Red Hat Integration 2019-07

Data Virtualization

Data Virtualization

Last Updated: 2019-08-21

Red Hat Integration 2019-07 Data Virtualization

Data Virtualization

Legal Notice

Copyright © 2019 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

Combine data from multiple sources so that applications can connect to a single, virtual data model

Table of Contents

CHAPTER 1. HIGH-LEVEL OVERVIEW OF DATA VIRTUALIZATION 3

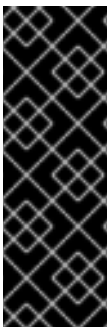
CHAPTER 2. CREATING VIRTUAL DATABASES 4
2.1. USING MAVEN TO CREATE A JAVA SHELL PROJECT FOR YOUR VIRTUAL DATABASE 5
2.2. CREATING AND DEPLOYING SECRETS 5
2.3. SECRET OBJECTS FOR STORING DATA SOURCE INFORMATION 6
2.4. SPECIFYING PROJECT DEPENDENCIES IN THE POM.XML FILE 7
2.5. DEFINING THE STRUCTURE FOR VIRTUAL DATABASES IN A DDL FILE 8
2.6. DDL FILES 9
2.7. ADDING JAVA APPLICATION AND CLASS FILES 10
2.8. SAMPLE DATASOURCES.JAVA FILE 11
2.9. SPECIFYING APPLICATION PROPERTIES 12
2.10. SAMPLE APPLICATION.PROPERTIES FILE 12
2.11. DEPLOYMENT CONFIGURATION FILES (DEPLOYMENTCONFIG.YML) 13
2.12. SETTING THE DEPLOYMENT CONFIGURATION 14
2.13. CONNECTION SETTINGS FOR OTHER DATA SOURCES 15
2.13.1. Settings to connect to Salesforce as a data source 16
2.13.2. Settings to connect to Google Sheets as a data source 18
2.14. MAKING VIRTUAL DATABASES AVAILABLE TO API CONSUMERS 19
2.14.1. Configuring access for OData clients 20
2.14.2. Configuring access for JDBC clients 21
2.14.3. Identifying the OData endpoint of a virtual database 22
2.15. BUILDING A DATA VIRTUALIZATION PROJECT 23

CHAPTER 3. SECURING ODATA APIS FOR A VIRTUAL DATABASE BY USING 3SCALE AND RH-SSO 24
3.1. UPDATING THE VIRTUAL DATABASE CONFIGURATION 24
3.1.1. Adding SSO dependencies to pom.xml 25
3.1.2. Adding SSO settings to application.properties 25
3.1.3. Creating a configuration map 26
3.1.4. Updating SSO environment variables in the deploymentconfig.yml file 26
3.1.5. Defining user roles in the DDL file 27
3.2. UPDATING THE 3SCALE CONFIGURATION 27

CHAPTER 4. MIGRATING LEGACY VIRTUAL DATABASE FILES TO DDL FORMAT 29
4.1. VALIDATING A LEGACY VIRTUAL DATABASE XML FILE AND VIEWING IT IN DDL FORMAT 29
4.2. CONVERTING A LEGACY VIRTUAL DATABASE XML FILE AND SAVING IT AS A DDL FILE 30

CHAPTER 1. HIGH-LEVEL OVERVIEW OF DATA VIRTUALIZATION

Data virtualization is a container-native service that provides integrated access to multiple diverse data sources, including relational databases, files, web services, and SaaS repositories through a single uniform API. Applications and users connect to a virtual database over standard interfaces (OData REST, or JDBC/ODBC) and can interact with data from all configured data sources as if it were stored in a single relational database.



IMPORTANT

Data virtualization is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

CHAPTER 2. CREATING VIRTUAL DATABASES

Prerequisites

You design and create a virtual database and then deploy it to an OpenShift container. After you create the virtual database, you can make it available to API consumers, which can connect to it as if it were a single relational database.

For this Technology Preview release, Fabric8 and Maven Spring Boot plugins are required to build and deploy virtual databases on OpenShift. The Spring Boot Maven plugin converts the Teiid library into an executable Spring Boot JAR file. The Fabric8 Maven plugin helps to build a container image that is based on the Spring Boot executable and optionally deploy that container to OpenShift. Both plugins are available from the public Red Hat maven repository, and are downloaded on demand at build time after you specify dependencies in your **pom.xml** file.

You can also create a virtual database in Fuse Online. However, in the current Technology Preview, virtual database created in Fuse Online provide a more limited set of features. Most developers will use Maven and Spring Boot to create and deploy a virtual database.

For information about data sources that you can use with Teiid, the open source project on which data virtualization is based, see: Data sources in the Teiid Reference Guide.

Prerequisites

- You have the 2019-07 release of Red Hat Integration.
- You have Developer access to an OpenShift server and you are familiar with using the OpenShift console and CLI.
- You have a DDL file for the virtual database that you want to create, or you know how to write SQL code and create DDL files.
- You have Maven 3.0 installed.
- You have the Java Platform, Standard Edition 11 Development Kit (JDK 11) or greater installed.
- You can connect to a postgreSQL or other relational database, or to another data source. In this Technology Preview, along with relational databases, you can connect to the following data sources:
  - Salesforce databases
  - Google Sheets spreadsheets
- You have administrator access to the configured data source.

Summary of steps:

1. Use Maven to generate a Java shell project.
2. Create and deploy a secret object in OpenShift .
3. Specify core project dependencies and data source dependencies in the **pom.xml** file.
4. Add a DDL file to define the structure of the virtual database .
5. Add **Application.java** and **DataSources.java** files.

6. Define static application properties in the **application.properties** file.
7. Add a **deploymentconfig.yml** file to define environment variables to configure the virtual database.
8. Make the virtual database available to API consumers .
9. Run Maven commands to deploy the virtual database image to OpenShift .

2.1. USING MAVEN TO CREATE A JAVA SHELL PROJECT FOR YOUR VIRTUAL DATABASE

Use Maven to generate a Java shell project that you can then modify.

Prerequisites

- You have Maven 3.0 installed.
- You have Java JDK 11 or greater installed.

Procedure

1. On your local workstation, change to a directory where you want to create the Java project for your virtual database.
2. Run the following Maven command to generate a plain Java shell project:

```
mvn archetype:generate -DgroupID=<domainSuffix>.<domainName> -DartifactID=
<virtualDbName> -DarchetypeArtifactId=maven-archetype-quickstart -
DinteractiveMode=false
```

For example:

```
mvn archetype:generate -DgroupId=com.example -DartifactId=sampledb -
DarchetypeArtifactId=maven-archetype-quickstart  -DinteractiveMode=false
```

All of the artifacts that you need to create a Java project are saved to the project directory.

2.2. CREATING AND DEPLOYING SECRETS

Create and deploy secret objects to store sensitive values for the environment variables that provide your virtual database with the information that it needs to connect to data sources.

Prerequisites

- You have the login credentials and other information that are required to access the data source.

Procedure

1. Create a secrets file to contain the credentials for your data source, and save it locally as a **.yml** file. For example,

Sample secrets.yml file

```
apiVersion: v1
kind: Secret
metadata:
  name: postgresql
type: Opaque
stringData:
  database-user: bob
  database-name: sampledb
  database-password: bob_password
```

2. Deploy the secret object on OpenShift.

   a. Log in to OpenShift, and open the project that you want to use for your virtual database. For example,
   oc login --token=<token> --server=https://<server>oc project <projectName>

   b. Run the following command to deploy the secret file:
   oc create -f ./secret.yaml

Additional resources

- For more information about how to use secrets on OpenShift, see Providing sensitive data to pods in the OpenShift documentation.

2.3. SECRET OBJECTS FOR STORING DATA SOURCE INFORMATION

Use secret objects to securely store sensitive information about how your virtual database connects to its data sources.

For OpenShift to open a connection to a data source on behalf your virtual database application, it must pass login credentials and other information to the data source. Typically, to maintain security administrators limit access to database credentials, and do not expose credentials to developers directly. To enable developers indirect access to credentials, it's possible to deploy *secret objects* on OpenShift to securely store and pass credentials.

Use secrets in combination with environment variables. Rather than specify static values directly in your configuration files, you can configure OpenShift to retrieve values for environment variables from secret objects. When a key in an environment variable refers to a secret object, to obtain the key value, OpenShift examines the secret to find a token that has a name that matches the key name. It extracts the token value and then passes it to the environment variable.

For example, the following environment variable is set to retrieve the value for the **database-user** key from a secret object that has the name **postgresql**.

```
- name: SPRING_DATASOURCE_SAMPLEDB_USERNAME
  valueFrom:
    secretKeyRef:
      name: postgresql
      key: database-user
```

When the data virtualization service needs to retrieve the value for the preceding environment variable, it accesses the secret object with the name **postgresql** and reads the value of the **database-user** key.

In the following secret object, the **database-user** token is assigned the value **bob**. OpenShift passes that value to the environment variable.

Sample secrets.yml file

```
apiVersion: v1
kind: Secret
metadata:
  name: postgresql
type: Opaque
stringData:
  database-user: bob
```

2.4. SPECIFYING PROJECT DEPENDENCIES IN THE POM.XML FILE

To provide your Java shell project with the details that are required to build the project, edit the **pom.xml** file to specify project dependencies, plugins to run, and build profiles.

Some properties in **pom.xml** are common to all data source types. For example, the Teiid Spring Boot starter and the Spring Boot Maven plugin are required to connect to both a postgreSQL database and a Salesforce database. Other properties, such as the drivers that a data source requires, are specific to individual data sources.

Use the sample **pom.xml** file in the Teiid OpenShift repository as the basis for your own file. The sample file contains settings for a postgreSQL database, but the settings in the **<dependencyManagement>**, **<build>**, and **<profiles>** elements apply to any data source. The build resource must be set to the Spring Boot Maven plugin. This plugin, **spring-boot-maven-plugin**, converts the virtual database schema project into a Spring Boot executable Uber JAR file that becomes the basis for an OpenShift container image. The OpenShift profile element must be set to use the Fabric8 Maven plugin (**<artifactId>fabric8-maven-plugin</artifactId>**), which helps to build a container image from the executable JAR, and optionally deploy it into OpenShift.

Modify the values in the **<dependencies>** element as needed to enable connectivity to the data sources that you want to use.

NOTE

Driver modules for most databases are included in the Red Hat Maven repository. At build time, the drivers are downloaded automatically based on the dependency statements in **pom.xml**. For some proprietary data sources, drivers might not be publicly available. If a driver is not available in the repository, download the driver from a third-party site, and deploy it to your local Maven repository.

Prerequisites

- You have a Java shell project for your virtual database application.
- You are familiar with editing Maven **pom.xml** files.
- Download the sample **pom.xml** file from the Teiid OpenShift repository.
- If the driver for your database is not available from the public Maven repository, you have downloaded the driver and deployed it to your local Maven repository.

Procedure

1. Replace the default **pom.xml** file that you created in your Java shell project with the file that you download from the Teiid OpenShift repository.
2. Edit the **pom.xml** to specify the name of the OpenShift project in the **<fabric8 namespace>** element.
3. Set the value of the properties version element to the data virtualization version that you are using. For example,

```
<properties>
    <version.teiid.spring.boot>${version.teiid.spring-boot} </version.teiid.spring.boot>
</properties>
```

   ${version.teiid.spring-boot} represents the build version of the code that is available in the Maven repository. Substitute the value of the build version for the product that you are working with.
4. Specify your data source dependencies in the **<dependencies>** element. If you want to connect a postgreSQL database, you can use the values in the **<dependencies>** element as they are.

Additional resources

- For information about changes that you must make to the **<dependencies>** element of the **pom.xml** file to support Google Sheets or Salesforce, see Section 2.13, "Connection settings for other data sources".

2.5. DEFINING THE STRUCTURE FOR VIRTUAL DATABASES IN A DDL FILE

After you complete changes to the **pom.xml** file, you're ready to define the structure of your virtual database. You define a virtual database through a text-based DDL file. You can supply an existing DDL file, if you have one, or you can create one.

NOTE

If you have a **.vdb** or **.xml** file from an earlier data-virtualization product that you want to reuse, you must convert the file into DDL format before you can use it. For more information about how to convert legacy virtual database files to DDL format, see Chapter 4, *Migrating legacy virtual database files to DDL format* .

If you want to create your own DDL file you can use the sample DDL file in the Teiid OpenShift repository as a guide. But be mindful that the entries in your DDL file are unique to your environment and must include settings that are specific to the data sources that you use.

The design of a DDL file can be complex and is beyond the scope of this documentation. For more information about using SQL in DDL files that support data virtualization, see the Teiid Reference Guide.

Prerequisites

- You have a DDL file for the virtual database that you want to create.

- You know how to write SQL code and you are familiar with creating DDL files.

Procedure

1. Add your DDL file to the **src/main/resources** directory. You can create a new file from scratch, modify the sample DDL file, or use an existing file.

2. Use SQL statements to specify the structural elements of your virtual database.

2.6. DDL FILES

DDL files contain SQL commands that describe and define the structure of the virtual database, such as its views or schema, procedures, functions, tables, records, columns, and other fields.

When you are ready to build your virtual database, the data virtualization service reads the information in the DDL file and uses that information to generate the virtual database image.

Structures that you define in the DDL file are converted to relational database tables. Translators in the data virtualization service import and convert data from your data sources, and use the data to create views in the virtual database.

A typical DDL file define database elements by using statements similar to those in the following list:

- The name of the virtual database. For example:

```
CREATE DATABASE customer OPTIONS (ANNOTATION 'Customer VDB');
USE DATABASE customer;
```

- The name of the translator, or foreign data wrapper, that is needed to interpret data from the data source. For example,

```
CREATE FOREIGN DATA WRAPPER postgresql;
```

- The name of the data source server, and the name of the resource adapter that provides the connections details for the external data source. For example,

```
CREATE SERVER sampledb TYPE 'NONE' FOREIGN DATA WRAPPER postgresql
OPTIONS ("resource-name" 'sampledb');
```

NOTE: Some of the other files that you use to configure your virtual database depend on the name of the data source that is defined in the **CREATE SERVER** statement of the DDL file. For example, the bean name and properties prefix in the **Datasources.java** file, the properties in **application.properties** file, and the environment variables in the **deploymentconfig.yml** file all include the data source name. To wire the various configuration files together, it's important to use the data source name consistently.

- The names of the foreign schema from which you want to import metadata, and the virtual schema into which you want to import that metadata. For example:

```
CREATE SCHEMA accounts SERVER sampledb;
```

```
CREATE VIRTUAL SCHEMA portfolio;

SET SCHEMA accounts;
IMPORT FOREIGN SCHEMA public FROM SERVER sampledb INTO accounts
OPTIONS("importer.useFullSchemaName" 'false');
```

- Views in the virtual database and the mapping between data in the data source and in the virtual database view. For example,

```
CREATE VIEW CustomerZip(id bigint PRIMARY KEY, name string, ssn string, zip string) AS
  SELECT c.ID as id, c.NAME as name, c.SSN as ssn, a.ZIP as zip
  FROM accounts.CUSTOMER c LEFT OUTER JOIN accounts.ADDRESS a
  ON c.ID = a.CUSTOMER_ID;
```

2.7. ADDING JAVA APPLICATION AND CLASS FILES

In the application library of your virtual database project, create a Java application file, **Application.java**, and a data source class file, **DataSources.java**.

Application.java is the main Spring Boot application file that bootstraps the virtual database application. **DataSources.java** adds @Bean methods for each data source that you want to use as an input to your virtual database. The Java class serves as a resource adapter, which provides access to the data source.

When the virtual database starts, the virtualization service reads the metadata and generates an internal model from it. The service then uses that model to read and write to the virtual database. Entities within a data source become available in the virtual database for users to access as tables. For example, if you use a Salesforce database as a data source, then **SObjects** in Salesforce become available as tables in the virtual database.

Prerequisites

- You have Java 11 or greater installed.
- You have a Java shell project that you generated with Maven.

Procedure

1. Create the following **Application.java** file in your Java class folder (for example, **src/main/java/com/example**).

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

You can remove the default **app.java** file that Maven generates when you create the project.

2. Create a **Datasources.java** file in the class folder, and add a bean method for each data source that you want to connect to your virtual database. For an example of a **Datasources.java** file that is designed to work with a postgreSQL database, see the Section 2.8, "Sample Datasources.java file".

2.8. SAMPLE DATASOURCES.JAVA FILE

The **Datasources.java** file adds a class to represent a connection to a data source. The file also establishes a prefix in the **ConfigurationProperties** argument (**spring.datasource.sampledb**). This prefix must be used in the names of data source properties that you specify in the **application.properties** file.

You can define multiple data sources in **Datasources.java** by adding multiple classes, each with its own prefix designation. In each case you must add corresponding entries to the DDL file and to the properties or deployment configuration files.

To associate the Java bean with the data source that is defined in your DDL file, the bean name must be the same as the name of the **SERVER** and **resource-name** properties in the DDL file. For example, the following sample file establishes a connection to a PostgreSQL database called **sampledb**, which is the name that is assigned in the DDL file to the data source **SERVER** object and to its **resource-name** definition.

```
package com.example;

import javax.sql.DataSource;

import org.springframework.boot.jdbc.DataSourceBuilder;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DataSources {

    @ConfigurationProperties(prefix = "spring.datasource.sampledb") (1)
    @Bean
    public DataSource sampledb() {  (2)
        return DataSourceBuilder.create().build();
    }

}
```

1. The prefix must match the prefix that you assign to properties that you define in the **application.properties** file.

2. The name **sampledb** in the prefix definition and in the method name must match the name in the **SERVER** and **resource-name** objects that are defined in the virtual database DDL file. The Spring Boot framework automatically associates the names of methods in the **Datasources.java** file with the names of data sources in the DDL file.

NOTE

The preceding sample file is designed to work with a postgreSQL database. For information about how to adapt the file for use with other data sources, see Section 2.13, "Connection settings for other data sources".

2.9. SPECIFYING APPLICATION PROPERTIES

You define static properties for your virtual database application in an **application.properties** file in the **/src/main/resource** directory. Static properties are configuration settings that remain constant across different environments. After you deploy a virtual database on OpenShift, any modifications that you make to the **application.properties** file are not effective unless you rebuild and redeploy your virtual database.

At minimum the **application.properties** file must contain a value for the **teiid.vdb-file** property, which names the DDL file that defines the structure of the the virtual database. For example, **teiid.vdb-file=customer-vdb.ddl**.

You can also use the **application.properties** file to define other properties, such as data source properties, including their names, their drivers, and the URLs, user names, and passwords that are required to connect to them. Using static properties to assign values for these can be useful in a test environment. But if you deploy your virtual database in multiple OpenShift environments, it's best to use environment variables to dynamically assign unique values for each environment. For more information about using environment variables in your virtual database configuration, see Section 2.12, "Setting the deployment configuration".

If you define data source properties in the **application.properties** file, you must prefix the configuration properties string that you specified in the **Datasources.java** file. The prefix establishes a connection between the properties and the Java class. For example, if you establish the configuration properties prefix **spring.datasource.sampledb** in the **Datasources.java** file, then you must precede the names of the properties that you define in your **application.properties** file with that string. For example,

```
spring.datasource.sampledb.username=<username>
spring.datasource.sampledb.password=<password>
```

Prerequisites

- You have a DDL file in **/src/main/resources** that defines your virtual database structure.
- You have a **Datasources.java** file in your Java class folder that specifies an application prefix.

Procedure

1. Add the file **application.properties** to the **src/main/resources** folder of your Java project.
2. In the file, add the property **teiid.vdb-file** and set its value to the name of the DDL file in the **src/main/resources** folder of your Java project, for example, **customer-vdb.ddl**.
3. (Optional) Add properties to specify connection information for your data source, such as its name, URL, login credentials, and drivers. For an example of an **application.properties** file for a postgreSQL data source, see Section 2.10, "Sample application.properties file"

2.10. SAMPLE APPLICATION.PROPERTIES FILE

The following **application.properties** file includes settings for connecting to a postgreSQL database. You might use a file similar to this for testing purposes, but for an OpenShift deployment, it's best to specify data source properties in a **deploymentconfig.yml** file.

You can specify source-specfic properties for other types of data sources. For more information, see Section 2.13, "Connection settings for other data sources".

```
spring.datasource.sampledb.jdbc-url=jdbc:postgresql://localhost/sampledb 1 2
spring.datasource.sampledb.username=user 3
spring.datasource.sampledb.password=user
spring.datasource.sampledb.driver-class-name=org.postgresql.Driver 4
spring.datasource.sampledb.platform=sampledb 5

# VDB location
teiid.vdb-file=customer-vdb.ddl 6

# true to allow JDBC connections
teiid.jdbc-enable=true

#logging.level.org.teiid=DEBUG
```

1. The JDBC URL that the virtual database uses to connect to a local postgreSQL database as its data source.
2. The prefix that is used in each of these properties matches the prefix that is defined in the **Datasources.java** file.
3. The user name and password values listed here are displayed in plain text. To secure these credentials in a production deployment on OpenShift, use environment variables in a deployment configuration file to reference the secret object that defines these values.
4. The driver that is required to connect to the data source. This driver is defined in the **pom.xml** file.
5. The name of the data source.
6. The name of the DDL file.

2.11. DEPLOYMENT CONFIGURATION FILES (DEPLOYMENTCONFIG.YML)

A deployment configuration file stores settings that govern how the Fabric8 Maven plugin builds and deploys the container image for your virtual database.

The **deploymentconfig.yml** file can also define environment variables for the properties that are required to configure data sources for your virtual databases. The environment variables that you define in the **deploymentconfig.yml** file map to properties in the **application.properties** file. But unlike settings in the properties file, the settings that you define in the **deploymentconfig.yml** file are dynamic. That is, if you add an environment variable or change its value, you do not have to rebuild the virtual database service to put the change into effect.

Environment variables and their corresponding properties have similar names, but they are formatted differently. Separators in the property names are converted from dots or dashes to underscores, and alphabetic characters are converted to uppercase.

Table 2.1. Format of properties versus environment variables

Property in **application.properties**	Environment variable in **deploymentconfig.yml**	
spring.datasource.sampledb.jdbc-url	SPRING_DATASOURCE_SAMPLEDB_JDBCURL	

Because you commonly deploy virtual databases to multiple OpenShift environments, for example a staging and a production environment, you typically specify different data source properties in each environment. For example, the login credentials for accessing a data source in your staging environment probably differ from the credentials that you need to access the data source in the production environment. To define unique values in each environment, you can use environment variables.

The environment variables in **deploymentconfig.yml** replace any static properties that you might set in the **application.properties** file. If a property is defined in both files, the value in the **deploymentconfig.yml** file takes precedence.

You use a single version of the file across environments and use secret objects to isolate the unique details of each environment. Instead of specifying static values for environment variables directly in the file, you can store the values for each deployment environment in secret objects that are unique to each environment. The value of each environment variable in the file contains only a key reference, which specifies the name of a secret object, and the name of a token in the secret. The token stores the actual value. At runtime, environment variables retrieve their values from the tokens.

By using secrets to store the values of your environment variables, you can use a single version of the **deploymentconfig.yml** across environments. The secret objects that you deploy in each environment must have the same name, but each secret object contains token values that are specific to its environment.

Additional resources

- For more information about using secrets, see Section 2.3, "Secret objects for storing data source information".
- For information about adding a **deploymentconfig.yml** file, see Section 2.12, "Setting the deployment configuration".

2.12. SETTING THE DEPLOYMENT CONFIGURATION

You set the deployment configuration by editing a **deploymentconfiguration.yml** file. You can define environment variables in the file for each data source that the virtual database uses.

Prerequisites

- You have a copy of the sample **deploymentconfiguration.yml** file from the Teiid OpenShift repository.
- You have information about the connection settings for your data sources.
- If you want to use secrets to store values for your environment variables, you have information about the name of the secret for your virtual database and the names of the tokens that you want to refer to in your environment variables.

Procedure

1. In the **/src/main/fabric8** folder of your Java project, create a **deploymentconfiguration.yml** file.

2. Add environment variables and other settings that are consistent with your environment.

Additional resources

- For more information about the **deploymentconfiguration.yml** file, see Section 2.11, "Deployment configuration files (**deploymentconfig.yml**)".
- For more information about secrets, see Section 2.3, "Secret objects for storing data source information".
- For information about deployment configuration settings for other data sources, see Section 2.13, "Connection settings for other data sources" .

2.13. CONNECTION SETTINGS FOR OTHER DATA SOURCES

To enable a virtual database to connect to a data source, you must provide connection details such as the name of the data source, the driver to use, the user credentials and so forth. You specify these settings across several files.

The sample files in the Teiid OpenShift repository, or elsewhere in this documentation provide configuration information that is consistent with using a postgreSQL database as the data source for your virtual database. If you want to a different data source, you must modify settings in the postgreSQL versions of the following files:

pom.xml file

Specifies the dependencies for a data source, such as the drivers that are required to connect to the data source. At build time, if the driver for your database type is publically available, Teiid Spring Boot downloads the required drivers automatically.

application.properties file

Defines static application properties that cannot be changed after you deploy the application to OpenShift, for example the name of the DDL file.

Deploymentconfig.yml

Defines application properties through dynamic environment variables so that you can specify values to correspond to a particular deployment environment

Datasources.java

Specifies a Java class to represent the connection to the data source. The service name that you specify in the annotation and as the name of the method must match exactly the name of the SERVER that is listed in the DDL file.

DDL file

Defines the virtual database structure including specific mapping from the source schema to the virtual schema.

When the virtual database service starts, the data virtualization service scans the application's packages for dependency annotations and uses the information to build the metadata to create the virtual database and deploy it to the OpenShift server.

Prerequisites

- You have reviewed the postgreSQL versions of the files in the preceding list.

2.13.1. Settings to connect to Salesforce as a data source

If you want to use a Salesforce database as a data source for your virtual database, you must add some source-specific settings. You can use the postgreSQL files in the Teiid OpenShift repository as a starting point, but you'll have to modify the files to adapt them for use with Salesforce.

The following files contain information that you must modify to use a Salesforce database as a data source:

- **pom.xml**
- **application.properties**
- **Deploymentconfig.yml**
- **Datasources.java**
- DDL file

Pom.xml settings for using Salesforce as a data source

To support Salesforce as a data source, you must add the following dependencies in the **pom.xml** file:

- teiid-spring-boot-starter
- spring-data-salesforce

For example:

```
<dependency>
  <groupId>org.teiid</groupId>
  <artifactId>teiid-spring-boot-starter</artifactId>
</dependency>
<dependency>
   <groupId>org.teiid.spring</groupId>
   <artifactId>spring-data-salesforce</artifactId>
</dependency>
```

Application.properties settings to add to use Salesforce as a data source

You can specify values in **application.properties** to configure Salesforce as a data source, as shown in the following table. Each of the properties uses the prefix **spring.teiid.data.salesforce**

Property Name	Description	Default
url	Login URL	https://login.salesforce.com/services/Soap/u/45.0
requestTimeout	Request timeout	Integer.MAX_VALUE
connectTimeout	Connection timeout	Integer.MAX_VALUE
password	**Password**	**N/A**

Property Name	Description	Default
pollingInterval	Polling interval for bulk results	500
clientId	OAuth2 client ID	N/A
clientSecret	OAuth2 client secret	N/A
refreshToken	OAuth2 refresh token	N/A
userName	User name	N/A
password	**Password**	**N/A**

Datasources.java file for connecting to Salesforce as a data source

The **Datasources.java** file creates a class that Teiid can recognize as a Salesforce data source. The class acts as a resource adapter to enable the virtual database to access information in the data source.

```
package org.teiid.spring.example;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.teiid.spring.data.salesforce.SalesforceConfiguration;
import org.teiid.spring.data.salesforce.SalesforceConnectionFactory;

@Configuration
public class DataSources {

    @Bean
    public SalesforceConnectionFactory accounts(SalesforceConfiguration config) {
        return new SalesforceConnectionFactory(config);
    }

    @Bean
    @ConfigurationProperties("spring.teiid.data.salesforce")
    public SalesforceConfiguration salesforceConfig() {
        return new SalesforceConfiguration();
    }
}
```

The preceding class defines an **accounts** bean. When you create the virtual database, the data virtualization service recognizes the class as a data source, reads its metadata, and generates an internal model from it. The data virtualization service can then read from and write to it.

DDL file changes to connect a virtual database to Salesforce as a data source

Update the name of the **SERVER** object and the **FOREIGN DATA WRAPPER** to reflect the data source. For example, assign the following values to objects in the DDL file:

- Set the name of the **SERVER** object to **salesforce**

- Set the name of the **FOREIGN DATA WRAPPER** to **salesforce**

2.13.2. Settings to connect to Google Sheets as a data source

If you want to use a Google Sheets as a data source for your virtual database, you must add some source-specific settings. You can use the postgreSQL files in the Teiid OpenShift repository as a starting point, but you'll have to modify the files to adapt them for use with Google Sheets.

The following files contain information that you must modify to use Google Sheets as a data source:

- **pom.xml**
- **application.properties**
- **Deploymentconfig.yml**
- **Datasources.java**
- DDL file

Pom.xml settings for using Google Sheets as a data source

To support Google Sheets as a data source, you must add the following dependencies in the **pom.xml** file:

- teiid-spring-boot-starter
- spring-data-google

For example:

```
<dependency>
  <groupId>org.teiid</groupId>
  <artifactId>teiid-spring-boot-starter</artifactId>
</dependency>
<dependency>
   <groupId>org.teiid.spring</groupId>
   <artifactId>spring-data-google</artifactId>
</dependency>
```

Application.properties settings to add to use Google Sheets as a data source.

You can specify values in **application.properties** to configure Google Sheets as a data source, as shown in the following table. Each of the properties uses the prefix **spring.teiid.data.google-sheets**

Property Name	Description	Default
spread-sheet-name	Name of the Google Spreadsheet	N/A
refresh-token	**OAuth2 refresh token**	**N/A**

Property Name	Description	Default
spread-sheet-id	Spreadsheet ID	Sheet Id, Look in URL of the spreadsheet. For more info see https://developers.google.com/sheets/api/guides/concepts#spreadsheet_id
client-id	Client ID	OAuth2 client ID
client-secret	Client secret	OAuth2 client secret
refresh-token	**OAuth2 refresh token**	**N/A**

Datasources.java file for connecting to Google Sheets as a data source

The **Datasources.java** file creates a class that Teiid can recognize as a Google Sheets data source. The class acts as a resource adapter to enable the virtual database to access information in the data source.

```
@Configuration
public class DataSources {

    @Bean
    public SpreadsheetConnectionFactory accounts(SpreadSheetConfiguration config) {
        return new SpreadsheetConnectionFactory(config);
    }

    @Bean
    @ConfigurationProperties("spring.teiid.data.google.sheets")
    public SpreadSheetConfiguration sheetsConfig() {
        return new SpreadSheetConfiguration();
    }
}
```

The preceding class defines an **accounts** bean. When you create the virtual database, the data virtualization service recognizes the class as a data source, reads its metadata, and generates an internal model from it. The data virtualization service read from and write to it.

DDL file changes to connect a virtual database to Google Sheets as a data source

Update the name of the **SERVER** object and the **FOREIGN DATA WRAPPER** to reflect the data source. For example, assign the following values to objects in the DDL file:

- Set the name of the **SERVER** object to **google-spreadsheet**.
- Set the name of the **FOREIGN DATA WRAPPER** to **google-spreadsheet**.

2.14. MAKING VIRTUAL DATABASES AVAILABLE TO API CONSUMERS

To enable API consumers to access the virtual database, define services and routes for JDBC or OData protocols for the virtual database service on OpenShift.

After you define a protocol service for a virtual database, any client in the same OpenShift cluster can access the virtual database. Only applications that are in the same cluster have access. Remote clients do not have access.

To enable OData access for remote clients, you must define an OData route to the virtual database service.

External applications JDBC clients do not use routes to access virtual database services. Instead, JDBC clients depend on the OpenShift load balancer service to allocate external IP addresses that external clients can use to access services in the cluster.

The following table lists the configuration files that you add in **/src/main/fabric8** to configure services and routes for a virtual database.

Table 2.2. Fabric8 configuration files

Name	Purpose	Sample file
Deployment configuration	Controls overall deployment	deploymentconfig.yml
JDBC service configuration	Specifies JDBC service	jdbc-svc.yml
OData service configuation	Specifies OData service	odata-svc.yml
OData route configuration	Specifies OData route	odata-route.yml

After you define a protocol service for a virtual database, clients that share an OpenShift cluster with the virtual database can access it. Remote clients do not have access. To enable OData access for remote clients, you must define an OData route to the virtual database service.

External applications JDBC clients do not use routes to access virtual database services. Instead, JDBC clients depend on the OpenShift load balancer service to allocate external IP addresses that external clients can use to access services in the cluster.

You configure services and routes by adding configuration files in **/src/main/fabric8**. Download the sample Fabric8 configuration files from the Teiid OpenShift repository. The following table list the configuration files to add in **/src/main/fabric8** to configure services and routes for the virtual database.

If you choose not to create a service for one of the protocols, or to expose a route for that protocol, omit the corresponding file from the **/fabric8** directory.

The service and route configuration files specify default values for port numbers and timeout values. Unless you have a specific reason for modifying those settings, you can retain the default values.

Prerequisites

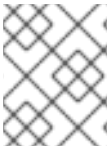
- You downloaded the sample configuration files that you need from the Teiid OpenShift repository.

2.14.1. Configuring access for OData clients

The relational model of the data in a virtual database is automatically mapped to JSON or XML to allow applications to consume the data through OData APIs. However, if you want the virtual database to be available to OData clients, you must explicitly define an OData service and route. The default settings in

the **odata-svc.yml** and **odata-route.yml** files that are available from the Teiid OpenShift sample repository are configured to enable an OData service and route.

Do not edit instances of the variable **${project.artifactId}** that appear in the configuration files. At build time, these variables are replaced automatically with information from elements in the **pom.xml** file.



NOTE

OData routes are created automatically for virtual databases that you create in Fuse Online.

Prerequisites

- You have the sample template **odata-svc.yml** and **odata-route.yml** files from the Teiid OpenShift repository.
- You have completed the configuration tasks that are summarized in Chapter 2, *Creating virtual databases*.

Procedure

1. Add the following dependency in your **pom.xml** file:

```
<dependency>
  <groupId>org.teiid</groupId>
  <artifactId>spring-odata</artifactId>
</dependency>
```

2. In the **/src/main/fabric8** folder of your Java project, add an **odata-svc.yml** file to create an OData service).
3. If you want to make the virtual database available to OData clients outside of the OpenShift cluster, add an **odata-route.yml** file to create and OData route.

2.14.2. Configuring access for JDBC clients

To enable JDBC client applications to access your virtual database, you must configure a JDBC service. After the JDBC service is enabled, OpenShift applications that share the cluster with the virtual database can access the database over JDBC. Third-party client applications that are outside of the OpenShift cluster have no JDBC access to the virtual database.

To provide JDBC access to external clients, you do not create a JDBC route as you do to enable OData access to external clients. Rather, you must configure an OpenShift load balancer service to configure ingress cluster traffic. After that you must provide external applications with the IP address that the load balancer service assigns to the virtual database.

Do not edit instances of the variable **${project.artifactId}** that appear in the configuration files. At build time, these variables are replaced automatically with information from elements in the **pom.xml** file.

Prerequisites

- You have the sample **jdbc-svc.yml** file from the Teiid OpenShift repository.
- You have completed the configuration tasks that are summarized in Chapter 2, *Creating virtual databases*.

Procedure

1. To create a JDBC service, add the file **jdbc-svc.yml** to the **/src/main/fabric8** folder of your Java project.

NOTE

If you add the **jdbc-svc.yml** file from the Teiid OpenShift sample repository, the JDBC service is enabled by default.

1. Create a file with the name **ingress** and add the following contents to it:

```
apiVersion: v1
kind: Service
metadata:
  name: rdbms-example-ingress
spec:
  ports:
  - name: teiid
    port: 31000
  type: LoadBalancer
  selector:
    app: rdbms-example
  sessionAffinity: ClientIP
```

2. Log in to OpenShift and run the following command to deploy the file to OpenShift:

```
$oc create -f -ingress
```

3. Run the following command to determine the IP port:

```
$oc get svc rdbms-example-ingress
```

4. Share the port number that is returned with your API clients.

Additional resources

- For more information about Configuring ingress cluster traffic using a load balancer , see the OpenShift documentation.

2.14.3. Identifying the OData endpoint of a virtual database

After you deploy a virtual database, you can share the OData URL with application developers so that they can use REST APIs to query and retrieve data from the virtual database.

NOTE

After you create a virtual database in Fuse Online, its OData route is exposed automatically on the **Data Virtualizations** page.

To retrieve the OData URL for virtual databases that you create using the Spring Boot Maven plugin, you run an OpenShift command. You then append **/odata** to the URL that command returns to establish the API endpoint for the service. .

NOTE

In this Technology Preview release, there is no relationship between virtual databases that you create by running the Maven Spring Boot plugin and those that you create in Fuse Online. As a result, the **Data virtualization** page in Fuse Online displays only the virtual databases does not show virtual databases that you build and create outside of Fuse Online.

Prerequisites

- In Fuse Online, the **ExposeVia3Scale** property is set to **true** during the Fuse Online installation. This setting does not apply to virtual databases that you create with the Maven Spring Boot plugin.

Procedure

- Log in to OpenShift and run the following command to obtain the OData URL for the service: **oc describe route <virtual-database-service-name>**

  For example, **oc describe route rdbms-example**

2.15. BUILDING A DATA VIRTUALIZATION PROJECT

After you complete the configuration tasks for your virtual database, you are ready to run Maven to build the project.

The Spring Boot Maven plugin creates a self-contained Uber JAR that includes all of the application code and dependencies in a single JAR file. The resulting JAR file serves as the basis for an OpenShift image. The OpenShift profile includes a Fabric8 Maven plugin that compiles the current build with the Uber JAR to generate an image that you can deploy to OpenShift.

After you make certain changes to the virtual database configuration, for example adding protocol services or routes for the virtual database, you must re-compile the project to update the deployed image.

Prerequisites

- You have completed all of the configuration tasks listed in Chapter 2, *Creating virtual databases*.

Procedure

- Log in to OpenShift and run the following command:

```
mvn clean install -Popenshift -Dfabric8.namespace=`oc project -q`
```

CHAPTER 3. SECURING ODATA APIS FOR A VIRTUAL DATABASE BY USING 3SCALE AND RH-SSO

You can integrate data virtualization with 3scale and Red Hat's single sign-on technology to apply advanced authorization and authentication controls to the OData endpoints for your virtual database services. By using 3Scale as a gateway to your API, you ensure that only authorized users have access, and you can control the level of access a user has based on who they are (role-based access).

By default, after you create a virtual database, the OData interface to it is discoverable by the 3Scale API management system, as long as 3Scale system is defined to same cluster and namespace. Using 3scale's API management features, you can control access to the OData API, and track usage. By further securing access to the API through SSO, you can define user roles and implement role-based access to the API endpoints. After you complete the configuration, you can control access in the virtual database at the level of the view, column, or data source.

The Red Hat SSO technology uses OpenID-Connect as the authentication mechanism to secure the API, and uses OAuth2 as the authorization mechanism. When a user logs in, 3scale negotiates authentication with the Red Hat SSO package. If the authentication succeeds, 3scale passes a security token to the OData API for verification. The OData API then reads permissions from the token and applies them to the data roles that are defined for the virtual database.

Prerequisites

- You have installed and configured Red Hat's single sign-on technology. For more information about how to configure SSO to secure OData APIs for a virtual database, see the Teiid OpenShift documentation.
- You have 3scale API Management installed in the OpenShift cluster that hosts your virtual database.
- You have configured integration between 3scale and SSO. For more information, see Configuring Red Hat Single Sign-On integration in Using the Developer Portal.
  - You have configured the Red Hat SSO technology for OpenID Connect.
  - You have created SSO security realms.
  - You have specified an Admin client ID, protocol, and access type.
  - You have assigned the *realm-management* and *manage-clients* roles.
  - You created API users and specified credentials.
  - You configured 3scale to use OpenID-Connect as the authentication mechanism and OAuth2 as the authorization mechanism.

3.1. UPDATING THE VIRTUAL DATABASE CONFIGURATION

After you finish setting up single sign-on in OpenShift, you must modify virtual database settings before you can use SSO to secure your OData APIs. You must update the following artifacts:

- **pom.xml**
- **application.properties**

- Configuration map
- **deploymentconfig.yml**
- DDL file

After you update the virtual database configuration, update the 3scale configuration to complete the integration.

Prerequisites

- You completed the SSO configuration.

3.1.1. Adding SSO dependencies to pom.xml

Prerequisites

- You completed the configuration to integrate 3scale with SSO.

Procedure

- Edit the **pom.xml** file to add the following SSO dependencies:

```
<dependency>
  <groupId>org.teiid</groupId>
  <artifactId>spring-keycloak</artifactId>
</dependency>
```

3.1.2. Adding SSO settings to application.properties

Update properties in the **application.properties** file to support integration with SSO.

Procedure

1. Log in to the Red Hat Single Sign-On Admin Console to find the values for the following authentication properties:
   - keycloak.realm
   - keycloak.auth-server-url
   - keycloak.resource
   - keycloak.ssl-required
   - keycloak.public-client
   - keycloak.principal-attribute
2. Open the **application.properties** file from **src/main/resources/** and add values for the preceding properties. For example:

```
keycloak.realm = 3scale-sso
keycloak.auth-server-url = http://keycloak-staging.dev.openshiftapps.com/auth
```

```
keycloak.resource = di
keycloak.ssl-required = external
keycloak.public-client = true
keycloak.principal-attribute=preferred_username
```

3.1.3. Creating a configuration map

In **application.properties** you assign static properties that do not change between deployment environments. For settings that can change between environments, such as authentication URLs and credentials, you might want to apply different values depending on the environment. To make it easy to apply different values for environment-specific settings, you can specify their values as properties in a configuration map. In each environment in which you deploy a virtual database, use a configuration map with the same name, but vary the map contents (the authentication URL and secret) to match the environment. Then, in each environment where you deploy the virtual database, the virtual database retrieves its version of the authentication values from the map.

You run an OpenShift command to create a configuration map and add values for the following properties:

- **keycloak.auth-server-url**
- **keycloak.credentials.secret**

Procedure

1. Run the following command:

```
oc create configmap my-config --from-literal=keycloak.auth-server-url=http://<keycloakURL>
--from-literal=keycloak.credentials.secret=<secretValue>
```

For example,

```
oc create configmap my-config --from-literal=keycloak.auth-server-url=http://keycloak-
staging.dev.example.com/auth --from-literal=keycloak.credentials.secret=xxxxxxxx-xxxx-
xxxx-xxxx-xxxxxxxxxxxx
```

The preceding command creates a configuration map in OpenShift with the name *my-config*, which you can then reference from the deployment configuration file (**deploymentconfig.yml**).

3.1.4. Updating SSO environment variables in the deploymentconfig.yml file

In the **deploymentconfig.yml** file you can specify environment variables to apply different settings in each environment where you deploy a virtual database. For example, if you are deploying a virtual database to both a staging environment and a production environment, you can specify different environment variables in each to customize the settings for that environment.

To support SSO, you must add environment variables to the **deploymentconfig.yml** file.

Procedure

- From **src/main/fabric8**, open the **deploymentconfig.yml** file and add the following environment variables:

```
- name: KEYCLOAK_AUTHSERVERURL
```

```
    valueFrom:
      configMapKeyRef:
        name: <config-name>
        key: keycloak.auth-server-url
  - name: KEYCLOAK_CREDENTIALS_SECRET
    valueFrom:
      configMapKeyRef:
        name: <config-name>
        key: keycloak.credentials.secret
```

For example,

```
- name: KEYCLOAK_AUTHSERVERURL
  valueFrom:
    configMapKeyRef:
      name: my-config
      key: keycloak.auth-server-url
- name: KEYCLOAK_CREDENTIALS_SECRET
  valueFrom:
    configMapKeyRef:
      name: my-config
      key: keycloak.credentials.secret
```

If you deploy a virtual database in multiple environments that use different realm or client settings, be sure to specify unique realm and client properties in the SSO configuration that you establish for each environment.

3.1.5. Defining user roles in the DDL file

To apply unique data roles to different users, you must define those roles in the DDL file.

Procedure

1. Add the following lines to the DDL file in **src/main/resources**:

```
CREATE ROLE ReadRole WITH JAAS ROLE ReadRole;
GRANT SELECT ON TABLE "<tableName.fieldName>" TO ReadRole
```

For example,

```
CREATE ROLE ReadRole WITH JAAS ROLE ReadRole;
GRANT SELECT ON TABLE "portfolio.CustomerZip" TO ReadRole
```

In the preceding example, the first line creates a role with the name **ReadRole**. Map this role to a role with the same name that you created in the SSO configuration. The role names can be different, but for simplicity, it's best to use the same name. The second line grants SELECT permission to the **portfolio.CustomerZip** View to users who are assigned the **ReadRole**.

After you make changes to the project to integrate the 3scale and enable SSO, rebuild the virtual database image and deploy it to OpenShift. For information about how to build and deploy the image, see Section 2.15, "Building a data virtualization project" .

3.2. UPDATING THE 3SCALE CONFIGURATION

Prerequisites

- You have configured SSO.
- You have made updates to the virtual database service to support SSO.

Procedure

1. Login to your 3scale admin portal and select the service for which you want to enable OpenId Connect integration with SSO.
2. Click the **APIs** tab, select the virtual database service, and click the **Integration** link.
3. Click **edit integration settings**.
4. In the **Authentication** section, click **OpenID Connect**, and click **Update Service**.
5. From the **Integration & Configuration** page, click the **edit APIcast configuration**.
6. Click **Authentication Settings** to view the authentication options.
7. In the **OpenID Connect Issuer** field, type your SSO client credentials and the URL of your SSO server.
8. Click **Update Staging Environment**
9. Create a new application so that 3scale can synchronize it with SSO.
    a. Create a new Developer account. For more information, see Creating a new developer account and API credentials in the 3scale Admin Portal Guide.
    b. In the Developer account, create an application. For more information, see Set up applications.
    c. Select an application plan for the virtual database API that you want to secure. You can select an existing application plan or create a new one. For information about how to create an application plan, see How to create an application plan in the 3scale Admin Portal Guide.
    d. Record the API credentials (the client secret and client ID). These credentials become available after you configure OpenID Connect. You must supply these credentials to access the API. If no value is displayed for the client secret, click **Add random key** to populate the field.
    e. Specify a redirect URL. You use the redirect URL to test the integration in Postman or other REST clients.
    f. In the Redirect URL field click **Edit**, type a callback link, for exmaple, https://openidconnect.net/callback, and then click **Update**.

CHAPTER 4. MIGRATING LEGACY VIRTUAL DATABASE FILES TO DDL FORMAT

The data virtualization Technology Preview requires that you define the structure of virtual databases in SQL-MED DDL (data definition language) format. By contrast, the structure of legacy Teiid virtual databases, such as those that run on Wildfly, or on the Red Hat JBoss Data Virtualization offering, are defined by using files that are in **.xml** or **.vdb** format.

You can reuse the virtual database designs that you developed for a legacy deployment, but you must first update the format of the files. A migration tool is available to convert your files. After your convert the files you can rebuild the virtual databases as container images and deploy them to OpenShift.

You can use the migration utility in the following two ways:

To validate a VDB file only

Use this method to check whether the utility can a successfully convert a VDB file. The utility converts the VDB file and reports validation errors to the terminal. If there are no validation errors, the utility displays the resulting DDL, but it does not save the converted DDL to a file.

To validate and a VDB file and save it to a DDL file

The file is saved only if there are no validation errors.

The migration tool works on **.xml** files only. Files with a **.vdb** file extension are file archives that contain multiple folders. If you have legacy files in **.vdb** format, use Teiid Designer to export the files to **.xml** format, and then run the migration tool to convert the resulting **.xml** files.

Prerequisites

- You have a legacy virtual database file in **.xml** format.
- You download the **settings.xml** file from the Teiid OpenShift repository. Maven uses the information in the file to run the migration tool.

4.1. VALIDATING A LEGACY VIRTUAL DATABASE XML FILE AND VIEWING IT IN DDL FORMAT

You can run a test conversion on a legacy virtual database to check for validation errors and view the resulting DDL file. When you run the migration tool in this way, the converted DDL file is not saved.

Procedure

1. Open the directory that contains the **settings.xml** file that you downloaded from the Teiid OpenShift repository, and type the following command:

```
$mvn -s settings.xml exec:java -Dvdb={path_to_vdb_xml_file}
```

For example:

```
$mvn -s settings.xml exec:java -Dvdb={rdbms-example/src/main/resources/vdb.xml}
```

The migration tool checks the specified **.xml** file, and reports any validation errors. If there are no validation errors, the migration tool displays a **.ddl** version of the virtual database on the screen.

4.2. CONVERTING A LEGACY VIRTUAL DATABASE XML FILE AND SAVING IT AS A DDL FILE

You can run the migration tool to convert a legacy virtual database file to **.ddl** format, and then save the **.ddl** file to a specified directory. The migration tool checks the **.xml** file that you provide for validation errors. If there are no validation errors, the migration tool converts the file to **.ddl** format and saves it to the file name and directory that you specify.

The migration tool checks the **.xml** file that you provide for validation errors. If there are no validation errors, the migration tool converts the file to `.ddl' format and save it to the filename and directory that you specify.

Procedure

- Open the directory that contains the **settings.xml** file that you downloaded from the Teiid OpenShift repository, and type the following command:

  ```
  $mvn -s settings.xml exec:java -Dvdb={path_to_vdb_xml_file} -Doutput=
  {path_to_save_ddl_file}
  ```

  For example:

  ```
  $mvn -s settings.xml exec:java -Dvdb={rdbms-example/src/main/resources/vdb.xml} -
  Doutput=[rdbms-example/src/main/resources/vdb.ddl]
  ```