Key Concepts and Features

Given below are the key concepts used in WSO2 Data Services Server (WSO2 DSS) and the capabilities or features that are available with the product. If you are already familiar with these concepts and features, move to the tutorials to see how these features work.

- Create data services using the management console
- Expose various datasources
- Exposing data as web services/REST resources
- Expose data as OData services
- Using namespaces for data services
- Process requests in batches
- Invoke multiple operations simultaneously (boxcarring/request box)
- Expose data through various transports
- Handling data federation
- Manage large data chunks using streaming
- Transactional data access and services (distributed transactions)
- RESTful data access
- Transform and validate data
- Secured and managed data access
- Monitor data services

Create data services using the management console

WSO2 DSS allows you to decouple data (from various datasources) from the infrastructure where it is stored and expose them as data services. You can create a comprehensive data service in a few simple steps using the management console of WSO2 DSS. There are three ways of creating data services in WSO2 DSS.

- Creating a data service from scratch

The management console of WSO2 DSS provides a convenient wizard that will guide you to create the service from scratch using a few simple steps. The following diagram depicts the steps you need to follow.

Let’s briefly look at each step of the Create New Data Service wizard:

1. **Define the data service**: In the first step, you must specify the general details of the data service, such as the name and description of the data service. In addition, you can enable advanced options for your data service such as Namespaces, Batch Requests, Data Streaming, Transport settings and Distributed Transaction settings.

2. **Add Datasources**: In the second step, you must specify the details of the datasources that should be exposed by the data service. You can add multiple datasources to the data service to achieve data federation. The parameters that you can configure for a datasource depends on the type of datasource. For example, if you use an RDBMS or Cassandra as the datasource, you can enable OData.

3. **Define Queries**: In the third step, you must specify queries to process the data in a particular datasource. For example, create queries to search for data in the datasource, or insert/update/delete data. The query you define will consist of parameters that you will use to carry out the query function. For example, to insert data into the datasource, you must specify input mapping parameters and to view data from the datasource, you must specify output mapping parameters.

4. **Define Operations**: In the fourth step, you can define an operation connected to a single query if you want to expose the data as a SOAP service. The operation will be used to invoke the query.

5. **Expose data as a REST resource**: In the next step, you have the option of exposing the data from your data services as REST resources. To do this, you will be defining a resource connected to each q
Uploading an already created data service (.dbs file)

If you already have a data service created, you can directly upload the .dbs file of the data service using the management console. See Uploading a Created Data Service for instructions.

Generating a data service using a defined database structure

If a database structure is already defined, the data service will be automatically generated for the datasource that you specify. See Generating a Data Service for instructions.

Expose various datasources

When you create a data service, you can service enable data from various datasources.

- Supported datasources: Any RDBMS, CSV, Excel, Carbon Datasources, Cassandra, Google Spreadsheets, RDF, JNDI Datasources, Custom Datasources, Any Web page via scraping, MongoDB.
- Supported databases: MSSQL, DB2, Oracle, OpenEdge, TerraData, MySQL, PostgreSQL/EnterpriseDB, H2, Derby or any database with a JDBC driver.

Exposing data as web services/REST resources

Data from various datasources can be exposed as SOAP services or REST-style web resources using WSO2 DSS. When creating a data service, define an operation to expose data as a SOAP service or define a resource to expose data in REST-style.

Expose data as OData services

From DSS 3.5.0 onwards, OData protocol version 4 (OASIS standards) is supported, which mainly provides support for CRUD operations. You can easily expose databases as an odata service by selecting the OData check box in the Create New Data Service wizard. Currently, support for Odata service is available for RDBMS datasources and Cassandra datasources. If you have enabled Odata for your data service, you can complete the data service creation process without defining queries or operations for the service. This Odata service will now be accessible from the following endpoints:

- For super tenant: http://localhost:9763/odata/{dataserviceName}/{datasourceId}/
- For normal tenants: http://localhost:9763/odata/t/{tenantId}/{dataserviceName}/{datasourceId}/

You can find a sample demonstration of an OData service here.

Using namespaces for data services

The service namespace uniquely identifies a Web service and is specified by the <targetNamespace> element in the WSDL that represents the service. A data service is simply a Web service with specialized functionality. As a data service implementation is based on XML, namespace handling is an important aspect to it.

When developing a data service, you get to specify namespaces in several places. One is as shown above. Others are result row namespace, namespaces for specific elements in a query result and for complex results. For
Process requests in batches

You can enable batch requesting for operations that contain multiple parameters for a single request. When a data service is created with the batch request mode enabled, for all the in-only operations (i.e., operations that do not have any return value), the system creates a corresponding batch operation automatically. This batch operation takes in an array of parameters, compared to the single parameter list a non-batch operation takes.

Batch requests can only be used with in-only operations. That is, the query cannot have a result element or out-type parameters. Batch requests are typically used with insert operations.

For a practical demonstration, see the Batch Processing Sample.

Invoke multiple operations simultaneously (boxcarring/request box)

Using request_box: When using the generated wsdl, it will contain all the service level details you need to know in order to invoke a “request_box” request. It’s simply a wrapper element (“request_box”), which wraps the operations you need to call. You can put any number of operations within that “request_box” element any number of times. The order of operations depend on the flow you need to follow. Furthermore, this will return the result of the last operation you have put in the “request_box” wrapper element. The “request_box” operation will be automatically added to the data service.

From DSS 3.5.0 onwards we are deprecating the back end approach used for boxcarring. Instead, we are introducing a new request type (request_box), where you can call multiple operations via a single service call. In WSO2 DSS 3.5.0, both boxcarring and request box is disabled in a data service by default. You can enable both these options by clicking the Enable Boxcarring check box when you create a new data service. The same "enableBoxcarring="true" attribute in the .dbs file applies to both boxcarring and request box.

About Boxcarring

Boxcarring is a method of grouping a set of service calls together and executing them at once. Where applicable, a boxcarring session works in a transactional manner such as when used with an RDBMS datasource. WSO2 Data Services Server facilitates boxcarring by grouping service calls in the server side. As a result, special service clients are not required and as usual, successive service calls can be made to the server to participate in a boxcarring session.

The following control operations get automatically added to the data service when boxcarring is used:

- begin_boxcar : A boxcarring session must be started by calling this operation. Once this is called, the server is notified that the subsequent operation calls must belong to this boxcarring session. The server stores these calls without executing them immediately.
- end_boxcar : After begin_boxcar is called, the actual operations that belong to the boxcarring session follow. The last step is executing the grouped operations and ending the boxcarring session. This is done with the end_boxcar operation. Once this is called, all the grouped operations are executed at once and the boxcarring session ends. After calling end_boxcar, results are not returned to the client because there are multiple service calls executed at once. If there are any results in the operations of the boxcarring session, they are discarded.
- abort_boxcar : If an error occurs in a boxcarring session, the user can choose to end the session by calling the abort_boxcar operation. This invalidates the boxcarring session and removes all pending operations in it.

For boxcarring to function properly, you must use a transport that supports session management, such as HTTP. The service client must also support session management by returning session cookies sent by the server. Boxcarring uses the session information to keep track of operations belonging to one batch. This session information will be sent along the request from the beginning of the boxcarring session to its end. That is, the cookie from the
Session management in Axis2 clients

Axis2 service clients have full support for session management. The following code snippet shows the process of enabling a session in the client and invoking operations in a boxcarring session.

```java
RDBMSSampleStub stub = new RDBMSSampleStub(epr);
stub._getServiceClient().getOptions().setManageSession(true);
stub.begin_boxcar();
stub.addEmployee(49001, "Smith", "John", "john@test.com", 10000.0);
stub.incrementEmployeeSalary(5000.0, 1002);
stub.incrementEmployeeSalary(4500.0, 1003);
stub.end_boxcar();
```

In line 2 of the code segment above, the client is requested to handle sessions. Without this command, the boxcarring sessions are not created in the server side. The three operation calls that follow the begin_boxcar operation belong to the newly-created boxcarring session. When end_boxcar is called, the earlier three operations are executed together as a group, in one transaction.

**Note:** When using MySQL as the RDBMS, InnoDB Storage Engine must be used.

Session management in CXF clients

If the client is CXF, when executing a boxcarring session, you must enable robust in only option with the session management functionality. This is done by setting the endpoint property `org.apache.cxf.oneway.robust` to `true` to activate it at the endpoint. A sample CXF client code is shown below:

```java
private static final QName SERVICE_NAME = new QName("http://ws.wso2.org/dataservice/CXFClientService", "CXFClientDataService");
CXFClientDataService ss = new CXFClientDataService(wsdlURL, SERVICE_NAME);
CXFClientDataServicePortType port = ss.getSOAP11Endpoint();
BindingProvider bp = (BindingProvider)port;
bp.getRequestContext().put(BindingProvider.SESSION_MAINTAIN_PROPERTY, true);
bp.getRequestContext().put(Message.ROBUST_ONEWAY, Boolean.TRUE);
port.beginBoxcar();
```

When you work with boxcarring in a clustered setup, you need to enable **sticky sessions** in your load balancer. Otherwise, each request will be treated as an individual request, and the transaction behavior will not work as expected for the whole boxcarring session. Note that you do not need sticky sessions when using request box, because request box is always a single request that is sent to a single node in the cluster. The sticky sessions feature is required only when we need to send multiple requests to a single node in the cluster, as in the case of boxcarring.

- See the Request Box Sample for a practical demonstration of the request box.
- See the Boxcarring Sample for a practical demonstration of boxcarring.
Expose data through various transports

The transport settings of a data service allow you to specify the transport protocols that should be applicable when receiving messages to the data service. However, this facility to select the transport types as and when you create the data service will directly add the transport settings to the .dbs file of the data service. See the documentation on product administration for information on the transports used in DSS.

Handling data federation

Data from multiple datasources can be exposed through a single data service.

Data federation can be achieved in three ways when you use WSO2 DSS:

- Expose multiple datasources using a single data service.
- Use Nested Queries in your data service. This will allow you to feed the result you get from one query as input to another query. That is, data can be combined into a single response or resource.

For a practical demonstration of how nested queries can be used for data federation, see the Nested Query Sample.

Manage large data chunks using streaming

Data service streaming helps manage large data chunks sent back to the client as the response to a data service query. When streaming is enabled, the data is sent to the client as it is generated, without memory building up in the server. By default, streaming is enabled in data services.

Transactional data access and services (distributed transactions)

Distributed transaction support helps you manage two or more transactions, often using multiple databases, in a coordinated way. A transaction manager is set up in the middle of these transactions for effective coordination and management. This feature uses Java Transaction API (JTA), which allows distributed transactions to be carried out across multiple XA resources in a Java environment. You can also override this transaction manager.

For a practical demonstration of distributed transaction, see Distributed Transactions Sample in Samples.

Distributed transactions with JMS

JMS has the ability to participate in distributed transactions. You can use JMS as the communication channel and run your database transactions in a distributed manner. It guarantees no message loss or failures and ensures single, successful message delivery.
If a JTA transaction is already started, incoming messages will automatically bind with it. In this case, the transaction is started using the JMS transport receiver, if the receiver is enabled as follows:

- Enable the JMS transport receiver in `<PRODUCT_HOME>/repository/conf/axis2.xml` file. The `axis2.xml` file contains properties related to Apache ActiveMQ JMS broker by default. Change the settings as suitable to the JMS broker you use.
- Add dependent libraries to `<PRODUCT_HOME>/repository/components/lib`.
- Add the following two properties to configure the transport receiver to use JTA transactions:
  - `jta`
  - `true`

After the JMS transport is configured, you can use any JMS client to send messages to the configured JMS broker. For example, you can use the Try-it tool in WSO2 Data Services Server. Be sure to enable the JMS transport sender by un-commenting the code block for JMSSender in `<PRODUCT_HOME>/repository/conf/axis2.xml` file.

**Overriding the default transaction manager**

The default transaction manager of a server can be overridden by providing the location to a JNDI name that the transaction manager is bound to. This is especially useful when the server is embedded in other application servers that do not use the standard JNDI name to bind to the transaction manager. In this case, you can provide a custom JNDI name to detect the transaction manager.

Shown below is an example how WebLogic's JTA transaction manager overrides a data service's default transaction manager.

![Edit Data Service (DTPSampleService)](image)

**RESTful data access**

Data can be exposed as REST resources in addition to SOAP services.

- Create collections of REST resources for CRUD operations.
- Support for both JSON and XML media types.
- Secure resources with HTTP(S) Basic Auth integrated to enterprise identity systems (via WSO2 Identity Server).

For a practical demonstration of this feature, see the sample on REST-Style Resources Sample.

### Transform and validate data

- Transform via XSLT and XQuery.
- Built-in validators for standard data types.
- Customizable validators via Java extensions.
- Schema validation via integration with WSO2 Enterprise Service Bus.

For a practical demonstration of this feature, see the XSLT Transformation Sample.

### Secured and managed data access

- Fine-grained authorization with XACML (via integration to WSO2 Enterprise Service Bus).
- Comprehensive user management including LDAP via integration to WSO2 Identity Server.

### Monitor data services

WSO2 DSS provides a dashboard for monitoring all the running statistics related to data services. You can obtain more advanced monitoring capabilities for your data services by integrating with WSO2 Data Analytics Server.
(WSO2 DAS).