# Table of Contents

1. Working with AS Tooling ................................................................. 7  
1.1 Installing the AS Tooling Plug-In ............................................. 7  
1.2 Creating an Axis2 Project .......................................................... 11  
1.3 Creating a Web Application ...................................................... 15  
1.4 Creating JAX-WS and JAX-RS Service Projects ......................... 20  
2. WSO2 Application Server Documentation ....................................... 23  
2.1 About WSO2 AS ........................................................................ 24  
2.1.1 Introducing AS .................................................................... 25  
2.1.2 Features ............................................................................. 27  
2.1.3 Architecture ....................................................................... 29  
2.1.4 About this Release .............................................................. 32  
2.2 Getting Started ........................................................................ 32  
2.2.1 Quick Start Guide ................................................................ 33  
2.2.2 Downloading the Product ..................................................... 38  
2.2.3 Installation Prerequisites ...................................................... 38  
2.2.4 Installing the Product ........................................................... 40  
2.2.4.1 Installing on Linux or OS X ........................................... 40  
2.2.4.2 Installing on Solaris ......................................................... 42  
2.2.4.3 Installing on Windows ..................................................... 44  
2.2.4.4 Installing as a Windows Service ....................................... 47  
2.2.4.5 Installing as a Linux Service ........................................... 54  
2.2.5 Running the Product ............................................................ 56  
2.3 User Guide ................................................................................ 61  
2.3.1 Application Development and Deployment ............................ 62  
2.3.1.1 Developing Web Applications ......................................... 63  
2.3.1.1.1 JAX-WS and JAX-RS Support .................................... 65  
2.3.1.1.2 Developing JAX-RS Applications ................................. 65  
2.3.1.1.3 Developing JAX-WS Applications ............................... 72  
2.3.1.1.4 Versioning Web Applications ..................................... 78  
2.3.1.2 Developing Jaggery Applications ..................................... 79  
2.3.1.3 Deploying Applications in AS .......................................... 79  
2.3.1.4 Managing and Monitoring Applications ............................ 82  
2.3.1.4.1 Using the Application Dashboard ................................. 82  
2.3.1.4.2 Web Application Statistics Monitoring with BAM .......... 86  
2.3.2 Services Development and Deployment .................................... 91  
2.3.2.1 Deploying Axis2 Services .............................................. 92  
2.3.2.2 Deploying JAR Services .................................................. 94  
2.3.2.3 Deploying Spring Services ............................................. 96  
2.4 Samples .................................................................................... 99  
2.4.1 Deploying your First Custom Web Application ...................... 100  
2.4.2 The Entitlement Servlet Filter Sample .................................... 101  
2.4.3 JAX-RS Samples .................................................................. 104  
2.4.3.1 Building and Running JAX-RS Samples ......................... 104  
2.4.3.2 JAX-RS Basics .............................................................. 106  
2.4.3.3 JAX-RS Advanced .......................................................... 107  
2.4.3.4 JAX-RS Sample on Content Negotiation ......................... 110  
2.4.4 JAX-WS Samples ................................................................. 111
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.4.1 Building and Running JAX-WS Samples</td>
<td>112</td>
</tr>
<tr>
<td>2.4.4.2 Asynchronous Invocation</td>
<td>113</td>
</tr>
<tr>
<td>2.4.4.3 Build and Invoke a Service Using a WSDL</td>
<td>114</td>
</tr>
<tr>
<td>2.4.4.4 Develop Services in Code-First Approach</td>
<td>115</td>
</tr>
<tr>
<td>2.4.4.5 JAX-WS Handlers</td>
<td>115</td>
</tr>
<tr>
<td>2.4.4.6 MTOM Sample for SWA and XOP</td>
<td>116</td>
</tr>
<tr>
<td>2.4.4.7 WS-Security Sample on Signature and Encryption</td>
<td>116</td>
</tr>
<tr>
<td>2.4.4.8 WS-Security Sample on UsernameToken and Timestamp</td>
<td>116</td>
</tr>
<tr>
<td>2.4.5 Jaggery Samples</td>
<td>117</td>
</tr>
<tr>
<td>2.4.5.1 Building Jaggery Samples</td>
<td>117</td>
</tr>
<tr>
<td>2.4.5.2 Available Jaggery Samples</td>
<td>118</td>
</tr>
<tr>
<td>2.4.6 WebSockets Sample</td>
<td>118</td>
</tr>
<tr>
<td>2.4.7 SSO Sample</td>
<td>119</td>
</tr>
<tr>
<td>2.5 Configuration Guide</td>
<td>121</td>
</tr>
<tr>
<td>2.5.1 Clustered Deployment</td>
<td>122</td>
</tr>
<tr>
<td>2.5.2 Working with Databases</td>
<td>122</td>
</tr>
<tr>
<td>2.5.2.1 Setting up the Physical Database</td>
<td>122</td>
</tr>
<tr>
<td>2.5.2.1.1 Setting up a MySQL Cluster</td>
<td>123</td>
</tr>
<tr>
<td>2.5.2.1.2 Setting up Derby</td>
<td>123</td>
</tr>
<tr>
<td>2.5.2.1.3 Setting up H2</td>
<td>130</td>
</tr>
<tr>
<td>2.5.2.1.4 Setting up IBM DB2</td>
<td>138</td>
</tr>
<tr>
<td>2.5.2.1.5 Setting up IBM Informix</td>
<td>144</td>
</tr>
<tr>
<td>2.5.2.1.6 Setting up Microsoft SQL</td>
<td>147</td>
</tr>
<tr>
<td>2.5.2.1.7 Setting up MySQL</td>
<td>151</td>
</tr>
<tr>
<td>2.5.2.1.8 Setting up OpenEdge</td>
<td>155</td>
</tr>
<tr>
<td>2.5.2.1.9 Setting up Oracle</td>
<td>159</td>
</tr>
<tr>
<td>2.5.2.1.10 Setting up Oracle RAC</td>
<td>163</td>
</tr>
<tr>
<td>2.5.2.1.11 Setting up PostgreSQL</td>
<td>167</td>
</tr>
<tr>
<td>2.5.2.1.12 Setting up MariaDB</td>
<td>171</td>
</tr>
<tr>
<td>2.5.2.2 Managing Datasources</td>
<td>175</td>
</tr>
<tr>
<td>2.5.2.2.1 Configuring an RDBMS Datasource</td>
<td>176</td>
</tr>
<tr>
<td>2.5.2.2.2 Configuring a Custom Datasource</td>
<td>185</td>
</tr>
<tr>
<td>2.5.3 Working with the Registry</td>
<td>187</td>
</tr>
<tr>
<td>2.5.3.1 Introduction to Registry</td>
<td>188</td>
</tr>
<tr>
<td>2.5.3.2 Managing the Registry</td>
<td>188</td>
</tr>
<tr>
<td>2.5.3.2.1 Managing Breadcrumb</td>
<td>189</td>
</tr>
<tr>
<td>2.5.3.2.2 Managing Metadata</td>
<td>190</td>
</tr>
<tr>
<td>2.5.3.2.3 Managing Properties</td>
<td>193</td>
</tr>
<tr>
<td>2.5.3.2.4 Managing Entries and Content</td>
<td>195</td>
</tr>
<tr>
<td>2.5.3.2.5 Managing Role Permissions</td>
<td>202</td>
</tr>
<tr>
<td>2.5.3.3 Searching the Registry</td>
<td>203</td>
</tr>
<tr>
<td>2.5.3.4 Setting up the Registry Space</td>
<td>205</td>
</tr>
<tr>
<td>2.5.3.4.1 All Partitions in a Single Server</td>
<td>206</td>
</tr>
<tr>
<td>2.5.3.4.2 Config and Governance Partitions in a Remote Registry</td>
<td>207</td>
</tr>
<tr>
<td>2.5.3.4.3 Governance Partition in a Remote Registry</td>
<td>212</td>
</tr>
<tr>
<td>2.5.3.4.4 Config and Governance Partitions in Separate Nodes</td>
<td>218</td>
</tr>
<tr>
<td>2.5.3.4.5 Using WS-Discovery in Governance Registry</td>
<td>229</td>
</tr>
<tr>
<td>2.5.4 Working with Transports</td>
<td>230</td>
</tr>
<tr>
<td>2.5.4.1 Introduction to Transports</td>
<td>231</td>
</tr>
<tr>
<td>2.5.4.2 AS Transports</td>
<td>231</td>
</tr>
</tbody>
</table>
## Contents

2.5.4.2.1 HTTP Servlet Transport ................................................................. 231  
2.5.4.2.2 HTTPS Servlet Transport ............................................................ 236  
2.5.4.2.3 MailTo Transport ................................................................. 237  
2.5.4.2.4 JMS Transport ............................................................................. 239  
2.5.4.2.5 TCP Transport ............................................................................ 243  
2.5.4.2.6 Local Transport ........................................................................... 244  
2.5.4.2.7 UDP Transport ............................................................................ 245  
2.5.4.3 Configuring Transports Globally ...................................................... 245  
2.5.5 Working with Security ......................................................................... 249  
2.5.6 Working with Multiple Tenants ............................................................ 250  
2.5.6.1 Managing Tenants ............................................................................ 250  
2.5.6.2 Multitenant Architecture ................................................................. 251  
2.5.6.3 Tenant-aware Load Balancing Using the WSO2 Elastic Load Balancer ................................................................. 254  
2.5.7 Working with Logs .............................................................................. 256  
2.5.7.1 Configure Logging using the Management Console ......................... 257  
2.5.7.2 HTTP Access Logging ..................................................................... 260  
2.5.7.3 Configure Logging using Config Files .............................................. 263  
2.5.8 Tuning Performance ............................................................................. 264  
2.5.9 JNDI Resources .................................................................................. 268  
2.5.9.1 Carbon JNDI Resources ................................................................ 268  
2.5.9.2 Exposing Datasources as JNDI Resources ......................................... 270  
2.6 Administration Guide ............................................................................... 275  
2.6.1 Working with Users, Roles and Permissions ........................................ 276  
2.6.1.1 Introduction to User Management .................................................... 276  
2.6.1.2 Configuring the User Realm ............................................................ 279  
2.6.1.2.1 Configuring the System Administrator ....................................... 280  
2.6.1.2.2 Configuring the Authorization Manager ..................................... 282  
2.6.1.2.3 Configuring User Stores ............................................................. 284  
2.6.1.3 Managing Users, Roles and Permissions .......................................... 318  
2.6.1.3.1 Changing a Password ................................................................ 319  
2.6.1.3.2 Configuring Roles ...................................................................... 320  
2.6.1.3.3 Configuring Users ...................................................................... 322  
2.6.2 Working with Features ........................................................................ 325  
2.6.2.1 Managing the Feature Repository ................................................... 326  
2.6.2.2 Installing Features .......................................................................... 327  
2.6.2.2.1 Feature Categorization .............................................................. 329  
2.6.2.3 Uninstalling Features ...................................................................... 334  
2.6.2.4 Converting the Product into a Back-end/Front-end Server ............... 335  
2.6.2.5 Recovering from Unsuccessful Feature Installation ......................... 336  
2.6.3 Monitoring the Application Server ........................................................ 338  
2.6.3.1 Collecting Data Statistics of Services .............................................. 338  
2.6.3.2 Monitoring Logs ............................................................................. 341  
2.6.3.2.1 Application Logs ...................................................................... 342  
2.6.3.2.2 System Logs .............................................................................. 344  
2.6.3.3 Monitoring Performance Statistics ................................................... 346  
2.6.3.4 Monitoring SOAP Messages ............................................................ 355  
2.6.3.5 Monitoring Message Flows .............................................................. 357  
2.6.3.6 Monitoring Messages with TCPMon ............................................... 359  
2.6.3.7 JMX-Based Monitoring ................................................................. 363  
2.6.4 Working with Composite Applications ............................................... 364  

---

Copyright © WSO2 Inc. 2014
2.6.4.1 Introduction to Composite Applications ........................................... 364
2.6.4.2 Creating and Deploying a Composite Application .......................... 365
2.6.4.3 Deleting Composite Applications ................................................. 366
2.6.4.4 Managing Server Roles ................................................................. 367
2.6.5 Running the Product on a Preferred Profile ...................................... 369
2.6.6 Migrating the Application Server ..................................................... 370
2.6.7 Calling Admin Services from Apps ................................................ 371
2.6.8 Working with Applications ............................................................... 375
  2.6.8.1 Configuring Applications for AS .................................................. 376
  2.6.8.2 Configuring Runtime Environments ............................................ 378
  2.6.8.3 Customize Logging for Web Applications .................................... 381
  2.6.8.4 HTTP Session Management ....................................................... 383
  2.6.8.5 Setting up Virtual Hosts .............................................................. 385
  2.6.8.6 Using SSO with Web Applications ............................................. 387
  2.6.8.7 Enable XACML Fine Grained Authorization for Web Apps ............. 390
  2.6.8.8 Enabling Java EE 6 Web Profile ................................................ 395
2.6.9 Working with Services ................................................................. 396
  2.6.9.1 Creating an Axis2 Module .......................................................... 396
  2.6.9.2 Deploying Axis2 Modules ............................................................ 403
  2.6.9.3 Managing Dependencies and Common Libraries ......................... 404
    2.6.9.3.1 Class Loading for Web Services ......................................... 405
    2.6.9.3.2 Uploading Shared Axis2 Artifacts ...................................... 407
    2.6.9.3.3 Restarting the Axis2 Configuration .................................... 409
    2.6.9.3.4 Deleting Common Axis2 Libraries ...................................... 409
2.6.10 Upgrading from a Previous Release ............................................ 410
2.7 Developer Guide ............................................................................. 415
  2.7.1 Working with the Source Code ...................................................... 416
  2.7.2 Basic Registry API Knowledge ...................................................... 416
    2.7.2.1 Adding a Resource with API .................................................. 416
    2.7.2.2 Creating a Resource from a Given URL .................................... 417
    2.7.2.3 Checking for the Existence of a Resource .................................. 417
    2.7.2.4 Deleting a Resource ................................................................. 418
    2.7.2.5 Managing Commenting ............................................................. 418
    2.7.2.6 Managing Ratings ................................................................. 419
    2.7.2.7 Managing Tags of a Resource ................................................ 419
    2.7.2.8 Managing Versions of a Resource ......................................... 420
    2.7.2.9 Reading a Resource ................................................................. 421
    2.7.2.10 Renaming a Resource ............................................................. 421
    2.7.2.11 Retrieving Metadata of a Resource ........................................ 422
    2.7.2.12 Moving and Copying a Resource ........................................... 422
    2.7.2.13 Managing Relationships of a Resource .................................... 422
    2.7.2.14 Executing a Query ................................................................. 423
    2.7.2.15 Searching for Content ........................................................... 423
    2.7.2.16 Retrieving Logs ................................................................. 424
    2.7.2.17 Managing Aspects ................................................................. 424
    2.7.2.18 Managing Symbolic Links ...................................................... 425
  2.7.3 CarbonContext API ..................................................................... 426
  2.7.4 WSO2 Patch Application Process ................................................. 432
  2.7.5 Customizing the Management Console ........................................... 434
  2.7.6 Enabling SPI Fly Support .............................................................. 436
# Table of Contents

2.7.7 Java Documentation ................................................................. 437  
2.8 Reference Guide ................................................................. 437  
  2.8.1 Configuration Files ............................................................. 438  
    2.8.1.1 Configuring axis2.xml ...................................................... 438  
    2.8.1.2 Configuring carbon.xml .................................................. 449  
    2.8.1.3 Configuring catalina-server.xml ....................................... 455  
    2.8.1.4 Configuring identity.xml ................................................ 466  
    2.8.1.5 Configuring master-datasources.xml .................................. 471  
    2.8.1.6 Configuring registry.xml ............................................... 473  
    2.8.1.7 Configuring user-mgt.xml .............................................. 476  
    2.8.1.8 Configuring config-validation.xml ................................. 477  
    2.8.1.9 Configuring sso-sp-config.properties ............................. 480  
  2.8.2 Changing the Default Ports .............................................. 483  
  2.8.3 Tools ............................................................................ 483  
    2.8.3.1 Try It Tool ................................................................. 484  
    2.8.3.2 Generate Java Code for WSDL ....................................... 488  
    2.8.3.3 Generate WSDL for Java Code ....................................... 490  
    2.8.3.4 Validating a WSDL ...................................................... 492  
    2.8.3.5 JavaScript Scraping Assistant ....................................... 493  
    2.8.3.6 JavaScript Stub Generator .......................................... 496  
    2.8.3.7 Tools of the Carbon Platform ....................................... 498  
      2.8.3.7.1 Capturing the State of the System in Error Situations .... 498  
      2.8.3.7.2 Changing User Passwords in the Carbon Database ........ 499  
      2.8.3.7.3 Encrypting and Decrypting Simple Texts ................... 500  
      2.8.3.7.4 View and Resend Messages ................................... 501  
  2.8.4 Lazy Loading of Tenants and Artifacts ................................ 506  
  2.8.5 Product Startup Options .................................................. 507  
  2.9 FAQ ............................................................................. 508  
  2.10 Getting Support ............................................................... 509  
  2.11 Glossary ..................................................................... 510  
  2.12 Site Map .................................................................... 513  

Copyright © WSO2 Inc. 2014
Working with AS Tooling

This section walks you through how to install the AS tooling plug-in, based on WSO2 Developer Studio Kernel, using which you can create and manage AS artifacts.

- Installing the AS Tooling Plug-In
- Creating an Axis2 Project
- Creating a Web Application
- Creating JAX-WS and JAX-RS Service Projects
Installing the AS Tooling Plug-In

The AS tooling plug-in is based on WSO2 Developer Studio Kernel, which gives the capabilities of a complete Eclipse-based development environment for the AS. You can develop services, features and artifacts and manage their links and dependencies through a simplified graphical editor.

You can install multiple WSO2 product plug-ins on top of the same Eclipse instance.

There are 3 ways to install the tooling plug-in. The following two methods require you to install Eclipse Luna separately in your system, if you do not have it already:

- Install the plug-in on Eclipse Luna using the P2 URL
- Install the plug-in on Eclipse Luna using the P2.zip file

The third method uses a complete plug-in installation with pre-packaged Eclipse, so that you do not have to install Eclipse separately:

- Install the plug-in with pre-packaged Eclipse

Install the plug-in on Eclipse Luna using the P2 URL

1. Make sure you have Eclipse IDE for Java EE Developers (Luna SR2) installed.
2. Open Eclipse and click Help > Install New Software.
3. On the dialog box that appears, click Add.

4. Give Appserver Tool as the name and http://product-dist.wso2.com/p2/developer-studio-
kernel/4.0.0/appserver-tools/releases/ as the location and click OK.

5. Select all the check boxes and click Next.

6. Read and accept the license agreements and click Finish.

7. If a security warning appears saying that the authenticity or validity of the software cannot be established, click OK.

8. Restart Eclipse to complete the installation.

Install the plug-in on Eclipse Luna using the P2.zip file
1. Make sure you have Eclipse IDE for Java EE Developers (Luna SR2) installed.
3. Open Eclipse and click Help > Install New Software.
4. On the dialog box that appears, click Add.

5. Give Appserver Tool as the name and click Archive.

6. Navigate to the downloaded .zip file and click OK.
7. Select all the check boxes and click Next.
8. Read and accept the license agreements and click Finish.

9. If a security warning appears saying that the authenticity or validity of the software cannot be established, click OK.

10. Restart Eclipse to complete the installation.
Install the plug-in with pre-packaged Eclipse

Download the distribution according to your operating system from http://product-dist.wso2.com/products/developer-studio-for-appserver/4.0.0/wso2-developer-studio-for-appserver_4.0.0.zip.
Creating an Axis2 Project

You can create an Axis2 project for use with the Application Server.

Creating an Axis2 service project

You can create a new, empty project and develop the service as a Java class inside the project, or create the project and service in one step using an existing WSDL file.

To create an empty service project:

1. On the Developer Studio Dashboard, click Axis2 Service Project, leave the first option selected, and click Next. The Create New Axis2 Service dialog box appears.

2. Do the following:
   1. Type a unique name for the project.
   2. Specify the package and class name.
   3. Optionally specify the location where you want to save the project (or leave the default location specified).
4. Optionally specify the working set, if any, that you want to include in this project.

3. A Maven POM file will be generated automatically for this project. If you want to include parent POM information in the file from another project in this workspace, click Next, select the Specify Parent from Workspace check box, and then select the parent project.

4. Click Finish.
   The project is created, and the service class is open in the editor.

5. Create a method in the service class. For example:

   ```java
   package org.wso2.cs.helloservices;
   public class HelloService{
       public String sayHello(String name){
           return "Hello"+name;
       }
   }
   ```

6. Generate the WSDL as follows:
   1. Right-click the project and choose Generate WSDL.
   2. Specify the location and name for the file and click OK.

The WSDL file is generated and appears in the editor.

**To create a service project from a WSDL:**

1. On the Developer Studio Dashboard, click Axis2 Service Project, click the option to create the project from a WSDL, and click Next.
2. Browse to the WSDL file, specify a name for the project, and then optionally specify the location and working sets as described in the previous steps.
3. Optionally click Next and modify the Axis2 options.
   For more information on Axis2 configuration, see: http://axis.apache.org/axis2/java/core/docs/adv-userguide.html
4. Click Finish.

The project is created with all the generated stubs created from the WSDL you specified.

Creating an Axis2 web service client

You can create an Axis2 client project to generate client classes from a WSDL.

2. Specify the WSDL, and click Next.
3. Specify the src folder where you want to create the client and click Next. If you want to create a new project for this client, click Create a new Project and enter the required information.
4. Configure the Axis2 options and click Finish.
   For more information on Axis2 configuration, see: http://axis.apache.org/axis2/java/core/docs/adv-userguide.html
The client classes are created in the source folder you specified.

**Creating an Axis2 web service skeleton**

You can create an Axis2 web service skeleton project to generate classes from a WSDL.

1. On the Developer Studio Dashboard, click **Generate Axis2 Web Service Skeleton**.
2. Specify the WSDL, and click **Next**.
3. Specify the src folder where you want to create the skeleton and click **Next**. If you want to create a new project for this skeleton, click **Create a new Project** and enter the required information.
4. Configure the Axis2 options and click **Finish**.
   
   For more information on Axis2 configuration, see: [http://axis.apache.org/axis2/java/core/docs/adv-userguide.html](http://axis.apache.org/axis2/java/core/docs/adv-userguide.html)

The classes are created in the source folder you specified.

**Generating an Axis2 service archive**

You can generate an Axis2 Service Archive (AAR file) by exporting the project.

1. Right-click the Axis2 service project and select **Export Project as Deployable Archive**.
2. Select the project and location where you want to save the archive and click **Finish**.

**Deploying the project**

You can add the project as a dependency of a Composite Application project that you deploy on the WSO2 Application Server.
Creating a Web Application

You can use Developer Studio to create a web application (webapp), which you can host on the Application Server. You can create a new webapp or import an existing webapp from a WAR file.

As you create the webapp, you can configure facets, runtimes, an EAR project, and the context root. For information on these concepts, see the topic on creating a dynamic web project in the Eclipse help (click Help -> Help Contents and search for "dynamic web project").

Creating a new webapp

Follow these steps to create a new webapp. Alternatively, you can import an existing webapp.

1. Open the Developer Studio Dashboard (click Developer Studio -> Open Dashboard) and click Web Application.
2. Select Create New Web Application and click Next.
3. Type a unique name for the webapp project, and then either leave the default location, or clear the check box to specify a new location where you want to create the project.
4. Specify the server runtime environment (the container in which the webapp will run). To enable the webapp to run with WSO2 Carbon products, click **New Runtime**, expand the WSO2 folder and select the version of Carbon.
5. Specify the carbon home path and click Finish.

6. Specify the dynamic web module version (the version of the Servlet API) that is compatible with the target container.
7. Specify the configuration you want to use as the starting configuration for this webapp. To modify the facets and limit the runtimes that this project is compatible with, click Modify.
8. Optionally add this project to an Enterprise Archive file (EAR) and working sets.
9. Click Next, and either leave the default web module settings or configure the settings as needed.
10. Click Next and optionally configure the Maven settings.
11. Click Finish.

The new webapp project has now been created in the workspace. You can export this webapp to a web archive (WAR) file by right-clicking the project and choosing Export -> War File. You can then import it into another project or environment.

Importing an existing webapp

Follow these steps to import an existing webapp from a WAR file. Alternatively, you can create a new webapp.

1. Open the Developer Studio Dashboard (click Developer Studio -> Open Dashboard) and click Web Application.
2. Select Import Web Application and click Next.
3. Specify the WAR file by typing its full path name or clicking Browse and navigating to the file.
4. Click **Next** if you want to specify Maven information.
5. Click **Finish**.
Creating JAX-WS and JAX-RS Service Projects

You can create JAX-WS (SOAP) and JAX-RS (REST) service projects for use with the Application Server. You can create a new, empty project and develop the service as a Java class inside the project, or create the project and service in one step using an existing WSDL file.

To create an empty project:

1. On the Developer Studio Dashboard, click JAX-WS or JAX-RS, leave the first option selected, and click Next. The Create New JAX-WS Service dialog box appears.

![Create New JAX-WS Service dialog box]

2. Do the following:
   1. Type a unique name for the project.
   2. Specify the package and class name.
   3. Optionally specify the location where you want to save the project (or leave the default location specified).
   4. Optionally specify the working set, if any, that you want to include in this project.

3. A Maven POM file will be generated automatically for this project. If you want to include parent POM information in the file from another project in this workspace, click Next, click the Specify Parent from Workspace check box, and then select the parent project.

4. Click Finish.

The project is created. You now create the web service as a Java class inside this project.

To create the Java service class:
1. Right-click the project you just created, and then choose **New -> JAX-WS Service Class** or **New -> JAX-RS Service Class**.
   The New JAX-WS Class dialog box appears.

   ![New JAX-RS class dialog box](image)

   - **Source folder**: JAX-RS/src/main/java
   - **Package**: org.wso2.devstudio.sample
   - **Name**: MyWebServiceImpl
   - **Superclass**: java.lang.Object

   The New JAX-RS Class dialog box appears.
   - **Service Interface**
     - **Package**: org.wso2.devstudio.sample
     - **Name**: MyWebService

   - **Create Service interface**
   - **Add sample webservice method to new class**

   - **Finish**

2. Specify the information for the bean class and optionally the interface class, and then click **Finish**.

   The classes are created inside the project.

**To create a project and service from a WSDL:**

1. On the Developer Studio Dashboard, click JAX-WS or JAX-RS, click the option to create the project from a WSDL, and click **Next**.
2. Enter a name for the project, browse to the WSDL file, and specify the CXF home and a custom package name.
   - You can download the **CXF Binary Distribution** and extract it. The location where you extract it is your CXF Home.

   The service class has been generated and contains the public endpoint interface described by the WSDL without the implementation. You now develop the service implementation class inside the project.

   You can export the project as a web application (WAR) file or you can add it as dependency of a Composite Application project that you deploy on the WSO2 Application Server. To create a deployable archive
(WAR), right-click the project, select **Export Project as Deployable Archive**, and choose the destination location.
Welcome to the WSO2 Application Server (AS) 5.3.0 documentation! The WSO2 AS is a lean, 100% open source, cloud-enabled, multi-tenant platform for developing, hosting and maintaining various types of Web services and Web applications.

Following are the key sections in the documentation that you should read to get up and running quickly. You can also browse the topics in the table of contents or search for a specific term or phrase. To download a PDF of this document or a selected part of it, click here (only generate one PDF at a time). You can also use this link to export to HTML or XML.

<table>
<thead>
<tr>
<th><strong>About WSO2 AS</strong></th>
<th><strong>Getting Started</strong></th>
<th><strong>User Guide</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>The topics in this section introduce Application Server (AS), including the business cases it solves, its features, architecture and how to get help or get involved.</td>
<td>Provides instructions for downloading and installing WSO2 AS and accessing its features.</td>
<td>Explores the features and functionality of the Application Server, solution development, testing, debugging and deployment.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Samples</strong></th>
<th><strong>Configuration Guide</strong></th>
<th><strong>Administration Guide</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-life business use cases of the product.</td>
<td>Explores different options to customize Application Server according to user-specific needs.</td>
<td>Describes all aspects of administering the AS, including deployment, user management, and more.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Developer Guide</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>This section has useful developer information on CarbonContext APIs and building the product from source.</td>
</tr>
</tbody>
</table>
About WSO2 AS

The topics in this section introduce you to WSO2 Application Server, including the business cases it solves, its features, and architecture.

- Introducing AS
- Features
- Architecture
- About this Release

Introducing AS

The following topics describe the application server's evolution, its place in SOA with an introduction to the WSO2 Application Server.

- What is an Application Server
- Evolution of Application Servers
- The WSO2 Application Server
- The WSO2 Application Server in context of SOA

What is an Application Server

An Application Server (AS) is a software program which accommodates the hosting, deployment and management of various applications. It is released under Apache Software License Version 2.0, one of the most business-friendly licenses available today. Residing in the middle tier between an organization's database layer and the user-interaction layer, an application server facilitates communication and connection between the front and back tiers.

For example, if a user triggers a particular control in a user interface, like clicking Submit, it typically invokes a service deployed in the application server. The deployed service or application defines the course of actions that need to be taken when the button is clicked. Few possibilities are data retrieval from the database and manipulation before presenting to the end-user or invoking other services as defined in the application's business logic. These operations are typically handled by an application server.

In addition to business logic implementation, application servers also centrally manage most reusable operations like security, monitoring, governance, high-availability etc. Separation and central management of an organization's pool of applications and services are key reasons for the wide acceptance and adoption of application servers in distributed enterprise architectures today.

Evolution of Application Servers

The application server concept emerged in early client-server environments. The term was originally used to differentiate servers that hosted databases and related services from file/print servers.

The client-server architecture was introduced back then to address some of the drawbacks of the file-server architecture, where file servers acted as mere shared storages and performed no logical processing. In contrast, the client-server environment had a server which hosted the database and performed related operations (similar to an application server). Many clients shared the server. End-user interaction was handled by the client computers whereas logical processing was cooperatively shared between clients and the server.

In a typical client-server environment, application developers had to write most of the program code for operations such as database interactions, client request management, session management and security, in addition to the actual business logic. Since every developer had to write similar code for common functionalities, packaging and
hosting them in a centrally-accessible server increased reusability and reduced the overhead. This triggered the emergence of the 3-tier (or N-tier) architecture, with reusable functionalities instilled in application servers in middle tier/s. This enables developers to focus simply on an application’s business logic implementation.

Application server functionality has enormously expanded over the years. Nowadays, most enterprises have 3-tiered (or N-tiered) distributed architectures with Web-enabled applications and services hosted and managed in application servers.

**The WSO2 Application Server**

WSO2 Application Server is a lean, open source, high-performant application server with Web application and Web services deployment and management capabilities.

It is fully open source and developed based on the lean, component-based WSO2 Carbon platform. It utilizes Apache Tomcat and is capable of hosting any kind of Web application deployable in Tomcat. Users can simply and effectively create, consume and manage their applications and services in a unified manner through the Application Server’s "Management Console" user-interface.

Unlike many over-bloated, proprietary solutions, WSO2 has created a new platform that cuts out unnecessary layers, supporting the full WS-* stack in a simple, clean, lightweight model. It is reliable, scalable and supports secure communication and transactional integration based on the concept of loose coupling. Web applications inherit features such as authentication/authorization and user management from the WSO2 Carbon platform.

The WSO2 Application Server brings together a number of open source projects into a single, seamless, easy-to-use offering. Instead of having you download different parts of the Apache Web services stack separately and assemble them, WSO2 has already done that for you. With WSO2 AS, you get a single, integrated package with lots of additional features, which are inherited from the powerful WSO2 Carbon platform. The WSO2 Application Server utilizes Apache WS* stack and REST protocol features, provides enterprise-class security features, JAX-WS support, governance, monitoring, clustering, high-availability, load balancing, tracing, logging and more.

WSO2 has positioned the application server product in 3 different phases. Currently in use is the WSO2 Application Server, which is the successor of WSO2 Web Services Application Server (WSAS). WSO2 WSAS was launched in 2008 with its predecessor being WSO2 Tungsten.

**The WSO2 Application Server in context of SOA**

An enterprise’s application servers are typically accessed by a variety of heterogeneous clients, within a LAN and over the Internet. Modern-day enterprises acknowledge the advantages of building Web-enabled applications on Web service interfaces. Web services have also become the fundamental building block of a service oriented architecture (SOA) implementation.

As a result, the WSO2 Application Server is designed to be portable and interacts with the outside using platform-agnostic technologies such as XML/SOAP messaging. This gives clients access to an enterprise’s applications and services irrespective of their infrastructures.
The WSO2 Application Server has extensive Web service management capabilities, which makes it ideal for SOA implementations. It utilizes Apache WS-* stack for Web services related functionalities along with Axis2 components such as Rampart, Sandesh2, Neethi etc. It also offers the unique capability to run Web applications and Web services on the same server and manage them simply through a unified graphical console.

WSO2 Application Server supports deployment of different types of services such as spring, jar, axis2 and JAX-WS. Service type depends on the way the service is developed. However, finally all services are deployed within the Axis2 engine and all features will be available for all service types.

Features

WSO2 Application Server has the key features listed below. For the relevant versions of the applications used as features, see Compatibility of WSO2 Products.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
</table>
| Java EE 6 Web Profile support                | Application Server (AS) 5.3.0 supports Java EE 6, which uses Apache TomEE. To use this feature in AS 5.3.0, you have to configure a new runtime called “Javaee”. With this new runtime, you will be allowed to deploy Java EE web apps in AS.  
  • See the topic on enabling Java EE 6 Web Profile for instructions on how to enable the JavaEE 6 Web profile for your server. |
<p>| WebSocket support for web applications       | Starting from version 5.3.0, WSO2 AS provides support for Websocket 1.1 API as defined by the JSR-356 specification. This is provided by Apache Tomcat, which is embedded in AS. See the following sample demonstration on how web sockets work in AS 5.3.0. |</p>
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Apache Aries SPI Fly** | The WSO2 Carbon platform integrates ‘Apache Aries SPI Fly’, which provides OSGi support for JRE SPI mechanisms.  
- See the topic on enabling SPI Fly support for details. |
| **Publishing web application statistics from AS to BAM** | The Application Server (AS) previously had the capability of only monitoring statistics through WSO2’s Business Activity Monitor (BAM). However, now the Application Server has the capability of monitoring web application statistics as well using BAM.  
- See the topic on web application statistics monitoring with BAM for details. |
| **Support for multiple AS modes/profiles** | Multi-profile support allows the user to run the product pack in a preferred profile. The available profiles are:  
- Web Application Profile  
- Axis2 Service Profile  
- Jaggery Profile |
| **Distributed Caching implementation based on Hazelcast** | Cluster wide distributed caching is supported by using Hazelcast implementation. Thereby, the caching API, response caching etc. can be used in a distributed manner. |
| **Clustering support for High Availability and High Scalability** | You can install multiple instances of WSO2 products in a cluster to ensure that if one instance becomes unavailable or is experiencing high traffic, another instance will seamlessly handle the requests. Starting from AS 5.2.1, we provide clustering support through Hazelcast. For more information see, Clustering Deployment. |
| **Secondary user store support** | Secondary user stores can be dynamically configured via the Application Server UI. In addition, users are allowed to configure their own user stores, out-of-the-box. |
| **CarbonContext API** | This is an API for developers to access certain core services (such as registry, caching and more) from the Carbon runtime. |
| **Support for Servlet 3, JSP 2.2, EL 2.2, JSTL 1.2 specifications** |  |
| **Full JAX-WS 2.2 and JAX-RS 2.0 specification support** | WSO2 AS provides out-of-the-box support for JAX-WS 2.2 and JAX-RS 2.0 specifications via CXF. |
| **Integration of Jaggery** | Jaggery is a server side scripting framework. It allows you to write web apps and HTTP-focused web services for all aspects of the application: front-end, communication, server-side logic and persistence in pure Javascript. |
### Multi-tenant support for standalone deployment

In addition to the ability of creating tenants in cloud deployments, the user can create and use tenants in standalone mode as well.

<table>
<thead>
<tr>
<th>100% Apache Tomcat compliance</th>
<th>WSO2 Application Server is based on Apache Tomcat and is compliant with Tomcat.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lazy loading</strong> for tenants</td>
<td>Lazy loading support for tenants allows you to load the tenants on request. For example, the tenant is loaded when a tenant user is logged in, and a web application/service request is sent to a tenant's artifact. This starts the server much faster, due to efficient memory usage, since all the tenants and its artifacts are not loaded at a given time.</td>
</tr>
<tr>
<td><strong>Lazy loading</strong> for web applications and services</td>
<td>This allows the user to load the web applications and web services only when it's being requested. This starts the server faster, due to efficient memory usage, since all the artifacts are not loaded at a given time.</td>
</tr>
<tr>
<td>Tooling support</td>
<td>Application Server related artifacts can be easily generated using WSO2 Developer Studio, which is an Eclipse plugin.</td>
</tr>
<tr>
<td><strong>JMX and Web interface based monitoring and management</strong></td>
<td>By default, JMX is enabled in WSO2 Carbon. Therefore, the WSO2 Application Server facilitates JMX-based monitoring. Some of the monitoring and management capabilities available in the Application Server's Management Console are also exposed via JMX.</td>
</tr>
<tr>
<td><em><em>WS-</em> and REST support</em>*</td>
<td>WSO2 Application Server (AS) supports most of the WS specifications including WS-Security, WS-Trust, WS-Policy, WS-Secure Conversation, WS-Reliable Messaging, WS-Discovery etc. AS also includes support for REST Web services via JAX-RS and WSDL 2.0</td>
</tr>
<tr>
<td><strong>UI, command line, and IDE-based tools for Web service and Web application development</strong></td>
<td>You can develop Web services (such as, Axis2 etc.) and Web applications (such as, JAX-WS, JAX-RS and generic webapps) via Developer Studio, which is an Eclipse plugin.</td>
</tr>
<tr>
<td><strong>Equinox P2 based provisioning support</strong></td>
<td>The user can install/update/uninstall features to/from a WSO2 Carbon-based products with WSO2 Carbon.</td>
</tr>
<tr>
<td><strong>WSDL2Java/Java2WSDL/WSDL 1.1, and UI-based try it (invoke any remote Web service), TCPMon</strong></td>
<td>The WSO2 Application Server ships these tools since these are very useful for web service developers.</td>
</tr>
</tbody>
</table>

### Architecture
The WSO2 Application Server has Apache Axis2 framework as its core and is distributed under Apache License 2.0. It has full support for Apache Tomcat, which is the underlying Web application container and can be installed standalone or on top of Tomcat, WebSphere, WebLogic, Jboss or as part of a Java EE compliant servlet container.

Because the WSO2 Application Server is part of WSO2 Carbon, the component-based middleware platform, it consists of a set of pluggable components. Developers have the choice to deploy only the required components and leave unwanted functionality which adds to the bulk.

The diagram below describes WSO2 Application Server architecture from a messaging perspective.

1. Different Web channels such as client applications or mobile phones, communicate with an enterprise’s application server through the Internet or a private network. These Web channels typically use XML/SOAP or REST messages.
2. The "transport" in the message receiver’s end manages concurrent connections and listens to messages contacting the application server via a particular protocol. For example, an “HTTP transport listener” detects XML/SOAP messages communicated in the HTTP protocol.
3. A transport which detects a message identifies if it needs any additional processing, also known as Quality of Service (QoS) operations such as security checks. The transport then sends the message through a message pipe containing handlers to take care of the additional processing.
4. Once the message has gone through the message pipe, it is passed on to the Message Receiver.
5. The Message Receiver determines the Web Service that needs to be invoked out of the services deployed in the application server and executes it.

This is how a message sent from a Web channel communicates with the Application Server and invokes a Web Service. Similarly, Web Application clients can call Web applications deployed inside the Application Server directly through Tomcat transports (HTTP/S).

All these components can be managed and monitored through the WSO2 Application Server’s Management Console.

**Application server components**

The following scheme visually presents the component-based architecture of the WSO2 Application Server.
The WSO2 Application Server is a light-weight, high-performant and integrated Web services middleware platform. For its Web service related functionalities, it uses components from the Apache WS-* stack. Apache Axis2 is the heart of the WSO2 Application Server's SOAP processing engine. Other components like Rampart, Sandesha2, Neethi etc. provide the other WS-* capabilities.

Transports

The WSO2 Application Server supports a number of transports through which the clients can send requests. As listed in the figure above, HTTP/S, JMS, Mail, XMPP and TCP are the main transports that are supported. In addition to that, any transport can be easily plugged in using the APIs provided by Axis2. For more information, see Working with Transports.

Web Services

The WSO2 Application Server supports different types of services. A service's type depends on the way the service is developed. However, finally all services are deployed within the Axis2 engine and features are available for all service types. For more information, see the topic on Services development and deployment.

Applications

In the standalone mode, the WSO2 Application Server runs on top of an embedded Tomcat instance. This is from where it inherits the Web application deployment capabilities. It also supports development and deployment of other types of applications, including JAX-WS and JAX-RS.

For more information, see Developing Applications and Deploying Applications.

Tools

The WSO2 Application Server comes with different kinds of tools for running, testing and monitoring services and applications.

Carbon Context

The WSO2 Application Server has an in-built registry/repository to store the configuration and configuration metadata. It provides the facility to use a remote repository as well.

The WSO2 Governance Registry is something the WSO2 Application Server inherits from the WSO2 Carbon platform, the component-based platform upon which all WSO2 products are built. WSO2 Governance Registry acts as the metadata repository for the deployed services. In addition to that, users can use it for governance features as well. For more information, see Working with the Registry.
In addition, the Application Server can be deployed on other Application servers (as a Webapp) like WebSphere, WebLogic and Jboss as well.

**About this Release**

**What is new in this release**

The WSO2 Application Server version 5.3.0 (WSO2 AS 5.3.0) is the successor of version 5.2.1. It contains several new features and bug fixes. Given below is the list of new features enabled in WSO2 AS 5.3.0.

- Supporting Java EE 6 web profile, which enables the Java EE runtime environment. See this link for documentation.
- Supporting Java Web Sockets 1.0 API. See the following sample demonstration on how web sockets work in AS 5.3.0.
- Possibility to enable virtual hosts, which allows you to deploy applications in multiple domains. See this link for documentation.
- Possibility to manage sessions of web applications by enabling session persistence. See this link for documentation.
- Possibility to enable the Single Sign On functionality for applications. See this link for documentation.
- Integration with Apache Aries SPI Fly, which supports OSGi JRE SPI. See this link for documentation.
- Possibility to discover JAX-RS/JAX-WS applications (deployed in AS 5.3.0) using WSO2 Governance Registry. Note that this feature will be deprecated in future. See this link for documentation.

**Removed features**

The data services feature is removed from WSO2 AS 5.3.0. Therefore, the following functionality is no longer available in WSO2 AS:

- Creating, uploading and generating data services.
- Scheduling tasks for data services.

If required, you can add this feature to your WSO2 AS 5.3.0 product pack from the WSO2 feature repository using the instructions given here.

**Compatible versions**

WSO2 AS 5.3.0 is based WSO2 Carbon 4.4.1 and is expected to be compatible with any of the WSO2 products that are based on any Carbon 4.4.x version. If you get any compatibility issues, please contact the WSO2 team. For information on third-party software requirements with AS 5.3.0, see Installation Prerequisites. For more information on the products in each Carbon platform release, see the Release Matrix.

**Fixed issues**

For the list of fixed issues, go to WSO2 Application Server 5.3.0 - Fixed Issues.

**Known issues**

For the list of known issues, go to WSO2 Application Server 5.3.0 - Known Issues.
Getting Started

The section provides instructions to download, install, run, and get started quickly with the WSO2 Application Server.

- Quick Start Guide
- Downloading the Product
- Installation Prerequisites
- Installing the Product
- Running the Product

Quick Start Guide

The purpose of this guide is to get you started on deploying applications in WSO2 Application Server (WSO2 AS) as quickly as possible. See the following topics for details:

- Introduction and key concepts
- Installing and setting up WSO2 AS
- Deploying and managing your first application
  - Step 1: Deploying an application
  - Step 2: Running the application
  - Step 3: Monitoring the application
- Deep diving into the Application Server

Introduction and key concepts

WSO2 Application Server (WSO2 AS) is a complete solution for hosting, deploying and managing applications and services. It lies as the middle tier between the backend (database layer) and the front end (consumers of WSO2 AS) of a system. The following diagram depicts how consumers (web channels and client applications) connect to an application deployed in WSO2 AS:

The main function of WSO2 AS is to deploy applications that are designed to perform certain tasks, such as retrieving data from a database and manipulating the data. External Web channels connect to the Web applications deployed in the AS to consume the services performed by the applications. For example, if a user triggers a control in a Web channel (like clicking Submit), it sends a message to the application deployed in the AS. The application deployed in the AS defines the course of actions that need to be taken to serve the Web channel (such as retrieving data from the database and presenting it to the end-user).

WSO2 AS also comes with a list of features to help you deploy and manage Web applications easily and securely. You get the required configurations already bundled and built in the binary distribution. You can use it to start deploying applications and services immediately. When you use the AS in a production environment, you can
enhance the configurations in order to obtain a more robust system that caters to your requirements.

---

**Installing and setting up WSO2 AS**

Follow the steps given below to download WSO2 DSS.

1. Download WSO2 AS from [here](#).
2. Extract the zip file to a location in your computer. This location will be referred to as `<AS_HOME>` from hereon.

Before you start the server, the following prerequisites should be in place:

1. Ensure that you have JDK 7/8 installed in your computer.
2. You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

WSO2 AS is now installed with the required settings.

---

**Deploying and managing your first application**

Let's deploy a sample Web application called HelloWorldWebapp and expose it to an external Web channel.

**Step 1: Deploying an application**

You are now ready to deploy your first application in WSO2 AS. You will be using the sample Web application ([HelloWorldApp](#)) for this demonstration. Follow the steps given below.

1. Download the sample HelloWorldWebapp Web application from [here](#) and save it to a location in your computer.
2. Open a terminal and navigate to the `<AS_HOME>/bin` directory and execute the AS startup script using one of the following commands:
   - On Windows: `wso2server.bat`
   - On Linux: `sh wso2server.sh`
3. When the product is started, the URL of the Management Console will be shown in the terminal as follows:
   ```
   INFO {org.wso2.carbon.ui.internal.CarbonUIServiceComponent} - Mgt Console URL: https://10.100.5.65:9443/carbon/
   ```
4. Copy this URL to your browser to open the Management Console.
5. Log in to the Management Console using the default administrator credentials: admin/admin.
6. In the left navigator, go to the **Main** tab and click **Web Applications** under the **Add** menu.

7. The **Upload** page opens.
1. Click Choose File and browse for the WAR file of the HelloWorldWebapp application.

2. Enter a version for the Web application in the Version field. We will used '1.0.0' as the first version of this application.

   By versioning the WAR file, you can add more applications with the same name later. Read more about versioning of Web applications from [here](#).

8. Once the file is uploaded successfully, click Applications -> List. The HelloWorldWebapp Web application will be listed in the Running Applications page as shown below.

You have deployed a sample Web application in the server. Let's run it.

**Step 2: Running the application**

Follow the steps below to connect to the application that you deployed in the previous step.

1. In the Management Console, click Applications -> List to open the Running Applications page.
2. See that the HelloWorldWebapp Web application is listed as an application deployed in the server.
3. Click the Go To URL link associated with the HelloWorldWebapp Web application that you deployed earlier.
4. Now that you have the URL of the application deployed in WSO2 AS, you can connect to the application from any external Web channel such as your browser.
5. The application opens in your browser as shown below.

   ![Image of application URL]

   Hello 1!

**Step 3: Monitoring the application**
The Management Console of WSO2 AS allows you to access the statistics of a running server instance using Application Dashboards. Follow the steps given below.

1. Click **Applications -> List** in the left navigator of the Management Console to open the **Running Applications** page.
2. Click the **HelloWorldWebapp** application that is listed there. This will take you to the dashboard dedicated for the application.

You will see the following information displayed on the dashboard:

1. The **Application Details** panel will show all the general information about the Web application.

<table>
<thead>
<tr>
<th><strong>Context</strong></th>
<th>/HelloWorldWebapp</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Display Name</strong></td>
<td>HelloWorldWebapp.war</td>
</tr>
<tr>
<td><strong>State</strong></td>
<td>Started</td>
</tr>
<tr>
<td><strong>Last Modified</strong></td>
<td>2015-10-13 10:14:27</td>
</tr>
</tbody>
</table>

2. The **Session Statistics** panel will show all the statistical information about the user sessions of the applications. It also provides a graphical view of the average system response time.

   | **Active Sessions** | 1 |
   | **Expired Sessions** | 1 |
   | **Maximum Concurrent Active Sessions** | 1 |
   | **Rejected Sessions** | 0 |
   | **Average Session Lifetime** | 1834 s |
   | **Maximum Session Lifetime** | 1834 s |
   | **Maximum Session Inactivity Interval** | 1800 s |

3. In the **Statistics** panel, you will find the following statistics of a running web application. Note that these statistics are not applicable to **Jaggery** applications.

   - **Request Count** - Total number of requests received to web applications.
   - **Response Count** - Total number of responses served by the web applications.
   - **Fault Count** - Total number of requests that triggered faults.
   - **Maximum Response Time** - Maximum time required to serve a request
   - **Minimum Response Time** - Minimum time required to serve a request
   - **Average Response Time** - Average time required to serve a request.
Deep diving into the Application Server

If you have followed the use case given above, you should now be familiar with using the management console of WSO2 AS and its basic functionality. WSO2 AS comes with many features that you can configure to support your enterprise requirements. Listed below are some of the more advance features that you can use when you manage applications in WSO2 AS.

- Enabling SaaS mode and CORS for your applications. See this link for instructions.
- Customizing logging management for your applications. See this link for instructions.
- Deploying multiple domains in a single server instance by configuring virtual hosts. See this link for instructions.
- Using Single-Sign-On with SAML for your web applications by integrating with WSO2 Identity Server. See this link for instructions.
- Using XACML fine grained authorization for your web applications by integrating with WSO2 Identity Server. See this link for instructions.
- Configuring class loading for your web applications using separate runtime environments. See this link for instructions.

For more in-depth information about WSO2 AS see the following links:
**User Guide:** Explains the various types of web applications and how they can be deployed in WSO2 AS.

**Configuration Guide:** Explains the various configuration options that are available for your product, such as enhancing Security, Multitenancy support, Logging management, Database management etc.

**Administration Guide:** Explains how you can further enhance the features of your product.

### Downloading the Product

Follow the instructions below to download WSO2 AS. You can also download and build the source code.

2. Click the **Download** button in the upper right-hand corner of the page to download the latest version. To download an older version, click the **Previous Releases** link and then select the version that you want.
3. Enter the required details in the form, and click **Download**.

The binary distribution contains the AS binary files for both MS Windows and Linux operating systems, compressed into a single ZIP file. This distribution is recommended for many users.

After downloading the binary distribution, go to **Installation Prerequisites** for instructions on installing the necessary supporting applications.

### Installation Prerequisites

Prior to installing any WSO2 Carbon-based product, it is necessary to have the appropriate prerequisite software installed on your system. Verify that the computer has the supported operating system and development platforms before starting the installation.

**System requirements**

| Memory | • ~ 2 GB minimum  
|        | • ~ 512 MB heap size. This is generally sufficient to process typical SOAP messages but the requirements vary with larger message sizes and the number of messages processed concurrently. |

| Disk | • ~ 1 GB, excluding space allocated for log files and databases. |

**Environment compatibility**
Operating Systems / Databases

- All WSO2 Carbon-based products are Java applications that can be run on any platform that is Oracle JDK 7/8 compliant. Also, we do not recommend OpenJDK as we do not support it or test our products with it.
- All WSO2 Carbon-based products are generally compatible with most common DBMSs. For more information, see Working with Databases.
- It is not recommended to use Apache DS in a production environment due to issues with scalability. Instead, it is recommended to use an LDAP like OpenLDAP for user management.
- For environments that WSO2 products are tested with, see Compatibility of WSO2 Products.
- If you have difficulty in setting up any WSO2 product in a specific platform or database, please contact us.

Required applications

The following applications are required for running the product and its samples or for building from the source code. Mandatory installations are marked with an asterisk *.

<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
<th>Version</th>
<th>Download Links</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle Java SE Development Kit (JDK)*</td>
<td>To launch the product as each product is a Java application. To build the product from the source distribution (both JDK and Apache Maven are required). To run Apache Ant.</td>
<td>JDK 7 or 8. Oracle and IBM JRE 1.7 are also supported when running (not building) WSO2 products. <strong>We do not recommend OpenJDK as we do not support it or test our products with it.</strong></td>
<td><a href="http://java.sun.com/javase/downloads/index.jsp">http://java.sun.com/javase/downloads/index.jsp</a></td>
</tr>
<tr>
<td>Apache ActiveMQ JMS Provider</td>
<td>To enable the product’s JMS transport and try out JMS samples if any. The ActiveMQ client libraries must be installed in the product’s classpath before you can enable the JMS transport.</td>
<td>5.5.0 or later If you use any other JMS provider (e.g. Apache Qpid), install any necessary libraries and/or components.</td>
<td><a href="http://activemq.apache.org/">http://activemq.apache.org/</a></td>
</tr>
</tbody>
</table>
You are now ready to install WSO2 AS. Click one of the following links for instructions:

- Installing on Linux or OS X
- Installing on Solaris
- Installing on Windows
- Installing as a Windows Service

**Installing the Product**

Installing WSO2 AS is very fast and easy. Before you begin, be sure you have met the installation prerequisites, and then follow the installation instructions as given below.

- Installing on Linux or OS X
- Installing on Solaris
- Installing on Windows
- Installing as a Windows Service
- Installing as a Linux Service

**Installing on Linux or OS X**
Follow the instructions below to install the required applications and WSO2 AS on Linux or Mac OS X.

**Install the required applications**

1. Establish an SSH connection to the Linux machine or log in on the text Linux console.
2. Be sure your system meets the Installation Prerequisites. Java Development Kit (JDK) is essential to run the product.

**Installing the product**

1. If you have not done so already, download the latest version of AS as described in Downloading the Product.
2. Extract the archive file to a dedicated directory for the product, which will hereafter be referred to as `<AS_HOME>`.

**Setting JAVA_HOME**

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

Environment variables are global system variables accessible by all the processes running under the operating system.

1. In your home directory, open the `BASHRC` file in your favourite Linux text editor, such as vi, emacs, pico or mcedit.
2. Add the following two lines at the bottom of the file, replacing `/usr/java/jdk1.6.0_25` with the actual directory where the JDK is installed.

   ```
   export JAVA_HOME=/usr/java/jdk1.6.0_25
   export PATH=${JAVA_HOME}/bin:${PATH}
   ```

   The file should now look like this:

   ```
   #.bashrc
   # Source global definitions
   if [ -f /etc/bashrc ]; then
   . /etc/bashrc
   fi
   # User specific aliases and functions.
   export JAVA_HOME=/usr/java/jdk1.6.0_25
   export PATH=${JAVA_HOME}/bin:${PATH}
   export M2_HOME=/opt/apache-maven-3.8.3
   export PATH=${M2_HOME}/bin:${PATH}
   ```

3. Save the file.

   If you do not know how to work with text editors in a Linux SSH session, run the following command:
3. To verify that the `JAVA_HOME` variable is set correctly, execute the following command:

```bash
echo $JAVA_HOME
```

The system returns the JDK installation path.

### Setting system properties

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**
  Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.

- **Set the properties from an external registry**
  If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as, username and password to connect to the registry in a properties file, instead of in the Java code and secure the properties file with the secure vault. Read about secure vault in the section on security.

### SUSE Linux

When using SUSE Linux, it ignores `/etc/resolv.conf` and only looks at the `/etc/hosts` file. This means that the server will throw an exception on startup if you have not specified anything besides localhost. To avoid this error, add the following line above `127.0.0.1 localhost` in the `/etc/hosts` file:

```
<ip_address> <machine_name> localhost.
```

You are now ready to **run the product**.

### Installing on Solaris

**Before you begin, please see our compatibility matrix** to find out if this version of the product is fully tested on Solaris.
Follow the instructions below to install the required applications and the product on Solaris.

**Installing the supporting applications**

1. Establish an SSH connection to the Solaris machine or log in on the text console.
2. Be sure your system meets the Installation Prerequisites. Java Development Kit (JDK) is essential to run the product.

**Installing the product**

1. If you have not done so already, download the latest version of the product as described in Downloading the Product.
2. Extract the archive file to a dedicated directory for the product, which will hereafter be referred to as `<PRODUCT_HOME>`.

**Setting JAVA_HOME**

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer.

Environment variables are global system variables accessible by all the processes running under the operating system.

1. In your home directory, open the BASHRC file in your favorite text editor, such as vi, emacs, pico or mcedit.
2. Add the following two lines at the bottom of the file, replacing `/usr/java/jdk1.6.0_25` with the actual directory where the JDK is installed.

   ```bash
   export JAVA_HOME=/usr/java/jdk1.6.0_25
   export PATH=${JAVA_HOME}/bin:${PATH}
   ```

The file should now look like this:

```bash
# .bashrc
# Source global definitions
if [ -f /etc/bashrc ]; then
    . /etc/bashrc
fi

# User specific aliases and functions
export JAVA_HOME=/usr/java/jdk1.6.0_25
export PATH=${JAVA_HOME}/bin:${PATH}
export M2_HOME=/opt/apache-maven-3.6.3
export PATH=${M2_HOME}/bin:${PATH}
```

3. Save the file.

   If you do not know how to work with text editors in an SSH session, run the following command:

   ```bash
   cat >> .bashrc
   ```

   Paste the string from the clipboard and press "Ctrl+D."
4. To verify that the `JAVA_HOME` variable is set correctly, execute the following command: `echo $JAVA_HOME`

```
[suncom@wso2 ~]$ echo $JAVA_HOME
/usr/java/jdk1.6.0_25
[suncom@wso2 ~]$
```

The system returns the JDK installation path.

**Setting system properties**

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**
  Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and adds the properties you want to set, rather than editing the WSO2 startup script directly.

- **Set the properties from an external registry**
  If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as, username and password to connect to the registry, in a properties file instead of in the Java code and secure the properties file with the secure vault.

You are now ready to run the product.

**Installing on Windows**

Follow the instructions below to install the required applications and the product on Windows.

**Installing the supporting applications**

- Be sure your system meets the **Installation Prerequisites**. Java Development Kit (JDK) is essential to run the product.
- Be sure that the `PATH` environment variable is set to "C:\Windows\System32", because the `findstr` windows `exe` is stored in this path.

**Installing the product**

1. If you have not done so already, download the latest version of the product as described in **Downloading the Product**.
2. Extract the archive file to a dedicated directory for the product, which will hereafter be referred to as `<PRODUCT_HOME>`.

**Setting JAVA_HOME**

You must set your `JAVA_HOME` environment variable to point to the directory where the Java Development Kit (JDK) is installed on the computer. Typically, the JDK is installed in a directory under `C:\Program Files\Java`, such as `C:\Program Files\Java\jdk1.6.0_27`. If you have multiple versions installed, choose the latest one, which you can find by sorting by date.
Environment variables are global system variables accessible by all the processes running under the operating system. You can define an environment variable as a system variable, which applies to all users, or as a user variable, which applies only to the user who is currently logged in.

You can set JAVA_HOME using the system properties, as described below. Alternatively, if you just want to set JAVA_HOME temporarily in the current command prompt window, set it at the command prompt.

**Setting JAVA_HOME using the System Properties**

1. Right-click the "My Computer" icon on the desktop and choose **Properties**.

2. In the System Properties window, click the **Advanced** tab, and then click the **Environment Variables** button.
3. Click the New button under "System variables" (for all users) or under "User variables" (just for the user who is currently logged in).

4. Enter the following information:
   - In the "Variable name" field, enter: JAVA_HOME
   - In the "Variable value" field, enter the installation path of the Java Development Kit, such as: c:\Program Files\Java\jdk1.6.0_27

5. Click OK.

The JAVA_HOME variable is now set and will apply to any subsequent command prompt windows you open. If you have existing command prompt windows running, you must close and reopen them for the JAVA_HOME variable to take effect, or manually set the JAVA_HOME variable in those command prompt windows as described in the next section. To verify that the JAVA_HOME variable is set correctly, open a command window (from the Start menu, click Run, and then type CMD and click Enter) and execute the following command:
The system returns the JDK installation path.

**Setting JAVA_HOME temporarily using the Windows command prompt (CMD)**

You can temporarily set the JAVA_HOME environment variable within a Windows command prompt window (CMD). This is useful when you have an existing command prompt window running and you do not want to restart it.

1. In the command prompt window, enter the following command where `<JDK_INSTALLATION_PATH>` is the JDK installation directory and press Enter:
   
   ```
   set JAVA_HOME=<JDK_INSTALLATION_PATH>
   ```

   For example:

   ```
   set JAVA_HOME=c:\Program Files\java\jdk1.6.0_27
   ```

   The JAVA_HOME variable is now set for the current CMD session only.

2. To verify that the JAVA_HOME variable is set correctly, execute the following command:

   ```
   set JAVA_HOME
   ```

   The system returns the JDK installation path.

**Setting system properties**

If you need to set additional system properties when the server starts, you can take the following approaches:

- **Set the properties from a script**
  Setting your system properties in the startup script is ideal, because it ensures that you set the properties every time you start the server. To avoid having to modify the script each time you upgrade, the best approach is to create your own startup script that wraps the WSO2 startup script and add the properties you want to set, rather than editing the WSO2 startup script directly.

- **Set the properties from an external registry**
  If you want to access properties from an external registry, you could create Java code that reads the properties at runtime from that registry. Be sure to store sensitive data such as, username and password to connect to the registry in a properties file, instead of in the Java code and secure the properties file with the secure vault.

You are now ready to run the product.

**Installing as a Windows Service**

WSO2 Carbon and any Carbon-based product can be run as a Windows service as described in the following sections:

- **Prerequisites**
- **Setting up the YAJSW wrapper configuration file**
- **Setting up CARBON_HOME**
- **Running the product in console mode**
- **Working with the WSO2CARBON service**

**Prerequisites**
- Install JDK 1.6.24 or later or 1.7.* and set up the JAVA_HOME environment variable.
- Download and install a service wrapper library to use for running your WSO2 product as a Windows service. WSO2 recommends Yet Another Java Service Wrapper (YAJSW) version 11.03, and several WSO2 products provide a default wrapper.conf file in their <PRODUCT_HOME>/bin/yajsw/ directory. The instructions below describe how to set up this file.

### Setting up the YAJSW wrapper configuration file

The configuration file used for wrapping Java Applications by YAJSW is wrapper.conf, which is located in the <YAJSW_HOME>/conf/ directory and in the <PRODUCT_HOME>/bin/yajsw/ directory of many WSO2 products. Following is the minimal wrapper.conf configuration for running a WSO2 product as a Windows service. Open your wrapper.conf file, set its properties as follows, and save it in <YAJSW_HOME>/conf/ directory.

If you want to set additional properties from an external registry at runtime, store sensitive information like usernames and passwords for connecting to the registry in a properties file and secure it with secure vault.

---

**Minimal wrapper.conf configuration**

```
#********************************************************************
# working directory
#********************************************************************
wrapper.working.dir=${carbon_home}\\
# Java Main class.
# YAJSW: default is "org.rzo.yajsw.app.WrapperJVMMain"
# DO NOT SET THIS PROPERTY UNLESS YOU HAVE YOUR OWN IMPLEMENTATION
# wrapper.java.mainclass=
#********************************************************************
# tmp folder
# yajsw creates temporary files named in_. out_. err_. jna..
# per default these are placed in jna.tmpdir.
# jna.tmpdir is set in setenv batch file to <yajsw>/tmp
#********************************************************************
wrapper.tmp.path = ${jna_tmpdir}
#********************************************************************
# Application main class or native executable
# One of the following properties MUST be defined
#********************************************************************
# Java Application main class
wrapper.java.app.mainclass=org.wso2.carbon.bootstrap.Bootstrap
# Log Level for console output. (See docs for log levels)
wrapper.console.loglevel=INFO
# Log file to use for wrapper output logging.
wrapper.logfile=${wrapper_home}/log/wrapper.log
# Format of output for the log file. (See docs for formats)
#wrapper.logfile.format=LPTM
# Log Level for log file output. (See docs for log levels)
#wrapper.logfile.loglevel=INFO
# Maximum size that the log file will be allowed to grow to before
# the log is rolled. Size is specified in bytes. The default value
# of 0, disables log rolling by size. May abbreviate with the 'k' (kB) or
# 'm' (mB) suffix. For example: 10m = 10 megabytes.
# If wrapper.logfile does not contain the string ROLLNUM it will be
```
automatically added as suffix of the file name
wrapper.logfile.maxsize=10m
# Maximum number of rolled log files which will be allowed before old
# files are deleted. The default value of 0 implies no limit.
wrapper.logfile.maxfiles=10
# Title to use when running as a console
wrapper.console.title="WSO2 Carbon"
#********************************************************************
# Wrapper Windows Service and Posix Daemon Properties
#********************************************************************
# Name of the service
wrapper.ntservice.name="WSO2CARBON"
# Display name of the service
wrapper.ntservice.displayname="WSO2 Carbon"
# Description of the service
wrapper.ntservice.description="Carbon Kernel"
#********************************************************************
# Wrapper System Tray Properties
#********************************************************************
# enable system tray
wrapper.tray = true
# TCP/IP port. If none is defined multicast discovery is used to find the
# port
# Set the port in case multicast is not possible.
wrapper.tray.port = 15002
#********************************************************************
# Exit Code Properties
#********************************************************************
# Restart on non zero exit code
#********************************************************************
wrapper.on_exit.0=SHUTDOWN
wrapper.on_exit.default=RESTART
#********************************************************************
# Trigger actions on console output
#********************************************************************
# On Exception show message in system tray
wrapper.filter.trigger.0=Exception
wrapper.filter.script.0=scripts\trayMessage.gv
wrapper.filter.script.0.args=Exception
#********************************************************************
# genConfig: further Properties generated by genConfig
#********************************************************************
placeHolderSoGenPropsComeHere=
wrapper.java.command = ${java_home}\bin\java
wrapper.java.classpath.1 = ${java_home}\lib\tools.jar
wrapper.java.classpath.2 = $(carbon_home)\bin\*.jar
wrapper.app.parameter.1 = org.wso2.carbon.bootstrap.Bootstrap
wrapper.app.parameter.2 = RUN
wrapper.java.additional.1 =
-XXbootclasspath/a:${carbon_home}\lib\xboot\*.jar
wrapper.java.additional.2 = -Xms256m
wrapper.java.additional.3 = -Xmx1024m
wrapper.java.additional.4 = -XX:MaxPermSize=256m
wrapper.java.additional.5 = -XX:+HeapDumpOnOutOfMemoryError
wrapper.java.additional.6 =
-XX:HeapDumpPath=${carbon_home}\repository\logs\heap-dump.hprof
wrapper.java.additional.7 =
-Dcom.sun.management.jmxremote
wrapper.java.additional.8 =
-Djava.endorsed.dirs=${carbon_home}\lib\endorsed;${java_home}\jre\lib\endorsed
wrapper.java.additional.9 =
-Dcarbon.registry.root=\/
wrapper.java.additional.10 =
-Dcarbon.home=${carbon_home}
wrapper.java.additional.11 =
-Dwso2.server.standalone=true
wrapper.java.additional.12 =
-Djava.command=${java_home}\bin\java
wrapper.java.additional.13 =
-Djava.io.tmpdir=${carbon_home}\tmp
wrapper.java.additional.14 =
-Dcatalina.base=${carbon_home}\lib\tomcat
wrapper.java.additional.15 =
-Djava.util.logging.config.file=${carbon_home}\repository\conf\log4j.properties
wrapper.java.additional.16 =
-Dcarbon.config.dir.path=${carbon_home}\repository\conf
wrapper.java.additional.17 =
-Dcarbon.logs.path=${carbon_home}\repository\logs
wrapper.java.additional.18 =
-Dcomponents.repos=${carbon_home}\repository\components\plugins
wrapper.java.additional.19 =
-Dconf.location=${carbon_home}\repository\conf
wrapper.java.additional.20 =
-Dcom.atomikos.icatch.file=${carbon_home}\lib\transactions.properties
wrapper.java.additional.21 =
-Dcom.atomikos.icatch.hide_init_file_path=true
wrapper.java.additional.22 =
-Dorg.apache.jasper.runtime.BodyContentImpl.LIMIT_BUFFER=true
wrapper.java.additional.23 =
Setting up CARBON_HOME

Extract the Carbon-based product that you want to run as a Windows service, and then set the Windows environment variable CARBON_HOME to the extracted product directory location. For example, if you want to run ESB 4.5.0 as a Windows service, you would set CARBON_HOME to the extracted wso2esb-4.5.0 directory.

Running the product in console mode

You will now verify that YAJSW is configured correctly for running the Carbon-based product as a Windows service.

1. Open a Windows command prompt and go to the <YAJSW_HOME>/bat/ directory. For example:

   ```
   cd C:\Documents and Settings\yajsw_home\bat
   ```

2. Start the wrapper in console mode using the following command:

   ```
   runConsole.bat
   ```

   For example:

   ```
   C:\Documents and Settings\yajsw_home\bat>runConsole.bat
   ```

If the configurations are set properly for YAJSW, you will see console output similar to the following and can now access the WSO2 management console from your web browser via https://localhost:9443/carbon.
Working with the WSO2CARBON service

To install the Carbon-based product as a Windows service, execute the following command in the `<YAJSW_HOME>/bat` directory:

```
installService.bat
```

The console will display a message confirming that the WSO2CARBON service was installed.

To start the service, execute the following command in the same console window:

```
startService.bat
```

The console will display a message confirming that the WSO2CARBON service was started.
To stop the service, execute the following command in the same console window:

```
stopService.bat
```

The console will display a message confirming that the WSO2CARBON service has stopped.

To uninstall the service, execute the following command in the same console window:

```
uninstallService.bat
```

The console will display a message confirming that the WSO2CARBON service was removed.
Installing as a Linux Service

Follow the sections below to run a WSO2 product as a Linux service:

- **Prerequisites**
- Setting up CARBON_HOME
- Running the product as a Linux service

**Prerequisites**

Install JDK 1.6.24 or later or 1.7.* and set up the JAVA_HOME environment variable.

**Setting up CARBON_HOME**

Extract the WSO2 product to a preferred directory in your machine and set the environment variable CARBON_HOME to the extracted directory location.

**Running the product as a Linux service**

1. To run the product as a service, create a startup script and add it to the boot sequence. The basic structure of the startup script has three parts (i.e., start, stop and restart) as follows:
#!/bin/bash

case "" in
  start)
    echo "Starting the Service"
    ;;
  stop)
    echo "Stopping the Service"
    ;;
  restart)
    echo "Restarting the Service"
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
esac

Given below is a sample startup script. <PRODUCT_HOME> can vary depending on the WSO2 product's directory.

#!/bin/sh
export JAVA_HOME="/usr/lib/jvm/jdk1.7.0_07"

startcmd='"<PRODUCT_HOME>/bin/wso2server.sh start > /dev/null &"'
restartcmd='"<PRODUCT_HOME>/bin/wso2server.sh restart > /dev/null &"'
stopcmd='"<PRODUCT_HOME>/bin/wso2server.sh stop > /dev/null &"

case "$1" in
  start)
    echo "Starting the WSO2 Server ..."
    su -c "${startcmd}" user1
    ;;
  restart)
    echo "Re-starting the WSO2 Server ..."
    su -c "${restartcmd}" user1
    ;;
  stop)
    echo "Stopping the WSO2 Server ..."
    su -c "${stopcmd}" user1
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
esac

In the above script, the server is started as a user by the name user1 rather than the root user. For example, su -c "${startcmd}" user1

2. Add the script to /etc/init.d/ directory.
If you want to keep the scripts in a location other than `/etc/init.d/` folder, you can add a symbolic link to the script in `/etc/init.d/` and keep the actual script in a separate location. Say your script name is `prodserver` and it is in `/opt/WSO2/` folder, then the commands for adding a link to `/etc/init.d/` is as follows:

- Make executable: `sudo chmod a+x /opt/WSO2/prodserver`
- Add a link to `/etc/init.d/`: `sudo ln -snf /opt/WSO2/prodserver /etc/init.d/prodserver`

3. Install the startup script to respective runlevels using the command `update-rc.d`. For example, give the following command for the sample script shown in step1:

```
sudo update-rc.d prodserver defaults
```

The `defaults` option in the above command makes the service to start in runlevels 2, 3, 4, and 5 and to stop in runlevels 0, 1, and 6.

A runlevel is a mode of operation in Linux (or any Unix-style operating system). There are several runlevels in a Linux server and each of these runlevels is represented by a single digit integer. Each runlevel designates a different system configuration and allows access to a different combination of processes.

4. You can now start, stop and restart the server using `service <service name> {start|stop|restart}` command. You will be prompted for the password of the user (or root) who was used to start the service.

### Running the Product

To run WSO2 products, you start the product server at the command line. You can then access the management console to configure and manage the product.

- Before you begin
- Starting the Server
- Accessing the Management Console
- Stopping the Server

**Before you begin**

Before you start the server, note the following:

- The `config-validation.xml` file in the `<PRODUCT_HOME>/repository/conf/etc` directory contains a list of recommended system parameters, which are validated against your system when the server starts. See Configuring `config-validation.xml` for details on modifying these parameters before starting the server.
- The Management Console uses the default HTTP-NIO transport, which is configured in the `catalina-servers.xml` file in the `<PRODUCT_HOME>/repository/conf/tomcat` directory. This transport must be properly configured in this file for the Management Console to be accessible.

**Starting the Server**

To start the server, you run the script `wso2server.bat` (on Windows) or `wso2server.sh` (on Linux/Solaris) from the `bin` folder. Alternatively, you can install and run the server as a Windows service.
To start and stop the server in the background mode of Linux, run `wso2server.sh start` and `wso2server.sh stop` commands.

1. Open a command prompt:
   - On Windows, choose **Start -> Run**, type `cmd` at the prompt, and press Enter.
   - On Linux/Solaris, establish a SSH connection to the server or log into the text Linux console.
2. Execute one of the following commands, where `<PRODUCT_HOME>` is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td><code>&lt;PRODUCT_HOME&gt;/bin/wso2server.bat --run</code></td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td><code>sh &lt;PRODUCT_HOME&gt;/bin/wso2server.sh</code></td>
</tr>
</tbody>
</table>

The operation log appears. When the product server is running, the log displays the message "WSO2 Carbon started in ‘n’ seconds”.

### Starting the server as a Worker Node:

If you want to provide access to the production environment without allowing any user group (including admin) to log in to the management console, you must start the product as a worker node. You can enable the worker node for your product in two ways:

1. Use the `-DworkerNode` startup command when you start the product as shown below:
   - **On Windows**: `<PRODUCT_HOME>/bin/wso2server.bat --run -DworkerNode`
   - **On Linux/Solaris**: `sh <PRODUCT_HOME>/bin/wso2server.sh -DworkerNode`

2. You can set permanently set the `-DworkerNode` system property to ‘true’ in your product startup script. When you execute the product startup script, the worker profile will be started automatically.

   ```
   '-DworkerNode=false'
   ```

Note that this profile only consists of the back-end features of the Application Server and does not include stubs or any UI features. However, in WSO2 AS, if you require any stubs to be activated in addition to the back-end features when you run the worker profile, you can manually add the relevant JARs to the web application's `lib` folder before starting the server. For information about other profiles available for AS, see [Running the Product on a Preferred Profile](#).

If you want to check any additional options available to be used with the startup commands, type `-help` after the command, such as: `sh <PRODUCT_HOME>/bin/wso2server.sh -help`.

The operation log appears. When the product server is running, the log displays the message "WSO2 Carbon started in ‘n’ seconds”.

### Accessing the Management Console

Once the server has started, you can run the Management Console by opening a Web browser and typing in the management console's URL. The URL is displayed as the last line in the start script's console and log. For example:
The URL should be in the following format: https://<Server Host>:9443/carbon

You can use this URL to access the Management Console on this computer from any other computer connected to the Internet or LAN. When accessing the Management Console from the same server where it's installed, you can type "localhost" instead of the IP address: https://localhost:9443/carbon

At the sign-in screen, sign in to the Management Console using admin as both the username and password. You can then use the Management Console to manage the product. The tabs and menu items in the navigation pane on the left may vary depending on the features you have installed. For example, following is the Management Console for WSO2 Carbon:

![Management Console](image)

To view information about a particular page, click the Help link in the top right corner of that page, or click the Docs link to open this documentation for full information on managing the product.

When the Management Console Sign-in page appears, the web browser will typically display an "insecure connection" message, which requires your confirmation before you can continue.

The Management Console is based on the HTTPS protocol, which is a combination of HTTP and SSL protocols. This protocol is generally used to encrypt the traffic from the client to server for security reasons. The certificate it works with is used for encryption only, and does not prove the server identity, so when you try to access the Management Console, a warning of untrusted connection is usually displayed. To continue working with this certificate, some steps should be taken to "accept" the certificate before access to the site is permitted. If you are using the Mozilla Firefox browser, this usually occurs only on the first access to the server, after which the certificate is stored in the browser database and marked as trusted. However, with other browsers the insecure connection warning might be displayed every time you access the server.

This scenario is suitable for testing purposes, or for running the program on the company's internal networks. If you want to make the Management Console available to external users, your organization should obtain a certificate signed by a well-known certificate authority, which verifies that the server actually has the name it is accessed by and that this server belongs to the given organization.
If you leave the Management Console unattended, the session will time out. The default timeout value is 15 minutes, but you can change this in the `<PRODUCT_HOME>/repository/conf/tomcat/carbon/WEB-INF/web.xml` file as follows:

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

**Restricting access to the management console and web applications:**

You can restrict access to the management console of your product by binding the management console with selected IP addresses. Note that you can either restrict access to the management console only, or you can restrict access to all web applications in your server as explained below.
To control access only to the management console, add the IP addresses to the `<PRODUCT_HOME>/repository/conf/tomcat/carbon/META-INF/context.xml` file as follows:

```xml
<Valve
    className="org.apache.catalina.valves.RemoteAddrValve"
    allow="<IP-address-01>|<IP-address-02>|<IP-address-03>"/>
```

The RemoteAddrValve Tomcat valve defined in this file will only apply to the Carbon management console, and thereby all outside requests to the management console will be blocked.

To control access to all web applications deployed in your server, add the IP addresses to the `<PRODUCT_HOME>/repository/conf/context.xml` file as follows:

```xml
<Valve
    className="org.apache.catalina.valves.RemoteAddrValve"
    allow="<IP-address-01>|<IP-address-02>|<IP-address-03>"/>
```

The RemoteAddrValve Tomcat valve defined in this file will apply to each web application hosted on the Carbon server. Therefore, all outside requests to any web application will be blocked.

You can also restrict access to particular servlets in a web application by adding a Remote Address Filter to the `web.xml` file (stored in the `<PRODUCT_HOME>/repository/conf/tomcat/` directory), and by mapping that filter to the servlet url. In the Remote Address Filter that you add, you can specify the IP addresses that should be allowed to access the servlet.

The following example from a `web.xml` file illustrates how access to the management page (`/carbon/admin/login.jsp`) is granted only to one IP address:

```xml
<filter>
    <filter-name>Remote Address Filter</filter-name>
</filter>
<filter-class>org.apache.catalina.filters.RemoteAddrFilter</filter-class>

    <init-param>
        <param-name>allow</param-name>
        <param-value>127.0.0.1</param-value>
    </init-param>
</filter>

<filter-mapping>
    <filter-name>Remote Address Filter</filter-name>
    <url-pattern>/carbon/admin/login.jsp</url-pattern>
</filter-mapping>
```

Note: Any configurations (including valves) defined in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file applies to all web applications and is globally available across server, regardless of host or cluster. See the official Tomcat documentation for more information about using [remote host filters](https://tomcat.apache.org/).
Stopping the Server

To stop the server, press Ctrl+C in the command window, or click the Shutdown/RuStart link in the navigation pane in the Management Console.
User Guide

This section provides information about the features, functionality, solution development, testing and debugging options of the WSO2 Application Server.

- **Application Development and Deployment**
- **Services Development and Deployment**

Application Development and Deployment

WSO2 Application Server is built with many features to facilitate the hosting, deployment and management of various applications. The following diagram depicts the process that you need to follow when working with applications in WSO2 AS:

![Diagram of Application Development and Deployment process]

See the following topics for details of the above:

- Developing applications for WSO2 AS
- Deploying applications in WSO2 AS
- Managing applications deployed in WSO2 AS
- Connecting to applications deployed in WSO2 AS

Developing applications for WSO2 AS

Applications that can be deployed in WSO2 AS include **Web Applications** (generic web applications, JAX-RS/JAX-WS applications) and **Jaggery Applications**. The following topics explain the different types of applications deployable in WSO2 AS and some instructions and configurations that should be followed when developing them:

- Developing Web Applications
- Developing Jaggery Applications

Note that you can conveniently create applications using WSO2 Developer Studio. If you are already done with building your application, you can directly proceed to the next step, which is deploying applications in AS.

Configuring WSO2 AS for applications
WSO2 AS inherits many features from the Carbon platform, which allows you the flexibility of configuring your server to provide the necessary support for your web applications. See the section on working with web applications in the administration guide to find information on the server settings that will directly apply to your applications.

Deploying applications in WSO2 AS

Once you are done with developing your application, you can easily deploy it in WSO2 AS using the management console. See the topic on deploying applications in AS for step-by-step instructions.

Managing applications deployed in WSO2 AS

In the management console of WSO2 AS, you will find a separate dashboard for each of the deployed applications for monitoring statistics. You can also use WSO2 BAM with WSO2 AS, for more advanced monitoring capabilities. See the topic on managing and monitoring applications in AS for detailed information.

Connecting to applications deployed in WSO2 AS

WSO2 AS supports a variety of transports for connecting with applications deployed in AS. Among these, the HTTP and HTTPS servlet transports are used by the applications hosted on AS, to connect with other clients and applications.

- By default WSO2 AS is configured with the message listener and message sender implementations that are necessary for sending and receiving messages through the HTTP/HTTPS protocol.
- The transport receiver of the HTTP protocol is implemented in the Carbon Core component.
- The transport sender is implemented as a Tomcat connector in the catalina-server.xml file (stored in the <PRODUCT_HOME>/repository/conf/tomcat/ folder).

Developing Web Applications

Web components packaged in Web Application Archives (WAR) can be deployed in WSO2 AS as web applications. This includes any generic web application, JAX-RS/JAX-WS application or Spring application that is packaged into WAR files. For more information on WAR files, please refer to http://ant.apache.org/manual/Tasks/war.html.

Note that it is not allowed to have Web applications named 't.war'.

The following topics explains these different types of web applications deployable in AS, and how they should be developed for WSO2 AS:

- **Generic web applications**
- **JAX-RS/JAX-WS applications**
- **Spring applications**

Generic web applications

Once you have developed your web application, see that the following is in place before you build the .war file:

1. The class files and the web.xml file should be in the archive directory. The hierarchical directory structure of the web archive for a generic web application is shown below.
2. If you want to configure your web application by enabling SaaS mode or CORS, see the topic on Configuring Applications for AS.

3. If you want to use more advanced configurations for your web application, such as enabling Single Sign On (SSO), enabling fine grained authorization using XACML, see the Working with Applications topic.

**JAX-RS/JAX-WS applications**

JAX-WS and JAX-RS are popular Java APIs for implementing XML-based web applications and RESTful web applications respectively. WSO2 AS uses Apache CXF as the framework for deploying and managing JAX-WS/JAX-RS web applications. You can easily develop JAX-WS/JAX-RS web applications using WSO2 Developer Studio.

The following topics explain in detail how you can use the JAX-WS/JAX-RS specifications to develop and secure your applications:

- Developing JAX-RS Applications
- Developing JAX-WS Applications

**Spring applications**

See the following topics for information:

**Spring dependencies**

- Only the following subset of the spring packages (Spring Framework version 3.1.0) are delivered with AS:  
  - org.springframework.ws spring-xml (version : 2.0.3.RELEASE)  
  - org.springframework spring-core (version : 3.1.0)  
  - org.springframework spring-beans (version : 3.1.0)  
  - org.springframework spring-context (version : 3.1.0)  
  - org.springframework spring-web (version : 3.1.0)  

- If the application needs any other spring packages, they need to be deployed with the application itself.
- Projects that are built with Spring 2.5 or higher versions will be backward compatible with delivered version of Spring Framework. However, it is recommended to build your application with Spring 3.1.0 before deployment.
- If the application is based on a higher version of the Spring framework, then that version needs to be deployed with the application. The framework which is deployed with the application will be given the priority in the run time.

**CXF environment**

- The following sub packages of Spring 3.0.7 are shipped with the Application Server CXF environment.  
  - spring-aop  
  - spring-asm
JAX-WS and JAX-RS Support

JAX-WS and JAX-RS services are supported in WSO2 AS by the Apache CXF framework as explained below. To enable this framework, the following feature should be installed in your product.

**Name:** WSO2 Carbon - JAX-WS/JAX-RS Webapp Management Feature

**Identifier:** org.wso2.carbon.jaxws.webapp.mgt.feature.group

Note that this feature is enabled in WSO2 AS by default.

For information on how these services are implemented and deployed in AS, see the following topics: Developing JAX-WS services, Developing JAX-RS services and Deploying Applications in AS.

**CXF framework in WSO2 AS**

CXF (http://cxf.apache.org) is an open source services framework for developing services using JAX-WS and JAX-RS (including other APIs). There are numerous benefits of using the Apache CXF framework for web service development. In addition to JAX-WS and JAX-RS, Apache CXF also supports Spring integration and a variety of web service specifications, including WS-Addressing, WS-Policy, WS-ReliableMessaging and WS-Security. It also comes with a business-friendly Apache license.

WSO2 Application Server had the ability to host CXF-based applications in previously releases as well. However, because of the integration that now comes with this feature, there is more flexibility and you can use features of the Carbon platform within CXF applications. Further, this feature enables CXF users to directly deploy their web applications in a running Carbon instance in a few easy steps. That is, you do not have to provide any CXF libraries since they are already provided by the WSO2 Carbon OSGi environment. The user only has to focus on the business logic of the applications while the Carbon platform takes care of deployment.

In addition to the significant ease in hosting and managing, CXF-based applications deployed in a Carbon instance automatically inherits all the underlying capabilities of the platform. Some of them are listed below.

- Ability to use the Carbon user store through CarbonContext.
- Ability to use WSO2 Registry APIs through CarbonContext.
- Fine-grained authorization through WSO2 Identity Server features.
- All benefits inherited from OSGi.
- User-friendly management console for web application management.
- Hot-deployment and hot-update of CXF-based web applications.

**Developing JAX-RS Applications**

JAX-RS is an annotation-driven Java API that can be used for exposing Java beans as HTTP based services. For example, RESTful web applications are developed using JAX-RS annotations to define the resources exposed by the service. Web applications developed using JAX-RS annotations consist of the following qualities:
Use of HTTP/S transport protocols for message transfer.
The content that is transported by a JAX-RS web service can be in any format, e.g., XML, JSON, TEXT etc.
Security can be enabled for a JAX-RS web application either at the HTTP Header level, or by using TLS protocol.

Use WSO2 Developer Studio to easily develop your JAX-WS/JAX-RS Application. It will generate the needed Web application structure and you only have to concentrate on annotations and the business logic.

See the following topics for information on JAX-RS applications in WSO2 AS:

- Working with JAX-RS annotations
- Configuring JAX-RS applications for AS
  - Securing JAX-RS applications
  - Other configuration options
- Building JAX-RS applications for AS

Working with JAX-RS annotations

JAX-RS annotations are defined in JSR 311. They are also a way of mapping Java with HTTP requests. For example, @GET, @PUT, @POST, @DELETE and @HEAD are JAX-RS annotations that directly map to HTTP requests by the same name. See the official Java documentation, for details about JAX-RS annotations and RESTful web services.

Shown below is an example of a simple web application that uses the following JAX-RS annotations: @Path, @GET, @Consumes and @Produces.

```java
@Path("/hello")
public class HelloWorldService {
    @GET
    @Consumes("text/plain")
    @Produces("text/xml")
    @Path("/{user}")
    public String hello(@PathParam("user") String user) {   }
}
```

In this example, the @GET annotation is used to specify how the web application should respond to HTTP GET requests coming from a client. The @Consumes annotation determines the media type that the web service can accept from the client and the @Produces annotation determines the media type that the web service can return to the client in response to the request.

Configuring JAX-RS applications for AS

The following sections explain some of the configuration options that are available for JAX-RS applications:

- Securing JAX-RS applications
- Other configuration options

Securing JAX-RS applications

When you define JAX-RS applications, you can enable authentication and authorization for your application based on the users and roles that are defined in the AS user store. This is done by using the @RolesAllowed annotation in your web service to map user roles that are already defined in the user store.
The following steps demonstrate how a JAX-RS application is secured by filtering the users that can access the service:

1. For authentication, you should first enable the basic-auth security constraint in the web.xml file of your web application. In AS, web applications are authenticated against the Carbon user stores. The Working with Users, Roles and Permissions section explains how user stores are set up and configured. By default, WSO2 Carbon products come with a default JDBC-based user store, and do not use tomcat-users.xml (which comes by default with Apache Tomcat). This allows the web application to be authenticated against user stores such as LDAP, ActiveDirectory etc. So, when you set BASIC auth as a security-constraint via web.xml, the users will be authenticated against the users and roles set in a Carbon user store. See the following example:

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>CXF Jax-RS security</web-resource-name>
    <url-pattern>/services/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>admin</role-name>
    <role-name>newrole</role-name>
  </auth-constraint>
</security-constraint>
```

2. You must add a cxf interceptor in the cxf-servlet.xml file. The roles you set using the @RolesAllowed annotation is authorized by CXF using the request interceptor, org.apache.cxf.interceptor.security.SecureAnnotationsInterceptor. CXF receives the underlying container's SecurityContext for this task. Since we have set the Carbon user store as the default user realm, CXF will use that. See the following example:
3. Then, you can go ahead and add the relevant annotation to the needed resource methods in your JAX-RS resource class. See the following examples:

- **@RolesAllowed** annotation: The required roles for authentication and authorization should be specified for the application. In the example given below, only the users with the “admin” role will be able to access the resource method, “getCustomer”.

```java
@GET
@RolesAllowed("admin")
@Produces("application/xml")
@Path("/customers/{id}/")
public Customer getCustomer(@PathParam("id") String id)
{         System.out.println("----invoking getCustomer, Customer id is: " + id);long idNumber = Long.parseLong(id);Customer c =
customers.get(idNumber); return c;}
```

- **@DenyAll** annotation: Block all users in the user store from accessing the service.
3.4. The annotations such as @RolesAllowed comes from the jsr250-api Common Annotations library. This is not shipped with WSO2 AS by default. So, you need to either include this library in your web application, or directly copy it to the AS. Since this is for a CXF JAX-RS application, you may copy this library to `<AS_HOME>/lib/runtimes/cxf/` folder. The Maven dependency information for jsr250-api library is as follows:

```xml
<dependency>
  <groupId>javax.annotation</groupId>
  <artifactId>jsr250-api</artifactId>
  <version>1.0</version>
</dependency>
```

Shown below is a sample security configuration (web.xml file and cxf-servlet.xml file) for a JAX-RS service.

Sample web.xml file:
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
<display-name>JAX-WS/JAX-RS Webapp</display-name>
<Servlet>
<servlet-name>JAXServlet</servlet-name>
<display-name>JAX-WS/JAX-RS Servlet</display-name>
<description>JAX-WS/JAX-RS Endpoint</description>
<servlet-class>
org.apache.cxf.transport.servlet.CXFServlet
</servlet-class>
<init-param>
<param-name>service-list-stylesheet</param-name>
<param-value>servicelist.css</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>JAXServlet</servlet-name>
<url-pattern>/services/*</url-pattern>
</servlet-mapping>
<session-config>
<session-timeout>60</session-timeout>
</session-config>
<security-constraint>
<web-resource-collection>
<web-resource-name>CXF Jax-RS security</web-resource-name>
<url-pattern>/services/*</url-pattern>
</web-resource-collection>
<auth-constraint>
<role-name>admin</role-name>
<role-name>newrole</role-name>
</auth-constraint>
</security-constraint>
<login-config>
<auth-method>BASIC</auth-method>
<realm-name>default</realm-name>
</login-config>
</web-app>

Sample cxf-servlet.xml file:
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:jaxrs="http://cxf.apache.org/jaxrs"
      xsi:schemaLocation="
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
          http://cxf.apache.org/jaxrs
          http://cxf.apache.org/schemas/jaxrs.xsd">
    <bean id="serviceBean" class="demo.jaxrs.server.CustomerService"/>

    <!-- The SecureAnnotationsInterceptor honors the @RolesAllowed,
    @PermitAll and @DenyAll annotations -->
    <bean id="authorizationInterceptor" class="org.apache.cxf.interceptor.security.SecureAnnotationsInterceptor">
        <property name="securedObject" ref="serviceBean"/>
    </bean>

    <jaxrs:server id="customerService" address="/customers">
        <!-- set the interceptor for the jaxrs:server for in-bound messages
        to authorize the user -->
        <jaxrs:inInterceptors>
            <ref bean="authorizationInterceptor"/>
        </jaxrs:inInterceptors>
        <jaxrs:serviceBeans>
            <ref bean="serviceBean"/>
        </jaxrs:serviceBeans>
    </jaxrs:server>
</beans>

Other configuration options

- If you want to enable SaaS mode or CORS for your web applications see Configuring Applications for AS.
- If you want to use more advanced configurations for your web application, such as enabling Single Sign On (SSO), enabling fine grained authorization using XACML, see the topics listed under Working with Applications.

Building JAX-RS applications for AS

JAX-WS and JAX-RS applications can be bundled in a CXF application. Given below is the general folder structure of a CXF application. However, this can change depending on your application.

Please see the instructions on class loading for web applications, which explains how you can add the webapp-classloading.xml file to your web app. This XML is a custom file implemented by WSO2 to give flexibility in classloading. You must have this file in order to deploy JAX-WS and JAX-RS applications, because the CXF runtime is not visible to the webapps by default.
Developing JAX-WS Applications

JAX-WS is the Java API for XML-Based Web Services. It provides a simple method of developing Web applications with a high level of portability and is defined using Java annotations without any deployment descriptors. JAX-WS 2.0 specification (http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index3.html) is defined on top of the JAX-RPC 1.1 (http://jcp.org/aboutJava/communityprocess/final/jsr101/index2.html), which defines a standard set of APIs for RPC oriented Web applications for Java. RPC is not widely used anymore. JAX-WS also delegates all data binding aspects to JAXB (http://jcp.org/aboutJava/communityprocess/mrel/jsr222/index2.html).

JAX-WS development is dependent on Java annotations defined in JSR-181 (Web Services Metadata for the Java Platform), JAX-WS and JAXB. These annotations are used to let the JAX-WS runtime know about the nature of the Web application interface. The developer can customize the Web application interface using annotations and build the service according to his requirements.

Use WSO2 Developer Studio to easily develop your JAX-WS/JAX-RS Application. It will generate the needed Web application structure and you only have to concentrate on annotations and the business logic.

- Working with JAX-WS annotations
  - Java to WSDL Development Model with JAX-WS
  - WSDL to Java Development Model with JAX-WS
  - Configuration JAX-WS applications for AS
  - Building JAX-WS applications for AS

Working with JAX-WS annotations

Listed below are the most important annotations and their member-value pairs.

@WebService

Used to mark a Java class when implementing a Web service or to mark a Java interface as defining a Web service interface.

- name - Name of the wsdl:portType in WSDL 1.1 (http://www.w3.org/TR/wsdl).
- targetNamespace - The XML namespace used for the WSDL
- serviceName - Name of the wsdl:service in WSDL 1.1. This is not allowed on endpoint interfaces.
- portName - Name of the wsdl:port in WSDL 1.1. This is not allowed on endpoint interfaces.
- wsdlLocation - A URL which points to an already existing WSDL
- endpointInterface - Complete name of the endpoint interface, when the interface and the implementation are separated. This is not allowed on endpoint interfaces.

@WebMethod
Used to customize a method which is exposed as a Web service operation.

- **opérationName** - Name of the wsdl:operation matching the particular method
- **action** - The action for this operation
- **exclude** - Used to leave a method out of the Web Service interface

@Oneway

This annotation can be used to indicate that the given method maps to a one way operation. It has only an input message and no output message.

@WebParam

Used to customize the mapping of an individual method parameter to a Web service message part and XML element.

- **name** - Name of the parameter. This property relates to the style (document or rpc) of the operation.
- **partName** - Name of the wsdl:part which represents this parameter
- **targetNamespace** - XML namespace for this parameter
- **header** - If true, the parameter is pulled from a message header rather than the message body

@WebResult

Used to customize the mapping of the return value to a Web service message part and XML element.

- **name** - Name of the return value. This property relates to the style (document or rpc) of the operation.
- **partName** - Name of the wsdl:part which represents the return value
- **targetNamespace** - XML namespace for the return value
- **header** - If true, the return value is in a message header, rather than the message body

@SOAPBinding

Specifies the mapping of the Web service onto the SOAP message protocol.

- **style** - DOCUMENT or RPC
- **use** - LITERAL or ENCODED
- **parameterStyle** - WRAPPED or BARE

JAX-WS 2.x supports the following combinations:

- DOCUMENT/LITERAL/WRAPPED (default)
- DOCUMENT/LITERAL/BARE
- RPC/LITERAL/WRAPPED

There are two main Web service development models. They are:

- Java to WSDL (code first) approach
- WSDL to Java (contract first) approach

**Java to WSDL Development Model with JAX-WS**

In this approach, development starts from the code or the service implementation and the Web application engine generates the WSDL for the developer. The developer can write his POJO service and use annotations to control the structure of the WSDL. The JAX-WS runtime generates the WSDL according to these annotations used. The advantage of using annotations is that the Web application implementation can be changed without affecting the generated WSDL contract. Therefore, JAX-WS is the standard development technique when it comes to POJO development.

Some sample POJO services using JAX-WS are as follows:

**Sample 1:**
This simple example uses the @WebService annotation to convert a simple POJO into a Web application. When you use this annotation, all the public methods in the class become operations in the application by default. If you want to exclude a particular method, you have to specify it using the 'exclude' property of the @WebMethod annotation. Also, if some method should be mapped into a one way operation, you have to use the @Oneway annotation. The sample code of this POJO is as follows:

```java
package org.wso2.jaxws.sample01;

import javax.jws.WebService;
import javax.jws.Oneway;
import javax.jws.WebMethod;

@WebService
public class SimpleSample {

    public String echo(String echoInput) {
        return echoInput;
    }

    @Oneway
    public void ping(String pingInput) {
        System.out.println("Ping : " + pingInput);
    }

    @WebMethod(exclude = true)
    public String hideMethod(String hideInput) {
        return "Hiding : " + hideInput;
    }

}
```

Sample 2:

This sample service (StudentMarksService) contains two operations called 'computeAverage' and 'computeHighestMarks'. Both these methods get a Student object as a parameter. The sample code is as follows, showing how you can customize the WSDL service interface using annotations.

```java
StudentMraksService.java class

package org.wso2.jaxws.sample02;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;
import javax.jws.WebResult;
import javax.jws.soap.SOAPBinding;

@WebService(name = "StudentMarks",
            serviceName = "StudentMarksService",
            targetNamespace = "http://wso2.org/jaxws/sample02")
@SOAPBinding(
    name = "StudentMarks",
    serviceName = "StudentMarksService",
    targetNamespace = "http://wso2.org/jaxws/sample02"
)
```
style = SOAPBinding.Style.DOCUMENT,
       use = SOAPBinding.Use.LITERAL,
       parameterStyle = SOAPBinding.ParameterStyle.WRAPPED
    )

public class StudentMarksService {

    @WebMethod(
        operationName = "computeAverage",
        action = "urn:getAverage"
    )
    @WebResult(
        name = "average",
        targetNamespace = "http://wso2.org/jaxws/sample02"
    )
    public double getAverage(
        @WebParam(name = "student", targetNamespace = "http://wso2.org/jaxws/sample02")
        Student student) {

        double totalMarks = 0;
        for (int i = 0; i < student.getMarks().length; i++) {
            totalMarks += student.getMarks()[i];
        }
        return totalMarks / student.getMarks().length;
    }

    @WebMethod(
        operationName = "computeHighestMarks",
        action = "urn:getHighestMarks"
    )
    @WebResult(
        name = "highest",
        targetNamespace = "http://wso2.org/jaxws/sample02"
    )
    public int getHighestMarks(
        @WebParam(name = "student", targetNamespace = "http://wso2.org/jaxws/sample02")
        Student student) {

        int highest = 0, temp;
        for (int i = 0; i < student.getMarks().length; i++) {
            temp = student.getMarks()[i];
            if (temp > highest) {
                highest = temp;
            }
        }
        return highest;
    }

}
Student.java class

```java
package org.wso2.jaxws.sample02;

public class Student {
    private String name;
    private int age;
    private int[] marks;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int[] getMarks() {
        return marks;
    }

    public void setMarks(int[] marks) {
        this.marks = marks;
    }
}
```

You can simply compile these sample classes, create .jar archives and deploy them in a running Carbon instance.

**WSDL to Java Development Model with JAX-WS**

In this method, the WSDL contract is used to generate the application’s skeleton and other binding classes through the usage of some WSDL to Java tool (Look under the Tools section for more information). The tool generates the Service Endpoint Interface (SEI) and the JAXB binding classes according to the WSDL and the schema. Service developer only has to write the application class by implementing the SEI.

A WSDL to Java sample using JAX-WS is as follows:

**Sample 1:**
The same "StudentMarksService" mentioned above is used here.

First, generate the code using some WSDL to Java tool. Create a folder (Ex: wsdl2java) to place your generated source files. Next step is to write the application implementation class. Use some IDE (e.g., Developer Studio) and create a project using the generated classes. Then write the application class by implementing the SEI. Use the @WebService annotation to map the properties according to your WSDL and make sure you point to your WSDL file using the 'wsdlLocation' property. Service class is as follow:

```java
package org.wso2.jaxws.sample02.service;

import org.wso2.jaxws.sample02.StudentMarksPort;
import org.wso2.jaxws.sample02.Student;

import javax.jws.WebService;

@WebService(
    serviceName = "StudentMarksService",
    portName = "StudentMarksPort",
    targetNamespace = "http://wso2.org/jaxws/sample02",
    endpointInterface = "org.wso2.jaxws.sample02.StudentMarksPort",
    wsdlLocation = "StudentMarksService.wsdl"
)
public class StudentMarksImpl implements StudentMarksPort {

    public double computeAverage(Student student) {
        double totalMarks = 0;
        for (int i = 0; i < student.getMarks().size(); i++) {
            totalMarks += student.getMarks().get(i);
        }
        return totalMarks / student.getMarks().size();
    }

    public int computeHighestMarks(Student student) {
        int highest = 0, temp;
        for (int i = 0; i < student.getMarks().size(); i++) {
            temp = student.getMarks().get(i);
            if (temp > highest) {
                highest = temp;
            }
        }
        return highest;
    }
}
```

Compile your project and create the .jar archive to be deployed. Note that you have to include your WSDL file at the root level of your .jar archive.

For more information on these operation styles, see the following articles on non JAX-WS contract-first development with Apache Axis2.

- Contract First Development with Axis2 (Part 1)
- Contract First Development with Axis2 (Part 2)
Configuration JAX-WS applications for AS

- If you want to enable SaaS mode or CORS for your web applications see Configuring Applications for AS.
- If you want to use more advanced configurations for your web application, such as enabling Single Sign On (SSO), enabling fine grained authorization using XACML, see the topics listed under Working with Applications.

Building JAX-WS applications for AS

JAX-WS and JAX-RS applications can be bundled in a CXF application. Given below is the general folder structure of a CXF application. However, this can change depending on your application.

Please see the instructions on class loading for web applications, which explains how you can add the webapp-classloading.xml file to your web app. This XML is a custom file implemented by WSO2 to give flexibility in classloading. You must have this file in order to deploy JAX-WS and JAX-RS applications, because the CXF runtime is not visible to the webapps by default.

```
HelloworldService.war
WEB-INF/
    web.xml
    cxf-servlet.xml
    wsdl
    lib/
    classes/
META-INF/
    webapp-classloading.xml
```

Versioning Web Applications

Starting from Application Server 5.2.1, AS provides support for webapp versioning. With webapp versioning, you can have two webapps with the same context-prefix, but with different context-post-fixes set as the versions. For example, you can have two webapps with contexts as shown below:

```
helloworld/1.0
helloworld/2.0
helloworld
example/version1
```

We have facilitated this feature using a special convention for the webapp file name. We have closely followed the Tomcat Context Naming Convention, but the concepts used are different. With Apache Tomcat, the web application versions refers to internal versioning with the same context path for both the web applications. But with Application Server, what we have implemented is a way where the context path is modified to mention the version. If you read the above mentioned article, the naming convention we follow is pretty similar to the use of single hash symbol ‘#’ in webapp name.

To use webapp versioning with AS, set web application (war) file name similar to following examples mentioned in
Developing Jaggery Applications

Jaggery is a framework to write Web applications and HTTP-focused Web services purely in JavaScript, for all aspects of the application: front-end, server-side logic, communication and persistence. It is 100% open source and delivered under Apache license 2.0.

There is a myriad of different front-end and back-end technologies with multiple levels of data communication involved in Web development today. The biggest challenge in working with a pool of different technologies is preserving information integrity and consistency when it is passed through and converted from one technology to another. Jaggery successfully addresses most of these challenges.

The most powerful use case of Jaggery is handling any part of a Web application development through a single JavaScript framework - a single language and a single data format. This eliminates mismatches across different layers introduced by data conversion, and enables to create both applications and APIs at the same time, bridging the gap between Web applications and HTTP-focused Web services.

Given below are some of the main features and capabilities of Jaggery.

- Jaggery is developed on top of the component-based WSO2 Carbon platform, which is the award-winning enterprise run-time for application and Web service hosting. It consists of just the right set of components required for efficient functioning. As a result, Jaggery is lean and lightweight.
- Because of its component-based architecture, Jaggery can be easily provisioned by plugging in additional components.
- Jaggery is extremely agile and flexible. Most JavaScript application development mechanisms restrict developers to a framework-specific structure or pattern, demanding an additional learning curve. In contrast, Jaggery does not require adherence to a specific Web development standard or procedure. Users have the flexibility to write their scripts any way they want and simply execute on the Jaggery deployment engine without compilation or any kind of pre-processing. Development is as simple as writing your script and just dropping into the Jaggery framework for auto-deployment. Jaggery smoothly and seamlessly decouples developers from the framework, allowing them to simply concentrate on the business logic.
- Jaggery talks smoothly in JSON. Clients can instantly send JSON information to the server and the application is capable of understanding the data format.
- Jaggery has embedded URL pattern recognition and pattern matching libraries. Being able to handle URL patterns is a useful feature in Web application and API development.
- In-built capability to communicate with databases such as embedded H2, MySQL, Oracle etc.
- Jaggery scripts can be executed through a shell, which assists debugging.

Developing Jaggery applications

For more information on Jaggery, a quick-start guide, samples, documentation and hands-on experience of this evolving product, go to http://jaggeryjs.org.

Configuration Jaggery applications for AS

If you want to enable SaaS mode or CORS for your applications, see Configuring Applications for AS.

Deploying Applications in AS
WSO2 Application Server is a program for hosting, deploying and managing various applications. See Application Development and Deployment for an overview of the process. Once you have developed the required applications, you can deploy them in AS using the management console as explained in the following instructions.

First, start WSO2 Application Server and log in to the management console. You can then deploy your applications using the Manage/Applications menu in the Main tab on the navigator:

![Manage/Applications](image)

The following topics explain how different types of applications should be deployed in AS:

- Deploying web applications
- Deploying Jaggery applications

**Deploying web applications**

This topic explains the steps that you have to follow in order to deploy web applications as web archive files (WAR files). These include various types of web applications, such as generic web applications, JAX-RS/JAX-WS applications as well as Spring applications.

**Before you begin**, note that the support for web application deployment in AS is provided by the following features, which are installed in AS by default:

- The capability to deploy and manage generic web applications is provided by the following feature:
  
  **Name**: WSO2 Carbon - Webapp Management Feature  
  **Identifier**: org.wso2.carbon.webapp.mgt.feature.group

- The capability to deploy and manage JAX-RS/JAX-WS web applications is provided by the following feature:
  
  **Name**: WSO2 Carbon - JAX-WS/JAX-RS Webapp Management Feature  
  **Identifier**: org.wso2.carbon.jaxws.webapp.mgt.feature.group

Follow the instructions below to upload a Web archive file to Application Server.

1. Log in to the product's management console and click **Web Applications** under **Applications** in the navigator. The **Upload Web Applications** screen appears.

2. Browse and select your Web archive file.
3. If you are logged in as the super tenant, you can select the host for your web application. In the above example, the host is the default 'localhost'. If you have virtual hosts configured in your AS, other domains will be listed in addition to 'localhost'. This means that you can deploy the same application in multiple domains.
4. You can use the plus sign (+) to add multiple web applications.
5. Click Upload. If the file is uploaded successfully, a message appears prompting you to refresh the page. Click OK.

6. The .war file will be listed in the Running Applications page.

If the Web application is faulty, a 'Faulty Web Application Groups' link will appear in the Running Applications page. You can click the link to view the errors.

Instead of using the management console, you can manually place the .war file containing all the required files in the `<PRODUCT_HOME>/repository/deployment/server/webapps/` directory. These archive files will be deployed as web applications when the server starts.

Deploying Jaggery applications

This topic explains the steps that you have to follow in order to deploy Jaggery applications in AS.

WSO2 supports the deployment of Jaggery applications based on Apache Tomcat through the following feature in the WSO2 feature repository:

**Name:** WSO2 Carbon - Jaggery Server Feature

**Identifier:** org.wso2.carbon.jaggery.server.feature.group

The Jaggery application management feature is bundled in AS (version 5.0.0 onwards) by default.
Follow the instructions below to deploy the file in a running Carbon instance.

1. Before deploying a Jaggery application through the management console, bundle it as a '.zip' archive with all the resources and third-party libraries related to Jaggery applications.
2. Log in to the product's management console. Click Add under the Jaggery Applications menu.
3. The Upload Jaggery Applications page appears. Select your file and click Upload.

   **Upload Jaggery Applications**

   ![Upload Jaggery Applications](image)

   If you want to add multiple services at the same time, click on the '+' sign next to Browse.

4. If the Jaggery application was deployed successfully, a message appears confirming the deployment and opens the Running Applications page. For example,

   **Running Applications**

   ![Running Applications](image)

5. If the Jaggery application is faulty, a "Faulty Jaggery Application Groups" link will appear. You can click the link to view the errors.

   Instead of using the Management Console, you can manually place the .zip file in `<PRODUCT_HOME>/repository/deployment/server/jaggeryapps` and it will be deployed. Instead of creating a zip file, you can also add the resource files such as style sheets, images, java scripts etc. in the said location (preferably within a folder) along with the index.jag file. The files will be auto deployed.

**Managing and Monitoring Applications**

WSO2 products provide a range of monitoring tools to analyse statistics related to web applications (JAX-WS/JAX-RS/WebApps). The application dashboards available on the management console of WSO2 AS allows you to access the statistics of a running WSO2 AS instance. You can further extend the monitoring capabilities using the toolboxes in WSO2 BAM. See the topics below for details.

- Using the Application Dashboard
- Web Application Statistics Monitoring with BAM

**Using the Application Dashboard**

Every application that is deployed in WSO2 AS has a dedicated dashboard for the management and monitoring of
an application. See the following topics for detailed information:

- Accessing an Application Dashboard
- Using the Application Dashboard
  - Viewing the application details
  - Viewing session statistics
  - Managing session operations
  - Enabling BAM statistics monitoring for the application
  - Monitoring application statistics

**Accessing an Application Dashboard**

Follow the instructions below to access the dashboard of an application.

1. Log in to the product’s management console.
2. On the **Main** menu, under **Applications**, click **List**. The **Running Applications** page appears.
3. Select an application and click on it to access its dashboard.

Click this link to see an example of an Application Dashboard
Using the Application Dashboard

The following topics explain the various panels that are available on the application dashboard and how they can be used:

- Viewing the application details
- Viewing session statistics
- Managing session operations
- Enabling BAM statistics monitoring for the application
- Monitoring application statistics

Viewing the application details

The Application Details panel provides general information about the application, such as the Context, the name of the application, the file type, the status of the application (started or stopped) etc.

Viewing session statistics

The Session Statistics panel provides statistical information about the user sessions of the applications. It also provides a graphical view of the average system response time.

Managing session operations

The Operations panel allows you to manage the running session of an application by performing the following operations:

- Expire sessions - If you enter a value for session expiry in minutes and click Expire sessions, all sessions
of the application that have been idle for the given number of minutes or more will be expired.

- **Expire All Sessions** - When you click this link, all existing sessions of the application will be expired.
- **Reload** - This option allows you to reload the application. For example, after doing some modification to the application's business logic, it can be reloaded to the Application Server.
- **Stop** - This option allows you to stop the application running. If the web application is currently not running, then its **Operations** section has only the **Start** options.

**Enabling BAM statistics monitoring for the application**

In the **Services** panel, you will find the **Enable BAM Statistics** option, which is used to enable application monitoring via WSO2 BAM. Note that in addition to this check box, there are other configurations that you must complete in order to use the BAM statistics monitoring feature for your application. For the complete instructions, see [Web Application Statistics Monitoring with BAM](#).

**Monitoring application statistics**

In the **Statistics** panel, you will find the following statistics of a running web application. Note that these statistics are not applicable to Jaggery applications.

- **Request Count** - Total number of requests received to web applications.
- **Response Count** - Total number of responses served by the web applications.
- **Fault Count** - Total number of requests that triggered faults.
- **Maximum Response Time** - Maximum time required to serve a request
- **Minimum Response Time** - Minimum time required to serve a request
- **Average Response Time** - Average time required to serve a request.
This statistics monitoring functionality is provided by the following feature:

**Name:** WSO2 Carbon - Statistics Feature

**Identifier:** org.wso2.carbon.system.statistics.feature.group

### Web Application Statistics Monitoring with BAM

WSO2 BAM is a monitoring tool, which performs the task of aggregating, analyzing and presenting information about business activities. Thereby, BAM can be used with AS in order to effectively monitor the statistics of web applications in AS.

To use BAM with AS, you must first configure the AS settings that allow web application data from AS to be published in BAM. Also, each web application in AS, which you want to monitor in BAM should be individually configured to allow data publishing in BAM. In BAM, the `bam_webapp_statistics` default toolbox is dedicated for monitoring the web applications.

- Configuring web application statistics monitoring with BAM
- Viewing the web application statistics

Note that the statistics available on the management console of AS are only valid for each running session of the server. That is, if you restart the server, the statistics shown on the management console will reset. Therefore, if you want to have persistent statistics of web applications, you must use WSO2 BAM for monitoring statistics.
Configuring web application statistics monitoring with BAM

Perform the following steps to ensure that you have the necessary configurations for monitoring web application statistics with BAM:

- **Step 1: Start the Application Server**
- **Step 2: Start the Business Activity Monitor**
- **Step 3: Configure the Application Server (AS)**
- **Step 4: Configuring the web application that need monitoring**
- **Step 5: Configure the Business Activity Monitor (BAM)**
- **Step 6: Invoke the web application that has enabled statistics monitoring**
- **Step 7: Schedule or execute the webapp statistics**

**Step 1: Start the Application Server**

To start the application server for web application monitoring with BAM:

1. Do the following changes to the given files before you start AS.
   1. Go to `<AS_HOME>/repository/conf/` folder and open `carbon.xml`. Look for `<offset>0</offset>` and change it to 1.
   2. Go to `<AS_HOME>/repository/conf/etc/` folder and open `bam.xml`. Look for `<WebappDataPublishing>disable</WebappDataPublishing>` and enable it.
2. Start the Application Server. For more information, see Starting the server.

**Step 2: Start the Business Activity Monitor**

To start BAM and to setup databases for statistics:

1. Start BAM. For more information, see Starting the server.
2. If you need to change the database that stores statistic data, do the following:
   1. Create a database.
   2. Configure the database in `<BAM_HOME>/repository/conf/datasources/master-datasources.xml` as `WSO2BAM_DATASOURCE`.

**Step 3: Configure the Application Server (AS)**

To configure the AS for publishing statistics in BAM:

1. Log in and open the AS management console.
   
   Note that web app data publishing settings need to be configured separately for the tenants.

2. Go to the Configure tab and click Webapp Data Publishing.
3. Select the Enable webapp stats check box.
4. Leave the Stream Name, Version, Nick Name, Description as it is. These are the default values from the `bam_webapp_statistics` toolbox.

   **Note that if you change any of the details relevant to the toolbox, e.g., stream name, version etc, you must accordingly modify the hive script of the toolbox in BAM.**

5. Enter the Receiver URL.
   This refers to the Thrift URL, which is normally `tcp://{host}:7611`. If you need to use SSL for this, the URL will be `tcps://{host}:7711`. If you change the offset of the BAM server to 1, both ports will be incremented by 1 (7612, 7712).

6. Enter the Cassandra Username.
7. Enter the Cassandra Password.
8. Click **Update** to successfully configure the AS for Data publishing to BAM.

**Step 4: Configuring the web application that need monitoring**

Once you have configured your AS with a BAM tool box for monitoring statistics, you must enable monitoring on each of the web applications that you want to monitor. Use either one of the following approaches to configure the web application before deploying.

1. **Enable monitoring via the UI**
2. **Enable monitoring via web.xml**

   The UI configurations has precedence over `web.xml` configurations. As a result, if you disable web app statistics monitoring from the UI, the configurations in the `web.xml` will not work.

**Enable monitoring via the UI**

1. On the **Main** tab, under the **Applications** menu, click **List**.
2. Click the **Context** link for the respective web application.
   The selected application dashboard will appear.
3. Click **Activate** to enable BAM statistics.

Add the following context parameters to the `web.xml` file corresponding to your web application.

```xml
<context-param>
  <param-name>enable.statistics</param-name>
  <param-value>true</param-value>
</context-param>
```

**Step 5: Configure the Business Activity Monitor (BAM)**

In the previous steps, we have configured the web applications in AS as required for monitoring with BAM. Now,
follow the steps given below to install and set up the toolbox in BAM.

1. Login and open the management console of BAM.

2. On the Main menu, under BAM Toolbox, click Add.

3. Select the ToolBox From URL option and enter the following URL: http://product-dist.wso2.com/downloads/business-activity-monitor/tool-boxes/2.5.0/WebApps_Stats_Monitoring.tbox.

   This is the URL of the WebApps_Stats_Monitoring toolbox (default toolbox for web application data monitoring). Note that all the toolboxes are hosted in the product web site for WSO2 BAM.

   If you have changed any details corresponding to the toolbox when you configured the AS (see the step on configuring AS), you must change the hive script of the toolbox according to those changes.

4. Click Install. Refresh after a few seconds to see if the toolbox is installed.

**Step 6: Invoke the web application that has enabled statistics monitoring**

To invoke the web application in AS:

1. Login and open the AS management console.

2. On the Main menu, under the Applications caption, click List.

3. Click the respective web application context link. The application dashboard will appear.

4. Click the web application services link (e.g., /jaxrs_basic/services).

**Step 7: Schedule or execute the webapp statistics**

To execute the statistics on BAM:

1. Login and open the BAM management console.

2. On the Main menu, under Analytics, click List. The webapp statistics will be displayed.

3. Execute the webapp statistics or schedule its running sequence.

**Viewing the web application statistics**

To view the statistics with the web statistics on BAM:

1. On the Main menu of the BAM management console, click Dashboard.

2. Login to the dashboard using "admin" as the username and password.

3. Select the Server, Tenant and Web App to view web application statistics via the BAM Dashboard.
Services Development and Deployment

Services of various types (Axis2 services, JAR services and Spring services) can be deployed in WSO2 AS easily by using the management console. See the following topics for more information:

- Axis2 services
- Spring services
- Engaging global modules for services
- Deploying services in AS

Axis2 services

An Axis2 archive file is just a .jar file with a specific directory structure as shown below.

```
Test.aar
  META-INF
    services.xml
  Test.class
```

- To learn how to create an Axis2 service using WSO2 Developer Studio, refer to: http://markmail.org/download.d.qxy?id=tsctq7ex5rnwexls&number=1
- To learn how to create an Axis2 service using Eclipse, refer to: http://wso2.org/library/tutorials/create-axis2-web-service-in-3-steps-using-eclipse

Spring services

To develop a simple Spring service, you have to provide a Spring context.xml file, which has a .jar file containing the bean classes. Given below is an example of a simple Spring context.xml file specifying a particular bean with
an ID and a class name found in the .jar file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="SimpleEchoBean"
       class="org.wso2.test.EchoBean"/>
</beans>
</beans>
```

The Echo Bean for this example is shown below. It can contain any type of service implementation.

```java
package org.wso2.test;
/*** Example EchoBean***/

public class EchoBean {
    public String echoString(String input) { return input; }

    public int echoInt(int input) { return input; }
}
```

For more information, refer to http://axis.apache.org/axis2/java/core/docs/spring.html

Engaging global modules for services

With Axis2, you can use modules to extend the functions of services deployed in your server. See the following topics for detailed information on how you can create axis2 modules, add them to the server and engage them for your services.

- Creating an Axis2 Module
- Deploying Axis2 Modules

Deploying services in AS

See the following topics for step-by-step instructions on how to deploy services in WSO2 AS:

- Deploying Axis2 Services
- Deploying JAR Services
- Deploying Spring Services

Deploying Axis2 Services

All WSO2 products come with Axis2 capabilities, which allows you to deploy services as archive files. An Axis2 Service is deployed in your product in the form of an Axis2 archive file (.aar), which has all the service-related resources, service classes, and third-party libraries bundled together.
Once you have developed the Axis2 service, you can upload the .aar file to a running WSO2 AS instance as shown below.

1. Log in to the management console and select **Main -> Services -> Add -> AAR Service.**

2. The **Add Service Archives** page appears. Select your .aar file and click **Upload**. For example, specify a service hierarchy.

   **Service Hierarchy**
   Specifying a service hierarchy is optional. It allows you to customize your service endpoint reference. If the business logic of the same service changes over time, you can maintain multiple versions of the service by using service hierarchy at deployment.

   For example, consider a service named **Version**. If the service hierarchy is not mentioned, then the service endpoint reference is ../services/Version. With service hierarchy specified, for example as 1.0.0, the endpoint reference is ../services/1.0.0/Version. Specifying a hierarchy is important if you want to manage services separately.

3. If the file is uploaded successfully, a message appears prompting you to refresh the page. Click **OK**.

4. Refresh the **Deployed Services** page in the management console to view the newly added service.
Deploying JAR Services

Follow the instructions below to deploy a Jar service in a running WSO2 AS instance.

1. Log on to the product's management console and select Main -> Services -> Add -> Jar Service.

2. The Add Jar Service page opens. The Artifact and Service Group Name fields are mandatory. Others are optional.

   **Add Jar Service**

   ![Add Jar Service Form](image)

   **Service Hierarchy** Specifying a service hierarchy is optional. It allows you to customize your service endpoint reference. If the business logic of the same service changes over time, you can maintain multiple versions of the service by using service hierarchy at deployment.

The file name of the archive is always used as the service name unless you have a different name attributed to the service file. For example, if the name of the archive file is Test.aar, then the name of the service will be Test.
For example, consider a service named "Version". If the service hierarchy is not mentioned, then the service endpoint reference is ../services/Version. With service hierarchy specified, for example as 1.0.0, the endpoint reference is ../services/1.0.0/Version. Specifying a hierarchy is important if you want to manage services separately.

3. Specify the service archive details and click Next.

Add Jar Service

You can add more files by clicking Add more dependencies. The "-" sign removes the option to add more files.

4. If the above steps are successful, you will be shown all the classes in the .jar file/files that you uploaded.

You can specify the following properties in the selected classes:

- **Service Name**: Name of the Web service created from the selected class.
- **Deployment Scope**: There are four ways in which you can deploy a service. The scope of the service relates to the life cycle of the service instance. They are defined as follows:
  - Application - Service instance lifetime is equal to the server lifetime. Service class will be initialized only once for a running server. This is useful in developing stateful Web
services.
- Request - Service class is initialized for each and every request.
- SOAP Session - Axis2 supports a SOAP session using a SOAP specific cookie. Here, the service instance lifetime is equal to the lifetime of the SOAP session.
- Transport Session - Service instance lifetime is equal to the lifetime of the transport session.
  - **Use Original WSDL**: Determines whether or not to use the original WSDL in the service archive. This property can only be specified if you have already uploaded a WSDL in the above steps. If this is set to 'False', then the WSDL will be generated since it is not specified earlier.

5. Select one or more classes to be exposed as Web services and click **Next**.

![Select Classes to Expose as Web Services](image)

6. You will be shown the set of classes selected along with the operations in each of those classes. All methods are selected by default. Deselect the ones you do not wish to expose as Web service operations and click **Finish**.

![Select Methods to Be Exposed as Web Service Operations](image)

7. If the file is uploaded successfully, a message appears prompting you to refresh the page. Click **OK**.

![Service archive successfully created](image)

8. Refresh the **Deployed Services** page in the product's management console to view the newly added service listed.

![Deployed Services](image)

**Deploying Spring Services**
To develop a simple Spring service, you have to provide a Spring context.xml file, which has a .jar file containing the bean classes. Given below is an example of a simple Spring context.xml file specifying a particular bean with an ID and a class name found in the .jar file.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="SimpleEchoBean" class="org.wso2.test.EchoBean">
  </bean>
</beans>
```

The Echo Bean for this example is shown below. It can contain any type of service implementation.

```java
package org.wso2.test;
/** Example EchoBean**/

public class EchoBean {
    public String echoString(String input) {
        return input;
    }

    public int echoInt(int input) {
        return input;
    }
}
```

For more information, refer to [http://axis.apache.org/axis2/java/core/docs/spring.html](http://axis.apache.org/axis2/java/core/docs/spring.html)

**Deploying Spring Services**

Follow the instructions below to deploy a Spring service to a running WSO2 AS instance.

1. Log on to the product's management console and select **Main -> Service -> Add -> Spring Service**.

2. In the **Add Spring Service** window that appears, specify the relevant files and click **Upload**. For example,
**Service Hierarchy**

Specifying a service hierarchy is optional. It allows you to customize your service endpoint reference. If the business logic of the same service changes over time, you can maintain multiple versions of the service by using service hierarchy at deployment.

For example, consider a service named "SpringTest". If the service hierarchy is not mentioned, then the service endpoint reference is ../services/SpringTest. With service hierarchy specified, for example as 1.0.0, the endpoint reference is ../services/1.0.0/SpringTest. Specifying a hierarchy is important if you want to manage services separately.

If the files are not faulty, the **Select Beans to Deploy** page appears.

If the beans specified in the spring context.xml file are mentioned in the .jar file, they will be displayed on this page.

3. Select the **Include** check box associated with the the beans you want to expose as Web services and click **Generate**.
4. If the generation is successful, a message appears prompting you to refresh the page. Click OK.

5. Refresh the Deployed Services page in your product's management console to view the newly added services listed.

If your service implementation depends on any third-party libraries, you can copy them to the `<AS_HOME>/repository/components/extensions` directory and restart the server to get your service started.
Samples

This section discusses the following use cases of the product.

- Deploying your First Custom Web Application
- The Entitlement Servlet Filter Sample
- JAX-RS Samples
- JAX-WS Samples
- Jaggery Samples
- WebSockets Sample
- SSO Sample

Deploying your First Custom Web Application

The WSO2 Application Server supports Apache Tomcat based Web application deployment. All you need to do is to bundle your Web application as a .war archive with all your Web application related resources and all third-party libraries, and then deploy the archive. This sample demonstrates how you can get started quickly with WSO2 Application Server, by deploying your custom web application.

You can use Maven or Apache Ant to deploy the applications as follows:

- Using Maven to deploy applications
- Using Apache Ant to deploy applications

Using Maven to deploy applications

Apache Maven 3 is recommended to build the samples.

Follow the steps given below before you build and run the sample.

1. Open a command prompt and navigate to the base directory of the sample (i.e., `<PRODUCT_HOME>/samples`).
2. Build the demo and create a WAR file using the following command.

   ```
   mvn clean install
   ```

   This will create the sample webapp WAR file (`wso2appserver-samples-hello-webapp-1.0.war`) in the target folder of the `<AS_HOME>/samples/<sample-name>` directory.

3. Start the WSO2 Application Server by following the instructions in Running the Product.
4. You can deploy the web app in two ways:
   - Manually copy the WAR file to the `<AS_HOME>/repository/deployment/server/webapps` directory.
   - Deploy the WAR file you created in step 2 using the management console as detailed here.

Using Apache Ant to deploy applications
You can find the sample discussed here in the `<AS_HOME>/samples/HelloWorldWebapp` directory.

The steps are as follows:

1. Run the WSO2 Application Server. For instructions see, [Running the Product](#).
2. In a command prompt, switch to the samples directory. For example, in Windows:

   ```
   cd <AS_HOME>/samples/HelloWorldWebapp
   ```

3. Enter the following command:

   ```
   ant
   ```

   This will copy the sample webapp (wso2appserver-samples-hello-webapp-1.0.war file) to `<AS_HOME>/repository/deployment/server/webapps/` directory. If you log into the Application Server, you will see the deployed webapp on the Running Applications screen.

### Related Links

- JAX-RS Samples
- JAX-WS Samples
- Jaggery Samples

### The Entitlement Servlet Filter Sample

This sample demonstrates how fine grained authorization is enabled and used for web applications. Read more about XACML fine grained authorization in web applications.

In this sample we create a Web App with Entitlement Servlet Filter engaged. All the dependencies are packed in to the lib. So this sample can be run in any other webapp container. You have to simply host the Web App in the container and edit the `pom.xml` to give the URL of the web application. Thereafter, you can check the functionality.

- Prerequisites
- Building the Sample
- Running the sample

**Prerequisites**

1. Before you start an instance of WSO2 IS, be sure that the port offset is given as "1" in the `carbon.xml` file.

The configurations in the `web.xml` file of the sample web app should match the configurations in your running IS instance. For example, if you have changed the `remoteServiceUrl` value in the `web.xml` of the sample (stored in sample directory's `<AS_HOME>/samples/EntitlementFilter/src/main/webapp/WEB-INF` folder), you must change the IS portoffset accordingly.
2. Start an instance of WSO2 Identity Server.
3. Import the sample XACML policy stored in the `<AS_HOME>/samples/EntitlementFilter/src/main/resources` directory to IS using the management console.

**Building the Sample**

The base directory of this sample has the build.xml file which is used to build the necessary Web App and to deploy it in WSO2 App Server.

Run the relevant command to deploy the web app:

- **Using Maven**
  1. Create a WAR file for the sample using the following command:

```
mvn clean install
```

  2. Deploy the generated WAR file on WSO2 AS with the related logs on the console:

```
mvn -Pdeploy
```

- **Using Ant**

```
ant
```

Start the Application Server and access the Management Console. Go to the webapp service listing page. You will see the deployed service.

**Running the sample**

1. You have to run the run-client.sh or run-client.bat script. It has all the arguments and classpaths configured to run the sample.
2. In the console it will show the result for several Entitlement Decision Scenarios. Following output will be shown in the console if you run the sample correctly:

```
***********Starting the Entitlement Servlet Filter Sample***********

Sending Request For a Web Page Which Requires Authorization
Subject : admin
Resource : /Entitlement_Sample_WebApp/protected.jsp
Action : GET
Environment : Not Specified
***Response BEGIN ***
<html>
<head><title>Protected Page</title></head>
<body>Only Authorized Users Can View This</body>
</html>
***Response END ***

Sending Request For a Web Page Which Not Requires Authorization
```
Subject : admin
Resource : /Entitlement_Sample_WebApp/index.jsp
Action : GET
Environment : Not Specified
***Response BEGIN ***
<html>
<head><title>Index Page</title></head>
<body>Anybody Can Access This Page....</body>
</html>

***Response END ***

Sending Request For a Web Page Which Requires Authorization with False Name
Subject : andunslg
Resource : /Entitlement_Sample_WebApp/protected.jsp
Action : GET
Environment : Not Specified
***Response BEGIN ***
Server returned HTTP response code: 401 for URL:
http://localhost:9763/Entitlement_Sample_WebApp/protected.jsp
***Response END ***

Sending Request For a Web Page Which Requires Authorization with False Action
Subject : admin
Resource : /Entitlement_Sample_WebApp/protected.jsp
Action : POST
Environment : Not Specified
***Response BEGIN ***
<html>
<head><title>Protected Page</title></head>
<body>Only Authorized Users Can View This</body>
</html>

***Response END ***

Sending Request For a Web Page Which Requires Authorization But Policy is not defined
Subject : admin
Resource : /Entitlement_Sample_WebApp/other.jsp
Action : GET
Environment : Not Specified
***Response BEGIN ***
<html>
<head><title>Index Page</title></head>
<body>Anybody Can Access This Page....</body>
</html>
JAX-RS Samples

- Building and Running JAX-RS Samples
- JAX-RS Basics
- JAX-RS Advanced
- JAX-RS Sample on Content Negotiation

Building and Running JAX-RS Samples

Using Maven

Apache Maven 3 is recommended to build the samples.

Follow the steps given below before you build and run the sample:

1. Open the sample folder in your product pack using a command line and go to the `<PRODUCT_HOME>/samples/Jaxws-Jaxrs/<JAX-RS-Sample-Folder>` directory. The folders for Jax-rs samples are listed here: JAX-RS Sample Folder.
2. Build the demo and create a WAR file using the following command.

   ```
   mvn clean install
   ```

3. Deploy the generated WAR file in WSO2 AS with the related logs on the console using the following command:

   ```
   mvn -Pdeploy
   ```

   This command will copy the WAR file to the `<AS_HOME>/repository/deployment/server/webapps` directory.

4. Start the WSO2 Application Server by executing one of the following commands, where `<PRODUCT_HOME>` is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td><code>&lt;PRODUCT_HOME&gt;/bin/wso2server.bat --run</code></td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td>sh <code>&lt;PRODUCT_HOME&gt;/bin/wso2server.sh</code></td>
</tr>
</tbody>
</table>

5. Log in to the management console and click List on the Main menu, under Manage/Applications. This
opens the Running Applications page. The deployed applications will be listed here.

6. You can open the required web application by clicking the Find Services action.

If you want to remove the target directory, run the following command inside the `<PRODUCT_HOME>/samples/Jaxrs-Jaxrs/<JAX-RS-Sample-Folder>` directory:

```
mvn clean
```

Using Apache Ant

You need Apache Ant 1.6.2 or higher to build the samples.

1. Open the sample folder in your product pack using a command line and go to the `<PRODUCT_HOME>/samples/Jaxws-Jaxrs/<JAX-RS-Sample-Folder>` directory. The folders for Jax-rs samples are listed here: JAX-RS Sample Folder.

2. To deploy the sample, run the following command:

```
ant
```

3. Start the Application Server and log in to the management console.

4. You can run the sample in two ways:
   1. Log in to the management console. On the Main menu, under Manage/Applications, click List to open the Running Applications page. The deployed applications will be listed here.
   2. Using the command line, go the JAX-RS sample directory and execute the following command:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td>run-client.bat</td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td>sh run-client.sh</td>
</tr>
</tbody>
</table>

The results of the sample will show on the command line.

For different options, run:

```
sh run-client.sh -help
```

JAX-RS Sample Folder

<table>
<thead>
<tr>
<th>Sample</th>
<th>JAX-RS Sample Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAX-RS Basics</td>
<td>jaxrs_basic</td>
</tr>
<tr>
<td>JAX-RS Advanced</td>
<td>jaxrs_starbucks_service</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>JAX-RS Sample on Content Negotiation</td>
<td>jaxrs_content_negotiation</td>
</tr>
</tbody>
</table>

**JAX-RS Basics**

This sample demonstrates a basic REST-based Web Services using JAX-RS (JSR-311).

- Building the sample
- Running the sample

**Building the sample**

1. Open a command line, and navigate to the `<AS_HOME>/samples/Jaxws-Jaxrs/jaxrs_basic` directory.
2. Run the relevant command to deploy the web app:
   - Using Maven
     1. Create a WAR file for the sample using the following command:
        ```sh
mvn clean install
        ```
     2. Deploy the generated WAR file on WSO2 AS with the related logs on the console:
        ```sh
        mvn -Pdeploy
        ```
   - Using Ant
     ```sh
     ant
     ```
3. Start the WSO2 Application Server by executing one of the following commands, where `<PRODUCT_HOME>` is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td><code>&lt;PRODUCT_HOME&gt;\bin\wso2server.bat --run</code></td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td><code>sh &lt;PRODUCT_HOME&gt;/bin/wso2server.sh</code></td>
</tr>
</tbody>
</table>

4. Log in to the management console and click **List** on the **Main** menu, under **Manage/Applications**. This opens the **Running Applications** page. The deployed applications will be listed here.

5. You can open the required web application by clicking the **Find Services** action.

**Running the sample**

You can invoke the JAX-RS web application as shown below.
• Get the customer instance for customer ID 123 by sending an HTTP GET request to http://localhost:9763/jaxrs_basic/services/customers/customerservice/customers/123. The XML document returns the following:

```xml
<Customer>
   <id>123</id>
   <name>John</name>
</Customer>
```

• Get the product 323 that belongs to order 223 by sending an HTTP GET request to http://localhost:9763/jaxrs_basic/services/customers/customerservice/orders/223/products/323. The XML document returns the following:

```xml
<Product>
   <description>product 323</description>
   <id>323</id>
</Product>
```

• Add the customer named Jack by sending an HTTP POST request to http://localhost:9763/jaxrs_basic/services/customers/customerservice/customers as shown below.

```xml
<Customer>
   <name>Jack</name>
</Customer>
```

• Update the customer with ID 123 as shown below by sending an HTTP PUT request to http://localhost:9763/jaxrs_basic/services/customers/customerservice/customers.

```xml
<Customer>
   <id>123</id>
   <name>John</name>
</Customer>
```

The client code demonstrates how to send HTTP GET/POST/PUT/DELETE requests; whereas, the server code demonstrates how to build a RESTful endpoint through JAX-RS (JSR-311) APIs.

**JAX-RS Advanced**

This sample shows some advanced functionalities of REST-based Web Services using JAX-RS (JSR-311). Please note that this sample was called jaxrs_sample_02 in the AS 5.1.0 release. It has been renamed to provide better clarity.

• Building the sample
• Running the sample

**Building the sample**

1. Open a command line, and navigate to the \<AS_HOME>\samples\Jaxws-Jaxrs\jaxrs_starbucks_se
1. Create a WAR file for the sample using the following command:

   ```mvn clean install```

2. Deploy the generated WAR file on WSO2 AS with the related logs on the console:

   ```mvn -Pdeploy```

3. Start the WSO2 Application Server by executing one of the following commands, where `<PRODUCT_HOME>` is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td><code>&lt;PRODUCT_HOME&gt;/bin/wso2server.bat --run</code></td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td><code>sh &lt;PRODUCT_HOME&gt;/bin/wso2server.sh</code></td>
</tr>
</tbody>
</table>

4. Log in to the management console and click **List** on the **Main** menu, under **Manage/Applications**. This opens the **Running Applications** page. The deployed applications will be listed here.

5. You can open the required web application by clicking the **Find Services** action.

**Running the sample**

This JAX-RS webapp provides following services:

- A RESTful Starbucks outlet service is provided on URL `http://localhost:9763/jaxrs_starbucks_service/service/Starbucks_Outlet_Service` (say serviceURL). Users access this URI to perform operations of drink orders.

- A HTTP GET request to URL `${serviceURL}/orders/123` returns a drink order added with the generated id 123. The JSON document returns:
• A HTTP POST request to URL `${serviceURL}/orders` with the following data adds an order:

```xml
<Order>
  <drinkName>Mocha Flavored Coffee</drinkName>
  <additions>Caramel</additions>
</Order>
```

An id for this order is generated at server-side and returned in the response. The response body looks like the following in JSON format:

```
{
  "Order":{
    "additions":"Caramel",
    "drinkName":"Mocha Flavored Coffee",
    "locked":false,
    "orderId":"ee1a9ec2-c8a5-4afe-8585-74df591f9990"
  }
}
```

• A HTTP PUT request to URL `${serviceURL}/orders` with the following data, where the 'orderId' is set to the one received in the response body when the order was added:

```json
{
  "Order":{
    "orderId":"ee1a9ec2-c8a5-4afe-8585-74df591f9990",
    "additions":"Chocolate Chip Cookies"
  }
}
```

• It updates the 'additions' of the order "ee1a9ec2-c8a5-4afe-8585-74df591f9990" from "Caramel" to "Chocolate Chip Cookies". The response will be in XML-format as follows.
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Order>
  <additions>Chocolate Chip Cookies</additions>
  <drinkName>Full Leaf Green Tea</drinkName>
  <locked>true</locked>
  <orderId>ee1a9ec2-c8a5-4afe-8585-74df591f9990</orderId>
</Order>

A HTTP DELETE request to URL `${serviceURL}/orders/ee1a9ec2-c8a5-4afe-8585-74df591f9990` with an empty request body will delete the order details of order "ee1a9ec2-c8a5-4afe-8585-74df591f9990".

The client code demonstrates how to send HTTP GET/POST/PUT/DELETE requests whereas the server code demonstrates how to build a RESTful endpoint through JAX-RS (JSR-311) APIs.

**JAX-RS Sample on Content Negotiation**

This sample demonstrates how to implement content negotiation in the Application Server. Content negotiation is a mechanism which allows to maintain multiple representations of the same resource, and serve the particular representation relevant to a given request and what the requesting server needs.

- **Building the sample**
- **Running the sample**

**Building the sample**

1. Open a command line, and navigate to the `<AS_HOME>/samples/Jaxws-Jaxrs/jaxrs_content_negotiation` directory.
2. Run the relevant command to deploy the web app:
   - Using Maven
     1. Create a WAR file for the sample using the following command:
        
        ```
        mvn clean install
        ```
     2. Deploy the generated WAR file on WSO2 AS with the related logs on the console:
        
        ```
        mvn -Pdeploy
        ```
   - Using Ant
     
     ```
     ant
     ```
3. Start the WSO2 Application Server by executing one of the following commands, where `<PRODUCT_HOME>` is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
On Windows

\<PRODUCT_HOME>\bin\wso2server.bat --run

On Linux/Solaris

sh <PRODUCT_HOME>/bin/wso2server.sh

4. Log in to the management console and click List on the Main menu, under Manage/Applications. This opens the Running Applications page. The deployed applications will be listed here.

5. You can open the required web application by clicking the Find Services action.

Running the sample

A RESTful customer service is provided on URL http://localhost:9763/jaxrs_content_negotiation/services/. Users access this URL to operate on a customer. The REST server provides the following services:

- An HTTP GET request to URL http://localhost:9763/jaxrs_content_negotiation/services/customers/customerservice/customers/123 with "accept header" set to "application/xml" returns a customer instance in XML format. The XML document returns:

  <Customer>
    <id>123</id>
    <name>John</name>
  </Customer>

- An HTTP GET request to URL http://localhost:9763/jaxrs_content_negotiation/services/customers/customerservice/customers/123 with "accept header" set to "application/json" returns a customer instance in JSON format. The JSON document returns:

  {"Customer":{"id":"123","name":"John"}}

- An HTTP GET request to URL http://localhost:9763/jaxrs_content_negotiation/services/customers/customerservice/customers/123 without setting "accept header" explicitly, returns a customer instance in XML format. This is because the accept header will be absent from the request when using HTTP Client. In that case, the CXF will treat the "Accept" content type as "*/". The XML document returns:

  <Customer>
    <id>123</id>
    <name>John</name>
  </Customer>

JAX-WS Samples

- Building and Running JAX-WS Samples
- Asynchronous Invocation
- Build and Invoke a Service Using a WSDL
- Develop Services in Code-First Approach
- JAX-WS Handlers
- MTOM Sample for SWA and XOP
- WS-Security Sample on Signature and Encryption
- WS-Security Sample on UsernameToken and Timestamp
Building and Running JAX-WS Samples

Using Maven

Follow the steps given below before you build and run the sample:

1. Open the sample folder in your product pack using a command line and go to the <PRODUCT_HOME>/samples/Jaxws-Jaxrs/<JAX-WS-Sample-Folder> directory. The folders for Jax-ws samples are listed here: JAX-WS Sample Folder.

2. Build the demo and create a WAR file using the following command.

   ```
   mvn clean install
   ```

3. Deploy the generated WAR file in WSO2 AS with the related logs on the console using the following command:

   ```
   mvn -Pdeploy
   ```

   This command will copy the WAR file to the <AS_HOME>/repository/deployment/server/webapps directory.

4. Start the WSO2 Application Server by executing one of the following commands, where <PRODUCT_HOME> is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td>&lt;PRODUCT_HOME&gt;/bin/wso2server.bat --run</td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td>sh &lt;PRODUCT_HOME&gt;/bin/wso2server.sh</td>
</tr>
</tbody>
</table>

5. Log in to the management console and click List on the Main menu, under Manage/Applications. This opens the Running Applications page. The deployed applications will be listed here.

6. You can open the required web application by clicking the Find Services action.

   If you want to remove the target directory, run the following command inside the <PRODUCT_HOME>/samples/Jaxws-Jaxrs/<JAX-WS-Sample-Folder> directory:

   ```
   run mvn clean
   ```

Using Apache Ant

You need Apache Ant 1.6.2 or higher to build the samples.
1. Open the sample folder in your product pack using a command line and go to the `<PRODUCT_HOME>/samples/Jaxws-Jaxrs/JAX-WS-Sample-Folder` directory. The folders for Jax-rs samples are listed here: JAX-WS Sample Folder.

2. To deploy the sample, run the following command:

   ```
   ant
   ```

3. Start the Application Server and log in to the management console.

4. You can run the sample in two ways:
   1. Log in to the management console. On the **Main** menu, under **Manage/Applications**, click **List** to open the **Running Applications** page. The deployed applications will be listed here.
   2. Using the command line, go to the JAX-WS sample directory and execute the following command:

```
<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td>run-client.bat</td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td>sh run-client.sh</td>
</tr>
</tbody>
</table>
```

The results of the sample will show on the command line.

For different options, run:

```
sh run-client.sh -help
```

### JAX-WS Sample Folder

<table>
<thead>
<tr>
<th>Sample</th>
<th>JAX-WS Sample Folder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous Invocation</td>
<td>async_jaxws</td>
</tr>
<tr>
<td>Build and Invoke a Service Using a WSDL</td>
<td>wsd1_first_jaxws</td>
</tr>
<tr>
<td>Develop Services in Code-First Approach</td>
<td>java_first_jaxws</td>
</tr>
<tr>
<td>JAX-WS Handlers</td>
<td>handlers_jaxws</td>
</tr>
<tr>
<td>MTOM Sample for SWA and XOP</td>
<td>mtom_jaxws</td>
</tr>
<tr>
<td>WS-Security Sample on Signature and Encryption</td>
<td>sign_encrypt_jaxws</td>
</tr>
<tr>
<td>WS-Security Sample on UsernameToken and Timestamp</td>
<td>ws_security\ut</td>
</tr>
</tbody>
</table>
Asynchronous Invocation

This sample illustrates the use of JAX-WS asynchronous invocation model using Document/Literal Style.

For more information on asynchronous invocation, refer to the JAX-WS 2.0 specification here: http://jcp.org/aboutJava/communityprocess/pfd/jsr224/index.html

The asynchronous model allows client threads to continue after making a two-way invocation, without being blocked while awaiting a response from the server. Once the response is available, it is delivered to the client application asynchronously, using one of two alternative approaches as follows:

- **Callback**: the client application implements the "javax.xml.ws.AsyncHandler" interface to accept notification of the response availability.
- **Polling**: the client application periodically polls a "javax.xml.ws.Response" instance to check if the response is available.

This Sample illustrates both approaches. Additional methods are generated on the Service Endpoint Interface (SEI) to provide this asynchrony, named by convention with the suffix "Async". As many applications will not require this functionality, the asynchronous variants of the SEI methods are omitted by default to avoid loading the SEI with unnecessary baggage. In order to enable generation of these methods, a bindings file (wsdl/asyncBindings.xml) is passed to the wsdl2java generator.

**Building and running the sample**

This sample can be found at `<AS_HOME>/samples/Jaxws-Jaxrs/async_jaxws`

For information on building and running the sample, see Building and Running JAX-WS Samples.

**Build and Invoke a Service Using a WSDL**

This sample demonstrates how to build and call a Web service using a given WSDL file. This approach is also called contract-first.

For more information on contact-first Web services, refer to: http://cxf.apache.org/docs/defining-contract-first-webservices-with-wsdl-generation-from-java.html

This Sample mainly addresses SOAP over HTTP in Document/Literal or Document/Literal wrapped style. For other transports or styles the configuration may look different.

This sample consists of three sections as follows:

- **Creating the server and client code stubs from the WSDL**.
- **Service implementation**.
- **Client implementation**.

**Code generation**

Code generation in Maven is done using the cxf-codegen-plugin. For information, refer to http://cxf.apache.org/docs/maven-cxf-codegen-plugin-wsdl-to-java.html. Code generation is activated using a binding.xml file. In this case, the normal java Date is used for xsd:date and xsd:DateTime. If this is not present, the XMLGregorianCalendar will be used.
Another common use of the binding file is to generate asynchronous stubs. The line `jaxws:enableAsyncMapping` has to be uncommented to use this. More information about the binding file can be found here: http://jax-ws.java.net/jax-ws-20-fcs/docs/customizations.html.

**Service implementation**

The service is implemented in the class `CustomerServiceImpl`, which simply implements the previously-generated service interface. The method `getCustomersByName` demonstrates how a query function looks like. The idea is to search and return all customers with the given name. If the searched name is null, the method returns an exception to indicate that no matching customer was found. (In a real implementation, a list of zero objects can typically be used). For any other name, the method returns a list of two customer objects. The number of objects can be increased to test how fast CXF works for larger data.

**Client implementation**

The main client code lives in the class `CustomerServiceTester`, which needs a proxy to the service and demonstrates some calls and their expected outcome using junit assertions.

The first call is a request `getCustomersByName` for all customers with name “Smith” and the result is checked. Then, the same method is called with the invalid name “None”. In this case a `NoSuchCustomerException` is expected. The third call shows that the one way method `updateCustomer` will return instantly even if the service needs some time to process the request.

The classes `CustomerServiceClient` and `CustomerServiceSpringClient` show how to get a service proxy using JAX-WS and how to wire it to your business class (in this case, `CustomerServiceTester`).

**Building and running the sample**

This sample can be found at `<AS_HOME>\samples\Jaxws-Jaxrs\wsdl_first_jaxws`.

For information on building and running the sample, see Building and Running JAX-WS Samples.

1. If you have a EPR location other than http://localhost:9763/java_first_jaxws/services/hello_world, edit the run-client script.
2. Prior to running the client, it is recommended to confirm that the generated WSDL (http://{ip}:{port}/java_first_jaxws/services/hello_world?wsdl) can be seen from a web browser.

**Develop Services in Code-First Approach**

This sample demonstrates how to develop a service using the code-first approach using JAX-WS APIs and jsr-181.

**Building and running the sample**

This sample can be found at `<AS_HOME>\samples\Jaxws-Jaxrs\java_first_jaxws`.

For information on building and running the sample, see Building and Running JAX-WS Samples.

**JAX-WS Handlers**

This sample demonstrates how JAX-WS handlers are used in the WSO2 Application Server. The server uses a SOAP protocol handler, which logs incoming and outgoing messages to the console. The server code registers a handler using the `@HandlerChain` annotation within the service implementation class. In this sample, `LoggingHandler` is the `SOAPHandler` that logs the entire SOAP message content to stdout.
The client includes a logical handler that checks the parameters on outbound requests and short-circuits the invocation in certain circumstances. This handler is specified programmatically.

**Building and running the sample**

This sample can be found at `<AS_HOME>\samples\Jaxws-Jaxrs\handlers_jaxws`

For information on building and running the sample, see Building and Running JAX-WS Samples.

When specifying the handler file path in the `@HandlerChain` annotation, the complete path from root should be given as:

```
@HandlerChain(file = "/demo/handlers/common/demo_handlers.xml")
```

Relative paths such as "/common/demo_handlers.xml" do not render in WSO2 Carbon. The full package hierarchy should be used.

**MTOM Sample for SWA and XOP**

This sample demonstrates the use of a SOAP message with an attachment and XML-binary optimized packaging.

**Building and running the sample**

This sample can be found at `<AS_HOME>\samples\Jaxws-Jaxrs\mtom_jaxws`

For information on building and running the sample, see Building and Running JAX-WS Samples.

**WS-Security Sample on Signature and Encryption**

This sample shows how WS-Security support can be enabled in JAX-WS.

WS-Security can be configured to the Client and Server endpoints by adding WSS4JInterceptors. Both Server and Client can be configured for outgoing and incoming interceptors. Various actions like, Timestamp, UsernameToken, Signature, Encryption, etc., can be applied to the interceptors by passing appropriate configuration properties.

The logging feature is used to log the inbound and outbound SOAP messages and display these to the console. In all other respects, this sample is based on the basic hello_world sample.

The samples in this directory use STRONG encryption. The default encryption algorithms included in a JRE are not adequate for these samples.

For the sample, the Java Cryptography Extension (JCE), Unlimited Strength Jurisdiction Policy Files available on Oracle’s JDK download page must be installed from here: [http://www.oracle.com/technetwork/java/javase/downloads/index.html](http://www.oracle.com/technetwork/java/javase/downloads/index.html); otherwise, there will be errors on invalid key lengths.

**Building and running the sample**

This sample can be found at `<AS_HOME>\samples\Jaxws-Jaxrs\ws_security\sign_encrypt_jaxws`

For information on building and running the sample, see Building and Running JAX-WS Samples.

**WS-Security Sample on UsernameToken and Timestamp**
This sample demonstrates how WS-Security support in JAX-WS services is enabled.

WS-Security can be configured to the client and server endpoints by adding WSS4J Interceptors. Both server and client can be configured for outgoing and incoming interceptors. Various actions like, Timestamp, UsernameToken, Signature, Encryption, etc., can be applied to the interceptors by passing appropriate configuration properties.

The logging feature is used to log the inbound and outbound SOAP messages and display these to the console. In all other respects, this sample is based on the basic hello_world sample.

The samples in this directory use STRONG encryption. The default encryption algorithms included in a JRE are not adequate for these samples.

For the sample, the Java Cryptography Extension (JCE), Unlimited Strength Jurisdiction Policy Files available on Oracle's JDK download page must be installed from here: (http://www.oracle.com/technetwork/java/javase/downloads/index.html); otherwise, there will be errors on invalid key lengths.

Building and running the sample

This sample can be found at <AS_HOME>\samples\Jaxws-Jaxrs\ws_security\ut

For information on building and running the sample, see Building and Running JAX-WS Samples.

Jaggery Samples

The Jaggery samples, demonstrate some functionality of Jaggery deployed as Jaggery applications in the WSO2 Web Service Application Server.

The following sections describe each Jaggery sample and also how to build the samples:

- Building Jaggery Samples
- Available Jaggery Samples

Building Jaggery Samples

You need Apache Ant 1.6.2 or higher to build the samples.

The steps are as follows:

1. Run the WSO2 Application Server by executing one of the following commands, where <PRODUCT_HOME> is the directory where you installed the product distribution:

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td>&lt;PRODUCT_HOME&gt;\bin\wso2server.bat --run</td>
</tr>
<tr>
<td>On Linux/Solaris</td>
<td>sh &lt;PRODUCT_HOME&gt;/bin/wso2server.sh</td>
</tr>
</tbody>
</table>

2. In a command prompt, switch to the samples directory:
   For example, in Windows:
3. Enter the following command:

```ant```

This will copy the directories into `<PRODUCT_HOME>/repository/deployment/server/jaggeryapps` directory. If you log into the Application Server, you will see the deployed jaggery applications on the Running Application screen.

**Available Jaggery Samples**

- **Coffee Shop**
- **TaskMaster**
- **Freshometer**
- **Jaggery Documentation**

**Coffee Shop**

The coffee shop sample will explain RESTful application features with Jaggery. It is a nice client end-user interface where the user can make an order for coffee. There are seven main REST calls that happen in this application with PUT, POST, GET and DELETE. This sample contains a database connection, URL mapping and web request in Jaggery.

**TaskMaster**

The TaskMaster is a task manager application which makes use of the session in Jaggery. The tasks are saved and can be viewed via the main page.

**Freshometer**

The Fresh-o-meter application allows you to search for any movie title, and retrieve its information (Cast, Movie Poster, Release year, Plot and rating) as well as related videos from YouTube. It acts as a central movie information search engine. The user needs to be online to be able to use this application.

**Jaggery Documentation**

The Documentation section of the jaggery site runs on Jaggery. It is the centralized and in-depth area to view descriptions, demos and samples and try out jaggery code. The documentation site also includes downloads and sample apps.

**WebSockets Sample**

WSO2 AS 5.3.0 supports WebSocket 1.1 API as defined by [the JSR-356](#) specification. This is provided by Apache Tomcat, which is embedded in AS. You can run the following sample web application to see how WebSockets work in AS.

Note that you do not need additional configurations on AS server-level to enable Websocket support for web applications.
Running the sample

The "Example" sample is deployed in AS by default. You can run this sample as explained below.

1. Start WSO2 Application Server.

2. Log in to the management console and click **List** under **Applications** to open the **Running Applications** page.

3. The "example" sample will be listed as a web application. Click **Go to URL** to open the web application. Alternatively, you can directly point your browser to the URL of this sample, which is http://localhost:9763/example/. Note that the port number in the URL needs to be updated if you have set a port offset for the server in the `<AS_HOME>/repository/conf/carbon.xml` file.

4. When you open the URL, click **Websocket** examples. The following samples will be listed:

   1. Echo example: This sample echoes whatever the message you send to the server via websocket. There are several websocket endpoints listed here to illustrate three different mechanisms that implement websocket services.
      - programmatic API - Invoking a Websocket endpoint that is registered programmatically. This is the interface-driven approach where the developer implements the `javax.websocket.Endpoint` interface to create the service.
      - annotation API (basic) - Invoking a websocket service that is defined using java annotations. This is the annotation-driven approach where the developer interacts with the websocket lifecycle events via annotated POJOs.
      - annotation API (stream) - This also follows the annotation-driven approach, and the messages sent from the server are streamed.

   1. Chat example: This simulates a standard chat application. In this application, the messages received are broadcast to all the users that are connected to this websocket service.
   2. Multiplayer snake example
   3. Multiplayer drawboard example

**SSO Sample**

This sample will demonstrate how SAML SSO can be used to log in to web applications deployed in WSO2 Application Server. Go to this link for more information about using SSO with web applications.

**Before you begin:**

1. Ensure that **WSO2 Application Server** and **WSO2 Identity Server** are both installed in your system.
2. The following server configurations should be completed for WSO2 AS and WSO2 IS. The instructions are given here.
   - Generic SSO configurations should be included in the `sso-sp-config.properties` file stored in the `<AS_HOME>/repository/conf/security/` directory.
   - Register SSO service providers in WSO2 IS for each web application.
   - Update the IS and AS hostnames in the `<AS>/repository/conf/security/sso-sp-config.properties` file and the `<IS_HOME>/repository/conf/identity.xml` file.
   - In WSO2 IS, update the resident IDP provider Entity Id.
Enable the SSOValve in the `<AS_HOME>/repository/conf/tomcat/catalina-server.xml` file of AS.

**Building the sample**

1. Open a command line, and navigate to the `<AS_HOME>/samples/SSOValveSample/bar-app` directory.
2. Run the relevant command to deploy the web app:
   - **Using Maven**
     1. Create a WAR file for the sample using the following command:
        ```
        mvn clean install
        ```
     2. Deploy the generated WAR file on WSO2 AS with the related logs on the console:
        ```
        mvn -Pdeploy
        ```
   - **Using Ant**
     ```
     ant
     ```
     This will deploy the `bar-app` web application in AS.
3. In the same way, navigate to the `<AS_HOME>/samples/SSOValveSample/foo-app` directory and deploy the `foo-app` web application.
4. Start the Application Server and log in to the management console.
5. You will find the `bar-app` and `foo-app` web applications listed in the Running Applications page.
7. Now you can run the samples as explained below.

**Running the sample**

1. Go to the Running Applications page of AS.
2. Click Go to URL for the bar-app web application. You will be directed to the following page, which allows you to log into the web application through WSO2 IS:
3. Sign in to access the application.
4. Now, go back to the Running Applications page and click Go to URL for the foo-app web application. Note that you will be automatically logged in to the web application without any need to enter credentials.
Configuration Guide

This section explores different options to customize the WSO2 Application Server according to user-specific needs.

- Clustered Deployment
- Working with Databases
- Working with the Registry
- Working with Transports
- Working with Security
- Working with Multiple Tenants
- Working with Logs
- Tuning Performance
- JNDI Resources

Clustered Deployment

You can install multiple instances of WSO2 products in a cluster to ensure that if one instance becomes unavailable or is experiencing high traffic, another instance will seamlessly handle the requests. See the documentation on clustering WSO2 Application Server for comprehensive details.

Working with Databases

The default databases that WSO2 products uses to store registry, user manager and product-specific data are the H2 databases in `<PRODUCT_Home>/repository/database` as follows:

- `WSO2CARBON_DB.h2.db`: used to store registry and user manager data

These embedded H2 databases are suitable for development, testing, and some production environments. For most production environments, however, we recommend you to use an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc.

You can use the scripts provided with WSO2 products to install and configure several other types of relational databases, including MySQL, IBM DB2, Oracle, and more.

- Setting up the Physical Database
- Managing Datasources

Setting up the Physical Database

The topics in this section describe how to use scripts in `<PRODUCT_HOME>/dbscripts/` folder to set up each type of physical database.

- Setting up a MySQL Cluster
- Setting up Derby
- Setting up H2
- Setting up IBM DB2
- Setting up IBM Informix
- Setting up Microsoft SQL
- Setting up MySQL
- Setting up OpenEdge
- Setting up Oracle
- Setting up Oracle RAC
- Setting up PostgreSQL
- Setting up MariaDB
Setting up a MySQL Cluster

For instructions on setting up any WSO2 product with a MySQL cluster, see this article, which is published in the WSO2 library.

Setting up Derby

You can set up either an embedded Derby database or a remote one according to the information given below:

- Setting up Embedded Derby
- Setting up Remote Derby

Setting up Embedded Derby

The following sections describe how to replace the default H2 databases with embedded Derby:

- Setting up the database
- Setting up the drivers
- Setting up datasources configurations
- Creating database tables

Setting up the database

Follow the steps below to set up an embedded Derby database:

1. Download Apache Derby.
2. Install Apache Derby on your computer.

For instructions on installing Apache Derby, see the Apache Derby documentation.

Setting up the drivers

Copy derby.jar, derbyclient.jar, and derbynet.jar from the <DERBY_HOME>/lib/ directory to the <PRODUCT_HOME>/repository/components/extensions/ directory (the classpath of the WSO2 Carbon web application).

Setting up datasources configurations

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Embedded Derby database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file as shown below.
<datasource>
    <name>WSO2_CARBON_DB</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/WSO2CarbonDB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:derby://localhost:1527/db;create=true</url>
            <username>regadmin</username>
            <password>regadmin</password>
            <driverClassName>org.apache.derby.jdbc.EmbeddedDriver</driverClassName>
            <maxActive>80</maxActive>
            <maxWait>60000</maxWait>
            <minIdle>5</minIdle>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
            <defaultAutoCommit>false</defaultAutoCommit>
        </configuration>
    </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
</tbody>
</table>
### validationQuery
The SQL query that will be used to validate connections from this pool before returning them to the caller.

### validationInterval
The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.

### defaultAutoCommit
This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS driver by default.

When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.

---

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see [Tomcat JDBC Connection Pool](#).

---

Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see [Setting up datasource configurations](#).
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
   <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

   ```xml
   <Configuration>
   <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

---

### Creating database tables

You can create database tables by executing the database scripts as follows:

1. Run the `ij` tool located in the `<DERBY_HOME>/bin/` directory as illustrated below:

   ```sql
   ```
2. Create the database and connect to it using the following command inside the `ij` prompt:

```java
connect 'jdbc:derby:repository/database/WSO2CARBON_DB;create=true';
```

Replace the database file path in the above command with the full path to your database.

3. Exit from the `ij` tool by typing the `exit` command.

```java
exit;
```

4. Log in to the `ij` tool with the username and password that you set in `registry.xml` and `user-mgt.xml`:

```java
connect 'jdbc:derby:repository/database/WSO2CARBON_DB' user 'regadmin' password 'regadmin';
```

5. Use the scripts given in the following locations to create the database tables:

   - To create tables for the registry and user manager database (WSO2CARBON_DB), run the below command:

   ```bash
   run '<PRODUCT_HOME>/dbscripts/derby.sql';
   ```

Now the product is running using the embedded Apache Derby database.

6. Restart the server.

You can create database tables automatically when starting the product for the first time by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

The product is configured to run using an embedded Apache Derby database.

In contrast to setting up with remote Derby, when setting up with the embedded mode, set the database driver name (the `driverClassName` element) to the value `org.apache.derby.jdbc.EmbeddedDriver` and the database URL (the `url` element) to the database directory location relative to the installation. In the above sample configuration, it is inside the `<DERBY_HOME>/WSO2_CARBON_DB/` directory.

Setting up Remote Derby

The following sections describe how to replace the default H2 databases with a remote Derby database:

- Setting up the database
Setting up the database

Follow the steps below to set up a remote Derby database.

1. Download Apache Derby.
2. Install Apache Derby on your computer.

   For instructions on installing Apache Derby, see the Apache Derby documentation.

3. Go to the <DERBY_HOME>/bin/ directory and run the Derby network server start script. Usually it is named startNetworkServer.

Setting up the drivers

Copy derby.jar, derbyclient.jar, and derbynet.jar from the <DERBY_HOME>/lib/ directory to the <PRODUCT_HOME>/repository/components/extensions/ directory (the classpath of the Carbon web application).

Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Remote Derby database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file as shown below.
<datasource>
    <name>WSO2_CARBON_DB</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/WSO2CarbonDB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:derby://localhost:1527/db;create=true</url>
            <username>regadmin</username>
            <password>regadmin</password>
            <driverClassName>org.apache.derby.jdbc.ClientDriver</driverClassName>
            <maxActive>80</maxActive>
            <maxWait>60000</maxWait>
            <minIdle>5</minIdle>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
            <defaultAutoCommit>false</defaultAutoCommit>
        </configuration>
    </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
</tbody>
</table>
validationInterval
The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.

defaultAutoCommit
This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default.

When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see Tomcat JDBC Connection Pool.

In contrast to setting up with embedded Derby, in the remote registry you set the database driver name (the `driverName` element) to the value `org.apache.derby.jdbc.ClientDriver` and the database URL (the `url` element) to the database remote location.

Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.

2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
   <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

   ```xml
   <Configuration>
   <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

Creating database tables

You can create database tables by executing the following script(s):

1. Run the `ij` tool located in the `<DERBY_HOME>/bin` directory.
2. Create the database and connect to it using the following command inside the `ij` prompt:

```sql
connect
'jdbc:derby://localhost:1527/db;user=regadmin;password=regadmin;create=true';
```

Replace the database file path, user name, and password in the above command to suit your requirements.

3. Exit from the `ij` tool by typing the `exit` command as follows:

```sql
exit;
```

4. Log in to the `ij` tool with the username and password you just used to create the database.

```sql
connect 'jdbc:derby://localhost:1527/db' user 'regadmin' password 'regadmin';
```

5. You can create database tables manually by executing the following scripts.

   - To create tables in the registry and user manager database (`WSO2CARBON_DB`), use the below script:

```sql
run '<PRODUCT_HOME>/dbscripts/derby.sql';
```

6. Restart the server.

You can create database tables automatically when starting the product for the first time by using the `-Dsetup` parameter as follows:

   - For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
   - For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

The product is now configured to run using a remote Apache Derby database.

**Setting up H2**

You can set up either an embedded H2 database or a remote one using the instructions given below:

- Setting up Embedded H2
- Setting up Remote H2

**Setting up Embedded H2**

The following sections describe how to replace the default H2 databases with Embedded H2:

- Setting up the database
- Setting up the drivers
- Setting up datasource configurations
- Creating database tables
Setting up the database

Download and install the H2 database engine in your computer.

For instructions on installing DB2 Express-C, see H2 installation guide.

Setting up the drivers

WSO2 currently ships H2 database engine version h2-1.2.140.* and its related H2 database driver. If you want to use a different H2 database driver, take the following steps:

1. Delete the following H2 database-related JAR file, which is shipped with WSO2 products:
   `<PRODUCT_HOME>/repository/components/plugins/h2-database-engine_1.2.140.wso2v3.jar`

2. Find the JAR file of the new H2 database driver (`<H2_HOME>/bin/h2-*.jar`, where `<H2_HOME>` is the H2 installation directory) and copy it to your WSO2 product's `<PRODUCT_HOME>/repository/components/lib` directory.

Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Embedded H2 database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.
The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>parameter</td>
<td>description</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
<tr>
<td>validationInterval</td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td>defaultAutoCommit</td>
<td>This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see Tomcat JDBC Connection Pool.

Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB` datasource above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
   <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

   ```xml
   <Configuration>
   <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

Creating database tables

To create the database tables, connect to the database that you created earlier and run the following scripts in the H2 shell or web console:

- To create tables in the registry and user manager database (`WSO2CARBON_DB`), use the below script:
Follow the steps below to run the script in Web console:

1. Run the `.h2.sh` command to start the Web console.
2. Copy the script text from the SQL file.
3. Paste it into the console.
4. Click Run.
5. Restart the server.

You can create database tables automatically **when starting the product for the first time** by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

### Setting up Remote H2

The following sections describe how to replace the default H2 databases with Remote H2:

- Setting up the remote H2 database
- Setting up the drivers
- Setting up datasource configurations
- Creating database tables

### Setting up the remote H2 database

Follow the steps below to set up a Remote H2 database.

1. Download and install the H2 database engine on your computer as follows.

   For instructions on installing, see the H2 installation guide.

   ```
   client@wso2:~$ dbb$ wget -c http://www.h2database.com/h2-2011-09-11.zip
   Resolving www.h2database.com... 80.74.147.171
   Connecting to www.h2database.com|80.74.147.171|:80... connected.
   HTTP request sent, waiting response... 200 OK
   Length: 5007851 (5.7M) [application/zip]
   Saving to: "h2-2011-09-11.zip"
   15% [==================================] 1923304 111K/s eta 45s
   ```

2. Go to the `<H2_HOME>/bin/` directory and run the H2 network server starting script as follows, where `<H2_HOME>` is the H2 installation directory:
3. Run the H2 database server with the following commands:
   - For Linux:
     $ ./h2.sh
   - For Windows:
     $ h2.bat

   The script starts the database engine and opens a pop-up window.

4. Click **Start Browser** to open a web browser containing a client application, which you use to connect to a database. If a database does not already exist by the name you provided in the **JDBC URL** text box, H2 will automatically create a database.

**Setting up the drivers**

WSO2 currently ships H2 database engine version h2-1.2.140.* and its related H2 database driver. If you want to use a different H2 database driver, take the following steps:

1. Delete the following H2 database-related JAR file, which is shipped with WSO2 products:
   `<PRODUCT_HOME>/repository/components/plugins/h2-database-engine_1.2.140.wso2v3.jar`

2. Find the JAR file of the new H2 database driver (`<H2_HOME>/bin/h2-*.jar`, where `<H2_HOME>` is the H2 installation directory) and copy it to your WSO2 product's `<PRODUCT_HOME>/repository/components/lib/` directory.

**Setting up datasource configurations**

A datasource is used to establish the connection to a database. By default, **WSO2_CARBON_DB** datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Remote H2 database to replace the default H2 database, either **change the default configurations of the WSO2_CARBON_DB datasource**, or **configure a new datasource** to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/` directory.
<datasource>
   <name>WSO2_CARBON_DB</name>
   <description>The datasource used for registry and user manager</description>
   <jndiConfig>
      <name>jdbc/WSO2CarbonDB</name>
   </jndiConfig>
   <definition type="RDBMS">
      <configuration>
         <url>jdbc:h2:tcp://localhost/~/registryDB;create=true</url>
         <username>regadmin</username>
         <password>regadmin</password>
         <driverClassName>org.h2.Driver</driverClassName>
         <maxActive>80</maxActive>
         <maxWait>60000</maxWait>
         <minIdle>5</minIdle>
         <testOnBorrow>true</testOnBorrow>
         <validationQuery>SELECT 1</validationQuery>
         <validationInterval>30000</validationInterval>
         <defaultAutoCommit>false</defaultAutoCommit>
      </configuration>
   </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
</tbody>
</table>
### validationInterval

The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.

### defaultAutoCommit

This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default.

When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.

---

**For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see Tomcat JDBC Connection Pool.**

---

Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
   <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

   ```xml
   <Configuration>
   <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

---

**Creating database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts in H2 shell or web console:

- To create tables in the registry and user manager database (`WSO2CARBON_DB`), use the below script:

  ```
  <PRODUCT_HOME>/dbscripts/h2.sql
  ```
Follow the steps below to run the script in Web console:

1. Run the `./h2.sh` command to start the Web console.
2. Copy the script text from the SQL file.
3. Paste it into the console.
4. Click Run.
5. Restart the server.

You can create database tables automatically **when starting the product for the first time** by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

**Setting up IBM DB2**

The following sections describe how to replace the default H2 databases with IBM DB2:

- Prerequisites
- Setting up the database and users
- Setting up DB2 JDBC drivers
- Setting up datasource configurations
- Creating database tables

**Prerequisites**
Download the latest version of **DB2 Express-C** and install it on your computer.

For instructions on installing DB2 Express-C, see this [ebook](#).

---

### Setting up the database and users

Create the database using either **DB2 command processor** or **DB2 control center** as described below.

#### Using the DB2 command processor

1. Run DB2 console and execute the `db2start` command in CLI to open DB2.
2. Create the database using the following command:
   ```
   create database <DB_NAME>
   ```
3. Before issuing a SQL statement, establish the connection to the database using the following command:
   ```
   connect to <DB_NAME> user <USER_ID> using <PASSWORD>
   ```
4. Grant required permissions for users as follows:
   ```
   connect to DB_NAME
   grant <AUTHORITY> on database to user <USER_ID>
   ```

   For example:

   ![DB2 Command Processor Example](image)

   For more information on DB2 commands, see the **DB2 Express-C Guide**.

#### Using the DB2 control center

1. Open the DB2 control center using the `db2cc` command as follows:
   ```
   greg@so2:-/sqllib/bin$ ./db2cc
   ```

2. Right-click **All Databases** in the control center tree (inside the object browser), click **Create Database**, and then click **Standard** and follow the steps in the **Create New Database** wizard.
3. Click **User and Group Objects** in the control center tree to create users for the newly created database.

4. Give the required permissions to the newly created users.
Setting up DB2 JDBC drivers

Copy the DB2 JDBC drivers (db2jcc.jar and db2jcc_license_c00.jar) from <DB2_HOME>/SQLLIB/java/directory to the <PRODUCT_HOME>/repository/components/lib/ directory.

<DB2_HOME> refers to the installation directory of DB2 Express-C, and <PRODUCT_HOME> refers to the directory where you run the WSO2 product instance.

Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the IBM DB2 database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource
Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.

```xml
<datasource>
    <name>WSO2_CARBON_DB</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/WSO2CarbonDB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:db2://SERVER_NAME:PORT/DB_NAME</url>
            <username>regadmin</username>
            <password>regadmin</password>
            <driverClassName>com.ibm.db2.jcc.DB2Driver</driverClassName>
            <maxActive>80</maxActive>
            <maxWait>360000</maxWait>
            <minIdle>5</minIdle>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
            <defaultAutoCommit>false</defaultAutoCommit>
        </configuration>
    </definition>
</datasource>
```

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
<tr>
<td>validationInterval</td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td>defaultAutoCommit</td>
<td>This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/ master-datasources.xml` file, see [Tomcat JDBC Connection Pool](#).

### Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/ master-datasources.xml` file. Change its elements with your custom values. For instructions, see [Setting up datasource configurations](#).

2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/ registry.xml` file.

```xml
<dbConfig name="wso2registry">
    <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
</dbConfig>
```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/ user-mgt.xml` file.

```xml
<Configuration>
    <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
</Configuration>
```

### Creating database tables

To create the database tables, connect to the database that you created earlier and run the following scripts in the...
DB2 Express-C command editor.

1. To create tables in the registry and user manager database (WSO2CARBON_DB), use the below script:

   ```<PRODUCT_HOME>/dbscripts/db2.sql```

2. Restart the server.

   You can create database tables automatically **when starting the product for the first time** by using the `-D setup` parameter as follows:

   - For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
   - For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

---

**Setting up IBM Informix**

The following sections describe how to replace the default H2 databases with IBM Informix:

- Prerequisites
- Creating the database
- Setting up Informix JDBC drivers
- Setting up datasource configurations
- Creating database tables

**Prerequisites**

Download the latest version of IBM Informix and install it on your computer.

**Creating the database**

Create the database and users in Informix. For instructions on creating the database and users, see Informix product documentation.

Do the following changes to the default database when creating the Informix database.

- Use page size as 4K or higher when creating the dbspace as shown in the following command (i.e. denoted by `-k 4`):
  
  ```onspaces -c -S testspace4 -k 4 -p /usr/informix/logdir/data5.dat -o 100 -s 3000000```

- Add the following system environment variables.

  ```
  export DB_LOCALE=en_US.UTF-8
  export CLIENT_LOCALE=en_US.UTF-8
  ```

- Create a sbspace other than the dbspace by executing the following command:
  
  ```onspaces -c -S testspace4 -k 4 -p /usr/informix/logdir/data5.dat -o 100 -s 3000000```

- Add the following entry to the `<INFORMIX_HOME>/etc/onconfig` file, and replace the given example sbspace name (i.e. `testspace4`) with your sbspace name: `SBSPACENAME testspace4`
Setting up Informix JDBC drivers

Download the Informix JDBC drivers and copy them to your WSO2 product's `<PRODUCT_HOME>/repository/components/lib/` directory.

Use Informix JDBC driver version 3.70.JC8, 4.10.JC2 or higher.

Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, `WSO2_CARBON_DB` datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the IBM Informix database to replace the default H2 database, either change the default configurations of the `WSO2_CARBON_DB` datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default `WSO2_CARBON_DB` datasource

Follow the steps below to change the type of the default `WSO2_CARBON_DB` datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.

```xml
<datasource>
    <name>WSO2AM_DB</name>
    <description>The datasource used for API Manager database</description>
    <jndiConfig>
        <name>jdbc/WSO2AM_DB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <!-- IP ADDRESS AND PORT OF DB SERVER -->
            <url>jdbc:informix-sqli://localhost:1533/AM_DB;CLIENT_LOCALE=en_US.utf8;DB_LOCALE=en_us.utf8;IFX_USE_STRENC=true;</url>
            <username>wso2carbon</username>
            <password>wso2carbon</password>
            <driverClassName>com.informix.jdbc.IfxDriver</driverClassName>
            <maxActive>80</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
            <defaultAutoCommit>false</defaultAutoCommit>
        </configuration>
    </definition>
</datasource>
```

The elements in the above configuration are described below:
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000. You need to add the following configuration when specifying the connection URL as shown in the example above: <code>CLIENT_LOCALE=en_US.utf8;DB_LOCALE=en_us.utf8;_STRENC=true;</code></td>
</tr>
<tr>
<td>username</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>password</td>
<td></td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from the pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without any extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning to the caller.</td>
</tr>
<tr>
<td>validationInterval</td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td>defaultAutoCommit</td>
<td>This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS driver by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see [Tomcat JDBC Connection Pool](#).

**Configuring new datasources to manage registry or user management data**

Follow the steps below to configure new datasources to point to the new databases you create to manage registry...
1. Add a new datasource with similar configurations as the WSO2_CARBON_DB datasource above to the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.
2. If you are setting up a separate database to store registry-related data, update the following configurations in the <PRODUCT_HOME>/repository/conf/registry.xml file.

```xml
<dbConfig name="wso2registry">
  <dataSource>j dbc/ MY_DATASOURCE_NAME</dataSource>
</dbConfig>
```

3. If you are setting up a separate database to store user management data, update the following configurations in the <PRODUCT_HOME>/repository/conf/user-mgt.xml file.

```xml
.Configuration>
  <Property name="dataSource">jdbc/ MY_DATASOURCE_NAME</Property>
</Configuration>
```

**Creating database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts.

1. To create tables in the registry and user manager database (WSO2CARBON_DB), use the below script:

   ```shell
   <PRODUCT_HOME>/dbscripts/informix.sql
   ```

2. Restart the server.

   You can create database tables automatically when starting the product for the first time by using the -Dsetup parameter as follows:

   - For Windows: <PRODUCT_HOME>/bin/wso2server.bat -Dsetup
   - For Linux: <PRODUCT_HOME>/bin/wso2server.sh -Dsetup

**Setting up Microsoft SQL**

The following sections describe how to replace the default H2 database with MS SQL:

- Setting up the database and users
- Setting up the JDBC driver
- Setting up datasource configurations
- Creating the database tables

**Setting up the database and users**
Follow the steps below to set up the Microsoft SQL database and users.

**Enable TCP/IP**

1. In the start menu, click Programs and launch Microsoft SQL Server 2005.
2. Click Configuration Tools, and then click SQL Server Configuration Manager.
3. Enable TCP/IP and disable Named Pipes from protocols of your Microsoft SQL server.
4. Double click TCP/IP to open the TCP/IP properties window, and set Listen All to Yes on the Protocol tab.
5. On the IP Address tab, disable TCP Dynamic Ports by leaving it blank and give a valid TCP port, so that Microsoft SQL server will listen on that port.

   The best practice is to use port 1433, because you can use it in order processing services.

6. Similarly, enable TCP/IP from SQL Native Client Configuration and disable Named Pipes. Also check whether the port is set correctly to 1433.
7. Restart Microsoft SQL Server.

Create the database and user

1. Open Microsoft SQL Management Studio to create a database and user.
2. Click New Database from the Database menu, and specify all the options to create a new database.
3. Click New Login from the Logins menu, and specify all the necessary options.

Grant permissions

Assign newly created users the required grants/permissions to log in, create tables, and insert, index, select, update, and delete data in tables in the newly created database, as the minimum set of SQL server permissions.

**Setting up the JDBC driver**

Download and copy the sqljdbc4 Microsoft SQL JDBC driver file to the WSO2 product's `<PRODUCT_HOME>/repository/components/lib/` directory. Use `com.microsoft.sqlserver.jdbc.SQLServerDriver` as the `<driverClassName>` in your datasource configuration in `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file.

**Setting up datasource configurations**

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Microsoft SQL database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

**Changing the default WSO2_CARBON_DB datasource**

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.
<datasource>
   <name>WSO2_CARBON_DB</name>
   <description>The datasource used for registry and user manager</description>
   <jndiConfig>
      <name>jdbc/WSO2CarbonDB</name>
   </jndiConfig>
   <definition type="RDBMS">
      <configuration>
         <url>jdbc:sqlserver://<IP>:1433;databaseName=wso2greg</url>
         <username>regadmin</username>
         <password>regadmin</password>
         <driverClassName>com.microsoft.sqlserver.jdbc.SQLServerDriver</driverClassName>
         <maxActive>80</maxActive>
         <maxWait>60000</maxWait>
         <testOnBorrow>true</testOnBorrow>
         <validationQuery>SELECT 1</validationQuery>
         <validationInterval>30000</validationInterval>
         <defaultAutoCommit>false</defaultAutoCommit>
      </configuration>
   </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. Change the <code>&lt;IP&gt;</code> with the IP of the server. The best practice is to use port 1433, because you can use it in order processing services.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>validationInterval</td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td>defaultAutoCommit</td>
<td>This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see Tomcat JDBC Connection Pool.

**Configuring new datasources to manage registry or user management data**

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the WSO2_CARBON_DB datasource above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

```xml
<dbConfig name="wso2registry">
    <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
</dbConfig>
```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

```xml
<Configuration>
    <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
</Configuration>
```

**Creating the database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts.

1. To create tables in the registry and user manager database (WSO2CARBON_DB), use the below script:
2. Restart the server.

You can create database tables automatically when starting the product for the first time by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

**Setting up MySQL**

The following sections describe how to replace the default H2 databases with MySQL:

- Setting up the database and users
- Setting up the drivers
- Setting up datasource configurations
- Creating database tables
- Changing the registry/user management databases

**Setting up the database and users**

Follow the steps below to set up a MySQL database:

1. Download and install MySQL on your computer using the following command:

```
For instructions on installing MySQL on MAC OS, go to Homebrew.
```

```
sudo apt-get install mysql-server mysql-client
```

2. Start the MySQL service using the following command:

```
sudo /etc/init.d/mysql start
```

3. Log in to the MySQL client as the root user (or any other user with database creation privileges).

```
mysql -u root -p
```

4. Enter the password when prompted.

```
In most systems, there is no default root password. Press the Enter key without typing anything if you have not changed the default root password.
```

5. In the MySQL command prompt, create the database using the following command:

```
create database regdb;
```
For users of Microsoft Windows, when creating the database in MySQL, it is important to specify the character set as latin1. Failure to do this may result in an error (error code: 1709) when starting your cluster. This error occurs in certain versions of MySQL (5.6.x), and is related to the UTF-8 encoding. MySQL originally used the latin1 character set by default, which stored characters in a 2-byte sequence. However, in recent versions, MySQL defaults to UTF-8 to be friendlier to international users. Hence, you must use latin1 as the character set as indicated below in the database creation commands to avoid this problem. Note that this may result in issues with non-latin characters (like Hebrew, Japanese, etc.). The database creation command should be as follows:

```sql
mysql> create database <DATABASE_NAME> character set latin1;
```

For users of other operating systems, the standard database creation commands will suffice. For these operating systems, the database creation command should be as follows:

```sql
mysql> create database <DATABASE_NAME>;
```

6. Give authorization of the database to the regadmin user as follows:

```sql
GRANT ALL ON regdb.* TO regadmin@localhost IDENTIFIED BY "regadmin";
```

7. Once you have finalized the permissions, reload all the privileges by executing the following command:

```sql
FLUSH PRIVILEGES;
```

8. Log out from the MySQL prompt by executing the following command:

```sql
quit;
```

**Setting up the drivers**

Download the MySQL Java connector JAR file, and copy it to the `<PRODUCT_HOME>/repository/components/lib` directory.

**Tip:** Be sure to use the connector version that is supported by the MySQL version you use. If you come across any issues due to version incompatibility, follow the steps below:

1. Shut down the server and remove all existing connectors from `<PRODUCT_HOME>/repository/components/lib` and `<PRODUCT_HOME>/repository/components/dropins`.
2. Download the connector JAR that is compatible with your current MySQL version.
3. Copy the JAR file only to `<PRODUCT_HOME>/repository/components/lib`. Files will be copied automatically to the dropins folder at the server startup.
4. Start the server with the `-Dsetup` parameter as `sh wso2server.sh -Dsetup`.

**Setting up datasource configurations**

A datasource is used to establish the connection to a database. By default, `WSO2_CARBON_DB` datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the MySQL database to replace the default H2 database, either change the default configurations of the `WSO2_CARBON_DB` datasource, or configure a new datasource to point it to the new database as explained below.

**Changing the default `WSO2_CARBON_DB` datasource**

Follow the steps below to change the type of the default `WSO2_CARBON_DB` datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://localhost:3306/regdb</url>
      <username>regadmin</username>
      <password>regadmin</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>80</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for MySQL is 3306</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
</tbody>
</table>
### validationQuery
The SQL query that will be used to validate connections from this pool before returning them to the caller.

### validationInterval
The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.

### defaultAutoCommit
This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default.

When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.

---

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/ master-datasources.xml` file, see [Tomcat JDBC Connection Pool](#).

---

**Configuring new datasources to manage registry or user management data**

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/`. Change its elements with your custom values. For instructions, see [Setting up datasource configurations](#).

2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/ registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
      <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/ user-mgt.xml` file.

   ```xml
   <Configuration>
      <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

**Creating database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts

1. To create tables in the registry and user manager database (WSO2CARBON_DB), use the below script:

   ```sql
   You may have to enter the password for each command when prompted.
   ```
If you are using MySQL version 5.7 or later, use the following script instead:

```
mysql -u regadmin -p -Dregdb '<PRODUCT_HOME>/dbscripts/mysql5.7.sql';
```

If you prefer to populate MySQL 5.7 or later with the -Dsetup command, rename `mysql5.7.sql` to `mysql.sql`.

2. Restart the server.

You can create database tables automatically when starting the product for the first time by using the -Dsetup parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

Changing the registry/user management databases

If you change the database that comes by default or set up a separate database for registry or user management related data, follow the below instructions.

1. Add the datasource to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.

Setting up OpenEdge

The following sections describe how to replace the default H2 databases with OpenEdge (OE):

- Setting up the database and users
- Setting up the drivers
- Setting up datasource configurations
- Creating database tables

Setting up the database and users

Follow the steps below to set up an OpenEdge (OE) database.

1. Download and install OpenEdge on you computer.
2. Go to the `<OE_HOME>/bin/` directory and use the `proenv` script to set up the environment variables.
3. Add `<OE_HOME>/java/prosp.jar` to the CLASSPATH environment variable.
4. Create an empty database using the `prodb` script as follows. This script creates a database by copying an existing database provided with the installation.
prodB CARBON_DB <OE-installation-directory>/empty8

5. Start the database using the proserve script as follows. Provide the database name and a port as arguments to this script using the -db and -S parameters.

   proserve -db CARBON_DB -S 6767

6. Use the sqlexp script to start the default SQL explorer that comes with the OpenEdge installation. Connect to the database you just created by using the -db and -S parameters as follows:

   sqlexp -db CARBON_DB -S 6767

7. Use the following commands to create a user and grant that user the required permissions to the database:

   CREATE USER 'wso2carbon','wso2carbon';
   GRANT dba,resource TO 'wso2carbon';
   COMMIT;

8. Log out from the SQL explorer by typing the following command: exit

Setting up the drivers

Copy the <OE_HOME>/java/openedge.jar file to your WSO2 product's <PRODUCT_HOME>/repository/components/lib/ directory.

Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the OpenEdge database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file as shown below.
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:datadirect:openedge://localhost:6767;databaseName=CARBON_DB</url>
      <username>regadmin</username>
      <password>regadmin</password>
      <driverClassName>com.ddtek.jdbc.openedge.OpenEdgeDriver</driverClassName>
      <maxActive>80</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
</tbody>
</table>
The SQL query that will be used to validate connections from this pool before returning them to the caller.

The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.

This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default.

When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see Tomcat JDBC Connection Pool.

**Configuring new datasources to manage registry or user management data**

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB` datasource above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.

2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
     <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

   ```xml
   <Configuration>
     <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

**Creating database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts

- To create tables in the registry and user manager database (`WSO2CARBON_DB`), use the below script:
Follow the steps below to create the database tables by executing the scripts.

1. Modify the OpenEdge script provided with the product to create the tables manually. Make a backup of the `<PRODUCT_HOME>/dbscripts/openedge.sql` script under the name `openedge_manual.sql`.
2. Replace all the "/" symbols in the `openedge_manual.sql` script with the ":" symbol.
3. At the end of the `openedge_manual.sql` script, add the following line and save the script:
   
   ```sql
   COMMIT;
   ```
4. Run the modified script using the SQL explorer as follows:

   ```
   sqlexp -db CARBON_DB -S 6767 -user wso2carbon -password wso2carbon <PRODUCT_HOME>/dbscripts/openedge_manual.sql
   ```
5. Restart the server.

You can create database tables automatically **when starting the product for the first time** by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

### Setting up Oracle

The following sections describe how to replace the default H2 databases with Oracle:

- Setting up the database and users
- Setting up the JDBC driver
- Setting up datasource configurations
- Creating the database tables

**Setting up the database and users**

Follow the steps below to set up a Oracle database.

1. Create a new database by using the Oracle database configuration assistant (dbca) or manually.
2. Make the necessary changes in the Oracle `tnsnames.ora` and `listener.ora` files in order to define addresses of the databases for establishing connections to the newly created database.
3. After configuring the `.ora` files, start the Oracle instance using the following command:

   ```
   $ sudo /etc/init.d/oracle-xe restart
   ```
4. Connect to Oracle using SQL*Plus as SYSDBA as follows:

   ```
   $ ./$<ORACLE_HOME>/config/scripts/sqlplus.sh sysadm/password as SYSDBA
   ```
5. Connect to the instance with the username and password using the following command:
5. $ connect

6. As SYSDBA, create a database user and grant privileges to the user as shown below:

   Create user <USER_NAME> identified by password account unlock;
   grant connect to <USER_NAME>;
   grant create session, create table, create sequence, create trigger to <USER_NAME>;
   alter user <USER_NAME> quota <SPACE_QUOTA_SIZE_IN_MEGABYTES> on '<TABLE_SPACE_NAME>';
   commit;

7. Exit from the SQL*Plus session by executing the quit command.

Setting up the JDBC driver

1. Copy the Oracle JDBC libraries (for example, `<ORACLE_HOME>/jdbc/lib/ojdbc14.jar`) to the `<PRODUCT_HOME>/repository/components/lib/` directory.
2. Remove the old database driver from the `<PRODUCT_HOME>/repository/components/dropins/` directory.

   If you get a timezone region not found error when using the ojdbc6.jar with WSO2 servers, set the Java property as follows:

   ```shell
   export JAVA_OPTS="-Duser.timezone='+05:30'"
   ```

   The value of this property should be the GMT difference of the country. If it is necessary to set this property permanently, define it inside the `wso2server.sh` as a new `JAVA_OPT` property.

Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, **WSO2_CARBON_DB** datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Oracle database to replace the default H2 database, either change the default configurations of the **WSO2_CARBON_DB** datasource, or configure a new datasource to point it to the new database as explained below.

**Changing the default WSO2_CARBON_DB datasource**

Follow the steps below to change the type of the default **WSO2_CARBON_DB** datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.
The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td>validationQuery</td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
</tbody>
</table>
### validationInterval

The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.

### defaultAutoCommit

This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default.

When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.

---

The default port for Oracle is 1521.

---

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/ master-datasources.xml` file, see Tomcat JDBC Connection Pool.

---

**Configuring new datasources to manage registry or user management data**

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/ master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/ registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
     <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/ user-mgt.xml` file.

   ```xml
   <Configuration>
     <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

---

**Creating the database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts in SQL*Plus:

1. To create tables in the registry and user manager database (`WSO2CARBON_DB`), use the below script:
SQL> @$<PRODUCT_HOME>/dbscripts/oracle.sql

2. Restart the server.

You can create database tables automatically when starting the product for the first time by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

Setting up Oracle RAC

The following sections describe how to replace the default H2 databases with Oracle RAC:

- Setting up the database and users
- Setting up the JDBC driver
- Setting up datasource configurations
- Creating the database tables

Oracle Real Application Clusters (RAC) is an option for the Oracle Database for clustering and high availability in Oracle database environments. In the Oracle RAC environment, some of the commands used in `oracle.sql` are considered inefficient. Therefore, the product has a separate SQL script `oracle_rac.sql` for Oracle RAC. The Oracle RAC-friendly script is located in the `dbscripts` folder together with other `.sql` scripts.

To test products on Oracle RAC, rename `oracle_rac.sql` to `oracle.sql` before running `-Dsetup`.

Setting up the database and users

Follow the steps below to set up an Oracle RAC database.

1. Set environment variables `ORACLE_HOME`, `PATH`, and `ORACLE_SID` with the corresponding values `/oracle/app/oracle/product/11.2.0/dbhome_1`, `$PATH:<ORACLE_HOME>/bin`, and `orcl1` as follows:

   ```bash
   [oracle@node1 ~]$ export ORACLE_HOME=/oracle/app/oracle/product/11.2.0/dbhome_1
   [oracle@node1 ~]$ export PATH=$PATH:$ORACLE_HOME/bin
   [oracle@node1 ~]$ export ORACLE_SID=orcl1
   ```

2. Connect to Oracle using SQL*Plus as SYSDBA.
3. Create a database user and grant privileges to the user as shown below:

```
CREATE user <USER_NAME> identified by password account unlock;
grant connect to <USER_NAME>;
grant create session, create table, create sequence, create trigger to <USER_NAME>;
alter user <USER_NAME> quota <SPACE_QUOTA_SIZE_IN_MEGABYTES> on '<TABLE_SPACE_NAME>';
commit;
```

4. Exit from the SQL*Plus session by executing the `quit` command.

**Setting up the JDBC driver**

Copy the Oracle JDBC libraries (for example, the `<ORACLE_HOME>/jdbc/lib/ojdbc14.jar` file) to the `<PRODUCT_HOME>/repository/components/lib/` directory.

Remove the old database driver from the `<PRODUCT_HOME>/repository/components/dropins/` directory when you upgrade the database driver.
Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the Oracle RAC database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

Changing the default WSO2_CARBON_DB datasource

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file as shown below.

   ```xml
   <datasource>
     <name>WSO2_CARBON_DB</name>
     <description>The datasource used for registry and user manager</description>
     <jndiConfig>
       <name>jdbc/WSO2CarbonDB</name>
     </jndiConfig>
     <definition type="RDBMS">
       <configuration>
         <url>jdbc:oracle:thin:@(DESCRIPTION=(LOAD_BALANCE=on)
           (ADDRESS=(PROTOCOL=TCP)(HOST=racnode1) (PORT=1521))
           (ADDRESS=(PROTOCOL=TCP)(HOST=racnode2) (PORT=1521))
           (CONNECT_DATA=(SERVICE_NAME=rac)))</url>
         <username>regadmin</username>
         <password>regadmin</password>
         <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
         <maxActive>80</maxActive>
         <maxWait>60000</maxWait>
         <minIdle>5</minIdle>
         <testOnBorrow>true</testOnBorrow>
         <validationQuery>SELECT 1 FROM DUAL</validationQuery>
         <validationInterval>30000</validationInterval>
         <defaultAutoCommit>false</defaultAutoCommit>
       </configuration>
     </definition>
   </datasource>
   ```

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a DB2 instance is 50000.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>maxActive</code></td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td><code>maxWait</code></td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td><code>minIdle</code></td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td><code>testOnBorrow</code></td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td><code>validationQuery</code></td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
<tr>
<td><code>validationInterval</code></td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td><code>defaultAutoCommit</code></td>
<td>This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see [Tomcat JDBC Connection Pool](#).

### Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBON_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see [Setting up datasource configurations](#).
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file:

   ```xml
   <dbConfig name="wso2registry">
     <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.
Creating the database tables

To create the database tables, connect to the database that you created earlier and run the following scripts in SQL*Plus:

1. To create tables in the registry and user manager database (WSO2CARBON_DB), use the below script:

   ```sql
   SQL> @$<PRODUCT_HOME>/dbscripts/oracle.sql
   ```

2. Restart the server.

   You can create database tables automatically when starting the product for the first time by using the -Dsetup parameter as follows:
   - For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
   - For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

Setting up PostgreSQL

The following sections describe how to replace the default H2 databases with PostgreSQL:

- Setting up the database and login role
- Setting up the drivers
- Setting up datasource configurations
- Creating database tables

Setting up the database and login role

Follow the steps below to set up a PostgreSQL database.

1. Install PostgreSQL on your computer as follows:

   ```bash
   client@wso2:$ sudo apt-get install postgresql
   ```

2. Start the PostgreSQL service using the following command:

   ```bash
   client@wso2:$ sudo /etc/init.d/postgresql start
   Starting PostgreSQL 8.4 database server: main.
   client@wso2:$
   ```

3. Create a database and the login role from a GUI using the PGAdminIII tool.
4. To connect PGAdminIII to a PostgreSQL database server, locate the server from the object browser, right-click the client, and click Connect. This will show you the databases, tablespaces, and login roles as follows:
4. To create a database, click **Databases** in the tree (inside the object browser), and click **New Database**.
5. In the **New Database** dialog box, give a name to the database (for example: gregdb) and click **OK**.
6. To create a login role, click **Login Roles** in the tree (inside the object browser), and click **New Login Role**. Enter the role name and a password.

   These values will be used in the product configurations as described in the following sections. In the sample configuration, **gregadmin** will be used as both the role name and the password.

7. Optionally enter other policies, such as the expiration time for the login and the connection limit.
8. Click **OK** to finish creating the login role.

### Setting up the drivers

1. Download the PostgreSQL JDBC4 driver.
2. Copy the driver to your WSO2 product's `<PRODUCT_HOME>/repository/components/lib` directory.

### Setting up datasource configurations

A datasource is used to establish the connection to a database. By default, **WSO2_CARBON_DB** datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the PostgreSQL database to replace the default H2 database, either change the default configurations of the **WSO2_CARBON_DB** datasource, or configure a new datasource to point it to the new database as explained below.

#### Changing the default **WSO2_CARBON_DB** datasource

Follow the steps below to change the type of the default **WSO2_CARBON_DB** datasource.
1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.

```xml
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:postgresql://localhost:5432/gregdb</url>
      <username>regadmin</username>
      <password>regadmin</password>
      <driverClassName>org.postgresql.Driver</driverClassName>
      <maxActive>80</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <defaultAutoCommit>true</defaultAutoCommit>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>
```

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for a PostgreSQL instance is 5432.</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user.</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver.</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td><strong>testOnBorrow</strong></td>
<td>Whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td><strong>validationInterval</strong></td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td><strong>defaultAutoCommit</strong></td>
<td>Whether to commit database changes automatically or not. This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see Tomcat JDBC Connection Pool.

**Configuring new datasources to manage registry or user management data**

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the WSO2_CARBON_DB datasource above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see Setting up datasource configurations.

2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/registry.xml` file.

   ```xml
   <dbConfig name="wso2registry">
     <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

   ```xml
   <Configuration>
     <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

**Creating database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts.
1. To create tables in the registry and user manager database (WSO2CARBON_DB), use the below script:

```
<PRODUCT_HOME>/dbscripts/postgresql.sql
```

2. Restart the server.

You can create database tables automatically **when starting the product for the first time** by using the `-Dsetup` parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

### Setting up MariaDB

The following sections describe how to replace the default H2 databases with MariaDB, which is a drop-in replacement for MySQL:

- Setting up the database and users
- Setting up the drivers
- Setting up datasource configurations
- Creating database tables

#### Setting up the database and users

Follow the steps below to set up MariaDB. See Tested DBMSs for information on the MariaDB versions that we have tested the WSO2 products with.

1. Download, install and start MariaDB on your computer. See [https://downloads.mariadb.org/](https://downloads.mariadb.org/).

   You can install MariaDB standalone or as a galera cluster for high availability. Database clustering is independent of WSO2 product clustering. For more information on setting up a galera cluster, see the MariaDB Galera Cluster documentation.

   For instructions on installing MariaDB on MAC OS, go to [Homebrew](https://brew.sh).  

2. Log in to MariaDB as the root user (or any other user with database creation privileges).

   ```
   mysql -u root -p
   ```

3. Enter the password when prompted.

   In most systems, there is no default root password. Press the Enter key without typing anything if you have not changed the default root password.

4. In the MySQL command prompt, create the database using the following command:
create database regdb;

5. Give authorization of the database to the regadmin user as follows:
   
   ```sql
   GRANT ALL ON regdb.* TO regadmin@localhost IDENTIFIED BY "regadmin";
   ```

6. Once you have finalized the permissions, reload all the privileges by executing the following command:
   
   ```sql
   FLUSH PRIVILEGES;
   ```

7. Log out from the MySQL prompt by executing the following command:
   
   ```sql
   quit;
   ```

**Setting up the drivers**

Download the MySQL Java connector **JAR file**, and copy it to the `<PRODUCT_HOME>/repository/components/lib/` directory.

```sql
Note that you must use the MySQL connector that is compatible with your MariaDB version. For example, mysql-connector-java-5.1.36-bin.jar is compatible with MariaDB version 10.0.20. See Tested DBMSs for information on which version of a WSO2 product has been tested for compatibility with which version of MariaDB and MySQL connector.
```

**Setting up datasource configurations**

A datasource is used to establish the connection to a database. By default, WSO2_CARBON_DB datasource is used to connect to the default H2 database, which stores registry and user management data. After setting up the MariaDB database to replace the default H2 database, either change the default configurations of the WSO2_CARBON_DB datasource, or configure a new datasource to point it to the new database as explained below.

**Changing the default WSO2_CARBON_DB datasource**

Follow the steps below to change the type of the default WSO2_CARBON_DB datasource.

1. Edit the default datasource configuration in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file as shown below.

```
Do not change the datasource name WSO2_CARBON_DB in the below configuration.
```
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://localhost:3306/regdb</url>
      <username>regadmin</username>
      <password>regadmin</password>
      <defaultAutoCommit>false</defaultAutoCommit>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>80</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>

The elements in the above configuration are described below:

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>url</td>
<td>The URL of the database. The default port for MariaDB is 3306</td>
</tr>
<tr>
<td>username and password</td>
<td>The name and password of the database user</td>
</tr>
<tr>
<td>driverClassName</td>
<td>The class name of the database driver</td>
</tr>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated at the same time from this pool. Enter any negative value to denote an unlimited number of active connections.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum number of milliseconds that the pool will wait (when there are no available connections) for a connection to be returned before throwing an exception. You can enter zero or a negative value to wait indefinitely.</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of active connections that can remain idle in the pool without extra ones being created, or enter zero to create none.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and another attempt will be made to borrow another.</td>
</tr>
<tr>
<td><strong>validationQuery</strong></td>
<td>The SQL query that will be used to validate connections from this pool before returning them to the caller.</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>validationInterval</strong></td>
<td>The indication to avoid excess validation, and only run validation at the most, at this frequency (time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
</tr>
<tr>
<td><strong>defaultAutoCommit</strong></td>
<td>This property is not applicable to the Carbon database in WSO2 products because auto committing is usually handled at the code level, i.e., the default auto commit configuration specified for the RDBMS driver will be effective instead of this property element. Typically, auto committing is enabled for an RDBMS drivers by default. When auto committing is enabled, each SQL statement will be committed to the database as an individual transaction, as opposed to committing multiple statements as a single transaction.</td>
</tr>
</tbody>
</table>

For more information on other parameters that can be defined in the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file, see [Tomcat JDBC Connection Pool](#).

### Configuring new datasources to manage registry or user management data

Follow the steps below to configure new datasources to point to the new databases you create to manage registry and/or user management data separately.

1. Add a new datasource with similar configurations as the `WSO2_CARBN_DB datasource` above to the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file. Change its elements with your custom values. For instructions, see [Setting up datasource configurations](#).
2. If you are setting up a separate database to store registry-related data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/` file:

   ```xml
   <dbConfig name="wso2registry">
   <dataSource>jdbc/MY_DATASOURCE_NAME</dataSource>
   </dbConfig>
   ```

3. If you are setting up a separate database to store user management data, update the following configurations in the `<PRODUCT_HOME>/repository/conf/` file:

   ```xml
   <Configuration>
   <Property name="dataSource">jdbc/MY_DATASOURCE_NAME</Property>
   </Configuration>
   ```

### Creating database tables

To create the database tables, connect to the database that you created earlier and run the following scripts.

1. To create tables in the registry and user manager database (`WSO2CARBN_DB`), use the below script:
1. mysql -u regadmin -p -Dregdb < 'PRODUCT_HOME>/dbscripts/mysql.sql';

2. Restart the server.

You can create database tables automatically when starting the product for the first time by using the -Dsetup parameter as follows:

- For Windows: `<PRODUCT_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<PRODUCT_HOME>/bin/wso2server.sh -Dsetup`

Managing Datasources

A datasource provides information that a server can use to connect to a database or to an external data store. The topics given below explain the ways in which new datasources can be created and edited in WSO2 products.

- Creating datasources using the management console
- Creating datasources using configuration files
- Editing datasource information

Creating datasources using the management console

The following feature in the WSO2 Feature Repository allows you to manage datasources using the management console of your product:

Name: WSO2 Carbon - datasource management feature
Identifier: org.wso2.carbon.datasource.feature.group

If this feature is not included in your product by default, you can install it using the Working with Features section of this documentation. With this feature, you can add any RDBMS datasource or your own custom datasources using the management console, which can later be edited or deleted for your product instance if required. Note that all datasources created using the management console are saved to the registry of your product. The steps for creating datasources are discussed in the following topics:

- Configuring an RDBMS Datasource
- Configuring a Custom Datasource

Creating datasources using configuration files

In all WSO2 products, you also have the option of creating datasources by simply creating a configuration file with the datasource information and storing it in the `<PRODUCT_HOME>/repository/conf/datasources/` directory. For example, all WSO2 products are shipped with the default WSO2_CARBON_DB database, and the datasource that is used to connect to this database is defined in the `master-datasources.xml` file stored in the datasources folder. You can also create your own custom datasources by creating configuration files and storing them in the same folder. For example, see the `custom-datasources.xml` file. The datasources defined using configuration files can be viewed using the management console if the datasource management feature is installed.
When adding an RDBMS datasource, be sure to copy the JDBC driver JAR file for your database to `<PRODUCT_HOME>/repository/components/lib`.

**Editing datasource information**

After datasources are created (either using the management console or by creating a separate configuration file), you can access them by clicking **Data Sources** on the **Configure** tab of the management console as shown below.

You can also edit and delete the datasources that are saved in the registry (i.e. datasources created using the management console). However, you cannot edit or delete the datasources that are added using configuration files in the `<PRODUCT_HOME>/repository/conf/datasources/` directory using the management console. For example, to modify the default `<WSO2_CARBON_DB>` datasource, you have to change the `master-datasources.xml` file.

**Configuring an RDBMS Datasource**

Follow the steps given below to create RDBMS datasources for your product server. RDBMS datasources allow you to service enable data from relational databases.

1. Log in to the management console of your product and click **Data Sources** on the **Configure** tab. The datasources that are already created will be listed.
2. Click **Add Data Source** to open the following screen:
3. Select ‘RDBMS’ for the **Data Source Type** field to define an RDBMS configuration.

4. Enter values for the remaining fields as explained below.
   - **Name**: Name of the datasource (must be a unique value).
   - **Data Source Provider**: Specify the datasource provider.
   - **Driver**: The class name of the JDBC driver to use.
   - **URL**: The connection URL to pass to the JDBC driver to establish the connection
   - **User Name**: The connection user name to pass to the JDBC driver to establish the connection
   - **Password**: The connection password to pass to the JDBC driver to establish the connection
   - **Expose as a JNDI Data Source**: Allows you to specify the JNDI data source.
   - **Data Source Configuration Parameters**: Allows you to specify the datasource connection pool parameters when creating an RDBMS datasource

5. Be sure to copy the JDBC driver relevant to the database engine to the `<PRODUCT_HOME>/repository/components/dropins` and `<PRODUCT_HOME>/repository/components/lib` directories. For example, if you are using MySQL, you would specify `com.mysql.jdbc.Driver` as the driver and would copy `mysql-connector-java-5.XX-bin.jar` to these directories. If you do not copy the driver to these directories when you create the datasource, you will get an exception similar to "Cannot load JDBC driver class com.mysql.jdbc.Driver".

6. After creating datasources, they appear on the **Data Sources** page. If required, you can edit or delete them by clicking **Edit** or **Delete**.

   This is the default RDBMS datasource configuration provided by WSO2. You can also write your own RDBMS configuration by selecting the **custom datasource** option.

---

Configuring a JNDI Datasource

Java Naming and Directory Interface (JNDI) is a Java application programming interface (API) that provides naming and directory functionality for Java software clients to discover and look up data and objects via a name. It helps decouple object creation from the object look-up. When you have registered a datasource with JNDI, others can discover it through a JNDI look-up and use it. Click **Expose as a JNDI Data Source** to display the JNDI fields as shown below.
Following are descriptions of the JNDI fields:

- **Name**: Name of the JNDI datasource that will be visible to others in object look-up
- **Use Data Source Factory**: To make the datasource accessible from an external environment, you must use a data source factory. When this option is selected, a reference object will be created with the defined datasource properties. The data source factory will create the datasource instance based on the values of the reference object when accessing the datasource from an external environment. In the datasource configuration, this is set as follows: `<jndiConfig useDataSourceFactory="true">`
- **JNDI Properties**: Properties related to the JNDI datasource (such as password). When you select this option, set the following properties:
  - `java.naming.factory.initial`: Selects the registry service provider as the initial context
  - `java.naming.provider.url`: Specifies the location of the registry when the registry is being used as the initial context

**Configuring the Datasource Provider**

A datasource provider connects to a source of data such as a database, accesses its data, and returns the results of the access queries. When creating an RDBMS datasource, you can use the default provider or link to an external provider.

**Default datasource provider**

To use the default datasource provider, select **default**, and then enter the connection properties Driver, URL, User Name, and Password as follows:
External datasource provider

If you need to add a datasource supported by an external provider class such as `com.mysql.jdbc.jdbc2.optional.MysqlXADataSource`, select **External Data Source**, click **Add Property**, and then enter the name and value of each connection property you need to configure. Following is an example datasource for an external datasource provider.

Configuring the Datasource Connection Pool Parameters

When the server processes a database operation, it spawns a database connection from an associated datasource. After using this connection, the server returns it to the pool of connections. This is called **datasource connection pooling** and is a recommended way to gain more performance/throughput in the system. In datasource connection pooling, the physical connection is not dropped with the database server unless it becomes stale or the datasource connection is closed.

RDBMS datasources in WSO2 products use Tomcat JDBC connection pool (`org.apache.tomcat.jdbc.pool`). It is common to all components that access databases for data persistence, such as the registry, user management (if configured against a JDBC userstore), etc.
You can configure the datasource connection pool parameters, such as how long a connection is persisted in the pool, using the datasource configuration parameters section that appears in the management console when creating a datasource. Click and expand the option as shown below:
Following are descriptions of the parameters you can configure. For more details on datasource configuration parameters, see [Tomcat JDBC Connection Pooling](#) and the [DBCP configuration guide](#).

The default values of the following parameters might not be optimal for the specific hardware/server configurations in your environment. We recommend you to carry out load tests in your environment to tune them accordingly.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction Isolation</td>
<td>The default TransactionIsolation state of connections created by this pool.</td>
</tr>
</tbody>
</table>
| | • TRANSACTION_UNKNOWN  
| | • TRANSACTION_NONE  
| | • TRANSACTION_READ_COMMITTED  
| | • TRANSACTION_READ_UNCOMMITTED  
| | • TRANSACTION_REPEATABLE_READ  
| | • TRANSACTION_SERIALIZABLE  
| Initial Size | (int)  
| | The initial number of connections created when the pool is started. Default value is 0. |
| Max. Active | (int)  
| | The maximum number of active connections that can be allocated from this pool at the same time.  
| | The default value is 100. |
| Min. Idle | (int)  
| | The minimum number of established connections that should be kept in the pool at all times.  
| | The connection pool can shrink below this number if validation queries fail. Default value is 0. (also see `testWhileIdle`) |
| Max. Wait | (int)  
| | Maximum number of milliseconds that the pool waits (when there are no available connections) for  
<p>| | a connection to be returned before throwing an exception. Default value is 30000 (30 seconds). |</p>
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation Query</td>
<td>(String)</td>
<td>The SQL query used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw a SQLException. The default value is null. Example values are <code>SELECT 1(mysql)</code>, <code>select 1 from dual(oracle)</code>, <code>SELECT 1(MS Sql Server)</code>.</td>
</tr>
<tr>
<td>Test On Return</td>
<td>(boolean)</td>
<td>Used to indicate if objects will be validated before returned to the pool. NOTE - for a true value to have any effect, the validationQuery parameter must be set to a non-null string. The default value is false.</td>
</tr>
<tr>
<td>Test On Borrow</td>
<td>(boolean)</td>
<td>Used to indicate if objects will be validated before borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another. NOTE - for a true value to have any effect, the validationQuery parameter must be set to a non-null string. In order to have a more efficient validation, see validationInterval. Default value is false.</td>
</tr>
<tr>
<td>Test While Idle</td>
<td>(boolean)</td>
<td>The indication of whether objects will be validated by the idle object evictor (if any). If an object fails to validate, it will be dropped from the pool. NOTE - for a true value to have any effect, the validationQuery parameter must be set to a non-null string. The default value is false and this property has to be set in order for the pool cleaner/test thread to run (also see timeBetweenEvictionRunsMillis).</td>
</tr>
<tr>
<td>Time Between Eviction Runs Mills</td>
<td>(int)</td>
<td>The number of milliseconds to sleep between runs of the idle connection validation/cleaner thread. This value should not be set under 1 second. It dictates how often we check for idle, abandoned connections, and how often we validate idle connections. The default value is 5000 (5 seconds).</td>
</tr>
<tr>
<td>Minimum Evictable Idle Time</td>
<td>(int)</td>
<td>The minimum amount of time an object may sit idle in the pool before it is eligible for eviction. The default value is 60000 (60 seconds).</td>
</tr>
<tr>
<td>Remove Abandoned</td>
<td>(boolean)</td>
<td>Flag to remove abandoned connections if they exceed the removeAbandonedTimeout. If set to true a connection is considered abandoned and eligible for removal if it has been in use longer than the removeAbandonedTimeout Setting this to true can recover db connections from applications that fail to close a connection. See also logAbandoned. The default value is false.</td>
</tr>
<tr>
<td>Remove Abandoned Timeout</td>
<td>(int)</td>
<td>Timeout in seconds before an abandoned(in use) connection can be removed. The default value is 60 (60 seconds). The value should be set to the longest running query your applications might have.</td>
</tr>
<tr>
<td>Log Abandoned</td>
<td>(boolean)</td>
<td>Flag to log stack traces for application code which abandoned a Connection. Logging of abandoned Connections adds overhead for every Connection borrow because a stack trace has to be generated. The default value is false.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Auto Commit</td>
<td>(boolean) The default auto-commit state of connections created by this pool. If not set, default is JDBC driver default (If not set then the setAutoCommit method will not be called.)</td>
<td></td>
</tr>
<tr>
<td>Default Read Only</td>
<td>(boolean) The default read-only state of connections created by this pool. If not set then the setReadOnly method will not be called. (Some drivers don't support read only mode, ex: Informix)</td>
<td></td>
</tr>
<tr>
<td>Default Catalog</td>
<td>(String) The default catalog of connections created by this pool.</td>
<td></td>
</tr>
<tr>
<td>Validator Class Name</td>
<td>(String) The name of a class which implements the org.apache.tomcat.jdbc.pool.Validator interface and provides a no-arg constructor (may be implicit). If specified, the class will be used to create a Validator instance which is then used instead of any validation query to validate connections. The default value is null. An example value is com.mycompany.project.SimpleValidator.</td>
<td></td>
</tr>
<tr>
<td>Connection Properties</td>
<td>(String) The connection properties that will be sent to our JDBC driver when establishing new connections. Format of the string must be [propertyName=property;]* NOTE - The &quot;user&quot; and &quot;password&quot; properties will be passed explicitly, so they do not need to be included here. The default value is null.</td>
<td></td>
</tr>
<tr>
<td>Init SQL</td>
<td>The ability to run a SQL statement exactly once, when the connection is created.</td>
<td></td>
</tr>
<tr>
<td>JDBC Interceptors</td>
<td>Flexible and pluggable interceptors to create any customizations around the pool, the query execution and the result set handling.</td>
<td></td>
</tr>
<tr>
<td>Validation Interval</td>
<td>(long) avoid excess validation, only run validation at most at this frequency - time in milliseconds. If a connection is due for validation, but has been validated previously within this interval, it will not be validated again. The default value is 30000 (30 seconds).</td>
<td></td>
</tr>
<tr>
<td>JMX Enabled</td>
<td>(boolean) Register the pool with JMX or not. The default value is true.</td>
<td></td>
</tr>
<tr>
<td>Fair Queue</td>
<td>(boolean) Set to true if you wish that calls to getConnection should be treated fairly in a true FIFO fashion. This uses the org.apache.tomcat.jdbc.pool.FairBlockingQueue implementation for the list of the idle connections. The default value is true. This flag is required when you want to use asynchronous connection retrieval. Setting this flag ensures that threads receive connections in the order they arrive. During performance tests, there is a very large difference in how locks and lock waiting is implemented. When fairQueue=true there is a decision making process based on what operating system the system is running. If the system is running on Linux (property os.name=Linux. To disable this Linux specific behavior and still use the fair queue, simply add the property org.apache.tomcat.jdbc.pool.FairBlockingQueue.ignoreOS=true to your system properties before the connection pool classes are loaded.</td>
<td></td>
</tr>
<tr>
<td>Abandon When Percentage Full</td>
<td>(int) Connections that have been abandoned (timed out) wont get closed and reported up unless the number of connections in use are above the percentage defined by abandonWhenPercentageFull. The value should be between 0-100. The default value is 0, which implies that connections are eligible for closure as soon as removeAbandonedTimeout has been reached.</td>
<td></td>
</tr>
<tr>
<td>Max Age</td>
<td>(long) Time in milliseconds to keep this connection. When a connection is returned to the pool, the pool will check to see if the now - time-when-connected &gt; maxAge has been reached, and if so, it closes the connection rather than returning it to the pool. The default value is 0, which implies that connections will be left open and no age check will be done upon returning the connection to the pool.</td>
<td></td>
</tr>
<tr>
<td>Use Equals</td>
<td>(boolean) Set to true if you wish the ProxyConnection class to use String.equals and set to false when you wish to use == when comparing method names. This property does not apply to added interceptors as those are configured individually. The default value is true.</td>
<td></td>
</tr>
</tbody>
</table>
Suspect Timeout | (int) Timeout value in seconds. Default value is 0. Similar to to the removeAbandonedTimeout value but instead of treating the connection as abandoned, and potentially closing the connection, this simply logs the warning if logAbandoned is set to true. If this value is equal or less than 0, no suspect checking will be performed. Suspect checking only takes place if the timeout value is larger than 0 and the connection was not abandoned or if abandon check is disabled. If a connection is suspect a WARN message gets logged and a JMX notification gets sent once.

Alternate User Name Allowed | (boolean) By default, the jdbc-pool will ignore the DataSource.getConnection(username,password) call, and simply return a previously pooled connection under the globally configured properties username and password, for performance reasons.

The pool can however be configured to allow use of different credentials each time a connection is requested. To enable the functionality described in the DataSource.getConnection(username,password) call, simply set the property alternateUsernameAllowed to true. Should you request a connection with the credentials user1/password1 and the connection was previously connected using different user2/password2, the connection will be closed, and reopened with the requested credentials. This way, the pool size is still managed on a global level, and not on a per schema level. The default value is false.

### Configuring a Custom Datasource

This topic explains the purpose of using custom datasources and how you can define custom datasource implementations using the management console. Alternatively, you can simply create datasources using the default RDBMS configuration provided in WSO2 products.

- About custom datasources
- Creating custom datasources

#### About custom datasources

Custom datasources allows you to define your own datasource implementation. There are two options for writing a custom datasource, and these two options cover most of the common business use cases as follows:

- **Custom tabular datasources**: Used to represent data in tables, where a set of named tables contain data rows that can be queried later. A tabular datasource is typically associated with an SQL data services query. This is done by internally using our own SQL parser to execute SQL against the custom datasource. You can use the `org.wso2.carbon.dataservices.core.custom.datasource.TabularDataBasedDS` interface to implement tabular datasources. For a sample implementation of a tabular custom datasource, see `org.wso2.carbon.dataservices.core.custom.datasource.InMemoryDataSource`. Also, this is supported in Carbon datasources with the following datasource reader implementation: `org.wso2.carbon.dataservices.core.custom.datasource.CustomTabularDataSourceReader`.

- **Custom query datasources**: Used when the datasource has some form of query expression support. Custom query datasources are implemented using the `org.wso2.carbon.dataservices.core.custom.datasource.CustomQueryBasedDS` interface. You can create any non-tabular datasource using the query-based approach. Even if the target datasource does not have a query expression format, you can create your own. For example, you can support any NoSQL type datasource this way. For a sample implementation of a query-based custom datasource, see `org.wso2.carbon.dataservices.core.custom.datasource.EchoDataSource`. This is supported in Carbon datasources with the following datasource reader implementation: `org.wso2.carbon.dataservices.core.custom.datasource.CustomQueryDataSourceReader`.

#### Samples

- **InMemoryDSSample** is a sample data service (shipped with DSS by default), which contains both
Creating custom datasources

You can create custom data sources as shown below.

1. Go to the Configure tab on the management console and click Data Sources to open the Data Sources screen.
2. Then click Add Data Source. The following screen will open:

3. Enter "Custom" as the datasource type.
4. In the Custom Data Source Type field, enter "DS_CUSTOM_TABULAR" (to store data in tables) or "DS_CUSTOM_QUERY" (to store non-tabular data accessed through a query).
5. In the Name and Description fields, enter a unique name for the datasource.
6. In the Configuration section, specify the xml configuration of the datasource. See the examples given below.

- XML configuration for a custom tabular datasource (DS_CUSTOM_TABULAR type):
• XML configuration for a custom query datasource (DS_CUSTOM_QUERY):

```xml
<configuration>
  <customDataSourceClass>org.wso2.carbon.dataservices.core.custom.datasource.EchoDataSource</customDataSourceClass>
  <customDataSourceProps>
    <property name="p1">val1</property>
    <property name="p2">val2</property>
  </customDataSourceProps>
</configuration>
```

In the "init" methods of all custom datasources, user-supplied properties will be parsed to initialize the datasource accordingly. Also, a property named "__DATASOURCE_ID__", which contains a UUID to uniquely identify the current datasource, will be passed. This can be used by custom datasource authors to identify the datasources accordingly, such as datasource instances communicating within a server cluster for data synchronization.

After creating datasources, they appear on the Data Sources page. You can edit and delete them as needed by clicking the Edit or Delete links.

**Working with the Registry**
This chapter contains the following information:

- Introduction to Registry
- Managing the Registry
- Searching the Registry
- Setting up the Registry Space

**Introduction to Registry**

A registry is a content store and a metadata repository. Various SOA artifacts such as services, WSDLs and configuration files can be stored in a registry, keyed by unique paths. A path is similar to a Unix file path. In WSO2 products, all configurations pertaining to modules, logging, security, datasources and other service groups are stored in the registry by default.

The Registry kernel of WSO2 products provide the basic registry and repository functionality and contains three major partitions as follows:

- **Local Data Repository** - Used to store settings/metadata specific to the product. This registry is not intended to be shared among multiple servers.
- **Configuration Registry** - Used to store product-specific configurations. These configurations can be shared across multiple instances of the same product like a cluster.
- **Governance Registry** - Used to store user-specified metadata and resources and can be shared across an organization.

These registry instances are mounted to a single top level registry to provide a single, unified view. Mount points of the three registries are /_system/local, /_system/config and /_system/governance respectively. You can browse the contents of the registry used by the product through its management console as shown below.

All WSO2 products use the services provided by the Registry kernel to establish their own registry space, which is utilized for storing data and persisting configurations. Here are some of the features provided by the WSO2 Registry interface:

- Provides the facility to organize resources into collections.
- Maintains multiple versions of resources.
- Manages social aspects such as rating of resources.
- AtomPub interfaces to publish, view and manage resources from remote or non-Java clients.
Managing the Registry

Follow the instructions below to access the registry user interface.

1. Log on to the product’s Management Console and select *Browse* under *Registry*.

2. The *Browse* page appears.

Components of Registry User Interface

- **Breadcrumb** - Shows the current directory hierarchy.
- **Metadata** - Shows metadata for the resource/collection.
- **Properties** - Shows properties for the resource/collection.
- **Entries** - Shows the contents of the resource/collection.
- **Permissions** - Shows the defined role permissions to use the resource/collection.

Managing Breadcrumb

Use the breadcrumb to navigate backward in the current branch of the directory path by clicking on a directory name in the breadcrumb.

1. In the *Browse* window, click the *Tree View* tab to see the branch.
2. Click on a particular directory name to see its details in the *Detail view* tab.

### Managing Metadata

The *Metadata* panel displays the following properties of the resource or the collection:

- **Created** - Shows the time when a resource was created and the author of a resource/collection.
- **Last Updated** - Shows the time when a resource was updated and the author of alterations/collection.
- **Media Type** - An associated media type of the resource/collection.
- **Checkpoint** - Allows to create a checkpoint (URL for the permanent link) of a resource/collection.
- **Versions** - Allows to view versions of a resource/collection.
- **Description** - Description of the resource/collection.

For example,
Checkpoint Creation

1. To create a checkpoint, click on the *Create Checkpoint* link.

2. If the checkpoint was successfully created, a message will be displayed. Click *OK*.

Viewing Versions

1. Click on the *View versions* link.
2. The *Versions* page appears.

- **Version** - Shows the number of a resource/collection version.
- **Last Modified Date** - Shows the last date of updating.
- **Last Modified By** - Shows the author of alterations.
- **Actions**
  - **Details** - Allows to get to the *Browse* page of a particular resource/collection version.
  - **Restore** - Allows to restore a resource/collection version.

**Editing Description**

1. To edit a description of a resource/collection, click on the *Edit* link.
2. Edit the description of a resource/collection in the text area and click **Save**.

### Managing Properties

The *Properties* panel displays the properties of the currently selected resource or collection. New properties can be **added**, while existing properties can be **edited** or **deleted**.

### Adding a Property

1. To add a property, click on the *Add New Property* link.

2. In the *Add New Property* panel, enter a unique name of a property and its value. Click *Add*. 
Editing a Property

1. Click on the *Edit* link of a particular property in the *Action* column.

2. Edit the name and the value of a property in the active fields and click *Save*.

Deleting a Property

1. To delete a property, click on the *Delete* link of a certain property in the *Action* column.

2. Confirm your request by clicking *Yes*.
Managing Entries and Content

If the currently selected entity is a collection, the Content panel is called the Entries panel and shows the child entries under that collection. It provides details of each entry. An entry can be either another collection or a resource. Here you can also add a new resource, add a new collection and create links.

If the currently selected entity is a resource, the Content panel provides a user interface through which one can display, edit, upload, and download the content.

The Entries Panel

The following information is given as shown in the example screenshot below.

- Add Resource
- Add Collection
- Create Link
- Child Resources - The list of child entries provides the following information:
  - Name - The name of a child resource.
  - Created On - The date when a child resource was created.
  - Author - The author who created a child resource.

You can also see the detailed information about the resource by clicking on the Info icon. The following information is available as shown in the example screenshot below.

- Media Type
- Feed
- Rating

To see the available actions over a resource, click on the Actions icon.
The following actions over the resources are available:

- **Rename** - Allows to rename a resource.
- **Move** - Allows to move a resource to a new directory.
- **Delete** - Allows to delete a resource.
- **Copy** - Allows to copy a resource to a specified directory.

**Tip**

All these options are available not for all the resources.

**Renaming a Resource**

1. To rename a resource, click *Rename* and enter a new name to the field.

2. Click on the *Rename* button to save a new name of a resource.

**Moving a Resource**

1. To move a resource to a new directory, click *Move* and specify *Destination Path*.

2. Click *Move*.

**Deleting a Resource**

1. To delete a resource, click *Delete* and confirm your request by clicking *Yes* in the message that appears if the resource is deleted successfully.
Copying a Resource

1. To copy a resource to some directory, click Copy and specify Destination Path.

2. Click Copy.

If the resource was successfully copied, a message appears. Click OK.

Adding a Collection

Follow the instructions below to add a new collection.

1. To add a new collection, click Add Collection.

2. Specify the following options:
   - **Name** - The unique name of a collection.
   - **Media Type** - Select media type of a collection from the drop-down menu:
     - application/vnd.wso2.esb
     - application/vnd.apache.synapse
     - application/vnd.apache.axis2
     - application/vnd.wso2.wsas
     - Other
Description - A brief description of a collection.

3. Click Add.

Adding a Resource

You can add a resource to a certain collection for more convenient usage of the Registry.

Follow the instructions below to add a new child entry to a collection.

1. To add a new resource, click on the Add Resource link.

2. In the Add Resource panel, select Method from the drop-down menu.

The following methods are available:

- Upload content from file
- Import content from URL
- Create Text content
- Create custom content

Uploading Content from File

1. If this method was selected, specify the following options:

   - File - The path of a file to fetch content (XML, WSDL, JAR etc.) Use the Browse button to upload a file.
   - Name - The unique name of a resource.
   - Media type - Can add a configured media type or leave this unspecified to enforce the default media type.
   - Description - The description of a resource.

2. Click Add once the information is added as shown in the example below.
Importing Content from URL

1. If this method was selected, specify the following options:
   - URL - The full URL of the resource to fetch content from URL.
   - Name - The unique name of a resource.
   - Media type - Can add a configured media type or leave this unspecified to enforce the default media type.
   - Description - The description of a resource.

2. Click Add once the information is added.

Text Content Creation

1. If this method was selected, specify the following options:
   - Name - The unique name of a resource.
   - Media type - Can add a configured media type or leave this unspecified to enforce the default media type.
   - Description - The description of a resource.
   - Content - The resource content. You can use either Rich Text Editor or Plain Text Editor to enter.

2. Click Add once the information is added.
Custom Content Creation

1. If this method was selected, choose the Media Type from the drop-down menu and click Create Content.

Media Types

Each collection and resource created and stored on the repository has an associated media type. However, you also have the option to leave this unspecified enforcing the default media type. There are two main ways to configure media types for resources.

- The first method is by means of a one-time configuration, which can be done by modifying the "mime.types" file found in `<PRODUCT_HOME>`\repository\conf\etc directory. This can be done just once before the initial start-up of the server.
- The second method is to configure the media types via the server administration console. The first method does not apply for collections, and the only available mechanism is to configure the media types via the server administration console.

Initially the system contains the media types defined in the mime.types file will be available for resources and a set of default media types will be available for collections.

Managing media types for resources can be done via the server administration console, by editing the properties of the `/system/mime.types/index` collection. This collection contains two resources, collection and custom.ui. To
manage media types of collections and custom user interfaces, edit the properties of these two resources.

Link Creation

Follow the instructions below to create a link on a resource/collection.

1. Symbolic links and Remote links can be created in a similar way to adding a normal resource. To add a link, click Create Link in the Entries panel.

2. Select a link to add from the drop-down menu.

   **A Symbolic Link**

   When adding a Symbolic link, enter a name for the link and the path of an existing resource or collection which is being linked. It creates a link to the particular resource.

   **A Remote Link**

   You can mount a collection in a remotely-deployed registry instance to your registry instance by adding a Remote link. Provide a name for the Remote link in the name field. Choose the instance to which you are going to mount and give the path of the remote collection which you need to mount for the path field, or else the root collection will be mounted.
Managing Role Permissions

The Permissions panel shows the defined role permissions, allows to add new role permissions and edit existing ones.

Adding New Role Permissions

1. In the New Role Permission pane, select a role to set a permission.

2. Select an action from the drop-down menu. The following actions are available:
   - Read
   - Write
   - Delete
   - Authorize
3. Select whether to allow the action the selected role or deny.

4. Click on the Add Permission button.

5. A new permission appears in the Defined Role Permissions list.

Editing Role Permissions

1. You can also edit the defined role permissions using the check boxes in the Defined Role Permissions list.

2. After editing the permissions, click on the Apply All Permissions button to save the alterations.

Searching the Registry

All resources found in the Registry can be searched through the product's Management Console. Search can be refined by resource name, created date range, updated date range, tags, comments, property name, property value, media type etc.
Tip

To search for matches containing a specific pattern, use the "%" symbol.

Follow the instructions below to find a necessary resource in the Registry.

1. Log on to the product's Management Console and select Metadata under Registry.

2. The Search window appears.
The search can be refined by:

- **Resource Name**
- **Created Date Range** - The date when a resource was created.
- **Updated Date Range** - The date when a resource was updated.
- **Update Author** - The author of a resource updating.
- **Create Author** - The author of a resource creation.
- **Tags**
- **Comments**
- **Property Name**
- **Property Value**
- **Media Type**

**Tip**

Created or updated dates can be either entered in the format of MM/DD/YYYY or picked from the calendar interface provided.

3. Fill the search criteria and click on the **Search** button. The results are displayed in the Search Results window.

**Setting up the Registry Space**

WSO2 Carbon is the base platform for all WSO2 products and its Registry kernel provides the basic registry and repository functionality. Products based on Carbon use the services provided by the Registry kernel to establish their own registry spaces utilized for storing data and persisting configuration. Here are some of the features provided by the WSO2 Registry interface:

- Provides the facility to organize resources into collections.
- Maintains multiple versions of resources.
- Manages social aspects such as rating of resources.
- AtomPub interfaces to publish, view and manage resources from remote or non-Java clients.

**Registry partitions**

The Registry space provided to each product contains three major partitions.

- **Local Repository** : Used to store configuration and runtime data that is local to the server. This partition is not to be shared with multiple servers and can be browsed under /_system/local in the registry browser.
- **Configuration Repository** : Used to store product-specific configuration. This partition can be shared across multiple instances of the same product. (eg: sharing ESB configuration across a ESB cluster) and can be
browsed under /_system/config in the registry browser.

- **Governance Repository**: Used to store configuration and data that are shared across the whole platform. This typically includes services, service descriptions, endpoints or datasources and can be browsed under /_system/governance in the registry browser.

### Default setup:

These registry instances are mounted to a single top level registry to provide a single, unified view. Mount points of the three registries are /_system/local, /_system/config and /_system/governance respectively. You can browse the contents of the registry used by the product through its management console.

### Setting up the partitions

Any WSO2 Carbon-based product has the following options when configuring and using a registry space:

- Use the registry space shipped by default with the product.
- Use a remote registry instance/s for the registry partitions that can be shared across multiple WSO2 Carbon-based product instances.

This guide explains the second option using WSO2 Governance Registry as the remote registry instance. Two of the three partitions can be shared across multiple product instances in a typical production environment. Therefore, we can identify four main deployment strategies for the three partitions as follows.

- **All Partitions in a Single Server**
- **Config and Governance Partitions in a Remote Registry**
- **Governance Partition in a Remote Registry**
- **Config and Governance Partitions in Separate Nodes**

In all of the above four sections, any of the WSO2 products can be mounted to a remote WSO2 Governance Registry (G-Reg) instance. Examples discussed here use JDBC based configuration model as that is the recommended approach for a production deployment setup.

### All Partitions in a Single Server

**Strategy 1: Local Registry**

![Pattern #1: Local Registry](image)

Figure 1: All registry partitions in a single server instance.

The entire registry space is local in a single server instance and not shared. This is the default setup in WSO2 products. This strategy is recommended for a stand-alone deployment of a single product instance, but can also be used if there are two or more instances of a product that do not need to share data or configurations among them.
This strategy requires no additional configurations.

**Config and Governance Partitions in a Remote Registry**

In this deployment strategy, the configuration and governance spaces are shared among instances of a group/cluster. For example, two WSO2 Application Server instances that have been configured to operate in a clustered environment can have a single configuration and governance registry which is shared across each node of the cluster. A separate instance of the WSO2 Governance Registry (G-Reg) is used to provide the space used in common.

Figure 2: Config and governance partitions in the remote Governance Registry instance.

Configuration steps are given in the following sections.

- Creating the Database
- Configuring Governance Registry as the Remote Registry Instance
- Configuring Carbon Server Nodes

**Creating the Database**

In a production setup, it is recommended to use an Oracle or MySQL database for the governance registry. As an example, we use MySQL database named ‘registrydb’. Instructions are as follows:

1. Access MySQL using the command:
   
   ```
   mysql -u root -p
   ```

2. Enter the password when prompted.

3. Create ‘registrydb’ database.
   
   ```
   create database registrydb;
   ```

The MySQL database for G-Reg is now created.
Configuring Governance Registry as the Remote Registry Instance

Database configurations are stored in $CARBON_HOME/repository/conf/datasources/master-datasources .xml file for all carbon servers. By default, all WSO2 products use the in-built H2 database. Since Governance Registry in this example is using a MySQL database named 'registrydb', the master-datasources .xml file needs to be configured so that the datasource used for the registry and user manager in Governance Registry is the said MySQL database.


2. Navigate to $G-REG_HOME/repository/conf/master-datasources .xml file where G-REG_HOME is the Governance Registry distribution home. Replace the existing WSO2_CARBON_DB datasource with the following configuration:

```xml
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://x.x.x.x:3306/registrydb</url>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>
```

Change the values of the following elements according to your environment.

- `<url>` : URL of the MySQL database.
- `<username>` and `<password>` : username and password of the mySQL database.
- `<validationQuery>` : Validate and test the health of the DB connection.
- `<validationInterval>` : specified time intervals at which the DB connection validations should run.

3. Navigate to $G-REG_HOME/repository/conf/axis2/axis2.xml file in all Carbon-based product instances to be connected with the remote registry, and enable hazelcast clustering with the following configuration.

```xml
<clustering
  class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent"
  enable="true">
```
The above configuration is required only when caching is enabled for the Carbon server instances and `<enableCache>` parameter is set to true. This provides cache invalidation at the event of any updates on the registry resources.


5. Start the Governance Registry server with `-Dsetup` so that all the required tables are created in the database. For example, in Linux

```
sh wso2server.sh -Dsetup
```

The Governance Registry server is now running with all required user manager and registry tables for the server also created in 'registrydb' database.

**Configuring Carbon Server Nodes**

Now that the shared registry is configured, let's take a look at the configuration of Carbon server nodes that use the shared, remote registry.

1. Download and extract the relevant WSO2 product distribution from the 'Products' menu of [https://wso2.com](https://wso2.com). In this example, we use two server instances (of any product) by the names CARBON-Node1 and CARBON-Node2.

2. We use the same datasource used for Governance Registry above as the registry space of Carbon-based product instances.

**Configure master-datasources.xml File**

3. Configure `$CARBON_HOME/repository/conf/datasource/master-datasources.xml` where `CARBON_HOME` is the distribution home of any WSO2 Carbon-based product you downloaded in step 1. Then, add the following datasource for the registry space.
<datasource>
  <name>WSO2_CARBON_DB_GREG</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB_GREG</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://x.x.x.x:3306/registrydb</url>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>

Change the values of the relevant elements accordingly.

**Configuring registry.xml File**

4. Navigate to $CARBON_HOME/repository/conf/registry.xml file and specify the following configurations for both server instances setup in step 1.

Add a new db config to the datasource configuration done in step 3 above. For example,

```xml
<dbConfig name="remote_registry">
  <dataSource>jdbc/WSO2CarbonDB_GREG</dataSource>
</dbConfig>
```

Specify the remote Governance Registry instance with the following configuration:

```xml
<remoteInstance url="https://x.x.x.x:9443/registry">
  <id>instanceid</id>
  <dbConfig>remote_registry</dbConfig>
  <cacheId>root@jdbc:mysql://x.x.x.x:3306/registrydb</cacheId>
  <readOnly>false</readOnly>
  <enableCache>true</enableCache>
  <registryRoot>/</registryRoot>
</remoteInstance>
```

Change the values of the following elements according to your environment.
Define the registry partitions using the remote Governance Registry instance. In this deployment strategy, we are mounting the config and governance partitions of the Carbon-based product instances to the remote Governance Registry instance. This is graphically represented in Figure 2 at the beginning.

```xml
<mount path="/system/config" overwrite="true">
    <instanceId>instanceid</instanceId>
    <targetPath>/system/nodes</targetPath>
</mount>
<mount path="/system/governance" overwrite="true">
    <instanceId>instanceid</instanceId>
    <targetPath>/system/governance</targetPath>
</mount>
```

- **mount path**: Registry collection of Carbon server instance that needs to be mounted
- **mount overwrite**: Defines if an existing collection/resource at the given path should be overwritten or not. Possible values are:
  - true: The existing collection/resource in the specified location will always be deleted and overwritten with the resource/s in the remote registry instance.
  - false: The resource/s will not be overwritten. An error will be logged if a resource exists at the existing location.
  - virtual: If the existing location has a resource/collection, it will be preserved but virtual view of the remote registry resource/s can be viewed. The original resource/collection can be viewed once the remote registry configuration is removed.
- **target path**: Path to the remote Governance Registry instance where the registry collection is mounted.

**Configuring axis2.xml File**

1. Navigate to `$CARBON_HOME/repository/conf/axis2/axis2.xml` file where `CARBON_HOME` is the distribution home of any WSO2 Carbon-based products to be connected with the remote registry. Enable carbon clustering by copying the following configuration to all Carbon server instances:

```xml
<clustering class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent" enable="true"/>
```

The above configuration is needed only when caching is enabled in the server instances and `<enableCache>` parameter set to true. Clustering enables cache invalidation in configured nodes at the event of any changes to the registry resources by any of the Carbon server nodes in the deployment setup.

2. Copy 'MySQL JDBC connector jar' (http://dev.mysql.com/downloads/connector/j/5.1.html) to `$G-REG_HOME/repository/components/lib` in both Carbon server instances.

3. Start both servers and note the log entries that indicate successful mounting to the remote Governance Registry instance. For example,
4. Navigate to the registry browser in the Carbon server's management console and note the config and governance partitions indicating successful mounting to the remote registry instance. For example,

**Governance Partition in a Remote Registry**

In this deployment strategy, only the governance partition is shared among instances of a group/cluster. For example, a WSO2 Application Server instance and a WSO2 ESB instance that have been configured to operate in a clustered environment can have a single governance registry which is shared across each node of the cluster. A separate instance of the WSO2 Governance Registry (G-Reg) is used to provide the space used in common.
Figure 3: Governance partition in the remote Governance Registry instance.

Configuration steps are given in the following sections.

- Creating the database
- Configuring the Governance Registry instance
- Configuring the Carbon server nodes

Creating the database

In a production setup, it is recommended to use an Oracle or MySQL database for the governance registry. As an example, we use MySQL database named ‘registrydb’. Instructions are as follows:

1. Access MySQL using the command:

   ```
   mysql -u root -p
   ```

2. Enter the password when prompted.
3. Create ‘registrydb’ database.

   ```
   create database registrydb;
   ```

The MySQL database for G-Reg is now created.

Configuring the Governance Registry instance

Database configurations are stored in $PRODUCT_HOME/repository/conf/datasources/master-datasources.xml file for all carbon servers. By default, all WSO2 products use the in-built H2 database. Since Governance Registry in this example is using a MySQL database named ‘registrydb’, the master-datasources.xml file needs to be configured so that the datasource used for the registry and user manager in Governance Registry is the said MySQL database.

2. Navigate to $G-REG_HOME/repository/conf/datasources/master-datasources.xml file where G-REG_HOME is the Governance Registry distribution home. Replace the existing WSO2_CARBON_DB datasource with the following configuration:
Change the values of the following elements according to your environment.

- `<url>`: URL of the MySQL database.
- `<username>` and `<password>`: The username and password of the MySQL database.
- `<validationQuery>`: Validate and test the health of the DB connection.
- `<validationInterval>`: The specified time intervals at which the DB connection validations should run.

3. Navigate to `$PRODUCT_HOME/repository/conf/axis2/axis2.xml` file in all Carbon-based product instances to be connected with the remote registry, and enable clustering with the following configuration.

```xml
<clustering
    class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent" enable="false"/>
```

The above configuration is required only when caching is enabled for the Carbon server instances and `<enableCache>` parameter is set to true. This provides cache invalidation at the event of any updates on the registry resources.

5. Start the Governance Registry server with -Dsetup so that all the required tables are created in the database. For example, in Linux:

```
sh wso2server.sh -Dsetup
```

The Governance Registry server is now running with all required user manager and registry tables for the server
Configuring the Carbon server nodes

Now that the shared registry is configured, let's take a look at the configuration of Carbon server nodes that use the shared, remote registry.

1. Download and extract the relevant WSO2 product distribution from the 'Products' menu of https://wso2.com.
   In this example, we use two server instances (of any product) by the names CARBON-Node1 and CARBON-Node2 and the configuration is given for one server instance. Similar steps apply to the other server instance as well.

2. We use the same datasource used for Governance Registry above as the registry space of Carbon-based product instances.

3. Configure $PRODUCT_HOME/repository/conf/datasource/master-datasources.xml where PRODUCT_HOME is the distribution home of any WSO2 Carbon-based product you downloaded in step 1. Then, add the following datasource for the registry space.

```xml
<datasource>
    <name>WSO2_CARBON_DB_GREG</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/WSO2CarbonDB_GREG</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:mysql://x.x.x.x:3306/registrydb</url>
            <username>root</username>
            <password>root</password>
            <driverClassname>com.mysql.jdbc.Driver</driverClassname>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>
```

Change the values of the relevant elements accordingly.

4. Navigate to $PRODUCT_HOME/repository/conf/registry.xml file and specify the following configurations for both server instances setup in step 1.

Add a new db config to the datasource configuration done in step 3 above. For example:
4. Configuring registry.xml file

```xml
<dbConfig name="remote_registry">
  <dataSource>jdbc/WSO2CarbonDB_GREG</dataSource>
</dbConfig>
```

Specify the remote Governance Registry instance with the following configuration:

```xml
<remoteInstance url="https://x.x.x:9443/registry">
  <id>instanceid</id>
  <dbConfig>remote_registry</dbConfig>
  <cacheId>root@jdbc:mysql://x.x.x.x:3306/registrydb</cacheId>
  <readOnly>false</readOnly>
  <enableCache>true</enableCache>
  <registryRoot>/</registryRoot>
</remoteInstance>
```

Change the values of the following elements according to your environment:

- `<remoteInstance url>`: URL of the remote G-Reg instance.
- `<dbConfig>`: The dbConfig name specified for the registry database configuration.
- `<cacheId>`: This provides information on where the cache resource resides.
- `<enableCache>`: Whether caching is enabled on the Carbon server instance.

In terms of Atom and WS this is mandatory, as this URL is used to connect to the remote instance. In the case of JDBC it is not mandatory but it is required for certain features such as Atom Feeds and Subscriptions available for the repository. Subscriptions for other components of the ESB (for example), are not affected by this parameter.

In the case of a remote mount, the content actually resides remotely and can be updated from multiple destinations. It is important that all of these updates are globally available. Due to this, subscriptions and atom feeds are managed/generated from a single location (i.e., the registry to which you mount to). So, if you need to use these features available in G-Reg/ESB, then you need to have G-Reg up and running and also a proper URL in this field.

In most instances, features such as Atom Feeds and Subscriptions available for the Repository are not required. In such cases, it is best to provide a value like "https://localhost:9443/registry" for this parameter - since the schema validation expects the existence of this parameter though it does not actually need to point to a running instance of WSO2 Governance Registry.

Define the registry partitions using the remote Governance Registry instance. In this deployment strategy, we are mounting the governance partition of the Carbon-based product instances to the remote Governance Registry instance. This is graphically represented in Figure 3 at the beginning of this page.
<mount path="/_system/governance" overwrite="true">
  <instanceId>instanceid</instanceId>
  <targetPath>/_system/governance</targetPath>
</mount>

- mount path : Registry collection of Carbon server instance that needs to be mounted
- mount overwrite : Defines if an existing collection/resource at the given path should be overwritten or not. Possible vales are:
  - true - The existing collection/resource in the specified location will always be deleted and overwritten with the resource/s in the remote registry instance.
  - false - The resource/s will not be overwritten. An error will be logged if a resource exists at the existing location.
  - virtual - If the existing location has a resource/collection, it will be preserved but virtual view of the remote registry resource/s can be viewed. The original resource/collection can be viewed once the remote registry configuration is removed.
- target path : Path to the remote Governance Registry instance where the registry collection is mounted.

5. Navigate to $PRODUCT_HOME/repository/conf/axis2/axis2.xml file where PRODUCT_HOME is the distribution home of any WSO2 Carbon-based products to be connected with the remote registry. Enable carbon clustering by copying the following configuration to all Carbon server instances:

```xml
<clustering
class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent" enable="false">

The above configuration is needed only when caching is enabled in the server instances and <enableCache> parameter set to true. Clustering enables cache invalidation in configured nodes at the event of any changes to the registry resources by any of the Carbon server nodes in the deployment setup.


7. Start both servers and note the log entries that indicate successful mounting to the remote Governance Registry instance. Also navigate to the registry browser in the Carbon server's management console and note the governance partition indicating successful mounting to the remote registry instance.

For large deployments it is recommended to have separate databases for config registry per product. Follow the configurations as below for such deployments.

Create a new datasource in master-datasources.xml.
<datasource>
    <name>WSO2_CARBON_AS_CONFIG</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/as_config</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:mysql://x.x.x.x:3306/registrydb</url>
            <username>root</username>
            <password>root</password>
            <driverClassName>com.mysql.jdbc.Driver</driverClassName>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>

Navigate to $PRODUCT_HOME/repository/conf/registry.xml file and specify the following configurations.

<!-- config db -->
<dbConfig name="config">
    <dataSource>jdbc/as_config</dataSource>
</dbConfig>

<remoteInstance url="https://localhost:9443/registry">
    <id>config</id>
    <dbConfig>config</dbConfig>
    <readOnly>false</readOnly>
    <registryRoot>registry</registryRoot>
    <enableCache>true</enableCache>
    <cacheId root@jdbc:mysql://10.45.67.89:3306/as_config></cacheId>
</remoteInstance>

Config and Governance Partitions in Separate Nodes

In this deployment strategy, let's assume 2 clusters of Carbon-based product Foo and Carbon-based product Bar that share a governance registry space by the name G-Reg 1. In addition, the product Foo cluster shares a configuration registry space by the name G-Reg 2 and the product Bar cluster shares a configuration registry space by the name G-Reg 3.
Figure 4: Config and governance partitions in separate registry instances.

Configuration steps are given in the following sections.

- Creating the Database
- Configuring the Remote Registry Instances
- Configuring the Foo Product Cluster
- Configuring the Bar Product Cluster

Creating the Database

In a production setup, it is recommended to use an Oracle or MySQL database for the governance registry. As an example, we use MySQL database named ‘registrydb’. Instructions are as follows:

1. Access MySQL using the command:

   ```
   mysql -u root -p
   ```

2. Enter the password when prompted.

3. Create 'registrydb' database.

   ```
   create database registrydb;
   ```

The MySQL database for G-Reg 1 is now created. Similarly create ‘registrydb2’ and ‘registrydb3’ as the MySQL databases for G-Reg 2 and G-Reg 3 respectively.

Configuring the Remote Registry Instances

Database configurations are stored in the `<GREG_HOME>/repository/conf/datasources/master-datasources.xml` file for all carbon servers. By default, all WSO2 products use the in-built H2 database. Since the Governance Registry nodes (G-Reg 1, G-Reg 2 and G-Reg 3) in this example are using MySQL databases
('registrydb', 'registrydb2' and 'registrydb3' respectively), the master-datasources.xml file of each node needs to be configured so that the datasources used for the registry, user manager and configuration partitions in Governance Registry are the said MySQL databases.


2. First, navigate to the <GREG_HOME>/repository/conf/datasources/master-datasources.xml file where GREG_HOME is the distribution home of Governance Registry of G-Reg 1. Replace the existing WSO2_CARBON_DB datasource with the following configuration:

```xml
<datasource>
  <name>WSO2_CARBON_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://10.20.30.41:3306/registrydb</url>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>
```

Change the values of the following elements according to your environment.

- `<url>`: URL of the MySQL database.
- `<username>` and `<password>`: Username and password of the mySQL database.
- `<validationQuery>`: Validate and test the health of the DB connection.
- `<validationInterval>`: Specified time intervals at which the DB connection validations should run.

3. Similarly, replace the existing WSO2_CARBON_DB datasource in G-Reg 2 with the following:
4. Repeat the same for G-Reg 3 as follows.

5. Navigate to the `<GREG_HOME>/repository/conf/axis2/axis2.xml` file in all instances and enable clustering with the following configuration.
The above configuration is required only when caching is enabled for the Carbon server instances and the `<enable Cache>` parameter is set to true. This provides cache invalidation at the event of any updates on the registry resources.

6. Copy the 'mySQL JDBC connector jar' ([http://dev.mysql.com/downloads/connector/j/5.1.html](http://dev.mysql.com/downloads/connector/j/5.1.html)) to the `<GREG_HOME>/repository/components/lib` directory in G-Reg 1, G-Reg 2 and G-Reg 3.

7. Start the Governance Registry servers with `-Dsetup` so that all the required tables will be created in the databases. For example, in Linux

   ```bash
   sh wso2server.sh -Dsetup
   ```

   The Governance Registry server instances are now running with all required user manager and registry tables for the server created in 'registrydb', 'registrydb1' and 'registrydb2' databases.

### Configuring the Foo Product Cluster

Now that the shared registry nodes are configured, let's take a look at the configuration of Carbon server clusters that share the remote registry instances. Namely, Foo product cluster shares G-Reg 1 and G-Reg 2 while Bar product cluster shares G-Reg 1 and G-Reg 3.

Include the following configurations in the master node of Foo product cluster.

**Configure master-datasources.xml file**

1. Configure `<PRODUCT_HOME>/repository/conf/datasource/master-datasources.xml` where `<PRODUCT_HOME>` is the distribution home of any WSO2 Carbon-based product. Then, add the following datasource for the registry space.

```xml
<clustering
   class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent"
   enable="true">
```
<datasource>
  <name>WSO2_CARBON_DB_GREG</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB_GREG</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://10.20.30.41:3306/registrydb</url>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>

<datasource>
  <name>WSO2_CARBON_DB_GREG_CONFIG</name>
  <description>The datasource used for configuration partition</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB_GREG_CONFIG</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>

Change the values of the relevant elements according to your environment.

**Configuring registry.xml file**

2. Navigate to the `<GREG_HOME>/repository/conf/registry.xml` file and specify the following configurations.
Add a new db config to the datasource configuration done in step 1 above. For example,

```xml
<dbConfig name="governance_registry">
    <dataSource>jdbc/WSO2CarbonDB_GREG</dataSource>
</dbConfig>
<dbConfig name="config_registry">
    <dataSource>jdbc/WSO2CarbonDB_GREG_CONFIG</dataSource>
</dbConfig>
```

Specify the remote Governance Registry instance with the following configuration:

```xml
<remoteInstance url="https://10.20.30.41:9443/registry">
    <id>governanceRegistryInstance</id>
    <dbConfig>governance_registry</dbConfig>
    <cacheId>root@jdbc:mysql://10.20.30.42:3306/registrydb</cacheId>
    <readOnly>false</readOnly>
    <enableCache>true</enableCache>
    <registryRoot>/</registryRoot>
</remoteInstance>
<remoteInstance url="https://10.20.30.42:9443/registry">
    <id>configRegistryInstance</id>
    <dbConfig>config_registry</dbConfig>
    <cacheId>root@jdbc:mysql://10.20.30.42:3306/registrydb2</cacheId>
    <readOnly>false</readOnly>
    <enableCache>true</enableCache>
    <registryRoot>/</registryRoot>
</remoteInstance>
```

Change the values of the following elements according to your environment.

- `<remoteInstance url>`: URL of the remote G-Reg instance.
- `<dbConfig>`: The dbConfig name specified for the registry database configuration.
- `<cacheId>`: This provides information on where the cache resource resides.
- `<enableCache>`: Whether caching is enabled on the Carbon server instance.

When adding the corresponding configuration to the `registry.xml` file of a slave node, set `<readOnly>true</readOnly>`. This is the only configuration change.

Define the registry partitions using the remote Governance Registry instance.
mount path: Registry collection of Carbon server instance that needs to be mounted
mount overwrite: Defines if an existing collection/resource at the given path should be overwritten or not. Possible values are:
  - true - The existing collection/resource in the specified location will always be deleted and overwritten with the resource/s in the remote registry instance.
  - false - The resource/s will not be overwritten. An error will be logged if a resource exists at the existing location.
  - virtual - If the existing location has a resource/collection, it will be preserved but virtual view of the remote registry resource/s can be viewed. The original resource/collection can be viewed once the remote registry configuration is removed.

target path: Path to the remote Governance Registry instance where the registry collection is mounted.

Configuring axis2.xml file

3. Navigate to the `<PRODUCT_HOME/repository/conf/axis2/axis2.xml>` file and enable clustering by copying the following configuration to all Carbon server instances:

   ```xml
   <clustering class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent" enable="true"/>
   ```

   The above configuration is needed only when caching is enabled in the server instances and when the `<enableCache>` parameter is set to true. Clustering enables cache invalidation in configured nodes at the event of any changes to the registry resources by any of the Carbon server nodes in the deployment setup.


Configuring the Bar Product Cluster

The instructions here are similar to that of the Foo product cluster discussed above. The difference is that Bar product cluster shares G-Reg 1 (Governance space) and G-Reg 3 (Config space) remote registry spaces whereas Foo product cluster shares G-Reg 1 and G-Reg 2 (Config space).

Include the following configurations in the master node of Foo product cluster.

Configure master-datasources. xml file

1. Configure the `<PRODUCT_HOME>/repository/conf/datasource/master-datasources.xml` file where `<PRODUCT_HOME>` is the distribution home of any WSO2 Carbon-based product. Then, add the following datasource for the registry space.
<datasource>
  <name>WSO2_CARBON_DB_GREG</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB_GREG</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://10.20.30.41:3306/registrydb</url>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>

<datasource>
  <name>WSO2_CARBON_DB_GREG_CONFIG</name>
  <description>The datasource used for configuration partition</description>
  <jndiConfig>
    <name>jdbc/WSO2CarbonDB_GREG_CONFIG</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://10.20.30.43:3306/registrydb2</url>
      <username>root</username>
      <password>root</password>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
    </configuration>
  </definition>
</datasource>

Change the values of the relevant elements according to your environment.

**Configuring registry.xml file**

2. Navigate to the `<GREG_HOME>/repository/conf/registry.xml` file and specify the following configurations.
Add a new db config to the datasource configuration done in step 1 above. For example,

```
<dbConfig name="governance_registry">
  <dataSource>jdbc/WSO2CarbonDB_GREG</dataSource>
</dbConfig>
<dbConfig name="config_registry">
  <dataSource>jdbc/WSO2CarbonDB_GREG_CONFIG</dataSource>
</dbConfig>
```

Specify the remote Governance Registry instance with the following configuration:

```
<remoteInstance url="https://10.20.30.41:9443/registry">
  <id>governanceRegistryInstance</id>
  <dbConfig>governance_registry</dbConfig>
  <cacheId>root@jdbc:mysql://10.20.30.42:3306/registrydb</cacheId>
  <readOnly>false</readOnly>
  <enableCache>true</enableCache>
  <registryRoot>/</registryRoot>
</remoteInstance>

<remoteInstance url="https://10.20.30.43:9443/registry">
  <id>configRegistryInstance</id>
  <dbConfig>config_registry</dbConfig>
  <cacheId>root@jdbc:mysql://10.20.30.42:3306/registrydb2</cacheId>
  <readOnly>false</readOnly>
  <enableCache>true</enableCache>
  <registryRoot>/</registryRoot>
</remoteInstance>
```

Change the values of the following elements according to your environment.

- `<remoteInstance url>`: URL of the remote G-Reg instance.
- `<dbConfig>`: The dbConfig name specified for the registry database configuration.
- `<cacheId>`: This provides information on where the cache resource resides.
- `<enableCache>`: Whether caching is enabled on the Carbon server instance.

When adding the corresponding configuration to the registry.xml file of a slave node, set `<readOnly>true</readOnly>`. This is the only configuration change.

Define the registry partitions using the remote Governance Registry instance.
<mount path="/system/config" overwrite="true">
  <instanceId>configRegistryInstance</instanceId>
  <targetPath>/system/nodes</targetPath>
</mount>

<mount path="/system/governance" overwrite="true">
  <instanceId>governanceRegistryInstance</instanceId>
  <targetPath>/system/governance</targetPath>
</mount>

- mount path: Registry collection of Carbon server instance that needs to be mounted
- mount overwrite: Defines if an existing collection/resource at the given path should be overwritten or not.
  Possible values are:
  - true - The existing collection/resource in the specified location will always be deleted and overwritten with the resource/s in the remote registry instance.
  - false - The resource/s will not be overwritten. An error will be logged if a resource exists at the existing location.
  - virtual - If the existing location has a resource/collection, it will be preserved but virtual view of the remote registry resource/s can be viewed. The original resource/collection can be viewed once the remote registry configuration is removed.
- target path: Path to the remote Governance Registry instance where the registry collection is mounted.

**Configuring axis2.xml file**

3. Navigate to the `<PRODUCT_HOME>/repository/conf/axis2/axis2.xml` file and enable clustering by copying the following configuration to all Carbon server instances:

```
<clustering
  class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent"
  enable="false"/>
```

The above configuration is needed only when caching is enabled in the server instances and when the `<enableCache>` parameter is set to true. Clustering enables cache invalidation in configured nodes at the event of any changes to the registry resources by any of the Carbon server nodes in the deployment setup.


5. Start both clusters and note the log entries that indicate successful mounting to the remote Governance Registry nodes.

6. Navigate to the registry browser in the Carbon server's management console of a selected node and note the config and governance partitions indicating successful mounting to the remote registry instances.

For large deployments it is recommended to have separate databases for config registry per product. Follow the configurations as below for such deployments.

Create a new datasource in the `master-datasources.xml` file.
<datasource>
    <name>WSO2_CARBON_AS_CONFIG</name>
    <description>The datasource used for registry and user manager</description>
    <jndiConfig>
        <name>jdbc/as_config</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <url>jdbc:mysql://x.x.x.x:3306/registrydb</url>
            <username>root</username>
            <password>root</password>
            <driverClassName>com.mysql.jdbc.Driver</driverClassName>
            <maxActive>50</maxActive>
            <maxWait>60000</maxWait>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
        </configuration>
    </definition>
</datasource>

Navigate to the <PRODUCT_HOME>/repository/conf/registry.xml file and specify the following configurations.

<!-- config db -->
<dbConfig name="config">
    <dataSource>jdbc/as_config</dataSource>
</dbConfig>

<remoteInstance url="https://localhost:9443/registry">
    <id>config</id>
    <dbConfig>config</dbConfig>
    <readOnly>false</readOnly>
    <registryRoot>/</registryRoot>
    <enableCache>true</enableCache>
    <cacheId>root@jdbc:mysql://10.45.67.89:3306/as_config</cacheId>
</remoteInstance>

**Using WS-Discovery in Governance Registry**

WSO2 Governance Registry has the capability to discover the services via ws-discovery and store the discovered services and related metadata. If you have set up WSO2 Governance Registry as a remote governance registry for your application, you have the option of configuring your server to use the governance registry as a WS-Discovery proxy. Any WS-Discovery compliant web services hosting server (E.g.: Axis2/JAX-WS/JAX-RS) can be configured to use Governance Registry as a WS-Discovery proxy, which would discover any deployed services for governing purposes such as LifeCycle Management.

The following instructions explain how to discover JAX-WS/JAX-RS web services in WSO2 Application Server using
WS-Discovery in the WSO2 Governance Registry:

1. Add the following property to the axis2.xml file of WSO2 AS to configure WSO2 Governance Registry (GReg) to work as a WS-Discovery proxy. In this example, the value of the parameter is the endpoint of the "DiscoveryProxy" service, which is deployed in Governance Registry to discover services.

   ```xml
   <parameter
    name="DiscoveryProxy">https://localhost:9443/services/DiscoveryProxy</parameter>
   ```

2. If required, you can define scopes and the metadata version as a <context-param> in the WEB-INF/web.xml. If the scopes and metadata version are not defined, the default version will be effective.
   - The following configuration adds the http://wso2.com/as scope:

   ```xml
   <context-param>
    <param-name>Scopes</param-name>
    <param-value>http://wso2.com/as</param-value>
   </context-param>
   ```

   - The following configuration adds the metadata version:

   ```xml
   <context-param>
    <param-name>MetadataVersion</param-name>
    <param-value>100</param-value>
   </context-param>
   ```

3. Start the two products (WSO2 Governance Registry and WSO2 Application Server) following the instructions in Running the Product.

   Note that you must add a port offset in the <PRODUCT_HOME>/repository/conf/carbon.xml file to ensure that both products are not using the same port.

4. Log in to the management console of WSO2 AS and add a JAX-WS service (E.g. java_first_jaxws.war). The service would be displayed on the service list.
5. Now, log in to the management console of WSO2 Governance Registry and go to the Services menu. The service list will display the services that were deployed in Application Server.

   The name of this service will be shown as the web application name followed by the name of service available in the web application. For example, if java_first_jaxws.war has a service called hello_world, then the service will be listed as java_first_jaxws_hello_world.

This means that Governance Registry has discovered all the web services deployed in Application Server and stored them in Governance Registry. Now these service endpoints can be used from Governance Registry for various purposes.

**Working with Transports**
This chapter contains the following information:

- Introduction to Transports
- AS Transports
- Configuring Transports Globally

Introduction to Transports

WSO2 Carbon is the base platform on which all WSO2 Java products are developed. Built on OSGi, WSO2 Carbon encapsulates all major SOA functionality. It supports a variety of transports which make Carbon-based products capable of receiving and sending messages over a multitude of transport and application-level protocols. This functionality is implemented mainly in the Carbon core, which combines a set of transport-specific components to load, enable, manage and persist transport related functionality and configurations.

All transports currently supported by WSO2 Carbon are directly or indirectly based on the Apache Axis2 transports framework. This framework provides two main interfaces for each transport implementation.

- `org.apache.axis2.transport.TransportListener` - Implementations of this interface should specify how incoming messages are received and processed before handing them over to the Axis2 engine for further processing.
- `org.apache.axis2.transport.TransportSender` - Implementations of this interface should specify how a message can be sent out from the Axis2 engine.

Each transport implementation generally contains a transport receiver/listener and a transport sender, since they use the interfaces above. The Axis2 transport framework enables the user to configure, enable and manage transport listeners and senders independent to each other, without having to restart the server. For example, one may enable only the JMS transport sender without having to enable JMS transport listener.

The transport management capability of WSO2 Carbon is provided by the following feature in the WSO2 feature repository:

**Name:** WSO2 Carbon - Transport Management Feature

**Identifier:** org.wso2.carbon.transport.feature

If transport management capability is not included in your product by default, you can add it by installing the above feature using the instructions given in section Feature Management.

AS Transports

WSO2 AS supports a variety of transports which makes WSO2 AS capable of receiving and sending messages over a multitude of transport and application level protocols.

The following transport implementations are supported:

- HTTP Servlet Transport
- HTTPS Servlet Transport
- MailTo Transport
- JMS Transport
- TCP Transport
- Local Transport
- UDP Transport

The links discuss these transport implementations in detail, highlighting the configuration parameters associated with them. For more information on configuring transports, see Working with Transports.

HTTP Servlet Transport
The transport receiver implementation of the HTTP transport is available in the Carbon core component. The transport sender implementation comes from the Tomcat http connector. This transport is shipped with WSO2 Carbon and all WSO2 Carbon-based products that use this transport as the default transport, except WSO2 ESB. By default, we use non-blocking Tomcat Java connector, `org.apache.coyote.http11.Http11NioProtocol`.

- This is a non-blocking HTTP transport implementation, which means that I/O the threads does not get blocked while received messages are processed.
- Although the `axis2.xml` file contains configurations for HTTP/S transports by default, they are not used by WSO2 products. Instead, the products use the HTTP/S transport configurations in Tomcat-level; therefore, changing the HTTP/S configurations in the `axis2.xml` file has no effect.

In the transport parameter tables, the literals displayed in italics under the "Possible Values" column should be considered as fixed literal constant values. Those values can be directly put into the transport configurations.

HTTP servlet transport should be configured in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file by configuring the connector parameters given below. The transport class that should be specified in the `catalina-server.xml` file is `<Connector protocol="org.apache.coyote.http11.Http11NioProtocol"/>

### Transport Connector Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>The port number on which this transport receiver should listen for incoming messages.</td>
<td>Yes</td>
<td>A positive integer less than 65535</td>
<td></td>
</tr>
<tr>
<td>proxyPort</td>
<td>When used, this transport listener will accept messages arriving through a HTTP proxy server which listens on the specified proxy port. Apache mod_proxy should be enabled in the proxy server. All the WSDLs generated will contain the proxy port value as the listener port.</td>
<td>No</td>
<td>A positive integer less than 65535</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>redirectPort</td>
<td>If this Connector is supporting non-SSL requests, and a request is received for which a matching <code>&lt;security-constraint&gt;</code> requires SSL transport, Catalina will automatically redirect the request to the port number specified here.</td>
<td>A positive integer less than 65535</td>
<td>9443</td>
<td></td>
</tr>
<tr>
<td>bindOnInit</td>
<td>Controls when the socket used by the connector is bound. By default it is bound when the connector is initiated and unbound when the connector is destroyed. If set to false, the socket will be bound when the connector is started and unbound when it is stopped.</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proxyPort</td>
<td>When used, this transport listener will accept messages arriving through a HTTP proxy server which listens on the specified proxy port. Apache mod_proxy should be enabled on the proxy server. All the WSDLs generated will contain the proxy port value as the listener port.</td>
<td>A positive integer less than 65535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>maxHttpHeaderSize</td>
<td>The maximum size of the HTTP request and response header in bytes.</td>
<td>A positive integer</td>
<td>4096</td>
<td></td>
</tr>
<tr>
<td>acceptorThreadCount</td>
<td>The number of threads to be used to accept connections. Increase this value on a multi CPU machine, although you would never really need more than 2. Also, with a lot of non keep alive connections, you might want to increase this value as well.</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>maxThreads</td>
<td>The maximum number of worker threads created by the receiver to handle incoming requests. This parameter largely determines the number of concurrent connections that can be handled by the transport.</td>
<td>A positive integer</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>minSpareThreads</td>
<td>The minimum number of threads always kept running. If not specified, the default will be used.</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>enableLookups</td>
<td>Use this parameter to enable DNS lookups in order to return the actual host name of the remote client. Disabling DNS lookups at transport level generally improves performance. By default, DNS lookups are disabled.</td>
<td>true, false</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>disableUploadTimeout</td>
<td>This flag allows the servlet container to use a different, longer connection timeout while a servlet is being executed, which in the end allows either the servlet a longer amount of time to complete its execution, or a longer timeout during data upload.</td>
<td>true, false</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>connectionUploadTimeout</td>
<td>Specifies the timeout, in milliseconds, to use while a data upload is in progress. This only takes effect if disableUploadTimeout is set to false.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Values</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------------------------</td>
<td>--------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>clientAuth</td>
<td>Set to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to want if you want the SSL stack to request a client Certificate, but not fail if one is not present. A false value (which is the default) will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.</td>
<td>true, false, want</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>maxKeepAliveRequests</td>
<td>The maximum number of HTTP requests which can be pipelined until the connection is closed by the server. Setting this attribute to 1 will disable HTTP/1.0 keep-alive, as well as HTTP/1.1 keep-alive and pipelining. Setting this to -1 will allow an unlimited amount of pipelined or keep-alive HTTP requests.</td>
<td>-1 or any positive integer</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>acceptCount</td>
<td>The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused.</td>
<td>A positive integer</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>Overrides the Server header for the http response. If set, the value for this attribute overrides the Tomcat default and any Server header set by a web application. If not set, any value specified by the application is used.</td>
<td>Any string</td>
<td>WSO2 Carbon Server</td>
<td></td>
</tr>
<tr>
<td>compression</td>
<td>The Connector may use HTTP/1.1 GZIP compression in an attempt to save server bandwidth. The acceptable values for the parameter is &quot;off&quot; (disable compression), &quot;on&quot; (allow compression, which causes text data to be compressed), &quot;force&quot; (forces compression in all cases), or a numerical integer value (which is equivalent to &quot;on&quot;, but specifies the minimum amount of data before the output is compressed). If the content-length is not known and compression is set to &quot;on&quot; or more aggressive, the output will also be compressed. If not specified, this attribute is set to &quot;off&quot;.</td>
<td>on, off, force</td>
<td>on</td>
<td></td>
</tr>
<tr>
<td>compressionMinSize</td>
<td>If compression is set to &quot;on&quot; then this attribute may be used to specify the minimum amount of data before the output is compressed.</td>
<td>A positive integer</td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>noCompressionUserAgents</td>
<td>Indicate a list of regular expressions matching user-agents of HTTP clients for which compression should not be used, because these clients, although they do advertise support for the feature, have a broken implementation.</td>
<td>A comma separated list of regular expressions</td>
<td>empty string</td>
<td></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
<td>Required</td>
<td>Possible Values</td>
<td>Default Value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------</td>
<td>-----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>compressableMimeType</td>
<td>Use this parameter to indicate a list of MIME types for which HTTP compression may be used.</td>
<td>A comma separated list of valid mime types</td>
<td>text/html, text/xml, text/plain</td>
<td></td>
</tr>
<tr>
<td>URIEncoding</td>
<td>This specifies the character encoding used to decode the URI bytes, after %xx decoding the URL.</td>
<td>URI encoding Character set name</td>
<td>ISO-8859-1</td>
<td></td>
</tr>
</tbody>
</table>

This servlet transport implementation can be further tuned up using the following parameters.

This is only a subset of all the supported parameters. The servlet HTTP transport uses the `org.apache.catalina.connector.Connector` implementation from Apache Tomcat. So the servlet HTTP transport actually accepts any parameter accepted by the connector implementation. For a complete list of supported parameters, see Apache Tomcat's connector configuration reference.

**Transport Sender Parameters**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROTOCOL</td>
<td>The version of HTTP protocol to be used for outgoing messages.</td>
<td>No</td>
<td>HTTP/1.0, HTTP/1.1</td>
<td>HTTP/1.1</td>
</tr>
<tr>
<td>Transfer-Encoding</td>
<td>Effective only when the HTTP version is 1.1 (i.e. the value of the PROTOCOL parameter should be HTTP/1.1). Use this parameter to enable chunking support for the transport sender.</td>
<td>No</td>
<td>chunked</td>
<td>Not Chunked</td>
</tr>
<tr>
<td>SocketTimeout</td>
<td>The socket timeout value in milliseconds, for out bound connections.</td>
<td>No</td>
<td>A positive integer</td>
<td>60000 ms</td>
</tr>
<tr>
<td>ConnectionTimeout</td>
<td>The connection timeout value in milliseconds, for out bound connections.</td>
<td>No</td>
<td>A positive integer</td>
<td>60000 ms</td>
</tr>
<tr>
<td>OmitSOAP12Action</td>
<td>Set this parameter to &quot;true&quot; if you need to disable the soapaction for SOAP 1.2 messages.</td>
<td>No</td>
<td>true, false</td>
<td>false</td>
</tr>
</tbody>
</table>

**Defining multiple tomcat connectors**

You have the option of defining multiple tomcat connectors in the `catalina-server.xml` file. Note that when you define multiple connectors, all the endpoints of the applications deployed in your WSO2 server will still be exposed through all the connector ports. However, you can configure your load balancer to ensure that only the relevant applications are exposed through the required connector port.

Therefore, you can use multiple connectors to strictly separate the applications deployed in your server as explained below.

1. See the example given below where two connectors are defined in the `catalina-server.xml` file.
2. Configure your load balancer so that the relevant applications are exposed through the required connector port.

**HTTPS Servlet Transport**

Similar to the HTTP transport, the HTTPS transport consists of a receiver implementation which comes from the Carbon core component and a sender implementation which comes from the Tomcat http connector. In fact, this transport uses exactly the same transport sender implementation as the HTTP transport. The transport receiver implementation of the HTTP transport is available in the Carbon core component. The transport sender implementation comes from the Tomcat http connector. This transport is shipped with WSO2 Carbon and all WSO2 Carbon-based products, which use this transport as the default transport, except WSO2 ESB. By default, we use non-blocking Tomcat Java connector, org.apache.coyote.http11.Http11NioProtocol.

Although the axis2.xml file contains configurations for HTTP/S transports by default, they are not used by WSO2 products. Instead, the products use the HTTP/S transport configurations in Tomcat-level; therefore, changing the HTTP/S configurations in the axis2.xml file has no effect.

HTTPS servlet transport should be configured in the <PRODUCT_HOME>/repository/conf/tomcat/catalina -server.xml file, same as with the HTTP servlet transport. The transport class that should be specified in the catalina-server.xml file is as follows: <Connector protocol="org.apache.coyote.http11.Http11NioProtocol"/>

**Transport connector parameters**

In addition to the configuration parameters supported by the HTTP servlet transport, HTTPS servlet transport supports the following configuration parameters:

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sslProtocol</td>
<td>Transport level security protocol to be used.</td>
<td>TLS, SSL</td>
<td>TLS</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Examples</td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>enableLookups</td>
<td>Set to true if you want calls to request.getRemoteHost() to perform DNS lookups in order to return the actual host name of the remote client. Set to false to skip the DNS lookup and return the IP address in String form instead (thereby improving performance).</td>
<td>true, false</td>
<td></td>
</tr>
<tr>
<td>clientAuth</td>
<td>Set to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to false if you want the SSL stack to request a client Certificate, but not fail if one isn't presented. A false value (which is the default) will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.</td>
<td>true, false</td>
<td></td>
</tr>
<tr>
<td>scheme</td>
<td>Set this attribute to the name of the protocol you wish to have returned by calls to request.getScheme().</td>
<td>http, https</td>
<td></td>
</tr>
<tr>
<td>secure</td>
<td>Set this attribute to true if you wish to have calls to request.isSecure() to return true for requests received by this Connector. You would want this on an SSL Connector or a non SSL connector that is receiving data from a SSL accelerator, like a crypto card, a SSL appliance or even a webserver.</td>
<td>true, false</td>
<td></td>
</tr>
<tr>
<td>SSLEnabled</td>
<td>Use this attribute to enable SSL traffic on a connector. To turn on SSL handshake/encryption/decryption on a connector set this value to true. When turning this value true you will want to set the scheme and the secure attributes as well to pass the correct request.getScheme() and request.isSecure() values to the servlets. For more information see, SSL Support.</td>
<td>true, false</td>
<td></td>
</tr>
<tr>
<td>keystoreFile</td>
<td>Path to the keystore which should be used for encryption/decryption.</td>
<td>A valid file path to a keystore file</td>
<td></td>
</tr>
<tr>
<td>keypass</td>
<td>Password to access the specified keystore.</td>
<td>A valid password</td>
<td></td>
</tr>
</tbody>
</table>

Similar to the servlet HTTP transport, this transport is also based on Apache Tomcat's connector implementation. For a complete list of supported parameters, see Tomcat connector configuration reference.

**Defining multiple tomcat connectors**

You have the option of defining multiple HTTPS connectors in the catalina-server.xml file in the same way as for HTTP connectors. See HTTP servlet transport for more information.

**MailTo Transport**

The polling MailTo transport supports sending messages (E-Mail) over SMTP and receiving messages over POP3 or IMAP. This transport implementation is available as a module of the WS-Commons Transports project. The receiver and sender classes that should be included in the Carbon configuration to enable the MailTo transport are org.apache.axis2.transport.mail.MailTransportListener and org.apache.axis2.transport.mail.MailTransportSender respectively. The JAR consisting of the transport implementation is named axis2-transport-mail.jar.

The mail transport receiver should be configured at service level. That is each service configuration should explicitly state the mail transport receiver configuration. This is required to enable different services to receive mails over.
different mail accounts and configurations. However, transport sender is generally configured globally so that all services can share the same transport sender configuration.

Service Level Transport Receiver Parameters

The MailTo transport listener implementation can be configured by setting the parameters described in JavaMail API documentation. For IMAP related properties, see Package Summary - IMAP. For POP3 properties, see Package Summary - POP3. In addition to the parameters described in the JavaMail API documentation, the MailTo transport listener also supports the following transport parameters.

Tip

In the following transport parameter tables, the literals displayed in italics under the Possible Values column should be considered as fixed literal constant values. Those values can be directly specified in the transport configuration.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport.mail.Address</td>
<td>The mail address from which this service should fetch incoming mails.</td>
<td>Yes</td>
<td>A valid e-mail address</td>
<td></td>
</tr>
<tr>
<td>transport.mail.Folder</td>
<td>The mail folder in the server from which the listener should fetch incoming mails.</td>
<td>No</td>
<td>A valid mail folder name (e.g., inbox)</td>
<td>inbox folder if that is available or else the root folder</td>
</tr>
<tr>
<td>transport.mail.Protocol</td>
<td>The mail protocol to be used to receive messages.</td>
<td>No</td>
<td>pop3, imap</td>
<td>imap</td>
</tr>
<tr>
<td>transport.mail.PreserveHeaders</td>
<td>A comma separated list of mail header names that this receiver should preserve in all incoming messages.</td>
<td>No</td>
<td>A comma separated list</td>
<td></td>
</tr>
<tr>
<td>transport.mail.RemoveHeaders</td>
<td>A comma separated list of mail header names that this receiver should remove from incoming messages.</td>
<td>No</td>
<td>A comma separated list</td>
<td></td>
</tr>
<tr>
<td>transport.mail.ActionAfterProcess</td>
<td>Action to perform on the mails after processing them.</td>
<td>No</td>
<td>MOVE, DELETE</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
Global Transport Sender Parameters

For a list of parameters supported by the MailTo transport sender, see Package Summary - SMTP. In addition to the parameters described there, the MailTo transport sender supports the following parameters.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport.mail.SMTPBccAddresses</td>
<td>If one or more e-mail addresses need to be specified as BCC addresses for outgoing mails, this parameter can be used.</td>
<td>No</td>
<td>A comma separated list of e-mail addresses</td>
<td></td>
</tr>
<tr>
<td>transport.mail.Format</td>
<td>Format of the outgoing mail.</td>
<td>No</td>
<td>Text, Multipart</td>
<td>Text</td>
</tr>
</tbody>
</table>
JMS Transport

JMS (Java Message Service) transport implementation also comes from the WS-Commons Transports project. All the relevant classes are packed into the axis2-transport-jms-<version>.jar and the following classes act as the transport receiver and the sender respectively.

- org.apache.axis2.transport.jms.JMSListener
- org.apache.axis2.transport.jms.JMSSender

The JMS transport implementation requires an active JMS server instance to be able to receive and send messages. We recommend using Apache ActiveMQ JMS server, but other implementations such as Apache Qpid and Tibco are also supported. You also need to put the client JARs for your JMS server in Carbon classpath. In case of Apache ActiveMQ, you need to put the following JARs in the classpath:

- activemq-core.jar
- geronimo-j2ee-management_1.0_spec-1.0.jar
- geronimo-jms_1.1_spec-1.1.1.jar

These JAR files can be obtained by downloading the latest version of Apache ActiveMQ (version 5.2.0 is recommended). Extract the downloaded archive and find the required dependencies in the $ACTIVEMQ_HOME/lib directory. You need to copy these JAR files over to $CARBON_HOME/repository/components/lib directory for Carbon to be able to pick them up at runtime.

Configuration parameters for JMS receiver and the sender are XML fragments that represent JMS connection factories. A typical JMS parameter configuration would look like this:

```xml
<parameter name="myTopicConnectionFactory">
  <parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
  <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
  <parameter name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
  <parameter name="transport.jms.ConnectionFactoryType">topic</parameter>
</parameter>
```

This is a bare minimal JMS connection factory configuration, which consists of four connection factory parameters. JMS connection factory parameters are described in detail below.

### JMS Connection Factory Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
</tr>
</thead>
</table>

In transport parameter tables, literals displayed in italic mode under the “Possible Values” column should be considered as fixed literal constant values. Those values can be directly put in transport configurations.
<table>
<thead>
<tr>
<th>Configuration Key</th>
<th>Description</th>
<th>Required</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.naming.factory.initial</td>
<td>JNDI initial context factory class. The class must implement the <code>java.naming.spi.InitialContextFactory</code> interface.</td>
<td>Yes</td>
<td>A v</td>
</tr>
<tr>
<td>java.naming.provider.url</td>
<td>URL of the JNDI provider.</td>
<td>Yes</td>
<td>A v</td>
</tr>
<tr>
<td>java.naming.security.principal</td>
<td>JNDI Username.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>java.naming.security.credentials</td>
<td>JNDI password.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>transport.Transactionality</td>
<td>Desired mode of transactionality.</td>
<td>No</td>
<td>nai</td>
</tr>
<tr>
<td>transport.UserTxnJNDIName</td>
<td>JNDI name to be used to require user transaction.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>transport.CacheUserTxn</td>
<td>Whether caching for user transactions should be enabled or not.</td>
<td>No</td>
<td>true</td>
</tr>
<tr>
<td>transport.jms.SessionTransacted</td>
<td>Whether the JMS session should be transacted or not.</td>
<td>No</td>
<td>true</td>
</tr>
<tr>
<td>transport.jms.SessionAcknowledgement</td>
<td>JMS session acknowledgment mode.</td>
<td>No</td>
<td>AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE, SESSION_TRANSACTED</td>
</tr>
<tr>
<td>transport.jms.ConnectionFactoryJNDIName</td>
<td>The JNDI name of the connection factory.</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>transport.jms.ConnectionFactoryType</td>
<td>Type of the connection factory.</td>
<td>No</td>
<td>queue</td>
</tr>
<tr>
<td>transport.jms.JMSSpecVersion</td>
<td>JMS API version.</td>
<td>No</td>
<td>1.1</td>
</tr>
<tr>
<td>transport.jms.UserName</td>
<td>The JMS connection username.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>transport.jms.Password</td>
<td>The JMS connection password.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>transport.jms.Destination</td>
<td>The JNDI name of the destination.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>transport.jms.DestinationType</td>
<td>Type of the destination.</td>
<td>No</td>
<td>queue</td>
</tr>
<tr>
<td>transport.jms.DefaultReplyDestination</td>
<td>JNDI name of the default reply destination.</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>transport.jms.DefaultReplyDestinationType</td>
<td>Type of the reply destination.</td>
<td>No</td>
<td>queue</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Default</td>
<td>Configuration Level</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td>transport.jms.MessageSelector</td>
<td>Message selector implementation.</td>
<td>No</td>
<td>Service Level</td>
</tr>
<tr>
<td>transport.jms.SubscriptionDurable</td>
<td>Whether the connection factory is subscription durable or not.</td>
<td>No</td>
<td>True</td>
</tr>
<tr>
<td>transport.jms.DurableSubscriberName</td>
<td>Name of the durable subscriber.</td>
<td>Yes if the subscription durable is turned on</td>
<td></td>
</tr>
<tr>
<td>transport.jms.PubSubNoLocal</td>
<td>Whether the messages should be published by the same connection they were received.</td>
<td>No</td>
<td>True</td>
</tr>
<tr>
<td>transport.jms.CacheLevel</td>
<td>JMS resource cache level.</td>
<td>No</td>
<td>None, Connection, Session, Consumer, Producer, Auto</td>
</tr>
<tr>
<td>transport.jms.ReceiveTimeout</td>
<td>Time to wait for a JMS message during polling. Set this parameter value to a negative integer to wait indefinitely. Set to zero to prevent waiting.</td>
<td>No</td>
<td>Number of milliseconds to wait</td>
</tr>
<tr>
<td>transport.jms.ConcurrentConsumers</td>
<td>Number of concurrent threads to be started to consume messages when polling.</td>
<td>No</td>
<td>Any Positive Integer - For topics this must always be 1</td>
</tr>
<tr>
<td>transport.jms.MaxConcurrentConsumers</td>
<td>Maximum number of concurrent threads to use during polling.</td>
<td>No</td>
<td>Any Positive Integer</td>
</tr>
<tr>
<td>transport.jms.IdleTaskLimit</td>
<td>The number of idle runs per thread before it dies out.</td>
<td>No</td>
<td>Any</td>
</tr>
<tr>
<td>transport.jms.MaxMessagesPerTask</td>
<td>The maximum number of successful message receipts per thread.</td>
<td>No</td>
<td>Any to infinity</td>
</tr>
<tr>
<td>transport.jms.InitialReconnectDuration</td>
<td>Initial reconnection attempts duration in milliseconds.</td>
<td>No</td>
<td>Any</td>
</tr>
<tr>
<td>transport.jms.ReconnectProgressFactor</td>
<td>Factor by which the reconnection duration will be increased.</td>
<td>No</td>
<td>Any</td>
</tr>
<tr>
<td>transport.jms.MaxReconnectDuration</td>
<td>Maximum reconnection duration in milliseconds.</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

JMS transport implementation has some parameters that should be configured at service level, in other words in service XML files of individual services.

**Service Level JMS Configuration Parameters**
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport.jms.ConnectionFactory</td>
<td>Name of the JMS connection factory the service should use.</td>
<td>No</td>
<td>A name of an already defined connection factory</td>
<td>default</td>
</tr>
<tr>
<td>transport.jms.PublishEPR</td>
<td>JMS EPR to be published in the WSDL.</td>
<td>No</td>
<td>A JMS EPR</td>
<td></td>
</tr>
</tbody>
</table>

TCP Transport

The TCP transport is implemented in the Apache WS-Commons Transport project. All the classes required for the TCP transport are packed into the bundle named `axis2-transport-tcp-<version>`. The following classes act as the transport receiver and sender respectively:

- `org.apache.axis2.transport.tcp.TCPServer`
- `org.apache.axis2.transport.tcp.TCPTransportSender`

The following topics explain the process of installing the TCP transport in WSO2 AS:

- **Step 1: Deploying the required bundles**
- **Step 2: Enabling the TCP transport in the product configurations**
- **Step 3: Restart the server**

**Step 1: Deploying the required bundles**

In order to use this transport, the `axis2-transport-tcp-<version>` OSGi bundle should be added to the server. This can be done in two ways:

- Install the TCP transport feature that is available in the WSO2 feature repository. See the instructions for installing features.

Note that there is a separate feature repository for each Carbon release. You will find all the repositories listed here. Therefore, when you install a feature for your product version, make sure to use the feature version that is relevant to your product version.

- Alternatively, you can manually download the `axis2-transport-tcp-<version>` bundle from here and add it to the `<AS_HOME>/repository/components/dropins` directory.

**Step 2: Enabling the TCP transport in the product configurations**

To enable the TCP transport, open the `<AS_HOME>/repository/conf/axis2.xml` file and add the following transport receiver/sender configurations:
Given below are the parameters that can be used in the axis2.xml file when you configure the 'transport receiver'. Note that the TCP transport sender does not accept any configuration parameters as of now.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Required</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport.tcp.port</td>
<td>The port on which the TCP server should listen for incoming messages.</td>
<td>No</td>
<td>A positive integer less than 65535.</td>
<td>8000</td>
</tr>
<tr>
<td>transport.tcp.hostname</td>
<td>The host name of the server to be displayed in WSDLs etc.</td>
<td>No</td>
<td>A valid host name or an IP address.</td>
<td></td>
</tr>
</tbody>
</table>

**Step 3: Restart the server**

Restart the server after completing the above steps to enable the TCP transport.

**Local Transport**

Apache Axis2's local transport implementation is used to make internal service calls and transfer data within the Axis2 instance. The following class implements the sender API:

- `org.apache.axis2.transport.local.LocalTransportSender`

The transport does not have a receiver implementation as of now.

It provides an opportunity for fast in-VM service call.

To use this transport, configure an endpoints with the `local://` prefix. For example, to make an in-VM call to the HelloService, use `local://HelloService`.

Configuring a Local Transport with WSO2 Products

Shown below is how to configure a local transport with any WSO2 Carbon-based product.

1. In the carbon.xml file at location `<PRODUCT_HOME>/repository/conf`, an endpoint is available as follows by default.

   `<ServerURL>local://${carbon.context}/services/</ServerURL>`

2. In the axis2.xml file at location `<PRODUCT_HOME>/repository/conf/axis2`, there is a transport sender named...
'local' specified as follows:

```xml
<transportSender name="local"
class="org.apache.axis2.transport.local.LocalTransportSender"/>
```

It has to be replaced with the following sender/receiver pair.

```xml
<transportReceiver name="local"
class="org.wso2.carbon.core.transports.local.CarbonLocalTransportReceiver"/>
<transportSender name="local"
class="org.wso2.carbon.core.transports.local.CarbonLocalTransportSender"/>
```

For more information about transports, refer to Working with Transports.

**UDP Transport**

The UDP transport implementation is in the Apache WS-Commons Transports project. The following classes implement the Axis2 transport listener and sender APIs respectively.

- `org.apache.axis2.transport.udp.UDPListener`
- `org.apache.axis2.transport.udp.UDPSender`

The axis2-transport-udp.jar archive file contains the above implementation classes.

To enable the UDP transport for samples, simple open the file repository/conf/axis2.xml in a text editor and add the following transport configurations. UDP transport component is shipped with the WSO2 ESB by default.

```xml
<transportReceiver name="udp"/>
<transportSender name="udp"/>
```

If you wish to use the sample Axis2 client to send UDP messages, you have to uncomment the UDP transport sender configuration in the samples/axis2Client/client_repo/conf/axis2.xml file.

**Configuring Transports Globally**

Transports can be configured and enabled in a global-level or a service-level. Globally enabled and configured transports effect all services deployed in a running Carbon instance. The following methods can be used to configure and enable transports globally:

- Using `axis2.xml` file
- Using `catalina-server.xml` file

**Using axis2.xml file**

WSO2 Carbon and all Carbon-based products are shipped with a configuration file named `axis2.xml`. This XML configuration file can be found at `<CARBON_HOME>/repository/conf/axis2` directory. This is similar to the `axis2.xml` file that ships with Apache Axis2 and Apache Synapse. It contains the global configuration for WSO2
Carbon and the Carbon-based products. The axis2.xml configuration generally includes configuration details for modules, phases, handlers, global configuration parameters and transports. The elements <transportReceiver> and <transportSender> are used to configure transport listeners and senders respectively. In the axis2.xml file that comes with WSO2 Carbon or any Carbon-based product, some transports are already configured and enabled by default, including the HTTP and HTTPS transports.

WSO2 products do not use the HTTP/S servlet transport configurations that are in the axis2.xml file. Instead, they use Tomcat-level servlet transports, which are used by the management console in <PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml file.

Given below is an example JMS transport receiver configuration in the axis2.xml file.

```xml
<transportReceiver name="jms"
  class="org.apache.axis2.transport.jms.JMSListener">
  <parameter name="myTopicConnectionFactory">
    <parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
    <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
    <parameter name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</parameter>
  </parameter>

  <parameter name="myQueueConnectionFactory">
    <parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
    <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
    <parameter name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
  </parameter>

  <parameter name="default">
    <parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
    <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
    <parameter name="transport.jms.ConnectionFactoryJNDIName">QueueConnectionFactory</parameter>
  </parameter>
</transportReceiver>
```

The <transportReceiver> element has the following attributes and sub-elements:
**name** - A mandatory attribute that indicates a unique name for the transport receiver.

**class** - A mandatory attribute that indicates the transport receiver implementation class.

**parameters** - Configuration parameters for the transport receiver. It should be included as child elements of the `<transportReceiver>` element.

Similarly, use the `<transportSender>` element to configure and enable transport senders in WSO2 products.

- The `axis2.xml` file will be loaded to memory only during server startup. Therefore, any changes made to the file while the server is up and running will not be applied until the server is restarted.
- Simply having `<transportReceiver>` and `<transportSender>` elements in the `axis2.xml` file causes those transports to be loaded and activated during server startup. Therefore, any dependency JARs required by those transport implementations must be included in the server classpath to prevent the server from running into exceptions at startup. In addition, an inaccurate transport configuration (for example, a wrong parameter value) might result in the transport not being enabled properly.

### Using `catalina-server.xml` file

In addition to the above, transport receivers can be configured globally using the `<CARBON_HOME>/repository/conf/tomcat/catalina-server.xml` file. The default HTTP/S configuration specified in the `catalina-server.xml` file is given below:

```xml
bindOnInit="false"
maxHttpHeaderSize="8192"
acceptorThreadCount="2"
maxThreads="250"
minSpareThreads="50"
disableUploadTimeout="false"
connectionUploadTimeout="120000"
maxKeepAliveRequests="200"
acceptCount="200"
server="WSO2 Carbon Server"
compression="on"
compressionMinSize="2048"
noCompressionUserAgents="gozilla, traviata"
compressableMimeType="text/html,text/javascript,application/x-javascript,application/javascript,application/xml,text/css,application/xslt+xml,text/xsl,image/gif,image/jpg,image/jpeg" URIEncoding="UTF-8"/>

bindOnInit="false"
sslProtocol="TLS"
maxHttpHeaderSize="8192"/>```
acceptorThreadCount="2"
maxThreads="250"
minSpareThreads="50"
disableUploadTimeout="false"
enableLookups="false"
connectionUploadTimeout="120000"
maxKeepAliveRequests="200"
acceptCount="200"
server="WSO2 Carbon Server"
clientAuth="false"
compression="on"
scheme="https"
secure="true"
SSLEnabled="true"
compressionMinSize="2048"
noCompressionUserAgents="gozilla, traviata"

compressableMimeType="text/html,text/javascript,application/x-javascript,application/javascript,application/xml,text/css,application/xslt+xml,text/xsl,image/gif,image/jpg,image/jpeg"
keystoreFile="${carbon.home}/repository/resources/security/wso2carbon.jks"
keystorePass="wso2carbon"
URIEncoding="UTF-8"/>

Currently only the default servlet transports of Carbon can be configured using the catalina-server.xml file.

For more details on config parameters, go to http://tomcat.apache.org/tomcat-7.0-doc/config/http.html

Working with Security

This section explains how you can set up the security configurations in your application server.

After you install WSO2 AS, it is recommended to change the default security settings according to the requirements of your production environment. Since AS is built on top of the WSO2 Carbon Kernel (version 4.4.1), the main security configurations applicable to AS are inherited from the Carbon kernel.

You can find detailed information on how to configure security in your AS as explained below.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring transport-level security</td>
<td>WSO2 products support a variety of transports that make them capable of receiving and sending messages over a multitude of transport and application-level protocols. By default, most of the WSO2 products come with the HTTP transport. The transport receiver implementation of the HTTP transport is available in Carbon. The transport sender implementation comes from the Tomcat HTTP connector, which is configured in the <code>&lt;AS_HOME&gt;/repository/conf/tomcat/catalina-server.xml</code> file. This also covers how you can protect your system from common security attacks. For example, 'Poodle' is a bug in the SSL version 3 protocol, which exposes critical data encrypted between clients and servers. 'Logjam' is a security threat (man-in-the-middle attack), which is caused by weak ciphers. These security vulnerabilities can be avoided by configuring transport-level security. For more information on securing the HTTP transport, see the topic on configuring transport level security in the WSO2 Carbon documentation.</td>
</tr>
</tbody>
</table>
| Configuring keystores       | A keystore is a repository that stores the cryptographic keys and certificates. These artifacts are used for encrypting sensitive information, and establishing trust between your server and outside parties that connect to your server. All WSO2 products come with a default keystore (wso2carbon.jks). In a production environment, it is recommended to replace it with a new keystore. You can also configure multiple keystores for different purposes. See the following in the WSO2 Carbon documentation:  
  * Learn how public key encryption and keystores are used.  
  * Learn how to create new keystores and replace the default one.  
  * Learn how configuration files should be updated to use the relevant keystore for different purposes. |
Securing sensitive passwords

As a secure vault implementation is available in all WSO2 products, you can encrypt the sensitive data such as passwords in configuration files using the Cipher tool.

See the following in the WSO2 Carbon documentation:

- How the secure vault is implemented in WSO2 products.
- How to encrypt passwords using the Cipher tool.
- How to resolve encrypted passwords.

Enabling JAVA security manager

See the topic on enabling JAVA security manager in the WSO2 Carbon documentation for details on how to prevent untrusted code from manipulating your system.

---

Working with Multiple Tenants

The goal of multitenancy is to maximize resource sharing by allowing multiple users (tenants) to log in and use a single server/cluster at the same time, in a tenant-isolated manner. That is, each user is given the experience of using his/her own server, rather than a shared environment. Multitenancy ensures optimal performance of the system's resources such as memory and hardware and also secures each tenant's personal data.

You can register tenant domains using the Management Console of WSO2 products.

This section covers the following topics:

- Managing Tenants
- Multitenant Architecture
- Tenant-aware Load Balancing Using the WSO2 Elastic Load Balancer

Managing Tenants

You can add a new tenant in the management console and then view it by following the procedure below. In order to add a new tenant, you should be logged in as a super user.

1. Click Add New Tenant in the Configure tab of your product's management console.

2. Enter the tenant information in Register A New Organization screen as follows, and click Save.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>The domain name for the organization, which should be unique (e.g., abc.com). This is used as a unique identifier for your domain. You can use it to log into the admin console to be redirected to your specific tenant. The domain is also used in URLs to distinguish one tenant from another.</td>
</tr>
</tbody>
</table>
### Select Usage Plan for Tenant

The usage plan defines limitations (such as number of users, bandwidth etc.) for the tenant.

### First Name/Last Name

The name of the tenant admin.

### Admin Username

The login username of the tenant admin. The username always ends with the domain name (e.g., admin@abc.com)

### Admin Password

The password used to log in using the admin username specified.

### Admin Password (Repeat)

Repeat the password to confirm.

### Email

The email address of the admin.

3. After saving, the newly added tenant appears in the **Tenants List** page as shown below. Click **View Tenants** in the **Configure** tab of the management console to see information of all the tenants that currently exist in the system. If you want to view only tenants of a specific domain, enter the domain name in the **Enter the Tenant Domain** parameter and click **Find**.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Email</th>
<th>Created Date</th>
<th>Active</th>
<th>Edit</th>
</tr>
</thead>
<tbody>
<tr>
<td>wso2.com</td>
<td><a href="mailto:frankie.avalone@gmail.com">frankie.avalone@gmail.com</a></td>
<td>2014/11/17 12:03:06</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>abc.com</td>
<td><a href="mailto:dean.martin@gmail.com">dean.martin@gmail.com</a></td>
<td>2014/11/17 13:43:48</td>
<td>✔️</td>
<td></td>
</tr>
</tbody>
</table>

### Multitenant Architecture

The multi tenant architecture of WSO2 products allows you to deploy Web applications, Web services, ESB mediators, mashups etc. in an environment that supports the following:

- **Tenant isolation**: Each tenant has its own domain, which the other tenants cannot access.
- **Data isolation**: Each tenant can manage its data securely in an isolated manner.
- **Execution isolation**: Each tenant can carry out business processes and workflows independent of the other tenants. No action of a tenant is triggered or inhibited by another tenant.
- **Performance Isolation**: No tenant has an impact on the performance of another tenant.

#### Architecture

A tenant is an isolated domain. The users within this domain can manage their own data and perform their own transactions without being affected by actions carried out in other domains.

These domains are allocated server space from the complete server space of a WSO2 product instance which is referred to as the **super tenant**.

The super tenant as well as each individual tenant has its own configuration and context module.

Each tenant has its own security domain. A domain has a set of users, and permissions for those users to access resources. Thus, a tenant is restricted by the users and permissions of the domain assigned to it. The artifact
repositories of the tenants are separated from each other.

An individual tenant can carry out the following activities within the boundaries of its own configuration and context module:

- Deploying artifacts
- Applying security
- User management
- Data management
- Request throttling
- Response caching

WSO2 Carbon provides a number of Admin services which have special privileges to manage the server. These admin services are deployed in the super tenant. Other tenants can make use of these admin services to manage their deployment. The admin services operate in a tenant aware fashion. Thus, privileges and restrictions that apply to any client using an admin service are taken into account.

**Resource sharing**

WSO2 Carbon supports the following methods for sharing resources among tenants:

- **Private Jet mode**: This method allows the load of a tenant ID to be deployed in a single tenant mode. A single tenant is allocated an entire service cluster. The purpose of this approach is to allow special privileges (such as priority processing and improved performance) to a tenant.
- **Separation at hardware level**: This method allows different tenants to share a common set of resources, but each tenant has to run its own operating system. This approach helps to achieve a high level of isolation, but it also incurs a high overhead cost.
- **Separation at JVM level**: This method allows tenants to share the same operating system. This is done by enabling each tenant to run a separate JVM instance in the operating system.
- **Native multitenancy**: This method involves allowing all the tenants to share a single JVM instance. This method minimises the overhead cost.

**Lazy loading**

Lazy loading is a design pattern used specifically in cloud deployments to prolong the initialization of an object or artifact until it is requested by a tenant or an internal process.

**Tenants**
Lazy loading of tenants is a feature that is built into all WSO2 products. This feature ensures that all the tenants are not loaded at the time the server starts in an environment with multiple tenants. Instead, they are loaded only when a request is made to a particular tenant. If a tenant is not utilized for a certain period of time (30 minutes by default), it will be unloaded from the memory.

You can change the default time period allowed for tenant inactiveness by adding `-Dtenant.idle.time=<time_in_minutes>` java property to the startup scrip of the product (`/wso2server.sh` file for Linux and `wso2server.bat` for Windows) as shown below.

```
JAVA_OPTS 
-Dtenant.idle.time=30 
```

### Artifacts

Lazy loading of artifacts is a feature that is used by some WSO2 products, which can be enabled in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file. The deployer that handles lazy loading of artifacts is called the GhostDeployer. A flag to enable or disable the Ghost Deployer is shown below. This is set to `false` by default because the Ghost Deployer works only with the HTTP/S transports. Therefore, if other transports are used, the Ghost Deployer does not have to be enabled.

```
<GhostDeployment>
  <Enabled>false</Enabled>
  <PartialUpdate>false</PartialUpdate>
</GhostDeployment>
```

When a stand-alone WSO2 product instance is started with lazy loading enabled, its services, applications and other artifacts are not deployed immediately. They are first loaded in the Ghost form and the actual artifact is deployed only when a request for the artifact is made. In addition, if an artifact has not been utilized for a certain period of time, it will be unloaded from the memory.

When lazy loading of artifacts is enabled for PaaS deployments, lazy loading applies both for tenants as well as a tenant artifacts. As a result, lazy loading is applicable on both levels for a tenant in a cloud environment. Therefore, the associated performance improvements and resource utilization efficiencies are optimal.

### Restrictions

The following restrictions are imposed to ensure that each individual tenant has the required level of isolation and maintains fine grained security control over its own services without affecting the other tenants.

- Only the super tenant can modify its own configuration. In addition, it can add, view and delete tenants.
- When a tenant logs into the system, it can only access artifacts deployed under its own configuration. One tenant cannot manipulate the code of another tenant.
- The super admin or tenant admin can add user stores to their own domain. Dynamic configurations are possible only for secondary user stores and the primary user store is not configurable at run time. This is because primary user stores are available for all tenants and allowing changes to the configuration at run time can lead to instability of the system. Therefore, the primary user store is treated as a static property in the implementation and it should be configured prior to run time.
- A tenants code cannot invoke sensitive server side functionality. This is achieved via Java security.
- Tenants share the transports provided by the system. They are not allowed to create their own transports.

### Request dispatching

This section describes how the multi tenancy architecture described above works in a request dispatching scenario.
When a Carbon server receives a request, the message is first received by the handlers and dispatchers defined for the server configuration (i.e. super tenant). The server configuration may include handlers that implement cross tenant policies and Service Level Agreement (SLA) management. For example, a priority based dispatcher can be applied at this stage to offer differentiated qualities of service to different clients. Once the relevant handlers and dispatchers are applied, the request is sent to the tenant to which it is addressed. Then the message dispatchers and handlers specific to that tenant will be applied.

The following example further illustrates how message dispatching is carried out in a multi tenant server.

For example, two tenants named foo.com and bar.com may deploy a service named MyService. When this service is hosted on the two tenants, they would have the following URLs.

http://example.com/t/foo.com/services/MyService
http://example.com/t/bar.com/services/MyService

The name of the tenant in the URL allows the tenant to be identified when the Carbon server receives a message which is addressed to a specific client. Alternatively, you may configure a CNAME record in DNS (Domain Name System) as an alias for this information.

If a request is addressed to the MyService service hosted by foo.com, the message handlers and dispatchers of the super tenant will be applied and the tenant foo.com will be identified by the tenant name in the URL. Then the request will be sent to foo.com where it will be processed.

**Scaling**

The multi tenancy architecture described above mainly refers to a scenario where a single instance of a Carbon server acts as a single multi tenant node. In a situation where a very high load of requests are handled, you may need multiple multi tenant nodes. In order to operate with multiple multi tenant nodes, you need load balancing. The load balancer you use also needs to be tenant-aware.

**Tenant-aware Load Balancing Using the WSO2 Elastic Load Balancer**

WSO2 Elastic Load Balancer is currently retired.

Tenant partitioning is required in a clustered deployment to be able to scale to large numbers of tenants. There can be multiple clusters for a single service and each cluster would have a subset of tenants as illustrated in the diagram below. In such situations, the load balancers need to be tenant aware in order to route the requests to the required tenant clusters. They also need to be service aware since it is the service clusters which are partitioned according to the clients.

The following example further illustrates how this is achieved in WSO2 Elastic Load Balancer (ELB).
A request sent to a load balancer has the following host header to identify the cluster domain:

https://appserver.cloud-test.wso2.com/carbon.as1.domain/carbon/admin/login.jsp

In this URL:

- **appserver.cloud-test.wso2.com** is the service domain which allows the load balancer to identify the service.
- **carbon.as1.domain.com** is the tenant domain which allows the load balancer to identify the tenant.

Services are configured with their cluster domains and tenant ranges in the `loadbalancer.conf` file. These cluster domains and tenant ranges are picked by the load balancer when it loads.

The following is a sample configuration of the loadbalancer.conf file.
In the above configuration, there is a host address which maps to the application server service. If required, you can enter multiple host addresses separated by commas.

There are two cluster domains defined in the configuration. The cluster domain named `carbon.as1.domain` is used to load the range of tenants with IDs 1-100. The other cluster domain named `carbon.as2.domain` is used to load the tenants with IDs 101-200.

If the tenant ID of `abc.com` is 22, the request will be directed to the `Carbon.AS1.domain` cluster.

**Working with Logs**

Logging is one of the most important aspects of a production-grade server. A properly configured logging system is vital for identifying errors, security threats and usage patterns. All WSO2 products are shipped with the log4j logging capabilities, which generates administrative activities and server side logs. The `log4j.properties` file, which governs how logging is performed by the server can be found in the `<PRODUCT_HOME>/repository/conf` directory.

Note that in addition to the logs from libraries that use Log4j, all logs from libraries (such as, Tomcat, Hazelcast and more) that use Java logging framework are also visible in the same log files. That is, when Java logging is enabled in Carbon, only the Log4j appenders will write to the log files. If the Java Logging Handlers have logs, these logs will be delegated to the log events of the corresponding Log4j appenders. A Pub/Sub registry pattern implementation has been used in the latter mentioned scenario to plug the handlers and appenders. The following default log4j appenders in the `log4j.properties` file are used for this implementation:

- `org.wso2.carbon.logging.appenders.CarbonConsoleAppender`
- `org.wso2.carbon.logging.appenders.CarbonDailyRollingFileAppender`

For more information on appenders, loggers, their log levels and logging, see [http://logging.apache.org/log4j](http://logging.apache.org/log4j).

Logging functionality is provided by the following feature in the WSO2 feature repository:

**Name** : WSO2 Carbon - Logging Management Feature  
**Identifier** : org.wso2.carbon.logging.mgt.feature.group

If the above feature is not bundled in your product by default, you can install it using the instructions given in the Working with Features section.

**Configuring products for logging**
Given below are ways to configure log4j files:

- Manually editing the log4j.properties file.
- Configure logging through the management console. These changes will apply at run time.

The latter is recommended because all changes made to log4j through the management console persist in the WSO2 Registry. Therefore, those changes will be available after the server restarts. Changes that you make to the logging configuration via the management console will get priority over what is defined in the actual log4j.properties file. However, if you modify the log4j.properties file and restart the server, the earlier log4j configuration that persisted in the registry will be overwritten. There is also an option in the management console to restore the original log4j configuration from the log4j.properties file.

**Managing log growth**

Log growth can be managed by the following configurations in the `<PRODUCT_HOME>/repository/conf/log4j.properties` file.

- Configurable log rotation: By default, log rotation is on a daily basis.
- Log rotation based on time as opposed to size: This helps to inspect the events that occurred during a specific time.
- Log files are archived to maximise the use of space.

The log4j-based logging mechanism uses appenders to append all the log messages into a file. That is, at the end of the log rotation period, a new file will be created with the appended logs and archived. The name of the archived log file will always contain the date on which the file is archived.

**Viewing logs**

In each product, users can configure and adjust the logging levels for each type of activity/transactions. There are several ways to view system and application logs of a running Carbon instance.

- Through the Management Console. See Monitoring Logs for details.
- Through the log files that are stored in the `<PRODUCT_HOME>/repository/logs` folder. This folder contains current logs in a log file with a date stamp. Older logs are archived in the `wso2carbon.log` file.
- Through the command prompt/shell terminal that opens when you run the "wso2server.bat"/"wso2server.sh" files to start the Carbon server.

See the following topics for information about logging:

- Configure Logging using the Management Console
- HTTP Access Logging
- Configure Logging using Config Files

**Configure Logging using the Management Console**

In most systems, logging properties should be specified before the server starts and cannot be changed while it is running. However, logging properties of a running Carbon instance can be changed through its management console, while the server is up and running.

There are three important components when configuring log4j. They are Loggers, Appenders, and Layouts. The Logging Management feature of WSO2, allows you to change these parameters globally as well as individually at
run time. When you change the parameters using the Management Console, first the server stores new values in
the database and then changes the appropriate components in the logging framework, enabling the logging
properties to be updated immediately.

To configure logging properties of the system and application logs of a Carbon server at run time:

1. Log on to the management console of your product and go to Configure -> Logging in the navigator.
2. The Logging Configuration window appears.

3. If you select the Persist All Configurations Changes check box, all the modifications will persist and they
will be available even after the server restarts.

4. The Logging Configuration window consists of three sections, which you can use to configure the layout
and the amount of information about the system activity that you want to receive. The following topics
describes each of these settings:

- Global Log4J Configuration
- Configure Log4J Appenders
- Configure Log4J Loggers

Global Log4J Configuration
This section allows you to assign a single log level and log pattern to all loggers.

- **Log Level** - Severity of the message. Reflects a minimum level that the logger requires. You can view the hierarchy of levels.
- **Log Pattern** - Defines the output format of the log file. This is the layout pattern which describes the log message format

If you click **Restore Defaults**, the Registry will be overwritten by logging configurations specified in the `log4j.properties` file.

### Configure Log4J Appenders

This section allows you to configure appenders individually. Log4j allows logging requests to print to multiple destinations. These output destinations are called 'Appenders'. You can attach several appenders to one logger.

- **Name** - The name of an appender. By default, a WSO2 product server is entered in this field with the following log appenders configured:
  - **CARBON_CONSOLE** - Logs to the console when the server is running.
  - **CARBON_LOGFILE** - Writes the logs to `<PRODUCT_HOME>/repository/logs/wso2carbon.log`.
  - **CARBON_MEMORY**
  - **CARBON_SYS_LOG** - Allow to separate the software that generates messages, from the system that stores them and the software that reports and analyzes them.
  - **CARBON_TRACE_LOGFILE**

- **Log pattern** - Defines the output format of the log file.
- **Sys Log Host** - The IP address of the system log server. The syslog server is a dedicated log server for many applications. It runs in a particular TCP port in a separate machine, which can be identified by an IP address.
- **Facility** - The log message type sent to the system log server.
- **Threshold** - Filters log entries based on their level. For example, threshold set to 'WARN' will allow the log entry to pass into appender. If its level is 'WARN', 'ERROR' or 'FATAL', other entries will be discarded. This is the minimum log level at which you can log a message.

The available categories of logs you can view:

- **TRACE** - Designates fine-grained informational events than DEBUG.
- **DEBUG** - Designates fine-grained informational events that are most useful to debug an application.
- **INFO** - Designates informational messages that highlight the progress of the application at coarse-grained level.
- **WARN** - Designates potentially harmful situations.
- **ERROR** - Designates error events that might still allow the application to continue running.
- **FATAL** - Designates very severe error events that will presumably lead the application to abort.

**Configure Log4J Loggers**

A Logger is an object used to log messages for a specific system or application component. Loggers are normally named using a hierarchical dot-separated namespace and have a 'child-parent' relationship. For example, the logger named 'root.sv' is a parent of the logger named 'root.sv.sf' and a child of 'root'.

When the server starts for the first time, all the loggers initially listed in the `log4j.properties` file appear on the logger name list. This section allows you to browse through all these loggers, define a log level and switch on/off additivity to any of them. After editing, the logging properties are read only from the database.

- **Logger** - The name of a logger.
- **Parent Logger** - The name of a parent logger.
- **Level** - Allows to select level (threshold) from the drop-down menu. After you specify the level for a certain logger, a log request for that logger will only be enabled if its level is equal or higher to the logger’s level. If a given logger is not assigned a level, then it inherits one from its closest ancestor with an assigned level. Refer to hierarchy of levels above.
- **Additivity** - Allows to inherit all the appenders of the parent Logger if set as ‘True’.

In this section, loggers can be filtered by the first characters (use the Starts With button) or by a combination of characters (use the Contains button).

**HTTP Access Logging**

HTTP Requests/Responses are logged in the access log(s) and are helpful to monitor your application's usage activities, such as the persons who access it, how many hits it receives, what the errors are etc. This information is useful for troubleshooting. As the runtime of WSO2 products are based on Apache Tomcat, you can use the `Access_Log_Valve` variable in Tomcat 7 as explained below to configure HTTP access logs in WSO2 products. In addition, you can customize the access logs based on the supported Access Log Valve attributes.

**Configuring HTTP access logging**

To configure HTTP access logs:
1. Edit the Access_Log_Valve variable as follows in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file, which is the server descriptor file for the embedded Tomcat integration.

```xml
<Valve className="org.apache.catalina.valves.AccessLogValve"
directory="${carbon.home}/repository/logs"
prefix="localhost_access_log_sample."
suffix=".log"
pattern="%{xxx}i %{xxx}o"
resolveHosts="false"/>
```

The pattern attribute defines the formatting layout, which consists of the information fields from the requests and responses that should be logged. The description of the formatting layout specified in the above pattern attribute is given below. For more example configurations, see Customizing access logs by pattern.

- `xxx` - header name we need to log
- `%{xxx}i` - incoming request headers
- `%{xxx}o` - outgoing response headers

For more information on all possible values that can be given in the pattern parameter, go to: http://tomcat.apache.org/tomcat-7.0-doc/config/valve.html#Access_Log_Valve

If you do not wish to customize the formatting layout, you can simply use the word 'common' or 'combined' for the pattern attribute to select a standard format.

- common - %h %l %u %t "%r" %s %b
- combined - %h %l %u %t "%r" %s %b "%{Referer}i" "%{User-Agent}i"

The optimized access only supports 'common' and 'combined' as the value for this attribute.

2. Restart the server. This will create a log file named `localhost_access_log_sample.{DATE}.log` inside the `<PRODUCT_HOME>/repository/logs` directory.

**Customizing access logs by pattern**

Given below are a few sample configurations, for ways in which the access logs can be customized by the pattern attribute:

- **Example 1: Logging request headers**
- **Example 2: Logging response headers**
- **Example 3: Logging other variable values**
- **Example 4: Logging URL encoded parameters**

**Example 1: Logging request headers**

The configuration is as follows:
This sample configuration logs the Content-type, Accept and Accept-encoding headers of every request coming to the server. For example, in the following example, we use the RequestInfoExample to send the HTTP request:

```
GET http://<IP>:<PORT>/example/servlets/servlet/RequestInfoExample?abc=xyz
```

It records the following log entry in the `localhost_access_log_sample.{DATE}.log` file.

```
text/plain; charset=utf-8        */*        gzip, deflate, sdch
```

**Example 2: Logging response headers**

The configuration is as follows:

```
<Vale className="org.apache.catalina.valves.AccessLogValve"
directory="${carbon.home}/repository/logs"
prefix="localhost_access_log_test."
suffix=".log"
pattern="%{Content-Type}o %{Content-Length}o %{Date}o %{Server}o"
resolveHosts="false"/>
```

The above configuration sample logs the Content-type, Content-Length, Date and Server headers of every response coming from the server as follows:

```
text/html;charset=ISO-8859-1       662       Tue, 09 Jul 2013 11:21:50 GMT
WSO2 Carbon
```

**Example 3: Logging other variable values**

The configuration is as follows:
<Valve className="org.apache.catalina.valves.AccessLogValve"
directory="${carbon.home}/repository/logs"
prefix="localhost_access_log_test."
suffix=".log"
pattern="%r %q %h"
resolveHosts="false"/>

The above sample configuration logs the first line of the request (method and request URI), query string (prepended with a '?' if it exists) and remote hostname (or IP) of every request coming to the server as follows.

```
"GET /example/servlets/servlet/RequestInfoExample?abc=xyz HTTP/1.1"
?abc=xyz     10.100.0.67
```

Example 4: Logging URL encoded parameters

You cannot use the AccessLogValve to log URL encoded parameters. However, you can use the ExtendedAccessLogValve attribute for this purpose. In this example only two values (namely, className and pattern) are modified from the previous configuration.

The configuration is as follows:

<Valve className="org.apache.catalina.valves.ExtendedAccessLogValve"
directory="${carbon.home}/repository/logs"
prefix="localhost_access_log_extended."
suffix=".log"
pattern="x-P(param1) x-P(param2)"
resolveHosts="false"/>

Send the POST request together with the URL encoded values such as param1=value1 and param2=value2 as follows:

```
POST http://<IP>:<PORT>/example/servlets/servlet/RequestInfoExample
```

The above sample configuration logs the following:

```
'value1'     'value2'
```

Configure Logging using Config Files

All WSO2 products are shipped with the log4j logging capabilities, which generates administrative activities and server side logs. There are two ways of configuring how logging works in your server: by using the management console or by manually updating the configuration files.

Given below are some instructions on how to configure logging using configuration files in WSO2 AS.
• Updating the log4j.properties file
• Updating the log levels for JavaEE

**Updating the log4j.properties file**

The `log4j.properties` file, which governs how logging is performed by the server can be found in the `<PRODUCT_HOME>/repository/conf` directory. All the logging configurations that you enable using the management console can be manually configured in this configuration file.

**Updating the log levels for JavaEE**

From AS 5.3.0 onwards, you can enable JavaEE as a runtime in your server. Thereby, if you want to change the log levels of web apps deployed in the JavaEE runtime, you need to configure the relevant log levels in the `<AS_HOME>/repository/conf/etc/logging-bridge.properties` file. The default levels given in this file are as follows:

```
OpenEJB.level = WARNING
OpenEJB.options.level = WARNING
OpenEJB.server.level = WARNING
OpenEJB.startup.level = WARNING
OpenEJB.startup.service.level = WARNING
OpenEJB.startup.config.level = WARNING
OpenEJB.hsql.level = INFO
OpenEJB.rs.level = INFO
OpenEJB.ws.level = INFO
OpenEJB.tomcat.level = INFO
CORBA-Adapter.level = WARNING
Transaction.level = WARNING
org.apache.activemq.level = SEVERE
org.apache.geronimo.level = SEVERE
openjpa.level = WARNING
OpenEJB.cdi.level = WARNING
org.apache.webbeans.level = WARNING
org.apache.openejb.level = WARNING
```

**Tuning Performance**

This section describes some recommended performance tuning configurations to optimize WSO2 AS. It assumes that you have set up WSO2 AS on a server running Unix/Linux, which is recommended for a production deployment.

• **OS-Level Settings**
• **JVM Settings**
• **JDBC Pool Configuration**

**Important**

• Performance tuning requires you to modify important system files, which affect all programs running on the server. We recommend you to familiarize yourself with these files using Unix/Linux documentation before editing them.
• The parameter values we discuss below are just examples. They might not be the optimal values for the specific hardware configurations in your environment. We recommend that you carry out load tests on your environment to tune the AS accordingly.
**OS-Level Settings**

1. To optimize network and OS performance, configure the following settings in `/etc/sysctl.conf` file of Linux. These settings specify a larger port range, a more effective TCP connection timeout value, and a number of other important parameters at the OS-level.

   ```
   net.ipv4.tcp_fin_timeout = 30
   fs.file-max = 2097152
   net.ipv4.tcp_tw_recycle = 1
   net.ipv4.tcp_tw_reuse = 1
   net.core.rmem_default = 524288
   net.core.wmem_default = 524288
   net.core.rmem_max = 67108864
   net.core.wmem_max = 67108864
   net.ipv4.tcp_rmem = 4096 87380 16777216
   net.ipv4.tcp_wmem = 4096 65536 16777216
   net.ipv4.ip_local_port_range = 1024 65535
   ```

   When we have the localhost port range configuration lower bound to 1024, there is a possibility that some processes may pick the ports which are already used by WSO2 servers. Therefore, it's good to increase the lower bound as sufficient for production, e.g., 10,000.

   ```
   * soft nofile 4096
   * hard nofile 65535
   ```

   Optimal values for these parameters depend on the environment.

2. To alter the number of allowed open files for system users, configure the following settings in `/etc/security/limits.conf` file of Linux.

3. To alter the maximum number of processes your user is allowed to run at a given time, configure the following settings in `/etc/security/limits.conf` file of Linux (be sure to include the leading `*` character). Each carbon server instance you run would require up to 1024 threads (with default thread pool configuration). Therefore, you need to increase the nproc value by 1024 per each carbon server (both hard and soft).

   ```
   * soft nproc 20000
   * hard nproc 20000
   ```

**JVM Settings**

If one or more worker nodes in a clustered deployment require access to the management console, you need to increase the entity expansion limit in the `<AS_HOME>/bin/wso2server.bat` file (for Windows) or the `<AS_HOME>/bin/wso2server.sh` file (for Linux/Solaris) as show below. The default entity expansion limit is 64000.
**JDBC Pool Configuration**

Within the WSO2 platform, we use Tomcat JDBC pooling as the default pooling framework due to its production ready stability and high performance. The goal of tuning the pool properties is to maintain a pool that is large enough to handle peak load without unnecessarily utilising resources. These pooling configurations can be tuned for your production server in general in the `<AS_HOME>/repository/conf/datasources/master-datasources.xml` file.

The following parameters should be considered when tuning the connection pool:

- The application's concurrency requirement.
- The average time taken to run a database query.
- The maximum number of connections the database server can support.

The table below indicates some recommendations on how to configure the JDBC pool. For more details about recommended JDBC configurations, see [Tomcat JDBC Connection Pool](#).

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
<th>Tuning Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxActive</td>
<td>The maximum number of active connections that can be allocated from the connection pool at the same time. The default value is <strong>100</strong>.</td>
<td>The maximum latency (approximately) = ( \frac{P}{M} ) * T, where, ( M ) = maxActive value, ( P ) = Peak concurrency value, ( T ) = Time (average) taken to process a query. That is, if the maximum latency (approximately) = ( \frac{P}{M} ) * T, then, the maxWait = ( \frac{P}{M} ) * T + buffer time.</td>
</tr>
<tr>
<td>maxWait</td>
<td>The maximum time that requests are expected to wait in the queue for a connection to be released. This property comes into effect when the maximum number of active connections allowed in the connection pool (see maxActive property) is used up.</td>
<td>Adjust this to a value slightly higher than the maximum latency for a request, so that a buffer time is added to the maximum latency. That is, if the maximum latency (approximately) = ( \frac{P}{M} ) * T, then, the maxWait = ( \frac{P}{M} ) * T + buffer time.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>minIdle</td>
<td>The minimum number of connections that can remain idle in the pool, without extra ones being created. The connection pool can shrink below this number if validation queries fail. Default value is 0.</td>
<td>This value should be similar or near to the average number of requests that will be received by the server at the same time. With this setting, you can avoid having to open and close new connections every time a request is received by the server.</td>
</tr>
<tr>
<td>maxIdle</td>
<td>The maximum number of connections that can remain idle in the pool.</td>
<td>The value should be less than the maxActive value. For high performance, tune maxIdle to match the number of average, concurrent requests to the pool. If this value is set to a large value, the pool will contain unnecessary idle connections.</td>
</tr>
<tr>
<td>testOnBorrow</td>
<td>The indication of whether connection objects will be validated before they are borrowed from the pool. If the object validation fails, the connection is dropped from the pool, and there will be an attempt to borrow another connection.</td>
<td>When the connection to the database is broken, the connection pool does not know that the connection has been lost. As a result, the connection pool will continue to distribute connections to the application until the application actually tries to use the connection. To resolve this problem, set “Test On Borrow” to “true” and make sure that the “ValidationQuery” property is set. To increase the efficiency of connection validation and to improve performance, validationInterval property should also be used.</td>
</tr>
<tr>
<td>validationInterval</td>
<td>This parameter controls how frequently a given validation query is executed (time in milliseconds). The default value is 30000 (30 seconds). That is, if a connection is due for validation, but has been validated previously within this interval, it will not be validated again.</td>
<td>Deciding the value for the &quot;validationInterval&quot; depends on the target application's behavior. Therefore, selecting a value for this property is a trade-off and ultimately depends on what is acceptable for the application. If a larger value is set, the frequency of executing the Validation Query is low, which results in better performance. Note that this value can be as high as the time it takes for your DBMS to declare a connection as stale. For example, MySQL will keep a connection open for as long as 8 hours, which requires the validation interval to be within that range. However, note that the validation query execution is usually fast. Therefore, even if this value is only large by a few seconds, there will not be a big penalty on performance. Also, specially when the database requests have a high throughput, the negative impact on performance is negligible. For example, a single extra validation query run every 30 seconds is usually negligible. If a smaller value is set, a stale connection will be identified quickly when it is presented. This maybe important if you need connections repaired instantly, e.g. during a database server restart.</td>
</tr>
</tbody>
</table>
Validation Query

The SQL query used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw an SQLException. The default value is null. Example values are `SELECT 1(mysql), select 1 from dual(oracle), SELECT 1(MS Sql Server).` Specify an SQL query, which will validate the availability of a connection in the pool. This query is necessary when `testBorrow` property is true.

When it comes to web applications, users are free to experiment and package their own pooling framework such BoneCP.

JNDI Resources

- Carbon JNDI Resources
- Exposing Datasources as JNDI Resources

Carbon JNDI Resources

Java Naming and Directory Interface (JNDI) is a Java Application Programming Interface (API) providing naming and directory functionality for Java software clients to discover and look up data and objects via a name. Carbon maintains platform level JNDI resources such as Carbon Datasources, Transaction Manager etc. Applications deployed in WSO2 Application Server (such as, web applications, Axis2 services etc.) can access them using an `InitialContext` object.

Example

```java
Context initCtx = new InitialContext();
DataSource ds = (DataSource) initialContext.lookup("jndi_name");
```

Carbon JNDI resources inherit the multitenant behavior from the Carbon platform. As a result, JNDI resources registered at tenant level can only be accessed by that particular tenant.

JNDI resources for web apps

WSO2 Applications Server deploys the web applications on embedded Tomcat server. As a result, the JNDI support provided by the Tomcat server is available for the web application. In addition to Tomcat level JNDI resources, web applications can lookup Carbon JNDI resources as mentioned above.

Registering a Tomcat JNDI resource

For more details on Tomcat JNDI resources, go to [http://tomcat.apache.org/tomcat-7.0-doc/jndi-resources-howto.html](http://tomcat.apache.org/tomcat-7.0-doc/jndi-resources-howto.html)
**Webapp level registering**

To register a JNDI resource on webapp level, place the JNDI resource in the `context.xml` file of the webapp. Since each webapp's context is isolated from each other, this ensures that the resources will be available to the webapp context only.

Example

```
<Resource name="jdbc/TestDataSource" auth="Container"
   type="javax.sql.DataSource"
   driverClassName="com.mysql.jdbc.Driver"
   maxActive="100" maxIdle="30" maxWait="10000"
   url="jdbc:mysql://localhost:3306/test_db"
   username="root" password="root"/>
```

The above example defines a database resource and its properties. You can define any number of resources. Properties of a resource should go inside its `<Resource>` element.

**Global registering**

Place resources under the `<GlobalNamingResources>` element in Tomcat's server configuration file (`<PRODUCT_HOME>repository/conf/tomcat/catalina-server.xml` in WSO2 products).

Example

```
<GlobalNamingResources>
   <Resource name="jdbc/TestDataSource" auth="Container"
      type="javax.sql.DataSource"
      driverClassName="com.mysql.jdbc.Driver"
      maxActive="100" maxIdle="30" maxWait="10000"
      url="jdbc:mysql://localhost:3306/test_db"
      username="root" password="root"/>
</GlobalNamingResources>
```

Then, the resources can be referenced by linking them in the webapp's `context.xml` file. Global resources are visible to all webapps deployed in the server.

Example

```
<ResourceLink name="jdbc/TestDataSource"
   global="jdbc/TestDataSource"
   type="javax.sql.DataSource"/>
```

Global naming resources are available for all the tenants. Web applications deployed in any tenant can access them by referencing them inside the web applications.

**Accessing JNDI resources from web applications**

Application Server 5.2.1 allows web applications to access both Tomcat and Carbon JNDI resources using an Init
object. However, Tomcat level JNDI resources get the precedence over Carbon level JNDI resources. As a result, when a lookup call is made using an InitialContext object, it first looks up in the Tomcat JNDI resources of that web app. If there are no resources bound for the given name in the Tomcat JNDI context of that web app, then it will look up in the Carbon JNDI context. If there is a resource bound for that name in the Carbon context, then it will return that resource.

In web applications Carbon JNDI resources are available only for lookup calls. That means methods such as listBindings() will return only the list of the Tomcat level JNDI resources.

Accessing Tomcat JNDI resources

Application Server 5.2.1 allows web applications to access Tomcat JNDI resources in the same manner as a native Tomcat server.

Example

```java
Context initCtx = new InitialContext();
DataSource ds = (DataSource) initialContext.lookup("java:comp/env/jdbc/TestDataSource");
```

Or

```java
Context initCtx = new InitialContext(environment);
Context context = (Context) initCtx.lookup("java:comp/env");
DataSource ds = (DataSource) context.lookup("jdbc/TestDataSource");
```

Exposing Datasources as JNDI Resources

Java Naming and Directory Interface (JNDI) is a Java application programming interface (API) providing naming and directory functionality for Java software clients to discover and look up data and objects via a name. WSO2 Application Server supports JNDI InitialContext implementation by inheriting the JNDI implementation of Tomcat. JNDI helps decouple object creation from the object look-up. When you have registered a datasource with JNDI, others can discover it through a JNDI look-up and use it.

The following sections describe how to register and look up JNDI resources, and how to expose datasources as JNDI resources.

- Registering JNDI Resources
- JNDI Look-up
- Accessing JNDI Resources within the Webapps

**Registering JNDI Resources**

First, register JNDI resources for webapps context. Registering can be done in two ways - webapp-level or global level.

**Webapp-Level Registering**

To register a JNDI resource on a webapp-level, place the JNDI resource in the context.xml file of the webapp. Since each webapp’s context is isolated from each other, this ensures that the resources will be available to the webapp
context only.

For example,

```xml
<Resource name="jdbc/TestDataSource" auth="Container"
    type="javax.sql.DataSource"
    driverClassName="com.mysql.jdbc.Driver"
    maxActive="100" maxIdle="30" maxWait="10000"
    url="jdbc:mysql://localhost:3306/test_db"
    username="root" password="root"/>
```

The example above defines a database resource and its properties. You can define any number of resources. Properties of a resource should go inside its `<Resource>` element.

Global Registering

Place resources under the `<GlobalNamingResources>` element in tomcat's server server configuration file (<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml in WSO2 products). Then, the resources can be referenced by linking them in the webapp's context.xml file. Global resources are visible to all webapps deployed in the server.

For example,

```xml
<ResourceLink name="jdbc/TestDataSource"
    global="jdbc/TestDataSource"
    type="javax.sql.DataSource"/>
```

Note

The classes used to define resource-type should be in the class path.

Finally, have a reference to those resources in the web.xml of the webapp. For example,

```xml
<resource-ref>
    <description>Your Description</description>
    <res-ref-name>jdbc/TestDataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

JNDI Look-up

Now that you are done registering JNDI resources, let's see how to look them up. In the example code segment below, SelectorContext is the Tomcat JNDI Context implementation. Similarly, build the Catalina selector context using the InitialContext environment, using which you can look up the JNDI resources.
Hashtable environment = new Hashtable();
environment.put("java.naming.factory.initial",
"org.wso2.carbon.tomcat.jndi.CarbonJavaURLContextFactory");
environment.put("java.naming.webapp.bound",
true);
Context initCtx = new InitialContext(environment);
Context initialContext = (Context) initCtx.lookup("java:comp/env");

Once done, you can look up for the resource defined earlier. For example,

DataSource ds = (DataSource) initialContext.lookup("jdbc/TestDataSource");

**Accessing JNDI Resources within the Webapps**

You can expose datasources as JNDI resources from the management console of the Application Server.

Next, let's take a look at how to access these JNDI resources within the webapps that are hosted in the same Application Server instance. This was not possible in previous versions due to the way JNDI context isolation is implemented for webapps at tomcat level. The implementation states that one webapp's JNDI context cannot be accessed by another webapp, unless the JNDI resources are registered globally. Since webapps run isolated from each other, their contexts will also be isolated. The JNDI context of carbon is also usually isolated from webapps.

Shown below is how the naming context factory class (i.e. org.apache.naming.java.javaURLContextFactory), which is used as the default Initial Context Factory class in WSO2 Carbon products, is implemented:

```java
public Context getInitialContext(Hashtable<?, ?> environment) throws NamingException {
    if (ContextBindings.isThreadBound() ||
        (ContextBindings.isClassLoaderBound())) {
        // Redirect the request to the bound initial context
        return new SelectorContext(
            (Hashtable<String, Object>) environment, true);
    }

    // If the thread is not bound, return a shared writable context
    if (initialContext == null) {
        synchronized (javaURLContextFactory.class) {
            if (initialContext == null) {
                initialContext = new NamingContext(
                    (Hashtable<String, Object>) environment, MAIN);
            }
        }
    }
}
```

In order to access JNDI resources within webapps, WSO2 has written a InitialContextFactory, which extends the above class and overrides the above methods. This new implementation of javaUrlContextFactory in WSO2 Carbon products does not check whether the current context is bound to a thread or class loader. It returns the JNDI of the
Carbon runtime.

The following segment is set as the default Initial Context Factory class in $\langle$PRODUCT_HOME$\rangle$/repository/conf/carbon.xml file.

```
```

When you want to access the Carbon JNDI Context to lookup for Datasources, add the following JNDI Initial Context Factory Environment property:

```
java.naming.factory.initial :
"org.wso2.carbon.tomcat.jndi.CarbonJavaURLContextFactory"
```

The following steps explain how this works.

1. Log on to the product's management console and select "Datasources."

![Data Sources](image)

2. Click on the "Add datasource" link in the "Datasources" page.

![Add Data Source](image)

3. Select RDBMS as the **Datasource Type**. Provide the rest of the details accordingly. You will find an option to expose this datasource as a JNDI Datasource at the end of the screen.

In this example, we are exposing the new Datasource as `jdbc/MyCarbonDataSource`. 
**New Data Source**

- **Name**: Name of the JNDI datasource which will be visible to others in object look-up.

- **Use Datasource Factory**: If the datasource needs to be accessed from an external environment, the "Datasource Factory" option should be used. When it is selected, a reference object is created with defined datasource properties. When accessing the datasource from an external environment, the "Datasource Factory" creates a datasource instance based on the values of the reference object. In the configuration, this is set as useDataSourceFactory="true".

- **JNDI Properties**: Properties related to the jndi datasource (username, password etc). If the "Datasource Factory" is selected, following properties should be specified.
  - `java.naming.factory.initial` - Used to select the registry service provider as the initial context.
  - `java.naming.provider.url` - Specifies the location of the registry when the registry is being used as the initial context.

4. Click **Save** to complete. It will expose the Datasource.

5. The code segment below shows how to access this exposed Datasource within a webapp.
try {
    Hashtable environment = new Hashtable();
    environment.put("java.naming.factory.initial",
    "org.wso2.carbon.tomcat.jndi.CarbonJavaURLContextFactory");
    Context initContext = new InitialContext(environment);
    Object result = initContext.lookup("jdbc/MyCarbonDataSource");
    if (result != null) {
        // Do your work here
    } else {
        System.out.println("Cannot find MyCarbonDataSource");
    }
} catch (NamingException e) {
    e.printStackTrace();
}

In the above code segment, we create an InitialContext with the relevant environment properties. It gives the JNDI context of Carbon, using which you can look for any object that is bound to that context. Here you are looking `jdbc/MyCarbonDataSource` up.

Similarly, you can look up any Datasource, which is exposed as a JNDI resource, from webapps.
**Administration Guide**

This section explores various product deployment and configuration options useful for system administrators.

- Working with Users, Roles and Permissions
- Working with Features
- Monitoring the Application Server
- Working with Composite Applications
- Running the Product on a Preferred Profile
- Migrating the Application Server
- Calling Admin Services from Apps
- Working with Applications
- Working with Services
- Upgrading from a Previous Release

**Working with Users, Roles and Permissions**

This section contains the following information:

- Introduction to User Management
- Configuring the User Realm
- Managing Users, Roles and Permissions

**Introduction to User Management**

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. The following documentation introduces the main concepts in User Management, such as users, roles, permissions, user stores etc. and how they are used in WSO2 products.

User management involves defining and managing users, roles, and their access levels in a system. A user management dashboard or console provides system administrators with a high-level view of a system's active user sessions, their log-in statuses, the privileges of each user, and their activity in the system, enabling system admins to make business-critical, real-time security decisions. A typical user management implementation involves a wide range of functionality such as adding/deleting users, controlling user activity through permissions, managing user roles, defining authentication policies, managing external user stores, manual/automatic log-out, and resetting user passwords.

Any user management system has the following basic components:

- **Users**: Users are consumers who interact with your organizational applications, databases, and other systems. A user can be a person, a device, or another application/program within or outside of the organization's network. Because users interact with internal systems and access data, security-conscious organizations need to define which data and functionality each user can access by assigning permissions.

- **Permissions**: A permission is a delegation of authority or a right that is assigned to a user or a group of users to perform an action on a system. Permissions can be granted to or revoked from a user, user group, or user role automatically or by a system administrator. For example, if a user has the permission to log in to a system, the permission to log out is automatically granted as well.

- **User roles**: A user role is a grouping of permissions. In addition to assigning individual permissions to users, admins can create user roles and assign those roles to users. For example, you might create user roles called VP, Manager, and Employee, each of which has a different set of permissions, and then assign those roles to users based on their position in the company. Then, if you need to modify the permissions of all your managers, you can simply modify the Manager user role, and all users with that role will have their permissions updated automatically.
The following diagram illustrates how the user management functionality is structured to work in WSO2 products:

- **User stores**: A user store is the database where information about the users and user roles is stored, including log-in name, password, first name, last name, and e-mail address.

- **RDBMS (for Authentication and Authorization)**: This RDBMS stores information of the role-based permissions.

According to the default configuration in WSO2 products, the embedded H2 RDBMS that is shipped with the product is used as the user store as well as the RDBMS for storing information related to permissions.

- **Realm configuration**: The user realm consists of the configurations required to initialise the user realm. The user-mgt.xml file stored in the \<PRODUCT_HOME>/repository/conf/ directory is used as the realm configuration XML. This includes setting up the User Store Manager, the Authorization Manager and the System Administrator. These configurations are explained below.

<table>
<thead>
<tr>
<th>User Store Manager</th>
<th>The User Store Manager is responsible for managing the underlying user store. It is represented by the UserStoreManager Java interface. There can be different User Store Manager implementations to connect with different user stores, but you can configure only one User Store Manager implementation in a single user realm (that is, a single WSO2 Carbon instance). The User Store Manager can be operated in both read/write mode and read-only mode. In read-only mode, you can only connect with an existing user store. WSO2 products provide the following default User Store Manager implementations:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>- JDBCUserStoreManager (read and write)</td>
</tr>
<tr>
<td></td>
<td>- LDAPUserStoreManager (read-only)</td>
</tr>
<tr>
<td></td>
<td>- ApacheDSUserStoreManager (read and write)</td>
</tr>
</tbody>
</table>
You can write a custom user store manager implementation by implementing `UserStoreManager` or by extending `AbstractUserStoreManager` or one of the default implementations.

**Using JDBCUserStoreManager**

The `JDBCUserStoreManager` class uses a schema that is specific to WSO2 Carbon. It contains the following tables:

- **UM_USER**: Contains user names and passwords
- **UM_ROLE**: Contains role names
- **UM_USER_ROLE**: Contains user role mappings
- **UM_USER_ATTRIBUTE**: Contains user attributes. There can be any attribute ID and a value for that attribute ID that is associated with a user’s profile.

You can find the full schema of these tables from the database script files in the `<PRODUCT_HOME>/dbscripts` directory. Note that these scripts also contain schemas for other tables that are used for user management and registry functions. If your organization contains an existing JDBC user store that you want to use with a WSO2 product, you must extend `JDBCUserStoreManager` and write a new implementation for your user store according to your schema.

**Authorization Manager**

The Authorization Manager uses role-based access control (RBAC) to protect resources related to the WSO2 Carbon platform. The default implementation of the Authorization Manager is `JDBCAuthorizationManager`, which uses a permission model specific to WSO2 Carbon and uses the authorization data that is stored in tables in the JDBC database. You can replace this implementation with a custom implementation (for example, if you want to use a XACML authorization manager) and use it with WSO2 products.
Related Topics

- Configuring the User Realm: This topic explains how you can set up and configure the user management realm.
- Managing Users, Roles and Permissions: This topic explains how you can manage the Users, Roles and Permissions using the management console.

Configuring the User Realm

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. The following documentation explains the configurations that should be done in WSO2 products in order to set up the User Management module.

The complete functionality and contents of the User Management module is called a user realm. The realm includes the user management classes, configurations and repositories that store information. Therefore, configuring the User Management functionality in a WSO2 product involves setting up the relevant repositories and updating the relevant configuration files.

The following diagram illustrates the required configurations and repositories:

See the following topics for instructions:

- Configuring the System Administrator
• Configuring the Authorization Manager
• Configuring User Stores

Configuring the System Administrator

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. This documentation explains the main settings relevant to the system administrator.

The `admin` user is the super tenant that will be able to manage all other users, roles and permissions in the system by using the management console of the product. Therefore, the user that should have admin permissions is required to be stored in the primary user store when you start the system for the first time. The documentation on setting up primary user stores will explain how to configure the administrator while configuring the user store. The information under this topic will explain the main configurations that are relevant to setting up the system administrator.

Note the following key facts about the system administrator in your system:

- The admin user and role is always stored in the primary user store in your system.
- An administrator is configured for your system by default. This `admin` user is assigned to the `admin` role, which has all permissions enabled.
- The permissions assigned to the default `admin` role cannot be modified.

Before you begin:

Ensure that you have a primary user store (for storing users and roles) and an RDBMS (for storing information related to permissions). See the related documentation for instructions on how to set up these repositories.

Updating the administrator

The `<Configuration>` section at the top of the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file allows you to configure the administrator user in your system as well as the RDBMS that will be used for storing information related to user authentication (i.e. role-based permissions).
<Realm>
  <Configuration>
    <AddAdmin>true</AddAdmin>
    <AdminRole>admin</AdminRole>
    <AdminUser>
      <UserName>admin</UserName>
      <Password>admin</Password>
    </AdminUser>
    <EveryOneRoleName>everyone</EveryOneRoleName> <!-- By default users in this role see the registry root -->
    <Property name=""></Property>
    ............
  </Configuration>
  ...  
</Realm>

Note the following regarding the configuration above.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;AddAdmin&gt;</td>
<td>When true, this element creates the admin user based on the AdminUser element. It also indicates whether to create the specified admin user if it doesn't already exist. When connecting to an external read-only LDAP or Active Directory user store, this property needs to be false if an admin user and admin role exist within the user store. If the admin user and admin role do not exist in the user store, this value should be true, so that the role is added to the user management database. However, if the admin user is not there in the user store, we must add that user to the user store manually. If the AddAdmin value is set to true in this case, it will generate an exception.</td>
</tr>
<tr>
<td>&lt;AdminRole&gt;</td>
<td>This is the role that has all administrative privileges of the WSO2 product, so all users having this role are admins of the product. You can provide any meaningful name for this role. This role is created in the internal H2 database when the product starts. This role has permission to carry out any actions related to the Management Console. If the user store is read-only, this role is added to the system as a special internal role where users are from an external user store.</td>
</tr>
<tr>
<td>&lt;AdminUser&gt;</td>
<td>Configures the default administrator for the WSO2 product. If the user store is read-only, the admin user must exist in the user store or the system will not start. If the external user store is read-only, you must select a user already existing in the external user store and add it as the admin user that is defined in the &lt;AdminUser&gt; element. If the external user store is in read/write mode, and you set &lt;Add Admin&gt; to true, the user you specify will be automatically created.</td>
</tr>
<tr>
<td>&lt;UserName&gt;</td>
<td>This is the username of the default administrator or super tenant of the user store. If the user store is read-only, the admin user MUST exist in the user store for the process to work.</td>
</tr>
</tbody>
</table>

<Password>

Do NOT put the password here but leave the default value. If the user store is read-only, this element and its value are ignored. This password is used only if the user store is read-write and the Admin value is set to true.

Note that the password in the user-mgt.xml file is written to the primary user store when the server starts for the first time. Thereafter, the password will be validated from the primary user store and not from the user-mgt.xml file. Therefore, if you need to change the admin password stored in the user store, you cannot simply change the value in the user-mgt.xml file. To change the admin password, you must use the Change Password option from the management console. See the related topics for instructions.

<EveryOneRoleName>

The name of the "everyone" role. All users in the system belong to this role.

Related Topics

- **Configuring the Primary User Stores**: This topic explains how the primary user store is set up and configured for your product.
- **Configuring the Authorization Manager**: This topic explains how the repository (RDBMS) for storing authorization information (role-based permissions) is configured for your product.
- **Changing a Password**: This topic explains how you can change the admin password using the management console of the product.

Configuring the Authorization Manager

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the user-mgt.xml file found in the <PRODUCT_HOME>/repository/conf/ directory. This documentation explains how to set up a repository for storing authorization information (role-based permissions) and how to change the relevant configurations.

According to the default configuration in WSO2 products, the Users, Roles and Permissions are stored in the same repository (i.e., the default, embedded H2 database). However, you can change this configuration in such a way that the Users and Roles are stored in one repository (User Store) and the Permissions are stored in a separate repository. A user store can be a typical RDBMS, an LDAP or an external Active Directory. See the related topics for information on how user stores are configured.

The repository that stores Permissions should always be an RDBMS. The Authorization Manager configuration in the user-mgt.xml file (stored in the <PRODUCT_HOME>/repository/conf/ directory) connects the system to this RDBMS.

Follow the instructions given below to set up and configure the Authorization Manager.

- **Step 1: Setting up the repository**
- **Step 2: Updating the user realm configurations**

**Step 1: Setting up the repository**

By default, the embedded H2 database is used for storing permissions. You can change this as follows:
1. Change the default H2 database or set up another RDBMS for storing permissions.
   2. When you set up an RDBMS for your system, it is necessary to create a corresponding datasource, which allows the system to connect to the database.
      - If you are replacing the default H2 database with a new RDBMS, update the master-datasource.xml file (stored in the <PRODUCT_HOME>/repository/conf/datasources/ directory) with the relevant information.
      - Alternatively, create a new XML file with the datasource information of your new RDBMS and store it in the same <PRODUCT_HOME>/repository/conf/datasources/ directory.

Refer the related topics for detailed information on setting up databases and configuring datasources.

**Step 2: Updating the user realm configurations**

Once you have set up a new RDBMS and configured the datasource, the user-mgt.xml file (user realm configuration) should be updated as explained below.

**Setting up the database connection**

Update the datasource information using the <Property> element under <Configuration>. Given below are the properties that are set by default.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dataSource</td>
<td>This is the jndi name of the datasource that is used for referring to the datasource. In the following example, the jndi name of the default datasource defined in the &lt;PRODUCT_HOME&gt;/repository/conf/datasources/master-datasources.xml file linked from the user-mgt.xml file.</td>
</tr>
<tr>
<td>isCascadeDeleteEnabled</td>
<td>This property is set to 'true' by default, which enables cascade delete for the UM_USER_PERMISSION and UM_ROLE_PERMISSION tables when a permission gets deleted from the UM_PERMISSION table. That is, if a record in the parent table is deleted the corresponding records in the child table will be automatically deleted.</td>
</tr>
</tbody>
</table>

You can add more optional configurations using the <Property> element:

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
<th>Mand</th>
</tr>
</thead>
<tbody>
<tr>
<td>testOnBorrow</td>
<td>It is recommended to set this property to 'true' so that object connections will be validated before being borrowed from the JDBC pool. For this property to be effective, the validationQuery parameter in the &lt;PRODUCT_HOME&gt;/repository/conf/datasources/master-datasources.xml file should be a non-string value. This setting will avoid connection failures. See the section on performance tuning of WSO2 products for more information.</td>
<td>Option</td>
</tr>
</tbody>
</table>

**Configuring the Authorization Manager**

Shown below is how the Authorization Manager is enabled in the user-mgt.xml file.
<AuthorizationManager
class="org.wso2.carbon.user.core.authorization.JDBCAuthorizationManager">
  <Property
  name="AdminRoleManagementPermissions">/permission</Property>
  <Property name="AuthorizationCacheEnabled">true</Property>
</AuthorizationManager>

- The org.wso2.carbon.user.core.authorization.JDBCAuthorizationManager class enables the Authorization Manager for your product.
- The AdminRoleManagementPermissions property sets the registry path where the authorization information (role-based permissions) are stored. Note that this links to the repository that you defined in Step 1.
- It is recommended to enable the GetAllRolesOfUserEnabled property in the AuthorizationManager as follows:

  <Property name="GetAllRolesOfUserEnabled">true</Property>

Although using the user store manager does not depend on this property, you must consider enabling this if there are any performance issues in your production environment. Enabling this property affects the performance when the user logs in. This depends on the users, roles and permission stats.

- By default, the rules linked to a permission (role name, action, resource) are not case sensitive. If you want to make them case sensitive, enable the following property:

  <Property name="CaseSensitiveAuthorizationRules">true</Property>

**Related Topics**

- Configuring User Stores: This topic explains how the repositories for storing information about Users and Roles are configured.
- Setting Up the Physical Database: This section explains how you can set up a new RDBMS and configure it for your system.
- Managing Datasources: This section explains the purpose of defining datasources and how they are configured for a product.

**Configuring User Stores**

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the user-mgt.xml file found in the <PRODUCT_HOME>/repository/conf/ directory. This topic explains the different types of user stores that can be configured for a WSO2 product.

The user management feature in WSO2 products allows you to maintain multiple user stores for your system that are used to store the users and their roles. You can set up any of the following types of user stores:

- JDBC user stores
- Active Directory user stores
- Read-Only LDAP user stores
- Read-Write LDAP user stores

The Primary User Store in every WSO2 product is configured in the <PRODUCT_HOME>/repository/conf/user-mgt.xml file. By default, the embedded H2 database (JDBC) that is shipped with WSO2 products is configured
as the primary user store, except for WSO2 Identity Server, which has an embedded LDAP as its primary user store. You can change the default configuration by replacing the default database according to your requirement. The primary user store is shared among all the tenants in the system.

With the user management feature, any number of **Secondary User Stores** can be easily set up for your system using the management console. This will automatically create an XML file with the configurations corresponding to the secondary user store in the same `<PRODUCT_HOME>/repository/conf/` directory. Alternatively, you can manually create the configuration file and store it in this directory without using the management console.

Although, information about users and roles are stored in the repositories that we call User Stores, which can be of any of the types described above, the permissions attached to roles are always stored in an RDBMS. According to the default configuration in WSO2 products, the embedded H2 database is used for storing permissions as well as users and roles. The instructions in this section explain how you can change the default user store. See the section on configuring the authorization manager for information on how to set up the RDBMS for storing permissions.

### Related Topics

See the following topics for instructions on setting up user stores:

- Configuring the Primary User Store
- Configuring Secondary User Stores
- Working with Properties of User Stores

See the topic on configuring the Authorization Manager for instructions on how to set up a repository for storing permissions.

### Configuring the Primary User Store

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. This documentation explains the process of setting up a primary user store for your system.

### The default User Store

The primary user store that is configured by default in every WSO2 product is a JDBC user store, which reads/writes into the internal database of the product server. By default, the internal database is H2 (except for WSO2 IS, which uses an LDAP as the default user store). This database is used by the Authorization Manager (for user authentication information) as well as the User Store Manager (for defining users and roles).

Instead of using the embedded database, you can set up a separate repository and configure it as your primary user store. Since the user store you want to connect to might have different schemas from the ones available in the embedded user store, it needs to go through an adaptation process. WSO2 products provide the following adapters, for connecting to LDAP, Active Directory and JDBC. Thereby, these adapters enable you to authenticate users from different types of user stores.

<table>
<thead>
<tr>
<th>User store manager class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager</td>
<td>UseReadOnlyLDAPUserStoreManager external LDAP user stores.</td>
</tr>
</tbody>
</table>
org.wso2.carbon.user.core.ldap.ReadWriteLDAPUserStoreManager | Use ReadWriteLDAPUserStoreManager for both read and write operations. This user store configuration is uncommented in the code in the file for WSO2 Identity Server.

org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager | Use ActiveDirectoryUserStoreManager to configure an Active Directory Domain Service (AD DS) or Active Directory Lightweight Directory Service (AD LDS). This can be used only for read-only, you must use org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager.

org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager | Use JDBCUserStoreManager for both internal and external JDBC user stores. This is the user store configuration which is uncommented in the code in the user-mgt.xml file for all WSO2 products, except WSO2 Identity Server.

The user-mgt.xml file already has sample configurations for all of the above user stores. To enable the required user store configuration, you must uncomment them in the code and comment out the ones that you do not need as explained in the following topics.

- Configuring a JDBC User Store
- Configuring a Read-Only LDAP or Active Directory User Store
- Configuring a Read-Write Active Directory User Store
- Configuring a Read-Write LDAP User Store

Configuring a JDBC User Store

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the user-mgt.xml file found in the <PRODUCT_HOME>/repository/conf/ directory. This file is shipped with user store manager configurations for all possible user store types (JDBC, read-only LDAP/Active Directory, read-write LDAP and read-write Active directory). The instructions given below explains how to configure an RDBMS (JDBC) as the primary user store for the WSO2 server.

The default User Store

The primary user store that is configured by default in every WSO2 product is a JDBC user store, which reads/writes into the internal database of the product server. By default, the internal database is H2 for all WSO2 products excluding the Identity Server. This database is used by the Authorization Manager (for user authentication information) as well as the User Store Manager (for defining users and roles). In the case of the WSO2 Identity Server, the default user store is an LDAP (Apache DS) that is shipped with the product.

When you configure a JDBC user store as the primary user store, you can either use the default configuration or you can change it in the following ways:

- You can set up two separate databases for the Authorization Manager and the User Store Manager.
- It is not recommended to use the default H2 database in production. Therefore, you can replace this as instructed in the related documentation.

Therefore, before you begin, ensure that the RDBMS that you want to use as the JDBC user store is correctly set up for your system. Then, follow the steps given below to configure a JDBC user store as the primary user store in your product.
Step 1: Configuring the JDBC user store manager
Step 2: Updating the system administrator
Step 3: Updating the datasources
Step 4: Starting the server

Step 1: Configuring the JDBC user store manager

Before you begin

- If you create the user-mgt.xml file yourself, be sure to save it in the <PRODUCT_HOME>/repository/conf/ directory.
- The class attribute for JDBC is <UserStoreManager class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">.

To configure a JDBC user store as the primary user store, you must change the JDBCUserStoreManager section in the <PRODUCT_HOME>/repository/conf/user-mgt.xml file.

1. Uncomment the following section:

   ```xml
   <UserStoreManager
class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
   ```

2. Specify the connection to the RDBMS inside the JDBC user store manager according to your requirement. For more information about user store properties, see the related documentation.

   - **Internal JDBC User Store**
   - **External JDBC User Store**

   The following sample shows how to configure the internal RDBMS as the JDBC user store:
The following sample shows how to configure an external RDBMS as the JDBC user store:

```xml
<UserStoreManager
class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
  <Property
name="TenantManager">org.wso2.carbon.user.core.tenant.JDBCTenantManager</Property>
  <Property name="ReadOnly">false</Property>
  <Property name="MaxUserNameListLength">100</Property>
  <Property name="IsEmailUserName">false</Property>
  <Property name="DomainCalculation">default</Property>
  <Property name="PasswordDigest">SHA-256</Property>
  <Property name="StoreSaltedPassword">true</Property>
  <Property name="UserNameUniqueAcrossTenants">false</Property>
  <Property name="PasswordJavaRegEx">^[^!\#\%\^*+=\{\}\|\<\>\,\'\"\]$</Property>
  <Property name="PasswordJavaScriptRegEx">^[^!\#\%\^*+=\{\}\|\<\>\,\'\"\]</Property>
  <Property name="UsernameJavaRegEx">^[^!\#\%=~\~\^*+=\{\}\|\<\>\,\'\"\]$</Property>
  <Property name="UsernameJavaScriptRegEx">^[^!\#\%=~\~\^*+=\{\}\|\<\>\,\'\"\]</Property>
  <Property name="RolenameJavaRegEx">^[^!\#\%=~\~\^*+=\{\}\|\<\>\,\'\"\]$</Property>
  <Property name="RolenameJavaScriptRegEx">^[^!\#\%=~\~\^*+=\{\}\|\<\>\,\'\"\]</Property>
  <Property name="UserRolesCacheEnabled">true</Property>
</UserStoreManager>
```
<Property name="PasswordJavaScriptRegEx">^[\S]{5,30}$</Property>
<Property name="UsernameJavaRegEx">^[\S]{5,30}$</Property>
<Property name="UsernameJavaScriptRegEx">^[\S]{5,30}$</Property>
<Property name="RolenameJavaRegEx">^[\S]{5,30}$</Property>
<Property name="RolenameJavaScriptRegEx">^[\S]{5,30}$</Property>
<Property name="SCIMEnabled">false</Property>
<Property name="SelectUserSQL">SELECT * FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?</Property>
<Property name="GetRoleListSQL">SELECT UM_ROLE_NAME, UM_TENANT_ID, UM_SHARED_ROLE FROM UM_ROLE WHERE UM_ROLE_NAME LIKE ? AND UM_TENANT_ID=? AND UM_SHARED_ROLE = '0' ORDER BY UM_ROLE_NAME</Property>
<Property name="GetSharedRoleListSQL">SELECT UM_ROLE_NAME, UM_TENANT_ID, UM_SHARED_ROLE FROM UM_ROLE WHERE UM_ROLE_NAME LIKE ? AND UM_SHARED_ROLE = '1' ORDER BY UM_ROLE_NAME</Property>
<Property name="IsRoleExistingSQL">SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND UM_TENANT_ID=?</Property>
<Property name="GetUserListOfSharedRoleSQL">SELECT UM_USER_NAME FROM UM_SHARED_USER_ROLE INNER JOIN UM_USER ON UM_SHARED_USER_ROLE.UM_USER_ID = UM_USER.UM_ID WHERE UM_ROLE.UM_ROLE_NAME= ? AND UM_SHARED_USER_ROLE.UM_USER_TENANT_ID = UM_USER.UM_TENANT_ID AND UM_SHARED_USER_ROLE.UM_ROLE_TENANT_ID = UM_ROLE.UM_TENANT_ID</Property>
<Property name="IsUserExistingSQL">SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?</Property>
<Property name="GetUserPropertyForProfileSQL">SELECT
WSO2 Application Server, version 5.3.0

2.


<Property name="GetProfileNamesSQL">SELECT DISTINCT UM_PROFILE_ID FROM UM_USER_ATTRIBUTE WHERE UM_TENANT_ID = ?</Property>

<Property name="GetUserProfileNamesSQL">SELECT DISTINCT UM_PROFILE_ID FROM UM_USER_ATTRIBUTE WHERE UM_USER_ID = (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?) AND UM_TENANT_ID = ?</Property>

<Property name="GetUserIDFromUserNameSQL">SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?</Property>

<Property name="GetUserNameFromTenantIDSQL">SELECT UM_USER_NAME FROM UM_USER WHERE UM_TENANT_ID = ?</Property>

<Property name="GetTenantIDFromUserNameSQL">SELECT UM_TENANT_ID FROM UM_USER WHERE UM_USER_NAME = ?</Property>

<Property name="AddUserSQL">INSERT INTO UM_USER (UM_USER_NAME, UM_USER_PASSWORD, UM_SALT_VALUE, UM_REQUIRE_CHANGE, UM_CHANGED_TIME, UM_TENANT_ID) VALUES (?, ?, ?, ?, ?, ?)</Property>

<Property name="AddUserToRoleSQL">INSERT INTO UM_USER_ROLE (UM_USER_ID, UM_ROLE_ID, UM_TENANT_ID) VALUES ((SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?), (SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?), ?)</Property>

<Property name="AddRoleSQL">INSERT INTO UM_ROLE (UM_ROLE_NAME, UM_TENANT_ID) VALUES (?, ?)</Property>

<Property name="AddSharedRoleSQL">UPDATE UM_ROLE SET UM_SHARED_ROLE = ? WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?</Property>

<Property name="AddRoleToUserRoleSQL">INSERT INTO UM_USER_ROLE (UM_ROLE_ID, UM_USER_ID, UM_TENANT_ID) VALUES ((SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?), (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?), ?)</Property>

<Property name="AddSharedRoleToUserRoleSQL">INSERT INTO UM_SHARED_USER_ROLE (UM_ROLE_ID, UM_USER_ID, UM_USER_TENANT_ID, UM_ROLE_TENANT_ID) VALUES ((SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?), (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?), ?, ?)</Property>


<Property name="RemoveUserFromRoleSQL">DELETE FROM UM_USER_ROLE WHERE UM_USER_ID = (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?) AND UM_ROLE_ID = (SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?) AND UM_TENANT_ID = ?</Property>

<Property name="RemoveRoleFromUserRoleSQL">DELETE FROM UM_USER_ROLE WHERE UM_USER_ID = (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID = ?) AND UM_ROLE_ID = (SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID = ?) AND UM_TENANT_ID = ?</Property>
WHERE UM_ROLE_ID=(SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND UM_TENANT_ID=?) AND UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?) AND UM_TENANT_ID=?

PROPERTY name="DeleteRoleSQL">DELETE FROM UM_ROLE WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID=?</PROPERTY>

PROPERTY name="OnDeleteRoleRemoveUserRoleMappingSQL">DELETE FROM UM_USER_ROLE WHERE UM_ROLE_ID=(SELECT UM_ID FROM UM_ROLE WHERE UM_USER_NAME=? AND UM_TENANT_ID=?) AND UM_TENANT_ID=?

PROPERTY name="DeleteUserSQL">DELETE FROM UM_USER WHERE UM_USER_NAME = ? AND UM_TENANT_ID=?


PROPERTY name="OnDeleteUserRemoveUserAttributeSQL">DELETE FROM UM_USER_ATTRIBUTE WHERE UM_USER_ID=(SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?) AND UM_TENANT_ID=?


PROPERTY name="UpdateRoleNameSQL">UPDATE UM_ROLE set UM_ROLE_NAME=? WHERE UM_ROLE_NAME = ? AND UM_TENANT_ID=?


PROPERTY name="UserNameUniqueAcrossTenantsSQL">SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=?</PROPERTY>

PROPERTY name="IsDomainExistingSQL">SELECT UM_DOMAIN_ID FROM UM_DOMAIN WHERE UM_DOMAIN_NAME=? AND UM_TENANT_ID=?

PROPERTY name="AddDomainSQL">INSERT INTO UM_DOMAIN (UM_DOMAIN_NAME, UM_TENANT_ID) VALUES (?, ?)

PROPERTY name="AddUserToRoleSQL-mssql">INSERT INTO UM_USER_ROLE (UM_USER_ID, UM_ROLE_ID, UM_TENANT_ID) SELECT (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?), (SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND UM_TENANT_ID=?), (?)

PROPERTY name="AddRoleToUserSQL-mssql">INSERT INTO UM_USER_ROLE (UM_ROLE_ID, UM_USER_ID, UM_TENANT_ID) SELECT (SELECT UM_ID FROM UM_ROLE WHERE UM_ROLE_NAME=? AND UM_TENANT_ID=?), (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?), (?)

PROPERTY name="AddUserRolePropertySQL-mssql">INSERT INTO UM_USER_ATTRIBUTE (UM_USER_ID, UM_ATTR_NAME, UM_ATTR_VALUE, UM_PROFILE_ID, UM_TENANT_ID) SELECT (SELECT UM_ID FROM UM_USER WHERE UM_USER_NAME=? AND UM_TENANT_ID=?), (?), (?), (?), (?)

PROPERTY name="AddUserToRoleSQL-openedge">INSERT INTO UM_USER_ROLE (UM_USER_ID, UM_ROLE_ID, UM_TENANT_ID) SELECT UU.UM_ID,


The sample for the external JDBC user store consists of properties pertaining to various SQL statements. This is because the schema may be different for an external user store, and these adjustments need to be made in order to streamline the configurations with WSO2 products.

3. Add the `PasswordHashMethod` property to the UserStoreManager configuration for JDBCUserStoreManager. For example:

   ```xml
   <UserStoreManager
class="org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager">
   <Property name="PasswordHashMethod">SHA</Property>
   ...
   </UserStoreManager>
   ``

   The `PasswordHashMethod` property specifies how the password should be stored. It usually has the following values:

   - SHA: Uses SHA digest method.
   - MD5: Uses MD5 digest method.
   - PLAIN_TEXT: Plain text passwords.

   In addition, it also supports all digest methods in [http://docs.oracle.com/javase/6/docs/api/java/security/MessageDigest.html](http://docs.oracle.com/javase/6/docs/api/java/security/MessageDigest.html).

4. If you are setting up an external JDBC user store, you need to set the following property to 'true' to be able to create roles in the primary user store.

   ```xml
   <Property name="WriteGroups">false</Property>
   ``

Step 2: Updating the system administrator

The `admin` user is the super tenant that will be able to manage all other users, roles and permissions in the system by using the management console of the product. Therefore, the user that should have admin permissions is required to be stored in the user store when you start the system for the first time. If the JDBC user store is read-only, then we need to always use a user ID that is already in the user store as the super tenant. Otherwise, if the JDBC user store can be written to, you have the option of creating a new admin user in the user store when you start the system for the first time. Refer the related topics for information about the system administrator.

These two alternative configurations can be done as explained below.

- If the user store is read-only, find a valid user that already resides in the RDBMS. For example, say a valid username is AdminSOA. Update the `<AdminUser>` section of your configuration as shown below. You do not have to update the password element as it is already set in the user store.
If the user store can be written to, you can add the super tenant user to the user store. Therefore, `<AddAdmin>` should be set to `true` as shown below.

```
<AddAdmin>true</AddAdmin>
<AdminRole>admin</AdminRole>
<AdminUser>
  <UserName>admin</UserName>
  <Password>admin</Password>
</AdminUser>
```

In the realm configuration section, set the value of the `MultiTenantRealmConfigBuilder` property to `org.wso2.carbon.user.core.config.multitenancy.SimpleRealmConfigBuilder`. For example:

```
<Property name="MultiTenantRealmConfigBuilder">org.wso2.carbon.user.core.config.multitenancy.SimpleRealmConfigBuilder</Property>
```

### Step 3: Updating the datasources

Whenever there is an RDBMS set up for your system, it is necessary to create a corresponding datasource, which allows the system to connect to the database. The datasource for the internal H2 database that is shipped with WSO2 products by default, is configured in the `master-datasources.xml` file, which is stored in the `<PRODUCT_HOME>/repository/conf/datasources/` directory. Refer the related topics for detailed information on setting up databases and configuring datasources.

1. There are two possible methods for updating datasources:
   - Shown below is how the `master-datasources.xml` file is configured to connect to the default H2 database in your system. If you have replaced the default database with a new RDBMS, which you are now using as the JDBC users store, you have to update the `master-datasource.xml` file with the relevant information.
1. **datasource**
   
   ```xml
   <name>WSO2_CARBON_DB</name>
   <description>The datasource used for registry and user manager</description>
   <jndiConfig>
     <name>jdbc/WSO2CarbonDB</name>
   </jndiConfig>
   <definition type="RDBMS">
     <configuration>
       <url>jdbc:h2:repository/database/WSO2CARBON_DB;DB_CLOSE_ON_EXIT=FALSE;LOCK_TIMEOUT=60000</url>
       <username>wso2carbon</username>
       <password>wso2carbon</password>
       <driverClassName>org.h2.Driver</driverClassName>
       <maxActive>50</maxActive>
       <maxWait>60000</maxWait>
       <testOnBorrow>true</testOnBorrow>
       <validationQuery>SELECT 1</validationQuery>
       <validationInterval>30000</validationInterval>
     </configuration>
   </definition>
   </datasource>
   
   Alternatively, instead of using the master-datasource.xml file, you can also create a new XML file with the datasource information of your new RDBMS and store it in the same directory.

2. Now, the datasource configuration and the user store manager configuration in the user-mgt.xml file should be linked together. You can do this by referring to the datasource information (typically defined in the master file) from the user-mgt.xml file as explained below.
   
   • The RDBMS that is used for storing Authorization information is configured under the <Configuration> section in the user-mgt.xml file, by adding `<Property name="dataSource"> as shown below. The following example refers to the default H2 database.

   ```xml
   <Configuration>
     ......
     <Property name="dataSource">jdbc/WSO2CarbonDB</Property>
   </Configuration>
   
   If you are using the same RDBMS as the user store in your system, this datasource reference will suffice.

   • However, if you have set up a separate RDBMS as the user store, instead of using a common RDBMS for Authorization information as well as the user store, you must refer to the datasource configuration from within the User Store Manager configuration in the user-mgt.xml file by adding the `<Property name="dataSource"> property.

Step 4: Starting the server
1. Add the JDBC driver to the classpath by copying its JAR file into the `<PRODUCT_HOME>/repository/components/lib` directory.
2. Start the server.

Related Topics
- Setting Up the Physical Database: This section explains how you can set up a new RDBMS and configure it for your system.
- Managing Datasources: This section explains the purpose of defining datasources and how they are configured for a product.
- Configuring the System Administrator: This section provides information about the system administrator user.
- Properties of Primary User Stores: This section describes each of the properties used in the user-mgt.xml file for configuring the primary user store.

**Configuring a Read-Only LDAP or Active Directory User Store**

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. This file is shipped with user store manager configurations for all possible user store types (JDBC, read-only LDAP/Active Directory and read-write LDAP/Active Directory). The instructions given below explains how to configure a read-only LDAP or Active Directory as the primary user store for the WSO2 server.

**The default User Store**

The primary user store that is configured by default in the user-mgt.xml file is a JDBC user store, which reads/writes into the internal database of the product server. By default, the internal database is H2 for all WSO2 products excluding the Identity Server. This database is used by the Authorization Manager (for user authentication information) as well as the User Store Manager (for defining users and roles). In the case of the WSO2 Identity Server, the default user store is an LDAP (Apache DS) that is shipped with the product.

Note that the RDBMS used in the default configuration can remain as the database used for storing Authorization information.

Follow the given steps to configure a read-only LDAP/AD as the primary user store:

- Step 1: Setting up the read-only LDAP/AD user store manager
- Step 2: Updating the system administrator
- Step 3: Starting the server

Step 1: Setting up the read-only LDAP/AD user store manager

**Before you begin**

- If you create the `user-mgt.xml` file yourself, be sure to save it in the `<PRODUCT_HOME>/repository/conf/` directory.
- The class attribute for a read-only LDAP/Active Directory is `<UserStoreManager class="org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager">`.

1. Uncomment the following user store in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file: `<UserStoreManager class="org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager">`. Also, ensure
that you comment out the configurations for any other user stores in the same file.

2. Given below is a sample for the LDAP/AD user store configuration in read-only mode. You can change the values to match your LDAP/AD. For descriptions of the user store properties, see the related documentation.

```xml
<UserManager>
  <Realm>
    ...
    <UserStoreManager class="org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager">
      <Property name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDPTenantManager</Property>
      <Property name="ReadOnly">true</Property>
      <Property name="Disabled">false</Property>
      <Property name="MaxUserNameListLength">100</Property>
      <Property name="ConnectionURL">ldap://localhost:10389</Property>
      <Property name="ConnectionName">uid=admin,ou=system</Property>
      <Property name="PasswordHashMethod">PLAIN_TEXT</Property>
      <Property name="UserSearchBase">ou=system</Property>
      <Property name="UserNameListFilter">(objectClass=person)</Property>
      <Property name="UserNameSearchFilter">(&amp;(objectClass=person)(uid=?))</Property>
      <Property name="UserNameAttribute">uid</Property>
      <Property name="ReadGroups">true</Property>
      <Property name="GroupSearchBase">ou=system</Property>
      <Property name="GroupNameListFilter">(objectClass=groupOfNames)</Property>
      <Property name="GroupNameSearchFilter">(&amp;(objectClass=groupOfNames)(cn=?))</Property>
      <Property name="GroupNameAttribute">cn</Property>
      <Property name="SharedGroupNameAttribute">cn</Property>
      <Property name="SharedGroupSearchBase">ou=SharedGroups,dc=wso2,dc=org</Property>
      <Property name="SharedGroupNameListFilter">(objectClass=groupOfNames)</Property>
      <Property name="SharedTenantNameListFilter">(objectClass=organizationalUnit)</Property>
      <Property name="SharedTenantNameAttribute">ou</Property>
      <Property name="SharedTenantObjectClass">organizationalUnit</Property>
      <Property name="MembershipAttribute">member</Property>
      <Property name="UserRolesCacheEnabled">true</Property>
    </UserStoreManager>
  </Realm>
</UserManager>
```
<Property name="ReplaceEscapeCharactersAtUserLogin">true</Property>
<Property name="MaxRoleNameListLength">100</Property>
<Property name="SCIMEnabled">false</Property>
1. Update the connection details to match your user store. For example:

   <Property name="ConnectionURL">ldap://localhost:10389</Property>

   For Active Directory, the connectionURL should have the following format:

   <Property name="ConnectionURL">ldap://<AD_host-ip>:<AD_listen_port></Property>

   If you are using ldaps (secured) to connect to the Active Directory as shown below, you need to import the certificate of Active Directory to the client-truststore.jks of the WSO2 product. See the topic on configuring keystores for information on how to add certificates to the trust-store.

   <Property name="ConnectionURL">ldaps://10.100.1.100:636</Property>

2. Obtain a user who has permission to read all users/attributes and perform searches on the user store from your LDAP/Active Directory administrator. For example, if the privileged user is AdminLDAP and the password is 2010#Avrudu, update the following sections of the user store configuration as shown below. Note that this user does NOT have to be the system administrator that you define here.

   <Property name="UserNameAttribute">uid</Property>
   <Property name="UserNameSearchFilter">uid=AdminLDAP,ou=system</Property>
   <Property name="UserNameSearchFilter">%</Property>

3. Update <Property name="UserSearchBase"> with the directory name where the users are stored. When LDAP searches for users, it will start from this location of the directory.

   <Property name="UserSearchBase">ou=system</Property>

4. Set the attribute to use as the username, typically either cn or uid for LDAP. Ideally, <Property name="UserNameAttribute"> and <Property name="UserNameSearchFilter"> should refer to the same attribute. If you are not sure what attribute is available in your user store, check with your LDAP/Active Directory administrator.

   For example:
5. Set the ReadGroups property to 'true', if it should be allowed to read roles from this user store. When this property is 'true', you must also specify values for the GroupSearchBase, GroupSearchFilter and GroupNameAttribute properties. If the ReadGroups property is set to 'false', only Users can be read from the user store. You can set the configuration to read roles from the user store by reading the user/role mapping based on a membership (user list) or backlink attribute as shown below.

To read the user/role mapping based on a membership (This is used by the ApacheDirectory server and OpenLDAP):

- Enable the ReadGroups property.

```xml
<Property name="ReadGroups">true</Property>
```

- Set the GroupSearchBase property to the directory name where the Roles are stored. That is, the roles you create using the management console of your product will be stored in this directory location. Also, when LDAP searches for users, it will start from this location of the directory. For example:

```xml
<Property name="GroupSearchBase">ou=system,CN=Users,DC=wso2,DC=test</Property>
```

- Set the GroupSearchFilter and GroupNameAttributes. For example:

```xml
<Property name="GroupSearchFilter">(objectClass=groupOfNames)</Property>
<Property name="GroupNameAttribute">cn</Property>
```

- Set the MembershipAttribute property as shown below:

```xml
<Property name="MembershipAttribute">member</Property>
```

To read roles based on a backlink attribute, use the following code snippet instead of the above:
6. For Active Directory, you can use `<Property name="Referral">follow</Property>` to enable referrals within the user store. The AD user store may be partitioned into multiple domains. However, according to the use store configurations in the `user-mgt.xml` file, we are only connecting to one of the domains. Therefore, when a request for an object is received to the user store, the `<Property name="Referral">follow</Property>` property ensures that all the domains in the directory will be searched to locate the requested object.

Step 2: Updating the system administrator

The admin user is the super tenant that will be able to manage all other users, roles and permissions in the system by using the management console of the product. The `<Configuration>` section in the `user-mgt.xml` file contains the super admin information. Update this configuration for the read-only LDAP/AD as explained below.

```xml
<Configuration>
  <AddAdmin>False</AddAdmin>
  <AdminRole>admin</AdminRole>
  <AdminUser>
    <UserName>AdminSOA</UserName>
    <Password>XXXXXX</Password>
  </AdminUser>
  .........
</Configuration>
```

- `<AddAdmin>`: This should be set to ‘False’ as it will not be allowed to create users and roles in a read-only user store.
- `<AdminRole>`: The admin role you enter here should already exist in the read-only user store. Otherwise, you must enter an internal role, which will be saved to the internal database of the system when the system starts the first time.
- `<AdminUser>`: Since we are configuring a read-only LDAP as the primary user store, the user that should have admin permissions is required to be stored in the user store when you start the system for the first time. For example, say a valid username is AdminSOA. Update the `<AdminUser>` section of your configuration as shown above. You do not have to update the password element as it is already set in the user store.

See the related documentation for more information about the system administrator.

Step 3: Starting the server

Start your server and try to log in as the admin user you specified. The password is the admin user’s password in the LDAP server.

Related Topics
Configuring the System Administrator: This section provides information about the system administrator user.

Properties of Primary User Stores: This section describes each of the properties used in the user-mgt.xml file for configuring the primary user store.

Setting up Keystores: This topics explains how keystores are used in WSO2 products.

Configuring a Read-Write Active Directory User Store

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the user-mgt.xml file found in the <PRODUCT_HOME>/repository/conf/ directory. This topic provides instructions on how to configure an external Active Directory as the primary user store for the WSO2 server, by changing the default configuration given in this file.

The default User Store

The primary user store that is configured by default in the user-mgt.xml file is a JDBC user store, which reads/writes into the internal database of the product server. By default, the internal database is H2 for all WSO2 products excluding WSO2 Identity Server. This database is used by the Authorization Manager (for user authentication information) as well as the User Store Manager (for defining users and roles). In the case of the WSO2 Identity Server, the default user store is an LDAP (Apache DS) that is shipped with the product.

Note that the RDBMS used in the default configuration can remain as the database used for storing Authorization information.

Follow the given steps to configure an external Active Directory as the primary user store:

- **Step 1: Setting up the external AD user store manager**
- **Step 2: Updating the system administrator**
- **Step 3: Starting the server**

Step 1: Setting up the external AD user store manager

Before you begin

- If you create the user-mgt.xml file yourself, be sure to save it in the <PRODUCT_HOME>/repository/conf directory.
- The class attribute for an external AD is <UserStoreManager class="org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager">.

1. Enable the ActiveDirectoryUserStoreManager class in the user-mgt.xml file by uncommenting the code. When it is enabled, the user manager reads/writes into the Active Directory user store.

   Note that these configurations already exist in the user-mgt.xml file so you only need to uncomment them and make the appropriate adjustments. Also ensure that you comment out the configurations for other user stores which you are not using.

2. The default configuration for the external read/write user store in the user-mgt.xml file is as given below. Change the values according to your requirement. For more information about user store properties, see...
<UserStoreManager class="org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager">
    <Property name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDAPTenantManager</Property>
    <Property name="defaultRealmName">WSO2.ORG</Property>
    <Property name="Disabled">false</Property>
    <Property name="kdcEnabled">false</Property>
    <Property name="ConnectionURL">ldaps://10.100.1.100:636</Property>
    <Property name="ConnectionName">CN=admin,CN=Users,DC=WSO2,DC=Com</Property>
    <Property name="ConnectionPassword">A1b2c3d4</Property>
    <Property name="PasswordHashMethod">PLAIN_TEXT</Property>
    <Property name="UserSearchBase">CN=Users,DC=WSO2,DC=Com</Property>
    <Property name="UserNameAttribute">cn</Property>
    <Property name="isADLDSRole">false</Property>
    <Property name="isADLDSRole">false</Property>
    <Property name="UserNameListFilter">(objectClass=user)</Property>
    <Property name="UserNameSearchFilter">(&(objectClass=user)(cn=?)</Property>
    <Property name="UserNameJavaScriptRegEx">^[\S]{3,30}$</Property>
    <Property name="UserNameJavaRegEx">^[a-zA-Z0-9-_]{3,30}$</Property>
    <Property name="PasswordJavaScriptRegEx">^[\S]{5,30}$</Property>
    <Property name="RolenameJavaScriptRegEx">^[\S]{3,30}$</Property>
    <Property name="RolenameJavaRegEx">^[a-zA-Z0-9-_]{3,30}$</Property>
    <Property name="ReadGroups">true</Property>
    <Property name="WriteGroups">true</Property>
    <Property name="EmptyRolesAllowed">true</Property>
    <Property name="GroupSearchBase">CN=Users,DC=WSO2,DC=Com</Property>
    <Property name="GroupEntryObjectClass">group</Property>
    <Property name="GroupNameAttribute">cn</Property>
    <Property name="SharedGroupNameAttribute">cn</Property>
    <Property name="SharedGroupSearchBase">ou=SharedGroups,dc=wso2,dc=org</Property>
    <Property name="SharedGroupEntryObjectClass">groups</Property>
    <Property name="SharedTenantNameListFilter">{object=organizationalUnit}</Property>
<Property name="SharedTenantNameAttribute">ou</Property>
<Property name="SharedTenantObjectClass">organizationalUnit</Property>
<Property name="MembershipAttribute">member</Property>
<Property name="GroupNameListFilter">(objectcategory=group)</Property>
<Property name="GroupNameSearchFilter">(&amp;(objectClass=group)(cn=?))</Property>
<Property name="UserRolesCacheEnabled">true</Property>
<Property name="Referral">follow</Property>
<Property name="BackLinksEnabled">true</Property>
<Property name="MaxRoleNameListLength">100</Property>
When working with Active Directory;

- It is best to enable the **GetAllRolesOfUserEnabled** property in the **AuthorizationManager** as follows. See the documentation on configuring the **Authorization Manager** for more information.

```xml
<AuthorizationManager
class="org.wso2.carbon.user.core.authorization.JDBCAuthorizationManager">
  <Property
  name="AdminRoleManagementPermissions">/permission</Property>
  <Property
  name="AuthorizationCacheEnabled">true</Property>
  <Property
  name="GetAllRolesOfUserEnabled">true</Property>
</AuthorizationManager>
```

Although using the user store manager does not depend on this property, you must consider enabling this if there are any performance issues in your production environment. Enabling this property affects the performance when the user logs in. This depends on the users, roles and permissions.

- If you are using **ldaps** (secured) to connect to the Active Directory as shown in the example below, you need to import the certificate of Active Directory to the **client-truststore.jks** of the WSO2 product. See the topic on configuring **keystores** for information on how to add certificates to the trust-store.

```xml
<Property
  name="ConnectionURL">ldaps://10.100.1.100:636</Property>
```

3. For Active Directory, you can use `<Property name="Referral">follow</Property>` to enable referrals within the user store. The AD user store may be partitioned into multiple domains. However, according to the use store configurations in the **user-mgt.xml** file, we are only connecting to one of the domains. Therefore, when a request for an object is received to the user store, the `<Property name="Referral">follow</Property>` property ensures that all the domains in the directory will be searched to locate the requested object.

4. Set the attribute to use as the username, typically either `cn` or `uid` for LDAP. Ideally, `<Property name="UserNameAttribute">` and `<Property name="UserNameSearchFilter">` should refer to the same attribute. If you are not sure what attribute is available in your user store, check with your LDAP/Active Directory administrator.
5. Set the ReadGroups property to 'true', if it should be allowed to read roles from this user store. When this property is 'true', you must also specify values for the GroupSearchBase, GroupSearchFilter and GroupNameAttribute properties. If the ReadGroups property is set to 'false', only Users can be read from the user store. You can set the configuration to read roles from the user store by reading the user/role mapping based on a membership (user list) or backlink attribute as shown below.

To read the user/role mapping based on a membership (This is used by the ApacheDirectory server and OpenLDAP):

- Enable the ReadGroups property.

```xml
<Property name="ReadGroups">true</Property>
```

- Set the GroupSearchBase property to the directory name where the Roles are stored. That is, the roles you create using the management console of your product will be stored in this directory location. Also, when LDAP searches for users, it will start from this location of the directory. For example:

```xml
<Property name="GroupSearchBase">ou=system,CN=Users,DC=wso2,DC=test</Property>
```

- Set the GroupSearchFilter and GroupNameAttributes. For example:

```xml
<Property name="GroupSearchFilter">(objectClass=groupOfNames)</Property>
<Property name="GroupNameAttribute">cn</Property>
```

- Set the MembershipAttribute property as shown below:

```xml
<Property name="MembershipAttribute">member</Property>
```

To read roles based on a backlink attribute, use the following code snippet instead of the above:
Step 2: Updating the system administrator

The admin user is the super tenant that will be able to manage all other users, roles and permissions in the system by using the management console of the product. Therefore, the user that should have admin permissions is required to be stored in the user store when you start the system for the first time. Since the Active Directory user store can be written to, you have the option of creating a new admin user in the user store when you start the system for the first time. Alternatively, you can also use a user ID that already exists in the user store. See the related documentation for more information about the system administrator.

These two alternative configurations can be done as explained below.

- Find a valid user that already resides in the user store. For example, say a valid username is AdminSOA. Update the <AdminUser> section of your configuration as shown below. You do not have to update the <AdminPassword> element as it is already set in the user store.

```xml
<AddAdmin>False</AddAdmin>
<AdminRole>admin</AdminRole>
<AdminUser>
  <UserName>AdminSOA</UserName>
  <Password>XXXXXX</Password>
</AdminUser>
```

- Since the user store can be written to, you can add the super tenant user to the user store. Therefore, <AddAdmin> should be set to true as shown below.

```xml
<AddAdmin>true</AddAdmin>
<AdminRole>admin</AdminRole>
<AdminUser>
  <UserName>admin</UserName>
  <Password>admin</Password>
</AdminUser>
```

Step 3: Starting the server

Start your server and try to log in as the admin user you specified.

Related Topics

- Configuring the User Realm: This section provides information about setting up the system administrator and the authorization manager.
Setting up Keystores: This section explains how keystores are configured and used in a system.

Properties of Primary User Stores: This section describes each of the properties used in the user-mgt.xml file for configuring the primary user store.

**Configuring a Read-Write LDAP User Store**

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the user-mgt.xml file found in the `<PRODUCT_HOME>/repository/conf/` directory. This file is shipped with user store manager configurations for all possible user store types (JDBC, read-only LDAP/Active Directory, read-write LDAP and read-write Active directory). The instructions given below explains how to configure a read-write LDAP as the primary user store for the WSO2 server.

---

**The default User Store**

The primary user store that is configured by default in the user-mgt.xml file of WSO2 products is a JDBC user store, which reads/writes into the internal database of the product server. By default, the internal database is H2. This database is used by the Authorization Manager (for user authentication information) as well as the User Store Manager (for defining users and roles). In the case of the WSO2 Identity Server, the default user store is an LDAP (Apache DS) that is shipped with the product.

Note that the RDBMS used in the default configuration can remain as the database used for storing Authorization information.

Follow the given steps to configure a read-write LDAP as the primary user store:

- **Step 1: Setting up the read-write LDAP user store manager**
- **Step 2: Updating the system administrator**
- **Step 3: Starting the server**

**Step 1: Setting up the read-write LDAP user store manager**

Once the above points are made note of and completed, you can start configuring your external read-write LDAP as the primary user store.

1. Enable the `<ReadWriteLDAPUserStoreManager>` user store manager class in the user-mgt.xml file by uncommenting the relevant code. When it is enabled, the user manager reads/writes into the LDAP user store.

   Note that these configurations already exist in the user-mgt.xml file so you only need to uncomment them and make the appropriate adjustments. Also ensure that you comment out the configurations for other user stores that you are not using; in short, you can only configure one primary user store.
2. The default configuration for the external read/write user store in the `user-mgt.xml` file is as follows. You may have to change some of these values according to your requirements. For more information about user store properties, see the related documentation.

```xml
<UserStoreManager
    class="org.wso2.carbon.user.core.ldap.ReadWriteLDAPUserStoreManager">
    <Property
        name="TenantManager">org.wso2.carbon.user.core.tenant.CommonHybridLDAPTenantManager</Property>
    <Property
        name="ConnectionURL">ldap://localhost:${Ports.EmbeddedLDAP.LDAPServerPort}</Property>
    <Property
        name="ConnectionName">uid=admin,ou=system</Property>
    <Property
        name="PasswordHashMethod">SHA</Property>
    <Property
        name="UserNameListFilter">(objectClass=person)</Property>
    <Property
        name="UserNameObjectClass">wso2Person</Property>
    <Property
        name="UserNameSearchBase">ou=Users,dc=wso2,dc=org</Property>
    <Property
        name="UserNameSearchFilter">(&amp;(objectClass=person)(uid=?))</Property>
    <Property
        name="UserNameAttribute">uid</Property>
    <Property
        name="PasswordJavaScriptRegEx">\[\S\]{5,30}</Property>
    <Property
        name="UsernameJavaScriptRegEx">\[\S\]{3,30}</Property>
    <Property
        name="UsernameJavaRegEx">^[^!#$;\^*+=]\[\S\]{3,30}$</Property>
    <Property
        name="RolenameJavaRegEx">^[^!#$;\^*+=]\[\S\]{3,30}$</Property>
    <Property
        name="ReadGroups">true</Property>
    <Property
        name="WriteGroups">true</Property>
    <Property
        name="EmptyRolesAllowed">true</Property>
    <Property
        name="GroupSearchBase">ou=Groups,dc=wso2,dc=org</Property>
    <Property
        name="GroupNameListFilter">(objectClass=groupOfNames)</Property>
    <Property
        name="GroupNameObjectClass">groupOfNames</Property>
    <Property
        name="GroupNameSearchFilter">(&amp;(objectClass=groupOfNames)(cn=?))</Property>
    <Property
        name="GroupNameAttribute">cn</Property>
    <Property
        name="SharedGroupNameAttribute">cn</Property>
    <Property
        name="SharedGroupSearchBase">ou=SharedGroups,dc=wso2,dc=org</Property>
    <Property
        name="SharedGroupNameListFilter">(object=organizationalUnit)</Property>
```

<Property name="SharedTenantNameAttribute">ou</Property>
<Property name="SharedTenantObjectClass">organizationalUnit</Property>
<Property name="MembershipAttribute">member</Property>
<Property name="UserRolesCacheEnabled">true</Property>
1. To read and write to an LDAP user store, it is important to ensure that the `ReadGroups` and `WriteGroups` properties in the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file are set to `true`.

   ```xml
   <Property name="ReadGroups">true</Property>
   <Property name="WriteGroups">true</Property>
   ```

2. Set the attribute to use as the username, typically either `cn` or `uid` for LDAP. Ideally, `<Property name="UserNameAttribute">` and `<Property name="UserNameSearchFilter">` should refer to the same attribute. If you are not sure what attribute is available in your user store, check with your LDAP administrator.

   ```xml
   <Property name="UserNameAttribute">uid</Property>
   ```

3. Specify the following properties that are relevant to connecting to the LDAP in order to perform various tasks.

   ```xml
   <Property name="ConnectionURL">ldap://localhost:${Ports.EmbeddedLDAP.LDAPS erverPort}</Property>
   <Property name="ConnectionName">uid=admin,ou=system</Property>
   <Property name="ConnectionPassword">admin</Property>
   ```

4. Set the `ReadGroups` property to ‘true’, if it should be allowed to read roles from this user store. When this property is ‘true’, you must also specify values for the `GroupSearchBase`, `GroupSearchFilter` and `GroupNameAttribute` properties. If the `ReadGroups` property is set to ‘false’, only Users can be read from the user store. You can set the configuration to read roles from the user store by reading the user/role mapping based on a membership (user list) or backlink attribute as shown below.

   To read the user/role mapping based on a membership (This is used by the `ApacheDirectory` server and `OpenLDAP`):
   
   - Enable the `ReadGroups` property.

     ```xml
     <Property name="ReadGroups">true</Property>
     ```
   
   - Set the `GroupSearchBase` property to the directory name where the Roles are stored. That is, the roles you create using the management console of your product will be stored in this directory location. Also, when LDAP searches for users, it will start from this location of the directory. For example:
Step 2: Updating the system administrator

The **admin** user is the super tenant that will be able to manage all other users, roles and permissions in the system by using the management console of the product. Therefore, the user that should have admin permissions is required to be stored in the user store when you start the system for the first time. Since the LDAP user store can be written to, you have the option of creating a new admin user in the user store when you start the system for the first time. Alternatively, you can also use a user ID that already exists in the LDAP. See the related documentation for information about the system administrator.

These two alternative configurations can be done as explained below.

- If the user store is read-only, find a valid user that already resides in the user store. For example, say a valid username is AdminSOA. Update the `<AdminUser>` section of your configuration as shown below. You do not have to update the password element as it is already set in the user store.
If the user store can be written to, you can add the super tenant user to the user store. Therefore, `<AddAdmin>` should be set to true as shown below.

```
<AddAdmin>true</AddAdmin>
<AdminRole>admin</AdminRole>
<AdminUser>
  <UserName>admin</UserName>
  <Password>admin</Password>
</AdminUser>
```

Step 3: Starting the server

Start your server and try to log in as the admin user you specified in Step 2.

Related Topics

- **Configuring the System Administrator**: This section provides information about the system administrator user.
- **Properties of Primary User Stores**: This section describes each of the properties used in the `user-mgt.xml` file for configuring the primary user store.

**Configuring Secondary User Stores**

The default configurations of WSO2 products have a single, embedded user store (primary user store). If required, you can configure WSO2 products to connect to several secondary user stores as well. After configuration, users from different stores can log in and perform operations depending on their roles/permissions. You can also configure your own customized user stores and connect them with the products as secondary stores.

The topics below explain how to configure secondary user stores manually or using the management console:

- **Configuring using the management console**
- **Configuring manually**

Before you begin:

If you are setting up a database other than the default H2 that comes with the product to store user information, select the script relevant to your database type from the `<PRODUCT_HOME>/dbscripts` folder and run it on your database. It creates the necessary tables.

Configuring using the management console

1. Log in to the management console and click **User Store Management** sub menu under **Configure** menu.
2. The **User Store Management** page opens. Initially, there are no secondary user stores.
3. Click **Add Secondary User Store**.

4. In the User Store Manager Class list, select the type of user store you are creating:

<table>
<thead>
<tr>
<th>User store manager</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.wso2.carbon.user.core.ldap.ReadOnlyLDAPUserStoreManager</td>
<td>Use <code>ReadOnlyLDAPUserStoreManager</code> for read-only LDAP user stores.</td>
</tr>
<tr>
<td>org.wso2.carbon.user.core.ldap.ReadWriteLDAPUserStoreManager</td>
<td>Use <code>ReadWriteLDAPUserStoreManager</code> for both read and write operations.</td>
</tr>
<tr>
<td>org.wso2.carbon.user.core.ldap.ActiveDirectoryUserStoreManager</td>
<td>Use <code>ActiveDirectoryUserStoreManager</code> to configure Active Directory Domain Service (AD DS) or Active Directory Lightweight Directory Service (AD LDS). This can be used for read/write operations. If you need to use AD DS only read-only, you must use <code>ReadOnlyLDAPUserStoreManager</code>.</td>
</tr>
<tr>
<td>org.wso2.carbon.user.core.jdbc.JDBCUserStoreManager</td>
<td>Use <code>JDBCUserStoreManager</code> can be configured for read/write operations.</td>
</tr>
</tbody>
</table>

You can also populate this drop-down list with custom user store manager implementations by adding them to the server. A sample custom user store manager can be found in the repository.

5. Enter a unique domain name with no underscore (_) characters, and optionally enter a description for this user store.

6. Enter values for the properties, using the descriptions in the Descriptions column for guidance. The properties that appear vary based on the user store manager class you selected, and there may be additional properties in an Optional or Advanced section at the bottom of the screen. See the related topics for descriptions of user store properties.
7. Ensure that all the mandatory fields are filled and a valid domain name is given and click Add.

8. A message appears saying that the user stores are being added.

9. Refresh the page after a few seconds to check the status.

10. If the new user store is successfully added, it will appear in the User Store Management page.

11. After adding to the server, you can edit the properties of the new secondary user store and enable/disable it in a dynamic manner.

Configuring manually

By default, the configuration of the primary user store is saved in the `user-mgt.xml` file. When you create a secondary user store using the management console as explained above, its configuration is saved to an XML file with the same name as the domain name you specify. Alternatively, you can create this XML file manually and save it as follows:

- When you configure multiple user stores, you must give a unique domain name to each user store in the `<DomainName>` element. If you configure a user store without specifying a domain name, the server throws an exception at start up.
- If it is the configuration of a super tenant, save the secondary user store definitions in `<PRODUCT_HOM`
If it is a general tenant, save the configuration in <PRODUCT_HOME>/repository/tenants/<tenantid>/userstores directory.

The secondary user store configuration file must have the same name as the domain with an underscore (_) in place of the period. For example, if the domain is wso2.com, name the file as wso2_com.xml.

One file only contains the definition for one user store domain.

**Related topics**

- **Properties of User Stores**: This topic explains the properties that are used when defining user stores.

**Working with Properties of User Stores**

The following table provides descriptions of the key properties you use to configure primary user stores.

<table>
<thead>
<tr>
<th>Property name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MaxUserNameListLength</td>
<td>Controls the number of users listed in the user store of a WSO2 product. Setting this property to 0 displays all users.</td>
</tr>
<tr>
<td>ConnectionURL</td>
<td>Connection URL to the user store server. In the case of default LDAP in Carbon, the port is specified in the carbon.xml file, and a reference to that port is included in this configuration.</td>
</tr>
<tr>
<td>ConnectionName</td>
<td>The username used to connect to the database and perform various operations on the user store. The value you specify is used as the DN (Distinguished Name) attribute of the user. This property is mandatory.</td>
</tr>
<tr>
<td>DisplayNameAttribute</td>
<td>This is an optional property. The Display Name Attribute is the name by which users will be listed when you search for users in the management console (Go to Configuration -&gt; Users tab).</td>
</tr>
<tr>
<td>PasswordHashMethod</td>
<td>Password hash method to use when storing user entries in the user store.</td>
</tr>
<tr>
<td>UserNameListFilter</td>
<td>Filtering criteria for listing all the user entries in the user store. This query or filter is used when doing search operations on users. In this case, the search operation only provides the objects created from the specified class.</td>
</tr>
<tr>
<td>UserEntryObjectClass</td>
<td>Object class used to construct user entries. By default, it is a custom object class defined with the name wso2Person.</td>
</tr>
<tr>
<td>UserSearchBase</td>
<td>DN of the context or object under which the user entries are stored in the user store. When the user store searches for users, it will start from this location of the directory. Different databases have different search bases.</td>
</tr>
<tr>
<td>UserNameSearchFilter</td>
<td>Filtering criteria used to search for a particular user entry.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UserNameAttribute</td>
<td>The attribute used for uniquely identifying a user entry. Users can login using this attribute. The name of the attribute is considered as the username.</td>
</tr>
<tr>
<td>UsernameWithEmailJavaScriptRegEx</td>
<td>This property defines the JavaScript regular expression pattern when the property is set to true in configuration file. If you need to support both email as a user name and normal user names, you can set this property as follows.</td>
</tr>
<tr>
<td>UsernameJavaScriptRegEx</td>
<td>Policy that defines the password format.</td>
</tr>
<tr>
<td>UsernameJavaRegEx</td>
<td>The regular expression used by the front-end components for username validation.</td>
</tr>
<tr>
<td>UsernameJavaRegEx</td>
<td>A regular expression used to validate usernames. By default, strings have a length of 5 to 30. Only non-empty characters are allowed. You can provide ranges of alphabets, numbers and also ranges of ASCII values in the Regex.</td>
</tr>
<tr>
<td>RolenameJavaScriptRegEx</td>
<td>The regular expression used by the front-end components for role name validation.</td>
</tr>
<tr>
<td>RolenameJavaRegEx</td>
<td>A regular expression used to validate role names. By default, strings have a length of 5 to 30. Only non-empty characters are allowed.</td>
</tr>
<tr>
<td>ReadGroups</td>
<td>Specifies whether groups should be read from the user store. If this following group configurations are NOT mandatory: GroupSearchBase, GroupSearchFilter, GroupNameListFilter, GroupNameSearchFilter, GroupNameAttribute, MembershipAttribute, EmptyRolesAllowed.</td>
</tr>
<tr>
<td>WriteGroups</td>
<td>Specifies whether groups should be written to user store.</td>
</tr>
<tr>
<td>EmptyRolesAllowed</td>
<td>Specifies whether the underlying user store allows empty groups to be created. Usually LDAP servers do not allow you to create empty groups.</td>
</tr>
<tr>
<td>GroupSearchBase</td>
<td>DN of the context under which user entries are stored in the user store.</td>
</tr>
<tr>
<td>GroupSearchFilter</td>
<td>The query used to search for groups.</td>
</tr>
<tr>
<td>GroupNameListFilter</td>
<td>Filtering criteria for listing all the group entries in the user store. GroupSearchBase returns objects created from this class.</td>
</tr>
<tr>
<td>GroupEntryObjectClass</td>
<td>Object class used to construct group entries.</td>
</tr>
<tr>
<td>GroupNameSearchFilter</td>
<td>Filtering criteria used to search for a particular group entry.</td>
</tr>
<tr>
<td>GroupNameAttribute</td>
<td>Attribute used for uniquely identifying a user entry. This attribute is equivalent to the username.</td>
</tr>
<tr>
<td>MembershipAttribute</td>
<td>Attribute used to define members of groups.</td>
</tr>
<tr>
<td>UserRolesCacheEnabled</td>
<td>This is to indicate whether to cache the role list of a user. By default, those changes should be instantly reflected in the Carbon instance.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>UserDNPattern</td>
<td>(LDAP) The pattern for the user's DN, which can be defined to improve performance by specifying a pattern that is more specific to the user's DN.</td>
</tr>
<tr>
<td>ReplaceEscapeCharactersAtUserLogin</td>
<td>(LDAP) If the user name has special characters, it replaces them to validate the user logging in. Only \ and \ are identified as escape characters.</td>
</tr>
<tr>
<td>TenantManager</td>
<td>Includes the location of the tenant manager.</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>(LDAP and JDBC) Indicates whether the user store of this realm operates in read-only mode.</td>
</tr>
<tr>
<td>IsEmailUserName</td>
<td>(JDBC) Indicates whether the user's email is used as their username. If true, the user's email is used instead of the username provided during login.</td>
</tr>
<tr>
<td>DomainCalculation</td>
<td>(JDBC) Can be either default or custom (this applies when the realm operates in read-only mode).</td>
</tr>
<tr>
<td>PasswordDigest</td>
<td>(JDBC) Digesting algorithm of the password. Has values such as, F, SHA, MD5, etc.</td>
</tr>
<tr>
<td>StoreSaltedPassword</td>
<td>(JDBC) Indicates whether to salt the password.</td>
</tr>
<tr>
<td>UserNameUniqueAcrossTenants</td>
<td>(JDBC) An attribute used for multi-tenancy.</td>
</tr>
<tr>
<td>PasswordJavaRegEx</td>
<td>(LDAP and JDBC) A regular expression to validate passwords. By default, strings having a length between 5 to 30 with non-empty characters are allowed.</td>
</tr>
<tr>
<td>PasswordJavaScriptRegEx</td>
<td>The regular expression used by the front-end components for password validation.</td>
</tr>
<tr>
<td>UsernameJavaRegEx</td>
<td>A regular expression to validate usernames. By default, strings having a length between 5 to 30 with non-empty characters are allowed.</td>
</tr>
<tr>
<td>UsernameJavaScriptRegEx</td>
<td>The regular expression used by the front-end components for username validation.</td>
</tr>
<tr>
<td>RolenameJavaRegEx</td>
<td>A regular expression to validate role names. By default, strings having a length between 5 to 30 with non-empty characters are allowed.</td>
</tr>
<tr>
<td>RolenameJavaScriptRegEx</td>
<td>The regular expression used by the front-end components for rolename validation.</td>
</tr>
<tr>
<td>MultiTenantRealmConfigBuilder</td>
<td>Tenant Manager specific realm config parameter. Can be used to build different types of realms for the tenant.</td>
</tr>
<tr>
<td>SharedGroupEnabled</td>
<td>This property is used to enable/disable the shared role functionality.</td>
</tr>
<tr>
<td>SharedGroupSearchBase</td>
<td>Shared roles are created for other tenants to access under the mentioned DN.</td>
</tr>
<tr>
<td>SharedTenantObjectClass</td>
<td>Object class for the shared groups created.</td>
</tr>
<tr>
<td>SharedTenantNameAttribute</td>
<td>Name attribute for the shared group.</td>
</tr>
<tr>
<td>SharedTenantNameListFilter</td>
<td>This is currently not used.</td>
</tr>
</tbody>
</table>

## Managing Users, Roles and Permissions

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. The following documentation provides information about the parameters and settings used for managing users, roles, and permissions.
WSO2 products support the role-based authentication model where privileges of a user are based on the role attached. Each role is configured with zero or more permissions. Therefore, the set of permissions owned by a user is determined by the roles assigned to that user. If a user has several roles assigned, their permissions are added together.

By default, all WSO2 products have the following roles configured:

- **Admin** - Provides full access to all features and controls. By default, the admin user is assigned to both the Admin and the Everyone roles.
- **Internal/Everyone** - Every new user is assigned to this role by default. It does not include any permissions.
- **Internal/System** - This role is not visible in the Management Console.

More roles may be configured by default, depending on the type of features installed in your product. For example, in WSO2 Storage Server (which has the Cassandra feature and RSS Manager feature installed), the following roles will also be defined by default: Internal/Cassandra and Internal/RSSManager.

Permissions assigned to the Admin role cannot be modified.

Before you begin your configurations, note the following:

- Only system administrators or other users with Security level permissions can add, modify and remove users and roles. See the related topics for details on permissions.
- Your product has a primary user store where the users/roles that you create using the management console are stored by default. The default configurations for this user store are as follows. RegEx configuration s ensure that parameters like the length of a user name/password meet the requirements of the user store.

    ```
    PasswordJavaRegEx-------- ^[\S]{5,30}$
    PasswordJavaScriptRegEx-- ^[\S]{5,30}$
    UsernameJavaRegEx-------- ^~!#$;%*+={}\{3,30}$
    UsernameJavaScriptRegEx-- ^[\S]{3,30}$
    RolenameJavaRegEx-------- ^~!#$;%*+={}\{3,30}$
    RolenameJavaScriptRegEx-- ^[\S]{3,30}$
    ```

    When creating users/roles, if you enter a username, password etc. that does not conform to the RegEx configurations, the system throws an exception. You can either change the RegEx configuration or enter values that conform to the RegEx. If you change the default user store or set up a secondary user store, configure the RegEx accordingly under the user store manager configurations in `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file.

- The permission model of WSO2 products is hierarchical. Permissions can be assigned to a role in a fine-grained or a coarse-grained manner.

**Related Topics**

- Changing a Password
- Configuring Roles
- Configuring Users

**Changing a Password**

If you are a user with admin privileges, you can change your own password or reset another user’s password using
To change a user's password:

1. Log in to the management console of your product.
2. On the **Configure** tab, click **Users and Roles**.

3. To change your own password, click **Change My Password**, enter your current password and new password, and click **Change**.
4. If you are an admin user and need to change another user's password (such as if they have forgotten their current password and need you to reset it), do the following:
   1. Click **Users**.
   2. Find the user's account on the **Users** screen and click **Change Password** in the **Actions** column.
   3. Enter a new temporary password and click **Change**.
   4. Inform the user of their new temporary password and instruct them to log in and change it as soon as possible.

**Configuring Roles**

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. The instructions given in this topic explain how you can add and manager user roles from the management console.

Roles contain permissions for users to manage the server. You can create different roles with various combinations of permissions and assign them to a user or a group of users. Through the Management Console, you can also edit and delete an existing user role.

WSO2 supports the role-based authentication model where privileges of a user are based on the role to which it is attached. By default, WSO2 products come with the following roles:

- **Admin** - Provides full access to all features and controls. By default, the admin user is assigned to both the Admin and the Everyone roles.
- **Everyone** - Every new user is assigned to this role by default. It does not include any permissions by default.
- **System** - This role is not visible in the Management Console.

If a user has several assigned roles, their permissions are added together.

- Adding a user role
- Editing or deleting a role
- Updating role names

**Adding a user role**

Follow the instructions below to add a user role.

1. Go to the **Configure** tab in the management console and click **Users and Roles**.
2. Click **Roles**. This link is only visible to users with **Security** level permissions role. By default, the admin user has this permission enabled. See more information about permissions in the related topics.
3. Click **Add New Role**. The following screen will open:
3. Do the following:
   1. In the **Domain** list, specify the user store where you want to create this role. This list includes the primary user store and any other secondary user stores that are configured for your product. See the related topics for information on configuring user stores.
   2. Enter a unique name for this role.
   3. Click **Next**.

4. Select the permissions that you want users with this role to have. See more information about permissions in the related topics.
   Note that when you assign this role to a user, you can override the role's permissions and customize them for the user.

5. Select the existing users to whom this role should be assigned. You can also assign this role to users later, but if you are creating this role in an external user store that does not allow empty roles, you must assign it to at least one user. You can search for a user by name, or view all users by entering * in the search field.

6. Click **Finish**.

The role is created and is listed on the **Roles** page. You can now edit the role as needed.

**Editing or deleting a role**

If you need to do modifications to a role, select the domain (user store) where the role resides, and then use the relevant links in the **Actions** column on the **Roles** screen:

- Rename the role
- Change the default permissions associated with this role
- Assign this role to users
- View the users who are assigned this role
- Delete the role if you no longer need it

If the role is in an external user store to which you are connected in read-only mode, you will be able to view the existing roles but not edit or delete them. However, you can still create new editable roles.

**Updating role names**

If you need to do modifications to the role names, you need to do one of the following:

- Update before the first startup (recommended)
- Update after the product is used for sometime

Update before the first startup (recommended)
The default role names (admin and everyone) can be changed before starting the WSO2 product by editing `<PRODUCT_HOME>/repository/conf/user-mgt.xml`. See more information about configuring the system administrator.

```xml
<Configuration>
  <AdminRole>admin</AdminRole>
  <AdminUser>
    <UserName>admin</UserName>
    <Password>admin</Password>
  </AdminUser>
  <EveryOneRoleName>everyone</EveryOneRoleName> <!-- By default users in this role sees the registry root -->
  <Property name="dataSource">jdbc/WSO2CarbonDB</Property>
  <Property name="MultiTenantRealmConfigBuilder">org.wso2.carbon.user.core.config.multitenancy.SimpleRealmConfigBuilder</Property>
</Configuration>
```

The following are the changes that need to be made in the configurations above:

- Change `<AdminRole>admin</AdminRole>` to `<AdminRole>administrator</AdminRole>.
- Change `<EveryOneRoleName>everyone</EveryOneRoleName>` to `<EveryOneRoleName>Your role</EveryOneRoleName>.

Update after the product is used for sometime

You do not have to do this when updating before the first startup. The following steps guide you through updating the role names:

1. Do the configuration changes indicated in the above section.
2. You need to do the following user store level changes for existing users if you have changed the role names as mentioned earlier.
   - If you are connected to JDBCUserStoreManager you need to update the UM_USER_ROLE table with the existing users after changing the admin and everyone role names. Also if you have changed the permission of everyone role, the UM_ROLE_PERMISSION has to be updated with the permissions to the new role.

   The schema can be located by referring to the data source defined in the user-mgt.xml file. The data source definition can be found under `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml`.

   - If you are connected to ReadWriteLdapUserStoreManager, you need to populate the members of the previous admin role to the new role under the Groups.
3. After the changes, restart the server.

Related Topics

- **Configuring the System Administrator**: This section describes how the system administrator user and role is set up and configured.
- **Configuring User Stores**: This section explains how user stores (which are repositories storing information about Users and Roles) are set up and configured.
Configuring Users

User management functionality is provided by default in all WSO2 Carbon-based products and is configured in the `user-mgt.xml` file found in the `<PRODUCT_HOME>/repository/conf/` directory. The instructions given in this topic explain how you can add and manage users from the management console.

To enable users to log into the management console, you create user accounts and assign them roles, which are sets of permissions. You can add individual users or import users in bulk.

- Adding a new user and assigning roles
- Importing users
- Customizing the user's roles and permissions
- Customizing a user's profile
- Deleting an existing user
- Related Topics

Adding a new user and assigning roles

Follow the instructions below to add a new user account and configure its role.

1. Go to the Configure tab in the management console and click Users and Roles.
2. Click Users. This link is only visible to users with the Admin role.
3. Click Add New User. The following screen will open:

   ![Add User Screen](image)

   **Step 1: Enter user name**

   **Enter user name**

<table>
<thead>
<tr>
<th>Domain</th>
<th>PRIMARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Name*</td>
<td></td>
</tr>
<tr>
<td>Password*</td>
<td></td>
</tr>
<tr>
<td>Password Repeat*</td>
<td></td>
</tr>
</tbody>
</table>

   4. Do the following:
      1. In the Domain list, specify the user store where you want to create this user account. This list includes the primary user store and any other secondary user stores that are configured for your product. See the related topics for information on configuring user stores.
      2. Enter a unique user name and the password that the person will use to log in. By default, the password must be at least five characters and should have at least one character with a capital letter, characters, numbers and special characters.
      3. Click Next.
      4. Optionally, select the role(s) you want this user to have. If you have many roles in your system, you can search for them by name.
      5. Click Finish.

   A new user account is created with the specified roles and is listed on the Users page.
Importing users

In addition to manually adding individual users, you can import multiple users in bulk if you have exported them to a comma-separated values (.csv) file or Microsoft Excel (.xls) file. It is possible to import the **username** and **password** directly from the CSV/Excel to the product. Other user attributes can be imported only if **claim URLs are defined for such attributes** in the product. For example, consider that you have claim URLs defined for your product as shown below. These will allow you to import the user's **email address**, **full name**, **last name**, **given name** and **role** in addition to the **username** and **password**.

The username, password and other attributes (claim URLs) that you import should be given in a CSV file as shown below. Note that the first line of the file will not be imported considering that it is not a username.

<table>
<thead>
<tr>
<th>Username, Password, Claims</th>
</tr>
</thead>
</table>

This is only supported if you have configured your user store as a JDBCUserStoreManager. See the related topics for information on how JDBC user stores are configured. It is recommended to upload a maximum of 500,000 users at a time. If you need to upload more users, you can upload them in separate batches of 500,000 each. You can also specify the size of the file that you can upload in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file using the `TotalFileSizeLimit` element. This value is in MB.

```
<TotalFileSizeLimit>100</TotalFileSizeLimit>
```

1. On the **Users** screen, click **Bulk Import Users**.
2. Browse and select the file that contains the user data.
3. Specify a default password to assign to all the users you are importing and click **Finish**. This password is valid for only 24 hours, so you should inform your users that they must log in and change their password within 24 hours.

Customizing the user's roles and permissions

Each role specifies a set of permissions that the user will have when assigned to that role. After creating a user, you can assign and remove roles for that user by clicking **Assign Roles** in the **Actions** column. To see which users a role is already assigned to, click **View Users** next to the role.

You can also customize which permissions apply to this user by clicking **View Roles** in the **Actions** column of the **Users** screen and then selecting the permissions from each role that you want this user to have. See the related topics for information about permissions.

Customizing a user's profile

Each individual user has a profile that can be updated to include various details. To do this, click **User Profile** on the **Users** screen. Make the changes required and click **Update**. You can also add multiple profiles for a user.

**Note:** You can only add new profiles if you are connected to a JDBC user store. You also need to have
Do the following in order to add new profiles.

1. On the **Configure** tab in the Management Console, click **Users and Roles**.
2. Click **Users**. This link is only visible to users with the Admin role.
3. Click the **User Profile** link.
4. You can add multiple profiles using the **Add New Profile** link and create any number of profiles for your user as long as the user is located in a JDBC user store.

Deleting an existing user

Follow the instructions below to delete a user.

1. Go to the **Configure** tab in the management console and click **Users and Roles**.
2. Click **Users**. This link is only visible to users with **User Management** level permissions. See the related topics for more information about permissions.
3. In the **Users** list, click **Delete** next to the user you want to delete, and then click **Yes** to confirm the operation.

Related Topics

- **Configuring the System Administrator**: This section describes how the system administrator user and role is set up and configured.
- **Configuring User Stores**: This section explains how user stores (which are repositories storing information about Users and Roles) are set up and configured.

Working with Features

Each enterprise middleware product is a collection of reusable software units called features. Similarly, WSO2 Application Server (AS) consists of a collection of features where a single feature is a list of components and/or other features.

Provisioning software is the act of placing an individual software application or a complete software stack onto a target system. Provisioning WSO2 AS is installing/updating/uninstalling features to/from the Carbon base platform on top of which the entire WSO2 product stack is developed. It is also possible to easily revert to a previous feature configuration using the provisioning support.

Features can be easily installed to any WSO2 product using the Feature Manager that comes with the product. WSO2 Feature Manager is powered by Equinox P2 and allows you to connect to a remote or local P2 repository and get any feature installed into the product's runtime.

P2 can be used as a provisioning platform for any OSGi-based application. P2 has enabled easy provisioning capabilities in WSO2 Carbon, thereby increasing the user friendliness in building customized SOA products using the Carbon platform. Users can download the WSO2 Carbon framework or any other WSO2 product and extend them by simply installing various features. The WSO2 Feature Manager provides a convenient user interface to perform common provisioning operations and related repository management functions.

See also

- Managing the Feature Repository
- Installing Features
- Uninstalling Features
- Converting the Product into a Back-end/Front-end Server
Managing the Feature Repository

- Accessing the available repository list
- Adding a repository
- Editing a repository
- Enabling/disabling a repository
- Deleting a repository

**Accessing the available repository list**

1. Log in to the product's management console.
2. On the **Configure** menu, click **Features**. The **Feature Management** page appears.
3. Click on the **Repository Management** tab.
   
   The **Manage Repositories** page will appear, which allows you to view or modify the feature management settings.

**Adding a repository**

1. Access the available repositories list.
2. Click **Add Repository**.
3. Provide a convenient name for the repository being added.
4. Enter the repository location using one of the following methods:
   - **URL** - this option is used when you are adding an external repository.
     - Select the **URL** option.
     - Enter the Equinox P2 repository URL.

   The official WSO2 Carbon features are available in Equinox P2 repository at: [http://dist.wso2.org/p2/carbon/releases](http://dist.wso2.org/p2/carbon/releases). WSO2 Application Server 5.2.1 is based on WSO2 Carbon 4.2.0. Therefore, the P2 feature repository 4.2.0 is the earliest, compatible version.

   Equinox P2 provides provisioning technology for OSGi-based applications such as WSO2 Carbon.

   - **Location** - this option is used when you are adding a repository that you have downloaded to your computer.
     - Select the **Local** option.
     - Enter the directory path of the repository on your local machine.

**Editing a repository**
1. **Access the available repositories list.**
2. Click on the **Edit** link, corresponding to the repository that you wish to modify. The **Edit Existing Repository** page appears.
3. Edit the repository name and click **Save**.

This page allows you to change the repository name only. If you want to change the URL, you need to remove the existing repository and then add the new repository.

**Enabling/disabling a repository**

1. **Access the available repositories list.**
2. Click on the **Enable** or **Disable** link associated with the repository.

By default, all the repositories are enabled. When you perform a provisioning operation, metadata and artifacts are searched only from the enabled repositories.

**Deleting a repository**

1. Click the **Remove** link, respective to the repository to be removed.
2. Click **Yes**, to confirm your deletion request.

**Installing Features**

The manual way of provisioning Carbon is dropping bundles and configuration files that belong to a feature. This method is not recommended since it can cause errors. Besides, finding the exact set of components and configuring files is a complex task. Components have inter-dependencies with other components. Some components depend on specific versions of other components.

As a result, to overcome these issues WSO2 has the **Equinox P2** integrated with Carbon. It enables user-friendly provisioning capabilities by allowing users to download WSO2 Carbon or any other WSO2 product and simply extend them by installing various features.

If you are on Windows, be sure to point the `-Dcarbon.home property` in the product's startup script (`wso2server.bat`) to the product's distribution home (e.g., `-Dcarbon.home=C:\Users\VM\Desktop\wso2as -5.2.1`). Then, restart the server. If not, you might not be able to install features through the management console.

Follow the instructions below to install new features to any product of the WSO2 products stack:

1. Log into the product's Management Console.
2. On the **Configure** menu, click **Features**. The **Feature Management** page will appear.
3. Click **Available Features**.
4. Select a relevant repository.
   - If no repositories have been added or the required repository is not available, add a new repository. For more information see, **Adding a Repository**.
5. Some repositories contain multiple versions of features. If you are only interested in the latest versions, select the **Show only the latest versions** option.
6. A feature category is a logical grouping of the features that constitute a particular Carbon based product. Select the **Group features by category** option enables you to easily view and select the entire list of features of a particular product at once. For more information, see **Feature Categorization**.

---

Copyright © WSO2 Inc. 2014

327
If you do not select this option when looking for features, you will see an uncategorized, flat feature list from which individual features can be selected separately.

7. Click **Find Features**. The available features will be listed.
8. Select the features you wish to install.

```
To find a particular feature, you can use the search function. Enter the name of a feature (or a part of the name) and press **Enter**.
```

```
This search will return only available features; excluding the ones already installed.
```

9. Click **Install**.
10. Verify the feature to be installed and click **Next**.
11. Read and accept the terms of the license agreement.
12. Click **Next**.
    The installation process starts. It may take a few minutes to download the necessary components.
13. Once the installation process is complete, click **Finish**.
14. Restart the server for the changes to take effect.
    Based on the newly added features, you will be able to see the additional functionalities.

**Viewing feature information**
When features of a repository are loaded, you can install them or view the corresponding feature details.

To view feature details:

1. Click on the respective "+" link to navigate to the feature.
2. Click on the corresponding More Info link. The Feature Information page appears with the following information:
   - Name
   - Identifier
   - Version
   - Provider
   - Description
   - Copyright
   - License Agreement

Feature Categorization

Feature repository provided by WSO2 contains a number of features that can be installed into WSO2 Carbon based products. The logically related features have been categorized in the feature repository, to make it easier for the user to search and install the required features more effectively. These feature categories are grouped under products. As illustrated in the following screenshot, there are high level feature categories such as Application Server and Governance Registry. Under these product based feature categories, there are another set of feature categories based on the product features. The user has the option of installing either the whole product or only the required features of a product. The feature categorization can be seen on the Available Features page after selecting/adding the repository and thereafter, searching for features that are grouped by category. For more information, see Installing Features.

![Feature Categorization Screenshot](image)

Application Server specific features
The following table contains the Application Server specific features categorization:

**Y** - Available by default in the Application Server.

**N** - Not available by default in the Application Server.

<table>
<thead>
<tr>
<th>Feature Categorization</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Application Server</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Application Hosting</strong></td>
<td>Features required for webapp and web services hosting.</td>
</tr>
<tr>
<td>• Axis2 Service Hosting</td>
<td>Y</td>
</tr>
<tr>
<td>• Data Service Hosting</td>
<td>Y</td>
</tr>
<tr>
<td>• Doc Request Processor</td>
<td>N</td>
</tr>
<tr>
<td>• Gadget Request Processor</td>
<td>N</td>
</tr>
<tr>
<td>• Jaggery aggregate</td>
<td>Y</td>
</tr>
<tr>
<td>• Jar Service Hosting</td>
<td>Y</td>
</tr>
<tr>
<td>• Javascript Service Hosting</td>
<td>Y</td>
</tr>
<tr>
<td>• Javascript Stub Generator</td>
<td>Y</td>
</tr>
<tr>
<td>• Javascript Web Scraping</td>
<td>Y</td>
</tr>
<tr>
<td>• Mashup Application Deployer</td>
<td>Y</td>
</tr>
<tr>
<td>• New Datasource Management</td>
<td>Y</td>
</tr>
<tr>
<td>• Source Request Processor</td>
<td>N</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Spring Service Hosting</td>
<td>Y</td>
</tr>
<tr>
<td>Webapp Management</td>
<td>Y</td>
</tr>
<tr>
<td>CXF Runtime Environment</td>
<td>Y</td>
</tr>
<tr>
<td><strong>Application Management</strong></td>
<td>Features required for artifact management.</td>
</tr>
<tr>
<td>Application Management</td>
<td>Y</td>
</tr>
<tr>
<td>AppServer - Utils Features</td>
<td></td>
</tr>
<tr>
<td>Axis2 Repository Management</td>
<td>Y</td>
</tr>
<tr>
<td>JAX-WS/JAX-RS Webapp Management</td>
<td>Y</td>
</tr>
<tr>
<td>Service Management</td>
<td>Y</td>
</tr>
<tr>
<td>Webapp Application Deployer</td>
<td>Y</td>
</tr>
<tr>
<td>Webapp Application Management</td>
<td>Y</td>
</tr>
<tr>
<td><strong>AppServer Styles</strong></td>
<td>Application Server UI theme features.</td>
</tr>
<tr>
<td>AppServer - Style Features</td>
<td>Y</td>
</tr>
<tr>
<td><strong>AppServer Tools</strong></td>
<td>Features that provide a useful set of tools for web services.</td>
</tr>
<tr>
<td>Try it</td>
<td>Y</td>
</tr>
<tr>
<td>WSDL Tools</td>
<td>Y</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td>Miscellaneous features.</td>
</tr>
</tbody>
</table>
### Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coordination Core</td>
<td>N</td>
</tr>
<tr>
<td>Deployment Synchronizer</td>
<td>Y</td>
</tr>
<tr>
<td>JMS Transport Core</td>
<td>Y</td>
</tr>
<tr>
<td>Mail Transport Core</td>
<td>Y</td>
</tr>
<tr>
<td>Mex Module</td>
<td>Y</td>
</tr>
<tr>
<td>Reporting</td>
<td>N</td>
</tr>
<tr>
<td>Tasks</td>
<td>Y</td>
</tr>
<tr>
<td>Transaction Manager</td>
<td>Y</td>
</tr>
<tr>
<td>Transport Management</td>
<td>Y</td>
</tr>
<tr>
<td>WS-Discovery Admin UI</td>
<td>Y</td>
</tr>
<tr>
<td>WS-Discovery Core</td>
<td>Y</td>
</tr>
<tr>
<td>Xfer Module</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Monitoring**

Features for server and deployment artifacts monitoring.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAM Webapp Stats Data Agent Aggregate</td>
<td>Y</td>
</tr>
<tr>
<td>Data Bridge - Data Publisher Aggregate</td>
<td>Y</td>
</tr>
<tr>
<td>Logging Management</td>
<td>Y</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>Message Flows</td>
<td></td>
</tr>
<tr>
<td>SOAP Tracer</td>
<td></td>
</tr>
<tr>
<td>Statistics</td>
<td></td>
</tr>
<tr>
<td>Registry/Repository</td>
<td>Registry features.</td>
</tr>
<tr>
<td>Registry Core</td>
<td></td>
</tr>
<tr>
<td>Registry Resource Properties</td>
<td></td>
</tr>
<tr>
<td>Registry UI Menu</td>
<td></td>
</tr>
<tr>
<td>Single Sign On(SAML2)</td>
<td>Features that enables the use of Single-Sign-On to log in to AS.</td>
</tr>
<tr>
<td>Identity Provider Management Aggregated</td>
<td>N</td>
</tr>
<tr>
<td>Tenancy Management</td>
<td>Multi-tenant features.</td>
</tr>
<tr>
<td>Stratos AppServer - Dashboard UI Features</td>
<td>Y</td>
</tr>
<tr>
<td>Stratos AppServer - Dashboard UI Features</td>
<td>Y</td>
</tr>
<tr>
<td>Stratos - Common Composite</td>
<td>Y</td>
</tr>
<tr>
<td>Stratos - Deployment Features</td>
<td>Y</td>
</tr>
<tr>
<td>Stratos - Throttling Agent</td>
<td>Y</td>
</tr>
<tr>
<td>Stratos - Usage Agent</td>
<td>Y</td>
</tr>
<tr>
<td>User Management</td>
<td>User Management features to configure users, roles etc.</td>
</tr>
<tr>
<td>Feature</td>
<td>Status</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------</td>
</tr>
<tr>
<td>Entitlement Servlet Filter</td>
<td>Y</td>
</tr>
<tr>
<td>Synapse</td>
<td>Y</td>
</tr>
<tr>
<td>User Management</td>
<td>Y</td>
</tr>
<tr>
<td>Application Server Runtimes (Optional)</td>
<td>A set of runtimes that can be installed in Application Server.</td>
</tr>
<tr>
<td>J2EE Runtime Environment</td>
<td>N</td>
</tr>
<tr>
<td>CXF Runtime Environment</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Uninstalling Features**

1. Log in to the product’s Management Console.
3. Click Installed Features. The Installed Features page allows you to browse through the list of installed features.
4. Select the features that you need to uninstall. If you wish to uninstall all the features, select the Select all in this page option.
5. Click Uninstall. A page will appear containing details of the features to be uninstalled.

6. Verify the information and click Next. If the uninstallation is successful, a success message appears.
7. Click Finish and restart the server to apply the changes.

**Converting the Product into a Back-end/Front-end Server**

- Converting to a back-end server
- Converting to a front-end server

WSO2 Carbon supports back-end front-end separation, where you can manage multiple backend servers using a single front-end server. You can convert a given product either to a backend server or to a front-end server. This conversion can be performed using the Feature Management user interface.

**Converting to a back-end server**

If you want to get only a back-end server, you have to uninstall all the front-end features as mentioned below:

1. Log in to the product's Management Console.
3. Click Installed Features. The Installed Features page appears.
4. Select "Front-end", from the Filtered by option. This will list all the front-end features that are currently installed in the system.
5. Press Uninstall, to uninstall all the front-end features in the current installation.
Converting to a front-end server

If you want to get only a front-end server, you have to uninstall all the back-end features as mentioned below:

1. Log in to the product’s Management Console.
2. On the Configure menu, click Features.
   The Feature Management page will appear.
3. Click Installed Features.
   The Installed Features page appears.
   This will list all the back-end features that are currently installed in the system.
5. Press Uninstall, to uninstall all the back-end features in the current installation.

Recovering from Unsuccessful Feature Installation

After installing features, if you encounter server issues or startup failures, you can revert the current configuration by restoring a previous one using either the management console or the command line. The latter is recommended if you cannot start the server.

Use the following steps to check your feature installation history and revert the server back to a previous installation. In this recovery process, some features might get installed and some uninstalled.

- Reverting using the management console
- Reverting using the command line

Reverting using the management console

1. Log in to the management console.
2. On the Configure menu, click Features.
   The Feature Management page appears.
3. Click on the Installation History tab.
   The Installation History page appears.

4. Click on the configuration that you wish to revert to.
   The actions that take place when reverting to the selected configuration is listed.
5. Click **Revert** to revert the current configuration to a previous configuration.

**Reverting using the command line**

If you cannot start the server after an unsuccessful feature installation, use the following steps to revert to a previous installation:

1. Start your product with the `-DosgiConsole` system property.
2. Once the server is started, type the following command:
   
   ```
   osgi> getInstallationHistory
   ```

   You will get the following list of states:

   
   ```
   1376883697814 August 19, 2013 at 09:11:37 IST
   1376883697957 August 19, 2013 at 09:11:37 IST
   1376883700725 August 19, 2013 at 09:11:40 IST
   1376883701385 August 19, 2013 at 09:11:41 IST
   1376883704884 August 19, 2013 at 09:11:44 IST
   1376883712770 August 19, 2013 at 09:11:52 IST
   1376883715952 August 19, 2013 at 09:11:55 IST
   1376883743493 August 19, 2013 at 09:12:23 IST
   1376933879416 August 19, 2013 at 23:07:59 IST
   1376940017503 August 20, 2013 at 00:50:17 IST
   ```

3. Check what features are installed and uninstalled in a given state, by entering the following command:
   
   ```
   osgi> getInstallationHistory <timestamp>
   ```

   For example:
   
   ```
   osgi> getInstallationHistory 1376933879416
   ```

   The output will be as follows:
-- Installed features in this configuration

-- Uninstalled features in this configuration
WSO2 Carbon - Service Management Feature 4.2.0
WSO2 Stratos - Deployment Features 2.2.0
WSO2 Stratos - Common Composite Feature 2.2.0
WSO2 Stratos - Usage Agent Feature 2.2.0
WSO2 Stratos - Throttling Agent Feature 2.2.0
WSO2 Stratos AppServer - Dashboard UI Features 2.2.0
WSO2 Stratos AppServer - Dashboard UI Features 2.2.0

4. Decide to which state you need to revert the system to, and thereafter use the following command:

`osgi> revert <timestamp>`

For example:

`osgi> revert 1376933879416`

The output will be as follows:

Successfully reverted to 1376933879416
Changes will get applied once you restart the server.

Monitoring the Application Server

This chapter contains the following information:

- Collecting Data Statistics of Services
- Monitoring Logs
- Monitoring Performance Statistics
- Monitoring SOAP Messages
- Monitoring Message Flows
- Monitoring Messages with TCPMon
- JMX-Based Monitoring

Note that some of the above statistics may not be available in some WSO2 products, depending on the availability of the relevant feature in its distribution. If you want a particular functionality that is not bundled with the distribution by default, you need to install the relevant feature. This can be done via the Feature Management page in the management console, by navigating to the Configure menu and clicking Features.

Collecting Data Statistics of Services

Service data analysis is crucial in understanding and optimizing their effective usage. For this reason, functionality to capture and collect statistics related to service data is integrated by default into service-hosting WSO2 products.
such as the WSO2 Application Server (AS), WSO2 Data Services Server (DSS), WSO2 Business Process Server (BPS), WSO2 Business Rules Server (BRS) etc. Once captured, these statistics are transferred to a data analysis product like WSO2 Business Activity Monitor (BAM) or WSO2 Complex Event Processor for processing and deriving business-critical information.

WSO2 BAM receives data events via its Thrift API. This uses a binary protocol and enables fast data transmission between the service-hosting product and BAM server. BAM's Service Data Agent can be configured early so that the statistics to be extracted from service data can be pre-defined. In the configuration panel, the user can specify the BAM server (Thrift server) related information and properties to be extracted from the service of the service-hosting product. The user should define the Event Stream related parameters uniquely identified by a name and a version.

When using data publisher API to publish data in a periodic manner to WSO2 BAM/CEP, the eviction time and eviction idle time for the connections should be higher than the periodic interval. This is required to re-use the created socket connections from the pool, avoiding closure of it and creation of new connections. The default eviction period is 5.5 seconds (5500 milliseconds). If you are publishing events in a periodic interval as more than 5.5s, you need to tune the `<secureEvictionTimePeriod>` parameter accordingly, in the `<PRODUCT_HOME>/repository/conf/data-bridge/thrift-agent-config.xml` file of the agent in the client side, by increasing this default value.

See the following topics for details:

- Using on-premise deployment of WSO2 AS
- Using Cloud deployment of WSO2 AS

**Using on-premise deployment of WSO2 AS**

If you are using an on-premise deployment of WSO2 AS, the service data publishing capability is provided by the following feature in the WSO2 feature repository:

**Name**: WSO2 Carbon - BAM Service Data Agent Aggregate

**Identifier**: org.wso2.carbon.bam.service.agent.feature.group

If it is not included in your product by default, you can add it by installing the above feature using the instructions given in section, Feature Management.

Follow the instructions below to setup service data publishing in a running Carbon instance.

1. Before starting AS, enable service data publishing as explained below:

   Go to `{AS_HOME}/repository/conf/etc/` and open `bam.xml`. Look for `<ServiceDataPublishing>disable</ServiceDataPublishing>` and enable it.

2. Start the Application Server. For more information, see Starting the server.

3. Log on to the product's Management Console select "Configure -> Service Data Publishing."
4. The "Service Data Publisher Configuration" window opens. Fill it appropriately. For example,

Service Data Publisher Configuration

Enable Service stats: Tick to get the activity statistics.
Enable Activity Service: This enables message tracing with BAM, CEP or other data analysis servers. Typically, a request is sent to an endpoint after being passed through some intermediate stages. Messaging tracing enables to track each messages and visualize the message body in each of these intermediate stages.
**Note**

**Enable Activity Service** feature will be implemented from the next major release of BAM.

**Stream Definition Configuration:**

- **Stream Name**: Stream Name can be any string with alpha-numeric characters
- **Version**: Stream Version distinguishes different streams with the same Stream Name. Default version should be 1.0.0.
- **Nick Name**: This is a user preferred nick name to the Stream Name in alpha-numeric characters
- **Description**: A description describing about the particular stream defined by Stream Name, Stream Version pair. Description should also consists of alpha-numeric characters.

**BAM Credentials:**

- **BAM URL**: Enter the IP address of the BAM server. And the port should be the thrift port (ex: tcp://127.0.0.1:7611). You can add a comma-separated list of server URLs to ensure that load is distributed among many servers and to minimize effects of a failure of one server node.

  See the documentation on [setting up multi receiver and load balancing data agent](#), for details on how BAM handles load distribution using multiple servers.

- **Username**: Put the BAM Thrift log-in user name. Default value is **admin**.
- **Password**: Put the BAM Thrift log-in password. Default value is **admin**.

  **Properties**: A BAM deployment can receive events from different data agents. In order to uniquely identify events triggering from one agent, a property can be associated with it. This feature is particularly useful in clustered environments. Click **Update** once done.

  The service-hosting server is now ready to collect and send service data to any data analysis server like the BAM and CEP. Specific instructions to receive this data and analyze it can be found in the respective documentation of the data analysis servers.

**Using Cloud deployment of WSO2 AS**

In a cloud deployment of WSO2 AS, it is not required to enable data publishing to BAM using the 'BAM Service Data Agent Aggregate' feature as explained above. Instead, you can simply add the BAM url to the `<PRODUCT_HOME>/repository/conf/carbon.xml` file as shown below.

```
<BamServerURL>https://bamhost:bamport/services/</BamServerURL>
```

Note that cloud deployment is enabled by the following setting in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file:

```
<IsCloudDeployment>true</IsCloudDeployment>
```

**Monitoring Logs**
The following topics describe how logs can be managed using the management console:

- **Application Logs**
- **System Logs**

**Application Logs**

Application logs are where events invoked by an application or a program running in a system are recorded. Similarly, the Application Logs of a running Carbon instance display the log events of its deployed Web applications and Web services. The “Application Logs” page has been introduced as a fine-grained view of **System Logs**. While system logs display log events of the entire system holistically, the application logs page allows the user to select a particular application and view its logs only.

The log files can be retrieved in two ways:

- If syslog-ng is configured, log files are taken from the remote location where the log files are hosted using syslog-ng server.
- If syslog-ng is not configured, log files are taken from the local file system (super-tenant or Stand-alone apps).

For more information on logs and how to change log properties according to your preferences, refer to section **Working with Logs**.

Statistics on application logs are provided by the following feature:

**Name** : WSO2 Carbon - Logging Management Feature

**Identifier** : org.wso2.carbon.logging.mgt.feature.group

If the above feature is not bundled in your product by default, you can install it using the instructions given in section, **Feature Management**.

Follow the instructions below to access statistics on Application Logs.

1. Log on to the product's management console and select "Monitor -> Application Logs."

2. The "Application Logs" page appears. This page displays logs of a selected application in a bottom-up manner. For example,
2. You can see a drop-down list from which a deployed Web Services or a Web applications can be selected to view its log files.

3. You can see a drop-down list from which a deployed Web Services or a Web applications can be selected to view its log files.

Note

The log messages displayed on this page are obtained from a memory appender. Therefore, the severity (log level) of the displayed log messages are equal to or higher than the threshold of the memory appender. For more information on appenders, loggers, their log levels and logging, refer to http://logging.apache.org/log4j

4. In the "View" list, select the category of logs you want to view. The available categories are:
   - TRACE - Trace messages.
   - DEBUG - Debug messages.
   - INFO - Information messages.
   - WARN - Warning messages.
   - ERROR - Error messages.
   - FATAL - Fatal error messages.
   - ALL - Displays all categories of logs.

For Example,
5. You can also find a certain log using the search function. Enter a keyword (or part of a keyword) and click "Search." When a search criteria is given, the "View" is displayed as "Custom". For example,

![Search Interface]

The location of the log files on disk is specified in the log4j.configuration file.

### System Logs

The "System Logs" page displays all events of a running Carbon instance. Further, it facilitates downloading and viewing log files according to user preferences. The log files can be retrieved in two ways:

- If syslog-ng is configured, log files are taken from the remote location where the log files are hosted using syslog-ng server.
- If syslog-ng is not configured, log files are taken from the local file system (super-tenant or Stand-alone apps).

You can refer to more information about logs and how to set log properties in section Working with Logs. Statistics on system logs are provided by the following feature:

**Name**: WSO2 Carbon - Logging Management Feature  
**Identifier**: org.wso2.carbon.logging.mgt.feature.group

If it is not bundled in your product by default, you can install the above feature using the instructions given in section Feature Management.

Follow the instructions below to access statistics on System Logs.

1. Log on to the product's management console and select "Monitor -&gt; System Logs."

![System Logs Section]

2. The "System Logs" page appears. This page displays logs in a bottom-up manner.

For example,
2. To view old archived logs, click the "Show archived logs" tab at the bottom of the "System Logs" page.

3. To view old archived logs, click the "Show archived logs" tab at the bottom of the "System Logs" page.

4. In the "View" list, select the category of logs you want to view. The available categories are:

   - **TRACE** - Trace messages.
   - **DEBUG** - Debug messages.
   - **INFO** - Information messages.
   - **WARN** - Warning messages.
   - **ERROR** - Error messages.
   - **FATAL** - Fatal error messages.
Monitoring Performance Statistics

WSO2 products provide a range of access and performance statistics on the running Carbon instance as well as a service. These statistics include information about memory availability, request count, server name, server start time, system up time, active services, total memory, average, minimum, maximum response times etc. Statistics are accessible through the Management Console of a running Carbon instance and the functionality is provided by the following feature:

**Name:** WSO2 Carbon - Statistics Feature

**Identifier:** org.wso2.carbon.system.statistics.feature.group

If this feature is not bundled with the WSO2 product by default, you can install it using the instructions provided in the Working with Features section. Once installed, the necessary menus will appear in the Management Console. Access and performance statistics are available in two levels as follows:

- System Level Statistics
- Service Level Statistics
- Operation Level Statistics

### System-Level Statistics

Follow the instructions below to access system-level statistics.

1. Log on to the product’s Management Console and select System Statistics from menu Monitor. For example,
2. The System Statistics page appears as follows.

The following information is available:

- Service Summary
- Server Information
- Response Time Graph
- Memory Graph
- Statistics Configuration Panel

**Service Summary**
This panel provides the following information:

- **Average Response Time** - The average amount of time taken by the mediation channel to mediate a message (in milliseconds).
- **Minimum Response Time** - The least amount of time taken by the mediation channel to mediate a message (in milliseconds).
- **Maximum Response Time** - The greatest amount of time taken by the mediation channel to mediate a message (in milliseconds).
- **Total Request Count** - The total number of messages received and mediated through the mediation channel.
- **Total Response Count** - The total number of messages send and mediated through the mediation channel.
- **Total Fault Count** - The number of messages that triggered faults while being mediated through the channel.
- **Active Services** - The number of currently active services.

Server Information

This panel provides the following information:

- **Host** - Shows the IP address of the server.
- **Server Start Time** - Shows the time when the server started.
- **System Up Time** - Shows the time the server has been up and running.
- **Memory Allocated** - Shows the memory capacity of the server.
- **Memory Usage** - Shows the used memory of the server.

Response Time Graph
This graph shows the temporal variation of the Average Response time.

**Memory Graph**

This graph shows the temporal variation of the server Memory.

**Statistics Configuration Panel**

The *Statistics Configuration* panel is provided to customize the *System Statistics* display by entering values to the appropriate fields and clicking *Update*.
• **Statistics Refresh Interval (ms)** - Allows to specify the interval of the statistics refresh. A smaller value refreshes the statistics display faster.

• **Response Time Graph** - Allows to specify the X and Y parameters of the Response Time graph.
  - **X-Scale (units)**
  - **X-Width (px)**

• **Memory Graph** - Allows to specify the X and Y parameters of the Memory graph.
  - **X-Scale (units)**
  - **X-Width (px)**

If you want to reset the previous values before submitting the page after editing, click **Reset**. The **Restore Defaults** button sets the default values in the corresponding fields.

**Service-Level Statistics**

In addition to system-level statistics, you can also access service-level statistics through a selected service’s dashboard. Service dashboard and management capabilities in your product are provided by the following features:

**Name**: WSO2 Carbon - Service Management Feature

**Identifier**: org.wso2.carbon.service.mgt.feature.group

Not all WSO2 products have service management capabilities bundled in them by default. If required, you can add the functionality by installing the above features using instructions in section **Feature Management**. Once installed, the necessary functionality and menu items will appear in the Management Console.

Follow the instructions below to access service-level statistics:

1. Log on to the product’s Management Console and select **List from menu Web Services**.

2. The **Deployed Services** window appears. Click on a service to access its dashboard. For example,

3. The service’s dashboard appears. It contains the statistics panel in addition to others.
4. The Statistics panel provides the following statistical information about a service.

- Request Count
- Response Count
- Fault Count
- Maximum Response Time
- Minimum Response Time
- Average Response Time

<table>
<thead>
<tr>
<th>Statistics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Count</td>
<td>1</td>
</tr>
<tr>
<td>Response Count</td>
<td>0</td>
</tr>
<tr>
<td>Fault Count</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Response Time</td>
<td>37 ms</td>
</tr>
<tr>
<td>Minimum Response Time</td>
<td>37 ms</td>
</tr>
<tr>
<td>Average Response Time</td>
<td>37.0 ms</td>
</tr>
</tbody>
</table>

It provides a graphical view of the average response time as follows:
Operation-Level Statistics

You can also access operation-level statistics of a service's operation through its dashboard. Service and operation dashboards are provided by the following features:

**Name**: WSO2 Carbon - Service Management Feature

**Identifier**: org.wso2.carbon.service.mgt.feature.group

All WSO2 service hosting products have service management capabilities bundled in them by default. If the product you are using does not have this capability added by default, you can add it by installing the above features using instructions in section Feature Management. Once installed, the necessarily functionality and menu items will appear in the Management Console.

Follow the instructions below to access operation-level statistics:

1. Log on to the product's Management Console and select *List* from menu *Web Services*.

2. The *Deployed Services* window appears. Click on a service to access its dashboard. For example,
3. The *Service Dashboard* page appears. Click *Operations* in the *Quality of Service Configuration* panel.

4. The *Operations of Service* page appears. Click on a selected operation to view its *Operation Dashboard*. 
5. The operation's statistics panel appears in the *Operation Dashboard*. For example,

6. The *Statistics* panel of an operation provides the same statistics shown about a service.

- Request Count
- Response Count
- Fault Count
- Maximum Response Time
- Minimum Response Time
- Average Response Time

It also provides a graphical view of the average response time.
Monitoring SOAP Messages

The SOAP Tracer tool provided by WSO2 shows the SOAP messages, the respective SOAP message requests and responses when the services are deployed on the server, and the invoked operations. This functionality is provided by the following feature:

**Name**: WSO2 Carbon - SOAP Tracer Feature  
**Identifier**: org.wso2.carbon.soaptracer.feature.group

If the above feature is not included in your product by default, you can install it using the instructions given in section Feature Management.

By default, this feature is turned off. You have to turn it on to enable tracing.

Turning on the SOAP Tracer feature may impose a significant performance overhead. This is because all the SOAP messages will always be completely built (deferred building will not be done) and stored in the database by WSO2 Data Services. Hence this option should be used with caution.

Follow the instructions below to access the SOAP Tracer.

1. Log on to the product's management console and select "Monitor -> SOAP Tracer."

2. In the drop-down menu, select "Yes."

3. The tracer will show the messages of the operations that were invoked. For example,
By using the SOAP Tracer, you can see the SOAP messages with their time-stamps, service name, operation invoked and the number of requests to the server. The most recent SOAP messages are listed at the top. When a particular SOAP message is selected, its “Request” and “Response” can be viewed.

**Note**

This tracer does not apply to operations invoked in the admin services. They are filtered out.

4. Use the “Search” function to find a message. Fill in the field with a word (or a part of word) in a message and click on the “Search button.

5. You will see the message in the “Messages” list and its full description will be shown in the “Request” or “Response” text area. For example,
Monitoring Message Flows

Message Flows provide graphical or textual views of the globally-engaged handlers in the system at a given time. This functionality is provided by the following feature:

Name : WSO2 Carbon - Message Flows Feature
Identifier : org.wso2.carbon.message.flows.feature.group

If the above feature is not included in your product by default, you install it using the instructions given in section Feature Management.

Modules use handlers to engage in different message flows at defined phases. You can observe the handlers invoked in each phase of each flow in real time. For example, the Apache/Rampart module defines handlers in the security phase of each flow, which handles the security aspects of the messages that are transferred through these flows. Therefore, if the Rampart module is engaged, you can see the Apache/Rampart handlers in the message flows in real time.

Follow the instructions below to access the Message Flows.

1. Log on to the product's management console and select "Monitor -> Message Flows".

2. The Message Flows page with the graphical view of the message flows appears. There are four different flows
defined in the system:

- **In Flow**: A correct message coming into the system.
- **Out Flow**: A correct message going out of the system.
- **In Fault Flow**: A faulty message coming into the system.
- **Out Fault Flow**: A faulty message going out of the system.

In each flow, a message passes through a set of phases to reach the service. These phases vary according to the currently engaged modules within the system. The interface displays the current phases in each and every flow as shown in the diagram below.

3. In the graphical view of the message flows, click the links to get a view of the engaged handlers in each phase. For example, the figure below shows the handlers engaged in the Addressing phase at system start up.

4. You can see the text view of message flows. Click on the "Show Text View" link.
5. The page with the text view of message flows appears. The textual view provides the name and the fully qualified classes of all handlers within each and every phase.

**Monitoring Messages with TCPMon**

TCPMon is a tool that can be used to view and monitor the messages passed along a TCP-based conversation. Therefore, this is a convenient tool, particularly useful for debugging when you develop Web services. TCPMon is an Apache project distributed under Apache 2.0 License. This tool is not dependent on any third party libraries. Its user interface is based on a swing UI and works on almost all platforms that support Java.

The most common usage of TCPMon is as an intermediary, which monitors the communication between the client (front end) and the back end server. That is, the messages sent from the client are received by the intermediary instead of the back end server.

- Starting TCPMon
• Monitoring Messages between Client and Server

Starting TCPMon

TCPMon is available in the `<PRODUCT_HOME>/bin` directory of any WSO2 Carbon based product distribution. Alternatively, you can download TCPMon from Apache and run the tool.

Running TCPMon (from Carbon product pack)

Ensure that the following prerequisites are fulfilled in order to run TCPMon.

• Install JDK 1.4 or later version.
• Set the `JAVA_HOME` variable. This setting is required only if you are using the TCPMon available in the WSO2 Carbon based product pack.

For information on how to set the `JAVA_HOME` variable, go to Installing the Product, select the instructions relevant to your operating system and refer the ‘Setting JAVA_HOME’ section.

To run the TCPMon available with your WSO2 Carbon product pack:

1. Go to `<AS_HOME>/bin` directory of your product pack.
2. Execute the following command to run the tool.

   For Windows
   
   `tcpmon.bat`
   
   For Linux
   
   `./tcpmon.sh`

Running TCPMon (downloaded from Apache)

To download TCPMon from Apache and run the tool:

1. Download TCPMon from the following location: `http://archive.apache.org/dist/ws/tcpmon/1.0/tcpmon-1.0-bin.zip`.
2. Extract tcpmon-1.0-bin.zip archive.
3. Go to the build of the extracted directory to find the execution script.
4. Execute the following command to run the tool.

   For Windows
   
   `tcpmon.bat`
   
   For Linux
Monitoring Messages between Client and Server

The following diagram depicts a typical communication between the front end client and the back end server. 80 is the listening port of the back end server which receives the messages from the client:

![Diagram of typical communication](image)

The following diagram depicts how TCPMon is placed between the client and the server in order to monitor the messages. 8081 is the listening port in TCPMon which receives the messages from the client instead of the back end server:

![Diagram of TCPMon placement](image)

As an intermediary, TCPMon only receives messages and forwards them to the back end server. Therefore, it is a safe tool to be used for debugging purposes.

Note that TCPMon cannot be used to view messages transferred over https protocol.

To monitor messages from client to server using TCPMon:

1. Start TCPMon. Follow the instructions on Starting TCPMon.
2. Give 8081 (the listening port of TCPMon) in the **Listen Port** field (This could be any unused port in your local machine).
3. Give the address of the back end server as the target hostname. For example, if you are monitoring messages sent to **www.apache.org**, enter this web address as the hostname.
4. Give 80 as the target port, which is the listening port of **www.apache.org**.
5. Click **Add** to save the setting.

7. A new tab in TCPMon will indicate the 8081 port. You can view the requests and responses passing through TCPMon as shown below.
The options at the bottom of the screen can be used to have the messages in XML format (useful in debugging Web services), to save and resend the messages and also to switch the layout of the message windows.

JMX-Based Monitoring

Java Management Extensions (JMX) is a technology that lets you implement management interfaces for Java applications. JConsole is a JMX-compliant monitoring tool, which comes with the Java Development Kit (JDK) 1.5 or later versions. Therefore, when you use a WSO2 product, JMX is enabled by default, which allows you to monitor the product using JConsole. When you start JConsole, the MBeans tab will show the MBeans generated for your product. While some of these MBeans (ServerAdmin and DataSource) are common to all WSO2 products, some MBeans are specific to WSO2 AS.

Go to the WSO2 administration guide for detailed instructions on how to configure JMX for a WSO2 product, how to use JConsole for monitoring a product and for descriptions of the common MBeans used by all WSO2 products.
Working with Composite Applications

The topics given below explain the concept of 'Composite Applications' (C-Apps) and about how C-Apps can be deployed and managed.

- Introduction to Composite Applications
- Creating and Deploying a Composite Application
- Deleting Composite Applications
- Managing Server Roles

Introduction to Composite Applications

A Composite Application (C-App) is a collection of artifacts deployable on a WSO2 product instance. C-App files have CAR extensions and can be deployed to different runtimes. Each runtime will only deploy the artifacts which match with the role that the runtime is playing (e.g., ESB runtime will not deploy a data service in the CAR file, unless the default configuration is altered). These artifacts are usually JAVA-based or XML configurations, designed differently for each product in the WSO2 Carbon platform. You can deploy these artifacts to generate services.

A single WSO2 product can have numerous artifacts such as Axis2 services, dataservices, synapse configurations, endpoints, proxy services, mediators, registry resources, BPEL workflows etc. Usually, these artifacts are created in a development environment and then moved one by one into staging/production environments. Manually configuring artifacts to build up the entire solution this way is a time-consuming task. Instead, you can bundle configuration files and artifacts in a C-App and port Web service based solutions across environments more easily. C-Apps allow you to export your entire solution as a single archive file.

A single C-App can consist of artifacts that are deployable in various different products. Therefore, when you deploy a C-App in a particular WSO2 product, all its artifacts might not be deployed in that particular product instance. To sort out which can be deployed in which, the ServerRole property is used.

Structure of a C-App

The structure of a typical C-App, contains individual directories of the artifacts and a file named artifacts.xml, which contains the metadata about the artifact content listed inside the C-App. The following code illustrates a sample artifacts.xml file which provides in-depth details about the sample C-App and its content. This file contains the name of the C-App (in this case sampleCApp), its version and the type of the artifact. The relevant deployer for the artifact is identified using the artifact type; for C-Apps this is "carbon/application". In addition,
the artifacts.xml file also contains details about its content artifacts such as its name, version and server role of the artifact. The server role will be considered to recognize whether the given artifact is relevant to that running server. If the serverRole is matched with the WSO2 product that the C-App was deployed for, then those artifacts will be deployed using the CappDeployer; however, if the serverRole is not matched the artifact will be ignored.

Sample artifacts.xml file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<artifacts>
  <artifact name="sampleCApp" version="1.0.0" type="carbon/application">
    <dependency artifact="testArtifact01" version="1.0.0" serverRole="ApplicationServer"/>
    <dependency artifact="testArtifact02" version="1.0.0" include="true" serverRole="ApplicationServer"/>
    <dependency artifact="testArtifact03" version="1.0.0" include="true" serverRole="ApplicationServer"/>
  </artifact>
</artifacts>
```

The relevant deployable artifact (e.g., if its an Axis2 service the file extension used is .aar and if it is a webapp the file extension used is .war) and other relevant files needed for that specific artifact are found inside the artifact directory. This also contains an XML file named artifact.xml that contains the meta-data information about that artifact. This contains the artifact name, version, type of the artifact, server role and the name of the artifact file. For example if its an Axis2 service the type will be “service/axis2” and for webapps the type will either be “web/application” or “webapp/jaxws” based on the type of the webapp.

Sample artifact.xml files

```xml
<?xml version="1.0" encoding="UTF-8"?>
<artifact name="testArtifact02" version="1.0.0" type="service/axis2" serverRole="ApplicationServer">
  <file>testArtifact02.aar</file>
</artifact>
```

Creating and Deploying a Composite Application

The steps below show how to use WSO2 Developer Studio to develop individual artifacts and export them as a single C-App artifact. You can then deploy the C-App on the WSO2 product instance. Use this method to move the configurations of your server from one environment to another.

1. See User Guide in the Developer Studio documentation to create artifacts.
2. See Packaging Artifacts Into Deployable Archives in the Developer Studio documentation to export the C-App as a Carbon Application Archive (CAR) file.

   Next, you can upload the CAR file into your new product instance as follows.
3. Log in to the product's management console and select Add menu under the Applications menu.

   ![Carbon Applications]
   ![List]
   ![Add]

4. Select your CAR file and click Upload.
Add Applications

- The CAR files you deploy this way are dropped to <PRODUCT_HOME>/repository/carbonapps/{tenant-ID}. (tenant-ID) is 0 in a single-tenanted environment. 0 is the super tenant ID.
- As an alternative to uploading the C-App using the management console, you can copy the CAR archive into <PRODUCT_HOME>/repository/carbonapps/{tenant-ID} manually and it will be deployed.
- Manual deployment in a multi-tenanted environment is not recommended if you are unaware of the tenant ID.

5. If successfully deployed, the C-App will be listed on the Applications List window. For example,

Applications List

If the Carbon application does not have artifacts that are applicable to the particular product you are using, nothing will be listed on this window.

Deleting Composite Applications

Follow the instructions below to delete a deployed composite application from the product. It is not recommended to delete individual artifacts of a C-App. The recommended practice is to delete the entire C-App.

Note

As an alternative to deleting the C-App through the management console UI, you can delete the CAR archive from <PRODUCT_HOME>/repository/carbonapps/{tenant-ID} manually. {tenant-ID} is the ID of the tenant, which is 0 in a single-tenant environment. 0 is the super tenant ID. Manual removing a C-App in a multi-tenanted environment is not recommended if you are unaware of the tenant ID.

1. Log on to the product's management Console and click List under Carbon Applications. For example,

2. You will be able to see the deployed C-App on this page as shown in the example given below. Click the Delete
link associated with a certain application to remove it permanently.

### Applications List

<table>
<thead>
<tr>
<th>Applications</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataServices</td>
<td>Delete</td>
</tr>
</tbody>
</table>

3. You will be asked to confirm your request.

---

**Managing Server Roles**

A server role is a parameter that is mentioned in `<PRODUCT_HOME>/repository/conf/carbon.xml` file of all WSO2 products. When a C-App is being deployed, it reads the ServerRoles property from the `carbon.xml` and deploys only the resources that match the server-role value in the file. Each product has a different default ServerRoles property as follows:

- WSO2 Application Server - "ApplicationServer"
- WSO2 Business Activity Monitor - "BusinessActivityMonitor"
- WSO2 Business Process Server - "BusinessProcessServer"
- WSO2 Business Rules Server - "BusinessRulesServer"
- WSO2 Data Services Server - "DataServicesServer"
- WSO2 Enterprise Service Bus - "EnterpriseServiceBus"
- WSO2 Gadget Server - "GadgetServer"
- WSO2 Governance Registry - "GovernanceRegistry"
- WSO2 Identity Server - "IdentityServer"
- WSO2 Mashup Server - "MashupServer"

The following is an example list of C-App resources that map to default server roles.

- `ApplicationServer` - foo.aar, jax-wx.war
- `EnterpriseServiceBus` - proxy.xml
- `BusinessProcessServer` - my_bpel.zip
- `JaggeryServer` - jaggery_app.jag

In C-App development time, you can specify a server role for each and every artifact in the C-App. For example, say you are developing an Axis2 service and planning to deploy all your services in a single Application Server instance in the production setup. You can set the server role as `appserver1`.

---

**Server Roles Manager**

Server roles manager is a component to manage the server roles property for WSO2 Carbon based products. Due to the functionality of the server roles manager, users do not have to manually modify the `carbon.xml` to include the server-roles related to the product feature that they have added to a Carbon product instance. The server roles manager stores both `carbon.xml` file's default product roles as well as the user/tenant specific server roles in the configuration registry. So, when a C-App is deployed in Carbon, the C-App deployer checks for auto-mentioned server roles from the registry instead of the `carbon.xml` file.
In the server roles manager, the ServerRoles properties are of two types:

- **Default** - All the server roles picked from that particular product instance’s carbon.xml.
- **Custom** - All other server roles added by the users.

The following methods can be used to set the ServerRole:

- Using the management console to set the ServerRole property
- Using carbon.xml file to set the ServerRole property
- Using a system property to set the ServerRole property

**Using the management console to set the ServerRole property**

This is the easiest and the most recommended way to configure your server roles. Follow the instructions below to add a new server role:

1. Log in to the management console of your product and click **Server Roles** menu under the Configure menu.

2. Click **Add New Server Role**, enter the Role Name and click **Add**. You can add any textual name as a server role without special characters except underscore.

3. The newly added server role is displayed in the server roles list.

You can delete the server role using the **Delete** icon associated with it.

You cannot undo a deletion once performed. Users can even delete a default server role. Once deleted, the server role manager will not pick up the deleted server role from the carbon.xml file, next time the server starts.

**Using carbon.xml file to set the ServerRole property**

Find the ServerRoles element in <PRODUCT_HOME>/repository/conf/carbon.xml file. For example,
You can set multiple role elements here to specify the roles acted by the current server. For example, if you want this server to act as appserver1 and dataservices1 servers, configuration is as follows:

```xml
<ServerRoles>
  <Role>appserver1</Role>
  <Role>dataservices1</Role>
</ServerRoles>
```

Before setting the above, ensure that the current server has capability to deploy Axis2 services and data services. When you deploy a C-App artifact on this server, all artifacts which have the above two server roles will be deployed on the current instance. Others will be ignored.

**Using a system property to set the ServerRole property**

You can use the system property `ServerRoles` to specify the server roles that can be acted by the current product instance. When you start the server, pass the server roles as a comma separated list. For example,

```
sh wso2server.sh -DserverRoles=appserver1,dataservices1
```

Once you use the management console to set server roles, you can't change that configuration using the other two methods. Server roles are stored in the Registry when they are configured through the management console. Values in the Registry are always given priority over others.

**Running the Product on a Preferred Profile**

When a WSO2 product server starts, it starts all features and related artifacts bundled in the product. Multi-profile support allows you to run the product on a selected profile so that only features specific to that profile along with the common features start up with the server. This enables better resource utilization.

To run the product on a preferred profile:

1. Download the product distribution.
   For more information, see [Downloading the Product](#).
2. Open a command prompt:
   - On Windows, choose Start > Run, type `cmd` at the prompt, and press Enter.
   - On Linux/Solaris, establish a SSH connection to the server or log into the Linux console.
3. Execute one of the following commands to start a product on a preferred profile, where `<PRODUCT_HOME>` is the directory where you installed the product distribution and `. For more information on the preferred profile, see [AS profiles](#).

<table>
<thead>
<tr>
<th>OS</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>On Windows</td>
<td><code>&lt;PRODUCT_HOME&gt;/bin\wso2server.bat -Dprofile=&lt;preferred-profile&gt; --run</code></td>
</tr>
</tbody>
</table>
For example, in Linux/Solaris to start the Application Server with the support for Axis2 services only, type the following command:

```
sh <PRODUCT_HOME>/bin/wso2server.sh -Dprofile=axis2service
```

4. Once the server has started, run the Management Console by opening a Web browser and typing in the server’s URL. The URL is displayed as the last line in the start script’s console and log. The URL should be in the following format: https://<Server Host>:9443/carbon

AS profiles

The following are the list of profiles that corresponds to the respective services/applications:

<table>
<thead>
<tr>
<th>Service/Application Name</th>
<th>Profile Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>web application</td>
<td>webapp</td>
</tr>
<tr>
<td>Axis2 service</td>
<td>axis2service</td>
</tr>
<tr>
<td>Jaggery</td>
<td>jaggery</td>
</tr>
</tbody>
</table>

How multi-profiling works

Starting a product on a preferred profile starts only a subset of features bundled in the product. In order to identify what feature bundles apply to which profile, each product maintains a set of bundles.info files in the <PRODUCT_HOME>/repository/components/<profile-name>/configuration/org.eclipse.equinox.simpleconfigurator directories. The bundles.info files contain references to the actual bundles. Note that <profile-name> in the directory path refers to the name of the profile. For example, when there is a product profile named webapp, references to all the feature bundles required for the webapp profile to function are in a bundles.info file saved in <PRODUCT_HOME>/repository/components/webapp/configuration/org.eclipse.equinox.simpleconfigurator directory.

Note that when you start the server without using a preferred profile, the server refers to the <PRODUCT_HOME>/repository/components/default/configuration/org.eclipse.equinox.simpleconfigurator/bundles.info file by default. This file contains references to all bundles in the <PRODUCT_HOME>/repository/components/plugins directory, which is where all the components/bundles of a product are saved.

Migrating the Application Server

If you have multiple instances of AS and you want to move your data and configurations from one instance to another (such as moving from development to test or production), you can create a Composite Application project (C-App) and deploy the C-App in the new environment as described in Creating and Deploying a Composite Application. This approach works when the new instance you are migrating to uses the same AS release and has the same database versions as the previous instance.

See Upgrading from a Previous Release in the following situations:

- The new environment you are migrating to has a different database version. In this case, you must upgrade the older database.
- You want to upgrade from a previous AS release to a new one.
Calling Admin Services from Apps

WSO2 products are managed internally using SOAP Web services known as **admin services**. WSO2 products come with a management console UI, which communicates with these admin services to facilitate administration capabilities through the UI.

A service in WSO2 products is defined by the following components:

- **Service component**: provides the actual service
- **UI component**: provides the Web user interface to the service
- **Service stub**: provides the interface to invoke the service generated from the service WSDL

There can be instances where you want to call back-end Web services directly. For example, in test automation, to minimize the overhead of having to change automation scripts whenever a UI change happens, developers prefer to call the underlying services in scripts. The topics below explain how to discover and invoke these services from your applications.

**Discovering the admin services**

By default, the WSDLs of admin services are hidden from consumers. Given below is how to discover them.

1. Set the `<HideAdminServiceWSDLs>` element to false in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file.
2. Restart the server.
3. Start the WSO2 product with the `-DosgiConsole` option, such as `sh <PRODUCT_HOME>/bin/wso2server.sh -DosgiConsole` in Linux.
4. When the server is started, hit the enter/return key several times to get the OSGI shell in the console.
5. In the OSGI shell, type: `osgi> listAdminServices`
6. The list of admin services of your product are listed. For example:

```
Admin services deployed on this server:
```

7. To see the service contract of an admin service, select the admin service's URL and then paste it in your browser with `?wsdl` at the end. For example: `https://localhost:9443/services/UserAdmin?wsdl`

```
In products like WSO2 ESB and WSO2 API Manager, the port is 8243 (assuming 0 port offset). However, you should be accessing the Admin Services via the management console port, which is 9443 when there is no port offset.
```

8. Note that the admin service's URL appears as follows in the list you discovered in step 6:

```
```
Invoking an admin service

Admin services are secured using common types of security protocols such as HTTP basic authentication, WS-Security username token, and session based authentication to prevent anonymous invocations. For example, the UserAdmin Web service is secured with the HTTP basic authentication. To invoke a service, you do the following:

1. Authenticate yourself and get the session cookie.
2. Generate the client stubs to access the back-end Web services.

To generate the stubs, you can write your own client program using the Axis2 client API or use an existing tool like SoapUI (4.5.1 or later) or wsdl2java.

The wsdl2java tool, which comes with WSO2 products by default hides all the complexity and presents you with a proxy to the back-end service. The stub generation happens during the project build process within the Maven POM files. It uses the Maven ant run plug-in to execute the wsdl2java tool.

You can also use the Java client program given here to invoke admin services. All dependency JAR files that you need to run this client are found in the /lib directory.

Authenticate the user

The example code below authenticates the user and gets the session cookie:
import org.apache.axis2.AxisFault;
import org.apache.axis2.transport.http.HTTPConstants;
import org.wso2.carbon.authenticator.stub.AuthenticationAdminStub;
import org.wso2.carbon.authenticator.stub.LoginAuthenticationExceptionException;
import org.wso2.carbon.authenticator.stub.LogoutAuthenticationExceptionException;
import org.apache.axis2.context.ServiceContext;
import java.rmi.RemoteException;

public class LoginAdminServiceClient {
    private final String serviceName = "AuthenticationAdmin";
    private AuthenticationAdminStub authenticationAdminStub;
    private String endPoint;

    public LoginAdminServiceClient(String backEndUrl) throws AxisFault {
        this.endPoint = backEndUrl + "/services/" + serviceName;
        authenticationAdminStub = new AuthenticationAdminStub(endPoint);
    }

    public String authenticate(String userName, String password) throws RemoteException, LoginAuthenticationExceptionException {
        String sessionCookie = null;
        if (authenticationAdminStub.login(userName, password, "localhost")) {
            System.out.println("Login Successful");
            ServiceContext serviceContext = authenticationAdminStub._getServiceClient().getLastOperationContext().getServiceContext();
            sessionCookie = (String) serviceContext.getProperty(HTTPConstants.COOKIE_STRING);
            System.out.println(sessionCookie);
        }
        return sessionCookie;
    }

    public void logOut() throws RemoteException, LogoutAuthenticationExceptionException {
        authenticationAdminStub.logout();
    }
}

To resolve dependency issues, if any, add the following dependency JARs location to the class path: <PRO
Generate the client stubs

After authenticating the user, give the retrieved admin cookie with the service endpoint URL as shown in the sample below. The service management service name is ServiceAdmin. You can find its URL (e.g., https://localhost:9443/services/ServiceAdmin) in the service.xml file in the META-INF folder in the respective bundle that you find in $PRODUCT_HOME/repository/components/plugins.

```java
import org.apache.axis2.AxisFault;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;
import org.wso2.carbon.service.mgt.stub.ServiceAdminStub;
import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaDataWrapper;
import java.rmi.RemoteException;

public class ServiceAdminClient {
    private final String serviceName = "ServiceAdmin";
    private ServiceAdminStub serviceAdminStub;
    private String endPoint;

    public ServiceAdminClient(String backEndUrl, String sessionCookie) throws AxisFault {
        this.endPoint = backEndUrl + "/services/" + serviceName;
        serviceAdminStub = new ServiceAdminStub(endPoint);
        //Authenticate Your stub from sessionCookie
        ServiceClient serviceClient;
        Options option;
        serviceClient = serviceAdminStub._getServiceClient();
        option = serviceClient.getOptions();
        option.setManageSession(true);
        option.setProperty(org.apache.axis2.transport.http.HTTPConstants.COOKIE_STRING, sessionCookie);

        option.setProperty(org.apache.axis2.transport.http.HTTPConstants.COOKIE_STRING, sessionCookie);
    }

    public void deleteService(String[] serviceGroup) throws RemoteException {
        serviceAdminStub.deleteServiceGroups(serviceGroup);
    }

    public ServiceMetaDataWrapper listServices() throws RemoteException {
        return serviceAdminStub.listServices("ALL", "*", 0);
    }
}
```
The following sample code lists the back-end Web services:

```java
import org.wso2.carbon.authenticator.stub.LoginAuthenticationExceptionException;
import org.wso2.carbon.authenticator.stub.LogoutAuthenticationExceptionException;
import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaData;
import org.wso2.carbon.service.mgt.stub.types.carbon.ServiceMetaDataWrapper;
import java.rmi.RemoteException;

public class ListServices {
    public static void main(String[] args)
            throws RemoteException, LoginAuthenticationExceptionException,
            LogoutAuthenticationExceptionException {
        System.setProperty("javax.net.ssl.trustStore",
                "$ESB_HOME/repository/resources/security/wso2carbon.jks");
        System.setProperty("javax.net.ssl.trustStorePassword", "wso2carbon");

        System.setProperty("javax.net.ssl.trustStoreType", "JKS");
        String backEndUrl = "https://localhost:9443";

        LoginAdminServiceClient login = new
                LoginAdminServiceClient(backEndUrl);
        String session = login.authenticate("admin", "admin");
        ServiceAdminClient serviceAdminClient = new
                ServiceAdminClient(backEndUrl, session);
        ServiceMetaDataWrapper serviceList =
                serviceAdminClient.listServices();
        System.out.println("Service Names:");
        for (ServiceMetaData serviceData : serviceList.getServices()) {
            System.out.println(serviceData.getName());
        }

        login.logOut();
    }
}
```

**Working with Applications**

See the following topics for instructions on how the AS can be configured to facilitate the requirements of web applications that are deployed:

- Configuring Applications for AS
- Configuring Runtime Environments
- Customize Logging for Web Applications
- HTTP Session Management
- Setting up Virtual Hosts
- Using SSO with Web Applications
Configuring Applications for AS

The content given here explains how you can enable some optional configurations for your application. The instructions given here are applicable to all applications (generic web applications, JAX-RS/JAX-WS applications and Jaggery application) that are deployable in WSO2 AS.

- Enable SaaS mode for applications
- Enabling CORS for applications

Enable SaaS mode for applications

You can enable SaaS mode for your applications by configuring the META-INF/context.xml file relevant to each web application. This configuration allows the web application to be shared among multiple tenants.

To enable SaaS mode:

1. Open the META-INF/context.xml file of your web application from the archive directory.
2. Update the following section to enable/disable SaaS mode for your application:

```xml
<Context>
  <Realm
    className="org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm"
    enableSaaS="true"
    saasRules="*"
  />
</Context>
```

- The enableSaaS parameter can be set to "true" or "false". The value you give for this parameter will depend on the use case.
- The saasRules parameter controls how the web application should be shared among tenants. For example, consider that a tenant wants to share its web application with wso2.com, foo.com and bar.com. You can change the parameter value according to the use case as shown below.

If all tenants can access this application, enter the following:

```xml
<Context>
  <Realm
    className="org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm"
    enableSaaS="true"
    saasRules="*"
  />
</Context>
```

If all tenants except foo.com and bar.com can access this application, enter the following:

```xml
<Context>
  <Realm
    className="org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm"
    enableSaaS="true"
    saasRules="* foo.com,bar.com"
  />
</Context>
```
<Context>
  <Realm
  className="org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm"
  enableSaaS="true"
  saasRules="*, !foo.com,!bar.com" />
</Context>

If only foo.com and bar.com (all users) can access this application, enter the following:

<Context>
  <Realm
  className="org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm"
  enableSaaS="true"
  saasRules="foo.com,bar.com" />
</Context>

If only Sam and admin in foo.com tenant and all users in bar.com tenant can access this application, enter the following:

<Context>
  <Realm
  className="org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm"
  enableSaaS="true"
  saasRules="foo.com;users=Sam,admin,bar.com" />
</Context>

If you have not configured SaaS for your application as explained above, you will be warned with the following message when the application is invoked: "To enable SaaS mode for the webapp, "<webapp-name>"", configure the CarbonTomcatRealm in META-INF/context.xml".

**Enabling CORS for applications**

If required, you can enable CORS (cross origin resource sharing) for web applications by adding the CORS filter to the web.xml file as shown in the following example.
<filter>
  <filter-name>CorsFilter</filter-name>
  <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
  <init-param>
    <param-name>cors.allowed.origins</param-name>
    <param-value>*</param-value>
  </init-param>
  <init-param>
    <param-name>cors.allowed.methods</param-name>
    <param-value>GET,POST,HEAD,OPTIONS,PUT,DELETE,PATCH</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>CorsFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

The parameters used in this example are as follows:

- The `cors.allowed.origins` parameter is used to specify the web domains that should be allowed to share resources with the web application. The domain names should be specified as parameter values. If the parameter value is set to "*" as shown above, or if this parameter is not used at all, resource sharing will be allowed for all origins (all web domains).

- The `cors.allowed.methods` parameter is used to specify the type of requests for which CORS should be enabled. You can list the allowed methods as parameter values.

### Configuring Runtime Environments

WSO2 AS leverages a concept called runtime environments to separately maintain different runtimes for loading the relevant classes for your web applications. The WSO2 web application management feature provides the capability to load classes from different locations to your applications.

Most web applications that are deployed in production environments depend on external, third-party libraries for different functionality. Through the WSO2 Application Server's runtime environment, you can control class loading per server or application. The following runtimes that are available with WSO2 AS are sufficient for most users, but you can also define your own runtimes. There are four default runtime environments available in AS 5.3.0, which are as follows:

- **Tomcat Environment**: This is the minimal runtime, which is identical to a pure Tomcat runtime. It only has Tomcat, Servlet, JSP, EL and JSTL available in the server-level classpath. If you want additional JARs, you should package them with the web application or place them in the Tomcat environment's extension directory.

- **Carbon Environment**: This consists of both the Tomcat environment and the WSO2 Carbon runtime. It does not provide CXF or Spring dependencies. If you want additional JARs, you should package them with the web application or place them in the WSO2 Carbon environment's extension directory.

- **CXF Environment**: This consists of the Tomcat environment, CXF and Spring. It does not provide the WSO2 Carbon runtime. If you want additional JARs, you should package them with the web application or place them in the CXF environment's lib directory.

- **Javaee Environment**: This consists of the Javaee runtime and the WSO2 Carbon runtime. If you want additional JARs, you should package them with the web application or place them in the Javaee environment's lib directory.

See the following topics for more information:
Setting up the runtime environment

To configure class loading, you should add the class loading configuration in a file named `webapp-classloading.xml` and place it in the META-INF folder of a web application. This folder should be created inside the `src/main/webapp` folder of your web application. All the artifacts related to a web application are saved in the following directory: `<PRODUCT_HOME>/repository/deployment/server/<Web_Application_Type>/<Web_Application_Name>`. For example: `<PRODUCT_HOME>/repository/deployment/server/webapps/SampleApp`.

The Carbon runtime is the default runtime in WSO2 products. If you want to use the runtime environments other than Carbon, provide the `webapp-classloading.xml` file with the correct configurations. Otherwise, the application will be deployed in the default Carbon runtime environment.

The `webapp-classloading.xml` file takes the following format:

```
<Classloading xmlns="http://wso2.org/projects/as/classloading">
  <Environments>{Runtime Environment Names} </Environments>
</Classloading>
```

- Shown below is the `webapp-classloading.xml` file configuration to specify CXF as the runtime environment:

```
<Classloading xmlns="http://wso2.org/projects/as/classloading">
  <Environments>CXF</Environments>
</Classloading>
```

- You might want to access some Carbon features in a CXF application. To achieve this, specify a comma-separated list of environments in the `webapp-classloading.xml` file. The following example specifies both CXF and Carbon as the runtime environments:

```
<Classloading xmlns="http://wso2.org/projects/as/classloading">
  <Environments>Spring,Carbon</Environments>
</Classloading>
```

- From AS 5.3.0 onwards, Java EE 6 Web Profile support is available as a feature, which allows you to enable the Javaee runtime environment, in addition to Tomcat, Carbon and CXF runtime environments. See the topic on enabling Java EE 6 Web Profile for step-by-step instructions on configuring the Java EE runtime.

Extending the runtime environments

This section explains how to place your external dependencies in a running server. This allows you to share dependencies with a number of applications without packaging them with each and every application. To do this, place the common dependencies in the relevant directories given below.

- In Tomcat Environment - `<PRODUCT_HOME>/lib/runtimes/ext`
- In Carbon Environment - `<PRODUCT_HOME>/repository/components/lib`
- In CXF, Javaee or any Custom Environment – Use the environment's `lib` directory. For example: `<PRODUCT_HOME>/lib/runtimes/cxf`

Practice caution when placing dependency files in Tomcat environment's `ext` directory as those dependencies will be visible to all other environments. For example, if you place incompatible Spring dependencies in the `<PRODUCT_HOME>/lib/runtimes/ext` directory, it can cause problems with the existing Spring dependencies in the CXF runtime environment.

If there are such incompatible dependencies, the recommended best practice is to package them with the web application in the Tomcat environment itself. Libraries that are only used by a particular web application can be put into the `<webapp_name>.war/WEB-INF/lib` directory.

Adding Custom Runtime Environments

To define custom runtime environments, add a new element as `<ExclusiveEnvironments>` in the `<PRODUCT_HOME>/repository/conf/tomcat/webapp-classloading-environments.xml` file.

To define a custom runtime environment for Spring:

1. Modify the `webapp-classloading-environments.xml` file with the following entries.

   ```xml
   <ExclusiveEnvironments>
   <ExclusiveEnvironment>
   <Name>Spring</Name>
   <Classpath>${carbon.home}/lib/runtimes/spring/*.jar;${carbon.home}/lib/runtimes/spring/</Classpath>
   </ExclusiveEnvironment>
   </ExclusiveEnvironments>
   ```

2. Next, create and copy of the related Spring dependencies in the `<PRODUCT_HOME>/lib/runtimes/spring` directory.
3. Add the following entries to the `webapp-classloading.xml` file of each web application so that they can be used in the Spring runtime environment defined above.

   ```xml
   <Classloading xmlns="http://wso2.org/projects/as/classloading">
   <Environments>Spring,Carbon</Environments>
   </Classloading>
   ```

Note that adding custom runtime environments to your system without studying their impact can cause unfavorable results. For example, assume an Application Server (AS) instance contains the following configurations:

- CXF (runtime provided by AS) contains CXF 2.7.5 and Spring 3.0.7 dependencies.
- Spring (custom runtime) contains Spring 3.2.1 dependencies.

If a web application consumes both of the above environments, the following problems will arise:

1. The web application's classpath contains dependencies from two Spring versions as 3.0.7 and 3.2.0.
1. Create a Maven project to build the web application. Note that Apache Maven WAR Plugin is used to build the .war file.

2. Create a log4j.properties file inside the src/main/resources folder. You can simply copy the log4j

The Spring project does not recommend this as it will cause several class loading issues.

2. CXF 2.7.5 itself uses Spring 3.0.7. It is possible that a particular CXF version is not properly tested for compliance with another version of Spring. You should study the CXF project recommendations to find a suitable Spring version.

Upgrading Existing Runtime Environments

The CXF runtime environment comes by default with core CXF 2.7.5 and core Spring 3.0.7 dependencies. If you want to upgrade to a different CXF version, there are two recommendations:

- Upgrade the CXF runtime environment: You can replace the existing CXF/Spring dependent JARs in the <PRODUCT_HOME>/lib/runtimes/cxf directory with new CXF/Spring JARs.

- Instead of upgrading server-level CXF dependencies, you can package all the required JARs in the Web-INF/lib directory of the web application and deploy that in the Tomcat runtime environment.

If you want to add optional CXF/Spring JARs, copy them to the <PRODUCT_HOME>/lib/runtimes/cxf directory after ensuring that they are compatible with the existing JARs.

Class loading pattern for web applications is always child-first and this can not be changed. Therefore, location 2 is always given the highest priority when the same library exists in both locations.

Customize Logging for Web Applications

WSO2 Application Server provides a platform for hosting and managing web applications. Multitenancy further allows you to maintain your own deployment area inside the server. During the development phase of a web application, it is necessary to see the errors published on the server console in order to troubleshoot problems and errors.

WSO2 Application Server already provides container level logging facility. It supports most of the commonly used logging API's, such as log4j, common-logging, java-utils-logging, etc. Therefore, if you have inserted log statements to your web application, you will be able to access the logs either on the product's startup console or the log file stored in the <AS_HOME>/repository/logs/directory. The default log file configured for your product is wso2caarbon.log. The log4j configurations for your server are specified in the log4j.properties file stored in the <PRODUCT_HOME>/repository/conf/directory, which you can customize as required. See the topic on Working with Logs for more information.

However, if it is necessary to manage customized logs for your web application individually, a customized log4j.properties file should be packaged in the .war file of the web application along with the relevant libraries and configurations.

Note that tenant aware logging and app-context aware logging are built using the carbon context. Therefore, to use tenant aware logging and app context aware logging, it is recommended to use the AS container level logging facility without doing any customizing the individual web applications if it is there.

Given below are the steps that you must follow in order to enable customized logging for your web application:

1. Create a Maven project to build the web application. Note that Apache Maven WAR Plugin is used to build the .war file.

2. Create a log4j.properties file inside the src/main/resources folder. You can simply copy the log4j
.properties file in the <AS_HOME>/repository/conf/ directory and do the necessary customizations. Note that the default resource directory for all Maven projects is src/main/resources.

3. Add the Maven dependency for log4j to the pom.xml of the web application as shown below. This will package apache-logging-log4j.jar when the .war file is build.

```xml
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.17</version>
</dependency>
```

Be sure that <packagingExcludes>WEB-INF/lib/*.*,jar</packagingExcludes> does not exist in the pom.xml of the web application.

4. Add the following Maven dependency if you are using the commons API:

```xml
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>1.1.3</version>
</dependency>
```

5. Update the web application by adding the necessary log lines and specifying the Java classes that need to be imported. See the following example, which uses the Apache Commons API for logging:

1. Add the following import statements to your web application's Java classes when you need to use JCL SPI (Jakarta Commons Logging Service Provider Interface):

```java
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
```

2. For every class in which you want to use Commons logging, you need to declare and initialize a log using the following:

```java
private Log log = LogFactory.getLog(ClassName.class);
```

3. To log a message to a logger use the appropriate method provided by org.apache.commons.logging.Log interface as the log priority.

```java
log.info(Object message);
log.debug(Object message);
log.error(Object message);
log.warn(Object message);
```
Example: See the following simple GET method which generates logs with different priorities:

```java
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("/")
public class CommonsLogGenerator {
    private Log log = LogFactory.getLog(CommonsLogGenerator.class);

    @GET
    @Path("/logging")
    public String createLogs() {
        log.info("INFO LOG");
        log.debug("DEBUG LOG");
        log.error("ERROR LOG");
        log.warn("WARN LOG");
        return "hello";
    }
}
```

6. Build the .war file using Maven. Check the following:
   - The default resource directory for all Maven projects (src/main/resources) will be copied to the target/classes and the WEB-INF/classes folders in the .war file.
   - The JARs related to log4j (apache-logging-log4j.jar) will be bundled in the WEB-INF/lib folder.
   - The directory structure is preserved by the build process.

7. Start WSO2 AS and deploy the web application's .war file.
8. Now, when you invoke the web application, the logs will be created according to the configurations you specified in the log4j.properties file.

### HTTP Session Management

In WSO2 Application Server, the HTTP sessions of web applications are handled by the underlying, embedded tomcat instance. By default, tomcat provides two classes to manage the sessions of web applications: "StandardManager" and "PersistenceManager".

Additionally, the following features are available in AS for managing the sessions of web applications:

1. In the webapp management UI, a list of all the active sessions are listed.
2. Sessions are terminated when the session is inactive for the specified session expiry time. Sessions can also be explicitly terminated using the management console.

Please note that this UI-based session termination still does not work in a clustered setup. The sessions will be terminated only on the node from which the UI was loaded. Also, if you have a clustered environment, you can read more about HTTP session replication from here.

In WSO2 Application Server, "CarbonTomcatSessionManager" and "CarbonTomcatSessionPersistentManager" are the extensions to "StandardManager" and "PersistenceManager" implementations respectively. The main feature of these two manager implementations is that both are tenant aware and ensures that the sessions of one tenant
cannot be accessed by other tenants. "CarbonTomcatSessionPersistentManager" has the capability to store the currently active and idle sessions to a storage scheme specified by the "Store" configuration. By default, there are two types of stores that can be used with the "PersistanceManager". They are File-based store and JDBC-based store. See the tomcat documentation for more information on these storage implementations.

You can enable "CarbonTomcatSessionPersistentManager" (session persistence) for web applications by updating the context.xml file stored in the <AS_HOME>/repository/conf/tomcat folder. See the following topics for instructions:

- Enabling File-based store
- Enabling JDBC-based store

**Enabling File-based store**

Open the context.xml file and add the following to enable a File-based store with CarbonTomcatSessionPersistentManager.

```xml
<Manager className='org.wso2.carbon.webapp.mgt.CarbonTomcatSessionPersistentManager'>
    <Store className='org.apache.catalina.session.FileStore' directory='directory-name'/>
</Manager>
```

**Enabling JDBC-based store**

1. Set up a MySQL database and create a new table using the following SQL command:

```sql
CREATE TABLE tomcat_sessions (
    session_id VARCHAR(100) NOT NULL PRIMARY KEY,
    valid_session CHAR(1) NOT NULL,
    max_inactive INT NOT NULL,
    last_access BIGINT NOT NULL,
    app_name VARCHAR(255),
    session_data MEDIUMBLOB,
    KEY kapp_name(app_name)
);
```

The above table will be used by the JDBC store to swap out the sessions when required and to swap in the session when they are loaded back into the memory during a server restart.

2. The following is the default configuration in the context.xml file for enabling a JDBC-based store using the MySQL database.
3. For the database connectivity from Application Server, we need to place the relevant JDBC connector driver in the `<AS_HOME>/repository/components/lib` directory and start the server.

Setting up Virtual Hosts

Virtual host support is an in-built feature of Apache Tomcat that allows you to deploy multiple domains in a single Tomcat server instance. This feature allows the super tenant of your Application Server to deploy web applications with different domain names, which are all hosted with the same ip address. Previous releases of WSO2 Application Server loads web applications to the `<AS_HOME>/repository/deployment/server/webapps/` directory. With the new feature in WSO2 AS 5.3.0, there will be a separate folder in the `<AS_HOME>/repository/deployment/server/` directory, which corresponds to each host domain created by the super tenant.

Note that virtual hosts can only be created and used by the super tenant. All other users will be allowed to deploy web applications to the default host configured for the product.

See the following topics for instructions:

- Defining virtual hosts for the server
- Deploying applications in virtual hosts

Defining virtual hosts for the server

To enable virtual host support and to add new hosts in WSO2 AS:

1. All the required hosts for your server should be added to the `<AS_HOME>/repository/conf/tomcat/catalina-server.xml` file as explained below.

   - The `catalina-server.xml` file comes with the following host enabled by default:
The "host name" attribute specifies the domain name while the "appBase" attribute defines the location that stores the web apps. Note that the "appBase" must be located inside the <AS_HOME>/repository/deployment/server/webapps/ folder. In the above example, the default domain name is set to "localhost" and the location of web apps is the "webapps" folder.

- You can add a new hostname with a new appBase location by adding the following to the catalina-server.xml file:

```
<Host name="www.vhost.com" unpackWARs="true" deployOnStartup="false" autoDeploy="false"
appBase="${carbon.home}/repository/deployment/server/vhost/>
```

2. Setup the DNS configuration by mapping the new host name to the preferred IP address as shown below. If you are testing locally, the "hosts" file is located in the /etc/hosts folder for linux. In a production environment, the DNS server should be updated with the domain names accordingly.

```
(your-ip-address)        (host-name)
```

(eg: 10.100.5.86  www.vhost.com)

**Deploying applications in virtual hosts**

The virtual hosts for your server are now configured and you can start adding web applications to any appBase location. Note that this enables you to deploy web applications with the same name into different appBases. Web apps can be added to the product in two ways:

- Simply, copy the WAR file into the relevant appBase location (e.g.: add example.war file to <PRODUCT_HOME>/repository/deployment/server/vhost). When you start WSO2 AS, the web application (example.war) will be deployed and can be accessed through the following address: http://www.vhost.com:9763/<webapp-name>. Note that if you change the default offset value (in the <PRODUCT_HOME>/repository/conf/carbon.xml file), the port number should get updated accordingly.
- Use the management console of the product as shown below. You can select the host name when you add a web app from the UI.

See the topic on deploying applications in AS for instructions on using the management console.
Virtual hosts in a clustered deployment:

If you have enabled deployment synchronization support in an environment of AS clusters, the web applications that you add to any appBase location in the manager node will not be committed from the manager node to the svn repo. Therefore, the appBase locations of all the virtual hosts will also follow the same behaviour as the default “webapps” directory. For depsync support to work in a worker/manager setup, all the nodes should have a similar configuration in the catalina-server.xml file with the relevant host entries.

Example: Manager node and a worker node with depsync enabled in a cluster.

1. Consider that a manager node has a new host entry in addition to the default entry as follows:

   ```xml
   <Host name="www.vhost.com" unpackWARs="true"
   deployOnStartup="false" autoDeploy="false"
   appBase="${carbon.home}/repository/deployment/server/vhost/">
       ......
   </Host>
   ```

2. If the manager node deploys a webapp under the newly added host, the webapp will unpacked in the `<PRODUCT_HOME>/repository/deployment/server/vhost/` location.

3. Due to the depsync effect, the .war file in the above location will be synced to the same location of the worker node. For that webapp to deploy properly and to be invoked via the same host name, both the worker and manager nodes should follow the same configuration when adding Host entries to the catalina-server.xml file.

Using SSO with Web Applications

Single Sign On or SSO is a popular way of managing a log-in session throughout several applications or programs. It allows users to access multiple applications through a single log-in session, without having to enter credentials multiple times. WSO2 provides the capability of enabling SSO for user applications via WSO2 Identity Server.

From the WSO2 Application Server perspective, SSO can be configured within multiple applications deployed in WSO2 AS, using WSO2 IS. According to normal practice, application developers need to include certain code in the application in order to enable SSO. The use of the SSO Valve reduces this complexity and allows application developers to easily enable SSO for a web application deployed in WSO2 AS. With the SSO Valve, SSO can be enabled for applications with minimal configuration changes.

Configuring the servers (WSO2 AS and WSO2 IS)

The following configurations should be done in WSO2 AS:

1. Open the sso-sp-config.properties file, which is stored in the `<AS_HOME>/repository/conf/security/` directory and update the "IdPEntityId". By default, this value is set to the localhost as shown below.

   ```properties
   #IdPEntityId
   SAML2.IdPEntityId=localhost
   ```
The sso-sp-config.properties file is the global configuration file for generic SSO configurations in WSO2 AS. See the descriptions of all the properties that you can configure in this file from here.

2. Register the SSO valve in WSO2 AS by appending the following in the `<AS_HOME>/repository/conf/tomcat/catalina-server.xml` file. Be sure to add this entry just after "CarbonStuckThreadDetectionValve".

```xml
<V valve="org.wso2.carbon.webapp.mgt.sso.SAMLSSOValve"/>
```

3. The SSO valve is capable of handling Single Log Out redirections to the Assertion Consumer URL of the web application by redirecting to the root context of the web app. This capability can be turned on/off using the "handleConsumerURLAfterSLO" property in the sso-sp-config.properties file:

```properties
#Enable this if needed to automatically redirect from acs page after SLO
handleConsumerURLAfterSLO=true
```

The following configurations should be done in WSO2 IS:

1. The respective SSO Service Providers need to be registered in WSO2 Identity Server for each web application.

Note that the parameters/values defined in the sso-sp-config.properties file of WSO2 AS should correspond to the parameters defined for the service providers registered in WSO2 IS.

1. Since the valve automatically determines the SSO issuer-id, the service provider issuer-id needs to be in the following format:

For super tenant web applications: issuer-id = webapp-name
For tenant web applications: issuer-id = t_tenant-name_webapp-name

Examples:

- When the foo.war web application is deployed for the Super Tenant, the issuer-id = foo.
- When the bar.war web application is deployed in wso2.com tenant, the issuer-id = t_wso2.com_bar.

2. The 'Assertion Consumer URL' for the service providers should be set to the same value specified in the sso-sp-config.properties file. Shown below is the URL given in the default sso-sp-config.properties file.

```properties
#The URL of the SAML 2.0 Assertion Consumer
#SAML.ConsumerUrl=http://localhost:8080/avis.com/home.jsp
```

2. Update "IdentityProviderURL" and “EntityId” in `<IS_HOME>/repository/conf/identity.xml` with the correct IS hostname.
In WSO2 IS, update the resident IDP "Entity Id" with the same value as the "EntityId". Shown below is the default configuration.

### Resident Identity Provider

![Resident Identity Provider Configuration](image)

### Inbound Authentication Configuration

- **OpenID Configuration**
- **SAML2 Web SSO Configuration**
  - **Identity Provider Entity Id:** localhost
  - **SSO URL:** https://localhost:9443/samlss
  - **Logout Url:** https://localhost:9443/samlss

### Configuring the web applications

You must enable SSO for web applications by adding the following context parameter to the web application's `web.xml`:

```xml
<context-param>
  <param-name>enable.saml2.sso</param-name>
  <param-value>true</param-value>
</context-param>
```

### Configuring SSO valve to access web applications in multiple tenants

When the SSO valve is engaged, you can allow tenants to log in to web applications that are deployed for other tenants. For example, consider that there are two tenants: "a.com" and "b.com". Tenant a.com deploys an application named foo.war, which needs to be accessed by tenant b.com. In order to allow this, the following configurations should be in place:

1. Tenant b.com needs to create an SP (service provider) in WSO2 IS similar to the SP created by a.com. Note that the SP configurations for b.com should have the same issuer id, etc. In this case issuer id for b.com would be: t_a.com_foo.
2. If SSO request validation is required (this is optional), tenant b.com needs to import tenant a.com's public key into tenant a.com's keystore. Then in the aforementioned SP configuration for the foo app, you can enable the "Enable Signature Validation in Authentication Requests and Logout Requests" option and select the certificate of a.com. See the topic on setting up keystores for more information on keystores.

### Logging in to web applications using SSO

Note that the "EntityId" in `<IS_HOME>/repository/conf/identity.xml` should be the same as the "SAML2.IdPEntityId" defined in `<AS_HOME>/repository/conf/security/sso-sp-config.properties`.

3. In WSO2 IS, update the resident IDP "Entity Id" with the same value as the "EntityId". Shown below is the default configuration.
Once you have the above configurations in place, you can simply deploy your web applications in WSO2 AS. Note that WSO2 IS should be started and running in parallel to AS for this feature to work. You will now be able to log in to the web application deployed on WSO2 AS using SSO. Try this sample for a demonstration.

**Enable XACML Fine Grained Authorization for Web Apps**

This section explains how you can enable fine grained authorization for your web applications by integrating with WSO2 Identity Server. See the following topics for details:

- About Fine Grained Authorization
- About XACML policies
- Providing XACML Fine Grained Authorization to WebApp Requests
- Samples

**About Fine Grained Authorization**

When a resource (web application) is deployed in an application server, it is necessary to specify the users that should be granted authority to use the application. Typically, authority is granted to the user of the web application depending on the type of user, resource and the action that the user performs with the resource. However, if we can use more details from the user scenario to determine the authority that should be granted to the user, we call it **Fine Grained Authorization**. For example, in addition to the type of user, resource and the action performed on the resource, you can use fine grained criteria such as the environment, time, user's role etc., to grant authority.

Example of a fine grained authority requirement: "This document can be edited only by PersonX, who is a Teacher and between 8am to 10am in the school office premises".

**About XACML policies**

To evaluate such a requirement against a user request, we have to document those fine grained details of the requirement. These documents are called **Policies** and **XACML** is used to document this type of policies. We can evaluate a user's requirements against these XACML policies using an XACML engine. **WSO2 Identity Server** can be used for this purpose. WSO2 IS has an XACML Policy Engine where users can evaluate their requests and it provides many functionalities related to XACML Authorization.

**Providing XACML Fine Grained Authorization to WebApp Requests**

**WSO2 Application Server**, **Apache Tomcat** or any other web container can be used to host our web apps. If it is required to provide fine grained access (authority) to our Web Apps, WSO2 Identity Server can be used as the **XACML Policy Decision Point (PDP)**. This PDP can be accessed via a web service called Entitlement Service. We use the servlet filter named **Entitlement Servlet Filter** as the **Policy Enforcement Point (PEP)** for Web App authorization. This allows us the flexibility of using it in any Web App container. The **Entitlement Servlet Filter** uses a proxy to communicate with WSO2 Identity Server.

The following diagram shows how the servlet filter receives the decision on user authority:
To get an entitlement decision, we need some parameters like UserName, ResourceName, Action and Environment. We can map the resource name to the servlet to which the request is sent. Environment will be the WebApp. Action will be the HTTP action GET, POST etc. In order to get the user name of the person who sent the Web App request, the following Java EE authentication mechanisms are used:

- Basic Authentication
- Client Cert Authentication
- Digest Authentication
- Form Authentication

To grant authority, we have to authenticate the person. After the authentication, we can obtain the username in the servlet filter using the above mentioned methods. All the parameters can be obtained to get an entitlement decision. As shown in the diagram, when a request comes to a particular Web App which has the engaged Entitlement Servlet Filter, the following parameters are obtained: UserName, ResourceName, Action and Environment. Then the PDP Proxy is initialized to communicate with WSO2 IS. After that, the parameters are sent as an XACML request and the entitlement decision is received. Depending on the entitlement decision received, the request which has came to the Web App is either stopped or passed.

The next critical step in this process is for the user to engage the Entitlement Servlet Filter. For that, we use the web.xml. From this file, the servlet filter will read necessary parameters in order to initialize the communication with WSO2 IS. The following shows an example web.xml which configures the Entitlement Servlet Filter.

**Example web.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <display-name>Entitlement_Sample_WebApp</display-name>
```
<!-- The scope in which the subject would be available. Legal values are basicAuth, request-param, request-attribute, session -->
<!-- This param is optional. If not provided would be set to default value -->
<context-param>
    <param-name>subjectScope</param-name>
    <param-value>basicAuth</param-value>
</context-param>

<!-- The name of the identifier by which to identify the subject -->
<!-- This param is Mandatory. Should be provided -->
<context-param>
    <param-name>subjectAttributeName</param-name>
    <param-value>username</param-value>
</context-param>

<!-- The username to perform EntitlementService query-->
<!-- This param is Mandatory. Should be provided -->
<context-param>
    <param-name>userName</param-name>
    <param-value>admin</param-value>
</context-param>

<!-- The password to perform EntitlementService query -->
<!-- This param is Mandatory. Should be provided -->
<context-param>
    <param-name>password</param-name>
    <param-value>admin</param-value>
</context-param>

<!-- The URL to perform EntitlementService query-->
<!-- This param is Mandatory. Should be provided in this format -->
<!-- If the transport type is SOAP give the url like https://localhost:9443/services/ -->
<!-- If the transport type is Thrift give the url like https://localhost:9443/ -->
<context-param>
    <param-name>remoteServiceUrl</param-name>
    <param-value>https://localhost:9443/services/</param-value>
</context-param>

<!-- EntitlementFilter Settings -->
<filter>
    <filter-name>EntitlementFilter</filter-name>
    <filter-class>org.wso2.carbon.identity.entitlement.filter.EntitlementFilter</filter-class>
</filter>

<!-- Client Class that extends AbstractEntitlementServiceClient. Legal values are basicAuth, soap and thrift. Default is 'thrift'. -->
<init-param>
    <param-name>client</param-name>
</init-param>
<param-value>basicAuth</param-value>
</init-param>

<!-- Decision caching at PEPProxy. Legal values are simple and carbon. This parameter is optional. If not specified no caching is done. -->
<init-param>
  <param-name>cacheType</param-name>
  <param-value>simple</param-value>
</init-param>

<!-- Maximum number of cached entries. Legal values are between 0 and 10000. Only works with caching. -->
<init-param>
  <param-name>maxCacheEntries</param-name>
  <param-value>1000</param-value>
</init-param>

<!-- Time interval for which cached entry is valid. Only works with simple cache type. -->
<init-param>
  <param-name>invalidationInterval</param-name>
  <param-value>100000</param-value>
</init-param>

<!-- URL to redirect to if authorization fails -->
<!-- This param is Mandatory. Should be provided -->
<init-param>
  <param-name>authRedirectUrl</param-name>
  <param-value>/index.jsp</param-value>
</init-param>

<!-- This will be used if the transport type is thrift. This is mandatory -->
<init-param>
  <param-name>thriftHost</param-name>
  <param-value>localhost</param-value>
</init-param>

<!-- This will be used if the transport type is thrift. This is optional. If not provided would be set to default value -->
<init-param>
  <param-name>thriftPort</param-name>
  <param-value>10500</param-value>
</init-param>

</filter>

<!-- Filter mappings used to configure URLs that need to be authorized -->
<filter-mapping>
  <filter-name>EntitlementFilter</filter-name>
  <url-pattern>/protected.jsp</url-pattern>
</filter-mapping>
<filter-mapping>
<!-- Filter mappings used to configure URLs that need to be authorized -->
  <filter-mapping>
    <filter-name>EntitlementFilter</filter-name>
    <url-pattern>/other.jsp</url-pattern>
  </filter-mapping>
</filter-mapping>

<!-- Mandatory mapping that needs to be present to work with PEP cache update authorization -->
<filter-mapping>
  <filter-name>EntitlementFilter</filter-name>
  <url-pattern>/updateCacheAuth.do</url-pattern>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
<servlet>
  <servlet-name>EntitlementCacheUpdateServlet</servlet-name>
  <servlet-class>org.wso2.carbon.identity.entitlement.filter.EntitlementCacheUpdateServlet</servlet-class>
</servlet>

<!-- HTTPS port of the web container used when redirecting request to come over https port for cache update authentication -->
<init-param>
  <param-name>httpsPort</param-name>
  <param-value>9453</param-value>
</init-param>

<!-- Authentication mode for cache update. Legal values are webapp and wso2is -->
<init-param>
  <param-name>authentication</param-name>
  <param-value>webapp</param-value>
</init-param>

<!-- Authentication page used for cache update authentication. Legal values are default and custom -->
<init-param>
  <param-name>authenticationPage</param-name>
  <param-value>default</param-value>
</init-param>

<!-- Authentication page URL used for cache update authentication. Works only with custom for authenticationPage -->
<init-param>
  <param-name>authenticationPageUrl</param-name>
  <param-value>/updateCache.html</param-value>
</init-param>
</servlet>

<!-- Servlet mapping needed for cache update authentication -->
<servlet-mapping>
  <servlet-name>EntitlementCacheUpdateServlet</servlet-name>
</servlet-mapping>
<url-pattern>/updateCache.do</url-pattern>
</servlet-mapping>
</web-app>

**Samples**

For a demonstration of this functionality, run the Entitlement Servlet sample as explained here.

**Enabling Java EE 6 Web Profile**

Starting from WSO2 AS 5.3.0, AS supports Java EE 6 Web Profile. This support is provided using Apache TomEE, which is a project developed at Apache Software Foundation, and is a Java EE 6 Web Profile certified stack. With the Java EE runtime environment, you can now deploy Java EE web applications (that uses Apache TomEE) in WSO2 AS.

The following configurations should be in place in order to use this feature:

- Enabling the Java EE runtime
- Configuring TomEE

**Enabling the Java EE runtime**

WSO2 leverages a concept called runtime environments to maintain separate runtimes. You can enable the following runtime environments in WSO2 AS: Carbon, Tomcat, CXF and Java EE. See the topic on configuring runtime environments for more descriptive information. Shown below is the configuration to specify 'Javaee' as the runtime environment.

1. Open the webapp-classloading.xml file stored in the META-INF folder of your web application. This folder should be created inside the src/main/webapp folder of your web app. For example, check the webapp-classloading.xml file of a sample web application (javaee-examples) stored in the src/main/webapp/META-INF folder.
2. Enter the following class loading details to enable the Java EE runtime:

   ```
   <Classloading xmlns="http://wso2.org/projects/as/classloading">
     <Environments>Javaee</Environments>
   </Classloading>
   ```

3. You can add other runtimes in addition to Java EE, by entering a comma separated list for the `<Environments>` element.

4. The support for Javaee is provided along with Apache TomEE. If you use the Javaee runtime, then TomEE should also be configured in AS as explained below.

**Configuring TomEE**

The following configurations are required if the Java EE runtime is enabled for your AS as explained above. There are two Apache TomEE configurations:

- Setting the default TomEE configurations:

  1. Open the tomee.xml file located in the `<AS_HOME>/repository/conf/tomee` directory.

  2. In this file, you can configure theDatasources, Resources etc. See the following example where the "myDerbyDatasource" datasource is configured as a resource.
3. See the documentation on Apache TomEE configurations for more examples on how to configure the tomee.xml file.

- Setting the jvm system properties that will be used by TomEE:
  1. Open the system.properties file located in the <AS_HOME>/repository/conf/tomee directory.
  2. Update the required properties. For a comprehensive list of system properties, see http://tomee.apache.org/properties-listing.html.

Working with Services

The following topics explain the how you can create axis2 modules and use them in WSO2 AS to extend the server:

- Creating an Axis2 Module
- Deploying Axis2 Modules
- Managing Dependencies and Common Libraries

Creating an Axis2 Module

Axis2 provides extended support for modules. For details on modules in Axis2, refer to the Architecture Guide.Shown below is how a simple logging module is developed. This module contains one handler that just logs the message that is passed through it. Axis2 uses MAR (Module Archive) to deploy modules in Axis2.

The following steps explain the process of creating and packaging an axis2 module:

- Step 1: Creating the Module Implementation
- Step 2: Creating the Handlers
- Step 3: Creating the module.xml
- Step 4: Updating the axis2.xml
- Step 5: Engaging the module
  - Engaging the module at service level
  - Engaging the module at operation level
  - Engaging the module globally
- Step 6: Packaging in a MAR (Module Archive)

**Step 1: Creating the Module Implementation**

LoggingModule is the implementation class of the Axis2 module. Axis2 modules should implement the org.apache.axis2.modules.Module interface with the following methods:
public void init(ConfigurationContext configContext, AxisModule module) throws AxisFault; //Initialize the module
public void shutdown(ConfigurationContext configurationContext) throws AxisFault; //End of module processing
public void engageNotify(AxisDescription axisDescription) throws AxisFault;
public void applyPolicy(Policy policy, AxisDescription axisDescription) throws AxisFault;
public boolean canSupportAssertion(Assertion assertion);
```java
public class LogHandler extends AbstractHandler implements Handler {
    private static final Log log = LogFactory.getLog(LogHandler.class);
    private String name;

    public String getName() {
        return name;
    }

    public InvocationResponse invoke(MessageContext msgContext) throws AxisFault {
        log.info(msgContext.getEnvelope().toString());
        return InvocationResponse.CONTINUE;
    }

    public void revoke(MessageContext msgContext) {
        log.info(msgContext.getEnvelope().toString());
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

**Step 3: Creating the module.xml**

The `module.xml` contains the deployment configurations for a particular module. It contains details such as the implementation class of the module (in this example it is the `LoggingModule` class and various handlers that will run in different phases). The `module.xml` for the logging module will be as follows:
There are four flows defined in the module.xml:

- **InFlow** - Represents the handler chain that will run when a message is coming in.
- **OutFlow** - Represents the handler chain that will run when the message is going out.
- **OutFaultFlow** - Represents the handler chain that will run when there is a fault, and the fault is going out.
- **InFaultFlow** - Represents the handler chain that will run when there is a fault, and the fault is coming in.

The following set of tags describes the name of the handler, handler class, and the phase in which this handler is going to run.

- **handler**
  - **name** - Is the name given for the particular instance of this handler class.
  - **class** - Is the actual implementation class for this handler. Since we are writing a logging handler, we can reuse the same handler in all these phases. However, this may not be the same for all the modules.
- **order**
  - **phase** - Describes the phase in which this handler runs.
To learn more about Phase rules, refer to the article Axis2 Execution Framework: [http://www.developer.com/java/web/article.php/3529321](http://www.developer.com/java/web/article.php/3529321)

**Step 4: Updating the axis2.xml**

In this handler the loggingPhase is defined by the module writer. It is not a predefined handler phase, hence the module writer should introduce it to the axis2.xml (not the services.xml) so that the Axis2 engine knows where to place the handler in different flows (inFlow, outFlow, etc.).

The following XML lines show the respective changes made to the axis2.xml in order to deploy the logging module in the "Axis2" engine. This is an extract of the phase section of axis2.xml.

```xml
<phaseOrder type="inflow">
    <!-- System pre defined phases -->
    <phase name="TransportIn"/>
    <phase name="PreDispatch"/>
    <phase name="Dispatch"
        class="org.apache.axis2.engine.DispatchPhase">
        <handler name="AddressingBasedDispatcher"
            class="org.apache.axis2.dispatchers.AddressingBasedDispatcher">
            <order phase="Dispatch"/>
        </handler>
        <handler name="RequestURIBasedDispatcher"
            class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher">
            <order phase="Dispatch"/>
        </handler>
        <handler name="SOAPActionBasedDispatcher"
            class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher">
            <order phase="Dispatch"/>
        </handler>
        <handler name="SOAPMessageBodyBasedDispatcher"
            class="org.apache.axis2.dispatchers.SOAPMessageBodyBasedDispatcher">
            <order phase="Dispatch"/>
        </handler>
        <handler name="InstanceDispatcher"
```
<class="org.apache.axis2.engine.InstanceDispatcher">
  <order phase="PostDispatch"/>
</phase>

<!-- System pre defined phases -->
<!-- After Postdispatch phase module author or service author can add any phase he wants -->
  <phase name="OperationInPhase"/>
  <phase name="loggingPhase"/>
</phaseOrder>

<phaseOrder type="outflow">
  <!-- user can add his own phases to this area -->
  <phase name="OperationOutPhase"/>
  <phase name="loggingPhase"/>
  <!-- system predefined phases -->
  <!-- these phases will run irrespective of the service -->
  <phase name="PolicyDetermination"/>
  <phase name="MessageOut"/>
</phaseOrder>

<phaseOrder type="INfaultflow">
  <!-- user can add his own phases to this area -->
  <phase name="OperationInFaultPhase"/>
  <phase name="loggingPhase"/>
</phaseOrder>

<phaseOrder type="Outfaultflow">
  <!-- user can add his own phases to this area -->
  <phase name="OperationOutFaultPhase"/>
  <phase name="loggingPhase"/>
  <phase name="PolicyDetermination"/>
</phaseOrder>
The custom phase loggingPhase is placed in all the flows, hence that phase will be called in all the message flows in the engine. Since the module is associated with this phase, the LogHandler inside the module will now be executed in this phase.

Step 5: Engaging the module

Up to this point, we have created the required classes and configuration descriptions for the logging module and the required phases. Next step is to "engage" (use) this module. You can engage the module at three levels:

- Engaging the module at service level
- Engaging the module at operation level
- Engaging the module globally

Engaging the module at service level

The following example shows how the module is engaged for the example MyService. It is necessary to modify the services.xml of MyService in order to engage this module. The code for the service can be found in the Axis2_HOME/samples/userguide/src/userguide/example2 directory.

```xml
<service name="MyServiceWithModule">
  <description>
    This is a sample Web service with a logging module engaged.
  </description>

  <module ref="logging"/>

  <parameter name="ServiceClass" locked="xsd:false">userguide.example2.MyService</parameter>
  <messageReceiver class="org.apache.axis2.receivers.RawXMLINOutMessageReceiver"/>
  <operation name="echo"/>
  <operation name="ping"/>
</service>
```

In this example, the service name was changed. In addition, the line <module ref="logging"/> was added to services.xml. This informs the "Axis2" engine that the module logging should be engaged for this service. The handler inside the module will be executed in their respective phases as described by the module.xml.

Engaging the module at operation level

Modules can be engaged for selected operations in a service by updating the services.xml of the service. The following example shows how the "logging" module is engaged for the operation named "listModulesForOperation" in a service.
<operation name="listModulesForOperation">
  <parameter name="AuthorizationAction" locked="false">
    /permission/admin/manage/modify/service</parameter>
  <module ref="logging"/>
</operation>

Engaging the module globally

In order to globally engage modules in WSO2 products that are based on Carbon 4.4.1 or a later version, you need to update the `<AS_HOME>/repository/conf/axis2/axis2.xml` file with the following:

```
<module ref="logging"/>
```

Note that the "Addressing" module is engaged by default in the `axis2.xml` as shown below.

```
<module ref="addressing"/>
```

**Step 6: Packaging in a MAR (Module Archive)**

Before deploying the module, it is necessary to create the MAR file for this module. This can be done using the JAR command and then renaming the created JAR file. Else you can find the `logging.mar` that has already been created in the `Axis2_HOME/samples/userguide` directory.

**Deploying Axis2 Modules**

The WSO2 SOA platform inherits the capabilities of Axis2, which allows the platform to be extended using modules. After writing a module, you can add it to a Carbon server. The global modules affect all the services deployed within the server.

If you are deploying a module in Axis2 manually, you must create a directory with the name "modules" in the `webapps/axis2/WEB-INF` directory of their servlet container, and then copying the MAR file to that directory. For more information, refer to [http://axis.apache.org/axis2/java/core/docs/modules.html](http://axis.apache.org/axis2/java/core/docs/modules.html)

Follow the instructions below to upload a Module to the server using the management console:

```
The Axis2 module management capability is provided by the following feature in the WSO2 feature repository:

Name: WSO2 Carbon - Module Management UI Feature
Identifier: org.wso2.carbon.module.mgt.ui.feature.group

Name: WSO2 Carbon - Module Management Core Feature
Identifier: org.wso2.carbon.module.mgt.server.feature.group

The module management UI and server features above are currently bundled in the service management feature given below:

Name: WSO2 Carbon - Service Management Feature
Identifier: org.wso2.carbon.service.mgt.feature.group
```
The service management feature is added by default in all service hosting WSO2 middleware products. If it is not included in the product distribution you are using, you can install it by following the instructions given in the Feature Management section.

1. Log in to the product's management console and click Add under Main -> Modules.

2. Click Browse to select the module archive file (MAR) you want to upload. For example,

   ![Add modules](image)

   Tip
   
   Use the "+" button to upload more than one module. The "-" button removes the additional upload option.

3. Click Upload. If the file is uploaded successfully, a message appears prompting you to restart the server to activate the module. Click OK.

4. When the server restarts, the module will be active and displayed on the Deployed Modules page.

**Managing Dependencies and Common Libraries**

This chapter contains the following information:

- Class Loading for Web Services
- Uploading Shared Axis2 Artifacts
- Restarting the Axis2 Configuration
- Deleting Common Axis2 Libraries
Class Loading for Web Services

In most scenarios, web services that are deployed in production environments depend on external, third-party libraries for different functionality. The web services deployed in WSO2 Application Server can load classes from different locations. Therefore, it is important to understand how you can place your external dependencies within WSO2 Application Server in the manner most suitable to your requirements.

There are different kinds of services that you can deploy in WSO2 Application Server through the management console. See Service Development and Deployment for more information. These deployed service archives are written to the relevant folder in the following location of the file system: `{AS_HOME}/repository/deployment/server`. In this folder, there are separate sub folders (deployment folders) for the different types of services. For example, axis2 services are deployed in `{AS_HOME}/repository/deployment/server/axis2services`.

See the following topics for information on how classes are loaded to your web services and about the possible class loading configurations:

- Classes for Web Services
- Child-first class loading
- Parent-first class loading
- Change class loading for a particular service

Classes for Web Services

The web services deployed in your Application Server can load classes from four different locations as explained below:

- `{PRODUCT_HOME}/repository/components/lib`
  When a .jar file is copied to this location, it will be converted to an OSGi bundle and injected to the Carbon OSGi runtime. Any other OSGi bundle can import packages provided by the copied .jar file. We recommend users to copy external libraries to this location when it is necessary to make the dependency available within the OSGi environment. For example, consider a situation where you have installed mediation features in the WSO2 Application Server and want to share the same library between a mediator and an AAR service.

  If your external dependency is already an OSGi bundle, you can directly copy that bundle to `{AS_HOME}/repository/components/dropins` folder and the behavior will be the same as above.

- `{PRODUCT_HOME}/repository/deployment/server/lib`
  As mentioned above, all sub folders relevant to different service types exist in the "server" folder. So the 'lib' folder is the place to copy libraries which should be shared across different services of different service types.

    You have the option to use the management console of AS in order to upload the axis2 artifacts to this directory as explained here. This possibility is enabled in AS by the Axis2 Repository Management feature that is enabled in AS by default.

- `{PRODUCT_HOME}/repository/deployment/server/<deployment_folder>/lib`
  The `{deployment_folder}` here is the relevant folder for the service type (For example, `{axis2services}`). This 'lib' folder is the place to copy libraries that should be shared across different services of the same service type. For example, if you have two different Axis2 services that share the same dependency, you can
You have the option to use the management console of AS in order to upload the axis2 artifacts to this directory as explained here. This possibility is enabled in AS by the **Axis2 Repository Management** feature that is enabled in AS by default.

- **The 'lib' folder inside the service archive file**
  You can include your external libraries to the service archive file itself. Create a 'lib' folder on the top level of your archive file and include the dependency inside that folder. This is recommended when the external library is used only by the particular service. For example, if your .aar service depends on a .jar, embed the .jar file inside the .aar file as shown below.

  ```
  StudentMarks.aar
  META-INF
  services.xml
  lib
  foo.jar
  org
  wso2
  sample
  marks.class
  ```

**Child-first class loading**

The WSO2 Application Server follows child-first class loading by default. You can see that by opening the `<AS_HOME>/repository/conf/axis2.xml` file and checking the following parameter.

```
<parameter name="EnableChildFirstClassLoading">true</parameter>
```

This means that the priority of class-loading options mentioned above reduces in the following order: 4 > 3 > 2 > 1. When loading classes, highest priority is given to location 4 above and the least to location 1. For example, if you have two different versions of the same library in 1 and 3 above and your .aar service depends on that library, classes will be loaded from the library in location 3.

**Parent-first class loading**

If you want to enable parent-first class loading, you can do that by opening the `<AS_HOME>/repository/conf/axis2.xml` file and setting the value of the following parameter to 'false'.

```
<parameter name="EnableChildFirstClassLoading">false</parameter>
```

This means that the priority of class-loading options mentioned above reduces in the following order: 1 > 2 > 3 > 4. When loading classes, the highest priority is given to location 1 above and the least to location 4. For example, if you have two different versions of the same library in 1 and 3 above and your .aar service depends on that library, classes will be loaded from the library in location 1.

**Change class loading for a particular service**
If you want a particular Axis2 service to have a different setting for the "EnableChildFirstClassLoading" parameter, you can add the above parameter to the META-INF/services.xml file of the .aar file.

```xml
<parameter name="EnableChildFirstClassLoading">false</parameter>
```

For example, consider that child-first class loading is enabled at server level (default setting), and therefore the priority of class-loading options reduces in the following order: 4 > 3 > 2 > 1. In this scenario, if you have two different versions of the same library in locations 1 and 4, and your service depends on the library in location 4, the classes will be loaded for the service from the library in location 4 as specified in the META-INF/services.xml file.

You can only control the behaviour of location 4 using this method. Therefore, use this method only when you have embedded your libraries inside the service archive.

**Uploading Shared Axis2 Artifacts**

Axis2 services can be deployed to a running Carbon instance as archive files, given that the required features are installed. See the section on developing and deploying services in AS for step-by-step instructions. Some of these Axis2 service archives may require additional libraries. As explained in the section on class loading for services, these external libraries can be applied to services at individual service level or they can be shared across multiple services.

With the **Axis2 Repository Management Feature** that is installed in WSO2 AS by default, you are able to directly upload the common libraries to the Axis2 Repository using the management console. You can also remove uploaded/shared libraries that are no longer necessary.

Follow the instructions below to upload shared libraries.

1. Log in to the product's management console and click **Carbon** under **Configure -> Repositories**.
2. The **Artifact Dependency Manager** page opens. To upload shared libraries, click **Add dependencies** link next to the relevant lib directory icon.

**Artifact Dependency Manager**

![Artifact Dependency Manager](image)

This uploads libraries that can be used by Axis2 artifacts that are deployed under the Parent directory of that particular lib directory. For example, libraries uploaded to the lib directory of axis2services can also be shared by Axis2 artifacts deployed under axis2services directory.
3. The artifact upload page appears. Browse for the .jar file and click **Upload** as shown below.

![Artifact Dependency Manager](image)

4. If the file is uploaded successfully, a message will appear asking if you want to restart Axis2 Configuration now. Click **Yes** and similarly confirm the message that follows.

![WSO2 Carbon](image)

5. The Axis2 configuration is restarted and the following message is displayed. Click **OK**.

![WSO2 Carbon](image)

6. You can see the uploaded .jar file in the lib folder.
Restarting the Axis2 Configuration

In order to load the newly uploaded common libraries, the Axis2 configuration needs to be restarted. If a restart is required, a message will be displayed asking you to restart the configuration.

1. Simply click the link to restart the Axis2 Configuration.

2. A message appears prompting you to confirm the restart. Click "Yes."

The restart option would be available only if the server was not restarted after uploading the libraries via Axis2 Repository Manager.

Deleting Common Axis2 Libraries

Follow the instructions below to delete shared libraries.

1. Log on to the product's management console and select "Carbon" under "Configure -> Repositories."
2. The "Artifact Dependency Manager" page opens. Click the delete link associated with the artifact you want to delete.

3. Confirm the message that is prompted.

**Upgrading from a Previous Release**

The following steps describe how to upgrade WSO2 Application Server from version 5.2.1 to 5.3.0. To upgrade from a version older than 5.2.1, start from the documentation that was released immediately after your current release and upgrade incrementally. For more information on release versions, see the Release Matrix.

If you want to migrate your WSO2 AS configurations from one instance to another (such as when promoting your instance from test to production) using the same AS version, see Migrating the Application Server.
The instructions on this page take you through the steps for upgrading from AS 5.2.1 to AS 5.3.0. This includes upgrading the database changes as well as the configurations that should follow the upgrade.

Preparing to upgrade

The following prerequisites must be completed before upgrading:

- Make a backup of the user and registry databases of AS 5.2.1
- Make a backup of <AS_HOME_5.2.1> folder to backup the product configurations.
- Download WSO2 Application Server 5.3.0 from http://wso2.com/products/application-server/.

You cannot rollback the upgrade process. However, it is possible to restore a backup of the previous database and restart the upgrade progress.

Upgrading the database/registry

There are no registry and user database schema changes between AS 5.2.1 and 5.3.0. Therefore, you do not need to do any database schema migration.

Migrating the configurations

The following topics explain the configuration changes that need to be updated for AS 5.3.0:

- Updating the configuration files
- Migrating third party libraries
- Migrating the services and artifacts

Updating the configuration files

The following are the updates that need to be done to the configuration files in AS 5.3.0.

Note that configuration files should not be copied directly between servers.

1. Create a new database for AS 5.3.0 and restore the backup of the old database in this new database.
2. To connect AS 5.3.0 to the upgraded database, configure the following files:
   1. Configure the <AS_HOME_5.3.0>/repository/conf/datasources/master-datasources.xml file as shown in the following example:
<datasource>
user manager</description>
<name>WSO2_CARON_DB</name>
<description>The datasource used for registry and</description>
<jndiConfig>
<name>jdbc/WSO2CarbonDB</name>
</jndiConfig>
<definition type="RDBMS">
<configuration>
<url>jdbc:mysql://localhost:3306/new_database</url>
<username>username</username>
<password>password</password>
<driverClassName>com.mysql.jdbc.Driver</driverClassName>
<maxActive>80</maxActive>
<maxWait>60000</maxWait>
<minIdle>5</minIdle>
<testOnBorrow>true</testOnBorrow>
<validationQuery>SELECT 1</validationQuery>
<validationInterval>30000</validationInterval>
</configuration>
</definition>
</datasource>

2. Go to the <AS _HOME_5.3.0>/repository/conf/ directory and update the datasource references
in the user-mgt.xml and registry.xml files to match the updated configurations in the master-datasources.xml file. The following are sample configurations if the datasource is "jdbc/WSO2CarbonDB":

registry.xml

<dbConfig name="wso2registry">
<dataSource>jdbc/WSO2CarbonDB</dataSource>
</dbConfig>

user-mgt.xml

<UserManager>
<Realm>
<Configuration>
...
<Property name="dataSource">jdbc/WSO2CarbonDB</Property>
</Configuration>
...
</Realm>
</UserManager>
3. Data services feature is no longer shipped with WSO2 Application Server. If you need to use the data services hosting feature, you have two options:
   1. Install the Data Services hosting feature from the public p2-repo. See Installing Features for instructions.
   2. Use the WSO2 Data Services Server product which can be downloaded from http://wso2.com/products/data-services-server/. See the About this Release page to check if the latest WSO2 DSS is compatible with the current AS version.

4. The configurations for SaaS web applications has changed in AS 5.3.0. In previous releases, SaaS configurations were enabled in the web.xml file of the web application by adding a context-param called carbon.enable.saas. In AS 5.3.0, SaaS is configured via the context.xml file that needs to be placed under the META-INF/ folder of the web application. See more details about configuring SaaS applications in AS 5.3.0 from here.

5. Prior to AS 5.3.0, the primary keystore configured in the carbon.xml file was used for securing transports. In AS 5.3.0, the keystore used for transports should be separately configured in the catalina-server.xml file.

   The “RegistryKeyStore” configuration in carbon.xml is removed. See the section on configuring keystores for more information.

6. Check for any other configurations that were done for AS 5.2.1 (based on your solutions), and update the configuration files in AS 5.3.0 accordingly. For example, external user stores, caching, mounting, etc.

Migrating third party libraries

If there are third party libraries used with AS 5.2.1 that you want to migrate, copy them to the following directories in AS 5.3.0 as applicable.

   1. If you have used JDBC drivers etc, copy them to the <AS_HOME>/repository/components/lib directory.
   2. If you have used OSGi bundles such as SVNKit etc, copy them to the <AS_HOME>/repository/components/dropins directory.

Migrating the services and artifacts

You can migrate all artifacts relevant to the super tenant as well as the ordinary tenants by copying the following directories from the old server to the new server.

   1. To migrate the super tenant’s artifacts, copy the <AS_HOME>/repository/deployment/server/ directory from AS 5.2.1 to AS 5.3.0.
   2. If you are using multi-tenancy, copy the <AS_HOME>/repository/tenants directory from AS 5.2.1 to AS 5.3.0.
   3. Since the Axis2 Quality of Services UI to apply policies such security has been removed from the management console, users need to use alternative mechanisms and apply the policies for their services. For Axis2 AAR services, the policies can be applied through the services.xml.

   In AS 5.3.0, it is not possible to globally engage modules using the management console. Therefore, you need to update the axis2.xml file in the <AS_HOME>/repository/conf/axis2 directory as shown below. You can find more information on engaging modules for axis2 services from here.
<axisconfig name="AxisJava2.0">
    <module ref="addressing"/>
    ....
</axisconfig>

Testing the upgrade

1. When the database upgrade scripts are executed, the following are some of the new tables that will be created in the database:
   - UM_DOMAIN
   - UM_SYSTEM_USER
   - UM_SYSTEM_ROLE
   - UM_SYSTEM_USER_ROLE
2. Verify that all the required scenarios are working as expected as shown below. This confirms that the upgrade is successful.
   1. Start the AS 5.3.0 server instance once the configurations are done.
   2. Make sure that the server starts up fine without any errors.
   3. Test the deployed artifacts:
      1. Log in to the management console as the super tenant.
      3. Verify the web application list shown there.
      4. Invoke a web application to verify that it works.
      5. Then, navigate to Main -> Services -> List.
      6. Verify the services list shown there.
      7. Invoke a service to verify that it works.
      8. Then, navigate to Main -> Carbon Applications -> List.
      9. Verify the CApp list shown there.
   4. Verify that the Users and Roles are picked up:
      1. Navigate to Configure -> Accounts & Credentials -> Users and Roles
      2. Verify that the list of users and roles are shown correctly.
      3. View the permissions of a chosen role, and make sure that the permissions are correct.
      5. If you are using multitenancy,
         1. Log in to the management console using the super tenant credentials.
         2. Navigate to Configure -> View Tenants.
         3. Verify the tenant list shown there.
         4. Then, log in to the system as a tenant and make sure that the log in is successful.
Developer Guide

This section describes the developer aspect of how to work with WSO2 Application Server.

- Working with the Source Code
- Basic Registry API Knowledge
- CarbonContext API
- WSO2 Patch Application Process
- Customizing the Management Console
- Enabling SPI Fly Support
- Java Documentation

Working with the Source Code

The source code of WSO2 Application Server as well as all other WSO2 products are maintained in GitHub repositories. If you are a developer, you can clone the source code from these Git repositories, and if required, you can do modifications and build a customized product on your own. WSO2 invites interested parties to contribute their changes and suggestions to WSO2.

See the documentation on working with the source code for more information.

Basic Registry API Knowledge

This section describes the basic functions of the Registry API. For example:

- Reading a resource from the Registry
- Adding a resource to the Registry
- Renaming a resource, etc.

You can do all this through the Management Console. See Working with the Registry to learn how to manage the AS Registry using the UI.

Visit the following pages to learn more information about the basic Registry functions:

- Adding a Resource with API
- Creating a Resource from a Given URL
- Checking for the Existence of a Resource
- Deleting a Resource
- Managing Commenting
- Managing Ratings
- Managing Tags of a Resource
- Managing Versions of a Resource
- Reading a Resource
- Renaming a Resource
- Retrieving Metadata of a Resource
- Moving and Copying a Resource
- Managing Relationships of a Resource
- Executing a Query
- Searching for Content
- Retrieving Logs
- Managing Aspects
- Managing Symbolic Links

Adding a Resource with API

Add a resource to the registry instance
To add a resource to the registry instance, create a Resource object and then pass the created resource object to the registry instance’s `put` method:

1. Create a collection called `/c1/c2`

```java
Collection collection = registry.newCollection();
```

2. Call the `put` method.

```java
registry.put("/c1/c2", collection);
```

**Access the created resource**

If you call the `get` method, then you will be able to access that created resource.

**Add a resource with content**

To add a resource with content, create a Resource object and then set content.

```java
Resource r1 = registry.newResource();
String str = "My File Content";
r1.setContent(str.getBytes());
registry.put("/c1/c2/r1", r1);
```

**Related Links**

*Adding a Resource* - For information on adding a resource using the Management Console.

**Creating a Resource from a Given URL**

If a resource needs to be created with content from any given URL, the following line of code can be used to add a resource from a given URL to a suggested location. The real location to which the resource was added is returned. The `template_resource` should contain *metadata of the resource* that a user wants to add.

```java
String real_location = registry.importResource("c1/c2/r3",
"http://wso2.com", template_resource);
```

*metadata of the resource* is defined as the properties of the resource, media type of the resource, author of the resource, create date of the resource, last updates time of the resource, etc.

**Checking for the Existence of a Resource**

We can use the following code to confirm whether the resource exists:
boolean value = registry.resourceExists("/c1/c2/r1");

If the resource does exist, a boolean value of true will be returned and if it does not exist, the false value is returned.

Deleting a Resource

Additionally, you can use the registry instance to delete resources. Deleting a resource is a matter of calling the delete method of the registry. For example, to delete the /c1/c2/r1 resource, use the following code:

```
registry.delete("/c1/c2/r1");
```

Related Links

Managing Entries or Content - For information on deleting a resource using the Management Console.

Managing Commenting

Commenting on a resource

You can also comment on a resource using the registry. The following lines of code illustrate how to create a comment object and call the registry instance:

```
Comment c1 = new Comment();
c1.setText("This is my comment");
String commentPath = registry.addComment("/c1/c2/r2", c1);
```

The above lines of code will add a comment to the resource named: c1/c2/r2

Edit a comment

You can also make changes to comments you have already made using the registry instance. For this you need the path and new text for the comment that is being added. For example, to change from "This is my comment" to "This is cool," do the following:

```
registry.editComment(commentPath, "This is cool"); // Here commentPath have a value something like "/c1/c2/r2;comment:1"
```

Remove a comment
A comment can be removed using the comment path. The following code shows how this is done:

```java
registry.removeComment(commentpath);
```

**Retrieving comments made about a resource**

Comments about a specific resource can be retrieved using the following code:

```java
registry.getComments("/c1/c2/r2");
```

### Managing Ratings

**Rating a resource**

To rate a resource based on our judgment, you can use the registry instance. Rating a resource can be done using the following line of code:

```java
registry.rateResource("c1/c2/r2", 4);
```

**Retrieving the average rating of a resource**

An average of all ratings of a particular resource can be retrieved. This can be done using the following line of code:

```java
registry.getAverageRating("c1/c2/r2");
```

**Retrieving the rating given to the resource by a particular user**

When a rating given by a particular user for a resource is needed, it can be done using the following line of code:

```java
registry.getRating("c1/c2/r2", "wso2_admin"); // rating given for r2 by wso2 admin
```

### Managing Tags of a Resource

**Adding a tag**

```java
registry.getTag("c1/c2/r2", "wso2_admin");
```
You can perform tagging operations using the registry. To tag a resource, you need the resource path and the tagging words. For example, to tag a resource named /c1/c2/r2 as "rename resource," do the following:

```java
registry.applyTag("/c1/c2/r2", "rename resource");
```

**Retrieving all tags of a given resource**

You can use the registry to retrieve tags for a given resource. It will return an array of type `Tag` and you can iterate the array to see the content.

```java
Tag[] tags = registry.getTags("/c1/c2/r2");
```

**Deleting a tag**

You can remove a tag using the tag name. Use the following code:

```java
registry.removeTag("/c1/c2/r2","rename resource");
```

**Retrieving resources with a specific tag**

Resources that have a specific tag can be retrieved using the following code:

```java
registry.getResourcePathsWithTag("wso2");
```

**Managing Versions of a Resource**

**Enabling auto versioning of resources**

Resources can be automatically versioned when they are added or updated. However, this feature is disabled by default. It can be enabled by adding following to `PRODUCT_HOME/repository/conf/registry.xml`

```xml
<versionResourcesOnChange>true</versionResourcesOnChange>
```

**Creating versions of resources**

Collections are not versioned automatically due to performance considerations. However, you can create versions of collections or resources using the API. It is recommended that collection versions are created only for making checkpoints of the sub-tree you are working on. When connecting to the server from the client side, it is possible to create versions.
Use the following command to create a version of your resource/collection:

```java
registry.createVersion("/c1/c2");
```

---

**Retrieving versions of a given resource**

You can list all of the versions of a given resource using the code given below. The result would be an array of String type, containing links to the different versions of the resource.

```java
String [] versions = registry.getVersions("/c1/c2");
```

---

**Restoring to an old version**

Since the Registry comes with versioning, we can restore a resource to any of its versions. This can be done using the registry as well. In the Retrieving versions of a given resource section, we discussed how to retrieve versions for a given resource. The following line of code demonstrates how we can restore back an old version of the registry instance. Note that in the process of restoration, the content, properties, comments, tags, ratings, associations and dependencies will be restored, but subscriptions and permissions related to resources will not be restored.

```java
registry.restoreVersion("/c1/c2;version:2");
```

---

**Related Links**

Managing Metadata - for information on checking versions using the Management Console.

**Reading a Resource**

Once the Registry instance is in place (embedded registry or remote registry), navigation is straightforward. It is the same as working with the Registry API. If we have a resource called /foo/c1 in the Registry, you can access it using the Registry instance as given below:

```java
Resource resource = registry.get("/foo/c1");
```

The resource object will represent the actual resource object in the Registry. However, if the resource is a collection, then the object will represent the collection.

Once we have a registry instance, we do not need to pass a complete URL for all invocations. It is adequate to pass only a relative path to a resource.

---

**Related Links**

Managing Entries or Content - for information on how to read resources via the Management Console.
Renaming a Resource

You can rename individual resources or collections. To rename a resource, use the following lines of code. The same code can be used to rename a collection as well.

```java
registry.rename("/c1/c2/r1", "/c1/c2/r2");
```

The above line of code renames the resource `/c1/c2/r1` to `/c1/c2/r2`.

Retrieving Metadata of a Resource

To retrieve metadata associated with a resource, such as author, created date, etc. We can use the registry instance with the following line of code:

```java
registry.getMetaData("c1/c2/r2");
```

Moving and Copying a Resource

Moving a resource | Copying a resource

Moving a resource

If a resource needs to be moved, the following code can be used:

```java
registry.move("c1/c2/r3", "c1/c2/r4");
```

Copying a resource

If a resource needs to be copied, the following code can be used:

```java
registry.copy("c1/c2/r3", "c1/c2/r4");
```

Related Links

Managing Entries or Content - For more information on moving and copying a resource via the Management Console.

Managing Relationships of a Resource

Creating a relationship between two resources | Removing a relationship between two resources | Retrieving relationships from a resource
Creating a relationship between two resources

A relationship (for example, dependency) can be created between two resources using the following line of code:

```java
registry.addAssociation("c1/c2/r3", "c1/c2/r4", "depends on"); // in this case r3 depends on r4
```

Removing a relationship between two resources

A relationship can be removed between two resources using the following line of code:

```java
registry.removeAssociation("c1/c2/r3", "c1/c2/r4", "depends on"); // in this case r3 depends on r4
```

Retrieving relationships from a resource

Relationships of a resource between two resources can be retrieved using the following line of code:

```java
registry.addAssociation("c1/c2/r3", "c1/c2/r4", "depends on"); // in this case r3 depends on r4

Association all[] = registry.getAllAssociations("c1/c2/r3"); // all associations of r3
String sourcePath1 = all[0].getSourcePath(); // equals to "c1/c2/r3"

Association dependsAll[] = registry.getAssociations("c1/c2/r3", "depends on"); // "depends on" associations of r3
String sourcePath2 = dependsAll[0].getSourcePath(); // equals to "c1/c2/r3"

Association r4Assoc[] = registry.getAssociations("c1/c2/r4", "depends on"); // "depends on" associations of r3
boolean hasAssociations = (r4Assoc.length == 0); // this returns true
```

Executing a Query

A custom query stored in the registry can be executed by giving the path of the stored query and giving a Map of the required parameters. For example:

```java
registry.executeQuery("/path/to/storedQuery", parameter_map); // parameter map contains key value pairs required by the query
```

Searching for Content
A search for content can be launched by passing the required keywords as parameters in the following code:

```java
registry.searchContent("WSO2 Application Server"); // searches through resource content for WSO2 Application Server
```

### Retrieving Logs

Logs can be retrieved based on username, time period, action performed, and specific resource. For this purpose, the method `getLogs` returns an array while the method `getLogCollection` returns a collection.

```java
LogEntry[] logentries = registry.getLogs("c1/c2/r3", LogEntry.ALL, "admin", new Date(2010,4,7), new Date(2010,4,12), true);

LogEntryCollection leCollection = registry.getLogCollection("c1/c2/r3", LogEntry.ALL, "admin", new Date(2010,4,7), new Date(2010,4,12), true);
```

### Managing Aspects

#### Retrieve available aspects

The available aspects in the Registry can be retrieved through the following code. It returns a string array containing the names of the aspects:

```java
String[] aspects = registry.getAvailableAspects();
```

#### Adding a new aspect

An aspect is a way of associating custom behaviors with resources. The Registry is shipped with an implemented `Aspect` known as `SimpleLifeCycle` by default. The following code shows how this aspect can be added to the registry:

```java
Aspect simpleLifeCycle = new SimpleLifeCycle();
registry.addAspect("SimpleLifeCycle", simpleLifeCycle);
```

#### Associating an aspect with a resource

An aspect can be associated with a Resource. The following code associated the `SimpleLifeCycle` to be associated with a resource:
### Invoking an aspect

An aspect needs to be invoked with the appropriate actions to make use of its custom behavior. The following code invokes the `SimpleLifeCycle` to be associated with a resource:

```java
registry.invokeAspect("c1/c2/r3", "SimpleLifeCycle", SimpleLifecycle.ACTION);
```

### Retrieving actions of an aspect

An action associated with an aspect with respect to a particular resource can be retrieved using the following line of code:

```java
registry.getAspectActions("c1/c2/r3", "SimpleLifeCycle");
```

### Removing an aspect from registry

An aspect can be removed from the registry with the following line of code:

```java
registry.removeAspect("SimpleLifeCycle");
```

### Managing Symbolic Links

#### Creating a symbolic link/mounting a registry

A symbolic link to a registry can be created in the following manner. The same code can be used to mount a registry to a specific location. If a remote registry needs to be mounted, the same method can be used with an extra parameter.

```java
registry.createLink("/config", "/_system/config");

registry.createLink("/config", "/_system/config", "http://remote.registry.url/`);```

#### Removing a symbolic link/mount point of a registry

A symbolic link or a mount point of a registry created above can be removed using the following line of code:

```java
```
registry.removeLink("/config");

Related Link

**Link Creation** - For more information on link creating via the Management Console.

**CarbonContext API**

- **PrivilegedCarbonContext**
  - Obtaining the PrivilegedCarbonContext
    - getThreadLocalCarbonContext()
  - Setting information into PrivilegedCarbonContext
    - setTenantId(int tenantId)
    - setTenantId(int tenantId, boolean resolveTenantDomain)
    - setUsername(String username)
    - setTenantDomain(String tenantDomain)
    - setTenantDomain(String tenantDomain, boolean resolveTenantId)
    - setApplicationName(String applicationName)
    - setUserRealm(UserRealm userRealm)
    - setRegistry(RegistryType type, Registry registry)
  - Switching tenant flows
    - PrivilegedCarbonContext.startTenantFlow()
    - PrivilegedCarbonContext.endTenantFlow()
  - Getting Information from PrivilegedCarbonContext
    - getTenantDomain(boolean resolve)
    - getTenantId(boolean resolve)
    - getRegistry(RegistryType type)
    - Object getOSGiService(Class clazz)
    - List<Object> getOSGiServices(Class clazz)
  - Miscellaneous Methods
    - public static void unloadTenant(int tenantId)
    - public static void destroyCurrentContext()

- **CarbonContext**
  - Obtaining the CarbonContext
    - getThreadLocalCarbonContext()
  - Retrieving information
    - public int getTenantId()
    - String getUsername()
    - String getTenantDomain()
    - Registry getRegistry(RegistryType type)
    - UserRealm getUserRealm()
    - CarbonQueue<?> getQueue(String name)
    - Context getJNDIContext(Hashtable properties) throws NamingException
    - Context getJNDIContext() throws NamingException
    - String[] discover(URI[] scopes)

**PrivilegedCarbonContext**

*PrivilegedCarbonContext* is a special subclass of *CarbonContext*, which allows you to perform privileged operations such as, setting the tenant ID and domain, starting or ending tenant flows and more. This class can only be used by Carbon components that have the permission to get hold of an instance of the *PrivilegedCarbonContext*. An instance of *PrivilegedCarbonContext* can only be obtained using the static methods outlined below.

**Obtaining the PrivilegedCarbonContext**
The following method has to be statically invoked, and will return an instance of PrivilegedCarbonContext.

getThreadLocalCarbonContext()

Obtains the PrivilegedCarbonContext by using data stored in the current thread: PrivilegedCarbonContext.getThreadLocalCarbonContext(). As a result, in cases like the deployers, where you can be sure that the deployment scheduler thread would set the ThreadLocal data, you should directly call the getThreadLocalCarbonContext method. You should have an idea, under which thread you are executing. It is better to resolve the CarbonContext outside that util method, as opposed to resolving the CarbonContext within the method.

Setting information into PrivilegedCarbonContext

If the data backed by the PrivilegedCarbonContext has been created for the first time, you have to populate information (such as, tenant ID, tenant domain etc.) needed by the downstream code, so that the code can simply get hold of an instance of CarbonContext or PrivilegedCarbonContext, and start using it. Populating this data can be done using the following setter methods.

- setTenantId(int tenantId)
- setTenantId(int tenantId, boolean resolveTenantDomain)
- setUsername(String username)
- setTenantDomain(String tenantDomain)
- setTenantDomain(String tenantDomain, boolean resolveTenantId)
- setApplicationName(String applicationName)
- setUserRealm(UserRealm userRealm)
- setRegistry(RegistryType type, Registry registry)

setTenantId(int tenantId)

This method sets the tenant ID in the CarbonContext. The tenant domain corresponding to this tenant ID will not be resolved.

setTenantId(int tenantId, boolean resolveTenantDomain)

This method sets the tenant ID in the CarbonContext. The tenant domain corresponding to this tenant ID will be resolved, if resolveTenantDomain is set to true.

setUsername(String username)

If there is a user logged in, this method sets that user’s username in the PrivilegedCarbonContext.

setTenantDomain(String tenantDomain)

This method sets the tenant domain on this CarbonContext instance. This method will not automatically resolve the tenant ID based on the tenant domain.

setTenantDomain(String tenantDomain, boolean resolveTenantId)

This method sets the tenant domain on this CarbonContext instance. If resolveTenantId is set to true, the tenant ID corresponding to the tenantDomain will be resolved internally.

setApplicationName(String applicationName)

This method sets the name of the webapp or service to which the request was destined.

setUserRealm(UserRealm userRealm)

This method sets the tenant specific UserRealm.
setRegistry(RegistryType type, Registry registry)

This method sets the tenant specific registry.

RegistryTypes

- USER_CONFIGURATION - The configuration registry of the currently logged in user.
- USER_GOVERNANCE - The governance registry of the currently logged in user.
- SYSTEM_CONFIGURATION - The configuration registry of the system.
- LOCAL_REPOSITORY - The local repository of the system.

Switching tenant flows

During an execution flow, sometimes you will need to switch from super tenant mode to tenant mode, do some work as that tenant, and then get back to super tenant mode. In such a scenario, you will do the following:

1. Start the Tenant Flow.
2. Set the tenant ID and the tenant domain in the new CarbonContext data holder that gets created when you start the tenant flow.
3. Carryout the respective action.
4. End the tenant flow.

Always follow the following template when you carryout the latter mentioned steps:

```java
try{
    PrivilegedCarbonContext.startTenantFlow();
    PrivilegedCarbonContext privilegedCarbonContext = PrivilegedCarbonContext.getThreadLocalCarbonContext();
    privilegedCarbonContext.setTenantId(tenantId);
    privilegedCarbonContext.setTenantDomain(tenantDomain);
    // set other stuff like registry etc. if needed
    doSomething();
} finally {
    PrivilegedCarbonContext.endTenantFlow();
}
```

Inside the doSomething() method, now if you call CarbonContext or PrivilegedCarbonContext.getThreadLocalCarbonContext, you will get the newly created data. The code that started the tenant flow is referred to as the upstream code, and all the code that gets called after starting the tenant flow are referred to as the downstream code. One example where you may start a tenant flow is when the super tenant runs tasks on behalf of a tenant. In the latter mentioned instance it may have a loop; and within that loop start a tenant flow for tenants, run the task as that tenant, get back to the super tenant mode (endTenantFlow), switch back to another tenant(startTenantFlow), run the task of the second tenant, get back to super tenant mode, and so on.

In most situations you will not need to start a tenant flow since you will not generally switch to tenant mode from super tenant mode.
**PrivilegedCarbonContext.startTenantFlow()**

This method starts a new tenant flow, and creates a new holder for tenant data. Thereafter, until `endTenantFlow` is called, the `getCarbonContext()` and `getThreadLocalCarbonContext()` methods will return the data related to the newly created tenant data holder. Once `startTenantFlow` is called, set the tenant ID, tenant domain and other tenant specific data.

**PrivilegedCarbonContext.endTenantFlow()**

This method will end the tenant flow and restore the previous `CarbonContext`.

The following diagram depicts during an execution of a thread, how we can start as a super tenant, then switch to tenant ‘x’ and carry out some actions, and later switch to tenant ‘y’ and carry out some actions and continue.

---

**Getting Information from PrivilegedCarbonContext**

- `getTenantDomain(boolean resolve)`
- `getTenantId(boolean resolve)`
- `getRegistry(RegistryType type)`
- `Object getOSGiService(Class clazz)`
- `List<Object> getOSGiServices(Class clazz)`

**getTenantDomain(boolean resolve)**

This method retrieves the tenant domain, and if `resolve` is set to `true`, it will try to resolve the domain using the tenant ID.

**getTenantId(boolean resolve)**

This method retrieves the tenant ID, and if `resolve` is set to `true`, it will try to resolve the tenant ID using the tenant domain.

**getRegistry(RegistryType type)**

This method retrieves the Registry of the tenant of type `RegistryType`.

**RegistryTypes**
- SYSTEM_CONFIGURATION - The system configuration registry of the tenant.
- SYSTEM_GOVERNANCE - The governance registry of the tenant.

**Object getOSGiService(Class clazz)**

This method obtains the first instance of the OSGi services found for the interface or class `clazz`.

Note that this method is depreciated in Carbon 4.3.0. The new method is `getOSGiService(Class clazz, Hashtable<String, String> props)`.

**List<Object> getOSGiServices(Class clazz)**

This method obtains all OSGi service instances found for interface or class `clazz`.

Note that this method is depreciated in Carbon 4.3.0. The new method is `List<Object> getOSGiServices(Class clazz, Hashtable<String, String> props)`.

**Miscellaneous Methods**

- `public static void unloadTenant(int tenantId)`
- `public static void destroyCurrentContext()`

**public static void unloadTenant(int tenantId)**

This method unloads the tenant.

**public static void destroyCurrentContext()**

This method destroys the current `ThreadLocal CarbonContext`.

**CarbonContext**

The `CarbonContext` is designed for normal tenants to retrieve information from the Carbon runtime. In the super tenant mode, for this to work the relevant data has to be set so that tenants can retrieve information using the `CarbonContext`.

**Obtaining the CarbonContext**

- `getThreadLocalCarbonContext()`

**getThreadLocalCarbonContext()**

This method obtains the `CarbonContext` by using the data stored in the current thread: The `CarbonContext.getThreadLocalCarbonContext()`.

**Retrieving information**

The following methods allow tenants to retrieve information relevant to those tenants from the Carbon runtime.

- `public int getTenantId()`  
- `String getUsername()`  
- `String getTenantDomain()`  
- `Registry getRegistry(RegistryType type)`
public int getTenantId()
This method retrieves the tenant ID.

String getUsername()
This method retrieves the tenant's username.

String getTenantDomain()
This method retrieves the tenant domain.

Registry getRegistry(RegistryType type)
This method retrieves the tenant specific registry.

Registry Types
- USER_CONFIGURATION - The configuration registry of the currently logged in user.
- USER_GOVERNANCE - The governance registry of the currently logged in user.
- SYSTEM_CONFIGURATION - The configuration registry of the system.
- SYSTEM_GOVERNANCE - The governance registry of the system.
- LOCAL_REPOSITORY - The local repository of the system.

UserRealm getUserRealm()
This method retrieves the user realm of the current tenant.

CarbonQueue<?> getQueue(String name)
This method obtains a named queue instance.

Context getJNDIContext(Hashtable properties) throws NamingException
This method obtains a JNDI-context with the given initialization properties.

Method description
properties - the properties required to create the JNDI-context instance.
NamingException - if the operation fails, the naming exception will be thrown.

Context getJNDIContext() throws NamingException
This method obtains a JNDI-context.

Method description
NamingException - if the operation fails, the naming exception will be thrown.

String[] discover(URI[] scopes)
This method is used to discover a set of service endpoints belonging the defined scopes. Therefore, when this method is executed the list of service endpoints will be returned.
Method
description

scopes – the scopes in which to look-up for the service.

WSO2 Patch Application Process

You apply patches to WSO2 products either as individual patches or through a service pack. A service pack is recommended when the number of patches increase. The following sections explain the WSO2 patch application process:

- Applying service packs to the Kernel
- Applying individual patches to the Kernel
- Verifying the patch application
- Overview of the patch application process

Before you begin

- You can download all WSO2 Carbon Kernel patches from here.
- Before you apply a patch, check its README.txt file for any configuration changes required.

Applying service packs to the product

Carbon 4.2.0 Kernel supports service packs. A service pack is a collection of patches in a single pack. It contains two elements:

- The lib directory: contains all the JARs relevant to the service pack.
- The servicepack_patches.txt text file: contains the list of JARs in the service pack.

Follow the steps below to apply service packs to your product.

1. Copy the service pack file to the <PRODUCT_HOME>/repository/components/servicepacks/ directory. For example, the image below shows how a new service pack named servicepack001 is added to this directory.

```
servicepacks
  └── servicepack001
      └── lib
          ├── org.wso2.carbon.base_4.2.0.SNAPSHOT.jar
          │    ├── org.wso2.carbon.bridge_4.2.0.SNAPSHOT.jar
          │    └── org.wso2.carbon.core.bootup.validator_4.2.0.SNAPSHOT.jar
          │    └── org.wso2.carbon.core.common_4.2.0.SNAPSHOT.jar
          │    └── org.wso2.carbon.registry.api_4.2.0.SNAPSHOT.jar
          └── servicepack_patches.txt
```

2. Start your product. The following steps will be executed:

   1. Before applying any patches, the process first creates a backup folder named patch0000 inside the <PRODUCT_HOME>/repository/components/patches/ directory, which will contain the original content of the <PRODUCT_HOME>/repository/components/plugins/ directory. This step enables you to revert back to the previous state if something goes wrong during operations.

   2. The latest service pack in the <PRODUCT_HOME>/repository/components/servicepacks/ directory will be applied. That is, the patches in the service pack will be applied to the <PRODUCT_HOME>/
2. In addition to the service pack, if there are individual patches added to the `<PRODUCT_HOME>/repository/components/patches/` directory, those will also be incrementally applied to the plugins directory.

The metadata file available in the service pack will maintain a list of the applied patches by service pack. Therefore, the metadata file information will be compared against the `<PRODUCT_HOME>/repository/components/patches/` directory, and only the patches that were not applied by the service pack will be incrementally applied to the plugins directory.

### Applying individual patches to the product

You can apply each patch individually to your system as explained below. Alternatively, you can apply patches through service packs as explained above.

1. Copy the patches to the `<PRODUCT_HOME>/repository/components/patches/` directory.
2. Start the Carbon server. The patches will then be incrementally applied to the plugins directory.

Before applying any patches, the process first creates a backup folder named `patch0000` inside the `<PRODUCT_HOME>/repository/components/patches/` directory, which will contain the original content of the `<PRODUCT_HOME>/repository/components/plugins/` directory. This step enables you to revert back to the previous state if something goes wrong during operations.

Prior to Carbon 4.2.0, users were expected to apply patches by starting the server with `wso2server.sh -DapplyPatches`. Now, you do not have to issue a special command to trigger the patch application process. It starts automatically if there are changes in either the `<PRODUCT_HOME>/repository/components/servicepacks/` directory or the `<PRODUCT_HOME>/repository/components/patches/` directory. It verifies all the latest JARs in the servicepacks and patches directories against the JARs in the plugins directory by comparing MD5s of JARs.

### Verifying the patch application

After the patch application process is completed, the patch verification process ensures that the latest service pack and other existing patches are correctly applied to the `<PRODUCT_HOME>/repository/components/plugins/` folder.

- All patch related logs are recorded in the `<PRODUCT_HOME>/repository/logs/patches.log` file.
- The `<PRODUCT_HOME>/repository/components/patches/` directory contains the list of patched JARs and the md5 values.
- A list of all the applied service packs and patches are in the `<PRODUCT_HOME>/repository/components/default/configuration/prePatched.txt` file.

Do not change the data in the `<PRODUCT_HOME>/repository/components/default/configuration/prePatched.txt` file. The patch application process gets the pre-patched list from this file and compares the list with the patches available in the servicepack and patches directories. If you change the data in this file, you will get a startup error when applying patches.

### Overview of the patch application process

- The metadata file available in the service pack will maintain a list of the applied patches by service pack. Therefore, the metadata file information will be compared against the `<PRODUCT_HOME>/repository/components/patches/` directory, and only the patches that were not applied by the service pack will be incrementally applied to the plugins directory.

Applying individual patches to the product

You can apply each patch individually to your system as explained below. Alternatively, you can apply patches through service packs as explained above.

1. Copy the patches to the `<PRODUCT_HOME>/repository/components/patches/` directory.
2. Start the Carbon server. The patches will then be incrementally applied to the plugins directory.

Before applying any patches, the process first creates a backup folder named `patch0000` inside the `<PRODUCT_HOME>/repository/components/patches/` directory, which will contain the original content of the `<PRODUCT_HOME>/repository/components/plugins/` directory. This step enables you to revert back to the previous state if something goes wrong during operations.

Prior to Carbon 4.2.0, users were expected to apply patches by starting the server with `wso2server.sh -DapplyPatches`. Now, you do not have to issue a special command to trigger the patch application process. It starts automatically if there are changes in either the `<PRODUCT_HOME>/repository/components/servicepacks/` directory or the `<PRODUCT_HOME>/repository/components/patches/` directory. It verifies all the latest JARs in the servicepacks and patches directories against the JARs in the plugins directory by comparing MD5s of JARs.

### Verifying the patch application

After the patch application process is completed, the patch verification process ensures that the latest service pack and other existing patches are correctly applied to the `<PRODUCT_HOME>/repository/components/plugins/` folder.

- All patch related logs are recorded in the `<PRODUCT_HOME>/repository/logs/patches.log` file.
- The `<PRODUCT_HOME>/repository/components/patches/` directory contains the list of patched JARs and the md5 values.
- A list of all the applied service packs and patches are in the `<PRODUCT_HOME>/repository/components/default/configuration/prePatched.txt` file.

Do not change the data in the `<PRODUCT_HOME>/repository/components/default/configuration/prePatched.txt` file. The patch application process gets the pre-patched list from this file and compares the list with the patches available in the servicepack and patches directories. If you change the data in this file, you will get a startup error when applying patches.
The diagram below shows how the patch application process is implemented when you start the server.

Customizing the Management Console

The Management Console user interface (https://localhost:9443/carbon) of a Carbon product consists of two layers:

1. **UI inherited from WSO2 Carbon platform** contains the templates, styles (css files), and images that are stored in the core Carbon UI bundle stored in `<PRODUCT_HOME>/repository/components/plugins/org.wso2.carbon.ui_<version-number>.jar` where `<version-number>` is the version of the Carbon kernel that the product is built on. This bundle is responsible for the overall look and feel of the entire Carbon platform.

2. **UI unique to each product** contains all the styles and images that override the ones in core Carbon platform. This file is in `<PRODUCT_HOME>/repository/components/plugins/org.wso2.<product-name>.styles_<version-number>.jar` where `<version-number>` is the version of the product.

The following topics explain how to download a Carbon product and customize its user interface.

- Setting up the development environment
- Customizing the user interface
- Starting the server

Setting up the development environment

To download and set up the product environment for editing, take the following steps.

1. Download your product.
2. Extract the ZIP file into a separate folder in your hard drive.
3. Go to the `<PRODUCT_HOME>/repository/components/plugins/` directory to find the required JAR files:
   - `org.wso2.carbon.ui_<version-number>.jar`
   - `org.wso2.<product-name>.styles_<version-number>.jar`
4. Copy the JAR files to a separate location on your hard drive. Since the JAR files are zipped, you must unzip them to make them editable.

You can now customize the look and feel of your product by modifying the contents of the JAR files as described in the next section.
Customizing the user interface

Customizing the product interface involves changing the layout/design of the Carbon framework as well as changing the styles and images specific to the product. The following topics explain how some of the main changes to the product interface can be done.

- Changing the layout
- Changing the styles on the Carbon framework
- Changing the product specific styles and images

Changing the layout

The layout of the Carbon framework is built using a tiles JSP tag library. The use of tiles allows us to break the presentation of the layout into small JSP snippets that perform a specific function. For example, header.jsp and footer.jsp are the tiles corresponding to the header and footer in the layout. The template.jsp file controls the main layout page of the Carbon framework, which holds all the tiles together. That is, the header part in the template.jsp file is replaced with the <tiles:insertAttribute name="header"/> tag, which refers to the header.jsp file. The template.jsp file as well as the JSP files corresponding to the tiles are located in the org.wso2.<product-name>.styles_<version-name>.jar/web/admin/layout/ directory.

Therefore, changing the layout of your product primarily involves changing the template.jsp page (main layout page) and the JSP files of the relevant JSP tiles.

Ensure that you do not change or remove the ID attributes on the .jsp files.

Changing the styles on the Carbon framework

The global.css file, which determines the styles of the Carbon framework, is located in the org.wso2.carbon.ui_<version-name>.jar/web/admin/css/ directory. You can edit this file as per your requirement. Alternatively, you can apply a completely new stylesheet to your framework instead of the default global.css stylesheet.

To apply a new style sheet to the carbon framework:

1. Copy your new CSS file to this same location.
2. Open the template.jsp file located in the org.wso2.carbon.ui_<version-name>.jar/web/admin/layout/ directory, which contains the main layout of the page and the default JavaScript libraries.
3. Replace global.css with the new style sheet by pointing the String globalCSS attribute to the new stylesheet file.

```java
//Customization of UI theming per tenant
String tenantDomain = null;
String globalCSS = "../admin/css/global.css";
String mainCSS = "";
```

Changing the product specific styles and images

The styles and images unique to your product are located in the org.wso2.<product-name>.styles_<version-number>.jar folder. To modify product specific styles and images, take the following steps.

1. Copy the necessary images to the org.wso2.<product-name>.styles_<version-number>.jar/web/styles/images/ directory. For example, if you want to change the product banner, add the new image
file to this directory.

2. Open the `main.css` file located in the `org.wso2.<product-name>.styles_<version-name>.jar/web/styles/css/` directory.

3. To specify a new product banner, change the `background-image` attribute of `org.wso2.<product-name>.styles_<version-name>.jar/web/styles/css/main.css` file as follows:

```css
/* ---------------- header styles ------------------ */
div#header-div {
    background-image: url( ../images/newproduct-header-bg.png);
    height:70px;
}
```

Note that the size of the images you use will affect the overall UI of your product. For example, if the height of the product logo image exceeds 28 pixels, you must adjust the `main.css` file in the `org.wso2.<product-name>.styles_<version-name>.jar/web/styles/css/` directory to ensure that the other UI elements of your product aligns with the product logo.

Starting the server

In the preceding steps, you have done the changes to the product interface after copying the JAR files to a separate location on your hard drive. Therefore, before you start your production server, these files must be correctly copied back to your production environment as explained below.

2. Change the name of the ZIP file to `org.wso2.carbon.ui_<version-number>.jar` and `org.wso2.<product-name>.styles_<version-number>.jar` respectively.
3. Copy these two new JAR files to the `<PRODUCT_HOME>/repository/components/plugins/` directory in your product installation.
4. Start the server.

Enabling SPI Fly Support

The WSO2 Carbon platform supports OSGi JRE SPI mechanisms by integrating ‘Apache Aries SPI Fly’, which provides OSGi support for JRE SPI mechanisms, including the usage of java.util.ServiceLoader, META-INF/services and similar methods. Developers can integrate Apache Aries SPI-Fly support by adding the relevant bundles to the `<AS_HOME>/repository/components/dropins/` directory as explained below. Since the Carbon Runtime is higher than the OSGi 4.3 compliant framework that supports WeavingHooks, we can use either `Dynamic` or `Static` weaving in the Application Server.

The following steps explain how SPI Fly support can be enabled for AS:

1. Copy the relevant bundles from Apache Aries to the `<AS_HOME>/repository/components/dropins/` directory. The bundles you should copy depend on whether `Dynamic` weaving or `Static` weaving should be enabled as explained below.
   - For `Dynamic` weaving, copy the following bundles:
     - SPI Fly Dynamic Weaving Bundle
     - Aries Util
     - ASM
   - For `Static` weaving, copy the following bundles:
     - SPI Fly Static Weaving Bundle
     - Aries Util
     - ASM
1. For Static weaving, copy the following bundles:

   - SPI Fly Static Weaving Bundle
   - Aries Util

2. These bundles will provide the necessary support for SPI Fly. Along with them, you can apply your SPI providers and consumers to the dropins directory.

Sample bundles for SPI providers and consumers are given here. Even though these are the same bundles that are provided by Aries, we have modified the client bundle to address an issue with the static weaving bundle.

3. If Static weaving is used, it is necessary to process the consumer bundles before the provider bundles. You can do this by executing the following command on a command line:

   ```
   java -jar
   org.apache.aries.spifly.static.tool-1.0.0-jar-with-dependencies.jar
   <provider-bundle>.jar
   ```

   This command will change the bytecode of the bundle so that static weaving can be used at runtime. Note that the following tool is used in the above command: org.apache.aries.spifly.static.tool.

4. Start the server.

5. Check the console logs and you will observe the logs from the consumer bundle that invoked the SPI.

See the Apache Aries documentation for samples and more information.

**Java Documentation**

The Java document for Application Server describes all the classes, interfaces, and methods of the Application Server APIs.
Reference Guide

The following topics provide reference information for working with WSO2 AS:

- Configuration Files
- Changing the Default Ports
- Tools
- Lazy Loading of Tenants and Artifacts
- Product Startup Options

Configuration Files

The following sections provide detailed definitions for the main configuration files used in Carbon-based products.

- Configuring axis2.xml
- Configuring carbon.xml
- Configuring catalina-server.xml
- Configuring identity.xml
- Configuring master-datasources.xml
- Configuring registry.xml
- Configuring user-mgt.xml
- Configuring config-validation.xml
- Configuring sso-sp-config.properties

Configuring axis2.xml

Users can change the default functionality-related configurations by editing the `<PRODUCT_HOME>/repository/conf/axis2/axis2.xml` file using the information given below. This information is provided as reference for users who are already familiar with the product features and want to know how to configure them. If you need introductory information on a specific concept, such as message receivers and formatters, see the relevant topics in the User Guide.

Click on the table and use the left and right arrow keys to scroll horizontally. For sample values, see the Example below the table.

XML Elements

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attributes</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
<th>Mandatory/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;axisconfig&gt;</td>
<td>name</td>
<td>The root element. The name is defined as: name= &quot;AxisJava2.0&quot;</td>
<td></td>
<td></td>
<td>Mandatory</td>
</tr>
<tr>
<td>&lt;module&gt;</td>
<td>ref</td>
<td>A globally engaged module. The ref attribute specifies the module name.</td>
<td></td>
<td></td>
<td>Mandatory</td>
</tr>
</tbody>
</table>
### `<parameter>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>A parameter is a name-value pair. All top-level parameters (those that are direct sub-elements of the root element) will be transformed into properties in AxisConfiguration and can be accessed in the running system. The <code>name</code> attribute (required) specifies the parameter name. If you set the <code>locked</code> attribute to true (default is false), this parameter's value cannot be overridden by services and other configurations.</td>
</tr>
<tr>
<td><code>locked</code></td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

### `<listener>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class</code></td>
<td>A registered listener that will be automatically informed whenever a change occurs in AxisConfiguration, such as when a service or module is deployed or removed. The <code>class</code> attribute specifies this listener's implementation class, which must implement the AxisObserver interface. Registering listeners is useful for additional features such as RSS feed generation, which will provide service information to subscribers.</td>
</tr>
<tr>
<td></td>
<td>Optional</td>
</tr>
</tbody>
</table>

### `<messageReceivers>`

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The container element for messages receiver definitions.</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

### `<messageReceiver>`

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>class</code></td>
<td>A message receiver definition. The <code>class</code> attribute (required) specifies the message receiver implementation class. The <code>mep</code> attribute (required) specifies the message exchange pattern supported by this message receiver. Each message receiver definition supports only one MEP.</td>
</tr>
<tr>
<td></td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

| `mep`     |               |
### <messageFormatters>

The container element for message formatter definitions, which are used to serialize outgoing messages to different formats (such as JSON). The format for a message can be specified by setting the "messageType" property in the MessageContext. It can also be specified as a parameter in service.xml (for service-based configuration) in addition to axis2.xml (for global configuration).

### <messageFormatter> contentType class

A message formatter definition. The `contentType` attribute specifies which message types are handled by this formatter, and the `class` attribute specifies the formatter implementation class.

### <messageBuilders>

The container element for the message builder definitions, which are used to process the raw payload of incoming messages and convert them to SOAP.

### <messageBuilder> contentType class

A message builder definition. The `contentType` attribute specifies which message types are handled by this builder, and the `class` attribute specifies the builder implementation class.

### <transportReceiver> name class

A transport receiver definition, one for each transport type. The `name` attribute specifies the short name to use when referring to this transport in your configurations (http, tcp, etc.), and the `class` attribute specifies the receiver implementation class that provides the logic for receiving messages via this transport. You can specify `<parameter>` elements to pass any necessary information to the transport.
<table>
<thead>
<tr>
<th>Element</th>
<th>Attribute</th>
<th>Description</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;transportSender&gt;</code></td>
<td></td>
<td>Just like <code>&lt;transportReceiver&gt;</code>, except <code>&lt;transportSender&gt;</code> allows you to define transport senders, which are used to send messages via the transport.</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>
| `<phaseOrder>`    | `type`    | Specifies the order of phases in the execution chain of a specific type of flow (specified by the `type` attribute), which can be one of the following:  
  - InFlow  
  - OutFlow  
  - InFaultFlow  
  - OutFaultFlow  
You add phases using the `<phase>` sub-element. In the `In` phase orders, all phases before the Dispatch phase are global phases and after Dispatch are operation phases. In the `Out` phase orders, phases before the MessageOut phase are global phases and after MessageOut are operation phases. | Mandatory   |
| `<phase>`         | `name`    | The phase definition. The `name` attribute specifies the phase name. You can add the `<handler>` sub-element to execute a specific handler during this phase.                                                      | Mandatory   |
| `<handler>`       | `name`    |                                                                                                                                                | Optional    |
|                   | `class`   | The handler (message processing functionality) to execute during this phase. Handlers are combined into chains and phases to provide customizable functionality such as security, reliability, etc. Handlers must be multi-thread safe and should keep all their state in Context objects (see the `org.apache.axis2.context` package). |             |
| `<order>`         | `phase`   |                                                                                                                                                | Optional    |
### <clustering>

<table>
<thead>
<tr>
<th>class</th>
<th>enable</th>
<th>Used to enable clustering. The class attribute specifies the clustering agent class. The enable attribute is false by default; set it to true to enable clustering.</th>
<th>Optional</th>
</tr>
</thead>
</table>

### <property>

<table>
<thead>
<tr>
<th>name</th>
<th>value</th>
<th>Optional</th>
</tr>
</thead>
</table>

### <members>

| <members> | The list of static or well-known members. These entries will only be valid if the "membershipScheme" above is set to "wka" | N/A | Optional |

### <member>

| <member> | N/A | Optional |

### <hostName>

| <hostName> | N/A | Optional |

### <port>

| <port> | N/A | Optional |

### <groupManagement>

| <groupManagement> | Enable the groupManagement entry if you need to run this node as a cluster manager. Multiple application domains with different GroupManagementAgent implementations can be defined in this section. | Optional |

| enable | FALSE |

### <applicationDomain>

| <applicationDomain> | N/A | Optional |

| name | port | subDomain | agent | description |}

**Example**

The following example shows excerpts from an axis2.xml file.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
...
<axisconfig name="AxisJava2.0">
```
<!-- ================================================= -->
<!--                  Parameters                       -->
<!-- ================================================= -->

...<br>

<!-- If you want to enable file caching for attachments change this to true -->
<parameter name="cacheAttachments" locked="false">false</parameter><br>
<br>
<!-- Attachment file caching location relative to CARBON_HOME -->
<parameter name="attachmentDIR" locked="false">work/mtom</parameter><br>
<br>
<!-- Attachment file cache threshold size -->
<parameter name="sizeThreshold" locked="false">4000</parameter><br>
<br>
...

<!-- ================================================= -->
<!--                  Listeners                        -->
<!-- ================================================= -->

<!-- This deployment interceptor will be called whenever before a module is initialized or -->
<!-- service is deployed -->
<listener class="org.wso2.carbon.core.deployment.DeploymentInterceptor"/>

...<br>

<!-- ================================================= -->
<!--                  Deployers                        -->
<!-- ================================================= -->

<!-- Deployer for the dataservice. -->
<!-- <deployer extension="dbs" directory="dataservices" class="org.wso2.dataservices.DBDeployer"/>-->

<!-- Axis1 deployer for Axis2 -->
<!-- <deployer extension="wsdd" class="org.wso2.carbon.axis1services.Axis1Deployer" directory="axis1services"/>-->

...

<!-- ================================================= -->
<!--                Message Receivers                  -->
<!-- ================================================= -->

<!-- This is the set of default Message Receivers for the system, if you want to have -->
<!-- message receivers for any of the other Message exchange Patterns (MEP) implement it -->
<!-- and add the implementation class to here, so that you can refer from any operation -->
<!-- Note : You can override this for particular service by adding this -->
same element to the -->
   <!-- services.xml with your preferences -->
   <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/robust-in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/in-out"
class="org.apache.axis2.rpc.receivers.RPCMessageReceiver"/>
   </messageReceivers>

   <!-- Following content type to message formatter mapping can be used to
implement support -->
   <!-- for different message format serializations in Axis2. These
message formats are -->
   <!-- expected to be resolved based on the content type. -->
   <messageFormatters>
      <messageFormatter contentType="application/x-www-form-urlencoded"
class="org.apache.axis2.transport.http.XFormURLEncodedFormatter"/>
      <messageFormatter contentType="multipart/form-data"
class="org.apache.axis2.transport.http.MultipartFormDataFormatter"/>
      <messageFormatter contentType="application/xml"
class="org.apache.axis2.transport.http.ApplicationXMLFormatter"/>
      <messageFormatter contentType="text/xml"
class="org.apache.axis2.transport.http.SOAPMessageFormatter"/>
      <messageFormatter contentType="application/soap+xml"
class="org.apache.axis2.transport.http.SOAPMessageFormatter"/>
      <messageFormatter contentType="text/plain"
class="org.apache.axis2.format.PlainTextFormatter"/>
...
   </messageFormatters>

   <!-- Following content type to builder mapping can be used to implement
support for -->
<messageBuilders>
  <messageBuilder contentType="application/xml"
      class="org.apache.axis2.builder.ApplicationXMLBuilder"/>
  <messageBuilder contentType="application/x-www-form-urlencoded"
      class="org.apache.synapse.commons.builders.XFormURLEncodedBuilder"/>
  <messageBuilder contentType="multipart/form-data"
      class="org.apache.axis2.builder.MultipartFormDataBuilder"/>
  <messageBuilder contentType="text/plain"
      class="org.apache.axis2.format.PlainTextBuilder"/>
...
</messageBuilders>

<!-- Transport Ins (Listeners) -->
<transportReceiver name="http"
  class="org.apache.synapse.transport.passthru.PassThroughHttpListener">
  <parameter name="port" locked="false">8280</parameter>
  <parameter name="non-blocking" locked="false">true</parameter>
  <!--parameter name="bind-address" locked="false">hostname or IP address</parameter-->
  <!--parameter name="WSDL/EPRPrefix" locked="false">https://apachehost/port/somepath</parameter-->
  <parameter name="httpGetProcessor" locked="false">org.wso2.carbon.transport.nhttp.api.PassThroughNHttpGetProcessor</parameter>
  <!--<parameter name="priorityConfigFile" locked="false">location of priority configuration file</parameter-->-->
</transportReceiver>

<!-- Transport Outs (Senders) -->
<transportSender name="http"
  class="org.apache.synapse.transport.passthru.PassThroughHttpSender">
  <parameter name="non-blocking" locked="false">true</parameter>
  <!--parameter name="warnOnHTTP500" locked="false">*</parameter-->-->
  <!--parameter name="http.proxyHost" locked="false">localhost</parameter-->
  <!--parameter name="http.proxyPort" locked="false">3128</parameter-->-->

<transportSender>

</transportSender>

<module ref="addressing"/>

<clustering class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent" enable="false">
  <parameter name="AvoidInitiation">true</parameter>
  <members>
  <member>
    <hostName>127.0.0.1</hostName>
    <port>4000</port>
  </member>
  </members>
</clustering>

This parameter indicates whether the cluster has to be automatically initialized when the AxisConfiguration is built. If set to "true" the initialization will not be done at that stage, and some other party will have to explicitly initialize the cluster.

The list of static or well-known members. These entries will only be valid if the "membershipScheme" above is set to "wka"

...
<!-- Enable the groupManagement entry if you need to run this node as a cluster manager. Multiple application domains with different GroupManagementAgent implementations can be defined in this section. -->

<groupManagement enable="false">
  <applicationDomain name="wso2.esb.domain" description="ESB group"
    agent="org.wso2.carbon.core.clustering.hazelcast.HazelcastGroupManagementAgent"
    subDomain="worker" port="2222"/>
</groupManagement>
</clustering>

<!-- ================================================= -->
<!--                   Transactions                    -->
<!-- ================================================= -->

<!-- Uncomment and configure the following section to enable transactions support -->

<!--<transaction timeout="30000">
  <parameter name="java.naming.factory.initial">org.apache.activemq.jndi.ActiveMQInitialContextFactory</parameter>
  <parameter name="java.naming.provider.url">tcp://localhost:61616</parameter>
  <parameter name="UserTransactionJNDIName">UserTransaction</parameter>
  <parameter name="TransactionManagerJNDIName">TransactionManager</parameter>
</transaction> -->

-->  

<!-- ================================================= -->
<!--                    Phases                         -->
<!-- ================================================= -->

<phaseOrder type="InFlow">
  <!-- System pre defined phases -->
  <!-- The MsgInObservation phase is used to observe messages as soon as they are received. In this phase, we could do some things such as SOAP message tracing & keeping track of the time at which a particular message was received   -->
  NOTE: This should be the very first phase in this flow
</phaseOrder>
<phase name="MsgInObservation">
  <handler name="TraceMessageBuilderDispatchHandler"
    class="org.apache.synapse.transport.passthru.util.TraceMessageBuilderDispatchHandler"/>
</phase>

<phase name="Validation"/>
<phase name="Transport">
  <handler name="RequestURIBasedDispatcher" class="org.apache.axis2.dispatchers.RequestURIBasedDispatcher">
    <order phase="Transport"/>
  </handler>

  <handler name="CarbonContextConfigurator" class="org.wso2.carbon.mediation.initializer.handler.CarbonContextConfigurator">
  </handler>

  </handler>

  <handler name="SOAPActionBasedDispatcher" class="org.apache.axis2.dispatchers.SOAPActionBasedDispatcher">
    <order phase="Transport"/>
  </handler>

  <!--handler name="SMTPFaultHandler" class="org.wso2.carbon.core.transports.smtp.SMTPFaultHandler">
    <order phase="Transport"/>
  </handler-->

  <handler name="CacheMessageBuilderDispatchandler" class="org.wso2.carbon.mediation.initializer.handler.CacheMessageBuilderDispatchandler">
  </handler>
</phase>

<phaseOrder type="OutFlow">
  <!-- Handlers related to unified-endpoint component are added to the UEPPhase -->
  <phase name="UEPPhase"/>
  <!-- user can add his own phases to this area -->
  <phase name="RMPhase"/>
  ...
</phaseOrder>
Configuring carbon.xml

Users can change the configurations related to the default Carbon functionality by editing the `<PRODUCT_HOME>/repository/conf/carbon.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

### XML Elements

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Server&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Name&gt;</td>
<td></td>
<td>Product Name.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;ServerKey&gt;</td>
<td></td>
<td>Machine readable unique key to identify each product.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;Version&gt;</td>
<td></td>
<td>Product Version.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;HostName&gt;</td>
<td></td>
<td>Host name or IP address of the machine hosting this server e.g. <a href="http://www.wso2.org">www.wso2.org</a>, 192.168.1.10 This is will become part of the End Point Reference of the services deployed on this server instance.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;MgtHostName&gt;</td>
<td></td>
<td>Host name to be used for the Carbon management console.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;ServerURL&gt;</td>
<td></td>
<td>The URL of the back end server. This is where the admin services are hosted and will be used by the clients in the front end server. This is required only for the Front-end server. This is used when separating the BE server from the FE server.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;IndexPageURL&gt;</td>
<td></td>
<td>The URL of the index page. This is where the user will be redirected after signing in to the carbon server.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;ServerRoles&gt;</td>
<td></td>
<td>For cApp deployment, we have to identify the roles that can be acted by the current server. The following property is used for that purpose. Any number of roles can be defined here. Regular expressions can be used in the role. Ex : &lt;Role&gt;.*&lt;/Role&gt; means this server can act as any role.</td>
<td>String</td>
</tr>
<tr>
<td>&lt;Role&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;BamServerURL&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Package&gt;</td>
<td></td>
<td>The fully qualified name of the server.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;WebContextRoot&gt;</code></td>
<td>Webapp context root of WSO2 Carbon.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td><code>&lt;RegistryHttpPort&gt;</code></td>
<td>In-order to get the registry http Port from the back-end when the default http transport is not the same.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ItemsPerPage&gt;</code></td>
<td>Number of items to be displayed on a management console page. This is used at the backend server for pagination of various items.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;InstanceMgtWSEndpoint&gt;</code></td>
<td>The endpoint URL of the cloud instance management Web service.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td><code>&lt;Ports&gt;</code></td>
<td>Ports used by this server</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Offset&gt;</code></td>
<td>Ports offset. This entry will set the value of the ports defined below to the define value + Offset. e.g. Offset=2 and HTTPS port=9443 will set the effective HTTPS port to 9445.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;JMX&gt;</code></td>
<td>The JMX Ports.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;RMIRegistryPort&gt;</code></td>
<td>The port RMI registry is exposed.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;RMIServerPort&gt;</code></td>
<td>The port RMI server should be exposed.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;EmbeddedLDAP&gt;</code></td>
<td>Embedded LDAP server specific ports.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;LDAPServerPort&gt;</code></td>
<td>Port which embedded LDAP server runs.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;KDCServerPort&gt;</code></td>
<td>Port which KDC (Kerberos Key Distribution Center) server runs.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;EmbeddedQpid&gt;</code></td>
<td>Embedded Qpid broker ports.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;BrokerPort&gt;</code></td>
<td>Broker TCP Port.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;BrokerSSLPot&gt;</code></td>
<td>SSL Port.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;JNDIProviderPort&gt;</code></td>
<td>Override datasources JNDIproviderPort defined in bps.xml and datasources.properties files.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ThriftEntitlementReceivePort&gt;</code></td>
<td>Override receive port of thrift based entitlement service.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;JNDI&gt;</code></td>
<td>JNDI Configuration.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;DefaultInitialContextFactory&gt;</code></td>
<td>The fully qualified name of the default initial context factory.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td><code>&lt;Restrictions&gt;</code></td>
<td>The restrictions that are done to various JNDI Contexts in a Multi-tenant environment.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;AllTenants&gt;</code></td>
<td>Contexts that are common to all tenants.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td><code>&lt;UrlContexts&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;UrlContext&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Scheme&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>&lt;SuperTenantOnly&gt;</td>
<td>Contexts that will be available only to the super-tenant.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;UrlContexts&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;UrlContext&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Scheme&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;IsCloudDeployment&gt;</td>
<td>Property to determine if the server is running on a cloud deployment environment. This property should only be used to determine deployment specific details that are applicable only in a cloud deployment, i.e when the server is deployed “-as-a-service.”</td>
<td>Boo</td>
<td></td>
</tr>
<tr>
<td>&lt;EnableMetering&gt;</td>
<td>Property to determine whether usage data should be collected for metering purposes.</td>
<td>Boo</td>
<td></td>
</tr>
<tr>
<td>&lt;MaxThreadExecutionTime&gt;</td>
<td>The Max time a thread should take for execution in seconds.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;GhostDeployment&gt;</td>
<td>A flag to enable or disable Ghost Deployer. By default this is set to false. That is because the Ghost Deployer works only with the HTTP/S transports. If you are using other transports, don't enable Ghost Deployer.</td>
<td>Boo</td>
<td></td>
</tr>
<tr>
<td>&lt;Enabled&gt;</td>
<td>When &lt;GhostDeployment&gt; is enabled, the lazy loading feature will apply to artifacts deployed. That is, when a tenant loads, only the specific artifact requested by the service will be loaded.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;PartialUpdate&gt;</td>
<td>&lt;PartialUpdate&gt; is a further enhancement to lazy loading of artifacts, which applies when &lt;DeploymentSynchronizer&gt; is enabled in a clustered environment.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Axis2Config&gt;</td>
<td>Axis2 related configurations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;RepositoryLocation&gt;</td>
<td>Location of the Axis2 Services &amp; Modules repository. This can be a directory in the local file system, or a URL. E.g. 1. /home/wso2wsas/repository/ - An absolute path 2. repository - In this case, the path is relative to CARBON_HOME 3. file:///home/wso2wsas/repository/ 4. <a href="http://wso2wsas/repository/">http://wso2wsas/repository/</a>.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;DeploymentUpdateInterval&gt;</td>
<td>Deployment update interval in seconds. This is the interval between repository listener executions.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;ConfigurationFile&gt;</td>
<td>Location of the main Axis2 configuration descriptor file, a.k.a. axis2.xml file. This can be a file on the local file system, or a URL. E.g. 1. /home/repository/axis2.xml - An absolute path 2. conf/axis2.xml - In this case, the path is relative to CARBON_HOME 3. file:///home/repository/axis2.xml 4. <a href="http://repository/conf/axis2.xml">http://repository/conf/axis2.xml</a></td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;ServiceGroupContextIdleTime&gt;</td>
<td>ServiceGroupContextIdleTime, which will be set in ConfigurationContext for multiple clients which are going to access the same ServiceGroupContext Default Value is 30 Sec.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;ClientRepositoryLocation&gt;</td>
<td>This repository location is used to create the client side configuration context used by the server when calling admin services.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;clientAxis2XmlLocation&gt;</td>
<td>This axis2 xml is used in creating the configuration context by the FE server calling to BE server.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>&lt;HideAdminServiceWSDLs&gt;</td>
<td>If this parameter is set, the WSDL file on an admin service will not give the admin service WSDL. By default, this parameter is set to &quot;true&quot;. Note that setting this parameter to false will expose WSO2 Storage Server operations through a WSDL.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;HttpAdminServices&gt;</td>
<td>WARNING-Use With Care! Uncommenting bellow parameter would expose all AdminServices in HTTP transport. With HTTP transport your credentials and data routed in public channels are vulnerable for sniffing attacks. Use this parameter ONLY if your communication channels are confirmed to be secured by other means.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;ServiceUserRoles&gt;</td>
<td>The default user roles which will be created when the server is started up for the first time.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;EnableEmailUserName&gt;</td>
<td>Enable following config to allow Emails as usernames.</td>
<td>Boo</td>
<td></td>
</tr>
<tr>
<td>&lt;Security&gt;</td>
<td>Security configurations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;KeyStore&gt;</td>
<td>KeyStore which will be used for encrypting/decrypting passwords and other sensitive information.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Location&gt;</td>
<td>Keystore file location.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Type&gt;</td>
<td>Keystore type (JKS/PKCS12 etc.)</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Password&gt;</td>
<td>Keystore password.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;KeyAlias&gt;</td>
<td>Private Key alias.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;KeyPassword&gt;</td>
<td>Private Key password.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;TrustStore&gt;</td>
<td>System wide trust-store which is used to maintain the certificates of all the trusted parties.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Location&gt;</td>
<td>Trust-store file location.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Type&gt;</td>
<td>Trust-store type.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Password&gt;</td>
<td>Trust-store password.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;NetworkAuthenticatorConfig&gt;</td>
<td>The Authenticator configuration to be used at the JVM level. We extend the java.net.Authenticator to make it possible to authenticate to given servers and proxies.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Credential&gt;</td>
<td></td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Pattern&gt;</td>
<td>The pattern that would match a subset of URLs for which this authenticator would be used.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Type&gt;</td>
<td>The type of this authenticator. Allowed values are: 1. server 2. proxy.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Username&gt;</td>
<td>The username used to log in to server/proxy.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Password&gt;</td>
<td>The password used to log in to server/proxy.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;TomcatRealm&gt;</td>
<td>The Tomcat realm to be used for hosted Web applications. Allowed values are: 1. UserManager 2. Memory If this is set to 'UserManager', the realm will pick users &amp; roles from the system's WSO2 User Manager. If it is set to 'memory', the realm will pick users &amp; roles from CARBON_HOME/repository/conf/tomcat/tomcat-users.xml.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;DisableTokenStore&gt;</td>
<td>Option to disable storing of tokens issued by STS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;TokenStoreClassName&gt;</td>
<td>Security token store class name. If this is not set, default class will be org.wso2.carbon.security.util.SecurityTokenStore</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;WorkDirectory&gt;</td>
<td>The temporary work directory.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;HouseKeeping&gt;</td>
<td>House-keeping configuration.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;AutoStart&gt;</td>
<td>True - Start House-keeping thread on server startup false - Do not start House-keeping thread on server startup. The user will run it manually as and when he wishes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Interval&gt;</td>
<td>The interval in <em>minutes</em>, between house-keeping runs.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;MaxTempFileLifetime&gt;</td>
<td>The maximum time in <em>minutes</em>, temp files are allowed to live in the system. Files/directories which were modified more than &quot;MaxTempFileLifetime&quot; minutes ago will be removed by the house-keeping task.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;FileUploadConfig&gt;</td>
<td>Configuration for handling different types of file upload and other file uploading related config parameters. To map all actions to a particular FileUploadExecutor, use &lt;Action&gt;*&lt;/Action&gt;.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;TotalFileSizeLimit&gt;</td>
<td>The total file upload size limit in MB.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Mapping&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Actions&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Action&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Class&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;HttpGetRequestProcessors&gt;</td>
<td>Processors which process special HTTP GET requests such as ?wsdl, ?policy etc. In order to plug in a processor to handle a special request, simply add an entry to this section. The value of the Item element is the first parameter in the query string(e.g. ?wsdl) which needs special processing The value of the Class element is a class which implements org.wso2.carbon.transport.HttpGetRequestProcessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Processor&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Item&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Class&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;DeploymentSynchronizer&gt;</code></td>
<td>Deployment Synchronizer Configuration. Enabled when running with &quot;svn based&quot; dep sync. In master nodes you need to set both AutoCommit and AutoCheckout to true and in worker nodes set only AutoCheckout to true.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Enabled&gt;</code></td>
<td>Boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;AutoCommit&gt;</code></td>
<td>Boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;AutoCheckout&gt;</code></td>
<td>Boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;RepositoryType&gt;</code></td>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SvnUrl&gt;</code></td>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SvnUser&gt;</code></td>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SvnPassword&gt;</code></td>
<td>String</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SvnUrlAppendTenantId&gt;</code></td>
<td>Boolean</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;MediationConfig&gt;</code></td>
<td>Mediation persistence configurations. Only valid if mediation features are available i.e. ESB.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;LoadFromRegistry&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SaveToFile&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Persistence&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;RegistryPersistence&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ServerInitializers&gt;</code></td>
<td>Server initializing code, specified as implementation classes of org.wso2.carbon.core.ServerInitializer. This code will be run when the Carbon server is initialized.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Initializers&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;RequireCarbonServlet&gt;</code></td>
<td>Indicates whether the Carbon Servlet is required by the system, and whether it should be registered.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;H2DatabaseConfiguration&gt;</code></td>
<td>Carbon H2 OSGI Configuration By default non of the servers start. name=&quot;web&quot; - Start the web server with the H2 Console name=&quot;webPort&quot; - The port (default: 8082) name=&quot;webAllowOthers&quot; - Allow other computers to connect name=&quot;webSSL&quot; - Use encrypted (HTTPS) connections name=&quot;tcp&quot; - Start the TCP server name=&quot;tcpPort&quot; - The port (default: 9092) name=&quot;tcpAllowOthers&quot; - Allow other computers to connect name=&quot;tcpSSL&quot; - Use encrypted (SSL) connections name=&quot;pg&quot; - Start the PG server name=&quot;pgPort&quot; - The port (default: 5435) name=&quot;pgAllowOthers&quot; - Allow other computers to connect name=&quot;trace&quot; - Print additional trace information; for all servers name=&quot;baseDir&quot; - The base directory for H2 databases; for all servers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;StatisticsReporterDisabled&gt;</code></td>
<td>Disables the statistics reporter by default.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;EnableHTTPAdminConsole&gt;</code></td>
<td>Enables HTTP for WSO2 servers so that you can access the Admin Console via HTTP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;FeatureRepository&gt;</code></td>
<td>Default Feature Repository of WSO2 Carbon.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;RepositoryName&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML element</td>
<td>Attribute</td>
<td>Description</td>
<td>Data type</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>&lt;Server&gt;</td>
<td></td>
<td>A Server element represents the entire Catalina servlet container. Therefore, it must be the single outermost element in the conf/server.xml configuration file. Its attributes represent the characteristics of the servlet container as a whole.</td>
<td>String</td>
</tr>
<tr>
<td></td>
<td>shutdown</td>
<td>The command string that must be received via a TCP/IP connection to the specified port number, in order to shut down Tomcat.</td>
<td>String</td>
</tr>
</tbody>
</table>

### Configuring catalina-server.xml

Users can change the default configurations by editing the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

**XML Elements**

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;APIManagement&gt;</td>
<td></td>
<td>Configure API Management.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>&lt;Enabled&gt;</td>
<td></td>
<td>Uses the embedded API Manager by default. If you want to use an external API Manager instance to manage APIs, configure below externalAPIManager.</td>
<td>Boolean</td>
<td></td>
</tr>
<tr>
<td>&lt;ExternalAPIManager&gt;</td>
<td></td>
<td>Uncomment and configure API Gateway and Publisher URLs to use external API Manager instance.</td>
<td>String</td>
<td></td>
</tr>
<tr>
<td>&lt;LoadAPIContextsInServerStartup&gt;</td>
<td></td>
<td></td>
<td>Boolean</td>
<td></td>
</tr>
</tbody>
</table>
| port | The TCP/IP port number on which this server waits for a shutdown command. Set to -1 to disable the shutdown port.  
  
  Note: Disabling the shutdown port works well when Tomcat is started using Apache Commons Daemon (running as a service on Windows or with jsvc on un*xes). It cannot be used when running Tomcat with the standard shell scripts though, as it will prevent shutdown.bat|.sh and catalina.bat|.sh from stopping it gracefully. | Int | 8005 |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Service&gt;</td>
<td>A Service element represents the combination of one or more Connector components that share a single Engine component for processing incoming requests. One or more Service elements may be nested inside a Server element.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>The display name of this Service, which will be included in log messages if you utilize standard Catalina components. The name of each Service that is associated with a particular Server must be unique.</td>
<td>String</td>
<td>Catalina</td>
</tr>
<tr>
<td>className</td>
<td>Java class name of the implementation to use. This class must implement the org.apache.catalina.Service interface. If no class name is specified, the standard implementation will be used.</td>
<td>String</td>
<td>org.wso2.carbon.tomcat.ext.service.ExtendedStandardService</td>
</tr>
<tr>
<td><strong>port</strong></td>
<td>The TCP port number on which this Connector will create a server socket and await incoming connections. Your operating system will allow only one server application to listen to a particular port number on a particular IP address. If the special value of 0 (zero) is used, then Tomcat will select a free port at random to use for this connector. This is typically only useful in embedded and testing applications.</td>
<td>Int</td>
<td>9763</td>
</tr>
<tr>
<td>URIEncoding</td>
<td>This specifies the character encoding used to decode the URI bytes, after %xx decoding the URL.</td>
<td>Int</td>
<td>UTF-8</td>
</tr>
<tr>
<td>compressableMimeType</td>
<td>The value is a comma separated list of MIME types for which HTTP compression may be used.</td>
<td>String</td>
<td>text/html,text/javascript,application/x-javascript,application/javascript,application/xml,text/css,application/xslt+xml,text/xsl,image/gif,image/jpg,image/jpeg</td>
</tr>
<tr>
<td>noCompressionUserAgents</td>
<td>The value is a regular expression (using java.util.regex) matching the user-agent header of HTTP clients for which compression should not be used, because these clients, although they do advertise support for the feature, have a broken implementation.</td>
<td>String</td>
<td>gozilla, traviata</td>
</tr>
<tr>
<td>compressionMinSize</td>
<td>If compression is set to &quot;on&quot; then this attribute may be used to specify the minimum amount of data before the output is compressed.</td>
<td>Int</td>
<td>2048</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compression</td>
<td>The Connector may use HTTP/1.1 GZIP compression in an attempt to save server bandwidth. The acceptable values for the parameter is &quot;off&quot; (disable compression), &quot;on&quot; (allow compression, which causes text data to be compressed), &quot;force&quot; (forces compression in all cases), or a numerical integer value (which is equivalent to &quot;on&quot;, but specifies the minimum amount of data before the output is compressed). If the content-length is not known and compression is set to &quot;on&quot; or more aggressive, the output will also be compressed. If not specified, this attribute is set to &quot;off&quot;. Note: There is a tradeoff between using compression (saving your bandwidth) and using the sendfile feature (saving your CPU cycles). If the connector supports the sendfile feature, e.g. the NIO connector, using sendfile will take precedence over compression. The symptoms will be that static files greater that 48 Kb will be sent uncompressed. You can turn off sendfile by setting useSendfile attribute of the connector, as documented below, or change the sendfile usage threshold in the configuration of the DefaultServlet in the default conf/web.xml or in the web.xml of your web application.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>server</td>
<td>Overrides the Server header for the http response. If set, the value for this attribute overrides the Tomcat default and any Server header set by a web application. If not set, any value specified by the application is used. Most often, this feature is not required.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acceptCount</td>
<td>The maximum queue length for incoming connection requests when all possible request processing threads are in use. Any requests received when the queue is full will be refused.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>maxKeepAliveRequests</td>
<td>The maximum number of HTTP requests which can be pipelined until the connection is closed by the server. Setting this attribute to 1 will disable HTTP/1.0 keep-alive, as well as HTTP/1.1 keep-alive and pipelining. Setting this to -1 will allow an unlimited amount of pipelined or keep-alive HTTP requests.</td>
<td>Int</td>
<td>200</td>
</tr>
<tr>
<td>connectionUploadTimeout</td>
<td>Specifies the timeout, in milliseconds, to use while a data upload is in progress. This only takes effect if disableUploadTimeout is set to false.</td>
<td>Int</td>
<td>120000</td>
</tr>
<tr>
<td>disableUploadTimeout</td>
<td>This flag allows the servlet container to use a different, usually longer connection timeout during data upload.</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>minSpareThreads</td>
<td>The minimum number of threads always kept running.</td>
<td>Int</td>
<td>50</td>
</tr>
<tr>
<td>maxThreads</td>
<td>The maximum number of request processing threads to be created by this Connector, which therefore determines the maximum number of simultaneous requests that can be handled. If an executor is associated with this connector, this attribute is ignored as the connector will execute tasks using the executor rather than an internal thread pool.</td>
<td>Int</td>
<td>250</td>
</tr>
<tr>
<td>acceptorThreadCount</td>
<td>The number of threads to be used to accept connections. Increase this value on a multi CPU machine, although you would never really need more than 2. Also, with a lot of non keep alive connections, you might want to increase this value as well.</td>
<td>Int</td>
<td>2</td>
</tr>
<tr>
<td>maxHttpHeaderSize</td>
<td>The maximum size of the request and response HTTP header, specified in bytes.</td>
<td>Int</td>
<td>8192</td>
</tr>
<tr>
<td>bindOnInit</td>
<td>Controls when the socket used by the connector is bound. By default it is bound when the connector is initiated and unbound when the connector is destroyed. If set to false, the socket will be bound when the connector is started and unbound when it is stopped.</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td><strong>redirectPort</strong></td>
<td>If this Connector is supporting non-SSL requests, and a request is received for which a matching <code>&lt;security-constraint&gt;</code> requires SSL transport, Catalina will automatically redirect the request to the port number specified here.</td>
<td><strong>Int</strong></td>
<td><strong>9443</strong></td>
</tr>
<tr>
<td><strong>protocol</strong></td>
<td>Sets the protocol to handle incoming traffic.</td>
<td><strong>String</strong></td>
<td>org.apache.coyote.http11.Http11NioProtocol</td>
</tr>
<tr>
<td><strong>SSLEnabled</strong></td>
<td>Use this attribute to enable SSL traffic on a connector. To turn on SSL handshake/encryption/decryption on a connector set this value to true. The default value is false. When turning this value to true you will want to set the scheme and the secure attributes as well to pass the correct request.getScheme() and request.isSecure() values to the servlets. See SSL Support for more information.</td>
<td><strong>Boolean</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>secure</strong></td>
<td>Set this attribute to true if you wish to have calls to request.isSecure() to return true for requests received by this Connector. You would want this on an SSL Connector or a non SSL connector that is receiving data from a SSL accelerator, like a crypto card, a SSL appliance or even a webserver.</td>
<td><strong>Boolean</strong></td>
<td>true</td>
</tr>
<tr>
<td><strong>scheme</strong></td>
<td>Set this attribute to the name of the protocol you wish to have returned by calls to request.getScheme(). For example, you would set this attribute to &quot;https&quot; for an SSL Connector.</td>
<td><strong>String</strong></td>
<td>https</td>
</tr>
<tr>
<td><strong>clientAuth</strong></td>
<td>Set to true if you want the SSL stack to require a valid certificate chain from the client before accepting a connection. Set to false if you want the SSL stack to request a client Certificate, but not fail if one isn't presented. A false value will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.</td>
<td><strong>Boolean</strong></td>
<td>false</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td>--------</td>
</tr>
<tr>
<td>enableLookups</td>
<td>Set to true if you want calls to request.getRemoteHost() to perform DNS lookups in order to return the actual host name of the remote client. Set to false to skip the DNS lookup and return the IP address in String form instead (thereby improving performance). By default, DNS lookups are disabled.</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>sslProtocol</td>
<td>The SSL protocol(s) to use (a single value may enable multiple protocols - see the JVM documentation for details). The permitted values may be obtained from the JVM documentation for the allowed values for algorithm when creating an SSLContext instance e.g. Oracle Java 6 and Oracle Java 7. Note: There is overlap between this attribute and sslEnabledProtocols.</td>
<td>String</td>
<td>TLS</td>
</tr>
<tr>
<td>keystoreFile</td>
<td>This setting allows you to use separate keystore and security certificates for SSL connections. The location of the keystore file and the keystore password can be given for these parameters. Note that by default, these parameters point to the location and password of the default keystore in the Carbon server.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>keystorePass</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Engine&gt;</td>
<td>The Engine element represents the entire request processing machinery associated with a particular Catalina Service. It receives and processes all requests from one or more Connectors, and returns the completed response to the Connector for ultimate transmission back to the client. Exactly one Engine element MUST be nested inside a Service element, following all of the corresponding Connector elements associated with this Service.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>name</td>
<td>Logical name of this Engine, used in log and error messages. When using multiple Service elements in the same Server, each Engine MUST be assigned a unique name.</td>
<td>String</td>
<td>Catalina</td>
</tr>
<tr>
<td>Attribute</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>---------</td>
<td>---------------------</td>
</tr>
<tr>
<td>defaultHost</td>
<td>The default host name, which identifies the Host that will process requests directed to host names on this server, but which are not configured in this configuration file. This name MUST match the name attributes of one of the Host elements nested immediately inside.</td>
<td>String</td>
<td>localhost</td>
</tr>
<tr>
<td>&lt;Realm&gt;</td>
<td>A Realm element represents a &quot;database&quot; of usernames, passwords, and roles (similar to Unix groups) assigned to those users. Different implementations of Realm allow Catalina to be integrated into environments where such authentication information is already being created and maintained, and then utilize that information to implement Container Managed Security as described in the Servlet Specification. You may nest a Realm inside any Catalina container Engine, Host, or Context). In addition, Realms associated with an Engine or a Host are automatically inherited by lower-level containers, unless explicitly overridden.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>className</td>
<td>Java class name of the implementation to use. This class must implement the org.apache.catalina.Realminterface.</td>
<td>String</td>
<td>org.wso2.carbon.tomcat.ext.realms.CarbonTomcatRealm</td>
</tr>
</tbody>
</table>
| `<Host>` | The Host element represents a virtual host, which is an association of a network name for a server (such as "www.mycompany.com") with the particular server on which Tomcat is running. For clients to be able to connect to a Tomcat server using its network name, this name must be registered in the Domain Name Service (DNS) server that manages the Internet domain you belong to - contact your Network Administrator for more information.

In many cases, System Administrators wish to associate more than one network name (such as www.mycompany.com and company.com) with the same virtual host and applications. This can be accomplished using the Host Name Aliases feature discussed below.

One or more Host elements are nested inside an Engine element. Inside the Host element, you can nest Context elements for the web applications associated with this virtual host. Exactly one of the Hosts associated with each Engine MUST have a name matching the defaultHost attribute of that Engine.

Clients normally use host names to identify the server they wish to connect to. This host name is also included in the HTTP request headers. Tomcat extracts the host name from the HTTP headers and looks for a Host with a matching name. If no match is found, the request is routed to the default host. The name of the default host does not have to match a DNS name (although it can) since any request where the DNS name does not match the name of a Host element will be routed to the default host. |
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Usually the network name of this virtual host, as registered in your Domain Name Service server. Regardless of the case used to specify the host name, Tomcat will convert it to lower case internally. One of the Hosts nested within an Engine MUST have a name that matches the defaultHost setting for that Engine. See Host Name Aliases for information on how to assign more than one network name to the same virtual host.</td>
<td>String</td>
<td>localhost</td>
</tr>
<tr>
<td>appBase</td>
<td>The Application Base directory for this virtual host. This is the pathname of a directory that may contain web applications to be deployed on this virtual host. You may specify an absolute pathname, or a pathname that is relative to the $CATALINA_BASE directory. See Automatic Application Deployment for more information on automatic recognition and deployment of web applications. If not specified, the default of webapps will be used.</td>
<td>String</td>
<td><code>${carbon.home}/repository/deployment/server/webapps/</code></td>
</tr>
<tr>
<td>autoDeploy</td>
<td>This flag value indicates if Tomcat should check periodically for new or updated web applications while Tomcat is running. If true, Tomcat periodically checks the appBase and xmlBase directories and deploys any new web applications or context XML descriptors found. Updated web applications or context XML descriptors will trigger a reload of the web application. See Automatic Application Deployment for more information.</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>deployOnStartup</td>
<td>This flag value indicates if web applications from this host should be automatically deployed when Tomcat starts. See Automatic Application Deployment for more information.</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>unpackWARs</td>
<td>Set to true if you want web applications that are placed in the appBase directory as web application archive (WAR) files to be unpacked into a corresponding disk directory structure, false to run such web applications directly from a WAR file. WAR files located outside of the Host's appBase will not be expanded.</td>
<td>Boolean</td>
<td>true</td>
</tr>
<tr>
<td>&lt;Valve</td>
<td>The Access Log Valve creates log messages.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
files in the same format as those created by standard web servers. These logs can later be analyzed by standard log analysis tools to track page hit counts, user session activity, and so on. The files produced by this Valve are rolled over nightly at midnight. This Valve may be associated with any Catalina container (Context, Host, or Engine), and will record ALL requests processed by that container.

Some requests may be handled by Tomcat before they are passed to a container. These include redirects from /foo to /foo/ and the rejection of invalid requests. Where Tomcat can identify the Context that would have handled the request, the request/response will be logged in the AccessLog(s) associated Context, Host and Engine. Where Tomcat cannot identify the Context that would have handled the request, e.g. in cases where the URL is invalid, Tomcat will look first in the Engine, then the default Host for the Engine and finally the ROOT (or default) Context for the default Host for an AccessLog implementation. Tomcat will use the first AccessLog implementation found to log those requests that are rejected before they are passed to a container.

The output file will be placed in the directory given by the directory attribute. The name of the file is composed by concatenation of the configured prefix, timestamp and suffix. The format of the timestamp in the file name can be set using the fileDateFormat attribute. This timestamp will be omitted if the file rotation is switched off by setting rotatable to false.

Warning: If multiple AccessLogValve instances are used, they should be configured to use different output files.

If sendfile is used, the response bytes will be written asynchronously in a separate thread and the access log valve will not know how many bytes were actually written. In this
case, the number of bytes that was passed to the sendfile thread for writing will be recorded in the access log valve.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
<th>Data Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>Java class name of the implementation to use.</td>
<td>String</td>
<td>org.wso2.carbon.tomcat.ext.valves.CarbonContextCreatorValve, org.apache.catalina.valves.AccessLogValve</td>
</tr>
<tr>
<td>pattern</td>
<td>A formatting layout identifying the various information fields from the request and response to be logged, or the word common or combined to select a standard format.</td>
<td>String</td>
<td>combined</td>
</tr>
<tr>
<td>suffix</td>
<td>The suffix added to the end of each log file name.</td>
<td>String</td>
<td>.log</td>
</tr>
<tr>
<td>prefix</td>
<td>The prefix added to the start of each log file name.</td>
<td>String</td>
<td>http_access_</td>
</tr>
<tr>
<td>directory</td>
<td>Absolute or relative path name of a directory in which log files created by this valve will be placed. If a relative path is specified, it is interpreted as relative to $CATALINA_BASE. If no directory attribute is specified, the default value is &quot;logs&quot; (relative to $CATALINA_BASE).</td>
<td>String</td>
<td>${carbon.home}/repository/logs</td>
</tr>
<tr>
<td>threshold</td>
<td>Minimum duration in seconds after which a thread is considered stuck. If set to 0, the detection is disabled. Note: since the detection is done in the background thread of the Container (Engine, Host or Context) declaring this Valve, the threshold should be higher than the backgroundProcessorDelay of this Container.</td>
<td>Int</td>
<td>600</td>
</tr>
</tbody>
</table>

**Configuring identity.xml**

Users can change the default configurations by editing the `<PRODUCT_HOME>/repository/conf/identity.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

**XML Elements**

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
<th>Data type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Server&gt;</td>
<td>xmlns</td>
<td></td>
<td>String</td>
<td></td>
</tr>
<tr>
<td><code>&lt;JDBCPersistenceManager&gt;</code></td>
<td>Identity related data source configuration.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;DataSource&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Name&gt;</code></td>
<td>Include a data source name (jndiConfigName) from the set of data sources defined in master-datasources.xml. String N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SkipDBSchemaCreation&gt;</code></td>
<td>If the identity database is created from another place and if it is required to skip schema initialization during the server start up, set the property to &quot;true&quot;. Boolean FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OpenID&gt;</code></td>
<td>OpenID related configurations.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OpenIDServerUrl&gt;</code></td>
<td>This is the URL that the OpenID server (servlet) is running in. String N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OpenIDUserPattern&gt;</code></td>
<td>URL of the pattern that can be configured for the user’s OpenID. String N/A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OpenIDSkipUserConsent&gt;</code></td>
<td>Set to false if the users must be prompted for approval. Boolean FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OpenIDRememberMeExpiry&gt;</code></td>
<td>Expiry time of the OpenID RememberMe token in minutes. Int 0 Minute</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;UseMultifactorAuthentication&gt;</code></td>
<td>Multifactor authentication configuration. Boolean FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;DisableOpenIDDumbMode&gt;</code></td>
<td>To enable or disable OpenID dumb mode. Boolean FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SessionTimeout&gt;</code></td>
<td>OpenID session timeout in seconds. Int 36000 S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;AcceptSAMLSSOLogin&gt;</code></td>
<td>Skips authentication if the valid SAML2 Web SSO browser session is available. Boolean FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;ClaimsRetrieverImplClass&gt;</code></td>
<td>User claim retrieving module for OpenID.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OAuth&gt;</code></td>
<td>OAuth related configurations.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;AccessTokenDefaultValidityPeriod&gt;</code></td>
<td>Default validity period for Access Token in seconds. Int 3600 Sec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;TimestampSkew&gt;</code></td>
<td>Timestamp skew in seconds. Int 300 Sec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;EnableOAuthCache&gt;</code></td>
<td>Enable OAuth caching. This cache has the replication support. Boolean TRUE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>&lt;TokenPersistencePreprocessor&gt;</td>
<td>Configure the security measures needed to be done prior to storing the token in the database, such as hashing, encrypting, etc.</td>
<td>String</td>
<td>org.wso2.carbon.identity.oauth2.preprocessor.PlainTokenPersistencePreprocessor</td>
<td></td>
</tr>
<tr>
<td>&lt;SupportedResponseTypes&gt;</td>
<td>Supported OAuth2.0 response types.</td>
<td>String</td>
<td>token, code</td>
<td></td>
</tr>
<tr>
<td>&lt;SupportedGrantTypes&gt;</td>
<td>Supported OAuth2.0 grant types.</td>
<td>String</td>
<td>authorization, refresh_token, password, id_token</td>
<td></td>
</tr>
<tr>
<td>&lt;OAuthCallbackHandlers&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;OAuthCallbackHandler&gt;</td>
<td>OAuth callback handler module class name.</td>
<td>String</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>&lt;EnableAssertions&gt;</td>
<td>Assertions can be used to embed parameters into the access token.</td>
<td>Boolean</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>&lt;UserName&gt;</td>
<td>This enables you to add the user name as an additional parameter if you require it.</td>
<td>Boolean</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>&lt;EnableAccessTokenPartitioning&gt;</td>
<td>This should be set to true when using multiple user stores and keys should be saved into different tables according to the user store. By default, all the application keys are saved into the same table. UserName Assertion should be 'true' to use this.</td>
<td>Boolean</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>&lt;AccessTokenPartitioningDomains&gt;</td>
<td>This includes the user store domain names and mapping to the new table name. E.g., if you provide 'A:foo.com', foo.com should be the user store domain name and 'A' represents the relevant mapping of the token store table, i.e., tokens will be added to a table called IDN_OAUTH2_ACCESS_TOKEN_A.</td>
<td>String</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>&lt;AuthorizationContextTokenGeneration&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Enabled&gt;</td>
<td>This mentions whether token generation is enabled or not.</td>
<td>Boolean</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>&lt;TokenGeneratorImplClass&gt;</td>
<td>Token generation class name.</td>
<td>String</td>
<td>org.wso2.carbon.identity.oauth2.token.JWTTokenGenerator</td>
<td></td>
</tr>
<tr>
<td>&lt;ClaimsRetrieverImplClass&gt;</td>
<td>Claim retrieving class name for generating a token.</td>
<td>String</td>
<td>org.wso2.carbon.identity.oauth2.token.DefaultClaimsRetriever</td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ConsumerDialectURI&gt;</code></td>
<td>Claim Dialect URI that is used for claim retrieving.</td>
<td></td>
<td><a href="http://ws">http://ws</a></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SignatureAlgorithm&gt;</code></td>
<td>Signature algorithm used for sign the token.</td>
<td></td>
<td>SHA256</td>
<td></td>
</tr>
<tr>
<td><code>&lt;AuthorizationContextTTL&gt;</code></td>
<td>Token time to live value.</td>
<td>Long</td>
<td>15 Min</td>
<td></td>
</tr>
<tr>
<td><code>&lt;SAML2Grant&gt;</code></td>
<td>Configuration related to SAML2 Grant type.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;OpenIDConnect&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;IDTokenBuilder&gt;</code></td>
<td>IDToken generator implementation class name.</td>
<td>String</td>
<td>org.wso2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;IDTokenIssuerID&gt;</code></td>
<td>The value of TokenIssuerID of the IDToken. This is a unique value and should be changed according to the deployment values.</td>
<td>String</td>
<td>OIDCAu</td>
<td></td>
</tr>
<tr>
<td><code>&lt;IDTokenSubjectClaim&gt;</code></td>
<td>This is the claim used as the subject of the IDToken. You can use different claims such as <a href="http://wso2.org/claims/emailaddress">http://wso2.org/claims/emailaddress</a>.</td>
<td>String</td>
<td><a href="http://ws">http://ws</a></td>
<td></td>
</tr>
<tr>
<td><code>&lt;IDTokenCustomClaimsCallbackHandler&gt;</code></td>
<td>Claim callback implementation class name. This is used to return custom claims with the IDToken.</td>
<td>String</td>
<td>org.wso2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;IDTokenExpiration&gt;</code></td>
<td>The expiration value of the IDToken in seconds.</td>
<td>Int</td>
<td>300 Sec</td>
<td></td>
</tr>
<tr>
<td><code>&lt;UserInfoEndpointClaimDialect&gt;</code></td>
<td>Defines which claim dialect should be returned from the User Endpoint.</td>
<td>String</td>
<td><a href="http://ws">http://ws</a></td>
<td></td>
</tr>
<tr>
<td><code>&lt;UserInfoEndpointClaimRetriever&gt;</code></td>
<td>Defines the implementation name of the class which builds the claims for the user info endpoint's response.</td>
<td>String</td>
<td>org.wso2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;UserInfoEndpointRequestValidator&gt;</code></td>
<td>Implementation name of the class that validates the user info request against the specification.</td>
<td>String</td>
<td>org.wso2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;UserInfoEndpointAccessTokenValidator&gt;</code></td>
<td>Implementation name of the class that validates the access token.</td>
<td>String</td>
<td>org.wso2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;UserInfoEndpointResponseBuilder&gt;</code></td>
<td>Implementation name of the class that builds the user info request.</td>
<td>String</td>
<td>org.wso2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;SkipUserConsent&gt;</code></td>
<td>Set to false if the users must be prompted for approval.</td>
<td>Boolean</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td><code>&lt;MultifactorAuthentication&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;XMPPSettings&gt;</code></td>
<td>XMPP setting for multifactor authentication.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;XMPPConfig&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;XMPPProvider&gt;</code></td>
<td>XMPP provider name.</td>
<td>String</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Copyright © WSO2 Inc. 2014
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;XMPPServer&gt;</td>
<td>XMPP server name.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;XMPPPort&gt;</td>
<td>XMPP server's port.</td>
<td>Int</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;XMPPExt&gt;</td>
<td>XMPP domain.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;XMPPUserName&gt;</td>
<td>User name for login to XMPP server.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;XMPPPassword&gt;</td>
<td>Password for login to XMPP server.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;SSOService&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;IdentityProviderURL&gt;</td>
<td>Unique identifier for IDP. This would be passed as Issuer in SAML2 response.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;SingleLogoutRetryCount&gt;</td>
<td>Number of retries that must be done if a single logout request is not received from the SP.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;SingleLogoutRetryInterval&gt;</td>
<td>Interval between two re-tries.</td>
<td>Int</td>
<td>60 Sec</td>
</tr>
<tr>
<td>&lt;TenantPartitioningEnabled&gt;</td>
<td>This would add the tenant domain as parameter into the ACS URL.</td>
<td>Boolean</td>
<td>FALSE</td>
</tr>
<tr>
<td>&lt;SessionTimeout&gt;</td>
<td>Remember me session timeout in seconds.</td>
<td>Int</td>
<td>36000 S</td>
</tr>
<tr>
<td>&lt;AttributesClaimDialect&gt;</td>
<td>Claim Dialect URI that is used for claim retrieving.</td>
<td>String</td>
<td><a href="http://ws">http://ws</a></td>
</tr>
<tr>
<td>&lt;AcceptOpenIDLogin&gt;</td>
<td>Skips authentication if the valid OpenID login session is available.</td>
<td>Boolean</td>
<td>FALSE</td>
</tr>
<tr>
<td>&lt;ClaimsRetrieverImplClass&gt;</td>
<td>Claim retrieving class name name for generating a token.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;SAMLResponseValidityPeriod&gt;</td>
<td>SAML Token validity period in minutes.</td>
<td>Int</td>
<td>5 Minute</td>
</tr>
<tr>
<td>&lt;UseAuthenticatedUserDomainCrypto&gt;</td>
<td>When set to true, this is useful in tenant mode setup with older versions of API Manager. This indicates that the SAML2 SSO SAML Response must be signed using the authenticated user's tenant keystore.</td>
<td>Boolean</td>
<td>FALSE</td>
</tr>
<tr>
<td>&lt;EntitlementSettings&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ThriftBasedEntitlementConfig&gt;</td>
<td>Thrift transport configurations for entitlement service.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;EnableThriftService&gt;</td>
<td>Enable thrift transport.</td>
<td>Boolean</td>
<td>FALSE</td>
</tr>
<tr>
<td>&lt;ReceivePort&gt;</td>
<td>Thrift listening port.</td>
<td>Int</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;ClientTimeout&gt;</td>
<td>Thrift session time out in seconds.</td>
<td>Int</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;KeyStore&gt;</td>
<td>Thrift key store configurations used for SSL.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Configuring master-datasources.xml

Users can change the default configurations by editing the `<PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml` file using the information in the following table.

**XML Elements**

Click on the table and use the left and right arrow keys to scroll horizontally. For sample values, see the [Example](#) below the table.

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;datasources-configuration&gt;</code></td>
<td>xmlns</td>
<td>The root element. The namespace is specified as: xmlns:svns=&quot;<a href="http://org.wso2.securevault/configuration">http://org.wso2.securevault/configuration</a>&quot;</td>
<td></td>
</tr>
<tr>
<td><code>&lt;providers&gt;</code></td>
<td></td>
<td>The container element for the datasource providers.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;provider&gt;</code></td>
<td></td>
<td>The datasource provider, which should implement org.wso2.carbon.ndatasource.common.spi.DataSourceReader. The datasources follow a pluggable model in providing datasource type implementations using this approach.</td>
<td>Fully qualified Java class</td>
</tr>
<tr>
<td><code>&lt;datasources&gt;</code></td>
<td></td>
<td>The container element for the datasources.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;datasource&gt;</code></td>
<td></td>
<td>The root element of a datasource.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td></td>
<td>Name of the datasource.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;description&gt;</code></td>
<td></td>
<td>Description of the datasource.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;jndiConfig&gt;</code></td>
<td></td>
<td>The container element that allows you to expose this datasource as a JNDI datasource.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td></td>
<td>The JNDI resource name to which this datasource will be bound.</td>
<td>String</td>
</tr>
</tbody>
</table>
**<environment>**

The container element in which you specify the following JNDI properties:

- `java.naming.factory.initial`: Selects the registry service provider as the initial context.
- `java.naming.provider.url`: Specifies the location of the registry when the registry is being used as the initial context.

**<definition>**

The container element for the data source definition. Set the type attribute to RDBMS, or to custom if you're creating a custom type. The "RDBMS" data source reader expects a "configuration" element with the sub-elements listed below.

**<configuration>**

The container element for the RDBMS properties.

**<url>**

The connection URL to pass to the JDBC driver to establish the connection.

**<username>**

The connection user name to pass to the JDBC driver to establish the connection.

**<password>**

The connection password to pass to the JDBC driver to establish the connection.

**<driverClassName>**

The class name of the JDBC driver to use. Fully qualified Java class

**<maxActive>**

The maximum number of active connections that can be allocated from this pool at the same time.

**<maxWait>**

Maximum number of milliseconds that the pool waits (when there are no available connections) for a connection to be returned before throwing an exception.

**<testOnBorrow>**

Specifies whether objects will be validated before being borrowed from the pool. If the object fails to validate, it will be dropped from the pool, and we will attempt to borrow another. When set to true, the `validationQuery` parameter must be set to a non-null string.

**<validationQuery>**

The SQL query used to validate connections from this pool before returning them to the caller. If specified, this query does not have to return any data, it just can't throw a SQLException. The default value is null. Example values are `SELECT 1(mysql)`, `select 1 from dual(oracle)`, `SELECT 1(MS Sql Server)`. String
<validationInterval>
To avoid excess validation, only run validation at most at this frequency (interval time in milliseconds). If a connection is due for validation, but has been validated previously within this interval, it will not be validated again. The default value is 30000 (30 seconds).
</validationInterval>

**Example**

```xml
<datasources-configuration
xmlns:svns="http://org.wso2.securevault/configuration">
  <providers>
    <provider>
      org.wso2.carbon.ndatasource.rdbms.RDBMSDataSourceReader
    </provider>
  </providers>
  <datasources>
    <datasource>
      <name>WSO2_CARBON_DB</name>
      <description>The datasource used for registry and user manager</description>
      <jndiConfig>
        <name>jdbc/WSO2CarbonDB</name>
      </jndiConfig>
      <definition type="RDBMS">
        <configuration>
          <url>
            jdbc:h2:repository/database/WSO2CARBON_DB;DB_CLOSE_ON_EXIT=FALSE;LOCK_TIME_OUT=60000
          </url>
          <username>wso2carbon</username>
          <password>wso2carbon</password>
          <driverClassName>org.h2.Driver</driverClassName>
          <maxActive>50</maxActive>
          <maxWait>60000</maxWait>
          <testOnBorrow>true</testOnBorrow>
          <validationQuery>SELECT 1</validationQuery>
          <validationInterval>30000</validationInterval>
        </configuration>
      </definition>
    </datasource>
  </datasources>
</datasources-configuration>
```

**Configuring registry.xml**

Users can change the default configurations by editing the `<PRODUCT_HOME>/repository/conf/registry.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

**XML Elements**
<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;wso2registry&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;currentDBConfig&gt;</td>
<td></td>
<td>The server can only handle one active configuration at a time. The currentDBConfig parameter defined in the registry.xml is used to specify the database configuration that is active at present. The valid name of a database configuration defined on the registry.xml file. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Data+base+Configuration+Details">database+Configuration+Details</a></td>
</tr>
<tr>
<td>&lt;readOnly&gt;</td>
<td></td>
<td>To run the registry in read-only mode, set the readOnly element to true. Setting the read-only mode allows you to run an immutable instance of registry repository. This setting is valid on a global level. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Registry+Configuration+Details">registry+Configuration+Details</a></td>
</tr>
<tr>
<td>&lt;enableCache&gt;</td>
<td></td>
<td>To enable registry caching, set the enableCache element to true. Once caching is enabled, repetitive read operations will be executed against the cache instead of the database. This setting is valid on a global level.</td>
</tr>
<tr>
<td>&lt;registryRoot&gt;</td>
<td></td>
<td>The registryRoot parameter can be used to define the apparent root of the running instance of the server. This setting is valid on a global level. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Registry+Configuration+Details">registry+Configuration+Details</a></td>
</tr>
<tr>
<td>&lt;dbConfig&gt;</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>&lt;dataSource&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;handler&gt;</td>
<td>class</td>
<td>Handlers are pluggable components, that contain custom processing logic for handling resources. All handlers extend an abstract class named Handler, which provides default implementation utilities useful for concrete Handler implementations. Handler implementations can provide alternative behaviors for basic resource related operations, by overwriting one or more methods in the Handler class. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Handler+Configuration+Details">handler+Configuration+Details</a></td>
</tr>
<tr>
<td>&lt;filter&gt;</td>
<td>class</td>
<td></td>
</tr>
<tr>
<td>&lt;remoteInstance&gt;</td>
<td></td>
<td>In order to mount an external registry, you have to define the remoteInstance. This could use either the JDBC-based configuration, the Atom-based configuration model or the WebService-based configuration model. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Remote+Instance+and+Mount+Configuration+Details">remote+Instance+and+Mount+Configuration+Details</a></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>url</td>
<td>The URL of the remote instance.</td>
<td></td>
</tr>
<tr>
<td>&lt;ID&gt;</td>
<td>Remote instance ID.</td>
<td></td>
</tr>
<tr>
<td>&lt;username&gt;</td>
<td>Username of the remote registry login.</td>
<td></td>
</tr>
<tr>
<td>&lt;password&gt;</td>
<td>Password of the remote registry login.</td>
<td></td>
</tr>
<tr>
<td>&lt;dbConfig&gt;</td>
<td>The database configuration to use.</td>
<td></td>
</tr>
<tr>
<td>&lt;readOnly&gt;</td>
<td>To run the registry in read-only mode set the readOnly element to true. The registry will be an immutable instance of the registry repository. This setting is valid only for the specific remote instance. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Registry+Configuration+Details">http://docs.wso2.org/display/Governance501/Registry+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td>&lt;enableCache&gt;</td>
<td>To enable registry caching, set the enableCache element to true. The operations will be executed against the cache instead of the database. This setting is valid only for the specific remote instance.</td>
<td></td>
</tr>
<tr>
<td>&lt;registryRoot&gt;</td>
<td>The registryRoot parameter can be used to define whether the application is running in read-only mode. This setting is valid only for the specific remote instance. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Registry+Configuration+Details">http://docs.wso2.org/display/Governance501/Registry+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td>&lt;mount&gt;</td>
<td>Once a remote instance has been defined, a collection on the remote instance can be mounted to the local instance. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Remote+Instance+and+Mount+Configuration+Details">http://docs.wso2.org/display/Governance501/Remote+Instance+and+Mount+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td>path</td>
<td>The path to which the mount will be added to.</td>
<td></td>
</tr>
<tr>
<td>overwrite</td>
<td>Whether an existing collection at the given path would be overwritten or not.</td>
<td></td>
</tr>
<tr>
<td>&lt;instanceID&gt;</td>
<td>Remote instance ID.</td>
<td></td>
</tr>
<tr>
<td>&lt;targetPath&gt;</td>
<td>The path on the remote registry.</td>
<td></td>
</tr>
<tr>
<td>&lt;versionResourcesOnChange&gt;</td>
<td>You can configure whether you want to auto-version the resources (non-collection) by setting the versionResourcesOnChange element to true. In this configuration it will create a version for the resources whenever they are updated. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources">http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources</a></td>
<td></td>
</tr>
<tr>
<td>&lt;staticConfiguration&gt;</td>
<td>While most configuration options can be changed after the first run of the server, the staticConfiguration parameter is an exception. Any Static Configuration changes will not take effect until the server is restarted. For more information, see the Governance Registry documentation here: <a href="http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources">http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources</a></td>
<td></td>
</tr>
<tr>
<td>&lt;versioningProperties&gt;</td>
<td>Whether the properties are versioned when a snapshot is created.</td>
<td></td>
</tr>
<tr>
<td>&lt;versioningComments&gt;</td>
<td>Whether the comments are versioned when a snapshot is created.</td>
<td></td>
</tr>
<tr>
<td>&lt;versioningTags&gt;</td>
<td>Whether the tags are versioned when a snapshot is created.</td>
<td></td>
</tr>
<tr>
<td>&lt;versioningRatings&gt;</td>
<td>Whether the ratings are versioned when a snapshot is created.</td>
<td></td>
</tr>
</tbody>
</table>
Configuring user-mgt.xml

Users can change the default user management functionality related configurations by editing the `<PRODUCT_HOME>/repository/conf/user-mgt.xml` file using the information given below.

Click on the table and use the left and right arrow keys to scroll horizontally.

**XML Elements**

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
<th>Data type</th>
<th>Default value</th>
<th>Mandatory/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;UserManager&gt;</code></td>
<td></td>
<td>User kernel configuration for Carbon server.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Realm&gt;</code></td>
<td></td>
<td>Realm configuration.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Configuration&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;AddAdmin&gt;</code></td>
<td></td>
<td>Specifies whether the admin user and admin role will be created in the primary user store. This element enables the user to create additional admin users in the user store. If the <code>&lt;AdminUser&gt;</code> element does not exist in the external user store, it will be automatically created only if this property is set to true. If the value is set to false, the given admin user and role should already exist in the external user store.</td>
<td>Boolean</td>
<td>true</td>
<td>Mandatory</td>
</tr>
<tr>
<td><code>&lt;AdminRole&gt;</code></td>
<td></td>
<td>The role name that is used as an admin role for the Carbon server.</td>
<td>String</td>
<td>N/A</td>
<td>Mandatory</td>
</tr>
<tr>
<td><code>&lt;AdminUser&gt;</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td><code>&lt;UserName&gt;</code></td>
<td>User name that is used to represent an admin user for the Carbon server.</td>
<td>String</td>
<td>N/A</td>
<td>Mandatory</td>
</tr>
<tr>
<td>d&gt;</td>
<td><code>&lt;Password&gt;</code></td>
<td>Password of the admin user. If the admin user needs to be created in the Carbon server.</td>
<td>String</td>
<td>N/A</td>
<td>Optional</td>
</tr>
<tr>
<td></td>
<td><code>&lt;EveryOneRo leName&gt;</code></td>
<td>By default, every user in the user store is assigned to this role.</td>
<td>String</td>
<td>N/A</td>
<td>Mandatory</td>
</tr>
<tr>
<td></td>
<td><code>&lt;Property&gt;</code></td>
<td>User realm configuration specific property values.</td>
<td>String</td>
<td>N/A</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>
User Store manager implementation classes and their configurations for use realm. Use the ReadOnlyLDAPUserStoreManager to do read-only operations for external LDAP user stores.

To do both read and write operations, use the ReadWriteLDAPUserStoreManager for external LDAP user stores.

If you wish to use an Active Directory Domain Service (AD DS) or Active Directory Lightweight Directory Service (AD LDS), use the ActiveDirectoryUserStoreManager. This can be used for both read-only and read/write operations.

Use JDBCUserStoreManager for both internal and external JDBC user stores.

Configuring config-validation.xml

The <PRODUCT_HOME>/repository/conf/etc/config-validation.xml file contains the recommended system configurations for a server. When you start the server, the system configurations will be validated against these recommendations, and warnings will be published if conflicts are found. See more details on system requirements for your product on Installation Prerequisites, and the procedure for starting a server in Running the Product.

Given below are the default recommendations in the config-validation.xml file. If required, you may change some of these recommendations on this file according to the conditions in your production environment.

System Validation

Following are the system parameter values recommended for the purpose of running a WSO2 product server.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Parameter Description</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Required processor speed.</td>
<td>800</td>
</tr>
<tr>
<td>RAM</td>
<td>Required RAM in your environment.</td>
<td>2048</td>
</tr>
<tr>
<td>swap</td>
<td>Required space in hard disk to use for virtual memory.</td>
<td>2048</td>
</tr>
<tr>
<td>freeDisk</td>
<td>Free disk space required in your environment.</td>
<td>1024</td>
</tr>
</tbody>
</table>
| ulimit            | The limit of resources per user. This value indicates the limit on the number of file descriptors a process may have. This property is specified in the product startup script as shown below. For example, see the product startup script for Linux: <PRODUCT_HOME>/bin/wso2server.sh:

```
#ulimit -n 100000
```

<table>
<thead>
<tr>
<th></th>
<th>Parameter Description</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>initHeapSize</td>
<td>The initial heap size that applies if the JVM requires more memory than is allocated by default.</td>
<td>256</td>
</tr>
<tr>
<td>maxHeapSize</td>
<td>The maximum heap size that applies if the JVM requires more memory than is allocated by default.</td>
<td>512</td>
</tr>
<tr>
<td>maxPermGenSize</td>
<td>The maximum heap size of the permanent generation of heap.</td>
<td>256</td>
</tr>
</tbody>
</table>

If the values set for these parameters in your environment are less than the recommendations, the following warnings will be published when you start your server.

- WARN - ValidationResultPrinter CPU speed (MHz): <systemCPU> of the system is below the recommended minimum speed : <recommended value>
- WARN - ValidationResultPrinter RAM size (MB): <systemRam> of the system is below the recommended minimum size : <recommended value>
- WARN - ValidationResultPrinter Swap Memory size (MB): <systemSwap> of the system is below the recommended minimum size : <recommended value>
- WARN - ValidationResultPrinter Maximum free Disk Space (MB): <systemDisk> of the system is below the recommended minimum size : <recommended value>
- WARN - ValidationResultPrinter Open files limit : <openFileLimit> of the system is below the recommended minimum count : <recommended value>

**JVM Validation**

The following JVM heap size values are recommended by default in the config-validation.xml file.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Parameter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>initHeapSize</td>
<td>The initial heap size that applies if the JVM requires more memory than is allocated by default.</td>
<td>256</td>
</tr>
<tr>
<td>maxHeapSize</td>
<td>The maximum heap size that applies if the JVM requires more memory than is allocated by default.</td>
<td>512</td>
</tr>
<tr>
<td>maxPermGenSize</td>
<td>The maximum heap size of the permanent generation of heap.</td>
<td>256</td>
</tr>
</tbody>
</table>

These parameters are specified in the product startup script as shown below, where, "-Xms", "-Xmx" and "-XX" correspond to "initHeapSize", "maxHeapSize" and "maxPermGenSize" respectively. For example, see the product startup script for Linux: <PRODUCT_HOME>/bin/wso2server.sh.
If these heap size values in your product startup script are less than the recommended values, the following warnings will be published when you start your server:

- WARN - ValidationResultPrinter Initial Heap Memory (MB) : <system value> of the running JVM is set below the recommended minimum size :<recommended value>
- WARN - ValidationResultPrinter Maximum Heap Memory (MB) : <system value> of the running JVM is set below the recommended minimum size :<recommended value>
- WARN - ValidationResultPrinter Maximum PermGen space (MB) :<system value> of the running JVM is set below the recommended minimum size :<recommended value>

**System Property Validation**

According to the `config-validation.xml` file, values are required to be specified for the following properties in your system. Note that it is not recommended to remove this validations as these are mandatory settings.

- carbon.home
- carbon.config.dir.path
- axis2.home

The `carbon.home` and `carbon.config.dir.path` properties are given in the product startup script as shown below. For example, see the product startup script for Linux: `<PRODUCT_HOME>/bin/wso2server.sh`

```
$JAVA_OPTS 
-Dcarbon.home="$CARBON_HOME" 
-Dcarbon.config.dir.path="$CARBON_HOME/repository/conf" 
```

The `axis2.home` property is given in the product startup script as shown below. For example, see the product startup script for Linux: `<PRODUCT_HOME>/bin/wso2server.sh`

```
# Set AXIS2_HOME. Needed for One Click JAR Download 
AXIS2_HOME=$CARBON_HOME 
```

If the values for these properties are null in the product startup script, the following warning message will be published when you start the server: "Value is not set for the required system property : <property-value>".

**Supported OS Validation**

The product has been tested for compatibility with the following operating systems, which are listed in the `config-validation.xml` file. Therefore, by default, the system is validated against these operating systems.

- Linux
- Unix
- Mac OS
- Windows Server 2003
- Windows XP
- Windows Vista
- Windows 7
- Mac OS X
If the OS in your environment is not included in this list, the following warning message will be published when you start the server: "WARN - ValidationResultPrinter The running OS : <value> is not a tested Operating System for running WSO2 Carbon."

Configuring sso-sp-config.properties

The sso-sp-config.properties file is the global configuration file for generic SSO configurations in AS.

- **File Details**
- **Descriptions of Properties**

**File Details**

- **File Name:** sso-sp-config.properties
- **Location:** <PRODUCT_HOME>/repository/conf/security

**Descriptions of Properties**

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EnableSAML2SSOLogin=true</td>
<td>Use this property to enable users to log in using SAML2SSO.</td>
</tr>
<tr>
<td>EnableOpenIDLogin=false</td>
<td>Use this property to enable users to log in using OpenID.</td>
</tr>
<tr>
<td>SkipURIs=</td>
<td>URIs to skip authentication.</td>
</tr>
<tr>
<td>handleConsumerURLAfterSLO=true</td>
<td>This property is used to automatically redirect the consumer URL from the acs page after SLO.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Value</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>LoginUrl</td>
<td>index.jsp</td>
</tr>
<tr>
<td>SAML2SSOURL</td>
<td>samlso</td>
</tr>
<tr>
<td>SAML.IssuerID</td>
<td>avis.com</td>
</tr>
<tr>
<td>ApplicationServerURL</td>
<td><a href="https://localhost:9444">https://localhost:9444</a></td>
</tr>
<tr>
<td>SAML.ConsumerUrlPostFix</td>
<td>/acs</td>
</tr>
<tr>
<td>SAML.ConsumerUrl</td>
<td><a href="http://localhost:8080/avis.com/home.jsp">http://localhost:8080/avis.com/home.jsp</a></td>
</tr>
<tr>
<td>SAML2.IdPURL</td>
<td><a href="https://localhost:9443/samlso">https://localhost:9443/samlso</a></td>
</tr>
<tr>
<td>SAML2.IdPEntityId</td>
<td><a href="https://localhost:9443/samlso">https://localhost:9443/samlso</a></td>
</tr>
<tr>
<td>SSOAgentSessionBeanName</td>
<td>SSOAgentSessionBean</td>
</tr>
<tr>
<td>Property</td>
<td>Value</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>SAML.AttributeConsumingServiceIndex</td>
<td>1701087467</td>
</tr>
<tr>
<td>SAML2.RelayState</td>
<td>index.jsp</td>
</tr>
<tr>
<td>SAML2.EnableSLO</td>
<td>true</td>
</tr>
<tr>
<td>SAML2.SLOURL</td>
<td>logout</td>
</tr>
<tr>
<td>SAML2.EnableResponseSigning</td>
<td>true</td>
</tr>
<tr>
<td>SAML2.EnableAssertionEncryption</td>
<td>false</td>
</tr>
<tr>
<td>SAML2.EnableRequestSigning</td>
<td>true</td>
</tr>
<tr>
<td>SAML2.IsForceAuthn</td>
<td>false</td>
</tr>
<tr>
<td>SAML2.IsPassiveAuthn</td>
<td>false</td>
</tr>
<tr>
<td>SAML.SSOAgentCredentialImplClass</td>
<td>org.wso2.carbon.identity.sso.agent.saml.SSOAgentKeyStoreCredential</td>
</tr>
<tr>
<td>SAML.KeyStorePassword</td>
<td>wso2carbon</td>
</tr>
</tbody>
</table>
SAML.IdPCertAlias=wso2carbon

SAML.PrivateKeyAlias=wso2carbon

SAML.PrivateKeyPassword=wso2carbon

SAML.Request.Query.Param=&forceAuth=true

Changing the Default Ports

When you run multiple WSO2 products, multiple instances of the same product, or multiple WSO2 product clusters on the same server or virtual machines (VMs), you must change their default ports with an offset value to avoid port conflicts. Port offset defines the number by which all ports defined in the runtime such as the HTTP/S ports will be changed. For example, if the default HTTP port is 9763 and the port offset is 1, the effective HTTP port will change to 9764. For each additional WSO2 product instance, you set the port offset to a unique value.

Default Ports:

The list of default ports used by all WSO2 products, including WSO2 AS, is given in the administration guide of the WSO2 Carbon platform.

Setting port offset for the server

The default port offset value is 0. There are two ways to set an offset to a port:

- Pass the port offset to the server during startup. The following command starts the server with the default port incremented by 3:
  ```bash
  ./wso2server.sh -DportOffset=3
  ```
- Set the Ports section of `<PRODUCT_HOME>/repository/conf/carbon.xml` as follows:
  ```xml
  <Offset>3</Offset>
  ```

When you set the server-level port offset in WSO2 AS as shown above, all the ports used by the server will change automatically. However, this may not be the case with some WSO2 products. See the documentation for your respective products for information on how this server-level port offset affects other ports in the server.

Tools

The following tools are available in WSO2 AS. Some of the tools are inherited from the WSO2 Carbon platform.

- Try It Tool
- Generate Java Code for WSDL
- Generate WSDL for Java Code
- Validating a WSDL
Try It Tool

- Invoking an external service through Try It
- Invoking Try It through a Axis web service's dashboard
- Invoking Try It through a JAX-WS application's dashboard
- Customizing Try It options
  - Choosing endpoints
  - Viewing the service information
  - Prioritizing Operations

Invoking an external service through Try It

The Try It tool is provided by WSO2 as a quick and easy way to test your Axis2 or JAX-WS services using its WSDL. You can also test any service that has a publicly-available WSDL of document/literal format as well. Try It provides you with a mechanism to test your WSDL by creating endpoints on the fly. It helps to test a WSDL before actually coding it, without the need of a third-party WSDL validator tool. This functionality is provided by the following feature of the WSO2 feature repository:

Name: WSO2 Carbon - Try It Feature
Identifier: org.wso2.carbon.tryit.feature.group

If the above feature is not included in your product by default, you can install it using the instructions given in section Feature Management.

The Try It tool does not support relative schema imports and WSDL imports at the moment. WSO2 will provide this feature in a future release.

Follow the instructions below to invoke the Try It tool.

1. Log on to the product's management console.
2. On the Tools menu, click Try It.
3. Type or paste the location of the WSDL. This may be a local file system path or a web URL.
4. Click Try It.

If your browser has "Pop up Blocking" enabled, you will receive a message saying a pop up was blocked. Allow pop ups for the current domain and try again. The available operations in the given WSDL appears. For operations that take arguments you will see primitive argument-type fields. The values specified in these fields will be passed to the operations. For no-argument operations you will only see a button with the same name as the operation.
5. There are two text areas on the **Try It** page, each for request and response. You can switch the layout using either **Horizontal** or **Vertical** buttons.

6. Fill in the necessary parameters.
7. Click **Send** to invoke the operation.
   
   The return value immediately appears in the response text area. For more information, see **Try It Tool#Customizing Try It options**.

**Invoking Try It through a Axis web service's dashboard**

The **Try It** page is a full Ajax client and interacts with the service just as any other client would, so it is a good way to verify that a service not only works, but is reachable by its clients. However, note that the **Try It** page attempts to contact the Web service at the endpoint URL provided. If the domain or URL scheme of the service is different from the endpoint, the browser may impose security restrictions. In this case, the Try It falls back to using a script-injection protocol to tunnel the Web service request. A wire trace of the communication with the browser will not show the actual Web service messages used to communicate with the service.

In addition, since the Try It page is fully asynchronous, it doesn’t block while a Web service is being invoked. You can invoke several operations before one of them completes (though it does not have a way to display the results of the same operation invoked multiple times).

The Try It tool can also be invoked from the dashboard of a selected service. Service dashboard is part of the service management functionality provided by the following feature:

**Name**: WSO2 Carbon - Service Management Feature

**Identifier**: org.wso2.carbon.service.mgt.feature.group

The above feature is bundled by default in all service-hosting WSO2 middleware products. If the product you are using does not have this feature installed, you can do so by following the instructions given in section **Feature Management**.

Follow the instructions below to try out the WSDL of a selected service.

1. Log on to the product’s management console.
2. On the **Main** menu, under **Services**, click **List**.
   
   The **Deployed Services** page appears, which lists out all the currently active services.
3. Click the **Try this service** link, respective to the selected service.
The Try this service link is also available when viewing the respective service dashboard.

When you click the Try this service link, you will be directed to the page where the operations available for your service will be displayed. When you give the parameters for the operation, a button corresponding to your service appears. You can check whether your service provides the desired output by using this option. For more information, see the steps from Step 5 to 7 and Try It Tool#Customizing Try It options.

The Try It page of a service can also be invoked by appending the service's endpoint reference with the string "tryit?". For example, http://<your-machine-ip>:<server port>/services/<service name>?tryit

Invoking Try It through a JAX-WS application’s dashboard

The Try It tool can also be invoked from the dashboard of a selected JAX-WS application. JAX-WS application dashboard is part of the application management functionality provided by the following feature:

Name : WSO2 Carbon - Service Management Feature
Identifier : org.wso2.carbon.service.mgt.feature.group

The above feature is bundled by default in all service-hosting WSO2 middleware products. If the product you are using does not have this feature installed, you can do so by following the instructions given in section Feature Management.

Follow the instructions below to try out the WSDL of a selected application.

1. Log on to the product's management console.
2. On the Main menu, under Applications, click List.
   The Running Applications page appears, which lists out all the currently active applications.
3. Click on the respective JAX-WS application.
   The application dashboard will appear.
4. Click the Try this service link.
   When you click the Try this service link, you will be directed to the page where the operations available for your service will be displayed. When you give the parameters for the operation, a button corresponding to
your service appears. You can check whether your service provides the desired output by using this option. For more information, see the steps from Step 5 to 7 and Try It Tool#Customizing Try It options.

**Customizing Try It options**

**Choosing endpoints**
You can change the endpoint for the service, if there are multiple endpoints. You can also specify a customized endpoint.

1. Click the "**Using Endpoint - ....**" link.
2. Select a new endpoint from the list.
3. Specify the address of the new endpoint.

![Using Endpoint - echoHttpSoap12Endpoint](image)

**Viewing the service information**
To view the information about the service click on the Service Information link.

![Service Information](image)

**Prioritizing Operations**
Use the small yellow icon with the plus (+) sign to prioritize operation.

![Priority Operations](image)

The chosen operation is added to the Priority Operations section.
This will be useful when you have a lot of operations in the Try It page. In such a scenario, you can add the operations you want into this section and switch among them easily.

**Generate Java Code for WSDL**

The "WSDL2Java" tool is used to generate Java code to provide or consume a Web service for a given WSDL. You can use this tool to develop Web services for contract first development. The tool provides many options to customize your code. The functionality is provided by the following feature:

<table>
<thead>
<tr>
<th>Name</th>
<th>org.wso2.carbon.wsdl.tools.feature.group</th>
</tr>
</thead>
</table>

If the above feature is not included in your product by default, you can install it using the instructions given in section Feature Management.

Follow the instructions below to use the "WSDL2Java" tool.

1. Log on to the product's management console and select "Tools -> WSDL2Java."

2. The "WSDL2Java" page appears. Fill in the fields according to your requirement. A url or path to a WSDL file is mandatory to generate the corresponding Java code. For example,

![Upload WSDL file](image)

4. Click "Generate." The Java code will be saved in a location selected by you as a zip file.

**Invoking the Tool from a Service's Dashboard**

Java code can also be generated for the WSDL file of a selected service, through its dashboard. Service dashboard is part of the service management functionality provided by the following feature:

**Name**: WSO2 Carbon - Service Management Feature

**Identifier**: org.wso2.carbon.service.mgt.feature.group

The above feature is bundled by default in all service-hosting WSO2 middleware products. If the product you are using doesn't have this feature installed, you can do so by following the instructions given in section Feature Management.

Follow the instructions below to invoke the WSDL2Java Tool.

1. Log on to the product's management console and select "List -> Web Services."

![Home](image)

2. The "Deployed Services" page appears which lists out all the currently active services. Select a service and click on it to view its dashboard. For example,
3. The service's dashboard appears. In the "Client Operations" panel, you can see the link "Generate Client."

4. Click on the "Generate Client" link to open the "WSDL2Java" page. For more instructions, refer to the steps from Step2 above.

**Generate WSDL for Java Code**

The *Java2WSDL* tool allows you to generate the WSDL for Java code that is already written to provide a Web service. This tool helps you to develop Web services in the code first approach, where the development starts with the code and thereafter you derive the WSDL from the source. The functionality is provided by the following feature:

**Name** : WSO2 Carbon - WSDL Tools Feature

**Identifier** : org.wso2.carbon.wsdltools.feature.group

If the above feature is not included in your product by default, you can install it using the instructions given in section Feature Management.

Follow the instructions below to use the *Java2WSDL* tool.

1. Log on to the product's management console and select "Tools -> Java2WSDL."
2. In the "Add Resources" panel, click "Browse" to locate the file you want to add. For example,

Tip

Only .jar files can be added as resources. If you want to add more than one file, click the plus sign next to the "Browse" button.

3. Enter values as required and click "Generate."

Note

Each field is described in the "Java2WSDL" user interface itself. The "class name" of the service implementation is a mandatory field.
Validating a WSDL

The WSDL Validator tool can be used to validate a WSDL document. You can upload your WSDL or you can provide a WSDL URL. The tool validates your WSDL and prints the result in the Validation Result section. The WSDL validator functionality is provided by the following feature:

Name : WSO2 Carbon - WSDL Tools Feature
Identifier : org.wso2.carbon.wsdl.tools.feature.group

If the above feature is not included in your product by default, you can install it using the instructions given in section Feature Management.

Follow the instructions below to use the WSDL Validator tool.

1. Log on to the product's management console and select "Tools -> WSDL Validator ."

2. The "WSDL Validator" page appears with the following options.

**Uploading a WSDL**

1. You can upload the WSDL document to be validated using the following user interface. Click the "Browse" button to upload a document and click on the "Validate From File" button to validate. The validation results will be shown below the panel. For example,

**Providing a WSDL URL**

1. You can provide the URL of your WSDL document to be validated using the following user interface. Click on the Validate From URL button. Validation result will be shown below the panel.
JavaScript Scraping Assistant

The JavaScript Scraping Assistant tool provides a menu-driven user interface to create a scraper file, which can extract the contents of a HTML page as an XML document. This functionality is provided by the following feature in the WSO2 feature repository.

Name : WSO2 Carbon - Javascript Web Scraping Feature
Identifier : org.wso2.carbon.jsscraper.feature.group

The scraping assistant tool is bundled by default in the WSO2 Application Server and WSO2 Mashup Server products. If it is not included in your product distribution, you can add it by installing the above feature using the instructions given in section Feature Management.

Follow the instructions below to invoke the tool.

1. Log on to the product’s Management Console and select “Scraping Assistant” in the “Tools” menu.

2. The "Scraping Assistant" window opens with an empty config tag.
3. The tool's menus can be used to write the XML configuration according to your requirements. For example,

3.1 select the “Add HTTP request” menu item.

It inserts a line to retrieve a page available at a given URL as follows:

```xml
<http url="url-to-fetch" method="post"/>
```

Replace the "url-to-fetch" section with the URL of the page you want to fetch.

3.2 Highlight the line of code inserted before and select the 'Convert HTML to XML' menu item. It converts the HTML to XML by including the code within the following tags:
3.3 Highlight the existing code segment and select the 'Convert to Variable' menu item. It includes the code within the following tags:

```
<var-def name="variable_1"></var-def>
```

3.4 Optionally, use a variable name that has some semantic value, to get a completed configuration as shown in the example below.

```
Scraping Assistant

<config>
  <var-def name="mashupSite">
    <html-to-xml outputtype="pretty">
      <http url="http://wso2.org/projects/mashup" method="post"/>
    </html-to-xml>
  </var-def>
</config>
```

3.5 This example scraper configuration can be used in script as shown in the example below:

```
function getString() {
  var config = <config>
    <var-def name="mashupSite">
      <html-to-xml outputtype="pretty">
        <http url="http://wso2.org/projects/mashup" method="post"/>
      </html-to-xml>
    </var-def>
  </config>
  var scraper = new Scraper(config);
  var result = scraper.response;
  return result;
}
```

3.6 The code segment above will fetch all content from the URL. You can now modify your configuration to filter out the information you don't need from this URL, or use logic within your script itself to extract the bits you need.

**Scraper Host Object**

The Scraper object we created before allows data to be extracted from HTML pages and presented in XML format. It provides a bridge to data sources that don't have XML or Web service representations at present. The scraping component wraps WebHarvest: http://web-harvest.sourceforge.net/index.php.

There are a few caveats when using the screen scraping language from within the Scraper object and within E4X,
as listed below:

- The result of the scrape must be saved in a variable. The contents of the variable appear as a property on the Scraper object.

```javascript
var config = <config>
  <var-def name='response'>
    <html-to-xml>
      <http method='get'
        url='http://ww2.wso2.org/~builder/'/>
    </html-to-xml>
  </var-def>
</config>;
```

- Currently, the result comes back as a string. When the result represents XML, you have to parse it into XML and also ensure that you remove the XML declaration. The XML constructor does not parse documents, but only node lists, and rejects the declaration as an illegal processing instruction:

```javascript
var scraper = new Scraper(config);
var result = scraper.response;
// strip off the XML declaration and parse as XML.
var resultXML = new XML(result.substring(result.indexOf('?>') + 2));
return resultXML;
```

- The WebHarvest language `<template>` instruction allows variables to be referenced, using the notation `${variable-name}`. The curly brackets conflict with the use of XML literals in E4X, where they cause evaluation of the enclosed data. To escape the curly brackets in E4X (so they will be interpreted by WebHarvest), use the character entity references `&#{7B};` and `&#{7D};` for `{` and `}` respectively.

### JavaScript Stub Generator

The easiest way to access a Web Service hosted externally is through a stub. The "JavaScript Stub Generator" is a productivity tool enabling client code generation according to a given WSDL. Its functionality is provided by the following feature in the WSO2 feature repository.

**Name**: WSO2 Carbon - Javascript Stub Generator Feature  
**Identifier**: org.wso2.carbon.jsstubgenerator.feature.group

The stub generation functionality is bundled by default in the WSO2 Application Server and WSO2 Mashup Server products. If it is not included in your product distribution, you can add it by installing the above feature using the instructions given in section Feature Management.

Follow the instructions below to invoke the tool.

1. Log on to the product's Management Console and select "JavaScript Stub Generator" in the "Tools" menu.
2. The "JavaScript Stub Generator" window appears.

The required WSDL can be uploaded as a file or can be given as a URL, where the stubs can be generated according to the E4X specification or according to DOM.

- E4X Stub: The E4X stub is to be used to access a service from a service.
- DOM Stub: The DOM stub is to be used in browser environments.

3. Once a WSDL file or WSDL file URL is given, click the "Generate from File" or "Generate from URL" button associated with the relevant upload option. You can see the generated stub in the text area. For example,
Shown below is how this script is imported to a Mashup.

Create a new Mashup, say 'Currency' and save it. You will find a Currency.resources directory created in your personal scripts directory: <PRODUCT_HOME>/repository/deployment/server/jsservices/admin. Create a new text file in the Currency.resources directory for the stub and paste the generated stub in it. Let's assume it 'convertor.stub.js'.

Edit the mashup and include an import statement to bring in your stub as follows:

```javascript
system.include("convertor.stub.js");
```

Tools of the Carbon Platform

The WSO2 Carbon platform has tools by default to:

- Capturing the state of the system in error situations.
- Changing user passwords in the Carbon database.
- Encrypting and decrypting simple texts.
- Viewing and resending messages.
- Securing the plain text password in carbon configuration files.

Capturing the State of the System in Error Situations

Carbondump is a tool for collecting all the necessary data from a running Carbon instance at the time of an error. The carboneump generates a zip archive with the collected data, which helps the WSO2 support team to analyze your system and determine the problem which caused the error. Therefore, it is recommended that you run this tool as soon as an error occurs in the Carbon instance.
When using the tool, you have to provide the process ID (pid) of the Carbon instance and the `<PRODUCT_HOME>` which is where your unzipped Carbon distribution files reside. The command takes the following format:

```
sh carbondump.sh [-carbonHome path] [-pid of the carbon instance]
```

For example,

```
In Linux: sh carbondump.sh -carbonHome /home/user/wso2carbon-3.0.0/ -pid 5151
In Windows: carbondump.bat -carbonHome c:\wso2carbon-3.0.0\ -pid 5151
```

The tool captures the following information about the system:

- Operating system information
  - OS (kernel) version
  - Installed modules lists and their information
  - List of running tasks in the system
- Memory information of the Java process
  - Java heap memory dump
  - Histogram of the heap
  - Objects waiting for finalization
  - Java heap summary. GC algo used, etc.
  - Statistics on permgen space of Java heap
- Information about the running Carbon instance
  - Product name and version
  - Carbon framework version (This includes the patched version)
  - `<PRODUCT_HOME>, <JAVA_HOME>`
  - configuration files
  - log files
  - H2 database files
- Thread dump
- Checksum values of all the files found in the `$CARBON_HOME`

### Changing User Passwords in the Carbon Database

The content on this page is under review.

If an admin user forgets his password, he cannot retrieve it using the Management Console. In such a scenario, has to change the password by running the `chpasswd` script located in `<PRODUCT_HOME>/bin` folder on the machine that hosts the Carbon server.

Before executing this script, the Carbon instance should be shut down.

In order to change a user's password, you need to provide the following information:

- **The Carbon database URL**: By default, WSO2 Carbon is shipped with H2 database so the default URL is
jdbc:h2:repository/database/WSO2CARBON_DB;DB_CLOSE_ON_EXIT=FALSE. This URL may change if a different database was specified during the installation. Then the URL will be in the form of jdbc:h2:<PRODUCT_HOME>/repository/database/WSO2CARBON_DB.

- **The Database driver class**: For the default H2 database, the driver will be automatically picked up by the system. If a different database is used, the driver class needs to be specified.

- **The database's username and password**: Again for the default H2 database, the default username and password will be used by the system. However, if a different username/password is used, you are required to reset the admin password.

- **The username and new password of the user whose password is to be changed**: If you do not provide these as command line arguments, you will be prompted for it during execution.

The command line options available for chpasswd is as follows:

<table>
<thead>
<tr>
<th>Command Line Option</th>
<th>Description</th>
<th>Mandatory?</th>
</tr>
</thead>
<tbody>
<tr>
<td>--db-url</td>
<td>The database URL</td>
<td>Yes</td>
</tr>
<tr>
<td>--db-driver</td>
<td>The database driver class</td>
<td>No</td>
</tr>
<tr>
<td>--db-username</td>
<td>The username for the database</td>
<td>No</td>
</tr>
<tr>
<td>--db-password</td>
<td>The password for the database</td>
<td>No</td>
</tr>
<tr>
<td>--username</td>
<td>The username of the user whose password is to be changed.</td>
<td>No</td>
</tr>
<tr>
<td>--new-password</td>
<td>The new password of the user whose password is to be changed.</td>
<td>No</td>
</tr>
</tbody>
</table>

For example:

```
chpasswd --db-url "jdbc:h2:CARBON_HOME/database/WSO2CARBON_DB"
```

On Windows: `chpasswd.bat --db-url "jdbc:h2:C:\Documents and Settings\user\wso2carbon\repository\database\WSO2CARBON_DB"

The following message is displayed if the password is updated successfully:

```
Password of user [username] updated successfully
```

If the database path includes directory names with spaces, the whole URL needs to be included within quotations.

**Encrypting and Decrypting Simple Texts**
The cipher tool encrypts and decrypts simple texts. You can launch the tool by running the ciphertool.sh and ciphertool.bat scripts available at `<PRODUCT_HOME>/bin` directory. The arguments accepted by this tool are as follows:

- `keystore` - If keys are in a store, its location
- `storepass` - Password to access the keyStore
- `keypass` - To get private key
- `alias` - Alias to identify key owner
- `storetype` - Type of the keystore. Default is JKS
- `keyfile` - If key is in a file
- `opmode` - Encrypt or decrypt. Default is encrypt
- `algorithm` - Encrypt or decrypt algorithm. Default is RSA
- `source` - Either cipher or plain text as an in-lined form
- `outencode` - Currently base64 and used for encoding the result
- `inencode` - Currently base64 and used to decode input
- `trusted` - Is KeyStore a trusted store? If this argument is provided, consider as a trusted store
- `passphrase` - if a simple symmetric encryption using a pass phrase shall be used

For example,

```
ciphertool.bat -source testpass -keystore resources/security/client-truststore.jks -storepass wso2carbon -alias wo2carbon -outencode base64 -trusted
```

View and Resend Messages

Users can view and monitor the messages passed along in a TCP-based conversation, using the TCPMon utility in the WSO2 Carbon base platform. TCPMon is a simple and easy-to-use tool, particularly useful when developing Web services. It is based on a swing UI and works on almost all platforms that support Java. You can find the script file used to run this tool by the name tcpmon.sh or tcpmon.bat in the `<PRODUCT_HOME>/bin` folder. TCPMon needs JRE 1.4 or higher to run and has no dependencies on third-party libraries.

Usage Patterns of the Tool

**As an Explicit Intermediate**

The most common usage of TCPMon is as an intermediary. It is called explicit since the client has to send the messages to the intermediary rather than to the original endpoint in order to monitor the messages. The following figure explains this concept.

![Diagram](image)

In order to start TCPMon in this configuration, you have to provide the target host name and port as well as the listening port on the Admin tab. For example,
Click 'Add' to open a new tab that displays the messages.
At this point, the requester should point to the listener port of the TCPMon instead of the original endpoint. For example, assume that we need to monitor all the messages that are sent to and from www.apache.org.

**Step 1:**

Add a listener with the host 'www.apache.org' and port 80. Set the listener to port 8080, which is any unused port in the local machine.

**Step 2:**

Point the browser to localhost:8080 instead of www.apache.org.
When the exchange of messages starts, it can be seen on the relevant tab.

The options at the bottom of the screen can be used to have the messages in XML format (useful in debugging Web services), save and resend the messages and also to switch the layout of the message windows.

As a Request Sender for Web Services

TCPMon can also be used as a request sender for Web services. The request SOAP message can be pasted on the send screen and sent directly to the server.
As a Proxy

TCPMon can act as a proxy. To start it in proxy mode, select the Proxy option. When acting as a proxy, TCPMon only needs the listener port to be configured.

Advanced Settings

TCPMon can simulate a slow connection, in which case the delay and the bytes to be dropped can be configured. This is useful when testing Web services.
Lazy Loading of Tenants and Artifacts

Lazy loading is a design pattern used specifically in cloud deployments to prolong the initialisation of an object or artifact until it is requested by a tenant or an internal process. Lazy loading of tenants is a feature that is built into all WSO2 products, which ensures that in an environment with multiple tenants, all tenants are not loaded at the time the server starts. Instead, they are loaded only when a request is made to a particular tenant. If a tenant is not utilised for a certain period of time, it will be unloaded from memory.

Lazy loading of deployment artifacts

Lazy loading of tenants is a default feature available in any WSO2 product instance. Additionally, the Ghost Deployment configuration in the `carbon.xml` file allows users to enable lazy loading for deployment artifacts as shown below. When this feature is enabled, the services, applications and other artifacts are first loaded in Ghost form. The actual artifact is deployed only when a request for the artifact is made. Also, if an artifact has not been utilized for a certain period of time, it will be unloaded from memory. Note that this setting is not enabled by default because the Ghost Deployer works only with the HTTP/S transports. That is, if other transports are used, we do not have to enable the Ghost Deployer.

```xml
<GhostDeployment>
  <Enabled>false</Enabled>
</GhostDeployment>
```

When lazy loading of artifacts is enabled for PaaS deployments, lazy loading applies both for tenants as well as a tenant’s artifacts. As a result, for a tenant in a cloud environment, lazy loading is applicable on both levels. Therefore, the associated performance improvement and resource utilization efficiencies are optimal.

Lazy loading implementation

Out of four popular variants (Lazy Initialization, Virtual Proxy, Value Holder, Ghost) of the lazy loading design pattern, the "Value Holder" and "Ghost" variants are adopted and used in the lazy loading implementation in WSO2 Carbon Framework.

- Use of Value Holder
- Use of GhostDeployer and GhostDispatcher/GhostWebappDeployerValve

Use of Value Holder

A value holder is an object, usually (but not necessarily) with a method by the name `getValue`, which the clients invoke to obtain a reference to the real object corresponding to a parameter passed in the method. The ValueHolder variant is used at the event of loading a tenant. The called method is as follow:

```java
TenantAxisUtils.getTenantConfigurationContext(String tenantDomain)
```

The `getTenantConfigurationContext` method is the equivalent of the `getValue` method in the ValueHolder variant. It takes a `tenantDomain` as a reference and returns the `ConfigurationContext`, if the tenant is already loaded. If the tenant domain does not have an Axis2 `ConfigurationContext`, the method will return a newly created Axis2 ConfigurationContext. In both the latter mentioned methods, when loading an entire Axis
Configuration, all artifacts belonging to that particular tenant are also loaded. A more efficient practice is to lazy load those artifacts, by loading them only when it is requested by the system or a tenant. This prevents processing overhead and unnecessary memory usage. This is where the Ghost variant of the lazy loading pattern is used.

**Use of GhostDeployer and GhostDispatcher/GhostWebappDeployerValve**

When the server starts, you can see all the previously deployed services and web applications listed in its management console. However, these artifacts are only the Ghost instances that maintain references to the actual instances. *GhostDeployer* that handles lazy loading of artifacts, holds a map of deployers for all artifact types.

When a new artifact is deployed, the *GhostDeployer* creates a Ghost Axis2 Service containing a special parameter (Ghost Parameter) to distinguish the service as a Ghost service, and then registers that service into the AxisConfig. When a user requests for a particular service artifact, the *GhostDispatcher*, which is an Axis2 dispatching handler, checks the Ghost parameter added to the service to determine if it is the actual artifact or the ghost form. If it is the latter, the actual deployer is called to load the actual service, the rest of the metadata from the registry, service policies and other information regarding the artifact.

*GhostWebappDeployerValve*, on the other hand, is used with lazy loading for web applications. It is a Tomcat valve that intercepts the incoming http requests, and does the same work as the *GhostDispatcher* in Axis2. It intercepts the incoming web app request, checks for the Ghost parameter in the webapp and loads the actual web application if it is found to be in ghost form.

**Product Startup Options**

Given below are the options that are available when starting a WSO2 product. The product startup scripts are stored in the `<PRODUCT_HOME>/bin/` directory. When you execute the startup script, you can specify the relevant startup option as shown below.

```sh
sh wso2server.sh -<startup option>
```

See the topic on running a product for detailed information on how to start a WSO2 server and access the management console.

Listed below are some general options that can be used for starting the server.

<table>
<thead>
<tr>
<th>Startup Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-start</td>
<td>Starts the Carbon server using &quot;nohup&quot; in the background. This option is not available for Windows.</td>
</tr>
<tr>
<td>-stop</td>
<td>Stops the Carbon server process. This option is not available for Windows.</td>
</tr>
<tr>
<td>-restart</td>
<td>Restarts the Carbon server process. This option is not available for windows.</td>
</tr>
<tr>
<td>-cleanRegistry</td>
<td>Cleans the registry space. <strong>Caution:</strong> All registry data will be lost.</td>
</tr>
<tr>
<td>-debug &lt;port&gt;</td>
<td>Starts the server in remote debugging mode. The remote debugging port should be specified.</td>
</tr>
<tr>
<td>-version</td>
<td>Shows the version of the product that you are running.</td>
</tr>
</tbody>
</table>
Listed below are some system properties that can be used when starting the server.

<table>
<thead>
<tr>
<th>Startup Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-DosgiConsole=[port]</td>
<td>Starts the Carbon server with the Equinox OSGi console. If the optional 'port' parameter is provided, a telnet port will be opened.</td>
</tr>
<tr>
<td>-DosgiDebugOptions=[options-file]</td>
<td>Starts the Carbon server with OSGi debugging enabled. Debug options are loaded from the <code>&lt;PRODUCT_HOME&gt;/repository/cont/etc/osgi-debug.options</code>.</td>
</tr>
<tr>
<td>-Dsetup</td>
<td>Cleans the registry and other configurations, recreates DB, re-populates the configuration and starts the server. <strong>Note:</strong> It is not recommended to use this option in a production environment. Instead, you can manually run the DB scripts directly in the database.</td>
</tr>
<tr>
<td>-DserverRoles=&lt;roles&gt;</td>
<td>A comma separated list of roles used in deploying Carbon applications.</td>
</tr>
<tr>
<td>-DworkerNode</td>
<td>This property starts the server as a worker node.</td>
</tr>
<tr>
<td>-Dprofile=&lt;profileName&gt;</td>
<td>Starts the server with the specified profile, e.g., worker profile.</td>
</tr>
<tr>
<td>-Dtenant.idle.time=&lt;time&gt;</td>
<td>If a tenant is idle for the specified time, the tenant will be unloaded. The default tenant idle time is 30 minutes. This is required in clustered setups, which has master and worker nodes.</td>
</tr>
</tbody>
</table>
FAQ

- How do you track web sessions?
- Under what instances are web sessions terminated?
- How to configure GZip encoding in AS?
- How can I get the \(<AS\_HOME>/tmp\> directory cleared?
- How to protect my product server from security attacks caused by weak ciphers such as the Logjam attack (Man-in-the-Middle attack)?

**How do you track web sessions?**

This can be done via the webapp management UI as it show all the active sessions.

**Under what instances are web sessions terminated?**

Sessions are terminated when the session is inactive for the specified session expiry time. Sessions can also be explicitly terminated using the management console. Please note that the UI based session termination is not compatible in a clustered setup. The sessions will be terminated only on the node from which the UI was loaded.

**How to configure GZip encoding in AS?**

GZip can be configured using the \(\text{AS\_HOME/repo}\text{ratory/conf/\ tomcat/catalina-server.xml}\) file. This file is equivalent to the server.xml file in Apache Tomcat. The GZip can be configured for each connector in the catalina server. The important attributes of the catalina server that should be configured are compression, compressionMinSize, compressableMimeType, and noCompressionUserAgents. Go to Configuring catalina-server.xml for more details of these attributes.

**How can I get the \(<AS\_HOME>/tmp\> directory cleared?**

Based on the HouseKeeping task for carbon.xml, if the <AutoStart> property is set to true, all files that are more than 30 minutes old will be cleared every 10 minutes. If you are running AS on Linux, you can also write a cron job to handle the clearing of contents within the <AS\_HOME>/tmp directory.

**How to protect my product server from security attacks caused by weak ciphers such as the Logjam attack (Man-in-the-Middle attack)?**

You can disable weak ciphers as described in the section on disabling weak ciphers.
## Getting Support

In addition to this documentation, there are several ways to get help as you work on WSO2 products.

<table>
<thead>
<tr>
<th><img src="image.png" alt="Image" /></th>
<th><strong>Explore learning resources</strong>: For tutorials, articles, whitepapers, webinars, and other learning resources, look in the <strong>Resources</strong> menu on the WSO2 website. For training materials, click <strong>WSO2 Training</strong> on the <strong>Support &amp; Training</strong> menu. In products that have a visual user interface, click the Help link in the top right-hand corner to get help with your current task.</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image.png" alt="Image" /></td>
<td><strong>Try our support options</strong>: WSO2 offers a variety of development and production support programs, ranging from web-based support during normal business hours to premium 24x7 phone support. For support information, see <a href="http://wso2.com/support/">http://wso2.com/support/</a>.</td>
</tr>
<tr>
<td><img src="image.png" alt="Image" /></td>
<td><strong>Ask questions in the user forums</strong> at <a href="http://stackoverflow.com">http://stackoverflow.com</a>. Ensure that you tag your question with appropriate keywords such as <strong>WSO2</strong> and the product name so that our team can easily find your questions and provide answers. If you can't find an answer on the user forum, you can email the WSO2 development team directly using the relevant mailing lists described at <a href="http://wso2.org/mail">http://wso2.org/mail</a>.</td>
</tr>
<tr>
<td><img src="image.png" alt="Image" /></td>
<td><strong>Report issues</strong>, submit enhancement requests, track and comment on issues using our public bug-tracking system, and contribute samples, patches, and tips &amp; tricks (see the WSO2 Contributor License Agreement).</td>
</tr>
</tbody>
</table>
Glossary

Component | Default Endpoint | Endpoint | Embedded Registry Instance | Event Sources | J2EE | Keystore | Mediator | Port offset | Proxy Services | Sequence | Servlet | Servlet container | SOAP | synapse-config Directory | Tasks | WS-Addressing | WSDL | WSDL Endpoint | WSO2 AS Management Console

Component

Components in the Carbon platform add functionality to all WSO2 Carbon-based products. For example, the statistics component enables users to monitor system and service level statistics. A component in the Carbon platform is made up of one or more OSGi bundles, which is the modularization unit in OSGi similar to a JAR file in Java. For example, the statistics component contains two bundles: one is the back-end bundle that collects, summarizes, and stores statistics, and the other is the front-end bundle, which presents the data to the user through a user-friendly interface. This component-based architecture of the WSO2 Carbon platform gives developers flexibility to build efficient and lean products that best suit their unique business needs simply by adding and removing components.

Default Endpoint

A default endpoint is an endpoint defined for adding Quality of Service (QoS) and other configurations to the endpoint, which is resolved from the To address of the message context.

Endpoint

An endpoint is a specific destination for a message. It may be specified as an Address endpoint, WSDL endpoint, a Failover group, a Loadbalance group, and more. Endpoints can be added, edited, and deleted.

Embedded Registry Instance

The Embedded Registry Instance makes use of the embedded AS database. This is a H2 database and the data files are stored by default in the directory named AS_HOME/repository/database. If you run the AS in the embedded registry mode, you should be careful not to alter manually any files stored in this directory as that might lead to database corruption or data loss.

Event Sources

An Event Source is used to define the event source and the subscription manager configuration used in the eventing implementation of WSO2 AS. An Event Source provides a service URL that can be used to send subscriptions and events (events that are not required to go through a mediation flow).

J2EE

Abbreviation for "Java 2 Platform, Enterprise Edition". Now known as "Java Platform, Enterprise Edition" or "Java EE"

Keystore

The keystore used to encrypt administrator passwords and other confidential information in Carbon is configured in AS_HOME/repository/conf/carbon.xml file. This keystore configuration can be found under the <security> element of the carbon.xml file.
Mediator

A mediator is a full powered processing unit. It is a component that performs some mediation action on a message during the process flow. Thus, a mediator gets full access to a message at the point where it is defined to gain control and could inspect the message, modify it or take an external action depending on some attributes or values of the current message.

Port offset

The port offset feature allows you to run multiple WSO2 products, multiple instances of a WSO2 product, or multiple WSO2 product clusters on the same server or virtual machine (VM). The port offset defines the number by which all ports defined in the runtime such as the HTTP/S ports will be offset. For example, if the HTTP port is defined as 9763 and the portOffset is 1, the effective HTTP port will be 9764. Therefore, for each additional WSO2 product, instance, or cluster you add to a server, set the port offset to a unique value (the default is 0).

Port offset can be passed to the server during startup as follows:

```
./wso2server.sh -DportOffset=3
```

Alternatively, you can set it in the Ports section of `<PRODUCT_HOME>/repository/conf/carbon.xml` file as follows:

```
<Offset>3</Offset>
```

Proxy Services

A Proxy Service is a virtual service hosted in the AS runtime. The proxy service mediates any accepted requests and forwards them to a specified endpoint, most of the time to an actual Web Service. The responses coming back from the target endpoint are mediated back to the client which sent the original service request. Proxy services often make references to sequences, endpoints and local entries. A proxy service can enforce various QoS requirements on a request and can be exposed over a user specified transport. Therefore they are often used to expose existing Web Services over a different transport or a QoS configuration. For example, a proxy service can be used to expose an HTTP service over JMS or it can be used to expose an unsecured service in a secured manner using WS-Security standards. Unlike sequences and endpoints, which can be stored and loaded from the registry, proxy services cannot be loaded from the registry. However, a proxy service can make references to sequences and endpoints stored in the registry.

Sequence

A Sequence element is used to define a Sequence of mediators that can be invoked later as a Sequence. If the configuration defines a Sequence named "main", then it is considered as the main mediation Sequence of the AS. If such a Sequence is not defined locally and a registry has been specified, the registry is looked up for a key named "main" to find the main mediator Sequence. If the user does not define a main or fault Sequence, AS will create default Sequences.

Servlet

A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

Servlet container

The Servlet Container (such as Apache Tomcat) is an application that monitors a port on a given IP address. Servlets generate responses to HTTP requests. To do so, the container loads your servlet (if it is not in memory
already) and calls the methods defined in the interface. This is the foundation of servlet and JSP architecture.

---

**SOAP**

An XML-based, extensible message envelope format, with "bindings" to underlying protocols. The primary protocols are HTTP and HTTPS, although bindings for others, including SMTP and XMPP, have been written.

---

**synapse-config Directory**

The synapse-config Directory houses the Mediation Configuration (Synapse Configuration) of WSO2 AS and contains several files and subdirectories. Usually the following set of child directories is available in the synapse-config Directory: endpoints, event-sources, local-entries, priority-executors, proxy-services, sequences, tasks.

---

**Tasks**

A Task is a custom Java class that implements the `org.apache.synapse.startup` Task interface that defines a single `public void execute()` method. Such a Task can be scheduled and managed via the AS. The scheduling information for a Task can be specified in the `cron` format or a `simple` format by the user. A Task may also be specified as a one-time Task where required and can be used to trigger a callout or inject a message into the AS.

---

**WS-Addressing**

WS-Addressing was originally designed as the way to convey connection state or context in SOAP messages, emulating the contextualization mechanism provided by HTTP headers in conjunction with stored cookies. This makes it possible to contextualize message exchanges in a similar way, independently of the transport used.

---

**WSDL**

An XML format allows service interfaces to be described along with the details of their bindings to specific protocols. It is typically used to generate server and client code and for configuration.

---

**WSDL Endpoint**

WSDL Endpoint is an endpoint definition based on a specified WSDL document. The WSDL document can be specified either as a URI or as an inlined definition within the configuration.

---

**WSO2 AS Management Console**

WSO2 AS Management Console is a Web based control panel powered by JSP and AJAX which enables system administrators to interact with a running AS instance without touching any underlying configuration files. The Management Console allows the users to command and control proxy services, sequences, transports, local entries, registry, modules, endpoints and much more.