VMware vRealize Orchestrator 8.0 Plug-In Development Guide

25 FEB 2020

vRealize Orchestrator 8.0

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

Introduction to Developing vRealize Orchestrator plug-ins

1

This document describes how you can use a model-drive framework to develop plug-ins for vRealize Orchestrator 8.x.

This chapter includes the following topics:

Basic Plug-In Concepts

Plug-ins in vRealize Orchestrator rely on a few key features.

Checkpointing

The checkpointing feature makes it possible for vRealize Orchestrator to store the state of a running workflow and resume the workflow run from where it stopped. Checkpointing occurs every time a workflow completes a step and switches to the next element.

For example, you might run a workflow that invokes another workflow, **List path in guest**, and then calls a `set result` scripting element. When the workflow run switches from the **List path in guest workflow** to the scripting element, the engine creates a checkpoint by storing all input and output parameters, and variables.

The sample workflow verifies if a folder exists in the file system of a specified virtual machine.

The sample workflow includes four input parameters. Two of them are strings, one is a custom SDK object that derives from the vRealize Orchestrator vCenter Server plug-in, and the fourth one is a secure string.

In the current sample scenario, when the **List path in guest workflow** step is completed, the engine serializes the state of the workflow, which is called a token, and stores it in the vRealize Orchestrator database. When the workflow run passes to the `set result` scripting element, the engine deserializes the token and tries to resume the parameters and attributes that are associated with the workflow.

Parameter types are serialized in a different way. Serializing string types consists in storing and reading the value of the string. However, `VC:VirtualMachine` is a scripting object that contains data and methods, so serializing and deserializing it requires the use of a plug-in.

Scripting objects, whose serialization and deserialization involve using a plug-in, are known as SDK objects or finders. Such finder objects are not serialized by value but rather by a reference.

The reference has two components – object type, for example `VC:VirtualMachine` and the ID of the object. The engine delegates the deserialization of finder objects to a plug-in and invokes that plug-in by the prefix in the parameter name. The engine strips the `VC` prefix from the `VC:VirtualMachine` parameter type and invokes the `finder` and the associated `ID` for type `VirtualMachine`.

Finders

Custom SDK objects are defined by the underlying plug-ins. Finders create the custom SDK objects and pass them to the platform. The platform exposes an interface that includes three methods for retrieving these objects - `findById`, `findAll` and `findRelation`.

In the above checkpointing example, on every checkpoint, the engine invokes the `findById` method to restore the object. If you set a property to the virtual machine from the example, without storing this property on a third-party system, the next checkpoint retrieves a new object from the platform but does not save the assigned property. This means that the platform retains only a reference to the custom SDK object, without serializing the objects.

Developing Plug-Ins with Model-Driven

2

Model-driven is a framework that exposes the model of third-party libraries, which makes it suitable for creating plug-ins for specific integrations. Model-driven is included in the vRealize Orchestrator 8.x SDK.

You can find sample code at:

Protocol	Sample
HTTPS	https://github.com/dimitrovvlado/o11n-plugin-redis
SSH	git@github.com:dimitrovvlado/o11n-plugin-redis.git

This chapter includes the following topics:

Creating an Integration with Redis

You can use the model-driven framework included in the vRealize Orchestrator SDK to create an integration with the Redis data structure project.

Prerequisites

- Verify if Java Development Kit 8 is installed.
- Verify that you are familiar with Redis. See the Redis home page.
- Download and install the Maven build tool. See the Maven Apache Project.
- Migrate or Upgrade your vRealize Orchestrator Appliance to version 8.0 or later.
- Verify that you are familiar with Spring IoC. See Spring IoC.

Create the Skeleton of the Plug-In

The vRealize Orchestrator Appliance includes a Maven repository that contains the registered Maven archetypes. You can use these archetypes to develop a plug-in.

Procedure

1. On the vRealize Orchestrator welcome page, navigate to **Explore Developer Resources**.
2. In the **Develop an Orchestrator plug-in** section, copy the first command and run it in the command line of the machine you use for building the plug-in.

```
mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate -DarchetypeCatalog=https://{vro-host}/vco-repo/archetype-catalog.xml -DrepoUrl=https://{vro-host}/vco-repo -Dmaven.repo.remote=https://{vro-host}/vco-repo -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

With this command, Maven generates a project by using the archetype catalog in the internal Maven repository of the vRealize Orchestrator Appliance. The command is interactive and requires specifying several parameters.

a Select the numeric option that corresponds to the model-driven archetype.

```
Choose archetype:
1: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-plugin-archetype-inventory (o11nplugin-project-archetype)
2: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-package-archetype (o11nplugin-project-archetype)
3: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-archetype-inventory-annotation (o11nplugin-project-archetype)
4: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-plugin-archetype-simple (o11nplugin-project-archetype)
5: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-archetype-spring (o11nplugin-spring-archetype)
6: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-plugin-archetype-modeldriven (o11nplugin-project-archetype)
7: https://{vro_host}/vco-repo/archetype-catalog.xml -> com.vmware.o11n:o11n-client-archetype-rest (o11nplugin-project-archetype)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): : 6
```

b Enter the group ID according to the naming convention that your company uses. For more information, see the *Guide to naming conventions on groupId, artifactId and version* in the official Maven documentation.

```
Define value for property 'groupId': : com.vmware.o11n.plugin
[INFO] Using property: groupId = com.vmware.o11n.plugin
```

c Enter the `artifactId` that corresponds to the developed integration. In this example – Redis.

```
Define value for property 'artifactId': : redis
[INFO] Using property: artifactId = redis
```

d Enter a name for the package.

The package name should correspond to the `groupId` value and must be compliant with the Java packages naming convention. For more information, see *Naming a Package* in the official Java documentation.

```
Define value for property 'package':  com.vmware.o11n.plugin: : com.vmware.o11n.plugin.redis
[INFO] Using property: package = com.vmware.o11n.plugin.redis
```

e Enter name and alias of the plug-in.

```
Define value for property 'pluginAlias': : Redis
Define value for property 'pluginName': : Redis
```

The alias must not contain spaces because it is used as a prefix for the scripting objects. The archetype also uses the alias to prefix the adapter and the factory of the plug-in.

Note If the prefix contains spaces or invalid characters, Java classes with invalid names might be generated and the plug-in might not compile successfully.

When you output all parameters, you are prompted to confirm the information.

```
Confirm properties configuration:
groupId: com.vmware.o11n.plugin
groupId: com.vmware.o11n.plugin
artifactId: redis
artifactId: redis
version: 1.0.0-SNAPSHOT
package: com.vmware.o11n.plugin.redis
package: com.vmware.o11n.plugin.redis
pluginAlias: Redis
pluginName: Redis
repoUrl: https://{vro_host}/vco-repo
vcoVersion: {vro_version}
```

Note The `vcoVersion` value depends on the version of the vRealize Orchestrator SDK that you use.

3 Navigate to the project directory. Verify that the directory structure corresponds to the example.

```
drwxr-xr-x   9 vdimitrov  staff   306 Feb  7 11:08 .
drwxr-xr-x  46 vdimitrov  staff  1564 Feb  6 10:34 ..
drwxr-xr-x   4 vdimitrov  staff   136 Feb  7 11:08 o11nplugin-redis
drwxr-xr-x   4 vdimitrov  staff   136 Feb  7 11:08 o11nplugin-redis-core
drwxr-xr-x   4 vdimitrov  staff   136 Feb  7 11:08 o11nplugin-redis-custom
drwxr-xr-x   3 vdimitrov  staff   102 Feb  7 11:08 o11nplugin-redis-gen
drwxr-xr-x   6 vdimitrov  staff   204 Feb  7 11:08 o11nplugin-redis-package
-rw-r--r--   1 vdimitrov  staff  2519 Feb  7 11:08 pom.xml
```

4 Run the `mvn clean install` command to build the plug-in.

```
mvn clean install –Dmaven.wagon.http.ssl.insecure=true –Dmaven.wagon.http.ssl.allowall=true
```

Note When you build a project from the repository, you must add the vRealize Orchestrator IP address or DNS name in the root `pom.xml`. By default, the archetype does this automatically.

```
mvn clean install –Dmaven.wagon.http.ssl.insecure=true –Dmaven.wagon.http.ssl.allowall=true –
Drepo.host={vro_host}
```

If the build is successful, the console displays the `[INFO] BUILD SUCCESS` message.

Note If you generate the Redis integration plug-in with Orchestrator 7.0, the build might fail because of a missing file. To work around the problem, you must open the `{plug–in–home}/o11nplugin–redis–core/src/main/resources/com/vmware/o11n/plugin/redis/plugin.xml` file and append the following content:

```
<?xml version="1.0" encoding="UTF–8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema–instance" xmlns:context="http://
www.springframework.org/schema/context"
    xmlns:util="http://www.springframework.org/schema/util" xmlns:task="http://
www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/task http://www.springframework.org/
schema/task/spring–task–3.0.xsd
        http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/
spring–beans–3.0.xsd
        http://www.springframework.org/schema/context http://www.springframework.org/schema/
context/spring–context–3.0.xsd
        http://www.springframework.org/schema/util http://www.springframework.org/schema/util/
spring–util–3.0.xsd">
    <context:component–scan base–package="com.vmware.o11n.plugin.redis" />

    <import resource="classpath:com/vmware/o11n/plugin/sdk/spring/pluginEnv.xml" />
    <bean class="com.vmware.o11n.sdk.modeldriven.impl.PolicyService" />
    <bean class="com.vmware.o11n.sdk.modeldriven.impl.DefaultInventoryService" />
    <bean class="com.vmware.o11n.sdk.modeldriven.impl.DefaultObjectFactory" />
    <bean class="com.vmware.o11n.sdk.modeldriven.impl.DefaultCollectionFactory" />
    <bean class="com.vmware.o11n.sdk.modeldriven.impl.DefaultProxyResolver" />
    <bean class="com.vmware.o11n.sdk.modeldriven.impl.DefaultRuntimeConfiguration">
        <property name="properties">
            <util:properties location="com/vmware/o11n/plugin/redis_gen/runtime–
config.properties" />
        </property>
    </bean>

    <bean class="com.vmware.o11n.plugin.redis.RedisPluginFactory" id="pluginFactory"
        autowire–candidate="false" scope="prototype" />
</beans>
```

What to do next

- The `PluginFactory` class must match your factory, namely `RedisPluginFactory`.
- The `redis_gen` directory that contains the `runtime-config.properties` file must match the folder of your plug-in project.

File Structure

The home folder of the plug-in contains five Maven modules.

o11nplugin-redis

The `o11nplugin-redis` folder contains the plug-in deliverables.

.dar archive

- Contains the plug-in JAR files and the dependable JAR files.
- Contains the plug-in icons.
- Contains the `.package` archive that includes the workflows and actions.

.vmoapp file

- Contains one or multiple DAR archives.
- Contains the End-User License Agreement.

vso.xml descriptor file

- Contains information about the plug-in.
- Contains system information about the plug-in, such as name, scripting objects, inventory objects, name of the `.package` file.

Images folder

Contains icons that you can associate inventory objects with.

o11nplugin-redis-core

The `o11nplugin-redis-core` folder contains implementations that are related to the persistence, inventory objects and their mutual relations, scripting objects, caching, unit tests, and others. The `o11nplugin-redis-core` folder also includes the `PluginAdaptor` class, which is the entry point for the plug-in, and `PluginFactory`, which is a method of finding inventory objects but model-driven does not use this method.

plugin.xml file

The `plugin.xml` file is a spring configuration file. This file defines a set of beans for the model-driven framework.

o11nplugin-redis-custom

The `o11nplugin-redis-custom` folder is essential for the model-driven-based plug-ins because it stores all scripting objects and their finders.

CustomModule.java file

In the `CustomModule.java` file, you apply modifications to the `vso.xml` descriptor file.

```
public CustomModule() {
    this.plugin = new Plugin();
    plugin.setApiPrefix("Redis");
    plugin.setIcon("default-32x32.png");
    plugin.setDescription("Redis");
    plugin.setDisplayName("Redis");
    plugin.setName("Redis");
    plugin.setPackages(Collections.singletonList("o11nplugin-example-package-$
{project.version}.package"));
    plugin.setAdaptorClassName(com.vmware.o11n.plugin.redis.RedisPluginAdaptor.class);
}
```

In this file, you define the name and the prefix of the plug-in, and the content of the `.package` file. Here you also define the `PluginAdaptor` class that points the plug-in entry point to the platform.

CustomMapping.java file

This is the Java class that contains all scripting objects, singleton objects, and finders of the plug-in. In the `CustomMapping.java` file you bind all dynamic parts of the plug-in.

o11nplugin-redis-gen

The `o11nplugin-redis-gen` folder stores all the generated code. You do not use this module for building the Redis integration plug-in. The `pom.xml` file in this folder contains a custom Maven plug-in.

o11nplugin-redis-package

The `o11nplugin-redis-package` folder includes the text representation of the plug-in, such as workflows, actions, resource elements, configurations but without their corresponding binary files.

HelloWorld.xml file

The `HelloWorld.xml` file is a sample workflow text representation file.

Deploy a Plug-In

After you confirm that the plug-in built is successful, you upload the plug-in to the vRealize Orchestrator server.

Procedure

1 Log in the Control Center as **root**.

2 Select **Manage Plug-Ins**.

3 Click **Browse**, and navigate to the `o11nplugin-redis.vmoapp` file that is located in the `{plug-in-home}/o11nplugin-redis/target` directory.

4 Click **Upload**.

5 To install the plug-in, click **Install**.

6 Log in to the vRealize Orchestrator Client.

7 Navigate to the **Inventory** page, and verify that the Redis plug-in is listed.

Modify the Plug-In Descriptor File

You can customize the plug-in further, for example by changing the plug-in icon that appears in Control Center and in the vRealize Orchestrator Client.

Procedure

1 Open the `CustomModule.java` file in the `o11nplugin-redis-custom` folder and change the icon and the description of the Redis plug-in.

```
public CustomModule() {
   this.plugin = new Plugin();
    plugin.setApiPrefix("Redis");
    plugin.setIcon("redis.png");
   plugin.setDescription("Redis plug-in for vRO");
   plugin.setDisplayName("Redis");
   plugin.setName("Redis");
plugin.setPackages(Collections.singletonList("o11nplugin-redis-package-$
{project.version}.package"));
 plugin.setAdaptorClassName(com.vmware.o11n.plugin.redis.RedisPluginAdaptor.class);
}
```

2 To rebuild the plug-in, run the `mvn clean install` command.

```
mvn clean install -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

3 (Optional) During the installation, you can change the build number, the version, and the installation mode of the plug-in, by adding the corresponding properties to the install command

```
mvn clean install –Dbuild.number=999 –Dinstallation.mode=always
```

Note The `installation mode` property defines the behavior of the newly built plug-in content. The value of the installation mode can be either `never`,`always` or `version`.

Value	Description
Never	The new content never overwrites the existing content.
Always	The new content always overwrites the existing content. For example, if the new content has an earlier version number compared to the existing one.
Version	Compare the versions and update the plug-in only if the new version number is later than the existing one.

When the new build completes, the `vso.xml` file in the {`plug–in–home`}/`o11nplugin–redis/src/main/dar/VSO–INF` directory shows the new values for the icon and for the description.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<module name="Redis" version="${project.version}" build-number="${build.number}" image="images/
redis.png" display-name="Redis">
    <description>Redis plug-in for vRO</description>
    <installation mode="${installation.mode}">
        <action resource="packages/o11nplugin-redis-package-${project.version}.package"
type="install-package"/>
    </installation>
    <inventory type="_ROOT"/>
    <finder-datasources>
        <finder-datasource name="main-datasource" adaptor-
class="com.vmware.o11n.plugin.redis.RedisPluginAdaptor" concurrent-call="false" anonymous-login-
mode="internal"/>
    </finder-datasources>
    <finders>
        <finder hidden="true" datasource="main-datasource" type="_ROOT">
            <inventory-children/>
            <relations/>
        </finder>
    </finders>
    <scripting-objects/>
</module>
```

Create a Scripting Object

Besides changing the basic plug-in configuration options, you can add some code that makes the plug-in complete a task. You can create a scripting object and statically invoke its methods, by using a scripting singleton.

Procedure

1. Under the `{plug-in-home}/o11nplugin-redis-core/src/main/java/com/vmware/o11n/plugin/redis` folder, create a directory with the name **`singleton`**.

   This directory constitutes the `com.vmware.o11n.plugin.redis.singleton` package of the plug-in.

2. Create a `ConnectionManager` scripting class in the `com.vmware.o11n.plugin.redis.singleton` package.

3. Save the `ConnectionManager` scripting class as a `ConnectionManager.java` file in the singleton directory.

```
package com.vmware.o11n.plugin.redis.singleton;
/**
 * A scripting singleton for managing redis connections.
 */
public class ConnectionManager {
    /**
     * This method creates a redis connection by the provided name, host and
     * port.
     *
     * @param connectionName
     *            the name of the connection, only one with the same name can
     *            exist
     * @param redisHost
     *            the redis host, cannot be null
     * @param redisPort
     *            redis port
     * @return the ID of the newly created connection
     */
    public String save(String connectionName, String redisHost, int redisPort) {
        return null;
    }
}
```

4. Add the `singleton(ConnectionManager.class)` line to the `CustomMapping.java` file so that the plug-in can expose the `ConnectionManager` class as a scripting object.

```
package com.vmware.o11n.plugin.redis;

import com.vmware.o11n.plugin.redis.singleton.ConnectionManager;
import com.vmware.o11n.sdk.modeldrivengen.mapping.AbstractMapping;

public class CustomMapping extends AbstractMapping {
    @Override
    public void define() {
        singleton(ConnectionManager.class);
    }
}
```

5. To rebuild the plug-in, run the `mvn clean install` command.

6 Verify the changes in the `vso.xml` file.

```
<scripting-objects>
    <object create="false" java-
class="com.vmware.o11n.plugin.redis_gen.ConnectionManager_Wrapper" script-
name="_RedisConnectionManager">
        <description></description>
        <singleton script-name="RedisConnectionManager" datasource="main-datasource"/>
        <attributes/>
        <constructors/>
        <methods>
            <method script-name="save" java-name="save" return-type="String">
                <description></description>
                <parameters>
                    <parameter type="String" name="connectionName"></parameter>
                    <parameter type="String" name="redisHost"></parameter>
                    <parameter type="Number" name="redisPort"></parameter>
                </parameters>
            </method>
        </methods>
    </object>
</scripting-objects>
```

The vRealize Orchestrator platform can interpret the scripting-objects tag `RedisConnectionManage` is a scripting object associated to the `ConnectionManager` class.

With model-driven, you can extend objects with different functionalities. In this example, the `vso.xml` file points to a wrapper class, instead of pointing to the scripting object itself.

7 Install the modified version of the plug-in. See Deploy a Plug-In.

8 Log in to the vRealize Orchestrator Client.

9 From the left navigation panel, select **API Explorer**.

10 Expand the Redis plug-in to browse through the API elements.

The `ConnectionManager` class is exposed as a scripting object together with the `save` method.

Note By default, the code generator adds the name of the plug-in as a prefix of the plug-in objects. For example, `ConnectionManager` becomes `RedisConnectionManager`.

Add an Endpoint Configuration

Most vRealize Orchestrator plug-ins require an endpoint configuration. An endpoint is a location where the plug-in stores connection details for the instances, with which vRealize Orchestrator communicates.

Procedure

1 Under the `{plug-in-home}/o11nplugin-redis-core/src/main/java/com/vmware/o11n/plugin/redis` folder, create a directory with the name **`model`**.

This directory constitutes the `com.vmware.o11n.plugin.redis.model` package of the plug-in.

2 Create an endpoint configuration.

The following example shows a plain old Java object (POJO) that is used to store connection details for the plug-in endpoint, such as connection name, host name and host port.

```
package com.vmware.o11n.plugin.redis.model;
import org.springframework.util.Assert;
import com.vmware.o11n.sdk.modeldriven.Sid;
/**
 * Object for all the redis connection details.
 */
public class ConnectionInfo {
    /**
     * Name of the connection as defined by the user when creating a connection
     */
    private String name;
    /**
     * ID of the connection
     */
    private final Sid id;
    /**
     * Service URI of the third party system
     */
    private String host;
    /**
     * Port of the connection, defaults the to redis default port number
     */
    private int port = 6379;
    public ConnectionInfo() {
        this.id = Sid.unique();
    }
    /*
     * Verify that each ConnectionInfo has an ID.
     */
    public ConnectionInfo(Sid id) {
        super();
        Assert.notNull(id, "Id cannot be null.");
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getHost() {
        return host;
    }
    public void setHost(String host) {
        this.host = host;
    }
    public Sid getId() {
        return id;
    }
    public void setPort(int port) {
```

```
        this.port = port;
    }
    public int getPort() {
        return port;
    }
    @Override
    public String toString() {
        return "ConnectionInfo [name=" + name + ", id=" + id + ", uri=" + host + ", port=" + port
+ "]";
    }
}
```

3 Specify an ID, so that the default constructor can find a unique ID.

Plug-ins use `Sid` instead of a string ID. When you build hierarchies of object, each child object must know its parent ID. A `Sid` wraps multiple key-value objects and you can identify a single object by different properties.

For example, a single configuration can have multiple connections, each connection has a list of locations and each location has a list of virtual machine images. When you run certain operation, for example delete, on a virtual machine image, you must know the specific connection, to which you run the command. By using a `Sid`, you can identify the virtual machine image object by connection ID, name, and location.

4 Save the endpoint configuration as a `ConnectionInfo.java` file in the model directory.

5 Wrap `ConnectionInfo` to `Connection` to expose the actual object as a scripting object

```
package com.vmware.o11n.plugin.redis.model;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;
@Component
@Qualifier(value = "connection")
@Scope(value = "prototype")
public class Connection {
    /*
     * The connectionInfo which stands behind this live connection.
     */
    private ConnectionInfo connectionInfo;
    /*
     * There is no default constructor, the Connection must be initialized only
     * with a connection info argument.
     */
    public Connection(ConnectionInfo info) {
        init(info);
    }
    public synchronized ConnectionInfo getConnectionInfo() {
        return connectionInfo;
    }
    public String getDisplayName() {
        return getConnectionInfo().getName() + " [" + getConnectionInfo().getHost() + "]";
    }
    /*
```

```
     * Updates this connection with the provided info. This operation will
     * destroy the existing third party client, causing all associated
     * operations to fail.
     */
    public synchronized void update(ConnectionInfo connectionInfo) {
        if (this.connectionInfo != null && !
connectionInfo.getId().equals(this.connectionInfo.getId())) {
            throw new IllegalArgumentException("Cannot update using different id");
        }
        destroy();
        init(connectionInfo);
    }
    private void init(ConnectionInfo connectionInfo) {
        this.connectionInfo = connectionInfo;
    }
    public synchronized void destroy() {
        // Destroy the third party client
    }
}
```

Note After the `Connection` object is exposed as a scripting object, it will also be exposed as an inventory object.

6 Save the `Connection` object as a `Connection.java` file in the model directory.

7 Under the `{plug-in-home}/o11nplugin-redis-core/src/main/java/com/vmware/o11n/plugin/redis` folder, create a directory with name `config`.

This directory constitutes the `com.vmware.o11n.plugin.redis.config` package of the plug-in.

8 Add a connection, persister, and interface for object messaging to the `com.vmware.o11n.plugin.redis.config` package.

a Add `ConnectionPersister` and save it as a `ConnectionPersister.java` file.

Note `ConnectionPersister` invokes the vRealize Orchestrator Persistence SDK.

```
package com.vmware.o11n.plugin.redis.config;
import java.util.List;
import com.vmware.o11n.plugin.redis.model.ConnectionInfo;
import com.vmware.o11n.sdk.modeldriven.Sid;
public interface ConnectionPersister {
    /**
     * Returns a collection of all stored configurations (resources under a
     * folder with the plug-in name)
     */
    public List<ConnectionInfo> findAll();
    /**
     * Returns a collection by its ID or null if not found
     */
    public ConnectionInfo findById(Sid id);
    /**
     * Stores a connection info or updates it if already available. The
     * persister checks the availability of a connection by its ID
     */
    public ConnectionInfo save(ConnectionInfo connection);
    /**
     * Deletes a connection info. The persister will use the ID of the
     * connection
     */
    public void delete(ConnectionInfo connectionInfo);
    /**
     * Allows us to subscribe to the events of the persister. For example, if a
     * connection is deleted, the persister will trigger an event, notifying all
     * subscribers. This is an implementation of the observer pattern.
     */
    void addChangeListener(ConfigurationChangeListener listener);
    /**
     * Forces the persister to read all the configurations and trigger the
     * events. This method is invoked when the plug-in is loaded on server
     * start-up.
     */
    public void load();
}
```

b Add `DefaultConnectionPersister` and save it as a `DefaultConnectionPersister.java` file.

```
package com.vmware.o11n.plugin.redis.config;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import org.apache.commons.lang.Validate;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import com.vmware.o11n.plugin.redis.model.ConnectionInfo;
import com.vmware.o11n.sdk.modeldriven.Sid;
import ch.dunes.vso.sdk.endpoints.IEndpointConfiguration;
import ch.dunes.vso.sdk.endpoints.IEndpointConfigurationService;
@Component
public class DefaultConnectionPersister implements ConnectionPersister {
    private static final Logger log =
LoggerFactory.getLogger(DefaultConnectionPersister.class);
    /**
     * A list of listeners, who have subscribed to any configuration events,
     * such as connection updates and deletions.
     */
    private final Collection<ConfigurationChangeListener> listeners;
    public DefaultConnectionPersister() {
        // Initialise the listeners
        listeners = new CopyOnWriteArrayList<ConfigurationChangeListener>();
    }
    /*
     * Constants of the key names under which the connection values will be
     * stored.
     */
    private static final String ID = "connectionId";
    private static final String NAME = "name";
    private static final String SERVICE_HOST = "serviceHost";
    private static final String SERVICE_PORT = "servicePort";
    /*
     * The platform-provided service for configuration persistence
     */
    @Autowired
    private IEndpointConfigurationService endpointConfigurationService;
    /*
     * Returns a collection of all stored configurations for this plug-in only
     * The service is aware of the plug-in name, thus will return only
     * configurations for this plug-in.
     */
    @Override
    public List<ConnectionInfo> findAll() {
        Collection<IEndpointConfiguration> configs;
        try {
            // Use the configuration service to retrieve all configurations.
            // The service is aware of the plug-in name, thus will return only
            // configurations for this plug-in.
            configs = endpointConfigurationService.getEndpointConfigurations();
            List<ConnectionInfo> result = new ArrayList<>(configs.size());
            // Iterate all the connections
            for (IEndpointConfiguration config : configs) {
                // Convert the IEndpointConfiguration to our domain object - the
                // ConnectionInfo
                ConnectionInfo connectionInfo = getConnectionInfo(config);
                if (connectionInfo != null) {
                    log.debug("Adding connection info to result map: " + connectionInfo);
```

```
                    result.add(connectionInfo);
                }
            }
            return result;
        } catch (IOException e) {
            log.debug("Error reading connections.", e);
            throw new RuntimeException(e);
        }
    }
    /*
     * Returns a ConnectionInfo by its ID The service is aware of the plug-in
     * name, thus cannot return a configuration for another plug-in.
     */
    @Override
    public ConnectionInfo findById(Sid id) {
        // Sanity checks
        Validate.notNull(id, "Sid cannot be null.");
        IEndpointConfiguration endpointConfiguration;
        try {
            // Use the configuration service to retrieve the configuration
            // service by its ID
            endpointConfiguration =
endpointConfigurationService.getEndpointConfiguration(id.toString());
            // Convert the IEndpointConfiguration to our domain object - the
            // ConnectionInfo
            return getConnectionInfo(endpointConfiguration);
        } catch (IOException e) {
            log.debug("Error finding connection by id: " + id.toString(), e);
            throw new RuntimeException(e);
        }
    }
    /**
     * Save or update a connection info. The service is aware of the plug-in
     * name, thus cannot save the configuration under the name of another
     * plug-in.
     */
    @Override
    public ConnectionInfo save(ConnectionInfo connectionInfo) {
        // Sanity checks
        Validate.notNull(connectionInfo, "Connection info cannot be null.");
        Validate.notNull(connectionInfo.getId(), "Connection info must have an id.");
        // Additional validation - in this case we want the name of the
        // connection to be unique
        validateConnectionName(connectionInfo);
        try {
            // Find a connection with the provided ID. We don't expect to have
            // an empty ID
            IEndpointConfiguration endpointConfiguration = endpointConfigurationService
                    .getEndpointConfiguration(connectionInfo.getId().toString());
            // If the configuration is null, then we are performing a save
            // operation
            if (endpointConfiguration == null) {
                // Use the configuration service to create a new (empty)
                // IEndpointConfiguration.
                // In this case, we are responsible for assigning the ID of the
```

```
                // configuration,
                // which is done in the constructor of the ConnectionInfo
                endpointConfiguration = endpointConfigurationService
                        .newEndpointConfiguration(connectionInfo.getId().toString());
            }
            // Convert the ConnectionInfo the IEndpointConfiguration
            addConnectionInfoToConfig(endpointConfiguration, connectionInfo);
            // Use the configuration service to save the endpoint configuration
            endpointConfigurationService.saveEndpointConfiguration(endpointConfiguration);
            // Fire an event to all subscribers, that we have updated a
            // configuration.
            // Pass the entire connectionInfo object and let the subscribers
            // decide if they need to do something
            fireConnectionUpdated(connectionInfo);
            return connectionInfo;
        } catch (IOException e) {
            log.error("Error saving connection " + connectionInfo, e);
            throw new RuntimeException(e);
        }
    }
    /**
     * Delete a connection info. The service is aware of the plug-in name, thus
     * cannot delete a configuration from another plug-in.
     */
    @Override
    public void delete(ConnectionInfo connectionInfo) {
        try {
            // Use the configuration service to delete the connection info. The
            // service uses the ID

endpointConfigurationService.deleteEndpointConfiguration(connectionInfo.getId().toString());
            // Fire an event to all subscribers, that we have deleted a
            // configuration.
            // Pass the entire connectionInfo object and let the subscribers
            // decide if they need to do something
            fireConnectionRemoved(connectionInfo);
        } catch (IOException e) {
            log.error("Error deleting endpoint configuration: " + connectionInfo, e);
            throw new RuntimeException(e);
        }
    }
    /**
     * This method is used to load the entire configuration set of the plug-in.
     * As a second step we fire a notification to all subscribers. This method
     * is used when the plug-in is being loaded (on server startup).
     */
    @Override
    public void load() {
        List<ConnectionInfo> findAll = findAll();
        for (ConnectionInfo connectionInfo : findAll) {
            fireConnectionUpdated(connectionInfo);
        }
    }
    /**
     * Attach a configuration listener which will be called when a connection is
```

```
     * created/updated/deleted
     */
    @Override
    public void addChangeListener(ConfigurationChangeListener listener) {
        listeners.add(listener);
    }
    /*
     * A helper method which iterates all event subscribers and fires the update
     * notification for the provided connection info.
     */
    private void fireConnectionUpdated(ConnectionInfo connectionInfo) {
        for (ConfigurationChangeListener li : listeners) {
            li.connectionUpdated(connectionInfo);
        }
    }
    /*
     * A helper method which iterates all event subscribers and fires the delete
     * notification for the provided connection info.
     */
    private void fireConnectionRemoved(ConnectionInfo connectionInfo) {
        for (ConfigurationChangeListener li : listeners) {
            li.connectionRemoved(connectionInfo);
        }
    }
    /*
     * A helper method which converts our domain object the ConnectionInfo to an
     * IEndpointConfiguration
     */
    private void addConnectionInfoToConfig(IEndpointConfiguration config, ConnectionInfo
info) {
        try {
            config.setString(ID, info.getId().toString());
            config.setString(NAME, info.getName());
            config.setString(SERVICE_HOST, info.getHost());
            config.setInt(SERVICE_PORT, info.getPort());
        } catch (Exception e) {
            log.error("Error converting ConnectionInfo to IEndpointConfiguration.", e);
            throw new RuntimeException(e);
        }
    }
    /*
     * A helper method which converts the IEndpointConfiguration to our domain
     * object the ConnectionInfo
     */
    private ConnectionInfo getConnectionInfo(IEndpointConfiguration config) {
        ConnectionInfo info = null;
        try {
            Sid id = Sid.valueOf(config.getString(ID));
            info = new ConnectionInfo(id);
            info.setName(config.getString(NAME));
            info.setHost(config.getString(SERVICE_HOST));
            info.setPort(config.getAsInteger(SERVICE_PORT));
        } catch (IllegalArgumentException e) {
            log.warn("Cannot convert IEndpointConfiguration to ConnectionInfo: " +
config.getId(), e);
```

```
        }
        return info;
    }
    /*
     * A helper method which validates if a connection with the same name
     * already exists.
     */
    private void validateConnectionName(ConnectionInfo connectionInfo) {
        ConnectionInfo configurationByName =
getConfigurationByName(connectionInfo.getName());
        if (configurationByName != null
                && !
configurationByName.getId().toString().equals(connectionInfo.getId().toString())) {
            throw new RuntimeException("Connection with the same name already exists: " +
connectionInfo);
        }
    }
    /*
     * A helper method which gets a connection by name.
*/
private ConnectionInfo getConfigurationByName(String name) {
Validate.notNull(name, "Connection name cannot be null.");
Collection<ConnectionInfo> findAllClientInfos = findAll();
for (ConnectionInfo info : findAllClientInfos) {
if (name.equals(info.getName())) {
return info;
}
}
return null;
}
}
```

The `DefaultConnectionPersister` class includes the `IEndpointConfigurationService service` that is specific for vRealize Orchestrator. `IEndpointConfigurationService` stores in the resources of the vRealize Orchestrator platform simple key-value objects called `IEndpointConfiguration` objects. This service saves and deletes the configuration, and also validates connection configurations, if these configurations have a unique name system-wide.

c Add a collection of `ConfigurationChangeListener` interfaces.

d Save the interface as a `ConfigurationChangeListener.java` file in the configuration directory.

You can use the `ConfigurationChangeListener` interfaces as an extension point of the `IEndpointConfigurationService` service.

```
package com.vmware.o11n.plugin.redis.config;
import com.vmware.o11n.plugin.redis.model.ConnectionInfo;
/**
 * An extension point of the configuration persister. Serves the role of an
 * Observer when a certain connection is created, modified or deleted.
 */
public interface ConfigurationChangeListener {
    /**
     * Invoked when a connection is updated or created
     */
    void connectionUpdated(ConnectionInfo info);
    /**
     * Invoked when the ConnectionInfo input is deleted
     */
    void connectionRemoved(ConnectionInfo info);
}
```

This way, you implement the Observer pattern. If you want to cache all live connections, you must know which of the connections are deleted, so that you can remove them from the local cache.

e Create a local cache by using `ConnectionRepository` saved as a `ConnectionRepository.java` file in the configuration directory.

```
package com.vmware.o11n.plugin.redis.config;
import java.util.Collection;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.stereotype.Component;
import com.vmware.o11n.plugin.redis.model.Connection;
import com.vmware.o11n.plugin.redis.model.ConnectionInfo;
import com.vmware.o11n.sdk.modeldriven.Sid;
@Component
public class ConnectionRepository implements ApplicationContextAware, InitializingBean,
ConfigurationChangeListener {
    /*
     * Injecting the ConnectionPersister
     */
    @Autowired
    private ConnectionPersister persister;
    private ApplicationContext context;
    /*
     * The local map (cache) of live connections
     */
    private final Map<Sid, Connection> connections;
```

```
public ConnectionRepository() {
    connections = new ConcurrentHashMap<Sid, Connection>();
}
/**
 * Returns a live connection by its ID or null if no such connection has
 * been initialised by this ID
 */
public Connection findLiveConnection(Sid anyId) {
    return connections.get(anyId.getId());
}
/**
 * Returns all live connections from the local cache
 */
public Collection<Connection> findAll() {
    return connections.values();
}
/*
 * Spring-specifics - storing a reference to the spring context
 */
@Override
public void setApplicationContext(ApplicationContext context) throws BeansException {
    this.context = context;
}
/*
 * Spring specifics - this method is being called automatically by the
 * spring container after all the fields are set and before the bean is
 * being provided for usage. This method will be called when the plug-in is
 * being loaded - on server start-up.
 */
@Override
public void afterPropertiesSet() throws Exception {
    // Subscribing the Repository for any configuration changes that occur
    // in the Persister
    persister.addChangeListener(this);
    // Initialising the Persister. By doing that, the persister will invoke
    // the connectionUpdated() method
    // and since we are subscribed to those events, the local cache will be
    // populated with all the available connections.
    persister.load();
}
private Connection createConnection(ConnectionInfo info) {
    // This call will create a new spring-managed bean from the context
    return (Connection) context.getBean("connection", info);
}
/*
 * This method will be called from the ConnectionPersister when a new
 * connection is added or an existing one is updated.
 */
@Override
public void connectionUpdated(ConnectionInfo info) {
    Connection live = connections.get(info.getId());
    if (live != null) {
        live.update(info);
    } else {
        // connection just added, create it
```

```
            live = createConnection(info);
            connections.put(info.getId(), live);
        }
    }
    /*
     * This method will be called from the ConnectionPersister when a connection
     * is removed.
     */
    @Override
    public void connectionRemoved(ConnectionInfo info) {
        Connection live = connections.remove(info.getId());
        if (live != null) {
            live.destroy();
        }
    }
}
```

You can use `ConnectionRepository` as a local cache to read a single connection or all available connections. A data structure keeps all connections and every time the platform requests a connection, the local cache retrieves the same connection instance.

Note You can use `ConnectionRepository` for read operations and `ConnectionPersister` for create, update, and delete operations. When a connection is created, the persister notifies all listeners. For example, `ConnectionRepository` and the local cache stores every new connection.

For performance reasons, using `ConnectionRepository` is the preferred method for handling connections. Reading connections with `IEndpointConfigurationService` might lead to a unnecessary increase in plug-in load.

9 Use `ConnectionManager` from the `singleton` directory to expose the configuration-specific options in the scripting.

```
package com.vmware.o11n.plugin.redis.singleton;
import org.springframework.beans.factory.annotation.Autowired;
import.org.springframework.context.annotation.Scope;
import.org.springframework.stereotype.Component;
import com.vmware.o11n.plugin.redis.config.ConnectionPersister;
import com.vmware.o11n.plugin.redis.model.ConnectionInfo;
import com.vmware.o11n.plugin.sdk.spring.platform.GlobalPluginNotificationHandler;
/**
 * A scripting singleton for managing redis connections.
 */
@Component
@Scope(value = "prototype")
public class ConnectionManager {
    @Autowired
    private ConnectionPersister persister;
    /*
     * A vRO SDK object which is responsible for notifying the platform of
     * changes in the inventory of the plug-in
     */
    @Autowired
    private GlobalPluginNotificationHandler notificationHandler;
```

```
    /**
     * This method creates a redis connection by the provided name, host and
     * port.
     *
     * @param connectionName
     *            the name of the connection, only one with the same name can
     *            exist
     * @param redisHost
     *            the redis host, cannot be null
     * @param redisPort
     *            redis port
     * @return the ID of the newly created connection
     */
    public String save(String connectionName, String redisHost, int redisPort) {
        ConnectionInfo info = new ConnectionInfo();
        info.setName(connectionName);
        info.setHost(redisHost);
        if (redisPort > 0) {
            info.setPort(redisPort);
        }
        // Save the connection through the persister
        info = persister.save(info);
        // Invalidate all elements of the redis inventory
        notificationHandler.notifyElementsInvalidate();
        // Return the ID of the newly created connection
        return info.getId().toString();
    }
}
```

When you invoke the save method of the singleton scripting object, a new `ConnectionInfo` class is constructed. You save the connection by using the newly created persister.

Note You must annotate the class, otherwise the dependency injection does not work.

10 Build the plug-in again and deploy it to the vRealize Orchestrator server. See Deploy a Plug-In.

Create a Workflow

After you install the Redis plug-in and install it on your vRealize Orchestrator server, you can create workflows. The **Add redis connection workflow** is a sample workflow that requires three input parameters: a name, a host name, and a port.

Procedure

1 Log in to the vRealize Orchestrator Client.

2 Navigate to **Library > Workflows**.

3 Click **New Workflow**.

4 Enter a name for the workflow, and click **Create**.

5 On the **Inputs/Outputs** tab, add the following workflow input parameters.

Input Parameter	Type
name	string
host	string
port	number

6 On the **Schema** tab, add a scriptable task element.

7 Select the scriptable task and name it **Create connection**.

8 Select **Create connection**, and add the following script on the **Scripting** tab.

```
var connectionId = RedisConnectionManager.save(name, host, port);
System.log("Created connection with ID: " + connectionId);
```

9 Save the workflow, and click **Run**.

10 Enter the workflow input parameters, and click **Run**.

The printed connection ID is displayed on the **Logs** tab.

11 (Optional) To review the input parameters used in your workflow, navigate to **Assets > Resources**.

Export the Content

After you install a plug-in in vRealize Orchestrator, you can create and run workflows. If you want to include these workflows in the plug-in package, you must export the content of the workflows to the plug-in project.

Procedure

1 Log in to the vRealize Orchestrator Client.

2 Navigate to **Assets > Packages**.

3 Click **New Package**.

4 In the **Package name** text box, enter **`com.vmware.library.redis`**.

5 Select the **Content** tab, and click **Add**.

6 Select your Radis workflow. See Create a Workflow.

7 Click **Add**.

8 To finish creating the package, click **Create**.

9 On the machine where you created the Redis plug-in, open the `pom.xml` file of the package `o11nplugin-redis-package` module.

10 Modify the `packageName` property of the plug-in package configuration.

```
<plugin>
    <groupId>com.vmware.o11n.mojo.pkg</groupId>
    <artifactId>maven-o11n-package-plugin</artifactId>
    <version>${vco.version}</version>
    <extensions>true</extensions>
    <configuration>
        <packageName>com.vmware.o11n.library.redis</packageName>
        <!-- Set the local path to the *.vmokeystore file used to sign the content -->
        <keystoreLocation>${keystoreLocation}</keystoreLocation>
        <keystorePassword>${keystorePassword}</keystorePassword>
        <includes>
            <include>**/Library/Redis/**/*.element_info.xml</include>
        </includes>
        <packageFileName>o11nplugin-redis-package-${project.version}</packageFileName>
        <allowedMask>vef</allowedMask>
        <exportVersionHistory>false</exportVersionHistory>
    </configuration>
</plugin>
```

11 Go to the `o11nplugin-redis-package` folder and run the `mvn import` command.

```
mvn o11n-package:import-package -DserverUrl={vro_host}:{vro_port} -Dusername={vro_user} -
Dpassword={vro_pass} -Dmaven.wagon.http.ssl.insecure=true -Dmaven.wagon.http.ssl.allowall=true
```

You must replace the placeholders with the parameters of your environment. If your user name includes a domain, for example **`administrator@vsphere.local`**, you must add a slash (\) to your user name.

```
vsphere.local\administrator
```

Two new files appear under the `o11nplugin-redis-package/src/main/resources/Workflow/Library` folder.

Results

You have added new content to your plug-in project. To add more content, add this content to the plug-in package, and run the `mvn import` command.

Inventory Object Interfaces

With the Redis plug-in, you can store connections and make them visible in the plug-in inventory. Model-driven defines three types of interfaces that you can use to display objects in the plug-in inventory.

Interface	Description	Use
`ObjectRelater`	Describes the relation between two model objects.	In the example presented in step 2 of the Add an Endpoint Configuration procedure, this class holds the code that lists all virtual machines by their location.
`ObjectFinder`	Finds objects of a certain type by ID or by using a query.	Finders are a key concept in plug-in development. You use finders in plug-in checkpointing. See Basic Plug-In Concepts.
`Extension`	Extends the functionality of a model object.	You can add methods and fields to model objects in your code repository. If you expose model objects that derive from third-party libraries, you must wrap these objects and delegate calls to the wrapped object. **Note** Model-driven does this wrapping in a transparent way.

Add Inventory Objects

If you want to define the `Connection` class as a finder object, you must create a new finder object that implements the `ObjectFinder` interface. This automatically exposes the object as a scripting object but does not add it to the inventory tree.

Procedure

1 Create a `Connection` class as a finder object or modify the `Connection.java` file in the model directory.

```
@Component
@Qualifier(value = "connection")
@Scope(value = "prototype")
public class Connection implements Findable {
...
    @Override
    public Sid getInternalId() {
        return getConnectionInfo().getId();
    }
    @Override
    public void setInternalId(Sid id) {
        // do nothing, we set the Id in the constructor
    }
...
}
```

Note If you want the platform to query objects by a filter, the `Connection` class must implement the `Findable` interface.

2 Under the `{plug-in-home}/o11nplugin-redis-core/src/main/java/com/vmware/o11n/plugin/redis` folder, create a directory with name `finder`.

3 Create the finder and save it as a `ConnectionFinder.java` file in the `finder` directory.

```
package com.vmware.o11n.plugin.redis.finder;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import com.vmware.o11n.plugin.redis.config.ConnectionRepository;
import com.vmware.o11n.plugin.redis.model.Connection;
import com.vmware.o11n.sdk.modeldriven.FoundObject;
import com.vmware.o11n.sdk.modeldriven.ObjectFinder;
import com.vmware.o11n.sdk.modeldriven.PluginContext;
import com.vmware.o11n.sdk.modeldriven.Sid;
public class ConnectionFinder implements ObjectFinder<Connection> {


    @Autowired
    private ConnectionRepository connectionRepository;
    /*
     * When a connection is found, we need to assign an ID. However, the
     * Connection has its own ID, so we don't need to do anything here, aside
     * from return the connection.
     */
    @Override
    public Sid assignId(Connection obj, Sid relatedObject) {
        return obj.getInternalId();
    }
    /*
     * Finds the connection by ID
     */
    @Override
    public Connection find(PluginContext ctx, String type, Sid id) {
        return connectionRepository.findLiveConnection(id);
    }

    @Override
    public List<FoundObject<Connection>> query(PluginContext ctx, String type, String query) {
        Collection<Connection> allConnections = connectionRepository.findAll();
        List<FoundObject<Connection>> result = new LinkedList<>();
        boolean returnAll = "".equals(query);
        for (Connection connection : allConnections) {
            if (returnAll || connection.getName().toLowerCase().startsWith(query.toLowerCase()))
{
                FoundObject<Connection> foundObject = new FoundObject<>(connection);
                result.add(foundObject);
            }
```

```
        }
        return result;
    }
}
```

Note You can inject `ConnectionRepository` directly to the finder.

The find method is invoked every time the platform retrieves a Connection by its ID. When the plug-in returns a model object to the platform, it uses `assignedId`. The `assignedId` method sets an ID to the object that is returned to the platform. You can use this ID to restore the object later.

4 Expose the finder object as a scripting object, by defining it in `CustomMapping`.

```
wrap(Connection.class).andFind().using(ConnectionFinder.class).withIcon("connection.png");
```

5 Build the plug-in.

6 Install the modified version of the plug-in on the vRealize Orchestrator server.

7 Log in to the vRealize Orchestrator Client.

8 From the left navigational panel, select **API Explorer**.

9 Expand the Redis entry, and browse through the API elements.

The `Connection` object is exposed as a scripting object and is also defined as a type, or a findable object. You can set this type as an input, output, or a variable of a workflow.

Create a Redis Connection Workflow

Create a workflow that connects to the redis plug-in.

Procedure

1 Log in to the vRealize Orchestrator Client.

2 Navigate to **Library > Workflows**.

3 Click **New Workflow**.

4 Enter a name for the new workflow, and click **Create**.

5 Select the **Inputs/Outputs** tab, and click **New**.

6 Under the **Name** text box, enter **`connection`**.

7 Under the **Type** text box, enter **`Redis:Connection`**.

8 Save the workflow, and click **Run**.

The **Search** text box appears.

The search string you enter is delegated to the `query` method of the `ConnectionFinder`. If you leave the text box empty, the plug-in returns all available connections.

Note The `wrap` method exposes the `Connection` as a scripting object and the `andFind` method marks the object as a type. You cannot define an object as a findable type if you do not use a finder class. So, you can use a `ConnectionFinder`.

Although there is a findable object and you can use a query to search for this object, the plug-in inventory tree is empty. If you want to see the available connections in the inventory tree, you must define the relation of the findable object.

The platform cannot identify the parent object of the `Connection` object and cannot retrieve objects of this type. The platform can only search for the object by ID.

9 Under the `{plug-in-home}/o11nplugin-redis-core/src/main/java/com/vmware/o11n/plugin/redis` folder, create a directory with name `relater`.

10 Use the `ObjectRelated` interface to create a `RootHasConnections` relater

```
package com.vmware.o11n.plugin.redis.relater;
import java.util.ArrayList;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import com.vmware.o11n.plugin.redis.config.ConnectionRepository;
import com.vmware.o11n.plugin.redis.model.Connection;
import com.vmware.o11n.sdk.modeldriven.ObjectRelater;
import com.vmware.o11n.sdk.modeldriven.PluginContext;
import com.vmware.o11n.sdk.modeldriven.Sid;

public class RootHasConnections implements ObjectRelater<Connection> {


    @Autowired
    private ConnectionRepository connectionRepository;

    @Override
    public List<Connection> findChildren(PluginContext ctx, String relation, String parentType,
Sid parentId) {
        return new ArrayList<>(connectionRepository.findAll());
    }
}
```

The `RootHasConnections` relater creates the relation between the root of the inventory and a list of connections.

Note The relater only returns all connection objects. When you relate the root of a tree to an object, the arguments of the `findChildren` method become irrelevant.

11 Use the `define` method to define the relation in `CustomMapping`.

```
relateRoot().to(Connection.class).using(RootHasConnections.class).as("connections");
```

12 Build the plug-in.

13 Install the modified version of the plug-in on the vRealize Orchestrator server. See Deploy a Plug-In.

14 Log in to the vRealize Orchestrator Client.

15 Navigate to the **Inventory** page, and expand the Redis plug-in entry.

The **relateRoot()** method relates objects to the root of the tree. There is one additional method for adding inventory objects to the tree – `relate()`.

Wrap the Client

To create the integration with the third-party system, for which you design the plug-in, you must add the Maven dependencies.

You introduce a dependency with a small Java-based client called Jedis and the Apache common pool library, on which the Jedis library depends

Procedure

1 Define the dependency in the main `pom.xml` file.

```
...
<properties>
    <jedis.version>2.8.0</jedis.version>
    <commons.pool.version>2.4.2</commons.pool.version>
</properties>
...
<dependency>
    <groupId>redis.clients</groupId>
    <artifactId>jedis</artifactId>
    <version>${jedis.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
    <version>${commons.pool.version}</version>
</dependency>
```

2 Add a dependency in the `pom.xml` file in the `o11nplugin-redis-core` folder.

```
<dependency>
   <groupId>redis.clients</groupId>
   <artifactId>jedis</artifactId>
</dependency>
<dependency>
   <groupId>org.apache.commons</groupId>
   <artifactId>commons-pool2</artifactId>
</dependency>
```

3 Define the use of Jedis.

```
//Construct a pool of redis connections
JedisPool pool = new JedisPool(new JedisPoolConfig(), "localhost");

/// Jedis implements Closable. Hence, the jedis instance will be auto-closed after the last
statement.
try (Jedis jedis = pool.getResource()) {
  /// ... do stuff here ... for example
  jedis.set("foo", "bar");
  String foobar = jedis.get("foo");
  jedis.zadd("sose", 0, "car"); jedis.zadd("sose", 0, "bike");
  Set<String> sose = jedis.zrange("sose", 0, -1);
}
/// ... when closing your application:
pool.destroy();
```

Multiple threads, such as workflows and user interactions, will access the `Connection` class. You must make sure that you have either a pool of Redis connections, which is provided by default, or a thread-safe connection.

a When you create the `Connection` object, you initialize the `JedisPool`.

b Retrieve the Jedis object in the try-statement as a resource.

c Close the resource by returning it to the `JedisPool`, when it is not used any more.

d When the connection is destroyed, you must destroy the `JedisPool`.

```
@Component
@Qualifier(value = "connection")
@Scope(value = "prototype")
public class Connection implements Findable {
    ...
    private JedisPool pool;

    @ExtensionMethod
    public String ping() {
        try (Jedis jedis = getResource()) {
            return jedis.ping();
        }
    }

    @ExtensionMethod
    public String set(String key, String value) {
        try (Jedis jedis = getResource()) {
            return jedis.set(key, value);
        }
    }

    @ExtensionMethod
    public String get(String key) {
        try (Jedis jedis = getResource()) {
            return jedis.get(key);
        }
```

```
    }

    public synchronized void destroy() {
        if (pool != null) {
            pool.destroy();
        }
    }

    /**
     * Returns a redis connection from the pool.
     */
    public Jedis getResource() {
        return getPool().getResource();
    }

    /*
     * Lazy initialization of the pool.
     */
    private synchronized JedisPool getPool() {
        if (pool == null) {
            JedisPoolConfig jedisConfig = new JedisPoolConfig();
            pool = new JedisPool(jedisConfig, connectionInfo.getHost(),
connectionInfo.getPort());
        }
        return pool;
    }
    ...
}
```

Important Notes on the Jedis Code

The Jedis implementation initializes only when you attempt to perform an operation over the `Connection` object. This type of initialization is useful when you manage thousands of connections.

When you destroy a connection, you must destroy all related resources. To prevent a state of inconsistency, the `destroy` and the `getPool` methods are synchronized.

You must explicitly close the Jedis connection. For example, if you use the `ping` method, you retrieve a connection from the pool, run a call and close the connection, by returning it to the pool. If you want to run multiple calls, you must retrieve a connection multiple times. For performance reasons, you can expose the Jedis object to the users.

Create a Testing Redis Connection Workflow

After you add the Jedis functionality to the `Connection` object, you can use this functionality by creating a workflow.

Prerequisites

- Build the plug-in.
- Install the modified version of the plug-in to the Orchestrator server. See Deploy a Plug-In.

Procedure

1. Log in to the vRealize Orchestrator Client.
2. Navigate to **Library > Workflows**.
3. Click **New Workflow**.
4. Enter a name for the workflow, and click **Create**.
5. Use `Redis:Connection` as a connection type to create an input parameter of the workflow.
6. Create a **Scriptable task** element by inserting the following code:

```
connection.set("plugin:tutorial", "Testing redis connection – success");
var result = connection.get("plugin:tutorial");
System.log(result);
```

7. Save the workflow, and click **Run**.
8. After the workflow run finishes, select the **Logs** tab to verify the connection status.

Results

Example:

What to do next

Additional Relations

Redis is a data structure store and therefore you can use it as a database. In Redis, there can be multiple databases identified by a number and the number of the default database is 0.

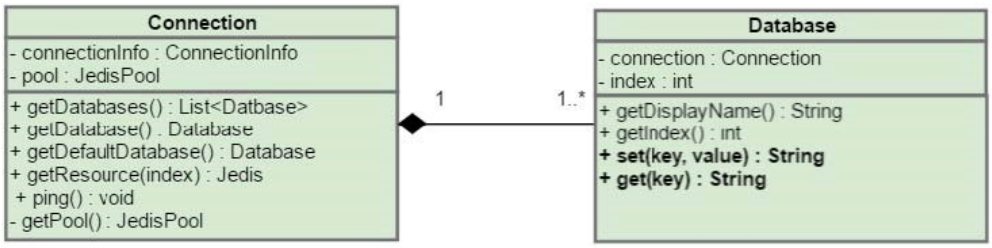
The commands you run through the Redis plug-in target the default Redis database, unless you specify otherwise.

```
Jedis jedis = getPool().getResource();
jedis.select(3); //Selects database with index 3
jedis.set("plugin:tutorial", "Using another database");
```

You can add to the plug-in the option to select a database that is different from the default one. To achieve this, you can use either of the two available methods.

Method	Description
Expose database index as part of the scripting API.	When a user calls a `set` method, for example, they can pass the database index as an extra function argument.
Present a new inventory object, Database, that wraps the index.	You must move all current methods from the `Connection` to the `Database` model object.

The following diagram displays the model of the second method.



You define the `Database` object in the `model` package. The `Database` object that you invoke instead of `get`, `set` or other methods, includes the following code.

```
private final Connection connection;
    private final int index;

    public Database(Connection connection, int index) {
        Assert.notNull(connection, "Connection cannot be null.");
        Assert.isTrue(index >= 0, "Index must be a positive number.");


        this.connection = connection;
        this.index = index;
    }
    public String getDisplayName() {
        return "db" + index;
    }
    public int getIndex() {
        return index;
    }
    @ExtensionMethod
    public String ping() {
        try (Jedis jedis = connection.getResource(index)) {
            return jedis.ping();
        }
    }
    @ExtensionMethod
    public String set(String key, String value) {
        try (Jedis jedis = connection.getResource(index)) {
            return jedis.set(key, value);
        }
    }
    @ExtensionMethod
    public String get(String key) {
        try (Jedis jedis = connection.getResource(index)) {
            return jedis.get(key);
        }
    }
    @ExtensionMethod
    public String get(String key) {
        try (Jedis jedis = connection.getResource(index)) {
            return jedis.get(key);
        }
```

```
    }
}
```

The `Connection` object also requires some changes. Some of them consist in removing methods, such as `get`, `set`, and `append`.

```
@Component
@Qualifier(value = "connection")
@Scope(value = "prototype")
public class Connection implements Findable {
    private static final int DEFAULT_REDIS_DATABASE_INDEX = 0;

    /*
     * The connectionInfo which stands behind this live connection.
     */

    private ConnectionInfo connectionInfo;
    private Map<Integer, Database> databases = null;
...
    public List<Database> getDatabases() {
        if (databases == null) {
            databases = new HashMap<>(16);
            //Issue a call to Redis, to see how many databases are configured, default is 16
            List<String> configs = getResource(DEFAULT_REDIS_DATABASE_INDEX).configGet("databases");
            int numberOfInstances = Integer.parseInt(configs.get(1));
            for (int index = 0; index < numberOfInstances; index++) {
                databases.put(index, new Database(this, index));
            }
        }
        return new ArrayList<>(databases.values());
    }

    @ExtensionMethod
    public Database getDatabase(int index) {
        return getDatabases().get(index);
    }

    @ExtensionMethod
    public Database getDefaultDatabase() {
        return getDatabase(DEFAULT_REDIS_DATABASE_INDEX);
    }
...
}
```

The modified `Connection` model initializes a map of database instances. You can find a database by its index. The `getDatabases()` method invokes a configuration command against the Redis instance and retrieves the count of the supported database instances. By default, the number of supported instances is 16.

You use `CustomMapping` to add a relation between a `Connection` object and a `Database` object.

```
    @Override
    public void define() {
      //@formatter:off
```

```
...
        wrap(Database.class).
            andFind().
            using(DatabaseFinder.class).
            withIcon("database.png");
...
        relate(Connection.class).
            to(Database.class).
            using(ConnectionHasDatabases.class).
            as("databases");
      //@formatter:on
    }
}
```

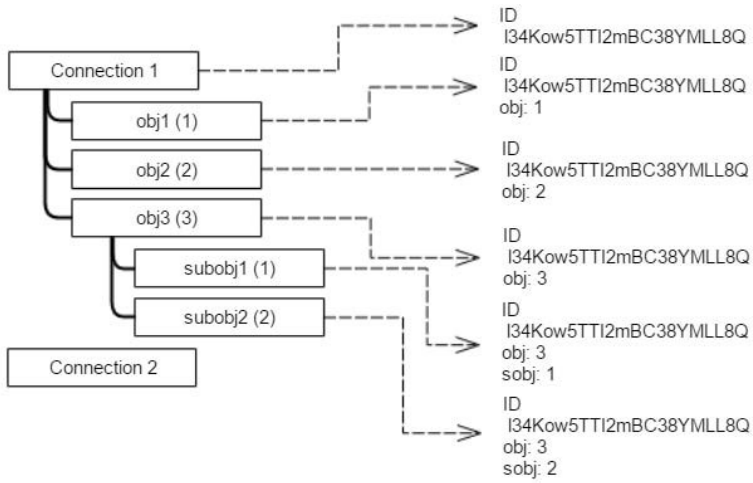
Note After you add a new inventory object, you must create a new `Finder` implementation and a new `Relater` implementation.

```
public class DatabaseFinder implements ObjectFinder<Database> {
    @Autowired
    private ConnectionRepository connectionRepository;


    @Override
    public Database find(PluginContext ctx, String type, Sid id) {
        Connection connection = connectionRepository.findLiveConnection(id);
        if (connection != null) {
            return connection.getDatabase((int) id.getLong("dbid", 0));
        }
        return null;
    }


    @Override
    public List<FoundObject<Database>> query(PluginContext ctx, String type, String query) {
        //Return null for now
        return null;
    }


    @Override
    public Sid assignId(Database obj, Sid relatedObject) {
        return relatedObject.with("dbid", obj.getIndex());
    }
}
```

Although the finder of the `Database` object is similar to `ConnectionFinder`, some major differences exist between these finders. While the `Connection` object is related to the root of the inventory tree and does not have a parent object, the `Database` object is a child object of the `Connection` object. When you invoke the `assignId` method, the `relatedObject` argument is the ID of the parent object, or the ID of the `Connection` object. You can track a child object by the ID of its parent object.

The `relatedObject.with("dbid", obj.getIndex());` implementation creates a new ID based on the connection ID. The `Database` object ID includes the `Connection` object ID and the index of the database instance. By using this method, you identify a single database instance among all `Connection` objects.

The following diagram shows the relations between several layers of objects and their corresponding IDs.



For example, `Connection 1` has an ID `l34Kow5TTI2mBC38YMLL8Q`. `Connection 1` also has sub objects: `obj1`, `obj2` and `obj3`, whose natural IDs are 1, 2, and 3 respectively. `obj3` has a set of subobjects, namely `subobj1` and `subobj2`.

There are three types of objects, so you need three finders – for `Connections`, for `obj` and for `subobj` objects.

The `assignId` method for the connection returns only the ID of the connection. The `assignId` method for the finder of the first-level objects, `obj`, returns the ID of the connection and the ID of the `obj`. The `assignId` method for the finder of the `subobj` objects returns the ID of the parent object and the ID of the subobject.

A structure, similar to a map, stores the values of the ID for each object.

Note During checkpointing, the vRealize Orchestrator server stores only the ID of the object, the method that mixes the IDs makes it possible to retrieve the connection ID of a subobject and, at the same time, retrieve the ID of the `Connection` object.

Defining the finder of the `Database` object is not enough to show the database objects in the inventory tree. By using `CustomMapping`, you must define the parent object of the `Database` object and pass it to the vRealize Orchestrator platform. By introducing the `ConnectionHasDatabase` class, you can find a set of databases for a certain connection.

```
public class ConnectionHasDatabases implements ObjectRelater<Database> {


    @Autowired
    private ConnectionRepository connectionRepository;
```

```
    @Override
    public List<Database> findChildren(PluginContext ctx, String relation, String parentType, Sid
parentId) {
        Connection connection = connectionRepository.findLiveConnection(parentId);
        if (connection != null) {
            return connection.getDatabases();
        }
        return Collections.emptyList();
    }
}
```

Similarly to using the `ConnectionFinder` class, when you know the ID of the `Connection` object, which is a parent object, you can use `ConnectionRepository` to find the connection instance. To retrieve the result, you must invoke the `getDatabases()` method.

If you run the workflow from the Create a Testing Redis Connection Workflow section, you receive the following error message: `TypeError: Cannot find function set in object DynamicWrapper (Instance) : [RedisConnection]-[class com.vmware.o11n.plugin.redis_gen.Connection_Wrapper] -- VALUE :.`

To resolve the error, you must recreate the scripting of the **Testing Redis Connection** workflow.

```
connection.defaultDatabase.set("plugin:tutorial", "Testing redis connection - success");
var result = connection.defaultDatabase.get("plugin:tutorial");
System.log(result);
```

The modified workflow uses the same `Redis:Connection` connection parameter but retrieves the default database.