Red Hat Integration 2020-Q2

Using Data Virtualization

TECHNOLOGY PREVIEW - User's guide to Data Virtualization

Last Updated: 2020-07-15

Red Hat Integration 2020-Q2 Using Data Virtualization

TECHNOLOGY PREVIEW - User's guide to Data Virtualization

Legal Notice

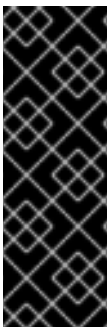
Abstract

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

Data virtualization is a container-native service that provides integrated access to multiple diverse data sources, including relational and noSQL 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 the data were served from a single relational database.



IMPORTANT

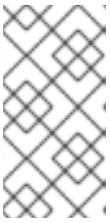
Data virtualization is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming 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/.

The Red Hat data virtualization technology is based on Teiid, the open source data virtualization project. For more information about Teiid, see the Teiid community documentation.

CHAPTER 2. CREATING VIRTUAL DATABASES

To add a virtual database, you must complete the following tasks:

1. Install the Data Virtualization Operator.
2. Design and develop the database.



NOTE

If you want to use a custom TLS certificate to encrypt virtual database traffic, you must obtain and configure the certificate before you deploy the virtual database. For more information, see Section 7.1, “Certificates and data virtualization”.

3. Create a custom resource (CR) file for deploying the database.
4. Deploy the virtual database to OpenShift by running the Data Virtualization Operator with the CR.

You can use the following methods to design a virtual database.

Create a virtual database from a DDL file

Define the entire contents of a virtual database, including the DDL, in a YAML file. For more information, see Chapter 3, *Creating a virtual database by embedding DDL statements in a custom resource (CR)* .

Create a virtual database as a Maven artifact

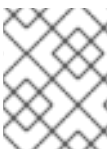
Create a virtual database from one or more DDL files and generate a Maven artifact for deployment For more information, see Chapter 4, *Creating a virtual database as a Maven artifact* .

In each of the methods, you use SQL data definition language (DDL) to specify the structure of the virtual database, and you then configure the data sources that you want the virtual database to read from and write to.

There are advantages and disadvantages to using each method, the runtime virtualizations that any of the methods create have equivalent features. Choose a method based on the complexity of your project and on whether you want to be able to test the virtualization as a standalone component or on OpenShift only.

After you define the virtual database, you use the Data Virtualization Operator to deploy the virtualization from a custom resource (CR). The custom resource that you use to deploy a virtual database varies with the method that you used to design the virtual database. For more information, see Chapter 6, *Running the Data Virtualization Operator to deploy a virtual database* .

After you set up connections to a data source, you can optionally configure authentication to Red Hat SSO to secure the connections, and enable single sign-on.



NOTE

You can also create virtual databases in Fuse Online (Technology Preview). Virtual databases that you create in Fuse Online provide a limited set of features.

Additional resources

- Section 6.1, “Installing the Data Virtualization Operator on OpenShift”.
- Chapter 7, *Securing data*.

2.1. COMPATIBLE DATA SOURCES

You can create virtual databases from a range of different data sources.

For each data source that you configure, you specify a set of properties in a custom resource (CR) YAML file. A *translator* property specifies the name of a component that provides the logic to interpret the commands and data exchanged that pass between the data source and the virtual database. Each data sources uses a specific named translator.

The following table lists the data source types that you can include in a virtual database, and the names of the translators for each data source:

Data source		Translator name
Amazon S3/ Ceph		amazon-s3
Google Sheets		google-spreadsheet
Data Grid (Infinispan)		infinispan-hotrod
MongoDB		mongodb
Relational databases		
	Amazon Athena	amazon-athena or jdbc-ansi
	Amazon Redshift	redshift
	Db2	db2
	Microsoft SQL Server (JDBC)	sqlserver
	MySQL	mysql
	Oracle	oracle
	PostgreSQL	postgresql
	SAP HANA (JDBC)	hana
OData		odata
OData4		odata4
OpenAPI		openapi

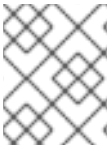
Data source	Translator name
REST	ws
Salesforce	salesforce
SFTP	file
SOAP	soap or ws

2.2. CREATING CUSTOM RESOURCES TO DEPLOY VIRTUALIZATIONS

Before you can use the Data Virtualization Operator to create a virtual database, you must specify properties for the data source in a custom resource (CR) file.

When you run the Data Virtualization Operator, it reads information from the CR that it needs to convert a data virtualization artifact into an image and deploy it to OpenShift.

Properties in the CR specify environment variables that store the credentials that the Operator requires to connect to a data source. You can specify the values directly in the CR, or provide references to an OpenShift *secret* that stores the values. For more information about creating secrets, see Section 7.2, "Using secrets to store data source credentials".



NOTE

Period characters (**.**) are not valid for use in environment variables. When you add variable names to the CR, use underscore characters (_) as separators.

The information that you add to the CR depends on the type of artifact that you created for the virtualization and the location of artifact. You can also supply configuration information in the CR.

NOTE

If you want OpenShift to create an HTTP endpoint for the deployed virtualization, add the property **spec/exposeVia3scale** to the CR, and set its value to **false**. If the value is set to **true** it is assumed that 3scale manages the endpoint, and no HTTP endpoint is created.

Additional resources

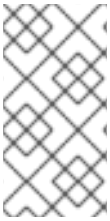
- Section 2.2.2, "Environment variables in custom resources"
- Section 3.1, "Creating a CR to deploy a DDL artifact"
- Section 4.2, "Creating a custom resource (CR) to deploy a Maven artifact"

2.2.1. Configuring an OpenShift load balancer service to enable external JDBC clients to access the virtual database

After you deploy a virtual database, it is automatically available to internal JDBC clients, that is, clients that are installed on the OpenShift cluster that hosts the virtual database. By default, external JDBC

clients are unable to access the virtual database service. To enable external clients to access the virtual database service, you must add an OpenShift load balancer service.

To configure a load balancer for the virtual database, you define an attribute in the custom resource. Afterwards, when you run the Data Virtualization Operator to build and deploy the virtual database, the Operator creates the load balancer service automatically.



NOTE

Although OpenShift typically requires you to create a route to the service that you want to expose, you do not have to create routes for virtual database services that you deploy with the Data Virtualization Operator. When the Operator deploys the virtual database, it automatically exposes the JDBC route to the virtual database service.

Prerequisites

- You have access to an OpenShift cluster that permits you to add a LoadBalancer Ingress Service.
- You have a custom resource (CR) to which you can add the attribute to enable the load balancer service.

Procedure

1. Add a load balancer service for the virtual database by setting the value of **spec.expose** in your virtual database CR to **LoadBalancer**.
   To provide flexibility in exposing other resources in the future, precede the value with a hyphen (**-**) to indicate that it is an element in an array, as in the following example:

```
apiVersion: teiid.io/v1alpha1
kind: VirtualDatabase
metadata:
  name: dv-customer
spec:
  replicas: 1
  expose:
    - LoadBalancer
....
```

2. After you deploy the virtual database, you can run the following command from a terminal window to identify the exposed host and port:

```
oc get svc VDB_NAME-external
```

   For example,

```
oc get svc dv-customer-external
```

   The command returns network information for the service, including the cluster IP address, external host name, and port number and type. For example:

```
NAME                    TYPE           CLUSTER-IP     EXTERNAL-IP                          PORT(S)
AGE
dv-customer-ingress  LoadBalancer  172.30.22.226  ad42f5d8b303045-
```

```
487804948.example.com  3306:30357/TCP  15m
```

Additional resources

- For information about deploying the virtual database, see Section 6.2, “Deploying virtual databases”.
- For more information about configuring an OpenShift load balancer service, see the OpenShift documentation.

2.2.2. Environment variables in custom resources

You set environment variables in the custom resource file to enable your virtual database to connect to data sources.

Because you typically deploy virtual databases to multiple OpenShift environments, such as to a staging and a production environment, you might want to define different data source properties for each environment. For example, the login credentials that you must provide to access a data source in the staging environment are probably different from the credentials that you use to access the data source in the production environment. To define unique values in each environment, you can use environment variables.

The environment variables that you define in a CR replace any static properties that you might set elsewhere. If you define a property in the properties file and in the CR, the value in the CR file takes precedence.

You can combine the use of environment variables and secret objects to specify and protect the unique details for each environment. Instead of specifying static values for environment variables directly in the CR, 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 CR 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 CR across environments. The secret objects that you deploy in each environment must have the same name, but in each environment you assign token values that are specific to the environment.

Additional resources

- For more information about using secrets, see Section 7.2, “Using secrets to store data source credentials”.
- For information about adding a CR file, see Section 2.2, “Creating custom resources to deploy virtualizations”.

CHAPTER 3. CREATING A VIRTUAL DATABASE BY EMBEDDING DDL STATEMENTS IN A CUSTOM RESOURCE (CR)

You can define the structure of a virtual database by adding DDL statements directly within a custom resource file. During deployment, the Operator runs a source-to-image (S2I) build on OpenShift based on the dependencies that it detects in the virtual database artifact. To prevent build failures, ensure that any dependencies that your virtual database requires, such as JDBC driver dependencies, can be found at build time.

Advantages of using DDL in the CR to create a virtual database

- Simple and minimalistic.
- Code and configuration for a virtualization are in a single file. No need to create a separate CR file.
- Easy to manage.

Disadvantages of using DDL in the CR to create a virtual database

- Embedding the DDL for the virtual database in the custom resource (CR) file results in a large file.
- Because the DDL is embedded in the CR YAML file, you cannot version the DDL and other aspects of the configuration independently.
- If you deploy to multiple environments, you must store properties in configuration maps or secrets to make them independent of the custom resource.

Prerequisites

- You have Developer or Administrator access to an OpenShift cluster in which the data virtualization operator is installed.
- You have a compatible data source and the OpenShift cluster can access it.
- The data virtualization operator has access to any Maven repositories that contain build dependencies for the virtual database.
- You have information about the connection settings for your data sources, including login credentials.
- You have a DDL file for the virtual database that you want to create, or you know how to write the SQL code to design the database.

Procedure

- Create a CR text file in YAML format and save it with a .yaml or .yml extension, for example **dv-customer.yaml**
  The following example shows the elements to include in a CR for a virtual database that uses a postgreSQL data source:

  Example: dv-customer.yaml

```
apiVersion: teiid.io/v1alpha1
kind: VirtualDatabase
metadata:
  name: dv-customer (1)
spec:
  replicas: 1 (2)
  datasources: (3)
    - name: sampledb
      type: postgresql
      properties:
        - name: username
          value: USER
        - name: password
          value: MYPASSWORD
        - name: jdbc-url
          value: jdbc:postgresql://accounts/accounts
build:
    source:
      dependencies: (4)
      - org.postgresql:postgresql:42.1.4
      ddl: | (5)
        CREATE DATABASE customer OPTIONS (ANNOTATION 'Customer VDB');
        USE DATABASE customer;

        CREATE SERVER sampledb TYPE 'NONE' FOREIGN DATA WRAPPER postgresql;

        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');

        SET SCHEMA portfolio;

        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;
      mavenRepositories: (6)
        central: https://repo.maven.apache.org/maven2
```

(1) The name of the virtual database.

(2) Specifies the number of instances to deploy. The default setting is 1.

(3) Specifies the data source properties for the virtual database. The properties in the example apply to a connection to a PostgreSQL database. For information about supported data sources and their properties, see Section 2.1, “Compatible data sources”.

(4) Specifies a list of Maven dependency JAR files in GAV format (groupId:artifactid:version). These files define the JDBC driver files and any custom dependencies for the data source. Typically, the Operator build automatically adds libraries that are available in public Maven repositories.

5 Defines the virtual database in DDL form. For information about how to use DDL to define a virtual database, see *DDL metadata for schema objects* in the Data virtualization

6 Specifies the location of any private or non-public repositories that contain dependencies or other virtual databases. You can specify multiple repositories. If dependencies are in a repository other than the public Maven Central repository, specify the repository location. For more information about using private Maven repositories, see Section 4.3, "Private Maven repositories".

After you create the YAML file, you can run the Data Virtualization Operator to deploy the virtual database to OpenShift. For more information, see Chapter 6, *Running the Data Virtualization Operator to deploy a virtual database*.

3.1. CREATING A CR TO DEPLOY A DDL ARTIFACT

If you create a virtual databases by embedding DDL directly in a CR, you already have the CR that the Data Virtualization Operator uses for deployment. For information about the CR for a DDL artifact, see Chapter 3, *Creating a virtual database by embedding DDL statements in a custom resource (CR)* .

Run the Data Virtualization Operator with the CR to generate the virtual database and deploy it to OpenShift.

Additional resources

- Chapter 6, *Running the Data Virtualization Operator to deploy a virtual database* .

CHAPTER 4. CREATING A VIRTUAL DATABASE AS A MAVEN ARTIFACT

You can use a Teiid Maven plugin to convert a DDL file into a Maven artifact. You define the structure of the virtual database in a DDL file and use the file to generate an artifact to deploy to a Maven repository. The Data Virtualization Operator can then deploy the artifact from the Maven repository to an OpenShift project.

This is an advanced method that provides a high level of flexibility and is suitable for complex projects. Using this method, you can create multi-module Maven projects in which you import one or more other virtual databases and incorporate them into your design.

You specify use of the Teiid plugin in your **pom.xml** file. You can also define other Maven dependencies in the **pom.xml** file. When you run the build, the plugin reads the file and resolves its contents.

Advantages of creating a virtual database as a Maven artifact

- Flexible, clean separation between the DDL code that represents the virtual database and other configuration settings.
- Enables easy deployment into multiple environments.
- Provides for versioning at the virtual database level.
- Enables importing of one virtual database into another, by adding **IMPORT DATABASE** statements to the DDL.
- Enables a virtual database to be shared across projects and teams in a consistent way.
- Supports continuous integration and continuous delivery (CI/CD) workflows.

Disadvantages of creating a virtual database as a Maven artifact

- Requires a working knowledge of Maven.

Prerequisites

- You have a compatible data source and the OpenShift cluster can access it.
- You know how to create a **pom.xml** file to specify the dependencies that are required to build your virtual database.
- You have information about the connection settings for your data sources, including login credentials.
- The Data Virtualization Operator has access to the Maven repositories that contain build dependencies for the virtual database.
- You have Maven 3.2 or later installed.

Procedure

1. From a text editor, create a POM file to define build dependencies. For example,

   Example: POM file for building a Maven-based virtual database

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.teiid</groupId>
  <artifactId>dv-customer</artifactId>
  <name>dv-customer</name>
  <description>Demo project to showcase maven based vdb</description>
  <packaging>vdb</packaging>
  <version>1.0</version>

  <build>
    <plugins>
      <plugin>
        <groupId>org.teiid</groupId>
        <artifactId>vdb-plugin</artifactId> (1)
        <version>1.2.0</version>
        <extensions>true</extensions>
        <executions>
          <execution>
            <goals>
              <goal>vdb</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

(1) For virtual databases that import from other virtual databases, supply a unique **artifactId** name for each virtual database that you want to import.

The preceding example can serve as a model for the **pom.xml** for your virtual database.

2. Create a Maven project to import the virtual database definition from a DDL file. For example:

```
vdb-project
├── pom.xml
└── src
    └── main
        └── vdb
            └── vdb.ddl
```

3. If you do not already have one, create a DDL file to specify the structure of the virtual database, and save it to the **/src/main/vdb** directory of your project. Maven-based virtual databases use the same DDL structure as other virtual databases. The DDL file for a Maven-based virtual database must have the name **vdb.ddl**.
   The following example shows a sample DDL file for a virtual database that uses a postgreSQL data source:

   Example: vdb.ddl

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

CREATE FOREIGN DATA WRAPPER postgresql;
CREATE SERVER sampledb TYPE 'NONE' FOREIGN DATA WRAPPER postgresql;

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');

SET SCHEMA portfolio;

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;
```

Additional resources

- For information about how to use DDL to define a virtual database, see *DDL metadata for schema objects* in the Data Virtualization Reference. Defining the complete DDL is beyond the scope of this document.
- For more information about importing an existing virtual database into the current one, see Section 4.4, “Virtual database import”.

4.1. BUILDING A VIRTUAL DATABASE ARTIFACT

After you have all components of your Maven-based virtual database project, you can build the artifact and deploy it to your Maven repository.

Prerequisites

- You set up your virtual database Maven project.
- You have a DDL file, saved as **vdb.ddl**, that describes the virtual database that you want to build.
- You have a **pom.xml** file that defines the dependencies for building the virtual database.

Procedure

1. Open a terminal window to the root folder of your Maven project, and type the following command:

   ```
   mvn deploy
   ```

   The command builds the virtual database and deploys it to a local or remote Maven repository. The Maven repository can be public or private. The command generates a ***PROJECT_NAME-VERSION*.vdb** file in your target repository.

After the virtual database artifact is available in a Maven repository, you can use a YAML-based custom resource to deploy the virtual database to OpenShift. For information about using YAML to create a custom resource for deploying virtual database Maven artifacts, see Section 4.2, "Creating a custom resource (CR) to deploy a Maven artifact".

For information about using the Data Virtualization Operator to deploy a virtual database, see Chapter 6, *Running the Data Virtualization Operator to deploy a virtual database* .

4.2. CREATING A CUSTOM RESOURCE (CR) TO DEPLOY A MAVEN ARTIFACT

Before you can deploy a virtualization that you create as a Maven artifact, you must create a CR that defines the location of the Maven repository. When you are ready to deploy the virtualization, you provide this CR to the Data Virtualization Operator.

Prerequisites

- You created a virtualization according to the instructions in Chapter 4, *Creating a virtual database as a Maven artifact*.
- You deployed the virtualization to a Maven repository that the Data Virtualization Operator can access.
- You have the login credentials to access the data source.
- You are familiar with the creation of custom resource files in YAML format.

Procedure

1. Open a text editor, create a file with the name of the virtualization, and save it with the extension **.yaml**, for example, **dv-customer.yaml**.
2. Add information to define the custom resource kind, name, and source. The following annotated example provides guidance on the contents to include in the CR:

   dv-customer.yaml

```
apiVersion: teiid.io/v1alpha1
kind: VirtualDatabase
metadata:
  name: dv-customer
spec:
  replicas: 1
  datasources:
    - name: sampledb
      type: postgresql
      properties:
        - name: username (1)
          value: user
        - name: password
          value: mypassword
        - name: jdbc-url
          value: jdbc:postgresql://sampledb/sampledb (2)
  resources:
    memory: 1024Mi
```

```
    cpu: 2.0
  build:
    source:
      maven: com.example:customer-vdb:1.0.0:vdb      3
    mavenRepositories:
      central: https://repo.maven.apache.org/maven2  4
```

1. Specifies the credentials for signing in to the data source. Although this example shows credentials that are defined within the CR, in production use, use secrets to specify credentials, rather than exposing them in plain text. For information about adding credentials to secrets, see Section 7.2, "Using secrets to store data source credentials" .
2. Specifies the URL for connecting to the data source.
3. Specifies the Maven location of the virtual database by providing the groupId, artifactId, and version (GAV) coordinates.
4. If you are using a private Maven repository, specify its URL. You can configure multiple repositories.

After you create the CR YAML file, you can run the Data Virtualization Operator to deploy the virtual database to OpenShift.

Run the Data Virtualization Operator with the CR to generate the virtual database and deploy it to OpenShift.

Additional resources

- Chapter 6, *Running the Data Virtualization Operator to deploy a virtual database* .

4.3. PRIVATE MAVEN REPOSITORIES

When you run the Data Virtualization Operator to build a virtual database, the Operator initiates a Maven build. The Maven build converts the DDL in the custom resource that you provide into a container image that can be deployed to OpenShift. If the Operator requires additional packages to complete the build process (for example, JDBC drivers, client libraries, and other dependency libraries), it retrieves these build dependencies from a Maven repository. By default, the Operator retrieves build dependencies from the public Maven Central repository.

In some environments the network configuration does not permit direct access to the internet, preventing the Operator from connecting to the Maven Central repository. To enable the Operator to build virtual databases when it cannot connect to the Maven Central repository, you can configure the use of a local, private Maven repository.

When you run the Operator to build a virtual database, it searches for a secret or ConfigMap object with a specific name (either **teiid-maven-settings** or ***VDB_NAME*-maven-settings**). If it finds a matching object, the Operator uses the Maven repositories specified in the **settings.xml** section of the object to resolve any dependencies. As long as the **settings.xml** key in the named object is correctly specified, no other configuration is required.

Methods for specifying private Maven repositories for virtual database builds

You can use several different methods to specify the local private Maven repository that the Operator uses to retrieve dependencies. The method that you choose depends the following factors:

- Whether the private Maven repository requires authentication
- Whether you want to use a single repository to build all of your virtual database.

The following table lists the methods that are available.

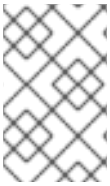
Table 4.1. Methods for specifying a private Maven repository

Method	Description	Limitations
Specify a global repository.	Applies to all virtual database in a namespace. List repositories in a **settings.xml** key that you add to an OpenShift secret or ConfigMap.	Secret or ConfigMap must use the name **teiid-maven-settings**
Specify unique repositories for individual virtual databases.	Applies to a single virtual database. Lists repositories in a **settings.xml** key that you add to an OpenShift secret or ConfigMap.	The secret or ConfigMap must mirror the name of the virtual database in the format ***VDB_NAME*-maven-settings.xml**
Specify repositories in the custom resource.	Lists repositories in the custom resource that you use to deploy the virtual database.	The Maven repository cannot require authentication.

4.3.1. Specifying private Maven repositories to build all virtual databases in a namespace

You can provide a full **settings.xml** file that specifies a single common Maven repository for the Operator to use in building any virtual database. Use this method if your Maven repository requires authentication, or if you have multiple repositories and you want to use a specific one for all your virtual database builds.

When the Operator builds the image for your virtual database, it checks for a ConfigMap named **teiid-maven-settings**. If it finds it, it then uses the **settings.xml** file in the ConfigMap for the build.



NOTE

You an override the use of a common repository by specifying a **settings.xml** to apply to a particular virtual database build. For more information, see Section 4.3.2, "Specifying private Maven repositories for building an individual virtual database".

Procedure

1. Create a ConfigMap or secret and assign the following value to the **metadata/name** key: **teiid-maven-settings**

2. Add a key to the ConfigMap or secret and assign the key the name **settings.xml**. The value of the key contains a full Maven settings file. The following example provides an excerpt that shows how to include a **settings.xml** file in a ConfigMap:

Sample ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: teiid-maven-settings
  namespace: myproject
data:
  settings.xml: |-
    <?xml version="1.0" encoding="UTF-8"?>
    <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
    <localRepository />
    <profiles>
       <profile>
          <id>maven-settings</id>
          <activation>
             <activeByDefault>true</activeByDefault>
          </activation>
          <repositories>
             <repository>
             <id>private</id>
             <url>https://myprivate.host.com/maven2</url>
             <snapshots>
                <enabled>false</enabled>
                <checksumPolicy>fail</checksumPolicy>
             </snapshots>
             <releases>
                <enabled>false</enabled>
                <checksumPolicy>fail</checksumPolicy>
             </releases>
             </repository>
          </repositories>
       </profile>
    </profiles>
    <activeProfiles>
      <activeProfile>maven-settings</activeProfile>
    </activeProfiles>
    </settings>
```

NOTE

The preceding example does not represent a working **settings.xml** file. You can use the example as the basis for your own ConfigMap, but you must provide details that are are specific to your Maven repository.

3. Save the ConfigMap YAML file as **maven-settings.yaml**.

4. Deploy the ConfigMap to OpenShift by typing following command:

```
oc create -f maven-settings.yaml
```

4.3.2. Specifying private Maven repositories for building an individual virtual database

In some cases, you might want the Data Virtualization Operator to use a specific Maven repository when it builds a particular virtual database. You can specify the Maven repository to use in a **settings.xml** file that you add to a ConfigMap or secret that applies only to a virtual database with a specific name.

The configuration that you add for a specific virtual database build takes precedence over any global configuration that you set for the namespace.

The repositories that you add designate for use in building an individual virtual database cannot require authentication.

Procedure

1. Create a ConfigMap or secret and assign the **metadata**/**name** key a value that matches the name of the virtual database, using the format ***VDB_NAME*-maven-settings.xml**. For example,

```
metadata:
  name: dv-customer-maven-settings
```

1. Add a key to the ConfigMap or secret and assign the key the name **settings.xml**. The value of the key contains a full Maven settings file.
2. Save the ConfigMap YAML file as **maven-settings.yaml**.
3. Deploy the ConfigMap to OpenShift by typing following command:

```
oc create -f maven-settings.yaml
```

Additional resources

- Section 4.3.1, “Specifying private Maven repositories to build all virtual databases in a namespace”

4.3.3. Specifying the private Maven repositories for building an individual virtual database in its custom resource

You can specify the private Maven repositories to use in building an individual virtual database in the custom resource for the virtual database. The method of specifying Maven repositories in a custom resource applies only to repositories that do not require authentication.

The repositories that you add to the CR cannot require authentication.

When you run the Data Virtualization Operator to build a virtual database, it uses the settings in the CR. For example,

dv-customer.yaml

```
apiVersion: teiid.io/v1alpha1
kind: VirtualDatabase
```

```
metadata:
  name: dv-customer
spec:
  replicas: 1
  build:
    source:
      ddl: |
        CREATE DATABASE customer OPTIONS (ANNOTATION 'Customer VDB');
        USE DATABASE customer;
        ...
      mavenRepositories: 1
        private: https://myprivate.host.com/maven2
        private2: https://myprivate.host2.com/maven2
```

1 Define one or more Maven private repositories to be used with the build.

If you specify Maven repositories in the CR, the Operator uses them in addition to the repositories specified in the **settings.xml** file.

4.4. VIRTUAL DATABASE IMPORT

You can use a Maven project to build a special type of virtual database that imports from one or more existing virtual databases. The project structure and build process for these importing virtual databases is the same as for other virtual databases that you create as Maven artifacts.

Virtual database import can be useful in organizations in which departments control access to their own data sources, but must share subsets of their original data with other teams. The data owners might not want to prevent outside groups from viewing details about the physical structure of the data. By setting up a virtual database, the owning group can establish an abstraction layer that exposes only targeted data without directly exposing your data sources.

Consuming groups typically have their own data stores that they use for data received from other groups. Instead of accessing the source data directly, they can create their own virtual database layer to import the data. Through this second abstraction layer, the consuming group can access the data in a way that insulates them from changes to the physical data store schema. When the group that owns the data updates the base virtual databases, changes can be easily pushed to the consuming groups virtual databases.

For example, suppose that a Sales team has the following two postgreSQL databases:

- Accounts Database (AccountsDB)
- Sales Database (SalesDB)

Should members of the Operations team require data from the Sales databases, the Sales team can create a virtual database to expose the source data. The Operations team can then consume the Sales data by creating their own virtual database, as in the following example:

Example: Operations team virtual database that imports data from two virtual databases created by the Sales team

```
CREATE DATABASE OperationsDB;
USE DATABASE OperationsDB;
```

```
IMPORT DATABASE AccountsDB VERSION 1.0;
IMPORT DATABASE SalesDB VERSION 1.0;
```

In the preceding example, the DDL defines the database **OperationsDB**, which then imports metadata from two source databases, **AccountsDB** and **SalesDB**. Users who connect to **OperationsDB** do not know anything about the two virtual databases that supply the data, but they have full access to the data that they expose. If future changes occur in the source databases, the Operations team can rebuild and redeploy a new version of the **OperationsDB** database to make the changes available to its users.

4.4.1. How virtual database importing works

A virtual database developer can deploy a virtual database to a Maven repository, where it is assigned an identifier, version number, and a defined location. The resulting Maven artifact can then be defined as a dependency to the build process of a second virtual database. When this second virtual database is built, the build reads the contents of the original virtual database from the Maven repository and incorporates it into current new virtual database.

For example, imagine that a Sales team developer creates the virtual databases **AccountsDB** and **SalesDB**, and then deploys them to a Maven repository. When the Operations team developer creates a Maven project for a secondary virtual database, **OperationsDB**, the project defines the **AccountsDB** and **SalesDB** databases as dependencies. The new project can then use an **IMPORT** statement to extract content from the two original virtual databases. After the developer deploys the **OperationsDB** database, users can connect to it to access data from both of its source databases.

Because the **OperationsDB** database is also available as a Maven artifact, it can be used as a data source by other virtual databases.

To deploy the **OperationsDB** virtual database, you must supply the Maven coordinates of the **OperationsDB** to the Data Virtualization Operator. During the build, the Operator retrieves the contents from the Maven repository, including the dependency information, and deploys the virtual database to OpenShift.

The Maven build process enables build tasks to be automated as part of an automated CI/CD workflow in which you can configure Maven to rebuild virtual databases automatically with no user intervention after changes occur in the sources.

4.4.2. POM and DDL for virtual databases that import from other virtual databases

You use a Maven build process to create a virtual database that import from other virtual databases.

The process and the project structure for developing virtual databases that use importing is the same as the process for creating any virtual database as a Maven artifact. You use the standard Maven project structure for your virtual database. Resources in the project must be added to the expected locations in the project structure. If you want to include additional metadata files with the virtual database, add them within this structure.

For more information about the structure for creating virtual databases as Maven artifacts, see Chapter 4, *Creating a virtual database as a Maven artifact* .

POM file

As with any virtual database that you create as a Maven artifact, the project for an importing virtual database must include a POM file. For an example of a POM file for building a virtual database as a Maven artifact, see Chapter 4, *Creating a virtual database as a Maven artifact* .

Along with the standard entries, the POM for an importing virtual database must list the following entries:

- The **artifactId** for each virtual databases to import from.
- Dependency definitions for each imported virtual database.

For example, if you want to import the virtual database *accountsdb* into the secondary virtual database *OperationsDB*, the POM file for the *OperationsDB* database must include a dependency entry for the *accountsdb* database as in the following example:

Example: Entry in the POM file for the OperationsDB database, listing the accountsdb virtual database as a dependency

```
<dependencies>
  <dependency>
    <groupId>org.example</groupId>
    <artifactId>accountsdb</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
```

If you want to import multiple virtual databases, the **<dependencies>** section of the POM must include separate **<dependency>** entries for each virtual database. The **<dependencies>** block is contained within the **<projects** section of the POM.

DDL

The DDL file for a Maven-based virtual database must have the name **vdb.ddl**. Maven-based virtual databases use the same DDL structure as other virtual databases.

The DDL for the importing virtual database must contain an **IMPORT** statement similar to the following statement:

```
IMPORT DATABASE IMPORTED_VIRTUAL_DATABASE_ VERSION VERSION_NUMBER__;
```

For example, if you want to import the virtual database *accountsdb* into a virtual database with the name *OperationsDB*, you would include the following entry in the DDL for the *OperationsDB* database:

```
IMPORT DATABASE AccountsDB VERSION 1.0;
```

The Maven build reads the contents of the **vdb.ddl** file for the **AccountsDB** and merges it into the virtual database that is being built. The result is a virtual database comprised of multiple virtual databases.

After you configure the source files for the Maven build, you can run the build to deploy the Maven artifact to a repository. The process for building the virtual database is the same as with any Maven-based virtual database. For more information, see Section 4.1, “Building a virtual database artifact”

4.4.3. DDL limitations

You should avoid the use of some statements in the DDL for a virtual database.

Foreign schema import

For example:

```
IMPORT FOREIGN SCHEMA public FROM SERVER sampledb INTO accounts;
```

The **IMPORT FOREIGN SCHEMA** statement is an expensive operation, that queries the underlying physical data source every time a pod restarts. Introducing this query places a strain on the underlying physical data source, increasing the time that it takes for the pod to start. The problem is magnified if you have multiple pods trying to access the data source at once.

Another problem with importing the foreign schema from another virtual database is that images deployed to OpenShift are assumed to be in an *immutable* state. That is, no matter how many times the image is stopped and started, their behavior should persist. However, if you define a SQL **IMPORT SCHEMA** operation in the DDL, the virtual database loads the schema from the source virtual database every time that the image starts up. As a result, the image contents can be modified, which runs counter to the principle of immutability for this architecture.

If you can guarantee that the underlying data source always returns the same metadata, there is no problem. Problems arise if the data source returns different metadata when the image starts.

To ensure that the contents of an image remain stable, it is best to define all metadata explicitly in the virtual database DDL, including tables, procedures, and any functions that data source represents.

ALTER statements to modify metadata

Do not use **ALTER TABLE** statements in the DDL for a virtual database if the DDL includes either of the following statements:

- **IMPORT FOREIGN SCHEMA**
- **IMPORT DATABASE**

ALTER TABLE statements are intended to modify the structure of a table by adding, removing, or modifying columns. However, in the case of imported virtual databases, the actual structure of the imported tables is not available at build time. From the perspective of the virtual database, the runtime metadata does not yet exist to be modified.

When you include an **ALTER TABLE** statement in the DDL with an **IMPORT** statement, it attempts to change a table or column that does not exist in the the virtual database metadata. A deployment failure can result, or if the virtual database is deployed, its contents might not include the expected data.

CHAPTER 5. DATA SOURCE CONFIGURATION

You can configure external services as data sources for a virtual database. To describe a data source, you add properties to a custom resource (CR) file. Some properties are common to multiple data sources. Other properties are specific to a particular data source. Property names are case-sensitive.

For every data source, you must provide the name of a *translator* that can interpret the commands and data that pass between the virtual database and the data source. For example, the web service translator converts SQL procedures executed to a virtual database to an HTTP call to send to a web service. Similarly, the translator can convert a JSON response to tabular results.

Translators can also include optional configurable properties that you can use to manage the behavior of the translator. Execution properties control how data is retrieved. Import settings determine the metadata that is the virtual database reads and imports.

Additional resources

- For more information about translators, see Data Virtualization Reference

The sections that follow describe the specific properties to set in the custom resource files to add data sources to virtual databases. Details about how to create the DDL to define the database structure are not covered.

5.1. CONFIGURATION PROPERTIES FOR S3 AND CEPH AS DATA SOURCES

You can configure an Amazon Simple Storage Service (S3) as a data source for a virtual database. Using a similar configuration, you can also use the Ceph storage platform as a data source.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For both S3 and Ceph, set the translator to *amazon-s3*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

Data source information

The custom resource that you use to create a virtual database from S3 or Ceph requires information about the service, For example, you must provide the access key and secret key that you use to sign request to AWS.

Dependencies

S3 and Ceph do not require you to specify any build dependencies.

The following table shows the data source information that is required in the data source properties of the custom resource:

Table 5.1. Data source properties for S3/Ceph

Property Name	Description	Required	Default value
region	S3 region. For example, **us-east-2** [a]	Yes	n/a

Property Name	Description	Required	Default value
bucket	The name of the S3 bucket.	Yes	n/a
accesskey	Access key ID for signing requests to AWS services.	Yes	n/a
secretkey	Secret access key for signing requests.	Yes	n/a

[a] The **region** property is not required for Ceph data sources.

Example: Excerpt from an S3 custom resource, showing the format for setting key properties

```
datasources:
  - name: sampledb
    type: amazon-s3
    properties:
      - name: region
        value: us-east-2
      - name: bucket
        value: mybucket
      - name: accesskey
        value: xxxxxxxx
      - name: secretkey
        value: xxxxxx
```

5.2. SETTINGS TO CONNECT TO GOOGLE SHEETS AS A DATA SOURCE

You can configure Google Sheets as a data source for a virtual database.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For Google Sheets sources, set the translator to *google-spreadsheet*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server. Each Sheet in a Google Sheets spreadsheet becomes available as a table in the virtual database.

For a virtual database to connect to Google Sheets, you must register your data virtualization service as a Google client application. During registration, you enable Google Sheets APIs and create credentials that the virtual database uses to access the APIs.

For information about how to register data virtualization as a Google client application, see the Google OAuth documentation.

The following table shows the data source information that is required in the data source properties of the custom resource:

Table 5.2. Data source properties for Google Sheets

Property Name	Description	Required	Default value
spreadSheetName	Name of the Google Sheets spreadsheet	Yes	n/a
spreadSheetId	Spreadsheet ID	Yes	Sheet ID in the URL of the spreadsheet. For more info see the Google Sheets API documentation.
clientId	OAuth2 client ID for Google Sheets	Yes	n/a
clientSecret	OAuth2 client secret for Google Sheets	Yes	n/a
refreshToken	OAuth2 refreshToken for Google Sheets	Yes	n/a

5.3. CONFIGURATION PROPERTIES FOR RED HAT DATA GRID (INFINISPAN) AS A DATA SOURCE

You can configure Red Hat Data Grid (Infinispan) as a data source for a virtual database.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For Data Grid or Infinispan sources, set the translator to *infinispan-hotrod*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

Dependencies

Data Grid or Infinispan provide the client libraries that you need. You do not have to specify any build dependencies.

The following table shows the information that is required in the data source properties of the custom resource:

Table 5.3. Data source properties for Data Grid

Property Name	Description	Required	Default value
url	URL to connect to Infinispan	Yes	n/a
username	User name	Yes	n/a

Property Name	Description	Required	Default value
password	Password	Yes	n/a
cacheName	Default cache name	No	n/a
authenticationRealm	Auth Realm	No	n/a
authenticationServerName	Auth Server	No	n/a

Example: Excerpt from an Infinispan custom resource, showing the format for setting key properties

```
datasources:
  - name: sampledb
    type: infinispan-hotrod
    properties:
      - name: url
        value: localhost:11222
      - name: user
        value: user
      - name: password
        value: pass
      - name: cacheName
        value: test
```

5.4. CONFIGURATION PROPERTIES FOR MONGODB AS A DATA SOURCE

You can configure MongoDB as a data source for a virtual database.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER.** For MongoDB sources, set the translator to *mongodb*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

Dependencies

MongoDB does not require you to specify any build dependencies.

The following tables list the properties that are required in the CR to create a virtual database that is based on a MongoDB database

Table 5.4. Data source properties for MongoDB

Property Name	Description	Required	Default value

Property Name	Description	Required	Default value
remoteServerList	List of MongoDb servers, for example: (localhost:27012).	Yes	n/a
user	User name.	Yes	n/a
password	Password.	Yes	n/a
database	Database name to connect to.	Yes	n/a
authDatabase	Database name for authorization.	No	n/a
ssl	Use SSL Connection?	No	n/a

Example: Excerpt from an MongoDB custom resource, showing the format for setting key properties

```
datasources:
  - name: sampledb
    type: mongodb
    properties:
      - name: user
        value: USER_NAME
      - name: password
        value: PASSWORD
      - name: remoteServerList
        value: localhost:27012
      - name: database
        value: DATABASE_NAME
```

For a complete list of the properties that you can set to control how data is translated between MongoDB and a virtual database, see the Data Virtualization Reference.

5.5. RELATIONAL DATABASES DATA SOURCES CONFIGURATION

To configure a virtual database to connect to a relational database for reading or writing tables, you provide information about the database by specifying a common set of properties in the custom resource, as shown in the following table:

Table 5.5. Data source properties for relational databases

Property name	Description	Required	Default value
jdbc-url	URL for the connection	Yes	n/a
username	User name	Yes	n/a

Property name	Description	Required	Default value
Password	Yes	n/a	n/a
jdbcDriverClass [a]	Driver class name	No	n/a
importer.schemaName	Schema name for import	Yes	n/a

[a] Depending on the database, multiple JDBC drivers might be available. To ensure that the build uses a suitable driver, some database types require you to specify a driver class.

You must also specify the Maven coordinates for the JDBC driver. For more information, see JDBC drivers.

JDBC drivers

To use a relational database as a source for a virtual database, you must provide a JDBC driver to manage the connection to the database. For some database types, such as postgreSQL and SQL Server, the JDBC driver is provided automatically during the virtual database build. However, for other databases, you must specify the driver to retrieve from the public Maven repository, or, if there is no publicly available driver, you must download the driver manually.

For downloaded drivers to be available to the build, you must add them to a private Maven repository, and then reference the repository in the virtual database CR.

If the source database requires that you specify build dependencies for the JDBC driver class, you specify these in the **build.source.dependencies** element in the CR. For most databases it is not necessary to define the driver class.

The following example shows an excerpt from a CR that defines the data source configuration for a sample postgreSQL database.

Example: Custom resource that defines data source properties in-line

```
spec:
  datasources: 1
    - name: sampledb 2
      type: postgresql 3
      properties:
        - name: username
          value: postgres
        - name: password
          value: postgres
        - name: jdbc-url 4
          value: jdbc:postgresql://database/postgres
        - name: jdbcDriverClass 5
          value: org.postgres.jdbc.Driver
  build:
    source:
      dependencies: 6
        - org.postgresql:postgresql:42.1.4
```

1. The **datasources** section lists the properties that define the connections to your data sources.
2. The custom name assigned to the source database.
3. The data source type. The type must match the translator that you specify in the **ddl** section of the CR.
4. The URL for the source database. The URL uses the format **jdbc:xxxx** where **xxxx** is the name of the data source. Requirements for specifying the full URL string vary by database vendor. Values in the string are not required to match names for the data source our translator.
5. Specifies the driver class for the JDBC driver.
6. Specifies a list of Maven dependency JAR files in GAV format (**groupId:artifactid:version**). These files define the JDBC driver files and any custom dependencies for the data source. For some database types the Data Virtualization Operator automatically adds the required JDBC libraries, and it is not necessary to specify the dependencies.

Property values defined as secrets

As an alternative to defining values for properties directly in the CR, you can define references to values in a **secret** object. This is especially important for securing sensitive data such as **Password** properties. For more information, see Section 7.2, “Using secrets to store data source credentials” .

To further tune the JDBC translator and schema import behavior, you can define additional properties. For more information see the Data Virtualization Reference.

Any Maven repository that you list must be available to the Data Virtualization Operator when it builds the virtual database. To provide the Operator with access to Maven resources that are not available from the public Maven Central repository, you can configure one or more private repositories. For more information, see Section 4.3, “Private Maven repositories”.

5.5.1. Configuration properties for Amazon Athena as a data source

You can configure an Amazon Athena query service as a data source for a virtual database.

Set the translator in the DDL for the virtual database to *amazon-athena or jdbc-ansi* with a matching **Server** definition.

The custom resource that you use to create a virtual database from Athena is the same as the CR for a standard JDBC source.

For more information, see Section 5.5, “Relational databases data sources configuration” .

The following table lists specific properties to use when you create a virtual database that is based on an Amazon Athena database.

Table 5.6. Property settings for using Amazon Athena as a data source

JDBC driver dependency	**jdbc-url** (Source database URL)	**jdbc-driver-class** name	JDBC driver download link

JDBC driver dependency	**jdbc-url** (Source database URL)	**jdbc-driver-class** name	JDBC driver download link
Based on downloaded driver [a]	**jdbc:awsathena://User=*ACCESS_KEY*;Password=*SECRET_KEY*;S3OutputLocation=*OUTPUT*;*PROPERTY1=VALUE*;*PROPERTY2=VALUE2*;**	com.simba.athena.jdbc.Driver [b]	https://docs.aws.amazon.com/athena/latest/ug/connect-with-jdbc.html[c]

[a] Obtain the driver from the link in the **Driver link** column of this table, and define a driver dependency that is based on the driver name in the **build/source/dependencies** section of the CR.

[b] Specify the driver class to ensure that the Data Virtualization Operator retrieves the correct driver from the JAR file.

[c] When you create a virtual database from an Amazon Athena source, the build does not automatically include the Athena JDBC driver. To supply the necessary driver, download it from the specified link, and add it to a Maven repository that the Data Virtualization Operator can access when it runs the OpenShift Source-To-Image (S2I) build.

For an example that shows how properties are defined in the custom resource for a virtual database that uses a relational database as its source, see Section 5.5, “Relational databases data sources configuration”.

5.5.2. Configuration properties for Amazon Redshift data sources

You can configure Amazon Redshift as a data source for a virtual database.

Set the translator in the DDL for the virtual database to **redshift** with a matching **SERVER** definition.

The custom resource that you use to create a virtual database from a Redshift source is the same as the CR for a standard JDBC source.

For more information, see Section 5.5, “Relational databases data sources configuration” .

The following table lists specific properties to use when you create a virtual database that is based on an Amazon Redshift database.

Table 5.7. Property settings for using Amazon Redshift as a data source

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link
'com.amazon.redshift:redshift-jdbc42:jar:1.2.1.1001' [a]	**jdbc:awsathena://User=*ACCESS_KEY*;Password=*SECRET_KEY*;S3OutputLocation=*OUTPUT*;*PROPERTY1=VALUE*;*PROPERTY2=VALUE2*;**	**com.amazon.redshift** [b]	N/A

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link

[a] When you create the custom resource for the virtual database, define a driver dependency with this value in the **build/source/dependencies** section.

[b] Specify the driver class to ensure that the Data Virtualization Operator retrieves the correct driver from the JAR file.

For an example that shows how properties are defined in the custom resource for a virtual database that uses a relational database as its source, see Section 5.5, “Relational databases data sources configuration”.

5.5.3. Configuration properties for Db2 as a data source

You can configure Db2 as a data source for a virtual database.

Set the translator in the DDL for the virtual database to *db2* with a matching **Server** definition.

The custom resource that you use to create a virtual database from Db2 is the same as the CR for a standard JDBC source.

For more information, see Section 5.5, “Relational databases data sources configuration” .

The following table lists specific properties to use when you create a virtual database that is based on a Db2 database.

Table 5.8. Property settings for using Db2 as a data source

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link
com.ibm.db2:jcc:jar:11.1.4.4 [a]	**jdbc:db2://*HOST*:50000/*DATABASE_NAME***	**com.ibm.db2.jcc.DB2Driver** [b]	N/A

[a] When you create the custom resource for the virtual database, define a driver dependency with this value in the **build/source/dependencies** section.

[b] Specify the driver class to ensure that the Data Virtualization Operator retrieves the correct driver from the JAR file.

For an example that shows how properties are defined in the custom resource for a virtual database that uses a relational database as its source, see Section 5.5, “Relational databases data sources configuration”.

5.5.4. Configuration properties for Microsoft SQL Server as a data source

You can configure Microsoft SQL Servers as a data source for a virtual database.

Set the translator in the DDL for the virtual database to **sqlserver** or **ms-sqlserver** with a matching **SERVER** definition.

The custom resource that you use to create a virtual database from a SQL Server source is the same as the CR for a standard JDBC source.

For more information, see Section 5.5, "Relational databases data sources configuration" .

The following table lists specific properties to use when you create a virtual database that is based on a SQL Server database.

Table 5.9. Property settings for using SQL Server as a data source

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link
com.microsoft.sqlserver:sqljdbc4:jar:4.0 (Optional) [a]	**jdbc:microsoft:sqlserver://*HOST*:1433**	**com.microsoft.sqlserver.jdbc.SQLServerDriver** (Optional)	N/A

[a] When you run the Data Virtualization Operator to build a virtual database that uses a SQL Server source, the build process automatically retrieves the required JDBC driver. It is not required to also define the driver dependency in the custom resource.

For an example that shows how properties are defined in the custom resource for a virtual database that uses a relational database as its source, see Section 5.5, "Relational databases data sources configuration".

5.5.5. Configuration properties for MySQL as a data source

You can configure MySQL as a data source for a virtual database.

Set the translator in the DDL for the virtual database to *mysql* with a matching **Server** definition.

The custom resource that you use to create a virtual database from MySQL is the same as the CR for a standard JDBC source.

For more information, see Section 5.5, "Relational databases data sources configuration" .

The following table lists specific properties to use when you create a virtual database that is based on a MySQL database.

Table 5.10. Property settings for using MySQL as a data source

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link
mysql:mysql-connector-java:jar:8.0.20 (Optional) [a]	**jdbc:mysql://*HOST*:3306/*DATABASE_NAME***	**com.mysql.jdbc.Driver** (Optional)	N/A

[a] When you run the Data Virtualization Operator to build a virtual database that uses a MySQL source, the build process automatically retrieves the required JDBC driver. It is not required to also define the driver dependency in the custom resource.

For an example that shows how properties are defined in the custom resource for a virtual database that uses a relational database as its source, see Section 5.5, “Relational databases data sources configuration”.

5.5.6. Configuration properties for Oracle Database as a data source

You can configure Oracle Database as a data source for a virtual database.

Set the translator in the DDL for the virtual database to **oracle** with a matching **SERVER** definition.

The custom resource that you use to create a virtual database from an Oracle Database source is the same as the CR for a standard JDBC source.

For more information, see Section 5.5, “Relational databases data sources configuration” .

The following table lists specific properties to use when you create a virtual database that is based on Oracle Database.

Table 5.11. Property settings for using Oracle Database as a data source

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link
com.oracle:ojdbc14:jar:10.2.0.4.0 [a]	**jdbc:oracle:thin:*HOST*:1521:orcl**	**oracle.jdbc.driver.OracleDriver** [b]	N/A

[a] When you create the custom resource for the virtual database, define a driver dependency with this value in the **build/source/dependencies** section.

[b] Specify the driver class to ensure that the Data Virtualization Operator retrieves the correct driver from the JAR file.

For an example that shows how properties are defined in the custom resource for a virtual database that uses a relational database as its source, see Section 5.5, “Relational databases data sources configuration”.

5.5.7. Configuration properties for postgreSQL as a data source

You can configure postgreSQL as a data source for a virtual database.

Set the translator in the DDL for the virtual database to **postgresql** with a matching **SERVER** definition.

The custom resource that you use to create a virtual database from a postgreSQL source is the same as the CR for a standard JDBC source. For more information, see Section 5.5, “Relational databases data sources configuration”.

The following table lists specific properties to use when you create a virtual database that is based on a postgreSQL database.

Table 5.12. Property settings for using postgreSQL as a data source

JDBC driver dependency	**jdbc-url** (URL for the source database)	**jdbc-driver-class** name	JDBC driver download link
org.postgresql:postgresql:jar:42.2.5 (Optional) [a]	**jdbc:postgresql://*HOST*:5432/*DATABASE_NAME***	**org.postgresql.Driver** (Optional)	N/A

[a] When you run the Data Virtualization Operator to build a virtual database that uses a PostgreSQL source, the build process automatically retrieves the required JDBC driver. It is not required to also define the driver dependency in the CR.

For an example that shows how these properties are specified in a CR, see Section 5.5, "Relational databases data sources configuration".

5.6. CONFIGURATION PROPERTIES FOR USING A REST SERVICE AS A DATA SOURCE

You can configure a REST service as a data source for a virtual database.

A common set of data source connection properties is required for all REST-based data sources. In addition to the common properties, services that are based on specific REST-based standards, such as OData or OpenAPI, require specific translators.

By default, translators are unable to parse the security configuration of a secured API. To enable translators to access data for a secured API, the CR must specify the security properties for the API.

Translator setting

For generic services that use REST directly, and that are not based on particular specifications, set the translator in the DDL for the virtual database to *rest* with a matching **SERVER** definition. Generic REST-based services lack built-in mechanisms for passing SQL query conditions to a REST API endpoint. As a result, the data virtualization service cannot automatically convert query criteria for these services into query parameters.

To pass SQL queries as XML or JSON payloads to the endpoints of these services, you must use the **invokeHttp** procedure, and use it to specify your query strings and headers.

Some REST-based data sources, such as OData, OpenAPI, and SOAP have specific translators that are based on the REST configuration.

For more information, see *Rest translator* in the Data Virtualization Reference.

The following tables show the data source information that is required in the data source properties of the custom resource:

Table 5.13. Data source properties for REST

Property Name	Description	Required	Default value
endpoint	Endpoint for the service.	Yes	n/a

securityType	Security type to use. Available options are *http-basic*, *openid-connect* or empty.	No	no security

If the *security type* is defined as *http_basic* you must also set the following properties:

Table 5.14. HTTP basic properties for REST data sources

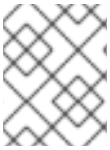
Property name	Description	Required	Default value
userName	User name	Yes	n/a
password	Password	Yes	n/a

If the *security type* is defined as *openid-connect*, you must set the following properties:

Table 5.15. OpenID Connect properties for REST data sources

Property Name	Description	Required	Default value
userName	User name	Yes	n/a
password	Password	Yes	n/a
clientId	ClientId from connected app.	Yes	n/a
clientSecret	clientSecret from connected app.	Yes	n/a
authorizeUrl	clientSecret from connected app.	Yes	n/a
accessTokenUrl	clientSecret from connected app.	Yes	n/a
scope	clientSecret from connected app.	No	n/a

Alternatively, for *openid-connect* you can specify the **refreshToken** property and avoid using the **userName** and **password** properties. The process obtaining a refresh token differs for different services. Describing how to obtain refresh tokens is beyond the scope of this document.



NOTE

To enable communications with REST data source endpoints over secure HTTP (HTTPS), you must have a truststore configured for the endpoint.

For information about configuring a custom TLS certificates, see xref:

For a complete list of the properties that you can set to control how data is translated between REST-based services and a virtual database, see the OData, OData V4, OpenAPI, and Web Services translator sections in the Data Virtualization Reference.

5.7. CONFIGURATION PROPERTIES FOR ODATA AS A DATA SOURCE

You can configure OData as a data source for a virtual database.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For OData sources, set the translator to *odata*. For an OData V4 service, use *odata4* A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

Because OData services are based on REST, they follow the same properties model as REST-based connections.

The following configuration showing **openid_connect** security type with a OData service

A sample configuration

```
datasources:
  - name: sampledb
    type: odata4
    properties:
      - name: endpoint
        value: https://dv-customer-myproject.apps-crc.testing/odata/accounts/customer
      - name: securityType
        value: openid-connect
      - name: clientId
        value: dv
      - name: clientSecret
        value: xxxxxxxxxxx
      - name: authorizeUrl
        value: https://keycloak-myproject.apps-crc.testing/auth/realms/master/protocol/openid-
connect/auth
      - name: accessTokenUrl
        value: https://keycloak-myproject.apps-crc.testing/auth/realms/master/protocol/openid-
connect/token
```

5.8. CONFIGURATION PROPERTIES FOR OPENAPI AS A DATA SOURCE

You can configure an OpenAPI service as a data source for a virtual database.

Because OpenAPI services are based on REST, they follow the same properties model as REST-based connections.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For OpenAPI sources, set the translator to *openapi*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

The *openapi* translator assumes that the endpoint in the API document is set to the target location /**openapi**, and it builds a source model that is based on that assumption.

If the API endpoint is set to a different target, a configuration setting must be specified so that the translator can locate the endpoint and import data correctly. The following examples show a DDL **SCHEMA** statement and an environment variable that you can set to specify the non-standard endpoint, /**swagger**.

DDL SCHEMA statement for defining a non-standard OpenAPI endpoint

```
CREATE SCHEMA sourceModel SERVER oService OPTIONS ("importer.metadataUrl"
'/swagger.json');
```

Example: Sample configuration that defines a non-standard OpenAPI endpoint

```
datasources:
  - name: sampledb
    type: openapi
    properties:
      - name: userName
        value: user
      - name: password
        value: pass
      - name: importer.metadataUrl
        value: /swagger.json
```

If the API is secured, the translator is unable to process the security configuration of the service automatically. The translator understands only the API document and its responses. To process security settings properly you must define them as REST properties, as described in Section 5.6, “Configuration properties for using a REST service as a data source”.

5.9. CONFIGURATION PROPERTIES FOR SALESFORCE AS A DATA SOURCE

You can configure Salesforce as a data source for a virtual database.

Salesforce uses OAuth 2.0 for authentication and authorization. Before you can set up a virtual database to import and query Salesforce data, you must obtain OAuth credentials for the virtual database from Salesforce. For information about how to set up OAuth, see Section 5.9.1, “Setting up an OAuth connection to Salesforce”

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For Salesforce sources, set the translator to *salesforce*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

Dependencies

Salesforce does not require you to specify any build dependencies.

The following tables list the properties that are required in the custom resource to create a virtual database that is based on a Salesforce database:

Table 5.16. Data source properties for Salesforce

Property Name	Description	Required	Default value
url	URL for salesforce.	No	https://login.salesforce.com/services/Soap/u/45.0
username	User account for salesforce.com.	Yes	n/a
password	Password for salesforce.com.	Yes	n/a
clientId	ClientId from connected app.	Yes	n/a
clientSecret	clientSecret from connected app.	No	n/a
refreshToken	Refresh Token [a]	No	n/a

[a] If your connected app uses refresh tokens to authenticate, rather than name and password, you must define the **refreshToken** property in the CR, in place of the user name and password properties. Information about obtaining refresh tokens is beyond the scope of this document. For information about how to obtain a refresh token for your connected app, see the Salesforce documentation.

The following example shows a configuration that uses simple user name and password login.

Example: Excerpt from a virtual database custom resource that connects to Salesforce by using name and password authentication

```
datasources:
  - name: sampledb
    type: salesforce
    properties:
      - name: userName
        value: user
      - name: password
        value: pass
```

You can obtain the **clientId** and **clientSecret** from Salesforce when you create your Salesforce application.

For a complete list of the properties that you can set to control how data is translated between Salesforce and a virtual database, see the Data Virtualization Reference.

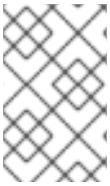
5.9.1. Setting up an OAuth connection to Salesforce

Before the data virtualization service can retrieve data from a Salesforce database, you must enable configure it as a connected app in Salesforce that is OAuth-enabled. After you configure OAuth, Salesforce generates a client ID and client secret that you must add to the CR file that defines the connection from the virtual database to Salesforce.

To configure OAuth you create a connected app in Salesforce that can request access to Salesforce data on behalf of the data virtualization service. In the settings for the connected app, you enable integration with the Salesforce API by using the OAuth 2.0.

Prerequisites

- You have a Salesforce.com account that has access to the data that you want to integrate in a virtual database.



NOTE

The following steps are based on Salesforce Classic. If you use a different version of Salesforce, you might use a different procedure. For more information about creating connected apps in Salesforce, see the Salesforce documentation.

Procedure

1. From Salesforce, log into your account.
2. Click **SetUp** in the profile menu.
3. In the **Build** section of the navigation sidebar, expand **Create**, and then click **Apps**.
4. In the **Connected Apps** section, click **New**.
5. Complete the required fields.
6. In the section **API (Enable OAuth Settings)**, select **Enable OAuth Settings** to display the OAuth settings.
7. Complete the required OAuth fields. In the **OAuth Scopes** field, you must select the following scopes:
   - Access and manage your data (api).
   - Access your basic information (id, profile, email, address, phone).
   - Allow access to your unique identifier (openid).
   - Full access (full).
   - Perform requests on your behalf at any time (refresh_token, offline_access).
8. Select **Require Secret for Web Server Flow**
9. Click **Save** and then click **Continue**.
10. Make a note of the values in the **Consumer Key** and **Consumer Secret** fields. These values are required for properties in the CR that specifies how the virtual database connects to Salesforce.

5.10. CONFIGURATION PROPERTIES FOR USING FTP/SFTP AS A DATA SOURCE

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For FTP sources, set the translator to *ftp*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

NOTE

To enable secure transmission over SFTP you must provide a TLS certificate. For more information about how to use certificates with data virtualization on OpenShift, see Section 7.1, "Certificates and data virtualization"

The following table shows the information that is required in the data source properties of the custom resource:

Table 5.17. Data source properties for SFTP

Property Name	Description	Required	Default value
host	Host name of the FTP server.	yes	n/a
port	Port of the FTP server.	No	21
username	User for remote server login	Yes	n/a
password	Password for remote server login.	Yes	n/a
parentDirectory	Directory that contains file data.	Yes	n/a
isFtps	FTP security.	No	false

Example: Excerpt from an FTP/SFTP custom resource, showing the format for setting key properties

```
datasources:
  - name: sampleftp
    type: ftp
    properties:
      - name: host
        value: localhost
      - name: parent-directory
        value: /path/to/file/
      - name: username
```

```
    value: user
  - name: password
    value: pass
```

5.11. CONFIGURATION PROPERTIES FOR SOAP AS A DATA SOURCE

You can configure SOAP as a data source for a virtual database.

Translator setting

The DDL for the virtual database defines a translator, or **FOREIGN DATA WRAPPER**. For SOAP sources, set the translator to *soap* or *ws*. A corresponding **SERVER** definition in the DDL represents the external data source server, and associates the translator with the external server.

The Web services or SOAP translator exposes stored procedures for calling web or SOAP services. Results from this translator are typically used with the TEXTTABLE or XMLTABLE table functions to process data formatted in CSV or XML.

Dependencies

SOAP data sources do not require you to specify any build dependencies.

The following table shows the data source information that is required in the data source properties of the custom resource:

Table 5.18. Data source properties for SOAP

Property Name	Applies to	Required	Default Value	Description
EndPoint	HTTP and SOAP	false	n/a	URL for HTTP; service endpoint for SOAP. Not required if using HTTP to invoke procedures that specify absolute URLs. Used as the base URL if an invoked procedure uses a relative URL.
SecurityType	HTTP and SOAP	false	none	Type of authentication to use with the web service. Allowed values [**None**,**HTTPBasic**]
AuthUserName	HTTP and SOAP	false	n/a	Name value for authentication, used in HTTPBasic and WsSecurity.

Property Name	Applies to	Required	Default Value	Description
AuthPassword	HTTP and SOAP	false	n/a	Password value for authentication, used in HTTPBasic and WsSecurity.
ConfigFile	HTTP and SOAP	false	n/a	CXF client configuration file or URL.
EndPointName	HTTP and SOAP	false	teiid	Local part of the endpoint QName to use with this connection. Must match the one defined in cxf file
ServiceName	SOAP	false	n/a	Local part of the service QName to use with this connection.
NamespaceUri	SOAP	false	http://teiid.org	Namespace URI of the service QName to use with this connection.
RequestTimeout	HTTP and SOAP	false	n/a	Timeout for request.
ConnectTimeout	HTTP and SOAP	false	n/a	Timeout for connection.
Wsdl	SOAP	false	n/a	WSDL file or URL for the web service.

Example: Excerpt from an SOAP custom resource, showing the format for setting key properties

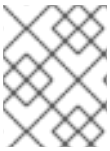
```
datasources:
  - name: soapCountry
    type: soap
    properties:
      - name: wsdl
        value: http://www.oorsprong.org/websamples.countryinfo/CountryInfoService.wso?WSDL
      - name: namespaceUri
        value: http://www.oorsprong.org/websamples.countryinfo
      - name: serviceName
```

```
    value: CountryInfoService
  - name: endPointName
    value: CountryInfoServiceSoap12
```

CHAPTER 6. RUNNING THE DATA VIRTUALIZATION OPERATOR TO DEPLOY A VIRTUAL DATABASE

The Data Virtualization Operator helps to automate the configuration and deployment of virtual databases.

The Operator processes a virtual database custom resource (CR) to deploy a virtual database object on OpenShift. By running the operator with different CRs, you can create virtual databases from a range of data sources.



NOTE

Virtual databases that you deploy to OpenShift in this Technology Preview are not available from Fuse Online.

6.1. INSTALLING THE DATA VIRTUALIZATION OPERATOR ON OPENSHIFT

Install the Data Virtualization Operator so that you can use it to deploy virtual database images to OpenShift from YAML-based custom resources (CRs).

You can install the data virtualization operator from the OperatorHub on OpenShift 4.2 and later.

After you add the operator to your OpenShift cluster, you can use it to build and deploy virtual database images from a range of data sources.

Prerequisites

- You have cluster-admin access to an OpenShift 4.2 or greater cluster.
- You have access to the OpenShift 4.2 or greater web console.
- You have Developer access to an OpenShift server and you are familiar with using the OpenShift console and CLI.

Procedure

1. From a terminal window, type the following commands to log in to the OpenShift cluster and create a pull secret that you can use to access the Red Hat image registry:

```
oc login
oc create secret docker-registry dv-pull-secret /
--docker-server=registry.redhat.io /
--docker-username=$username / (1)
--docker-password=$password /
--docker-email=$email_address
oc secrets link builder dv-pull-secret
oc secrets link builder dv-pull-secret --for=pull
```

(1) Use your Red Hat Customer Portal login credentials.

2. Log in to the OpenShift web console as a cluster administrator.

3. From the OpenShift menu, expand **Operators** and click **OperatorHub**.
4. Click **Red Hat Integration - Data Virtualization**, and then click **Install**.
5. From the **Create Operator Subscription** page, verify that the selected namespace matches the name of the project where you want to install the operator, and then click **Subscribe**. The **Installed Operators** page lists the **Data Virtualization Operator** and reports the status of the installation.
6. From the OpenShift menu, expand **Workloads** and click **Pods** to check the status of the Operator pod. After a few minutes, the pod for the Operator service begins to run.
7. To enable the data virtualization Operator to retrieve images from the Red Hat registry so that you can create virtual databases, link the secret that you created in Step 1 to the service account for the Operator.

```
oc secrets link dv-operator dv-pull-secret --for=pull
```

Additional resources

- Section 6.2, “Deploying virtual databases”.

6.2. DEPLOYING VIRTUAL DATABASES

After you create a virtual database and its corresponding CR file, run the Data Virtualization Operator to deploy the database to Openshift.

Prerequisites

- A cluster administrator added the Data Virtualization Operator to the OpenShift cluster where you want to deploy the virtual database.
- You have access to an OpenShift cluster in which the Data Virtualization Operator is installed.
- You have a CR in YAML format that provides information about how to configure and deploy the virtual database.
- The Operator has access to the Maven repositories that contain the dependencies that the build requires.
- OpenShift can access the data source that is referenced in the CR.

Procedure

1. From a terminal window, log in to OpenShift and open the project where you want to create the virtual database.
2. On your computer, change to the directory that contains the **.yaml** file that contains the CR.
3. Type the following command to run the operator to create the virtual database:

```
oc create -f <cr_filename.yaml>
```

Replace **<cr_filename.yaml>** with the name of the CR file for your data source. For example,

```
oc create -f dv-customer.yaml
```

After the deployment completes, a virtual database service is added to the OpenShift cluster. The name of the service matches the name that is specified in the custom resource.

4. Type the following command to verify that the virtual database is created:

```
oc get vdbs
```

OpenShift returns the list of virtual databases in the project.

5. To see whether a particular virtualization is available, type the following command:

```
oc get vdb <dv-name>
```

The deployed service supports connections from the following clients:

- JDBC clients through port 31000.
- postgreSQL clients, including ODBC clients, through port 5432.
- OData clients, through an HTTP endpoint and route.

CHAPTER 7. SECURING DATA

To prevent unauthorized access to data, you can implement the following measures:

- Encrypt communications between the virtual database and other database clients and servers.
- Use OpenShift secrets to store the values of properties.
- Configure integration with Red Hat Single Sign-On in OpenShift to enable OpenID-Connect authentication and OAuth2 authorization.
- Apply role-based access controls to your virtual database.
- Configure 3Scale to secure OData API endpoints.

7.1. CERTIFICATES AND DATA VIRTUALIZATION

You can use TLS to secure communications between the data virtualization service and other services. For example, you can use TLS to encrypt the traffic that the service exchanges during the following operations:

- Responding to queries from JDBC and postgreSQL clients.
- Responding to calls from REST or OData APIs over HTTPS.
- Communicating with a Keycloak/RH-SSO server.
- Communicating with SFTP data sources.

Certificate types

To encrypt traffic, you must add a TLS certificate for the virtual database service to the cluster. You can use either of two type of certificates to configure encryption. The certificate can be either a self-signed service certificate that is generated by the OpenShift certificate authority, or a custom certificate from a trusted third-party Certificate Authority (CA). If you use a custom certificate, you must configure it before you build and deploy the virtual database.

Certificate scope

After you configure a certificate for the data virtualization service, you can use the certificate for all of the data virtualization operations within the OpenShift cluster.

7.1.1. Service-generated certificates

Service certificates provide for encrypted communications with internal and external services alike. However, only internal services, that is, services that are deployed in the same OpenShift cluster, can validate the authenticity of a service certificate.

OpenShift service certificates have the following characteristics:

- Consist of a public key certificate (**tls.crt**) and a private key (**tls.key**) in PEM base-64-encoded format.
- Stored in an encryption secret in the OpenShift pod.
- Signed by the OpenShift CA.

- Valid for one year.
- Replaced automatically before expiration.
- Can be validated by internal services only.

Using a service-generated certificate is the simplest way to secure communications between a virtual database and other applications and services in the cluster. When you run the Data Virtualization Operator to create a virtual database, it checks for the existence of a secret that has the same name as the virtual database that is defined in the CR, for example, ***VDB_NAME*-certificates.**.

If the Operator detects a secret with a name that matches the virtual database name, it converts certificate and key in the secret into a Java Keystore. The Operator then configures the Keystore for use with the virtual database container that it deploys.

If the Operator does not find a secret with the name of the virtual database, it creates a *service-generated certificate* files in PEM format to define the public key certificate and private encryption key for the service. Also known as a service serving certificates, a service-generated certificate originates from the OpenShift Certificate Authority.

The following certificate files are created:

- **tls.crt** - TLS public key certificate
- **tls.key** - TLS private encryption key

The Operator stores the generated certificate files in a secret with the name of the virtual database: ***VDB_NAME*-certificates.**

When the certificate and key are converted to a Keystore, the Operator also adds the default Truststore from the Kubernetes **/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt** certificate to the cluster. After the Keystore and Truststore are deployed, the virtual database service can communicate securely with other services in the cluster, as long as they use service-based certificates. However, the virtual database cannot exchange secure communications with services that are use other certificate types, such as custom certificates, or other types of self-signed certificates. To enable communication with these services, and with services that hosted outside of the cluster, you must configure the service to use custom certificates.

Additional resources

For more information about service serving certificates in OpenShift, see the OpenShift Authentication documentation.

7.1.2. Custom certificates

To support secure communications between the virtual database service and applications outside of the OpenShift cluster, you can obtain custom certificates from a trusted third-party Certificate Authority.

External services do not recognize the validity of certificates that are generated by the OpenShift certificate authority. For an external services to validate custom TLS certificates, the certificates must originate from a trusted, third-party certificate authority (CA). Such certificates are universally recognized, and can be verified by any client. Information about how to obtain a certificate from a third-party CA is beyond the scope of this document.

You can add custom certificates to a virtual database by supplying information about the certificate in an encryption secret and deploying the secret to OpenShift before you run the Data Virtualization Operator to create the service. After you deploy an encryption secret to OpenShift, it becomes available

to the Data Virtualization Operator when it creates a virtual database. The Operator detects the secret with the name that matches the name of the virtual database in the CR, and it automatically configures the virtual database service to use the specified certificate to encrypt communications with other services.

7.1.3. Using custom TLS certificates to encrypt communications between a virtual database and other services

Data virtualization uses TLS certificates to encrypt network traffic between a virtual database service and the clients or data sources that the service communicates with.

To ensure that both internal and external services can authenticate the CA signature on the TLS certificates, the certificate must originate from trusted third-party certificate authorities (CA).

To configure the data virtualization service to use a custom certificate to encrypt traffic, you add the certificate details to an OpenShift secret and deploy the secret to the namespace where you want to create the virtual database. You must create the secret before you create the service.

To use custom TLS certificates, you first generate a keystore and truststore from the certificates. You then convert the binary content of those files into Base64 encoding. After you convert the content, you add the content to a secret and deploy it to OpenShift.

7.1.4. Creating a keystore from the private key and public key certificate

To support secure communications between the virtual database service and applications outside of the OpenShift cluster, you can obtain custom certificates from a trusted third-party Certificate Authority.

Adding a custom keystore and truststore for your TLS certificates provides the most flexible framework for securing communications, enabling you to define a configuration that works in any situation.

You must complete the following general tasks:

- Obtain a TLS certificate from a CA.
- Build a keystore in PKCS12 format from the public certificate and private key in the CA certificate.
- Convert the keystore to Base64 encoding.

Prerequisite

- You have a TLS certificate from a trusted, third-party CA.
- You have the private key (**tls.key**) and public key certificate (**tls.crt**) from the TLS certificate in PEM format.

Procedure

1. Taking as input the **tls.key** and **tls.crt** from the certificate in PEM format, run the following command to create a keystore.

   ```
   openssl pkcs12 -export -in tls.crt -inkey tls.key -out keystore.pkcs12
   ```

   The following table describes the elements of the openssl command:

Table 7.1. openssl command to generate a keystore from certificate files

Command element	Description
pkcs12	The openssl pkcs12 utility.
-export	Exports the file.
-in tls.crt	Identifies the certificate file.
-inkey tls.key	Identifies the key to import into the keystore.
-out keystore.pkcs12	Specifies the name of the keystore file to create.

2. From the **keystore.pkcs12** file that you generated in the previous step, type the following command to convert the file to Base64 encoding:

```
base64 keystore.pkcs12
```

3. Copy the contents of the command output to a YAML secret file.
For more information, see Section 7.1.7, "Creating an OpenShift secret to store the keystore and truststore".

7.1.5. Creating a truststore from the public key certificate

Prerequisite

- You have the Java 11 or later version of the **keytool** key and certificate management utility, which uses PKCS12 as the default format.
- You have a public key certificate (**tls.crt**) in PKCS12 format.

Procedure

In the steps that follow, after you generate Base64 encodings for the keystore and truststore keys, add the content to the YAML file.

1. From a terminal window, using the public key certificate, **tls.crt**, type the following command:

```
keytool -import -file tls.crt -alias myalias -keystore truststore.pkcs12
```

2. Type the following command to convert the output to Base64 encoding:

```
base64 truststore.pkcs12
```

3. Copy the contents of the command output and paste it into the secret.

7.1.6. Adding the keystore and truststore passwords to the configuration

To use the custom keystores, you must specify the passwords to use in virtual database operations. Provide the passwords as environment properties in the custom resource for the virtual database by setting the following properties.

- **TEIID_SSL_KEYSTORE_PASSWORD**
- **TEIID_SSL_TRUSTSTORE_PASSWORD**
- **KEYCLOAK_TRUSTSTORE_PASSWORD** (For use with Red Hat SSO/Keycloak, if the trust manager is not disabled)

The following example shows an excerpt from a virtual database CR in which the preceding variables are defined:

Example: Custom resource showing environment variables to define passwords for certificate keystore and truststores

```
apiVersion: teiid.io/v1alpha1
kind: VirtualDatabase
metadata:
  name: dv-customer
spec:
  replicas: 1
  env:
  - name: TEIID_SSL_KEYSTORE_PASSWORD
    value: KEYSTORE_PASSWORD
  - name: TEIID_SSL_TRUSTSTORE_PASSWORD
    value: TRUSTSTORE_PASSWORD
  - name: KEYCLOAK_TRUSTSTORE_PASSWORD
    value: SSO_TRUSTSTORE_PASSWORD

  ... additional content removed for brevity
```

After you configure the cluster to use the preceding certificates in the keystore and truststore, the virtual database can use the certificates to encrypt or decrypt communications with services.

After you deploy the TLS secret to OpenShift, run the Data Virtualization Operator to create a virtual database with the name that is specified in the secret. For more information, see Section 6.2, “Deploying virtual databases”.

When the Operator creates the virtual database, it searches for a secret that has a name that matches the name specified in the CR for the virtual database service. If it finds a matching secret, the Operator configures the service to use the secret to encrypt communications between the virtual database service and other applications and services.

7.1.7. Creating an OpenShift secret to store the keystore and truststore

- Create a YAML file
- Add the keystore to an OpenShift secret.
- Deploy the secret to OpenShift.

1. Create a YAML file to define a secret of type **Opaque** with the name **{vdb-name}-keystore**, and include the following information: +

- The keystore and truststore generated from a TLS key pair, in Base64 encoding.
- The name of the virtual database that you want to create.
- The OpenShift namespace in which you want to create the virtual database.

Prerequisites

- You have Developer or Administrator access to the OpenShift project where you want to create the secret and virtual database.
- You have the keystore and truststore in Base64 format that you generated from a custom TLS certificate.

Procedure

1. Create a YAML file to define a secret of type **kubernetes.io/tls**, and include the following information:
   - The public and private keys of the TLS key pair.
   - The name of the virtual database that you want to create.
   - The OpenShift namespace in which you want to create the virtual database.
     For example:

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: _VDB_NAME-keystore (1)
  namespace: PROJECT_NAME (2)

data: (3)
  keystore.pkcs12: >-
     -----BEGIN KEYSTORE-----
     [...]
     -----END KEYSTORE-----
  truststore.pkcs12: >-
     -----BEGIN TRUSTSTORE-----
     [...]
     -----END TRUSTSTORE-----
```

(1) The name of the secret. The secret name must match the name of the virtual database object in the CR YAML file that the Data Virtualization Operator uses to create a virtual database, for example, **dv-customer**.

(2) The OpenShift namespace in which the virtual database service is deployed, for example, **myproject**.

(3) The **data** value is made up of the contents of the TLS keystore certificate (**keystore.pkcs12**), and the truststore (**truststore.pkcs12**) in base64-encoded format.

2. Save the file as **dv-customer-keystore.yaml**.

3. Open a terminal window, sign in to the OpenShift project where you want to add the secret, and then type the following command:

```
$ oc apply -f tls_secret.yaml
```

The following example shows a YAML file for creating an OpenShift secret to store a keystore and truststore.

Example: dv-customer-keystore.yaml file

An example script to create the secret will be like below

dv-customer-keystore.yaml

```
kind: Secret
apiVersion: v1
metadata:
  name: dv-customer-keystore (1)
  namespace: my-project
data:
  keystore.pkcs12: >-

MIIKAwIBAzCCCc8GCSqGSIb3DQEHAaCCCcAEggm8MIIJuDCCBG8GCSqGSIb3DQEHBqCCBGA
wggRcAgEAMIIEVQYJKoZIhvcNAQcBMBwGCiqGSIb3DQEMAQYwDgQI7RFjrbx64PkCAggAgIIEKLn
2Y244Jw2O37QlmT+uS3XE0vErwJIIwpwR8nlu8YDPTU8UtN3nDXNkdKbolQVTCVnzFSbJ7RohoAEJ
dB3D8iDkVpFpvIbBYUvq3LB8obFuSuFP50IMprp9gnUjRit5/nOGdJIKiM3ZJQgE46gvYsQJiKo0CGlBf/
7J9CWh/zwp7fV4JzxZaW/4bkUaz1jegPx0lYEPW14U1INF0BCn0DnTffCHnSqQIIW+KwNj3uNtqVqLTt
4LoLTbfvCjYN+6q+0Ei67a05g8X2ZI7y7LJvfRGlAssVwqOlOMeiwajOtsJXXaN1WsZjURFVIJpmlWAcG/
72J9xUlA5YYUzdxI8GdaQis78b0lsvYPU0WqCMBmfoJMxhQuIfpZVqDgqTTaJhvrv7lw/VYLyJKG1N0pA
Q9dDnWUtje7MB+Q853ffjzZ5PDp8G+BoxUrsxABhesII8PwIb0JG66yxyBmgvpxlGVN1IyHLKZzn3/RUC
A8/WLjuD1SAmFQxfDoOir1LEnodXLEVLH+8/Ety0xvP5T9BFn/YVsPSjplhukkdfqiHDqxffg8aJlpfOC8AJ
4EVItb/W8fBQovQ+jhm1LpuQedA6fiaROYYHChaQM94y9HqPIveCEpKGkG47ohGWU/LCht/Da3iHhl6
h9BCX/U/PcsojKy8ZmzZTJb+oIRCx+A84X/hObGoqU+dOItQ//G37BIL7jIcQ9gwShtQhXmdCtrh10iNK
GxaDxyBBJS64+KeuAv16eyj3UHoR3Ux9P3RVzZ6bH+IrKsWRacg+JYzEZNAzo0NYkVCqgvbdC+fWDt
q6rQA2knjRhwwK/WU/=
  truststore.pkcs12: >-

MIIWlgIBAzCCFmIGCSqGSIb3DQEHAaCCFlMEghZPMIIWSzCCFkcGCSqGSIb3DQEHBqCCFjgwgh
Y0AgEAMIIWLQYJKoZIhvcNAQcBMBwGCiqGSIb3DQEMAQYwDgQIq4NIOxI8IoUCAggAgIIWAN8YK
MvjIo6qGX2Rz0SIKiDlUNySI5GKjt1RKicid9QIVfyKjWhjufqn+OXjhaxYJtZ+GgW3SdO1il0cHEGSQycEJ
PQ/diAMqmdgoyd1batEYxp1baR9wm4aqmYip0j3Xd84fpQylTs73tFOZYWJYPDqq27jYYbEUL0bOKko
MOvIftW6y18gT/E8XVYi7Gy81IJzNnhQkyt4bZO3/vyoEgvyUDGLCtFxSk4U9JiGk3RtzLW1HnOiGof1B/I
Js7vHe13QITJWqxhqKs4rWYj8pOiyrIhAcLtGMEUH9cyQ7gpYFvx5KObY//gEDr2MnRdR4cm79wuffg9
mUH96hvqwrm/dpJC1IP+dRM/9Alyn9KEuEiIWaUOxkHobvcCs04fh2Fw8GS4wdCAiB7Rj3e2U1duWdg
3MJ5Qxq4SVEZeTPkDKetqZTTWpzDiw8nxgZx7MGYAQ5kIYeWHWzVs9fFDuNFTnvhEb535KMz6qZ
YMjJdiZRVhX5XyCKyLyiBovQsdHDUkuubroJfUFe3VI7FNGNVJ1OIuqrIVJVYIpqER6khWoCOizm/L1P
WU8XS6fsR3ES296VaukzAyewQIpQhEek9XjRY=
type: Opaque
```

(1) **dv-customer** is name of the virtual database that will use the keystore.

Based on the format of the preceding example, create a secret file with the content from your keystore and truststore.

1. Type the following command to add the secret to OpenShift:

```
oc create -f dv-customer-keystore.yaml
```

7.2. USING SECRETS TO STORE DATA SOURCE CREDENTIALS

Create and deploy secret objects to store values for your environment variables.

Although secrets exist primarily to protect sensitive data by obscuring the value of a property, you can use them to store the value of any property.

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 YAML 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

3. Set an environment variable to retrieve its value from the secret.

   a. In the environment variable, use the format **valueFrom:/secretKeyRef** to specify that the variable retrieves it value from a key in the secret that you created in Step 1.
   For example, in the following excerpt, the **SPRING_DATASOURCE_SAMPLEDB_PASSWORD** retrieves its value from a reference to the **database-password** key of the **postgresql** secret:

   ```
   - name: SPRING_DATASOURCE_SAMPLEDB_PASSWORD
     valueFrom:
       secretKeyRef:
         name: postgresql
         key: database-password
   ```

The following example shows the use of secrets to define the **username** and **password** properties for a postgreSQL datbase.

Sample data source configuration that uses secrets to define properties

```
datasources:
  - name: sampledb
    type: postgresql
    properties:
      - name: username
        valueFrom:
          secretKeyRef:
            name: sampledb-secret
            key: username
      - name: password
        valueFrom:
          secretKeyRef:
            name: sampledb-secret
            key: password
      - name: jdbc-url
        value: jdbc:postgresql://database/postgres
```

In the preceding example, **sampledb** denotes a custom name that is assigned to the source database. The same name would be assigned to the **SERVER** definition for the data source in the DDL for the virtual database. For example, **CREATE SERVER sampledb FOREIGN DATA WRAPPER postgresql**.

Additional resources

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

7.3. SECURING ODATA APIS FOR A VIRTUAL DATABASE

You can integrate data virtualization with Red Hat Single Sign-On and Red Hat 3scale API Management to apply advanced authorization and authentication controls to the OData endpoints for your virtual database services.

The Red Hat Single Sign-On technology uses OpenID-Connect as the authentication mechanism to secure the API, and uses OAuth2 as the authorization mechanism. You can integrate data virtualization with Red Hat Single Sign-On alone, or along with 3scale.

By default, after you create a virtual database, the OData interface to it is discoverable by 3scale, as long as the 3scale system is defined to same cluster and namespace. By securing access to OData APIs through Red Hat Single Sign-On, 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. Only authorized users can access the API endpoint, and each user is permitted a level of access that is appropriate to their role (role-based access). By using 3scale as a gateway to your API, you can take advantage of 3scale's API management features, allowing you to tie API usage to authorized accounts for tracking and billing.

When a user logs in, 3scale negotiates authentication with the Red Hat Single Sign-On 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

- Red Hat Single Sign-On is running in the OpenShift cluster. For more information about deploying Red Hat Single Sign-On, see the Red Hat Single Sign-On for OpenShift documentation.
- You have Red Hat 3scale API Management installed in the OpenShift cluster that hosts your virtual database.
- You have configured integration between 3scale and Red Hat Single Sign-On. For more information, see Configuring Red Hat Single Sign-On integration in Using the Developer Portal.
  - 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.

7.3.1. Configuring Red Hat Single Sign-On to secure OData

You must add configuration settings in Red Hat Single Sign-On to enable integration with data virtualization.

Prerequisites

- Red Hat Single Sign-On is running in the OpenShift cluster. For information about deploying Red Hat Single Sign-On, see the link:Red Hat Single Sign-On for OpenShift[Red Hat Single Sign-On] documentation.
- You run the Data Virtualization Operator to create a virtual database in the cluster where Red Hat Single Sign-On is running.

Procedure

1. From a browser, log in to the Red Hat Single Sign-On Admin Console.
2. Create a realm for your data virtualization service.
   a. From the menu for the master realm, hover over **Master** and then click **Add realm**.
   b. Type a name for the realm, such as **datavirt**, and then click **Create**.
3. Add roles.
   a. From the menu, click **Roles**.
   b. Click **Add Role**.
   c. Type a name for the role, for example **ReadRole**, and then click **Save**.
   d. Create other roles as needed to map to the roles in your organization's LDAP or Active Directory. For information about federating user data from external identity providers, see the Server Administration Guide.
4. Add users.

a. From the menu, click **Users**, and then click **Add user**.

b. On the **Add user** form, type a user name, for example, **user**, specify other user properties that you want to assign, and then click **Save**.
Only the user field is mandatory.

c. From the details page for the user, click the **Credentials** tab.

d. Type and confirm a password for the user, click **Reset Password**, and then click **Change password** when prompted.

5. Assign roles to the user.

a. Click the **Role Mappings** tab.

b. In the **Available Roles** field, click **ReadRole** and then click **Add selected**.

6. Create a second user called **developer**, and assign a password and roles to the user.

7. Create a data virtualization client entry.
The client entry represents the data virtualization service as an SSO client application. .. From the menu, click **Clients**. .. Click **Create** to open the **Add Client** page. .. In the **Client ID** field, type a name for the client, for example, **dv-client**. .. In the **Client Protocol** field, choose **openid-connect**. .. Leave the **Root URL** field blank, and click **Save**.

You are now ready to add SSO properties to the CR for the data virtualization service.

7.3.2. Adding SSO properties to the custom resource file

After you configure Red Hat Single Sign-On to secure the OData endpoints for a virtual database, you must configure the virtual database to integrate with Red Hat Single Sign-On. To configure the virtual database to use SSO, you add SSO properties to the CR that you used when you first deployed the service (for example, **dv-customer.yaml**). You add the properties as environment variables. The SSO configuration takes effect after you redeploy the virtual database.

In this procedure you add the following Red Hat Single Sign-On properties to the CR:

Realm (KEYCLOAK_REALM)

The name of the realm that you created in Red Hat Single Sign-On for your virtual database.

Authentication server URL (KEYCLOAK_AUTH_SERVER_URL)

The base URL of the Red Hat Single Sign-On server. It is usually of the form https://host:port/auth.

Resource name(KEYCLOAK_RESOURCE)

The name of the client that you create in Red Hat Single Sign-On for the data virtualization service.

SSL requirement (KEYCLOAK_SSL_REQUIRED)

Specifies whether requests to the realm require SSL/TLS. You can require SSL/TLS for all requests, external requests only, or none.

Access type (KEYCLOAK_PUBLIC_CLIENT)

The OAuth application type for the client. Public access type is for client-side clients that sign in from a browser.

Prerequisites

- You ran the Data Virtualization Operator to create a virtual database.

- Red Hat Single Sign-On is running in the cluster where the virtual database is deployed.
- You have the CR YAML file, for example, **dv-customer.yaml** that you used to deploy the virtual database.
- You have have administrator access to the Red Hat Single Sign-On Admin Console.

Procedure

1. Log in to the Red Hat Single Sign-On Admin Console to find the values for the required authentication properties.
2. In a text editor, open the CR YAML file that you used to deploy your virtual database, and define authentication environment variables that are based on the values of your Red Hat Single Sign-On properties.
   For example:

```
env:
  - name: KEYCLOAK_REALM
    value: master
  - name: KEYCLOAK_AUTH_SERVER_URL
    value: http://rh-sso-datavirt.openshift.example.com/auth
    - name: KEYCLOAK_RESOURCE
      value: datavirt
  - name: KEYCLOAK_SSL_REQUIRED
    value: external
  - name: KEYCLOAK_PUBLIC_CLIENT
    value: true
```

3. Declare a build source dependency for the following Maven artifact for securing data virtualizations: **org.teiid:spring-keycloak**
   For example:

```
env:
  ....
  build:
    source:
      dependencies:
        - org.teiid:spring-keycloak
```

4. Save the CR.

You are now ready to define data roles in the DDL for the virtual database.

7.3.3. Defining data roles in the virtual database DDL

After you configure Red Hat Single Sign-On to integrate with data virtualization, to complete the required configuration changes, define role-based access policies in the DDL for the virtual database. Depending on how you deployed the virtual database, the DDL might be embedded in the CR file, or exist as a separate file.

You add the following information to the DDL file:

- The name of the role. Roles that you define in the DDL must map to roles that you created earlier in Red Hat Single Sign-On.

TIP

For the sake of clarity, match the role names in the DDL file to the role names that you specified in Red Hat Single Sign-On. Consistent naming makes it easier to correlate how the roles that you define in each location relate to each other.

- The database access to allow to users who are granted the specified role. For example, SELECT permissions on a particular table view.

Prerequisites

- You configured Red Hat Single Sign-On to work with data virtualization as described in Section 7.3.1, "Configuring Red Hat Single Sign-On to secure OData" .
- You added SSO properties to the CR file for the virtual database, as described in .

Procedure

1. In a text editor, open the file that contains the DDL description that you used to deploy the virtual database.
2. Insert statements to add any roles that you defined for virtual database users in Red Hat Single Sign-On. For example, to add a role with the name **ReadRole** add the following statement to the DDL:

```
CREATE ROLE ReadRole WITH FOREIGN ROLE ReadRole;
```

   Add separate **CREATE ROLE** statements for each role that you want to implement for the virtual database.
3. Insert statements that specify the level of access that users with the role have to database objects. For example,

```
GRANT SELECT ON TABLE "portfolio.CustomerZip" TO ReadRole
```

   Add separate **GRANT** statements for each role that you want to implement for the virtual database.
4. Save and close the CR or DDL file.
   You are now ready to redeploy the virtual database. For information about how to run the Data Virtualization Operator to deploy the virtual database, see Chapter 6, *Running the Data Virtualization Operator to deploy a virtual database*.

   After you redeploy the virtual database, add a redirect URL in the Red Hat Single Sign-On Admin Console. For more information, see Section 7.3.4, "Adding a redirect URI for the data virtualization client in the Red Hat Single Sign-On Admin Console".

7.3.4. Adding a redirect URI for the data virtualization client in the Red Hat Single Sign-On Admin Console

After you enable SSO for your virtual database and redeploy it, specify a redirect URI for the data virtualization client that you created in Section 7.3.1, "Configuring Red Hat Single Sign-On to secure OData".

Redirect URIs, or callback URLs are required for *public* clients, such as OData clients that use OpenID Connect to authenticate, and communicate with an identity provider through the redirect mechanism.

For more information about adding redirect URIs for OIDC clients, see the NameOfRHSSOServerAdmin.

Prerequisites

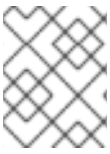
- You enabled SSO for a virtual database and used the Data Virtualization Operator to redeploy it.
- You have administrator access to the Red Hat Single Sign-On Admin Console.

Procedure

1. From a browser, sign in to the Red Hat Single Sign-On Admin Console.
2. From the security realm where you created the client for the data virtualization service, click **Clients** in the menu, and then click the ID of the data virtualization client that you created previously (for example, **dv-client**).
3. In the **Valid Redirect URIs** field, type the root URL for the OData service and append an asterisk to it. For example, **http://datavirt.odata.example.com/***
4. Test whether Red Hat Single Sign-On intercepts calls to the OData API.
   a. From a browser, type the address of an OData endpoint, for example:

      ```
      http://datavirt.odata.example.com/odata/CustomerZip
      ```

      A login page prompts you to provide credentials.
5. Sign in with the credentials of an authorized user.
   Your view of the data depends on the role of the account that you use to sign in.



NOTE

Some endpoints, such as **odata/$metadata** are excluded from security filtering so that they can be discovered by other services.

CHAPTER 8. VIRTUAL DATABASE MONITORING

Prometheus is an open-source systems and service monitoring and alerting toolkit that you can use to monitor services deployed in your Red Hat OpenShift environment. Prometheus collects and stores metrics from configured services at given intervals, evaluates rule expressions, displays the results, and can trigger alerts if a specified condition becomes true.

IMPORTANT

Red Hat support for Prometheus is limited to the setup and configuration recommendations provided in Red Hat product documentation.

Prometheus communicates with the data virtualization service to retrieve metrics and data. Users can query the data, or use a visualization took such as Grafana to view trends in a dashboard.

To enable monitoring of an OpenShift service, the service must expose an endpoint to Prometheus. This endpoint is an HTTP interface that provides a list of metrics and the current values of the metrics. When you use the Data Virtualization Operator to create a virtual database, the data virtualization service automatically exposes an HTTP endpoint to Prometheus. Prometheus periodically scrapes each target-defined endpoint and writes the collected data to its database.

After Prometheus is deployed to an OpenShift cluster, the metrics for any virtualization that you deploy to the same cluster are exposed to the Prometheus service automatically.

The following tables list the runtime and cache metrics that the data virtualization service exposes to Prometheus:

Table 8.1. Data virtualization runtime metrics exposed to Prometheus

Name	Description	Type
org.teiid.TotalRequestsProcessed	Total requests processed	Counter
org.teiid.WaitingRequestsCount	Requests queued for processing	Gauge
org.teiid.ActiveEngineThreadCount	Number of current engine threads	Gauge
org.teiid.QueuedEngineWorkItems	Number of queued work items	Gauge
org.teiid.LongRunningRequestCount	Number of long-running requests	Gauge
org.teiid.TotalOutOfDiskErrors	Total buffer out-of-disk errors	Counter
org.teiid.PercentBufferDiskSpaceInUse	Percent buffer disk space in use	Gauge
org.teiid.SessionCount	Number of client sessions	Gauge

Name	Description	Type
org.teiid.DiskSpaceUsedInMB	Disk space used (MB)	Gauge
org.teiid.ActiveRequestCount	Active requests	Gauge

Table 8.2. Data virtualization cache metrics exposed to Prometheus

Name	Description	Type
org.teiid.CacheRequestCount	Number of cache reads	Gauge
org.teiid.CacheTotalEntries	Number of cache entries	Gauge
org.teiid.CacheHitRatio	Ratio of cache hits to total attempts	Gauge

For more information about deploying Prometheus to monitor services on OpenShift, see the OpenShift documentation.

CHAPTER 9. 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.

Migration considerations

The following features that were supported in virtual databases in JBoss Data Virtualization and Teiid might be limited or unavailable in this Technology Preview release of data virtualization:

Data source compatibility

You cannot use all data sources with this release. For a list of compatible data sources, see Section 2.1, “Compatible data sources”.

Internal distributed materialization

Not available.

Resultset caching

Not available.

Use of runtime metadata to import other virtual databases

DDL must be used to specify metadata for virtual databases.

Runtime manipulation of multisource vdb sources

Not available.

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.

9.1. 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.

9.2. 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.

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
```