Oracle® Fusion Middleware

Integrating Oracle WebLogic Server with Helidon

12c (12.2.1.4.0)
F79200-01
July 2023

Oracle Fusion Middleware Integrating Oracle WebLogic Server with Helidon, 12c (12.2.1.4.0)

F79200-01

Copyright © 2023, Oracle and/or its affiliates.

Primary Author: Oracle Corporation

Contents

Preface

The *Integrating Oracle WebLogic Server with Helidon* document provides guidance on leveraging the integration of Oracle WebLogic Server and Helidon to establish a connection between applications deployed on Oracle WebLogic Server and Helidon microservice applications running in a Kubernetes cluster.

This preface includes the following topics:

- Audience
- Documentation Accessibility
- Diversity and Inclusion
- Related Resources
- Conventions

Audience

This document is intended for users who are responsible for using the Oracle WebLogic Server integration with Helidon in the following areas:

- WebLogic JMS (Java Messaging Service)
- REST Services
- Web Services
- Single sign-on (SSO) on Oracle Cloud Infrastructure (OCI)

This document provides information about integrating Oracle WebLogic Server 12c (12.2.1.4) with Helidon 3.x and 2.x.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at `https://www.oracle.com/corporate/accessibility/`.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `https://support.oracle.com/portal/` or visit `Oracle Accessibility Learning and Support` if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

Related Resources

For more information, see these Oracle resources:

- WebLogic Kubernetes Operator
- Helidon MP 2.x Upgrade Guide
- Helidon MP 3.x Upgrade Guide
- Helidon MP Tutorial
- Helidon MP Quickstart

Conventions

The following text conventions are used in this document.

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1 About the Oracle WebLogic Server and Helidon Integration

The Oracle WebLogic Server (WebLogic Server) and Helidon integration enables interaction between a Helidon microservice application and an application installed on WebLogic Server, which is deployed in a Kubernetes cluster managed by the WebLogic Kubernetes Operator (Operator).

> **Note:**
>
> This document provides information about the WebLogic Server and Helidon integration when both products are deployed in a Kubernetes environment. However, you can implement this integration on any supported platform.

Oracle WebLogic Server on Kubernetes enables you to efficiently build modern container applications with comprehensive Java services. Helidon is an open source, lightweight, fast, reactive, cloud native framework for developing Java microservices.

Owners of applications based on Oracle WebLogic Server seeking to modernize applications with microservices can use Oracle WebLogic Server and Helidon integration to implement communication and coordination between Oracle WebLogic Server-based applications and Helidon-based microservices. Such communication and coordination enables enterprise applications and microservices to coexist and cooperate in the realization of modernized architecture meeting the owners' requirements. The communication can take multiple forms, unidirectional or bi-directional, and the coordination targets security simplifications between Oracle WebLogic Server applications and Helidon microservices.

> **Note:**
>
> This document provides information about the Oracle WebLogic Server and Helidon integration when both products are deployed in a Kubernetes environment. However, you can implement this integration on any supported platform.

This document provides information about the Oracle WebLogic Server integration with Helidon 3.x and 2.x releases. To understand the differences between the two Helidon releases, see Helidon MP 2.x Upgrade Guide and Helidon MP 3.x Upgrade Guide.

The Oracle WebLogic Server and Helidon integration enables you to:

- Initiate bidirectional REST calls between Helidon and WebLogic Server. The REST integration enables WebLogic Server applications and Helidon microservices to communicate through RESTful Web Service invocations.
- Generate, produce, and consume a JMS message to and from a WebLogic Server Queue, Topic, Distributed Queue, and so on. The JMS integration enables Helidon microservices to publish and consume messages from WebLogic JMS Server.

- Initiate communication between a Helidon client and WebLogic Server Web Services. This integration enables Helidon microservices to interact with WebLogic Server applications through SOAP (Simple Object Access Protocol) Web Service calls from Helidon to WebLogic Server.
- The WebLogic Server and Helidon integration enables communication between a WebLogic cluster-hosted application and a Helidon microservice application by implementing Single Sign-on (SSO) authentication using Oracle Identity Cloud Service (IDCS).

This chapter includes the following topics:

- Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration
  Preparing the Kubernetes cluster for WebLogic Server and Helidon integration for REST, Java Message Service, Web Services, and Single Sign-on includes provisioning the WebLogic Server domain and the Helidon instances in a Kubernetes cluster, deploying the Operator, and deploying a load balancer or the Istio service mesh.
- Downloading the WebLogic Server Java Clients with Jakarta Package Names

Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration

Preparing the Kubernetes cluster for WebLogic Server and Helidon integration for REST, Java Message Service, Web Services, and Single Sign-on includes provisioning the WebLogic Server domain and the Helidon instances in a Kubernetes cluster, deploying the Operator, and deploying a load balancer or the Istio service mesh.

Preparing the Kubernetes cluster for these integrations (REST, Java Message Service, Web Services, and Single Sign-on) between WebLogic Server and Helidon includes provisioning the WebLogic Server domain and the Helidon instances in a Kubernetes cluster, deploying the Operator, and deploying a load balancer or the Istio service mesh.

Installing the Ingress Load Balancer

For effective traffic management between WebLogic Server and Helidon, you will need to deploy an Ingress controller or an Istio service mesh in the Kubernetes cluster. You can deploy any load balancer (or Ingress controller) in the Kubernetes cluster to balance traffic between WebLogic Server and the Helidon instances.

For more information about creating Traefik or Nginx Ingress controllers, see Ingress Controllers in the WebLogic Kubernetes Operator documentation.

Istio is a service mesh that provides a separate infrastructure layer to handle inter-service communication. For instructions to install Istio, see Install Istio.

Installing the WebLogic Kubernetes Operator

In this integration, the applications installed on WebLogic Server are deployed in a Kubernetes cluster that is managed by the Operator. You should install the Operator before provisioning the WebLogic Server domain. The Operator helps you manage all lifecycle operations of the WebLogic Server domain, such as provisioning, scaling, security, and lifecycle management which includes applying updates of the applications or to the WebLogic Server binaries in a rolling fashion.

For information about setting up the Operator, see Install the Operator.

Provisioning the WebLogic Server Domain

The Operator supports the following WebLogic Server domain home source types:

- **Model in Image:** The primary image contains the JDK and the WebLogic Server binaries. A separate, auxiliary image contains the WebLogic Deployment Tooling (WDT) model files, WDT variable files, and the application archive file. See Auxiliary Images in the WebLogic Kubernetes Operator documentation.
- **Domain Home in Persistent Volume (PV):** The primary image contains the JDK and the WebLogic Server binaries. The domain home and application binaries are in a shared PV. See Domain Home on a PV .

One of the many differences between these domain home source types is how you use them to create and update the WebLogic Server images so that the Operator can apply the update in a rolling fashion to ensure application availability. For information about the differences between these domain home source type, see Choose a Domain Home Source Type.

This document describes the provisioning of the WebLogic Server domain using the 'Model in Image' pattern. For a step-by-step guide on how to provision a domain with the 'Model in Image' pattern and create the required data sources, see Model in Image.

The Operator starts the WebLogic Server domain by using the Domain Custom Resource. There are two ways to start the domain:

- If you are starting the domain for the first time, you should provision the domain by creating the Domain Custom Resource.
- If the domain is already provisioned, you should edit the Domain Custom Resource to instruct the Operator to start the domain.

For more information about starting and stopping the WebLogic Server instances in your domain, see Domain Life Cycle.

For an overall understanding of running and managing the WebLogic Server domains, see Manage Domains.

Deploying Helidon

For information about the prerequisites and getting started with Helidon:

- See Get Started for Helidon 3.x.
- See Get Started for Helidon 2.x.

Downloading the WebLogic Server Java Clients with Jakarta Package Names

You can download the WebLogic Server 14.1.1.0 Java clients with Jakarta packages from Oracle Technology Network (OTN) or Oracle Software Delivery Cloud (OSDC). You can use these clients in remote Java client applications that contain libraries using the jakarta.* package naming convention and connect to server-side applications hosted in WebLogic Server 12.2.1.4 or WebLogic Server 14.1.1. 0.

To download the WebLogic Server Jakarta thin clients from OTN:

1. Go to the Free Oracle WebLogic Server Installers for Development page or the Oracle WebLogic Server Installers page. The following downloads are available under **WebLogic Server 14.1.1 Java clients with Jakarta package names**:
   - WebLogic Server 14.1.1 Thin T3 Jakarta client (`wlthint3client.jakarta.jar` file)
   - WebLogic Server 14.1.1 Web Services Jakarta client (`com.oracle.webservices.wls.jaxws-wlss-client.jakarta.jar` file)
2. Select the required file, check the Oracle License Agreement, and then click the file to download.

To download the clients from OSDC:

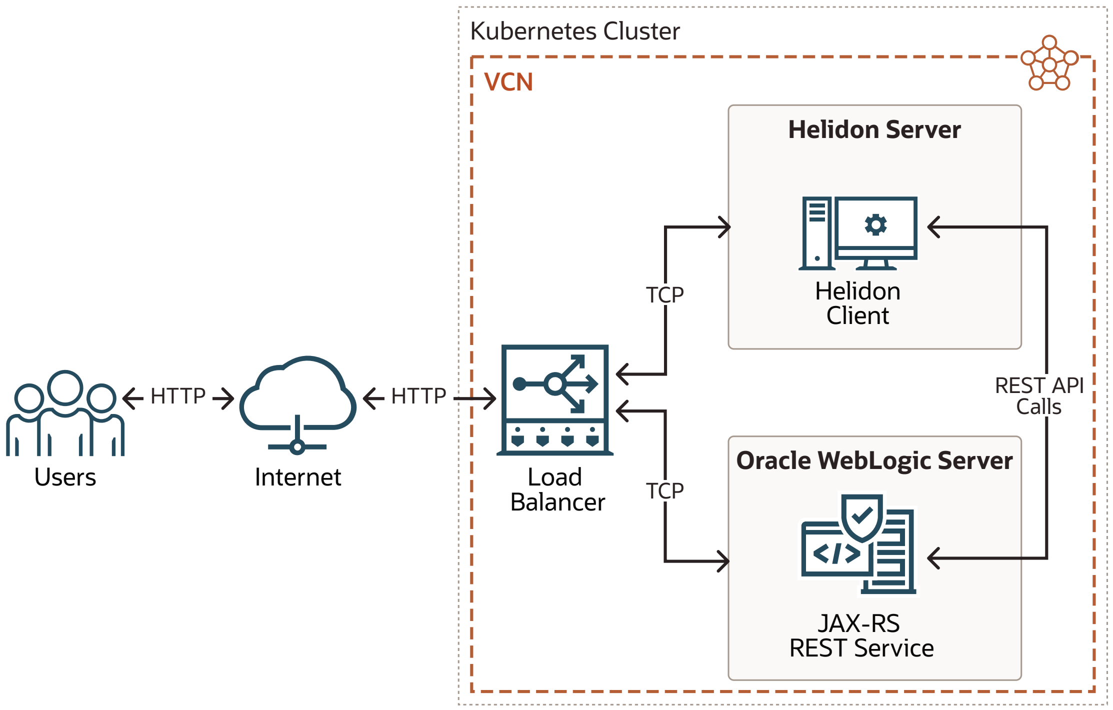
1. Sign in to Oracle Software Delivery Cloud.
2. Type in the search term 'Oracle WebLogic Server 14c' and click **Search**.
3. From the search results, select **Oracle WebLogic Server 14c 14.1.1.0.0 (Oracle WebLogic Server Enterprise Edition, Oracle WebLogic Server Standard Edition)**. It gets added to the downloads queue.
4. Click **View Items** and select **Continue** to view the list of items in your download queue. By default, all the items are selected for download.
5. Select only **Oracle WebLogic Server 14.1.1.0** from the list (uncheck the others), select **Platforms** as **GENERIC (ALL Platforms)**, and then click **Continue**.
6. Review and accept the Oracle License Agreement and click **Continue**.
7. From the downloads queue, select **Oracle WebLogic Server 14.1.1.0.0 for GENERIC (All Platforms)** if you want to download both the WebLogic 14.1.1 Jakarta clients, and click **Download**.
8. Sign out of the page after the download process is complete.
9. Before you extract the Jar files, rename the WebLogic Server 14.1.1 Web Services Jakarta client to `com.oracle.webservices.wls.jaxws-wlswss-client.jakarta.jar` and WebLogic Server 14.1.1 Thin T3 client to `wlthint3client.jakarta.jar`.

2

Integrating WebLogic Server REST Services with Helidon

The REST services integration with Helidon enables bidirectional REST calls between Helidon and Oracle WebLogic Server (WebLogic Server). RESTful integration between WebLogic Server and Helidon MP is easy to develop and maintain because both runtimes support JAX-RS for serving and calling the RESTful resources. With the Jakarta EE support, you can create the same RESTful resource or client, which will work in both environments. The following graphic illustrates the bidirectional communication between REST services, Helidon, Kubernetes cluster, VCN, and load balancer:

Figure 2-1 REST Services Integration with Helidon



The main difference in the usage of JAX-RS between Helidon and WebLogic Server is the version of the supported Jakarta specification. While WebLogic Server supports Jakarta EE 8, Helidon supports JAX-RS or the new Jakarta RESTful Web Services from Jakarta EE 9.1. The most notable difference between these two versions of Jakarta EE is the change in the package name, where `javax` is replaced with `jakarta`.

While imports from the `jakarta` namespace needs to be used in Helidon 3.x, for WebLogic Server and Helidon 2.x, `javax` should be used for the same JAX-RS code.

Helidon JAX-RS Imports for 3.x

```
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.Context;
import jakarta.ws.rs.core.MediaType;
```

Helidon JAX-RS Imports for 2.x

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
```

WebLogic Server JAX-RS Imports

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
```

This chapter includes the following topics:

- Prerequisites
- Using the JAX-RS Server
- Using the JAX-RS Client
  JAX-RS provides a convenient client API for calling the RESTful resources. The client enables you to prepare and execute the RESTful request call with a simple builder pattern API.

Prerequisites

To integrate WebLogic Server with Helidon for REST services, it is assumed that you have already deployed WebLogic Server and Helidon in a Kubernetes cluster with the WebLogic Kubernetes Operator (Operator). See Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration.
To deploy WebLogic Server, ensure that you have:

- A good understanding about the Operator. See WebLogic Kubernetes Operator.
- Installed and configured the Operator. See Model in Image.

To deploy Helidon, see:

- Prerequisites for 3.x
- Prerequisites for 2.x

To download the WebLogic Server Java clients with Jakarta packages, see Downloading the WebLogic Server Java Clients with Jakarta Package Names.

Using the JAX-RS Server

The JAX-RS resource is a simple bean with annotated methods representing routes and HTTP methods under a specific path. Annotated methods are invoked when a particular REST endpoint is called. All the mapping and routing are done by the actual implementation of the JAX-RS standard according to the JAX-RS annotations.
JAX-RS Example Resource

```
@Path("/greet")                                     (1)
public class GreetResource {

    @Path("/hello")
    @GET                                            (2)
    @Produces(MediaType.TEXT_PLAIN)                 (3)
    public Response getHello() {
        return Response.ok("Hello World!")          (4)
                       .build();
    }
}
```

A brief description of the above example:

(1) Path of the resource.

(2) HTTP method.

(3) Expected response content type.

(4) Returns the text payload with status 200.

JAX-RS is a very powerful tool where you can register your message body writers, readers, filters, or exception mappers. In both Helidon and WebLogic Server, Eclipse Jersey is used as the JAX-RS implementation. See Jersey User Guide.

For information about creating and deploying the JAX-RS RESTful resources in Helidon, see JAX-RS applications.

For information about developing and deploying the JAX-RS resources on WebLogic Server, see Developing RESTful Web Services.

Using the JAX-RS Client

JAX-RS provides a convenient client API for calling the RESTful resources. The client enables you to prepare and execute the RESTful request call with a simple builder pattern API.

JAX-RS Client Example

```
Client client = ClientBuilder.newClient();
String res = client
    .target("http://localhost:8080")          (1)
    .path("/greet")                           (2)
    .request("text/plain")                    (3)
    .get(String.class);                       (4)  (5)
```

A brief description of the above example:

(1) Creates a new WebTarget with the default root URL.

(2) Prepares the request to a particular context path.

(3) Sets the expected response content type.

(4) Executes the GET request and blocks until the response is received.

(5) Parameter sets the expected response payload type; available body readers are used for parsing to the correct response payload type.

You can also register your own message body writers, readers, filters, or exception mappers.

For information about creating JAX-RS clients in Helidon, see Jakarta REST (JAX-RS) Client.

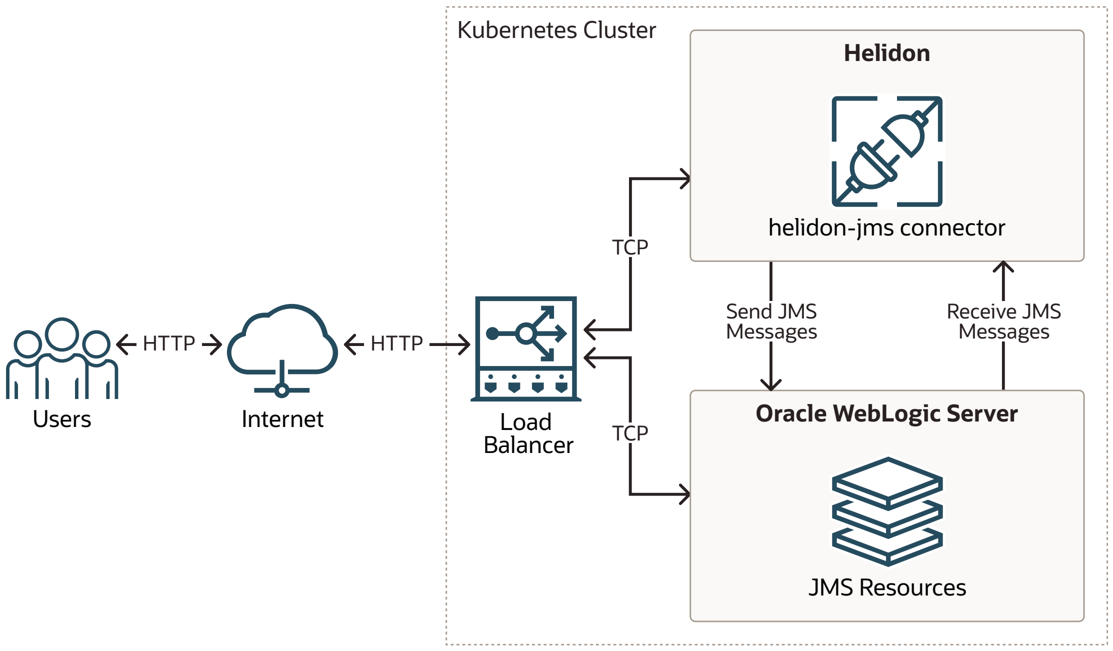
For information about developing and deploying JAX-RS clients on WebLogic Server, see Developing RESTful Web Services.

3

Integrating WebLogic Server JMS with Helidon

Integrating the Oracle WebLogic Server (WebLogic Server) Java Message Service (JMS) with Helidon enables the Helidon applications to send and receive messages to and from WebLogic Server asynchronously. The WebLogic Server applications and Helidon microservices communicate through messaging, in one or both directions.
The following graphic illustrates the transmission of messages between WebLogic Server JMS and Helidon:

Figure 3-1   WebLogic Server JMS Integration with Helidon



This chapter includes the following topics:

- Prerequisites
- Setting Up the JMS Integration with Helidon
- Troubleshooting Common JMS Issues
  Learn about the common issues you may encounter when setting up the integration between WebLogic Server and Helidon 3.x or 2.x.

Prerequisites

To integrate WebLogic Server with Helidon for JMS, it is assumed that you have already deployed WebLogic Server and Helidon in a Kubernetes cluster. See Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration.
In addition, ensure that you complete the following tasks:

- Configure/create the WebLogic Server JMS resources, before adding them as part of the Helidon JMS connector configurations.
- To send/receive JMS messages from Helidon to WebLogic Server JMS destinations, specify the WebLogic Server T3/T3S connection details. Enable the T3/T3S channels for communication in WebLogic Server and ensure that you are able to connect to T3/T3S from the Helidon application.
- Add the thin client JAR files to the local Maven repository and refer to it as part of the Maven dependencies.
  - **For Helidon 3.x**
    You can download the WebLogic Server 14.1.1 Thin T3 Jakarta client from Oracle Software Delivery Cloud (OSDC) for commercial use under WebLogic Server commercial licenses. Go to https://edelivery.oracle.com and download the package for Oracle WebLogic Server 14c 14.1.1.0.0 (Oracle WebLogic Server Enterprise Edition, Oracle WebLogic Server Standard Edition). To download the client for development use under the "Oracle Technology Network Free Developer License Terms", go to https://www.oracle.com/middleware/technologies/weblogic-server-downloads.html. For the steps to download, see Downloading the WebLogic Server Java Clients with Jakarta Package Names.
  - **For Helidon 2.x**
    You can locate the WebLogic Server Thin T3 client in your WebLogic Server installation under the `*WL_HOME*\server\lib` directory.
- Enabling the T3/T3S Channel in the WebLogic Kubernetes Operator
- Creating the Kubernetes Service for T3/T3S Channel for Communication
- Configuring the JMS Resources in WebLogic Server Using WebLogic Deploy Tooling

Enabling the T3/T3S Channel in the WebLogic Kubernetes Operator

WebLogic Server supports several ways to configure the T3/T3S channel in the WebLogic Kubernetes Operator (Operator). You can create the T3/T3S channel using one of the following options:

- WebLogic Server Administration Console: The Domain in Persistent Volume (PV) domain home pattern enables you to configure the channel using the WebLogic Server Administration Console. For more information about this pattern, see Domain Home on a PV.
- WebLogic Scripting Tool (WLST): The Domain in Persistent Volume (PV) domain home pattern enables you to configure the channel using WLST. See Domain Home on a PV.

- WebLogic Deploy Tooling (WDT) model: The Model in Image domain home pattern enables you to configure the channel using the WDT model files and application archives. See Model in Image.

For information about external WebLogic clients in WebLogic Kubernetes Operator, see External WebLogic Clients.

- Creating the T3/T3S Channel Using the Administration Console
- Creating the T3/T3S Channel Using the WLST Script
- Creating the T3/T3S Channel Using WebLogic Deploy Tooling

Creating the T3/T3S Channel Using the Administration Console

To create the T3/T3S channel for the Administration Server:

1. Log in to the WebLogic Server Administration Console and click **Lock & Edit** to obtain the configuration lock.
2. In the left pane of the console, expand **Environment** and select **Servers**.
3. On the Servers page, select the WebLogic Server Administration Server (for example: `admin-server`) from the list of servers.
4. Go to the **Protocols** tab, select the **Channels** tab, and then click **New**.
5. In the network channel **Name** field, enter `admin-t3-channel`, select **Protocol** as **t3**, and then click **Next**.
6. In the **Listen Port** field, enter `30014`, for **External Listen Address**, enter <*`Master IP`*>, and for **External Listen Port**, enter `30014`. Leave the **Listen Address** field blank.
7. Click **Finish** to create the network channel for admin-server.

To create the T3 channel for the dynamic cluster:

1. Log in to the WebLogic Server Administration Console and click **Lock & Edit** to obtain the configuration lock.
2. In the left pane of the console, expand **Environment** and select **Clusters**, and then select **Server Template**.
3. From the list of server templates, select the target server template (for example: server-template-1).
4. Go to the **Protocols** tab, click the **Channels** tab, and then click **New**.
5. In the network channel **Name** field, enter the name of the network channel (for example: cluster-t3-channel), select **Protocol** as `t3` or `t3s`, and then click **Next**.
6. In the **Listen Port** field, enter `30016`, for **External Listen Address**, enter <*`Master IP`*>, and for **External Listen Port**, enter `30016`. Leave the **Listen Address** field blank.
7. Click **Finish** to create the network channel for the dynamic cluster.

Note:

When WebLogic Server and Helidon are part of the same Kubernetes cluster, you can use the Fully Qualified Kubernetes Service Name as part of the public IP address.

Example of the FQDN name for the Administration Server:

```
wls-domain-admin-server.wls-domain-ns.svc.cluster.local
```

Example of the FQDN name for the dynamic cluster:

```
wls-domain-managed-server${id}.wls-domain-ns.svc.cluster.local
```

Creating the T3/T3S Channel Using the WLST Script

The following example script creates the T3 channel called `admin-t3-channel` that has a listen port `30014` and a public port `30014`:

```
import sys

admin_server = sys.argv[1]
admin_port = sys.argv[2]
user_name = sys.argv[3]
password = sys.argv[4]
domain_ns = sys.argv[5]

connect(user_name, password, 't3://' + admin_server + ':' + admin_port)

edit()
startEdit()
cd('/')

print('Create channel for admin server')
cd('/Servers/admin-server')
cmo.createNetworkAccessPoint('admin-t3-channel')
cd('NetworkAccessPoints/admin-t3-channel')
cmo.setProtocol('t3')
cmo.setListenPort(30014)
cmo.setPublicPort(30014)
##You need to use public IP address when Helidon and WebLogic Server 
are in different Kubernetes clusters.
cmo.setPublicAddress('wls-domain-admin-server.' + domain_ns + 
'.svc.cluster.local')
print('admin-t3-channel added')

print('Create channel for cluster')
cd('/ServerTemplates/server-template_1')
cmo.createNetworkAccessPoint('cluster-t3-channel')
cd('/ServerTemplates/server-template_1/NetworkAccessPoints/cluster-t3-
channel')
```

```
cmo.setProtocol('t3')
cmo.setListenPort(30016)
##You need to use public IP address when Helidon and WebLogic Server are in
different Kubernetes.
cmo.setPublicAddress('wls-domain-managed-server${id}.' + domain_ns +
'.svc.cluster.local')
cmo.setEnabled(true)
cmo.setHttpEnabledForThisProtocol(true)
cmo.setTunnelingEnabled(false)
cmo.setOutboundEnabled(false)
cmo.setTwoWaySSLEnabled(false)
cmo.setClientCertificateEnforced(false)
print('cluster-t3-channel added')

activate()
disconnect()
```

Creating the T3/T3S Channel Using WebLogic Deploy Tooling

You can create the T3/T3S channel resources by using WDT. The following example uses the WDT model configuration for creating the T3 channels for the Administration Server and the dynamic cluster:

```
topology:
    Server:
        admin-server:
            ListenAddress: wls-domain-admin-server
            NetworkAccessPoint:
                internal-t3:
                    ListenAddress: localhost
                    ListenPort: 7001
                admin-t3-channel:
                    # You need to use public IP address when Helidon and
WebLogic Server are in different Kubernetes.
                    PublicAddress: wls-domain-admin-server.wls-domain-
ns.svc.cluster.local
                    ListenPort: 30014
                    PublicPort: 30014
    ServerTemplate:
        server-template_1:
            Cluster: cluster-1
            ListenAddress: wls-domain-managed-server${id}
            ListenPort: 8001
            NetworkAccessPoint:
                cluster-t3-channel:
                    # You need to use public IP address when Helidon and
WebLogic Server are in different Kubernetes clusters.
                    PublicAddress: wls-domain-managed-server${id}.wls-domain-
ns.svc.cluster.local
                    ListenPort: 30016
```

Creating the Kubernetes Service for T3/T3S Channel for Communication

After you create the T3/T3S channels in the WebLogic Server domain, you should enable the channel for communication in the Kubernetes cluster. The following is an example of creating the Kubernetes service for T3 communication:

```
apiVersion: v1
kind: Service
metadata:
   name: adminserver-t3-external
   namespace: wls-domain-ns
   labels:
     weblogic.serverName: admin-server
spec:
  type: NodePort
  selector:
    weblogic.serverName: admin-server
  ports:
  - name: t3adminport
    protocol: TCP
    port: 30014
    targetPort: 30014
    nodePort: 30014
---
apiVersion: v1
kind: Service
metadata:
   name: cluster-t3-external
   namespace: wls-domain-ns
   labels:
     weblogic.clusterName: cluster-1
spec:
  type: NodePort
  selector:
    weblogic.clusterName: cluster-1
  ports:
  - name: t3clusterport
    protocol: TCP
    port: 30016
    targetPort: 30016
    nodePort: 30016
```

Note:

With this sample code, you can refer the T3/T3S Kubernetes services by using the following sample URLs:

- Refer the admin T3 external URL using `t3://adminserver-t3-external.wls-domain-ns.svc.cluster.local:30014`.
- Refer the cluster T3 external URL using `t3://cluster-t3-external.wls-domain-ns.svc.cluster.local:30016`.

For information about external WebLogic clients in WebLogic Kubernetes Operator, see External WebLogic Clients.

Configuring the JMS Resources in WebLogic Server Using WebLogic Deploy Tooling

A simple example of a model to deploy the JMS resources by targeting them to the WebLogic Administration Server and the dynamic cluster using WDT.

```
resources:
    FileStore:
        FileStoreCluster:
            Target: cluster-1
            Directory: wlsdeploy/stores/FileStoreCluster/
        FileStoreAdmin:
            Target: admin-server
            Directory: wlsdeploy/stores/FileStoreAdmin/
    JMSServer:
        JMSServerCluster:
            Target: cluster-1
            PersistentStore: FileStoreCluster
        JMSServerAdmin:
            Target: admin-server
            PersistentStore: FileStoreAdmin
            JmsSessionPool:
                JMSSessionPool0: {}
    JMSSystemResource:
        jmsmodulecluster:
            Target: cluster-1
            SubDeployment:
                jmssdcluster:
                    Target: JMSServerCluster
            JmsResource:
                ConnectionFactory:
                    dqcf:
                        DefaultTargetingEnabled: true
                        JNDIName: dqcf
                        ClientParams:
                            MessagesMaximum: 1
                        LoadBalancingParams:
                            ServerAffinityEnabled: false
                        TransactionParams:
                            XAConnectionFactoryEnabled: true
                UniformDistributedQueue:
                    udq:
                        JNDIName: udq
                        SubDeploymentName: jmssdcluster
                        MessagingPerformancePreference: 0
                        DeliveryFailureParams:
                            RedeliveryLimit: 600
                        DeliveryParamsOverrides:
                            RedeliveryDelay: 1000
                            DeliveryMode: Persistent
        jmsmoduleadmin:
            Target: admin-server
            SubDeployment:
                jmssdadmin:
                    Target: JMSServerAdmin
            JmsResource:
                ConnectionFactory:
                    qcf:
                        DefaultTargetingEnabled: true
                        JNDIName: qcf
                        ClientParams:
                            MessagesMaximum: 1
                        LoadBalancingParams:
                            ServerAffinityEnabled: false
                        TransactionParams:
                            XAConnectionFactoryEnabled: true
                Queue:
                    queue:
                        JNDIName: queuejndi
                        SubDeploymentName: jmssdadmin
                        MessagingPerformancePreference: 0
                        DeliveryFailureParams:
                            RedeliveryLimit: 600
                        DeliveryParamsOverrides:
                            RedeliveryDelay: 1000
                            DeliveryMode: Persistent
                Topic:
                    myTopic:
                        JNDIName: myTopic
                        SubDeploymentName: jmssdadmin
```

Setting Up the JMS Integration with Helidon

Before you begin the integration steps, ensure that you have created the required JMS resources and the T3/T3S channels. For information about creating the T3/T3S channels, see Enabling the T3/T3S Channel in the WebLogic Kubernetes Operator. Set up the integration between WebLogic Server Java Message Service (JMS) and Helidon by adding the required dependencies in the `pom.xml` file and configuring the WebLogic Server JMS connector. These dependencies enable reactive streaming and messaging along with JMS.

To set up the integration:

1. Add the following dependencies to the `pom.xml` file of Helidon:

   Dependency for Reactive Messaging

   ```
   <dependency>
      <groupId>io.helidon.microprofile.messaging</groupId>
      <artifactId>helidon-microprofile-messaging</artifactId>
   </dependency>
   ```

   Dependency for JMS Connector

   ```
   <dependency>
       <groupId>io.helidon.messaging.jms</groupId>
       <artifactId>helidon-messaging-jms</artifactId>
   </dependency>
   ```

   Dependency for Messaging Health

   ```
   <dependency>
       <groupId>io.helidon.microprofile.messaging</groupId>
       <artifactId>helidon-microprofile-messaging-health</artifactId>
   </dependency>
   ```

   Dependencies for the WLS Thin Client JAR File

   If you are using Helidon 3.x, add the **jakarta** thin client JAR file as part of the Maven compilation/runtime dependencies, as shown below:

   ```
   <dependency>
       <groupId>wlthint3client.jakarta</groupId>
       <artifactId>wlthint3client-jakarta</artifactId>
       <version>1.0</version>
   </dependency>
   ```

   The following example shows how you can add the jakarta thin client to the Maven repository:

   ```
   mvn install:install-file -Dfile=<JAR_FILE_PATH>/
   wlthint3client.jakarta.jar -DgroupId=wlthint3client.jakarta -
   DartifactId=wlthint3client-jakarta -Dversion=1.0
   ```

   If you are using Helidon 2.x, add the **javax** thin client JAR file as part of the Maven compilation/runtime dependencies, as shown below:

   ```
   <dependency>
       <groupId>wlthint3client</groupId>
       <artifactId>wlthint3client</artifactId>
       <version>1.0</version>
   </dependency>
   ```

The following example shows how you can add the javax thin client to the Maven repository:

```
mvn install:install-file -Dfile=<WLS_ORACLE_HOME>/wlserver/
server/lib/wlthint3client.jar -DgroupId=wlthint3client -
DartifactId=wlthint3client -Dversion=1.0
```

> **Note:**
>
> Ensure that the values for `groupId`, `artifactId`, and the `version` are identical to the values used in the `mvn install:install-file` command.

2. Configure the Helidon JMS connector. For more information about the connector, see the following documents:
   - For Helidon 3.x: Helidon MP - JMS Connector
   - For Helidon 2.x: Helidon MP - JMS Connector

   The configuration includes the following information:
   - The JMS environment properties that are used to lookup resources on the WebLogic Server:
     - WLS INITIAL_CONTEXT_FACTORY (`java.naming.factory.initial`): `weblogic.jms.WLInitialContextFactory`
     - SECURITY_PRINCIPAL (`java.naming.security.principal`): The user name for WebLogic Server.
     - SECURITY_CREDENTIALS (`java.naming.security.credentials`): The password for WebLogic Server.
     - PROVIDER_URL (`java.naming.provider.url`): The WebLogic Server T3/T3S connection URL.
   - The JMS resource details for the following:
     - JMS connection factory
     - JMS destination
     - JMS destination type

   The following example shows the **helidon-jms** connector configurations added to the `<src>/main/resources/application.yaml` file. In this example, JMS clients use the Java Naming and Directory Interface (JNDI) naming service. Hence, this example uses the `jndi.destination` key to refer the JMS destination name instead of using the `destination` key.

```
# User-defined properties
wls-username: <wls_username>
wls-password: <wls_password>
# WLS Admin server t3 connection URL
wls-admin-url: t3://localhost:7001
# WLS Admin T3 Kubernetes Service URL format within the same
kubernetes cluster
# wls-admin-url: t3://adminserver-t3-external.wls-domain-
```

```
ns.svc.cluster.local:30014
# WLS Cluster t3 connection URL
wls-cluster-url: t3://localhost:7003,localhost:7005,localhost:7007
# WLS Cluster T3 Kubernetes Service URL format within same Kubernetes
cluster
# wls-cluster-url: t3://cluster-t3-external.wls-domain-
ns.svc.cluster.local:30016

mp:
  messaging:
    connector:
      helidon-jms:
        jndi:
           #Default connection factory name. This can be overridden in
individual resource configurations
           jms-factory: qcf
           #JMS environment properties to lookup resources
           env-properties:
              java.naming.factory.initial:
weblogic.jms.WLInitialContextFactory
              java.naming.provider.url: ${wls-admin-url}
              java.naming.security.principal: ${wls-username}
              java.naming.security.credentials: ${wls-password}

    # Add all consumer resources-related configurations below incoming.
    incoming:
      #Identifier "from-wls-q" is used with the @Incoming annotation.
      from-wls-q:
        #Connector Name as specific in connector section. It is
predefined.
        connector: helidon-jms
        #JMS Destination Name in JNDI format.
        jndi.destination: queuejndi
        #JMS Destination Type.
        type: queue
        #JMS Connection Factory.
        jndi.jms-factory: qcf

    # Add all producer resources-related configurations below outgoing.
    outgoing:
      #Identifier "to-wls-q" is used with the @Outgoing annotation.
      to-wls-q:
        connector: helidon-jms
        jndi.destination: queuejndi
        type: queue
        jndi.jms-factory: qcf
```

This is an example of the **helidon-jms** connector configurations for a distributed queue:

```
mp:
  messaging:
    connector:
      helidon-jms:
        jndi:
           #Default connection factory name. This can be overridden
in individual resource configurations.
           jms-factory: qcf
           #JMS environment properties to lookup resources.
           env-properties:
              # Env properties
    outgoing:
      #Sample configurations for distributed queue.
      to-wls-dq:
        #Connector Name.
        connector: helidon-jms
        #JMS Dqueue JNDI Name.
        jndi.destination: dqcf
        type: queue
        #JMS DQueue Connection factory JNDI value.
        jndi.jms-factory: dqcf
        #Here wls-cluster-url refers to t3/t3s URL of WebLogic
Cluster.
        #JMS DQueue provider URL. It Overrides the default provider
value specified in the helidon-jms.jndi.env-properties section.
        jndi.env-properties.java.naming.provider.url: ${wls-cluster-
url}
```

Note:

The **helidon-jms** connector can also be used with the Create Destination Identifier (CDI) naming service to look up the configured JMS objects. In this case, you should use the `destination` key to refer to the JMS destination name instead of using the `jndi.destination` key.

3. After configuring the WebLogic Server JMS connector, add the Java code to send and receive messages to and from WebLogic Server.

   If you are using Helidon 3.x, you may use the following Java code example as reference:

```
package helidon.examples.quickstart.mp;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;
import org.eclipse.microprofile.reactive.messaging.Channel;
import org.eclipse.microprofile.reactive.messaging.Emitter;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Inject;

@ApplicationScoped
public class JMSQueue {
    //Inject channel to produce messages
    @Inject
    @Channel("to-wls-q")
    private Emitter<String> emitter;

    //Send Message
    public void sendMessage(String msg) {
        emitter.send(msg);
    }

    //Sample script to consume Messages
    @Incoming("from-wls-q")
    public void receive(String msg) {
        System.out.println("Process JMS message as per business
logic"+msg);
    }
}
```

If you are using Helidon 2.x, you may use the following Java code example as reference:

```
package helidon.examples.quickstart.mp;
import org.reactivestreams.FlowAdapters;
import org.reactivestreams.Publisher;
import java.util.concurrent.SubmissionPublisher;
import org.eclipse.microprofile.reactive.messaging.Incoming;
import org.eclipse.microprofile.reactive.messaging.Outgoing;
import javax.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class JMSQueue {
    SubmissionPublisher<String> emitter = new SubmissionPublisher<>();

    //Register publisher
    @Outgoing("to-wls-q")
    public Publisher<String> registerPublisher() {
        return FlowAdapters.toPublisher(emitter);
    }

    //Send Messages
    public void sendMessage(String msg) {
        emitter.submit(msg);
    }

    //Sample Script To Consume Messages
    @Incoming("from-wls-q")
    public void receive(String msg) {
        System.out.println("Process JMS message as per business
logic"+msg);
    }
}
```

4. (**This step is applicable only for Helidon 3.x.**) Add the `serial-config.properties` file in the `<src>/main/resources/META-INF/helidon/` location with the following content to address the deserialization filter issue reported with Helidon JEP-290 Implementation.

```
pattern=weblogic.**;java.util.**;java.lang.**;java.io.**;java.rmi.**;javax
.naming.**;jakarta.jms.**
```

For more information about deserialization filters, see JEP-290.

> **Note:**
>
> The suggested filter configuration helps only for selected use cases. In case of issues, you may need to use your own pattern that is suitable for your use case.

5. Start the WebLogic Server if it is not already up and running. For information about starting the WebLogic domain in Kubernetes, see Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration.

   You should have also created the JMS resources (such as Queue, Topic, Uniform Distributed Queue, and so on) and the T3/T3S channels.
6. Build the Helidon application using the following command:

   ```
   mvn clean package -DskipTests=true
   ```
7. Run the Helidon application using the following command:

   ```
   java -jar target/<Helidion-Project-Name>.jar
   ```

Troubleshooting Common JMS Issues

Learn about the common issues you may encounter when setting up the integration between WebLogic Server and Helidon 3.x or 2.x.

Issue 1

The `filter status: REJECTED` error is reported with the latest Patch Set Updated (PSU) for the WebLogic Server thin client jar file. Here is a sample of the error message:

```
<Error> <RJVM> <WL-000503> <Incoming message header or abbreviation
processing failed.
 java.io.InvalidClassException: filter status: REJECTED
        at java.base/
java.io.ObjectInputStream.filterCheck(ObjectInputStream.java:1414)
        at java.base/
java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.java:2055)
        at java.base/
java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1909)
        at java.base/
java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.java:223
5)
        at java.base/
java.io.ObjectInputStream.readObject0(ObjectInputStream.java:1744)
        at java.base/
java.io.ObjectInputStream.readObject(ObjectInputStream.java:514)
        at java.base/
java.io.ObjectInputStream.readObject(ObjectInputStream.java:472)
        at thinClientClassLoader//
weblogic.utils.io.FilteringObjectInputStream.readObjectValidated(Filter
ingObjectInputStream.java:177)
```

```
. . . .
>
023.02.04 13:57:10 INFO io.helidon.messaging.connectors.jms.JmsConnector
Thread[main,5,main]: JMS Connector gracefully stopped.
Exception in thread "main" io.helidon.messaging.MessagingException: Error
when preparing JNDI context.
        at
io.helidon.messaging.connectors.jms.ConnectionContext.<init>(ConnectionContex
t.java:64)
        at
io.helidon.messaging.connectors.jms.JmsConnector.getPublisherBuilder(JmsConne
ctor.java:429)
. . . .
```

Solution:

This issue is reported due to the deserialization filters used in Helidon. Add the `serial-config.properties` file in the `<src>/main/resources/META-INF/helidon/` location, with the following content to resolve the issue:

```
pattern=weblogic.**;java.util.**;java.lang.**;java.io.**;java.rmi.**;javax.na
ming.**;jakarta.jms.**
```

For more information about deserialization filters, see Helidon JEP-290 Implementation.

Note:

The suggested filter configuration helps only for selected use cases. In case of issues, you may need to use your own pattern that is suitable for your use case.

Issue 2

The Helidon 2.x application hangs when you create the JMS connector during server startup.

Solution:

Ensure that the `jaxws-wlswss-client` and `wlthint3client` jars are not used together. These jar files should not be used together.

Issue 3

In Helidon 3.x, class loader issues are reported with `wlthint3client` jar file. Here is a sample of the error message:

```
SEVERE: Default error handler: Unhandled exception encountered.
java.util.concurrent.ExecutionException: Unhandled 'cause' of this exception
encountered.
        at
io.helidon.webserver.RequestRouting$RoutedRequest.defaultHandler(RequestRouting.java:39
8)
. . .
Caused by: java.lang.ExceptionInInitializerError
        at weblogic.utils.LocatorUtilities.getService(LocatorUtilities.java:37)
        at
weblogic.jms.WLInitialContextFactory.getInitialContext(WLInitialContextFactory.java:124
)
. . .
Caused by: A MultiException has 2 exceptions.  They are:
1. java.lang.NoSuchMethodException: The class GlobalServiceLocator has no
constructor marked @Inject and no zero argument constructor
2. java.lang.IllegalArgumentException: Errors were discovered while reifying
SystemDescriptor(
        implementation=weblogic.server.GlobalServiceLocator
. . .
Caused by: java.lang.NoSuchMethodException: The class GlobalServiceLocator has
no constructor marked @Inject and no zero argument constructor
        at
org.jvnet.hk2.internal.Utilities.findProducerConstructor(Utilities.java:1326)
. . .
```

Solution:

The thin client that uses the `javax` namespace does not work with Helidon 3.x that uses the `jakarta` namespace. Therefore, download the jakarta thin client jar, `wlthint3client.jakarta.jar`, that uses the jakarta namespace and also handles multi-release jar files correctly. Add this jar to the local maven repository, build, and then run the application.

Issue 4

WebLogic Server JMS connector issues reported in Helidon 3.x.

Solution:

The new WebLogic Server connector initializes the `InitialContextFactory` interface within a different thread from the one which creates the destination. This feature makes the WebLogic Server's thread-based security unusable. See Understanding Thread-Based Security on Clients and Servers in *Developing JMS Applications for Oracle WebLogic Server*. The solution for resolving these issues is to switch to object-based security. See Understanding Object-Based Security in *Developing JMS Applications for Oracle WebLogic Server*.

Issue 5

Helidon serialization config filter does not trace the actual rejected classes by default. Here is a sample of the error message you will find in the logs:

```
java.io.InvalidClassException: filter status: REJECTED
```

Solution:

To find out which class has been actually rejected, set the `helidon.serialFilter.trace` system property to either `basic` or `full`.

```
java -Dhelidon.serialFilter.trace=basic -jar ./target/custom-mp.jar
```

Each accepted or rejected class is logged only once with the `basic` trace filter setting.

```
ALLOWED class: class java.util.LinkedList, arrayLength: -1, depth: 2,
references: 3, streamBytes: 84
REJECTED class: class java.util.ArrayList, arrayLength: -1, depth: 2,
references: 3, streamBytes: 90
```

You can compose proper serialization filter pattern with the list of REJECTED classes.

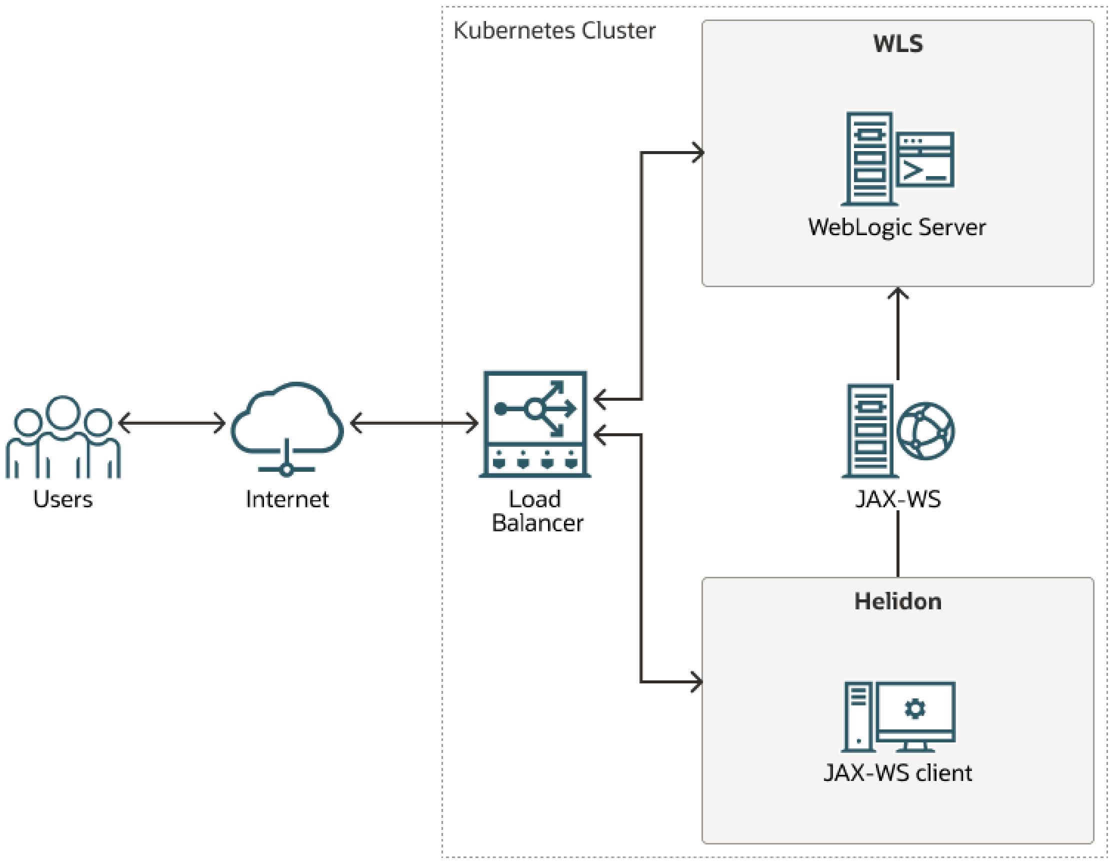
4

Integrating WebLogic Server Web Services with Helidon

The Oracle WebLogic Server (WebLogic Server) Web Services integration with Helidon enables a Helidon client to call on the WebLogic Server Web Services. This integration allows the Helidon microservices to interact with the WebLogic Server applications by using the SOAP Web Service calls from Helidon to WebLogic Server.

The following graphics illustrate the integration between WebLogic Server Web Services and Helidon:

Figure 4-1 Web Services Integration with Helidon



This chapter includes the following topics:

- Prerequisites
- Setting Up the Web Services Integration with Helidon

Prerequisites

To integrate WebLogic Server with Helidon for SOAP (Simple Object Access Protocol) Web Services, it is assumed that you have already deployed WebLogic Server and Helidon in a Kubernetes cluster. See Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration.
In addition, obtain the following jar files provided by WebLogic Server:

- **For Helidon 3.x**
  You can download the WebLogic Server 14.1.1 Web Services Jakarta client from Oracle Software Delivery Cloud (OSDC) for commercial use under WebLogic Server commercial licenses. Go to https://edelivery.oracle.com and download the package for Oracle WebLogic Server 14c 14.1.1.0.0 (Oracle WebLogic Server Enterprise Edition, Oracle WebLogic Server Standard Edition). To download the client for development use under the "Oracle Technology Network Free Developer License Terms", go to https://www.oracle.com/middleware/technologies/weblogic-server-downloads.html. For the steps to download the, see Downloading the WebLogic Server Java Clients with Jakarta Package Names.
- **For Helidon 2.x**
  You can locate the WebLogic Server 14.1.1 Web Services client in your WebLogic Server installation under the `*WL_HOME*/modules/clients` directory.

Setting Up the Web Services Integration with Helidon

WebLogic Server Web Services and Helidon integration enables the Helidon microservice application to communicate with the WebLogic Web Service deployed in WebLogic Server. Before you begin the integration steps, you should have created the JAX-WS Web Service using WebLogic Deploy Tooling (WDT) and included it as part of an auxiliary image. See Auxiliary Images. For information about developing WebLogic Server Web Services, see Examples of Developing JAX-WS Web Services in *Developing JAX-WS Web Services for Oracle WebLogic Server*.

To initiate a call from Helidon to an existing WebLogic Server Web Service:

1. Install the client jar file and include it as part of the Maven dependencies, as shown below:

   If you are using Helidon 3.x, install the `com.oracle.webservices.wls.jaxws-wlswss-client.jakarta.jar` client jar file, as shown below:

   ```
   <dependency>
       <groupId>com.oracle.webservices.wls.jaxws-wlswss-
   client.jakarta</groupId>
       <artifactId>com.oracle.webservices.wls.jaxws-wlswss-
   client.jakarta</artifactId>
       <version>1.0</version>
   </dependency>
   ```

If you are using Helidon 2.x, install the `com.oracle.webservices.wls.jaxws-wlswss-client.jar` client jar file, as shown below:

```
<dependency>
    <groupId>com.oracle.webservices.wls.jaxws-wlswss-client</groupId>
    <artifactId>com.oracle.webservices.wls.jaxws-wlswss-client</
artifactId>
    <version>1.0</version>
</dependency>
```

2. Use the `clientgen` WebLogic Server Web Services Ant task from the client jar file installed in Step 1 to generate the artifacts that client applications need. These artifacts are generated and added to the `target/generated-sources` folder.

   Add the `maven-antrun-plugin` plug-in to execute the `clientgen` Ant task during the `generate-sources` build phase, as shown in the following example:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-antrun-plugin</artifactId>
    <version>3.1.0</version>
    <executions>
        <execution>
            <id>ws-client-gen</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>run</goal>
            </goals>
            <configuration>
                <target>
                    <property name="wsdl-file">file://${basedir}/
DynamicWSImplService.wsdl</property>
                    <property name="compile_classpath"
                              refid="maven.compile.classpath"/>
                    <taskdef name="clientgen"

classname="weblogic.wsee.tools.anttasks.ClientGenTask"
                             classpath="${compile_classpath}"/>
                    <clientgen wsdl="${wsdl-file}"
                               wsdlLocation="${wsdl-file}"
                               destDir="${project.build.directory}/
generated-sources"
                               packageName="com.example.wlssoap"
                               generateRuntimeCatalog="false"
                               type="JAXWS"
                               copyWsdl="false"/>
                </target>
            </configuration>
        </execution>
    </executions>
</plugin>
```

   For more information about generating client artifacts, see Using the clientgen Ant Task To Generate Client Artifacts.

3. Use the `build-helper-maven-plugin` plug-in to add the `/target/generated-sources` directory with the generated client classes as an additional directory with sources.

```
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>build-helper-maven-plugin</artifactId>
    <version>3.3.0</version>
    <executions>
        <execution>
            <id>add-source</id>
            <phase>generate-sources</phase>
            <goals>
                <goal>add-source</goal>
            </goals>
            <configuration>
                <sources>
                    <source>${pom.basedir}/target/generated-
sources</source>
                </sources>
            </configuration>
        </execution>
    </executions>
</plugin>
```

4. Create the **`Jakarta`** based (for Helidon 3.x) or **`javax`** based (for Helidon 2.x) RESTful Web Service to invoke the WebLogic Web Service with the generated client classes, as shown in the following example.

```
@Path("/helidon-client")
@ApplicationScoped
public class HelidonWSEEClient {

    @Inject
    @ConfigProperty(name = "remote.wsdl.location")
    private String remoteWsdlLocation;

    @GET
    @Path("/getWLSWebserviceResult/subtract/{y}/from/{x}")
    @Produces(MediaType.APPLICATION_JSON)
    public JsonObject invokeWLSWebservice(@PathParam("x") int x,
                                          @PathParam("y") int y) {
        DynamicWSImplService testService = new
DynamicWSImplService();
        DynamicWSImpl testPort = testService.getDynamicWSImplPort();
        ((BindingProvider) testPort).getRequestContext()
                .put(BindingProvider.ENDPOINT_ADDRESS_PROPERTY,
remoteWsdlLocation);

        int response = testPort.subtract(x,y);

        return Json.createObjectBuilder().add("ws-response",
response).build();
```

```
    }
}
```

5. Build the Helidon client and invoke the WebLogic Server Web Service by compiling the Maven REST client using the following command:

   ```
   mvn clean package
   ```

6. Start the Helidon server using the following command:

   ```
   java -jar target/<Helidion-Project-Name>.jar
   ```

   When the Helidon server starts, the microservice gets deployed and becomes ready for use. You can access the microservice application locally by using the `http://<HELIDON_HOST>:<HELIDON_PORT>/helidon-client/getWLSWebserviceResult/subtract/5/from/10` URL.

   Where <*HELIDON_HOST*> and <*HELIDON_PORT*> refer to the host where the Helidon microservice application is running.

5

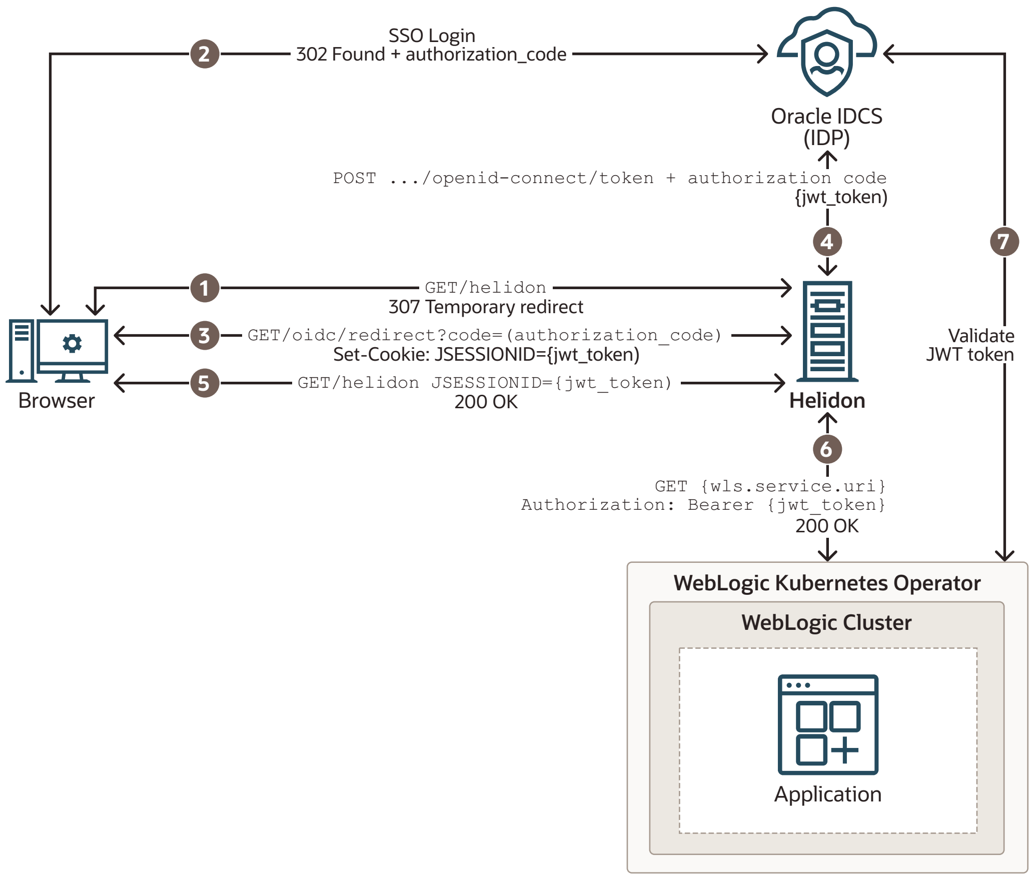
Integrating WebLogic Cluster and Helidon Applications for Single Sign-On on OCI Using IDCS

The Oracle WebLogic Server (WebLogic Server) and Helidon integration enables you to use the single sign-on (SSO) authentication mechanism for applications deployed on WebLogic Server and Helidon by using OpenID Connect (OIDC) with Oracle Identity Cloud Service (IDCS) in a Kubernetes environment. Using SSO between WebLogic Server applications and Helidon microservices simplifies security within a modernized application by sharing authentication while ensuring secure services. You can implement SSO in different ways. A common approach is to use a token-based authentication protocol such as OAuth or OIDC. The WebLogic Server and Helidon integration in a Kubernetes cluster enables SSO authentication to:

- Access the IDCS configured client application deployed in the WebLogic cluster.
- Access the Helidon REST endpoints configured with IDCS.
- Access the WebLogic Server application endpoints from the Helidon REST endpoints.

The following graphic illustrates the integration between WebLogic Server and Helidon for SSO using IDCS in an Oracle Cloud Infrastructure (OCI) environment:

Figure 5-1   WebLogic Cluster and Helidon Integration for SSO


SSO Login
302 Found + authorization_code
Oracle IDCS (IDP)
POST .../openid-connect/token + authorization code {jwt_token)
GET/helidon
307 Temporary redirect
GET/oidc/redirect?code=(authorization_code)
Set-Cookie: JSESSIONID={jwt_token)
GET/helidon JSESSIONID={jwt_token)
200 OK
Browser
Helidon
Validate JWT token
GET {wls.service.uri}
Authorization: Bearer {jwt_token}
200 OK
WebLogic Kubernetes Operator
WebLogic Cluster
Application

The description of the steps in the illustration:

1. The client uses a browser to call the protected Helidon resource without bearer token and gets redirected to the IDCS SSO login page.
2. The client is redirected back to the Helidon application with an authorization code after a successful sign-in on the login page.
3. When Helidon receives the authorization code, the JWT token is requested from IDCS, returned and saved as a JSESSIONID cookie.
4. The JWT token is requested with a new authorization code, client id, and client secret.
5. The last redirect leads back to the originally called resource, Helidon; this time with a valid bearer token in JSESSIONID.
6. The bearer token is propagated to the client call for the WebLogic Server application resource.
7. The WebLogic Server application resource is also secured with OIDC and validates the token against IDCS.

This chapter includes the following topics:

- Prerequisites
- Setting Up the WebLogic Cluster Integration with Helidon for Single Sign-On

Prerequisites

The prerequisites are based on WebLogic Server and Helidon integration in a Kubernetes cluster. The list may vary if you use any other supported platforms. See Preparing the Kubernetes Cluster for WebLogic Server and Helidon Integration.
Ensure that you have the following:

- A local machine with `kubectl` setup to access the Kubernetes cluster. For more information about this setup, see Set Up the kubeconfig File for the Cluster and Verify kubectl and Kubernetes Dashboard Access to the Cluster in the *Create a Cluster with Oracle Cloud Infrastructure Container Engine for Kubernetes* Tutorial.
- A WebLogic Kubernetes Operator (Operator) setup. See Operator Quick Start.
- An OCI load balancer with a public IP and the load balancer rules applied according to the WebLogic Server and Helidon application URL pattern. The following example shows the application of rules:

```
---
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: cquotes
  namespace: sample-domain1-ns
spec:
  routes:
  - kind: Rule
    match: PathPrefix(`/<WLS application url>)
    services:
    - kind: Service
      name: sample-domain1-cluster-cluster-1
      port: 8001
      sticky:
        cookie:
          httpOnly: true
          name: cookie
          secure: false
          sameSite: none
---
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
metadata:
  name: helidon-quickstart-mp
  namespace: default
spec:
  routes:
  - kind: Rule
    match: PathPrefix(`/<helidon rest url>`)
    services:
    - kind: Service
      name: helidon-quickstart-mp
      port: 8080
---
apiVersion: traefik.containo.us/v1alpha1
kind: IngressRoute
```

```
metadata:
  name: helidon-oidc
  namespace: default
spec:
  routes:
  - kind: Rule
    match: PathPrefix(`/oidc`)
    services:
    - kind: Service
      name: helidon-quickstart-mp
      port: 8080
```

- Access to IDCS with privileges to register applications. For instructions to create a user, see Create User Accounts in *Administering Oracle Identity Cloud Service*.
- Supported JDK (for example JDK 8 or later) and Maven (if required) to build the WebLogic cluster client applications.
- For deploying Helidon, see:
  - Prerequisites for 3.x
  - Prerequisites for 2.x
- A basic understanding of OpenID Connect (OIDC). See OpenID.

Setting Up the WebLogic Cluster Integration with Helidon for Single Sign-On

To facilitate the integration between WebLogic cluster and Helidon, WebLogic Server is deployed using the WebLogic Server Kubernetes Operator (Operator), in a Kubernetes cluster. The Helidon container is also deployed within the same Kubernetes cluster. The default OCI load balancer with a public IP is used to access the WebLogic Server Administration Console, the client applications (deployed in the WebLogic cluster), and the Helidon REST endpoints (exposed by the Helidon microservice application). The Helidon microservice application communicates with the WebLogic cluster applications through the REST endpoints that are integrated with Oracle Identity Cloud Service for authentication and authorization using SSO.

To set up the integration:

1. Integrate the WebLogic cluster applications with Oracle Identity Cloud Service.
   a. Register the application on Oracle Identity Cloud Service.

      The following client configuration details are important because you will use the values of these parameters in the source code of the client application to enable the Oracle Identity Cloud Service to communicate securely with the client application.

      - **Allowed Grant Types**: The grant type that the client application is allowed to use when it requests validation from Oracle Identity Cloud Service.
      - **Redirect URL**: The absolute URL of the client application where a user will be redirected to after successful authentication in Oracle Identity Cloud Service.
      - **Logout URL**: The URL that is called by Oracle Identity Cloud Service after a user logs out of the client application.

- **Post Logout Redirect URL**: The URL where a user will be redirected to after logging out of Oracle Identity Cloud Service.

After you finish registering and activating the client application, make a note of the **Client ID** and **Client Secret**. The client application will use the Client ID and Client Secret (similar to a login credential such as user ID and password) to communicate with Oracle Identity Cloud Service.

For more information about registering an application, see Add a Confidential Application in *Administering Oracle Identity Cloud Service*.

b. Integrate the client application with Oracle Identity Cloud Service.

In this step, you have to configure the client application to connect with Oracle Identity Cloud Service during authentication. Update the CLIENT_ID, CLIENT_SECRET (you will use the **Client ID** and **Client Secret** generated at the time of registering the client application in IDCS), and the IDCS_URL in the client application source code, as shown in the following example:

```
//YOUR IDENTITY DOMAIN AND APPLICATION CREDENTIALS
public static final String CLIENT_ID = "<your client id>";
public static final String CLIENT_SECRET = "<your client secret>";
public static final String IDCS_URL = "https://
example.identity.oraclecloud.com";
```

Note:

The **Client ID** and **Client Secret** are used to obtain the access token for SSO authentication.

c. Build and deploy the client application in the WebLogic cluster.

Build the client application using the WebLogic Server supported JDK, and then deploy the application in the WebLogic cluster. After a successful deployment, the client application will be displayed with an `Active` status in the Deployments section of the WebLogic Server Administration Console.

d. Verify the WebLogic Server application for SSO.

Access the client application URL `http://{OCI LB_IP}/<wls app>` using a browser. Here, *OCI LB_IP* is the public IP of the load balancer and *wls app* is the name of the WebLogic cluster application. The SSO endpoints should redirect to the IDCS login page for authentication.

After successful authentication, you should be able to view the application contents.

For an example of integrating an application with Oracle Identity Cloud Service, see Integrating Customer Quotes and Oracle Identity Cloud Service.

2. Integrate the Helidon application with Oracle Identity Cloud Service.

a. Register the Helidon MP client application on Oracle Identity Cloud Service.

The following parameters are important:

- **Redirect URL**: `<LB IP>/oidc/redirect`
- **Logout URL**: `<LB IP>/oidc/logout`
- **Post Logout Redirect URL**: `<Logout URL>`

- **Primary Audience**: `<LB IP>/` as per the Helidon application REST endpoints
- **Secondary Audience**: `<IDCS URI>` as per the IDCS tenancy

After you finish registering and activating the Helidon application, make a note of the **Client ID** and **Client Secret**.

b. Set up the Helidon application.

Create the Helidon MP sample application using Maven , as shown in the following example:

```
mvn -U archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-mp \
    -DarchetypeVersion=3.2.0 \
    -DgroupId=io.helidon.examples \
    -DartifactId=helidon-quickstart-mp \
    -Dpackage=io.helidon.examples.quickstart.mp
```

See *Set up Helidon* in the *Helidon MP OIDC Security Provider* Guide.

c. Configuring the OIDC dependencies for the Helidon application.

The project will be built and run from the `helidon-quickstart-mp` directory. Navigate to the directory:

```
cd helidon-quickstart-mp
```

i. Add the Maven dependency to the `pom.xml` file, as shown in the following example:

```
<dependency>
  <groupId>io.helidon.microprofile</groupId>
   <artifactId>helidon-microprofile-security</artifactId>
</dependency>
<dependency>
  <groupId>io.helidon.microprofile</groupId>
  <artifactId>helidon-microprofile-oidc</artifactId>
</dependency>
<dependency>
  <groupId>io.helidon.microprofile.jwt</groupId>
  <artifactId>helidon-microprofile-jwt-auth</artifactId>
</dependency>
```

ii. Update the `application.yaml` file according to your requirements and the IDCS configuration, as shown in the following example:

```
# These values should be as per IDCS configured application
values
security:
  config.require-encryption: false
  properties:
     # Oracle IDCS instance uri. Following URI may change
depending on IDCS instance.
```

```
     idcs-uri: "${idcs-uri}"
     # IDCS Registered application client-id and secret
     # Register Confidential application for Helidon at IDCS and
provide the client id and secret below.
     idcs-client-id: "${helidon-idcs-client-id}"
     idcs-client-secret: "${helidon-idcs-client-secret}"
     # Configure proxy if required.
     proxy-host: ""
     # Helidon application listening at port
     port: 8080
  providers:
    - abac:
    - oidc:
        client-id: "${security.properties.idcs-client-id}"
        # See [EncryptionFilter](https://helidon.io/docs/latest/apidocs/
io.helidon.config.encryption/io/helidon/config/encryption/EncryptionFilter.html for
details about encrypting passwords in configuration files.
        client-secret: "${security.properties.idcs-client-secret}"
        identity-uri: "${security.properties.idcs-uri}"
        # This redirect URI which should match at IDCS registered
application Redirect URL
        # Redirect URL at IDCS follows http://<hostname:8080 or
Load Balancer>/oidc/redirect
        redirect-uri: "/oidc/redirect"
        # scope-audience should match with IDCS Primary Audience ,
except adding "/" trailing character.
        # At IDCS it will be http://<hostname:8080 or Load
Balancer>/<REST endpoint>/.
        # Mismatch in scope-audience causes failure in generating
access token
        scope-audience: "http://${load-balancer-ip}/<helidon REST
endpoint>"
        # Mismatch in audience  causes failure in generating
access token
        audience: "${IDCS_URI}"
        # Front end host , it should be either hostname:8080 or
load balancer ip
        frontend-uri: "http://${load-balancer-ip}"
        server-type: "idcs"
        logout-enabled: true
        # Configured IDCS Logout URL "http://<LB|HOSTNAME:PORT>/
oidc/logout
        # Configured IDCS Post Logout Redirect URL "http://<LB|
HOSTNAME:PORT>/loggedout"
        # Post logout Helidon REST endpoint or URI
        post-logout-uri: "/${logout url}"
        propagate: true
        outbound:
           - name: "propagate-token"
             hosts: [ "${load-balancer-ip}" ]
        redirect: true
        cookie-use: true
        header-use: true
```

iii. Configure the Helidon REST endpoints for SSO.

You can SSO secure the Helidon REST endpoints by adding the `@Authenticated` annotation. See Section "Usage" in Adding Security.

The `@Authenticated` annotation is used to specify server resources with enforced authentication. The following is an example of using this annotation:

```
@Authenticated
@GET
@Produces(MediaType.APPLICATION_JSON)
  public JsonObject getDefaultMessage() {
      return createResponse("World");
  }
```

d. Deploy the Helidon MP application in the same Kubernetes cluster.

   i. Build the Docker image using the following command:

   ```
   docker build -t helidon-quickstart-mp .
   ```

   ii. Update the `app.yaml` file for image reference, based on the Helidon docker image created and hosted at the container registry:

   ```
   kind: Service
   apiVersion: v1
   metadata:
     name: helidon-quickstart-mp
     labels:
       app: helidon-quickstart-mp
   spec:
     type: NodePort
     selector:
       app: helidon-quickstart-mp
     ports:
     - port: 8080
       targetPort: 8080
       name: http
   ---
   kind: Deployment
   apiVersion: apps/v1
   metadata:
     name: helidon-quickstart-mp
   spec:
     replicas: 1
     selector:
       matchLabels:
         app: helidon-quickstart-mp
     template:
       metadata:
         labels:
           app: helidon-quickstart-mp
           version: v1
       spec:
         containers:
   ```

```
      - name: helidon-quickstart-mp
        image: ${docker image repo}/helidon-quickstart-mp
        imagePullPolicy: Always
        ports:
        - containerPort: 8080
```

iii. Deploy the Helidon application in the same Kubernetes cluster.
Run the following `kubectl` command on a local machine to deploy the Helidon application in the Kubernetes cluster:

```
kubectl create -f app.yaml
```

iv. Verify whether the Helidon deployment is successful by running the following `kubectl` commands:

```
kubectl get pods
```

```
kubectl get service helidon-quickstart-mp
```

For more information about the `quickstart`, see Helidon MP Quickstart.

e. Verify the Helidon application for SSO.

Access REST endpoint URL `http://{LB IP}/<SSO configured REST endpoint>` using a browser. Here, *LB_IP* is the public IP of the load balancer and *SSO configured REST endpoint* is the Helidon REST endpoint.

Upon successful SSO authentication at IDCS, you will receive a response from the REST endpoint.

3. Integrate the Helidon application and the WebLogic cluster applications for SSO.

a. Access the WebLogic cluster application SSO endpoints from the Helidon SSO REST endpoints. Helidon obtains the JWT token after the Helidon SSO endpoints are authenticated. You can use the token manually for calling other services or have the token propagated automatically with the JAX-RS client. You may use the following example for reference:

```
@Path("/helidon")
@ApplicationScoped
@Authenticated
public class HelidonResource {

    @Inject
    @ConfigProperty(name = "wls.service.url")
    private URI wlsServiceUri;

    @Inject
    private JsonWebToken jwt;

    @Authenticated
    @GET
    @RolesAllowed({"secret_role"})
    @Produces(MediaType.APPLICATION_JSON)
    public JsonObject getDefaultMessage(@Context SecurityContext
secCtx) {
        var user = secCtx.userName();
        var isInRole = secCtx.isUserInRole("secret_role");

        // Manually access the raw bearer token
        var bearerToken = jwt.getRawToken();

        // Bearer token is propagated automatically with the
OIDC outbound propagation,
        // no manual action is needed with the JAX-RS client
        JsonObject response = ClientBuilder.newClient()
                .target(wlsServiceUri)
                .request()
                .buildGet()
                .invoke(JsonObject.class);

        return Json.createObjectBuilder()
                .add("user", user)
                .add("is_secret_role", isInRole)
                .add("wls-response", response)
                .build();
    }
}
```

b. Access the Helidon SSO REST endpoint to verify the integration between Helidon application and the WebLogic cluster application.

   After a successful SSO authentication at the Helidon REST endpoint, you will be able to get access to the WebLogic cluster application.