# Table of Contents

1. WSO2 Message Broker Documentation .................................................. 5
   1.1 About WSO2 Message Broker ......................................................... 5
       1.1.1 Architecture ................................................................. 6
       1.1.2 Features ........................................................................ 9
       1.1.3 About this Release ........................................................... 9
   1.2 Quick Start Guide ........................................................................... 10
   1.3 Tutorials ...................................................................................... 14
       1.3.1 Working with Queues ......................................................... 15
           1.3.1.1 Managing Queues ....................................................... 16
           1.3.1.2 Using the Dead Letter Channel .................................. 21
       1.3.2 Working with Topics ............................................................ 22
           1.3.2.1 Managing Topics and Sub Topics ................................. 24
       1.3.3 Integrating with Other Products ............................................ 27
           1.3.3.1 Integrating WSO2 ESB .................................................. 27
           1.3.3.2 Integrating WSO2 CEP ............................................... 34
               1.3.3.2.1 Using WSO2 MB as A JMS Broker for WSO2 CEP Server 34
               1.3.3.2.2 Configure and Run Stock Quote Analyzer .................. 38
           1.3.3.3 Integrating WSO2 DSS ................................................ 51
           1.3.3.4 Integrating with Application Servers ................................ 55
               1.3.3.4.1 Integrating with Oracle Weblogic Server ............... 55
               1.3.3.4.2 Integrating with WSO2 AS ................................. 63
       1.3.4 Managing Subscriptions ........................................................... 73
   1.4 Samples ......................................................................................... 76
       1.4.1 Setting up the MB Samples ...................................................... 77
       1.4.2 JMS Client Samples ............................................................... 78
           1.4.2.1 Sending and Receiving Messages Using Queues ............. 79
           1.4.2.2 Sending and Receiving Messages Using Topics .............. 84
           1.4.2.3 Receiving Messages with JMS Message Listener ............ 89
           1.4.2.4 Receiving Messages from an MB Cluster ..................... 96
           1.4.2.5 JMS Selectors ............................................................. 101
           1.4.2.6 Sending and Receiving Messages with TTL ................. 109
       1.4.3 Creating a Durable Topic Subscription ....................................... 116
       1.4.4 Creating Hierarchical Topic Subscriptions ................................. 122
       1.4.5 CSharp Client Samples ............................................................ 129
           1.4.5.1 Publishing and Receiving Messages from a Queue .......... 129
           1.4.5.2 Publishing and Receiving Messages from a Topic .......... 134
       1.4.6 Using MQTT Transport .......................................................... 139
           1.4.6.1 Simple MQTT Client ................................................... 139
           1.4.6.2 MQTT Chat .............................................................. 146
           1.4.6.3 MQTT IoT ............................................................... 159
           1.4.6.4 MQTT Retain ............................................................ 178
       1.4.7 Using Transactional Sessions .................................................... 186
       1.4.8 Setting Message Expiration ...................................................... 197
   1.5 Deep Dive ...................................................................................... 205
       1.5.1 Installation Guide ................................................................. 206
           1.5.1.1 Downloading the Product .......................................... 206
           1.5.1.2 Installing the Product ................................................. 206

Copyright © WSO2 Inc. 2005-2014
1.5.1.2.1 Installation Prerequisites ................................................. 206
1.5.1.2.2 Installing on Linux .................................................. 209
1.5.1.2.3 Installing on Solaris ............................................... 211
1.5.1.2.4 Installing on Windows ............................................ 213
1.5.1.2.5 Installing as a Linux Service .................................. 216
1.5.1.3 Running the Product .................................................. 218
1.5.2 Product Administration .................................................... 221
  1.5.2.1 Upgrading from a Previous Release ............................... 225
  1.5.2.2 Performance Tuning Guide ...................................... 227
    1.5.2.2.1 Clustering Performance ...................................... 228
    1.5.2.2.2 Database Performance ....................................... 230
    1.5.2.2.3 Message Publisher and Consumer Performance .......... 230
    1.5.2.2.4 Tuning Flow Control ......................................... 235
  1.5.2.3 Troubleshooting WSO2 Message Broker .......................... 237
  1.5.2.4 Configuring Transports for WSO2 MB ............................ 243
  1.5.2.5 Changing the Default MB Database ............................. 247
  1.5.2.6 User Permissions for WSO2 MB .................................. 252
  1.5.2.7 Configuring Message Compression .............................. 260
  1.5.2.8 Enabling SSL Support ............................................ 261
  1.5.2.9 Port Offset Configuration ....................................... 263
  1.5.2.10 Handling Failover ................................................ 264
  1.5.2.11 Configuring the Message Delivery Strategy .................. 267
  1.5.2.12 Clustered Deployment ........................................... 269
    1.5.2.12.1 Configuring the DBMS for Storage ....................... 281
1.5.3 Analytics ................................................................. 284
  1.5.3.1 Working with WSO2 Carbon Metrics ............................. 284
    1.5.3.1.1 Enabling Metrics and Storage Types ....................... 284
    1.5.3.1.2 Configuring Metrics Properties ............................ 290
  1.5.3.2 Monitoring with Carbon Metrics ............................... 292
    1.5.3.2.1 Using JVM Metrics ........................................... 292
    1.5.3.2.2 Using Messaging Metrics ................................... 295
1.5.4 JMS Subscribers and Publishers ...................................... 299
  1.5.4.1 Creating Durable Topic Subscriptions ......................... 299
  1.5.4.2 Subscribing to Hierarchical Topics ............................ 303
  1.5.4.3 Redelivering Messages to Subscriber ........................ 305
  1.5.4.4 Setting the Routing Key for Messages ........................ 306
  1.5.4.5 Acknowledging Message Delivery ............................... 307
  1.5.4.6 Working with JMS Messages ..................................... 308
    1.5.4.6.1 JMS Message Types and Header Fields .................... 308
    1.5.4.6.2 Using Message Selectors .................................... 309
  1.5.4.7 Setting the Connection URL ..................................... 313
  1.5.4.8 Configuring Message Expiration ............................... 316
  1.5.4.9 Handling Distributed Transactions ............................. 318
1.5.5 Configuration Files ..................................................... 321
  1.5.5.1 Configuring broker.xml .......................................... 321
  1.5.5.2 Configuring qpid-config.xml .................................... 351
  1.5.5.3 Configuring qpid-virtualhosts.xml ............................. 359
  1.5.5.4 Configuring axis2.xml ........................................... 362
  1.5.5.5 Configuring carbon.xml ......................................... 373
  1.5.5.6 Configuring catalina-server.xml ............................... 379
1.5.5.7 Configuring master-datasources.xml ......................................................... 390
1.5.5.8 Configuring registry.xml ................................................................. 393
1.5.5.9 Configuring user-mgt.xml .............................................................. 395
1.5.5.10 Configuring hazelcast.properties ............................................... 397

2. Setting up with Remote Derby ................................................................. 398
3. Setting up with Embedded Derby .......................................................... 402
Welcome to the WSO2 Message Broker (MB) 3.2.0 documentation! Message Broker is a lightweight, easy-to-use, open source, distributed message brokering server available under the Apache Software License v2.0. Developed based on the award-winning WSO2 Carbon platform, all features of Message Broker are available as pluggable, configurable Carbon components with point-and-click installation simplicity.

### Get started with WSO2 Message Broker

If you are new to using WSO2 Message Broker, follow the steps given below to get started:

<table>
<thead>
<tr>
<th>Get familiar with WSO2 MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understand the basics of WSO2 MB and its architecture.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quick Start Guide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get started with WSO2 MB.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tutorials</th>
</tr>
</thead>
<tbody>
<tr>
<td>Go through the real-life use cases of WSO2 MB.</td>
</tr>
</tbody>
</table>

### Deep dive into WSO2 Message Broker

<table>
<thead>
<tr>
<th>Installation Guide</th>
<th>Product Administration</th>
<th>Analytics</th>
<th>JMS Publishers and Subscribers</th>
</tr>
</thead>
</table>

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.
About WSO2 Message Broker

WSO2 Message Broker (MB) is a fast, lightweight, user-friendly, open source distributed message brokering system, delivered under the Apache license 2.0. WSO2 MB allows system administrators and developers to easily configure JMS queues and topics, to be used in message routing, message stores and message processors. WSO2 MB is compliant with Advanced Message Queueing Protocol Version 0-91, Message Queueing and Telemetry Transport Version 3.1 and Java Message Service Specification version 1.1.

WSO2 MB is developed on top of the award-winning WSO2 Carbon platform, an OSGi-based framework that provides seamless modularity to your SOA via componentization. It also contains a range of optional components (add-ons) that can be installed to customize the behavior of the MB. Further, any existing features of the MB, which are not required for your environment can be easily removed using the underlying provisioning framework of Carbon. In brief, WSO2 MB is future-proof and can be fully customized and tailored to meet your exact SOA needs.

The WSO2 Message Broker is an on-going project. It undergoes continuous improvements and enhancements with each new release, which addresses new business challenges and customer expectations. WSO2 invites users, developers and enthusiasts to get involved or get the assistance from our development teams at many different levels through online forums, mailing lists and support options. We are committed to ensure you a fulfilling user experience at any level of involvement with the WSO2 Message Broker.

The following topics provide more information about Message Broker:

- Architecture
- Features
- About this Release

Architecture

The underlying messaging framework of the WSO2 Message Broker is powered by Andes, one of the distributed message brokering systems compatible with the leading Advanced Message Queueing Protocol (AMQP)(0-91)). In addition, WSO2 Message Broker is also compatible with the Message Queueing and Telemetry Transport (MQTT) 3.1.

This section describes the architecture at the following levels.

- Component Architecture
- Slot-based architecture

Component Architecture

The following diagram depicts the component-based architecture of the WSO2 Message Broker.
Transports
The WSO2 Message Broker supports the AMQP transport and the MQTT transport.

Inbound Disruptor
All incoming events are inserted into the inbound disruptor ring where many handlers work in parallel. Data is stored and deleted through the persistence layer.

Outbound Disruptor
This reads messages from the database concurrently, and passes them to the transport for delivery.

Slot Management
A queue can be divided into several slots. A slot is a chunk of messages which can be owned by one node at a time. The slot manager generates distributes slots between slot delivery workers based on the requirement. A publisher returns the last message ID to the slot manager after every 1000 messages.

Data Stores
These are used to save any information related to messaging such as AMQP exchanges, message content etc. Information relating to topic context and authentication are saved in the registry. Other information is saved in the message store.

Andes Kernel
This contains the WSO2 specific implementation, which is used when handling different messaging protocols.

Slot-based architecture
The slot-based message delivery system is designed to enable global queues to be shared among the nodes in an MB cluster.

A queue is mapped to a row in a message store and it can be divided into many slots. A slot is a chunk of messages in the row that can be owned by one MB node at a given time.

The slot manager communicates with both publishers and subscribers and acts as the coordinator for distributing slots among the nodes. A slot assignment map is maintained to track the slots that are assigned to nodes at a given
The activities of a slot manager can be illustrated as follows.

Communicating with publishers

Each publisher belongs to a queue/topic and its messages are published in the row of the message store mapped to this queue. A publisher returns its last message ID to the slot manager node in the MB cluster after every 1000 messages or after a timeout. The slot manager updates the Hazelcast Distributed Map with these IDs and uses them to generate slots when it receives requests for slots from slot delivery worker nodes.

The number of messages after which a publisher returns the last message ID to the slot manager can be changed by modifying the `windowSize` parameter in the `<MB_HOME>/repository/conf/broker.xml` file. See Configuring broker.xml for further information about this parameter.

Assigning slots

When a client subscribes to a queue/topic, a slot delivery worker requests for a slot. Then the slot manager first looks for returned slots (i.e. slots that were previously assigned to another subscriber node which has left the cluster) and assigns one of them if any are available. If there are no such slots, an empty slot (i.e. a slot with no messages currently published in it) is generated and assigned to the slot delivery worker.

Deleting a slot

Once a subscriber node has sent all the messages it has read from a slot and received acknowledgements, it sends a request to the slot manager to delete the slot. The slot manager removes the relevant entry from the slot assignment map once it receives the request to delete.

Reassigning slot when the last subscriber leaves

If a subscriber node to which a slot is assigned leaves the cluster, the slot manager reassigns the slot to another subscriber node to free the slot pool.

Delivering messages to subscribers

A slot delivery worker reads all the messages published in a slot assigned to it and passes them to the message flusher, which delivers them to subscribers who have subscribed to the relevant queue/topic in a Round Robin
manner. Messages that were not delivered due to a delivery failure and messages rejected by the subscriber are buffered queue-wise in the Message Flusher.

**Features**

WSO2 Message Broker brings messaging and eventing capabilities to your SOA framework. It is based on JMS (Java Message Service), and its features are basically implementations of the JMS specification, which means that any JMS client can communicate with WSO2 Message Broker. It can be used as a standalone message broker or a distributed message brokering system.

WSO2 Message Broker 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>
<tbody>
<tr>
<td>JMS Message Queuing</td>
<td>Publish and receive messages with:</td>
</tr>
<tr>
<td></td>
<td>• FIFO order</td>
</tr>
<tr>
<td></td>
<td>• Message durability</td>
</tr>
<tr>
<td></td>
<td>• Ability to handle large messages</td>
</tr>
<tr>
<td></td>
<td>• Ability to specify maximum delivery attempts</td>
</tr>
<tr>
<td></td>
<td>• Handling Failover available with clustered setup</td>
</tr>
<tr>
<td>JMS Message Pub/Sub</td>
<td>Use topics to subscribe to messages.</td>
</tr>
<tr>
<td></td>
<td>• Handling hierarchical topic structures</td>
</tr>
<tr>
<td></td>
<td>• Handling durable topic subscriptions</td>
</tr>
<tr>
<td>Featured Graphical Console</td>
<td>WSO2 Message Broker includes a set of management services and a graphical user interface to configure, manage, and monitor the running message broker, allowing you to:</td>
</tr>
<tr>
<td></td>
<td>• Create and delete message queues</td>
</tr>
<tr>
<td></td>
<td>• View message statistics</td>
</tr>
<tr>
<td></td>
<td>• Browse the registry</td>
</tr>
<tr>
<td></td>
<td>• View nodes that have joined the message broker cluster</td>
</tr>
<tr>
<td>Server Management via JMX</td>
<td>Manage queues, topics, and other server objects via JMX.</td>
</tr>
<tr>
<td>XA Transactions</td>
<td>Possibility to handle distributed transaction.</td>
</tr>
</tbody>
</table>

**About this Release**

WSO2 Message Broker version 3.2.0 is the successor of version 3.1.0. It is based on the Carbon 4.4.16.

WSO2 MB 3.2.0 includes the following enhancements and new features:

- Possibility to delay message redelivery to a subscriber.
- Possibility to enable per-message acknowledge of messages.
- Possibility to set the routing key for messages.
- Possibility to configure how message expiration is handled by the broker.
- Possibility to enable OAuth-based authentication and authorization for the MQTT transport.
- Possibility to reroute all messages from the Dead Letter Channel simultaneously.
- Handling distributed transactions.
- RDBMS-based clustering.

**Fixed and known issues**

- WSO2 Message Broker 3.2.0 - Fixed Issues
- WSO2 Message Broker 3.2.0 - Known Issues

**Compatible versions**

WSO2 MB 3.2.0 is based on WSO2 Carbon 4.4.16 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 MB 3.2.0, see Installation Prerequisites. For more information on the products in each Carbon platform release, see the Release Matrix.
Quick Start Guide

The purpose of this guide is to get you started on the main features of WSO2 Message Broker (MB) as quickly as possible. We will create queues and topics on WSO2 MB and then see how JMS clients can send messages to these queues/topics and receive messages from them.

- Key concepts
- Before you begin
  - Download and start WSO2 MB
  - Download sample JMS clients
- Working with Queues
  - Step 1: Create a queue in WSO2 MB
  - Step 2: Publish messages to the queue
  - Step 3: Subscribe to the queue and consume messages

Key concepts

Explained below are some of the key concepts in message brokering.

Queues

Queues in WSO2 MB are essentially message stores that can store messages from external JMS clients (publishers) and maintain them in an intermediate state until they are consumed by other JMS clients (subscribers).

Topics

Topics in WSO2 MB are used for realtime message brokering. You can maintain a hierarchy of topics or subjects in WSO2 MB for which JMS clients (subscribers) can subscribe. Other JMS clients (publishers) can then publish messages to these topics, which will immediately be consumed by the subscribers.

Subscribers and Publishers

A JMS client that is configured to send messages to a queue or topic is known as a publisher. A JMS client that is configured to receive messages published to a particular queue or topic is known as a subscriber.

Management Console

The management console of WSO2 MB is the user interface of the product, which can be used to conveniently create topics and queues and to manage subscriptions. External clients can then connect to WSO2 MB for publishing messages and consuming already published messages.

Before you begin

Install WSO2 MB and download JMS clients as explained below.

- Download and start WSO2 MB
- Download sample JMS clients

Download and start WSO2 MB

Follow the steps given below to download and start a WSO2 MB instance.
Download WSO2 Message Broker from here. Extract the ZIP file to a location on your computer. This location will be referred to as <MB_HOME> from hereon.

To start the product: Open a terminal, navigate to the <MB_HOME>/bin directory where all the startup scripts are stored and run the startup script:
- On Windows: wso2server.bat
- On Linux: sh wso2server.sh

The URL of the management console will be printed in the terminal as follows:

```
INFO {org.wso2.carbon.ui.internal.CarbonUIServiceComponent} - Mgt
Console URL : https://10.100.5.65:9443/carbon/
```

Download sample JMS clients

We will use sample JMS clients in this quick start guide to simulate how messages are published to WSO2 MB and how the published messages are consumed. First, you need to download and set up these clients as follows:

1. Click this link to download the QuickStartClients.zip file, which contains the following sample clients.
   - JMS queue publisher
   - JMS queue subscriber
   - JMS topic publisher
   - JMS topic subscriber
   - JMS durable topic subscriber
2. Extract this file to a location on your computer. We will call this <JMS_CLIENTS_HOME> from hereon.

You will need Apache Ant to execute the above JMS clients. Download Apache Ant using homebrew.

**Working with Queues**

Let's try out a simple scenario of brokering messages using a queue in WSO2 MB.

**Step 1: Create a queue in WSO2 MB**

We will now create a queue in WSO2 MB, using the product's management console. Follow the instructions given below.

1. Log in to the management console using the credentials of the default system administrator: admin/admin. You will now be logging into the super tenant domain.
2. In the Main tab, click Queues -> Add. The Add Queue screen will open.
3. Enter 'testQueue' as the name in the Queue Name field as shown below.
3. Click **Add Queue** and the new queue will be listed in the **Queue List** page as shown below.

You can access the **Queue List** page by clicking **Queues -> List** in the **Main** tab of the navigator.

**Step 2: Publish messages to the queue**

You can now use the sample JMS Queue Publisher client to publish messages to this queue. Follow the steps given below.

1. Open a command prompt and navigate to the `<JMS_CLIENTS_HOME>` directory on your computer.
2. Execute the following command, which will publish 10 messages to the queue named 'testQueue':

   ```
   ant queuePublisher
   ```

3. Go to the management console.
4. In the **Main** tab, click **Queues -> List**. You can see that 10 messages have been published to the queue as shown below.

   5. Click **Browse** to see details of the messages received.

   **Step 3: Subscribe to the queue and consume messages**

   Now, you need a JMS client to subscribe to the 'testQueue' queue and consume the messages. The sample JMS Queue Subscriber JMS client will be used for this purpose. Follow the steps given below.
1. Open a command prompt and navigate to the `<JMS_CLIENTS_HOME>` directory on your computer.
2. Execute the following command, to create a subscription to the 'testQueue' queue.

   ```
   ant queueSubscriber
   ```

3. The 10 messages in the queue have been successfully consumed by the subscriber. You can verify this from the terminal log:

   ```
   queueSubscriber:
   [java] log4j:WARN No appenders could be found for logger (org.wso2.andes.client.AMQDDestination).
   [java] log4j:WARN Please initialize the log4j system properly.
   [java] Waiting for messages
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received text message: Test Message Content
   [java] Received message count = 10
   ```

4. Go to the management console and click **Queues -> List** in the **Main** tab. You can see that the 10 messages that were stored in the queue are no longer there.
Tutorials

WSO2 MB is used for sending and receiving messages through queues and topics. See the following sections for information on how to use the main functions of WSO2 MB:

- Working with Queues
  - Managing Queues
  - Using the Dead Letter Channel
- Working with Topics
  - Managing Topics and Sub Topics
- Integrating with Other Products
  - Integrating WSO2 ESB
  - Integrating WSO2 CEP
  - Integrating WSO2 DSS
  - Integrating with Application Servers
- Managing Subscriptions

Working with Queues

Queues provide the facility for users to store messages in an intermediate location sent by an external party, and access them on-demand. WSO2 Message Broker uses a Message Queue to facilitate this feature, and all operations are carried out on that message queue. The queuing feature is similar to an inbox where anyone can send messages to and only the owner can consume the messages. Queues enable users to publish messages and receive them in the order that they are sent. These queues are natively persistent, which indicates that even after shutting down the server or a sudden crash happens, the messages still remain in the queue ready to be delivered.

The message queue architecture is depicted in the figure below.

The Queue management capability in WSO2 Message Broker is provided by the following feature in the WSO2
feature repository.

**Name**: WSO2 Carbon - Feature - Andes

**Identifier**: org.wso2.carbon.andes.feature.group

This feature can be removed or added to a different distribution if it is not already bundled with it using the instructions given in section, Server Provisioning.

**Managing Queues**

You can easily manage queues in WSO2 MB using the management console. This includes adding new queues, removing queues, viewing their contents and more. You can also specify advanced queue configurations to improve performance and functionality when you set up an instance of WSO2 Message Broker. For more information, see advanced configurations.

If you have multiple instances of Message Broker running, queues are distributed, and each queue is owned by a node in the cluster. If the node fails, and the client application is configured to use failover, it will be able to send and consume messages from the cluster without any failures or message loss. Clustering operations and synchronizing messages across the cluster are fully transparent to the user.

Following are the tasks you can perform to manage queues in the management console:

- **Prerequisites**

  - Creating new queues
    - Adding queues from the management console
    - Add queues through client subscriptions
    - Adding queues at server start
  - Viewing the message count
  - Viewing the queue contents
  - Publishing messages to queues
  - Purging queues
  - Deleting queues

**Prerequisites**

The possibility to work with queues in WSO2 MB are strictly secured by role-based permissions. Therefore, a user needs to be linked to a role with the relevant permissions in order to start managing queues in WSO2 MB. See the following links for details on how roles and permissions are set up for a user.

- Find out about the permissions required for working with queues.
- Creating roles and assigning permissions.
- Creating users and assigning roles.

**Creating new queues**

A new queue can be created in WSO2 MB using multiple methods. See the following sections for details.

- Adding queues from the management console
- Add queues through client subscriptions
- Adding queues at server start

**Adding queues from the management console**

The following steps describe how to add a queue using the management console. External applications can also add
d queues programmatically.

1. On the Main tab in the management console, click Queues -> Add.
2. Enter a name for the topic in the Queue Name field.

Note the following for queue names:

- The queue name cannot contain any of the following symbols: ~!@#$%^&*()+={}|<>"' or spaces.
- Do not include "/" literals or prefix the name with "tmp_".
- The "/" symbol (TENANT_SEPARATOR) can only be used to separate the tenant domain from the queue name when creating queues from external clients.

3. Now you need to grant permissions for users in your system to publish messages to the queue and consume messages from the queue. You will be granting these permissions to users based on the user’s role. As shown below, all the user roles in your tenant domain will be listed. You can select the relevant check box against each role:

   - If the Consume check box is selected, the users linked to the role has permission to consume messages from the queue.
   - If the Publish check box is selected, the users linked to the role has permission to publish messages to the queue.

Add Queue

If you want to create new user roles as queue publishers and queue consumers, see the documentation on creating users and roles.

By default, publishing and consuming permissions for queues (including JMS client queues) are granted only to the role to which the queue creator belongs. If required, the admin user can change these permissions (Configure > Users and Roles > Roles).

4. Click Add Queue.

5. If the queue is added successfully, a message appears. Click OK to view the Queue List where the newly added queue is listed:

```addddqueueislisted:
Add queues through client subscriptions

When a JMS client subscribes to WSO2 MB, the queue that is specified in the subscription will be automatically created in the broker (if it doesn't already exist). However, note that the subscriber client should connect to WSO2 MB with user credentials that have permission to create new queues in the broker. See the prerequisites section above for information on granting permissions to users.

For example consider the jndi.properties file given below, which specifies the connection details of a subscriber client connecting to WSO2 MB. As shown here, the client will be connecting to the queue named 'NewQueue' in WSO2 MB with the user 'sam'. If a queue by the name of 'NewQueue' does not exist in the WSO2 MB at the time of establishing this connection, and if the user 'sam' has permission to create new queues in WSO2 MB, a new queue will be created automatically.

```
# register some connection factories
# connectionfactory.[jndiname] = [ConnectionURL]
connectionfactory.QueueConnectionFactory =
  amqp://sam:sam@clientID/carbon?brokerlist='tcp://localhost:5677'

# register some queues in JNDI using the form
# queue.[jndiName] = [physicalName]
queue.NewQueue=NewQueue
```

Find out more on creating subscriptions to WSO2 MB.

Adding queues at server start

WSO2 Message Broker allows you to create queues automatically when the server is started. This can be done by updating the qpid-virtualhosts.xml file with the information about the queue you want created.

Follow the steps given below.

1. Open the qpid-virtualhosts.xml file from the `<MB_HOME>/repository/conf/advanced` directory.
2. Add a queue by adding a new code block under `<queues>`. See the following example, where a durable queue by the name of 'my-simple-queue' is added.

```
<queue>
  <name>my-simple-queue</name>
  <my-simple-queue>
    <exchange>amq.direct</exchange>
    <durable>true</durable>
  </my-simple-queue>
</queue>
```

Given below are the descriptions of the elements used above.
2. **<queue>** - This is the container element that holds the queue definition.
3. **<name>** - This element is used to give a name for the queue. In this example it is 'my-simple-queue'.
4. **<my-simple-queue>** - Once you define the queue name using the <name> element, you must open another element using the queue name as illustrated in this example.
5. **<exchange>** - Queues created in the broker will communicate with external clients through an exchange definition. This element should be set to amq.direct.
6. **<durable>** - This is also a mandatory element, which specifies that the queue should be durable.

3. You must now add an exchange configuration corresponding to the amq.direct exchange that you used when defining the queue.

```xml
<exchange>
    <type>direct</type>
    <name>amq.direct</name>
    <durable>true</durable>
</exchange>
```

4. Save the information.
5. Start the server and log in to the management console.
6. You can now view and manage the 'my-simple-queue' queue from the management console. See the section below on viewing queues for more information.

### Adding tenant-specific queues

Queues created by external client applications are listed the same way in the table when you are in the default "Super Tenant" mode. When a queue is created from a tenant other than the super tenant, the queue's name will be prefixed with the tenant domain name. For example, if the tenant domain is "a.com" and the queue name is "myQueue", the queue name will be displayed as "a.com/myQueue".

<table>
<thead>
<tr>
<th>Name</th>
<th>Message Count</th>
<th>View</th>
<th>Operations</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.com/myQueue</td>
<td>0</td>
<td>Browse</td>
<td>Publish Messages</td>
<td>Delete</td>
</tr>
</tbody>
</table>

When sending or consuming messages to/from this queue using an external client application, or when creating a queue from an external JMS client, the queue name must include the domain prefix (such as "a.com/myQueue" in the previous example). If the external client application needs the tenant username, which is in TENANT_USER@TENANT_DOMAIN format, to authenticate, replace the '@' symbol in the tenant-specific username with the '!' symbol.

**Viewing the message count**

To view the message count:

1. Access the **Queue List** by clicking **Queues -> List** on the **Main** tab.
2. The list of queues and the number of messages received will be indicated in the **Message Count** field.

This field displays the approximate number of persistent messages that have not been delivered to the client. However, note that if the number of messages flowing through the queue is high, the message count may be a higher value than the number of messages that remained in the store when the page was loaded.

**Viewing the queue contents**
To view the contents of a queue:

1. Access the **Queue List** by clicking **Queues -> List** on the **Main** tab.
2. Click **Browse** for a particular queue and the contents will be listed. See the following example:

   ![Queue List Example](image)

3. The table shows information about each message in the queue, including the content type and timestamp of the message. To view the message's body content, click **more...** in the **Message Summary** column.

   ![Message Summary](image)

By default, WSO2 Message Broker displays up to 100 messages in the queue content window. If you need to display more messages, you can increase the `<messageBrowsePageSize>` value in the `<MB_HOME>/repository/conf/broker.xml` file.

### Publishing messages to queues

You can simulate how messages are published to a queue by sending a sample text using the management console. Follow the steps given below.

1. Access the **Queue List** by clicking **Queues -> List** on the **Main** tab.
2. Click **Publish Messages** for the relevant queue.
3. In the **Send Message** page that opens, enter values for the following properties:
   - **Correlation ID**: The correlation ID for the message, which can be an application-specific 'String' value set by the user. This is not a mandatory field. This property is used for linking one message with another. According to the JMS specification, it typically links a reply message with its requesting message.
   - **JMS Type**: A 'String' value to define the message type if required. This is not a mandatory field.
   - **Number of Messages (Required)**: The number of sample messages to be sent to the queue. This field is mandatory.
   - **Message Body**: The text content of the sample message. This is not a mandatory field as the body will be containing a text called 'Type Message Here..' in case user doesn't add a message body.
4. Click **Send Message**. You can then verify that the messages were delivered by checking the **queue contents**.

### Purging queues

To purge queues from the server:

1. Access the **Queue List** by clicking **Queues -> List** on the **Main** tab.
2. Click **Purge Messages** for the relevant queue. This will remove all the messages from the queue and set the message count to zero.

### Deleting queues

To delete queues from the server:

1. Access the **Queue List** by clicking **Queues -> List** on the **Main** tab.
2. Click **Delete** for the relevant queue. This will remove all the persistent messages from the server.

**Using the Dead Letter Channel**

The Dead Letter Channel (DLC) is a sub-set of a queue, specifically designed to persist messages that are typically marked for deletion, providing you with a choice on whether to delete, retrieve or reroute the messages from the DLC. A DLC queue is created when the first subscription is made for a specific domain (superuser or tenant). By default, the message broker retries a defined number of times when a subscriber does not acknowledge a message. Once the retry count has been breached, the message is removed from the queue and placed in the DLC. The default number of retries is set to 10, however, this value is configurable. See the section on configuring the message retry count for more information.

---

External publishers/subscribers are not permitted to publish/subscribe to the DLC queue.

---

- Browsing through the DLC Queue
- Handling multi-tenancy
- Configuring the message retry count

**Browsing through the DLC Queue**

All messages that have breached the retry count will be dumped into the DLC queue. The content of the queue can be viewed by following the steps below.

1. Log in to the Management Console.
2. Click **Dead Letter Channel** - > **Browse** in the **Main** tab.
3. The DLC browser is displayed in the right side of the pane, as shown below. The message count specifies the number of messages in the DLC queue.

<table>
<thead>
<tr>
<th>Name</th>
<th>Message Count</th>
<th>View</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dead Letter Channel</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

4. Click **Browse** to view details of each message stored in the DLC. The **Destination** column shows the name of the original queue.
5. You can choose to delete, restore or reroute one or more messages:
   - **Delete**: Permanently delete the message from MB.
   - **Restore**: Re-insert the message to the queue from which the message originated.
   - **ReRoute**: Allows you to select a destination of choice instead of restoring the message to the original queue. When you click **ReRoute**, a pop-up menu appears, as shown below:

   Select the queue name to which the message should be re-routed and click **OK**. The message is then moved to the relevant queue.

   - **ReRoute All**: This option will be available if you have the relevant configuration enabled for the product. The `<allowReRouteAllInDLC>` property in the `broker.xml` file should be set to `true` before the broker is started. You can click this option to reroute all messages in the DLC to a destination.
Handling multitenancy

To ensure that messages can be distinguished per tenant, a separate DLC is created for every new tenant in the system. The signature of the DLC queue is similar to the example shown below.

\{(tenant_name)/DeadLetterChannel\}

This ensures that only the messages exchanged through queues in a specific tenant domain are inserted to the DLC queue.

Configuring the message retry count

The message retry count can be configured as follows:

1. Open the `broker.xml` file located in the `<MB_HOME>/repository/conf` folder.
2. Locate the `maximumRedeliveryAttempts` attribute and change the value accordingly. See the following example.

\<![maximumRedeliveryAttempts>15</maximumRedeliveryAttempts>\]

Working with Topics

In messaging systems, message routing is done on the basis of a concept called Topics: First, we need to have topics created in the message broker (WSO2 MB). A JMS client can then be configured to publish messages to this topic and other JMS clients can be configured as subscribers of this topic. When the publisher sends a message to
the topic, it will be dispatched to all the JMS clients that are subscribed to that topic as depicted in the following diagram:

For example, consider that we have a topic called 'SportsNews' created in the message broker. We can now have users (JMS clients) publishing messages related to sports news to this topic. Users (JMS clients) that are interested in sports news can subscribe to this topic and receive the messages that are published.

Here are some of the key qualities of message routing using WSO2 MB:

- Messages published to WSO2 MB are stored in the DB, which means that all messages published to WSO2 MB are inherently persistent.
- When a message is published to a topic, a copy of the topic is dispatched to all the subscribers.
- A subscription to a topic can be non-durable or durable depending on how the subscriber client implements the subscription. If the subscription is non-durable, the subscriber client will only receive the messages that are published while the subscriber client is active. If the subscription is durable, the subscriber client has the ability to recover the messages that were published to the topic while the subscriber was inactive.
- Topics in WSO2 MB are secured by role-based permissions. This means that a JMS client can publish messages or consume messages from a topic, only using the credentials of an authorized user.

The capability of managing topics and subscriptions in WSO2 Message Broker is provided by the following features in the WSO2 feature repository:

| Name: WSO2 Carbon - Event Feature |
| Identifier: org.wso2.carbon.event.feature.group |
| Name: WSO2 Carbon - Feature - Andes |
| Identifier: org.wso2.carbon.andes.feature.group |
Managing Topics and Sub Topics

Topics and sub topics can be easily added to WSO2 MB using the management console. The topics and sub topics you create will be arranged into a tree navigator, which you can expand and collapse to get an overview of the parent-child topic structure. You can also view details of a particular topic or sub topic by simply clicking on the topic in the tree navigator. Once a topic is added to MB, users will be able to publish information to the topic and subscribe to the information published.

Follow the instructions given below.

- **Prerequisites**
- Adding topics from the management console
- Adding sub topics from the management console
- Viewing topic details
- Deleting a topic

**Prerequisites**

The possibility to work with Topics in WSO2 MB are strictly secured by role-based permissions. Therefore, a user needs to be linked to a role with the relevant permissions in order to start managing Topics in WSO2 MB. See the following links for details on how roles and permissions are set up for a user.

- Find out about the permissions required for working with Topics.
- Creating roles and assigning permissions.
- Creating users and assigning roles.

**Adding topics from the management console**

To create a new topic using the management console:

1. Log in to the management console.
2. In the **Main** tab, click **Topics** -> **Add**. The **Add Topic** screen will open.
3. Enter a name for the topic in the **Topic** field.

**Note the following for the Topic name:**

- Topic name can start with any of the following: *, alphanumeric characters (a-z, A-Z or 0-9). These can be used any number of times delimited by dots.
- Topic name can end with the following: *, alphanumeric characters, hash (#).
- The “/” symbol (TENANT_SEPARATOR) can only be used to separate the tenant domain from the topic name when creating topics/subscribing to topics from external clients.
- The colon (" : ") can be used in Topic names if the `<allowStrictNameValidation>` property is set to `false` in the `broker.xml` file (stored in the `<MB_HOME>/repository/conf` folder). This is only applicable for AMQP.

4. Now you need to grant permissions for users in your system to publish and subscribe to the topic you are creating. You will be granting these permissions to users based on the user's role. As shown below, all the user roles in your tenant domain will be listed. You can select the relevant check box against each role:
   - If the **Subscribe** check box is selected, the users linked to the role has permission to subscribe to the topic.
If the **Publish** check box is selected, the users linked to the role has permission to publish information to the topic.

If you want to create new user roles for topic subscription and publishing see the documentation on creating users and roles.

### Add Topic

<table>
<thead>
<tr>
<th>Topic Name</th>
<th>Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SportsNews</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>Subscribe</th>
<th>Publish</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueConsumer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QueuePublisher</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TopicPublisher</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TopicSubscriber</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Internal/everyone</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Click **Add Topic** to create the topic.

6. If the topic is added successfully, a message appears. Click **OK** to view the topic browser where the newly added topic is listed in a tree view:

   ![Topic Browser](image)

   7. Click on the topic to view all the operations available. See the section on viewing topic details for more information.

---

**Adding sub topics from the management console**

You can add a sub topic under an existing topic as shown below.

1. Go to the **Topic Browser** to select the parent topic.
2. Click the topic to see the details.
3. Click **Add Subtopic** to create the sub topic. The information that should be provided when adding a subtopic is the same as when adding a main topic.
4. Once a sub topic is added, it will be displayed in the **Topic Browser** window under the main topic.

![Add Sub Topic](image)

**Viewing topic details**

Once you have added your topics to the **Topics Browser**, you can view detailed information about the topic as shown below.

1. Click on the topic in the **Topic Browser**. The **Details** option will be listed.
2. Click **Details** to open the **Topic Details** screen:
The following table shows descriptions of the parameters on this page:

<table>
<thead>
<tr>
<th>Detail</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permission Details</td>
<td>Permissions related to the topic can be viewed here. Permissions can be changed by clearing or selecting the provided check boxes and clicking <strong>Update Permissions</strong> once done.</td>
</tr>
<tr>
<td>JMS Subscription Details</td>
<td>All the durable and non durable JMS subscriptions are listed here.</td>
</tr>
<tr>
<td>Publish</td>
<td>Publish a sample XML message to a topic. When there is a subscription for the topic, the event sink URL receives the XML message specified here. Click the <strong>Publish</strong> button after placing the XML message in the provided space.</td>
</tr>
</tbody>
</table>

**Deleting a topic**

To delete a topic, the subscription count for that topic and its children should be zero. If not, an error message appears specifying that there are subscriptions for the topic or its children.

**Integrating with Other Products**

You can integrate WSO2 Message Broker with the following WSO2 products:

- Integrating WSO2 ESB
- Integrating WSO2 CEP
- Integrating WSO2 DSS
- Integrating with Application Servers

**Integrating WSO2 ESB**
This section describes how to integrate WSO2 Message Broker with WSO2 Enterprise Service Bus to facilitate message brokering needs of the ESB and to implement store and forward messaging pattern.

The first step is to set up WSO2 MB and WSO2 ESB.

Setting up WSO2 Message Broker

1. Download and install WSO2 MB according to the instructions in Installation Guide.

   It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts. Therefore, to avoid port conflicts, apply a port offset in the <MB_HOME>/repository/conf/carb on.xml file by changing the offset value to 1. For example,

   ```
   <Ports>
     <!-- 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 -->
     <Offset>1</Offset>
   </Ports>
   ```

2. Start the Message Broker by running <MB_HOME>/bin/wso2server.sh (on Linux) or <MB_HOME>/bin/wso2server.bat (on Windows).

Setting up WSO2 ESB

1. Download and install WSO2 ESB according to the instructions in Getting Started. The unzipped ESB distribution folder is referred to as ESB_HOME.

2. Enable the JMS transport of WSO2 ESB to communicate with the Message Broker by editing the <ESB_HOME>/repository/conf/axis2/axis2.xml file. Find a commented <transport receiver> block for MB and uncomment it as shown below.

   ```
   <!--Uncomment this and configure as appropriate for JMS transport support with WSO2 MB 2.x.x -->
   <transportReceiver name="jms" class="org.apache.axis2.transport.jms.JMSListener">
     ...
   </transportReceiver>
   ```

   Also, uncomment the <transport sender> block for JMS in the same file as shown below.

   ```
   <transportSender name="jms" class="org.apache.axis2.transport.jms.JMSSender"/>
   ```
3. Copy the following jar files from the `<MB_HOME>/client-lib` folder to the `<ESB_HOME>/repository/components/lib` folder.
   - `andes-client-3.1.1.jar`
   - `geronimo-jms_1.1_spec-1.1.0.wso2v1.jar`
   - `org.wso2.securevault-1.0.0-wso2v2.jar`

4. Open the `<ESB_HOME>/repository/conf/JNDI.properties` file and make a reference to the running Message Broker as shown below.

   Use `carbon` as the virtualhost. Define a queue called `JMSMS`. Comment out the topic since it is not needed for this scenario. However, in order to avoid getting the `javax.naming.NameNotFoundException` exception during server startup, make a reference to the Message Broker from the `TopicConnectionFactory` as well.

   ```
   # register some connection factories
   # connectionfactory.[jndiname] = [ConnectionURL]
   connectionfactory.QueueConnectionFactory =
   amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
   connectionfactory.TopicConnectionFactory =
   amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'
   # register some queues in JNDI using the form
   # queue.[jndiname] = [physicalName]
   queue.JMSMS=JMSMS
   queue.StockQuotesQueue = StockQuotesQueue
   ```

   The connection factory specifies the URL to be used by a client using WSO2 MB in order to connect to it. 5673 in this example is the port listening for messages on TCP when the AMQP transport is used. This port changes depending on the port offsets done in different scenarios. See Configuring a Client to Access Broker When Port Offset is Change for more information.

   If you want to create a queue that is specific to a particular tenant (e.g., a tenant with `test.com` domain), the following is required.

   - The connection factory entries should have the tenant's credentials instead of the super tenant's credentials.
   - The queue name should have the tenant's domain as a prefix as shown below.
     ```
     test.com/StockQuotesQueue
     ```

   See Managing Tenant-specific Subscriptions for detailed information.

5. Start WSO2 ESB by running `<ESB_HOME>/bin/wso2server.sh` (on Linux) or `<ESB_HOME>/bin/wso2server.bat` (on Windows).
   Now you will have both the Enterprise Service Bus and the Message Broker running.

6. We need some background services to be available for testing purposes. Therefore, run an ANT task to deploy the `SimpleStockQuoteService` in the simple axis2server as follows.

   - Navigate to `<ESB_HOME>/samples/axis2Server/src/SimpleStockQuoteService` and run the ant command to build the sample and deploy background services.
Next, run `<ESB_HOME>/samples/axis2server/axis2Server.sh` (on Linux) to start the Axis2 server.

Enter `http://127.0.0.1:9000/services/SimpleStockQuoteService?wsdl` in the address bar of your browser and press the Enter key to verify that the service is running.

You now have two instances of WSO2 Message Broker and WSO2 ESB configured, up and running. Next, proceed to integrate the Message Broker with ESB, for which there are two implementation options as follows.

- Using JMS Endpoints and JMS Proxy Services
- Using Message Stores and Processors

Integration Using JMS Endpoints and JMS Proxy Services

This section describes how to integrate WSO2 Message Broker with WSO2 ESB as JMS endpoints. A user case is used for this purpose.

**Use case**

Store a message received to a http proxy of the ESB in a JMS queue. Then consume that queue, get the message and send it to the actual endpoint.

**Execution Steps**

1. Create a JMS proxy named `StockQuotesQueue`. It is the same name defined in the `jndi.properties` file above. By creating this proxy, you will be creating a consumer to make a subscription to be used in this user case. The Synapse configuration of the proxy looks as follows. See Adding a Proxy Service for detailed instructions to create a proxy service.

```xml
<proxy xmlns="http://ws.apache.org/ns/synapse" name="StockQuotesQueue"
      transports="jms" statistics="disable" trace="disable"
      startOnLoad="true">
  <target>
    <inSequence>
      <log level="full"/>
      <property name="OUT_ONLY" value="true"/>
      <send>
        <endpoint>
          <address uri="http://localhost:9000/services/SimpleStockQuoteService"/>
        </endpoint>
      </send>
    </inSequence>
  </target>
  <description></description>
</proxy>
```

Once this proxy service is configured, the `StockQuotesQueue` JMS queue will be created in the MB management console (see Browsing Queues for more information).

2. Create the HTTP proxy to send messages to the JMS Queue. Synapse configuration of this HTTP proxy is as follows. Since this is a one-way message, the `OUT_ONLY` property is set to `true`, and the `FORCE_SC_ACCEPTED` property is defined to send a 202 response to the client that invokes this proxy.
The message flow paths are completed. When the StockQuoteProxy proxy service is invoked, it will send the message to the queue. This message will be consumed by the StockQuotesQueue JMS proxy and sent to the actual endpoint.

**Testing the Integration**

Send the following SOAP message to StockQuoteProxy proxy service using the SOAP UI.

```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmlns:ser="http://services.samples" xmlns:xsd="http://services.samples/xsd">
  <soap:Header/>
  <soap:Body>
    <ser:placeOrder>
      <!--Optional:-->
      <ser:order>
        <!--Optional:-->
        <xsd:price>10</xsd:price>
        <!--Optional:-->
        <xsd:quantity>100</xsd:quantity>
        <!--Optional:-->
        <xsd:symbol>IBM</xsd:symbol>
      </ser:order>
    </ser:placeOrder>
  </soap:Body>
</soap:Envelope>
```
The above message will be logged in the ESB console as output.

The log of the SimpleAxis2Server will be as follows.

```
Tue Jan 15 15:31:28 IST 2013 samples.services.SimpleStockQuoteService ::
Accepted order #2 for : 100 stocks of IBM at $ 10.0
```

**Integrate Using Message Stores and Processors**

This section describes how to integrate WSO2 Message Broker with WSO2 ESB using message stores and processors.

**Execution Steps**

Perform the following steps in WSO2 ESB.

1. Create a message store with the following configuration. See Adding a Message Store for further information.

   ```xml
   <messageStore name="JMSMS"
   class="org.wso2.carbon.message.store.persistence.jms.JMSMessageStore"
   xmlns="http://ws.apache.org/ns/synapse">
   <parameter name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFile
   InitialContextFactory</parameter>
   <parameter name="java.naming.provider.url">repository/conf/jndi.properties</parameter>
   <parameter name="store.jms.destination">JMSMS</parameter>
   <parameter name="store.jms.JMSSpecVersion">1.1</parameter>
   <parameter name="store.jms.cache.connection">false</parameter>
   </messageStore>
   ```

2. Define an endpoint to send the message. In this example, the backend just set-up in the previous step is used.

   ```xml
   <endpoint name="SimpleStockQuoteService">
   <address
   uri="http://127.0.0.1:9000/services/SimpleStockQuoteService"/>
   </endpoint>
   ```

3. Define a message forwarding processor with the following configuration. See Adding a Message Processor for further information.
<messageProcessor
class="org.apache.synapse.message.processors.forward.ScheduledMessageForwardingProcessor" name="Processor1" messageStore="JMSMS">
  <parameter name="max.delivery.attempts">4</parameter>
  <parameter name="interval">4000</parameter>
</messageProcessor>

4. Define a proxy service with the following configuration.

```xml
<proxy name="InOnlyProxy" transports="https http" startOnLoad="true" trace="disable">
  <target>
    <inSequence>
      <property name="FORCE_SC_ACCEPTED" value="true" scope="axis2"/>
      <property name="OUT_ONLY" value="true"/>
      <property name="target.endpoint" value="SimpleStockQuoteService"/>
      <log level="full"/>
      <store messageStore="JMSMS"/>
    </inSequence>
  </target>
</proxy>
```

Messages sent to this proxy service are stored in the JMSMS queue in the Message Broker which serves as a message store. If messages are sent while the message processor is disabled, you will notice an increase in the message count of the JMSMS queue in the MB Management Console. See Browsing Queues for more information.

**Note**

The following section describes the In Only service invocation with the Message Forwarding Processor. Follow this article to implement the others.

## Testing the Integration

Using the SOAP UI, send the following SOAP message to the InOnlyProxy proxy service you created.
The above message will be logged in ESB console. SOAP UI will get the 202 Accepted message.

The following message will be logged in the axis2server console.

```
samples.services.SimpleStockQuoteService :: Generating quote for : IBM
```

To adapt to your specific environment, simply replace the following with a suitable name.

```
<endpoint name="SimpleStockQuoteService">
    <address uri="http://test"/>
</endpoint>
```

### Integrating WSO2 CEP

WSO2 Message Broker can be configured as a JMS broker for WSO2 Complex Event Processing Server.

- Using WSO2 MB as A JMS Broker for WSO2 CEP Server
- Configure and Run Stock Quote Analyzer

### Using WSO2 MB as A JMS Broker for WSO2 CEP Server

The following example shows how you can use the WSO2 Message broker as the JMS broker for WSO2 Complex Event Processor.

Before you configure the WSO2 CEP:

1. See Installation Prerequisites for instructions on installing WSO2MB.
2. Copy the following broker-specific jars from `<MB_HOME>/client-lib` to `<CEP_HOME>/repository/components/lib`.
   - andes-client-xx.jar
   - geronimo-j2ee-management_1.1_spec-1.0.1xx.jar
   - log4j-1.2.13.jar
1. Open the `<CEP_HOME>/repository/conf/jndi.properties` file and register a connection factory named `TopicConnectionFactory` by entering the following in the `register some connection factories` section.

   ```
   connectionfactory.TopicConnectionFactory=amqp://admin:admin@clientid/carbon?brokerlist='tcp://localhost:5672'
   ```

2. Register the `BasicStockQuotes` topic by adding the following to the `register some topics in JNDI using the form` section of the same file.

   ```
   topic.BasicStockQuotes = BasicStockQuotes
   ```


4. Start WSO2 CEP using one of the following commands based on your operating system.

   - On Windows: `./wso2server.bat -DportOffset=1 -Dqpid.dest_syntax=BURL`
   - On Linux/Solaris: `./wso2server.sh -DportOffset=1 -Dqpid.dest_syntax=BURL`

   By setting the `portOffset=1` property, the ports used in CEP server are offset by 1 to avoid port conflicts with MB. This is only required if both MB and CEP servers are running on the same node. We are also forcing the usage of Binding URL(BURL) address syntax by setting `qpid.dest_syntax` system property to `BURL`.

5. Click `Input Event Adapters` in the `Configure` tab of the CEP management console to open the `Available Input Adapters` page. Then click `Add Input Event Adapter` to open the `Create a New Input Event Adapter` page. Create a JMS input event adapter to receive messages from WSO2 MB with the following configurations and save.

   - `Event Adapter Name`: MBInputAdapter
   - `Event Adapter Type`: JMS
   - `JNDI Provider URL`: repository/conf/jndi.properties
   - `The JMS connection username`: admin
   - `The JMS connection password`: admin
   - `Connection Factory JNDI Name`: TopicConnectionFactory
   - `Destination Type`: topic
   - `Enable Durable Subscription`: false

   The above configuration would be as follows in the source view:
<?xml version="1.0" encoding="UTF-8"?>
<inputEventAdapter name="MBInputAdapter" statistics="disable"
    trace="disable" type="jms"
    xmlns="http://wso2.org/carbon/eventAdaptermanager">
    <property
        name="java.naming.provider.url">repository/conf/jndi.properties</property>
    <property
        name="transport.jms.SubscriptionDurable">false</property>
    <property
        name="transport.jms.UserName">admin</property>
    <property
        name="java.naming.factory.initial">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</property>
    <property
        name="transport.jms.Password">admin</property>
    <property
        name="transport.jms.ConnectionFactoryJNDIName">TopicConnectionFactory</property>
    <property
        name="transport.jms.DestinationType">topic</property>
</inputEventAdapter>

6. Click **Output Event Adapters** in the **Configure** tab of the CEP management console to open the **Available Output Adapters** page. Then click **Add Output Event Adapter** to open the **Create a New Output Event Adapter** page. Create a JMS input event adapter to receive messages from WSO2 MB with following configurations and save.

- **Event Adapter Name**: MBOutputAdapter
- **Event Adapter Type**: JMS
- **JNDI Initial Context Factory Class**: org.wso2.andes.jndi.PropertiesFileInitialContextFactory
- **JNDI Provider URL**: repository/conf/jndi.properties
- **The JMS connection username**: admin
- **The JMS connection password**: admin
- **Connection Factory JNDI Name**: TopicConnectionFactory
- **Destination Type**: topic
- **Enable Durable Subscription**: false

The above configuration would be as follows in the source view:
7. In the Main tab of the CEP management console, click Event Streams to open the Available Event Streams page. Click the Inflows link of the org.wso2.sample.stock.quote.basic:1.0.0 stream and then click Receive to External Event Stream (via Event Builder) to open the Create a New Event Formatter pop-up screen. Create a new event builder as follows:

- **Event Builder Name**: MBAllStockQuotesBasic
- **Input Event Adapter Name**: MBInputAdapter
- **Topic/Queue Name**: AllStockQuotes
- **Input Mapping Type**: json

Click Advanced to expand the screen and display the JSON Mapping section. Enter two JSON mappings as shown in the table below.

<table>
<thead>
<tr>
<th>JSONPath</th>
<th>Mapped To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.StockQuoteEvent.LastTradeAmount</td>
<td>price double</td>
</tr>
<tr>
<td>$.StockQuoteEvent.StockSymbol</td>
<td>symbol string</td>
</tr>
</tbody>
</table>

8. Open the Available Event Streams page again. Click the Outflows link of the org.wso2.sample.stock.quote.basic:1.0.0 stream and then click Publish to External Event Stream (via Event Formatter) to open the Create a New Event Formatter pop-up screen. Create a new event formatter as follows:

- **Event Formatter Name**: MBBasicStockQuotes
- **Streams**: price double, symbol string
- **Output Event Adapter Name**: MBOutputAdapter
- **Destination**: AllStockQuotes
- **Header**: JMS_PrIORITY:4
- **Output Mapping Type**: map

Click Advanced to expand the screen and display the Map Mapping section. Enter two Map mappings as shown in the table below.

<table>
<thead>
<tr>
<th>JSONPath</th>
<th>Mapped To</th>
</tr>
</thead>
<tbody>
<tr>
<td>$.StockQuoteEvent.LastTradeAmount</td>
<td>price double</td>
</tr>
<tr>
<td>$.StockQuoteEvent.StockSymbol</td>
<td>symbol string</td>
</tr>
</tbody>
</table>
9. Enter the following properties in the `<CEP_HOME>/samples/consumers/jms/src/main/resources/jndi.properties` file.

In the section where initial context factories are entered:

```
java.naming.factory.initial =
org.wso2.andes.jndi.PropertiesFileInitialContextFactory
```

In the `use the following property to configure the default connector` section:

```
connectionfactory.ConnectionFactory=amqp://admin:admin@clientid/carbo
n?brokerlist='tcp://localhost:5672'
```

10. Run the following command in `<CEP_HOME>/samples/consumers/jms` directory to have the customer subscribed to the `BasicStockQuotes` topic.

```
ant topicConsumer -Dtopic=BasicStockQuotes
```

11. Enter the following properties in the `<CEP_HOME>/samples/producers/stock-quote/src/main/resources/jndi.properties` file.

In the section where initial context factories are entered:

```
java.naming.factory.initial =
org.wso2.andes.jndi.PropertiesFileInitialContextFactory
```

In the `use the following property to configure the default connector` section:

```
connectionfactory.ConnectionFactory=amqp://admin:admin@clientid/carbo
n?brokerlist='tcp://localhost:5672'
```

12. Run the `ant` command in the `<CEP_HOME>/samples/producers/stock-quote` directory.

Configure and Run Stock Quote Analyzer

This sample demonstrates how Siddhi engine can be used with JMS event broker to receive, process and publish XML messages.

In this sample CEP will receive stock quote information and fire outputs if the last traded amount vary by 2 percent with regards to the average traded price within past 2 minutes.
Here we will publish events using an JMS client to a JMS topic called AllStockQuotes and fired outputs of the bucket will be send to a JMS topic called FastMovingStockQuotes, which will be received using another JMS client and log in console.

Following is the configuration used in this sample.

```
<cep:bucket xmlns:cep="http://wso2.org/carbon/cep"
            name="XMLStockQuoteAnalyzer">
    <cep:description>
        This bucket analyzes stock quotes and trigger an event if the last traded amount vary by 2 percent with regards to the average traded price within past 2 minutes.
    </cep:description>
    <cep:engineProviderConfiguration engineProvider="SiddhiCEPRuntime">
        <cep:property name="siddhi.persistence.snapshot.time.interval.minutes">0</cep:property>
        <cep:property name="siddhi.enable.distributed.processing">false</cep:property>
    </cep:engineProviderConfiguration>
    <cep:input topic="AllStockQuotes" brokerName="MBJmsBroker">
        <cep:xmlMapping queryEventType="Tuple"
                        stream="allStockQuotesStream">
            <cep:xpathDefinition prefix="quotedata"
                               namespace="http://ws.cdyne.com/>
            <cep:property name="price" xpath="/quotedata:StockQuoteEvent/quotedata:LastTradeAmount" type="java.lang.Double"/>
            <cep:property name="symbol" xpath="/quotedata:StockQuoteEvent/quotedata:StockSymbol" type="java.lang.String"/>
        </cep:xmlMapping>
    </cep:input>
    <cep:query name="StockDetector">
        <cep:expression><![CDATA[
            from allStockQuotesStream#window.time(120000)
            insert into fastMovingStockQuotesStream
            symbol,avg(price) as avgPrice, price
            group by symbol
            having ((price > (avgPrice*1.02)) or ((avgPrice*0.98)>price ));
        ]]]
        <cep:output topic="FastMovingStockQuotes" brokerName="MBJmsBroker">
            <cep:xmlMapping>
                <quotedata:StockQuoteDataEvent
```
xmlns:quotedata="http://ws.cdyne.com/">
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <quotedata:StockSymbol>{symbol}</quotedata:StockSymbol>
  <quotedata:AvgLastTradeAmount>{avgPrice}</quotedata:AvgLastTradeAmount>
  <quotedata:LastTradeAmount>{price}</quotedata:LastTradeAmount>
</quotedata:StockQuoteDataEvent>
</cep:xmlMapping>
Prerequisites

- Apache Ant to build & deploy the Sample & Service, and to run the client. See Installation Prerequisites for instructions on installing Apache Ant.

Steps to configure the sample

The steps are as follows:

1. Install the WSO2 Complex Event Processor. See the Installation Guide for instructions.
2. Now start the WSO2 Complex Event Processor. See Running the Product for instructions.
3. Start WSO2 Message Broker with a port offset one (assuming setup is done on a single machine). See Running WSO2 MB for instructions.
4. Then configure WSO2 Message Broker as the JMS Broker for CEP server as described in Using WSO2 MB as A JMS Broker for WSO2 CEP Server.

   When CEP connects to MB, the wso2server.sh file needs to be updated with the following:

   ```
   system property -Dqpid.dest_syntax=BURL \
   ```

5. Copy the above bucket configuration to <CEP_HOME>/repository/deployment/server/cepbuckets folder. Note that we have used the JMS Broker "MBJmsBroker" created at step 4 in the bucket configuration.
6. Now the bucket will be deployed at the CEP side. Please check the logs to verify.

Steps to run the sample

In order to run the above sample we have to do two things.

1. Publish events to CEP server. We can do this by publishing events to the AllStockQuotes topic.
2. Subscribe for events generated by CEP server according to the query we have defined in bucket configuration above. This can be done in two ways.
   1. Subscribe an JMS topic message subscriber to the topic FastMovingStockQuotes and get the events.
   2. Subscribe a web service client to the topic FastMovingStockQuotes and receive the events using a web service client instead of using a JMS client.

Publishing events to CEP server

We will use a JMS client for this purpose.

Following class will create Initial Context to run the event publisher client. Note that this class is used in event subscriber JMS client as well.

```java
package org.wso2.cep.sample.jms.andes;
import javax.jms.TopicConnectionFactory;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
```
import java.util.Properties;
public class JNDIContext {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "ConnectionFactory";
    private static final String userName = "admin";
    private static final String password = "admin";
    private static String CARBON_CLIENT_ID = "clientid";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5673";
    private InitialContext initContext = null;
    private TopicConnectionFactory topicConnectionFactory = null;
    public static JNDIContext instance = null;
    private JNDIContext() {
        createInitialContext();
        createConnectionFactory();
    }
    public InitialContext getInitContext() {
        return initContext;
    }
    public TopicConnectionFactory getTopicConnectionFactory() {
        return topicConnectionFactory;
    }
    public static JNDIContext getInstance() {
        if (instance == null) {
            instance = new JNDIContext();
        }
        return instance;
    }
    /**
     * Create Connection factory with initial context
     */
    private void createConnectionFactory() {
        try {
            topicConnectionFactory = (TopicConnectionFactory) initContext.lookup("ConnectionFactory");
        } catch (NamingException e) {
            System.out.println("Can not create topic connection factory." + e);
        }
    }
    /**
     * Create Initial Context with given configuration
     */
    private void createInitialContext() {
        try {
            Properties properties = new Properties();
            properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
            properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
System.out.println("TCPConnectionURL: = " +
getTCPConnectionURL(userName, password));
initContext = new InitialContext(properties);
}
catch (NamingException e) {
    System.out.println("Can not create initial context with given
parameters." + e);
}
}

public String getTCPConnectionURL(String username, String password) {
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT).append("'")
        .toString();
package org.wso2.cep.sample.jms.andes.xmlMessage;
import org.apache.axiom.om.OMElement;
import org.apache.axiom.om.impl.builder.StAXOMBuilder;
import org.apache.axiom.om.util.StAXUtils;
import org.wso2.cep.sample.jms.andes.JNDIContext;
import javax.jms.JMSException;
import javax.jms.MessageProducer;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.naming.InitialContext;
import javax.xml.stream.XMLStreamException;
import javax.xml.stream.XMLStreamReader;
import java.io.ByteArrayInputStream;
public class AllStockQuotesPublisher {
    
    private static InitialContext initContext = null;
    private static TopicConnectionFactory topicConnectionFactory = null;
    public static void main(String[] args) throws XMLStreamException {
        String xmlElement1 = "<quotedata:AllStockQuoteStream xmlns:quotedata="http://ws.cdyne.com/">
            <quotedata:StockQuoteEvent>
                <quotedata:StockSymbol>MSFT</quotedata:StockSymbol>
                <quotedata:LastTradeAmount>126.36</quotedata:LastTradeAmount>
                <quotedata:StockChange>0.05</quotedata:StockChange>
                <quotedata:OpenAmount>25.05</quotedata:OpenAmount>
                <quotedata:DayHigh>25.46</quotedata:DayHigh>
                <quotedata:DayLow>25.01</quotedata:DayLow>
                <quotedata:StockVolume>20452658</quotedata:StockVolume>
                <quotedata:PrevCls>25.31</quotedata:PrevCls>
                <quotedata:ChangePercent>0.20</quotedata:ChangePercent>
                <quotedata:FiftyTwoWeekRange>22.73 - 31.58</quotedata:FiftyTwoWeekRange>
                <quotedata:EarnPerShare>2.326</quotedata:EarnPerShare>
                <quotedata:PE>10.88</quotedata:PE>
                <quotedata:CompanyName>Microsoft Corpora</quotedata:CompanyName>
        </quotedata:StockQuoteEvent>
    }
}
String xmlElement2 = "<quotedata:AllStockQuoteStream xmlns:quotedata="http://ws.cdyne.com/">
    <quotedata:StockQuoteEvent>
        <quotedata:StockSymbol>MSFT</quotedata:StockSymbol>
        <quotedata:LastTradeAmount>36.36</quotedata:LastTradeAmount>
        <quotedata:StockChange>0.05</quotedata:StockChange>
        <quotedata:OpenAmount>25.05</quotedata:OpenAmount>
        <quotedata:DayHigh>25.46</quotedata:DayHigh>
        <quotedata:DayLow>25.01</quotedata:DayLow>
        <quotedata:StockVolume>20452658</quotedata:StockVolume>
        <quotedata:PrevCls>25.31</quotedata:PrevCls>
        <quotedata:ChangePercent>0.20</quotedata:ChangePercent>
        <quotedata:FiftyTwoWeekRange>22.73 - 31.58</quotedata:FiftyTwoWeekRange>
        <quotedata:EarnPerShare>2.326</quotedata:EarnPerShare>
        <quotedata:PE>10.88</quotedata:PE>
        <quotedata:CompanyName>Microsoft Corpora</quotedata:CompanyName>
    </quotedata:StockQuoteEvent>
</quotedata:AllStockQuoteStream>
";

String xmlElement3 = "<quotedata:AllStockQuoteStream xmlns:quotedata="http://ws.cdyne.com/">
    <quotedata:StockQuoteEvent>
        <quotedata:StockSymbol>MSFT</quotedata:StockSymbol>
        <quotedata:LastTradeAmount>6.36</quotedata:LastTradeAmount>
        <quotedata:StockChange>0.05</quotedata:StockChange>
        <quotedata:OpenAmount>25.05</quotedata:OpenAmount>
        <quotedata:DayHigh>25.46</quotedata:DayHigh>
        <quotedata:DayLow>25.01</quotedata:DayLow>
        <quotedata:StockVolume>20452658</quotedata:StockVolume>
        <quotedata:PrevCls>25.31</quotedata:PrevCls>
        <quotedata:ChangePercent>0.20</quotedata:ChangePercent>
        <quotedata:FiftyTwoWeekRange>22.73 - 31.58</quotedata:FiftyTwoWeekRange>
        <quotedata:EarnPerShare>2.326</quotedata:EarnPerShare>
        <quotedata:PE>10.88</quotedata:PE>
        <quotedata:CompanyName>Microsoft Corpora</quotedata:CompanyName>
    </quotedata:StockQuoteEvent>
</quotedata:AllStockQuoteStream>
";
initContext = JNDIContext.getInstance().getInitContext();

topicConnectionFactory =
JNDIContext.getInstance().getTopicConnectionFactory();

AllStockQuotesPublisher publisher = new AllStockQuotesPublisher();

XMLStreamReader reader1 = StAXUtils.createXMLStreamReader(new
ByteArrayInputStream(xmlElement1.getBytes()));
StAXOMBuilder builder1 = new StAXOMBuilder(reader1);
OMElement OMMessage1 = builder1.getDocumentElement();
publisher.publish("AllStockQuotes", OMMessage1);

XMLStreamReader reader2 = StAXUtils.createXMLStreamReader(new
ByteArrayInputStream(xmlElement2.getBytes()));
StAXOMBuilder builder2 = new StAXOMBuilder(reader2);
OMElement OMMessage2 = builder2.getDocumentElement();
publisher.publish("AllStockQuotes", OMMessage2);

XMLStreamReader reader3 = StAXUtils.createXMLStreamReader(new
ByteArrayInputStream(xmlElement3.getBytes()));
StAXOMBuilder builder3 = new StAXOMBuilder(reader3);
OMElement OMMessage3 = builder3.getDocumentElement();
publisher.publish("AllStockQuotes", OMMessage3);

/**
 * Publish message to given topic
 * @param topicName - topic name to publish messages
 * @param message   - message to send
 */

public void publish(String topicName, OMElement message) {
    // create topic connection
    TopicConnection topicConnection = null;
    try {
        topicConnection =
topicConnectionFactory.createTopicConnection();
        topicConnection.start();
    } catch (JMSException e) {
        System.out.println("Can not create topic connection." + e);
        return;
    }
    // create session, producer, message and send message to given
destination(topic)
    // OMElement message text is published here.
    Session session = null;
    try {
        session = topicConnection.createTopicSession(false,
Session.AUTO_ACKNOWLEDGE);
        Topic topic = session.createTopic(topicName);
        MessageProducer producer = session.createProducer(topic);
        TextMessage jmsMessage =
session.createTextMessage(message.toString());
    producer.send(jmsMessage);
    producer.close();
    session.close();
    topicConnection.stop();
    topicConnection.close();
} catch (JMSException e) {
    System.out.println("Can not subscribe." + e);
}
}
Subscribing for filtered events and notifications from CEP server

a. Using a JMS client receiver

Following class acts as a JMS topic subscriber client. We register a subscription for filtered events we get from CEP triggered according to the query at bucket we have defined (Actually we are subscribing for a topic created at WSO2 Message broker, to which CEP will publish filtered events and notifications internally).

```java
package org.wso2.cep.sample.jms.andes.xmlMessage;
import org.wso2.cep.sample.jms.andes.JNDIContext;
import javax.jms.JMSException;
import javax.jms.MapMessage;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;
import javax.naming.InitialContext;
import javax.xml.stream.XMLStreamException;
import java.util.Enumeration;
public class FastMovingStockQuotesSubscriber implements MessageListener {
    private static InitialContext initContext = null;
    private static TopicConnectionFactory topicConnectionFactory = null;
    private boolean messageReceived = false;
    static String TOPIC = "FastMovingStockQuotes";
    public static void main(String[] args) throws XMLStreamException {
        initContext = JNDIContext.getInstance().getInitContext();
        topicConnectionFactory = JNDIContext.getInstance().getTopicConnectionFactory();
        new FastMovingStockQuotesSubscriber().subscribe(TOPIC);
    }
    public void subscribe(String topicName) {
        // create connection
        TopicConnection topicConnection = null;
        try {
            topicConnection = topicConnectionFactory.createTopicConnection();
        } catch (JMSException e) {
            System.out.println("Can not create topic connection." + e);
            return;
        }
        // create session, subscriber, message listener and listen on that topic
```
TopicSession session = null;
try {
    session = topicConnection.createTopicSession(false,
            javax.jms.Session.AUTO_ACKNOWLEDGE);
    Topic topic = session.createTopic(topicName);
    TopicSubscriber subscriber = session.createSubscriber(topic);
    subscriber.setMessageListener(this);
    topicConnection.start();
    synchronized (this) {
        while (!messageReceived) {
            try {
                this.wait();
            } catch (InterruptedException e) {
            }
        }
    }
} catch (JMSException e) {
    System.out.println("Can not subscribe." + e);
}

public void onMessage(Message message) {
    if (message instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message;
        try {
            System.out.println("output = " + textMessage.getText());
            synchronized (this) {
                messageReceived = true;
            }
        } catch (JMSException e) {
            System.out.println("error at getting text out of received message. = " + e);
        }
    } else if (message instanceof MapMessage) {
        try {
            Enumeration enumeration = ((MapMessage) message).getMapNames();
            for (; enumeration.hasMoreElements(); ) {
                System.out.println(((MapMessage) message).getString((String) enumeration.nextElement()));
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    } else {
        System.out.println("Received message is not a text/map message.");
    }
}
b. Using a web service message receiver

Deploying receiver service

First we have to deploy a web service at any WSO2 Server which would act as the receiver web service for messages from CEP server. We will use CEP itself to deploy such a web service.

The steps are as follows:

1. In a command prompt, switch to the FastMovingStockQuoteReceiverService services directory: `<CEP_HOME>/ samples/services/FastMovingStockQuoteReceiverService`
   For example, in Linux: `cd <CEP_HOME>/samples/services/FastMovingStockQuoteReceiverService`
2. From there, type ant. This will deploy the FastMovingStockQuoteReceiverService in CEP itself. You can follow the server logs to check whether FastMovingStockQuoteReceiverService.arr has been properly deployed. You will also be able to see the axis2 service in the services list.

Configuring receiver service

We need to configure the FastMovingStockQuoteReceiverService in order to receive the output events emitted by the bucket under the FastMovingStockQuotes topic. Here we will be creating FastMovingStockQuotes topic in the WSO2 Message Broker and subscribe FastMovingStockQuoteReceiverService on that topic.

The steps are as follows:

1. Sign In. Enter your user name and password to log on to the Message Broker Management Console.
2. Click **Add** under the **Topics** menu in the **Manage** section of the left panel.
3. Specify the topic name in the topic input text box. In this case, the topic name is "FastMovingStockQuotes" (the output topic). Click **Add Topic**. The topic is added to the server and you will be directed to the Topic Browser page.
4. Once you click on the topic in the topic browser page, you will be able see four links. Click the **Subscribe** link and you will be directed to the Subscribe page.
5. Create subscription with the following details. Once you are done click **Subscribe**.

```
topic   : FastMovingStockQuotes (Output topic)  
subscription mode: Topic only subscription
URL :  
http://localhost:9763/services/FastMovingStockQuoteService/getOMEleme  
nt
expiration Time  : select a future date from calender
```

Once you click **Subscribe**, you will be directed to the Topic Browser page.

6. You can verify whether you have correctly subscribed to the topic by clicking the **Details** link of that topic in the topic browser page.

Once you click on that, you will be directed to the “topic details” page and there you will find all the subscriptions for that topic and its children (in this case it does not exists) and permission on that topic. Apart from that with the **Publish** section, you can publish a test XML message to that topic and check whether your
subscription URL has been properly subscribed.

Running the Samples

1. If you are using a JMS subscriber client i.e "Using a JMS Client Receiver", build and run the topic subscriber class provided above. Else if you are using the "Web Service Message Receiver" above configurations under section (b) is enough.
2. Run the JMS publisher client.
3. It can be noticed that CEP analyzes the events we have published and fire outputs if the last traded amount vary by 2 percent with regards to the average traded price within past 2 minutes.

Observation

Observe the console where we are running the JMS subscriber client or server console where message receiving web service is deployed. You will see some logs like below.

Integrating WSO2 DSS

This page describes how to integrate WSO2 Message Broker with WSO2 Data Services Server to facilitate subscribing data services to JMS queues or topics and receiving messages from the broker. It contains the following sections:

- Setting up WSO2 Message Broker
- Setting up WSO2 DSS
- Testing the integration

Setting up WSO2 Message Broker

1. Download and install WSO2 MB according to the instructions in Installation Guide.
2. It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts. Therefore, to avoid port conflicts, apply a port offset in `<MB_HOME>/repository/conf/carbon.xml` file changing the offset value to 1. For example:
3. Start the Message Broker by running `<MB_HOME>/bin/wso2server.sh` (on Linux) or `<MB_HOME>/bin/wso2server.bat` (on Windows). Note that Message Broker must be up and running before starting the Data Services Server.

**Setting up WSO2 DSS**

1. Download and install the WSO2 DSS binary distribution (see the installation instructions in the DSS documentation). The unzipped DSS distribution folder is referred to as `<DSS_HOME>`.
2. WSO2 DSS does not have a default enabled JMS transport configuration for communicating with the Message Broker. Therefore, we need to add a `<transport receiver>` block for MB 2.x.x by editing `<DSS_HOME>/repository/conf/axis2/axis2.xml` as follows:

```xml
<Ports>
    <!-- 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
    -->
    <Offset>1</Offset>
</Ports>
```
3. Also, uncomment the `<transport sender>` block for JMS in the same file as follows:

```xml
<!-- uncomment this and configure to use connection pools for sending messages>  
<transportSender name="jms"  
class="org.apache.axis2.transport.jms.JMSSender"/>-->
```

4. Create and add the following `jndi.properties` file into the `<DSS_HOME>/repository/conf/directory` to point to the running Message Broker:
5. Copy the following client library JAR files from the `<MB_HOME>/client-lib` folder to `<DSS_HOME>/repository/components/lib`.

   - andes-client-0.13.wso2v8
   - geronimo-jms_1.1_spec-1.1.0.wso2v1

6. **Important**: If you are integrating with DSS version 3.0.1, replace `<DSS_HOME>/repository/components/plugins/axis2-transport-jms_1.1.0.wso2v7.jar` with the one attached here. This JAR file contains JMS transport changes that affect the integration of DSS 3.0.1 with MB. Future versions of Data Services Server will have this fix built in, so replacing this JAR will be not necessary.

7. Once the JMS transport is enabled for DSS, it will reflect in all the services deployed in Data Services Server, which can cause an unexpected error during server start up. To prevent this error, edit and add the following parameters into the `<service>` entry of `<DSS_HOME>/repository/deployment/server/dataservices/samples/RDBMSSample_services.xml`. 
Testing the integration

Deploy a sample data_service in DSS. A JMS queue for the name of this service will be generated in the Message Broker. Publish a message to this queue and it will be received by the corresponding data_service in DSS. There can be any message processing operations performed using these messages, inside the data_service now.

Integrating with Application Servers

WSO2 Message Broker (WSO2 MB) is typically used for brokering messages between JMS clients (JMS publishers and JMS subscribers). These JMS clients can work as a web application deployed in an application server. See the following topics for more information:

- Integrating with Oracle Weblogic Server
- Integrating with WSO2 AS

Integrating with Oracle Weblogic Server

In the sections that follow, we will look at how web applications deployed in Oracle Weblogic Server can communicate with WSO2 Message Broker (WSO2 MB) as JMS subscribers and JMS publishers.

- Setting up Oracle Weblogic Server and WSO2 MB
- Developing a web application (JMS Subscriber/Publisher)
  - Step 1: Writing the JMS client
  - Step 2: Building the web application
- Deploying the web application in Oracle Weblogic server
- Testing the web application with WSO2 MB

Setting up Oracle Weblogic Server and WSO2 MB

Note that these instructions have been tested with Weblogics server 12.1.2.

Follow the steps given below to set up Oracle Weblogic Server

1. Download Oracle Weblogics server 12.1.2 from here.
2. Follow the instructions given in the README.txt file to install and start the server.

Developing a web application (JMS Subscriber/Publisher)

To create a web application that can subscribe/publish to a queue or topic in WSO2 MB, you will need to write a JMS client (subscriber/publisher) and build it as a web application (WAR file), which can be deployed in the
Follow the steps given below to build a web application that can subscribe and publish to a queue in WSO2 MB.

**Step 1: Writing the JMS client**

First, we will write a JMS client that subscribes to a queue in WSO2 MB: Create a java project using the source files given below and compile it.

- SampleQueueSender.java
- SampleQueueReceiver.java
- Main.java

The following class is used to create the sample client that creates a queue named testQueue in WSO2 MB and sends messages to that queue:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class SampleQueueSender {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String QUEUE_NAME_PREFIX = "queue.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "testQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    public void sendMessages() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
        InitialContext ctx = new InitialContext(properties);
        QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        queueConnection = connFactory.createQueueConnection();
    }
}
```
queueConnection.start();
queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
// Send message
Queue queue = (Queue)ctx.lookup(queueName);
// create the message to send
TextMessage textMessage = queueSession.createTextMessage("Test Message Content");
javax.jms.QueueSender queueSender = queueSession.createSender(queue);
queueSender.send(textMessage);
queueSender.close();
queueSession.close();
queueConnection.close();

private String getTCPConnectionURL(String username, String password) {
// amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
            .append("amqp://").append(username).append(":").append(password)
            .append("@").append(CARBON_CLIENT_ID)
            .append("/").append(CARBON_VIRTUAL_HOST_NAME)
            .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
            .append(CARBON_DEFAULT_PORT).append("'").append(""")
            .toString();
The following class is used to create the sample client that receives the messages published to the testqueue queue in WSO2 MB:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.MessageConsumer;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class SampleQueueReceiver {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "testQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    public MessageConsumer registerSubscriber() throws NamingException, JMSException{
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put("queue."+ queueName,queueName);
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        queueConnection = connFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
        //Receive message
    }
}
```
Queue queue = (Queue) ctx.lookup(queueName);
MessageConsumer consumer = queueSession.createConsumer(queue);
return consumer;
}

public void receiveMessages(MessageConsumer consumer) throws NamingException, JMSException {
    TextMessage message = (TextMessage) consumer.receive();
    System.out.println("Got message from queue receiver==>" + message.getText());
    // Housekeeping
    consumer.close();
    queueSession.close();
    queueConnection.stop();
    queueConnection.close();
}

private String getTCPConnectionURL(String username, String password) {
    // amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT).append("'").
toString();
The `Main.java` class defines the main method for calling both the subscriber and publisher clients. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;

public class Main {
    public static void main(String[] args) throws NamingException, JMSException {
        SampleQueueReceiver queueReceiver = new SampleQueueReceiver();
        MessageConsumer consumer = queueReceiver.registerSubscriber();
        SampleQueueSender queueSender = new SampleQueueSender();
        queueSender.sendMessages();
        queueReceiver.receiveMessages(consumer);
    }
}
```

Once you build the above client, you will have the following class files:

- JmsQueue.class
- SampleQueueReceiver.class
- SampleQueueSender.class

**Step 2: Building the web application**

Now, let's build a web application using the source files (JMS client) given above. This web application will also have an interface for invoking the JMS client. Follow the steps given below.

1. Create a folder structure in your local machine by following the steps given below.
   
   1. Create the following folder structure in your local machine:

```
    -WebApp/
    -WEB-INF/
    -lib/
    -classes/
    -MyPackage/
        -web.xml
    Index.html
```

   2. And the class files you developed in step1 to the WEB-INF/lib/classes/MyPackage directory. You can download these class files from here.

   3. Update the `web.xml` file with the following content:
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                             http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
         version="2.4">
    <display-name>JMS Queue Sample Web Application</display-name>
    <description>
        This is a sample web application which sends and receives queue message using wso2 message broker.
    </description>
    <servlet>
        <servlet-name>JMSQueueServlet</servlet-name>
        <servlet-class>mypackage.JmsQueue</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>JMSQueueServlet</servlet-name>
        <url-pattern>/jmsqueue</url-pattern>
    </servlet-mapping>
</web-app>

4. Copy the JARs in the `<MB_HOME>/client-lib` directory to the `WEB-INF/lib` directory.
5. Update the `index.html` file with the following content:
<html>
<head>
<title>JMS Queue Sample Web Application</title>
</head>
<body bgcolor=white>
<table border="0">
<tr>
<td>
<h1>Sample Queue Web Application</h1>
</td>
</tr>
<tr>
<td>
<p>This is the home page for a sample web application which
sends and receives queue messages using wso2 message broker.</p>
</td>
</tr>
</table>
<p>Please click on following link to run the sample : </p>
<ul>
<li><a href="jmsqueue">Queue send receive sample
servlet</a></li> 
</ul>
</body>
</html>

Your <WEBAPP_HOME> directory should now have the following content:

```
-WebApp/
 -WEB-INF/
 -lib/
  - andes-client-3.0.1.jar
  - log4j-1.2.13.jar
  - org.wso2.securevault-1.0.0-wso2v2.jar
  - geronimo-jms_1.1_spec-1.1.0.wso2v1.jar
  - org.wso2.carbon.logging-4.4.1.jar
  - slf4j-1.5.10.wso2v1.jar
 -classes/
 -MyPackage/
  - JmsQueue.java
  - SampleQueueReceiver.java
  - SampleQueueSender.java
 -web.xml
Index.html
```

2. Open a terminal and navigate to the <WEBAPP_HOME> folder.

3. Execute the following command from the terminal in order to create the .war file: 
   ```
   Jar -cvf jmsQueue.war *
   ```
1. This will create the `jmsQueue.war` file in the `WebApp` folder.

**Deploying the web application in Oracle Weblogic server**

Deploy the web application in the Weblogic server. You can find detailed instructions in the Oracle documentation.

**Testing the web application with WSO2 MB**

Follow the steps given below to test how these JMS clients communicate with WSO2 MB.

1. Be sure that the WSO2 MB server is started.
2. Log in to the console of Weblogic server and invoke the `jmsQueue` web application.
3. Click **Queue send receive sample servlet** in the web application to execute the JMS queue send-receive sample operation.
4. You can verify this from the console logs of the broker as well: A subscription has been added and deleted for the queue named “testQueue”.

**Integrating with WSO2 AS**

In the sections that follow, we will look at how web applications deployed in WSO2 Application Server (WSO2 AS) can communicate with WSO2 Message Broker (WSO2 MB) as JMS subscribers and JMS publishers.

- Setting up WSO2 Message Broker
- Setting up WSO2 AS
- Developing a web application (JMS Subscriber/Publisher)
  - Step 1: Writing the JMS client
  - Step 2: Building the web application
- Deploying the web application in WSO2 AS
- Testing the web application with WSO2 MB

**Setting up WSO2 Message Broker**

Follow the steps given below to start WSO2 MB.

1. Download and install WSO2 MB.
2. Start WSO2 MB by executing one of the following startup scripts:
   - In Linux: `<MB_HOME>/bin/wso2server.sh`
   - In Windows: `<MB_HOME>/bin/wso2server.bat`

Note that WSO2 MB must be up and running before starting the application server.

**Setting up WSO2 AS**

Follow the steps given below to set up and start the application server.

1. Download and install WSO2 AS.
2. To avoid a port conflict with WSO2 MB, apply a port offset for WSO2 AS in the `carbon.xml` file (stored in the `<MB_HOME>/repository/conf` folder). See the following example:

   ```xml
   <Offset>1</Offset>
   ```

Since all WSO2 products use the same port (9443) in the default configuration, it is not possible to
3. Open the axis2.xml file stored in the `<AS_HOME>/repository/conf/axis2` folder and enable the JMS transport receiver for WSO2 MB by uncommenting the following:

```xml
<!--Configure for JMS transport support with WSO2 MB 2.x.x -->
<transportReceiver name="jms"
  class="org.apache.axis2.transport.jms.JMSListener">
  <parameter name="myTopicConnectionFactory" locked="false">
    <parameter name="java.naming.factory.initial" locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
    <parameter name="java.naming.provider.url" locked="false">repository/conf/jndi.properties</parameter>
    <parameter name="transport.jms.ConnectionFactoryJNDIName" locked="false">TopicConnectionFactory</parameter>
    <parameter name="transport.jms.ConnectionFactoryType" locked="false">topic</parameter>
  </parameter>
  <parameter name="myQueueConnectionFactory" locked="false">
    <parameter name="java.naming.factory.initial" locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
    <parameter name="java.naming.provider.url" locked="false">repository/conf/jndi.properties</parameter>
    <parameter name="transport.jms.ConnectionFactoryJNDIName" locked="false">QueueConnectionFactory</parameter>
    <parameter name="transport.jms.ConnectionFactoryType" locked="false">queue</parameter>
  </parameter>
  <parameter name="default" locked="false">
    <parameter name="java.naming.factory.initial" locked="false">org.wso2.andes.jndi.PropertiesFileInitialContextFactory</parameter>
    <parameter name="java.naming.provider.url" locked="false">repository/conf/jndi.properties</parameter>
    <parameter name="transport.jms.ConnectionFactoryJNDIName" locked="false">QueueConnectionFactory</parameter>
    <parameter name="transport.jms.ConnectionFactoryType" locked="false">queue</parameter>
  </parameter>
</transportReceiver>
```

4. Uncomment the `<transport sender>` block for JMS in the same file as follows:
4. Uncomment this and configure to use connection pools for sending messages:
   
   ```xml
   <transportSender name="jms"
   class="org.apache.axis2.transport.jms.JMSSender"/>
   ```

5. Add the following `jndi.properties` file to the `<AS_HOME>/repository/conf` directory to make sure that WSO2 AS is pointing to the running message broker.

   ```ini
   # register some connection factories
   # connectionfactory.[jndiname] = [ConnectionURL]
   connectionfactory.QueueConnectionFactory = amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5672'
   connectionfactory.TopicConnectionFactory = amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5672'
   ```

6. Copy the following JAR files from the `<MB_HOME>/client-lib` folder to the `<AS_HOME>/repository/components/lib` folder:
   - andes-client-3.1.1
   - geronimo.jms_1.1_spec-1.1.0.wso2v1
   - log4j-1.2.13
   - org.wso2.carbon.logging-4.4.1
   - org.wso2.securevault-1.0.0-wso2v2
   - slf4j-1.5.10.wso2v1

7. Download the `axis2-transport-all-1.0.0.jar` from here and add it to the `<AS_HOME>/repository/components/lib` directory.

8. Save all the files.

9. Now, start WSO2 AS by executing the startup script:
   - In Linux: `<AS_HOME>/bin/wso2server.sh`
   - In Windows: `<AS_HOME>/bin/wso2server.bat`

### Developing a web application (JMS Subscriber/Publisher)

To create a web application that can subscribe/publish to a queue or topic in WSO2 MB, you will need to write a JMS client (subscriber/publisher) and build it as a web application (WAR file), which can be deployed in the application server.

Follow the steps given below to build a web application that can subscribe and publish to a queue in WSO2 MB.

**Step 1: Writing the JMS client**

First, we will write a JMS client that subscribes to a queue in WSO2 MB: Create a java project using the source files given below and compile it:

- `SampleQueueSender.java`
- `SampleQueueReceiver.java`
- `Main.java`

The following class is used to create the sample client that creates a queue named `testQueue` in WSO2 MB and sends messages to that queue:
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class SampleQueueSender {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String QUEUE_NAME_PREFIX = "queue.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "testQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    public void sendMessages() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        queueConnection = connFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
        // Send message
        Queue queue = (Queue) ctx.lookup(queueName);
        // create the message to send
        TextMessage textMessage = queueSession.createTextMessage("Test Message Content");
        javax.jms.QueueSender queueSender = queueSession.createSender(queue);
        queueSender.send(textMessage);
        queueSender.close();
        queueSession.close();
        queueConnection.close();
    }
    private String getTCPConnectionURL(String username, String password) {
        // Implementation
    }
}
amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'

    return new StringBuffer()

    .append("amqp://").append(username).append(":").append(password)
    .append("@").append(CARBON_CLIENT_ID)
    .append("/").append(CARBON_VIRTUAL_HOST_NAME)

    .append("?brokerlist='tcp://")
    .append(CARBON_DEFAULT_HOSTNAME).append(":")
    .append(CARBON_DEFAULT_PORT).append("'\n")
    .toString();
The following class is used to create the sample client that receives the messages published to the `testqueue` queue in WSO2 MB:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.MessageConsumer;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

public class SampleQueueReceiver {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "testQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    public MessageConsumer registerSubscriber() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put("queue." + queueName, queueName);
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        queueConnection = connFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
        //Receive message
```
Queue queue = (Queue) ctx.lookup(queueName);
MessageConsumer consumer = queueSession.createConsumer(queue);
return consumer;
}

public void receiveMessages(MessageConsumer consumer) throws
NamingException, JMSException {
TextMessage message = (TextMessage) consumer.receive();
System.out.println("Got message from queue receiver==>" +
message.getText());
// Housekeeping
consumer.close();
queueSession.close();
queueConnection.stop();
queueConnection.close();
}

private String getTCPConnectionURL(String username, String password) {
// amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT).append("'")
        .toString();
The `Main.java` class defines the main method for calling both the subscriber and publisher clients. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;
public class Main {
    public static void main(String[] args) throws NamingException, JMSException {
        SampleQueueReceiver queueReceiver = new SampleQueueReceiver();
        MessageConsumer consumer = queueReceiver.registerSubscriber();
        SampleQueueSender queueSender = new SampleQueueSender();
        queueSender.sendMessages();
        queueReceiver.receiveMessages(consumer);
    }
}
```

Once you build the above client, you will have the following class files:

- JmsQueue.class
- SampleQueueReceiver.class
- SampleQueueSender.class

**Step 2: Building the web application**

Now, let's build a web application using the source files (JMS client) given above. This web application will also have an interface for invoking the JMS client. Follow the steps given below.

1. Create a folder structure in your local machine by following the steps given below.

   1. Create the following folder structure in your local machine:

   ```
   -WebApp/
     -WEB-INF/
     -lib/
     -classes/
     -MyPackage/
         -web.xml
     Index.html
   ```

   2. And the class files you developed in step1 to the WEB-INF/lib/classes/MyPackage directory. You can download these class files from here.

   3. Update the `web.xml` file with the following content:
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">
    <display-name>JMS Queue Sample Web Application</display-name>
    <description>
        This is a sample web application which sends and receives queue message using wso2 message broker.
    </description>
    <servlet>
        <servlet-name>JMSQueueServlet</servlet-name>
        <servlet-class>mypackage.JmsQueue</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>JMSQueueServlet</servlet-name>
        <url-pattern>/jmsqueue</url-pattern>
    </servlet-mapping>
</web-app>

4. Copy the JARs in the `<MB_HOME>/client-lib` directory to the `WEB-INF/lib` directory.
5. Update the `index.html` file with the following content:
<html>
<head>
<title>JMS Queue Sample Web Application</title>
</head>
<body bgcolor=white>
<table border="0">
<tr>
<td>
<h1>Sample Queue Web Application</h1>
</td>
</tr>
<tr>
<td>
<p>This is the home page for a sample web application which sends and receives queue messages using wso2 message broker.</p>
</td>
</tr>
</table>
<p>Please click on following link to run the sample:
<ul>
<li><a href="jmsqueue">Queue send receive sample servlet</a></li>
</ul>
</p>
</body>
</html>

Your <WEBAPP_HOME> directory should now have the following content:

```
-WebApp/
-WEB-INF/
 -lib/
  -andes-client-3.0.1.jar
  -log4j-1.2.13.jar
  -org.wso2.securevault-1.0.0-wso2v2.jar
  -geronimo-jms_1.1_spec-1.1.0.wso2v1.jar
  -org.wso2.carbon.logging-4.4.1.jar
  -slf4j-1.5.10.wso2v1.jar
-classes/
-MyPackage/
  -JmsQueue.java
  -SampleQueueReceiver.java
  -SampleQueueSender.java
-classes-
web.xml
Index.html
```

2. Open a terminal and navigate to the <WEBAPP_HOME> folder.

3. Execute the following command from the terminal in order to create the .war file: 
```
Jar -cvf jmsQueue.war *
```
This will create the `jmsQueue.war` file in the WebApp folder.

**Deploying the web application in WSO2 AS**

You can now deploy the `jmsQueue` WAR file in WSO2 AS using one of the following options:

- Log in to the management console of WSO2 AS and click **Web Applications** under **Applications** in the navigator to open the **Upload Web Applications** screen. Now you can upload the `jmsQueue.war` file to the server.

- Alternatively, you can simply add the WAR file to the `<AS_HOME>/repository/deployment/server/webapps` directory.

Now you have successfully deployed a web application (containing a JMS subscriber and JMS publisher) in WSO2 AS.

**Testing the web application with WSO2 MB**

Follow the steps given below to test how these JMS clients communicate with WSO2 MB.

1. Be sure that the WSO2 MB server is started.
2. Log in to the management console of WSO2 AS and invoke the `jmsQueue` web application.
3. Click **Queue send receive sample servlet** in the web application to execute the JMS queue send-receive sample operation.
4. You can verify this from the console logs of the broker as well: A subscription has been added and deleted for the queue named "testQueue".

**Managing Subscriptions**

When you have a message broker set up, it is important for administrators to always know the existing subscribers connected to the individual nodes in an MB cluster. It should also be possible to identify the respective nodes from which messages are consumed by each subscriber. These can be done using the Management Console of WSO2 MB.

- **Managing Queue subscriptions**
  - Viewing Queue subscriptions
  - Closing Queue subscriptions

- **Managing Topic subscriptions**
  - Viewing Topic subscriptions
  - Closing topic subscriptions

- **Closing subscriptions in an MB cluster**

**Managing Queue subscriptions**

You can view queue subscription details and close any subscription as required. See the following topics:

**Viewing Queue subscriptions**

Note that only authorized users will be allowed to view queue subscriptions using the Management Console as explained below. Find out about how permissions are enabled for a user from this link.

To view all the queue subscriptions, open the **Management Console** and click **Queue Subscriptions** in the **Main tab**. Whenever a queue subscription is created, the subscription page is updated. Subscription entries will exist as long as the subscriber connections are live.
The table below describes the information explained below:

<table>
<thead>
<tr>
<th>Heading</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exchange</td>
<td>The exchange to which the subscription is made. For queues, this is amq.direct.</td>
</tr>
<tr>
<td>Queue Name</td>
<td>Name of the queue subscription.</td>
</tr>
<tr>
<td>Destination</td>
<td>The JMS destination represented by the queue.</td>
</tr>
<tr>
<td>Durability</td>
<td>Determines whether or not the queue is durable.</td>
</tr>
<tr>
<td>Active</td>
<td>The status of the subscriber. If a connection to the server exists, the status will be 'Active'.</td>
</tr>
<tr>
<td>Owned Node</td>
<td>The node from which the subscription is made to the cluster. The IP address of the node is displayed.</td>
</tr>
</tbody>
</table>

In addition to the above, the number of messages tentatively pending for a subscriber are shown. While the subscriber is receiving messages, you can click **Refresh** to see the message count eventually decrease to zero.

**Closing Queue subscriptions**

Note that only authorized users will be allowed to close a queue subscription using the Management Console as explained below. Find out about how permissions are enabled for a user from this link.

If there are erroneous queue subscribers connected to a particular queue, you (as the broker administrator) can remove such subscriptions by using the **Close** operation associated with the subscription. The following message will be prompted:
Managing Topic subscriptions

Topic subscription can be managed in the same way as queue subscriptions.

Viewing Topic subscriptions

To view all the topic subscriptions, open the Management Console and click Topic Subscriptions in the Main tab. Topic subscriptions display similar information to queues. Some notable points are:

1. Topic subscriptions display a queue name with subscription information. This is the internal queue name where messages belonging to the subscriber are accumulated.
2. There are temporary topic subscriptions and durable topic subscriptions. Temporary topic subscriptions vanish as soon as the connection from the subscriber is closed, but durable topic subscribers are not removed from the broker. They will exist inside the broker cluster in the inActive mode.
3. The message count is not displayed for temporary topic subscriptions, whereas it is displayed for durable topic subscribers.
4. As inactive durable topic subscribers belong to the broker cluster, the Owned Node field is no longer applicable. If you make a subscription to the topic using the same subscription ID, that subscriber will be active once again.
5. If you have the necessary permissions, you can unsubscribe the inactive durable topic subscribers. All messages belonging to that subscriber are then removed from the broker cluster. You cannot unsubscribe active durable topic subscriptions using the management console.

Closing topic subscriptions

If there are erroneous queue subscribers connect to a particular queue, you (as the broker administrator) can remove such subscriptions by using the Close operation associated with the subscription. Closing a subscription will affect topic subscriptions as explained below.

Note the following:
The subscriber connection will be closed, and all associated resources are released.

### Non-durable topic subscriptions:
The subscriber connection will be closed, and all associated resources are released.

### Durable topic subscriptions:
The subscriber connection will be closed. However, the subscription will be listed as an 'inactive' subscription as shown below. All messages received by the topic subscriber is registered, cloned and persistently stored until the subscriber becomes active again.

#### Closing subscriptions in an MB cluster

In a clustered set up of WSO2 MB, subscribers are required to connect with an individual node in the cluster. Therefore, if you want to close a subscription, it is necessary to log in to the Management Console of the relevant MB node in the cluster. When you log in to the Management Console of one node in the cluster the Close operation will only be available for the subscriptions that are connected to the the MB instance to which you are logged in. For example, consider a broker cluster with 3 MB nodes (MB1, MB2 and MB3). You may have 3 subscribers connected to the 3 respective nodes: Sub1 connected to MB1, Sub2 connected to MB2 and Sub3 connected to MB3. In this scenario, you need to log in to the Management Console of MB1 in order to close Sub1. Therefore, this functionality of closing subscriptions is not symmetric across the cluster.
Samples

Several sample scenarios of the WSO2 Message Broker are explained in this section. These samples can be used as references to build your own application using various features of the Message Broker. The samples shipped with WSO2 Message Broker are stored in the `<CARBON_HOME>/samples` directory.

See the following topics for details:

- About MB samples
- Setting up and running the samples

About MB samples

The MB samples use several sample clients. For example, to illustrate a scenario where messages are published and received by subscribers, we need two clients: one for publishing messages to MB and another for subscribing and receiving messages.

These clients are defined via classes that are saved in the separate folders dedicated for each sample in the `<MB_HOME>/samples` directory. A sample may have more than one client to perform different actions. Since there are interdependencies between different clients used in a sample, the classes defining them need to be bound to each other. To achieve this, the sample folder where the classes of the clients are stored, also contains a class named `Main.class`, which defines the method for calling the other classes in the same directory.

For example, consider the "JMSQueueClient" sample, which you will find in the `<MB_HOME>/samples` directory. If you see the contents of the JMSQueueClient sample folder, you will find the following classes (inside the `Client/target/classes/org/sample/jms` folder):

- SampleQueueReceiver.class: The class file defining the JMS client for receiving messages from the queue.
- SampleQueueSender.class: The class file defining the JMS client for sending messages to the queue.
- Main.class: The main class that binds the two client classes.

Setting up and running the samples

See the following topics for instructions on how to setup and run the samples in WSO2 MB:

- Setting up the MB Samples
- JMS Client Samples
- Creating a Durable Topic Subscription
- Creating Hierarchical Topic Subscriptions
- CSharp Client Samples
- Using MQTT Transport
- Using Transactional Sessions
- Setting Message Expiration

Setting up the MB Samples

The following is a list of prerequisites that need to be carried out before running any of the MB samples.

1. To run the samples, you must have the correct versions of the following installed.
   - Java Development Kit/JRE
   - Apache Ant
   - Apache Maven
   - See the Installation Prerequisites section for more details.
2. If you want to run MB in DEBUG mode, you need to set the log level to DEBUG for the relevant appender as
2. A sample can be built using Apache Ant and/or Apache Maven. Make sure that the following dependencies are saved as required:
   1. A sample built using Apache Ant requires the following files to be saved in `<MB_HOME>/client-lib` directory. Note that these JARs are included in the product binary pack by default.
      1. `andes-client-3.1.1.jar`
      2. `geronimo-jms_1.1_spec-1.1.1.wso2v1.jar`
      3. `log4j-1.2.13.jar`
      4. `slf4j-1.5.10.wso2v1.jar`
   2. When you build MQTT samples, the Maven repositories required for the samples should be saved in the `pom.xml` file of the MQTT samples. For example, the dependencies for the Simple MQTT Client sample should be included in the `<MB_HOME>/samples/SimpleMqttClient/pom.xml` file. Note that these dependencies are added to the MQTT samples shipped by default with WSO2 MB.

```xml
<dependency>
    <groupId>org.wso2.andes.wso2</groupId>
    <artifactId>andes-client</artifactId>
    <version>0.13.wso2v3</version>
</dependency>
<dependency>
    <groupId>org.apache.geronimo.specs.wso2</groupId>
    <artifactId>geronimo-jms_1.1_spec</artifactId>
    <version>1.1.0.wso2v1</version>
</dependency>
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
<dependency>
    <groupId>slf4j.wso2</groupId>
    <artifactId>slf4j</artifactId>
    <version>1.5.10.wso2v1</version>
</dependency>
```

### JMS Client Samples

This set of samples demonstrate the use of JMS APIs with Message Broker to publish and subscribe messages.

- Sending and Receiving Messages Using Queues
- Sending and Receiving Messages Using Topics
- Receiving Messages with JMS Message Listener
- Receiving Messages from an MB Cluster
- JMS Selectors
- Sending and Receiving Messages with TTL

When creating a JMS connection to a queue or a topic using a sample JMS client, a single `javax.jms.Connection` can handle only 256 JMS sessions. You can change this setting by adding the `<maximumChannelCount>` property to the `qpid-config.xml` file (stored in the `<MB_HOME>/repository/conf/advanced` directory).
Sending and Receiving Messages Using Queues

This sample demonstrates how persistent queues can be created and used in WSO2 Message Broker using the JMS API. It first introduces a sample JMS client named `QueueSender` which is used to send messages to a known, created queue in WSO2 Message Broker. Then it introduces a sample JMS client named `QueueReceiver` to receive messages and print them in the console.

- Prerequisites
- About the sample
- Building the sample
- Analyzing the output

Prerequisites

See Prerequisites to Run the MB Samples for a list of prerequisites.

About the sample

The `<MB_HOME>/Samples/JMSQueueClient/src/org/sample/jms` directory has the following classes:

- `SampleQueueSender.java`
- `SampleQueueReceiver.java`
- `Main.java`

This class is used to create the sample client which sends messages to the queue named `testQueue` in WSO2 MB. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class SampleQueueSender {
    public static final String QPID_ICF =
            "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory."
            ;
    private static final String QUEUE_NAME_PREFIX = "queue."
            ;
    private static final String CF_NAME = "qpidConnectionFactory"
            ;
    String userName = "admin";
    String password = "admin";
    public SampleQueueSender()
```
private static String CARBON_CLIENT_ID = "carbon";
private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
private static String CARBON_DEFAULT_HOSTNAME = "localhost";
private static String CARBON_DEFAULT_PORT = "5672";
String queueName = "testQueue";
private QueueConnection queueConnection;
private QueueSession queueSession;
public void sendMessages() throws NamingException, JMSException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, 
    getTCPConnectionURL(userName, password));
    properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
    InitialContext ctx = new InitialContext(properties);
    // Lookup connection factory
    QueueConnectionFactory connFactory = (QueueConnectionFactory)
    ctx.lookup(CF_NAME);
    queueConnection = connFactory.createQueueConnection();
    queueConnection.start();
    queueSession = queueConnection.createQueueSession(false,
    QueueSession.AUTO_ACKNOWLEDGE);
    // Send message
    Queue queue = (Queue)ctx.lookup(queueName);
    // create the message to send
    TextMessage textMessage = queueSession.createTextMessage("Test
    Message Content");
    javax.jms.QueueSender queueSender =
    queueSession.createSender(queue);
    queueSender.send(textMessage);
    queueSender.close();
    queueSession.close();
    queueConnection.close();
}
private String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{p
    ort}'
    return new StringBuffer()
    .append("amqp://").append(username).append(":").append(password)
    .append("@").append(CARBON_CLIENT_ID)
    .append("/").append(CARBON_VIRTUAL_HOST_NAME)
    .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
    .append(CARBON_DEFAULT_PORT).append("'"))
    .toString();
}
This class is used to create the sample client which receives the message sent to the `testQueue` queue and prints it in the console. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.MessageConsumer;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class SampleQueueReceiver {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "testQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    public MessageConsumer registerSubscriber() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put("queue." + queueName, queueName);
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        queueConnection = connFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
        // Receive message
```
Queue queue = (Queue) ctx.lookup(queueName);
MessageConsumer consumer = queueSession.createConsumer(queue);
return consumer;
}
public void receiveMessages(MessageConsumer consumer) throws
NamingException, JMSException {
TextMessage message = (TextMessage) consumer.receive();
System.out.println("Got message from queue receiver==>");
message.getText();
// Housekeeping
consumer.close();
queueSession.close();
queueConnection.stop();
queueConnection.close();
}
private String getTCPConnectionURL(String username, String password) {
// amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
return new StringBuffer()
.append("amqp://").append(username).append(":").append(password)
.append("").append(CARBON_CLIENT_ID)
.append("/").append(CARBON_VIRTUAL_HOST_NAME)
.append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
.append(CARBON_DEFAULT_PORT).append("'")
.toString();
The `Main.java` class defines the main method for calling both the clients mentioned above. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;
public class Main {
    public static void main(String[] args) throws NamingException, JMSException {
        SampleQueueReceiver queueReceiver = new SampleQueueReceiver();
        MessageConsumer consumer = queueReceiver.registerSubscriber();
        SampleQueueSender queueSender = new SampleQueueSender();
        queueSender.sendMessages();
        queueReceiver.receiveMessages(consumer);
    }
}
```

The following should be noted if you are writing a similar sample:

- It is not possible to use the `@` symbol in the username or password.
  
  It is also not possible to use the percentage (%) sign in the password. When building the connection string URL inside the andes client code of MB, the URL is parsed. This parsing exception happens because the percentage (%) sign acts as the escape character in URL parsing. If using the percentage (%) sign in the connection string is required, use the respective encoding character for the percentage (%) sign in the connection string. For example:

  If you need to pass `adm%in` as the password, then the `%` should be encoded with its respective URL encoding character. Therefore, you have to send it as `adm%25in`.

  For a list of possible URL parsing patterns, see URL encoding reference.

- In addition to using `javax.jms.QueueSender` class to send the messages you can also use a `javax.jms.MessageProducer` client and send the messages to a destination queue. Following is the way of creating a JMS MessageProducer.

  ```java
  javax.jms.MessageProducer messageProducer = queueSession.createProducer(queue);
  messageProducer.send(textMessage);
  messageProducer.close();
  ```

- When subscribing and publishing to a queue in a tenant the qualified queue name, `DOMAIN_NAME/Q`
Building the sample

Run the ant command from `<MB_HOME>/Samples/JMSQueueClient` directory.

Analyzing the output

You will get the following log in your console.

```
[java] Got message from queue receiver==>Test Message Content
```

To view the queue in the MB Management Console, log into the MB Management Console. Then click **Browse** under **Queues** in the **Main** tab. You will see a queue named **TestQueue** automatically created.

Sending and Receiving Messages Using Topics

This sample demonstrates how to create a topic, and to subscribe to/publish messages in it.

- **About the sample**
- **Prerequisites**
- **Building the sample**
- **Analyzing the output**

**About the sample**

The `<MB_HOME>/Samples/JMSQueueClient/src/org/sample/jms` directory has the following classes.

- **SampleTopicPublisher.java**
- **SampleTopicSubscriber.java**
- **Main.java**

This class is used to publish messages to a topic named **MYTopic**. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
```
public class SampleTopicPublisher {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String topicName = "MYTopic";

    public void publishMessage() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        System.out.println("getTCPConnectionURL(userName,password) = " + getTCPConnectionURL(userName, password));
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        TopicConnectionFactory connFactory = (TopicConnectionFactory) ctx.lookup(CF_NAME);
        TopicConnection topicConnection = connFactory.createTopicConnection();
        topicConnection.start();
        TopicSession topicSession = topicConnection.createTopicSession(false, TopicSession.AUTO_ACKNOWLEDGE);
        // Send message
        Topic topic = topicSession.createTopic(topicName);
        // create the message to send
        TextMessage textMessage = topicSession.createTextMessage("Test Message");
        javax.jms.TopicPublisher topicPublisher = topicSession.createPublisher(topic);
        topicPublisher.publish(textMessage);
        topicPublisher.close();
        topicSession.close();
        topicConnection.stop();
        topicConnection.close();
    }

    private String getTCPConnectionURL(String username, String password) {
        // amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
        return new StringBuffer()
                .append("amqp://").append(username).append(":").append(password)
                .append("@carbon/carbon?brokerlist='tcp://").append(topicName).append(':');
    }
}
.append("?brokerlist='tcp://")
.append(CARBON_DEFAULT_HOSTNAME).append(":")
.append(CARBON_DEFAULT_PORT).append("'")
This class is used to subscribe to MYTopic. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class SampleTopicSubscriber {
    public static final String QPID_ICF = 
"org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String topicName = "MYTopic";
    TopicConnection topicConnection;
    TopicSession topicSession;
    public TopicSubscriber subscribe() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME,
        getTCPConnectionURL(userName, password));
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        TopicConnectionFactory connFactory = (TopicConnectionFactory)
        ctx.lookup(CF_NAME);
        topicConnection = connFactory.createTopicConnection();
        topicConnection.start();
        topicSession =
        topicConnection.createTopicSession(false, 
        TopicSession.AUTO_ACKNOWLEDGE);
    }
```
// Send message
Topic topic = topicSession.createTopic(topicName);
TopicSubscriber topicSubscriber =
topicSession.createSubscriber(topic);
return topicSubscriber;
}

public void receive(TopicSubscriber topicSubscriber) throws
NamingException, JMSException {
    Message message = topicSubscriber.receive();
    if (message instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message;
        System.out.println("Got message from topic subscriber = " +
        textMessage.getText());
    }
    // Housekeeping
    topicSubscriber.close();
topicSession.close();
topicConnection.stop();
topicConnection.close();
}

private String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT).append("'")
        .append(CARBON_DEFAULT_PORT).append("'")
This class defines the main method for calling both the clients mentioned above. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.naming.NamingException;
import javax.jms.TopicSubscriber;
public class Main {
    public static void main(String[] args) throws NamingException, JMSException {
        SampleTopicSubscriber topicSubscriber = new SampleTopicSubscriber();
        TopicSubscriber subscriber = topicSubscriber.subscribe();
        SampleTopicPublisher topicPublisher = new SampleTopicPublisher();
        topicPublisher.publishMessage();
        topicSubscriber.receive(subscriber);
    }
}
```

**Prerequisites**

See [Prerequisites to Run the MB Samples](#) for a list of prerequisites.

**Building the sample**

To run this sample, run the `ant` command from `<MB_HOME>/Samples/JMSTopicClient` directory.

**Analyzing the output**

Once you run this sample, you will see the following log in your console.

```
[java] Got message from topic subscriber = Test Message
```

To view the topic in the MB Management Console, log into the MB Management Console. Then click **Browse** under **Topics** in the **Main** tab. You will see a topic named **MYTopic** automatically created.

**Receiving Messages with JMS Message Listener**

This sample demonstrates how to receive messages asynchronously from a JMS queue or topic. The sample message listener will wait for messages to be received to each topic and queue.

- **About the sample**
- **Prerequisites**
- **Building the sample**
- **Analyzing the output**

**About the sample**
The `<MB_HOME>/Samples/JMSMessageListener/src/org/sample/jms` directory has the following classes.

- `MessageListenerClient.java`
- `SampleMessageListener.java`
- `Main.java`

This class is used to publish messages to a given queue and topic, and to listen to them asynchronously. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class MessageListenerClient {
    public static final String QPID_ICF = 
        "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory."
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String topicName = "foo.bar";
    String queueName = "queue";
    public void registerSubscribers() throws 
        NamingException, InterruptedException, JMSException {
        try {
            InitialContext ctx = initQueue();
            TopicConnectionFactory topicConnectionFactory =
                (TopicConnectionFactory) ctx.lookup("qpidConnectionfactory");
            TopicConnection topicConnection =
                topicConnectionFactory.createTopicConnection();
            topicConnection.start();
            TopicSession topicSession =
                topicConnection.createTopicSession(false,
            
```
Session.AUTO_ACKNOWLEDGE);
    Topic topic = (Topic) ctx.lookup(topicName);
    TopicSubscriber topicSubscriber =
        topicSession.createSubscriber(topic);
    topicSubscriber.setMessageListener(new
        SampleMessageListener(topicConnection, topicSession, topicSubscriber));
    publishMessagesToTopic();
    Thread.sleep(5000);
    QueueConnectionFactory connFactory = (QueueConnectionFactory)
        ctx.lookup(CF_NAME);
    QueueConnection queueConnection =
        connFactory.createQueueConnection();
    queueConnection.start();
    QueueSession queueSession =
        queueConnection.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
    Queue queue = (Queue) ctx.lookup(queueName);
    MessageConsumer queueReceiver =
        queueSession.createConsumer(queue);
    queueReceiver.setMessageListener(new
        SampleMessageListener(queueReceiver, queueSession, queueConnection));
    publishMessagesToQueue();

    } catch (NamingException e) {
        e.printStackTrace();
    } catch (JMSException e) {
        e.printStackTrace();
    }

private void publishMessagesToTopic() throws NamingException,
    JMSException, InterruptedException {
    InitialContext ctx = initQueue();
    TopicConnectionFactory tConnectionFactory =
        (TopicConnectionFactory)
            ctx.lookup("qpidConnectionfactory");
    TopicConnection tConnection =
        tConnectionFactory.createTopicConnection();
    tConnection.start();
    TopicSession tSession =
        tConnection.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
    Topic topic = (Topic) ctx.lookup(topicName);
    javax.jms.TopicPublisher topicPublisher =
        tSession.createPublisher(topic);
    for (int i = 0; i < 10; i++) {
        TextMessage topicMessage = tSession.createTextMessage("Topic Message - "+ (i + 1));
        topicPublisher.publish(topicMessage);
        Thread.sleep(1000);
    }
    tConnection.close();
    tSession.close();
    topicPublisher.close();
}
private void publishMessagesToQueue() throws NamingException, JMSException, InterruptedException {
    InitialContext ctx = initQueue();
    QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
    QueueConnection connection = connFactory.createQueueConnection();
    connection.start();
    QueueSession session = connection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
    Queue queue = (Queue) ctx.lookup(queueName);
    javax.jms.QueueSender queueSender = session.createSender(queue);
    for (int i = 0; i < 10; i++) {
        TextMessage queueMessage = session.createTextMessage("Queue Message - " + (i + 1));
        queueSender.send(queueMessage);
        Thread.sleep(1000);
    }
    connection.close();
    session.close();
    queueSender.close();
}

private InitialContext initQueue() throws NamingException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
    properties.put("queue." + queueName, queueName);
    properties.put("topic." + topicName, topicName);
    InitialContext ctx = new InitialContext(properties);
    return ctx;
}

private String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("brokerlist='tcp://'").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT).append(""")
        .toString();
This class is used as the message listener. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.QueueConnection;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;

public class SampleMessageListener implements javax.jms.MessageListener {
    private TopicConnection topicConnection;
    private TopicSession topicSession;
    private TopicSubscriber topicSubscriber;
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    private MessageConsumer queueReceiver;
    private int count = 0;
    public SampleMessageListener(MessageConsumer queueReceiver,
        QueueSession queueSession, QueueConnection queueConnection) {
        this.queueReceiver = queueReceiver;
        this.queueSession = queueSession;
        this.queueConnection = queueConnection;
        System.out.println("Starting Queue Listener....");
    }

    public SampleMessageListener(TopicConnection topicConnection,
        TopicSession topicSession, TopicSubscriber topicSubscriber) {
        this.topicConnection = topicConnection;
        this.topicSession = topicSession;
        this.topicSubscriber = topicSubscriber;
        System.out.println("Starting Topic Listener....");
    }

    /**
     * Override this method and add the operation which is needed to be done
     * when a message is arrived
     *
     * @param message - the next received message
     */
    @Override
    public void onMessage(Message message) {
        count++;
        TextMessage receivedMessage = (TextMessage) message;
        try {
            System.out.println("Got the message ==> " +
            receivedMessage.getText());
        } catch (JMSException e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```
receivedMessage.getText();
    if (count >= 10) {
        closeAll(receivedMessage);
    }
} catch (JMSException e) {
    e.printStackTrace();
}

private void closeAll(TextMessage receivedMessage) {
    try {
        if (receivedMessage.getText().contains("Queue")) {
            System.out.println("Closing Queue Listener............");
            queueConnection.close();
            queueSession.close();
            queueReceiver.close();
        } else {
            System.out.println("Closing Topic Listener............");
            topicConnection.close();
            topicSession.close();
            topicSubscriber.close();
        }
    } catch (JMSException e) {
        e.printStackTrace();
    }
}
This class defines the method for calling both the clients mentioned above. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.naming.NamingException;

public class Main {
    public static void main(String[] args) throws NamingException,
        JMSException, InterruptedException {
        MessageListenerClient messageListenerClient = new
            MessageListenerClient();
        messageListenerClient.registerSubscribers();
    }
}
```

**Prerequisites**

See [Prerequisites to Run the MB Samples](#) for a list of prerequisites.

**Building the sample**

Run the ant command from the `<MB_HOME>/Samples/JmsMessageListenerClient` directory.

**Analyzing the output**

You will get the following log in your console.
Receiving Messages from an MB Cluster

This sample demonstrates how to receive messages published to a queue in a particular MB instance can be received by a consumer of the same queue in another MB instance. In order to demonstrate this, two MB instances need to be run as follows.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Listener Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB1</td>
<td>5672</td>
</tr>
<tr>
<td>MB2</td>
<td>5673</td>
</tr>
</tbody>
</table>

In order to run two MB instances, you should do a Port Offset Configuration in `<MB_HOME>/repository/conf/carbon.xml`. Alternatively, use the following command to start one of the MB instances.

```
ws02server.sh -DportOffset=1
```

See the following topics for instructions:

**Prerequisites**

See Prerequisites to Run the MB Samples for a list of prerequisites.
Building the sample

The following classes need to be created for this sample.

- A client to make subscriptions to the queue named MyQueue at MB1. The code of this class should be as follows.

```java
import javax.jms.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class ConsumeClient {
    public void consumeMessage() {
        Properties initialContextProperties = new Properties();
        initialContextProperties.put("java.naming.factory.initial", "org.wso2.andes.jndi.PropertiesFileInitialContextFactory");
        String connectionString = "amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5672' ";
        initialContextProperties.put("connectionfactory.qpidConnectionFactory ", connectionString);
        initialContextProperties.put("queue.myQueue", "myQueue");
        try {
            InitialContext initialContext = new InitialContext(initialContextProperties);
            QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory) initialContext.lookup("qpidConnectionFactory");
            QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
            queueConnection.start();
            QueueSession queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
            Destination destination = (Destination) initialContext.lookup("myQueue");
            MessageConsumer messageConsumer = queueSession.createConsumer(destination);
            TextMessage textMessage = (TextMessage) messageConsumer.receive();
            System.out.println("Got message ==> " + textMessage.getText());
            try {
                Thread.sleep(9000);
            } catch (Exception e) {
                System.out.println(e);
            }
            messageConsumer.close();
            queueSession.close();
            queueConnection.stop();
            queueConnection.close();
        } catch (NamingException e) {
        }
    }
}
```
try {
    System.out.println("Consume Client Started. 
    
    ");
    ConsumeClient conClient = new ConsumeClient();
    conClient.consumeMessage(arg0);
    System.out.println("Message Consumed. ");
} catch (JMSException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}

public static void main(String[] args) {
    ConsumeClient sendConsumeClient = new ConsumeClient();
A client to publish messages to the queue named `MyQueue` at MB2. The code of this class should be as follows.

```java
import javax.jms.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

public class SendClient {
    public static void main(String[] args) {
        SendClient sendClient = new SendClient();
        sendClient.sendMessage();
    }

    public void sendMessage() {
        Properties initialContextProperties = new Properties();
        initialContextProperties.put("java.naming.factory.initial", "org.wso2.andes.jndi.PropertiesFileInitialContextFactory");
        String connectionString = "amqp://admin:admin@clientID/carbon?brokerlist='tcp://localhost:5673'",
        initialContextProperties.put("connectionfactory.qpidConnectionfactory", connectionString);
        initialContextProperties.put("queue.myQueue", "myQueue");
        try {
            InitialContext initialContext = new InitialContext(initialContextProperties);
            QueueConnectionFactory queueConnectionFactory = (QueueConnectionFactory)
            initialContext.lookup("qpidConnectionfactory");
            QueueConnection queueConnection = queueConnectionFactory.createQueueConnection();
            queueConnection.start();
            QueueSession queueSession = queueConnection.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
            TextMessage textMessage = queueSession.createTextMessage();
            textMessage.setText("Test message");
            System.out.println("Sending Message : " + textMessage.getText().length());
            // Send message
            Queue queue = (Queue) initialContext.lookup("myQueue");
            QueueSender queueSender = queueSession.createSender(queue);
        }
    }
}
```
queueSender.send(textMessage);
// Housekeeping
queueSender.close();
queueSession.close();
queueConnection.stop();
queueConnection.close();
} catch (NamingException e) {
    e.printStackTrace();
} catch (JMSException e) {
    e.printStackTrace();
Executing the sample

First run the message sender class to publish messages in the MyQueue queue at MB2. Then run the queue consumer class to consume messages published to the MyQueue queue from MB1.

Analyzing the output

It will be possible to view the message published in MB2 from MB3. You can check this in the output log of your console. Alternatively, you can check the queue contents of MyQueue in the Management Console of MB1.

JMS Selectors

This sample demonstrates how message selectors work when receiving messages from a JMS queue. It has a client which performs the role of a queue sender and another client which performs the role of a queue selector receiver. The queue selector receiver filters the messages it reads from the queue based on a selector string.

- About the sample
- Prerequisites
- Building the sample
- Analyzing the output

About the sample

The <MB_HOME>/Samples/JMSSelectors/src/org/sample/jms directory has the following classes:

- SampleQueueSelectorReceiver.java
- SampleQueueSender.java
- SelectorMainClass.java

This class defines a client that selects messages from the queue named testQueue based on a selector string. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.MessageConsumer;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
/**
 * This class contains methods and properties relate to Queue Receiver
 *(Subscriber)
 */
```
```java
public class SampleQueueSelectorReceiver {
    //JNDI Initial Context Factory. Don't change this
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";

    //Connection factory prefix
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    //Connection factory name
    private static final String CF_NAME = "qpidConnectionfactory";
    //Queue prefix
    private static final String QUEUE_NAME_PREFIX = "queue.";
    //username
    String userName = "admin";
    //password
    String password = "admin";
    //Client id it can be something
    private static String CARBON_CLIENT_ID = "carbon";
    //MB's Virtual host name should be match with this, default name is "carbon" can be configured
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    //IP Address of the host
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    //Standard AMQP port number
    private static String CARBON_DEFAULT_PORT = "5672";
    //Queue name
    String queueName = "testQueue";

    private QueueConnection queueConnection;
    private QueueSession queueSession;

    /**
     * Creating a Message Consumer Object
     * @param selector JMS Selector
     * @return Configured Message Consumer
     * @throws NamingException
     * @throws JMSException
     */
    public MessageConsumer registerSubscriber(String selector) throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        // Create a JMS connection
        queueConnection = connFactory.createQueueConnection();
        queueConnection.start();
        // Create JMS session object
        queueSession = queueConnection.createQueueSession(false,
```
QueueSession.AUTO_ACKNOWLEDGE);
    // Look up a JMS queue
    Queue queue = (Queue) ctx.lookup(queueName);
    // Create JMS consumer
    MessageConsumer consumer = queueSession.createConsumer(queue,
    selector, false);
    System.out.println("Starting Queue Listener....");
    System.out.println("JMS Selector : " + selector);
    return consumer;
}
/**
 * Receive Messages
 *
 * @param consumer Message consumer
 * @throws NamingException
 * @throws JMSException
 */
public void receiveMessages(MessageConsumer consumer) throws
    NamingException, JMSException {
    TextMessage message = (TextMessage) consumer.receive();
    System.out.println("Got message from queue receiver with conforming
to selectors ==>") + message.getText());
    // Housekeeping
    consumer.close();
    queueSession.close();
    queueConnection.stop();
    queueConnection.close();
}
/**
 * To construct Connection AMQP URL
 *
 * @param username username
 * @param password password
 * @return AMQP Connection URL
 */
private String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
    .append("amqp://").append(username).append(":").append(password)
    .append("@").append(CARBON_CLIENT_ID)
    .append("/").append(CARBON_VIRTUAL_HOST_NAME)
    .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
    .append(CARBON_DEFAULT_PORT)
    .append("'"
This class defines a client that publishes messages in the queue named `testQueue`. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * This class contains methods and properties relate to Queue Sender (Publisher)
 */
public class SampleQueueSender {
    //JNDI Initial Context Factory. Don't change this
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    //Connection factory prefix
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    //Connection factory name
    private static final String CF_NAME = "qpidConnectionfactory";
    //username
    String userName = "admin";
    //password
    String password = "admin";
    //Client id it can be something
    private static String CARBON_CLIENT_ID = "carbon";
    //MB's Virtual host name should be match with this, default name is "carbon" can be configured
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    //IP Address of the host
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    //Standard AMQP port number
    private static String CARBON_DEFAULT_PORT = "5672";
    //Queue prefix
    private static final String QUEUE_NAME_PREFIX = "queue.";
    //Queue name
    String queueName = "testQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;
    /**
     .toString();
     }
     }
```
public void sendMessages() throws NamingException, JMSException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME,
        getTCPConnectionURL(userName, password));
    properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
    InitialContext ctx = new InitialContext(properties);
    // Lookup connection factory
    QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
    // Create a JMS connection
    queueConnection = connFactory.createQueueConnection();
    queueConnection.start();
    // Create JMS session object
    queueSession = queueConnection.createQueueSession(false,
        QueueSession.AUTO_ACKNOWLEDGE);
    // Look up a JMS queue
    queue = (Queue) ctx.lookup(queueName);
    // Create the message to send
    System.out.println("Starting Queue Sender....");
    TextMessage textMessage = queueSession.createTextMessage("Test Message Content with properties LK & 1");
    // Create JMS String Property in text message
    textMessage.setStringProperty("Currency", "LK");
    // Create JMS Integer Property in text message
    textMessage.setIntProperty("quantity", 1);
    // Create JMS consumer
    javax.jms.QueueSender queueSender = queueSession.createSender(queue);
    queueSender.send(textMessage);
    System.out.println("Send Message from QueueSender : Currency = LK , Quantity = 1 ");
    // Send message
    // Create the message to send
    textMessage = queueSession.createTextMessage("Test Message Content with properties USD");
    textMessage.setStringProperty("Currency", "USD");
    queueSender.send(textMessage);
    System.out.println("Send Message from QueueSender : Currency = USD ");
    // Send message
    // Create the message to send
    textMessage = queueSession.createTextMessage("Test Message Content with properties LK & 4");
    textMessage.setStringProperty("Currency", "LK");
    textMessage.setIntProperty("quantity", 4);
    queueSender.send(textMessage);
    System.out.println("Send Message from QueueSender : Currency = LK , Quantity = 4 ");
}
Quantity = 4");
    // Send message
    // Create the message to send
    textMessage = queueSession.createTextMessage("Test Message Content
    with properties EUR");
    textMessage.setStringProperty("Currency", "EUR");
    queueSender.send(textMessage);
    System.out.println("Send Message from QueueSender : Currency = EUR
    ");
    // Send message
    // Create the message to send
    textMessage = queueSession.createTextMessage("Test Message Content
    with properties LK & 5");
    textMessage.setStringProperty("Currency", "LK");
    textMessage.setIntProperty("quantity", 5);
    queueSender.send(textMessage);
    System.out.println("Send Message from QueueSender : Currency = LK ,
    Quantity = 5
    ");
    // Send message
    // Create the message to send
    textMessage = queueSession.createTextMessage("Test Message Content
    with properties LK & 6");
    textMessage.setStringProperty("Currency", "LK");
    textMessage.setIntProperty("quantity", 6);
    System.out.println("Send Message from QueueSender : Currency = LK ,
    Quantity = 6
    ");
    queueSender.send(textMessage);
    // Send message
    // Create the message to send
    textMessage = queueSession.createTextMessage("Test Message Content
    without properties");
    queueSender.send(textMessage);
    queueSender.close();
    queueSession.close();
    queueConnection.close();
}
/**
 * To construct Connection AMQP URL
 * @param username username
 * @param password password
 * @return AMQP Connection URL
 */
private String getTCPConnectionURL(String username, String password) {
    //
    amqp:// {username}: {password} @carbon/carbon?brokerlist='tcp:// {hostname}:
    {port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
.append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
.append(CARBON_DEFAULT_PORT)
  .append("'")
This class defines the method to call both the clients mentioned above. The code of this class is as follows:

```java
package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;
/**
 * A message selector is a String that contains an expression.
 * The syntax of the expression is based on a subset of the SQL92 conditional expression syntax.
 * <p/>
 * JMS selectors work as filters
 */
public class SelectorMainClass {
    public static void main(String[] args) throws NamingException, JMSException {
        SampleQueueSelectorReceiver queueReceiver = new SampleQueueSelectorReceiver();
        //Message consumer with JMS Selector
        MessageConsumer consumer = queueReceiver.registerSubscriber("Currency ='LK' AND quantity < 3");
        SampleQueueSender queueSender = new SampleQueueSender();
        queueSender.sendMessages();
        queueReceiver.receiveMessages(consumer);
    }
}
```

**Prerequisites**

See [Prerequisites to Run the MB Samples](#) for a list of prerequisites.

**Building the sample**

Run the `ant` command from the `<MB_HOME>/samples/JMSSelectors` directory.

**Analyzing the output**

The following output log will appear in your console.
Sending and Receiving Messages with TTL

This sample demonstrates how the Time to Live(TTL) value can be set to messages which are published to WSO2 Message Broker. It uses a sample JMS client named QueueSender that will send messages with or without a TTL value for a queue in WSO2 Message Broker. It will then use a sample JMS client named QueueReceiver to receive the messages that are not expired at that time and print the number of received messages to the console.

- Prerequisites
- About the sample
- Building the sample
- Analyzing the output

Prerequisites

See Prerequisites to Run the MB Samples for a list of prerequisites.

About the sample

The <MB_HOME>/Samples/JmsExpirationSample/src/org/sample/jms directory has the following classes:

- SampleQueueSender.java
- SampleQueueReceiver.java
- Main.java

This class is used to create the sample client which sends messages to the queue named expirationTestQueue in WSO2 MB. The code of this class is as follows:

```java
package org.sample.jms;

import javax.jms.DeliveryMode;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
```
import javax.naming.NamingException;
import java.util.Properties;

/**
 * Sample sender to send the messages with/without TTL
 */
public class SampleQueueSender {

    public static final String QPID_ICF =
            "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String QUEUE_NAME_PREFIX = "queue.";
    private static final String CF_NAME = "qpidConnectionfactory";

    String userName = "admin";
    String password = "admin";

    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "expirationTestQueue";

    private QueueConnection queueConnection;
    private QueueSession queueSession;

    /**
     * Send the specified number of messages with the specified ttl.
     * @param noOfMessages Number of messages that need to be sent
     * @param timeToLive Time to live value for messages
     * @throws NamingException
     * @throws JMSException
     */
    public void sendMessages(int noOfMessages, long timeToLive) throws
                NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME,
                getTCPConnectionURL(userName, password));
        properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
        InitialContext ctx = new InitialContext(properties);
        // Lookup connection factory
        QueueConnectionFactory connFactory = (QueueConnectionFactory)
                ctx.lookup(CF_NAME);
        queueConnection = connFactory.createQueueConnection();
        queueConnection.start();
        queueSession = queueConnection.createQueueSession(false,
                QueueSession.AUTO_ACKNOWLEDGE);
        Queue queue = (Queue)ctx.lookup(queueName);
        // create the message to send
        TextMessage textMessage = queueSession.createTextMessage("Test
                Message Content");
        javax.jms.QueueSender queueSender =
                queueSession.createSender(queue);
        for(int i = 0; i < noOfMessages; i++){
//send the text message in persistent delivery mode with a time to
live value at priority level 4
    queueSender.send(textMessage,
        DeliveryMode.PERSISTENT, 4, timeToLive);
    }
    queueSender.close();
    queueSession.close();
    queueConnection.close();
}
/**
 * Creates amqp url.
 *
 * @param username The username for the amqp url.
 * @param password The password for the amqp url.
 * @return AMQP url.
 */
private String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT)
        .append("'"
This class is used to create the sample client which receives the message sent to the expirationTestQueue queue and prints it in the console. The code of this class is as follows:

```java
package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * Sample Receiver to receive the messages which were not expired
 */
public class SampleQueueReceiver {
    public static final String QPID_ICF = 
        "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "expirationTestQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;

    /**
     * Register Subscriber for a queue.
     * @return MessageConsumer The message consumer object of the subscriber.
     * @throws NamingException
     * @throws JMSException
     */
    public MessageConsumer registerSubscriber() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);

        return null;
    }
}
```
properties.put(CF_NAME_PREFIX + CF_NAME,
  getTCPConnectionURL(username, password));
properties.put("queue."+ queueName,queueName);
InitialContext ctx = new InitialContext(properties);
// Lookup connection factory
QueueConnectionFactory connFactory = (QueueConnectionFactory)
ctx.lookup(CF_NAME);
queueConnection = connFactory.createQueueConnection();
queueConnection.start();
queueSession = queueConnection.createQueueSession(false,
QueueSession.AUTO_ACKNOWLEDGE);
//Receive message
Queue queue = (Queue) ctx.lookup(queueName);
MessageConsumer consumer = queueSession.createConsumer(queue);
return consumer;
}
/**
 * Receive messages using the consumer.
 * @param consumer The message consumer object of the subscriber.
 * @throws NamingException
 * @throws JMSException
 */
public void receiveMessages(MessageConsumer consumer) throws
NamingException, JMSException {
  int receivedMessageCount = 0;
  //have 5 seconds as receive timeout value to stop the consumer
  while(null != consumer.receive(5000)){
    receivedMessageCount ++;
  }
  System.out.println("Received message count: "+
receivedMessageCount);
}
/**
 * Close the connections at the end of operation
 * @param consumer The message consumer object of the subscriber.
 * @throws JMSException
 */
public void closeConnections(MessageConsumer consumer) throws
JMSException{
  consumer.close();
  queueSession.close();
  queueConnection.stop();
  queueConnection.close();
}
/**
 * Creates amqp url.
 * @param username The username for the amqp url.
* @param password The password for the amqp url.
* @return AMQP url.
*/

private String getTCPConnectionURL(String username, String password) {
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
            .append("amqp://").append(username).append(":").append(password)
            .append("@").append(CARBON_CLIENT_ID)
            .append("/").append(CARBON_VIRTUAL_HOST_NAME)
            .append("?brokerlist='tcp://")
            .append(CARBON_DEFAULT_HOSTNAME).append(":")
            .append(CARBON_DEFAULT_PORT)
            .append("'")
The `Main.java` class defines the main method for calling both the clients mentioned above. The code of this class is as follows:

```java
package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;

/**<n
 * Sample executor class for message TTL
 */
public class Main {

    public static void main(String[] args) throws NamingException,
            JMSException {

        SampleQueueReceiver queueReceiver = new SampleQueueReceiver();
        MessageConsumer consumer = queueReceiver.registerSubscriber();

        //Send messages with very less time to live value
        System.out.println("Sending 5 messages with TTL value of 1sec");
        SampleQueueSender queueSenderWithTTL = new SampleQueueSender();
        queueSenderWithTTL.sendMessage(5, 1000);
        queueReceiver.receiveMessages(consumer);

        //send messages without time to live value
        System.out.println("Sending 5 messages without TTL");
        SampleQueueSender queueSenderWithoutTTL = new SampleQueueSender();
        queueSenderWithoutTTL.sendMessage(5, 0);
        queueReceiver.receiveMessages(consumer);

        //send messages with considerable time to live value
        System.out.println("Sending 5 messages TTL value of 10sec");
        SampleQueueSender queueSenderWithMediumTTL = new SampleQueueSender();
        queueSenderWithMediumTTL.sendMessage(5, 10000);
        queueReceiver.receiveMessages(consumer);

        //close the connection
        queueReceiver.closeConnections(consumer);
    }
}
```

**Building the sample**
Run the `ant` command from `<MB_HOME>/Samples/JMSQueueClient` directory.

Analyzing the output

You will get the following log in your console.

```
[java] Sending 5 messages with TTL value of 1sec
[java] Received message count: 0
[java] Sending 5 messages without TTL
[java] Received message count: 5
[java] Sending 5 messages TTL value of 10sec
[java] Received message count: 5
```

First, there were 5 messages published with a TTL value of 1sec and none of them got delivered since all of them were expired. Then, 5 more messages were sent without TTL and all of them got delivered. Finally, 5 messages were sent with a TTL value of 10 sec and all of them got delivered since they reached the recipient before the time of expiry.

Creating a Durable Topic Subscription

Durable topics keep messages persistently until a suitable consumer is available to consume them. Durable topic subscribers are used when an application needs to receive messages that are published even while the application is inactive. See Creating Durable Topic Subscriptions for more information.

- About the sample
- Prerequisites
- Executing the sample
- Analyzing the output

About the sample

The `<MB_HOME>/Samples/DurableTopicSubscriber/src/org/sample/jms` directory has the following classes.

- `DurableTopicSubscriber.java` class creates a durable topic subscription named `mySub1`.
- `SampleMessageListener.java` class creates a consumer for the durable topic subscription.
- `TopicPublisher.java` class creates a publisher to publish messages in the durable topic.
- `Main.java` defines the method for calling the three clients mentioned above.

Click the relevant tab to see the code.

- `DurableTopicSubscriber.java`
- `SampleMessageListener.java`
- `TopicPublisher.java`
- `Main.java`

```java
package org.sample.jms;
import javax.jms.*;
import javax.naming.Context;
import javax.naming.InitialContext;
```
import java.util.Properties;
public class DurableTopicSubscriber {
    public static final String ANDES_ICF = 
            "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "andesConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    private String topicName = "newTopic";
    private String subscriptionId = "mySub1";
    private boolean useListener = true;
    private int delayBetMessages = 200;
    private int messageCount = 10;
    private SampleMessageListener messageListener;
    private TopicConnection topicConnection;
    private TopicSession topicSession;
    private TopicSubscriber topicSubscriber;
    public void subscribe() {
        try {
            System.out.println("Starting the subscriber");
            Properties properties = new Properties();
            properties.put(Context.INITIAL_CONTEXT_FACTORY, ANDES_ICF);
            properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
            InitialContext ctx = new InitialContext(properties);
            // Lookup connection factory
            TopicConnectionFactory connFactory = (TopicConnectionFactory) ctx.lookup(CF_NAME);
            topicConnection = connFactory.createTopicConnection();
            topicConnection.start();
            topicSession =
                    topicConnection.createTopicSession(false, QueueSession.AUTO_ACKNOWLEDGE);
            // create durable subscriber with subscription ID
            Topic topic = (Topic) ctx.lookup(topicName);
            topicSubscriber = topicSession.createDurableSubscriber(topic, subscriptionId);
            if (!useListener) {
                for (int count = 0; count < messageCount; count++) {
                    Message message = topicSubscriber.receive();
                    System.out.println("count = " + count);
                    if (message instanceof TextMessage) {
                        TextMessage textMessage = (TextMessage) message;
                        System.out.println(count + ". textMessage.getText() = " + textMessage.getText());
                    }
                    if (delayBetMessages != 0) {
                        Thread.sleep(delayBetMessages);
                    }
                }
            }
        }
    }

public void stopSubscriber() throws JMSException {
    topicSubscriber.close();
    topicSession.close();
    topicConnection.close();

    public String getTCPConnectionURL(String username, String password) {
        return new StringBuffer()
            .append("amqp://")
            .append(username).append(":").append(password)
            .append("@").append(CARBON_CLIENT_ID)
            .append("/").append(CARBON_VIRTUAL_HOST_NAME)
            .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
            .append(CARBON_DEFAULT_PORT).append("'").append(""")
            .toString();
    }
package org.sample.jms;
import javax.jms.*;
public class SampleMessageListener implements MessageListener {
    private int delay = 0;
    private int currentMsgCount = 0;
    public SampleMessageListener(int delay) {
        this.delay = delay;
    }
    public void onMessage(Message message) {
        TextMessage receivedMessage = (TextMessage) message;
        try {
            System.out.println("Got the message ==> " + (currentMsgCount+1) + " - " + receivedMessage.getText());
            currentMsgCount++;
            if(delay != 0) {
                try {
                    Thread.sleep(delay);
                } catch (InterruptedException e) {
                    //silently ignore
                }
            }
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}

package org.sample.jms;
import javax.jms.JMSException;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;
public class TopicPublisher {
    public static final String ANDES_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory."
    private static final String CF_NAME = "andesConnectionfactory"
    String userName = "admin";
String password = "admin";
private static String CARBON_CLIENT_ID = "carbon";
private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
private static String CARBON_DEFAULT_HOSTNAME = "localhost";
private static String CARBON_DEFAULT_PORT = "5672";
String topicName = "newTopic";

public void publishMessage(int numOfMsgs) throws NamingException, JMSException, InterruptedException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, ANDES_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
    properties.put("topic."+topicName,topicName);
    InitialContext ctx = new InitialContext(properties);
    // Lookup connection factory
    TopicConnectionFactory connFactory = (TopicConnectionFactory) ctx.lookup(CF_NAME);
    TopicConnection topicConnection = connFactory.createTopicConnection();
    topicConnection.start();
    TopicSession topicSession = topicConnection.createTopicSession(false, QueueSession.AUTO_ACKNOWLEDGE);
    Topic topic = (Topic)ctx.lookup(topicName);
    // Create the messages to send
    TextMessage textMessage = topicSession.createTextMessage("Test Message");
    javax.jms.TopicPublisher topicPublisher = topicSession.createPublisher(topic);
    System.out.println("Sending " + numOfMsgs + " messages to Topic: " + topicName);
    for (int i = 0; i < numOfMsgs; i++)
    {
        topicPublisher.publish(textMessage);
        Thread.sleep(1000);
    }
    topicPublisher.close();
    topicSession.close();
    topicConnection.close();
}

public String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID).
        append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT).append(""");
Prerequisites

See Prerequisites to Run the MB Samples for a list of prerequisites.

Executing the sample

Run the ant command from the `<MB_Home>/samples/DurableTopicSubscriber` directory.

Analyzing the output

The scenario used in this sample to demonstrate durable topic subscriptions is as follows.

1. `durableTopicSubscriber` is run to create a durable topic subscriber.
2. 5 messages are sent to the `myTopic` topic. The messages will be received and printed by the subscriber named `durableTopicSubscriber`.
3. The `durableTopicSubscriber` is stopped.
4. The publisher is run again and 5 more messages are sent.
5. While running `durableTopicSubscriber` again, 5 different messages are sent to the same topic. You will see that all 10 messages (including the messages sent to the topic when the subscriber was absent) are
Consumed by the durableTopicSubscriber.

Creating Hierarchical Topic Subscriptions

This sample demonstrates how to publish messages to topics and subtopics in a topic hierarchy and to create hierarchical topic subscriptions.

- About the sample
- Prerequisites
- Executing the sample
- Analyzing the output

About the sample

The `<MB_HOME>/Samples/HierarchicalTopicsSubscriber/src/org/sample/jms` directory has the following classes:

- `SampleHierarchicalTopicsClient.java` class defines a client that subscribes to a hierarchical topic structure of which the main topic is Games.

- `TopicPublisher.java` class defines a client that publishes messages in the hierarchical topic structure mentioned above.

- `Main.java` class defines the method to call both the clients.

Click the relevant tab to see the code.

- `SampleHierarchicalTopicsClient.java`
- `TopicPublisher.java`
- `Main.java`

```java
package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.jms.TopicSubscriber;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

public class SampleHierarchicalTopicsClient extends Thread{
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
```
String userName = "admin";
String password = "admin";
private static String CARBON_CLIENT_ID = "carbon";
private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
private static String CARBON_DEFAULT_HOSTNAME = "localhost";
private static String CARBON_DEFAULT_PORT = "5672";
String topicName_1 = "Games";
String topicName_2 = "Games.Cricket";
String topicName_3 = "Games.Cricket.SL";
String topicName_4 = "Games.Cricket.India";
String topicName_5 = "Games.Cricket.India.Delhi";
String topicName_6 = "Games.Cricket.*";
String topicName_7 = "Games.Cricket.#";
private boolean isSubscriptionComplete = false;

@Override
public void run() {
    try {
        subscribe();
    } catch (NamingException e) {
        e.printStackTrace();
    } catch (JMSException e) {
        e.printStackTrace();
    }
}

public void subscribe() throws NamingException, JMSException {
    InitialContext ctx = init();
    // Lookup connection factory
    TopicConnectionFactory connFactory = (TopicConnectionFactory)
        ctx.lookup(CF_NAME);
    TopicConnection topicConnection =
        connFactory.createTopicConnection();
    topicConnection.start();
    //Create two topic sessions since a number of clients cannot be
    //connected from the same session
    TopicSession topicSession1 =
        topicConnection.createTopicSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
    TopicSession topicSession2 =
        topicConnection.createTopicSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
    Topic topic1 = topicSession1.createTopic(topicName_1);
    Topic topic2 = topicSession1.createTopic(topicName_2);
    Topic topic3 = topicSession1.createTopic(topicName_3);
    Topic topic4 = topicSession1.createTopic(topicName_4);
    Topic topic5 = topicSession1.createTopic(topicName_5);
    Topic topic6 = (Topic) ctx.lookup(topicName_6);
    Topic topic7 = (Topic) ctx.lookup(topicName_7);
    TopicSubscriber topicSubscriber1 =
topicSession1.createSubscriber(topic6);
TopicSubscriber topicSubscriber2 =
topicSession2.createSubscriber(topic7);

isSubscriptionComplete = true;
// Receive messages
Message message1;
System.out.println("Receiving messages for " + topicName_6 + ":");
while ((message1 = topicSubscriber1.receive(5000)) != null){
    if (message1 instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message1;
        System.out.println("Got Message from subscriber1 => " +
textMessage.getText());
    }
}

Message message2;
System.out.println("Receiving messages for " + topicName_7 + ":");
while ((message2 = topicSubscriber2.receive(5000)) != null){
    if (message2 instanceof TextMessage) {
        TextMessage textMessage = (TextMessage) message2;
        System.out.println("Got Message from subscriber2 => " +
textMessage.getText());
    }
}

topicSubscriber1.close();
topicSubscriber2.close();
topicSession1.close();
topicSession2.close();
topicConnection.stop();
topicConnection.close();

private InitialContext init() throws NamingException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, 
    getTCPConnectionURL(userName, password));
    properties.put("topic."+topicName_6,topicName_6);
    properties.put("topic."+topicName_7,topicName_7);
    return new InitialContext(properties);
}

private String getTCPConnectionURL(String username, String password) {
    //
    amqp://(username):{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
    .append("amqp://").append(username).append(":").append(password)
    .append("@").append(CARBON_CLIENT_ID)
    .append("/").append(CARBON_VIRTUAL_HOST_NAME)
.append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
.append(CARBON_DEFAULT_PORT).append("'")
    .toString();
}

public boolean isSubscriptionComplete(){


package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.jms.Topic;
import javax.jms.TopicConnection;
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicSession;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

public class TopicPublisher {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String topicName_1 = "Games";
    String topicName_2 = "Games.Cricket";
    String topicName_3 = "Games.Cricket.SL";
    String topicName_4 = "Games.Cricket.India";
    String topicName_5 = "Games.Cricket.India.Delhi";
    public void publishMessage() throws NamingException, JMSException {
        InitialContext ctx = init();
        // Lookup connection factory
        TopicConnectionFactory connFactory = (TopicConnectionFactory) ctx.lookup(CF_NAME);
        TopicConnection topicConnection = connFactory.createTopicConnection();
        topicConnection.start();
        TopicSession topicSession = topicConnection.createTopicSession(false,
                QueueSession.AUTO_ACKNOWLEDGE);
        Topic topic1 = (Topic) ctx.lookup(topicName_1);
        Topic topic2 = (Topic) ctx.lookup(topicName_2);
        Topic topic3 = (Topic) ctx.lookup(topicName_3);
        Topic topic4 = (Topic) ctx.lookup(topicName_4);
        return this.isSubscriptionComplete;
    }
}
Topic topic5 = (Topic) ctx.lookup(topicName_5);

javax.jms.TopicPublisher topicPublisher1 =
topicSession.createPublisher(topic1);
javax.jms.TopicPublisher topicPublisher2 =
topicSession.createPublisher(topic2);
javax.jms.TopicPublisher topicPublisher3 =
topicSession.createPublisher(topic3);
javax.jms.TopicPublisher topicPublisher4 =
topicSession.createPublisher(topic4);
javax.jms.TopicPublisher topicPublisher5 =
topicSession.createPublisher(topic5);

// Create the messages to send
TextMessage textMessage1 = topicSession.createTextMessage("Message for Games");
TextMessage textMessage2 = topicSession.createTextMessage("Message for Cricket");
TextMessage textMessage3 = topicSession.createTextMessage("Message for SL");
TextMessage textMessage4 = topicSession.createTextMessage("Message for India");
TextMessage textMessage5 = topicSession.createTextMessage("Message for Delhi");
topicPublisher1.publish(textMessage1);
topicPublisher2.publish(textMessage2);
topicPublisher3.publish(textMessage3);
topicPublisher4.publish(textMessage4);
topicPublisher5.publish(textMessage5);
topicSession.close();
topicConnection.close();

private InitialContext init() throws NamingException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
    properties.put("topic."+topicName_1,topicName_1);
    properties.put("topic."+topicName_2,topicName_2);
    properties.put("topic."+topicName_3,topicName_3);
    properties.put("topic."+topicName_4,topicName_4);
    properties.put("topic."+topicName_5,topicName_5);
    return new InitialContext(properties);
}

private String getTCPConnectionURL(String username, String password) {
    // amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
.append("@").append(CARBON_CLIENT_ID)
.append("/").append(CARBON_VIRTUAL_HOST_NAME)

.append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
.append(CARBON_DEFAULT_PORT).append("'")
```java
package org.sample.jms;

import javax.jms.JMSException;
import javax.naming.NamingException;
public class Main {
    public static void main(String[] args) throws NamingException, JMSException, InterruptedException {
        SampleHierarchicalTopicsClient hierarchicalTopicsClient = new SampleHierarchicalTopicsClient();
        hierarchicalTopicsClient.start();
        while (!hierarchicalTopicsClient.isSubscriptionComplete()){
            Thread.sleep(500);
        }
        TopicPublisher topicPublisher = new TopicPublisher();
        topicPublisher.publishMessage();
    }
}
```

### Prerequisites

See [Prerequisites to Run the MB Samples](#) for a list of prerequisites.

#### Executing the sample

Run the ant command from `<MB_HOME>/samples/HierarchicalTopicsSubscriber` directory.

#### Analyzing the output

When you run the sample, you will see the following in the output log in the console.

```
[java] Receiving messages for Games.Cricket.* :
[java] Receiving messages for Games.Cricket.# :
```

### CSharp Client Samples

This set of samples demonstrates the use of RabbitMQ .NET/C# client APIs with WSO2 Message Broker to publish and subscribe to messages from queues or topics.

- Publishing and Receiving Messages from a Queue
- Publishing and Receiving Messages from a Topic

### Publishing and Receiving Messages from a Queue


This sample demonstrates how persistent queues can be created and used in Message Broker using the RabbitMQ .NET/C# client. It first introduces a sample .NET client named QueuePublisher, which is used to publish messages to a known, created queue in WSO2 Message Broker. Then it introduces a sample .NET client named QueueConsumer to receive messages and print message content to the console.

- Prerequisites
- Building the sample
- Executing the sample

**Prerequisites**

In order to run this sample:

- Download and add the RabbitMQ.Client.dll file as a reference in your .NET project. You can download the file from http://www.rabbitmq.com/dotnet.html or from the WSO2 repository.
- See Prerequisites to Run the MB Samples for a list of other prerequisites.

**Building the sample**

1. Create a QueueConsumer .NET client to receive messages from the test-queue queue by adding a class with the following code.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RabbitMQ.Client;

namespace QueueConsumer
{
    class QueueConsumer
    {
        static void Main(string[] args)
        {
            QueueConsumer qConsumer = new QueueConsumer();
            qConsumer.GetMessage();
        }

        public void GetMessage()
        {
            //Setup the connection with the message broker
           ConnectionFactory factory = new ConnectionFactory();
            IProtocol protocol = Protocols.AMQP_0_9_1;
            factory.VirtualHost = "/carbon";
            factory.UserName = "admin";
            factory.Password = "admin";
            factory.HostName = "localhost";
            factory.Port = 5672;
            factory.Protocol = protocol;
            using (IConnection conn = factory.CreateConnection())
            {
                using (IModel ch = conn.CreateModel())
                {
```

```csharp
```
```csharp
    //Declare a queue to retrieve messages.
    ch.QueueDeclare("test-queue", true, false, false, null);
    //Create the binding between queue and the exchange
    ch.QueueBind("test-queue", "amq.direct", "test-queue");
    QueueingBasicConsumer consumer = new QueueingBasicConsumer(ch);
    ch.BasicConsume("test-queue", false, consumer);
    while (true)
    {
        try
        {
            RabbitMQ.Client.Events.BasicDeliverEventArgs e =
            (RabbitMQ.Client.Events.BasicDeliverEventArgs)consumer.Queue.Dequeue();
            byte[] body = e.Body;
            string message = Encoding.UTF8.GetString(body);
            Console.WriteLine("Received Message : " + message);
            ch.BasicAck(e.DeliveryTag, false);
        }
        catch (OperationCanceledException e)
        {
            Console.WriteLine(e);
            break;
        }
    }
```
Create a QueuePublisher .NET client to send messages to the test-queue queue by adding a class with the following code.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RabbitMQ.Client;

namespace RabbitMQ
{
    class QueuePublisher
    {
        static void Main(string[] args)
        {
            QueuePublisher publisher = new QueuePublisher();
            publisher.PublishMessage("This is a Test Message");
            Console.WriteLine("MessageSent");
            Console.ReadLine();
        }

        public void PublishMessage(string message)
        {
            // Setup the connection with the message broker
            ConnectionFactory factory = new ConnectionFactory();
            IProtocol protocol = Protocols.AMQP_0_9_1;
            factory.VirtualHost = "/carbon";
            factory.UserName = "admin";
            factory.Password = "admin";
            factory.HostName = "localhost";
            factory.Port = 5672;
            factory.Protocol = protocol;
            using (IConnection conn = factory.CreateConnection())
            {
                using (IModel ch = conn.CreateModel())
                {
                    IBasicProperties basicProperties = ch.CreateBasicProperties();
                    // Setting JMS Message ID.
                    // At least one QueueConsumer binding should exist before sending messages to the queue. Therefore, this QueueConsumer class should be run before the QueuePublisher class. Alternatively, you can manually create the test-queue queue in the MB Management Console. See Adding Queues for detailed instructions.
                }
            }
        }
    }
}
```
  // Setting content-type for message as we are sending a text message.
  basicProperties.ContentType = "text/plain";
  // Declare the exchange for the publisher. Here the exchange type is direct.
  ch.ExchangeDeclare("amq.direct", "direct");
  // Publish the message
}
Executing the sample

Run this sample from your C# project.

Publishing and Receiving Messages from a Topic

This sample demonstrates how durable or non-durable topics can be created and used in WSO2 Message Broker using the RabbitMQ .NET/C# client. It first introduces a sample .NET client named TopicPublisher, that publishes messages to a known, created topic in Message Broker. Then it introduces a sample .NET client named TopicConsumer that listens for messages and prints message contents to the console.

- Prerequisites
- Building the sample
- Executing the sample

Prerequisites

To run this sample:

- Download and add the RabbitMQ.Client.dll file as a reference in your .NET project. You can download this file from http://www.rabbitmq.com/dotnet.html or the WSO2 repository.
- See Prerequisites to Run the MB Samples for a list of other prerequisites.

Building the sample

1. Create a TopicConsumer .NET client to receive messages from the test-topic topic by adding a class with the following code.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RabbitMQ.Client;

namespace MB_TopicClient
{
    class TopicConsumer
    {
        static void Main(string[] args)
        {
            TopicConsumer topicConsumer = new TopicConsumer();
            topicConsumer.GetMessages();
        }

        public void GetMessages()
        {
            //Setup the connection with the message broker
            ConnectionFactory factory = new ConnectionFactory();
```
IProtocol protocol = Protocols.AMQP_0_9_1;
factory.VirtualHost = "/carbon";
factory.UserName = "admin";
factory.Password = "admin";
factory.HostName = "localhost";
factory.Port = 5672;
factory.Protocol = protocol;
using (IConnection conn = factory.CreateConnection())
{
    using (IModel ch = conn.CreateModel())
    {
        // Declare a topic exchange to be bound to retrieve messages, here we have used the default topic exchange of WSO2 MB
        ch.ExchangeDeclare("amq.topic", "topic");
        // Declare a topic name, here we use a non-durable topic. To make it durable use the 2nd parameter as 'true'
        ch.QueueDeclare("test-topic", false, false, false, null);
        // Bind the Topic in to the exchange
        ch.QueueBind("test-topic", "amq.topic", "test-topic");
        // Declare a consumer which listens on the messages published to 'test-topic' topic, we need to declare an exclusive subscriber, in order to get this work.
        // The syntax is BasicConsume(<queueName>, <noAck>, <consumerTag>, <noLocal>, <exclusive>, <arguments>, <Consumer>)
        QueueingBasicConsumer consumer = new QueueingBasicConsumer(ch);
        ch.BasicConsume("test-topic", false, "1", false, true, null, consumer);
        while (true)
        {
            try
            {
                byte[] body = e.Body;
                string message = Encoding.UTF8.GetString(body);
                Console.WriteLine("Received Message : " + message);
                ch.BasicAck(e.DeliveryTag, false);
            }
            catch (OperationCanceledException e)
            {
                Console.WriteLine(e);
                break;
            }
        }
    }
}
Create a client to send messages to the topic by adding a class with the following code.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using RabbitMQ.Client;

namespace MB_Topic_Publisher
{
    class TopicPublisher
    {
        static void Main(string[] args)
        {
            TopicPublisher topicPublisher = new TopicPublisher();
            topicPublisher.PublishMessage("Test Message");
            Console.WriteLine("Message Sent..");
            Console.ReadLine();
        }

        public void PublishMessage(string message)
        {
            //Setup the connection with the message broker
            ConnectionFactory factory = new ConnectionFactory();
            IProtocol protocol = Protocols.AMQP_0_9_1;
            factory.VirtualHost = "/carbon";
            factory.UserName = "admin";
            factory.Password = "admin";
            factory.HostName = "localhost";
            factory.Port = 5672;
            factory.Protocol = protocol;
            using (IConnection conn = factory.CreateConnection())
            {
                using (IModel ch = conn.CreateModel())
                {
                    // Declare a topic exchange to publish messages,
                    // here we have used the default topic exchange of WSO2 MB
                    ch.ExchangeDeclare("amq.topic", "topic");
                    IBasicProperties basicProperties =
                    ch.CreateBasicProperties();
                }
            }
        }
    }
}
```

At least one TopicConsumer binding should exist before sending messages to the topic. Therefore, this TopicConsumer class should be run before the TopicPublisher class. Alternatively, you can manually create the test-topic topic in the MB Management Console. See Adding Topics for detailed instructions.

2. Create a TopicPublisher .NET client to send messages to the test-topic topic by adding a class with the following code.
// Setting JMS Message ID.
    basicProperties.MessageId = "ID:" +
    System.Guid.NewGuid().ToString();
// Setting content-type for message as we are sending a text message.
    basicProperties.ContentType = "text/plain";
// Publish the message to the exchange, it will send it to the routing key which is our name 'myTopic'.
    // The syntax is ch.BasicPublish(<exchange_name>,
    // <topic_name>, <message_properties>,<message_body>)
    ch.BasicPublish("amq.topic", "test-topic",
    basicProperties, Encoding.UTF8.GetBytes(message));
} 
}
3. Add a Main.java class defining the method to call both the classes mentioned above.

**Executing the sample**

Run this sample from your C# project.

**Using MQTT Transport**

This section includes the following samples demonstrating how the MQTT transport can be used in real life scenarios:

- Simple MQTT Client
- MQTT Chat
- MQTT IoT
- MQTT Retain

**Simple MQTT Client**

This sample demonstrates how to send and receive messages in WSO2 Message broker via the MQTT transport.

- About the sample
- Prerequisites
- Building the sample
- Analyzing the output

**About the sample**

The `<MB_HOME>/Samples/SimpleMqttClient/src/main/java/org/wso2/sample/mqtt` directory has the following files:

- SimpleMQTTCallback.java
- QualityOfService.java
- Main.java

This file defines the callback handler client which handles the messages returned from MB and prints them in the output log of the console. The Callback handler handles messages returned from the Message Broker. These messages are categorized into 3 types as follows:

- Connection Lost
- Message Arrived
- Delivery Complete

The configuration of this class is as follows:
package org.wso2.sample.mqtt;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;

/**
* The MQTT client callback handler which handles message arrivals,
delivery completions and connection lost.
*/
public class SimpleMQTTCallback implements MqttCallback {
    private static final Log log = LogFactory.getLog(SimpleMQTTCallback.class);
    /**
     * Inform when connection with server is lost.
     * @param throwable Connection lost cause
     */
    @Override
    public void connectionLost(Throwable throwable) {
        log.error("Mqtt client lost connection with the server", throwable);
    }
    /**
     * Inform when a message is received through a subscribed topic.
     * @param topic The topic message received from
     * @param mqttMessage The message received
     * @throws Exception
     */
    @Override
    public void messageArrived(String topic, MqttMessage mqttMessage) throws Exception {
        log.info("Message arrived on topic : " + topic + " Message : \\
" + mqttMessage.toString() + \\
"");
    }
    /**
     * Inform when message delivery is complete for a published message.
     * @param iMqttDeliveryToken The message complete token
     */
    @Override
    public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
        for (String topic : iMqttDeliveryToken.getTopics()) {
            log.info("Message delivered successfully to topic : " + topic + \\
".");
        }
    }
}
This class defines the MQTT Quality of Service levels that need to be applied. The configuration of this class is as follows:

```java
package org.wso2.sample.mqtt;

/**
 * The quality of service levels in MQTT.
 */
public enum QualityOfService {

    /**
     * The message is delivered at most once, or it may not be delivered at all. Its delivery across the network is not acknowledged. The message is not stored. The message could be lost if the client is disconnected, or if the server fails. QoS0 is the fastest mode of transfer. It is sometimes called "fire and forget".
     */
    MOST_ONCE(0),

    /**
     * The message is always delivered at least once. It might be delivered multiple times if there is a failure before an acknowledgment is received by the sender. The message must be stored locally at the sender, until the sender receives confirmation that the message has been published by the receiver. The message is stored in case the message must be sent again.
     */
    LEAST_ONCE(1),

    /**
     * The message is always delivered exactly once. The message must be stored locally at the sender, until the sender receives confirmation that the message has been published by the receiver. The message is stored in case the message must be sent again. QoS2 is the safest, but slowest mode of transfer. A more sophisticated handshaking and acknowledgement sequence is used than for QoS1 to ensure no duplication of messages occurs.
     */
    EXACTLY_ONCE(2);

    private final int qos;

    private QualityOfService(int qos) {
        this.qos = qos;
    }

    /**
     * Get the corresponding value for the given quality of service.
     * Retrieve this value whenever quality of service level needs to feed into external libraries.
     *
     * @return The integer representation of this quality of service
     */
    }
```
*/
public int getValue() {
This class defines the method to be used for running both the classes mentioned above.

```java
package org.wso2.sample.mqtt;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
/**
 * This samples demonstrates how to write a simple MQTT client to send/receive message via MQTT in WSO2 Message Broker.
 */
public class Main {
    private static final Log log = LogFactory.getLog(Main.class);
    // Java temporary directory location
    private static final String JAVA_TMP_DIR = System.getProperty("java.io.tmpdir");
    // The MQTT broker URL
    private static final String brokerURL = "tcp://localhost:1883";

    /**
     * The main method which runs the sample.
     *
     * @param args Commandline arguments
     */
    public static void main(String[] args) {
        String subscriberClientId = "subscriber";
        String publisherClientId = "publisher";
        String topic = "simpleTopic";
        boolean retained = false;
        log.info("Running sample");
        byte[] payload = "hello".getBytes();
        try {
            // Creating mqtt subscriber client
            MqttClient mqttSubscriberClient = getNewMqttClient(subscriberClientId);
            // Creating mqtt publisher client
            MqttClient mqttPublisherClient = getNewMqttClient(publisherClientId);
            // Subscribing to mqtt topic "simpleTopic"
            mqttSubscriberClient.subscribe(topic,
                QualityOfService.LEAST_ONCE.getValue());
            // Publishing to mqtt topic "simpleTopic"
            mqttPublisherClient.publish(topic, payload,
                QualityOfService.LEAST_ONCE.getValue(), retained);
            mqttPublisherClient.disconnect();
        }
    }
}
```
mqttSubscriberClient.disconnect();
log.info("Clients Disconnected!");
}
catch (MqttException e) {
log.error("Error running the sample", e);
}

/**
 * Create a new MQTT client and connect it to the server.
 * @param clientId The unique mqtt client Id
 * @return Connected MQTT client
 * @throws MqttException
 */
private static MqttClient getNewMqttClient(String clientId) throws MqttException {
    //Store messages until server fetches them
    MqttDefaultFilePersistence dataStore = new MqttDefaultFilePersistence(JAVA_TMP_DIR + "/" + clientId);
    MqttClient mqttClient = new MqttClient(brokerURL, clientId, dataStore);
    SimpleMQTTCallback callback = new SimpleMQTTCallback();
    mqttClient.setCallback(callback);
    MqttConnectOptions connectOptions = new MqttConnectOptions();
    connectOptions.setUserName("admin");
    connectOptions.setPassword("admin".toCharArray());
    connectOptions.setCleanSession(true);
    mqttClient.connect(connectOptions);
return mqttClient;
}
}

**Prerequisites**

Before you build the sample, the prerequisites for MB samples should be in place.

**Building the sample**

If you are building an MQTT sample for the first time, you need to build the sample using Maven. This will download the Maven dependencies needed for your MQTT samples. Once the dependencies are downloaded, you can build any of the MQTT samples using either Maven or Ant.

**Using Maven:**

1. Navigate to the SimpleMqttSample sample folder in the `<MB_HOME>/samples` directory.
2. Execute the `mvn clean install` command to build and run the sample.
3. If the build is successful, the output will be printed in the terminal as shown below.

**Using Ant:**

Be sure that the Maven dependencies required for MQTT samples are already downloaded as explained here.

1. Navigate to the SimpleMqttSample sample folder in the `<MB_HOME>/samples` directory.
2. Execute the `ant` command to build and run the SimpleMqttSample sample.
3. If the build is successful, the output will be printed in the terminal as shown below.

Run the `ant` or the `mvn clean install` command from the `<MB_HOME>/samples/SimpleMQTTClient` directory.

**Analyzing the output**

When you run this sample, the following will be displayed in the output log of the console depending on the command you used:

For the `ant` command:

```
```

For the `mvn clean install` command:
INFO [org.wso2.sample.mqtt.Main] - Running sample
INFO [org.wso2.sample.mqtt.Main] - Clients Disconnected!

MQTT Chat

This sample demonstrates how WSO2 MB can be used to create a chat client that uses MQTT.

- About the sample
- Prerequisites
- Building the sample
- Analyzing the output

About the sample

The `<MB_HOME>/Samples/MqttChatClient/src/main/java/org/wso2/sample/mqtt` directory has the following classes:

- AndesMQTTClient.java
- ChatClient.java
- ChatWindow.java
- Main.java

This class holds a basic MQTT client. It also implements the Callback handler for this client. The code is as follows:

```java
package org.wso2.sample.mqtt;
import org.eclipse.paho.client.mqttv3.*;
import java.io.File;
/**
 * The MQTT clients which is used by the chat client to send/receive messages.
 */
public class AndesMQTTClient implements MqttCallback {
    /**
     * The Message Broker URL
     */
    private static final String brokerURL = "tcp://localhost:1883";
    /**
     * The temporary directory for mqtt client to work with
     */
    private static final String tmpDir = System.getProperty("java.io.tmpdir");
    /**
     * The MQTT client which is used to communicate with the server
     */
    private MqttClient mqttClient;
    /**
     * The unique MQTT client Id
     */
```
private final String clientId;

/**
 * Credentials to be used when connecting to MQTT server
 */
private static final String DEFAULT_USER_NAME = "admin";

private static final String DEFAULT_PASSWORD = "admin";

/**
 * Create a new MQTT client with the given client Id. Return after the connection is successful.
 */
public AndesMQTTClient(String clientId) throws MqttException {
    this.clientId = clientId;
    MqttConnectOptions options = new MqttConnectOptions();
    options.setCleanSession(true);
    options.setUserName(DEFAULT_USER_NAME);
    options.setPassword(DEFAULT_PASSWORD.toCharArray());
    mqttClient = new MqttClient(brokerURL, clientId, new MqttDefaultFilePersistence(tmpDir + File.separator + clientId));
    mqttClient.setCallback(this);
    mqttClient.connect(options);
}

/**
 * Subscribe to a given topic in given qos. Return after the subscription is complete.
 */
public void subscribe(String topic, int qos) throws MqttException {
    mqttClient.subscribe(topic, qos);
}

/**
 * Un-subscribe from a given topic after publishing to the chat server that this client is going to leave the given chat.
 */
public void unsubscribe(String topic) throws MqttException {
    mqttClient.publish(topic, (clientId + " has left the conversation").getBytes(), 2, false);
/**
 * Send message to a given topic.
 * 
 * @param topic   The topic to send message to
 * @param message The message to send
 * @param qos     The quality of service
 * @throws MqttException
 */
public void sendMessage(String topic, String message, int qos) throws MqttException {
    String encodedMessage = ChatWindow.encodeMessage(clientId, message);
    mqttClient.publish(topic, encodedMessage.getBytes(), qos, false);
}
/**
 * Disconnect the mqtt client from the server.
 * 
 * @throws MqttException
 */
public void disconnect() throws MqttException {
    mqttClient.disconnect();
}
/**
 * Handle if connection is lost with the server.
 * 
 * @param throwable Cause
 */
@Override
public void connectionLost(Throwable throwable) {
    ChatWindow.outputToChatWindow("Connection lost");
}
/**
 * Handle receiving a message from the server.
 * 
 * @param topic       The topic message received from
 * @param mqttMessage The received message
 * @throws Exception
 */
@Override
public void messageArrived(String topic, MqttMessage mqttMessage) throws Exception {
    synchronized (this.getClass()) {
        String chatFrom = null;
        // If message is received through the personal channel it is a
        // personal message. Otherwise it is a group
        // chat
        if (!clientId.equals(topic)) {
            chatFrom = topic;
        }
        ChatWindow.decodeAndOutputMessage(chatFrom, mqttMessage.toString());
*/

* On delivery complete notify it to the chat console.
*
* @param iMqttDeliveryToken Delivery information token
*/

@Override
public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
    synchronized (this.getClass()) {
        for (String topic : iMqttDeliveryToken.getTopics()) {
            String chatName = null;
            if (!topic.equals(clientId)) {
                chatName = topic;
            }
            ChatWindow.decodeAndOutputMessage(chatName, "Message Sent");
        }
    }
}
This class holds an AndesMQTTClient that subscribes/publishes via MQTT. The code is as follows:

```java
package org.wso2.sample.mqtt;
import org.eclipse.paho.client.mqttv3.MqttException;
/**
 * Represents a chat client which hosts a mqtt client.
 */
public class ChatClient {
    // For a chat client messages should be received exactly once, which is qos 2 in MQTT.
    private static final int qos = 2;
    private AndesMQTTClient mqttClient;
    /**
     * Create a chat client an initialises a mqtt client on it's name.
     *
     * @param name The name of the chat client
     * @throws MqttException
     */
    public ChatClient(String name) throws MqttException {
        mqttClient = new AndesMQTTClient(name);
        mqttClient.subscribe(name, qos);
    }
    /**
     * Start/Join a group chat.
     *
     * @param groupName The group name
     * @throws MqttException
     */
    public void startGroupConversation(String groupName) throws MqttException {
        mqttClient.subscribe(groupName, qos);
        ChatWindow.outputToChatWindow("Joined to the group : " + groupName);
    }
    /**
     * Leave a group chat.
     *
     * @param groupName The group name
     * @throws MqttException
     */
    public void endGroupConversation(String groupName) throws MqttException {
        mqttClient.unsubscribe(groupName);
        ChatWindow.outputToChatWindow("Left the group : " + groupName);
    }
}
```
* Send a chat message.
* @param chatName The person/group to send message to
  * @param message The message
  * @throws MqttException
  */
  public void sendMessage(String chatName, String message) throws MqttException {
    mqttClient.sendMessage(chatName, message, qos);
  }

/**
 * Close the chat client.
 * @throws MqttException
 */
  public void closeClient() throws MqttException {
This class represents a chat window in a chat application. The code is as follows:

```java
package org.wso2.sample.mqtt;
import org.eclipse.paho.client.mqttv3.MqttException;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.Scanner;
/**
 * Represents a chat console.
 */
public final class ChatWindow {
    /**
     * The new line character
     */
    private static final String NEW_LINE = "\n";
    /**
     * Message header and content separating string
     */
    private static final String SEPARATOR = "::";
    /**
     * Scanner to read user input
     */
    private static final Scanner scanner = new Scanner(System.in);
    /**
     * Output StreamWriter to write to the console
     */
    private static final BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(System.out));
    /**
     * The delimiter to separate each keyword in a user input command
     */
    private static final String COMMAND_DELIMITER = " ";
    /**
     * The command to exit
     */
    private static final String EXIT_COMMAND = "exit";
    /**
     * The command keyword to join a group chat
     */
    private static final String JOIN_GROUP_COMMAND = "join";
    /**
     * The command keyword to leave a group chat
     */
    private static final String LEAVE_GROUP_COMMAND = "leave";
    public void startChatWindow() {
        String command = makeCommandFromUserInput();
        switch (command) {
            case EXIT_COMMAND:
                System.out.println("Exiting the chat console.");
                break;
            case JOIN_GROUP_COMMAND:
                System.out.println("Joining the group chat.");
                break;
            case LEAVE_GROUP_COMMAND:
                System.out.println("Leaving the group chat.");
                break;
            default:
                System.out.println("Invalid command.");
        }
    }
    private String makeCommandFromUserInput() {
        String command = scanner.nextLine();
        String[] commandArray = command.split(COMMAND_DELIMITER);
        return commandArray[0];
    }
}
```
private static final String HELP_COMMAND = "help";
/**
 * The command keyword to get help
 */

private static final String HELP_STRING = "Use <alias/group message> to chat to a desired group or a person" +
    NEW_LINE + "<join group_name> to join a group chat" + NEW_LINE + "<leave group_name> to leave a group chat"
    + NEW_LINE + "<exit> to exit" + NEW_LINE;
/**
 * Print a given message to the chat window console
 */
public static void outputToChatWindow(String message) {
    try {
        writer.write(">"+ message + NEW_LINE);
        writer.flush();
    } catch (IOException ignore) {
        // Silently ignore since there is no other way than this method itself to print to the output
    }
}

public static String getInputFromChatWindow() {
    return scanner.nextLine();
}
/**
 * Decode a given message and output to the chat window console.
 * This is invoked when a new message is received to the chat client.
 */
public static void decodeAndOutputMessage(String chatName, String message) {
    StringBuilder output = new StringBuilder();
    if (null == chatName) {
        output.append("Personal message ");
    } else {
        output.append("chat with ").append(chatName).append(NEW_LINE);
    }
    String decoder[] = message.split(SEPARATOR);
    if (decoder.length == 1) { // Info message
        output.append("Info : ").append(decoder[0]);
    } else if (decoder.length == 2) { // chat message
        output.append("from ").append(decoder[0]).append(NEW_LINE).append(decoder[1]);
    } else {
        output.append("Invalid message received from the server.");
    }
output.append(NEW_LINE).append("Waiting for your input. Use <help> for more info").append(NEW_LINE);
    outputToChatWindow(output.toString());
}
/**
 * Encode a given message with sender::message.
 * @param sender  The message sender Id
 * @param message The message to send
 * @return The encoded message
 */
public static String encodeMessage(String sender, String message) {
    return sender + SEPARATOR + message;
}
/**
 * Request and read user input from console giving a message to specify the request.
 * @param message The input request message
 * @return User input line
 */
public static String getInput(String message) {
    ChatWindow.outputToChatWindow(message);
    return getInputFromChatWindow();
}
/**
 * Directly read user input from the console. Use when user has already been notified about what to input.
 * @return User input line
 */
public static String getInput() {
    return getInputFromChatWindow();
}
/**
 * Process a given user input and take actions accordingly.
 * - Set exit flag
 * - Send messages
 * - Join a group conversation
 * - Leave a group conversation
 * @param input      The user input line
 * @param chatClient The mqtt client to use when
 * @return Running condition
 * @throws MqttException
 */
public static boolean processInput(String input, ChatClient chatClient)
    throws MqttException {
    boolean running = true;
    if (EXIT_COMMAND.equalsIgnoreCase(input)) {
        running = false;
    } else if (HELP_COMMAND.equalsIgnoreCase(input)) {

printHelper();
} else {
  String[] inputArgs = input.split(COMMAND_DELIMITER, 2);
  int argsLength = inputArgs.length;
  if (2 == argsLength) {
    String arg1 = inputArgs[0];
    String arg2 = inputArgs[1];
    if (JOIN_GROUP_COMMAND.equalsIgnoreCase(arg1)) {
      chatClient.startGroupConversation(arg2);
    } else if (LEAVE_GROUP_COMMAND.equalsIgnoreCase(arg1)) {
      chatClient.endGroupConversation(arg2);
    } else {
      chatClient.sendMessage(arg1, arg2);
    }
  } else {
    outputToChatWindow("Incorrect command.");
    printHelper();
  }
}
return running;
/**
 * Print the help string to the output window.
 */
public static void printHelper() {
This class defines the method to call the other classes. The code is as follows:

```java
outputToChatWindow(HELP_STRING);
}
}
```
package org.wso2.sample.mqtt;
import org.eclipse.paho.client.mqttv3.MqttException;
import java.util.concurrent.TimeUnit;
/**
 * This sample demonstrates how to use WSO2 Message Broker to create a chat client which uses MQTT.
 * <p/>
 * The Main class which executes the sample.
 * - Creates several chat clients
 * - Initiates personal conversations
 * - Initiates group conversations
 */
public class Main {
    private static ChatClient chatClient;
    private static boolean running = true;
    /**
     * The main method which invokes the sample.
     * - Creates a chat client
     * - Takes user input
     * @param args Command line arguments
     * @throws MqttException
     * @throws InterruptedException
     */
    public static void main(String[] args) throws MqttException, InterruptedException {
        String alias = ChatWindow.getInput("Please enter your chat alias : ");
        chatClient = new ChatClient(alias);
        ChatWindow.printHelper();
        while (running) {
            String input = ChatWindow.getInput();
            running = ChatWindow.processInput(input, chatClient);
            TimeUnit.SECONDS.sleep(1L);
        }
        disconnect();
    }
    /**
     * Disconnect all the chat clients from the server.
     * @param args Command line arguments
     * @throws MqttException
     */
    private static void disconnect() throws MqttException {
        chatClient.closeClient();
    }
}
Before you build the sample, the **prerequisites for MB samples** should be in place.

**Building the sample**

If you are building an MQTT sample for the first time, you need to build the sample using Maven. This will download the Maven dependencies needed for your MQTT samples. Once the dependencies are downloaded, you can build any of the MQTT samples using either Maven or Ant.

**Using Maven:**

1. Navigate to the MqttChatClient sample folder in the `<MB_HOME>/samples` directory.
2. Execute the `mvn clean install` command to build and run the sample.
3. If the build is successful, you can analyze the output as shown below.

**Using Ant:**

Be sure that the Maven dependencies required for MQTT samples are already downloaded as explained here.

1. Navigate to the MqttChatClient sample folder in the `<MB_HOME>/samples` directory.
2. Execute the `ant` command to build and run the `mqtt retain` sample.
3. If the build is successful, you can analyze the output as shown below.

**Analyzing the output**

Once you run the sample, you can carry out the following activities.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending personal messages or group messages.</td>
<td>name/group_name message to send</td>
</tr>
<tr>
<td>Joining a group chat</td>
<td>join group_name</td>
</tr>
<tr>
<td>Leaving a group chat</td>
<td>join group_name</td>
</tr>
<tr>
<td>Exiting the sample</td>
<td>exit</td>
</tr>
</tbody>
</table>

**Example**

1. Run two instances of this sample by running the `ant` or the `mvn clean install` command from the `<MB_HOME>/samples/MqttChatClient` directory in two consoles.
2. You will be requested for a chat alias in both instances. Enter `abc` in one terminal and `def` in the other. The following would appear in the log of both instances.

   ```java
   [java] >Use <alias/group message> to chat to a desired group or a person
   [java] <join group_name> to join a group chat
   [java] <leave group_name> to leave a group chat
   [java] <exit> to exit
   ```

3. Enter the following command in the terminal of the `abc` chat alias.
3. Connect both chat aliases to a group chat using the following command.

   join Chat1Group

   You will get the following log for both instances.

   [java] >Joined to the group : Chat1Group

4. Enter the command `leave Chat1Group` in the terminal for ABC chat alias. The following would appear in the log.

   [java] >Left the group : Chat1Group

5. Type `exit` in the same terminal to stop the chat application.

**MQTT IoT**

This sample demonstrates how WSO2 MB can use the MQTT transport to publish data from running vehicles to a central server and then use that data for analysis.

- About the sample
- Prerequisites
- Building the sample
- Analyzing the output

**About the sample**

It creates mock vehicles and simulates the functionality of publishing data (such as the current speed, current acceleration and the engine temperature) from the vehicle to a topic hierarchy in MB every second.

Data is published to the topic hierarchy in the following format:

VehicleType/VehicleModel/VehicleId/Sensor

For example, a vehicle of type abc, model xyz and vehicle ID 123 would publish data as shown below to the relevant hierarchies.

- abc/xyz/123/engintemperature
- abc/xyz/123/speed
- abc/xyz/123/acceleration

To publish messages to a hierarchy, use "/" as the hierarchy separator as shown above. 3 MQTT clients will retrieve...
the real time data published from each vehicle through the broker as follows:

- Subscribe for +/+/+/enginetemperature to retrieve engine temperature values of all the vehicles. Then calculate the average value each second.
- Subscribe for car/# to retrieve all sensor data published by cars. Then find and output the car which has the maximum acceleration value for the given seconds.
- Subscribe for bike/bikeModel/# to retrieve all sensor data for the given bike model. Then output the latest speed readings of the bike.

For subscribing to each use case hierarchy, two wild cards have been used in this sample.

<table>
<thead>
<tr>
<th>Wild Card</th>
<th>Description</th>
</tr>
</thead>
</table>
| +         | This suggest to subscribe to a single level  
|           | • eg :- xyz/+/abc will subscribe to both xyz/1/abc and xyz/2/abc but not xyz/3/def |
| #         | This suggest to subscribe to all subtrees  
|           | • eg :- xyz/# will subscribe to all xyz/1/abc, xyz/2/abc/ and xyz/3/def |

The `<MB_HOME>/Samples/MqttIoT/src/main/java/org/wso2/sample/mqtt` directory has the following classes:

- AndesMQTTClient.java
- MQTTSampleConstants.java

The code of this class is as follows:

```java
package org.wso2.sample.mqtt;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.wso2.sample.mqtt.model.Vehicle;
import java.io.File;
import java.util.concurrent.ConcurrentHashMap;

/**
 * The mqtt client implementation for the sample.
 * This keeps hold of the latest data received in a topic related to sensors.
 */

public class AndesMQTTClient implements MqttCallback {
```
private static final Log log = LogFactory.getLog(AndesMQTTClient.class);
public static final String TEMPERATURE_PREFIX = "E:";
public static final String SPEED_PREFIX = "S:";
public static final String ACCELERATION_PREFIX = "A:";
private MqttClient mqttClient;

/**
 * Latest temperature readings received from the server <topic, value>
 */
private final ConcurrentHashMap<String, String> latestTemperatureReadings = new ConcurrentHashMap<String, String>();

/**
 * Latest Speed readings received from the server <topic, value> *
 */
private final ConcurrentHashMap<String, String> latestSpeedReadings = new ConcurrentHashMap<String, String>();

/**
 * Latest acceleration readings received from the server <topic, value> *
 */
private final ConcurrentHashMap<String, String> latestAccelerationReadings = new ConcurrentHashMap<String, String>();

/**
 * Create new mqtt client with the given clientId.
 * @param clientId The unique client Id
 * @param cleanSession Clean previous session data
 * @param userName User Name of the account
 * @param password User Password of the account
 * @throws MqttException
 */
public AndesMQTTClient(String clientId, boolean cleanSession, String userName, String password) throws MqttException {
    MqttConnectOptions options = new MqttConnectOptions();
    options.setCleanSession(cleanSession);
    options.setUserName(userName);
    options.setPassword(password.toCharArray());
    mqttClient = new MqttClient(MQTTSampleConstants.BROKER_URL, clientId,
     new MqttDefaultFilePersistence(MQTTSampleConstants.TMP_DIR + File.separator + clientId));
    mqttClient.setCallback(this);
    mqttClient.connect(options);
}

/**
* Subscribe to a topic.
* @param topic The topic to subscribe
* @param qos The quality of service level
* @throws MqttException
*/
public void subscribe(String topic, int qos) throws MqttException {
    mqttClient.subscribe(topic, qos);
}

/**
* Un-subscribe from a topic.
* @param topic The topic to un-subscribe from
* @throws MqttException
*/
public void unsubscribe(String topic) throws MqttException {
    mqttClient.unsubscribe(topic);
}

/**
* Send a message to mqtt server.
* @param topic The topic to send message to
* @param message The message string to send
* @param qos The quality of service level
* @throws MqttException
*/
public void sendMessage(String topic, String message, int qos) throws MqttException {
    mqttClient.publish(topic, message.getBytes(), qos, false);
}

/**
* Disconnect the mqtt client.
* @throws MqttException
*/
public void disconnect() throws MqttException {
    mqttClient.disconnect();
}

/**
* Connection lost message received from the server.
* @param throwable Connection lost cause
*/
public void connectionLost(Throwable throwable) {
    // We're only logging the connection lost here since this class is only responsible for handling callbacks
    // from server. If client tries to invoke any further operation on server it will create a server error which
    // will then be handled by the client.
log.warn("Server connection lost.", throwable);
}

/**
 * Handle received messages from mqtt server.
 * If the received message is from one of the vehicle sensors, keep the
 * latest one in memory to be retrieved by a
 * third party.
 * @param topic       The topic message received from
 * @param mqttMessage The mqtt message received
 * @throws Exception
 */
public void messageArrived(String topic, MqttMessage mqttMessage)
throws Exception {
    String message = mqttMessage.toString();
    String sensorReading = message.substring(2);
    if (message.startsWith(TEMPERATURE_PREFIX)) {
        latestTemperatureReadings.put(topic, sensorReading);
    } else if (message.startsWith(SPEED_PREFIX)) {
        latestSpeedReadings.put(topic, sensorReading);
    } else if (message.startsWith(ACCELERATION_PREFIX)) {
        latestAccelerationReadings.put(topic, sensorReading);
    }
}

public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
}

/**
 * Get last temperature readings received.
 * @return Received temperatures <topic, value>
 */
public ConcurrentHashMap<String, String> getLatestTemperatureReadings() {
    return latestTemperatureReadings;
}

/**
 * Get last speed readings.
 * @return Received speed readings <topic, value>
 */
public ConcurrentHashMap<String, String> getLatestSpeedReadings() {
    return latestSpeedReadings;
}

/**
 * Get last acceleration reading.
 * @return Received acceleration readings <topic, value>
 */
public ConcurrentHashMap<String, String> getLatestAccelerationReadings() {

The code of this class is as follows:

```java
package org.wso2.sample.mqtt;

/**
 * This holds constants used for the sample.
 */
public class MQTTSampleConstants {
    /**
     * Stop creating instance of this since this is only used to store constants.
     */
    private MQTTSampleConstants() {
    }

    // The URL of the Message Broker
    public static final String BROKER_URL = "tcp://localhost:1883";
    // The temp directory to use for mqtt client
    public static final String TMP_DIR = System.getProperty("java.io.tmpdir");
    /**
     * Credentials to be used when connecting to MQTT server
     */
    public static final String DEFAULT_USER_NAME = "admin";
    public static final String DEFAULT_PASSWORD = "admin";
}
```

The model directory in the mqtt directory has the following classes:

- Vehicle.java
- VehicleModel.java
- VehicleType.java
- Main.java

The code is as follows:

```java
package org.wso2.sample.mqtt.model;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.wso2.sample.mqtt.AndesMQTTClient;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;
```
/**
 * This represents a Vehicle and is responsible for collecting sensor data
 * for itself and sending them to the server.
 */

public class Vehicle {
    private String vehicleId;
    private VehicleModel vehicleModel;
    private AndesMQTTCClient mqttClient;
    private double engineTemperature; // Celsius
    private double speed; // km/h
    private double acceleration; // m/s^2

    public static final String ENGINE_TEMPERATURE = "engintemperature";
    public static final String SPEED = "speed";
    public static final String ACCELERATION = "acceleration";

    private final int qos = 0;
    private final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
    private final ScheduledFuture statusUpdateSchedule;
    private final Log log = LogFactory.getLog(Vehicle.class);

    /**
     * Create a new vehicle initialising vehicleId, Model and sensor data
     * update process.
     *
     * @param vehicleId    The vehicle Id to create
     * @param vehicleModel The model of the vehicle
     * @throws MqttException
     */
    public Vehicle(final String vehicleId, VehicleModel vehicleModel) throws MqttException {
        setVehicleId(vehicleId);
        setVehicleModel(vehicleModel);
        mqttClient = new AndesMQTTCClient(vehicleId, true,
                                            MQTTSampleConstants.DEFAULT_USER_NAME,
                                            MQTTSampleConstants.DEFAULT_PASSWORD);
        // send sensor statuses to the server periodically.
        statusUpdateSchedule = scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
            @Override
            public void run() {
                try {
                    // Send temperature reading
                    mqttClient.sendMessage(generateTopicHierarchy(ENGINE_TEMPERATURE),
                                            String.valueOf(getEngineTemperature()), qos);
                    // Send speed reading
                    mqttClient.sendMessage(generateTopicHierarchy(SPEED),
                                            String.valueOf(getSpeed()), qos);
                    // Send acceleration reading
                    mqttClient.sendMessage(generateTopicHierarchy(ACCELERATION),
                                            String.valueOf(getAcceleration()), qos);
                    log.info("Sensor readings of " + vehicleId + " sent to
                    
                } catch (MqttException e) {
                    log.error("Error sending sensor readings", e);
                }
            }
        }, 0, 10, TimeUnit.SECONDS);

        log.info("Vehicle created with id: " + vehicleId);
    }

    // Setters
    private void setVehicleId(String vehicleId) {
        this.vehicleId = vehicleId;
    }

    private void setVehicleModel(VehicleModel vehicleModel) {
        this.vehicleModel = vehicleModel;
    }

    // Getter
    public String getVehicleId() {
        return vehicleId;
    }

    public VehicleModel getVehicleModel() {
        return vehicleModel;
    }

    // Sensor reading getters
    public double getEngineTemperature() {
        return engineTemperature;
    }

    public double getSpeed() {
        return speed;
    }

    public double getAcceleration() {
        return acceleration;
    }
}
server.");
        } catch (MqttException e) {
            log.error("Error sending sensor data to server.", e);
        }
    }, 0, 1, TimeUnit.SECONDS);
    log.info("A "$ + vehicleModel.getVehicleType().getTypeName() + "$ of model "$ + vehicleModel.getModelName() + "$ "$ +
        "created. Id : "$ + vehicleId);
    }
    public String getVehicleId() {
        return vehicleId;
    }
    public void setVehicleId(String vehicleId) {
        this.vehicleId = vehicleId;
    }
    public VehicleModel getVehicleModel() {
        return vehicleModel;
    }
    public void setVehicleModel(VehicleModel vehicleModel) {
        this.vehicleModel = vehicleModel;
    }
    public double getEngineTemperature() {
        return engineTemperature;
    }
    public void setEngineTemperature(double engineTemperature) {
        this.engineTemperature = engineTemperature;
        log.info("Engine temperature of " + vehicleId + " updated to " +
                engineTemperature);
    }
    public double getSpeed() {
        return speed;
    }
    public void setSpeed(double speed) {
        this.speed = speed;
        log.info("Speed of " + vehicleId + " updated to " + speed);
    }
    public double getAcceleration() {
        return acceleration;
    }
    public void setAcceleration(double acceleration) {
        this.acceleration = acceleration;
        log.info("Acceleration of " + vehicleId + " updated to " +
                acceleration);
    }
    /**
     * Generate the hierarchy the vehicle should publish data to in mqtt.
     * @param leafTopic The leaf of the topic hierarchy
     * @return The topic hierarchy that is feed-able to the broker
     */
    public String generateTopicHierarchy(String leafTopic) {
        return vehicleModel.getVehicleType().getTypeName() + "/
                + vehicleModel.getModelName() + " "$ +
                vehicleId + " "$ +
                leafTopic + "."
            );
        }
    }
    }
}
vehicleModel.getModelName() + "/" + vehicleId + "/" + leafTopic;
}
/**
 * Stop sending sensor data to the server and disconnect clients.
 * @throws MqttException
 */
public void stopSensorDataUpload() throws MqttException {
    statusUpdateSchedule.cancel(true);
    scheduledExecutorService.shutdown();
mqttClient.disconnect();
}
}

The code is as follows:

```java
package org.wso2.sample.mqtt.model;

/**
 * This represents a Vehicle Model in a given Vehicle Type.
 */
public class VehicleModel {
    private String modelName;
    private VehicleType vehicleType;
    /**
     * Creates a new vehicle model with given name and type.
     *
     * @param name The vehicle model name
     * @param type The vehicle model type
     */
    public VehicleModel(String name, VehicleType type) {
        setModelName(name);
        setVehicleType(type);
    }

    public String getModelName() {
        return modelName;
    }

    public void setModelName(String modelName) {
        this.modelName = modelName;
    }

    public VehicleType getVehicleType() {
        return vehicleType;
    }

    public void setVehicleType(VehicleType vehicleType) {
        this.vehicleType = vehicleType;
    }
}
```

The code is as follows:
This defines the method to call all the other classes mentioned above. The code of this class is as follows:

```java
package org.wso2.sample.mqtt;
import org.apache.commons.lang.StringUtils;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.wso2.sample.mqtt.model.Vehicle;
import org.wso2.sample.mqtt.model.VehicleModel;
import org.wso2.sample.mqtt.model.VehicleType;
import java.util.*;
import java.util.concurrent.*;
/**
 * This samples demonstrates how WSO2 Message Broker MQTT can be used to
 * publish data from running vehicles to a
 * central server and use that data to analyze and come to conclusions.
 * <p/>
 * The main class which executes the sample.
 * - Creates mock vehicle types, vehicle models and vehicles
 * - Updates sensor readings periodically in each vehicle with a random
 * value mocking the sensor behaviours
 * - Read sensor data published by all vehicles via mqtt server and
 * generate mock output scenarios.
 * - Real time speed of a given vehicle
 * - Real time average temperature of all the vehicles
 * - Real time maximum speed of a given vehicle type
 */
public class Main {
```
private static final List<Vehicle> vehicleList = new ArrayList<Vehicle>();
private static final Map<vehicleTypes, VehicleType> vehicleTypeMap = new HashMap<vehicleTypes, VehicleType>();
private static final Set<VehicleModel> vehicleModelSet = new HashSet<VehicleModel>();
private static final Random random = new Random();
private static enum vehicleTypes {CAR, BIKE, VAN}
private static AndesMQTTClient temperatureClient;
private static AndesMQTTClient carClient;
private static AndesMQTTClient harleySpeedClient;
private static final ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(2);
private static ScheduledFuture vehicleStatusUpdater;
private static ScheduledFuture vehicleStatusProcessor;
/**
 * Time to run the sample in seconds
 */
private static final int RUNTIME = 20000;
private static final Log log = LogFactory.getLog(Main.class);
/**
 * Executes vehicle population, mock sensor reading update and sensor data reading.
 * @param args main command line arguments
 * @throws InterruptedException
 * @throws MqttException
 */
public static void main(String[] args) throws InterruptedException, MqttException {
    populateVehicles();
scheduleMockVehicleStatusUpdate();
    try {
        listenToVehicleSensorStatuses();
        // Let stats publish and stats processing commence for RUNTIME amount of time before exiting the sample
        Thread.sleep(RUNTIME);
        shutdown();
    } catch (MqttException e) {
        log.error("Error running the sample.", e);
    }
}
/**
 * Schedule to periodically update sensor readings of all vehicles.
 */
private static void scheduleMockVehicleStatusUpdate() {
    // Update sensors with random values.
    vehicleStatusUpdater = scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
        @Override
        public void run() {
            for (Vehicle vehicle : vehicleList) {
                vehicle.setAcceleration(random.nextInt(360));
            }
        }
    }, 0, 2, SECONDS);
}

vehicle.setSpeed(random.nextInt(200));
vehicle.setEngineTemperature(random.nextInt(350));
}
}
}, 0, 500, TimeUnit.MILLISECONDS);
/**
 * Populate vehicles types with CAR, BIKE and VAN.
 */
private static void populateVehicleTypes() {
  // CAR
  VehicleType car = new VehicleType(vehicleTypes.CAR.name());
  // BIKE
  VehicleType bike = new VehicleType(vehicleTypes.BIKE.name());
  // VAN
  VehicleType van = new VehicleType(vehicleTypes.VAN.name());
  vehicleTypeMap.put(vehicleTypes.CAR, car);
  vehicleTypeMap.put(vehicleTypes.BIKE, bike);
  vehicleTypeMap.put(vehicleTypes.VAN, van);
}
/**
 * Populate vehicle models.
 * - 5 models of type CAR
 * - 5 models of type BIKE
 * - 3 models of type VAN
 */
private static void populateVehicleModels() {
  populateVehicleTypes();
  VehicleType car = vehicleTypeMap.get(vehicleTypes.CAR);
  String bmwM3 = "BMWM3";
  String carreraGt = "PorscheCarreraGT";
  String mclarenF1 = "McLarnF1";
  String challenger = "DodgeChallenger";
  String mercielago = "LanborginiMercielago";
  vehicleModelSet.add(new VehicleModel(bmwM3, car));
  vehicleModelSet.add(new VehicleModel(carreraGt, car));
  vehicleModelSet.add(new VehicleModel(mclarenF1, car));
  vehicleModelSet.add(new VehicleModel(challenger, car));
  vehicleModelSet.add(new VehicleModel(mercielago, car));
  VehicleType bike = vehicleTypeMap.get(vehicleTypes.BIKE);
  String nightRod = "HarleyDavidsonNightRod";
  String h2r = "KawasakiH2R";
  String scrambler = "DucatiScramblerFullThrottle";
  String vfr = "HondaVFR800XCrossrunner";
  String gsx = "SuzukiGSX-S1000";
  vehicleModelSet.add(new VehicleModel(nightRod, bike));
  vehicleModelSet.add(new VehicleModel(h2r, bike));
  vehicleModelSet.add(new VehicleModel(scrambler, bike));
  vehicleModelSet.add(new VehicleModel(vfr, bike));
  vehicleModelSet.add(new VehicleModel(gsx, bike));
  VehicleType van = vehicleTypeMap.get(vehicleTypes.VAN);
  String odyssey = "HondaOdyssey";
  String grandCaravan = "DodgeGrandCaravan";
String sedona = "KiaSedona";

vehicleModelSet.add(new VehicleModel(odyssey, van));
vehicleModelSet.add(new VehicleModel(grandCaravan, van));
vehicleModelSet.add(new VehicleModel(sedona, van));

/**
 * Create mock vehicles, 1 per each model.
 * @throws MqttException
 */
private static void populateVehicles() throws MqttException {
  populateVehicleModels();
  int i = 0;
  // 1 vehicle per each vehicle model
  for (VehicleModel vehicleModel : vehicleModelSet) {
    vehicleList.add(new Vehicle("Vehicle" + i++, vehicleModel));
  }
}
/**
 * Read vehicle sensor updates from mqtt and output real time values.
 * @throws MqttException
 */
private static void listenToVehicleSensorStatuses() throws MqttException {
  temperatureClient = new AndesMQTTClient("temperatureClient", true,
MQTTSampleConstants.DEFAULT_USER_NAME,
MQTTSampleConstants.DEFAULT_PASSWORD);
  temperatureClient.subscribe("+/+/+/" + Vehicle.ENGINE_TEMPERATURE,
   1);
  carClient = new AndesMQTTClient("carClient", true,
MQTTSampleConstants.DEFAULT_USER_NAME,
MQTTSampleConstants.DEFAULT_PASSWORD);
  carClient.subscribe(vehicleTypes.CAR.name() + "/#", 1);
  harleySpeedClient = new AndesMQTTClient("harleySpeedClient", true,
MQTTSampleConstants.DEFAULT_USER_NAME,
MQTTSampleConstants.DEFAULT_PASSWORD);
  harleySpeedClient.subscribe(vehicleTypes.BIKE.name() + 
"/HarleyDavidsonNightRod/#", 1);
  // Print real time sensor data each second
  vehicleStatusProcessor =
  scheduledExecutorService.scheduleAtFixedRate(new Runnable() {
  @Override
  public void run() {
    StringBuilder outputString = new StringBuilder();
    // Print the speed of Harley
    ConcurrentHashMap<String, String> latestHarleyReadings =
    harleySpeedClient.getMap<String, String> latestHarleyReadings;
    for (Map.Entry<String, String> entry :
    latestHarleyReadings.entrySet()) {
      String latestReading = entry.getValue();
      // Print the speed of Harley
      outputString

outputString.append("Latest Harley Speed Reading : ").append(latestReading);
}

// Print the average temperature of all vehicles
ConcurrentHashMap<String, String> latestTemperatureReadings = temperatureClient
    .getLatestTemperatureReadings();

double totalTemperaturesSum = 0;
for (Map.Entry<String, String> entry :
    latestTemperatureReadings.entrySet()) {
    String latestReading = entry.getValue();
    if (!StringUtils.isEmpty(latestReading)) {
        totalTemperaturesSum = totalTemperaturesSum +
        Double.parseDouble(latestReading);
    }
}
outputString.append("\tLatest Average Temperature of all vehicles : ").append((totalTemperaturesSum / latestTemperatureReadings.size()));

// Print the CAR which has the maximum acceleration at the moment
ConcurrentHashMap<String, String> latestCarAccelerationReadings = carClient
    .getLatestSpeedReadings();

double maxAcceleration = 0;
String maxAccelerationVehicle = "Undefined";
for (Map.Entry<String, String> entry :
    latestCarAccelerationReadings.entrySet()) {
    String latestReading = entry.getValue();
    if (latestReading != null &&
        !"".equals(latestReading)) {
        double latestAcceleration =
            Double.parseDouble(latestReading);
        if (maxAcceleration < latestAcceleration) {
            maxAcceleration = latestAcceleration;
            maxAccelerationVehicle = entry.getKey();
        }
    }
}
outputString.append("\tCurrent Max car acceleration : ").append(maxAcceleration).append(" in ").append(maxAccelerationVehicle.replace("/speed", "")).
    log.info(outputString);
}
* @throws MqttException
*/
private static void shutdown() throws MqttException {
    log.info("Stopping sample");
    temperatureClient.disconnect();
    carClient.disconnect();
    harleySpeedClient.disconnect();
    for (Vehicle vehicle : vehicleList) {
        vehicle.stopSensorDataUpload();
    }
    vehicleStatusUpdater.cancel(true);
    vehicleStatusProcessor.cancel(true);
Prerequisites

Before you build the sample, the prerequisites for MB samples should be in place.

Building the sample

If you are building an MQTT sample for the first time, you need to build the sample using Maven. This will download the Maven dependencies needed for your MQTT samples. Once the dependencies are downloaded, you can build any of the MQTT samples using either Maven or Ant.

Using Maven:

1. Navigate to the MqttIot sample folder in the <MB_HOME>/samples directory.
2. Execute the mvn clean install command to build and run the sample.
3. If the build is successful, the output will be printed in the terminal as shown below.

Using Ant:

Be sure that the Maven dependencies required for MQTT samples are already downloaded as explained here.

1. Navigate to the MqttIot sample folder in the <MB_HOME>/samples directory.
2. Execute the ant command to build and run the sample.
3. If the build is successful, the output will be printed in the terminal as shown below.

Analyzing the output

Following is the output is printed in the terminal if the sample is successfully executed:

```java
INFO  [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle12 updated to 74.0
INFO  [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature of Vehicle12 updated to 254.0
```
Vehicle7 sent to server.

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Sensor readings of
```

Vehicle8 sent to server.

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Sensor readings of
```

Vehicle9 sent to server.

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Sensor readings of
```

Vehicle10 sent to server.

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Sensor readings of
```

Vehicle11 sent to server.

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle0 updated to 34.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle0
```

updated to 0.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle0 updated to 181.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle1 updated to 229.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle1
```

updated to 78.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle1 updated to 176.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle2 updated to 95.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle2
```

updated to 37.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle2 updated to 56.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle3 updated to 59.0

```java
```

updated to 44.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle3 updated to 168.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle4 updated to 97.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle4
```

updated to 182.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle4 updated to 225.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle5 updated to 200.0

```java
```

updated to 51.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle5 updated to 256.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Acceleration of
```

Vehicle6 updated to 5.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Sensor readings of
```

Vehicle12 sent to server.

```java
```

updated to 145.0

```java
INFO [org.wso2.sample.mqtt.model.Vehicle] - Engine temperature
```

of Vehicle6 updated to 284.0

```java
```
[java] INFO [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle7 updated to 142.0
[java] INFO [org.wso2.sample.mqtt.model.Vehicle] - Speed of Vehicle12 updated to 328.0
MQTT Retain

This sample demonstrates how WSO2 MB handles message arrivals, delivery completions and connections lost when the MQTT transport is used.

- About the sample
- Prerequisites
- Building the sample
- Analyzing the output

About the sample

The `<MB_HOME>/Samples/MqttRetainSample/src/main/java/org/wso2/sample/mqtt` directory has the following classes:

- SimpleMQTTCallback.java
- QualityOfService.java
- Main.java

This class defines a callback handler that handles message arrivals, delivery completions and connections lost. The code of this class is as follows:
package org.wso2.sample.mqtt;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;

/**
 * The MQTT client callback handler which handles message arrivals,
delivery completions and connection lost.
 */
public class SimpleMQTTCallback implements MqttCallback {
    private static final Log log =
        LogFactory.getLog(SimpleMQTTCallback.class);
    /**
     * Inform when connection with server is lost.
     *
     * @param throwable Connection lost cause
     */
    @Override
    public void connectionLost(Throwable throwable) {
        log.error("Mqtt client lost connection with the server",
                   throwable);
    }
    /**
     * Inform when a message is received through a subscribed topic.
     *
     * @param topic       The topic message received from
     * @param mqttMessage The message received
     * @throws Exception
     */
    @Override
    public void messageArrived(String topic, MqttMessage mqttMessage)
            throws Exception {
        log.info("Message arrived on topic : ":" + topic + ": Message :
" +
        mqttMessage.toString() + ":\n") ;
    }
    /**
     * Inform when message delivery is complete for a published message.
     *
     * @param iMqttDeliveryToken The message complete token
     */
    @Override
    public void deliveryComplete(IMqttDeliveryToken iMqttDeliveryToken) {
        for (String topic : iMqttDeliveryToken.getTopics()) {
            log.info("Message delivered successfully to topic : \":" + topic
                     + ":\.");
        }
    }
}
This class applies to the quality of service levels in MQTT. The code of this class is as follows:

```java
package org.wso2.sample.mqtt;
/**
 * The quality of service levels in MQTT.
 */
public enum QualityOfService {
    /**
     * The message is delivered at most once, or it may not be delivered at all. Its delivery across the network is not acknowledged. The message is not stored. The message could be lost if the client is disconnected, or if the server fails. QoS0 is the fastest mode of transfer. It is sometimes called "fire and forget".
     */
    MOST_ONCE(0),
    /**
     * The message is always delivered at least once. It might be delivered multiple times if there is a failure before an acknowledgment is received by the sender. The message must be stored locally at the sender, until the sender receives confirmation that the message has been published by the receiver. The message is stored in case the message must be sent again.
     */
    LEAST_ONCE(1),
    /**
     * The message is always delivered exactly once. The message must be stored locally at the sender, until the sender receives confirmation that the message has been published by the receiver. The message is stored in case the message must be sent again. QoS2 is the safest, but slowest mode of transfer. A more sophisticated handshaking and acknowledgement sequence is used than for QoS1 to ensure no duplication of messages occurs.
     */
    EXACTLY_ONCE(2);
    private final int qos;
    /**
     * Initialize with the given Quality of Service.
     * @param qos The quality of service level
     */
    private QualityOfService(int qos) {
        this.qos = qos;
    }
    /**
     * Get the corresponding value for the given quality of service.
     * Retrieve this value whenever quality of service level needs to feed into external libraries.
     *
     * @return The integer representation of this quality of service
     */
```
*/
public int getValue() {
This class defines the method that is used for calling both the classes mentioned above. The code of this class is as follows:

```java
package org.wso2.sample.mqtt;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * If MQTT Retain enabled broker should keep the retain enabled message for future subscribers.
 * This samples demonstrates how MQTT retain feature works.
 */
public class Main {
    private static final Log log = LogFactory.getLog(Main.class);
    /**
     * Java temporary directory location
     */
    private static final String JAVA_TMP_DIR = System.getProperty("java.io.tmpdir");
    /**
     * The MQTT broker URL
     */
    private static final String brokerURL = "tcp://localhost:1883";
    /**
     * topic name for subscriber and publisher
     */
    private static String topic;
    /**
     * retain state for published topic
     */
    private static boolean retained;
    /**
     * The main method which runs the sample.
     * @param args Commandline arguments
     */
    public static void main(String[] args) throws InterruptedException, IOException {
        BufferedReader bufferReader = new BufferedReader(new InputStreamReader(System.in));
    }
}
```
String subscriberClientId = "subscriber";
String publisherClientId = "publisher";
int maxWaitTimeUntilReceiveMessages = 1000;
// default topic name
String topic = "simpleTopic";
// default retain state
boolean retained = true;
// default payload
byte[] payload = "sample message payload".getBytes();
log.info("Retain Topic Sample");
getUserInputs(bufferReader);
log.info("Start sample with Topic Name " + topic + " with retain " + retained + ".");
try {
    // Creating mqtt subscriber client
    MqttClient mqttSubscriberClient =
        getNewMqttClient(subscriberClientId);
    // Creating mqtt publisher client
    MqttClient mqttPublisherClient =
        getNewMqttClient(publisherClientId);
    // Publishing to mqtt topic "simpleTopic"
    mqttPublisherClient.publish(topic, payload,
        QualityOfService.LEAST_ONCE.getValue(),
        retained);
    log.info("Publish topic message with retain enabled for topic name " + topic);
    // Subscribing to mqtt topic "simpleTopic"
    mqttSubscriberClient.subscribe(topic,
        QualityOfService.LEAST_ONCE.getValue());
    log.info("Subscribe for topic name " + topic);
    Thread.sleep(maxWaitTimeUntilReceiveMessages);
    mqttPublisherClient.disconnect();
    mqttSubscriberClient.disconnect();
    log.info("Clients Disconnected!");
} catch (MqttException e) {
    log.error("Error running the sample", e);
}
/**
 * Read user inputs and set to relevant parameters
 * @param bufferReader buffer text from character input stream
 * @throws IOException
 */
private static void getUserInputs(BufferedReader bufferReader) throws IOException {
    String lineSeparator = System.getProperty("line.separator");
    log.info("Enter topic name : ");
    String bufferReaderString = bufferReader.readLine();
    if (!bufferReaderString.isEmpty()) {
        topic = bufferReaderString;
    } else {
        log.info("Topic name not valid. Continuing with default topic"
name : " + topic);
}  
log.info("Set retain enable [Y/N] : ");
bufferReaderString = bufferReader.readLine();
if (bufferReaderString.contentEquals("Y")) {
  // set retain enable
  retained = true;
} else if (bufferReaderString.contentEquals("N")) {
  // set retain disable
  retained = false;
} else {
  log.info("Retain state not valid. Continuing with default retain state : " + retained);
  
  log.info(lineSeparator + "Enter Y to continue with " + topic + " topic name and" +
            " retain state " + retained + "." + lineSeparator +
            "Enter N to revise parameters [Y/N] : ");
  bufferReaderString = bufferReader.readLine();
  if (bufferReaderString.contentEquals("N")) {
    getUserInputs(bufferReader);
  }
}
/**
 * Create a new MQTT client and connect it to the server.
 * @param clientId The unique mqtt client id
 * @return Connected MQTT client
 * @throws MqttException
 */
private static MqttClient getNewMqttClient(String clientId) throws MqttException {
  // Store messages until server fetches them
  MqttDefaultFilePersistence dataStore = new MqttDefaultFilePersistence(JAVA_TMP_DIR + "/" + clientId);
  MqttClient mqttClient = new MqttClient(brokerURL, clientId, dataStore);
  SimpleMQTTCallback callback = new SimpleMQTTCallback();
  mqttClient.setCallback(callback);
  MqttConnectOptions connectOptions = new MqttConnectOptions();
  connectOptions.setUserName("admin");
  connectOptions.setPassword("admin".toCharArray());
  connectOptions.setCleanSession(true);
  mqttClient.connect(connectOptions);
Prerequisites

Before you build the sample, the prerequisites for MB samples should be in place.

Building the sample

If you are building an MQTT sample for the first time, you need to build the sample using Maven. This will download the Maven dependencies needed for your MQTT samples. Once the dependencies are downloaded, you can build any of the MQTT samples using either Maven or Ant.

Using Maven:

1. Navigate to the MqttRetainSample sample folder in the `<MB_HOME>/samples` directory.
2. Execute the `mvn clean install` command to build and run the sample.
3. If the build is successful, the output will be printed in the terminal as shown below.

Using Ant:

Be sure that the Maven dependencies required for MQTT samples are already downloaded as explained here.

1. Navigate to the MqttRetainSample sample folder in the `<MB_HOME>/samples` directory.
2. Execute the ant command to build and run the mqtt retain sample.
3. Once the sample is started, you will be asked to enter a topic name to publish/subscribe. Enter an appropriate name and proceed.
4. Then it will ask if the retain feature should be enabled for the given topic. Enter Y to enable the retain feature. You can check the behaviour before the retain feature by entering N as the value.
5. If all parameters are set correctly enter Y and proceed. Or else, you can re-enter values by type N.
6. If the build is successful, the output will be printed in the terminal as shown below.

Analyzing the output

Following is the output is printed in the terminal if the retain message is successfully received for future subscriber:
run:

```
[java] INFO  [org.wso2.sample.mqtt.Main] - Enter topic name : new1
[java] INFO  [org.wso2.sample.mqtt.Main] -
[java] Enter Y to continue with new1 topic name and retain state true.
[java] Enter N to revise parameters [Y/N] : Y

[java] INFO  [org.wso2.sample.mqtt.Main] - Publish topic message with retain enabled for topic name new1
[java] INFO  [org.wso2.sample.mqtt.Main] - Subscribe for topic name new1
```

Using Transactional Sessions

This sample demonstrates how transactional messages work with WSO2 MB.

- Prerequisites
- About the sample
- Building the sample
- Analyzing the output

Prerequisites

See Prerequisites to Run the MB Samples for a list of prerequisites.

About the sample

In this sample, a JMS subscriber connects to WSO2 MB and publishes messages to a queue using a 'transacted' session. Using this session ensures that the messages published will persist in WSO2 MB (i.e. will be stored to the DB) only when they are committed. Therefore, as demonstrated by this sample, publishing messages to WSO2 MB through a transacted session involves two steps:

1. The messages have to be **sent** from the publisher client.
2. The messages have to be **committed** from the publisher client.

The `<MB_HOME>/Samples/JMSQueueClient/src/org/sample/jms` directory has the following classes implementing the behaviour explained above.

- TransactionalQueuePublisher.java
- QueueConsumer.java
- Main.java
package org.sample.jms;
import org.apache.log4j.Logger;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * This class contains methods which is used in creating and using a
 * transactional JMS message publisher.
 */
public class TransactionalQueuePublisher {
    private static Logger log = Logger.getLogger(TransactionalQueuePublisher.class);
    /**
     * Andes initial context factory.
     */
    public static final String ANDES_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    /**
     * Connection factory name prefix.
     */
    public static final String CF_NAME_PREFIX = "connectionfactory.";
    /**
     * Andes connection factory name.
     */
    public static final String CF_NAME = "andesConnectionfactory";
    /**
     * The authorized username for the AMQP connection url.
     */
    private static final String userName = "admin";
    /**
     * The authorized password for the AMQP connection url.
     */
    private static final String password = "admin";
    /**
     * Client id for the AMQP connection url.
     */
    private static final String CARBON_CLIENT_ID = "carbon";
    /**
     * MB's Virtual host name should be match with this, default name is
     * "carbon" can be configured.
     */
    private static final String CARBON_VIRTUAL_HOST_NAME = "carbon";
    /**
     * IP Address of the host for AMQP connection url.
     */
private static final String CARBON_DEFAULT_HOSTNAME = "localhost";
/**
 * Standard AMQP port number for the connection url.
 */
private static final String CARBON_DEFAULT_PORT = "5672";
/**
 * Queue prefix for initializing context.
 */
private static final String QUEUE_NAME_PREFIX = "queue.";
/**
 * The queue connection in which the messages would be published.
 */
private QueueConnection queueConnection;
/**
 * The queue session in which the messages would be published.
 */
private QueueSession queueSession;
/**
 * The queue in which the messages would be published.
 */
private Queue queue;
/**
 * Creates a transactional JMS publisher.
 * @param queueName The name of the queue to which messages should be published.
 * @throws NamingException
 * @throws JMSException
 */
public TransactionalQueuePublisher(String queueName) throws NamingException, JMSException {
    // Creating properties for the initial context
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, ANDES_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
    properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
    // Creating initial context
    InitialContext initialContext = new InitialContext(properties);
    // Lookup connection factory
    QueueConnectionFactory connFactory = (QueueConnectionFactory) initialContext.lookup(CF_NAME);
    // Create a JMS connection
    queueConnection = connFactory.createQueueConnection();
    queueConnection.start();
    // Create JMS session object. Here we mentioned that the messages will be published transactionally to the broker.
    queueSession = queueConnection.createQueueSession(true, QueueSession.SESSION_TRANSACTED);
    // Look up a JMS queue
    queue = (Queue) initialContext.lookup(queueName);
// Adding a shutdown hook listener
Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        try {
            shutdownPublisher();
        } catch (JMSException jmsException) {
            throw new RuntimeException(jmsException.getMessage(),
            jmsException);
        }
    }
});

/**
* Publishes a JMS message.
* @param messageContent The message content to publish.
* @throws JMSException
*/
public void sendMessage(String messageContent) throws JMSException {
    // Create the message to send
    TextMessage textMessage = queueSession.createTextMessage(messageContent);
    // Sending a message
    QueueSender queueSender = queueSession.createSender(queue);
    queueSender.send(textMessage);
    log.info("Message sent : " + textMessage.getText());
}

/**
* Committing all messages that are being sent.
* @throws JMSException
*/
public void commitMessages() throws JMSException {
    log.info("Committing messages.");
    queueSession.commit();
}

/**
* Rollbacks all sent messages.
* @throws JMSException
*/
public void rollbackMessages() throws JMSException {
    log.info("Rollbacks all uncommitted messages.");
    queueSession.rollback();
}

/**
* Gets an AMQP connection string.
* @param username authorized username for the connection string.
* @param password authorizes password for the connection string.
* @return AMQP Connection URL
*/
private String getTCPConnectionURL(String username, String password) {
    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'
    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT)
        .append("'")
        .toString();
}
/**
* Shutting down the consumer.
* @throws JMSException
*/
public void shutdownPublisher() throws JMSException {
    log.info("Shutting down publisher.");
    // Housekeeping
    if (null != queueSession) {
        queueSession.close();
    }
    if (null != queueConnection) {
        queueConnection.close();
    }
package org.sample.jms;
import org.apache.log4j.Logger;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * This class contains methods and properties relate to Queue Receiver
 * (Subscriber)
 */
public class QueueConsumer {
    private static Logger log = Logger.getLogger(QueueConsumer.class);
    /**
     * Andes initial context factory.
     */
    public static final String ANDES_ICF =
        "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    /**
     * Connection factory name prefix.
     */
    public static final String CF_NAME_PREFIX = "connectionfactory.";
    /**
     * Andes connection factory name.
     */
    public static final String CF_NAME = "andesConnectionfactory";
    /**
     * The authorized username for the AMQP connection url.
     */
    private static final String userName = "admin";
    /**
     * The authorized password for the AMQP connection url.
     */
    private static final String password = "admin";
    /**
     * Client id for the AMQP connection url.
     */
    private static final String CARBON_CLIENT_ID = "carbon";
* MB's Virtual host name should be match with this, default name is "carbon" can be configured. *
/**
 * MB's Virtual host name should be match with this, default name is "carbon" can be configured.
 */
private static final String CARBON_VIRTUAL_HOST_NAME = "carbon";
/**
 * IP Address of the host for AMQP connection url.
 */
private static final String CARBON_DEFAULT_HOSTNAME = "localhost";
/**
 * Standard AMQP port number for the connection url.
 */
private static final String CARBON_DEFAULT_PORT = "5672";
/**
 * Queue prefix for initializing context.
 */
private static final String QUEUE_NAME_PREFIX = "queue.";
/**
 * The queue connection in which the messages would be published.
 */
private QueueConnection queueConnection;
/**
 * The queue session in which the messages would be published.
 */
private QueueSession queueSession;
/**
 * The message consumer for the subscriber.
 */
private MessageConsumer consumer;
/**
 * Creating a Message Consumer.
 *
 * @param queueName The name of the queue in which the subscriber should listen to.
 * @throws NamingException
 * @throws JMSException
 */
public QueueConsumer(String queueName) throws NamingException, JMSException {
    // Creating properties for the initial context
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, ANDES_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
    // Creating initial context
    InitialContext initialContext = new InitialContext(properties);
    // Lookup connection factory
    QueueConnectionFactory connFactory = (QueueConnectionFactory) initialContext.lookup(CF_NAME);
    // Create a JMS connection
    queueConnection = connFactory.createQueueConnection();
    queueConnection.start();
    // Create JMS session object
queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
// Look up a JMS queue
Queue queue = (Queue) initialContext.lookup(queueName);
// Create JMS consumer
consumer = queueSession.createConsumer(queue);
// Adding a shutdown hook listener
Runtime.getRuntime().addShutdownHook(new Thread() {
    @Override
    public void run() {
        try {
            shutdownConsumer();
        } catch (JMSException jmsException) {
            throw new RuntimeException(jmsException.getMessage(), jmsException);
        }
    }
});
/**
 * Receives a single message through the subscriber.
 * @return true if a message was received, else false
 * @throws NamingException
 * @throws JMSException
 */
public boolean receiveMessage() throws NamingException, JMSException {
    long waitingTime = 5000;
    Message receivedMessage = this.consumer.receive(waitingTime);
    if (null == receivedMessage) {
        log.info("No messages were received within " + waitingTime / 1000 + " seconds.");
        return false;
    } else {
        TextMessage message = (TextMessage) receivedMessage;
        log.info("Received message : ", message.getText());
        return true;
    }
}
/**
 * Gets an AMQP connection string.
 * @param username authorized username for the connection string.
 * @param password authorizes password for the connection string.
 * @return AMQP Connection URL
 */
private String getTCPConnectionURL(String username, String password) {
    //
    amqp://(username):(password)@carbon/carbon?brokerlist='tcp://(hostname):(port)'
    return new StringBuffer()
        .append("amqp://")
        .append(username)
        .append(":")
        .append(password)
```java
    .append("@").append(CARBON_CLIENT_ID)
    .append("/").append(CARBON_VIRTUAL_HOST_NAME)
    .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
    .append(CARBON_DEFAULT_PORT)
    .append("'"
            .toString();
}
/**
 * Shutting down the consumer.
 */
public void shutdownConsumer() throws JMSException {
    log.info("Shutting down consumer.");
    // Housekeeping
    if (null != consumer) {
        consumer.close();
    }
    if (null != queueSession) {
        queueSession.close();
    }
    if (null != queueConnection) {
        queueConnection.stop();
    }
    if (null != queueConnection) {
        queueConnection.close();
    }
```
package org.sample.jms;
import org.apache.log4j.Logger;
import javax.jms.JMSException;
import javax.naming.NamingException;
/**
 * The following class contains a publisher transactional sample. This
 * sample uses publisher transactions so that it
 * would help in recovering published messages in case if the server goes
 * down. This helps to prevent message loss.
 */
public class MainClass {
    private static final Logger log = Logger.getLogger(MainClass.class);
    /**
     * The main method for the transactional publishing sample.
     *
     * @param args The arguments passed.
     * @throws NamingException
     * @throws JMSException
     */
    public static void main(String[] args) throws NamingException, JMSException {
        // Creating a message consumer
        QueueConsumer queueConsumer = new QueueConsumer("Transactional-Queue");
        // Creating a transactional message publisher
        TransactionalQueuePublisher transactionalQueuePublisher = new TransactionalQueuePublisher("Transactional-Queue");
        log.info("-------Sample for Message Sending and Committing.-------");
        // Publishes a message
        transactionalQueuePublisher.sendMessage("My First Message.");
        // Attempts to receive a message. No messages were received here as
        // the send message was not committed.
        queueConsumer.receiveMessage();
        // Publishes a message
        transactionalQueuePublisher.sendMessage("My Second Message.");
        // Committing all published messages.
        transactionalQueuePublisher.commitMessages();
        // Receives a message.
        queueConsumer.receiveMessage();
        // Receives a message.
        queueConsumer.receiveMessage();
        log.info("-------Sample for Message Sending, Rollback and
        Committing.-------");
        // Publishes a message
        transactionalQueuePublisher.sendMessage("My Third Message.");
        // Attempts to receive a message. No messages were received here as
the sent message was not committed.
    queueConsumer.receiveMessage();
    // Rollbacks all published messages. This can be used in-case if
the server has gone down and in need of
    // recovering published messages.
    transactionalQueuePublisher.rollbackMessages();
    // Publishes a messages
    transactionalQueuePublisher.sendMessage("My Forth Message.");
    // Committing all published messages.
    transactionalQueuePublisher.commitMessages();
    // Receives a message.
    queueConsumer.receiveMessage();
    // Attempts to receive a message. No messages were received here as
all the messages were received.
    queueConsumer.receiveMessage();
    // Shutting down the sample.
Building the sample

Run the ant command from <MB_HOME>/Samples/TransactionalPublisher directory.

Analyzing the output

The result log shown above explains how the transactional session has worked when publishing messages:

```
[java] INFO : org.sample.jms.MainClass - ------Sample for Message Sending and Committing.------
[java] INFO : org.sample.jms.QueueConsumer - No messages were received within 5 seconds.
[java] INFO : org.sample.jms.TransactionalQueuePublisher - Committing messages.
[java] INFO : org.sample.jms.MainClass - ------Sample for Message Sending, Rollback and Committing.------
[java] INFO : org.sample.jms.QueueConsumer - No messages were received within 5 seconds.
[java] INFO : org.sample.jms.TransactionalQueuePublisher - Rollbacks all uncommitted messages.
[java] INFO : org.sample.jms.TransactionalQueuePublisher - Committing messages.
[java] INFO : org.sample.jms.QueueConsumer - No messages were received within 5 seconds.
[java] INFO : org.sample.jms.TransactionalQueuePublisher - Shutting down publisher.
```

Setting Message Expiration

This sample demonstrates how the Time to Live (TTL) can be set for messages published to WSO2 message broker (WSO2 MB). Go to Configuring Message Expiration for more information on this feature.
Prerequisites

See Prerequisites to Run the MB Samples for a list of prerequisites.

About the sample

This sample demonstrates first introduces a sample JMS client named QueueSender, which is used to send messages with or without a TTL value set for a queue in WSO2 Message Broker. Then it uses a sample JMS client named QueueReceiver to receive the messages, which are not expired at that time and prints the number of received messages on the console.

The <MB_HOME>/Samples/JmsExpirationSample/src/org/sample/jms directory has the following classes:

- SampleQueueSender.Java
- SampleQueueReceiver.Java
- Main.java

```java
package org.sample.jms;

import javax.jms.DeliveryMode;
import javax.jms.JMSException;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * Sample sender to send the messages with/without TTL
 */
public class SampleQueueSender {

    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory."
    private static final String QUEUE_NAME_PREFIX = "queue."
    private static final String CF_NAME = "qpidConnectionfactory"
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
```
String queueName = "expirationTestQueue";
private QueueConnection queueConnection;
private QueueSession queueSession;

/**
 * Send the specified number of messages with the specified ttl.
 * @param noOfMessages Number of messages that need to be sent
 * @param timeToLive Time to live value for messages
 * @throws NamingException
 * @throws JMSException
 */
public void sendMessages(int noOfMessages, long timeToLive) throws
NamingException, JMSException {
    Properties properties = new Properties();
    properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
    properties.put(CF_NAME_PREFIX + CF_NAME,
            getTCPConnectionURL(userName, password));
    properties.put(QUEUE_NAME_PREFIX + queueName, queueName);
    InitialContext ctx = new InitialContext(properties);
    // Lookup connection factory
    QueueConnectionFactory connFactory = (QueueConnectionFactory)
           ctx.lookup(CF_NAME);
    queueConnection = connFactory.createQueueConnection();
    queueConnection.start();
    queueSession = queueConnection.createQueueSession(false,
            QueueSession.AUTO_ACKNOWLEDGE);
    // Send message
    Queue queue = (Queue)ctx.lookup(queueName);
    // create the message to send
    TextMessage textMessage = queueSession.createTextMessage("Test
            Message Content");
    javax.jms.QueueSender queueSender =
            queueSession.createSender(queue);
    for(int i = 0; i < noOfMessages; i++){
        //send the text message in persistent delivery mode with a time to
        //live value at priority level 4
        queueSender.send(textMessage,
                DeliveryMode.PERSISTENT,4,timeToLive);
    }
    queueSender.close();
    queueSession.close();
    queueConnection.close();
}

/**
 * Creates amqp url.
 * @param username The username for the amqp url.
 * @param password The password for the amqp url.
 * @return AMQP url.
 */
private String getTCPConnectionURL(String username, String password) { 

    //
amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'

return new StringBuffer()

.append("amqp://").append(username).append(":").append(password)
.append("@").append(CARBON_CLIENT_ID)
.append("/").append(CARBON_VIRTUAL_HOST_NAME)

.append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
.append(CARBON_DEFAULT_PORT)
.append("'")
package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * Sample Receiver to receive the messages which were not expired
 */
public class SampleQueueReceiver {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "expirationTestQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;

    /**
     * Register Subscriber for a queue.
     * @return MessageConsumer The message consumer object of the subscriber.
     * @throws NamingException
     * @throws JMSException
     */
    public MessageConsumer registerSubscriber() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put("queue." + queueName, queueName);
        Context ctx = new InitialContext(properties);
        QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        try (QueueConnection conn = qcf.createQueueConnection()) {
            queueConnection = conn;
            Queue queue = conn.createQueue(queueName);
            try (QueueSession sess = conn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE)) {
                queueSession = sess;
                MessageConsumer consumer = queueSession.createConsumer(queue);
                return consumer;
            }
        }
    }

    private String getTCPConnectionURL(String userName, String password) {
        return String.format("tcp://%s:%s@%s:%s/%s", userName, password, CARBON_DEFAULT_HOSTNAME, CARBON_DEFAULT_PORT, CARBON_VIRTUAL_HOST_NAME);
    }
}

```java

package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSession;
import javax.jms.TextMessage;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Properties;

/**
 * Sample Receiver to receive the messages which were not expired
 */
public class SampleQueueReceiver {
    public static final String QPID_ICF = "org.wso2.andes.jndi.PropertiesFileInitialContextFactory";
    private static final String CF_NAME_PREFIX = "connectionfactory.";
    private static final String CF_NAME = "qpidConnectionfactory";
    String userName = "admin";
    String password = "admin";
    private static String CARBON_CLIENT_ID = "carbon";
    private static String CARBON_VIRTUAL_HOST_NAME = "carbon";
    private static String CARBON_DEFAULT_HOSTNAME = "localhost";
    private static String CARBON_DEFAULT_PORT = "5672";
    String queueName = "expirationTestQueue";
    private QueueConnection queueConnection;
    private QueueSession queueSession;

    /**
     * Register Subscriber for a queue.
     * @return MessageConsumer The message consumer object of the subscriber.
     * @throws NamingException
     * @throws JMSException
     */
    public MessageConsumer registerSubscriber() throws NamingException, JMSException {
        Properties properties = new Properties();
        properties.put(Context.INITIAL_CONTEXT_FACTORY, QPID_ICF);
        properties.put(CF_NAME_PREFIX + CF_NAME, getTCPConnectionURL(userName, password));
        properties.put("queue." + queueName, queueName);
        Context ctx = new InitialContext(properties);
        QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup(CF_NAME);
        try (QueueConnection conn = qcf.createQueueConnection()) {
            queueConnection = conn;
            Queue queue = conn.createQueue(queueName);
            try (QueueSession sess = conn.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE)) {
                queueSession = sess;
                MessageConsumer consumer = queueSession.createConsumer(queue);
                return consumer;
            }
        }
    }

    private String getTCPConnectionURL(String userName, String password) {
        return String.format("tcp://%s:%s@%s:%s/%s", userName, password, CARBON_DEFAULT_HOSTNAME, CARBON_DEFAULT_PORT, CARBON_VIRTUAL_HOST_NAME);
    }
}
```
InitialContext ctx = new InitialContext(properties);
// Lookup connection factory
QueueConnectionFactory connFactory = (QueueConnectionFactory) ctx.lookup(CF_NAME);
queueConnection = connFactory.createQueueConnection();
queueConnection.start();
queueSession = queueConnection.createQueueSession(false, QueueSession.AUTO_ACKNOWLEDGE);
//Receive message
Queue queue = (Queue) ctx.lookup(queueName);
MessageConsumer consumer = queueSession.createConsumer(queue);
return consumer;
}

/**
 * Receive messages using the consumer.
 * @param consumer The message consumer object of the subscriber.
 * @throws NamingException
 * @throws JMSException
 */
public void receiveMessages(MessageConsumer consumer) throws NamingException, JMSException {
    int receivedMessageCount = 0;
    //have 5 seconds as receive timeout value to stop the consumer
    while(null != consumer.receive(5000)){
        receivedMessageCount ++;
    }
    System.out.println("Received message count: " + receivedMessageCount);
}

/**
 * Close the connections at the end of operation
 * @param consumer The message consumer object of the subscriber.
 * @throws JMSException
 */
public void closeConnections(MessageConsumer consumer) throws JMSException{
    consumer.close();
    queueSession.close();
    queueConnection.stop();
    queueConnection.close();
}

/**
 * Creates amqp url.
 * @param username The username for the amqp url.
 * @param password The password for the amqp url.
 * @return AMQP url.
 */

private String getTCPConnectionURL(String username, String password) {

    //
    amqp://{username}:{password}@carbon/carbon?brokerlist='tcp://{hostname}:{port}'

    return new StringBuffer()
        .append("amqp://").append(username).append(":").append(password)
        .append("@").append(CARBON_CLIENT_ID)
        .append("/").append(CARBON_VIRTUAL_HOST_NAME)
        .append("?brokerlist='tcp://").append(CARBON_DEFAULT_HOSTNAME).append(":")
        .append(CARBON_DEFAULT_PORT)
        .append("'"");
}
package org.sample.jms;

import javax.jms.JMSException;
import javax.jms.MessageConsumer;
import javax.naming.NamingException;

/**
 * Sample executor class for message TTL
 */
public class Main {

    public static void main(String[] args) throws NamingException, JMSException {

        SampleQueueReceiver queueReceiver = new SampleQueueReceiver();
        MessageConsumer consumer = queueReceiver.registerSubscriber();

        //Send messages with very less time to live value
        System.out.println("Sending 5 messages with TTL value of 1sec");
        SampleQueueSender queueSenderWithTTL = new SampleQueueSender();
        queueSenderWithTTL.sendMessages(5, 1000);
        queueReceiver.receiveMessages(consumer);

        //Send messages without time to live value
        System.out.println("Sending 5 messages without TTL");
        SampleQueueSender queueSenderWithoutTTL = new SampleQueueSender();
        queueSenderWithoutTTL.sendMessages(5, 0);
        queueReceiver.receiveMessages(consumer);

        //Send messages with considerable time to live value
        System.out.println("Sending 5 messages TTL value of 10sec");
        SampleQueueSender queueSenderWithMediumTTL = new SampleQueueSender();
        queueSenderWithMediumTTL.sendMessages(5, 10000);
        queueReceiver.receiveMessages(consumer);

        //Close the connection
        queueReceiver.closeConnections(consumer);
    }
}

Building the sample

Run the ant command from the <MB_HOME>/Samples/JmsExpirationSample directory.

Analyzing the output
You will get the following log in your console.

```java
[java] Sending 5 messages with TTL value of 1sec
[java] Received message count: 0
[java] Sending 5 messages without TTL
[java] Received message count: 5
[java] Sending 5 messages TTL value of 10sec
[java] Received message count: 5
```

The first 5 messages were published with a TTL value of one second and none of them got delivered since they expired. In the second case, 5 messages were sent without a TTL value and all of them got delivered. In the last case, 5 messages were sent with a TTL value of ten seconds and all of them got delivered since they could reach the recipient before the messages expire.
Deep Dive

See the following topics for more information on features in WSO2 MB:

- Installation Guide
- Product Administration
- Analytics
- JMS Subscribers and Publishers
- Configuration Files

Installation Guide

This chapter contains the following information:

- Downloading the Product
- Installing the Product
- Running the Product

Downloading the Product

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

2. Click the Download button in the upper right corner of the page.
3. Enter the required details, and click Download.

The binary distribution contains the binary files for both MS Windows and Linux-based 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.

Installing the Product

This section provides information on installing source and binary distributions of WSO2 products on Windows and Linux.

- Installation Prerequisites
- Installing on Linux
- Installing on Solaris
- Installing on Windows
- Installing as a Linux Service

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 JMS messages but the requirements vary with larger message sizes and the number of messages processed concurrently. |
| Disk           | Disk space (for the database being used) will vary according to your requirements as messages are stored in the database. |

**Environment compatibility**
Operating Systems / Databases

- All WSO2 Carbon-based products are Java applications that can be run on any platform that is Oracle/IBM JDK 7/8 compliant. Also, we do not recommend or support OpenJDK. See the section on compatibility of WSO2 products for more information.

If you are using IBM JDK, the default configurations in the product has to be changed as follows:

- Open the Owasp.Csrfguard.Carbon.properties file (stored in the <MB_HOME>/repository/conf/security directory) and change the org.owasp.csrfguard.PRNG.Provider property to IBMJCE as shown below.

```
org.owasp.csrfguard.
PRNG.Provider=IBMJCE
```

- Open the broker.xml file (stored in the <MB_HOME>/repository/conf/ directory) and set the <certType> property to IbmX509 as shown below.

```
<certType>IbmX509</certType>
```

- 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 Message Broker and its samples or for building from the source code. Mandatory installs are marked with an asterisk *.

<table>
<thead>
<tr>
<th>Application</th>
<th>Purpose</th>
<th>Version</th>
<th>Download Links</th>
</tr>
</thead>
</table>

Copyright © WSO2 Inc. 2005-2014
### Oracle Java SE Development Kit (JDK)*
- 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.

<table>
<thead>
<tr>
<th>JDK</th>
<th>URL</th>
</tr>
</thead>
</table>

### Apache Ant
- To compile and run the product samples.

<table>
<thead>
<tr>
<th>Java</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7.0 or later</td>
<td><a href="http://ant.apache.org/">http://ant.apache.org/</a></td>
</tr>
</tbody>
</table>

### Git
- Download the source code and build the product from the source distribution.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux -</td>
<td><a href="http://git-scm.com/download/linux">http://git-scm.com/download/linux</a></td>
</tr>
<tr>
<td>Windows -</td>
<td><a href="http://git-scm.com/download/win">http://git-scm.com/download/win</a></td>
</tr>
</tbody>
</table>

### Apache Maven
- To build the product from the source distribution (both JDK and Apache Maven are required). If you are installing by downloading and extracting the binary distribution instead of building from the source code, you do not need to install Maven.

<table>
<thead>
<tr>
<th>Java</th>
<th>URL</th>
</tr>
</thead>
</table>

### Web Browser
- To access the product's Management Console. The Web Browser must be JavaScript enabled to take full advantage of the Management console.

On Windows Server 2003, you must not go below the medium security level in Internet Explorer 6.x.

### Installing on Linux

Follow the instructions below to install the required applications and the WSO2 product on Linux.

#### 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 the product as described in Downloading the
1. 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 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.

   ```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-3.8.3
   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:

   ```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:

   ```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.

---

**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

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_T_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.

   ```
   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-3.0.5
   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:

   ```
   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

**Before you begin, please see our compatibility matrix** to find out if this version of the product is fully tested on Windows.

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

#### Installing the required applications

1. Ensure that your system meets the **Installation Prerequisites** Java Development Kit (JDK) is essential to run the product.
2. Ensure that the `PATH` environment variable is set to "C:\Windows\System32", because the `findstr` Windows .exe file 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:

```
set JAVA_HOME
```

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 Linux Service

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

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

#### Prerequisites

Install JDK and set up the JAVA_HOME environment variable. For more information, see Installation Prerequisites.

#### Setting up CARBON_HOME

Extract the WSO2 product that you want to run as a Linux service and set the environment variable CARBON_HOME to the extracted product 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:

```bash
#!/bin/bash

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

For example, given below is a startup script written for WSO2 Application Server 5.2.0:
#!/bin/sh
export JAVA_HOME="/usr/lib/jvm/jdk1.7.0_07"

startcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh start > /dev/null &'
restartcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh restart > /dev/null &'
stopcmd='/opt/WSO2/wso2as-5.2.0/bin/wso2server.sh stop > /dev/null &'

case "$1" in
  start)
    echo "Starting WSO2 Application Server ..."
    su -c "${startcmd}" user1
    ;;
  restart)
    echo "Re-starting WSO2 Application Server ..."
    su -c "${restartcmd}" user1
    ;;
  stop)
    echo "Stopping WSO2 Application Server ..."
    su -c "${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 appserver 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/appserver`
- Add a link to `/etc/init.d/`: `sudo ln -snf /opt/WSO2/appserver /etc/init.d/appserver`

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 appserver 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. The following sections describe how to run the product.

- Starting the server
- Accessing the Management Console
- Stopping the server
- Related topics

**Starting the server**

To start the server, you run `<PRODUCT_HOME>/bin/wso2server.bat` (on Windows) or `<PRODUCT_HOME>/bin/wso2server.sh` (on Linux/Solaris/Mac OS) from the command prompt as described below. Alternatively, you can install and run the server as a Windows or Linux service (see the related topics section at the end of this page).

1. Open a command prompt by following the instructions below.
   - On Windows: Click **Start -> Run**, type `cmd` at the prompt, and then press **Enter**.
   - On Linux/Solaris/Mac OS: Establish an SSH connection to the server, log in to the text Linux console, or open a terminal window.

2. Execute one of the following commands:
   - To start the server in a typical environment:
     - On Windows: `<PRODUCT_HOME>/bin/wso2server.bat --run`
     - On Linux/Solaris/Mac OS: `sh <PRODUCT_HOME>/bin/wso2server.sh`
   - To start the server in the background mode of Linux: `sh <PRODUCT_HOME>/bin/wso2server.sh start`
     To stop the server running in this mode, you will enter: `sh <PRODUCT_HOME>/bin/wso2server.sh stop`
   - To provide access to the production environment without allowing any user group (including admin) to log into the Management Console:
     - On Windows: `<PRODUCT_HOME>/bin/wso2server.bat --run -DworkerNode`
     - On Linux/Solaris/Mac OS: `sh <PRODUCT_HOME>/bin/wso2server.sh -DworkerNode`

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'`

- To check for additional options you can use with the startup commands, type `-help` after the command, such as:
3. The operation log appears in the command window. When the product server has successfully started, 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 typing its URL in a Web browser. The following sections provide more information about running the Management Console:

- **Working with the URL**
- **Signing in**
- **Getting help**
- **Configuring the session time-out**
- **Restricting access to the Management Console and Web applications**

**Working with the URL**

The URL appears next to “Mgt Console URL” in the start script log that is displayed in the command window. For example:

```bash
```

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 is installed, you can type `localhost` instead of the IP address as follows: `https://localhost:9443/carbon`

You can change the Management Console URL by modifying the value of the `<MgtHostName>` property in the `<PRODUCT_HOME>/repository/conf/carbon.xml` file. When the host is internal or not resolved by a DNS, map the hostname alias to its IP address in the `/etc/hosts` file of your system, and then enter that alias as the value of the `<MgtHostName>` property in `carbon.xml`. For example:

```
In /etc/hosts:
127.0.0.1 localhost

In carbon.xml:
<MgtHostName>localhost</MgtHostName>
```

**Signing in**

At the sign-in screen, you can sign in to the Management Console using `admin` as both the username and password.

When the Management Console sign-in page appears, the Web browser typically displays 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. Therefore, 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. 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 actually belongs to the given organization.

**Getting help**

The tabs and menu items in the navigation pane on the left may vary depending on the features you have installed. To view information about a particular page, click the Help link at the top right corner of that page, or click the Docs link to open the documentation for full information on managing the product.

**Configuring the session time-out**

If you leave the Management Console unattended for a defined time, its login 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.

```xml
<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. 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 only applies to the Management Console, and thereby all outside requests to the Management Console are 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 applies to each Web application hosted on the WSO2 product server. Therefore, all outside requests to any Web application are blocked.
You can also restrict access to particular servlets in a Web application by adding a Remote Address Filter to the `<PRODUCT_HOME>/repository/conf/tomcat/web.xml` file 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 Console page (`/carbon/admin/login.jsp`) is granted only to one IP address.

```xml
<filter>
    <filter-name>Remote Address Filter</filter-name>
    <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>
```

Any configurations (including valves defined in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file) apply to all Web applications and are globally available across the server, regardless of the host or cluster. For more information about using remote host filters, see the Apache Tomcat documentation.

**Stopping the server**

To stop the server, press **Ctrl+C** in the command window, or click the **Shutdown/Redefine** link in the navigation pane in the Management Console. If you started the server in background mode in Linux, enter the following command instead:

```
sh <PRODUCT_HOME>/bin/wso2server.sh stop
```

**Related topics**

- HTTP-NIO Transport
- Installing as a Windows Service
- Installing as a Linux Service
- Product Startup Options

**Product Administration**

WSO2 Message Broker (WSO2 MB) is shipped with default configurations that will allow you to download, install and get started with your product instantly. However, when you go into production, it is recommended to change some of the default settings to ensure that you have a robust system that is suitable for your operational needs. Also, you may have specific use cases that require specific configurations to the server. If you are a product administrator, the follow content will provide an overview of the administration tasks that you need to perform when working with WSO2 Message Broker (WSO2 MB).

[ Upgrading from a previous release ][ Changing the default database ][ Configuring users, roles and permissions ]
Configuring security [(Configuring transports)] [(Configuring multitenancy)] [(Configuring the registry)] [(Performance tuning)] [(Changing the default ports)] [(Installing, uninstalling and managing product features)] [(Configuring custom proxy paths)] [(Customizing error pages)] [(Customizing the management console)] [(Applying patches)] [(Monitoring the server)] [(Troubleshooting WSO2 MB)] [(Clustering)]

Upgrading from a previous release

If you are upgrading from WSO2 MB 3.1.0 to WSO2 MB 3.2.0 version, see the upgrading instructions for WSO2 Message Broker.

Changing the default database

WSO2 MB contains two embedded H2 databases: The default Carbon database, which is used for storing user management and registry data, and the default broker-specific database.

You can change the default database configurations in WSO2 MB by setting up new physical databases, and updating the relevant configurations. We recommend the use of an industry-standard RDBMS such as Oracle, PostgreSQL, MySQL, MS SQL, etc. when you set up your production environment.

- For information on setting up a new database for your profile, see Setting up the Physical Database in the WSO2 Administration Guide.

  Add the database drivers to the `<MB_HOME>/repository/components/lib/` directory when setting up the database.

- Once you set up a new Carbon database, see Changing the Carbon Database for instructions on updating the configurations.
- Once you set up a new broker-specific database, see Changing the Default Broker Database for instructions on updating the configurations.

Configuring users, roles and permissions

The user management feature in your product allows you to create new users and define the permissions granted to each user. You can also configure the user stores that are used for storing data related to user management.

- For instructions on how to configure user management, see Working with Users, Roles and Permissions in the WSO2 Administration Guide.
- For descriptions of permissions that apply to WSO2 MB users, see Role-Based Permissions for WSO2 Message Broker.

Configuring security

After you install WSO2 MB, it is recommended to change the default security settings according to the requirements of your production environment. As MB is built on top of the WSO2 Carbon Kernel (version 4.4.6), the main security configurations applicable to MB are inherited from the Carbon kernel.

For instructions on configuring security in your server, see the following topics in the WSO2 Administration Guide.

- Configuring Transport-Level Security
- Using Asymmetric Encryption
- Using Symmetric Encryption
- Enabling Java Security Manager
Configuring transports

WSO2 Message Broker (WSO2 MB) uses two transport protocols for the purpose of brokering messages between publishers and subscribers. These protocols are the Advanced Message Queueing Protocol (AMQP) and the Message Queueing and Telemetry Transport (MQTT).

For instructions on configuring these transports, see Configuring Transports for WSO2 MB.

Note that WSO2 Message Broker does not require the transports enabled through the Carbon Kernel.

Configuring multitenancy

You can create multiple tenants in your product server, which will allow you to maintain tenant isolation in a single server/cluster. For instructions on configuring multiple tenants for your server, see Working with Multiple Tenants in the WSO2 Administration Guide.

Configuring the registry

A registry is a content store and a metadata repository for various artifacts such as services, WSDLs and configuration files. In WSO2 products, all configurations pertaining to modules, logging, security, data sources and other service groups are stored in the registry by default.

For instructions on setting up and configuring the registry for your server, see Working with the Registry in the WSO2 Administration Guide.

Performance tuning

You can optimize the performance of your WSO2 server by using configurations and settings that are suitable to your production environment. At a basic level, you need to have the appropriate OS settings, JVM settings etc. Since WSO2 products are all based on a common platform called Carbon, most of the OS, JVM settings recommended for production are common to all WSO2 products. Additionally, there will be other performance enhancing configuration recommendations that will depend on very specific features used by your product.

For instructions on the Carbon platform-level performance tuning recommendations, see Performance Tuning in the WSO2 Administration Guide.

For instructions on performance tuning recommendations that are specific to WSO2 MB functionality, see the topics given below.

- Clustering Performance
- Database Performance
- Message Publisher and Consumer Performance
- Tuning Flow Control

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.
For instructions on configuring posts, see Changing the Default Ports in the WSO2 Administration Guide.

---

**Installing, uninstalling and managing product features**

Each WSO2 product is a collection of reusable software units called features where a single feature is a list of components and/or other feature. By default, WSO2 MB is shipped with the features that are required for your main use cases.

For information on installing new features, or removing/updating an existing feature, see Working with Features in the WSO2 Administration Guide.

---

**Configuring custom proxy paths**

This feature is particularly useful when multiple WSO2 products (fronted by a proxy server) are hosted under the same domain name. By adding a custom proxy path you can host all products under a single domain and assign proxy paths for each product separately.

For instructions on configuring custom proxy paths, see Adding a Custom Proxy Path in the WSO2 Administration Guide.

---

**Customizing error pages**

You can make sure that sensitive information about the server is not revealed in error messages, by customizing the error pages in your product.

For instructions, see Customizing Error Pages in the WSO2 Administration Guide.

---

**Customizing the management console**

Some of the WSO2 products, such as WSO2 MB consist of a web user interface named the management console. This allows administrators to configure, monitor, tune, and maintain the product using a simple interface. You can customize the look and feel of the management console for your product.

For instructions, see Customizing the Management Console in the WSO2 Administration Guide.

---

**Applying patches**

For instructions on applying patches (issued by WSO2), see WSO2 Patch Application Process in the WSO2 Administration Guide.

---

**Monitoring the server**

Monitoring is an important part of maintaining a product server. Listed below are the monitoring capabilities that are available for WSO2 MB.

- **Monitoring server logs**: A properly configured logging system is vital for identifying errors, security threats and usage patterns in your product server. For instructions on monitoring the server logs, see Monitoring Logs in the WSO2 Administration Guide.

  - **Message tracing in WSO2 MB**: In WSO2 MB you have the option of tracing messages by enabling a trace log file. See the troubleshooting guide for instructions.
Monitoring using WSO2 metrics: WSO2 MB 3.5.0 is shipped with JVM Metrics, which allows you to monitor statistics of your server using Java Metrics. For instructions on setting up and using Carbon metrics for monitoring, see Using WSO2 Metrics in the WSO2 Administration Guide.

JMX-based monitoring: For information on monitoring your server using JMX, see JMX-based monitoring in the WSO2 Administration Guide.

Troubleshooting WSO2 MB

For details on how you can troubleshoot and trace errors that occur in your WSO2 MB server, see Troubleshooting WSO2 MB.

Clustering

In a production environment, WSO2 Message Broker can be clustered and configured with an external Database Management System (DBMS) of your choice. For instructions on the clustered deployment, see Clustered Deployment.

Upgrading from a Previous Release

The instructions on this page take you through the steps for upgrading from MB 3.1.0 to MB 3.2.0. Note that you cannot rollback the upgrade process. However, it is possible to restore a backup of the previous database and restart the upgrade progress.

Preparing to upgrade

The following prerequisites must be completed before upgrading:

1. Make a backup of the databases used for MB 3.1.0.
2. Also, copy the <MB_HOME>_3.1.0 directory in order to backup the product configurations.

NOTE: The downtime is limited to the time taken for switching databases when in the production environment.

Upgrading from MB 3.1.0 to MB 3.2.0

WSO2 MB 3.2.0 comes with several database changes compared to WSO2 MB 3.1.0 in terms of the data format used for storing. We are providing a simple tool that you can easily download and run to carry out this upgrade.

Follow the steps given below to upgrade from WSO2 MB 3.1.0 to WSO2 MB 3.2.0.

1. Disconnect all the subscribers and publishers for WSO2 MB 3.2.0.
2. Shut down the server.
3. Run the migration script to update the database:
   1. Open a terminal and navigate to the <MB_HOME>/dbscripts/mb-store/migration-3.1.0_to_
3.2.0 directory.
2. Run the migration script relevant to your database type. For example, if you are using an Oracle, use the following script: oracle-mb.sql.
4. Download and run the migration tool:
   1. Download the migration tool.
   2. Unzip the org.wso2.mb.migration.tool.zip file. The directory structure of the unzipped folder is as follows:

   ```
   TOOL_HOME
   | - lib
   | -- config.properties
   | -- tool.sh
   | -- README.txt
   |-- org.wso2.carbon.mb.migration.tool.jar
   ```

3. Download the relevant database connector and copy it to the lib directory in the above folder structure. For example, if you are upgrading your MySQL databases, you can download the MySQL connector JAR from [http://dev.mysql.com/downloads/connector/j/5.1.html](http://dev.mysql.com/downloads/connector/j/5.1.html) and copy it to the lib directory.
4. Open the config.properties file from the org.wso2.mb.migration.tool.zip file that you downloaded in step 4 above and update the database connection details shown below.

   ```
   #Configurations for the database
   dburl=
   driverclassname=
   dbuser=
   dbpassword=
   ```

   The parameter in the above file are as follows:

   - **dburl**: The URL for the database. For example, jdbc:mysql://localhost/wso2_mb
   - **driverclassname**: The database driver class. For example, com.mysql.jdbc.Driver for MySQL.
   - **dbuser**: The user name for connecting to the database.
   - **dbpassword**: The password for connecting to the database.

5. Update the datasource connection for the MB database in the store master-datasources.xml file (stored in the <MB_HOME_320>/repository/conf/datasources directory).
6. Run the migration tool:

   1. If you are on a Linux environment, open a command prompt and execute the following command: tool.sh.
   2. If you are on a non-Linux environment, execute org.wso2.carbon.mb.migration.tool.jar manually.

5. Start WSO2 MB 3.2.0.
6. Reconnect all the publishers and subscribers to MB 3.2.0.

### Testing the upgrade

Verify that all the required scenarios are working as expected as shown below. This confirms that the upgrade is successful.

1. Make sure that the server starts up fine without any errors.
2. 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.
4. Verify that all the messages to the Queues and Topics have been successfully published in the MB 3.2.0 instance.
5. Run some of the samples to see that the system is working fine.

Performance Tuning Guide

This section describes some recommended performance tuning configurations to optimize the WSO2 Message Broker. It assumes that you have set up the MB on a server running Unix/Linux, which is recommended for a production deployment. It is recommended to have at least one MB server node for failover. Therefore, a clustered deployment is recommended for most production systems with at least two MB server nodes.

- OS-level settings
- JVM-level settings
- WSO2 Carbon platform-level settings
- MB-level settings

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 you to carry out load tests on your environment to tune the ESB 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.

   It is not recommended to use `net.ipv4.tcp_tw_recycle = 1` when working with network address translation (NAT), such as if you are deploying products in EC2 or any other environment configured with NAT.

```
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
```
2. To alter the number of allowed open files for system users, configure the following settings in /etc/security/limits.conf file of Linux (be sure to include the leading * character).

```
* soft nofile 4096
* hard nofile 65535
```

Optimal values for these parameters depend on the environment.

**JVM-level settings**

If one or more worker nodes in a clustered deployment require access to the management console, increase the entity expansion limit as follows in the `<MB_HOME>/bin/wso2server.bat` file (for Windows) or the `<MB_HOME>/bin/wso2server.sh` file (for Linux/Solaris). The default entity expansion limit is 64000.

```
-DentityExpansionLimit=10000
```

**WSO2 Carbon platform-level settings**

In multitenant mode, the WSO2 Carbon runtime limits the thread execution time. That is, if a thread is stuck or taking a long time to process, Carbon detects such threads, interrupts and stops them. Note that Carbon prints the current stack trace before interrupting the thread. This mechanism is implemented as an Apache Tomcat valve. Therefore, it should be configured in the `<PRODUCT_HOME>/repository/conf/tomcat/catalina-server.xml` file as shown below.

```
<Valve
   className="org.wso2.carbon.tomcat.ext.valves.CarbonStuckThreadDetectionValve"
   threshold="600"/>
```

- The `className` is the Java class used for the implementation. Set it to `org.wso2.carbon.tomcat.ext.valves.CarbonStuckThreadDetectionValve`.
- The `threshold` gives the minimum duration in seconds after which a thread is considered stuck. The default value is 600 seconds.

**MB-level settings**

The following sections describe how you can configure the MB-level settings to optimize performance.

- Clustering Performance
- Database Performance
- Message Publisher and Consumer Performance
- Tuning Flow Control

**Clustering Performance**

This section explains how to tune the performance of the WSO2 MB in terms of deployment in a clustered environment.

<table>
<thead>
<tr>
<th>Improvement area</th>
<th>Parameter</th>
<th>Description</th>
<th>Location</th>
</tr>
</thead>
</table>

Copyright © WSO2 Inc. 2005-2014
| Memory | `-Xms256m -Xmx1024m`  
|        | `-XX:MaxPermSize=256m` | The memory allocated for the WSO2 MB. | **For Windows:** `<MB_HOME>/bin/wso2server.bat`  
|        |                      |                                          | **For Linux:** `<MB_HOME>/bin/wso2server.sh` |
| Cluster health | `hazelcast.max.no.heartbeat.seconds` | The maximum time period that should elapse between pings received from a worker node in an MB cluster before acknowledging that worker node to be dead. This value is specified in seconds. | `<MB_HOME>/repository/conf/hazelcast.properties` |

This parameter prevents the allocation of resources to inactive worker nodes, thereby avoiding unnecessary system overheads.

A short time duration can be specified in environments with a very high message flow and it is important to reallocate resources from inactive nodes to active nodes fast. A longer time duration can be specified in environments with lower message flows.
Cluster Recovery

**concurrentStorageQueueReads**

The number of storage queue reads carried out concurrently at a given time. This number should be set based on the number of storage queues that currently exist.

<MB_HOME>/repository/conf/broker.xml

---

**Database Performance**

This section explains how to tune the database performance of MB by configuring various parameters in the master-datasources.xml file.

<table>
<thead>
<tr>
<th>Improvement Area</th>
<th>Description</th>
<th>Performance Recommendations</th>
</tr>
</thead>
<tbody>
<tr>
<td>The maximum number of active connections</td>
<td>This is specified by entering a value for the maxActive parameter.</td>
<td>This should be set in proportion to the number of messages published and consumed. If there are too many active connections in the connection pool, some of them would be idle, incurring an unnecessary system overhead. The default/recommended value is 100.</td>
</tr>
<tr>
<td>The maximum waiting time.</td>
<td>The maxWait parameter specifies the maximum number of milliseconds the MB waits before an existing active connection is returned in order to make a call to the database when no active connections are currently available. An exception is thrown once this time duration has elapsed.</td>
<td>A lower number of milliseconds can be set if you want the database to be updated faster. If you increase the waiting time, it will reduce the performance of the MB in terms of the publish/consumption rate. The default/recommended value is 30000.</td>
</tr>
<tr>
<td>Testing objects borrowed from the pool</td>
<td>Select true or false for the testOnBorrow parameter to indicate whether a validation test should be performed on objects in the pool before borrowing them.</td>
<td>This parameter should be set to true if it is important to ensure the reliability of connections borrowed. However, performing validation tests would incur a system overhead. The default/recommended value is false.</td>
</tr>
<tr>
<td>Validation interval</td>
<td>The validationInterval parameter specifies the minimum number of milliseconds that should elapse after a validation test performed on an object in the connection pool before another test is performed. This parameter is relevant only if the testOnBorrow parameter is set to true.</td>
<td>The purpose of the parameter is to control system overhead that can be caused by excessive validation tests. The default/recommended value is 30000 milliseconds. You can increase this value to minimise the system overhead or reduce it if it is more important to ensure the reliability of the connections.</td>
</tr>
</tbody>
</table>

**Message Publisher and Consumer Performance**

This section explains the impact of different parameters in the <MB_HOME>/repository/conf/broker.xml file on the rate at which messages are published by publishers, as well as the rate at which the messages are...
consumed by the subscribers.

<table>
<thead>
<tr>
<th>Improvement Area</th>
<th>Parameter</th>
<th>Description</th>
<th>Perform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading messages from database and allocation of slots to consumers. See slots in the broker.xml file.</td>
<td>windowSize</td>
<td>This parameter specifies the size of a slot. A publisher returns its last message ID to the slot manager each time he/she publishes a number of messages equal to the number specified in this parameter.</td>
<td>Increase number Thus the allocate message even listener subsrcrit 1000.</td>
</tr>
<tr>
<td></td>
<td>messageAccumulationTimeout</td>
<td>If message publishers are slow, the time taken to fill the slot (up to &lt;windowSize&gt;) will be longer. This will add a latency to messages. Therefore, the broker will mark the slot as 'ready to deliver' after the time period specified by this parameter (in milliseconds), regardless of whether the slot is completely filled.</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>maxSubmitDelay</td>
<td>This parameter specifies the time interval in which, broker checks for slots that can be marked as 'ready to deliver' (i.e. slots that age more than the time specified in message AccumulationTimeout). The nodes, which will have subscribers for a given queue/topic will periodically poll for slots during this interval.</td>
<td>If the consumer is fast, the maxSubmitDelay would result in reduced CPU and I/O overhead on the DB. If the consumers are slow, there will be an increase in latency.</td>
</tr>
<tr>
<td></td>
<td>deleteThreadCount</td>
<td>This value specifies the number of parallel threads that should be included in a slot deletion task.</td>
<td>Increase message consumer reducing heap memory used by the server.</td>
</tr>
<tr>
<td></td>
<td>SlotDeleteQueueDepthWarningThreshold</td>
<td>Maximum number of slots (with pending messages) to delete per Slot Deleting Task. This configuration is used to raise a warning when the scheduled number of pending slots exceeds this limit. This indicates issues that can lead to message accumulation on the server.</td>
<td>-</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>thriftClientPoolSize</td>
<td>Maximum number of thrift client connections that should be created in the thrift connection pool.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>maxNumberOfReadButUndeliveredMessages</td>
<td>This parameter specifies the maximum number of messages undelivered messages that are allowed to be retained in the memory.</td>
<td>The default value for this parameter is 1000. Increasing this value can cause out-of-memory exceptions, but performance will be improved because the number of database calls will be reduced. Therefore, your allocated server memory capacity should be considered when configuring this parameter.</td>
<td></td>
</tr>
<tr>
<td>ringBufferSize</td>
<td>This parameter specifies the thread pool size of the queue delivery workers.</td>
<td>The default value for this parameter is 4096. An increased ring buffer size is also required if the slot window size is large and therefore, there is a large number of messages to be delivered.</td>
<td></td>
</tr>
<tr>
<td>parallelContentReaders</td>
<td>This parameter specifies the number of parallel readers used to read content from the message store.</td>
<td>The default value for this parameter is 5. Increasing this value would increase the speed of the message sending mechanism, however, the load on the message store would also be increased. A higher number of cores is required to increase these values.</td>
<td></td>
</tr>
<tr>
<td>parallelDeliveryHandlers</td>
<td>This parameter specifies the number of parallel delivery handlers used to deliver messages to subscribers.</td>
<td>The default value for this parameter is 5.</td>
<td></td>
</tr>
<tr>
<td>parallelDecompressionHandlers</td>
<td>This parameter specifies the number of parallel decompression handlers used to decompress messages before they are sent to subscribers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>contentReadBatchSize</td>
<td>This parameter specifies the number of messages included in the content retrieval query.</td>
<td>The contentReadBatchSize parameter is used to minimise the number of I/O operations. When large messages (i.e. larger than 1MB) are exchanged, the batch size should be reduced to avoid network latency.</td>
<td></td>
</tr>
<tr>
<td><strong>Acknowledgement Handling:</strong> This is the phase where the delivery of messages that have reached the consumers is acknowledged.</td>
<td><strong>ackHandlerCount</strong></td>
<td>This parameter specifies the number of message acknowledgment handlers to process acknowledgments concurrently.</td>
<td>The default value of 1 for the <code>ackHandlerCount</code> parameter should be increased in a high throughput scenario with a relatively high amount of messages being delivered to the consumers. The value for this parameter can be decreased when the value specified for the <code>ackHandlerBatchSize</code> parameter is high and as a result, each individual acknowledgement handler can handle a higher number of acknowledgements. Note that increasing the number of acknowledgement handlers when the number of messages being delivered and acknowledged is low, or when the value specified for the <code>ackHandlerBatchSize</code> parameter is high can result in idle acknowledgement handlers incurring an unnecessary system overhead.</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td><strong>ackHandlerBatchSize</strong></td>
<td>This parameter specifies the maximum number of acknowledgements that can be handled by an acknowledgement handler.</td>
<td>The default value of 100 for the <code>ackHandlerBatchSize</code> parameter should be increased when there is an increase in the number of messages being delivered to consumers. If the number of acknowledgements that can be handled by each individual acknowledgement handler is too low in a high throughput scenario, it will be required to increase the number of acknowledgement handlers. This would increase the number of calls made to the database, thereby increasing the system overhead.</td>
<td></td>
</tr>
<tr>
<td><strong>maxUnckedMessages</strong></td>
<td>The message delivery from the MB server to the client will be temporarily paused if the number of messages of which the delivery is not acknowledged reaches the number specified for the <code>maxUnckedMessages</code> parameter.</td>
<td>Increasing the value specified for the <code>maxUnckedMessages</code> parameter can cause out-of-memory exceptions, but the performance will be improved due to the reduction in the number of calls to the database. Therefore, your allocated server memory capacity should be considered when configuring this parameter.</td>
<td></td>
</tr>
<tr>
<td><strong>Content Handling:</strong> Handling incoming content chunks.</td>
<td><strong>contentChunkHandlerCount</strong></td>
<td>This parameter specifies the number of handlers that should be available to handle content chunks concurrently.</td>
<td>The default value of 3 for the <code>contentChunkHandlerCount</code> parameter should be increased when there is a significant number of large messages being published. A low value can be specified when the value for the <code>maxContentChunkSize</code> parameter is high in such situations, each individual handler will be able to handle a higher amount of content chunks. Note that increasing this value in a scenario where there are not many large messages published or when the maximum content chunk size is high can result in idle handlers causing an unnecessary system overhead.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Notes</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>maxContentChunkSize</td>
<td>This parameter specifies the maximum column size of a content chunk handler.</td>
<td>The default value of 65500 for the maxContentChunkSize parameter should be increased if there is an increased number of large messages being published. If the value specified for this parameter is too low, the value for the contentChunkHandlerCount parameter will have to be increased. This would increase the system overhead due to an increased number of calls made to the database.</td>
<td></td>
</tr>
<tr>
<td>allowCompression</td>
<td>This parameter specifies whether or not content compression is enabled. If enabled, messages published to MB will be compressed before storing in the DB, to reduce the content size.</td>
<td>This parameter is set to false by default. To increase the performance of the message broker, you can change this value in 'true'. Note that enabling content compression would increase the speed of the message sending mechanism. However, if message contents are very small, enabling content compression will reduce the performance of the message broker. This is because the message content will expand due to this setting.</td>
<td></td>
</tr>
<tr>
<td>contentCompressionThreshold</td>
<td>This parameter specifies the maximum message content size of an uncompressed message.</td>
<td>The default value of contentCompressionThreshold is 1000 in bytes. It is not recommended to use values less than 13 bytes. If the value specified for this parameter is too low, it will cause expansion of the message size, due to the lack of repeated content. Thus, it will reduce the performance of the message broker. Values greater than 1000 would increase performance.</td>
<td></td>
</tr>
</tbody>
</table>

**Inbound Events:**
These are messaging events relating to publishers

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bufferSize</td>
<td>This parameter specifies the size of the Disruptor ring buffer for inbound event handling.</td>
<td>It is recommended to increase the value for the bufferSize parameter when there is an increase in the rate of publishing. The default/recommended value is 65536.</td>
</tr>
<tr>
<td>parallelMessageWriters</td>
<td>This parameter specifies the number of parallel writers used to write content to the message store.</td>
<td>Increasing the value for the parallelMessageWriters parameter increases the speed of the message receiving mechanism. However, it would also increase the load on the data store. A higher number of cores are required to increase this value. The default/recommended value is 1.</td>
</tr>
<tr>
<td>messageWriterBatchSize</td>
<td>This parameter specifies the maximum batch size of the batch write operation for inbound messages.</td>
<td>The messageWriterBatchSize parameter should be used in high throughput scenarios to avoid database requests with a high load. A higher number of cores are required to increase this value. The default/recommended value is 70.</td>
</tr>
</tbody>
</table>
### purgedCountTimeout

This parameter specifies the number of milliseconds to wait for a queue to complete a purge event in order to update the purge count. If the time specified here elapses before the queue completes the purge event, the purge count will not be updated to include the event.

### Failover

**vHostSyncTaskInterval**

This parameter specifies the time interval after which the virtual host syncing task can sync host details across the cluster.

### Message Counter

**counterTaskInterval**

This parameter specifies the delay which should occur between the end of one execution and the start of another, in milliseconds.

### Message Deletion

**contentRemovalTaskInterval**

This parameter specifies the ask interval for the content removal task which will remove the actual message content from the store in the background.

### Tuning Flow Control

Flow control is typically employed for controlling fast producers from overloading slow consumers in producer-consumer scenarios. There may be several reasons for a fast producer-slow consumer scenario. For example, the consumer can be on a low resource footprint; or the message broker, which lies in the middle of the producer and the consumer, may get overloaded at a particular moment due to message accumulation within the broker itself. This can cause message broker instances to run out of resources, such as memory.

WSO2 Message Broker primarily supports buffer limit-based flow controlling. This involves blocking message acceptance when the rate at which messages are transmitted reaches a *high limit*, and unblocking it when this rate reaches a *low limit*.

This is further illustrated in the following matrix.

<table>
<thead>
<tr>
<th>Global</th>
<th>Per-publisher</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High Limit</strong></td>
<td>When the total number of message content chunks reach the <em>global high limit</em>, flow control is enabled and message acceptance is blocked for all the publishers.</td>
</tr>
</tbody>
</table>
If flow control is currently enabled, it will be disabled only when the total number of message content chunks reach the **global low limit**. Once this limit is reached, all the publishers are notified so that they can resume sending messages.

If flow control is currently enabled for an individual publisher, it will be disabled only when the total number of message content chunks reach the **per-publisher low limit**. Once this limit is reached, the publisher will be notified so that he/she can resume sending messages.

This section explains how to tune the performance of the MB in relation to flow control. The parameters described below can be configured in the `broker.xml` file.

<table>
<thead>
<tr>
<th>Improvement Area</th>
<th>Description</th>
<th>Performance Recommendations</th>
</tr>
</thead>
</table>
| Enabling/disabling flow control based on message content chunk limits | You can set the global limits and channel-specific (i.e., per publisher) limits for message content chunks in the `broker.xml` file as shown below.  
- To define the global message content limits, set the `<lowLimit>` and `<highLimit>` parameters as shown below. These limits are applicable to all channels collectively.  
  ```xml
  <flowControl>
    <!-- This is the global buffer limits which enable/disable the flow control globally -->
    <global>
      <lowLimit>800</lowLimit>
      <highLimit>8000</highLimit>
    </global>
  </flowControl>
  ```  
- To define the channel specific buffer limits, set the `<lowLimit>` and `<highLimit>` parameters as shown below. These limits will be applicable to each channel.  
  ```xml
  <flowControl>
    <!-- This is the channel specific buffer limits which enable/disable the flow control locally. -->
    <bufferBased>
      <lowLimit>100</lowLimit>
      <highLimit>1000</highLimit>
    </bufferBased>
  </flowControl>
  ```  | Having a large number as the higher limit would increase the number of messages stored in memory before they are stored in databases. This would result in a higher overall message publishing rate, but with reduced reliability.  
If the difference between the higher limit and the lower limit is too small, it would cause frequent enabling and disabling of flow control. This would reduce the overall message publishing rate.

Default or recommended values are as follows.  
**Global limits:**  
- Low limit: 800  
- High limit: 8000  
**Buffer based limits:**  
- High limit: 100  
- Low limit: 1000
Enabling/disabling flow control based on the memory

The `globalMemoryRecoveryThresholdRatio` parameter allows you to specify the memory consumption ratio at which flow control should be enabled (if it is currently disabled). This ratio is calculated using the following formula.

\[
\text{Used Memory/Allocated Memory}
\]

You can also specify the time interval at which the server should check whether the above ratio is reached via the `memoryCheckInterval` parameter.

If the publisher throughput is very high compared to the consumer throughput, this ratio should be reduced (to a maximum value of 1) to avoid out of memory scenarios. At the same time, the value for the `memoryCheckInterval` parameter should be reduced to perform more frequent checks on memory availability.

Troubleshooting WSO2 Message Broker

You can troubleshoot and trace possible errors that can occur with WSO2 Message Broker (WSO2 MB) in a given environment by using the methods given below.

- Debugging
- Message tracing
- Head dump and thread stack analysis
- Using wireshark to analyze protocol communication
- Detecting database anomalies
- Retrieving logs from the JMS client
- Monitoring JAVA metrics
- Identifying common warnings/logs

Debugging

The following table provides descriptions of the important classes in WSO2 MB that will be useful when you debug a session.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inbound</td>
<td>org.wso2.andes.kernel.disruptor.inbound.InboundEventManager</td>
</tr>
<tr>
<td></td>
<td>org.wso2.andes.kernel.disruptor.inbound.MessagePreProcessor</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.disruptor.inbound.ContentChunkHandler</td>
<td>This processor will take the message content chunks, convert them to the andes core chunk size and delegate the rest of the work to the MessageWriter.</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.disruptor.inbound.MessageWriter</td>
<td>This processor will write the message metadata and content chunks to the storage database using a batch approach.</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.disruptor.inbound.StateEventHandler</td>
<td>Upon saving the message to storage, this handler is triggered to notify a message received event, or to notify a message acknowledged event from the consumer.</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.disruptor.inbound.InboundTransactionEvent</td>
<td>This event is used to communicate the transaction commit/rollback events from the publisher.</td>
</tr>
<tr>
<td><strong>Outbound</strong></td>
<td></td>
</tr>
<tr>
<td>org.wso2.andes.kernel.disruptor.delivery.DeliveryEventHandler</td>
<td>This processor is used to deliver the message to one/all of the active subscriptions (based on the message destination).</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.MessageFlusher</td>
<td>This class is used to handover the message to the consumer after reading from the internal message buffer (readButUndeliveredMessages).</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.slot.SlotDeliveryWorker</td>
<td>There are multiple slot delivery workers managed by the SlotDeliveryWorkerManager. These will read messages from the database after selecting a slot range from the coordinator. The messages are then pushed to the message flusher for delivery.</td>
</tr>
<tr>
<td>org.wso2.andes.kernel.slot.SlotManagerClusterMode</td>
<td>This is where the coordinator logic resides within WSO2 MB. All slots are managed and distributed through this class across the cluster.</td>
</tr>
</tbody>
</table>
### AMQP

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.wso2.andes.server.AMQChannel</td>
<td>A channel is used for delivering and accepting messages to/from the broker. Each AMQP consumer/publisher has its own unique channel with a channel ID.</td>
</tr>
<tr>
<td>org.wso2.andes.amqp.QpidAndesBridge</td>
<td>This is used as the bridge between the Qpid messaging events and Andes events.</td>
</tr>
</tbody>
</table>

### MQTT

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.dna.mqtt.wso2.AndesMQTTBridge</td>
<td>This is used as the bridge between the moquette messaging events and Andes events.</td>
</tr>
<tr>
<td>org.dna.mqtt.moquette.messaging.spi.impl.ProtocolProcessor</td>
<td>This handles all events coming through the moquette disruptor (such as subscriber-connect, pub-acks etc.) and connects to the AndesMQTTBridge as required to bridge the MQTT functionality.</td>
</tr>
<tr>
<td>org.wso2.andes.mqtt.connectors.PersistenceStoreConnector</td>
<td>This class acts as an interface before storing MQTT messages to the message store, validating the message format, in addition to handling events such as consumer/publisher creation and closing in terms of the message store.</td>
</tr>
<tr>
<td>org.wso2.andes.mqtt.MQTTTopicManager</td>
<td>This class handles the lifecycle of MQTT subscriptions and also takes part in routing a given message to matching subscribers.</td>
</tr>
</tbody>
</table>

### Message tracing

This is an MB-specific logging implementation for tracing a message through its inbound event until it is delivered to the consumer application. This implementation has minimal impact on the performance of the broker functionality. To enable message tracing in WSO2 MB:

1. Open the `log4j.properties` file stored in the `<MB_HOME>/repository/conf` folder.
2. Uncomment the following:

```sh
#log4j.logger.org.wso2.andes.tools.utils.MessageTracer=TRACE,CARBON_TRACE_LOGFILE
```

Once message tracing is enabled, you can start the server and execute a `grep` command with the relevant
message ID you want to trace. This will print all the logs related to your message ID on your terminal.

**Head dump and thread stack analysis**

As with any other java product, if the MB cluster fails due to a resource exhaustion, the heap and thread dumps will always point you towards the cause of the leak. Therefore, it is important to be able to retrieve heap and thread dumps from an environment at the point when an error occurs. This will avoid the necessity of reproducing the exact issue again (specially in case of production issues). A resource exhaustion can happen for two reasons:

- Due to a bug in the system.
- An actual limitation of resources based on low configuration values.

You can easily create a a heap dump and thread dump using the CarbonDump tool that is shipped with your product. These will also provide information about the product version and any patch inconsistencies.

**Using wireshark to analyze protocol communication**

Wireshark is a network traffic analysis tool with great filtering features. Given that WSO2 MB uses the AMQP and MQTT protocols (which are different from HTTP), wireshark is a good way of capturing the network traffic and verifying if the packets are going in the expected order with correct data.

**Detecting database anomalies**

This section explains how you can identify errors by evaluating the condition of the database. Even though most of the database schema is self explanatory, it is still good to know the special cases where the slot ranges are being stored and how the safe zone is being evaluated. The following diagram illustrates the slot-based message delivery algorithm:

![Diagram illustrating slot-based message delivery](image)

Given that the coordinator is the decision maker on all operations, information on slots are also required to be maintained in a central location. Therefore, all the slot related information in the database are stored in mainly four tables as shown below.
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB_SLOT</td>
<td>Each slot, the assigned node ID and the current status are maintained here.</td>
</tr>
<tr>
<td>MB_SLOT_MESSAGE_ID</td>
<td>Whenever a node communicates a possible slot range to the coordinator, the node will decide on the appropriate message ID range to be included in the slot and update this table with the last <code>endMessageID</code> submitted by the node for a given queue/topic.</td>
</tr>
<tr>
<td>MB_NODE_TO_LAST_PUBLISHED_ID</td>
<td>This table contains the last published message ID for each node in order to calculate the global safe zone (minimum messageID from all nodes) that is required for deleting slots upon completion.</td>
</tr>
<tr>
<td>MB_QUEUE_TO_LAST_ASSIGNED_ID</td>
<td>Whenever a slot is given by the coordinator to an MB node for processing, its <code>endMessageID</code> is updated in this table against the destination name.</td>
</tr>
</tbody>
</table>

With the above information, you can infer the following validations in the database at any given time:

1. There should not be any slots in the `MB_SLOT` table if the `MB_METADATA` table is empty. This is an eventual guarantee. Even if there are slots queued for deletion, this rule must still be satisfied after some time.

2. There should be no records in the `MB_METADATA` table if the `MB_CONTENT` table is empty (one-to-one relationship).

3. Given the minimum message ID in the `MB_NODE_TO_LAST_PUBLISHED_ID` table, all slots within the `MB_SLOT` table with the “assigned” status (state = 2) and the `endMessageID` less than the minimum published ID should be deleted (or at-least be cleared after some time).

**Retrieving logs from the JMS client**

You can simply monitor the logs from the JMS clients connecting to WSO2 MB by enabling the following startup property on the clients:

```
-Damqj.protocol.logging.level=true
```

**Monitoring JAVA metrics**

The metrics dashboard of WSO2 MB provides general JVM metrics as well as MB-specific metrics to help you identify how the broker is running in a loaded/relaxed environment. This functionality will give you information such as the unexpected increases of delivery channels, latencies of database reads/writes etc., which will help you identify possible errors in the system. See the documentation on metrics for instructions on how to configure and use the metrics dashboard.

**Identifying common warnings/logs**

The following table details some of the most common warning messages/logs that can be encountered when working with WSO2 MB. You will also find here the possible causes and solutions for such warnings/logs.
<table>
<thead>
<tr>
<th>Warning</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>[WARN] Invalid message state transition from <code>&lt;state1&gt;</code> suggested: <code>&lt;state2&gt;</code> Message ID: 93293291982</td>
<td>This means that the message lifecycle has deviated from the expected execution path. Example: getting <code>SCHEDULED_TO_SEND</code>.</td>
</tr>
<tr>
<td>[WARN] Invalid State transition from <code>&lt;stateA&gt;</code> suggested: <code>&lt;stateB&gt;</code> Slot ID: `MyQueue</td>
<td>22309482...`</td>
</tr>
<tr>
<td>[WARN] Error when trying to read property <code>&lt;property&gt;</code>. Switching to default value: <code>&lt;defaultValue&gt;</code></td>
<td>This can happen if the MB configuration file (<code>broker.xml</code>) is not up to date, or if it has been edited.</td>
</tr>
<tr>
<td>[INFO] Local subscription ADDED [TestQueue]ID=635@NODE/10.100.5.115:4000/T=1456518837302/D=true/X=false/O=null/E=amq.direct/ET=org.wso2.andes.server.exchange.DirectExchange$1@2db707df/EUD=0/S=true {org.wso2.andes.subscription.SubscriptionStore}</td>
<td>This log is printed whenever a new queue/topic subscription is added to the cluster. The lifecycle of a subscription is <code>ADDED -&gt; DELETED</code>. In the case of durable subscriptions, a subscription may be disconnected before being deleted. If a subscription is disconnected, the messages will still persist until the subscriber is deleted. Note that this log is not printed if a durable subscription is disconnected and added for the second time after the cluster starts.</td>
</tr>
</tbody>
</table>
Configuring Transports for WSO2 MB

WSO2 Message Broker (WSO2 MB) uses two transport protocols for the purpose of brokering messages between publishers and subscribers. These protocols are the Advanced Message Queueing Protocol (AMQP) and the Message Queueing and Telemetry Transport (MQTT).

The following topics explain how these protocols are enabled and configured for WSO2 MB:

- **Advanced Message Queuing Protocol (AMQP)**
  - Enabling the transport
  - Configuring the SSL connection
  - Related Parameters
- **Message Queueing and Telemetry Transport (MQTT)**
  - Enabling the transport
  - Configuring the SSL connection
  - Configuring authentication and authorization

**Advanced Message Queuing Protocol (AMQP)**

The Advanced Message Queueing Protocol (AMQP) is a wire-level messaging protocol used by WSO2 MB for message queueing. The `<MB_HOME>/repository/conf/broker.xml` file contains parameters related to configuring the AMQP transport.

**Enabling the transport**

The AMQP transport is enabled by default, as shown in the following extract of the `broker.xml` file:

```xml
<amqp enabled="true">
  <defaultConnection enabled="true" port="5672" />
</amqp>
```

As shown above, if the value of this parameter is `true`, the AMQP transport is enabled and the AMQP protocol will be applied to messages sent to the specified listening port. The default listening port specified for the AMQP transport is 5672. That is, the AMQP broker will be initialized with this port by default. This value will be incremented based on the offset specified in the `carbon.xml`.

**Configuring the SSL connection**

You can configure a SSL connection for the AMQP transport using the `<sslConnection>` element in the `broker.xml` file. See Enabling SSL Support in the Broker for information.

**Related Parameters**
Once you have enabled the AMQP transport, you can update the following related parameters as required.

```xml
<amqp enabled="true">
    ............
    <maximumRedeliveryAttempts>10</maximumRedeliveryAttempts>
    <allowSharedTopicSubscriptions>false</allowSharedTopicSubscriptions>
    <allowStrictNameValidation>true</allowStrictNameValidation>
    ............
</amqp>
```

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>maximumRedeliveryAttempts</td>
<td>The maximum number of times WSO2 MB should attempt to redeliver a message that has not reached a subscriber. For example, when this value is set to 10, another 10 attempts will be made to deliver the message. The default value can be changed depending on your reliability requirements. Read more about message redelivery.</td>
<td>10</td>
</tr>
<tr>
<td>allowSharedTopicSubscriptions</td>
<td>If this parameter is true, a durable subscription to a topic can be shared among multiple subscribers. That is, multiple clients can subscribe to a topic in WSO2 MB using the same client ID. Read more about durable subscriptions to topics.</td>
<td>false</td>
</tr>
<tr>
<td>allowStrictNameValidation</td>
<td>If this parameter is true, the queue names and topic names will be validated according to the AMQP specification. When this parameter is set to false, it is possible to use &quot;;&quot; in topic names. Read more about this in ‘Adding topics from management console’.</td>
<td>true</td>
</tr>
</tbody>
</table>

Message Queueing and Telemetry Transport (MQTT)

The Message Queueing and Telemetry Transport (MQTT) is a lightweight, broker-based publish/subscribe messaging protocol, which enables an extremely lightweight publish/subscribe messaging model. WSO2 MB 3.0.0 and later versions fully support MQTT version 3.1.0, and partially supports version 3.1.1.

The MQTT protocol allows a message to be sent to a topic based on three levels of QoS (Quality of Service) as explained below.

- **QoS 1 - At Most One** - At this level, messages are delivered to subscribers in the most efficient manner. A message is dispatched only once.
- **QoS 2 - At Least One** - At this level, the system will ensure that a message is received by the subscriber at least once. The level of delivery is assured through acknowledged delivery.
- **QoS 3 - Exactly Once** - At this level, the message is delivered only once to its subscriber. This level is also defined as Assured Delivery.

Just as the AMQP transport, the MQTT transport can be configured using the `<MB_HOME>/repository/conf/broker.xml` file.

Enabling the transport

The MQTT transport is enabled by default, as shown in the following extract of the `broker.xml` file:
As shown above, If the value for this parameter is true, the MQTT transport is enabled and the MQTT protocol will be applied to messages that are sent to the specified listening port. The listening port for the MQTT transport is 1883. The MQTT broker will be initialized with this specified port by default. This value will be incremented based on the offset specified in the carbon.xml.

### Configuring the SSL connection

You can configure a secure SSL connection for the MQTT transport using the `<sslConnection>` element in the `broker.xml` file. See Enabling SSL Support in the Broker for information.

### Configuring authentication and authorization

Authentication and authorization of the MQTT connection can be configured using the following settings in the `broker.xml` file.

```xml
<mqtt enabled="true">
    ............
    <security>
        <authentication>OPTIONAL</authentication>
        <authenticator>org.wso2.carbon.andes.authentication.andes.CarbonBasedMQTTAuthenticator</authenticator>
        <!--authenticator class="org.wso2.carbon.andes.authentication.andes.OAuth2BasedMQTTAuthenticator">
            <property name="hostURL">https://localhost:9443/services/OAuth2TokenValidationService</property>
            <property name="username">admin</property>
            <property name="password">admin</property>
            <property name="maxConnectionsPerHost">10</property>
            <property name="maxTotalConnections">150</property>
        </authenticator-->
        <authorization>NOT_REQUIRED</authorization>
        <authorizer class="org.wso2.carbon.andes.authorization.andes.CarbonPermissionBasedMQTTAuthorizer">
            <property name="connectionPermission">/permission/admin/mqtt/connect</property>
        </authorizer>
    </security>
</mqtt>
```

The above configurations are explained below:
• The `<authentication>` element instructs the MQTT server on whether clients should always send credentials when establishing a connection. Possible values are as follows:

<table>
<thead>
<tr>
<th>OPTIONAL</th>
<th>This is the default value. If an MQTT client sends credentials, the server will validate them. If the client does not send credentials, the server will allow the client to establish the connection without authentication. This behavior adheres to the MQTT 3.1 specification.</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQUIRED</td>
<td>If the MQTT client doesn't send credentials or if the credentials are invalid, the server will reject the connection. Note that if authentication is REQUIRED, the permissions linked to the credentials may also be checked depending on the value specified for <code>&lt;authorization&gt;</code> element.</td>
</tr>
</tbody>
</table>

• The `<authenticator>` element specifies the class that implements authentication. By default, the `org.wso2.carbon.andes.authentication.andes.CarbonBasedMQTTAuthenticator` class is enabled, which authenticates the user's credentials against the Carbon user store.

If required, you can disable the default authenticator and enable the `org.wso2.carbon.andes.authentication.andes.OAuth2BasedMQTTAuthenticator` authenticator class as shown below. This class enables OAuth-based authentication and authorization for MQTT.

```xml
<mqtt enabled="true">
    ..............
    <security>
        ............
        <authenticator class="org.wso2.carbon.andes.authentication.andes.OAuth2BasedMQTTAuthenticator">
            <property name="hostURL">https://localhost:9443/services/OAuth2TokenValidationService</property>
            <property name="username">admin</property>
            <property name="password">admin</property>
            <property name="maxConnectionsPerHost">10</property>
            <property name="maxTotalConnections">150</property>
        </authenticator>
    </security>
    ..............
</mqtt>
```

• The `<authorization>` element instructs the MQTT server on whether clients should have permission to publish messages to the broker or to subscribe to the broker. Possible values are as follows:

<table>
<thead>
<tr>
<th>NOT_REQUIRED</th>
<th>This is the default value. The MQTT client does not require permission for the purpose of publishing messages or to subscribe.</th>
</tr>
</thead>
</table>
The permissions granted to the MQTT client will be checked before allowing the client to publish messages. This check will execute the class given in the `<authorizer>` element that is explained below. Note that the `<authentication>` element should be set to REQUIRED for authorization to be REQUIRED.

- The `<authorizer>` element specifies the permissions required by a user to connect to the broker. This is applicable if the `<authorization>` element is set to REQUIRED.

```
<mqtt enabled="true">
    ..............
    <security>
        ........
        <authorizer
            class="org.wso2.carbon.andes.authorization.andes.CarbonPermissionBase
            dMQTTAuthorizer">
            <property
              name="connectionPermission">/permission/admin/mqtt/connect</property>
        </authorizer>
        </security>
    </mqtt>
```

### Changing the Default MB Database

In addition to the Carbon database, WSO2 Message Broker (WSO2 MB) requires a separate database for storing data that is specific to WSO2 MB. By default, WSO2 MB is shipped with an embedded H2 database (`WSO2MB_DB.h2.db`) for this purpose. These databases are stored in the `<MB_HOME>/repository/database` directory.

For instructions on changing the default Carbon database, see Changing the Carbon Database in the WSO2 Product Administration Guide.

Given below are the steps you need to follow in order to change the default WSO2 MB database.

- Setting up the database
- Creating the datasource connection
  - Optional DB configurations
- Updating other configuration files
- Creating database tables

### Setting up the database

You can set up one of the following databases for storing WSO2 MB data:

- Setting up a MySQL database
- Setting up an MS SQL database
- Setting up an Oracle database
Creating the datasource connection

A datasource is used to establish the connection to a database. By default, the WSO2_MB_STORE_DB datasource is configured in the master-datasources.xml file for the purpose of connecting to the default H2 database that stores MB-specific data.

After setting up the MySQL database to replace the default H2 database, either change the default configurations of the WSO2_MB_STORE_DB datasource, or configure a new datasource to point it to the new database as explained below.

Follow the steps below.

1. Open the <PRODUCT_HOME>/repository/conf/datasources/master-datasources.xml file and locate the <datasource> configuration element.

2. You simply have to update the url pointing to your database, the username and password required to access the database and the driver details as shown below. Further, be sure to disable auto committing by setting the <defaultAutoCommit> element to false for the MB database. When auto committing is disabled, multiple SQL statements will be committed to the database as one transaction, as opposed to committing each SQL statement as an individual transaction.

Optionally, you can update the other elements for your database connection.

- MySQL
- MS SQL
- Oracle

```
<datasource>
  <name>WSO2_MB_STORE_DB</name>
  <description></description>
  <jndiConfig>
    <name>jdbc/WSO2MBStoreDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://localhost:3306/wso2_mb</url>
      <username>wso2carbon</username>
      <password>wso2carbon</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>
```
<datasource>
  <name>WSO2_MB_STORE_DB</name>
  <description></description>
  <jndiConfig>
    <name>WSO2MBStoreDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:jtds:sqlserver://localhost:1433/wso2_mb</url>
      <username>sa</username>
      <password>sa</password>
      <driverClassName>com.microsoft.sqlserver.jdbc.SQLServerDriver</driverClassName>
      <maxActive>200</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>

<datasource>
  <name>WSO2_MB_STORE_DB</name>
  <description></description>
  <jndiConfig>
    <name>WSO2MBStoreDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
      <url>jdbc:oracle:thin:@localhost:1521/orcl</url>
      <maxActive>100</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1 FROM DUAL</validationQuery>
      <validationInterval>30000</validationInterval>
      <username>scott</username>
      <password>tiger</password>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>
## Optional DB configurations

<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>
<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>
</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](#).

### Updating other configuration files

1. Open the `<MB_HOME>/repository/conf/broker.xml` file. This is the root configuration file of WSO2 MB. The changes made to this file must be done in all the MB nodes.
2. In the `broker.xml` file we need to use the Oracle message store and Andes context store. To do this, uncomment or add the following configuration.

```xml
...  
<messageStore class="org.wso2.andes.store.rdbms.RDBMSMessageStoreImpl">
   <property name="dataSource">WSO2MBStoreDB</property>
</messageStore>

<andescontextStore class="org.wso2.andes.store.rdbms.RDBMSAndesContextStoreImpl">
   <property name="dataSource">WSO2MBStoreDB</property>
</andescontextStore>
... 
```
The elements in the above configuration are described below.

- The fully qualified name of the respective implementation class should be defined under the class attributes of `messageStore` and `andesContextStore` elements. This implementation class will be used by MB to persist relevant information.

- The `<property>` elements are used to define different properties for each store. The minimal property for starting each store is the `dataSource` property. Depending on the implementation, the required properties may differ.

**Creating database tables**

To create the database tables, connect to the database that you created earlier and run the following scripts:

- MySQL
- MS SQL
- Oracle

To create the database tables, connect to the database that you created earlier and run the following scripts.

1. To create tables in the MB-specific database (`WSO2_MB`), use the below script:

   ```
   mysql -u root -p -DWSO2_MB <
   '<WSO2MB_HOME>/dbscripts/mb-store/mysql-mb.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: `<MB_HOME>/bin/wso2server.bat -Dsetup`
   - For Linux: `<MB_HOME>/bin/wso2server.sh -Dsetup`

To create the database tables, connect to the database that you created earlier and run the following scripts.

1. To create tables in MB-specific database (wso2mb), use the below script:

   ```
   <WSO2MB_HOME>/dbscripts/mb-store/mssql-mb.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: `<MB_HOME>/bin/wso2server.bat -Dsetup`
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 MB-specific database, use the below script:

   ```sql
   SQL> @$<MB_HOME>/dbscripts/mb-store/oracle-mb.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: `<MB_HOME>/bin/wso2server.bat -Dsetup`
- For Linux: `<MB_HOME>/bin/wso2server.sh -Dsetup`

**User Permissions for WSO2 MB**

This section explains in detail how the Management Console of a WSO2 product can be used for configuring the permissions granted to a user role. You will also find detailed descriptions on all the types of permissions that can be granted.

- **Introduction to role-based permissions**
- **Configuring permissions for a role**
- **Descriptions of permissions**
  - Log-in permissions
  - Super Tenant permissions
  - Tenant-level permissions
    - Configuration permissions
    - Permissions for managing Queues and Topics
    - General management permissions
    - Permissions for monitoring

**Introduction to role-based permissions**

The **User Management** module in WSO2 products enable role-based access. With this functionality, the permissions enabled for a particular role determines what that user can do using the Management Console of a WSO2 product. Permissions can be granted to a role at two levels:

- **Super tenant level**: A role with super tenant permissions is used for managing all the tenants in the system and also for managing the key features in the system, which are applicable to all the tenants.
- **Tenant level**: A role with tenant level permissions is only applicable to individual tenant spaces.

By default, every WSO2 product comes with the following User, Role and Permissions configured:

- The **Admin** user and **Admin** role is defined and linked to each other in the user-mgt.xml file, stored in the `<PRODUCT_HOME>/repository/conf/` directory as shown below.
<AddAdmin>true</AddAdmin>
<AdminRole>admin</AdminRole>
<AdminUser>
    <UserName>admin</UserName>
    <Password>admin</Password>
</AdminUser>

- The Admin role has all the permissions in the system enabled by default. Therefore, this is a super tenant, with all permissions enabled.

You will be able to log in to the Management Console of the product with the Admin user defined in the `<user-mgt.xml>` file. You can then create new users and roles and configure permissions for the roles using the Management Console. However, note that you cannot modify the permissions of the Admin role. The possibility of managing users, roles and permissions is granted by the User Management permission. See the documentation on configuring the system administrator for more information.

### Configuring permissions for a role

To configure the permissions for a role:

1. Click Users and Roles in the Configure tab of the navigator. All the roles created in the system will be listed in the Roles page as shown below.

   ![Roles Page](image)

   - **Name**: `admin`  
   - **Actions**: Assign Users, View Users

2. Click Permissions to open the permissions navigator for the role as shown below.

   ![Permissions Navigator](image)
Note that there may be other categories of permissions enabled for a WSO2 product, depending on the type of features that are installed in the product.

3. You can select the relevant check boxes to enable the required permissions for your role. The descriptions of all the available permissions are explained below.

### Descriptions of permissions

Let us now go through each of the options available in the permissions navigator to understand how they apply to functions in WSO2 MB.

- **Log-in permissions**
- **Super Tenant permissions**
- **Tenant-level permissions**
  - **Configuration permissions**
  - **Permissions for managing Queues and Topics**
  - **General management permissions**
  - **Permissions for monitoring**

### Log-in permissions

The **Login** permission defined under **Admin** permissions allows users to log in to the Management Console of the product. Therefore, this is the primary permission required for using the Management Console.

### Super Tenant permissions

The following table describes the permissions at **Super Tenant** level. These are also referred to as **Super Admin** permissions.

<table>
<thead>
<tr>
<th>Permission</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Configuration permissions:</strong></td>
<td>The <strong>Super Admin/Configuration</strong> permissions are used to grant permission to the key functions in a product server, which are common to all the tenants. In each WSO2 product, several configuration permissions will be available depending on the type of features that are installed in the product.</td>
</tr>
<tr>
<td><strong>Feature Management</strong></td>
<td>- <strong>Feature Management</strong> permission ensures that a user can control the features installed in the product using the Management Console. That is, the <strong>Features</strong> option will be enabled under the <strong>Configure</strong> menu. See the topic on feature management for more information.</td>
</tr>
<tr>
<td><strong>Logging</strong></td>
<td>- <strong>Logging</strong> permission enables the possibility to configure server logging from the Management Console. That is, the <strong>Logging</strong> option will be enabled under the <strong>Configure</strong> menu. See the topic on logging management for more information.</td>
</tr>
</tbody>
</table>
Management permissions:

The **Super Admin/Manage** permissions are used for adding new tenants and monitoring them.

- **Modify/Tenants** permission enables the **Add New Tenant** option in the **Configure** menu of the Management Console, which allows users to add new tenants.
- **Monitor/Tenants** permission enables the **View Tenants** option in the **Configure** menu of the Management Console.

See the topic on configuring multiple tenants for more information.

Server Admin permissions:

Selecting the **Server Admin** permission enables the **Shutdown.Restart** option in the **Main** menu of the Management Console.

Tenant-level permissions

The following table describes the permissions at **Tenant** level. These are also referred to as **Admin** permissions.

Note that when you select a node in the **Permissions** navigator, all the subordinate permissions that are listed under the selected node are also automatically enabled.

Configuration permissions

The following table explains the permissions required for performing various configuration tasks in the WSO2 MB.

<table>
<thead>
<tr>
<th>Permission level</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin/Configure</td>
<td>When the <strong>Admin/Configure</strong> permission node is selected, the following menus are enabled in the Management Console:</td>
</tr>
<tr>
<td></td>
<td>- <strong>Configure</strong> menu/<strong>Datasources</strong>: Not applicable to MB.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Configure</strong> menu/Server Roles**: Not applicable to MB.</td>
</tr>
<tr>
<td></td>
<td>- Additionally, all permissions listed under <strong>Configure</strong> in the permissions navigator are selected automatically.</td>
</tr>
</tbody>
</table>
Admin/Configure/Security

When the Admin/Configure/Security permission node is selected, the following menus are enabled in the Configure menu of the Management Console:

- **Keystores**: See the topic on managing keystores for information.
- This permission will also enable the Roles option under Configure/Users and Roles. See the topic on configuring users, roles and permissions for more information.
- Additionally, all permissions listed under Security in the permissions navigator are selected automatically.

Admin/Configure/Security/Identity Management/User Management

This permission enables the possibility to add users from the Management Console. That is, the Users option will be enabled under Configure/Users and Roles.

Admin/Configure/Security/Identity Management/Profile Management

This permission enables the profiles of all the users. You can view the profile in the Configure tab, Users and Roles -> Users link.

Admin/Configure/Security/Identity Management/Password Management

This permission enables the Change Password option for the users listed in the User Management/Users and Roles/Users screen, which allows the log in user to change the passwords.

### Permissions for managing Queues and Topics

WSO2 Message Broker is primarily used for brokering messages between external applications using Queues and Topics. Explained below are the role-based permissions applicable for working with queues and topics in WSO2 MB.

- Permissions required for working with **Queues**:

<table>
<thead>
<tr>
<th>Permission level</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin/Queue/Add</td>
<td>This permission enables the option to Add queues. You will be able to add new queues and view a list of the available queues with this permission. To be able to delete, purge messages to a queue or browse details of a queue, you need the following permissions. Note that a user that has permission to Add new queues, by default obtains permission to consume messages from all queues created by the same user and to publish messages to the same queues.</td>
</tr>
<tr>
<td>Admin/Manage/Queue/Browse</td>
<td>This permission enables the Browse option for Queues. When you go to the Main tab and click Queues -&gt; List, you will see the Browse link enabled for each queue.</td>
</tr>
<tr>
<td>Admin/Manage/Queue/Delete</td>
<td>This permission enables the Delete option for Queues. When you go to the Main tab and click Queues -&gt; List, you will see the Delete link enabled for each queue.</td>
</tr>
<tr>
<td>Admin/Manage/Queue/Purge</td>
<td>This permission enables the Purge Messages option for Queues. When you go to the Main tab and click Queues -&gt; List, you will see the Purge Messages link enabled for each queue.</td>
</tr>
</tbody>
</table>
### Admin/Manage/Dead Letter Channel

This permission enables users to see any queue information that is stored in the Dead Letter Channel. When this node is selected, the following permissions will be automatically granted:

- **Browse**: Allows users to browse details of a queue stored in the Dead Letter Channel.
- **Delete**: Allows users to delete any queue stored in the Dead Letter Channel.
- **Reroute**: Allows users to reroute a queue stored in the Dead Letter Channel to any other queue chosen by the user.
- **Restore**: Allows users to restore a queue stored in the Dead Letter Channel to the queue from which it originated.

### Permissions required for working with Topics:

<table>
<thead>
<tr>
<th>Permission level</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin/Manage/Topic/Add</td>
<td>This permission enables the possibility of adding topics and subtopics. When you go to the <strong>Main</strong> tab, the <strong>Add</strong> option will be enabled <strong>Topics</strong>, which can be used to add a new topic. When you go to <strong>Topics -&gt; List</strong> and select a particular topic, the <strong>Add Subtopic</strong> link will also be enabled.</td>
</tr>
<tr>
<td>Admin/Manage/Topic/Details</td>
<td>This permission enables the possibility of checking the details of topics and subtopics. When you go to <strong>Topics -&gt; List</strong> and select a particular topic, the <strong>Delete</strong> link will be enabled.</td>
</tr>
</tbody>
</table>

Note that a user that has permission to **Add** new topics, by default obtains permission to **subscribe** and **publish** to all the topics that are created by the same user.

Note that the **Admin/Manage/Resources/Browse** permission node should also be enabled for topic deletion to be allowed.

<table>
<thead>
<tr>
<th>Permission level</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin/Manage/Subscription/Queue</td>
<td>This permission enables the possibility of viewing details of queue subscribers. The <strong>Subscription -&gt; Queue Subscription List</strong> option will be available in the <strong>Main</strong> tab.</td>
</tr>
<tr>
<td>Admin/Manage/Subscription/CloseQueueSubscriptions</td>
<td>This permission in addition to 'Admin/Manage/Subscription/Queue' will allow users to close queue subscriptions.</td>
</tr>
</tbody>
</table>
Admin/Manage/Subscription/Topic

This permission enables the possibility of viewing details of topic subscribers. The **Subscription -> Topic Subscription List** option will be available in the **Main** tab.

Admin/Manage/Subscription/CloseTopicSubscriptions

This permission in addition to 'Admin/Manage/Subscription/Topic' will allow users to close topic subscriptions.

### Subscribing to Topics/Queues

Explained above are the list of role-based permissions that are required by users in order to create and manage queues/topics from the Management Console.

Note that the permission to create topics/queues also includes the permissions for publishing messages to that topic/queue and consuming the messages published to that topic/queue.

Once queues and topics are created in the Management Console, other users should be able to publish to these topics/queues and consume the messages that are published. Therefore, the creator of the topic/queue should grant permissions to other user roles at the time of creating the topic/queue as shown below.

- **When adding a topic from the Management Console**, all the available user roles will be listed as shown below. The topic creator can then select the relevant check box to grant the relevant permissions. See the detailed instruction on **creating topics in WSO2 MB**.

![Add Topic](image)

- **When adding a queue from the Management Console**, all the available user roles will be listed as shown below. The queue creator can then select the relevant check box to grant the relevant permissions. See the detailed instruction on **creating queues in WSO2 MB**.
General management permissions

Listed below are the permissions for some of the general functions applicable to WSO2 MB.

<table>
<thead>
<tr>
<th>Permission level</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin/Manage/Add</td>
<td>This permission enables the <strong>Cassandra Keypaces</strong> menu under the <strong>Main</strong> navigator menu. This option allows users to add and manage keyspaces in a Cassandra cluster.</td>
</tr>
<tr>
<td>Admin/Manage/Resources/Browse</td>
<td>This permission enables the <strong>Browse</strong> option under the <strong>Registry</strong> menu in the main navigator. This option allows users to browse the resources stored in the registry by using the <strong>Registry</strong> tree navigator.</td>
</tr>
<tr>
<td>Admin/Manage/Search</td>
<td>This permission enables the <strong>Search</strong> option under the <strong>Registry</strong> sub menu in the <strong>Main</strong> menu. This option allows users to search for specific resources stored in the registry by filling in the search criteria.</td>
</tr>
</tbody>
</table>

Permissions for monitoring

<table>
<thead>
<tr>
<th>Permission level</th>
<th>Description of UI menus enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Admin/Monitor/Logs</td>
<td>When this node is selected, the following menus are enabled in the <strong>Monitor</strong> tab of the Management Console:</td>
</tr>
<tr>
<td></td>
<td>- <strong>Monitor</strong> menu/<strong>System Logs</strong>: See the topic on system logs for information on how to use this option.</td>
</tr>
<tr>
<td></td>
<td>- <strong>Monitor</strong> menu/<strong>Application Logs</strong>: See the topic on application logs for information on how to use this option.</td>
</tr>
</tbody>
</table>
Admin/Monitor/Metrics

When this node is selected, the following menus are enabled in **Monitor** tab of the Management Console:

- **Metrics/JVM Metrics**: Used for monitoring system statistics common to all products.
- **Metrics/Messaging Metrics**: Used for monitoring MB-specific statistics.

### Configuring Message Compression

All messages published to WSO2 MB are stored in a database, which guarantees message persistence. From WSO2 MB 3.1.0 onwards, you can reduce the overhead on the database by compressing (i.e., reducing the message content size) the messages using LZ4 compression. This will reduce the number of message content chunks, which in turn will reduce the number of records in the database. This is an optional configuration that you can set up in the message broker.

Find out more about how this configuration improves the **performance of your MB server**.

Follow the steps given below to enable message content compression for your server.

1. Open the `broker.xml` file from the `<MB_HOME>/repository/conf` directory.
2. Enable compression by setting the following property to 'true'.

   ```xml
   <!-- This is the configuration to allow compression of message contents, before store messages into the database.-->
   <allowCompression>false</allowCompression>
   ```

3. Specify the message content compression threshold using the following parameter. This is the minimum content size a message should have in order to be compressed. Messages smaller than this size will not be compressed, even if compression is enabled for the server.

   ```xml
   <!-- This is the configuration to change the value of the content compression threshold (in bytes). Message contents less than this value will not compress, even compression is enabled. The recommended message size of the smallest message before compression is 13 bytes. Compress messages smaller than 13 bytes will expand the message size by 0.4% -->
   <contentCompressionThreshold>1000</contentCompressionThreshold>
   ```

   Note the following when you set this parameter:

   - If you set a very low value, the message size will increase due to the lack of repeated content.
   - It is not recommended to compress messages that are smaller than 13 bytes.
   - This value effects the performance of your server: Lower values will reduce server performance. Higher values (greater than 1000) would increase performance.

4. Specify values for the following parameters, which will determine the speed of the message sending mechanism:
The `<parallelDecompressionHandlers>` parameter specifies the total number of parallel decompression handlers that are used to decompress the message content before they are sent to subscribers.

The `<parallelContentReaders>` parameter specifies the number of parallel readers used to read content from the message store.

The `<parallelDeliveryHandlers>` parameter specifies the number of parallel delivery handlers used to deliver messages to subscribers.

```xml
<parallelDecompressionHandlers>5</parallelDecompressionHandlers>
<parallelContentReaders>5</parallelContentReaders>
<parallelDeliveryHandlers>5</parallelDeliveryHandlers>
```

Note the following when you set this parameter:

- Reducing this value would decrease the speed of the message sending mechanism, but also reduce the load on the message store.
- Increasing this value would increase the speed of the message sending mechanism because multiple handlers are working in parallel. However, the load on the message store would also increase.
- A higher number of cores is required to increase this value.
- The speed of the message sending mechanism depends on all three parameters given above. Therefore, it is recommendation to set the same (or nearly same) value for all of them.

**Enabling SSL Support**

WSO2 Message Broker provides support to send/receive messages via secured connections using the SSL/TLS protocol. The following instructions describe how to configure the MB server and JMS clients to communicate via encrypted connections using SSL.

- **Enabling SSL in the broker**
- **Configuring JMS Clients to use SSL**
- **Configuring JMS Clients for Failover with SSL**

**Enabling SSL in the broker**

To enable SSL in the server side, change the following entries in the `<MB_HOME>/repository/conf/broker.xml` file under the relevant transport (AMQP or MQTT). See Configuring Transports for WSO2 MB for more information on the available transports.
The parameters in the above configuration are as follows.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SSL Connection</strong></td>
<td>This contains the basic configurations relating to the SSL connection. Setting the <code>enabled=&quot;true&quot;</code> attribute ensures that SSL is enabled by default when the MB server is started. The <code>port=&quot;&quot;</code> attribute sets the default SSL listener port for messages/command sent via the relevant transport.</td>
</tr>
<tr>
<td></td>
<td>• The default port for the AMQP transport is 8672.</td>
</tr>
<tr>
<td></td>
<td>• The default port for the MQTT transport is 8883.</td>
</tr>
<tr>
<td><strong>Location</strong></td>
<td>The location where the keystore/truststore used for securing SSL connections is stored. By default this is the default keystore (<code>wso2carbon.jks</code>) and truststore (<code>client-truststore.jks</code>) that is shipped with WSO2 MB.</td>
</tr>
<tr>
<td></td>
<td>Note that these (keystore and truststore) should always created for the super tenant. Find out more about setting up keystores for your MB server.</td>
</tr>
<tr>
<td><strong>Password</strong></td>
<td>The password of the keystore/truststore.</td>
</tr>
<tr>
<td><strong>Certification Type</strong></td>
<td>The type of SSL certificate used for the keystore/truststore. <strong>SunX509</strong> is the standard name of the algorithm used by the key managers. This value should be changed accordingly if the system is running on a different JVM. For example, <strong>IbmX509</strong> for the IBM JVM.</td>
</tr>
</tbody>
</table>

**Configuring JMS Clients to use SSL**

SSL parameters are configured and sent to the broker as broker options in the `TCPConnectionURL` defined by the client. You need to set the 'ssl=true' property in the url and specify the keystore and client trust store paths and passwords. Use the connectionurl format shown below to pass the SSL parameters:
String connectionURL =
"amqp://<USERNAME>:<PASSWORD>@carbon/carbon?brokerlist='tcp://<IP>:<SSL_PORT>?ssl='true'&ssl_cert_alias='<CERTIFICATE_ALIAS_IN_TRUSTSTORE>'&trust_store='<PATH_TO_TRUST_STORE>'&trust_store_password='<TRUSTSTORE_PASSWORD>'&key_store='<PATH_TO_KEY_STORE>'&key_store_password='<KEYSTORE_PASSWORD>'";

Setting the 'ssl_cert_alias' property is not mandatory and can be used as an optional way to specify which certificate the broker should use if the trust store contains multiple entries.

Example: Consider that you have WSO2 Enterprise Service Bus (WSO2 ESB) as the JMS client. Shown below is an example connectionurl using the default keystores and trust stores in WSO2 ESB:

String connectionUrl =

When you configure WSO2 ESB to communicate with the broker (WSO2 MB) using SSL, the SSL url should be configured in the jndi.properties file for WSO2 ESB (stored in the <ESB_HOME>/repository/conf directory). Go to this link for detailed instructions on how WSO2 ESB integrates with WSO2 MB.

Configuring JMS Clients for Failover with SSL

For example, if you have configured a WSO2 Message Broker cluster, you might need to configure failover. If those broker nodes have different certs in place, when configuring a failover connection url at the client side, you can individually specify a client trust store and a keystore for each broker in the broker list. Or else, you can import the certs of all brokers in the cluster to a single trust store with different cert aliases and differentiate the cert to use when failing over by the alias.

Port Offset Configuration

Changing the Default Port of WSO2 Message Broker

Like other WSO2 products, Message Broker also uses the 9443 port in the default configuration. However, there can be instances where this default port needs to be changed. Change the <Offset> entry in the <MB_HOME>/repository/conf/carbon.xml file.

The following block handles the port configurations in the carbon.xml file:
<Ports>
  <!-- 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 -->
  <Offset>0</Offset>
  <!-- The JMX Ports -->
  <JMX>
    <!-- The port RMI registry is exposed-->
    <RMIRegistryPort>9999</RMIRegistryPort>
    <!-- The port RMI server should be exposed-->
    <RMIServerPort>11111</RMIServerPort>
  </JMX>
  <!-- Embedded LDAP server specific ports -->
  <EmbeddedLDAP>
    <!-- Port which embedded LDAP server runs -->
    <LDAPServerPort>10389</LDAPServerPort>
    <!-- Port which KDC (Kerberos Key Distribution Center) server runs -->
    <KDCServerPort>8000</KDCServerPort>
  </EmbeddedLDAP>
  <!-- Override datasources JNDIproviderPort defined in bps.xml and datasources.properties files -->
  <!--<JNDIProviderPort>2199</JNDIProviderPort>-->
  <!-- Override receive port of thrift based entitlement service.-->
  <ThriftEntitlementReceivePort>10500</ThriftEntitlementReceivePort>
</Ports>

**Configuring a Client to Access Broker When Port Offset is Changed**

As the Message Broker's port offset is changed, the default TCP Port of the broker also gets increased by the value set as the port offset. Hence, when using an external client to connect with the broker, TCP connection url's port should be changed to reflect the port offset value change.

Therefore if `<Offset>` is set as '1', the TCP Connection URL which is in the form of "amqp://{username}:{password}@carbon/carbon?brokerlist=tcp://{hostname}:{port}" must be changed to "amqp://{username}:{password}@carbon/carbon?brokerlist=tcp://{hostname}:{port+1}" from this onwards.

As the default Broker TCP port value for the AMQP transport is 5672, in this case it should be changed to 5673. Similarly, when the MQTT transport is used, the default Broker TCP port value 1883 should be changed to 1884.

**Handling Failover**

High availability is an important aspect associated with any server implementation that ensures a certain degree of operational continuity when unplanned downtime events impact parts of a system. JMS 1.1 does not provide this;
instead, most of the time it is vendor specific.

The idea behind failover is stopping a single-point-of-failure in a system. As the broker is the middle man storing and forwarding the messages, if that server goes down, the entire message flow of the system will go down no matter what other servers and functions are involved. In order to make a robust messaging system, it is mandatory to have a failover mechanism.

In order to achieve this, a few instances of message broker servers are set up and running in the system, while the system (generally) uses one broker. If that broker goes down, it automatically switches to the second broker and continues messaging. If the second broker fails, it will try the next one and so on. Thus, the whole system will not have any downtime.

This page covers the following topics:

- Failover with WSO2 Message Broker and WSO2 ESB
- Testing the setup

**Failover with WSO2 Message Broker and WSO2 ESB**

Follow the instructions below to set up a failover mechanism using MB and ESB:

- Setting up MB instances in a single machine
- Setting up MB instances in different machines
- Setting up a WSO2 ESB instance

### Setting up MB instances in a single machine

1. Download the latest version of MB and extract it into a folder.
2. Make an exact copy of MB in three different locations and rename them as MB1, MB2 and MB3. Those will be the 3 separate MB instances.

   ```xml
   <Ports>
   <!-- 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
   -->
   <Offset>1</Offset>
   </Ports>
   
   6. The 3 MB instances are ready. Start each instance of MB by running one of the following commands:
      - `<MB_HOME>/bin/wso2server.sh` (on Linux)
      - `<MB_HOME>/bin/wso2server.bat` (on Windows)
   7. The MB instances will start on ports 5673, 5674 and 5675 respectively.

### Setting up MB instances in different machines

1. Download the latest version of MB and extract it into a folder.
2. Make an exact copy of that folder in three separate machines. Let's assume MB1 is in the machine with IP1, MB2 in the machine with IP2 and MB3 in the machine with IP3.
3. Start all the above instances by running one of the following commands:
• `<MB_HOME>/bin/wso2server.sh` (on Linux)
  • `<MB_HOME>/bin/wso2server.bat` (on Windows)

All servers will start with port offset 0, which is 5672.

Setting up a WSO2 ESB instance

1. Start ESB in the default port (port offset 0). This is possible if you used the single machine setup above, as those servers were started with different port offsets. If you are using different machines, use another machine to start WSO2 ESB. Instructions can be found in Configure with WSO2 Message Broker.

   It is not possible to start multiple WSO2 products with their default configurations simultaneously in the same environment. Since all WSO2 products use the same port in their default configuration, there will be port conflicts.

2. There is only a single difference to enable failover across the three brokers we have setup. That is when specifying the following:

```
connectionfactory.ConnectionFactory
connectionfactory.QueueConnectionFactory
connectionfactory.TopicConnectionFactory
```

For example, if you hope to use a single machine MB setup as described above,

```
connectionfactory.QueueConnectionFactory =
```

If you hope to use the broker setup made across several machines as described above,

```
connectionfactory.QueueConnectionFactory =
```

The parameters used above are described in detail below:

**Brokerlist option**

```
brokerlist='<broker url>[;<broker url>]'```

The broker list defines the various brokers that can be used for this connection. A minimum of one broker
URL is required. Additional URLs are semi-colon (';') delimited.

**Broker URL format**

```
<transport>://<host>[:<port>][?<option>='<value>'[&<option>='<value>']]`
```

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>retries</td>
<td>1</td>
<td>The number of times to retry connection to this Broker</td>
</tr>
<tr>
<td>ssl</td>
<td>false</td>
<td>Use ssl on the connection</td>
</tr>
<tr>
<td>connecttimeout</td>
<td>30000</td>
<td>How long in (milliseconds) to wait for the connection to succeed</td>
</tr>
<tr>
<td>connectdelay</td>
<td>none</td>
<td>How long in (milliseconds) to wait before attempting to reconnect</td>
</tr>
</tbody>
</table>

**Brokerlist failover option**

```
failover='<method>[?<options>]'
```

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>singlebroker</td>
<td>This will only use the first broker in the list.</td>
</tr>
<tr>
<td>roundrobin</td>
<td>This method tries each broker in turn.</td>
</tr>
<tr>
<td>nofailover</td>
<td>[New in 0.5] This method disables all retry and failover logic.</td>
</tr>
</tbody>
</table>

The current default is to use the singlebroker method when only one broker is present, and the roundrobin method with multiple brokers. The method value in the URL may also be any valid class on the classpath that implements the FailoverMethod interface.

**Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclecount</td>
<td>1</td>
<td>The number of times to loop through the list of available brokers before failure.</td>
</tr>
</tbody>
</table>

**Testing the setup**

When sending messages to a queue (you can configure SOAP UI to send messages continuously one after the other) using the setup described above, kill MB1. You will notice that failover happens at ESB to the second broker i.e MB2. The logs at the ESB side will indicate that failover occurred.

If there were any previous messages at MB1, they will not be delivered.

Usually, the solution for that (and many other problems) is broker clustering. For information on the advantages of clustering and WSO2 Message Broker's support on clustering, see [Clustering Message Broker](#).

**Configuring the Message Delivery Strategy**

When you work with topics, messages that are not acknowledged by subscriber clients can cause your MB server to go out of memory. The message delivery strategy configuration is introduced in WSO2 MB to control this issue. This configuration allows you to control how messages that accumulate in memory should be handled by the server. See
the following topics for more details:

- Understanding message delivery patterns in WSO2 MB
- Changing the default message delivery strategy

**Understanding message delivery patterns in WSO2 MB**

All messages published to MB are stored in a database, which makes them inherently persistent. The MB will then read the messages from the database and deliver them to the relevant subscribers. The way messages flow through MB to subscribers works differently for Queues and Topics.

**Queues** are typically used for storing messages. A JMS client can then subscribe to the queue at any time and consume the messages stored for the queue. Unlike with Queues, **Topics** in WSO2 MB are typically used for realtime message brokering. That is, messages published to a topic are required to be delivered to all the active subscribers instantly (in realtime). Messages that are published to a topic while a subscriber is inactive will be handled according to the durability of the subscription as explained below.

### Durability of topic subscriptions:

- If the topic subscriber creates a non-durable subscription to the topic, the messages published to the topic while the subscriber is inactive will be lost.
- If the topic subscriber creates a durable subscription, the subscriber will be able to recover all messages that were published to the topic while the subscriber was inactive.

Read more about the [durability of topic subscriptions](#).

**Message delivery for non-durable topic subscriptions**

When there are non-durable topic subscriptions for a topic, the messages are temporarily stored in memory of the MB server until all the subscribers acknowledge that the message is received. However, sometimes a subscriber can be late to acknowledge, which will cause messages to accumulate in memory. The following steps describe how this message flow works:

1. Messages published to MB are first stored in the database.
2. MB fetches the messages from the DB and dispatches them to all the subscriber clients instantly.
3. MB maintains a list of the dispatched messages in memory as meta data until the messages are successfully received by all the clients.
4. The messages that are successfully received will send an acknowledgement back to the MB.
5. Messages that are acknowledged by all the subscribers are removed from the meta data in memory.
6. The messages that are not acknowledged by the subscriber will be retained in memory until such acknowledgement is received.

It is important to note here that the messages are not dispatched to subscriber clients from the MB according to the rate at which the messages are consumed by the subscriber. For example, consider that MB dispatches 1000 messages per second to each subscriber: Subscriber A may be consuming messages at the rate of 500 messages per second, whereas subscriber B will consume all 1000 messages per second. In this scenario, 500 messages that are not acknowledged by subscriber A will always be accumulating in MB memory. If the number of unacknowledged messages increases beyond a certain point, the MB server could run out of memory. Therefore, when you work with topics, you can change the message delivery strategy according to the requirement.

**Changing the default message delivery strategy**

Follow the steps given below.

1. Open the `broker.xml` file from the `<MB_HOME>/repository/conf` directory.
2. Locate the parameters defining the topic message delivery strategy:
3. Update the `<strategyName>` parameter using one of the following values:

- **DISCARD_NONE**: This is the default setting, which ensures that none of the messages are discarded from memory in MB. All the unacknowledged messages will be retained in memory of the MB.
- **DISCARD_ALLOWED**: If you set this value, messages dispatched from the MB will be removed from memory without waiting for an acknowledgement from the subscriber clients.
- **SLOWEST_SUB_RATE**: If you set this value, messages will be dispatched to subscriber clients at the rate of the slowest subscriber. For example, if subscriber A consumes messages at the rate of 500 messages per second and subscriber B consumes messages at a rate of 1000 messages per second, MB will always dispatch messages at the rate of 500 messages per second to both the subscribers.

4. Update the `<deliveryTimeout>` parameter with time period in seconds that is allowed before MB removes any unacknowledged messages.

### Clustered Deployment

In a production environment, WSO2 Message Broker (WSO2 MB) can be clustered and configured with an external Database Management System (DBMS) of your choice.

- **Advantages of clustering WSO2 MB**
- **Recommended deployment pattern**
- **Setting up the DBMS**
- **Configuring the WSO2 MB nodes in the cluster**
  - Configuring the broker.xml
  - Configuring the axis2.xml
  - Configuring user-mgt.xml
  - Configuring master-datasources.xml
  - Configuring registry.xml
  - Configuring cluster coordination
  - Handling network partitioning in Hazelcast-based clustering
- **Starting the servers**

#### Advantages of clustering WSO2 MB

Clustering will give you a very scalable product. This is useful because adding more WSO2 MB nodes to your cluster enables you to publish your messages or do subscriptions in a load balanced way. With clustering, your product can be scaled up to meet high messaging demands and if there are performance issues, the product can be scaled down according to the requirement. Therefore, WSO2 Message Broker can deliver high-performance results as opposed to many commercial and conventional Message Brokers that have very low performance when the "size of a message" becomes too large.

#### Recommended deployment pattern

The following is the common deployment model referred to in the configurations. This is the recommended
deployment pattern for WSO2 Message Broker and depicts the minimum number of broker nodes necessary to achieve high availability and high efficiency in the cluster.

Figure 1: Message Broker Nodes Configured With An External DBMS Cluster.

In this deployment model, assume that all these nodes are on different hosts. There are two WSO2 Message Broker nodes and any number of nodes that you prefer for the external DBMS (this can even be a single node).

<table>
<thead>
<tr>
<th>Server</th>
<th>IP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB Server 01</td>
<td>192.168.0.102</td>
</tr>
<tr>
<td>MB Server 02</td>
<td>192.168.0.103</td>
</tr>
<tr>
<td>RDBMS</td>
<td>192.168.0.104</td>
</tr>
</tbody>
</table>

Although this example specifies two WSO2 Message Broker nodes as a minimum requirement for optimal performance, you can have more Message Broker nodes depending on your requirement.

- It is vital to time synchronize all the nodes across the cluster to make the Message Broker nodes function properly.
- Ensure that required ports are open in the firewall or not bound to any other service. The list of ports used by WSO2 Message Broker can be found in Default Ports of WSO2 Products.

Use the following instructions to configure the two Message Broker nodes and connect them to an external DBMS.

Setting up the DBMS

The following steps describe how to download and install MySQL Server, create the databases, configure the datasources, and configure WSO2 MB to connect to them.

Note: MySQL is used as an example here, but you can use any database for this and change the configurations accordingly. See Configuring the DBMS for Storage for more options.
1. Download and install MySQL Server.
2. Download the MySQL JDBC driver.

3. Unzip the downloaded MySQL driver zipped archive, and copy the MySQL JDBC driver JAR (mysql-connector-java-x.x.xx-bin.jar) into the <PRODUCT_HOME>/repository/components/lib directory.

4. Define the host name for configuring permissions for the new database by opening the /etc/hosts file and adding the following line:

   `<MYSQL-DB-SERVER-IP> carbondb.mysql-wso2.com`

   You would do this step only if your database is on a separate server (not on your local machine).

5. Enter the following command in a terminal/command window, where `username` is the username you want to use to access the databases:

   `mysql -u username -p`

6. When prompted, specify the password that will be used to access the databases with the username you specified.

7. Create the databases using the following commands, where `<PRODUCT_HOME>` is the path to any of the product instances you installed, and `username` and `password` are the same as those you specified in the previous steps:

   `mysql> create database <DATABASE_NAME> character set latin1;`

   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 following is how your database creation command should look.

   `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 following is how your database creation command should look.

   `mysql> create database <DATABASE_NAME>;`
mysql> create database wso2_mb;
mysql> use wso2_mb;
mysql> source <MB_HOME>/dbscripts/mb-store/mysql-mb.sql;
mysql> grant all on wso2_mb.* TO username@localhost identified by "password";

mysql> create database userdb;
mysql> use userdb;
mysql> source <MB_HOME>/dbscripts/mysql.sql;
mysql> grant all on userdb.* TO username@localhost identified by "password";

mysql> create database regdb;
mysql> use regdb;
mysql> source <MB_HOME>/dbscripts/mysql.sql;
mysql> grant all on regdb.* TO username@localhost identified by "password";

Note: Ensure that MySQL is configured so that all nodes can connect to it.

Configuring the WSO2 MB nodes in the cluster

Once you have set up the DBMS for your cluster as explained previously, you can update the configuration files in each node of the WSO2 MB cluster as explained below.

- Configuring the broker.xml
- Configuring the axis2.xml
- Configuring user-mgt.xml
- Configuring master-datasources.xml
- Configuring registry.xml
- Configuring cluster coordination
- Handling network partitioning in Hazelcast-based clustering

Configuring the broker.xml

1. Open the <MB_HOME>/repository/conf/broker.xml file. This is the root configuration file of WSO2 MB. The changes made to this file must be done in all the WSO2 MB nodes.

2. Do thrift-related configurations. Here you must configure the thriftServerHost value to point to the IP address of the MB server node.
   - MB Server 1
   - MB Server 2
<coordination>
    <nodeID>default</nodeID>
    <thriftServerHost>192.168.0.102</thriftServerHost>
    <thriftServerPort>7611</thriftServerPort>
    <thriftSOTimeout>0</thriftSOTimeout>
</coordination>

<coordination>
    <nodeID>default</nodeID>
    <thriftServerHost>192.168.0.103</thriftServerHost>
    <thriftServerPort>7611</thriftServerPort>
    <thriftSOTimeout>0</thriftSOTimeout>
</coordination>

See the following table for details on these configurations.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coordination</td>
<td>This configuration is related to MB thrift communications.</td>
</tr>
<tr>
<td>nodeID</td>
<td>In a clustered deployment, an ID is assigned to each MB node via the cluster node identifier. This element can be used to override the cluster node identifier for this MB node. If the value for this element is left as default, the default node ID is generated using the IP and a universally unique identifier (UUID). <strong>The node ID of each member in a cluster must be unique.</strong></td>
</tr>
<tr>
<td>thriftServerHost</td>
<td>This is a sub-element of the &lt;coordination&gt; tag. WSO2 MB uses Apache Thrift for communications relating to message delivery. Therefore, an Apache Thrift server is started in each MB node in a clustered deployment. This element should point to the IP address of the Apache Thrift server. This should point to the IP address of the MB node that hosts the thrift server. The default value for this is localhost. For example, if you are configuring a Message Broker node hosted in 192.168.0.102 as the thrift server, this value should be 192.168.0.102.</td>
</tr>
<tr>
<td>thriftServerPort</td>
<td>This is another sub-element of the &lt;coordination&gt; tag. This should point to the port of the thrift server in MB. The default port is 7611. <strong>It is recommended to use this port for all broker nodes in your cluster.</strong></td>
</tr>
<tr>
<td>thriftSOTimeout</td>
<td>This is used to handle half-open TCP connections between the broker nodes in a cluster. In such situations, the socket may need to have a timeout value to invalidate the connection (in milliseconds). A timeout of zero is interpreted as an infinite timeout.</td>
</tr>
</tbody>
</table>

**Configuring the axis2.xml**

This section provides all the configurations related to enabling and correctly configuring clustering.

1. Open the `<MB_HOME>/repository/conf/axis2/axis2.xml` file. The changes made to this file must be done in both broker nodes.
2. Enable **Hazelcast** clustering by doing the following configuration.

```xml
<clustering class="org.wso2.carbon.core.clustering.hazelcast.HazelcastClusteringAgent" enable="true"/>
```

3. Specify the membership scheme that you plan to use for clustering. See **Clustering Overview** for more information on membership schemes.

```xml
<parameter name="membershipScheme">wka</parameter>
```

4. Configure the `localMemberHost` and the `localMemberPort` of the server to point to the IP address of the host where the MB server resides. This has to be done for each server.

- **MB Server 1 (192.168.0.102)**
  ```xml
  <parameter name="localMemberHost">192.168.0.102</parameter>
  <parameter name="localMemberPort">4000</parameter>
  ```

- **MB Server 2 (192.168.0.103)**
  ```xml
  <parameter name="localMemberHost">192.168.0.103</parameter>
  <parameter name="localMemberPort">4000</parameter>
  ```

5. When using the “wka” membership scheme, each member of the cluster should be configured with the information about other cluster members. In this case, the other broker node must be defined here. For example, if the broker node you are configuring is on 192.168.0.102, you must configure the other broker node, which is hosted on 192.168.0.103, as indicated below.

- **MB Server 1 (192.168.0.102)**
  ```xml
  <members>
  <member>
    <hostName>192.168.0.103</hostName>
    <port>4000</port>
  </member>
  </members>
  ```

- **MB Server 2 (192.168.0.103)**
  ```xml
  <members>
  <member>
    <hostName>192.168.0.102</hostName>
    <port>4000</port>
  </member>
  </members>
  ```

**Configuring user-mgt.xml**

1. Open the `<MB_HOME>/repository/conf/user-mgt.xml` file on each MB server.

2. Update the `dataSource` property in `user-mgt.xml` to the `jndiConfig` name used in `master-datasources.xml` for user store database configuration.
Configuring master-datasources.xml

In this configuration file, you must configure the datasources to point to the databases that you configured earlier.

>Note: MySQL is used as an example here, but you can use any database for this and change the configurations accordingly. See Configuring the DBMS for Storage for more options.

Also, make sure to replace the username and password with the username and password used by your database.

1. Open the `<MB_HOME>/repository/conf/datasources/master-datasources.xml` file. The changes made to this file must be done in both broker nodes.
2. Remove the H2-based WSO2_MB_STORE_DB configuration. This is the default configuration. To do this, remove or comment out the following code snippet.

```xml
<datasource>
  <name>WSO2_MB_STORE_DB</name>
  <description>The datasource used for message broker database</description>
  <jndiConfig>
    <name>WSO2MBStoreDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:h2:repository/database/WSO2MB_DB;DB_CLOSE_ON_EXIT=FALSE;LOCATION=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>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>
```

3. Uncomment or add the following MySQL-based WSO2_MB_STORE_DB configuration.
3. Add the user store database configuration.

```xml
<datasource>
  <name>WSO2_USER_STORE_DB</name>
  <jndiConfig>
    <name>jdbc/WSO2UserStoreDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <url>jdbc:mysql://192.168.0.104/userdb</url>
      <username>root</username>
      <password>root</password>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>
```

4. Add the user store database configuration.

```xml
<datasource>
  <name>WSO2_USER_STORE_DB</name>
  <jndiConfig>
    <name>jdbc/WSO2UserStoreDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <driverClassName>com.mysql.jdbc.Driver</driverClassName>
      <url>jdbc:mysql://192.168.0.104/userdb</url>
      <username>root</username>
      <password>root</password>
      <maxActive>50</maxActive>
      <maxWait>60000</maxWait>
      <minIdle>5</minIdle>
      <testOnBorrow>true</testOnBorrow>
      <validationQuery>SELECT 1</validationQuery>
      <validationInterval>30000</validationInterval>
      <defaultAutoCommit>false</defaultAutoCommit>
    </configuration>
  </definition>
</datasource>
```

5. Add the registry database configuration.
<datasource>
  <name>WSO2_GOV_DB</name>
  <description>The datasource used for registry and user manager</description>
  <jndiConfig>
    <name>jdbc/WSO2GovDB</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <url>jdbc:mysql://192.168.0.104/regdb</url>
      <username>root</username>
      <password>root</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>

Configuring registry.xml

In this configuration file, you must correctly mount the registry database.

1. Open the `<MB_HOME>/repository/conf/registry.xml` file. The changes made to this file must be done in all the WSO2 MB nodes.
2. Add the following configurations to mount the registry and do not replace any existing configurations.

```
<dbConfig name="wso2govregistry">
  <dataSource>jdbc/WSO2GovDB</dataSource>
</dbConfig>

<remoteInstance url="https://localhost:9443/registry">
  <id>govdb</id>
  <dbConfig>wso2govregistry</dbConfig>
  <readOnly>false</readOnly>
  <enableCache>true</enableCache>
  <registryRoot>/_system/governance</registryRoot>
</remoteInstance>

<mount path="/._system/governance" overwrite="true">
  <instanceId>govdb</instanceId>
  <targetPath>/_system/governance</targetPath>
</mount>
```

When mounting configurations, make note of the following.
• The `dataSource` you specify under the `<dbConfig name="sharedregistry">` tag must match the `jndiConfig` name you specified in the `master-datasources.xml` file.
• The registry `mountpath` is used to identify the type of registry. For example, `"/_system/config"` refers to configuration registry, and `"/_system/governance"` refers to the governance registry.
• The `dbconfig` entry enables you to identify the datasource you configured in the `master-datasources.xml` file. We use the unique name `sharedregistry` to refer to that datasource entry.
• The `remoteInstance` section refers to an external registry mount. We can specify the read-only/read-write nature of this instance as well as caching configurations and the registry root location. In the case of a Message Broker node, the `readOnly` property must be set to `false` as we require all broker nodes to have both read and write permissions. The `remoteInstance url` is used internally in Governance Registry for notification functionality. For all other non-Governance Registry server instances, you can leave this property as it is. If your MB cluster is not configured with a WSO2 Governance Registry, you do not need to change this URL entry for new values.
• You must define a unique name “id” for each remote instance, which is then referred to from mount configurations. In the above example, the unique ID for the remote instance is `govdb`.
• In each of the mounting configurations, we specify the actual mount path and target mount path. The `targetPath` can be any meaningful name. In this instance, it is `/_system/mbNodes`.

**Configuring cluster coordination**

WSO2 MB 3.2.0 introduces cluster coordination through an RDBMS. This means that the coordination between the nodes in a cluster can be managed through an RDBMS, just as message persistence. Therefore, the RDBMS that is connected to the MB nodes in the cluster is, by default, used for message persistence as well as cluster coordination. Shown below is the configuration in the `broker.xml` file (stored in the `<MB_HOME>/repository/conf/` directory), which enables RDBMS-based cluster coordination. If required, you can disable the following configuration, which will allow the hazelcast engine to manage cluster coordination. However, note that you need to configure the cluster to handle network partitioning when hazelcast-based cluster coordination is used.
Handling network partitioning in Hazelcast-based clustering

Network partitioning in the network used for cluster coordination can sometimes disrupt the cluster. WSO2 MB 3.2.0 introduces a new configuration to handle network partitioning that may occur when cluster coordination is managed by the hazelcast engine (as opposed to the RDBMS as explained above).

For example, consider a cluster of three WSO2 MB nodes that uses two separate networks for cluster coordination and message persistence. The network with the RDBMS will persist messages, subscriptions and queues etc., and the cluster network communicates information about subscriptions, queue additions, and to decide on the coordinator of the cluster. If one of the nodes in the cluster disconnects from the network used for cluster coordination, that node will separate from the cluster and function as a separate node/cluster (separate partition) as illustrated in the diagram below. That is, the three-node cluster will now be working as two separate clusters. However, the disconnected node will still be up and running, and message persistence will continue uninterrupted through the message persistence network. This will cause inconsistencies in the system because message persistence and cluster coordination will not be synchronized. In such a situation, it is necessary to stop the disconnected node from accepting messages.
The above situation can be prevented in MB 3.2.0 by configuring the minimum node count of the cluster. For example, if the cluster size is 5 and we have configured the minimum node count to 3 nodes, during a network partition (or when nodes crash or shut down) any partition that has 3 or more nodes will keep functioning, while the other partitions will stop processing messages.

Follow the steps given below to enable this feature.

1. Open the `<MB_HOME>/repository/conf/broker.xml` file and set the `<networkPartitionsDetection>` element shown below to `enabled="true"`. Note that this configuration is disabled by default as shown below.

   ```xml
   <?xml version="1.0"?>
   <networkPartitionsDetection enabled="false">
     <minimumClusterSize>1</minimumClusterSize>
   </networkPartitionsDetection>
   ```

2. Update the `<minimumClusterSize>` property in the above configuration to specify the minimum node count the cluster should maintain in order to operate. Note that if this value should be at least two for cluster coordination to work. If the number of nodes in the cluster becomes less than that configured value, the cluster will not accept any incoming traffic. That is, all subscriptions will be disconnected.

Starting the servers
Start the MB servers by executing the `<MB_HOME>/bin/wso2server.sh` file. 
$ ./wso2server.sh

### Configuring the DBMS for Storage

The following topics contain instructions on how to configure various databases with the WSO2 Message Broker cluster for storage purposes.

- Configuring MSSQL as the RDBMS
- Configuring MySQL as the RDBMS
- Configuring Oracle as the RDBMS

#### Configuring MSSQL as the RDBMS

To configure and run MSSQL as your RDBMS, you must do the following.

1. Open the `<MB_HOME>/repository/conf/datasources/master-datasources.xml` file. This is where datasources are configured to point to the databases used by the Message Broker. This file consists of commented out configurations for datasources. The datasource configuration for MSSQL is among these. The changes made to this file must be done in both broker nodes.
2. Uncomment or add the following configuration into the `master-datasources.xml` file. Update the JDBC URL to correctly point to your database and enter the username and password for a MSSQL database user with the proper permissions.

```xml
<datasource>
    <name>WSO2_MB_STORE_DB</name>
    <jndiConfig>
        <name>jdbc/MSSQLWSO2MBStoreDB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <defaultAutoCommit>false</defaultAutoCommit>
            <dataSourceClassName>com.microsoft.sqlserver.jdbc.SQLServerXADataSource</dataSourceClassName>
            <dataSourceProps>
                <property name = "URL">jdbc:sqlserver://127.0.0.1\SQLExpress</property>
                <property name = "databaseName">wso2mb</property>
                <property name = "user">sa</property>
                <property name = "password">mssql</property>
            </dataSourceProps>
        </configuration>
    </definition>
</datasource>
```

3. Open the `<MB_HOME>/repository/conf/broker.xml` file. This is the root configuration file of Message Broker. The changes made to this file must be done in all the WSO2 Message Broker nodes.
4. In the `broker.xml` file we need to use the MSSQL message store and Andes context store. To do this, uncomment or add the following configuration.
Configuring MySQL as the RDBMS

To configure and run MySQL as your RDBMS, you must do the following.

1. Open the `<MB_HOME>/repository/conf/datasources/master-datasources.xml` file. This is where datasources are configured to point to the databases used by the Message Broker. This file consists of commented out configurations for datasources. The datasource configuration for MySQL is among these. The changes made to this file must be done in both broker nodes.

2. Uncomment or add the following configuration into the `master-datasources.xml` file. Update the JDBC URL to correctly point to your database and enter the username and password for a MySQL database user with the proper permissions.

```xml
<datasource>
  <name>MYSQL_DATA_SOURCE</name>
  <jndiConfig>
    <name>jdbc/MySQLMessageStore</name>
  </jndiConfig>
  <definition type="RDBMS">
    <configuration>
      <defaultAutoCommit>false</defaultAutoCommit>

      <url>jdbc:mysql://localhost/WSO2_MB</url>
      <username>root</username>
      <password>root</password>
    </configuration>
  </definition>
</datasource>
```

3. Open the `<MB_HOME>/repository/conf/broker.xml` file. This is the root configuration file of Message Broker. The changes made to this file must be done in all the WSO2 Message Broker nodes.

4. In the `broker.xml` file we need to use the MySQL message store and Andes context store. To do this, uncomment or add the following configuration.
Configuring Oracle as the RDBMS

To configure and run Oracle as your RDBMS, you must do the following.

1. Open the `<MB_HOME>/repository/conf/datasources/master-datasources.xml` file. This is where datasources are configured to point to the databases used by the Message Broker. This file consists of commented out configurations for datasources. The datasource configuration for Oracle is among these. The changes made to this file must be done in both broker nodes.

2. Uncomment or add the following configuration into the `master-datasources.xml` file. Update the JDBC URL to correctly point to your database and enter the username and password for an Oracle database user with the proper permissions.

```
<datasource>
    <name>ORACLE_DATA_SOURCE</name>
    <jndiConfig>
        <name>WSO2MBStoreDB</name>
    </jndiConfig>
    <definition type="RDBMS">
        <configuration>
            <defaultAutoCommit>false</defaultAutoCommit>
            <driverClassName>oracle.jdbc.driver.OracleDriver</driverClassName>
            <url>jdbc:oracle:thin:@localhost:1521/orcl</url>
            <maxActive>100</maxActive>
            <maxWait>60000</maxWait>
            <minIdle>5</minIdle>
            <testOnBorrow>true</testOnBorrow>
            <validationQuery>SELECT 1</validationQuery>
            <validationInterval>30000</validationInterval>
            <username>scott</username>
            <password>tiger</password>
        </configuration>
    </definition>
</datasource>
```
3. Open the `<MB_HOME>/repository/conf/broker.xml` file. This is the root configuration file of Message Broker. The changes made to this file must be done in all the WSO2 Message Broker nodes.
4. In the `broker.xml` file we need to use the Oracle message store and Andes context store. To do this, uncomment or add the following configuration.

```xml
...  
<messageStore  
class="org.wso2.andes.store.rdbms.RDBMSMessageStoreImpl">  
    <property name="dataSource">WSO2MBStoreDB</property>  
</messageStore>  
<contextStore  
class="org.wso2.andes.store.rdbms.RDBMSAndesContextStoreImpl">  
    <property name="dataSource">WSO2MBStoreDB</property>  
</contextStore>
```

### Analytics

This chapter contains the following information:

- Working with WSO2 Carbon Metrics
- Monitoring with Carbon Metrics

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 which is not bundled with the distribution by default, you need to install the relevant feature using the Configure-> Features menu in your product's Management Console.

#### Note

The screenshots may vary depending on the product and configuration options you are using.

### Working with WSO2 Carbon Metrics

WSO2 Carbon Metrics provides an API for WSO2 Carbon components to use the Metrics library. The following topics explain the configurations that we need to have in order to use metrics.

- Enabling Metrics and Storage Types
- Configuring Metrics Properties

#### Enabling Metrics and Storage Types

Given below are the configurations that should be in place for your MB server in order to use the metrics feature. You need to first enable metrics for your server and then enable the required storage types (reporters), which will be used for storing the metrics data. See the following topics for instructions:

- Enabling metrics
- Configuring the storage of metrics
- Sample configuration
Enabling metrics

To enable metrics for your product, set the Enabled parameter under the Metrics element to true in the `<MB_HOME>/repository/conf/metrics.xml` file. Alternatively, you can enable metrics at the time of starting the MB server by using the following command:

```
-Dmetrics.enabled=true
```

Once metrics are enabled, the Metrics dashboard will be updated for WSO2 MB.

Configuring the storage of metrics

WSO2 MB is configured by default to store the information from metrics in the following reporters: JMX, CSV and JDBC. These reporters are configured in the metrics.xml file (stored in the `<MB_HOME>/repository/conf` directory). You can disable metrics for individual reporters by setting the Enabled parameter to false.

If you set the the Enabled parameter under the metrics element to false in the metrics.xml file, metrics will be disabled for all the reporters and it is not possible to enable metrics for individual reporters.

See the following topics for information on configuring each of the available storage types.

- JMX
- CSV
- JDBC

**JMX**

The following parameters in the metrics.xml file can be used to configure a JMX storage for metrics data.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Mandatory/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>This parameter specifies whether metrics monitoring is enabled for JMX or not.</td>
<td>Boolean</td>
<td>true</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

**CSV**

The following parameters in the metrics.xml file can be used to configure a CSV storage for metrics data.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Mandatory/Optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled</td>
<td>This parameter specifies whether metrics monitoring is enabled for CSV or not.</td>
<td>Boolean</td>
<td>false</td>
<td>Mandatory</td>
</tr>
<tr>
<td>Location</td>
<td>The location where the CSV files are stored.</td>
<td>String</td>
<td>${carbon.home}/repository/logs/metrics/</td>
<td></td>
</tr>
</tbody>
</table>
### PollingPeriod

The time interval between polling activities that are carried out to update the metrics dashboard based on latest information. For example, if the polling period is 60, polling would be carried out every 60 milliseconds.

<table>
<thead>
<tr>
<th>PollingPeriod</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The time interval between polling activities that are carried out to update the metrics dashboard based on latest information. For example, if the polling period is 60, polling would be carried out every 60 milliseconds.</td>
<td>Integer</td>
<td>60</td>
</tr>
</tbody>
</table>

### JDBC

The following parameters in the metrics.xml file can be used to configure a JDBC storage for metrics data.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Enabled</strong></td>
<td>This parameter specifies whether metrics monitoring is enabled for JDBC or not.</td>
</tr>
<tr>
<td><strong>DataSourceName</strong></td>
<td>The name of the datasource used.</td>
</tr>
<tr>
<td><strong>PollingPeriod</strong></td>
<td>The time interval between polling activities that are carried out to update the metrics dashboard based on latest information. For example, if the polling period is 60, polling would be carried out every 60 milliseconds.</td>
</tr>
</tbody>
</table>

**ScheduledCleanup**

This element contains parameters relating to scheduled cleanup. The possible values are `Enabled`, `ScheduledCleanupPeriod`, and `DaysToKeep`. Scheduled cleanup involves scheduling a task to clear metric data in the database after a specified time interval. This is done to avoid excessive memory usage.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ScheduledCleanup/Enabled</strong></td>
<td>This parameter specifies whether scheduled cleanup is enabled or not.</td>
</tr>
<tr>
<td><strong>ScheduledCleanup/ScheduledCleanupPeriod</strong></td>
<td>The number of milliseconds that should elapse after a clean-up task before the next clean-up task is carried out.</td>
</tr>
</tbody>
</table>
### ScheduledCleanup/DaysToKeep

The number of days during which the scheduled clean-up task should be carried out.

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;datasources-configuration&gt;</td>
<td>xmlns</td>
<td>The root element. The namespace is specified as: &quot;xmlns:svns: <a href="https://org.wso2.securevault/configuration">https://org.wso2.securevault/configuration</a>&quot;</td>
</tr>
</tbody>
</table>

If you have enabled JDBC, then you also need to specify a datasource configuration, which will be used to create the connection between WSO2 MB and the JDBC data storage system. The `metrics-datasources.xml` is used for configuring this datasource for metrics.

Parameters that can be configured for a datasource are as follows:

<table>
<thead>
<tr>
<th>XML element</th>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;providers&gt;</td>
<td></td>
<td>The container element for the datasource providers.</td>
</tr>
<tr>
<td>&lt;provider&gt;</td>
<td></td>
<td>The datasource provider, which should implement <code>org.wso2.carbon.ndatasource.common.spi.DataSourceReader</code>. The datasources follow a pluggable model in providing datasource type implementations using this approach.</td>
</tr>
<tr>
<td>&lt;datasources&gt;</td>
<td></td>
<td>The container element for the datasources.</td>
</tr>
<tr>
<td>&lt;datasource&gt;</td>
<td></td>
<td>The root element of a datasource.</td>
</tr>
<tr>
<td>&lt;name&gt;</td>
<td></td>
<td>Name of the datasource.</td>
</tr>
<tr>
<td>&lt;description&gt;</td>
<td></td>
<td>Description of the datasource.</td>
</tr>
<tr>
<td>&lt;jndiConfig&gt;</td>
<td></td>
<td>The container element that allows you to expose this datasource JNDI datasource.</td>
</tr>
<tr>
<td>&lt;name&gt;</td>
<td></td>
<td>The JNDI resource name to which this datasource will be bound.</td>
</tr>
<tr>
<td>&lt;environment&gt;</td>
<td></td>
<td>The container element in which you specify the following properties:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>java.naming.factory.initial</code>: Selects the registry service provider as the initial context.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>java.naming.provider.url</code>: Specifies the location of the registry when the registry is being used as the initial context.</td>
</tr>
<tr>
<td>&lt;definition&gt;</td>
<td>type</td>
<td>The container element for the data source definition. Set the type attribute to RDBMS, or to custom if you're creating a custom type &quot;RDBMS&quot; data source reader expects a “configuration” element with the sub-elements listed below.</td>
</tr>
<tr>
<td>&lt;configuration&gt;</td>
<td></td>
<td>The container element for the RDBMS properties.</td>
</tr>
<tr>
<td>&lt;url&gt;</td>
<td></td>
<td>The connection URL to pass to the JDBC driver to establish the connection.</td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>&lt;username&gt;</td>
<td>The connection user name to pass to the JDBC driver to establish connection.</td>
<td></td>
</tr>
<tr>
<td>&lt;password&gt;</td>
<td>The connection password to pass to the JDBC driver to establish connection.</td>
<td></td>
</tr>
<tr>
<td>&lt;driverClassName&gt;</td>
<td>The class name of the JDBC driver to use.</td>
<td></td>
</tr>
<tr>
<td>&lt;maxActive&gt;</td>
<td>The maximum number of active connections that can be allocated from this pool at the same time.</td>
<td></td>
</tr>
<tr>
<td>&lt;maxWait&gt;</td>
<td>Maximum number of milliseconds that the pool waits (when there are no available connections) for a connection to be returned before throwing an exception.</td>
<td></td>
</tr>
<tr>
<td>&lt;testOnBorrow&gt;</td>
<td>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 <code>validationQuery</code> parameter must be set to a non-null string.</td>
<td></td>
</tr>
<tr>
<td>&lt;validationQuery&gt;</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 an SQLException. The default 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>
<td></td>
</tr>
<tr>
<td>&lt;validationInterval&gt;</td>
<td>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).</td>
<td></td>
</tr>
</tbody>
</table>

If you have a clustered setup of WSO2 MB, you can create one JDBC data store (database) and point all nodes in the cluster to the same database. That is, the `metrics-datasources.xml` file (stored in the `<MB_HOME>/repository/conf/datasources` directory) should be updated with the same database information for all the MB nodes in the cluster. Data from each node will be published with the respective IP address, which allows the data to identified separately.

### Sample configuration

Shown below is a sample `metrics.xml` file with the default configurations specifying the types of storages enabled for metrics data. See the above topics for instructions.

```xml
<!--
  This is the main configuration file for metrics
-->  
<Metrics xmlns="http://wso2.org/projects/carbon/metrics.xml">
```
<!-- Enable Metrics -->
<Enabled>false</Enabled>
<!-- Metrics reporting configurations -->

<Reporting>
  <JMX>
    <Enabled>true</Enabled>
  </JMX>
  <CSV>
    <Enabled>false</Enabled>
    <Location>${carbon.home}/repository/logs/metrics/</Location>
    <PollingPeriod>60</PollingPeriod>
  </CSV>
  <JDBC>
    <Enabled>true</Enabled>
    <!-- Source of Metrics, which will be used to identify each metric in database -->
    <!-- Commented to use the hostname -->
    <Source>Carbon</Source>
    <!-- JNDI name of the data source to be used by the JDBC Reporter. -->
    <!-- This data source should be defined in a *-datasources.xml file in conf/datasources directory. -->
    <DataSourceName>jdbc/WSO2MetricsDB</DataSourceName>
    <PollingPeriod>60</PollingPeriod>
    <ScheduledCleanup>
      <!-- Schedule regular deletion of metrics data older than a set number of days. -->
      <!-- It is strongly recommended that you enable this job to ensure your metrics tables do not get extremely large. Deleting data older than seven days should be sufficient. -->
      <Enabled>true</Enabled>
      <ScheduledCleanupPeriod>86400</ScheduledCleanupPeriod>
      <DaysToKeep>7</DaysToKeep>
    </ScheduledCleanup>
  </JDBC>
</Reporting>
Configuring Metrics Properties

The `<MB_HOME>/repository/conf/metrics.properties` file specifies properties that correspond to the gauges in the Metrics Dashboard. See the topic on using JVM metrics for details on using the metrics dashboard for JVM metrics. The level defined for a property in this file determines the extent to which the relevant gauge in the dashboard should be updated with information. The different levels that can be defined for properties are as follows:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>Designates no informational events.</td>
</tr>
<tr>
<td>Info</td>
<td>Designates informational metric events that highlight the progress of the application at coarse-grained level.</td>
</tr>
<tr>
<td>Debug</td>
<td>Designates fine-grained informational events that are most useful to debug an application.</td>
</tr>
<tr>
<td>Trace</td>
<td>Designates finer-grained informational events than the DEBUG.</td>
</tr>
<tr>
<td>All</td>
<td>Designates all the informational events.</td>
</tr>
</tbody>
</table>

If no specific level is configured for a property in the `metrics.properties` file, the metrics root level will apply. The root level is defined as shown in the following example in the `metrics.properties` file.

```
metrics.rootLevel=OFF
```

If you want to change the current root level, you can also use the following command.

```
-Dmetrics.rootLevel=INFO
```

The levels in `metrics.properties` file can be configured to any hierarchy. However, if the level defined for an individual property is different to the level defined for its parent in the hierarchy, the level defined for the individual property will overrule that of the parent. For example, if we have `metric.level.jvm.memory=INFO` in the `<MB_HOME>/repository/conf/metrics.properties` file, all metrics under `jvm.memory` will have INFO as the configured level. However, if you have `metric.level.jvm.memory.heap=TRACE`, the TRACE level would apply for the `metric.level.jvm.memory.heap` property even though it is a child property of `jvm.memory`.

The properties that are included in this file by default are as follows:

- JVM’s direct and mapped buffer pools
- Class loading
- GC
- Memory
- Operating system load
- Threads

JVM’s direct and mapped buffer pools
<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric.level.jvm.buffers</td>
<td>OFF</td>
<td>The gauge showing the current number of distinct buffers.</td>
</tr>
</tbody>
</table>

**Class loading**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric.level.jvm.class-loading</td>
<td>INFO</td>
<td>The gauge showing the number of classes currently loaded for the JVM.</td>
</tr>
</tbody>
</table>

**GC**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric.level.jvm.gc</td>
<td>DEBUG</td>
<td>The gauge for showing garbage collection and memory usage. Monitoring this allows you to identify memory leaks and memory thrash, which have a negative impact on performance.</td>
</tr>
</tbody>
</table>

**Memory**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric.level.jvm.memory</td>
<td>INFO</td>
<td>The gauge for showing the used and committed memory in WSO2 MB.</td>
</tr>
<tr>
<td>metric.level.jvm.memory.heap</td>
<td>INFO</td>
<td>The gauge for showing the used and committed heap in WSO2 MB.</td>
</tr>
<tr>
<td>metric.level.jvm.memory.non-heap</td>
<td>INFO</td>
<td>The gauge for showing the used code cache and used CMS Perm Gen in WSO2 MB.</td>
</tr>
<tr>
<td>metric.level.jvm.memory.total</td>
<td>INFO</td>
<td>The gauge for showing the total memory currently available for the JVM.</td>
</tr>
<tr>
<td>metric.level.jvm.memory.pools</td>
<td>OFF</td>
<td>The gauge for showing the used and available memory for JVM in the memory pool.</td>
</tr>
</tbody>
</table>

**Operating system load**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metric.level.jvm.os</td>
<td>INFO</td>
<td>The gauge for showing the current load imposed by the JVM on the operating system.</td>
</tr>
</tbody>
</table>

**Threads**

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
<td>Level</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>metric.level.jvm.threads</td>
<td>OFF</td>
<td>The parent property of all the gauges relating to the JVM thread pool. The metric level defined for this property will apply to all the rest of the properties in this table. The metric level set via this property to a child property can be overruled if a different level is set for it.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.count</td>
<td>DEBUG</td>
<td>The gauge for showing the number of active and idle threads currently available in the JVM thread pool.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.daemon.count</td>
<td>DEBUG</td>
<td>The gauge for showing the number of active daemon threads currently available in the JVM thread pool.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.blocked.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads that are currently blocked in the JVM thread pool.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.deadlock.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads that are currently deadlocked in the JVM thread pool.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.new.count</td>
<td>OFF</td>
<td>The gauge for showing the number of new threads generated in the JVM thread pool.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.runnable.count</td>
<td>OFF</td>
<td>The gauge for showing the number of runnable threads currently available in the JVM thread pool.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.terminated.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads terminated from the JVM thread pool since you started running the WSO2 MB instance.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.timed_waiting.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads in the Timed_Waiting state.</td>
</tr>
<tr>
<td>metric.level.jvm.threads.waiting.count</td>
<td>OFF</td>
<td>The gauge for showing the number of threads in the Waiting state in the JVM thread pool. One or more other threads are required to perform certain actions before these threads can proceed with their actions.</td>
</tr>
</tbody>
</table>

**Monitoring with Carbon Metrics**

WSO2 MB 3.0.0 and later versions are shipped with the WSO2 Carbon Metrics API, which allows you to monitor statistics of your MB server using Java Metrics. The Java Metrics library consists of a variety of metrics that can be used for monitoring. With the WSO2 Carbon Metrics API, we have enabled all the metrics that are required to effectively monitor WSO2 products.

See the topic on [working with WSO2 Carbon metrics](#) for details on how metrics are enabled in WSO2 MB.

The metrics used in WSO2 MB are categorized into **JVM Metrics** and **Messaging Metrics**. You can access them from the management console of your product as explained in the following topics:

- **JVM Metrics**: Used for monitoring system statistics. These statistics are common to all WSO2 products.
- **Messaging Metrics**: Used for monitoring MB-specific statistics.

**Using JVM Metrics**

JVM metrics are the Java metrics enabled in WSO2 MB for the purpose of monitoring general statistics related to server performance.
The metrics feature is enabled in WSO2 MB by default. See the topic on working with WSO2 Carbon metrics for details on how metrics are enabled.

Follow the steps given below for instructions on using the JVM Metrics dashboard.

1. Log in to the management console of MB and click Monitor -> Metrics -> JVM Metrics.

2. The View Metrics page will open. At the top of this page, you will find the following panel:

3. First, select the Source from the drop-down list. In a clustered setup, you must specify the MB node that you want to monitor.

4. You can specify the time interval for which the statistics displayed are valid. By default you will see statistics from the last 5 minutes.

5. In the Views section, you will find buttons corresponding to the different types of information that you want view. You can click the required button to view the statistics. Given below are the statistics corresponding to each button:
   - **CPU**
• **Memory**

**JVM Memory (MB)**

- Heap Init
- Heap Used
- Heap Committed
- Heap Max
- Non-Heap Init
- Non-Heap Used
- Non-Heap Committed
- Non-Heap Max

**Physical Memory Details (MB)**

- Free Physical Memory Size
- Total Physical Memory Size
- Free Swap Space Size
- Total Swap Space Size
- Committed Virtual Memory Size

• **Threading**
Using Messaging Metrics

Messaging metrics are the Java metrics enabled in WSO2 MB for the purpose of monitoring MB-specific statistics.

The metrics feature is enabled in WSO2 MB by default. See the topic on working with WSO2 Carbon metrics for details on how metrics are enabled.
Follow the steps given below for instructions on using the **Messaging Metrics** dashboard.

1. Log in to the management console of MB and click **Monitor -> Metrics -> Messaging Metrics**.

![Dashboard Screenshot](image)

2. The **View Metrics** page will open. At the top of this page, you will find the following panel:

![View Metrics Panel](image)

3. First, select the **Source** from the drop-down list. In a clustered setup, you must specify the MB node that you want to monitor.

4. You can specify the time interval for which the statistics displayed are valid. By default you will see statistics from the last 5 minutes.

5. In the **Views** section, you will find buttons corresponding to the different types of metrics that you want view. You can click the relevant button to view the statistics. Given below are the statistics corresponding to each button:

   - **Disruptor**

   ![Disruptor Graph](image)

   **Metric**

   | Total Messages in Inbound Disruptor |

   | Description |

   | The **Disruptor** is a new open-source concurrency framework, designed as a high performance mechanism for inter-thread messaging. The current number of messages in the inbound disruptor can be viewed here. |
### Publish & Subscribe

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Queue Subscribers</td>
<td>This metric shows the total number of active queue subscribers for a particular MB node. This is an INFO level metric.</td>
</tr>
<tr>
<td>Total_topic Subscribers</td>
<td>The total number of active topic subscribers.</td>
</tr>
<tr>
<td>Total Channels</td>
<td>The total number of active channels.</td>
</tr>
</tbody>
</table>

### Messages & Acknowledgements

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
</table>

Total Acks in Inbound Disruptor: The current number of acknowledgments in the inbound disruptor.

Total Messages in Outbound Disruptor: The current number of messages in the outbound disruptor.
<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Messages Received/sec</strong></td>
<td>This metric provides the number of messages received per second by a particular MB node. This metric is calculated when a message reaches the server.</td>
</tr>
<tr>
<td><strong>Messages Published/sec</strong></td>
<td>This metric provides the number of messages published per second. This metric is calculated when the server publishes a message to a subscriber.</td>
</tr>
<tr>
<td><strong>Acknowledges Received/sec</strong></td>
<td>This metric provides the number of acknowledgments received from publishers per second.</td>
</tr>
<tr>
<td><strong>Acknowledges Sent/sec</strong></td>
<td>This metric provides the number of acknowledgments sent to publishers per second.</td>
</tr>
</tbody>
</table>

**Database**

- **Database Read Time (milliseconds)**
- **Database Read Rate (reads/second)**
JMS Subscribers and Publishers

- Creating Durable Topic Subscriptions
- Subscribing to Hierarchical Topics
- Redelivering Messages to Subscriber
- Setting the Routing Key for Messages
- Acknowledging Message Delivery
- Working with JMS Messages
- Setting the Connection URL
- Configuring Message Expiration
- Handling Distributed Transactions
Creating Durable Topic Subscriptions

Durable topics persist messages, which allows the messages to be received later if a subscriber is not online. A durable topic subscription is useful when a subscriber client needs to be able to receive messages that are published even when the client is inactive.

- Creating durable topic subscriptions
- Sharing a durable topic subscription
  - Understanding shared durable topic subscriptions
  - Shared durable subscriptions in a clustered setup
  - Enabling shared durable topic subscription in WSO2 MB

Creating durable topic subscriptions

If a client subscriber creates a durable subscription to the topic, the subscriber will be able to recover the messages that were published to the topic while the client was inactive. For example, a durable topic subscriber will receive all the messages that are published to the topic while the subscriber is active. However, if the subscriber becomes inactive for a time period and later returns to active state, the messages that were published during the inactive period will be fetched from the database and dispatched to the subscriber.

The following points will elaborate on the different elements that you need to be mindful of when creating a durable topic subscription:

- Creating a durable topic subscription is similar to creating a nondurable subscription, but you must additionally provide a name that identifies the durable subscription as shown below.

```java
// Create a durable subscriber, supplying a uniquely-identifying name
TopicSubscriber sub = session.createDurableSubscriber(topic, "mySub1_0001");
```

- Shown below is the connection URL used for subscribing to WSO2 MB during an AMQP session. See the topic on setting the connection URL for more information.

```java
connectionfactory.Connectionfactory = amqp://admin:admin@clientID/carbon?brokerlist='tcp://192.168.10.5:5672'
```

- Reusing a durable topic subscription: To reuse a durable subscription that was created previously, the subscriber client must create a durable topic subscriber using a session with the same client identifier as that associated with the durable subscription.

Tenant-specific durable subscriptions

When creating a JMS subscription in tenant mode, the username, topic name and subscription ID should be set in the following manner:

```java
username = username!tenantdomain
Example:  username = testuser!test.com;

topicName = tenantdomain/topicName;
Example:  topicName = test.com/testTopic;
```
Unsubscribing non-durable topic subscriptions

Nondurable message consumers in the publish/subscribe domain automatically deregister themselves when their close() method is called or when they fall out of scope. However, if you want to terminate a durable subscription, you must explicitly notify the broker. To do this, use the unsubscribe() method of the session and pass in the name that identifies the durable subscription:

```java
// Unsubscribe the durable subscriber created above
session.unsubscribe( "mySub1_0001" );
```

It is not possible to unsubscribe a durable consumer from inside the onMessage() method in the listener. If you need to unsubscribe, write a separate class/thread etc. with a different session, give the subscription ID and unsubscribe.

Sharing a durable topic subscription

As explained in the previous section, durable topic subscriptions are unique. That is, the subscription ID and client ID, which are used for identifying the durable subscription should be unique. This means that it is not possible to have multiple subscriber clients sharing a single subscription. Now let's look at what is meant by 'sharing' a durable subscription and how this possibility can be enabled for WSO2 MB.

Understanding shared durable topic subscriptions

Consider the following scenario:

In the above example, there are two subscribers (A and B) with durable subscriptions to the topic and each subscriber performs a specific task with the messages received through the subscription. Therefore, when messages are published to the topic, copies of the messages are dispatched to both subscribers and each subscriber will perform a unique task. This is how the default configuration of WSO2 MB works. You will note here that it is not possible to have two subscriber clients performing the same task. For example, consider that the task that should be performed by subscriber A is time consuming and requires a lot of CPU memory. In such a situation,
we need to be able to scale the work load of subscriber A among multiple subscriber clients. The following diagram depicts this requirement:

As shown above, what we need is to establish a single subscription where the messages received are shared by multiple subscriber clients using the round-robin method. In the default set up of WSO2 MB, this is not possible, because when there are multiple subscriber clients connecting to the topic, copies of the same messages are dispatched to all the subscriber clients. Therefore, sharing a subscription is only possible if the subscription ID can be shared by all the subscriber clients.

Shared durable subscriptions in a clustered setup

Now, let's look at how this functionality of sharing a durable topic subscription works when the brokering facility is distributed among multiple MB nodes. Consider the following scenario:

In this example, we have a cluster of 3 WSO2 MB nodes and all 3 nodes have the 'SportsNews' topic created. The following points will elaborate how this clustered setup works:

- There are multiple durable topic subscribers (Sub-1, Sub-2, Sub-3 and Sub-4) connecting to the MB cluster using a shared subscription ID (Sub1). Note that the subscribers Sub-2 and Sub-3 are connecting to the same MB node in the cluster.
- There are 3 other clients (Pub-1, Pub-2 and Pub-3) publishing messages to the SportsNews topic.

When a client publishes messages to a topic, the client must connect to a particular MB node in the cluster. In the above example, publishers Pub-1, Pub-2 and Pub-3 connects to MB nodes A, B and C respectively. However, since all three nodes belong to a clustered setup, the messages published to each node are received by the slot manager of the cluster and not by the specific node to which the publisher sends the message. The slot manager then allocates all the received messages among the 3 nodes. This means that the messages published to the MB cluster can be dispatched to the subscriber clients from any one of the 3 nodes in the cluster.
• All the messages published to the cluster will be evenly distributed among the subscribers, because the subscription is shared. Therefore, the total number of messages that are published to the MB cluster will equal the total number of messages that are received by all the subscribers \((x_1+x_2+x_3 = y_1+y_2+y_3+y_4)\).

Note that if you unsubscribe one shared topic subscriber, it will affect all other durable topic subscriptions with the same subscription ID.

**Enabling shared durable topic subscription in WSO2 MB**

You can enable shared topic subscriptions in WSO2 MB by following the steps given below.

1. Open the broker.xml file stored in the `<MB_HOME>/repository/conf` folder.
2. Enable the `allowSharedTopicSubscriptions` element as shown below. Note that this property is only applicable to AMQP.

```xml
<broker>
  <transports>
    <amqp enabled="true">
      <allowSharedTopicSubscriptions>true</allowSharedTopicSubscriptions>
    </amqp>
  </transports>
</broker>
```

3. The JMS clients that are subscribing to the topic should create durable subscriptions using the same subscription ID. See the previous section on creating durable topics subscriptions.

**Subscribing to Hierarchical Topics**

Many publish/subscribe message systems support a tree-organized topic arrangement, many of which predate JMS. For example, suppose the server in question supports topic hierarchies, and it is managing the topic "Games" and the subtopics "Cricket" and "FootBall". In this case, the characteristics of the server's publish/subscribe model and/or configuration parameters determine whether a subscription to "Games" would also constitute a subscription to all subtopics of "Games", so that a message published to "Cricket" and a message published to "FootBall" would be distributed to subscribers of "Games".

Although the JMS specification does not require support for topic hierarchies, they can be a powerful tool, and WSO2 Message Broker supports them. Following is an example configuration demonstrating hierarchical topics.
When subscribing to topics in the above hierarchy, you have the following options:

- **Topic Only** - Shown below is how the JMS client can subscribe to a specific topic only. For example, if you subscribe to the "Cricket" topic, the client will receive messages that are published to "Cricket" only. The client will not receive messages published to the "Sri Lanka", "India", "Football" or "Games" topics.

```java
String topicName_parent = "Games";
String topicName_child = "Games.Cricket";

Topic childTtopic = (Topic) ctx.lookup(topicName_child);
TopicSubscriber topicSubscriber =
topicSession.createSubscriber(childTtopic);
```

- **Immediate Children** - Shown below is how the JMS client subscribes to the first level of sub-topics but not to the topic itself. For example, if you subscribe to the "Games" topic, you will get messages published to "Football" or "Cricket" but not messages published to "Games", "Sri Lanka", or "India".

```java
String topicName_parent = "Games";
String topicName_child = "Games.*";
Topic childTtopic = (Topic) ctx.lookup(topicName_child);
TopicSubscriber topicSubscriber =
topicSession.createSubscriber(childTtopic);
```

- **Topic and Children** - Shown below is how the JMS client subscribes to the topic and all its sub-topics. For example, if you subscribe to the "Cricket" topic using, the client will receive messages published to "Cricket", "Sri Lanka", and "India".

```java
String topicName_parent = "Games";
String topicName_child = "Games.*";
Topic childTtopic = (Topic) ctx.lookup(topicName_child);
TopicSubscriber topicSubscriber =
topicSession.createSubscriber(childTtopic);

Note that subtopics can also have sub-topics repeating recursively.
String topicName_parent = "Games";
String topicName_child = "Games.Cricket.#";
Topic childTtopic = (Topic) ctx.lookup(topicName_child);
TopicSubscriber topicSubscriber =
topicSession.createSubscriber(childTtopic);

The **Immediate Children** and **Topic and Children** options explained above ensure that the client is automatically subscribed to the immediate child topic or all child topics of the parent topic, even if a given child topic restricts subscriber permission to the user’s role.

When creating a JMS subscription in tenant mode, the username, topic name and subscription ID should be set in the following manner:

```
username = username!tenantdomain
Example: username = testuser!test.com;

topicName = tenantdomain/topicName;
Example: topicName = test.com/testTopic;

subscription Id=tenantdomain/subscription id
Example: subscription Id=test.com/K1;
```

For a sample of using hierarchical topics, see [Creating Hierarchical Topic Subscriptions](#).

### Redelivering Messages to Subscriber

When you use queues and topics in WSO2 Message Broker, messages will be removed from the message store once the message consumers acknowledge that the messages are received.

You can configure message redelivery to a subscriber as follows:

- Configuring the maximum number of attempts to redeliver a message
- Delaying message redelivery to a subscriber

#### Configuring the maximum number of attempts to redeliver a message

If you want to limit the number of times the message broker attempts to redeliver the message, you can set the `maximumRedeliveryAttempts` element in the `broker.xml` file as follows:

```
<!--Broker will drop the message after the configured number of delivery
attempts for each message.-->
<maximumRedeliveryAttempts>10</maximumRedeliveryAttempts>
```

Note that the above configuration specifies the total number of attempts to redeliver the message after the original delivery attempt. For example, when the first attempt to send the message fails, there will be another 10 attempts to
redeliver the message. After the maximum number of attempts to redeliver the message are breached, the message is sent to the Dead Letter Channel. This is useful when the client application does not acknowledge the message because an operation on the message failed.

If the message is successfully delivered on a redelivery attempt, the JMSRedelivered field is set to 'true' in the message header, allowing the client to determine whether the message was delivered by the original attempt or on a later attempt.

**Delaying message redelivery to a subscriber**

With this feature, a subscriber client will be able to delay the redelivery of messages to the subscriber (when the message has already been rejected). Any message that comes to the client with the redelivery flag true will get delayed. The delay can be set using a custom system property. New messages that are received by the client does not get blocked due to the redelivered message. Therefore, redelivered messages are unordered.

When using this feature, it may be necessary to use “Per-Message Acknowledgements” depending on the use case.

This system property can be set as shown below. This value is given in milliseconds. The default value is 0.

```
System.setProperty("AndesRedeliveryDelay", "10000");
```

**Setting the Routing Key for Messages**

The Dead Letter Channel (DLC) is a sub-set of a queue, which is used for storing messages that have not been delivered to the intended subscriber. The DLC provides you the option of deleting these messages, retrieving them or rerouting them to another queue. In the case of rerouting the message to a different queue, the message will be received by the clients subscribed to this second queue. Therefore, since the subscribers of the second queue are not expecting to receive messages from another queue, it will be necessary for such subscriber clients to identify the destination (subscriber) to which the message was originally sent.

To achieve this, we are using a custom JMS property that embeds the routing key to the message. We are embedding the routing key to a JMS property of the message from the andes-client side. The embedding is enabled only when the “AndesSetRoutingKey” system property is set to a non-null value (any value other than null) in the publisher client. When the mentioned system property is set, each message will have a JMS property of type "string" with the “JMS_ANDES_ROUTING_KEY” name, which will contain the routing key.

**Configuring the publisher client**

The custom JMS property should be enabled for the publisher client as shown below.

```
System.setProperty("AndesSetRoutingKey", "1");
```

**Configuring the subscriber client**

The subscriber client can retrieve the JMS property in the message as shown below.

```
System.out.println("PROP:" +
message.getStringProperty("JMS_ANDES_ROUTING_KEY");
```
Acknowledging Message Delivery

A subscriber client receiving messages from a queue or topic in the broker should be configured to send an acknowledgement back to the broker when the messages are received. There are several acknowledgement methods that can be used by the subscriber.

- Configuring the time taken to acknowledge messages
- Configuring standard JMS message acknowledgment patterns
- Configuring per-message acknowledgment

Configuring the time taken to acknowledge messages

There are several acknowledgement models defined in JMS specification 1.1. To configure the time within which consumers can acknowledge messages, you can set the AndesAckWaitTimeOut entry in the JMS client as follows:

```java
System.setProperty("AndesAckWaitTimeOut", "30000");
```

If the acknowledgment fails within the above time, the client informs the MB server that the message is rejected. The message is then scheduled to be redelivered later by the server.

Configuring standard JMS message acknowledgment patterns

The following are acknowledgment patterns introduced by JMS:

- Auto Acknowledge
- Duplicates Allowed
- Client Acknowledge
- Transacted Acknowledgement

Configuring per-message acknowledgment

Per-message acknowledgment can be used to ensure that the subscriber client acknowledges each message that is received. The subscriber can enable this feature by using one of the following options when creating the session.

- Use the org.wso2.andes.jms.Session.PER_MESSAGE_ACKNOWLEDGE enum available in the andes-client JAR as shown below.

```java
import org.wso2.andes.jms.Session;
QueueSession queueSession = queueConnection.createQueueSession(false,
Session.PER_MESSAGE_ACKNOWLEDGE);
```

- Alternatively, you can use the value 259 as shown below.

```java
QueueSession queueSession = queueConnection.createQueueSession(false, 259);
```

Following is a sequence diagram on how an example scenario will work:
Working with JMS Messages

- JMS Message Types and Header Fields
- Using Message Selectors

JMS Message Types and Header Fields

Explained below are the message types and message headers that are supported by WSO2 Message Broker (WSO2 MB). Find more information on how to use message selectors from here.

Note that message selectors are not supported in a clustered setup of WSO2 Message Broker (WSO2 MB).

JMS Message Types supported by WSO2 MB

WSO2 MB supports all five types of JMS messages named TextMessage, BytesMessage, MapMessage, ObjectMessage and StreamMessage. A JMS client can send or receive any type of message from the above five and the content of the messages can be viewed using the WSO2 MB Queue Browser window. However, viewing the message body of an ‘ObjectMessage’ using the queue browser is not supported.
**JMS Message Headers supported by WSO2 MB**

A JMS client can create a message and set various fields of the message header before it is sent to a queue in WSO2 MB. However, as supported by the JMS specification, there are several message header fields which cannot be explicitly set by a JMS client. Hence even though client sets these fields, those will be replaced at the JMS provider level once it is received by WSO2 MB.

The following table displays a list of JMS message headers and in which level they can be configured.

<table>
<thead>
<tr>
<th>JMS Message Header</th>
<th>Can be Set by Whom</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>JMS providers can set this field when a message is sent. However, this header should also be set at broker level in WSO2 MB, because the client level header will be ignored by the broker.</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>As WSO2 MB uses persistent storage in Standalone mode, the DeliveryMode will be set to ‘1’ by default. It is not possible to change this at client level.</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>This header is not supported in WSO2 MB. The value passed by the producer/publisher will be delivered to the client without changing the value.</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>This header is not supported in WSO2 MB. Even if the JMS provider sets a priority, the broker will ignore this header and messages will be delivered in the order that they were sent to the broker.</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>JMS providers can set this field when a message is sent. However, this header should also be set at broker level in WSO2 MB, because the client level header will be ignored by the broker.</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>The value passed by the producer/publisher will be delivered to the client without changing the value.</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>By using a JMS Client</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>By using a JMS Client</td>
</tr>
<tr>
<td>JMSType</td>
<td>By using a JMS Client</td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td>From the JMS provider only</td>
</tr>
</tbody>
</table>

**Using Message Selectors**

A message selector allows a JMS consumer to be more selective about the messages that it receives from a particular topic or queue. A message selector uses message properties and message headers as criteria in conditional expressions. These expressions use Boolean logic to declare the messages that are delivered to a client. The message consumer will only receive messages where the headers and properties match the selector. There are different patterns that can be used in selector strings to filter messages and the broker (JMS provider) filters messages according to that query. It is not possible for a message selector to filter messages on the basis of the content of the message body. See the list of supported message types and header fields in WSO2 MB.

Note that message selectors are not supported in a clustered setup of WSO2 Message Broker (WSO2 MB).

See the following topics for details:

- Understanding how message selectors work with WSO2 MB
Understanding how message selectors work with WSO2 MB

The way selectors work depends on the type of subscription that is created from JMS clients. The following sections explain the different scenarios:

**Queue subscriptions**
- Only messages that match the selector will be delivered to the subscriber.
- If at least one subscriber out of a few subscribers for a queue has a matching selector, the message will be delivered to that subscriber.
- If more than one subscriber has a matching selector, the messages will be delivered in round robin manner between those subscribers.
- If none of the subscribers have matching selectors, that message will be placed in the Dead Letter Channel (DLC).

**Topic subscriptions (non-durable)**
- Only messages that match the selector will be delivered to the subscriber.
- If none of the subscribers have selectors matching the message, the message is dropped.
- Selectors are not evaluated when messages are received by the Broker. Messages will be written anyway if there are subscribers for that topic, and the selectors are evaluated upon delivery.

**Topic subscriptions (durable)**
- Only messages that match the selector will be delivered to the durable topic subscriber.
- When keeping a copy of a message coming for a topic for the particular subscription ID, selectors are not evaluated. They are evaluated upon delivery only.
- If a message does not match with the selector for a particular durable topic subscription, it will be routed to the Dead Letter Channel (DLC).

**Messages routed to DLC**
- If none of the subscribers have matching selectors with a message and if the message is for a durable queue or **durable subscription**, the message is routed to the DLC.
- User has the option to delete messages selectively if they are unnecessary.
- It is possible to create a new queue and route all unmatched messages to that queue if required.
- It is possible to create a new subscriber with a matching selector for some or all messages routed to the DLC and route all messages to the original queue again. Then, if there are newly created matching subscriptions, those messages will be delivered accordingly. Other messages will go to DLC queue once again.

**Using message headers as selector criteria**
A message header contains a number of predefined fields that contain values that both clients and providers can use to identify and to route messages. See the list of supported message types and header fields in WSO2 MB.

The following are examples of message header selector strings that are supported:

<table>
<thead>
<tr>
<th>Message Header</th>
<th>Selector String Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSDestination</td>
<td>JMSDestination=destqueue</td>
</tr>
<tr>
<td>JMSMessageID</td>
<td>JMSMessageID=JMSMessageID</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>JMSTimestamp=1396370353826</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>JMSCorrelationID='sri Lanka'</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td></td>
</tr>
<tr>
<td>JMSType</td>
<td>JMSType='AAAA'</td>
</tr>
</tbody>
</table>

The following message headers will not be filtered by the broker as they are handled by the JMS provider and the default values override the selector string:

- JMSDeliveryMode
- JMSExpiration
- JMSPriority
- JMSRedelivered

Using message properties as selector criteria

If you need values in addition to those provided by the header fields, you can create and set properties for the messages. See the list of supported message types and header fields in WSO2 MB.

The following is a list of message property selector strings that are supported:

<table>
<thead>
<tr>
<th>Message Properties</th>
<th>Selector String Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByteProperty</td>
<td>ByteMsgID=127</td>
</tr>
<tr>
<td>ShortProperty</td>
<td>ShortMsgID=32,767</td>
</tr>
<tr>
<td>IntProperty</td>
<td>IntMsgID=400000000</td>
</tr>
<tr>
<td>LongProperty</td>
<td>LongMsgID=121232222</td>
</tr>
<tr>
<td>FloatProperty</td>
<td>FloatMsgID = 0.0f</td>
</tr>
<tr>
<td>DoubleProperty</td>
<td>DoubleMsgID = 20011111000.12120</td>
</tr>
<tr>
<td>StringProperty</td>
<td>StrMsgID='100'</td>
</tr>
<tr>
<td>BooleanProperty</td>
<td>BoolMsgID=false</td>
</tr>
<tr>
<td>ObjectProperty</td>
<td></td>
</tr>
</tbody>
</table>

Examples of selector patterns

Given below are some sample selector patterns.
Selector Patterns | Example
---|---
Header='value' OR Header='value' | JMSType='AAAA' OR JMSPriority=4
Property='value' AND Property='value' | Country='SL' AND ID=1
Header='value' OR Property='value' | JMSType='AAA' OR msgID='1'
Header='value' AND Property='value' AND Property='value' | JMSType = 'AAA' AND color = 'red' AND weight=3500
(Property='value' OR Header='value') AND Property='value' | (Country='SL' OR JMSType='AAA') AND ID=1

**Subscribing to topics/queues using selectors**

The `createConsumer` and `createDurableSubscriber` methods allow you to specify a message selector as an argument when you create a message consumer.

You can create a message selector subscriber for a topic as shown in the example given below. Note that the `nolocal` parameter is set to 'false' as it is not supported by WSO2 MB. However, messages will be delivered to the subscriber irrespective of the `nolocal` parameter.

```java
String messgeSelector="JMSType='AAAA';
topicSession.createSubscriber(topic,messgeSelector,false)
```

The following examples illustrate how you can create a subscriber using selectors. The `messageSelector` property is set to "releaseYear < 1980" in all of these examples.

**Queue subscription**

```java
QueueSession queueSession =
    queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);

//Receive message
Queue queue = (Queue) ctx.lookup(queueName);

MessageConsumer queueReceiver = queueSession.
    createConsumer(queue, "releaseYear < 1980");
```

**Non-durable topic subscription**
TopicSession topicSession =
    topicConnection.createTopicSession(false,
    QueueSession.AUTO_ACKNOWLEDGE);

Topic topic = topicSession.createTopic(topicName);

javax.jms.TopicSubscriber topicSubscriber = topicSession.
    createSubscriber(topic, "releaseYear < 1980", false);

Durable topic subscription

TopicSession topicSession =
    topicConnection.createTopicSession(false,
    QueueSession.AUTO_ACKNOWLEDGE);
Topic topic = topicSession.createTopic(topicName);

javax.jms.TopicSubscriber topicSubscriber = topicSession.
    createDurableSubscriber(topic, "mySub4", "releaseYear < 1980", false);

Publishing messages using selectors

When publishing messages you need to add the ‘releaseYear’ JMS property
and set the value as shown below.

javax.jms.QueueSender queueSender = queueSession.createSender(queue);
int currentMsgCount = 0;

while(currentMsgCount < this.messageCount) {
    TextMessage textMessage = queueSession.
        createTextMessage("test: (" + currentMsgCount + ")");
textMessage.setLongProperty("releaseYear", 1990);
queueSender.send(textMessage);
System.out.println("Sent message: " + currentMsgCount);
currentMsgCount ++;
}

Setting the Connection URL

When a JMS client connects to WSO2 Message Broker (WSO2 MB), the connection parameters are specified using
the connectionfactory interface as shown below.

- connectionfactory.ConnectionFactory = <Connection_URL>
- connectionfactory.QueueConnectionFactory = <Connection_URL>
- connectionfactory.TopicConnectionFactory = <Connection_URL>

See the following topics:
Parameters used in the connection URL

The connection URL takes the following format, when the AMQP transport is used:

```
amqp://[<user>:<pass>@]<clientid>[/<virtualhost>]?<option>='<value>'[&<option>='<value>']
```

For example:

```
connectionfactory.QueueConnectionFactory =
amqp://admin:admin@clientID/carbon?brokerlist='tcp://IP1:5672'
```

Note that the clientID is used for identifying the client and is only applicable for **durable topic subscriptions**. However, since this value is not validated in WSO2 MB, you can enter any arbitrary string as the clientID.

Now, let’s look at the parameters used in the above URL format:

- `<user>:<pass>@`: This is the username:password that will be used to connect to WSO2 MB. Note that this user should have the required permissions granted in WSO2 MB. See the section for user permissions in WSO2 MB for more information on how permissions are defined.

- `/<virtualhost>`: The name of the virtual host, where the virtual host is a path that acts as a namespace. A name consists of any combination of the following: At least one alphanumerical value [A-Za-z0-9] and optionally special characters [.-_+!=:].

- `<option>='<value>'`: You can enter multiple options with values as explained below. These options should be separated by `&`.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| brokerlist | The list of brokers to use for this connection. The value should contain the URL of the broker as well as any other optional values. You can add any number of broker URLs separated by `,` as shown below.  

```
brokerlist='<broker url>;<broker url>'
```

See the topic below on Using the brokerlist option for details on how to configure the broker URL. |
| failover | This option controls how failover occurs when you have a list of brokers. The value used with the failover option should contain the failover method as well as any other optional values. See the topic below on Using the failover option for details. |

Using the 'brokerlist' option

When you use the brokerlist option, the broker URL should be as follows:
Now, let’s look at each of the parameters used in the broker URL:

- `<transport>`: The transport should be TCP.

  To support more transports, the "client.transportTransportConnection" class needs to be updated along with the parsing to handle the transport.

- `<host>`: The IP address.
- `<port>`: The default AMQP port of 5672 is used if no port is specified.
- `<option>='value'`: Each broker can have additional options that are specific to that broker. The following are the options that are currently implemented:

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>retries</td>
<td>1</td>
<td>The number of retry attempts when connecting to the broker.</td>
</tr>
<tr>
<td>ssl</td>
<td>false</td>
<td>Use SSL on the connection.</td>
</tr>
<tr>
<td>connecttimeout</td>
<td>30000</td>
<td>How long (in milliseconds) to wait for the connection to succeed.</td>
</tr>
<tr>
<td>connectdelay</td>
<td>none</td>
<td>How long (in milliseconds) to wait before attempting to reconnect.</td>
</tr>
</tbody>
</table>

### Using the ‘failover’ option

When you use the failover option, the following format should be followed:

```
failover='method[?options]'
```

Now, let’s look at the parameters in the above format:

- `<method>`: Listed below are the supported methods. Note that ‘singlebroker’ is used when only one broker is present and the 'roundrobin' method is used with multiple brokers. The 'nofailover' method is useful if you are using a 3rd party tool that has its own reconnection strategy that you wish to use.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>singlebroker</td>
<td>This will only use the first broker in the list.</td>
</tr>
<tr>
<td>roundrobin</td>
<td>This method tries each broker in turn.</td>
</tr>
<tr>
<td>nofailover</td>
<td>[New in 0.5] This method disables all retry and failover logic.</td>
</tr>
</tbody>
</table>

The `<method>` value in the URL may also be any valid class on the classpath that implements the FailoverMethod interface.
**<option>: The following options should be used for the failover method.**

<table>
<thead>
<tr>
<th>Option</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclecount</td>
<td>1</td>
<td>The number of times to loop through the list of available brokers before failure.</td>
</tr>
</tbody>
</table>

**Example**

For example, see how multiple brokers are defined in the connection URL, along with failover parameters.

```java
connectionfactory.QueueConnectionFactory = 
```

**Configuring Message Expiration**

According to the default setting in WSO2 Message Broker, messages published to a queue do not expire. Therefore, if required, a publisher can set an expiration period when publishing messages. This is done by setting the timeToLive (TTL) value at the publisher level.

If the specified timeToLive value is 0, the message never expires. When a message is published to the broker, the message expiration time is calculated by adding the TTL value sent by the publisher to the current time. Messages that are not delivered before the specified expiration time are deleted. The destruction of obsolete messages conserves storage and computing resources. See the section below on deleting expired messages to understand how the deletion process works.

See the following topics for instructions:

- Setting the expiration time for messages
  - Setting a default expiration time for all messages
  - Setting an expiration time for a specific message
- Deleting expired messages

See **Setting Message Expiration** for a sample demonstration of this functionality.

**Setting the expiration time for messages**

The TTL value for messages can be specified by the publisher in two ways:

- **Setting a default expiration time for all messages**

  You can use the `setTimeToLive` method of the `MessageProducer` interface to set a default expiration time for all messages sent by that producer as shown below.
MessageProducer messageProducer =
        session.createProducer(requestDestination);
    //time to live value in milliseconds
    messageProducer.setTimeToLive(1000);

- Setting an expiration time for a specific message

You can use the long form of the send or the publish method to set an expiration time for a specific message. The fourth argument sets the expiration time in milliseconds. In the following example, the TTL value of a message is set to 10 seconds:

```
QueuePublisher.publish(message, DeliveryMode.NON_PERSISTENT, 3,
                      10000);
```

Deleting expired messages

Messages will not be delivered to consumers after the TTL expires. These expired messages will be deleted in batches periodically. The following values set in the `broker.xml` (stored in the `<MB_HOME>/repository/conf` directory) file controls the process of deleting expired messages.

1. When messages are received by the broker, they are first stored in a database. The system periodically checks the database for received messages that have expired the TTL and deletes them. Note that at this stage, the system only checks for messages that are not assigned to a slot delivery worker. The following property in the `broker.xml` file specifies the time gap (in seconds) between periodic message deletion from the database.

```
<periodicMessageDeletionInterval>900</periodicMessageDeletionInterval>
```

2. When checking the database for expired messages, the messages that are about to be assigned to a slot delivery worker should not be deleted (even if they have expired) since that will interfere with the slot delivery functionality. Therefore, the following property in the `broker.xml` file specifies a safety buffer to identify the messages (number of slot allocations waiting to be assigned) that will not be affected by the periodic message deletion. In the following example, the first three slot allocations that are waiting to be assigned to the slot delivery worker from the database should not be affected by periodic message deletion even if the TTL of the allocated messages have expired. These expired messages will later be deleted at the message delivery path.

```
<safetySlotCount>3</safetySlotCount>
```

3. Once messages are assigned to the slot delivery worker, the delivery process is started. If any expired messages are identified in this delivery path, they are added to a queue for batch deletion. The following property in the `broker.xml` file specifies the time gap (in seconds) between batch deletes in the delivery path.
Handling Distributed Transactions

WSO2 Message Broker 3.2.0 supports XA transactions (distributed transactions). This capability ensures that a message delivered to multiple queues in WSO2 MB can be reverted if at least one queue fails to accept the message.

For example, consider a use case, where an ESB is configured to distribute messages to three separate queues (mbqueue1, mbqueue2 and mbqueue3) defined in WSO2 MB. When the message is dispatched from the ESB to the queues, if the message delivery to at least one queue fails, the message should be rolled back without delivering to any one of the queues.

Before you begin:

You can configure the maximum number of distributed transactions that can be handled in parallel by WSO2 MB. To do this, set the following parameter in the broker.xml file (stored in the `<MB_HOME>/repository/conf` directory).

```xml
<transaction>
  <maxParallelDtxChannels>20</maxParallelDtxChannels>
</transaction>
```

Let's see how this use case can be achieved using WSO2 Enterprise Service Bus (WSO2 ESB) as the ESB.

1. Follow the instructions on configuring WSO2 ESB with WSO2 MB.
2. Start WSO2 MB and create three queues: mbqueue1, mbqueue2, and mbqueue3.
3. Start WSO2 ESB and add a proxy service to mediate messages to WSO2 MB. To handle XA transactions, the proxy service should be configured as shown below. In this example, WSO2 ESB listens to a JMS queue named MyJMSQueue and consumes messages and sends messages to multiple JMS queues in a transactional manner.
<property expression="$body" name="BEFORE"/>
</log>
<property expression="get-property('MessageID')"
    name="MESSAGE_ID_B"
    scope="operation"
    type="STRING"/>
<property description="FailureResultProperty"
    name="failureResultProperty"
    scope="default">
    <result xmlns="">failure</result>
</property>
</log>
<enrich>
    <source clone="true" xpath="$ctx:failureResultProperty"/>
    <target type="body"/>
</enrich>
<property expression="$body" name="AFTER"/>
</log>
<property name="BEFORE1" scope="axis2" type="STRING"
    value="ABCD"/>
<callout
    <source type="envelope"/>
    <target
        xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
</callout>
<callout
    <source type="envelope"/>
    <target
        xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
</callout>
<callout

<source type="envelope"/>
<target
xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
>
</callout>
<drop/>
</inSequence>
<faultSequence>
<log level="custom">
  <property name="Transaction Action" value="Rollbacked"/>
</log>
<callout
<source type="envelope"/>
<target
xmlns:s11="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:s12="http://www.w3.org/2003/05/soap-envelope"
>
</callout>
</faultSequence>
</target>
</callout>
<parameter name="transport.jms.Destination">MyJMSQueue</parameter>
<parameter name="transport.jms.ContentType">
<rules xmlns=""
>
  <jmsProperty>contentType</jmsProperty>
  <default>application/xml</default>
</rules>
4. Now, you can disable one queue in the WSO2 MB and send a message to the ESB. The proxy service will attempt to dispatch the message to all three queues. However, since one queue is unavailable, the message will not be delivered to any of the queues.

**Configuration Files**

You can configure the behavior of WSO2 Message Broker using the following files. Click a file to see a reference to the properties you can configure.

- Configuring broker.xml
- Configuring qpid-config.xml
- Configuring qpid-virtualhosts.xml
- Configuring axis2.xml
- Configuring carbon.xml
- Configuring catalina-server.xml
- Configuring master-datasources.xml
- Configuring registry.xml
- Configuring user-mgt.xml
- Configuring hazelcast.properties

**Configuring broker.xml**

The *broker.xml* file located in the `<MB_HOME>/repository/conf` directory contains the configurations relating to optimization, messaging transports, persistent store information, performance tuning and cluster coordination.

This file is compliant with the **Cipher Tool**.

The following is a tree of the XML elements in the file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--
~ Copyright (c) 2015, WSO2 Inc. (http://www.wso2.org) All Rights Reserved.
~ WSO2 Inc. licenses this file to you under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License.
~ You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0
~ Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the
```
This is the root configuration file of WSO2 Message Broker (MB). Links to configurations of associated libraries are also specified here.

[Note for developers] - If you intend to rename or modify a property name, remember to update relevant, org.wso2.andes.configuration.enums.AndesConfiguration, enum value using the Xpath expression of the property.

This file is cipher-tool compliant. Refer PRODUCT_HOME/repository/conf/security/cipher-text.properties for examples.-->

<broker>

  <coordination>
    <!-- You can override the cluster node identifier of this MB node using the nodeID. 
    If it is left as "default", the default node ID will be generated for it. (Using IP + UUID).
    The node ID of each member should ALWAYS be unique.-->
    <nodeID>default</nodeID>
    <!-- Thrift is used to maintain and sync slot (message groups) ranges between MB nodes. -->
    <thriftServerHost>localhost</thriftServerHost>
    <thriftServerPort>7611</thriftServerPort>
    <!-- Thrift server reconnect timeout. Value specified in SECONDS-->
    <thriftSOTimeout>0</thriftSOTimeout>
    <!-- Hazelcast reliable topics are used to share all notifications across the MB cluster (e.g. subscription changes), And this property defines the time-to-live for a notification since its creation. (in Seconds) -->
    <clusterNotificationTimeout>10</clusterNotificationTimeout>
    <!-- Configurations related RDBMS based coordination algorithm -->
    <rdbmsBasedCoordination enabled="true">
      <!-- Heartbeat interval used in the RDBMS base coordination algorithm in milliseconds -->
      <heartbeatInterval>5000</heartbeatInterval>
      <!-- Time to wait before informing others about coordinator change in milliseconds. This value should be larger than a database read time including network latency and should be less than heartbeatInterval -->
      <coordinatorEntryCreationWaitTime>3000</coordinatorEntryCreationWaitTime>
      <!-- Time interval used to poll database for membership related events in milliseconds. -->
      <eventPollingInterval>4000</eventPollingInterval>
      <!-- Enabling this will make the cluster notifications such as Queue changes(additions and deletions), Subscription changes, etc. sent within the cluster be synchronized using RDBMS. If set to false, Hazelcast
<rdbmsBasedClusterEventSynchronization enabled="true">
<!--Specifies the interval at which, the cluster events will be read from the database. Needs to be declared in milliseconds. Setting this to a very low value could downgrade the performance where as setting this to a large value could increase the time taken for a cluster event to be synchronized in all the nodes in a cluster.-->
<eventSyncInterval>1000</eventSyncInterval>
</rdbmsBasedClusterEventSynchronization>
</coordination>
<!-- You can enable/disable specific messaging transports in this section. By default all transports are enabled. This section also allows you to customize the messaging flows used within WSO2 MB. NOT performance related, but logic related. -->
<transports>
  <amqp enabled="true">
    <bindAddress>0.0.0.0</bindAddress>
    <defaultConnection enabled="true" port="5672" />
    <sslConnection enabled="true" port="8672">
      <keyStore>
        <location>repository/resources/security/wso2carbon.jks</location>
        <password>wso2carbon</password>
        <certType>SunX509</certType>
      </keyStore>
      <trustStore>
        <location>repository/resources/security/client-truststore.jks</location>
        <password>wso2carbon</password>
        <certType>SunX509</certType>
      </trustStore>
      <maximumRedeliveryAttempts>10</maximumRedeliveryAttempts>
    </sslConnection>
    <allowSharedTopicSubscriptions>false</allowSharedTopicSubscriptions>
    <allowStrictNameValidation>true</allowStrictNameValidation>
  </amqp>
</transports>
<location>repository/resources/security/wso2carbon.jks</location>
<password>wso2carbon</password>
<certType>SunX509</certType>
</keyStore>
</trustStore>
<location>repository/resources/security/client-truststore.jks</location>
<password>wso2carbon</password>
<certType>SunX509</certType>
</trustStore>
<sslConnection>
  <maximumRedeliveryAttempts>10</maximumRedeliveryAttempts>
</sslConnection>
<allowSharedTopicSubscriptions>false</allowSharedTopicSubscriptions>
<allowStrictNameValidation>true</allowStrictNameValidation>
<!-- Refer repository/conf/advanced/qpid-config.xml for further AMQP-specific configurations.-->
</amqp>
<mqtt enabled="true">
  <bindAddress>0.0.0.0</bindAddress>
  <defaultConnection enabled="true" port="1883" />
  <sslConnection enabled="true" port="8883">
    <keyStore>
      <location>repository/resources/security/wso2carbon.jks</location>
      <password>wso2carbon</password>
      <certType>SunX509</certType>
    </keyStore>
    <trustStore>
      <location>repository/resources/security/client-truststore.jks</location>
      <password>wso2carbon</password>
      <certType>SunX509</certType>
    </trustStore>
  </sslConnection>
  <allowSharedTopicSubscriptions>false</allowSharedTopicSubscriptions>
  <allowStrictNameValidation>true</allowStrictNameValidation>
</mqtt>
<location>repository/resources/security/wso2carbon.jks</location>
<password>wso2carbon</password>
<certType>SunX509</certType>
</keyStore>
</trustStore>
<location>repository/resources/security/client-truststore.jks</location>
  <password>wso2carbon</password>
  <certType>SunX509</certType>
</trustStore>
</sslConnection>

<!-- All receiving events/messages will be in this ring buffer. -->
Ring buffer size
of MQTT inbound event disruptor. Default is set to 32768 (1024 * 32)
Having a large ring buffer will have an increase memory usage
and will improve performance
and vise versa -->
<inboundBufferSize>32768</inboundBufferSize>

<!-- Messages delivered to clients will be placed in this ring buffer. -->
Ring buffer size of MQTT delivery event disruptor. Default is set to 32768 (1024 * 32)
Having a large ring buffer will have an increase memory usage
and will improve performance
and vise versa -->
<deliveryBufferSize>32768</deliveryBufferSize>

<security>
  <!-- Instructs the MQTT server whether clients should always send credentials when establishing a connection. -->
  Possible values:
  OPTIONAL: This is the default value. MQTT clients may or may not send credentials server will validates it.
  If client doesn't send credentials then server will not authenticate, but allows client to establish the connection.
  This behavior adheres to MQTT 3.1 specification.
  REQUIRED: Clients should always provide credentials when connecting.
  If client doesn't send credentials or they are invalid server rejects the connection.
  -->
  <authentication>OPTIONAL</authentication>

<!-- Class name of the authenticator to use. class should inherit from org.dna.mqtt.moquette.server.IAuthenticator -->
Note: default implementation authenticates against
carbon user store

  based on supplied username/password

  <!--
  Authenticator
  class="org.wso2.carbon.andes.authentication.andes.CarbonBasedMQTTAuthenticator"/
  -->
  <authenticator
  class="org.wso2.carbon.andes.authentication.andes.OAuth2BasedMQTTAuthenticator">
    <property
      name="hostURL">https://localhost:9443/services/OAuth2TokenValidationService</property>
    <property name="username">admin</property>
    <property name="password">admin</property>
    <property
      name="maxConnectionsPerHost">10</property>
    <property
      name="maxTotalConnections">150</property>
  </authenticator>

  <!--
  Instructs the MQTT server whether clients should be authorized before either publishing or subscribing
  Possible values:
  NOT_REQUIRED: This is the default value. MQTT clients will skip the authorization check
  REQUIRED: Clients will authorized before publishing. this will execute the class given in authorizer
  Note: authentication should be REQUIRED for authorization to be REQUIRED.
  -->
  <authorization>NOT_REQUIRED</authorization>

  <!--
  Class name of the authorizer to use. class should inherit from
  org.dna.mqtt.moquette.server.IAutherizer
  Note: default implementation authorizes against carbon permission with the topic.
  -->
  <authorizer
    class="org.wso2.carbon.andes.authorization.andes.CarbonPermissionBasedMQTTAuthorizer">
    <property
      name="connectionPermission">/permission/admin/mqtt/connect</property>
  </authorizer>

  <!--
  Depending on the database type selected in master-datasources.xml, you must enable the relevant Data access classes here. Currently WSO2 MB Supports RDBMS (any RDBMS store).
  These stores are accessed for two purposes.
  -->
  </security>
</mqtt>
</transports>
1. For message persistence ("messageStore")
2. To persist and access other information relevant to messaging protocols ("contextStore").

<!-- By default WSO2 MB runs with H2 persistent store. If you plan to use a different store, point to the relevant dataSource or uncomment the database appropriately.

RDBMS

If you are running an RDBMS you can use the existing RDBMS implementation of stores by pointing to the correct data source by updating the property "dataSource". Data source entry should be present in
<MB_HOME>/repository/conf/datasources/master-datasources.xml.

<!-- RDBMS MB Store Configuration -->

<persistence>
  <!-- RDBMS MB Store Configuration -->
  <messageStore
  class="org.wso2.andes.store.rdbms.RDBMSMessageStoreImpl">
    <property name="dataSource">WSO2MBStoreDB</property>
    <property name="storeUnavailableSQLStateClasses">08</property>
    <property name="integrityViolationSQLStateClasses">23,27,44</property>
    <property name="dataErrorSQLStateClasses">21,22</property>
    <property name="transactionRollbackSQLStateClasses">40</property>
  </messageStore>
  <contextStore
  class="org.wso2.andes.store.rdbms.RDBMSAndesContextStoreImpl">
    <property name="dataSource">WSO2MBStoreDB</property>
    <property name="storeUnavailableSQLStateClasses">08</property>
    <property name="integrityViolationSQLStateClasses">23,27,44</property>
    <property name="dataErrorSQLStateClasses">21,22</property>
    <property name="transactionRollbackSQLStateClasses">40</property>
  </contextStore>
  <cache>
    <!-- Size of the messages cache in MBs. Setting '0' will disable the cache. -->
    <size>256</size>
    <!-- Expected concurrency for the cache (4 is guava default) -->
    <concurrencyLevel>4</concurrencyLevel>
    <!-- Number of seconds cache will keep messages after they are added (unless they are consumed and deleted). -->
    <expirySeconds>2</expirySeconds>
    <!-- Reference type used to hold messages in memory. weak - Using java weak references ( - results higher cache misses) strong - ordinary references ( - higher cache hits, but -->
  </cache>
</persistence>
not good if server is going to run with limited heap size + under severe load).

<!--
<valueReferenceType>strong</valueReferenceType>
<![--Prints cache statistics in 2 minute intervals in carbon log (and console)-->

<printStats>false</printStats>
</cache>

<!-- This class decides how unique IDs are generated for the MB node. This id generator is expected to be thread safe and a implementation of interface org.wso2.andes.server.cluster.coordination.MessageIdGenerator
NOTE: This is NOT used in MB to generate message IDs. -->

<idGenerator>org.wso2.andes.server.cluster.coordination.TimeStampBasedMessageIdGenerator</idGenerator>

<![-- This is the Task interval (in SECONDS) to check whether communication is healthy between message store (/Database) and this server instance. -->

<storeHealthCheckInterval>10</storeHealthCheckInterval>

<![--Publisher transaction related configurations.-->

<transaction>

<![--Maximum batch size (Messages) in kilobytes for a transaction. Exceeding this limit will result in a failure in the subsequent commit (or prepare) request. Default is set to 1MB. Limit is calculated considering the payload of messages.-->

<maxBatchSizeInKB>10240</maxBatchSizeInKB>

<![-- Maximum number of parallel dtx enabled channel count. Distributed transaction requests exceeding this limit will fail.-->

<maxParallelDtxChannels>20</maxParallelDtxChannels>

</transaction>

<![-- This section allows you to tweak memory and processor allocations used by WSO2 MB. Broken down by critical processes so you have a clear view of which parameters to change in different scenarios. -->

<performanceTuning>

<slots>

<![--Rough estimate for size of a slot. What is meant by size is the number of messages contained within bounties of a slot. -->

>windowSize>1000</windowSize>

<![-- If message publishers are slow, time taken to fill the slot (up to <windowSize>) will be longer. This will add an latency to messages. Therefore broker will mark the slot as


ready to deliver before even the slot is entirely filled after specified time.

NOTE: specified in milliseconds.

<!-- Time interval which broker check for slots that can be marked as 'ready to deliver'
(- slots which have a aged more than 'messageAccumulationTimeout')

NOTE: specified in milliseconds.

<!-- Number of MessageDeliveryWorker threads that should be started-->
<deliveryThreadCount>5</deliveryThreadCount>

<!-- Number of parallel threads to execute slot deletion task.
Increasing this value will remove slots whose messages are read/delivered to consumers/acknowledged faster reducing heap memory used by server.-->
<deleteThreadCount>5</deleteThreadCount>

<!-- Max number of pending message count to delete per Slot Deleting Task. This config is used to raise a WARN when pending scheduled number of slots exceeds this limit (indicate of an issue that can lead to message accumulation on server.-->
<SlotDeleteQueueDepthWarningThreshold>1000</SlotDeleteQueueDepthWarningThreshold>

<!-- Maximum number of thrift client connections that should be created in the thrift connection pool -->
<thriftClientPoolSize>10</thriftClientPoolSize>

<!-- Maximum number of undelivered messages that can have in memory. Increasing this value increase the possibility of out of memory scenario but performance will be improved -->
<maxNumberOfReadButUndeliveredMessages>1000</maxNumberOfReadButUndeliveredMessages>

<!-- This is the ring buffer size of the delivery disruptor. This value should be a power of 2 (E.g. 1024, 2048, 4096). Use a small ring size if you want to reduce the memory usage.-->
<ringBufferSize>4096</ringBufferSize>

<!-- Number of parallel readers used to used to read content from message store. Increasing this value will speed-up the message sending mechanism. But the load on the data store will increase. -->
<parallelContentReaders>5</parallelContentReaders>
<!-- Number of parallel decompression handlers used to decompress messages before send to subscribers. Increasing this value will speed-up the message decompressing mechanism. But the system load will increase. -->

<parallelDecompressionHandlers>5</parallelDecompressionHandlers>
<!-- Number of parallel delivery handlers used to send messages to subscribers. Increasing this value will speed-up the message sending mechanism. But the system load will increase. -->

<parallelDeliveryHandlers>5</parallelDeliveryHandlers>
<!-- The size of the batch represents, at a given time the number of messages that could be retrieved from the database. -->

<contentReadBatchSize>65000</contentReadBatchSize>
<contentCache>
<!-- Specify the maximum number of entries the cache may contain. -->

<maximumSize>100</maximumSize>
<!-- Specify the time in seconds that each entry should be automatically removed from the cache after the entry's creation. -->

<expiryTime>120</expiryTime>
</contentCache>
<!--When delivering topic messages to multiple topic subscribers one of following stratigies can be choosen. -->

1. DISCARDNONE - Broker do not loose any message to any subscriber. When there are slow subscribers this can cause broker go Out of Memory.

2. SLOWEST_SUB_RATE - Broker deliver to the speed of the slowest subscribers. This can cause fast to starve. But eliminate Out of Memory issue.

3. DISCARD_ALLOWED - Broker will try best to deliver. To eliminate Out but not acked message count to <maxUnackedMessages>. If it is breached, and <deliveryTimeout> is also breached message can either be lost or actually sent but ack is not honoured.

--> <topicMessageDeliveryStrategy>
<strategyName>DISCARD_NONE</strategyName>

<!-- If you choose DISCARD_ALLOWED topic message delivery strategy, broker keep messages in memory until ack is done until this timeout.
If an ack is not received under this timeout, ack will be simulated internally and real acknowledgement is discarded.
deliveryTimeout is in seconds -->
<deliveryTimeout>60</deliveryTimeout>
</topicMessageDeliveryStrategy>
</delivery>
<ackHandling>
 <!--Number of message acknowledgement handlers to process acknowledgements concurrently. These acknowledgement handlers will batch and process acknowledgements. -->
<ackHandlerCount>1</ackHandlerCount>
<!--Maximum batch size of the acknowledgement handler. Andes process acknowledgements in batches using Disruptor Increasing the batch size reduces the number of calls made to database by MB. Depending on the database optimal batch size this value should be set.
Batches will be of the maximum batch size mostly in high throughput scenarios.
Underlying implementation use Disruptor for batching hence will batch message at a lesser value than this in low throughput scenarios -->
<ackHandlerBatchSize>100</ackHandlerBatchSize>
<!-- Message delivery from server to the client will be paused temporarily if number of delivered but unacknowledged message count reaches this size. Should be set considering message consume rate. This is to avoid overwhelming slow subscribers. -->
<maxUnackedMessages>1000</maxUnackedMessages>
</ackHandling>
<contentHandling>
 <!-- Within Andes there are content chunk handlers which convert incoming large content chunks into max content chunk size allowed by Andes core. These handlers run in parallel converting large content chunks to smaller chunks. If the protocol specific content chunk size is different from the max chunk size allowed by Andes core and there are significant number of large messages published, then having multiple handlers will increase performance. -->
<contentChunkHandlerCount>3</contentChunkHandlerCount>
<!-- Andes core will store message content chunks according to this chunk size. Different database will have limits and performance gains by tuning this
parameter. For instance in MySQL the maximum table column size for content is less than 65534, which is the default chunk size of AMQP. By changing this parameter to a lesser value we can store large content chunks converted to smaller content chunks within the DB with this parameter. -->

```xml
<maxContentChunkSize>65500</maxContentChunkSize>
```

<!-- This is the configuration to allow compression of message contents, before store messages into the database.-->

```xml
<allowCompression>false</allowCompression>
```

<!-- This is the configuration to change the value of the content compression threshold (in bytes). Message contents less than this value will not compress, even compression is enabled. The recommended message size of the smallest message before compression is 13 bytes. Compress messages smaller than 13 bytes will expand the message size by 0.4% -->

```xml
<contentCompressionThreshold>1000</contentCompressionThreshold>
```

</contentHandling>

</inboundEvents>

<!--Number of parallel writers used to write content to message store. Increasing this value will speed-up the message receiving mechanism. But the load on the data store will increase.-->.

```xml
<parallelMessageWriters>1</parallelMessageWriters>
```

<!--Size of the Disruptor ring buffer for inbound event handling. For publishing at higher rates increasing the buffer size may give some advantage on keeping messages in memory and write. Note: Buffer size should be a value of power of two -->

```xml
<bufferSize>65536</bufferSize>
```

<!--Maximum batch size of the batch write operation for inbound messages. MB internals use Disruptor to batch events. Hence this batch size is set to avoid database requests with high load (with big batch sizes) to write messages. This need to be configured in high throughput messaging scenarios to regulate the hit on database from MB -->

```xml
<messageWriterBatchSize>70</messageWriterBatchSize>
```

<!--Timeout for waiting for a queue purge event to end to get the purged count. Doesn't affect actual purging. If purge takes time, increasing the value will improve the possibility of retrieving the correct purged count. Having a lower value doesn't stop purge event. Getting the purged count is affected by this -->

```xml
<purgedCountTimeout>180</purgedCountTimeout>
```
<!-- Number of parallel writers used to write content to message store for transaction based publishing. Increasing this value will speed up commit duration for a transaction. But the load on the data store will increase.-->
<transactionMessageWriters>1</transactionMessageWriters>
</inboundEvents>

<!-- Message expiration can be set for each messages which are published to Wso2 MB. After the expiration time, the messages will not be delivered to the consumers. Eventually they got deleted inside the MB.-->
<messageExpiration>
  <!-- When messages delivered, in the delivery path messages were checked whether they are already expired. If expired at that time add that message to a queue for a future batch delete. This interval decides on the time gap between the batch deletes. Time interval specified in seconds.-->
  <preDeliveryExpiryDeletionInterval>10</preDeliveryExpiryDeletionInterval>
  <!-- Periodically check the database for new expired messages which were not assigned to any slot delivery worker so far and delete them. This interval decides on the time gap between the periodic message deletion. Time interval specified in seconds.-->
  <periodicMessageDeletionInterval>900</periodicMessageDeletionInterval>
  <!-- When checking the database for expired messages, the messages which were handled by the slot delivery worker should no be touched since that mess up the slot delivery worker functionality. Those messages anyways get caught at the message delivery path. So there is a need to have a safe buffer of slots which can be allocated to a slot delivery worker in the near future. The specified number of slots from the last assigned should not be touched by the periodic deletion task.-->
  <safetySlotCount>3</safetySlotCount>
</messageExpiration>
</performanceTuning>

<!-- This section is about how you want to view messaging statistics from the admin console and how you plan to interact with it. -->
<managementConsole>
  <!-- Maximum number of messages to be fetched per page using Andes message browser when browsing queues/dlc -->
  <messageBrowsePageSize>100</messageBrowsePageSize>
  <!-- This property defines the maximum message content length that can be displayed at the -->
</managementConsole>
management console when browsing queues. If the message length exceeds the value, a truncated content will be displayed with a statement "message content too large to display." at the end. default value is 100000 (can roughly display a 100KB message.)

* NOTE : Increasing this value could cause delays when loading the message content page.-->

<!--Enable users to reroute all messages from a specific destination(queue or durable topic) to a specific queue.-->

<allowReRouteAllInDLC>true</allowReRouteAllInDLC>

</managementConsole>

<!-- Memory and resource exhaustion is something we should prevent and recover from.
This section allows you to specify the threshold at which to reduce/stop frequently intensive operations within MB temporarily. -->

<!-- highLimit - flow control is enabled when message chunk pending to be handled by inbound disruptor reaches above this limit
lowLimit - flow control is disabled (if enabled) when message chunk pending to be handled by inbound disruptor reaches below this limit -->

<flowControl>

<!-- This is the global buffer limits which enable/disable the flow control globally -->

<global>

<lowLimit>800</lowLimit>

<highLimit>8000</highLimit>

</global>

<!-- This is the channel specific buffer limits which enable/disable the flow control locally. -->

<bufferBased>

<lowLimit>100</lowLimit>

<highLimit>1000</highLimit>

</bufferBased>

</flowControl>

<!-- Message broker keeps track of all messages it has received as groups. These groups are termed 'Slots' (To know more information about Slots and message broker install please refer to online wiki). Size of a slot is loosely determined by the configuration <windowSize> (and the number of parallel publishers for specific topic/queue). Message broker cluster (or in single node) keeps track of slots which constitutes for a large part of operating state
before the cluster went down. When first message broker node of the cluster starts up, it will read the database to recreate the internal state to previous state.

```xml
<recovery>
  <!--
  There could be multiple storage queues worked before entire cluster (or single node) went down.
  We need to recover all remaining messages of each storage queue when first node startup and we can read remaining message concurrently of each storage queue. Default value to set here to 5. You can increase this value based on number of storage queues exist. Please use optimal value based on number of storage queues to speed up warm startup.
  -->
  <concurrentStorageQueueReads>5</concurrentStorageQueueReads>
  <!-- Virtual host sync interval seconds in for the Virtual host syncing Task which will sync the Virtual host details across the cluster -->
  <vHostSyncTaskInterval>900</vHostSyncTaskInterval>

  <!-- Enables network partition detection ( and surrounding functionality, such as disconnecting subscriptions, enabling error based flow control if the minimal node count becomes less than configured value. -->
  <networkPartitionsDetection enabled = "false">
    <!-- The minimum node count the cluster should maintain for this node to operate. if cluster size becomes less that configured value This node will not accept any incoming traffic ( and disconnect subscriptions) etc. -->
    <minimumClusterSize>1</minimumClusterSize>
  </networkPartitionsDetection>
</recovery>

<!-- Specifies the deployment mode for the broker node (and cluster). Possible values {default, standalone}. default - Broker node will decide to run HA (master/slave) or fully distributed mode. Decision is taken based on the node has a clustering mechanism enabled or not. If the node is not configured to join a cluster it will run in HA mode (refer to axis2.xml for more information). If the node can join a cluster it will start in fully clustered mode.
```
standalone - This is the simplest mode a broker can be started. Node will assume datastore is not shared with another node. Therefore it will not try to coordinate with other nodes (possibly non-existent) to provide HA or clustering.

--> 
<deployment>
Click an element below for more information on that element.

- `<broker>`
- `<coordination>`
- `<transports>`
- `<persistence>`
- `<transaction>`
- `<performanceTuning>`
- `<managementConsole>`
- `<flowControl>`
- `<recovery>`

**<broker>**

This is the top level element of the `broker.xml` file. It contains all the other elements needed to configure WSO2 MB.

**<coordination>**

This element contains configurations relating to the Apache Thrift communications. These configurations are important when MB is run in a clustered mode.

### Configurable sub elements

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;nodeID&gt;</code></td>
<td>In a clustered deployment, a ID is assigned to each MB node via the cluster node identifier. This element can be used to override the cluster node identifier for this MB node. If the value for this element is left as <code>default</code>, the default node ID is generated using the IP (Internet Protocol) and UUID.</td>
<td>String</td>
<td><code>default</code></td>
<td>N/A</td>
<td>Mandatory</td>
<td>The node of ea mem in a clust should be uniqui.</td>
</tr>
</tbody>
</table>
### `<thriftServerHost>`

Apache Thrift is embedded in WSO2 MB and is used for communications related to message delivery. In each MB node of a cluster, the value for this property (`thrift server host`) should be the IP address of that respective node.

**Note:** One of the nodes in an MB cluster will be used as the coordinating MB node at server startup. The Thrift server configured for the coordinating MB node will serve as the central Thrift server, which will be used by all nodes in the cluster to consume thrift services. The coordinating MB node may switch to another node in the cluster in case the current coordinator becomes unavailable.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;thriftServerHost&gt;</code></td>
<td>String</td>
<td>localhost</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### `<thriftServerPort>`

This should point to the port of the Apache Thrift server in WSO2 MB.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Default</th>
<th>Mandatory</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;thriftServerPort&gt;</code></td>
<td>String</td>
<td>7611</td>
<td>N/A</td>
</tr>
</tbody>
</table>
## <thriftServerReconnectTimeout>

The number of seconds taken by the thrift server to reconnect after a time out.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>5</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

## <thriftSOTimeout>

This is used to handle half-open TCP connections between the broker nodes in a cluster. In such situations, the socket may need to have a timeout value to invalidate the connection. (in milliseconds). A timeout of zero is interpreted as an infinite timeout.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
<td>N/A</td>
<td>Optional</td>
<td></td>
</tr>
</tbody>
</table>

## <transports>

This element contains all the messaging transports used by WSO2 MB. These messaging transports can be enabled/disabled as required. All the transports are enabled by default. This element also contains configuration parameters that enable you to customise the logical aspects of the message flows used within WSO2 MB.

### Configurable sub elements

The following configurable sub elements are common to both AMQP and MQTT transports.

<table>
<thead>
<tr>
<th>Element/Attribute Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optinal</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bindAddress&gt;</td>
<td>The IP address to which all transport listeners are bound (e.g. amqp and mqtt listeners).</td>
<td>Integer</td>
<td>The IP of the resident node</td>
<td></td>
<td>Mandatory</td>
<td>This is the address exposed from the server and not the host name inferred from carbon.xml.</td>
</tr>
<tr>
<td><code>&lt;defaultConnection&gt;</code></td>
<td>This element is used to specify default values with regard to the connection of the transport. It contains the <code>enabled</code> and <code>port</code> attributes.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>enabled</code></td>
<td>If the value is <code>true</code>, the relevant transport is enabled for WSO2 MB.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Boolean</td>
<td>true</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>port</code></td>
<td>The default listening port for messages/commands from the MB server via the relevant transport.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integer</td>
<td>5672</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;sslConnection&gt;</code></td>
<td>This element is used to specify default values with regard to the SSL (Security Sockets Layer) connection of the transport. It contains the <code>enabled</code> and <code>port</code> attributes.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>enabled</code></td>
<td>If the value is <code>true</code>, SSL is enabled for WSO2 MB.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Boolean</td>
<td>true</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>The default listening SSL port for messages/commands from the MB server via the relevant transport. This port is used only if the enabled attribute of the sslConnection element is set to true.</td>
<td>Integer</td>
<td>8672</td>
<td>Mandatory if SSL is enabled.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**<keyStore>**

This element contains parameters relating to the keystore directory which include location and password. See the section on working with security for details on keystores.

| location | The path to the keystore directory. See the section on working with security for details on keystores. | String | repository/resources/security/wso2carbon.jks | repository/resources/security/wso2carbon.jks | Mandatory |

| password | The password for the key store. | String | wso2carbon | wso2carbon | Mandatory |

**<trustStore>**

This element contains parameters relating to the truststore which include location and password.
<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;location&gt;</code></td>
<td>The location of the truststore.</td>
<td>String</td>
<td>repository/resources/security/client-truststore.jks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;password&gt;</code></td>
<td>The password for the truststore</td>
<td>String</td>
<td>wso2carbon</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following configurable sub elements are specific to the **AMQP transport**.

The maximum number of times the WSO2 MB should attempt to redeliver a message which has not reached a subscriber. For example, when the value for this is 10, ten more attempts will be made to deliver the message. The default value can be changed depending on your reliability requirements. See the topic on maximum delivery attempts for more information on this configuration.
<allowSharedTopicSubscriptions>
If the value for this parameter is true, it is possible to have multiple durable subscriptions with the same subscription ID. As a result, when a topic is used, only one of the subscribers will get the given message based in a Round Robin manner.

<allowStrictNameValidation>
If this value is set to false, it is possible to use the colon (":"), symbol in topic names. Note that this only applies when amqp is used.

The following configurable sub elements are specific to the MQTT Transport.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;inboundBufferSize&gt;</td>
<td>The ring buffer size of the MQTT inbound event disruptor. Increasing this value would improve performance, but it would also increase the memory consumption.</td>
<td>Integer</td>
<td>32768</td>
</tr>
<tr>
<td>&lt;deliveryBufferSize&gt;</td>
<td>The ring buffer size of the MQTT delivery event disruptor. As for &lt;inboundBufferSize&gt;, increasing this value would improve performance, but it would also increase the memory consumption.</td>
<td>Integer</td>
<td>32768</td>
</tr>
<tr>
<td>&lt;security&gt;</td>
<td>This element contains parameters relating to authentication for the MQTT transport which include &lt;authentication&gt; and &lt;authenticator&gt;.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<authentication>
This parameter is used to specify whether clients should always send credentials when establishing a connection. Possible values are as follows.

**Optional**: If this is selected, credentials of the client are validated if they are sent and the connection is established if the credentials are valid. However, if no credentials are sent, the connection is established without performing a validation test. This behaviour adheres to MQTT 3.1 specification.

**Required**: If this is selected, the server always performs an authentication test before establishing a connection. The connection is rejected if no credentials are provided by the client or if the credentials sent are not valid.

<size>

<authenticator>
The fully qualified class name of the authenticator. Class should be inherited from `org.dna.mqtt.moquette.server.IAuthenticator`.

Default implementation authenticates against the carbon user store based on the supplied username/password.

</authenticator>

<persistence>
This element contains configuration parameters relating to data persistence.

**Configurable sub elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;messageStore&gt;</td>
<td>The class that is used to access an external RDBMS database to operate on messages.</td>
<td>String</td>
<td>org.wso2.andes.store.jdbc.JDBCMessageStoreImpl</td>
</tr>
<tr>
<td>&lt;contextStore&gt;</td>
<td>The class that is used to access an external RDBMS database to operate on server context. e.g. subscriptions.</td>
<td>String</td>
<td>org.wso2.andes.store.jdbc.JDBCAndesContextStoreImpl</td>
</tr>
</tbody>
</table>
<idGenerator>
The ID generation class that is used to maintain unique IDs for each message that arrives at the server.

org.wso2.andes.server.cluster.coordination.TimeStampBasedMessageIdGenerator

<storeHealthCheckInterval>
The length of the time interval between two checks made by the system to ensure that communication between the message store and the MB server instance is healthy.

Integer 10

<transaction>
This element contains configurations relating to publisher transactions.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
<th>Fixed</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;dbConnectionPoolSize&gt;</td>
<td>The number of connections reserved at a given time for transactional database tasks. A transaction reserves a database connection until the transaction is committed, rolled back or closed.</td>
<td>Integer</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;maxBatchSizeInKB&gt;</td>
<td>The maximum number of kilobytes included in a transaction. Exceeding this number will cause failure in commit requests.</td>
<td>Integer</td>
<td>10240</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<performanceTuning>
This element contains configurations relating to the memory and processor allocations of the WSO2 MB server.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default</th>
<th>Fixed</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
</table>
Each subscriber is assigned a **slot** by WSO2 MB. The subscriber reads messages from this assigned slot. A slot comprises of a chunk of messages in a row.

This element contains all parameters relating to slots which include **slotRetainTimeInMemory**, **WindowSize**.

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;slots&gt;</td>
<td>Each subscriber is assigned a slot by WSO2 MB. The subscriber reads messages from this assigned slot. A slot comprises of a chunk of messages in a row. This element contains all parameters relating to slots which include slotRetainTimeInMemory, windowSize.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;slotRetainTimeInMemory&gt;</td>
<td>The maximum time duration for which a slot can be retained in memory. Once the time specified in this parameter elapses, the messages in the slot are updated to the coordinator node in the cluster.</td>
<td>Integer</td>
<td>1000</td>
</tr>
<tr>
<td>&lt;WindowSize&gt;</td>
<td>This parameter is used to specify a rough estimate for the size of the slot. e.g., If the size of the slot is 1000, it can expand to 3000 when there are 3 nodes.</td>
<td>Integer</td>
<td>1000</td>
</tr>
<tr>
<td>&lt;windowCreationTimeout&gt;</td>
<td>This parameter defines the timeout for the slot window creation task. In a slow message publishing scenario, this parameter accounts for the delay in the delivery of each message. e.g., If one message is published per minute, then each message will be delivered only after they are submitted to the slot coordinator once this timeout has taken place.</td>
<td></td>
<td>3000</td>
</tr>
<tr>
<td>&lt;delivery&gt;</td>
<td>This element contains all the parameters relating to message delivery which include maxNumberOfReadButUndeliveredMessages, ringBufferSize, parallelContentReaders, and parallelDeliveryHandlers.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;maxNumberOfReadButUndeliveredMessages&gt;</td>
<td>The maximum number of undelivered messages that can be stored in the memory. Increasing the value for this parameter can cause out-of-memory exceptions, but the performance will be improved.</td>
<td>Integer</td>
<td>1000</td>
</tr>
<tr>
<td>&lt;ringBufferSize&gt;</td>
<td>The thread pool size of the queue delivery workers. This should be increased if there are a lot of unique queues in the system at a given time.</td>
<td>Integer</td>
<td>4096</td>
</tr>
<tr>
<td>&lt;parallelContentReaders&gt;</td>
<td>The number of parallel readers used to read content from the message store. Increasing this value would increase the speed of the message sending mechanism. However, it would also cause the load on the data store to increase.</td>
<td>Integer</td>
<td>5</td>
</tr>
<tr>
<td>&lt;parallelDeliveryHandlers&gt;</td>
<td>The number of parallel delivery handlers used to send messages to subscribers. Increasing this value will increase the speed of the message sending mechanism. However, it would also cause the load on the data store to increase.</td>
<td>Integer</td>
<td>5</td>
</tr>
<tr>
<td>&lt;contentReadBatchSize&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element</td>
<td>Description</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ackHandling&gt;</code></td>
<td>This element contains parameters relating to message acknowledgement which include <code>maxUnackedMessages</code>, <code>ackHandlerBatchSize</code>, and <code>ackHandlerBatchSize</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;maxUnackedMessages&gt;</code></td>
<td>The maximum number of unacknowledged messages allowed. The message delivery from the server to the client is temporarily paused when the number specified for this parameter is reached. This number should be specified considering the message consumption rate.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ackHandlerBatchSize&gt;</code></td>
<td>The maximum batch size of the acknowledgement handler. Andes processes acknowledgements in batches using Disruptor. Increasing the batch size reduces the number of calls made to database by MB. The value for this parameter should be set based on database optimization. The maximum batch size is applicable in high throughput scenarios. Messages are handled in smaller batches using the Disruptor in low throughput scenarios.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ackHandlerCount&gt;</code></td>
<td>The number of acknowledgement handlers used to process acknowledgements concurrently. Acknowledgements are batched and processed by these acknowledgement handlers.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td><code>&lt;contentHandling&gt;</code></td>
<td>This element contains parameters relating to content handling which include <code>contentChunkHandlerCount</code> and <code>maxContentChunkSize</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;contentChunkHandlerCount&gt;</code></td>
<td>The number of message chunk handlers used to process content chunks concurrently. The content chunks are batched and processed by these content chunk handlers. The value for this parameter should be set based on database optimization.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td><code>&lt;maxContentChunkSize&gt;</code></td>
<td>The maximum column size for a content chuck handler.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td><code>&lt;inboundEvents&gt;</code></td>
<td>This element contains parameters relating to the inbound events, which include <code>parallelMessageWriters</code>, <code>bufferSize</code>, and <code>messageWriterBatchSize</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;parallelMessageWriters&gt;</code></td>
<td>The number of parallel writers used to write content to the message store. Increasing this value will increase the speed of the message receiving mechanism. However, it would also cause the load on the data store to increase.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td><code>&lt;bufferSize&gt;</code></td>
<td>The size of the Disruptor ring buffer for inbound event handling. Increasing the buffer size is recommended when there is an increased rate of publishing. In such situations, the messages can be stored in the memory.</td>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Configuration</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td><code>&lt;messageWriterBatchSize&gt;</code></td>
<td>The maximum batch size of the batch write operation for inbound messages. MB internals use Disruptor to batch events. Hence this batch size is set to avoid database requests with a high load (i.e. with large batch sizes) to write messages. This should be configured in high throughput messaging scenarios to regulate the load on the database from MB.</td>
<td>Integer</td>
<td>7</td>
</tr>
<tr>
<td><code>&lt;purgedCountTimeout&gt;</code></td>
<td>The number of milliseconds to wait for a queue to complete a purge event in order to update the purge count. If the time specified here elapses before the queue completes the purge event, the purge count will not be updated to include the event. The value for this parameter can be increased to ensure that the purge count is accurate.</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td><code>&lt;transactionMessageWriters&gt;</code></td>
<td>The number of parallel writers used to write content to the message store for transaction based publishing. Increasing the value for this parameter reduces the time taken to commit a transaction, but the load on the data store would also increase.</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td><code>&lt;failover&gt;</code></td>
<td>This element contains parameters relating to failover which includes <code>vHostSyncTaskInterval</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;messageCounter&gt;</code></td>
<td>This element contains parameters relating to the message counter which includes <code>counterTaskInterval</code> and <code>countUpdateBatchSize</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;counterTaskInterval&gt;</code></td>
<td>Message counter tasks delay that occurs between the end of one execution and the start of another.</td>
<td>Integer</td>
<td>5</td>
</tr>
<tr>
<td><code>&lt;countUpdateBatchSize&gt;</code></td>
<td>The message count is updated in batches. Once the count exceeds the batch size, the message count update is moved to the Message Count Update task.</td>
<td>Integer</td>
<td>10</td>
</tr>
<tr>
<td><code>&lt;messageDeletion&gt;</code></td>
<td>This element contains parameters relating to message deletion which include <code>contentRemovalTaskInterval</code>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;contentRemovalTaskInterval&gt;</code></td>
<td>The task interval for the content removal task which will remove the actual message content from the store in the background. If the message rate is very high, this can be set to a lower value to minimise the number of delete requests per task.</td>
<td>Integer</td>
<td>3</td>
</tr>
</tbody>
</table>
### <topicMatching>

The method used for topic matching. If SIMPLE is selected, the default selector mechanism will be used. Alternatively, you can select BitMaps. BitMaps is more suitable when using topics and subscriptions in large quantity since it is faster.

### <managementConsole>

This element contains parameters relating to the management console.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;messageBrowsePageSize&gt;</td>
<td>The maximum number of messages to be displayed in a page at a given time when viewing messages in a queue or the dead letter channel.</td>
<td>Integer</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### <maximumMessageDisplayLength>

The maximum number of characters in a message that should be displayed in the Management Console when browsing queues. If the number of characters in a message exceeds the value specified for this parameter, the truncated message content will be displayed with the text message content too large to display. Increasing this value can cause delays when viewing the message content in the Management Console.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;maximumMessageDisplayLength&gt;</td>
<td></td>
<td>Integer</td>
<td>100000</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;highLimit&gt;</td>
<td>The total maximum number of message content chunks that should be stored in the buffer. When this limit is reached, flow control is enabled. Having a higher limit will increase the number of messages in memory before storing in DB. This results in a higher overall message publishing rate, but with reduced reliability.</td>
<td>Integer</td>
<td>8000</td>
<td></td>
</tr>
<tr>
<td>&lt;bufferBased&gt;</td>
<td>This element contains parameters relating to the channel specific buffer limits which enable and disable the flow control locally which include lowLimit and highLimit.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;lowLimit&gt;</td>
<td>The number of message content chunks that would disable flow control if they are enabled, that should be stored in the buffer. Having a small difference between the lower and higher limits will lead to frequent enabling and disabling of flow control, causing the message publishing rate to be reduced.</td>
<td>Integer</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>&lt;highLimit&gt;</td>
<td>The maximum number of message content chunks that should be stored in buffer per publisher. When this limit is reached, flow control is enabled for the relevant publisher. Having a higher limit will increase the number of messages in memory before storing in the DB. This would lead to higher message publishing rates, but with reduced reliability.</td>
<td>Integer</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>&lt;memoryBased&gt;</td>
<td>This element contains memory based parameters which include memoryCheckInterval, globalMemoryThresholdRatio, and globalMemoryRecoveryThresholdRatio.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;memoryCheckInterval&gt;</td>
<td>The time interval at which the server should check for memory consumption and apply flow control to recover.</td>
<td>Integer</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>&lt;globalMemoryThresholdRatio&gt;</td>
<td>The maximum ratio of memory allowed to be used by the server.</td>
<td>Integer</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>&lt;globalMemoryRecoveryThresholdRatio&gt;</td>
<td>The ratio at which the server should apply flow control to recover.</td>
<td>Integer</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>&lt;connectionBased&gt;</td>
<td>This element contains connection based parameters which include perConnectionMessageThreshold.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
<td>Fixed Values</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-------</td>
<td>---------------</td>
<td>--------------</td>
</tr>
<tr>
<td><code>&lt;perConnectionMessageThreshold&gt;</code></td>
<td>This allows you to apply flow control based on the message count on a given connection.</td>
<td>Integer</td>
<td>1000</td>
<td></td>
</tr>
</tbody>
</table>

**<recovery>**

This element contains configurations relating to cluster recovery. Cluster recovery involves retrieving messages from a storage queues after the cluster or a single node has been inactive.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;concurrentStorageQueueReads&gt;</code></td>
<td>The number of storage queue reads carried out concurrently at a given time. There can be multiple storage queues operating before a cluster or a single node becomes inactive. The value for this parameter should be entered depending on the number of storage queues.</td>
<td>Integer</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;vHostSyncTaskInterval&gt;</code></td>
<td>The time interval after which the virtual host syncing task can sync host details across the cluster.</td>
<td>integer</td>
<td>900</td>
<td></td>
<td></td>
<td>This value is specified in seconds.</td>
</tr>
</tbody>
</table>

**Configuring qpid-config.xml**

The `<MB_HOME>/repository/conf/advanced/qpid-config.xml` file is used to configure the QPID extension of WSO2 MB.

Note that this file is taken from the QPID broker and used only within the QPID extension in WSO2 MB. Some parameters in this file are overruled by the settings in the `broker.xml` file.
Following is a tree of the XML elements in the file:

```xml
<broker>
  <prefix>
  </prefix>
  <work>
  </work>
  <conf>
    <plugin-directory>
    </plugin-directory>
    <cache-directory>
    </cache-directory>
    <connector>
      <qpidnio>
      </qpidnio>
      <protection>
      </protection>
      <enabled>
        <readBufferLimitSize>
        </readBufferLimitSize>
        <writeBufferLimitSize>
        </writeBufferLimitSize>
      </enabled>
      <transport>
        <socketReceiveBuffer>
        </socketReceiveBuffer>
        <socketSendBuffer>
        </socketSendBuffer>
      </transport>
      <management>
      </management>
      <enabled>
      </enabled>
      <jmxport>
      </jmxport>
      <ssl>
        <enabled>
          <keyStorePath>
          </keyStorePath>
          <keyStorePassword>
        </enabled>
        <ssl>
        </ssl>
      </ssl>
      <advanced>
        <filterchain>
          <enablePooledAllocator>false</enablePooledAllocator>
          <enableDirectBuffers>false</enableDirectBuffers>
          <framesize>65535</framesize>
          <compressBufferOnQueue>false</compressBufferOnQueue>
          <enableJMSXUserID>false</enableJMSXUserID>
          <locale>en_US</locale>
        </advanced>
        <security>
          <pd-auth-manager>
            <principal-database>
              <class>
                <attributes>
                  <attribute>
                    <name>
                    </name>
                    <value>
                    </value>
                  </attribute>
                </attributes>
              </class>
            </principal-database>
            <msg-auth>
            </msg-auth>
            <virtualhosts>
            </virtualhosts>
            <heartbeat>
              <delay>
                <timeoutFactor>
                </timeoutFactor>
              </delay>
            </heartbeat>
            <queue>
              <auto_register>
                <viewMessageCounts>false</viewMessageCounts>
              </auto_register>
              <status-updates>ON</status-updates>
            </queue>
          </pd-auth-manager>
        </security>
      </advanced>
    </conf>
  </work>
</broker>
```
Click an element below for more information:

- `<broker>`
- `<connector>`
- `<management>`
- `<advanced>`
- `<security>`
- `<virtualhosts>`
- `<heartbeat>`
- `<queue>`
- `<status-updates>`

**<broker>**

This is the top level element of the qpid-config.xml file. It contains all the other elements needed to configure WSO2 MB. The setting of the prefixes for ANDES_HOME and QPID_WORK allows environment variables to be used throughout the config.xml file and removes the need for hard coding of paths in this file.

**Configurable sub elements**

These parameters relate to the original QPID distribution and are overruled by the configurations of the broker.xml file. Therefore they will not have the effect intended when they were introduced by QPID.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;prefix&gt;</td>
<td>This points to the directory in which the QPID source is located.</td>
<td>String</td>
<td><code>${ANDES_HOME}</code></td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;work&gt;</td>
<td>This points to the directory in which the temporary data stored within QPID are located. e.g., Log files.</td>
<td>String</td>
<td><code>${QPID_WORK}</code></td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;conf&gt;</td>
<td>This points to the directory in which the QPID configuration files are located.</td>
<td>String</td>
<td><code>${prefix}</code></td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>
<plugin-directory>
This points to the directory in which any plug-ins to QPID are located.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;plugin-directory&gt;</td>
<td>This points to the directory in which any plug-ins to QPID are located.</td>
<td>String</td>
<td>${ANDES_HOME}/lib/plugins</td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<cache-directory>
This points to the cache directory of QPID.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cache-directory&gt;</td>
<td>This points to the cache directory of QPID.</td>
<td>String</td>
<td>${QPID_WORK}/cache</td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<connector>
This element contains parameters relating to the various aspects of the connection between the WSO2 MB and the QPID broker.

<qpidnio>
This is a sub element of the <connector> element.

Configurable sub elements

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;qpidnio&gt;</td>
<td>QPIDNIO is enabled by setting this parameter to true. When QPIDNIO is enabled, a multi-threaded MINA socket acceptor will be set up. This socket acceptor will make an attempt to boost the performance by allowing both reading from and writing to a socket simultaneously.</td>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;transport&gt;</td>
<td>This parameter determines which transport type to use to work with AMQP publishers/subscribers.</td>
<td>String</td>
<td>nio</td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;port&gt;</td>
<td>The port which corresponds with the port in which the non-secure Java Broker is run.</td>
<td>Integer</td>
<td>5672</td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>
**<socketReceiveBuffer>**
The buffer size which applies to messages received by the socket.

**Type**: Integer  
**Default Value**: 32768  
**Mandatory**: Mandatory

**<socketSendBuffer>**
The buffer size for messages sent from the socket.

**Type**: Integer  
**Default Value**: 32768  
**Mandatory**: Mandatory

**<protectio>**
This element contains parameters used to prevent the QPID broker from running out of memory because of run-away clients and unresponsive clients. The parameters in this section are overruled by the settings in the broker.xml file. Therefore they will not have the effect intended when they were introduced by QPID.

**Configurable sub elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>If the value is true, it means that the Protect I/O Configuration feature is enabled.</td>
<td>Boolean</td>
<td>false</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;readBufferLimitSize&gt;</td>
<td>The buffer limit for messages read by the QPID broker.</td>
<td>Integer</td>
<td>262144</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;writeBufferLimitSize&gt;</td>
<td>The buffer limit for messages sent by the QPID broker.</td>
<td>Integer</td>
<td>262144</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**<management>**
This configuration option is for managing the JMX console.

Since WSO2 MB uses its own management console, parameters in this section do not apply to WSO2 MB. The parameters relating to the WSO2 management console are described in the broker.xml file.

**Configurable sub elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>This element allows the user to switch the connectivity of the management console on/off. If the enabled tag is set to false you will not be able to connect a management console to this broker instance.</td>
<td>Boolean</td>
<td>true</td>
<td>true/false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;jmxport&gt;</td>
<td>The JMX Management port.</td>
<td>Int</td>
<td>8999</td>
<td>N/A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
<ssl>
SSL related configurations for the management console.

<enabled>
This is a sub element of the <ssl> element. This parameter determines whether SSL is enabled for the management console or not.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<keyStorePath>
This is a sub element of the <ssl> element. Update the path to your keystore location, or run the bin/create-example-ssl-stores (.sh|.bat) script from within the etc/ folder to generate an example store with a self-signed cert.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>${conf}/qpid.keystore</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<keyStorePassword>
The password for the keystore provided.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>password</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

<advanced>
The elements in this section are used by the QPID extension within WSO2MB. At present, we do not recommend any changes to these settings.

Configurable sub elements

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;filterchain&gt;</td>
<td>If the value is true, then the filter chain is activated.</td>
<td>Boolean</td>
<td>true</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Thus, many filters will be applied to the requests and each filter would</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>invoke the next filter in the chain.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;enablePooledAllocator&gt;</td>
<td>If this parameter is set to true, there will be a defined pool of ByteBuffers which will be used by the MINA socket acceptor to handle incoming messages.</td>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;enableDirectBuffers&gt;</td>
<td>If this parameter is set to true, the MINA socket acceptor will use direct ByteBuffers instead of heap ByteBuffers.</td>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
<td>Mandatory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>-------</td>
<td>---------------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;framesize&gt;</td>
<td>This parameter defines the buffer size when chunking the message content body as per the AMQP protocol.</td>
<td>Int</td>
<td>65535</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;compressBufferOnQueue&gt;</td>
<td>If this parameter is set to true, the ByteBuffers created by the MINA socket acceptor will be compressed for better performance.</td>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;enableJMSXUserID&gt;</td>
<td>If this parameter is set to true, it allows JMX user to connect to the QPID broker management console. However, since the QPID management console is not active within the WSO2 MB, this parameter is not used.</td>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;locale&gt;</td>
<td>This defines the language used by QPID to communicate its information/errors.</td>
<td>String</td>
<td>en_US</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

**<security>**

The security section specifies exactly one authentication manager (responsible for determining that a user's credentials are correct) and zero or more access plugins (which limit what the user may and may not do). No configuration changes are required in this section.

**Configurable sub elements**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element Name</td>
<td>Description</td>
<td>Type</td>
<td>Default Value</td>
</tr>
<tr>
<td>-----------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------</td>
<td>---------------</td>
</tr>
<tr>
<td>&lt;pd-auth-manager&gt;</td>
<td>This parameter contains the principle databases for which the authorization related parameters are set.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;principle-database&gt;</td>
<td>This element contains the principle database for which the authorisation related parameters apply.</td>
<td>String</td>
<td>N/A</td>
</tr>
<tr>
<td>&lt;class&gt;</td>
<td>This parameter contains the authentication class which applies to the principle database.</td>
<td>String</td>
<td>org.wso2.carbon.andes.authentication.andes.CarbonBasedPrincipal</td>
</tr>
</tbody>
</table>

**<virtualhosts>**

This element allows you to specify a location for the virtualhosts.xml file that you wish to use. If you are not using a subdirectory under $QPID_HOME you can provide a fully qualified path instead. For more information on the content of the virtualhosts.xml file please see Configuring qpid-virtualhosts.xml.

**<heartbeat>**

The Qpid broker sends an internal (only) heartbeat. This element is use to configure parameters relating to the frequency of this heartbeat. You need to to configure a proper delay for the heartbeat value if the connections will stay idle for a long time. For a description of all related configurations see Configure Broker and Client Heartbeating.

**Configurable sub elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;delay&gt;</td>
<td>This defines an interval between the pings sent to a subscriber to keep the connection alive.</td>
<td>Int</td>
<td>0</td>
<td>N/A</td>
<td></td>
<td>This value is given in seconds.</td>
</tr>
</tbody>
</table>
<timeoutFactor>

The time duration allowed for a subscriber to respond to a heartbeat request. If this time elapses before the response is received, the channel of communication between the server and the subscriber will end.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Float</td>
<td>2.0</td>
<td>N/A</td>
<td></td>
<td>This value is given in seconds.</td>
</tr>
</tbody>
</table>

<queue>

This parameter should NOT be changed lightly as it sets the broker up to automatically bind queues to exchanges. It could be used to prevent users from creating new queues at runtime, assuming that you have created all queues/topics etc at broker startup. However, it is recommended to leave the parameter unchanged for now.

Configurable sub elements

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;auto_register&gt;</td>
<td>This parameter determines whether to register the queue name in the server at the time it is created or not.</td>
<td>Boolean</td>
<td>true</td>
<td>true/false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;viewMessageCounts&gt;</td>
<td>If this parameter is set to true, the queue message counters are activated. As a result, the message count of each queue is displayed in the admin console by default. When this parameter is set to false, the message count can be read from the console on user demand.</td>
<td>Boolean</td>
<td>false</td>
<td>true/false</td>
<td>Optional</td>
<td>Setting this parameter to true has a direct impact on the broker performance. Therefore, change the value to true only when a performance degradation is acceptable.</td>
</tr>
</tbody>
</table>

<status-updates>

If the value for this parameter is on, status updates from the QPID broker will be generated.
Configuring qpid-virtualhosts.xml

The <MB_HOME>/repository/conf/advanced/qpid-virtualhosts.xml file allows you to configure virtual hosts for WSO2 Message Broker. Following is the tree of the XML elements in this file:

```xml
<virtualhosts>
  <default>
    <virtualhost>
      <name>
        <carbon>
          <housekeeping>
            <threadCount>
              <expiredMessageCheckPeriod>
                <exchanges>
                  <type>
                    <name>
                      <durable>
                        <exchange>
                          <type>
                            <name>
                              <queues>
                                <maximumQueueDepth>
                                <maximumMessageSize>
                                <maximumMessageAge>
                                <maximumMessageCount>
                  </virtualhosts>
                  </virtualhost>
                  </carbon>
                  </housekeeping>
                  </threadCount>
                  </expiredMessageCheckPeriod>
                  </exchanges>
                  </type>
                  </name>
                  </durable>
                  </exchange>
                  </type>
                  </name>
                  </queues>
                  </maximumQueueDepth>
                  </maximumMessageSize>
                  </maximumMessageAge>
                  </maximumMessageCount>
</virtualhosts>
```

Click an element below for more information about that element.

- `<virtualhosts>`
- `<virtualhost>`
- `<carbon>`
  - `<housekeeping>`
  - `<exchanges>`
  - `<queues>`

This configuration file contains details of all queues and topics, and associated properties, to be created on broker startup. These details are configured on a per virtual host basis. Note that if you do not prate this file with details of a queue or topic you intend to use, you must first create a consumer on a queue/topic before you can publish to it using the WSO2 MB. Thus, most application deployments need a `virtualhosts.xml` file with minimal detail.

### Configurable Sub Elements

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;default&gt;</td>
<td>This parameter sets the default virtual host for connections which do not specify a virtual host</td>
<td>String</td>
<td>carbon</td>
<td>one of defined virtual-hosts</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;virtualhost&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


Define a virtual host and all its configurations under this element. All sub sections are included under this element.

**Configurable Sub Elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;name&gt;</td>
<td>This parameter sets an identifiable name for the virtualhost</td>
<td>String</td>
<td>carbon</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>

**<carbon>**

All configuration options for **carbon** virtual host are defined under this element.

**<housekeeping>**

Housekeeping task configurations for virtual hosts. This section configures the clean-up threads that work on flushing out obsolete/expired messages from the AMQP exchanges.

**Configurable Sub Elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;threadCount&gt;</td>
<td>The number of clean-up threads.</td>
<td>Integer</td>
<td>2</td>
<td>N/A</td>
<td>Mandatory</td>
<td></td>
</tr>
<tr>
<td>&lt;expiredMessageCheckPeriod&gt;</td>
<td>The time intervals at which the QPID broker checks for expired messages.</td>
<td>Integer</td>
<td>20000</td>
<td>N/A</td>
<td>Mandatory</td>
<td>The value is specified in milliseconds.</td>
</tr>
</tbody>
</table>

**<exchanges>**

This element defines the types of additional AMQP exchange available for this virtual host. You should always have amq.direct (for queues) and amq.topic (for topics) by default. You can declare an additional exchange type for developer use only.

**<queues>**

Note that if you do not update this file with details of a queue or topic you intend to use, you should first create a consumer on a queue/topic before you can publish to it using WSO2 MB.

**Configurable Sub Elements**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Fixed Values</th>
<th>Mandatory/Optional</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;maximumQueueDepth&gt;</td>
<td>Defines the maximum number of messages that can be kept in the internal queue buffer for delivery. You can disable the parameter by setting it to 0.</td>
<td>0</td>
<td>N/A</td>
<td></td>
<td>Mandatory</td>
<td></td>
</tr>
</tbody>
</table>
### 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>
<tr>
<td>Tag</td>
<td>Name</td>
<td>Description</td>
<td>Mandatory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;parameter&gt;</td>
<td>name</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 name attribute (required) specifies the parameter name. If you set the locked attribute to true (default is false), this parameter's value cannot be overridden by services and other configurations.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>locked</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;listener&gt;</td>
<td>class</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 class 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>
<td>Optional</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;messageReceivers&gt;</td>
<td></td>
<td>The container element for message receiver definitions.</td>
<td>Mandatory</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;messageReceiver&gt;</td>
<td>class</td>
<td>A message receiver definition. The class attribute (required) specifies the message receiver implementation class. The mep attribute (required) specifies the message exchange pattern supported by this message receiver. Each message receiver definition supports only one MEP.</td>
<td>Mandatory</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>mep</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td>Optional/Mandatory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;messageFormatters&gt;</code></td>
<td>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 &quot;messageType&quot; 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).</td>
<td>Optional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;messageFormatter&gt;</code></td>
<td><strong>contentType</strong> and <strong>class</strong> attributes. The <code>contentType</code> attribute specifies which message types are handled by this formatter, and the <code>class</code> attribute specifies the formatter implementation class.</td>
<td>Optional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;messageBuilders&gt;</code></td>
<td>The container element for the message builder definitions, which are used to process the raw payload of incoming messages and convert them to SOAP.</td>
<td>Optional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;messageBuilder&gt;</code></td>
<td><strong>contentType</strong> and <strong>class</strong> attributes. The <code>contentType</code> attribute specifies which message types are handled by this builder, and the <code>class</code> attribute specifies the builder implementation class.</td>
<td>Optional</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;transportReceiver&gt;</code></td>
<td><strong>name</strong> and <strong>class</strong> attributes. A transport receiver definition, one for each transport type. The <code>name</code> attribute specifies the short name to use when referring to this transport in your configurations (http, tcp, etc.), and the <code>class</code> attribute specifies the receiver implementation class that provides the logic for receiving messages via this transport. You can specify <code>&lt;parameter&gt;</code> elements to pass any necessary information to the transport.</td>
<td>Mandatory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>&lt;transportSender&gt;</strong></td>
<td>Just like <strong>&lt;transportReceiver&gt;</strong>, except <strong>&lt;transportSender&gt;</strong> allows you to define transport senders, which are used to send messages via the transport.</td>
<td>Mandatory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
<td>-----------</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| **<phaseOrder>**     | 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>**          | 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>**        | 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). | Optional |
| **<order>**          |                                                                                                 | Optional |
### 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 -->

... If you want to enable file caching for attachments change this to true
<parameter name="cacheAttachments" locked="false">false</parameter>

<!-- Attachment file caching location relative to CARBON_HOME -->
<parameter name="attachmentDIR" locked="false">work/mtom</parameter>

<!-- Attachment file cache threshold size -->
<parameter name="sizeThreshold" locked="false">4000</parameter>

... 

<!-- Listeners -->

<!-- This deployment interceptor will be called whenever before a module is initialized or -->
<listener class="org.wso2.carbon.core.deployment.DeploymentInterceptor"/>

<!-- 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/wsd1/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
   <messageReceiver mep="http://www.w3.org/ns/wsd1/robust-in-only"
class="org.apache.axis2.rpc.receivers.RPCInOnlyMessageReceiver"/>
   <messageReceiver mep="http://www.w3.org/ns/wsd1/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 -->
<!-- different message formats in Axis2. These message formats are expected to be -->
<!-- resolved based on the content type. -->
<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="WSDLEPRPrefix" 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-->-
<!--<parameter name="http.nonProxyHosts" locked="false">localhost|moon|sun</parameter>-->
</transportSender>

...

<!-- ============================================================== -->
<!--                     Global Engaged Modules                      -->
<!-- ============================================================== -->

<!-- Comment this out to disable Addressing -->
<module ref="addressing"/>

<!-- ============================================================== -->
<!--                     Clustering                                -->
<!-- ============================================================== -->

To enable clustering for this node, set the value of "enable" attribute of the "clustering" element to "true". The initialization of a node in the cluster is handled by the class corresponding to the "class" attribute of the "clustering" element. It is also responsible for getting this node to join the cluster.

<!--
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.
-->

<parameter name="AvoidInitiation">true</parameter>

...

<!--
The list of static or well-known members. These entries will only be valid if the "membershipScheme" above is set to "wka"
-->

<members>
  <member>
    <hostName>127.0.0.1</hostName>
    <port>4000</port>
  </member>
</members>
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>

<!-- 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>-->
<phase name="MsgInObservation">
  <handler name="TraceMessageBuilderDispatcherHandler"/>
  </phase>
  <phase name="Validation"/>
  <phase name="Transport">
    <handler name="RequestURIBasedDispatcher"/>
  </phase>
  <handler name="CarbonContextConfigurator"/>
  <handler name="RelaySecurityMessageBuilderDispatcher"/>
  <handler name="SOAPActionBasedDispatcher"/>
  <handler name="SMTPFaultHandler"/>
  <handler name="CacheMessageBuilderDispatcher"/>
  </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>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Server&gt;</code></td>
<td>xmlns</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Name&gt;</code></td>
<td></td>
<td>Product Name.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;ServerKey&gt;</code></td>
<td></td>
<td>Machine readable unique key to identify each product.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;Version&gt;</code></td>
<td></td>
<td>Product Version.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;HostName&gt;</code></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><code>&lt;MgtHostName&gt;</code></td>
<td></td>
<td>Host name to be used for the Carbon management console.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;ServerURL&gt;</code></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><code>&lt;IndexPageURL&gt;</code></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><code>&lt;ServerRoles&gt;</code></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 : <code>&lt;Role&gt;</code>.*<code>&lt;/Role&gt;</code> means this server can act as any role.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;Role&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;BamServerURL&gt;</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Package&gt;</code></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></td>
<td>Webapp context root of WSO2 Carbon.</td>
<td>String</td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>&lt;RegistryHttpPort&gt;</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>&lt;ItemsPerPage&gt;</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>&lt;InstanceMgtWSEndpoint&gt;</td>
<td>The endpoint URL of the cloud instance management Web service.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Ports&gt;</td>
<td>Ports used by this server</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Offset&gt;</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>&lt;JMX&gt;</td>
<td>The JMX Ports.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;RMIRegistryPort&gt;</td>
<td>The port RMI registry is exposed.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;RMIserverPort&gt;</td>
<td>The port RMI server should be exposed.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;EmbeddedLDAP&gt;</td>
<td>Embedded LDAP server specific ports.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;LDAPServerPort&gt;</td>
<td>Port which embedded LDAP server runs.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;KDCServerPort&gt;</td>
<td>Port which KDC (Kerberos Key Distribution Center) server runs.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;EmbeddedQpid&gt;</td>
<td>Embedded Qpid broker ports.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;BrokerPort&gt;</td>
<td>Broker TCP Port.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;BrokerSSLPor​t&gt;</td>
<td>SSL Port.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;JNDIProviderPort&gt;</td>
<td>Override datasources JNDIproviderPort defined in bps.xml and datasources.properties files.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ThriftEntitlementReceivePort&gt;</td>
<td>Override receive port of thrift based entitlement service.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;JNDI&gt;</td>
<td>JNDI Configuration.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td>&lt;DefaultInitialContextFactory&gt;</td>
<td>The fully qualified name of the default initial context factory.</td>
<td>Strir</td>
<td></td>
</tr>
<tr>
<td>&lt;Restrictions&gt;</td>
<td>The restrictions that are done to various JNDI Contexts in a Multi-tenant environment.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;AllTenants&gt;</td>
<td>Contexts that are common to all tenants.</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;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>Property</td>
<td>Description</td>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td><code>&lt;IsCloudDeployment&gt;</code></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>Bool</td>
<td></td>
</tr>
<tr>
<td><code>&lt;EnableMetering&gt;</code></td>
<td>Property to determine whether usage data should be collected for metering purposes.</td>
<td>Bool</td>
<td></td>
</tr>
<tr>
<td><code>&lt;MaxThreadExecutionTime&gt;</code></td>
<td>The Max time a thread should take for execution in seconds.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;GhostDeployment&gt;</code></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>Bool</td>
<td></td>
</tr>
<tr>
<td><code>&lt;Enabled&gt;</code></td>
<td>When <code>&lt;GhostDeployment&gt;</code> 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><code>&lt;PartialUpdate&gt;</code></td>
<td><code>&lt;PartialUpdate&gt;</code> is a further enhancement to lazy loading of artifacts, which applies when <code>&lt;DeploymentSynchronizer&gt;</code> is enabled in a clustered environment.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Axis2Config&gt;</code></td>
<td>Axis2 related configurations.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;RepositoryLocation&gt;</code></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. <code>/home/wso2wsas/repository/</code> - 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><code>&lt;DeploymentUpdateInterval&gt;</code></td>
<td>Deployment update interval in seconds. This is the interval between repository listener executions.</td>
<td>Int</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ConfigurationFile&gt;</code></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. <code>/home/repository/axis2.xml</code> - An absolute path 2. conf/axis2.xml - In this case, the path is relative to CARBON_HOME 3. file:///home/carbon/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><code>&lt;ServiceGroupContextIdleTime&gt;</code></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><code>&lt;ClientRepositoryLocation&gt;</code></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><code>&lt;clientAxis2XmlLocation&gt;</code></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>Strir</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>
</tbody>
</table>
<TomcatRealm>
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 & roles from the system's WSO2 User Manager. If it is set to 'memory', the realm will pick users & roles from CARBON_HOME/repository/conf/tomcat/tomcat-users.xml.
</TomcatRealm>

<DisableTokenStore>
Option to disable storing of tokens issued by STS.
</DisableTokenStore>

<TokenStoreClassName>
Security token store class name. If this is not set, default class will be org.wso2.carbon.security.util.SecurityTokenStore
</TokenStoreClassName>

<WorkDirectory>
The temporary work directory.
</WorkDirectory>

<HouseKeeping>
House-keeping configuration.
</HouseKeeping>

<AutoStart>
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.
</AutoStart>

<Interval>
The interval in *minutes*, between house-keeping runs.
</Interval>

<MaxTempFileLifetime>
The maximum time in *minutes*, temp files are allowed to live in the system. Files/directories which were modified more than "MaxTempFileLifetime" minutes ago will be removed by the house-keeping task.
</MaxTempFileLifetime>

<FileUploadConfig>
Configuration for handling different types of file upload and other file uploading related config parameters. To map all actions to a particular FileUploadExecutor, use <Action>*</Action>.
</FileUploadConfig>

>TotalFileSizeLimit>
The total file upload size limit in MB.
</TotalFileSizeLimit>

<Mapping>
</Mapping>

<Actions>
</Actions>

>Action>
</Action>

<Class>
</Class>

<HttpGetRequestProcessors>
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
</HttpGetRequestProcessors>

<Processor>
</Processor>

<Item>
</Item>

<Class>
</Class>
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;DeploymentSynchronizer&gt;</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>Strir</td>
</tr>
<tr>
<td>&lt;Enabled&gt;</td>
<td></td>
<td>Boo</td>
</tr>
<tr>
<td>&lt;AutoCommit&gt;</td>
<td></td>
<td>Boo</td>
</tr>
<tr>
<td>&lt;AutoCheckout&gt;</td>
<td></td>
<td>Boo</td>
</tr>
<tr>
<td>&lt;RepositoryType&gt;</td>
<td></td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;SvnUrl&gt;</td>
<td></td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;SvnUser&gt;</td>
<td></td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;SvnPassword&gt;</td>
<td></td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;SvnUrlAppendTenantId&gt;</td>
<td></td>
<td>Boo</td>
</tr>
<tr>
<td>&lt;MediationConfig&gt;</td>
<td>Mediation persistence configurations. Only valid if mediation features are available i.e. ESB.</td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;LoadFromRegistry&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;SaveToFile&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Persistence&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;RegistryPersistence&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ServerInitializers&gt;</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>Strir</td>
</tr>
<tr>
<td>&lt;Initializers&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;RequireCarbonServlet&gt;</td>
<td>Indicates whether the Carbon Servlet is required by the system, and whether it should be registered.</td>
<td>Boo</td>
</tr>
<tr>
<td>&lt;H2DatabaseConfiguration&gt;</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>Strir</td>
</tr>
<tr>
<td>&lt;StatisticsReporterDisabled&gt;</td>
<td>Disables the statistics reporter by default.</td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;EnableHTTPAdminConsole&gt;</td>
<td>Enables HTTP for WSO2 servers so that you can access the Admin Console via HTTP.</td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;FeatureRepository&gt;</td>
<td>Default Feature Repository of WSO2 Carbon.</td>
<td>Strir</td>
</tr>
<tr>
<td>&lt;RepositoryName&gt;</td>
<td></td>
<td>Strir</td>
</tr>
</tbody>
</table>
Configure API Management.

**<RepositoryURL>**

String

**<APIManagement>**

Configure API Management.

String

**<Enabled>**

Uses the embedded API Manager by default. If you want to use an external API Manager instance to manage APIs, configure below externalAPIManager.

Boolean

**<ExternalAPIManager>**

Uncomment and configure API Gateway and Publisher URLs to use external API Manager instance.

String

**<APIServiceGatewayURL>**

**<APIPublisherURL>**

String

**<LoadAPIContextsInServerStartup>**

Boolean

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;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></td>
<td></td>
</tr>
<tr>
<td>shutdown</td>
<td></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>
<td>SHUTDOWN</td>
</tr>
<tr>
<td><strong>port</strong></td>
<td>The TCP/IP port number on which this server waits for a shutdown command. Set to -1 to disable the shutdown port.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>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</td>
<td>sh and catalina.bat</td>
<td>sh from stopping it gracefully.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Int</strong> 8005</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| **<Service>** | 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. |

<table>
<thead>
<tr>
<th><strong>name</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>String</strong> Catalina</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>className</strong></th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>String</strong> org.wso2.carbon.tomcat.ext.service.ExtendedStandardService</td>
</tr>
</tbody>
</table>

| **<Connector>** | |

| **<Connect[or]>** | |

Copyright © WSO2 Inc. 2005-2014
<p>| <strong>port</strong> | 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. | Int | 9763 |
| <strong>URIEncoding</strong> | This specifies the character encoding used to decode the URI bytes, after %xx decoding the URL. | Int | UTF-8 |
| <strong>compressableMimeType</strong> | The value is a comma separated list of MIME types for which HTTP compression may be used. | String | text/html,text/javascript,application/x-javascript,application/javascript,application/xml,text/css,application/xslt+xml,text/xsl,image/gif,image/jpg,image/jpeg |
| <strong>noCompressionUserAgents</strong> | 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. | String | gozilla, traviata |
| <strong>compressionMinSize</strong> | 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. | Int | 2048 |</p>
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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>String</td>
<td>on</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>String</td>
<td>WSO2 Carbon Server</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>Int</td>
<td>200</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>Attribute</td>
<td>Description</td>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<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>Int</td>
<td>9443</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. 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() andRequest.isSecure() values to the servlets. See SSL Support for more information.</td>
<td>Boolean</td>
<td>true</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>Boolean</td>
<td>true</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(). For example, you would set this attribute to &quot;https&quot; for an SSL Connector.</td>
<td>String</td>
<td>https</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 will not require a certificate chain unless the client requests a resource protected by a security constraint that uses CLIENT-CERT authentication.</td>
<td>Boolean</td>
<td>false</td>
</tr>
<tr>
<td>Property</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 keystorePass</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>&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>Parameter</td>
<td>Description</td>
<td>Type</td>
<td>Default</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>
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><strong>name</strong></th>
<th>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.</th>
<th>String</th>
<th>localhost</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>appBase</strong></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><strong>autoDeploy</strong></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><strong>deployOnStartup</strong></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><strong>unpackWARs</strong></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><strong>&lt;Valve</strong></td>
<td>The Access Log Valve creates log</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></th>
<th>Java class name of the implementation to use.</th>
<th>String</th>
<th>org.wso2.carbon.tomcat.ext.valves.CarbonContextCreatorValve, org.apache.catalina.valves.AccessLogValve</th>
</tr>
</thead>
<tbody>
<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.</td>
<td>Int</td>
<td>600</td>
</tr>
</tbody>
</table>

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.

**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>
</tr>
</thead>
</table>

Copyright © WSO2 Inc. 2005-2014
<table>
<thead>
<tr>
<th>XML Element</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;datasources-configuration&gt;</code></td>
<td>The root element. The namespace is specified as: <code>xml ns:svns=&quot;http://org.wso2.securevault/configuration&quot;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;providers&gt;</code></td>
<td>The container element for the datasource providers.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;provider&gt;</code></td>
<td>The datasource provider, which should implement <code>org.wso2.carbon.ndatasource.common.spi.DataSourceReader</code>. 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>The container element for the datasources.</td>
<td></td>
</tr>
<tr>
<td><code>&lt; datasource&gt;</code></td>
<td>The root element of a datasource.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;name&gt;</code></td>
<td>Name of the datasource.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;description&gt;</code></td>
<td>Description of the datasource.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;jndiConfig&gt;</code></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>The JNDI resource name to which this datasource will be bound.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;environment&gt;</code></td>
<td>The container element in which you specify the following JNDI properties:</td>
<td>Fully qualified Java class</td>
</tr>
<tr>
<td></td>
<td>• <code>java.naming.factory.initial</code>: Selects the registry service provider as the initial context.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• <code>java.naming.provider.url</code>: Specifies the location of the registry when the registry is being used as the initial context.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;definition&gt;</code></td>
<td>The container element for the data source definition.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;configuration&gt;</code></td>
<td>The container element for the RDBMS properties.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;url&gt;</code></td>
<td>The connection URL to pass to the JDBC driver to establish the connection.</td>
<td>URL</td>
</tr>
<tr>
<td><code>&lt;username&gt;</code></td>
<td>The connection user name to pass to the JDBC driver to establish the connection.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;password&gt;</code></td>
<td>The connection password to pass to the JDBC driver to establish the connection.</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;driverClassName&gt;</code></td>
<td>The class name of the JDBC driver to use.</td>
<td>Fully qualified Java class</td>
</tr>
<tr>
<td>Tag</td>
<td>Description</td>
<td>Type</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>----------</td>
</tr>
<tr>
<td><code>&lt;maxActive&gt;</code></td>
<td>The maximum number of active connections that can be allocated from this pool at the same time.</td>
<td>Integer</td>
</tr>
<tr>
<td><code>&lt;maxWait&gt;</code></td>
<td>Maximum number of milliseconds that the pool waits (when there are no available connections) for a connection to be returned before throwing an exception.</td>
<td>Integer</td>
</tr>
<tr>
<td><code>&lt;testOnBorrow&gt;</code></td>
<td>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.</td>
<td>Boolean</td>
</tr>
<tr>
<td><code>&lt;validationQuery&gt;</code></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 SELECT 1(mysql), select 1 from dual(oracle), SELECT 1(MS Sql Server).</td>
<td>String</td>
</tr>
<tr>
<td><code>&lt;validationInterval&gt;</code></td>
<td>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).</td>
<td>Long</td>
</tr>
</tbody>
</table>

**Example**
<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>Tag</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>&lt;currentDBConfig&gt;</code></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">http://docs.wso2.org/display/Governance501/Data+base+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td><code>&lt;readOnly&gt;</code></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">http://docs.wso2.org/display/Governance501/Registry+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td><code>&lt;enableCache&gt;</code></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>
<td></td>
</tr>
<tr>
<td><code>&lt;registryRoot&gt;</code></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">http://docs.wso2.org/display/Governance501/Registry+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td><code>&lt;dbConfig&gt;</code></td>
<td>name</td>
<td></td>
</tr>
<tr>
<td><code>&lt;dataSource&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;handler&gt;</code></td>
<td>Handlers are pluggable components, that contain custom processing logic. All handlers extend an abstract class named Handler, which provides default implementations for basic methods and a few utilities useful for concrete implementations. Handler implementations can provide alternative behaviors for basic 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">http://docs.wso2.org/display/Governance501/Handler+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td><code>class</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;filter&gt;</code></td>
<td>class</td>
<td></td>
</tr>
<tr>
<td><code>&lt;remoteInstance&gt;</code></td>
<td>In order to mount an external registry, you have to define the remote instance. 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">http://docs.wso2.org/display/Governance501/Remote+Instance+and+Mount+Configuration+Details</a></td>
<td></td>
</tr>
<tr>
<td><code>url</code></td>
<td>The URL of the remote instance.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;ID&gt;</code></td>
<td>Remote instance ID.</td>
<td></td>
</tr>
</tbody>
</table>
<username>
Username of the remote registry login.

<password>
Password of the remote registry login.

<dbConfig>
The database configuration to use.

<readOnly>
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 only for the specific remote instance.

For more information, see the Governance Registry documentation here: http://docs.wso2.org/display/Governance501/Registry+Configuration+Details

<enableCache>
To enable registry caching, set the enableCache element to true. Once executed against the cache instead of the database. This setting is valid only for the specific remote instance.

<registryRoot>
The registryRoot parameter can be used to define whether the application setting is valid only for the specific remote instance.

For more information, see the Governance Registry documentation here: http://docs.wso2.org/display/Governance501/Registry+Configuration+Details

<mount>
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: http://docs.wso2.org/display/Governance501/Remote+Instance+and+Mount+Configuration+Details

<path>
The path to which the mount will be added to.

<overwrite>
Whether an existing collection at the given path would be overwritten or not.

<instanceID>
Remote instance ID.

<targetPath>
The path on the remote registry.

<versionResourcesOnChange>
You can configure whether you want to auto-version the resources (non-collection) by setting versionResourcesOnChange element to true. In this configuration it will create a version for the resource whenever it is updated. For more information, see the Governance Registry documentation here: http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources

<staticConfiguration>
While most configuration options can be changed after the first run of the server, changing the Static Configuration (configuration details under the staticConfiguration parameter) will not be fully effective. If you need to change any Static Configuration and expect it to take effect, you will have to erase the contents of the database, and restart the server passing the -Dsetup system property which will re-generate the database.

You are supposed to change the static configuration section only before the first start-up. For more information, see the Governance Registry documentation here: http://docs.wso2.org/display/Governance501/Configuration+for+Static+%28One-time%29+and+Auto+Versioning+Resources

.VERSIONING_PROPERTIES
Whether the properties are versioned when a snapshot is created.

.VERSIONING_COMMENTS
Whether the comments are versioned when a snapshot is created.

.VERSIONING_TAGS
Whether the tags are versioned when a snapshot is created.

.VERSIONING_RATINGS
Whether the ratings are versioned when a snapshot is created.
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>&lt;UserManager&gt;</td>
<td></td>
<td>User kernel configuration for Carbon server.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Realm&gt;</td>
<td></td>
<td>Realm configuration.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;Configuration&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;AddAdmin&gt;</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>&lt;AdminRole&gt;</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>&lt;AdminUser&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&lt;UserName&gt;</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></td>
<td>&lt;Password&gt;</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>&lt;EveryOneRoleName&gt;</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>&lt;Property&gt;</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.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;UserStoreManager&gt;</td>
<td>User store configuration specific property values. See <a href="#">working with primary user store properties</a> for more information.</td>
<td>String</td>
<td>N/A</td>
<td>Optional</td>
</tr>
<tr>
<td>&lt;AuthorizationManager&gt;</td>
<td>Authorization manager implementation class and its configuration for user realm.</td>
<td>String</td>
<td>N/A</td>
<td>Mandatory</td>
</tr>
<tr>
<td>&lt;Property&gt;</td>
<td>Authorization manager configuration specific property values.</td>
<td>String</td>
<td>N/A</td>
<td>Optional</td>
</tr>
</tbody>
</table>

### Configuring hazelcast.properties

The `<MB_HOME>/repository/conf/hazelcast.properties` file is used to configure Hazelcast parameters that affect the performance of your MB cluster. The configurable parameters are as follows.

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Description</th>
<th>Type</th>
<th>Default Value</th>
<th>Notes</th>
</tr>
</thead>
</table>

Copyright © WSO2 Inc. 2005-2014
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Type</th>
<th>Value</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>hazelcast.max.no.heartbeat.seconds</td>
<td>The maximum time period that should elapse between pings received from a worker node in an MB cluster before acknowledging that worker node to be dead. This parameter prevents the allocation of resources to inactive worker nodes, thereby avoiding unnecessary system overheads.</td>
<td>Integer</td>
<td>600</td>
<td>This value is specified in seconds.</td>
</tr>
<tr>
<td>hazelcast.shutdownhook.enabled</td>
<td>This parameter specifies whether the Hazelcast shutdown hook is enabled or not.</td>
<td>Boolean</td>
<td>false</td>
<td>This parameter is not configurable.</td>
</tr>
<tr>
<td></td>
<td>When this parameter is set to false, the inbuilt shutdown hook thread of Hazelcast is disabled. This is because the WSO2 message broker is manually shutting down the hazelcast service.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Setting up with Remote Derby

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

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

Creating 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 drivers

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

Setting up datasource configurations

After creating the database, you create a datasource to point to it in the following files:

1. Edit the default datasource configuration in the <MB_HOME>/repository/conf/datasources/master-datasources.xml file. Replace the url, username, password and driverClassName settings with your custom values and also the other values accordingly 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>
    </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.

---

For more information on other parameters that can be defined in the `<MB_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.

---

2. Similarly, edit the database configurations in `<MB_HOME>/repository/conf/registry.xml`, `<MB_HOME>/repository/conf/user-mgt.xml` and `<MB_HOME>/repository/conf/identity.xml` files as well.

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.

   ```
   client@wso2:~/dtb/db-derby-10.8.1.2-bin/bin/bin$ ij
   ij version 10.8
   i> 
   ```

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

   ```
   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:

   ```
   exit;
   ```

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

   ```
   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:

     ```
     run '<MB_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: `<MB_HOME>/bin/wso2server.bat -Dsetup`
• For Linux: `<MB_HOME>/bin/wso2server.sh -Dsetup`

The product is now configured to run using a remote Apache Derby database.
Setting up with Embedded Derby

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

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

Creating 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 drivers

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

Setting up datasource configurations

After creating the database, you create a datasource to point to it in the following files.

1. Edit the default datasource configuration in the <MB_HOME>/repository/conf/datasources/master-datasources.xml file. Replace the url, username, password and driverClassName settings with your custom values and also the other values accordingly 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>
1. The validationInterval parameter is used 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.

2. Similarly, edit the database configurations in `<MB_HOME>/repository/conf/registry.xml`, `<MB_HOME>/repository/conf/user-mgt.xml` and `<MB_HOME>/repository/conf/identity.xml` files as well.

### 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:

   ```
   $ /ij
   ij version 10.8
   ij>
   ```

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

   ```
   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.

   ```
   exit;
   ```

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

   ```
   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:

     ```
     run '<MB_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: `<MB_HOME>/bin/wso2server.bat -Dsetup`
   - For Linux: `<MB_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.