Oracle® Communications Unified Inventory Management

Cloud Native Deployment Guide

Release 8.0.1
G50252-01
April 2026

Oracle Communications Unified Inventory Management Cloud Native Deployment Guide, Release 8.0.1

G50252-01

Contents

7 Extending the WebLogic Server Deploy Tooling (WDT) Model

8 Exploring Alternate Configuration Options

9 Integrating UIM

10 Running the SAF Sample for UIM Cloud Native

11 Upgrading the UIM Cloud Native Environment

12 Moving to UIM Cloud Native from a Traditional Deployment

About This Content

This document describes how to install and administer Oracle Communications Unified Inventory Management (UIM) cloud native deployment.

Audience

This document is for system administrators, database administrators, and developers who install and configure UIM. The person installing the software should be familiar with the following topics:

- Operating system commands
- Database configuration
- Oracle WebLogic Server
- Network management

Before reading this guide, you should have familiarity with UIM. See *UIM Concepts*.

UIM requires Oracle Database and Oracle WebLogic Server. See the documentation for these products for installation and configuration instructions.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

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

Overview of the UIM Cloud Native Deployment

Get an overview of Oracle Communications Unified Inventory Management (UIM) cloud native deployment, architecture, and the UIM cloud native package.

This chapter provides an overview of Oracle Communications Unified Inventory Management (UIM) deployed in a cloud native environment using container images and a Kubernetes cluster.

About the UIM Cloud Native Deployment

You can deploy UIM in a Kubernetes-based shared cloud (cluster) while implementing modern DevOps “Configuration as Code” principles to manage system configuration in a consistent manner. You can automate system lifecycle management. You set up your own cloud native environment and can then use the UIM cloud native toolkit to automate the deployment of UIM instances. By leveraging the pre-configured Helm charts, you can deploy UIM instances quickly ensuring your services are up and running in far less time than a traditional deployment.

UIM cloud native supports the following deployment models:

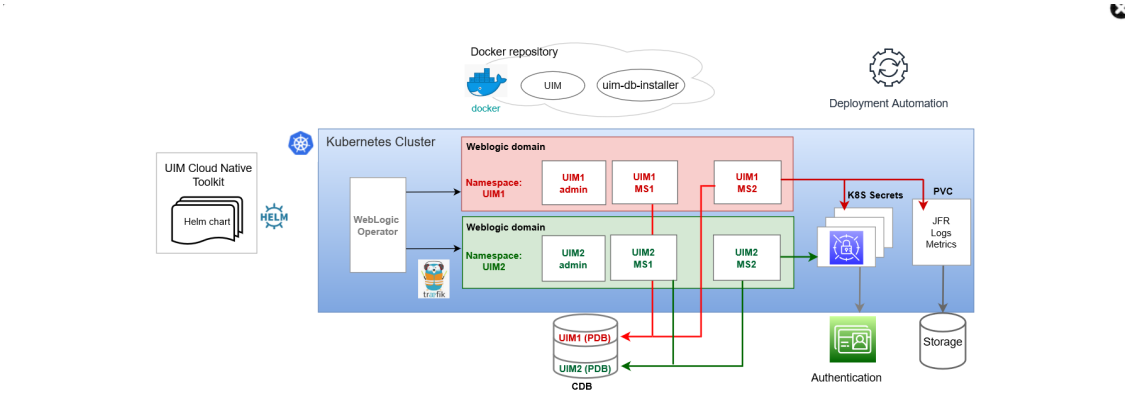
- **On Private Kubernetes Cluster**: UIM cloud native is certified for a general deployment of Kubernetes.
- **On Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE)**: UIM cloud native is certified to run on Oracle's hosted Kubernetes OKE service.

UIM Cloud Native Architecture

This section describes and illustrates the UIM cloud native architecture and the deployment environment.

The following diagram illustrates the UIM cloud native architecture.

Figure 1-1 UIM Cloud Native Architecture

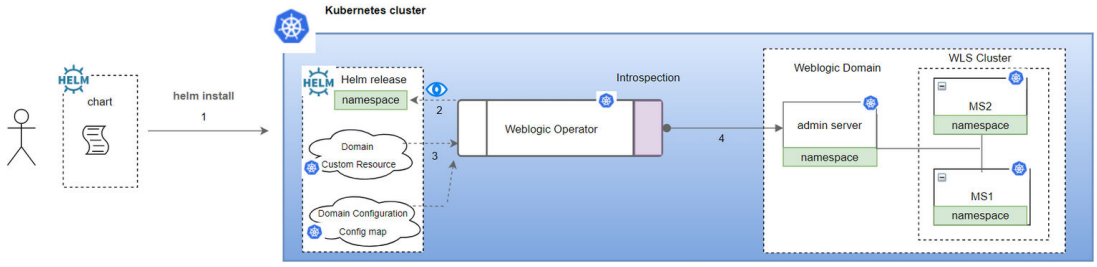


The UIM cloud native architecture requires components such as the Kubernetes cluster and WebLogic Kubernetes Operator, which are under your control to install and configure. A single WebLogic Operator can manage multiple UIM domains in multiple namespaces. Each domain is a dynamic cluster with multiple managed servers that is configured for integration with both optional and required components.

About the WebLogic Domain

The following diagram illustrates the UIM cloud native deployment environment and important concepts about producing a WebLogic domain that is capable of supporting UIM cloud native.

Figure 1-2 UIM Cloud Native Deployment Environment



In the deployment environment, the Helm chart that is provided with the UIM cloud native toolkit is deployed into the Kubernetes cluster producing two Kubernetes resources. These resources are then consumed by the WebLogic Kubernetes Operator (WKO).

About Kubernetes Custom Resource Definitions (CRD) and Domain Configuration Config Map

The Kubernetes API provides extensions called custom resources. To understand more about a Custom Resource Definition (CRD) and why it might be used, see the Kubernetes

CustomResourceDefinition (CRD) documentation at: https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/

To configure the operator for your WebLogic domain, you set up and configure your own domain resource. The domain resource does not replace the traditional configuration of the WebLogic domains found in the domain configuration files, but instead co-operates with those files to describe the Kubernetes artifacts of the corresponding domain. Refer to the Oracle WebLogic Kubernetes Operator User Guide to understand how to use a CRD to describe a WebLogic domain resource.

While the domain resource describes much of the operational details for a domain such as domain identification, secrets, pod creation, server instances, startup and shutdown, security, logging, clusters, admin and managed servers, and JVM options, the details about the more traditional configuration (deployed applications, JMS Queues, data sources and so on) are provided in a configuration map and are described using a metadata model specified by the WebLogic Deploy Tooling (WDT). The UIM cloud native toolkit provides the base configuration to produce these resources.

About Oracle WebLogic Server Deploy Tooling (WDT)

The WebLogic Server Deploy Tooling (WDT) has the following main purposes:

- It provides a metadata model that describes a WebLogic Server domain configuration.
- It provides scripts that perform domain lifecycle operations, simplifying the definition and the creation of domains. This capability provides an alternative to programmatic ways of defining domain configuration such as WebLogic Scripting Tool (WLST) or Java Mbeans manipulation.

The UIM cloud native toolkit leverages the WDT metadata model only. It does not use the scripting capabilities directly.

The toolkit provides the WDT metadata for a domain that is capable of supporting UIM. The toolkit enables you to easily override much of the base configuration through the use of Helm charts. Additionally, the toolkit framework allows you to add supplementary WDT metadata fragments to the domain. WDT provides tools that help with this task by inspecting an existing domain to produce the WDT metadata required for the configuration.

For more details about WDT, see the Oracle WebLogic Server Deploy Tooling documentation on GitHub at: https://github.com/oracle/weblogic-deploy-tooling

About UIM Configuration and Specification Layers

The UIM configuration defines the deployment footprint, layout, and tuning. Managing this configuration as a single monolithic unit is not optimal for sustainability or scalability. To address this, a layered configuration approach is adopted, enabling better modularity, maintainability, and risk management.

The following layers are defined, which include a set of values that are specific to the function of that layer:

- **Base**: This foundational layer is shared across all applications deployed using the common cloud native toolkit. It includes settings such as the ingress controller, SSL, authentication mechanisms, storage volumes, garbage collection (GC), log configuration, and so on.
- **UIM-Specific**: This layer includes parameters that are specific to a UIM instance such as external authentication providers, custom templates, SAF, JMS queues database identities, and cluster size.

- **Shape**: The shape layer defines the hardware resource utilization and the resulting tuning. Java Heap Size is an example of a configuration value found in the shape specification.
- **Database**: The database layer includes the parameters that are specific to schema operations (for example: create, upgrade, delete, purge schema), such as the DB installer image, table space info, and so on. The UIM instance do not read the parameters from this layer.

The layers are implemented as specification files written in YAML:

- **application-base**
- **app-uim**
- **<*shape*>/uim**
- **database**

Each UIM deployment requires a **project** and an **instance** name:

- **Project** refers to the Kubernetes namespace where UIM is deployed.
- **Instance** is a logical identifier used to differentiate multiple UIM deployments within the same namespace.

You can deploy multiple UIM instances within a single project. However, if you intend to use UIM services alongside the core UIM instance, Oracle recommends you to maintain a single UIM instance and add all services within one namespace. If you need more UIM instances, deploy them in separate namespaces.

Each of the above configuration layers: base, UIM-specific, database and shape, should be unique to the project and instance.

About Helm Overrides

The specification files are consumed in a hierarchical fashion. If a value is found in multiple specification files (layers), the one further up the hierarchy takes precedence. This allows the application specification to have the final control over its configuration by being able to override a value that is prescribed in either the shape or base specifications. This allows Oracle to define sealed, base configuration, while still providing you the control over the values used for the UIM instance.

Following are the specification files, listed in the order of the highest priority to the lowest:

- app-uim.yaml
- applications-base.yaml
- <*shape*>/uim.yaml
- <*values*>.yaml

While the specification for an **app-uim** points to the specification for the shape to be used (implying the order here may be out of sequence), the values found in the specification for the shape are loaded for processing before the values in the specification for the application.

The **app-uim** specification remains the final authority on any values that are found in multiple specification files.

About the Common Cloud Native Toolkit

The Common cloud native toolkit is an archive file that includes the default configuration files, utility scripts, and samples to deploy UIM in a cloud native environment. With UIM cloud native,

managing the domain Configuration as Code (CaC) is paramount. UIM cloud native provides guidance on effective management of this configuration to ensure that instances can be created in a standardized and repeatable fashion.

Contents of the Common Cloud Native Toolkit

- Helm charts for UIM and UIM database installer:
  - The Helm chart for UIM is located in **$COMMON_CNTK/charts/uim-app**.
  - The Helm chart for the UIM DB Installer is located in **$COMMON_CNTK/charts/uim-dbinstaller-app**.
- Mechanism to extend the domain and WDT samples and scripts for some common use cases
- Utility scripts to help with the lifecycle of WebLogic Kubernetes Operator
- Sample scripts to manage pre-requisite secrets. These are not pipeline-friendly.
- Scripts to manage the lifecycle of a UIM instance. These are pipeline friendly.

2

Planning and Validating Your Cloud Environment

In preparation for Oracle Communications Unified Inventory Management (UIM) cloud native deployment, you must set up and validate pre-requisite software. This chapter provides information about planning, setting up, and validating the environment for UIM cloud native deployment.

See the following topics:

- Required Components for UIM Cloud Native
- Planning Your Cloud Native Environment
- Planning Your Container Engine for Kubernetes (OKE) Cloud Environment
- Validating Your Cloud Environment

If you are already familiar with traditional UIM, for important information on the differences introduced by UIM cloud native, see "Differences Between UIM Cloud Native and UIM Traditional Deployments".

Required Components for UIM Cloud Native

In order to run, manage, and monitor the UIM cloud native deployment, the following components and capabilities are required. These must be configured in the cloud environment:

- Kubernetes Cluster
- Oracle Multitenant Container Database (CDB)
- Container Image Management
- Helm
- Oracle WebLogic Server Kubernetes Operator
- Load Balancer
- Domain Name System (DNS)
- Persistent Volumes
- Authentication
- Secrets Management
- Kubernetes Monitoring Toolchain
- Application Logs and Metrics Toolchain

For details about the required versions of these components, see "UIM Software Compatibility" in *UIM Compatibility Matrix*.

In order to utilize the full flexibility, reliability and value of the deployment, the following aspects must also be set up:

- Continuous Integration (CI) pipelines for custom images and cartridges

- Continuous Delivery (CD) pipelines for creating, scaling, updating, and deleting instances of the cloud native deployment

Planning Your Cloud Native Environment

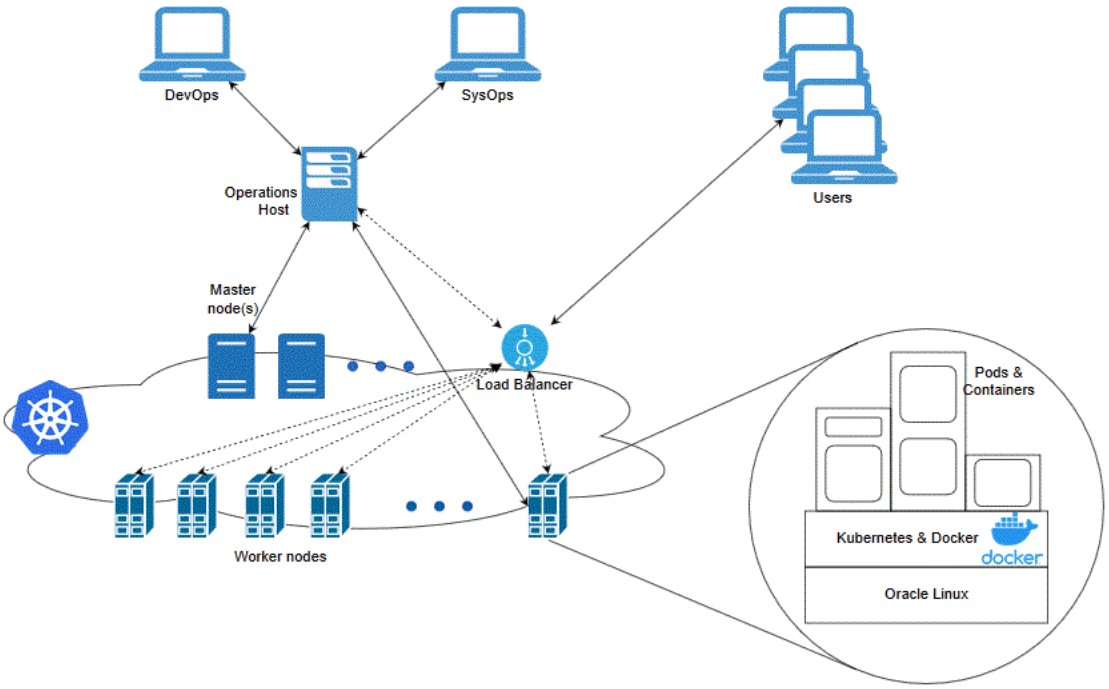
This section provides information about planning and setting up UIM cloud native environment. As part of preparing your environment for UIM cloud native, you choose, install, and set up various components and services in ways that are best suited for your cloud native environment. The following sections provide information about each of those required components and services, the available options that you can choose from, and the way you must set them up for your UIM cloud native environment.

For more information on Planning UIM Cloud Native environment, see "Planning UIM Installation" and "Planning UIM Cloud Native Upgrade"

Setting Up Your Kubernetes Cluster

For UIM cloud native, Kubernetes worker nodes must be capable of running Linux 8.x pods with software compiled for Intel 64-bit cores. A reliable cluster must have multiple worker nodes spread over separate physical infrastructure and a very reliable cluster must have multiple Master nodes spread over separate physical infrastructure.

The following diagram illustrates Kubernetes cluster and the components that it interacts with.



UIM cloud native requires:

- Kubernetes
  To check the version, run the following command:

  ```
  kubectl version
  ```

- Docker (for Linux version 7.x)

To check the version, run the following command:

```
docker version
```

- Podman (for Linux version 8.x)
  To check the version, run the following command:

  ```
  podman version
  ```

- Flannel
  To check the version, run the following command on the Master node running the kube-flannel pod:

  ```
  docker images | grep flannel
  kubectl get pods --all-namespaces | grep flannel
  ```

Typically, Kubernetes nodes are not used directly to run or monitor Kubernetes workloads. You must reserve worker node resources for processing Kubernetes workload. However, multiple users (manual and automated) of the cluster require a point from which to access the cluster and operate on it. This can be achieved by using kubectl commands (either directly on command line and shell scripts or through Helm) or Kubernetes APIs. For this purpose, set aside a separate host or set of hosts. Operational and administrative access to the Kubernetes cluster can be restricted to these hosts and specific users can be given named accounts on these hosts to reduce cluster exposure and promote traceability of actions.

Typically, the Continuous Delivery pipeline automation deploys directly on a set of such operations hosts (as in the case of Jenkins) or leverage runners deployed on such operations hosts (as in the case of GitLab CI). These hosts must run Linux, with all interactive-use packages installed to support tools such as Bash, Wget, cURL, Hostname, Sed, AWK, cut, and grep. An example of this is the Oracle Linux 8.x image on Oracle Cloud Infrastructure.

In addition, you need the appropriate tools to connect to your overall environment, including the Kubernetes cluster. For instance, for a Container Engine for Kubernetes (OKE) cluster, you must install and configure the Oracle Cloud Infrastructure Command Line Interface.

Additional integrations may need to include LDAP for users to be able to login to this host, appropriate NFS mounts for home directories, security lists and firewall configuration for access to overall environment, and so on.

Synchronizing Time Across Servers

It is important that you synchronize the date and time across all machines that are involved in testing, including client test drivers and Kubernetes worker nodes. Oracle recommends that you do this using Network Time Protocol (NTP), rather than manual synchronization, and strongly recommends it for Production environments. Synchronization is important in inter-component communications and in capturing accurate run-time statistics.

Provisioning Oracle Multitenant Container Database (CDB)

UIM cloud native architecture is best supported by the multitenant architecture that enables an Oracle database to function as a multitenant container database (CDB). A container database is either a Pluggable Database (PDB) or the root container. The root container is a collection of schemas, schema objects, and non-schema objects to which all PDBs belong. A PDB container for UIM cloud native contains the UIM schema and RCU schema. Each instance of UIM has its own PDB. UIM cloud native requires access to PDBs in an Oracle 19*c* Multitenant

database. For more information about the benefits of Oracle Multitenant Architecture for database consolidation, see *Oracle Database Concepts* for more information.

You can provision a CDB in an on-premise installation by following the instructions in *Oracle Database Installation Guide for Linux* for more information. Alternatively, you can set it up as an Oracle Cloud Infrastructure DB system. For details on the supported versions, see "UIM Software Compatibility" in *UIM Compatibility Matrix*. The provisioning process can vary based on the needs and the setup of your organization.

Provisioning an Empty PDB

To create an empty PDB:

1. Run the following SQL commands using the sys dba account for the CDB:

```
CREATE PLUGGABLE DATABASE <PDB_NAME> ADMIN USER <ADMIN_USER> IDENTIFIED BY
"<ADMIN_PASSWORD>" DEFAULT TABLESPACE "<TABLESPACE_NAME>" DATAFILE '+DATA'
SIZE 5M REUSE
AUTOEXTEND ON;
```

2. Log into the PDB as the sys dba account for the PDB (defined by the "_replace_this_text_with_admin_name_" parameter in the above commands) and adjust the PDB tablespace by running the following command:

> **Note**
>
> In the command, replace DATA with the proper name from v$asm_diskgroup.

```
create tablespace <TABLESPACE_NAME> datafile '+DATA' size 1024m reuse autoextend on
next 64m;
ALTER PLUGGABLE DATABASE DEFAULT TABLESPACE <TABLESPACE_NAME>;
```

About Container Image Management

A UIM cloud native deployment generates container images for UIM and UIM database installer. Additionally, images are downloaded for WebLogic Kubernetes Operator and HAProxy (depending on the choice of Ingress controllers).

Oracle highly recommends that you create a private container repository and ensure that all nodes have access to that repository. Images are saved in this repository and all nodes would then have access to the repository. This may require networking changes (such as routes and proxy) and include authentication for logging in to the repository.

Failing to ensure that all nodes have access to a centralized repository will mean that images have to be synced to the hosts manually or through custom mechanisms (for example, using scripts), which are error-prone operations as worker nodes are commissioned, decommissioned or even rebooted. When an image on a particular worker node is not available, then the pods using that image are either not scheduled to that node, wasting resources, or fail on that node. If image names and tags are kept constant (such as myapp:latest), the pod may pick up a pre-existing image of the same name and tag, leading to unexpected and hard to debug behaviors.

Installing Helm

UIM cloud native requires Helm, which delivers reliability, productivity, consistency, and ease of use.

In a UIM cloud native environment, using Helm enables you to achieve the following:

- You can apply custom domain configuration by using a single and consistent mechanism, which leads to an increase in productivity. You no longer need to apply configuration changes through multiple interfaces such as WebLogic Console, WLST, and WebLogic Server MBeans.
- Changing the UIM domain configuration in the traditional installations is a manual and multi-step process which may lead to errors. This can be eliminated with Helm because of the following features:
  - Helm Lint allows pre-validation of syntax issues before changes are applied
  - Multiple changes can be pushed to the running instance with a single upgrade command
  - Configuration changes may map to updates across multiple Kubernetes resources (such as domain resources, config maps and so on). With Helm, you merely update the Helm release and its responsibility to determine which Kubernetes resources are affected.
- Including configuration in Helm charts allows the content to be managed as code, through source control, which is a fundamental principle of modern DevOps practices.

In order to co-exist with older Helm versions in production environments, UIM requires Helm 3.3.4 or later saved as **helm** in PATH.

The following text shows sample commands for installing and validating Helm:

```
$ cd some-tmp-dir
$ wget https://get.helm.sh/helm-v3.12.0-linux-amd64.tar.gz
$ tar -zxvf helm-v3.12.0-linux-amd64.tar.gz

# Find the helm binary in the unpacked directory and move it to its desired
destination. You need root user.
$ sudo mv linux-amd64/helm /usr/local/bin/helm

# verify Helm version
$ helm version
version.BuildInfo{Version:"v3.12.0",
GitCommit:"472c5736ab01133de504a826bd9ee12cbe4e7904", GitTreeState:"clean",
GoVersion:"go1.18.10"}
```

For more information on helm version, see "UIM Cloud Native Deployment Software Compatibility" in *UIM Compatibility Matrix.*

Helm leverages **kubeconfig** for users running the `helm` command to access the Kubernetes cluster. By default, this is $HOME/.**kube/config**. Helm inherits the permissions set up for this access into the cluster. You must ensure that if RBAC is configured, then sufficient cluster permissions are granted to users running Helm.

About Load Balancing and Ingress Controller

Each UIM cloud native instance is a WebLogic cluster running in Kubernetes. To access application endpoints, you must enable HTTP/S connectivity to the cluster through an appropriate mechanism. This mechanism must be able to route traffic to the appropriate UIM cloud native instance in the Kubernetes cluster (as there can be many) and must be able to distribute traffic to the multiple Managed Server pods within a given instance. Each instance must be insulated from the traffic of the other instance. Distribution within an instance must allow for session stickiness so that UIM client UIs bind to a managed server wherever possible and therefore not require arbitrary re-authentication by the user. In the case of HTTPS, the load balance mechanism must enable TLS and handle it appropriately.

For UIM cloud native, an ingress controller is required to expose appropriate services from the UIM cluster and direct traffic appropriately to the cluster members. An external load balancer is an optional add-on.

The ingress controller monitors the ingress objects created by the UIM cloud native deployment, and acts on the configuration embedded in these objects to expose UIM HTTP and HTTPS services to the external network. This is achieved using NodePort services exposed by the ingress controller.

The ingress controller must support:

- Sticky routing (based on standard session cookie).
- Load balancing across the UIM managed servers (back-end servers).
- SSL termination and injecting headers into incoming traffic.

Examples of such ingress controllers include HAProxy, Voyager, and Traefik. The Common cloud native toolkit provides samples and documentation that use HAProxy as the ingress controller.

An external load balancer serves to provide a highly reliable singe-point access into the services exposed by the Kubernetes cluster. In this case, this would be the NodePort services exposed by the ingress controller on behalf of the UIM cloud native instance. Using a load balancer removes the need to expose Kubernetes node IPs to the larger user base, and insulates the users from changes (in terms of nodes appearing or being decommissioned) to the Kubernetes cluster. It also serves to enforce access policies. The Common cloud native toolkit includes samples and documentation that show integration with Oracle Cloud Infrastructure LBaaS when Oracle OKE is used as the Kubernetes environment.

Using Generic Ingress Controller

UIM cloud native supports annotation-based generic ingress creation. Which means, the use of the standard Kubernetes Ingress API (in contrast with a proprietary ingress Custom Resource Definition) that is verified by Kubernetes Conformance tests. The advantage of using a generic ingress is that it works for any Kubernetes certified ingress controller, provided that the ingress controller offers annotations (which are usually proprietary to the ingress controller) required for UIM.

Annotations applied to an ingress resource allow you to use advanced features such as connection timeout, URL rewrite, retry, additional headers, redirects, sticky cookie services, and so on and to fine-tune the functionality of that ingress resource. Different ingress controllers support different annotations. For more information on various ingress controllers, see Kubernetes documentation at: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/. Refer to this document to understand the annotations supported for your ingress controller.

The samples provided include Ingress HAProxy as Generic Ingress controller. If you have selected any other ingress controller, perform the corresponding steps for that ingress controller.

To install HAProxy ingress controller:

1. Add helm repo as follows:

   ```
   helm repo add haproxytech https://haproxytech.github.io/helm-charts
   ```

2. Create namespace for HAProxy:

   ```
   kubectl create namespace haproxy
   ```

3. Configure the **$COMMON_CNTK/samples/charts/haproxy/values.yaml** file as follows:

   a. Configure nodeports for HAProxy service:

      ```
      controller.service.nodePorts.http: 30505
      controller.service.nodePorts.https: 30543
      ```

   b. In case of multiple HAProxy controllers, or if you want to override default ingress class name, configure the as follows:

      ```
      controller.ingressClassResource.name: "haproxy"
      controller.ingressClass.name: "haproxy"
      ```

   c. You can override the default image name:

      ```
      controller.existingImagePullSecret: <image-pull-secret>
      controller.image.repository: <haproxy image name>
      controller.image.tag: <haproxy image tag>
      controller.image.pullPolicy: IfNotPresent
      ```

4. Install HAProxy:

   ```
   helm install haproxy-kubernetes-ingress haproxytech/kubernetes-ingress \
     --create-namespace \
     --namespace haproxy --values values.yaml
   ```

Any ingress controller that conforms to the standard Kubernetes ingress API and supports annotations needed by UIM should work. However, Oracle does not certify individual ingress controllers to confirm the **generic** compatibility.

For more information about ingress HAProxy controller, see https://github.com/haproxytech/kubernetes-ingress/blob/master/README.md

> **Note**
>
> - By default, Ingress HAProxy opens "**30505**" and "**30543**" ports on all nodes of cluster. Make sure these ports are not used by any other processes.
> - Use the same **IngressClassName** specified during the ingress controller installation when creating the UIM ingress. By default, the `className` is `haproxy`. If you have modified this value, ensure that you specify the same value in the **applications-base.yaml** file when creating the UIM ingress.

Using Domain Name System (DNS)

A Kubernetes cluster can have many routable entrypoints. Common choices are:

- External load balancer (IP and port)
- Ingress controller service (Master node IPs and ingress port)
- Ingress controller service (worker node IPs and ingress port)

You must identify the proper entrypoint for your Kubernetes cluster.

UIM cloud native requires hostnames to be mapped to routable entrypoints into the Kubernetes cluster. Regardless of the actual entrypoints (external load balancer, Kubernetes Master node, or worker nodes), users who need to communicate with the UIM cloud native instances require name resolution.

The access hostnames take the *prefix*.*domain* form. *prefix* and *domain* are determined by the specifications of the UIM cloud native configuration for a given deployment. *prefix* is unique to the deployment, while *domain* is common for multiple deployments.

The default *domain* in UIM cloud native toolkit is `uim.org`.

For a particular deployment, as an example, this results in the following addresses:

- `dev1.wireless.uim.org` (for HTTP access)
- `admin.dev1.wireless.uim.org` (for WebLogic Console access)
- `t3.dev1.wireless.uim.org` (for T3 JMS/SAF access)

These "hostnames" must be routable to the entry point of your Ingress Controller or Load Balancer. For a basic validation, on the systems that access the deployment, edit the local hosts file to add the following entry:

> **Note**
>
> The hosts file is located in **/etc/hosts** on Linux and MacOS machines and in **C:\Windows\System32\drivers\etc\hosts** on Windows machines.

```
ip_address  dev1.wireless.uim.org   admin.dev1.wireless.uim.org
t3.dev1.wireless.uim.org
```

However, the solution of editing the hosts file is not easy to scale and co-ordinate across multiple users and multiple access environments. A better solution is to leverage DNS services at the enterprise level.

With DNS servers, a more efficient mechanism can be adopted. The mechanism is the creation of a domain level A-record:

```
A-Record: *.uim.org IP_address
```

If the target is not a load balancer, but the Kubernetes cluster nodes themselves, a DNS service can also insulate the user from relying on any single node IP. The DNS entry can be configured to map **.uim.org* to all the current Kubernetes cluster node IP addresses. You must update this mapping as the Kubernetes cluster changes with adding a new node, removing an old node, reassigning the IP address of a node, and so on.

With these two approaches, you can set up an enterprise DNS once and modify it only infrequently.

Configuring Kubernetes Persistent Volumes

Typically, runtime artifacts in UIM cloud native are created within the respective pod filesystems. As a result, they are lost when the pod is deleted. These artifacts include application logs, Fusion MiddleWare logs, and JVM Java Flight Recorder data.

While this impermanence may be acceptable for highly transient environments, it is typically desirable to have access to these artifacts outside of the lifecycle of the UIM could native instance. It is also highly recommended to deploy a toolchain for logs to provide a centralized view with a dashboard. To allow for artifacts to be independent of the pod, UIM cloud native allows for them to be maintained on Kubernetes Persistent Volumes.

UIM cloud native does not dictate the technology that supports Persistent Volumes, but provides samples for NFS-based persistence and BV-based persistence. Additionally, for UIM cloud native on an Oracle OKE cloud, you can use persistence based on File Storage Service (FSS) or Block Volume (BV).

Regardless of the persistence provider chosen, persistent volumes for UIM cloud native use must be configured:

- With accessMode ReadWriteMany for NFS-based Persistence
- With accessMode ReadWriteOnce for BV-based Persistence
- With capacity to support intended workload

Log size and retention policies can be configured as part of the shape specification.

About NFS-based Persistence

For use with UIM cloud native, one or more NFS servers must be designated.

It is highly recommended to split the servers as follows:

- At least one for the development instances and the non-sensitive test instances (for example, for Integration testing)
- At least one for the sensitive test instances (for example, for Performance testing, Stress testing, and production staging)
- One for the production instance

In general, ensure that the sensitive instances have dedicated NFS support, so that they do not compete for disk space or network IOPS with others.

The exported filesystems must have enough capacity to support the intended workload. Given the dynamic nature of the UIM cloud native instances it is prudent to put in place a set of operational mechanisms to:

- Monitor disk usage and warn when the usage crosses a threshold
- Clean out the artifacts that are no longer needed

If a toolchain such as ELK Stack picks up this data, then the cleanup task can be built into this process itself. As artifacts are successfully populated into the toolchain, they can be deleted from the filesystem. You must take care to only delete log files that have rolled over.

About BV-based Persistence

In case of using Block Volumes with UIM cloud native, perform the following to create PV-PVC pairs for each block volume:

1. Create a Persistent Volume (PV) for each block volume. The PV should specify the storage capacity, access mode, and the path to the block volume on the host node.
2. Create a Persistent Volume Claim (PVC) for each server that requires access to the block volume. The PVC should specify the storage class, access mode, and the storage capacity required.
3. Bind each PVC to a PV.
4. Repeat the steps from **1** to **3** for every server that requires access to the block volume, including the db-installer, introspector, admin, and for each managed server.

> **Note**
>
> Creating separate PVs and PVCs for each server enables each server to have its own dedicated storage space.

About Authentication

UIM cloud native requires external LDAP to be configured for human users to access UIM application. For **fixed users**, either embedded LDAP or external LDAP can be used. It is recommended to use embedded LDAP for fixed users during instance creation. These fixed users can perform cartridge deployment and application administrative tasks.

When UIM cloud instances use external authentication, ensure that you create separate users and groups for each environment (or class of environments) in the external LDAP service. The specifications of this depend on the LDAP service provider.

UIM cloud native toolkit provides a sample configuration that uses OpenLDAP to demonstrate how to integrate with external LDAP server for human users. For details on setting up the OpenLDAP server and the layout of the data within it, see "Setting Up Authentication" for more information.

Management of Secrets

UIM cloud native leverages Kubernetes Secrets to store sensitive information securely. This sensitive information is, at a minimum, the database credentials and the WebLogic administrator credentials. Additional credentials may be stored to authenticate with the external LDAP system. Your custom cartridges may need to communicate with other systems, such as

Order and Service Management (OSM). The credentials for such systems too are managed as Kubernetes Secrets.

These secrets need to be secured over their lifecycle by the Kubernetes cluster administration. RBAC should be used to restrict the entities that can describe, view, or mount these credentials.

UIM cloud native scripts assume that a set of pre-requisite secrets exist when they are invoked. As such, creation of the secrets is a pre-requisite step in the pipeline. UIM cloud native toolkit provides a sample script to create some of the common secrets it needs, but this script is interactive and therefore not suitable for Continuous Delivery (CD) automation pipelines. The sample script serves to provide a basic mechanism to add secrets and illustrates the names and structure of the secrets that UIM cloud native requires.

You can create the secrets manually by using the sample script for each instance. The sample can be augmented to include additional custom secrets. This method requires exposing RBAC for creating secrets for a larger group of users, which might not be desirable. It can also result in human errors, such as mistyping a password, which will only be detected during the runtime of the UIM instance.

A more sustainable and scalable option is using a secrets management system. There are several secrets management systems available for use with Kubernetes. Choose a system that offers a secure API (to be called from the CD pipeline) and populates the sensitive information as secrets into Kubernetes, as opposed to populating into pods through environment variables. The installation, configuration, and validation of such a secrets management system is a pre-requisite to uptake UIM cloud native. For details on setting up the secrets management system, see the documentation of the system that you adopt.

Using Kubernetes Monitoring Toolchain

A multi-node Kubernetes cluster with multiple users and an ever-changing workload requires a capable set of tools to monitor and manage the cluster. There are tools that provide data, rich visualizations and other capabilities such as alerts. UIM cloud native does not require any particular system to be used, but recommends using such a monitoring, visualization and alerting capability.

For UIM cloud native, the key aspects of monitoring are:

- Worker capacity in CPU and memory. The pods take up non-trivial amount of worker resources. For example, pods configured for production performance use 32 GB of memory. Monitoring the free capacity leads to predictable UIM instance creation and scale-up.
- Worker node disk pressure
- Worker node network pressure
- Health of the core Kubernetes services
- Health of WebLogic Kubernetes Operator
- Health of Nginx (or other load balancer in the cluster)

The namespaces and pods that UIM cloud native uses provide a cross instance view of UIM cloud native.

About Application Logs and Metrics Toolchain

UIM cloud native generates all logs that traditional UIM and WebLogic Server typically generate. The logs can be sent to a shared filesystem for retention and for retrieval by a toolchain such as Elastic Stack.

In addition, UIM cloud native generates metrics and JVM Java Flight Recorder (JFR) data. UIM cloud native exposes metrics for scraping by Prometheus. These can then be processed by a metrics toolchain, with visualizations like Grafana dashboards. Dashboards and alerts can be configured to enable sustainable monitoring of multiple UIM cloud native instances throughout their lifecycles. The UIM JFR data can be retrieved by Java Mission Control or such similar tools to analyze the performance of UIM at the JVM level. Performance metrics include heap utilization, threads stuck, garbage collection, and so on.

Oracle highly recommends using a toolchain to effectively monitor UIM cloud native instances. The dynamic lifecycle in UIM cloud native, in terms of deploying, scaling and updating an instance, requires proper monitoring and management of the database resources as well. For non-sensitive environments such as development instances and some test instances, this largely implies monitoring the tablespace usage and the disk usage, and adding disk space as needed.

Another important facet is to track PDB usage to ensure PDBs that are no longer required are deleted so that the resources are freed up. Sensitive environments such as production and stress test instances require close monitoring of the database resources such as CPU, SGA/PGA, top-runner SQLs, and IOPS.

A key implication of the dynamic behavior of UIM cloud native on the database is when the instances are dehydrated. Very often, there is a requirement to have a UIM instance kept around even when it is not being actively used. Such an environment lies idle until it is needed again. With UIM cloud native, there is no retained state within the run-time instance. The information on creating the instance is in the CD artifacts (the various specification files), and all the UIM application information is in the PDB. As a result, when the instance is not actively needed, all Kubernetes resources for it can be freed up by deleting the instance. This does not delete the PDB. The CD artifacts and the PDB can be used to rehydrate the instance when required. In the meantime, if the instance is not required for a while (or if there is database capacity pressure), the PDB can be unplugged to no longer consume any run-time resources. An unplugged PDB can even be transferred to another CDB and plugged in there.

Role of Continuous Integration (CI) Pipelines

The roles of CI pipelines in a UIM cloud native environment are as follows:

- To generate standard UIM cartridge JAR files and store them in a central location with appropriate path and naming convention for deployment. Developers run this automation as they modify cartridges for testing. Standalone mechanisms that generate "official" cartridge builds for testing and production use also run automation.
- To generate custom UIM cloud native images. The UIM cloud native images contain all the components needed to run UIM cloud native. However, you may require some customizations to be addressed in the image such as, additional applications to be co-hosted by the UIM WebLogic cluster, UI-Customization, Localization. Few of these customizations are layered on top of the UIM cloud native image to generate a custom image. Automation can accomplish this by running customization scripts that are provided in the image builder toolkit. The generated images must be uploaded to the internal container repository for use by deployment. The path and naming convention must be

followed to designate images that are in development versus images that are ready for testing; and to version the images themselves.

UIM cloud native does not mandate the use of a specific set of tools for CI automation. Common choices are GitLab CI and Jenkins. As part of preparing for UIM cloud native, you must evaluate CI automation tools and choose one that fits your business needs and the desired source control mechanisms.

Role of Continuous Delivery (CD) Pipelines

The role of CD pipelines in a UIM cloud native environment is to perform operations on the target Kubernetes cluster to automate the full lifecycle of a UIM cloud native instance.

The following are the main operations you must implement:

- Create instance: This must drive off the source-controlled UIM cloud native specification files and run through the various stages (secrets creation, PDB creation, UIM database installation, UIM instance creation, load balancer creation) to create a new UIM cloud native instance. Variability should be built in for some key phases as secrets may already exist and may need to be updated, or PDB may already exist with or without UIM schema, and so on. As a result, this automation is written to a "create-or-update" pattern.
- Update instance: This must be a variant of the instance creation automation, skipping the PDB creation and perhaps the load balancer (Ingress) creation. The automation takes the source-controlled UIM cloud native specification files, which have presumably been modified in some way since the instance was created, and runs through the steps to make those changes appear in the provisioned UIM instance. The specification changes could be as simple as a change in the number of desired Managed Servers, or could be as complex as introducing a new UIM container image.
- Delete instance: This must clean up the Kubernetes resources used by the instance. Typically, the PDB is left alone to be handled separately, but it is possible to chain its deletion to the clean up operation as well.

UIM cloud native does not mandate the use of a particular set of tools for CD automation. Common choices are GitLab CD and Jenkins. As part of preparing for UIM cloud native, you must evaluate CD automation tools and choose one that fits your business needs and the target Kubernetes environment.

Planning Your Container Engine for Kubernetes (OKE) Cloud Environment

This section provides information about planning your cloud environment if you want to use Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE) for UIM cloud native. Some of the components, services, and capabilities that are required and recommended for a cloud native environment are applicable to the Oracle OKE cloud environment as well.

- **Kubernetes** and **Container Images**: You can choose from the version options available in OKE as long as the selected version conforms to the range described in the section about planning cloud native environment.
- **Container Image Management**: UIM cloud native recommends using Oracle Cloud Infrastructure Registry with OKE. Any other repository that you use must be able to serve images to the OKE environment in a quick and reliable manner. The UIM cloud native images are of the order of 3 GB each.
- **Oracle Multitenant Database**: It is strongly recommended to run Oracle DB outside of OKE, but within the same Oracle Cloud Infrastructure tenancy and the region as an Oracle

DB service (BareMetal, VM, or ExaData). The database version should be 19c. You can choose between a standalone DB or a multi-node RAC.

- **Helm** and **Oracle WebLogic Kubernetes Operator**: Install Helm and Oracle WebLogic Kubernetes Operator as described for the cloud native environment into the OKE cluster.
- **Persistent Volumes**: Use NFS-based persistence. UIM cloud native recommends the use of Oracle Cloud Infrastructure File Storage service in the OKE context.
- **Authentication** and **Secrets Management**: These aspects are common with the cloud native environment. Choose your mechanisms to deliver these capabilities and implement them in your OKE instance.
- **Monitoring Toolchains**: While the Oracle Cloud Infrastructure Console provides a view of the resources in the OKE cluster, it also enables you to use the Kubernetes Dashboard. Any additional monitoring capability must be built up.
- **CI** and **CD Pipelines**: The considerations and actions described for CI and CD pipelines in the cloud native environment apply to the OKE environment as well.

Compute Disk Space Requirements

Given the size of the UIM cloud native container images (approximately 3 GB), the size of the UIM cloud native containers, and the volume of the UIM logs generated, it is recommended that the OKE worker nodes have at least 40 GB of free space that the **/var/lib** filesystem can use. Add disk space if the worker nodes do not have the recommended free space in the **/var/lib** filesystem.

Work with your Oracle Cloud Infrastructure OKE administrator to ensure worker nodes have enough disk space. Common options are to use Compute shapes with larger boot volumes or to mount an Oracle Cloud Infrastructure Block Volume to **/var/lib/docker**.

> **Note**
>
> The reference to logs in this section applies to the container logs and other infrastructure logs. The space considerations still apply even if the UIM cloud native logs are being sent to an NFS Persistent Volume.

Connectivity Requirements

UIM cloud native assumes the connectivity between the OKE cluster and the Oracle CDBs is a LAN-equivalent in reliability, performance and throughput. This can be achieved by creating the Oracle CDBs within the same tenancy as the OKE cluster, and in the same Oracle Cloud Infrastructure region.

UIM cloud native allows for the full range of Oracle Cloud Infrastructure "cloud-to-ground" connectivity options for integrating the OKE cluster with on-premise applications and users. Selecting, provisioning, and testing such connectivity is a critical part of adopting Oracle Cloud Infrastructure OKE.

Using Load Balancer as a Service (LBaaS)

For load balancing, you have the option of using the services available in OKE. The infrastructure for OKE is provided by Oracle's IaaS offering, Oracle Cloud Infrastructure. In OKE, the Master node IP address is not exposed to the tenants. The IP addresses of the worker nodes are also not guaranteed to be static. This makes DNS mapping difficult to

achieve. Additionally, it is also required to balance the load between the worker nodes. In order to fulfill these requirements, you can use Load Balancer as a Service (LBaaS) of Oracle Cloud Infrastructure.

The load balancer can be created using the service descriptor in **$COMMON_CNTK/samples/charts/haproxy/oci-lb-haproxy.yaml**. The subnet ID referenced in this file must be filled in from your Oracle Cloud Infrastructure environment (using the subnet configured for your LBaaS). The port values assume you have installed HAProxy using the unchanged sample values.

The configuration can be applied using the following command (or for traceability, by wrapping it into a Helm chart):

```
$ kubectl apply -f oci-lb-haproxy.yaml

service/oci-lb-service-haproxy configured
```

The Load Balancer service is created for HAProxy pods in the haproxy namespace. Once the Load Balancer service is created successfully, an external IP address is allocated. This IP address must be used for DNS mapping.

```
$ kubectl get svc -n haproxy oci-lb-service-haproxy

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S)

oci-lb-service-haproxy LoadBalancer 10.96.103.118 100.77.24.178 80:32006/
TCP,443:32307/TCP
```

For additional details, see the following:

- "Creating Load Balancers to Distribute Traffic Between Cluster Nodes" in Oracle Cloud Infrastructure documentation.
- "Load Balancer Annotations" in Oracle GitHub documentation.

About Using Oracle Cloud Infrastructure Domain Name System (DNS) Zones

While a custom DNS service can provide the addressing needs of UIM cloud native even when UIM is running in OKE, you can evaluate the option of Oracle Cloud Infrastructure Domain Name System (DNS) zones capability. Configuration of DNS zones (and integration with on-premise DNS systems) is not within the scope of UIM cloud native.

Using Persistent Volumes and File Storage Service (FSS)

In the OKE cluster, UIM cloud native can leverage the high performance, high capacity, high reliability File Storage Service (FSS) as the backing for the persistent volumes of UIM cloud native. There are two flavors of FSS usage in this context:

- Allocating FSS by setting up NFS mount target
- Native FSS

To use FSS through an NFS mount target, see instructions for allocating FSS and setting up a Mount Target in "Creating File Systems" in the Oracle Cloud Infrastructure documentation. Note down the Mount Target IP address and the storage path and use these in the UIM cloud

native instance specification as the NFS host and path. This approach is simple to set up and leverages the NFS storage provisioner that is typically available in all Kubernetes installations. However, the data flows through the mount target, which models an NFS server.

FSS can also be used natively, without requiring the NFS protocol. This can be achieved by leveraging the FSS storage provisioner supplied by OKE. The broad outline of how to do this is available in the blog post "Using File Storage Service with Container Engine for Kubernetes" on the Oracle Cloud Infrastructure blog.

Leveraging Oracle Cloud Infrastructure Services

For your OKE environment, you can leverage existing services and capabilities that are available with Oracle Cloud Infrastructure. The following table lists the Oracle Cloud Infrastructure services that you can leverage for your OKE cloud environment.

Table 2-1 Oracle Cloud Infrastructure Services for OKE Cloud Environment

Type of Service	Service	Indicates Mandatory / Recommended / Optional
Developer Service	Container Clusters	Mandatory
Developer Service	Registry	Recommended
Core Infrastructure	Compute Instances	Mandatory
Core Infrastructure	File Storage	Recommended
Core Infrastructure	Block Volumes	Optional
Core Infrastructure	Networking	Mandatory
Core Infrastructure	Load Balancers	Recommended
Core Infrastructure	DNS Zones	Optional
Database	BareMetal, VM, and ExaData	Recommended

Validating Your Cloud Environment

Before you start using your cloud environment for deploying UIM cloud native instances, you must validate the environment to ensure that it is set up properly and that any prevailing issues are identified and resolved. This section describes the tasks that you should perform to validate your cloud environment.

You can validate your cloud environment by:

- Performing a smoke test of the Kubernetes cluster
- Validating the common building blocks in the Kubernetes cluster
- Running tasks and procedures in Oracle WebLogic Kubernetes Operator Quickstart

Performing a Smoke Test

You can perform a smoke test of your Kubernetes cloud environment by running nginx. This procedure validates basic routing within the Kubernetes cluster and access from outside the environment. It also allows for initial RBAC examination as you need to have permissions to perform the smoke test. For the smoke test, you need nginx 1.14.2 container image.

> **Note**
>
> The requirement of the nginx container image for the smoke test can change over time. See the content of the **deployment.yaml** file in step 3 of the following procedure to determine which image is required. Alternatively, ensure that you have logged in to Docker Hub so that the system can download the required image automatically.

To perform a smoke test:

1. Download the nginx container image from Docker Hub.

   For details on managing container images, see "About Container Image Management".

2. After obtaining the image from Docker Hub, upload it into your private container repository and ensure that the Kubernetes worker nodes can access the image in the repository.

   Oracle recommends that you download and save the container image to the private Docker repository even if the worker nodes can access Docker Hub directly. The images in the UIM cloud native toolkit are available only through your private Docker repository.

3. Run the following commands:

   ```
   kubectl apply -f https://k8s.io/examples/application/deployment.yaml # the
   deployment specifies two replicas
   kubectl get pods     # Must return two pods in the Running state
   kubectl expose deployment nginx-deployment --type=NodePort --name=external-
   nginx
   kubectl get service external-nginx    # Make a note of the external port
   for nginx
   ```

   These commands must run successfully and return information about the pods and the port for nginx.

4. Open the following URL in a browser:

   ```
   http://master_IP:port/
   ```

   where:

   - *master_IP* is the IP address of the Master node of the Kubernetes cluster or the external IP address for which routing has been set up
   - *port* is the external port for the **external-nginx** service

5. To track which pod is responding, on each pod, modify the text message in the web page served by nginx. In the following example, this is done for a deployment of two pods:

   ```
   $ kubectl get pods -o wide | grep nginx
   nginx-deployment-5c689d88bb-g7zvh   1/1     Running   0          1d
   10.244.0.149   worker1   <none>
   nginx-deployment-5c689d88bb-r68g4   1/1     Running   0          1d
   10.244.0.148   worker2   <none>
   $ cd /tmp
   $ echo "This is pod A - nginx-deployment-5c689d88bb-g7zvh - worker1" >
   index.html
   $ kubectl cp index.html nginx-deployment-5c689d88bb-g7zvh:/usr/share/nginx/
   html/index.html
   ```

```
$ echo "This is pod B - nginx-deployment-5c689d88bb-r68g4 - worker2" >
index.html
$ kubectl cp index.html nginx-deployment-5c689d88bb-r68g4:/usr/share/nginx/
html/index.html
$ rm index.html
```

6. Check the index.html web page to identify which pod is serving the page.
7. Check if you can reach all the pods by running refresh (Ctrl+R) and hard refresh (Ctrl+Shift+R) on the index.html Web page.
8. If you see the default nginx page, instead of the page with your custom message, it indicates that the pod has restarted. If a pod restarts, the custom message in the page gets deleted.

   Identify the pod that restarted and apply the custom message for that pod.
9. Increase the pod count by patching the deployment.

   For instance, if you have three worker nodes, run the following command:

   > **Note**
   >
   > Adjust the number as per your cluster. You may find you have to increase the pod count to more than your worker node count until you see at least one pod on each worker node. If this is not observed in your environment even with higher pod counts, consult your Kubernetes administrator. Meanwhile, try to get as much worker node coverage as reasonably possible.

   ```
   kubectl patch deployment nginx-deployment -p '{"spec":{"replicas":3}}' --
   type merge
   ```
10. For each pod that you add, repeat step 5 to step 8.

Ensuring that all the worker nodes have at least one nginx pod in the Running state ensures that all worker nodes have access to Docker Hub or to your private Docker repository.

Validating Common Building Blocks in the Kubernetes Cluster

To approach UIM cloud native in a sustainable manner, you must validate the common building blocks that are on top of the basic Kubernetes infrastructure individually. The following sections describe how you can validate the building blocks.

Network File System (NFS)

UIM cloud native uses Kubernetes Persistent Volumes (PV) and Persistent Volume Claims (PVC) to use a pod-remote destination filesystem for UIM logs and performance data. By default, these artifacts are stored within a pod in Kubernetes and are not easily available for integration into a toolchain. For these to be available externally, the Kubernetes environment must implement a mechanism for fulfilling PV and PVC. The Network File System (NFS) is a common PV mechanism.

For the Kubernetes environment, identify an NFS server and create or export an NFS filesystem from it.

Ensure that this filesystem:

- Has enough space for the UIM logs and performance data.

- Is mountable on all the Kubernetes worker nodes

Create an nginx pod that mounts an NFS PV from the identified server. For details, see the documentation about "Kubernetes Persistent Volumes" on the Kubernetes website. This activity verifies the integration of NFS, PV/PVC and the Kubernetes cluster. To clean up the environment, delete the nginx pod, the PVC, and the PV.

Ideally, data such as logs and JFR data is stored in the PV only until it can be retrieved into a monitoring toolchain such as Elastic Stack. The toolchain must delete the rolled over log files after processing them. This helps you to predict the size of the filesystem. You must also consider the factors such as the number of UIM cloud native instances that will use this space, the size of those instances, the volume of orders they will process, and the volume of logs that your cartridges generate.

Validating the Load Balancer

For a development-grade environment, you can use an in-cluster software load balancer. Common cloud native toolkit provides documentation and samples that show you how to use Nginx to perform load balancing activities for your Kubernetes cluster.

It is not necessary to run through Kubernetes Ingress as part of validating the environment. However, if the UIM cloud native instances have connectivity issues with HTTP/HTTPS traffic, and the UIM logs do not show any failures, it might be worthwhile to take a step back and validate using Kubernetes documentation about Ingress and Ingress Controller.

A more intensive environment, such as a test, a production, a pre-production, or performance environments can additionally require a more robust load balancing service to handle the HTTP/HTTPS traffic. For such environments, Oracle recommends using a load balancing hardware that is set up outside the Kubernetes cluster. A few examples of external load balancers are Oracle Cloud Infrastructure LBaaS for OKE, Google's Network LB Service in GKE, and F5's Big-IP for private cloud. The actual selection and configuration of an external load balancer is outside the scope of UIM cloud native itself, but is an important component to sort out in the implementation of UIM cloud native. For more details on the requirements and options, see "Integrating UIM".

To validate the ingress controller of your choice, you can use the same nginx deployment used in the smoke test described earlier. This is valid only when run in a Kubernetes cluster where multiple worker nodes are available to take the workload.

To perform a smoke test of your ingress setup:

1. Run the following commands:

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
kubectl get pods -o wide    # two nginx pods in Running state; ensure
these are on different worker nodes
cat > smoke-internal-nginx-svc.yaml <<EOF
apiVersion: v1
kind: Service
metadata:
  name: smoke-internal-nginx
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
```

```
  sessionAffinity: None
  type: ClusterIP
EOF
kubectl apply -f ./smoke-internal-nginx-svc.yaml
kubectl get svc smoke-internal-nginx
```

2. Create your ingress targeting the **internal-nginx** service. The following text shows a sample ingress annotated to work with the HAProxy ingress controller:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: smoke-haproxy-ingress
  annotations:
    haproxy.org/cookie-persistence: "uimhaproxycookie"
spec:
  ingressClassName: haproxy
  rules:
  - host: smoke.nginx.uimtest.org
    http:
      paths:
      - backend:
          service:
             name: smoke-internal-nginx
             port:
                 number: 80
        pathType: ImplementationSpecific
```

   If Ingress Object is created and points to Ingress-Haproxy Controller through **ingressClassName** property, the **Ingress-Haproxy** Controller creates a reverse proxy and a load balancer for the Nginx deployment. For more details, see Ingress HAProxy Controller documentation.

   If you plan to use other ingress controllers, refer to the documentation about the corresponding controllers for information on creating the appropriate ingress and make it known to the controller. The ingress definition should be largely reusable, with ingress controller vendors describing their own annotations that should be specified, instead of the HAProxy annotation used in the example.

3. Create a local DNS/hosts entry in your client system mapping **smoke.nginx.uimtest.org** to the IP address of the cluster, which is typically the IP address of the Kubernetes Master node, but could be configured differently.

4. Open the following URL in a browser:

   `http://smoke.nginx.uimtest.org:`*`Haproxy_Port`*`/`

   where *Haproxy_Port* is the external port that Ingress HAProxy has been configured to expose.

5. Verify that the web address opens and displays the Nginx default page.

Your ingress controller must support session stickiness for UIM cloud native. To learn how stickiness should be configured, refer to the documentation about the ingress controller you choose. For HAProxy, stickiness must be set up by providing annotation `haproxy.org/cookie-persistence: "uimhaproxycookie"` to ingress object. For testing purposes, you

can modify the **internal-nginx** service to enable stickiness by running the following commands:

```
kubectl delete ingress smoke-nginx-ingress
vi smoke-internal-nginx-svc.yaml
# Add an annotation section under the metadata section:
#   annotation:
#     nginx.ingress.kubernetes.io/affinity: "cookie"
#     nginx.ingress.kubernetes.io/affinity-mode: "persistent"
kubectl apply -f ./smoke-internal-nginx-svc.yaml
# now apply back the ingress smoke-nginx-ingress using the above yaml
definition
```

Other ingress controllers may have different configuration requirements for session stickiness. Once you have configured your ingress controller, and the `smoke-haproxy-ingress` and `smoke-internal-nginx` services as required, repeat the browser-based procedure to verify and confirm if nginx is still reachable. As you refresh (Ctrl+R) the browser, you should see the page getting served by one of the pods. Repeatedly refreshing the web page should show the same pod servicing the access request.

To further test session stickiness, you can either do a hard refresh (Ctrl+Shift+R) or restart your browser (you may have to use the browser in Incognito or Private mode), or clear your browser cache for the access hostname for your Kubernetes cluster. You may observe that the same nginx pod or a different pod is servicing the request. Refreshing the page repeatedly should stick with the same pod while hard refreshes should switch to the other pod occasionally. As the deployment has two pods, chances of a switch with a hard refresh are 50%. You can modify the deployment to increase the number of replica nginx pods (controlled by the `replicas` parameter under `spec`) to increase the odds of a switch. For example, with four nginx pods in the deployment, the odds of a switch with hard refresh rise to 75%. Before testing with the new pods, run the commands for identifying the pods to add unique identification to the new pods. See the procedure in "Performing a Smoke Test" for the commands.

To clean up the environment after the test, delete the following services and the deployment:

- `smoke-haproxy-ingress`
- `smoke-internal-nginx`
- `nginx-deployment`

Running Oracle WebLogic Kubernetes Operator Quickstart

Oracle recommends that you validate your new Kubernetes environment for UIM cloud native by performing the procedures described in Oracle WebLogic Kubernetes Operator Quickstart available at: https://oracle.github.io/weblogic-kubernetes-operator/quickstart/

The quickstart guide provides instructions for creating a WebLogic deployment in a Kubernetes cluster with the Oracle WebLogic Kubernetes Operator. The guide also provides instructions for downloading and installing a load balancer, and a domain. Follow the instructions provided above for Helm 3.x.

When you run and complete the tasks in the quickstart successfully, the following aspects of the cloud environment are tested and verified:

- Private Docker repository (or procedures to sync per-node Docker cache on a multi-node Kubernetes cluster)

- Initial view of the chosen in-cluster load balancers
- RBAC for WebLogic Kubernetes Operator
- Procedure to introduce secrets into the cloud environment
- Basic compatibility of the cloud environment with WebLogic Kubernetes Operator

The quickstart also contains instructions for cleaning up the environment after you finish the validation and testing. Perform these clean-up procedures to return the environment to the original state for UIM cloud native.

After completing the clean-up procedures, ensure that the WebLogic Kubernetes Operator CustomResourceDefinition (CRD) is removed from your cluster by running the following commands:

```
$ kubectl get crd domains.weblogic.oracle
# if this returns an existing CRD even after WKO quickstart cleanup, then run:
$ kubectl delete crd domains.weblogic.oracle
```

3

Creating the UIM Cloud Native Images

UIM cloud native requires container images be made available to create and manage UIM cloud native instances. This chapter describes how to create those UIM cloud native images.

UIM cloud native requires two container images. The UIM DB installer image is used to manage the UIM and Fusion MiddleWare schemas -create,delete,upgrade - as well as purging in the UIM schema. It is also used to generated encrypted weblogic credentials. The other image is the UIM image itself. This image is the basis for all of the long running pods - the WebLogic admin server and all the Managed Servers that comprise a UIM cloud native instance. Each image is built on top of a Linux base image and adds Java, Fusion MiddleWare components and UIM product components on top.

UIM Cloud native images are created using the UIM cloud native builder toolkit and a dependency manifest file. The UIM cloud native Image Builder is intended to be run as part of a Continuous Integration process that generates images. It needs to run on Linux and have access to the local Docker daemon. The versions of these are as per the UIM statement of certification in the UIM documentation. The dependency manifest is a file that describes all the versions and patches required to build out the image.

Downloading the UIM Cloud Native Image Builder

You download the UIM cloud native image builder from My Oracle Support at: https://support.oracle.com

The UIM cloud native image builder is bundled with the following components:

- UIM cloud native builder kit. The kit contains:
  - The UIM Domain WDT Model.
  - The UIM DB Installer scripts and manifest files.
- Staging directory structure.

Prerequisites for Creating UIM Images

The pre-requisites for building UIM cloud native images are:

- Docker client and daemon on the build machine.
- Installers for WebLogic Server and JDK. Download these from the Oracle Software Delivery Cloud:

  https://edelivery.oracle.com
- Required patches. Download these from My Oracle Support:

  https://support.oracle.com/
- Java, installed with JAVA_HOME set in the environment.
- Bash, to enable the `<tab>` command complete feature.

See "UIM Software Compatibility" in *UIM Compatibility Matrix* for details about the required and supported versions of these pre-requisite software.

Configuring the UIM Cloud Native Images

The dependency manifest file describes the input that goes into the UIM images. It is consumed by the image build process. The default configuration in the latest manifest file provides all the necessary components and required patches for creating the UIM cloud native images easily.

You can also modify the manifest file to extend it to meet your requirements. This enables you to:

- Specify any Linux image as the base, as long as its binary is compatible with Oracle Linux.
- Upgrade the Oracle Enterprise Linux version to a newer version to uptake a quarterly CPU.
- Upgrade the JDK version to a newer JDK version to uptake a quarterly CPU.
- Upgrade the Fusion Middleware version to a newer version. For example, you upgrade the Fusion Middleware version to a newer version when you initiate the upgrade to pick up new PSU or when Oracle recommends a new update.
- Change the set of patches applied on WebLogic Server, Coherence, Fusion Middleware, and OPatch to stay aligned with evolving UIM recommendations.
- Change the UIM artifacts to newer artifacts to uptake a new UIM patch.
- Choose a different **userid** and **groupid** for **oracle:oracle user:group** that the image specifies. The default is **1000:1000**.

The breakdown of each section in the dependency manifest file is as follows:

> **Note**
>
> The `schemaVersion` and `date` parameters are maintained by Oracle. Do not modify these parameters.
>
> Version numbers provided here are only examples. The manifest file used specifies the actual versions currently recommended.

- **UIM Cloud Native Infrastructure Image**

  While not required by UIM cloud native to create or manage UIM instances, this infrastructure image is a necessary building block of the final UIM container image.

  ```
  linux:
      vendor: Oracle
      version: 9-slim
      image: container-registry.oracle.com/os/oraclelinux:9-slim
  ```

  The `Linux` parameter specifies the base Linux image to be used as the base docker image. The version is the two-digit version from **/etc/redhat-release**.

  The vendor and version details are specified and used for:

  – Validation when an image is built.

– Querying at run-time. To troubleshoot issues, Oracle support requires you to provide these details in the manifest file used to build the image.

```
userGroup:
  username: oracle
  userid: 1000
  groupname: oracle
  groupid: 1000
```

The `userGroup` parameter specifies the default userId and groupId for `oracle`

```
 jdk:
    vendor: Oracle
    version: 21.0.6
    path: $CN_BUILDER_STAGING/java/jdk-2106-linux-x64 bin.tar.gz
```

The `jdk` parameter specifies the JDK vendor, version, and the staging path.

```
fmw:
    version: 14.1.2.0.0
    path: $CN_BUILDER_STAGING/fmw/install/
fmw_14.1.2.0.0_infrastructure_Disk1_1of1.zip
```

The `fmw` parameter specifies the Fusion Middleware version and staging path.

```
 oPatch:
    description: Weblogic Opatch
    patchNumber: 28186730
    patchId: 28186730_13.9.4.2.19
    path: $CN_BUILDER_STAGING/fmw/patch/p28186730_1394219_Generic.zip
```

The `oPatch` parameter specifies the Oracle Patch tool and staging path.

```
fmwPatch:
    - description: Coherence Cumulative Patch 14.1.2.0.1
      patchNumber: 37658370
      patchId: 37658370_14.1.2.0.0
      path: $CN_BUILDER_STAGING/fmw/patch/p37658370_141200_Generic.zip
    - description: WLS PATCH SET UPDATE 14.1.2.0.250102
      patchNumber: 37439198
      patchId: 37439198_14.1.2.0.0
      path: $CN_BUILDER_STAGING/fmw/patch/p37439198_141200_Generic.zip
    - description: OPSS Bundle Patch 12.2.1.4.220311(APR 2022 CPU)
      patchNumber: 33950717
      patchId: 33950717_12.2.1.4.0
      path: $CN_BUILDER_STAGING/fmw/patch/p33950717_122140_Generic.zip
    - description: FMW COMMON THIRD PARTY SPU 12.2.1.4.0 (OCT 2023 CPU)
      patchNumber: 35882299
      patchId: 35882299_12.2.1.4.0
      path: $CN_BUILDER_STAGING/fmw/patch/p35882299_122140_Generic.zip
    - description: FMW Control SPU Patch (OCT 2022 CPU)
      patchNumber: 34542329
      patchId: 34542329_12.2.1.4.0
```

```
      path: $CN_BUILDER_STAGING/fmw/patch/p34542329_122140_Generic.zip
    - description: LOGIN FAILS AFTER APPLYING 1.80.331 JDK (APR 2022 CPU)
      patchNumber: 33903365
      patchId: 33903365_12.2.1.4.0
      path: $CN_BUILDER_STAGING/fmw/patch/p33903365_122140_Generic.zip
    - description: DMS Metric table uses UUID for Keys
      patchNumber: 28334768
      patchId: 28334768_12.2.1.4.0
      path: $CN_BUILDER_STAGING/fmw/patch/p28334768_122140_Generic.zip
    - description: STUCK THREAD AT
```

The `fmwPatch` parameter specifies additional patches and their staging paths.

- **UIM Cloud Native Image**

> **Note**
>
> Do not modify this section other than for name, tag, and layertag. Rest of the parameters are maintained by Oracle.

```
uimCnImage:
  name: uim-cn-base
  tag: 8.0.0.0.0
  layertag: 8.0.0.0.0.1
  wdt:
    version: 4.3.3
    path: $CN_BUILDER_STAGING/cnsdk/tools/weblogic-deploy.zip
  modelfiles: $CN_BUILDER_STAGING/cnsdk/uim-model/uim-domain-config/uim-
base-domain.yaml,$CN_BUILDER_STAGING/cnsdk/uim-model/uim-domain-config/
properties/docker-build/domain.properties
  application: $CN_BUILDER_STAGING/cnsdk/uim-model/uim-app-archive.zip
  customApplication: $CN_BUILDER_STAGING/cnsdk/uim-model/uim-custom-
archive.zip
  customConfig: $CN_BUILDER_STAGING/cnsdk/uim-model/uim-custom-config
  configfiles: $CN_BUILDER_STAGING/cnsdk/uim-model/UIM
  dockerExtension: $CN_BUILDER_STAGING/cnsdk/uim-model/
additionalBuildCommands.txt
```

Where `name` is the name of the UIM image, `tag` is the tag name of the UIM image, and `layertag` is the tag name of the customized UIM image.

The `uimCnImage` section specifies details about the UIM artifacts required to build the UIM base image and UIM layered image. These include the inventory.ear, sharedLibraries, UIM application configuration files, customizations, WDT and base model files.

- **UIM Cloud Native DB Installer Image**

```
uimCnDbInstallerImage:
  name: uim-cn-db-installer
  tag: 8.0.0.0.0

  dbtools:
    vendor: Oracle
    path: $CN_BUILDER_STAGING/cnsdk/uim-db/ora_uim_dbtools.jar
```

The `uimCnDbInstallerImage` parameter specifies the DB Installer image name and version. This includes UIM DB Utility jar and Password Encryptor jar.

Creating the UIM Cloud Native Images

To create the UIM image or UIM DB Installer image, the image builder initially generates infrastructure image with the following:

- Starts with a base-level operating system image (for example, **oraclelinux:9-slim**).
- Creates user and group (for example, oracle:oracle).
- Updates the image with the necessary packages for installing Fusion Middleware.
- Installs Java, Fusion Middleware and applies patches.

To create the UIM image, the image builder does the following:

- Infrastructure image is reused if it exists, else it builds infrastructure image.
- Installs the UIM application base on the WDT model along with customizations.

To create the UIM DB Installer image, the image builder does the following:

- Infrastructure image is reused if it exists else it builds infrastructure image.
- Installs the UIM DB Installer with **ora_uim_dbtools.jar** and **password_encryptor.jar**.

You can specify any Linux image as the base, as long as its binary is compatible with Oracle Linux and conforms to the compatibility matrix. See "UIM Software Compatibility" in *UIM Compatibility Matrix* for details about the supported software.

The following packages must be installed onto the given base image, or be already present:

- gzip
- tar
- unzip

Creating the UIM and UIM DB Installer Images

To create the UIM and UIM DB Installer images:

1. Create the workspace directory:

   ```
   mkdir workspace
   ```

2. Obtain and untar the UIM image builder file: **uim-image-builder.tar.gz** to the workspace directory:

   ```
   tar -xf ./uim-image-builder.tar.gz --directory workspace
   ```

3. Download JDK to the **workspace/uim-image-builder/staging/javadirectory** directory. The JDK version to be downloaded is described in the dependency manifest file.

   ```
   cp jdk-2106-linux-x64 bin.tar.gz ./workspace/uim-image-builder/staging/
   java/jdk-2106-linux-x64 bin.tar.gz
   ```

4. From Oracle Software Delivery Cloud, download Fusion Middleware Infrastructure installer for the version listed in the dependency manifest file under the **fmw** tag. The downloaded package name may differ (for example, **V983368-01.zip**). and copy it to the **workspace/**

uim-image-builder/staging/fmw/install directory. The Fusion Middleware Infrastructure installer version to be download is described in the dependency manifest file under the `fmw` section.

```
cp fmw_14.1.2.0.0_infrastructure_Disk1_1of1.zip ./workspace/uim-image-
builder/staging/fmw/install/fmw_14.1.2.0.0_infrastructure_Disk1_1of1.zip
```

5. From https://github.com/oracle/weblogic-deploy-tooling/releases, download WDT 4.3.6 (weblogic-deploy.zip) and copy it to the **workspace/uim-image-builder/staging/cnsdk/tools** directory.
6. From https://github.com/oracle/weblogic-image-tool/releases, download WIT 1.15.0 (imagetool.zip) and copy it to the **workspace/uim-image-builder/staging/cnsdk/tools** directory.
7. Download all the listed patches to the **workspace/uim-image-builder/staging/fmw/patch** directory. The list of required patches is in the dependency manifest file in the `oPatch` and `fmwPatch` sections.
   You can download the patches using any of the following options:
   - (Recommended) Manually search for and download each OPatch/FMW patches from Oracle Support to the current working directory and then copy to the staging directory.

     ```
     cp pxxxxxx_xxxxx_Generic.zip ./workspace/uim-image-builder/staging/fmw/
     patch
     ```

   - Provide your My Oracle Support account credentials when invoking the **build-uim-images.sh** script, and let the builder download the patches automatically:

     > **Note**
     >
     > Some patches may not be retrievable in this manner. If the image build process fails with errors about a missing patch, use the recommended option.

     ```
     ./workspace/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
     s $STAGING -c uim -u MOS_username -p MOS_password
     ```

8. Run **customization.sh** if you made any customizations. For making customizations, see "Customizing Images". Export the variables as required and then run **customization.sh**:

   ```
   mkdir workspace/customization
   mkdir workspace/temp
   ```

   - Export the work space as follows. **WORKSPACEDIR** is the mandatory parameter that has the path where **uim-image-builder** is extracted.

     ```
     export WORKSPACEDIR=$(pwd)/workspace
     ```

- Export the custom folder as follows. **CUSTOMFOLDER** is the parameter for the path where the custom folder of **uim-app-archive.zip** is copied and modified for the customization.

```
export CUSTOMFOLDER=$(pwd)/workspace/customization
###extract custom folder of uim-app-archive.zip in this directory
```

- Export the temp directory as follows. **TEMPDIR** is the mandatory parameter to store the **temp** location where we have the merged folder of **uim-app-archive.zip** with customizations for any future reference.

```
export TEMPDIR=$(pwd)/workspace/temp
```

- Apply customizations to UIM application as follows:

```
./workspace/uim-image-builder/bin/customization.sh
```

> **Note**
>
> You can also export the following variables:
>
> - **CARTRIDGESDIR** is the parameter that has the path to copy solution cartridges. The default path is `$CUSTOMFOLDER/custom/cartridges`.
> - **PURGELIMIT** the optional parameter that sets the number of folders. Not setting any value defaults to having 3 existing folders in it.

9. Run **build-uim-images.sh** and pass the dependency manifest file, staging path, and the type of image to be created.

```
export DMANIFEST=$(pwd)/workspace/uim-image-builder/bin/
uim_cn_ci_manifest.yaml
export STAGING=$(pwd)/workspace/uim-image-builder/staging
```

- To create UIM image, use `-c uim` as shown:

```
./workspace/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
s $STAGING -c uim
```

- To create UIM DB installer image, use `-c dbinstaller` as shown:

```
./workspace/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
s $STAGING -c dbinstaller
```

- To create UIM layered image, use `-c layer` as shown:

```
./workspace/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
s $STAGING -c layer
```

> **Note**
>
> Run **customization.sh** before you create UIM layered image. This can used during solution development phase for creating updated image in case of Java code changes in the cartridges.

These steps can be included into your CI pipeline as long as the required components are already downloaded to the staging area.

Post-build Image Management

The UIM cloud native image builder creates images with names and tags based on the settings in the manifest file. By default, this results in the following images:

- uim-cn-infrastructure:14.1.2.0.0
- uim-cn-base:8.0.0.0.0
- uim-cn-db-installer:8.0.0.0.0
- uim-cn-base:8.0.0.0.1

> **Note**
>
> An optional layered image is created, if **Customization** is enabled and either Java Ruleset code or configuration files are present in the solution cartridges.

Once images are built in a CI pipeline, the pipeline uniquely tags the images and pushes them to an internal Docker repository. An uptake process can then be triggered for the new images:

- Sanity Test
- Development Test (for explicit retesting of scenarios that triggered the rebuild, if any)
- System Test
- Integration Test
- Pre-Production Test
- Production

Customizing Images

Various customizations such as UI, Localization, Web Services, and so on can be performed during the docker image creation. To apply these customizations, you have to run an additional script **uim-image-builder/bin/customization.sh**, in the UIM image builder toolkit before creating the UIM docker image. Based on the customization type, an additional layer is generated with the Layer tag, as defined in the **uim-image-builder/bin/uim_cn_ci_manifest.yaml** file. For example, if solution cartridges contain configuration files, Java ruleset code, and for custom applications, the layered build gets generated.

> **Note**
>
> For certain customization types, additional layer is built on the top of the base image. This approach makes a quick turnaround time for validating solution cartridge changes during the development phase. In case of any changes to the Javaruleset code or configuration files in the cartridges, you do not need to build the UIM Image but you can build only the layer after running **uim-image-builder/bin/customization.sh** as follows:
>
> ```
> ./workspace/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
> s $STAGING -c layer
> ```

Including User Interface Customizations and Localizing UIM Help in UIM Cloud Native Images

To include the user interface customizations in UIM cloud native images:

1. Customize the user interface and generate **inv.war** file after including the UI customizations. See "Overview" in *UIM Developer's Guide* for customizing the user interface and deploying the customizations.
2. Include **Localizing UIM Help** customizations in the same **inv.war** file. See "Localizing UIM" for localizing UIM Help and "Localizing the Network Plan and Design Process Page" for localizing Network Pland and Design in *UIM Developer's Guide*.
3. Place the **inv.war** file in the **$CUSTOMFOLDER\custom\ui_customization** folder.

> **Note**
>
> The CUSTOMFOLDER is the parameter for the path where the custom folder of **uim-app-archive.zip** is copied and modified for the customization.

4. If there are any logo customizations, place the **comms-platform-ui.jar** file in the **custom\ui_customization** folder.

Including Custom Web Services

To include the custom web services in UIM Cloud Native images:

1. Develop custom web services and create the WAR file. See "Web Services Overview" in *UIM Web Services Developer's Guide* for developing custom web services.
2. Place the web service's WAR file in the CUSTOMFOLDER at **custom\customWS**.

> **Note**
>
> The CUSTOMFOLDER is the parameter for the path where the custom folder of **uim-app-archive.zip** is copied and modified for the customization.

3. Update **application.xml** and **inventory-clusterPlan.xml** files with the new custom web service details.

During the image creation, these custom web services are packaged in **inventory.ear**.

4. After a new instance is created with the new image, verify the updated custom web services from the WebLogic console under **oracle.communications.inventory**.

Updating application.xml

Add the following information to the `<module>` element to identify the following for the custom web service:

- The WAR file name, such as ReferenceUim.war
- The WSDL file prefix, such as ReferenceUim

> **Note**
>
> Add the `<web-uri>` element for the WAR file name and the `<context-root>` element for the WSDL name, as shown in Example.

The sample is as follows:

```
<!-- Custom Web Service WAR -->
<module>
<web>
<web-uri>ReferenceUim.war</web-uri>
<context-root>ReferenceUim</context-root>
</web>
</module>
```

Updating inventory-clusterPlan.xml

You can find the inventory-clusterPlan.xml from the **CUSTOMFOLDER\custom\plans** directory.

To secure the custom webservice, update the deployment plan with required policies. You can use the security policy that comes with the UIM instance **Auth.xml**, the security policy that comes in the Reference Web Service ZIP file **SampleAuth.xml**, or create your own security policy file. Custom policies are available in the **<custom webservice war>/WEB-INF/policies** folder. The following example shows an update to inventory-**clusterPlan.xml** with default **Auth.xml** policy:

```
<module-override>
  <module-name>ReferenceUim.war</module-name>
  <module-type>war</module-type>
  <module-descriptor external="false">
    <root-element>weblogic-web-app</root-element>
    <uri>WEB-INF/weblogic.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>web-app</root-element>
    <uri>WEB-INF/web.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>weblogic-webservices</root-element>
    <uri>WEB-INF/weblogic-webservices.xml</uri>
```

```
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>webservices</root-element>
   <uri>WEB-INF/webservices.xml</uri>
  </module-descriptor>
  <module-descriptor external="false">
    <root-element>webservice-policy-ref</root-element>
    <uri>WEB-INF/weblogic-webservices-policy.xml</uri>
    <variable-assignment>
      <name>WsPolicy_policy:Auth.xml_Direction_13075993400140</name>
      <xpath>/webservice-policy-ref/port-policy/[port-
name="UIMReferenceUimHTTPPort"]/ws-policy/[uri="policy:Auth.xml"]/direction</
xpath>
    </variable-assignment>
    <variable-assignment>
      <name>WsPolicy_policy:Auth.xml_Direction_13075993400140</name>
      <xpath>/webservice-policy-ref/port-policy/[port-
name="UIMReferenceUimJMSPort"]/ws-policy/[uri="policy:Auth.xml"]/direction</
xpath>
    </variable-assignment>
  </module-descriptor>
</module-override>
```

Adding Third-party Libraries

To add third-party libraries:

1. Copy third-party libraries to **CUSTOMFOLDER\custom\libraries** folder.

   > **Note**
   >
   > The CUSTOMFOLDER is the parameter for the path where the custom folder of **uim-app-archive.zip** is copied and modified for the customization.

2. After a new instance is created with the new image, verify that the third-party libraries are available in the pod in the path **/UIM/lib**, by running the following command:

   ```
   kubectl exec <project>-<instance>-ms1 -n <project> -- ls /UIM/lib
   ```

Adding WebLogic Deployable Applications

To add WebLogic deployable applications, you can copy WebLogic deployable applications to **CUSTOMFOLDER\custom\applications** folder.

> **Note**
>
> The CUSTOMFOLDER is the parameter for the path where the custom folder of **uim-app-archive.zip** is copied and modified for the customization.

You can also add WebLogic deployable applications using extensions. This mechanism keeps the ear file together with the domain configuration in one location. This is best suited to applications that can be considered standard or fixed for all variants of a domain that are

required (test, development, and production). This is best suited in configuring MapViewer in the UIM cloud native instance.

The default WDT model for these applications is constructed while running the customization script as follows:

```
appDeployments:
  Application:
    '<application-name>':
      SourcePath: 'wlsdeploy/applications/custom/<application-name>.ear'
      ModuleType: ear
      StagingMode: nostage
      PlanStagingMode: nostage
      Target: '@@PROP:CLUSTER_NAME@@'
```

See "Deploying Entities to a UIM WebLogic Domain" for more information deploying the custom applications using extensions.

Adding Solution Cartridge Customizations

To add solution cartridge customizations, copy all cartridges to CARTRIDGEDIR or **$CUSTOMFOLDER/custom/cartridges**. You need to run the customization script that scans and packages the configuration files, images, and java ruleset codes into the container image and scans the cartridge jar files and extracts them to **$CARTRIDGES_PATH/unpack**.

Configuration Files

The configuration files are packaged into a layered image. After a new instance is created with the new image, verify the configuration files in the pod, by running the following command:

```
kubectl exec <project>-<instance>-ms1 -n <project> -- ls /UIM/config
```

Images

The image files are packaged into the base image. After a new instance is created with the new image, verify the images files in the pod, by running the following command:

```
kubectl exec <project>-<instance>-ms1 -n <project> -- ls /UIM/images
```

Localization

The localization changes are packaged into the base image. Verify the localization in UIM cloud native instance with newly generated image by changing the browser settings. See "Overview" in *UIM Developer's Guide* to build cartridges with localization files.

Custom Library or Java Ruleset Code

The Java ruleset code is packaged into a layered image. If ***Lib.jar** or ***aop.jar** exists in the extracted folder of **$CARTRIDGES_PATH/unpack**, then **uim_custom_lib.ear** is updated in **uim-custom-archive.zip** file.

The Java ruleset code is packaged into layered image as follows:

Table 3-1 Custom Library in Layered Images

Customization Type	CUSTOMFOLDER	CARTRIDGESDIR	Layered Image
User interface	The customized **inv.war** is placed in **$CUSTOMFOLDER\custom\ui_customization**.	NA	No layered image generated.
Custom Webservice	The customized war files are placed in **$CUSTOMFOLDER\custom\customWS**.	NA	No layered image generated.
Libraries	Third-party libraries are placed in **$CUSTOMFOLDER\custom\libraries**.	NA	Layered image is generated.
Applications	Place Weblogic Deployable applications in **$CUSTOMFOLDER\custom\applications**.	NA	Layered image is generated.
Solution cartridges	Optional if CARTRIDGEDIR is exported. Solution Cartridge jars are placed in **$CUSTOMFOLDER/custom/cartridges**.	Optional if CUSTOMFOLDER is exported. Solution Cartridge jar files are placed in $CARTRIDGESDIR.	Layered image generated for configuration files and Java Ruleset Code. Layered image not generated for images and localization.

Extending Entity Life Cycles

You can extend entity life cycles in UIM cloud native environment. See "Extending Life Cycles" in *UIM Developer's Guide* for more information on extending life cycles. The image builder tool kit includes the latest **customizations.sh** file that enables life cycle extensions support.

To set up your cloud native deployment for extending entity life cycles:

1. From **uim-app-archive.zip**, copy the **inventory-adapter.ear** and **core_lib.ear** files to a temporary location. For example: **D:/workspace/tmp**.
2. In Design Studio:
   a. Rename **COMPUTERNAME.properties** to match with the name of your computer and update its content to match with your environment.
   b. Update the **EAR_PATH** and **CORE_LIB_DIR** path values with the paths of the temporary location where you copied the **inventory-adapter.ear** and **core_lib.ear** files. The following text shows changes to the computer properties:

```
APP_NAME=inventory

JDK_HOME=C:/Program Files/Java/jdk-21.0.6
JAVA_HOME=${JDK_HOME}
ANT_HOME=C:/software/apache-ant-1.9.2

EAR_BUILD_DIR=generated

#location of the UIM home from the installer setup.
UIM_HOME=/weblogicDomainHome/UIM
```

```
DB_HOME=C:/app/orcluser/product/14.1.2/dbhome_1
DB_DRIVER=${DB_HOME}/jdbc/lib/ojdbc11.jar
DATABASE=oracle

#project home location this has to change based on the project location.
PROJECT_HOME=D:/Eclipse_Photon/UIM742/ora_uim_entity_sdk/src

#POMS location
POMS_SRCHOME=${PROJECT_HOME}/platformFiles/extract/objmgmt/poms
# POMS extract location.
POMS_ROOT=platformFiles/extract

APP_LIB=D:/tmp/UIM_SDK/lib  ### Path to UIM_SDK lib

#inventory-adapter.ear
EAR_PATH=D:/workspace/tmp

CLASSPATH=${JDK_HOME}/lib/tools.jar;${JDK_HOME}/jre/lib/rt.jar;$
{DB_DRIVER}

#This is required for finding the path for uim_core_lib.ear
CORE_LIB_DIR=D:/workspace/tmp
```

> **Note**
>
> Both **EAR_PATH** and **CORE_LIB_DIR** can have the same path values.

c. (Optional) Follow the steps described in the *Customizing Service Lifecycle to Introduce a New State*. (Doc ID 1918850.1) knowledge article on My Oracle Support to customize entity life cycles.

d. (Optional) Save the **build.xml** file.

e. Add any new custom metadata files to define the new custom state and build the project.

The build updates the .ear files that you copied to your local folder.

> **Note**
>
> During the build process, **uim-entities.jar** is modified. This **.jar** file contains the entity Java classes for all UIM entities and all custom entities. After the build, the **inventory-adapter.ear** and **uim_core_lib.ear** files in the **tmp** folder are updated along with the **uim_entities.jar** file.

3. Copy the updated **inventory-adapter.ear** and **uim_core_lib.ear** files from the **tmp** folder to **$CUSTOMFOLDER\custom\staticExtensions** folder.

> **Note**
>
> **CUSTOMFOLDER** is the parameter for the path where **uim-app-archive.zip** is copied and modified for customization.

4. Under the **model/content/product_home/config/resources/logging** directory of the **ora_uim_localization_reference** project, modify the **status.properties** file to display the new state on the UI.

> **Note**
>
> The UI-specific properties files are located in the **ora_uim_localization_reference** project, under the **model/content/product_home/config/resources/logging** directory. See "Overview" in *UIM Developer's Guide* for localizing the UI-specific files.

5. Add solution cartridge customizations to package the cartridge.

   See "Adding Solution Cartridge Customizations" for more information.

4

Creating a Basic UIM Cloud Native Instance

This chapter describes how to create a basic UIM cloud native instance in your cloud environment using the operational scripts and the base UIM configuration provided in the UIM cloud native toolkit. You can create a UIM instance quickly in order to become familiar with the process, explore the configuration, and structure your own project. This procedure is intended to validate that you are able to create a basic UIM instance in your environment. For information on creating your own project with custom configuration, see "Creating Your Own UIM Cloud Native Instance".

Before you can create a UIM instance, you must do the following:

- Download and extract the Common cloud native toolkit (COMMON CNTK) archive file.
- Assemble the specifications.
- Install WebLogic Operator. For more information, see "Installing the WebLogic Kubernetes Operator Container Image".
- Install Ingress Controller. For more information, see "About Load Balancing and Ingress Controller".

Installing the UIM Cloud Native Artifacts and the Toolkit

Build container images for the following using the UIM cloud native Image Builder:

- UIM core application
- UIM database installer

You must create a private Docker repository for these images, ensuring that all nodes in the cluster have access to the repository. See "About Container Image Management" for more information.

Download the Common cloud native toolkit archive and do the following:

- **On Oracle Linux**: Where Kubernetes is hosted on Oracle Linux, download and extract the tar archive to each host that has connectivity to the Kubernetes cluster.
- **On OKE**: For an environment where Kubernetes is running in OKE, extract the contents of the tar archive on each OKE client host. The OKE client host is the bastion host/s that is set up to communicate with the OKE cluster.

Assembling the Specifications

You must assemble the specification files to use the Common cloud native toolkit. To assemble the specification files:

1. Extract the archive.
2. Set up the environment.
3. Prepare the configuration.

Run the following command to assemble specification:

```
#Extract common-cntk
cd workspace
tar -xvf common-cntk.tar.gz

#Create spec directory
mkdir spec_dir

#Export variables
export COMMON_CNTK=(path_to_workspace)/common-cntk
export SPEC_PATH=(path_to_workspace)/spec_dir
export STRIMZI_NS=strimzi   #this is strimzi namespace for message bus, if
not applicable to you can keep default value `strimzi`

#Run assemble specification
$COMMON_CNTK/scripts/assemble-specifications.sh -p project -i instance -
s $SPEC_PATH
```

After assembling the specifications successfully without any errors, verify if the configuration files are correctly assembled by checking the contents of the $SPEC_PATH directory.

> **Note**
>
> All scripts in **common-cntk** are designed to read the required configuration files from the path specified by $SPEC_PATH.

Installing WebLogic Kubernetes Operator (WKO) and Ingress Controller

In a shared environment, multiple developers may create UIM instances in the same cluster, using a shared WebLogic Kubernetes Operator.

> **Note**
>
> There can be multiple operators in a Kubernetes cluster, and in that case, you must ensure that the namespaces managed by these operators do not overlap. A namespace can be managed by one operator.

For each cluster in your environment, you download and install the following:

- WebLogic Kubernetes Operator (WKO) application deployment.
- Ingress Controller deployment

Before installing the WKO and the Ingress Controller, do the following tasks:

- Remove the instances of the WKO and Ingress Controller that you installed to validate your cloud environment.

- Ensure that you have cleaned up the environment. See "Validating Your Cloud Environment" for instructions on cleaning up.
- Ensure that there are no WebLogic Server Operator artifacts in the environment.

Installing the WebLogic Kubernetes Operator Container Image

UIM cloud native package does not provide scripts to install or remove the WebLogic Kubernetes operator. See WKO Documentation for Installing and uninstalling Weblogic Operator.

See "UIM Software Compatibility" in *UIM Compatibility Matrix* for WKO recommended version.

For example, to download and install the WKO version:

1. See https://github.com/oracle/weblogic-kubernetes-operator/releases/.
2. Choose a namespace for the operator and set the `WLSKO_NS` environment variable to the Kubernetes namespace in which WKO will be deployed.

> **Note**
>
> Oracle recommends you use **--version=<*version*>** while installing. See "UIM Software Compatibility" *UIM Compatibility Matrix* for the corresponding WKO version.
>
> Oracle recommends you set the label to the same as namespace using `--set "domainNamespaceLabelSelector=<namespace>=enabled"` instead of the default label `"weblogic-operator=enabled"` as having multiple operators installed with the same label is not recommended.

3. After successful installation of WKO, validate that the operator is installed by running the following command:

```
kubectl get pods -n $WLSKO_NS
```

If you are using a version older than 3.1.0, the operator is supported to specify the namespaces that can be managed only through a list. Currently, the operator supports a list of namespaces, a label selector, or a regular expression matching namespace names.

> **Note**
>
> If you are upgrading from UIM CN 7.5.1 and an older version of the WebLogic Operator, see "Upgrading the UIM Cloud Native Environment" for more information.

Installing the Ingress Controller

You can use any Ingress Controller that conforms to the standard Kubernetes ingress API and that supports annotations required for UIM. See "About Load Balancing and Ingress Controller" and "Working with Ingress, Ingress Controller, and External Load Balancer" for more information.

> **Note**
>
> - Oracle does not certify individual Ingress Controllers to confirm this generic compatibility.
> - When selecting an Ingress Controller, you must validate if that supports the following requirements through Ingress annotations:
>   - Sticky sessions (session affinity) and cookie-based persistence.
>   - Upstream request header changes to:
>   * Clear `WL-Proxy-Client-IP` and `WL-Proxy-SSL`
>   * Set `X-Forwarded-Proto: https , WL-Proxy-SSL: true` When SSL Terminate on UIM cloud native.
>   - SSL passthrough, if external clients must connect to the Message Bus from outside of the cluster.

For information on the installation and the usage of the HAProxy Ingress controller. See About Load Balancing and Ingress Controller. To know the supported WebLogic version, see UIM Cloud Native Deployment Software Compatibility.

Creating a Basic UIM Instance

This section describes how to create a basic UIM instance. In this section, while creating a basic instance, the project name is considered as "sr" and instance name is considered as "quick".

Setting Environment Variables

UIM cloud native relies on access to certain environment variables to run seamlessly. Ensure the following variables are set in your environment:

- Path to your specification directory.
- Path to your Common cloud native toolkit directory.
- Path to your WebLogic operator namespace.

To set the environment variables:

1. Ensure that you have copied the corresponding files to the specification directory as mentioned in "Assembling the Specifications".

> **Note**
>
> You must provide full path of specification directory for **SPEC_PATH** variable as follows:
>
> ```
> $ export SPEC_PATH=<path to workspec>/spec_dir
> ```

2. Set the **COMMON_CNTK** variable for **common-cntk** directory path as follows:

   ```
   $ export COMMON_CNTK=<path to workspace>/common-cntk
   ```

3. Set the **WLSKO_NS** variable for WebLogic operator namespace as follows:

   ```
   $ export WLSKO_NS = wlsko namespace
   ```

Registering the Namespace

After you set the environment variables, register the namespace. If you are working with `wlsko` as the `targetNamespace`, then **RegisterNamespace** script offers an additional `-l` option that enables you to include the label selector used during the operator installation.

If a label selector is not added while installing the operator:

By default the `weblogic-enabled=true` label is added to your `$WLSKO_NS` namespace so that the operator can monitor it.

If a label selector is added while installing the operator:

Ensure that you include the same label using the `-l` option, as follows.

To register the namespace for **wlsko**, run the following command:

```
#if you have defined labelselector while installing operator Example:
wlsko=enabled
$COMMON_CNTK/scripts/register-namespace.sh -p <project> -t targets -l <label-
Selector>
#For example, $COMMON_CNTK/scripts/register-namespace.sh -p sr -t wlsko -l
wlsko=enabled
```

> **Note**
>
> - **wlsko** is the name of the targets for registering the namespace. The script uses **WLSKO_NS** to locate these targets.
> - For Generic IngressController, the registration of namespace is not required. To select the ingress controller, you need to provide the `ingressClassName` value under the **ingress.className** field in the **applications-base.yaml** file.
> - For more information about **ingressClassName**, see https://kubernetes.io/docs/concepts/services-networking/ingress/.

Creating Secrets

You must store sensitive data and credential information in the form of Kubernetes Secrets that the scripts and Helm charts in the toolkit consume. Managing secrets is out of the scope of the toolkit and must be implemented while adhering to your organization's corporate policies. Additionally, UIM cloud native does not establish password policies.

> **Note**
>
> The passwords and other input data such as RCU schema prefix length that you provide must adhere to the policies specified by the appropriate component.

As a prerequisite to use the toolkit for either installing the UIM database or creating a UIM instance, you must create secrets to access the following:

- UIM database
- RCU DB
- OPSS
- Operator artifacts for the instance
- WebLogic Server Admin (credentials used while creating the domain)

The toolkit provides sample scripts for this purpose. However, they are not pipeline-friendly. The scripts should be used for creating an instance manually and quickly, but not for any automated process for creating instances. The scripts also illustrate both the naming of the secret and the layout of the data within the secret that UIM cloud native requires. You must create secrets prior to running the **install-database.sh** or **create-applications.sh** scripts.

Ensure that you have assembled the specification before you the following script.

Run the following script to create the required secrets:

```
$COMMON_CNTK/scripts/manage-app-credentials.sh -p sr -i quick -s $SPEC_PATH -
a uim create wlsadmin,opssWP,wlsRTE,rcudb,uimdb
```

where:

- `uimdb` specifies the connectivity details and the credentials for connecting to the UIM PDB (UIM schema). This is consumed by the UIM DB installer and UIM runtime.

  > **Note**
  >
  > The uimdb secrets contain PDB sysdba user and uim main schema user. The names of these must be unique.

- `rcudb` specifies the connectivity details and the credentials for connecting to the UIM PDB (RCU schema). This is consumed by the UIM database installer and UIM and Fusion MiddleWare runtime.
- `wlsadmin` is the credential for the intended user that will be created with administrative access to the WebLogic domain.
- `opssWP` is the password for encrypting and decrypting the ewallet contents.
- `wlsRTE` is the password used to encrypt the operator artifacts for this instance. The merged domain model and the domain ZIP are available in the operator config map and are encoded using this password.

Verify that the following secrets are created:

```
sr-quick-database-credentials
sr-quick-embedded-ldap-credentials
```

```
sr-quick-weblogic-credentials
sr-quick-rcudb-credentials
sr-quick-opss-wallet-password-secret
sr-quick-runtime-encryption-secret
```

Additionally, the secret `opssWF` is created by the installation process and does not follow the same guidelines. It is therefore not a pre-requisite for creating a new instance. In scenarios where a database is being re-used for a different UIM instance, then this becomes a pre-requisite secret. For more details, see "Reusing the Database State".

Creating Secrets for LDAP System Users

To create secrets for LDAP system users:

1. Create a user information file, for example: **ldap_users.txt**, with the list of users and groups as follows:

   ```
   uim:uimadminuser:secret:uim-users
   uim:uimcmwsuser:secret:Administrators,Cartridge_Management_WebService
   uim:uimmetricsuser:secret:uim-metrics-users
   ```

2. Run the following command to create the required secrets for embedded LDAP system users:

   ```
   $COMMON_CNTK/samples/credentials/manage-uim-credentials.sh -p sr -i quick -
   c create -f "location to ldap_users.txt"
   ```

Installing the UIM and RCU Schemas

This procedure configures an empty PDB. Depending on the database strategy for your team, you may have already performed this procedure as described in "Planning Your Cloud Native Environment". Before continuing, confirm whether the PDB being used for creating the UIM instance has been cloned from a Master PDB that includes the schema installation. If the PDB already has the schema installed, skip this procedure and proceed to the Creating UIM Users and Groups topic.

After the PDB is created, it is configured with the UIM schema, the RCU schema, and the cluster leasing table.

To install the UIM and RCU schemas:

> **Note**
>
> YAML formatting is case-sensitive. While the next step uses vi editor for editing, if you are not familiar with editing YAML files, use a YAML editor to ensure that you do not make any syntax errors while editing. Follow the indentation guidelines for YAML, as incorrect spacing can lead to errors.

1. Edit the database.yaml specification file and update the DB installer image to point to the location of your image as shown below:

> **Note**
>
> Before changing the default values provided in the specification file, confirm that they align with the values used during PDB creation. For example, the default tablespace name should match the value used when PDB is created.

```
uim-dbinstaller:
  dbinstaller:
    image:
      name: "uim-cn-db-installer"
      tag: "latest"
```

2. If your environment requires a password to download the container images from your repository, create a Kubernetes secret with the Docker pull credentials. See the "Kubernetes documentation" for details. Refer the secret name in the **database.yaml** specification file.

```
# The image pull access credentials for the "docker login" into Docker
repository, as a Kubernetes secret.
# Uncomment and set if required.
# imagePullSecret: ""
```

3. Configure the tablespace details. By default, the `defaultTablespace` and `tempTablespace` is **SYSTEM** and **TEMP** respectively as follows:

```
db:
    defaultTablespace: "SYSTEM"
    tempTablespace: "TEMP"
```

4. Run the following script to start the UIM DB installer, which instantiates a Kubernetes Pod resource. The pod resource lives until the DB installation operation completes.

```
#(UIM Schema)
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 1

## once finished #(RCU Schema)
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 2
```

   You can invoke the script with `-h` to see the available options.

5. Check the console to see if the DB installer is installed successfully.
6. If the installation failed, run the following command to review the error message in the log:

```
kubectl logs -n sr sr-quick-dbinstaller
```

7. Clean up the failed pod by running the following command:

```
helm uninstall sr-quick-dbinstaller -n sr
```

8. Go back to step 4 and run the script again to install the UIM DB installer.

Generating Encrypted WebLogic Administrator's Password

To generate encrypted WebLogic administrator's password:

1. Run the following command to start the UIM DB installer, which initiates a Kubernetes Pod resource. The pod resource is available until the DB installation completes.

   ```
   $COMMON_CNTK/scripts/install-database.sh -p project -i instance -
   s $SPEC_PATH -a uim -c 8
   ```

   > **Note**
   >
   > To see further options available, use `-h` command.

2. Check the WebLogic console for successful DB installation. If the installation failed, run the following command to view the error message from the log:

   ```
   kubectl logs -n sr sr-quick-dbinstaller
   ```

3. Clean up the failed pod using the following command:

   ```
   helm uninstall sr-quick-dbinstaller -n sr
   ```

4. Verify the new secret that is created as follows:

   ```
   kubectl get secrets -n sr | grep encrypted
   sr-quick-weblogic-encrypted-credentials Opaque 1 5m32s
   ```

Configuring the Specification Files

Ensure that you have assembled the specification, and you have to edit and provide the values for the files in **$SPEC_PATH** location.

To configure the base specification, edit **$SPEC_PATH/project/instance/applications-base.yaml**:

1. Provide the **ingressController** details as follows for HAProxy ingress controller, if you are using any other ingress controller, provide the corresponding details:

   ```
   ingressController: "GENERIC"
   ingress:
     #provide appropriate ingressClass for controller, “haproxy” is default
   for Haproxy ingressController.
     className: "haproxy"
     annotations:
       haproxy.org/cookie-persistence: "uimhaproxycookie"
   ```

2. If your environment requires a password to download the container images from your repository, create a Kubernetes secret with the Docker pull credentials. For more

information, see Kubernetes documentation. Check the secret name in the project specification.

```
# The image pull access credentials for the "docker login" into Docker
repository, as a Kubernetes secret.
# uncomment and set if required.
#imagePullSecret:
# imagePullSecrets:
# -name: <image-pull-secret>
```

3. For your DNS resolution mechanism, change the default load balancer sub domain name as per your requirement:

```
hostSuffix: "uim.org"
```

4. Select the shape to be used. **shape** is the file that contains the resources for UIM service and must present at location **$SPEC_PATH/project/instance/shape** directory:

```
shape: dev
```

5. If external load balancer is used, provide the **loadbalancerport** of load balancer. Else, provide the HAProxy Ingress Controller NodePort as **loadBalancerPort**. **30505** is the default value of non-ssl NodePort of HAProxy in the sample files:

```
loadbalancerport: 30505
```

To configure the app specification, edit **$SPEC_PATH/project/instance/app-uim.yaml**:

1. Provide the image in your repository (name and tag):

```
** edit the image to reflect the UIM image name and location in your
docker repository
uim:
   image:
     name: "uim-cn-base"
     tag: "latest"
```

2. Provide the list of inventory users. All these users should be present in **project-instance-uimcn-cred-uim** secret:

```
inventoryUsers:
  - uimdev
  - cmwsdev
```

3. Specify the database details to enable **MultiDataSource** and **ServerAffinity**:

```
db:
datasourcesPrimary:
  port: 1521
  rcuPort: 1521
  # Below boolean is to enable server affinity, by default it is false.
  serverAffinityEnabled: false
  # Below boolean is to enable RAC DB Setup, by default it is false.
  multiDataSourceEnabled: false
  # If using RAC,enable multiDataSourceEnabled and provide list of SCAN
```

```
hostname/IP addresses
  # If not using RAC, comment it out "#uimScans:"
  #uimScans:
  # - scan1-ip
  # - scan2-ip
  #
  # If using RAC, provide either a list of VIP hostname/IP addresses
  # or a list of INSTANCE_NAMES
  # If not using RAC, comment these out "#uimVips:" and "#uimInstances:"
  #
  #uimVips:
  # - vip1-ip
  # - vip2-ip
  # --- OR ---
  #uimInstances:
  # - instance-1
  # - instance-2
  # By default rcu and uim schema are colocated, if we have different rcu
schema and different uim then below RAC details are to be provided.
  # If not provided it will not be considered as RAC.
rcuUimSchemaCoLocated: true
  # RCU db information
  # If using RAC, provide list of SCAN hostname/IP addresses
  # If not using RAC, comment it out "#rcuScans:"
  #rcuScans:
  # - scan1-ip
  # - scan2-ip
  #
  # If using RAC, provide either a list of VIP hostname/IP addresses
  # or a list of INSTANCE_NAMES
  # If not using RAC, comment these out "#rcuVips:" and "#rcuInstances:"
  #
  #rcuVips:
  # - vip1-ip
  # - vip2-ip
  # --- OR ---
  #rcuInstances:
  # - instance-1
  # - instance-2
```

Creating an Ingress

An ingress establishes connectivity to the UIM instances.

To create an Ingress, run the following command:

```
$COMMON_CNTK/scripts/create-ingress.sh -p sr -i quick -s $SPEC_PATH -a uim
Project Namespace : sr
Instance Fullname : sr-quick
LB_HOST           : quick.sr.uim.org
Ingress Controller: GENERIC
External LB IP    : 192.0.0.8

NAME: sr-quick-ingress
```

```
LAST DEPLOYED: Wed Jul  1 10:20:27 2020
NAMESPACE: sr
STATUS: deployed
REVISION: 1
TEST SUITE: None

Ingress created successfully...
```

Creating a UIM Instance

This procedure describes how to create a UIM instance in your environment using the scripts that are provided with the toolkit.

To create a UIM instance:

1. Run the following command:

   ```
   $COMMON_CNTK/scripts/create-applications.sh -p sr -i quick -s $SPEC_PATH -
   a uim
   ```

   The **create-applications.sh** script uses the Helm chart located in the **charts/uim-app** directory to create and deploy the domain custom resource and the domain config map for your instance. If the scripts fails, see the **Troubleshooting Issues** section at the end of this topic, before you make additional attempts.

   The instance creation process creates the opssWF secret, which is required for access to the RCU DB. It is possible to handle the wallet manually if needed. To do so, pass `-w` to the **create-applications.sh** script, which creates the wallet file at a location you choose. You can then use this wallet file to create a secret by using the manage instance credentials script.

2. Validate the important input details such as Image name and tag, specification files used (Values Applied), hostname, and port for ingress routing:

   ```
   $COMMON_CNTK/scripts/create-applications.sh -p sr -i quick -s $SPEC_PATH -
   a uim
   Calling helm lint==> Linting /scratch/sthatipa/cloudlab/uim-cntk/
   scripts/../charts/uim
   [INFO] Chart.yaml: icon is recommended
   1 chart(s) linted, 0 chart(s) failed
   Project Namespace : sr
   Instance Fullname : sr-quick
   LB_HOST : quick.sr.uim.org
   LB_PORT : 30505
   Image : uim-cn-base:7.5.1.0.1
   Shape : dev
   Values  Applied :
   Output wallet : n/a
   ```

   After the script finishes running, the log shows the following:

   ```
   NAME             READY   STATUS              RESTARTS   AGE
   sr-quick-admin   1/1     Running             0          2m12s
   sr-quick-ms1     0/1     ContainerCreating   0          1s
   ```

```
Provide opss wallet File for 'sr-quick' ...
For example : '/path-to-uim-cntk/sr-quick.ewallet'
opss wallet File:
secret/sr-quick-opss-walletfile-secret created

Instance 'sr/sr-quick' admin server is now running.
Creation of instance 'sr/sr-quick' has completed successfully.
```

The **create-applications.sh** script also provides some useful commands and configuration to inspect the instance and access it for use.

> **Note**
>
> While creating an instance on new RCU Schema, provide *-w* option to save OPSS eWallet data of instance. By default, this data is pushed into the **opssWF** secret automatically.

3. If you query the status of the pods, the **READY** state of the managed servers may display **0/1** for several minutes while the UIM application is starting.
   When the **READY** state shows **1/1**, your UIM instance is up and running. You can validate the instance by deploying UIM base cartridges and creating Custom Object instance in the UIM Home page.

The base hostname required to access this instance using HTTP is `quick.sr.uim.org`. See "Planning Your Cloud Native Environment" for details about hostname resolution.

The **create-applications** script prints out the following valuable information that you can use while working with your UIM domain:

- The T3 URL: `http://t3.quick.sr.uim.org:30505` This is required for external client applications such as JMS and WLST.
- Log in to the WebLogic Remote Console by:
  1. Open WebLogic Remote Console Application application.
  2. Choose startup task as **Add Admin Server Connection Provider**.
  3. Enter URL as follows:

     ```
     http://ServerName:Port
     ```

  4. Enter the WebLogic server administration user name and password.
- The URL for access to the UIM UIs, which is provided through the ingress controller that requires the host to be specified as: `http://quick.sr.uim.org:30505/Inventory/Login.jsp`.

Assigning Roles

To access UIM Home page, you need to assign roles. You can use EM console for role assignments.

To assign roles:

1. Create **uim-user-roles.txt** as follows:

   ```
   uimdev:uimuser
   cmwsdev:uimuser
   ```

2. Run the following command to assign the roles:

   ```
   $COMMON_CNTK/samples/credentials/assign-role.sh -p sr -i quick -f uim-user-roles.txt
   ```

Validating the UIM Instance

After creating an instance, you can validate it by checking the domain configuration and the client UIs.

Run the following command to display the domain configuration details of the UIM instance that you have created:

```
kubectl describe domain sr-quick -n sr
```

The command displays the domain configuration information.

To verify the client UIs:

- Log into the WebLogic console using the URL specified in the output of the **create-instance** script: http://admin.quick.sr.uim.org:30505/console

  You can use the console to verify the configuration that has been applied and to see that the UIM application is in an active state.
- Log into the UIM Task Web client user interface with the UIM administrator login credentials created as part of "Creating Secrets" using the URL (http://quick.sr.uim.org:30505/Inventory/Login.jsp) specified in the output of the **create-applications** script.

> **Note**
>
> After a UIM instance is created, it may take a few minutes for the UIM user interface to become active.

Scaling the UIM Application Cluster

Now that your UIM shape is up and running, you can explore the ability to dynamically scale the application cluster. Update the shape directory name that you have provided.

To scale the UIM application cluster, edit the configuration:

1. In the shape specification directory mentioned in **applications-base.yaml**, change the value for `clusterSize` in **uim.yaml** file manually. This change would ultimately be performed by an automated CI/CD pipeline.

   ```
   vi $SPEC_PATH/sr/quick/shapes/dev/uim.yaml

   # Change the cluster size to a value not larger than 18
   ```

```
#cluster size
clusterSize: 2
```

> **Note**
>
> You can watch the Kubernetes pods in your namespace shrink or grow in real-time. To watch the pods shrink or grow, in a separate terminal window, run the following command:
>
> ```
> kubectl get pods -n sr --watch
> ```

2. Upgrade the deployed Helm release:

   ```
   $COMMON_CNTK/scripts/upgrade-applications.sh -p sr -i quick -s $SPEC_PATH -
   a uim
   ```

   This pushes the new configuration to the deployed Helm release so the operator can take the necessary steps.

The WebLogic operator monitors changes to `clusterSize` and results in the operator spinning up or tearing down managed servers to align with the requested cluster size.

Deleting and Recreating Your UIM Instance

Deleting Your UIM Instance

To delete your UIM instance, run the following command:

```
$COMMON_CNTK/scripts/delete-applications.sh -p sr -i quick -s $SPEC_PATH -a
uim
```

Re-creating Your UIM Instance

When you delete a UIM instance, the database state for that instance still remains unaffected. You can re-create a UIM instance with the same project and the instance names, pointing to the same database.

> **Note**
>
> Ensure that you use the same specifications that you used for creating the instance and that the following secrets have not been deleted:
>
> - uimdb
> - rcudb
> - opssWF
> - opssWP
> - wlsRTE

To re-create a UIM instance, run the following command:

```
$COMMON_CNTK/scripts/create-applications.sh -p sr -i quick -s $SPEC_PATH -a
uim
```

> **Note**
>
> After re-creating an instance, client applications such as SoapUI and HermesJMS may need to be restarted to avoid using expired cache information.

Cleaning Up the Environment

To clean up the environment:

1. Delete the instance:

   ```
   $COMMON_CNTK/scripts/delete-applications.sh -p sr -i quick -s $SPEC_PATH -
   a uim
   ```

2. Delete the ingress:

   ```
   $COMMON_CNTK/scripts/delete-ingress.sh -p sr -i quick -s $SPEC_PATH -a uim
   ```

3. Delete the namespace, which in turn deletes the Kubernetes namespace and the secrets:

   ```
   $COMMON_CNTK/scripts/unregister-namespace.sh -p sr -d -t targets
   ```

   > **Note**
   >
   > `wlsko` is the name of the target for registration of the namespace. The script uses WLSKO_NS to find the target.

4. (Optional) To stop WebLogic Operator from monitoring the namespace, unregister it:

   - Ensure that you provide the `labelselector` you used while registering the namespace with `wlsko`. For example, for `wko412=enabled`, use the following command for unregistering. You can describe the project namespace and check the label to see which label selector is used.

     ```
     $COMMON_CNTK/scripts/unregister-namespace.sh -p <project> -t wlsko -l
     label

     #For example,
     $COMMON_CNTK/scripts/unregister-namespace.sh -p sr -t wlsko -l wlsko412
     ```

     > **Note**
     >
     > You do not have to provide the complete label, which is `key=value`. You can provide only the key of the label.

5. Drop the PDB as follows:

```
# Drop UIM Schema
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 5

# Drop RCU Schema
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 6

# If RCU dropped, You can delete the secret
kubectl delete secret -n project project-instance-opss-walletfile-secret
```

Troubleshooting Issues with the Scripts

This section provides information about troubleshooting some issues that you may come across when running the scripts.

If you experience issues when running the scripts, do the following:

- Check the operator logs to find out the details about the issue:

```
kubectl get pods -n $WLSKO_NS
# get the operator pod name to be used in the next command
kubectl logs -n $WLSKO_NS operator_pod
```

- Check the "Status" section of the domain to see if there is useful information:

```
kubectl describe domain -n sr sr-quick
```

"Timeout" Issue

In the logs, you may sometimes see the word "timeout" when the **create-applications** script fails. When you run the **create-applications** script, it may take a long time to pull the image, if you are doing it for the first time. In such a scenario, the script may fail and display the text "timeout" in the log.

To resolve this issue, try increasing the `podStartupDeadlineSeconds` parameter. The `podStartupDeadlineSeconds` parameter is a configurable parameter exposed in the instance specification that can be increased if required. Start with a very high timeout value and then monitor the average time it takes, because it depends on the speed with which the images are downloaded and how busy your cluster is. Once you have a good idea of the average time, you can reduce the timeout value accordingly to something that considers both the average time and some buffer.

```
# Modify the timeout value to start introspector pod. Mainly
# when using against slow DB or pulling image first time.
podStartupDeadlineSeconds: 800
```

After adjusting the parameter, clean up the failed instance and re-create the instance.

Cleanup Failed Instance

When a **create-applications** script fails, you must clean up the instance before making another attempt at instance creation.

> **Note**
>
> Do not retry running the **create-applications** script or the **upgrade-applications** script immediately to fix any errors, as they would return errors. The **upgrade-applications** script may work but re-running it does not complete the operation.

To clean up the failed instance:

1. Delete the instance:

```
$COMMON_CNTK/scripts/delete-applications.sh -p sr -i quick -s $SPEC_PATH -
a uim
```

2. Delete and recreate the RCU schema:

```
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 5
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 2
```

Recreating an Instance

If you face issues when creating an instance, do not try to re-run the **create-applications.sh** script as this will fail. Instead, perform the cleanup activities and then run the following command:

```
$COMMON_CNTK/scripts/create-applications.sh -p sr -i quick -s $SPEC_PATH -a
uim
```

Next Steps

A basic UIM cloud native instance should now be running in your environment. This process exposed you to some of the base functionality and concepts that are new to UIM cloud native. You can continue in your sandbox environment learning about more UIM cloud native capabilities by following the learning path.

If, however, your first priority is to understand details on infrastructure setup and structuring of UIM instances for your organization, then you should follow the infrastructure path.

To follow the infrastructure path, proceed to "Planning Infrastructure".

To follow the learning path, proceed to "Creating Your Own UIM Cloud Native Instance".

5

Planning Infrastructure

In "Creating a Basic UIM Cloud Native Instance", you learned how to create a basic UIM instance in your cloud native environment. This chapter provides details about setting up infrastructure and structuring UIM instances for your organization. However, if you want to continue in your sandbox environment learning about more UIM cloud native capabilities, then proceed to "Creating Your Own UIM Cloud Native Instance".

See the following topics:

- Sizing Considerations
- Managing Configuration as Code
- Setting Up Automation
- Securing Operations in Kubernetes

Sizing Considerations

The hardware utilization for a UIM cloud native deployment is approximately the same as that of a UIM traditional deployment.

Consider the following when sizing for your cloud native deployment:

- Oracle recommends sizing using a given production shape as a building block, adjusting the UIM cluster size to meet target order volumes.
- In addition to planning hardware for a production instance, Oracle recommends planning for a Disaster Recovery size and key non-production instances to support functional, integration and performance tests The Disaster Recovery instance can be created against an Active Data Guard Standby database when needed and terminated when no longer needed to improve hardware utilization.
- Non-production instances can likewise be created when needed, either against new or existing database instances.

Contact Oracle Support for further assistance with sizing.

Managing Configuration as Code

Managing Configuration as Code involves the following tasks:

- Creating Source Control Repository
- Managing UIM instances
- Deciding on the Scope
- Deployment Considerations
- Creating an Instance Using the Repository

Creating Source Control Repository

Managing Configuration as Code (CAC) is a central tenet of using UIM cloud native. You must create a source control repository to store all configuration that is necessary to re-create instance (or PDB) if it is lost. This does not include the toolkit scripts.

You must also set up a Docker repository for the UIM and UIM DB Installer images, as well as any custom versions of the UIM image for your use cases. For example, custom images are required to deploy a custom application **.ear** file. For more details on custom images, see "Customizing Images".

Managing UIM Instances

To extract the full benefits of UIM cloud native, it is imperative that you consider the management of the UIM instances before making potential configuration changes. The sections that follow describe how to structure your repositories to group project level artifacts, while allowing for other artifacts to be re-used (if needed) by the multiple UIM instances within a project.

Example Scenario

This section describes a scenario to help illustrate the concepts.

Let us assume that in an organization, UIM is used for two business purposes each of which is handled by two separate teams. The first team uses UIM to orchestrate wire line (triple play) orders for residential customers, and a second team uses UIM to process carrier ethernet for enterprises.

Deciding on the Scope

You must first decide on the scope of the project including how many instances are required. Choose meaningful names for your project and instance.

The organization in our example will have two projects named **resiwireline** and **bizwireless**. We can assume that each project team has a predefined "pre-production" instance for final validation or production changes, a geo-redundant production instance for disaster recovery, a final User Acceptance Testing (UAT) instance for business testing, a few small Quality Assurance (QA) systems and many small development instances.

The directory structure for your configuration repository should reflect the hierarchical relationship of the project/instance relationship as well as isolating different projects from each other.

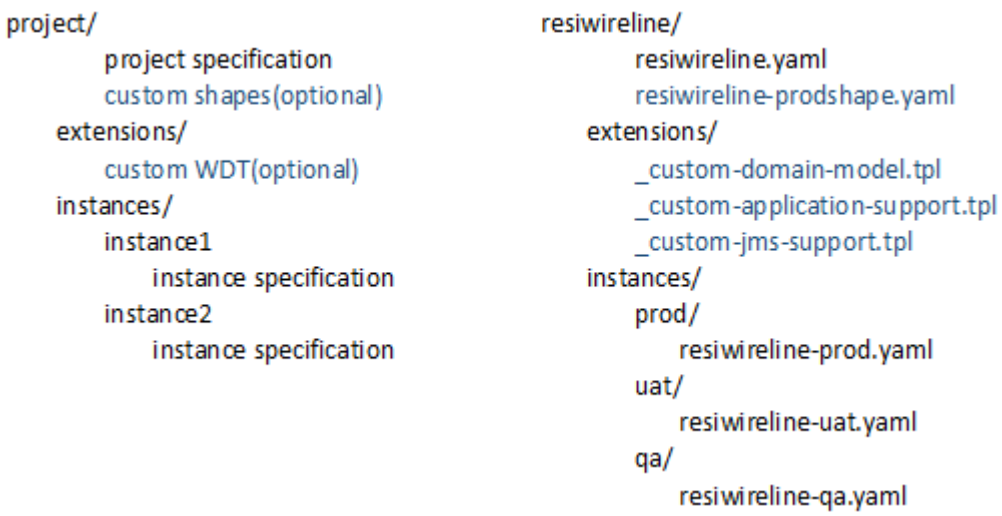
About the Repository Directory Structure

The project directory includes the instance directories as well as configuration that is common to all instances, whereas instance directory contains all individual specification files of an instance.

- Each project requires its own project specifications (YAML files).
- Optional artifacts such as the list of users that are located under the top level project directory.
- All artifacts under the project can be shared across the instances. Instance directories contain the individual specification files.

The following illustration shows the structure and hierarchy of the project directory with an example.

Figure 5-1 Project Directory Structure



Deployment Consideration

As the scenario shows, there will be many bits of configuration that may mix and match in different ways to produce a specific UIM instance. While all of these instances are pre-defined in the source control repository, they need not be deployed all the time.

Consider the following:

- For each project, one or more production instances may be deployed.
- It would be reasonable for pre-production to be deployed only when needed while first cloning the production DB.
- Likewise, the performance instance could also be deployed only when needed. Its PDB could be cloned from a specially generated PDB with synthetic test data, providing a consistent starting point.
- Likewise, the UAT instance could be deployed when needed, starting from similarly saved UAT PDB.
- The GR instance application would not be pre-deployed, but its database would be created in a DR site and synchronized from production via Active Data Guard.

Setting the Repository Path During Instance Creation

To offer flexibility in how the repository directory structure develops, the **create-applications.sh** script takes as input, the path to the specification files.

The **-s specPath** parameter is mandatory in **create-applications.sh**. The **specPath** structure should exactly match the structure that **assemble-specification** creates. Therefore, ensure that you run the script before performing any operation.

specPath would contain all the directories that contain specification files used for creating an instance:

- *repo*/**resiwireline**
- *repo*/**resiwireline/qa**:*repo*/**resiwireline/qa/shape**. This will include all specification files at the resiwireline project level, as well as the specification files in the **qa** instance directory.

Additionally, a separate parameter is used to point to the directory where custom extensions are found.

The **-m customExtPath** parameter is an optional parameter that can be passed into the **create-applications.sh** script.

customExtPath would point to all the directories where custom template files reside for the instance being created: *fileRepo*/**resiwireline/extensions**

Setting Up Automation

This section describes the complete sequence of activities for setting up a UIM environment with the aim of grouping repeatable steps into high-level categories. You should start to plan the steps that you can automate to some degree. This section does not include details on the changes that must be made to the specification files, which is described in "Creating a Basic UIM Instance".

> **Note**
>
> These steps exclude any one-time setup activities. For details on one-time setup activities, see the tasks you must do before creating a UIM instance in "Creating a Basic UIM Cloud Native Instance".

Where pre-requisite secrets are required, the toolkit provides sample scripts for this activity. However, the scripts are not pipeline-friendly. Use the scripts for manually standing up an instance quickly and not for any automated process for creating instances. These scripts are also important because they illustrate both the naming of the secret and the layout of the data within the secret that UIM cloud native requires. You must replace references to toolkit scripts for creating secrets with your own mechanism in your DevOps process.

Configuring Code for Creating a UIM Instance

To configure code for creating an instance, you assemble the configuration. While some of these activities could be automated, much of the work is manual in nature.

1. Assemble the configuration.
   To assemble the configuration:
   a. Run the following command to copy the default specification files to spec directory. It copies **applications-base.yaml**, **app-uim.yaml**, all config files, and shape files to provided **$SPEC_PATH** location:

      ```
      $COMMON_CNTK/scripts/assemble-specifications.sh -p project -i instance -
      s $SPEC_PATH
      ```

   b. Assemble the optional configuration files as needed. These files include custom WDT fragments and custom shapes for deployment.

2. Create pre-requisite secrets for UIM DB access, RCU DB access, UIM system users, OPSS, Introspector and the WLS Admin credentials used when creating the domain.

   ```
   $COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -
   s $SPEC_PATH -a uim create wlsadmin,opssWP,wlsRTE,rcudb,uimdb
   ```

   > **Note**
   >
   > Passwords and other secret input must adhere to the rules specified of the corresponding component.

3. Create custom secrets as required for creating UIM embedded LDAP users.

   ```
   $COMMON_CNTK/samples/credentials/manage-uim-credentials.sh -p project -i
   instance \
     -c create \
     -f user information file

   ** $COMMON_CNTK/samples/credentials/manage-uim-credentials.sh -h for help
   ```

4. Create other custom secrets as required by optional configuration.
5. Populate the embedded LDAP with all the fixed users in the *app-uim*.yaml file. During the creation of the UIM server instance, for all the users listed, an account is created in embedded LDAP with the same username and password as the Kubernetes secret:

   ```
   inventoryUsers:
     - uimadminuser
     - uimcmwsuser
     - uimmetricsuser
   ```

After the configuration and the input are available, the remaining activities are focused on Continuous Delivery, which can be automated.

1. Register a namespace per project:

   ```
   $COMMON_CNTK/scripts/register-namespace.sh -p project -t wlsko -l
   <lableSelector>

   # For example,
   $COMMON_CNTK/scripts/register-namespace.sh -p sr -t wlsko -l wlsko=enabled
   ```

2. Create one UIM PDB per instance:
   - If the Master UIM PDB exists in the CDB, clone the PDB. In this scenario, a Master PDB is created by cloning a seed PDB, deploying the UIM/RCU schema, and then optionally deploying cartridges. This Master is only valid for a specific UIM schema version.
   - If the Master CDB does not have the schema provisioned, do the following:
     a. Clone the seed PDB and then run the DB installer to create UIM and the RCU schema:

        ```
        $COMMON_CNTK/scripts/install-database.sh -p project -i instance -
        s $SPEC_PATH -a uim -c 1 (UIM Schema)
        ```

```
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -
s $SPEC_PATH -a uim -c 2 (RCU Schema)
```

Alternatively, the RCU schema can be reused. See "Reusing the RCU" section of "Recreating an Instance" for more information.

3. Create the Ingress:

```
$COMMON_CNTK/scripts/create-ingress.sh -p project -i instance -
s $SPEC_PATH -a uim
```

4. Create the instance.

```
$COMMON_CNTK/scripts/create-instance.sh -p project -i instance -
s $SPEC_PATH a uim
```

Deleting an Ingress

To delete an ingress, run the following command:

```
$COMMON_CNTK/scripts/delete-ingress.sh -p project -i instance -s $SPEC_PATH -
a uim
```

Deleting an Instance

This section describes the sequence of activities for deleting and cleaning up various aspects of the UIM environment.

To delete the application instance:

1. Run the following command:

```
$COMMON_CNTK/scripts/delete-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

2. Remove the instance content manually from the LDAP server using your LDAP Admin client. Specify ou=*project-instance*.

To clean up the PDB, drop it.

To clean up the configuration as code:

1. Delete the UIM instance and the database instance specification files.
2. Delete the secrets:

```
$COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -
s $SPEC_PATH -a uim delete uimldap,uimdb,rcudb,wlsadmin,opssWP,wlsRTE
```

3. Delete any additional custom secrets using kubectl.

Trying to streamline the processes and identifying when to omit certain activities and where other activities must be repeated can be challenging. For instance, dropping the UIM RCU schema is independent of deleting an instance, which happens through different script invocations. While the life-cycle of the UIM instance and the PDB should be aligned, there are also use cases where the business data in a PDB is required for re-use by a different UIM instance. For details on specific use cases, see "Reusing the Database State".

Securing Operations in Kubernetes Cluster

This section describes how to secure the operations of UIM cloud native users in a Kubernetes cluster. A well-organized deployment of UIM cloud native ensures that individual users have specific privileges that are limited to the requirements for their approved actions. The Kubernetes objects concerned are service accounts and RBAC objects.

All UIM cloud native users fall into the following three categories:

- Infrastructure Administrator
- Project Administrator
- UIM User

Infrastructure Administrator

Infrastructure Administrators perform the following operations:

- Install WebLogic Kubernetes Operator in its own namespace
- Create a project for UIM cloud native and configure it
- After creating a new project, run the **register-namespace.sh** script provided with the Common cloud native toolkit
- Before deleting a UIM cloud native project, run the **unregister-namespace.sh** script
- Delete a UIM cloud native project
- Manage the lifecycle of WebLogic Kubernetes Operator (restarting, upgrading, and so on)

Project Administrator

Project Administrators can perform all the tasks related to an instance level UIM cloud native deployment within a given project. This includes creating, updating, and deleting secrets, UIM cloud native instances, UIM cloud native DB Installer, and so on. A project administrator can work on one specific project. However, a given human user may be assigned Project Administrator privileges on more than one project.

UIM User

This class of users corresponds to the users described in the context of traditionally deployed UIM. These users can log into the user interfaces (UI) of UIM and can call the UIM APIs. These users are not Kubernetes users and have no privileges outside that granted to them within the UIM application. For details about user management, see "Unified Inventory Management System Administration Overview" in *UIM System Administrator's Guide* and "Setting Up Authentication" in this guide.

RBAC Requirements

The RBAC requirements for the WebLogic Kubernetes Operator are documented in its user guide. The privileges of the Infrastructure Administrator have to include these. In addition, the Infrastructure Administrator must be able to create and delete namespaces, operate on the WebLogic Kubernetes Operator's namespace. Depending on the specifics of your Kubernetes cluster and RBAC environment, this may require cluster-admin privileges.

The Project Administrator's RBAC can be much more limited. For a start, it would be limited to only that project's namespace. Further, it would be limited to the set of actions and objects that the instance-related scripts manipulate when run by the Project Administrator. This set of actions and objects is documented in the Common cloud native toolkit sample located in the **samples/rbac** directory.

Structuring Permissions Using the RBAC Sample Files

There are many ways to structure permissions within a Kubernetes cluster. There are clustering applications and platforms that add their own management and control of these permissions. Given this, the Common cloud native toolkit provides a set of RBAC files as a sample. You will have to translate this sample into a configuration that is appropriate for your environment. These samples are in **samples/rbac** directory within the toolkit.

The key files are **project-admin-role.yaml** and **project-admin-rolebinding.yaml**. These files govern the basic RBAC for a Project Administrator.

Do the following with these files:

1. Make a copy of both these files for each particular project, renaming them with the *project/namespace* name in place of "project". For example, for a project called "biz", these files would be **biz-admin-role.yaml** and **biz-admin-rolebinding.yaml**.
2. Edit both the files, replacing all occurrences of *project* with the actual *project/namespace* name.

   For the *project*-**admin-rolebinding.yaml** file, replace the contents of the "subjects" section with the list of users who will act as Project Administrators for this particular project.

   Alternatively, replace the contents with reference to a group that contains all users who will act as Project Administrators for this project.
3. Once both files are ready, they can be activated in the Kubernetes cluster by the cluster administrator using **kubectl apply -f** *filename*.

   It is strongly recommended that these files be version controlled as they form part of the overall UIM cloud native configuration.

In addition to the main Project Administrator role and its binding, the samples contain two additional and optional role-rolebinding sets:

- **project-admin-addon-role.yaml** and **project-admin-addon-rolebinding.yaml**: This role is per project and is an optional adjunct to the main Project Administrator role. It contains authorization for resources and actions in the project namespace that are not required by the toolkit, but might be of some use to the Project Administrator for debugging purposes.
- **wko-read-role.yaml** and **wko-read-rolebinding.yaml**: This role is available in the WebLogic Kubernetes Operator's namespace, and is an optional add-on to the Project Administrator's capabilities. It lets the user list the WKO pods and view their logs, which can be useful to debug issues related to instance startup and upgrade failures. This is suitable only for sandbox or development environments. It is strongly recommended that, even in these environments, WKO logs be exposed via a logs toolchain. The WebLogic Kubernetes Operator's Helm chart comes with the capability to interface with an ELK stack. For details, see https://oracle.github.io/weblogic-kubernetes-operator/managing-operators/#optional-elastic-stack-elasticsearch-logstash-and-kibana-integration.

6

Creating Your Own UIM Cloud Native Instance

This chapter provides information on creating your own UIM instance. This chapter provides information on how you can create a UIM instance that is tailored to the business requirements of your organization. However, if you want to first understand details on infrastructure setup and structuring of UIM instances for your organization, then see "Planning Infrastructure".

Before proceeding with creating your own UIM instance, you can look at the alternate and optional configuration options described in "Exploring Alternate Configuration Options".

When you created a basic instance, you used the operational scripts and the base configuration provided with the toolkit.

Creating your own instance involves various activities spanning both instance management and instance configuration and includes some of the following tasks:

- Customizing UIM Configuration Properties
- Deploying Cartridges
- Extending the WDT Model
- Working with Kubernetes Secrets
- Creating Inventory Users
- Assigning Application Roles to Inventory Users

Customizing UIM Configuration Properties

You use files to control many aspects of UIM performance and configuration. These system configuration files are located in ***UIM_builder_toolkit*/staging/cnsdk/uim-model/UIM/config**. Each file includes properties for which you can set values. See "Unified Inventory Management System Administration Overview" in *UIM System Administrator's Guide* for more information on property files and their contents.

In cloud native, these system configuration files are packaged into docker image while building the image and these files are available under **/UIM/config** folder within the pods.

Sample system configuration **custom-config.properties.sample** file is provided under **$SPEC_PATH/project/instance/config/uim/system-config** folder and will be available only after assembling the specifications. If you are doing it for the first time, run the **assemble-specifications.sh** script as mentioned in "Assembling the Specifications" and copy the sample property file to **custom-config.properties** and add key value to override the default value provided out-of-the-box for any specific system configuration property. The properties defined in **custom-config.properties** file are fed into the container using Kubernetes configuration maps. Any changes to these properties require the instance to be upgraded. Some properties can be updated dynamically and some may require rolling restart.

For properties with dynamic updates, run the following command after you change the property values:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

For properties with rolling restarts, run the following command after you change the property values:

```
$COMMON_CNTK/scripts/restart-applications.sh -p project -i instance -
s $SPEC_PATH -a uim -r ms
```

Sample custom-config.properties File

The sample **custom-config.properties** file is as follows:

```
#Add the overridden value for key in this custom-config.property file
ui.lastSavedSearch=true

# MapViewer Url, use this entry if mapviewer is running in a seperate domain
mapviewerUrl=http://hostname:port/mapviewer

#Timer properties
#Add new set of timers
#timer.MyTestingTimer.firstTime=120
#timer.MyTestingTimer.period=120
#timer.MyTestingTimer.listener=oracle.communications.customtimer.MyTestingTime
r
```

Adding New Properties

You can add new properties in either of the following ways:

- Define configuration files in solution cartridges. These configuration files are packaged into the customized docker image. See "Customizing Images" for more information. You need to build the image and restart the application in case of any changes to these properties.
- Update **custom-config.properties** in **$SPEC_PATH/project/instance/config/uim/system-config** to include new properties. These properties are available after you start the application and can be read using **SystemConfig** API.

Deploying Cartridges

Existing UIM cartridges that run on a traditional UIM deployment can still be used with UIM cloud native, but you prepare and deploy those cartridges differently when the cartridges have configuration files and Java code.

The cartridges can be categorized as follows:

- Simple cartridges that have entity specifications and Groovy or Drools code.
- Custom Extension cartridges that have Java code, configuration files, images, custom applications, Java libraries, Aspects, and localization.

Simple cartridges can be deployed on UIM Cloud Native running instance using Cartridge Management Tool or Design Studio. See "Overview" in *UIM Cartridge Guide* for more information.

To deploy Custom Extension cartridges in UIM Cloud Native environment:

1. Package the custom extension content of the cartridge into UIM docker image while building the image. The customized image should be generated with these cartridges. See "Customizing Images" for more information.

2. Deploy the cartridge on a running instance of UIM Cloud Native with the customized docker image. This can be done using CMT or Design Studio.

> **Note**
>
> You can follow the custom extension cartridge deployment by default in case you cannot identify the cartridge type.

Deploying Cartridges Using Design Studio

You can deploy cartridges directly from Design Studio using the Eclipse user interface or headless Design Studio. However, use Design Studio for deploying cartridges in scenarios where there is a lot of churn in the build, deploy and test cycle, but not for production environments.

In order to incorporate Design Studio into the larger UIM cloud native ecosystem, you need to have previously taken care of the mapping of the hostname to the Kubernetes cluster or the load balancer as described in "Planning and Validating Your Cloud Environment".

After confirming that this has been done, do the following in Design Studio:

- Ensure that the connection URL of the Design Studio environment project matches your UIM cloud native environment. This is likely: `http://`*instance.project*`.uim.org:30505/cartridge/wsapi`. The suffix `uim.org` is configurable.
- In the Design Studio workspace, depending on your network setup, you may need to set the **Proxy bypass** field in the **Network Connection** Preferences to: *instance.project*`.uim.org`

Deploying Cartridges Using Cartridge Management Tool

You can deploy cartridges using Cartridge Management Tool (CMT). Oracle recommends you to deploy cartridges using CMT in Continuous Deployment (CD).

Provide the following details into CMT:

- Connection URL that matches your UIM cloud native environment as follows:

  ```
  http://instance.project.uim.org:30505/cartridge/wsapi
  ```

  The suffix **uim.org** is configurable.
- UIM cluster name
- Cartridge file location of **Super Jar** or **Simple Jar**

The sample **build.properties** file is as follows:

```
url=http://quick.sr.uim.org:30505/cartridge/wsapi
username=<cartridge management web service user name>
password=<password>
fileLocation=<cartridge file location>
deploy.wl.target.name=uimcluster
```

To deploy the cartridge, run the following command:

```
ant -lib ../lib -f build.xml deploy-cartridge
```

Deploying Cartridges using SSL

You can deploy cartridges using SSL in either of the following ways:

- Using CMT
- Using Design Studio

Deploying Cartridges using SSL when SAML 2.0 Authentication is NOT Enabled on UIM CN in CMT

To deploy cartridges using SSL in CMT:

1. Upload the external server UIM certificate in JVM. The keytool is found in the **bin** directory of your jdk installation.
   - In Unix:

     ```
     ./keytool -importcert -v -trustcacerts -alias <alias> -file /
     path_to_copied_uim_certificate/commoncert.pem -keystore /
     path_to_jdk/jre/lib/security/cacerts -storepass <password>
     ```

   - In Windows (using the command prompt):

     ```
     keytool -import -alias <alias> -keystore "/path_to_jdk/jre/lib/security/
     cacerts" -file "/path_to_copied_uim_certificate/commoncert.pem"
     ```

   Use the default password of Java KeyStore.
2. In CMT **build.properties** under tag **sslKeyStore**, provide **commoncert.pem** file as follows:

   ```
   url=https://instance.project.uim.org:30443/cartridge/wsapi
   username=<cartridge management web service user name>
   password=<password>
   sslKeyStore="/path_to_copied_uim_certificate/commoncert.pem"
   ```

Deploying Cartridges using SSL when SAML 2.0 Authentication is NOT Enabled on UIM CN in Design Studio

To deploy cartridges using SSL when authentication is not enabled on UIM CN in Design Studio:

1. Upload the external server UIM certificate in **/path_to_jdk/jre/lib/security/cacerts** if not uploaded earlier:

   ```
   keytool -import -alias <alias> -keystore "/path_to_jdk/jre/lib/security/
   cacerts" -file "/path_to_copied_uim_certificate/commoncert.pem"
   ```

2. Provide the following `vmargs` in **eclipse.ini** file as follows:

```
-vmargs
-Djavax.net.ssl.trustStore=\path_to_jdk\jre\lib\security\cacerts
-Djavax.net.ssl.trustStorePassword=<password>
```

3. Run Eclipse as **Run As Administrator**.
4. Generate **jks** file from **commoncert.pem** and provide it in Eclipse under SSL tab under **Studio Environment** configuration as follows:

```
keytool -importcert -v -alias <alias> -file /path-to/<certificate>.crt -
keystore /path-to/<truststore>.jks -storepass <password>
```

5. Provide the HTTPS address for deploying the cartridge: **https:// instance.project.uim.org:30543/cartridge/wsapi**

Deploying Cartridge using SSL when SAML 2.0 Authentication is Enabled on UIM CN in CMT

To deploy cartridges using SSL when SAML 2.0 authentication is enabled on UIM CN in CMT:

1. Upload the external server IdP and UIM certificate in JVM. The keytool is found in the **bin** directory of your jdk installation.
   - In Unix:

```
./keytool -importcert -v -trustcacerts -alias <alias> -file /
path_to_copied_idp_certificate/idpcert.pem -keystore /
path_to_jdk/jre/lib/security/cacerts -storepass <password>
./keytool -importcert -v -trustcacerts -alias <alias> -file /
path_to_copied_uim_certificate/commoncert.pem -keystore /
path_to_jdk/jre/lib/security/cacerts -storepass <password>
```

   - In Windows (using the command prompt):

```
keytool -import -alias <alias> -keystore "/path_to_jdk/jre/lib/
security/ cacerts" -file "/path_to_copied_idp_certificate/idpcert.pem"
keytool -import-alias <alias> -keystore "/path_to_jdk/jre/lib/security/
cacerts" -file "/path_to_copied_uim_certificate/commoncert.pem"
```

2. Generate jks file from **idpcert.pem** and **commoncert.pem** and provide path to the jks file in CMT **build.properties**.

```
keytool -importcert -v -alias <alias> -file /
path_to_copied_uim_certificate/commoncert.pem -keystore /path-to/
<truststore>.jks -storepass <password>

#add idp certificate to same truststore
keytool -importcert -v -alias <alias> -file /path_to_copied_idp
_certificate/idpcert.pem -keystore /path-to/
<truststore>.jks -storepass <password>
```

3. In CMT **build.properties** under tag **sslKeyStore**, provide **truststore.jks** file as follows:

```
url=https://<instance>.<project>.<hostSuffix>:30543/cartridge/wsapi
username=<cartridge management web service user name>
```

```
password=<password>
sslKeyStore="/path_to_generated_keystore/<truststore>.jks"
```

Deploying Cartridges using SSL when SAML 2.0 Authentication is Enabled on UIM CN in Design Studio

To deploy cartridges using SSL when SAML 2.0 authentication is enabled on UIM CN in Design Studio:

1. Upload the external server IdP and UIM certificate in **/path_to_jdk/jre/lib/security/cacerts** if not uploaded earlier:

   ```
   keytool -import -alias <alias> -keystore "/path_to_jdk/jre/lib/security/
   cacerts" -file "/path_to_copied_IDP_certificate/idpcert.pem"

   keytool -import -alias <alias> -keystore "/path_to_jdk/jre/lib/security/
   cacerts" -file "/path_to_copied_uim_certificate/commoncert.pem"
   ```

2. Provide the following `vmargs` in **eclipse.ini** file as follows:

   ```
   -vmargs
   -Djavax.net.ssl.trustStore=\path_to_jdk\jre\lib\security\cacerts
   -Djavax.net.ssl.trustStorePassword=<password>
   ```

3. Run Eclipse as **Run As Administrator**.
4. Generate **jks** file from **commoncert.pem** and **idpcert.pem** and provide it in Eclipse under SSL tab under **Studio Environment** configuration as follows:

   ```
   keytool -importcert -v -alias <alias> -file /path-to/commoncert.pem -
   keystore /path-to/<truststore>.jks -storepass <password>

   #add idp certificate to same truststore crated above
   keytool -importcert -v -alias <alias> -file /path-to/idpcert.pem -
   keystore /path-to/<truststore>.jks -storepass <password>
   ```

5. Provide the HTTPS address for deploying the cartridge: **https://<instance>.<project>.<hostSuffix>:30543/cartridge/wsapi**

Adding New WDT Metadata

The Common cloud native toolkit provides the base WDT metadata in **$COMMON_CNTK/charts/uim-app/charts/uim/templates**. As the UIM application requires this WDT metadata for the proper functioning, this must not be edited. Instead, the toolkit provides a mechanism whereby new pieces of WDT metadata can be included in the final description of the domain.

See "Extending the WebLogic Server Deploy Tooling (WDT) Model" for complete details on the general process for providing custom WDT. The steps described must be repeated for a variety of WDT use cases.

To provide the required configuration for JMS queues, create custom JMS Resources as described in "Adding a JMS System Resource".

Handling of sensitive data from within the WDT metadata fragment is supported as described in the "Accessing Kubernetes Secrets from WDT Metadata".

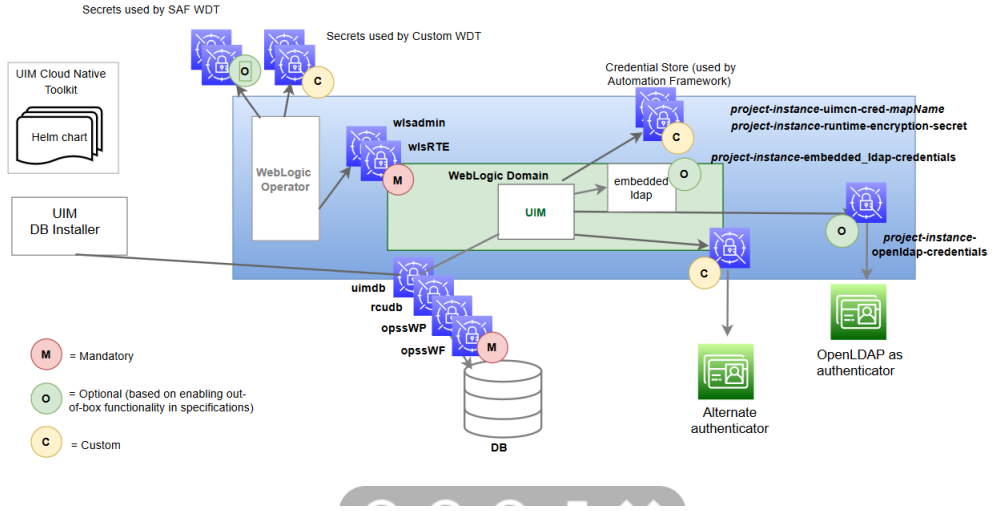
Working with Kubernetes Secrets

Secrets are Kubernetes objects that you must create in the cluster through a separate process that adheres to your corporate policies around managing secure data. Secrets are then made available to UIM cloud native by declaring them in your configuration.

When the UIM cloud native sample scripts are not used for creating secrets, the secrets you create must align to what is expected by UIM. The sample scripts contain guidelines for creating secrets.

The following diagram illustrates the role of Kubernetes Secrets in a UIM cloud environment:

Figure 6-1 Kubernetes Secrets in UIM Cloud Environment



There are three classifications of secrets, as shown in the above illustration:

- Mandatory (Pre-requisite) Secrets
- Optional Secrets
- Custom Secrets

About Mandatory Secret

Mandatory secrets must be created prior to running the cartridge management scripts or the instance creation script.

The toolkit provides the sample script: **$COMMON_CNTK/scripts/manage-app-credentials.sh** to create the secrets for you. Refer to the script code to see the naming and internal structure required for each of these secrets.

See the following topics for more details about Kubernetes Secrets:

- Creating Secrets
- Management of Secrets

About Optional Secrets

Optional secrets are dictated by enabling the out-of-the-box configuration. There is some functionality that is pre-configured in UIM cloud native and can be enabled or disabled in the specification files. When the functionality is enabled, these secrets must be created in the cluster before a UIM instance is created.

- If you use OpenLDAP for authentication, UIM cloud native relies on the following secret to have been created:

  ```
  project-instance-openldap-credentials
  ```

  The toolkit provides a sample script to create these secrets for you **$COMMON_CNTK/samples/credentials/manage-uim-ldap-credentials.sh** by passing in `-o secret`.

- When SAF is configured, SAF secrets are used. SAF secrets are similar to custom secrets and are declared in a specialized area within the **app-uim** specification that feeds into the SAF-specific WDT custom template.

  ```
  safDestinationConfig:
    - name: <SAF Configuration Name>
      t3Url: <Remote Destination URL>
      secretName: <Secret Name of Remote Destination Credentials>
  ```

About Custom Secrets

UIM cloud native provides a mechanism where WDT metadata can access sensitive data through a custom secret that is created in the cluster and then declared in the configuration. See "Accessing Kubernetes Secrets from WDT Metadata" to familiarize yourself with this process.

This class of secrets are required only if you need secrets for this mechanism.

To use custom secrets with WDT metadata:

> **Note**
>
> As an example, this procedure uses a WDT snippet for authentication.

1. Add secret usage in the WDT metadata fragment:

   ```
   Host: '@@SECRET:authentication-credentials:host@@'
   Port: '@@SECRET:authentication-credentials:port@@'
   ControlFlag: SUFFICIENT
   Principal: '@@SECRET:authentication-credentials:principal@@'
   CredentialEncrypted: '@@SECRET:authentication-credentials:credential@@'
   ```

2. Add the secret to the **app-uim** specification.

   ```
   project:
    customSecrets:
      secretNames: {} # This empty declaration should be removed if adding
   ```

```
items here.
   #secretNames:
   # - mysecret1
   # - mysecret2
```

3. Create the secret in the cluster, by using any one of the following methods:
   - Using Common cloud native toolkit scripts
   - Using a Template
   - Using the Command-line Interface

   In the example metadata shown in step 1, the secret must capture host, port, principal, and credential.

See "Mechanism for Creating Custom Secrets" for details about the methods.

Accommodating the Scope of Secrets

The WDT metadata fragments are defined at the project level as the project typically owns the solution definition. Accommodating this is a simple task.

To walk through this, we will use authentication as an example and introduce a UIM project that includes three instances: development, test, and production. The production environment has a dedicated authentication system, but the development and test instances use a shared authentication server.

To accommodate this scenario, the following changes must be made to each of the basic steps:

1. Define a naming strategy for the secrets that introduce scoping. For instance, secrets that need instance level control could prepend the instance name. In the example, this results in the following secret names:
   - `UIM-dev-authentication-credentials`
   - `UIM-test-authentication-credentials`
   - `UIM-prod-authentication-credentials`
2. Include the secret in the WDT fragment. In order for this scenario to work, a generic way is required to declare the "scope" or instance portion of the secret name. To do this, use the built-in Helm values:

   `.Values.name - references the full instance name (`*`project-instance`*`)`
   `.Values.namespace - references the project name (`*`project`*`)`

   If the fragment needs to support instance-level control, derive the instance name portion of the secret name.

```
Host: '@@SECRET:{{ .Values.name }}-authentication-credentials:host@@'
Port: '@@SECRET:{{ .Values.name }}-authentication-credentials:port@@'
ControlFlag: SUFFICIENT
Principal: '@@SECRET:{{ .Values.name }}-authentication-
credentials:principal@@'
CredentialEncrypted: '@@SECRET:{{ .Values.name }}-authentication-
credentials:credential@@'
```

3. Add the secret to the **app-uim** specification.

```
## Dev Instance Spec

 #Custom secrets
# Multiple secret names can be providedinstance:
customSecrets:
   secretNames: {} # This empty declaration should be removed if adding
items here.
   #secretNames:
   # - mysecret1
   # - mysecret2

 ## Test Instance spec

 #Custom secrets
# Multiple secret names can be provided
customSecrets:
  secretNames:
   - UIM-test-authentication-credentials

 ## Prod Instance Spec

#Custom secrets
# Multiple secret names can be provided
customSecrets:
 secretNames:
   - UIM-prod-authentication-credentials
```

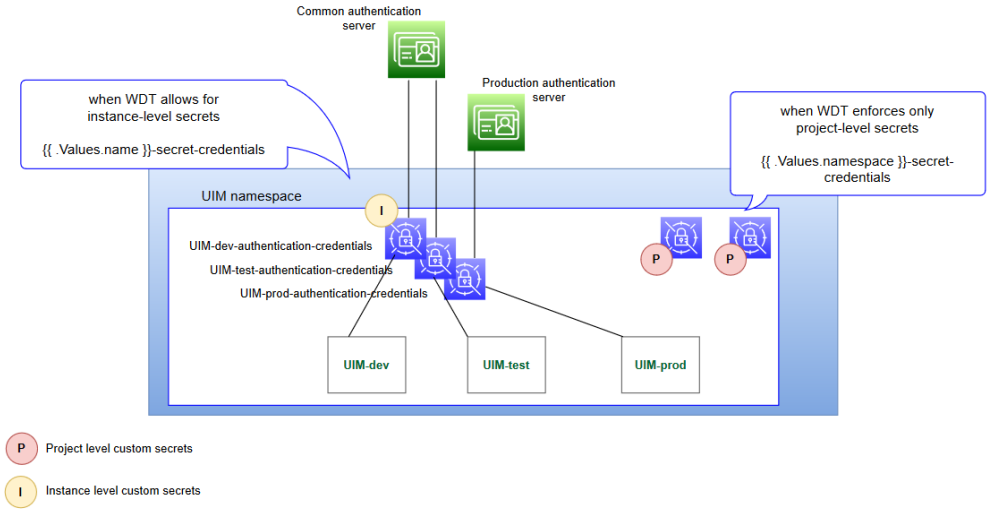
4. Create the secret in the cluster by following any one of the methods described in "Mechanism for Creating Custom Secrets". In our example, the secret would need to capture host, port, principal and credential. Each instance would need a secret created, but the values provided depend on which authentication system is being used.

```
# Dev secret creation

 kubectl create secret generic UIM-dev-authentication-credentials \
-n UIM \
--from-literal=principal=<value1> \
--from-literal=credential=<value2> \
--from-literal=host=<value3> \
--from-literal=port=<value4>

 # Test secret creation

kubectl create secret generic UIM-test-authentication-credentials \
-n UIM \
--from-literal=principal=<value1> \
--from-literal=credential=<value2> \
--from-literal=host=<value3> \
--from-literal=port=<value4>

  ##Production secret creation

 kubectl create secret generic UIM-prod-authentication-credentials \
-n UIM \
```

```
--from-literal=principal=<prodvalue1> \
--from-literal=credential=<prodvalue2> \
--from-literal=host=<prodvalue3> \
--from-literal=port=<prodvalue4>
```

The following diagram illustrates the secret landscape in this example:

Figure 6-2 Landscape of Secrets



Mechanism for Creating Custom Secrets

You can create custom secrets in any of the following ways:

- Using Scripts
- Using a Template
- Using the Command-line Interface

Using Scripts to Create Secrets

Functionality such as OpenLDAP and Embedded LDAP Store that can be enabled or disabled in UIM cloud native relies on pre-requisite secrets to be created. In such cases, the toolkit provides sample scripts that can create the secrets for you. While these scripts are useful for configuring instances quickly in development situations, it is important to remember that they are sample scripts, and not pipeline friendly. These scripts are also essential because when the secret is mandated by UIM cloud native, both the secret name and the secret data are available in the sample script that populates it.

As an example, the secrets used by the Credential Store mechanism must follow a specific naming convention:

projectName-instanceName-uimcn-cred-*mapName*

Using a Template

To create custom secrets using a template:

1. Save the secret details into a template file.

```
apiVersion: v2
kind: Secret
metadata:
 labels:
 weblogic.resourceVersion: domain-v2
 weblogic.domainUID: project-instance
 weblogic.domainName: project-instance
 namespace: project
 name: secretName
type: Opaque
stringData:
password_key: <password_key_value>
user_key: <user_key_value>
```

2. Run the following command to create the secret:

```
kubectl apply -f templateFile
```

Using the Command-line Interface

You can also specify the secret name and the details directly on the command-line interface:

```
kubectl create secret generic secretName \
-n project \
--from-literal=password_key=<password_key_value> \
--from-literal=user_key=<user_key_value>
```

Creating Inventory Users

This section describes how to use the sample scripts to create Inventory UI and Cartridge Deployment users and provide required project configuration in the UIM cloud native.

The sample scripts also provide the ability to populate the OpenLDAP server so that UIM can authenticate any inventory UI and cartridge deployment users.

Creating Users in Embedded LDAP

A fixed set of users can be created in Embedded LDAP. Sample scripts to create LDAP user secrets are placed in **$COMMON_CNTK/samples/credentials/manage-uim-credentials.sh**. Create a user information file with the list of users and WebLogic sever groups as follows:

```
uim:uimdev:secret:uim-users,uim-metrics-users
uim:uimqa:secret:uim-users,,uim-metrics-users
uim:cmwsdev:secret:Administrators,Cartridge_Management_WebService
uim:cmwsqa:secret:Administrators,Cartridge_Management_WebService
```

To create users in embedded LDAP, run the following command:

```
./manage-uim-credentials.sh  -p project  -i instance -c  create -f <text file>
```

Verify the Secret Creation

To verify the secret, run the following command:

```
kubectl get secrets -n project
#
NAME                                    TYPE
project-instance-uimcn-cred-uim          Opaque
```

Add all the inventory users to the inventory users in embedded LDAP under the **inventoryUsers** section in **app-uim** file. During the creation of the UIM server instance, for all the inventory users listed, an account is created in embedded LDAP with the same username and password and groups as the Kubernetes secret.

```
List all cartridge users and execute
# $COMMON_CNTK/samples/credentials/manage-uim-credentials.sh to create
# the cartridge user secret before creating the UIM CN instance.
#inventoryUsers: {} # This empty declaration should be removed if adding
items here.
inventoryUsers:
  - uimdev
  - uimqa
  - cmwsdev
  - cmwsqa
```

> **Note**
>
> Cartridge deployment can be performed by users `cmwsdev` and `cmwsqa`, where as users `uimdev` and `uimqa` can access Inventory UI.

Creating Users in OpenLDAP

UIM cloud native recommends to use external LDAP for maintaining user accounts. COMMON_CNTK includes sample scripts for OpenLDAP user creation. See "Setting Up Authentication" to configure OpenLDAP server.

UIM groups need to be created in the OpenLDAP server and assign them to a Human user before the user can access UIM functioning. The secrets giving access to OpenLDAP server for authentication purposes need to be set up and then enable the OpenLDAP integration by setting **true** to **uim.authentication.openldap.enabled** in **app-uim.yaml** file.

The sample **app-uim.yaml** is as follows:

```
# External authentication
# When enabled, kubernetes secret "<project>-<instance>-openldap-credentials"
# must exist
authentication:
  openldap:
    enabled: true
```

Creating Group and User

To create Human user and assign the user to a group in OpenLDAP, edit the **$COMMON_CNTK/samples/credentials/uim_users.txt** file.

For example:

```
uim:uimldapuser1:ldap:uim-users
uim:uimldapuser2:ldap:uim-users
```

Install OpenLDAP client on the host where you are running the scripts. Run the following command that installs the OpenLDAP clients:

```
sudo -s yum -y install openldap-clients
```

Run the sample script to populate the OpenLDAP server and create the secret as follows:

```
$COMMON_CNTK/samples/credentials/manage-uim-ldap-credentials.sh -p project -i
instance \
 -c create \
 -o account,secret \
 -H <Hostname> \
 -A <Admin dn> \
 -G <Domain dn> \
 -U <User dn>
```

Verify the secret creation as follows:

```
kubectl get secrets -n project

#
NAME                                    TYPE
project-instance-openldap-credentials          Opaque
```

Creating OpenLDAP Users

Create the OpenLDAP users as follows:

```
$COMMON_CNTK/samples/credentials/manage-uim-credentials.sh -p project -i
instance -c create -f ./uim_users.txt \
 -H <Hostname> \
 -A <Admin dn> \
 -G <Domain dn>
```

Verify the secret creation as follows:

```
kubectl get secrets -n project

#
NAME                                    TYPE
project-instance-uimcn-cred-uim          Opaque
```

The sample for creating OpenLDAP users is as follows:

```
$COMMON_CNTK/samples/credentials/manage-uim-credentials.sh -p project -i
instance -c create -f ./uim_users.txt \
 -H uimopenldap.snphxprshared1.gbucdsint02phx.oraclevcn.com \
 -A cn=Manager,dc=uimcn-ldap,dc=com \
 -G ou=Domains,dc=uimcn-ldap,dc=com
```

> **Note**
>
> The above sample prompts for OpenLDAP administrator's password and passwords for all the users you are creating. After the users are created successfully you can see a message similar to **LDAP User `uimldapuser1` created**.

Configuring Other LDAP Systems

The **manage-uim-credentials.sh** script supports the OpenLDAP system. To provide support for a different LDAP provider, you must modify the script. Also, the corresponding LDAP client or the API must be installed on the system where the script is processed.

You must modify the following functions within this script:

- `create_ldap_account`. This function creates the user account in the LDAP system and associates the user to the specified groups.
- `update_ldap_account`. This function updates the user password.
- `delete_ldap_account`. This function deletes the user from the LDAP system and disassociates this user from the specified group.
- `verify_ldap_account`. This function verifies that the specified user exists in the LDAP server.

For details on developing the functions, see the developer's guide of the target LDAP server that you want to use.

Assigning Application Roles to Inventory Users

Once inventory users are created, applications roles are to be assigned to perform operations in Inventory UI. See "Unified Inventory Management System Administration Overview" in *UIM System Administrator's Guide* for available application roles. Sample script is provided to assign roles to the inventory users. EM Console can also be used for role assignments.

To assign role, edit the **$COMMON_CNTK/samples/credentials/uim-user-roles.txt** file as follows:

```
uimldapuser1:uimuser,ProductAdministrator
uimldapuser2:uimuser
```

Run the sample script to assign roles to the users:

```
$COMMON_CNTK/samples/credentials/assign-role.sh -p project  -i instance  -f
<pathToTheFile>/uim-user-roles.txt
```

7

Extending the WebLogic Server Deploy Tooling (WDT) Model

While the Common cloud native toolkit provides a domain model that is sufficient to support the operation of the UIM application, there are a few aspects that you can customize to meet your business requirements. This chapter provides the general mechanism that UIM cloud native provides for how custom WebLogic Server Deploy Tooling (WDT) metadata can be used.

The following sections enable you to familiarize yourself with the basic extension mechanism. For details on using the sample scripts to add custom WDT metadata, see "Using the Sample Scripts to Extend the WDT Model".

About the Custom WDT Extension Mechanism

The Common cloud native toolkit exposes an extension mechanism to extend the base WDT domain configuration. For better management practices, you must specify different WDT model fragments in multiple **.tpl** files that can be included in instances as necessary.

All extensions must be located in your source control repository in a directory referred to as *customExtPath*, which is provided during instance creation. This does not need to be the same location as *specPath* that contains the specification files. See the illustration about the directory structure in "Managing Configuration as Code".

Using the WDT Model Tools

This section describes the WDT model tools that you can use when extending the WDT model.

The WDT model tools are available at: https://github.com/oracle/weblogic-deploy-tooling. The documentation available on GitHub describes various tools, which are included in the Common cloud native toolkit.

For a developer trying to modify or extend the WDT model for a custom UIM instance, the following tools are the most useful:

- WDT Discover Domain
- WDT Validate Model

WDT Discover Domain Tool

One way to generate the desired custom model is to manually create a WLS domain (using legacy installers, wlst scripts, console UI changes, and so on) that contains all the constructs that are required and is known to work, in terms of the custom use case. The WDT Discover Domain tool can be pointed at this WLS domain to generate a set of model files. These can be scanned and pruned to get the portions that are of custom interest. They can further be parameterized using WDT's properties files or using Helm values.

If WDT properties are used to parameterize, ensure that you add that properties file to the extension point in the custom implementation.

If Helm values are used to parameterize, ensure that you add these values to the appropriate location - base/app-uim/*shape* yamls.

To discover a domain, run the following commands on the prepared WLS admin server or standalone server:

```
# ensure ORACLE_HOME is properly set
cd $ORACLE_HOME
mkdir wdt && cd wdt
wget https://github.com/oracle/weblogic-deploy-tooling/releases/download/
weblogic-deploy-tooling-1.6.0/weblogic-deploy.zip
# Replace 1.6.0 with the actual WDT version as per UIM documentation
unzip weblogic-deploy.zip
cd weblogic-deploy/bin
./discoverDomain.sh -oracle_home $ORACLE_HOME \
 -domain_home domain-home \
 -archive_file archive \
 -model_file model \
 -domain_type domain-type \
 -admin_user admin-user \
 -admin_url t3-admin-url
```

where:

- *archive* and *model* are the directory+name of the files that the discovery tool creates. The model file is of primary importance in this situation.
- *domain-type* is JRF for UIM applications

The command extracts the model from the running WLS instance. Alternatively, if it is sufficient to extract the model from the domain configuration files, the `admin_user` and `admin_url` parameters can be left out.

WDT Validate Model Tool

This tool is useful in the following scenarios:

- When there is a need to see what attributes and sub-fields are available for a model element
- When there is a need to see if a model fragment is valid

Trying to test a newly written or even a modified model file by incorporating it into an instance creation is cumbersome and often an inefficient way to test your changes. You need to check the Introspector logs to see the details of any errors.

With the Validate Model tool, it is easier to validate the model file, especially if you are building the model iteratively.

Common WDT Extension Mechanism

This section describes the extension mechanism that is generic and common to all methods of extending WDT metadata.

Enabling the Extension Mechanism

To enable the extension mechanism:

1. Copy **$COMMON_CNTK/samples/uim/customExtensions/_custom-domain-model.tpl** to your source control repository *customExtPath*. This file is a single location where other template files, which store specific WDT metadata fragments, can be included for a UIM instance. This sets up the WDT fragments for re-use across a project, while allowing conditional inclusion based on instance level values in the specification files.
2. Enable the extension mechanism by setting the **custom** flag to `true` in the **app-uim** specification and including `_custom-domain-model.tpl`:

   ```
   custom:
     enabled: true
     #wdtFiles: {}
     wdtFiles:
      - _custom-domain-model.tpl
   ```

   The basic extension mechanism is now enabled.

For each WDT fragment that is destined for inclusion, perform the following additional steps:

- Provide the WDT fragment
- (Optional) Parameterize the WDT Fragment
- Load the WDT Fragment
- List the .tpl files
- Debug the changes in the Helm chart

Providing the WDT Fragment

Naming convention dictates that the template files start with an underscore _. For example, `_custom-extension-support.tpl`.

You can copy any one of the WDT fragments provided in the samples, or you can create your own. If you provide your own WDT fragment, then you will need to reverse engineer the required metadata using the WDT tooling. For these samples, see "Using the WDT Model Tools".

If you create your own .tpl file, ensure that the WDT fragment is enclosed in a `define` block as follows:

```
{{- define "uim.custom-extension-support" -}}
custom model fragment goes here
{{- end }}
```

(Optional) Parameterizing the WDT Fragment

Instead of hard coding the values into the WDT, you can parameterize the content so that specific values can be driven from the Helm chart. Determine which values fall into this category and then apply the following changes:

To parameterize the WDT fragment:

1. Update the WDT to use a parameter as illustrated in the following example:

   ```
   Host: 'external.provider.hostname'

   becomes....
   ```

```
Host: '{{ .Values.custom.extension.host }}'
```

2. Add values to the application instance in the **app-uim** specification found in the source control at $SPEC_PATH.

```
custom:
 enabled: true
 <extension>:
   host: provide_explicit_value_here
```

The custom area of the specification file is where you can add as much content as needed for your extension use cases. Oracle recommends that you keep the yaml structure as flat as possible.

Loading the WDT Fragment

The sample `_custom-domain-model.tpl` already has conditional inclusions for some of the samples provided in the toolkit. JMS, JDBC, SAF, and custom application archives can be enabled by providing the appropriate flag in the instance specification and including the specific **.tpl** file in the **app-uim** specification. For the samples, you do this task as described in "Using the Sample Scripts to Extend the WDT Model".

Load the model fragment into *extension_Directory*/_custom-domain-model.tpl as follows:

```
{{- define "uim.custom-domain-model" -}}
{{- $root := . }}
custom-<extension>-support.<index>.yaml: |+
 {{- include "uim.custom-extension-support" $root | nindent 2 }}
{{- end }}
```

> **Note**
>
> See the yaml naming convention that is specified by **wdt - filename.yaml**. The index used determines the loading order when there are multiple yaml files. Indexes below 80 are reserved for internal Oracle use.

The WDT may only need to be used conditionally. It is important to be able to exclude the fragment based on the values provided in the **app-uim** specification. In this case, `_custom-domain-model.tpl` should include the condition that needs to be met for the WDT to be included.

> **Note**
>
> Including the WDT in *extension_Directory*, which makes it available during instance creation, but not used does not pose any problems for Helm.

```
{{- define "uim.custom-domain-model" -}}
{{- $root := . }}
{{- if .Values.custom.<extension>.enabled }}
custom-extension-support.index.yaml: |+
 {{- include "uim.custom-extension-support" $root | nindent 2 }}
```

```
{{- end }}
{{- end }}
```

Listing the TPL Files in the Project

For each WDT fragment that is created in a .tpl file, it needs to be listed in the **app-uim** specification.

```
custom:
  enabled: true
  #wdtFiles: {}
  wdtFiles:
   - _custom-domain-model.tpl
   - new_wdt.tpl
```

Debugging Helm Chart Changes

When making changes to existing yaml files or creating new WDT fragments, it is useful to test the changes before attempting to create an instance.

You can use the following scripts provided with the toolkit to debug Helm chart changes:

- **$COMMON_CNTK/scripts/uim/lint-uim-instance-chart.sh**
- **$COMMON_CNTK/scripts/create-applications-dry-run.sh**

You can now create a UIM instance.

Using the Sample Scripts to Extend the WDT Model

This section provides instructions for extending the WDT model by using the sample scripts that are provided with the toolkit. You add custom WDT metadata to create your own UIM instances.

The toolkit includes sample scripts for the following:

- Adding a JDBC DataSource
- Adding a JMS System Resource
- Adding a Store-and-Forward-Agent and SAF Resources
- Deploying Entities to a UIM WebLogic Domain
- Extending the WDT Metadata for an External Authenticator

The general and common extension process described in "Common WDT Extension Mechanism" must be repeated for each of the use cases described in this section.

Adding a JDBC DataSource

The WDT fragment describing a **JDBCSystemResource** is provided in the **$COMMON_CNTK/samples/uim/customExtension/_custom-jdbc-support.tpl** sample file.

To incorporate this fragment into your UIM instance:

1. Enable the extension mechanism by setting the `custom` flag to **true** and add the **custom-domain-model** to the list of included wdtFiles in the **app-uim** specification:

```
custom:
  enabled: true
  wdtFiles:
   - _custom-domain-model.tpl
```

2. Provide the WDT fragment by copying **$COMMON_CNTK/samples/uim/customExtensions/_custom-jdbc-support.tpl** to the *customExtPath* in your source control repository.
3. Parameterize the WDT fragment. The fragment has already been parameterized and uses values specified in the shape file. You must update the remaining values enclosed in angular brackets. By default, this WDT reads the JDBC values from the shape that is provided during instance creation.

> **Note**
>
> Kubernetes Secrets can also be used to provide sensitive data such as username and password. See "Accessing Kubernetes Secrets from WDT Metadata" for details.

```
{{/* vim: set filetype=mustache: */}}
{{/* Copyright (c) 2021, Oracle and/or its affiliates. */}}
{{/*
Add custom JDBC resources
*/}}
{{- define "uim.custom-jdbc-support" -}}
resources:
  JDBCSystemResource:
    '<custom-conn-pool>':
      Target: '{{ .Values.clusterName}}'
      JdbcResource:
        JDBCDriverParams:
          URL: 'jdbc:oracle:thin:@//
@@SECRET:<custom_secret_name>:<dbconnectionstring_key>@@'
          PasswordEncrypted: '<password>'
          #PasswordEncrypted:
'@@SECRET:<custom_secret_name>:<password_key>@@'
          DriverName: oracle.jdbc.OracleDriver
          Properties:
            user:
              Value: '<user>'
              #Value: '@@SECRET:<custom_secret_name>:<user_key>@@'
            oracle.net.CONNECT_TIMEOUT:
              Value: {{ default
"10000" .Values.jdbc.oracleNetConnectTimeout }}
            oracle.jdbc.ReadTimeout:
              Value: {{ default
"3660000" .Values.jdbc.oracleJdbcReadTimeout }}
        JDBCConnectionPoolParams:
          InitialCapacity: {{ default "1" .Values.jdbc.initialCapacity }}
          MaxCapacity: {{ default "15" .Values.jdbc.maxCapacity }}
```

```
          MinCapacity: {{ default "1" .Values.jdbc.minCapacity }}
          ShrinkFrequencySeconds: {{ default
"900" .Values.jdbc.shrinkFrequencySeconds }}
          TestFrequencySeconds: {{ default
"300" .Values.jdbc.testFrequencySeconds }}
          TestConnectionsOnReserve: {{ default
"true" .Values.jdbc.testConnectionsOnReserve }}
          SecondsToTrustAnIdlePoolConnection: {{ default
"10" .Values.jdbc.secondsToTrustAnIdlePoolConnection }}
          StatementCacheSize: {{ default
"30" .Values.jdbc.statementCacheSize }}
          ConnectionCreationRetryFrequencySeconds: {{ default
"30" .Values.jdbc.connectionCreationRetryFrequencySeconds }}
          IgnoreInUseConnectionsEnabled: {{ default
"true" .Values.jdbc.ignoreInUseConnectionsEnabled }}
          InactiveConnectionTimeoutSeconds: {{ default
"0" .Values.jdbc.inactiveConnectionTimeoutSeconds }}
          StatementCacheType: '{{ default
"LRU" .Values.jdbc.statementCacheType }}'
          CountOfTestFailuresTillFlush: {{ default
"5" .Values.jdbc.countOfTestFailuresTillFlush }}
          CountOfRefreshFailuresTillDisable: {{ default
"5" .Values.jdbc.countOfRefreshFailuresTillDisable }}
          RemoveInfectedConnections: {{ default
"false" .Values.jdbc.removeInfectedConnections }}
          ConnectionReserveTimeoutSeconds: {{ default
"10" .Values.jdbc.connectionReserveTimeoutSeconds }}
          StatementTimeout: {{ default
"3630" .Values.jdbc.statementTimeout }}
        JDBCDataSourceParams:
          JNDIName: '<jdbc/custom-conn-pool>'
          GlobalTransactionsProtocol: 'None'
{{- end }}
```

4. The fragment is already configured for conditional loading based on the presence of the `jdbc` flag in the **app-uim** specification. Set the `jdbc` flag to `true`.

```
custom:
  enabled: true
  jdbc: true
```

5. Add the JDBC .tpl file to the **app-uim** specification:

```
custom:
  enabled: true
  jdbc: true
  wdtFiles:
   - _custom-domain-model.tpl
   - _custom-jdbc-support.tpl
```

You can now create the UIM instance.

6. You can create the UIM instance with *customExtPath* as follows:

```
$COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
s $SPEC_PATH -a uim -m <customExtPath>
```

Adding a JMS System Resource

The WDT fragment describing a JMS System Resource is provided in the **$COMMON_CNTK/samples/uim/customExtension/_custom-jms-support.tpl** sample file.

To incorporate this fragment into your UIM instance:

1. Enable the extension mechanism by setting the `custom` flag to **true** and add the **custom-domain-model** to the list of included wdtFiles in the **app-uim** specification:

```
custom:
  enabled: true
  wdtFiles:
   - _custom-domain-model.tpl
```

2. Provide the WDT fragment by copying **$COMMON_CNTK/samples/uim/customExtensions/_custom-jms-support.tpl** to the *customExtPath* in your source control repository. While this sample shows WDT for a JMS Queue and JMS Topic, any other JMS entity can be supplied instead. See "Using the WDT Model Tools" for details on establishing the correct WDT.
3. Parameterize the WDT fragment. The fragment has not been parameterized. The text enclosed in angular brackets must be replaced with specific values.
Alternatively, update the WDT to parameterize content and provide actual values in the **app-uim** specification.
4. The fragment is already configured for conditional loading based on the presence of the `jms` flag in the **app-uim** specification. Set the `jms` flag to `true`.

```
custom:
  enabled: true
  jms: true
```

5. Add the jms tpl file to the **app-uim** specification:

```
custom:
  enabled: true
  jms: true
  wdtFiles:
   - _custom-domain-model.tpl
   - _custom-jms-support.tpl
```

You can now create the UIM instance.

6. You can create the UIM instance with *customExtPath* as follows:

```
$COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
s $SPEC_PATH -a uim -m <customExtPath>
```

Adding a Store-and-Forward-Agent and SAF Resources

The WDT fragment that describes a SAF system resource is available in the **$COMMON_CNTK/samples/uim/customExtension/_custom-saf-support.tpl** sample file.

To include the WDT fragment in your UIM instance:

1. Enable the extension mechanism by setting the `custom` flag to `true` and add the custom-domain-model to the list of included WDT files in the **app-uim** specification as follows:

```
custom:
  enabled: true
  wdtFiles:
   - _custom-domain-model.tpl
```

2. Provide the WDT fragment by copying **$COMMON_CNTK/samples/uim/customExtensions/_custom-saf-support.tpl** to the **customExtPath** in your source control repository. See "Using the WDT Model Tools" for more information on setting the correct WDT.

> **Note**
>
> The sample file shows WDT for a SAF agent and SAF resources.

3. Set the parameters in the WDT fragment using the following commands.

> **Note**
>
> The fragment has parameters and uses values specified in the project file. By default, this WDT reads the SAF configuration values from the **app-uim.yaml** file that is provided during the instance creation.
> Kubernetes Secrets can also be used to provide sensitive data such as username and password. See "Accessing Kubernetes Secrets from WDT Metadata" for more information.

```
resources:
  SAFAgent:
    '<uim_saf_agent>':
        Store: '<inv_jms_store>'
        ServiceType: 'Sending-only'
        Target: '{{ default "c1" .Values.clusterName }}'
  JMSSystemResource:
    {{- range $destConfig := $.Values.safDestinationConfig }}
    'uim_{{- $destConfig.name -}}_saf_module':
        Target: '{{ default "c1" $root.Values.clusterName }}'
        SubDeployment:
          '<custom-subdeploy>':
            Target: '<uim_saf_agent>'
        JmsResource:
           SAFErrorHandling:
```

```
                'uim_{{$destConfig.name}}_saf_error':
                  LogFormat: "%header%, %properties%,%body%"
                  Policy: Log
               {{- if $destConfig.destinations }}
               SAFRemoteContext:
                'uim_{{$destConfig.name}}_saf_context':
                  SAFLoginContext:
                   Username: '@@SECRET:{{$destConfig.secretName}}:username@@'
                   PasswordEncrypted: '@@SECRET:
{{$destConfig.secretName}}:password@@'
                   LoginURL: '{{$destConfig.t3Url}}'
               SAFImportedDestinations:
                {{- range $destIndex, $dests := $destConfig.destinations }}
                'uim_{{$destConfig.name}}_saf_destinations_{{$destIndex}}':
                  SubDeploymentName: '<custom-subdeploy>'
                  JNDIPrefix: '{{- $dests.jndiPrefix -}}'
                  SAFErrorHandling: 'uim_{{$destConfig.name}}_saf_error'
                  SAFRemoteContext: 'uim_{{$destConfig.name}}_saf_context'
                  {{- if $dests.queues }}
                  SAFQueue:
                   {{- range $index, $queue := $dests.queues }}
                   'saf_queue_{{$index}}':
                     RemoteJndiName: '{{ $queue.queue.remoteJndi }}'
                     {{- if $queue.queue.localJndi }}
                     LocalJNDIName: '{{ $queue.queue.localJndi }}'
                     {{- end }} {{- /* end for if $queue.queue.localJndi  */
-}}
                {{- end }}    {{- /* end for range $dests.queues  */ -}}
                {{- end }}    {{- /* end for if $dests.queues */ -}}
                  {{- if $dests.topics }}
                  SAFTopic:
                   {{- range $index, $topic := $dests.topics }}
                   'saf_topic_{{$index}}':
                    UnitOfOrderRouting: Hash
                    NonPersistentQos: 'Exactly-Once'
                    RemoteJndiName: '{{ $topic.topic.remoteJndi }}'
                    {{- if $topic.topic.localJndi }}
                    LocalJNDIName: '{{ $topic.topic.localJndi }}'
                    {{- end }} {{- /* end for if $topic.topic.localJndi  */ -}}
                {{- end }}    {{- /* end for range $dests.topics  */ -}}
                {{- end }}    {{- /* end for if $dests.topics */ -}}
                {{- end }}      {{- /* end for range $destConfig.destinations
*/ -}}
                {{- end }}        {{- /* end for if $destConfig.destinations
*/ -}}
{{- end }}                  {{- /* end for
range .Values.safDestinationConfig */ -}}
```

4. The fragment is already configured for conditional loading based on the availability of the `saf` flag in the **app-uim** specification. Set the `saf` flag to `true` as follows:

```
custom:
  enabled: true
  saf: true
```

5. Add the **saf.tpl** file to the **app-uim** specification as follows:

```
custom:
  enabled: true
  saf: true
  wdtFiles:
   - _custom-domain-model.tpl
   - _custom-saf-support.tpl
```

6. Add the parameter values in the **app-uim.yaml** file as follows:

```
# When custom.saf is enabled, then below SAF destination configurations
will be used. Uncomment and provide the values here.
#safDestinationConfig: {} # This empty declaration should be removed if
adding items here.
safDestinationConfig:
  - name: <SAF Configuration Name>
    t3Url: <Remote Destination URL>
    secretName: <Secret Name of Remote Destination Credentials>
    destinations:
      - jndiPrefix: <JNDI Prefix>
        queues:
          - queue:
             localJndi: <Local Queue JNDI Name>
             remoteJndi: <Remote Queue JNDI Name>
```

7. Create the UIM instance with *customExtPath* as follows:

```
$COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
s $SPEC_PATH -a uim -m <customExtPath>
```

Deploying Entities to a UIM WebLogic Domain

You can deploy any WebLogic Server deployable entity, such as an application EAR or WAR to a UIM WebLogic domain. The deployment can be achieved in two ways:

- Packaging the application ear file in the Customized Image. See "Customizing Images" for more information on customizations.
- Using the extension mechanism.

To deploy an entity to a UIM WebLogic Domain using the extension mechanism:

1. Package the entity, for example, the application ear into an archive file and place it inside the container image used for creating UIM instances.

> **Note**
>
> The WebLogic domain tooling expects application binaries to be available at the correct path within the archive. A script is provided for your convenience that packages the application into the correct path.

```
cp application.ear samples/uim/customExtensions
cd samples/uim/customExtensions
./make-custom-archive.sh archive_file_name.zip application.ear plan.xml
```

2. Build a new container image:

```
cd samples/uim/customExtensions
docker build -t "image_name:tag" --build-arg base_image=uim_base_image --
build-arg archive=archive_file_name.zip .
```

3. (Optional) Build a new container image using Podman:

```
cd samples/uim/customExtensions
podman build -t "image_name:tag" --build-arg base_image=uim_base_image --
build-arg archive=archive_file_name.zip -f Dockerfile
```

4. Upload the generated image to your private Docker repository.
5. Add the domain configuration.
   In addition to copying the archive file into the base image, you must supply custom configuration. To use the extension mechanism:
   a. Enable the extension mechanism by setting the `custom` flag to **true** and add the **custom-domain-model** to the list of included wdtFiles in the **app-uim** specification:

```
custom:
  enabled: true
  wdtFiles:
   - _custom-domain-model.tpl
```

   b. Provide the WDT fragment by copying the following to the *customExtPath* in your source control repository.

```
cp $COMMON_CNTK/samples/uim/customExtensions/_custom-domain-model.tpl
customExtPath/
cp $COMMON_CNTK/samples/uim/customExtensions/_custom-application-
support.tpl customExtPath/
```

   c. Parameterize the WDT fragment. The fragment has already been parameterized.

```
appDeployments:
 Application:
 {{- .Values.custom.application_name }}:
 SourcePath: 'wlsdeploy/
applications/{{- .Values.custom.binary_name }}.ear'
 ModuleType: ear
 StagingMode: nostage
```

```
PlanStagingMode: nostage
Target: '@@PROP:CLUSTER_NAME@@'
```

d. Provide the values in the **app-uim** specification. The fragment is configured for conditional loading based on the presence of application flag in the **app-uim** specification. See **$COMMON_CNTK/charts/uim-app/charts/uim/templates/_custom-domain-model.tpl** in the toolkit.

```
custom:
  enabled: true
  application: true
  #additional values here
  application_name: myApplication
  binary_name: myApp
```

e. Add the application tpl file and update the image in the **app-uim** specification:

```
custom:
  enabled: true
  application: true
  wdtFiles:
   - _custom-domain-model.tpl
   - _custom-application-support.tpl
```

6. You can create the UIM instance with *customExtPath* as follows:

```
$COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
s $SPEC_PATH -a uim -m customExtPath
```

Deploying Mapviewer

This section describes the process for deploying MapViewer as an example of deploying an entity to a UIM WebLogic domain using the extension mechanism.

To deploy MapViewer:

1. Extract **mapviewer.ear** from the MapViewer installer JAR obtained from Oracle eDelivery: fmw_12.2.1.4.0_mapviewer.jar\Disk1\stage\Components\oracle.mapviewer\12.2.1.4.0\Data Files\filegroup1.jar\modules\oracle.mapviewer\
2. Extract the contents of **mapviewer.ear** and modify to **mapViewerConfig.xml** in **mapviewer.ear\web.war\WEB-INF\conf**:

```
Add below fragement within predefined datasource section after
map_data_source name="OracleMaps":
<map_data_source name="UIMDATA"
container_ds="jdbc/InventoryMapDataSource"
number_of_mappers="7" />
within <mds_config>, add below fragement:
<data_source name="UIMDATA">
<allow_predefined_themes>true</allow_predefined_themes>
<allow_dynamic_themes>true</allow_dynamic_themes>
</data_source>
```

3. Repackage **mapviewer.ear** with the updated **mapViewerConfig.xml**.

4. Package the updated **mapviewer.ear** file into an archive and add it in the container image to create the UIM instance:

```
 cp mapviewer.ear samples/customExtensions
cd samples/customExtensions
./make-custom-archive.sh mapviewer.zip mapviewer.ear
```

5. Build a new container image:

```
 cd samples/customExtensions
docker build -t "image_name:tag" --build-arg base_image=uim_base_image --
build-arg archive=mapviewer.zip
```

6. Upload the new container image to the repository.
7. Enable the custom extension mechanism by adding or updating the following entries in the project specification file, and update the image as follows:

```
custom:
enabled: true
application: true
application_name: MapViewer
binary_name: mapviewer
wdtFiles:
- _custom-domain-model.tpl
- _custom-application-support.tpl
```

8. Create **_custom-application-support.tpl** with the following content in **$UIM_CNTK/samples/customExtensions**:

```
 {{/* vim: set filetype=mustache: */}}
{{/* Copyright (c) 2021, Oracle and/or its affiliates. */}}
{{/*
Add application entities
*/}}
{{- define "uim.custom-application-support" -}}
appDeployments:
Application:
'MapViewer':
SourcePath: 'wlsdeploy/applications/{{- .Values.custom.binary_name }}.ear'
ModuleType: ear
StagingMode: nostage
PlanStagingMode: nostage
Target: '@@PROP:CLUSTER_NAME@@'
{{- end }}
```

9. Verify that the following fragment is already present in **$UIM_CNTK/samples/customExtensions/_custom-domain-model.tpl**:

```
 {{- if .Values.custom.application }}
custom-application-support.80.yaml: |+
{{- include "uim.custom-application-support" $root | nindent 2 }}
{{- end }}
```

10. Provide the WDT fragment by copying the following to the **customExtPath**:

```
 cp $UIM_CNTK/samples/customExtensions/_custom-domain-model.tpl
customExtPath/
cp $UIM_CNTK/samples/customExtensions/_custom-application-support.tpl
customExtPath/
```

11. Restart the instance:

```
$UIM_CNTK/scripts/restart-instance.sh -p project -i instance -s $SPEC_PATH
-m customExtPath -r full
```

MapViewer is deployed when the instance is restarted.

Extending the WDT Metadata for an External Authenticator

The Common cloud native toolkit provides out-of-the-box configuration for a WebLogic domain using OpenLDAP as the authenticator. Using a different provider (even a different LDAP provider) requires different WDT metadata, which is a significant undertaking. The configuration required to support an alternate WLS provider would need to be investigated and developed independently using an existing WebLogic domain. Oracle's WDT Discover Domain Tool can analyze an existing domain and generate the corresponding WDT model. The WDT model fragment can then be used to configure the UIM domain using the toolkit extension mechanism.

See the following documentation for information on configuring a WebLogic domain with alternative authentication providers:

- Configuring WebLogic to use LDAP
- Configuring Active Directory (AD) as an Authentication Provider in WebLogic

After the WDT is determined, it is provided during the creation process in the same way as other WDT metadata fragments. This section describes the process for setting up external authentication for UIM cloud native.

To set up external authentication:

1. Disable OpenLDAP by editing the **app-uim** specification in *specPath*:

```
authentication:
  openldap:
    enabled: false
```

2. Copy **$COMMON_CNTK/samples/uim/_custom-domain-model.tpl** to your source control repository at *customExtPath*.
3. Enable the extension mechanism by setting the `custom` flag to `true` in the **app-uim** specification and including the `_custom-domain-model.tpl`

```
custom:
  enabled: true
  wdtFiles:
   - _custom-domain-model.tpl
```

4. Determine and provide the WDT model fragment for the security provider in the WebLogic domain. Once you know the WDT fragment that needs to be supplied, save it into a file in your source control repository at the *customExtPath* (_custom-*provider*-support.tpl).

```
{{- define "uim.custom-provider-support" -}}
topology:
 SecurityConfiguration:
 Realm:
 myrealm:
 AuthenticationProvider:
 '!DefaultAuthenticator':
 '!DefaultIdentityAsserter':
 YouLDAPProviderStartHere:
 .... <specific details here>

 DefaultAuthenticator:
 DefaultAuthenticator:
 ControlFlag: SUFFICIENT
 UseRetrievedUserNameAsPrincipal: true
 DefaultIdentityAsserter:
 DefaultIdentityAsserter:
{{- end }}
```

> **Note**
>
> You can review the fragment for an OpenLDAP provider that is included in the toolkit: **$COMMON_CNTK/charts/uim-app/charts/uim/templates/_uim-openldap-support.tpl**

The security configuration WDT should respect sensitive data by using secrets. See "Accessing Kubernetes Secrets from WDT Metadata" for details on how to access secret data from within your WDT fragment.

5. (Optional) Update any parameters that should not be hard coded in the WDT fragment. Add these values to the **app-uim** specification under the "custom" section.

6. Load the model fragment by editing your *custom_extension_path*/ **_custom-domain-model.tpl** file:

```
{{- define "uim.custom-domain-model" -}}
{{- $root := . }}
custom-provider-support.index.yaml: |+
  {{- include "uim.custom-provider-support" $root | nindent 2 }}
{{- end }}


## If you would like conditional inclusion of the fragment...something
like this instead


{{- define "uim.custom-domain-model" -}}
{{- $root := . }}
{{- if .Values.custom.provider.flag}}
custom-provider-support.index.yaml: |+
  {{- include "uim.custom-<provider>-support" $root | nindent 2 }}
```

```
{{- end }}
{{- end }}
```

> **Note**
>
> Remember the yaml naming convention that is specified by wdt - filename.yaml. The index used determines the loading order when there are multiple yaml files. Indexes below 80 are reserved for internal Oracle use.

7. Add the tpl file that has the authentication provider WDT into the **app-uim** specification:

```
custom:
  enabled: true
  wdtFiles:
   - _custom-domain-model.tpl
   - _custom-provider-support.tpl
```

You can now create a UIM instance.

Extending WDT for Email Notification

When email notifications are enabled, event notifications are delivered to users at their configured email addresses. For more information, see "Extending Notifications" in *Developer's Guide*.

To configure email notifications, an SMTP server must be available and accessible from the UIM cloud native instance deployed on the cluster. Communication with the server occurs over the SMTP protocol. Therefore, ensure that SMTP traffic is permitted from your cluster, especially if the environment is behind a proxy.

To enable email notifications for UIM cloud native:

1. Create custom WDT extensions to configure email session as follows:
   a. Create a custom extension directory if not created:

```
mkdir -p $SPEC_PATH/$PROJECT/$INSTANCE/customExt
```

   b. Create a **_custom-mail-support.tpl** file in custom extension directory with the following content:

```
$ vim $SPEC_PATH/$PROJECT/$INSTANCE/customExt/_custom-mail-support.tpl

#_custom-mail-support.tpl file content below
{{- define "uim.custom-mail-support" -}}
{{- $root := . -}}
resources:
    MailSession:
        InventoryMailSession:
            JNDIName: mail/InventoryMailSession
            Target: '@@PROP:CLUSTER_NAME@@'
            Properties:
              mail.host: <Admin email Id>
              mail.smtp.host: <SMTP server>
              mail.smtp.port: <SMTP port>
```

```
                    mail.smtp.auth: <false/true>

    #Following Section is required only for Embedded LDAP,
    #In case of ExternalLDAP emails should be stored on LDAP Server
    #Start Embedded LDAP Users
    topology:
        Security:
            User:
        {{- range $inventoryUser := .Values.inventoryUsers }}
                '{{ $inventoryUser | replace "__" "@" }}':
                    Password: '@@SECRET:{{ $root.Values.name }}-uimcn-cred-
    uim:{{$inventoryUser}}@@'
                    GroupMemberOf: '@@SECRET:{{ $root.Values.name }}-uimcn-
    cred-uim:{{$inventoryUser}}_groups@@'
                    Description: 'Embedded LDAP User'
                    UserAttribute:
                      mail:
    {{ index $root.Values.custom.emails $inventoryUser }}
        {{- end }}
    #End Embedded LDAP Users
    {{- end }}
```

Provide appropriate values for the `<placeholders>` in the **_custom-mail-support.tpl** file. Based on the SMTP server configuration, you should specify additional properties such as `mail.user`, `mail.password`, `mail.smtp.ssl.enable`, and so on.

c. Copy the **_custom-domain-model.tpl** file from **$COMMON_CNTK/samples/uim/customExtensions** to **$SPEC_PATH/$PROJECT/$INSTANCE/customExt** and add the following entry to it:

```
{{- if .Values.custom.mail }}
custom-mail-support.90.yaml: |+
  {{- include "uim.custom-mail-support" $root | nindent 2 }}
{{- end }}
```

2. Enable the custom configuration and specify email addresses for all embedded LDAP users in the **$SPEC_PATH/$PROJECT/$INSTANCE/app-uim.yaml** file.

   a. Enable the custom flag, add new mail attribute, and provide the WDT file list.

   b. Provide email addresses for all provided embedded LDAP **inventoryUsers** as follows:

```
uim:
  #embedded ldap users list
  inventoryUsers:
    - user1
    - user2
    - user3

  custom:
    enabled: true
    mail: true
    #Map the embedded ldap users with their email address
    #Below emails will not be required in case of ExternalLDAP
    emails:
     user1: <user1 email address>
```

```
 user2: <user2 email address>
 user3: <user3 email address>
wdtFiles:
- _custom-domain-model.tpl
- _custom-mail-support.tpl
```

3. Add email template properties in **$SPEC_PATH/$PROJECT/$INSTANCE/config/uim/system-config/custom-config.properties**. The sample is as follows:

```
# Example template for ActivityAssignment

inventory.ActivityAssignmentEvent.message.template =
<html><body><pre><span style=\"font-family: tahoma, arial, helvetica, sans-serif; font-size: small;\">${notificationReceiver},<br /><br /> An activity named<strong> \"${activityName}\"</strong> activity from Work Order <strong>\"${workOrderName}\"</strong> has been assigned to you. <br /> <br /> Please note that this activity should start on <span style=\"color: #008000;\"><strong>${activityStartDate}</strong></span> and finish no later than <strong><span style=\"color: #ff0000;\">${activityEndDate}</span></strong>. </span><br /> <br /><span style=\"font-family: tahoma, arial, helvetica, sans-serif; font-size: small;\"> Login to <a href=\"${uimURL}\">Unified Inventory Management Application</a> to check the details. </span><br /><br /><br /><span style=\"font-family: tahoma, arial, helvetica, sans-serif; font-size: small;\"> Thanks,</span></pre><pre><span style=\"font-family: tahoma, arial, helvetica, sans-serif; font-size: small;\"> ${notificationOriginator}</span></pre></body></html>
```

4. Upgrade the UIM cloud native instance with custom extensions directory path argument as follows:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p $PROJECT -i $INSTANCE -s $SPEC_PATH -a uim -m $SPEC_PATH/$PROJECT/$INSTANCE/customExt
```

> **Note**
>
> For external LDAP, user emails should be managed directly on the LDAP server. You do not have to define them using the above configurations. Therefore, you can remove the related settings from the code, related to **uim.custom.emails** in **app-uim.yaml** and the topology section in **_custom-mail-support.tpl**.

Accessing Kubernetes Secrets from WDT Metadata

The process of handling sensitive data inside a WDT fragment involves the following:

- Creating Kubernetes secrets
- Declaring the secrets in the specification file
- Referencing the secrets from the WDT fragment

To access Kubernetes secrets from WDT metadata:

1. Create the secret.

Secrets must be created in the correct Kubernetes namespace. The namespace is already created when registering the namespace and aligns to your project name.

To create the secret using the command line, run the following command:

```
$kubectl -n project create secret generic secret_Name \
 --from-literal=key1=$value \
 --from-literal=key2=$value
```

2. Add the secret in the `custom` section of the **app-uim.yaml** specification in your source repository:

```
# Custom secrets
# replace the empty secret names with one or more secrets
instance:
  customSecrets:
    secretNames:
     - mysecret1
     - mysecret2
```

Once you have created and declared your custom secrets, they can be referenced from elsewhere in the WDT model.

3. Access the secret from inside a WDT fragment:

```
Field1: '@@SECRET:secret_name:key1@@'
Field2: '@@SECRET:secret_name:key2@@'
```

where *secret_name* represents the secret name and *key* represents one of the keys in the secret.

Troubleshooting WDT Issues

This section provides details about some procedures that you may have to run in order to resolve issues with WDT.

Starting and Terminating a WDT Pod

The UIM image includes the WDT tools that are often needed to debug or discover a WDT fragment. You can start a temporary pod that provides access to these tools. Before starting the pod, download the container image of the UIM base image to ensure that the download time does not exceed the duration of the Kubernetes pod creation timeout.

```
kubectl run wdt --generator=run-pod/v1 \
 --image UIM_base_image -- sleep infinity
```

When the pod is no longer needed, you can delete it:

```
kubectl delete pod wdt
```

Validating a Model YAML File

To validate a model YAML file:

1. Copy a model yaml into your temporary pod:

   ```
   kubectl cp model_file wdt:/tmp/model_file
   ```

2. Run the following command and wait for the prompt:

   ```
   kubectl exec -ti wdt /bin/bash
   ```

3. Validate the model file you copied:

   ```
   cd /u01/wdt/weblogic-deploy/bin
   ./validateModel.sh -oracle_home $ORACLE_HOME -model_file /tmp/model_file
   ```

4. When you are done validating, exit the pod:

   ```
   exit
   ```

The line numbers returned by the **validateModel** script are exclusive of the comment lines. Either strip the comments first or do the calculation to get the "real" line number in the file.

This process can be iterated by first reviewing the WDT errors and warnings, fixing the YAML file, and then re-running the above procedure. Repeat this as required.

> **Note**
>
> Model files can contain fragments of models, but each model element must have its full parentage, starting from `section`. For example, following is the sample if the fragment is the model element **JmsResource**:
>
> ```
> resources:
>  JMSSystemResource:
>  JmsResource:
>  model-fragment-to-validate
> ```

Displaying Valid Attributes and Child Attributes of a WDT Model

To display the attributes of a WDT model, run the following commands:

```
kubectl exec -ti wdt /bin/bash
# wait for prompt
cd /u01/wdt/weblogic-deploy/bin
./validateModel.sh -oracle_home $ORACLE_HOME \
 -print-usage path
exit
```

The *path* here is the WDT path to the model element of interest. For example, to see all the attributes and child attributes for SAFImportedDestinations, the path is `resources:/JMSSystemResource/JmsResource/SAFImportedDestinations`.

A common way to construct the path is to look for the element in a discovered model file and determine its yaml path. Another way is to start off with a path of `section:`, where `section`

is one of "domainInfo", "topology", "resources" or "appDeployments". By iteratively discovering the child attributes, the final path can be built-up.

To shorten this search process, add the `-recursive` flag to the validateModel.sh script command line. Care should be taken as the output can be quite large at the higher levels.

8
Exploring Alternate Configuration Options

The Common cloud native toolkit provides samples and documentation for setting up your UIM cloud native environment using standard configuration options. However, you can choose to explore alternate configuration options for setting up your environment, based on your requirements. This chapter describes alternate configurations you can explore, allowing you to decide how best to configure your UIM cloud native environment to suit your needs.

You can choose alternate configuration options for the following:

- Setting Up Authentication
- Enabling SAML Based Authentication Provider
- Enabling OAuth 2.0 Based Authentication Provider
- Working with Shapes
- Choosing Worker Nodes for Running UIM Cloud Native
- Working with Ingress, Ingress Controller, and External Load Balancer
- Using an Alternate Ingress Controller
- Reusing the Database State
- Setting Up Persistent Storage
- Managing Logs
- Managing UIM Cloud Native Metrics
- Managing WebLogic Monitoring Exporter (WME) Metrics

The sections that follow provide instructions for working with these configuration options.

Setting Up Authentication

By default, UIM uses the WebLogic embedded LDAP as the authentication provider. The UIM cartridge deployment users and application administrative users are created in embedded LDAP during instance creation. For human users, you may set up an optional authentication for the users who access UIM through user interfaces. See "Planning and Validating Your Cloud Environment" for information on the components that are required for setting up your cloud environment. The Common cloud native toolkit provides samples that you use to integrate components such as OpenLDAP, WebLogic Kubernetes Operator (WKO), and ingress controller. This section describes the tasks you must do for configuring optional authentication for UIM cloud native human users.

Perform the following tasks using the samples provided with the Common cloud native toolkit:

- Install and configure OpenLDAP. This is required to be done once for your organization.
- Install OpenLDAP clients. This is required to be performed on each host that installs and runs the toolkit scripts and when a Kubernetes cluster is shared by multiple hosts.
- In the OpenLDAP server, create the root node for each UIM instance.

Installing and Configuring OpenLDAP

OpenLDAP enables your organization to handle authentication for all instances of UIM. You install and configure OpenLDAP once for your organization.

To install and configure OpenLDAP:

1. Run the following command, which installs OpenLDAP:

```
$ sudo -s  yum -y install "openldap" "migrationtools"
```

2. Specify a password by running the following command:

```
$ sudo -s slappasswd
New password:
Re-enter new password:
```

3. Configure OpenLDAP by running the following commands:

```
$ sudo -s
$ cd /etc/openldap/slapd.d/cn=config
$ vi olcDatabase\=\{2\}hdb.ldif
```

4. Update the values for the following parameters:

> **Note**
>
> Ignore the warning about editing the file manually.

   - `olcSuffix: dc=uimcn-ldap,dc=com`
   - `olcRootDN: cn=Manager,dc=uimcn-ldap,dc=com`
   - `olcRootPW:`*ssha*
     where *ssha* is the SSHA that is generated

5. Update the **dc** values for the **olcAccess** parameter as follows:

```
olcAccess: {0}to * by
dn.base="gidNumber=0+uidNumber=0,cn=peercred,cn=external, cn=auth"
read by dn.base="cn=Manager,dc=uimcn-ldap,dc=com" read by * none
```

6. Test the configuration by running the following command:
   `sudo -s slaptest -u`

   Ignore the checksum warnings in the output and ensure that you get a success message at the end.

7. Run the following commands, which restart and enable LDAP:

```
sudo -s systemctl restart slapd
sudo -s systemctl enable slapd
sudo -s cp -rf /usr/share/openldap-servers/DB_CONFIG.example /var/lib/ldap/
DB_CONFIG
ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/cosine.ldif
ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/nis.ldif
ldapadd -Y EXTERNAL -H ldapi:/// -f /etc/openldap/schema/inetorgperson.ldif
```

8. Create a root node named **domain**, which will be the top parent for all UIM instances.

9. Run the following command to create a new file named **base.ldif**:

```
sudo -s  vi /root/base.ldif
```

10. Add the following entries to the **base.ldif** file:

```
dn: ou=Domains,dc=uimcn-ldap,dc=com
objectClass: top
objectClass: organizationalUnit
ou: Domains
```

11. Run the following commands to update the values in the **base.ldif** file:

```
ldapadd -x -W -D "cn=Manager,dc=uimcn-ldap,dc=com" -f /root/base.ldif
ldapsearch -x cn=Manager -b dc=uimcn-ldap,dc=com
```

12. Open the LDAP port 389 on all Kubernetes nodes in the cluster.

Installing OpenLDAP Clients

In environments where the Kubernetes cluster is shared by multiple hosts, you must install the OpenLDAP clients on each host. You use the scripts in the toolkit to populate the LDAP server with users and groups.

On the host on which you want to create a basic UIM instance, run the following command, which installs the OpenLDAP clients:

```
sudo -s yum -y install openldap-clients
```

Creating the Root Node

You must create the root node for each UIM instance before additional UIM non-automation user and UIM group can be created.

The toolkit provides a sample script (**$COMMON_CNTK/samples/credentials/managed-uim-ldap-credentials.sh**) that you can use to create the root node in the LDAP tree for the UIM instance.

Run the **$COMMON_CNTK/samples/credentials/managed-uim-ldap-credentials.sh** script by passing in **-o account**.

Enabling SAML Based Authentication Provider

The Common Cloud Native Tool Kit provides support for SAML-based authentication provider. This section describes the tasks you must do to configure an optional SAML-based authentication provider for UIM Cloud Native Deployment.

Prerequisite: Inventory application has to be registered with Authentication Provider to generate Metadata File which is required during UIM Image creation. In case, Authentication Provider is chosen to be **Identity Cloud Service**, see **Registering the Inventory Application in Identity Cloud Service** in the Knowledge article #Doc ID 2956673.1.

> **Note**
>
> If you have already configured one authentication provider, such as SAML or OAuth, you do not need another.

To enable SAML-based authentication provider, add the corresponding customizations to the **uim-cn-base** image and layered image as follows:

1. Extract **uim-app-archive.zip** for customization references.

   ```
   unzip workspace/uim-image-builder/staging/cnsdk/uim-model/uim-app-
   archive.zip -d workspace/customization
   ```

2. Export the following variables as required:

   ```
   mkdir workspace/temp
   export WORKSPACEDIR=$(pwd)/workspace
   export CUSTOMFOLDER=$(pwd)/workspace/customization
   export TEMPDIR=$(pwd)/workspace/temp
   export DMANIFEST=$(pwd)/workspace/uim-image-builder/bin/
   uim_cn_ci_manifest.yaml
   export STAGING=$(pwd)/workspace/uim-image-builder/staging
   ```

3. Update **$CUSTOMFOLDER/custom/plans/inventory-clusterPlan.xml** as follows:

   a. Change `logoutURL` address

      ```
      <variable>
            <name>logoutURL</name>
            <value>https://<instance>.<project>.uim.org:<LB_PORT>/
      saml2/sp/slo/init</value>
      </variable>
      ```

   b. Add a new variable assignment to **inv.war** and `weblogic-web-app` root element, in order to remove `cookie-path`:

      ```
      <module-override>
          <module-name>inv.war</module-name>
          <module-type>war</module-type>
          <module-descriptor external="false">
            <root-element>weblogic-web-app</root-element>
            <uri>WEB-INF/weblogic.xml</uri>
            <variable-assignment>
                  <name>cookie-path</name>
                  <xpath>/weblogic-web-app/session-descriptor/cookie-path</
      xpath>
                  <operation>remove</operation>
            </variable-assignment>
          </module-descriptor>
      ```

   c. Add a new module in order to remove `cookie-path` from **unified-topology-ui.war**:

      ```
      <module-override>
          <module-name>unified-topology-ui.war</module-name>
      ```

```
    <module-type>war</module-type>
    <module-descriptor external="false">
      <root-element>weblogic-web-app</root-element>
      <uri>WEB-INF/weblogic.xml</uri>
      <variable-assignment>
            <name>cookie-path</name>
            <xpath>/weblogic-web-app/session-descriptor/cookie-path</
xpath>
            <operation>remove</operation>
      </variable-assignment>
    </module-descriptor>
</module-override>
```

d. Add a new module in order to remove `cookie-path` for **InventoryRSOpenAPI.war**:

```
<module-override>
    <module-name>InventoryRSOpenAPI.war</module-name>
    <module-type>war</module-type>
    <module-descriptor external="false">
      <root-element>weblogic-web-app</root-element>
      <uri>WEB-INF/weblogic.xml</uri>
      <variable-assignment>
            <name>cookie-path</name>
            <xpath>/weblogic-web-app/session-descriptor/cookie-path</
xpath>
            <operation>remove</operation>
      </variable-assignment>
    </module-descriptor>
</module-override>
```

4. Update security files in **$CUSTOMFOLDER/custom/security/saml2**:

   a. Place Identity Provider (IdP) metadata file in **$CUSTOMFOLDER/custom/security/saml2/**.

   b. Copy **$CUSTOMFOLDER/custom/security/saml2/saml2idppartner.properties.sample** to **$CUSTOMFOLDER/custom/security/saml2/saml2idppartner.properties** and update the details of description and metadata file:

```
saml2.idp.partners=customidp
customidp.description=<IDP Partner>
customidp.metadata.file=<IDPMetadata.xml>
customidp.enabled=true
customidp.redirectUris=/Inventory/faces/*
customidp.virtualUserEnabled=true
```

5. Run the customization script and create UIM images, use `-c uim` as follows:

```
./workspace/uim-image-builder/bin/customization.sh
./workspace/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
s $STAGING -c uim
```

6. After you build the **uim-cn-base** image with layered tag, update the **app-uim.yaml** file as follows:

```
authentication:
  saml:
    enabled: true
    entityId: samlUIM #Use same entity id when configuring Idp Provider
```

7. Enable SSL incoming configuration on the UIM CN instance. See "Configuring Secure Incoming Access with SSL" for more information.
8. Create a UIM instance:

```
$COMMON_CNTK/scripts/create-applications.sh -p sr -i quick -s $SPEC_PATH -
a uim
```

> **Note**
>
> To Integrate UIM with ATA and Message Bus when authentication is enabled, update the **$SPEC_PATH/project/instance/config/uim/system-config/custom-config.properties** file with appropriate values. See "Checklists for Integration of Services" section from *Unified Inventory and Topology Deployment guide* for more information on integrating UIM with ATA and Message Bus.

Publishing UIM Cloud Native Service Provider Metadata File

If your identity provider supports SAML 2.0 client creation using the service provider metadata file, create a UIM metadata file in a cloud native environment as follows.

Open the pod sr-quick-admin and run following commands:

```
wlst.sh
connect('<weblogic-user-name>','<weblogic-password>','t3://sr-quick-ms1:8502')
serverRuntime()
cmo.getSingleSignOnServicesRuntime().publish('/logMount/UIMCNMetadata.xml',
false)
disconnect()
exit()
```

To copy the metadata file outside the pod (if PVC is enabled), the file is stored at pv-path locations by default. Alternatively, use the following command to copy to a required location:

```
kubectl cp sr-quick-admin:/logMount/UIMCNMetadata.xml ./UIMCNMetadata.xml -n
sr
```

Enabling OAuth 2.0 Based Authentication Provider

> **Note**
>
> If you have already configured one authentication provider, such as SAML or OAuth, you do not need another.

Prerequisite: Inventory application has to be registered with Authentication Provider to generate Metadata File which is required during UIM Image creation.

To enable OAuth-based authentication provider, add the corresponding customization to the **uim-cn-base** image and the layered image as follows:

1. Extract **uim-app-archive.zip** for customization references as follows:

   ```
   unzip workspace/uim-image-builder/staging/cnsdk/uim-model/uim-app-
   archive.zip -d workspace/customization
   ```

2. Export the following variables as required:

   ```
   mkdir workspace/temp
   export WORKSPACEDIR=$(pwd)/workspace
   export CUSTOMFOLDER=$(pwd)/workspace/customization
   export TEMPDIR=$(pwd)/workspace/temp
   export DMANIFEST=$(pwd)/workspace/uim-image-builder/bin/
   uim_cn_ci_manifest.yaml
   export STAGING=$(pwd)/workspace/uim-image-builder/staging
   ```

3. Update $CUSTOMFOLDER/custom/plans/inventory-clusterPlan.xml as follows:

   - Inside **<variable-definition>**, override the value of existing **endURL** variable with the IdP logout URL. For example: In case of Keycloak, it is **https://<keycloak-server>/realms/<realm>/protocol/openid-connect/logout?client_id=<CLIENT_ID>&post_logout_redirect_uri=https://<your-app-login-page>**:

     ```
     <variable>
           <name>endURL</name>
           <value>IdP_LOGOUT_URL_with_client_id_&_post_logout_redirect_uri</
     value>
     </variable>
     ```

   - Add new **variable-assignment** to **inv.war** and **weblogic-web-app** root element, to remove the cookie-path:

     ```
     <module-override>
         <module-name>inv.war</module-name>
         <module-type>war</module-type>
         <module-descriptor external="false">
           <root-element>weblogic-web-app</root-element>
           <uri>WEB-INF/weblogic.xml</uri>
           <variable-assignment>
                 <name>cookie-path</name>
     ```

```
                <xpath>/weblogic-web-app/session-descriptor/cookie-path</
xpath>
                <operation>remove</operation>
            </variable-assignment>
        </module-descriptor>
```

4. Copy **$CUSTOMFOLDER/custom/security/oidc/oidcAuth.properties.sample** to **$CUSTOMFOLDER/custom/security/oidc/oidcAuth.properties** and update the corresponding details:

   ```
   issuer=<oidc_issuer_url>
   clientId=<cliedt-id>
   clientSecret=<client-secret>
   redirectUrl=<http_or_https>://<Inventory_Host>:<Inventory_Port>/Inventory/
   faces/InventoryUIShell
   ```

   For more information, see "Preparing Web Applications for the OpenID Connect Provider" in *Administering Security for Oracle WebLogic Server*.

5. Run the customization script and create UIM images and use `-c uim`:

   ```
   $WORKSPACEDIR/uim-image-builder/bin/customization.sh
   ```

6. Modify image tags in the manifest yaml within the bin folder.
7. Add the JDK file, FMW infra file, FMW patches, WDT, and WIT zip files as mentioned in the manifest yaml to the staging folders.
8. Create UIM image as follows:

   ```
   $WORKSPACEDIR/uim-image-builder/bin/build-uim-images.sh -f $DMANIFEST -
   s $STAGING -c uim
   ```

Changing OAuth Configuration

To change OAuth configuration in **app-uim.yaml** file:

1. Change the **authentication.oauth.enabled** flag to **true** in the **$SPEC_PATH/$PROJECT/$INSTANCE/app-uim.yaml** file.
2. Restart UIM to apply the changes and test.

Working with Shapes

The Common cloud native toolkit provides the following pre-configured shapes. After assembling specifications, you can see the specification files at the following locations:

- **spec_path/project/instance/shapes/dev/uim.yaml**. This can be used for development, QA and user acceptance testing (UAT) instances.
- **spec_path/project/instance/shapes/devsmall/uim.yaml**. This can be used to reduce CPU requirements for small development instances.
- **spec_path/project/instance/shapes/prod/uim.yaml**. This can be used for production, pre-production, and disaster recovery (DR) instances.

- **spec_path/project/instance/shapes/prodlarge/uim.yaml**. This can be used for production, pre-production and disaster recovery (DR) instances that require more memory for UIM cartridges and order caches.
- **spec_path/project/instance/shapes/prodsmall/uim.yaml**. This can be used to reduce CPU requirements for production, pre-production and disaster recovery (DR) instances. For example, it can be used to deploy a small production cluster with two managed servers when the input request rate does not justify two managed servers configured with a **prod** or **prodlarge** shape. For production instances, Oracle recommends two or more managed servers. This provides increased resiliency to a single point of failure and can allow order processing to continue while failed managed servers are being recovered.

You can create custom shapes using the pre-configured shapes. See "Creating Custom Shapes" for details.

The pre-defined shapes come in standard sizes, which enable you to plan your Kubernetes cluster resource requirement.

The following table lists the sizing requirements of the shapes for a managed server:

Table 8-1 Sizing Requirements of Shapes for a Managed Server

Shape	Kube Request	Kube Limit	JVM Heap (GB)
prodlarge	80 GB RAM, 15 CPU	80 GB RAM, 15 CPU	64
prod	48 GB RAM, 15 CPU	48 GB RAM, 15 CPU	31
prodsmall	48 GB RAM, 7.5 CPU	48 GB RAM, 7.5 CPU	31
dev	8 GB RAM, 2 CPU	8 GB RAM	5
devsmall	N/A	N/A	5

The following table lists the sizing requirements of the shapes for an admin server:

Table 8-2 Sizing Requirements of Shapes for an Admin Server

Shape	Kube Request	Kube Limit	JVM Heap (GB)
prodlarge	8 GB RAM, 2 CPU	8 GB RAM	4
prod	8 GB RAM, 2 CPU	8 GB RAM	4
prodsmall	8 GB RAM, 2 CPU	8 GB RAM	4
dev	3 GB RAM, 1 CPU	3 GB RAM	1
devsmall	N/A	N/A	1

These values are encoded in the specifications and are automatically part of the individual pod configuration. The Kubernetes schedulers evaluate the Kube request settings to find space for each pod in the worker nodes of the Kubernetes cluster.

To plan the cluster capacity requirement, consider the following:

- Number of development instances required to be running in parallel: D
- Number of managed servers expected across all the development instances: Md (Md will be equal to D if all the development instances are 1 MS instances)
- Number of production (and production-like) instances required to be running in parallel: P
- Number of managed servers expected across all production instances: Mp
- Assume use of "dev" and "prod" shapes
- CPU requirement (CPUs) = D * 1 + Md * 2 + P * 2 + Mp * 15

- Memory requirement (GB) = D * 4 + Md * 8 + P * 8 + Mp * 48

> **Note**
>
> The production managed servers take their memory and CPU in large chunks. Kube scheduler requires the capacity of each pod to be satisfied within a particular worker node and does not schedule the pod if that capacity is fragmented across the worker nodes.

The shapes are pre-tuned for generic development and production environments. You can create a UIM instance with either of these shapes, by specifying the preferred one in the instance specification.

```
# Name of the shape. The UIM cloud native shapes are devsmall, dev,
prodsmall, prod, and prodlarge.
# Alternatively, custom shape name can be specified (as the filename without
the extension)
```

Creating Custom Shapes

You create custom shapes by copying the provided shapes and then specifying the desired tuning parameters. Do not edit the values in the shapes provided with the toolkit.

In addition to processor and memory sizing parameters, a custom shape can be used to tune:

- The number of threads allocated to UIM work managers
- UIM connection pool parameters

To create a custom shape:

1. Create a directory with the desired shape name at **$SPEC_PATH/project/instance/shapes/**.
2. Copy one of the pre-configured shapes and save it to the above shape directory of your source repository.
3. Update the tuning parameters as required in uim.yaml under newly create shape directory.
4. In the **applications-base.yaml** specification, specify the name of the shape you copied and renamed:

   ```
   shape: custom
   ```

5. Create the domain, ensuring that the location of your custom shape is included in the comma separated list of directories passed with `-s`.

   ```
   $COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
   s $SPEC_PATH -a uim
   ```

> **Note**
>
> While copying a pre-configured shape or editing your custom shape, ensure that you preserve any configuration that has comments indicating that it must not be deleted.

Choosing Worker Nodes for Running UIM Cloud Native

By default, UIM cloud native has its pods scheduled on all worker nodes in the Kubernetes cluster in which it is installed. However, in some situations, you may want to choose a subset of nodes where pods are scheduled.

For example, these situations include:

- Limitation on the deployment of UIM on specific worker nodes per team for reasons such as capacity management, chargeback, budgetary reasons, and so on.

To choose a subset of nodes where pods are scheduled, you can use the configuration in the **app-uim** specification file.

```
# If UIM CN instances must be targeted to a subset of worker nodes in the
# Kubernetes cluster, tag those nodes with a label name and value, and choose
# that label+value here.
#   keys:
#    - key    : any node label key
#    - operator : Valid operators are In, NotIn,Exists, DoesNotExist. Gt, and
Lt.
#    - values : values is an array of string values.
#               If the operator is In or NotIn, the values array must be non-
empty.
#               If the operator is Exists or DoesNotExist, the values array
must be empty (values can be removed from below).
#               If the operator is Gt or Lt, the values array must have a
single element, which will be interpreted as an integer.
#
# This can be overriden in instance specification if required.
# Node Affinity can be achieved by operator "In" and Node Anti-Affinity by
"NotIn"
# oracle.com/licensed-for-coherence is just an indicative example; any
# label and its values can be used for choosing nodes.
uimcnTargetNodes: {} # This empty declaration should be removed if adding
items here.
#uimcnTargetNodes:
#  nodeLabel:
#   keys:
#    - key: oracle.com/licensed-for-coherence
#      operator: In
#      values:
#       - true
#    - key: failure-domain.beta.kubernetes.io/zone
#      operator: NotIn
#      values:
#       - PHX-AD-2
#       - PHX-AD-3
```

Consider the following when you update the configuration:

- There is no restriction on node label key. Any valid node label can be used.
- There can be multiple valid values for a key.
- You can override this configuration in the instance specification yaml file, if required.

Examples

In the following example, pods are created on the nodes that have keys as **failure-domain.beta.kubernetes.io/zone** and the values as **PHX-AD-2** or **PHX-AD-3**:

```
# Example1
#uimcnTargetNodes: {}
uimcnTargetNodes:
  nodeLabel:
   keys:
    - key: failure-domain.beta.kubernetes.io/zone
      operator: In
      values:
      - PHX-AD-2
      - PHX-AD-3
```

In the following example, pods are created on the nodes that do not have keys as **name** and have keys as **failure-domain.beta.kubernetes.io/zone** and the values as neither **PHX-AD-2** nor **PHX-AD-3**.

```
# Example2
# uimcnTargetNodes: {} # This empty declaration should be removed if adding
items here.
uimcnTargetNodes:
  nodeLabel:
   keys:
    - key: name
      operator: DoesNotExist
    - key: failure-domain.beta.kubernetes.io/zone
      operator: NotIn
      values:
      - PHX-AD-2
      - PHX-AD-3
```

Working with Ingress, Ingress Controller, and External Load Balancer

A Kubernetes ingress is responsible for establishing access to back-end services. However, creating an ingress is not sufficient. An Ingress controller connects the back-end services with the front-end services that are external to Kubernetes through edge objects such as NodePort services, Load Balancers, and so on. In UIM cloud native, an ingress controller can be configured in the **applications-base** and **app-uim** specifications.

UIM cloud native supports annotation-based **generic ingress** creation that uses standard Kubernetes Ingress API as verified by Kubernetes Conformance tests. This can be used for any Kubernetes certified ingress controller, if that ingress controller offers annotations that are usually proprietary to the ingress controller, required for UIM. Annotations applied to an ingress resource allow you to use features such as connection timeout, URL rewrite, retry, additional headers, redirects, sticky cookie services, and so on, and to improve the performance of that ingress resource. The ingress controllers support a corresponding set of annotations. For information on annotations that are supported by your ingress controller and the list of various ingress controllers, see https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.

Any ingress controller, which conforms to the standard Kubernetes ingress API and supports annotations needed by UIM should work, although Oracle does not certify individual ingress controllers to confirm this **generic** compatibility.

These samples are provided to use HAProxy ingress controllers. For information about HAProxy Ingress Controller, see https://github.com/haproxytech/kubernetes-ingress/blob/master/README.md

The configurations required in your **app-uim** specification are as follows:

Update the **application-base.yaml** file from **$SPEC_PATH/project/instance** to use generic ingress as follow:

```
# valid values are GENERIC
ingressController: "GENERIC"
```

You need to provide the following annotations to enable cookies that can meet the hardware sizing requirements.

Provide an appropriate `ingressClassName` value for your ingress controller under the **ingress.className** field. Based on the value provided, an ingress object is created for that ingress class as follows:

```
ingress:
  className: haproxy ##provide ingressClassName value, default value for
haproxy ingressController is haproxy.
  annotations:
    haproxy.org/cookie-persistence: "uimhaproxycookie"
```

Annotations for Enabling SSL

Update the **application-base** specification to enable the SSL and also provide `loadbalancerport` value with appropriate LoadBalancerPort/NodePort of your Ingress Controller as follows:

```
loadbalancerport: 30543
tls:
  enabled: true
```

Update the **app-uim.yaml** specification file from **$SPEC_PATH/project/instance** to add annotations as follows:

```
uim:
   ingress:
    annotations:
      haproxy.org/ssl-redirect: "true"
      haproxy.org/backend-config-snippet: |
        http-request del-header WL-Proxy-Client-IP
        http-request del-header WL-Proxy-SSL
        http-request set-header X-Forwarded-Proto https
        http-request set-header WL-Proxy-SSL true
```

For more information on trust and identity provided in the above configuration, see "Setting Up Secure Communication with SSL".

Using an Alternate Ingress Controller

By default, UIM cloud native supports standard Kubernetes ingress API and provides sample files for integration. If your required ingress controller does not support one or more configurations through annotations on generic ingress, or you use your ingress controller's CRD instead, you can choose `"OTHER"`.

By choosing this option, UIM cloud native does not create or manage any ingress required for accessing the UIM cloud native services. However, you may choose to create your own ingress objects based on the service and port details mentioned in the tables that follow. The toolkit uses an ingress Helm chart (**$COMMON_CNTK/charts/uim-ingress/charts/uim-ingress/templates/generic-ingress.yaml**) and scripts for creating the ingress objects. If you want to use a generic ingress controller, these samples can be used as a reference and customized as necessary.

The host-based rules and the corresponding back-end Kubernetes service mapping are provided using the following definitions:

- domainUID: Combination of *project-instance*. For example, **sr-quick**.
- clusterName: The name of the cluster in lowercase. Replace any hyphens "-" with underscore "_". The default name of the cluster is **uimcluster**.

The following table lists the service name and service ports for Ingress rules:

Table 8-3 Service Name and Service Ports for Ingress Rules

Rule	Service Name	Service Port	Purpose
instance.project.loadBalancerDomainName	*domainUID*-cluster-*clusterName*	8502	For access to UIM through UI, Web Services, and so on.
t3.*instance.project.loadBalancerDomainName*	*domainUID*-cluster-*clusterName*	30303	UIM T3 Channel access for WLST, JMS, and SAF clients.
admin.*instance.project.loadBalancerDomainName*	*domainUID*-admin	8501 8504 (if ssl reencrypt strategy is enabled)	For access to UIM WebLogic Admin Console UI.

Ingresses need to be created for each of the above rules per the following guidelines:

- Before running **create-applications.sh**, ingress must be created.
- After running **delete-applications.sh**, ingress must be deleted.

You can develop your own code to handle your ingress controller or copy the sample `ingress-per-domain` chart and add additional template files for your ingress controller with a new value for the type (for example, NGINX).

- The reference sample for creation is: **$COMMON_CNTK/scripts/create-ingress.sh**
- The reference sample for deletion is: **$COMMON_CNTK/scripts/delete-ingress.sh**

> **Note**
>
> Regardless of the choice of Ingress controller, it is mandatory to provide the value of `loadbalancerPort` in one of the specification files. This is used for establishing front-end cluster.

Reusing the Database State

When a UIM instance is deleted, the state of the database remains unaffected, which makes it available for re-use. This is common in the following scenarios:

- When an instance is deleted and the same instance is re-created using the same project and the instance names, the database state is unaffected. For example, consider a performance instance that does not need to be up and running all the time, consuming resources. When it is no longer actively being used, its specification files and PDB can be saved and the instance can be deleted. When it is needed again, the instance can be rebuilt using the saved specifications and the saved PDB. Another common scenario is when developers delete and re-create the same instance multiple times while configuration is being developed and tested.
- When a new instance is created to point to the data of another instance with a new project and instance names, the database state is unaffected. A developer, who might want to create a development instance with the data from a test instance in order to investigate a reported issue, is likely to use their own instance specification and the UIM data from PDB of the test instance.

Additionally, consider the following components when re-using the database state:

- The UIM DB (schema and data)
- The RCU DB (schema and data)

Recreating an Instance

You can re-create a UIM instance with the same project and instance names, pointing to the same database. In this case, both the UIM DB and the RCU DB are re-used, making the sequence of events for instance re-creation relatively straightforward.

To recreate an instance, the following pre-requisites must be available from the original instance and made available to the re-creation process:

- PDB
- The project and instance specification files

Reusing the UIM Schema

To reuse the UIM DB, the secret for the PDB must still exist:

project-instance`-database-credentials`

project-instance`-database-credentials.`

This is the `uimdb` credential in the **manage-app-credentials.sh** script.

Reusing the RCU

To reuse the RCU, the following secrets for the RCU DB must still exist:

- *project-instance*`-rcudb-credentials`. This is the `rcudb` credential.
- *project-instance*`-opss-wallet-password-secret`. This is the `opssWP` credential.
- *project-instance*`-opss-walletfile-secret`. This is the `opssWF` credential.

To import the wallet file from previous installation of UIM:

1. Run the WLST *exportEncryptionKey* command in the previous domain where the RCU is been referenced. This generates *ewallet.p12* file.
2. Export the wallet file for generating OPSS wallet on the existing schema and use it to create UIM CN instance as follows:
   a. Connect to the previous UIM WebLogic domain that refers the RCU schemas as follows:

      ```
      $cd<FWM_HOME>/oracle_common/common/bin>$./wlst.shwls:/offline>
      exportEncryptionKey(jpsConfigFile, keyFilePath, keyFilePassword)
      Export of Encryption key(s) is done. Remember the password chosen, it
      will be required while importing the key(s)
      ```

      Where:

      - *keyFilePassword* is same as the OPSS wallet file password that is used during the secrets creation.
      - *jpsConfigFile* specifies the location of **jps-config.xml** file that corresponds to the location where the command is processed.
      - *keyFilePath* specifies the directory where **ewallet.p12** file is created. The content of this file is encrypted and secured by the value passed to *keyFilePassword*.
      - *keyFilePassword* specifies the password to secure **ewallet.p12** file. This password must be used while importing the file.

   b. Download the generated **ewallet.p12** file from **TEMP_LOCATION** folder and copy it into the Kubernetes worker node in **$SPEC_PATH**.
3. Convert the generated ewallet file to Base64 encoded format as follows:

   ```
   $cd $SPEC_PATH
   $base64 ewallet.p12 > ewalletbase64.P12
   ```

4. Create OPSSWF secret using the Base64 ewallet as follows:

   ```
   $COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -a
   uim create opssWF
   ```

5. Enter the Base64 ewallet file location: **$SPEC_PATH/ewalletbase64.p12**.

   > **Note**
   >
   > For `opssWP` and `wlsRTE`, use the same password that you used while exporting the wallet file.

6. Create the instance as you would normally do:

   ```
   $COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
   s $SPEC_PATH -a uim
   ```

> **Note**
>
> If the `opssWP` and `opssWF` secrets no longer exist and cannot be re-created from offline data, then drop the RCU schema and re-create it using the UIM DB Installer.

Creating a New Instance

If the original instance does not need to be retained, then the original PDB can be re-used directly by a new instance. If however, the instance needs to be retained, then you must create a clone of the PDB of the original instance. This section describes using a newly cloned PDB for the new instance.

If possible, ensure that the images specified in the app-uim specification (app-uim.yaml) match the images in the specification files of the original instance.

Reusing the UIM Schema

To reuse the UIM DB, the following secret for the PDB must be created using the new project and instance names. This is the `uimdb` credential in **manage-app-credentials.sh** and points to your cloned PDB:

```
project-instance-database-credentials
```

If your new instance must reference a newer UIM DB installer image in its specification files than the original instance, it is recommended to invoke an in-place upgrade of UIM schema before creating the new instance.

To upgrade or check the UIM schema:

```
# Upgrade the UIM schema to match new application's specification files
# Do nothing if schema already matches
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -s $SPEC_PATH
-a uim -c 3
```

> **Note**
>
> If the current instance details are different than the previous instance, to reuse the UIM schema, drop the table with suffix **WL_LLR_**.

You can choose a strategy for the RCU DB from one of the following options:

- Create a new RCU
- Reuse RCU

Creating a New RCU

If you only wish to retain the UIM schema data, then you can create a new RCU schema.

The following steps provide a consolidated view of RCU creation described in "Managing Configuration as Code".

To create a new RCU, create the following secrets:

- *project-instance*`-rcudb-credentials`. This is the `rcudb` credential and describes the new RCU schema you want in the clone.
- *project-instance*`-opss-wallet-password-secret`. This is the `opssWP` credential unique to your new instance

After these credentials are in place, prepare the cloned PDB:

```
# Create a fresh RCU DB schema while preserving UIM schema data
$COMMON_CNTK/scripts/install-database.sh -p project -i instance -s $SPEC_PATH
-a uim -c 2
```

With this approach, the RCU schema from the original instance is still available in the cloned PDB, but is not used by the new instance.

Reusing the RCU

Using the **manage-app-credentials.sh** script, create the following secret using your new project and instance names:

```
project-instance-rcudb-credentials
```

The secret should describe the old RCU schema, but with new PDB details.

- **Reusing RCU Schema Prefix**

  Over time, if PDBs are cloned multiple times, it may be desirable to avoid the proliferation of defunct RCU schemas by re-using the schema prefix and re-initializing the data. There is no UIM metadata stored in the RCU DB so the data can be safely re-initialized.

  project-instance`-opss-wallet-password-secret`. This is the `opssWP` credential unique to your new instance.

  To re-install the RCU, invoke DB Installer:

  ```
  $COMMON_CNTK/scripts/install-database.sh -p project -i instance -
  s $SPEC_PATH -a uim -c 2
  ```

- **Reusing RCU Schema and Data**

  In order to reuse the full RCU DB from another instance, the original `opssWF` and `opssWP` must be copied to the new environment and renamed following the convention: *project-instance*-**opss-wallet-password-secret** and *project-instance*-**opss-walletfile-secret**.

  This directs Fusion MiddleWare OPSS to access the data using the secrets.

Create the instance as you would normally do:

```
$COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Setting Up Persistent Storage

UIM cloud native can be configured to use a Kubernetes Persistent Volume to store data that needs to be retained even after a pod is terminated. This data includes application logs, JFR recordings and DB Installer logs, but does not include any sort of UIM state data. When an instance is re-created, the same persistent volume need not be available. When persistent

storage is enabled in the instance specification, these data files, which are written inside a pod are re-directed to the persistent volume.

Data from all instances in a project may be persisted, but each instance does not need a unique location for logging. Data is written to a *project-instance* folder, so multiple instances can share the same end location without destroying data from other instances.

The final location for this data should be one that is directly visible to the users of UIM cloud native. The development instances may simply direct data to a shared file system for analysis and debugging by cartridge developers. Whereas, formal test and production instances may need the data to be scraped by a logging toolchain such as EFK, that can then process the data and make it available in various forms. The recommendation therefore is to create a PV-PVC pair for each class of destination within a project. In this example, one for developers to access and one that feeds into a toolchain.

A PV-PVC pair would be created for each of these "destinations", that multiple instances can then share. A single PVC can be used by multiple UIM domains. The management of the PV and PVC lifecycles is beyond the scope of UIM cloud native.

The UIM cloud native infrastructure administrator is responsible for creating and deleting PVs or for setting up dynamic volume provisioning.

The UIM cloud native project administrator is responsible for creating and deleting PVCs as per the standard documentation in a manner such that they consume the pre-created PVs or trigger the dynamic volume provisioning. The specific technology supporting the PV is also beyond the scope of UIM cloud native. However, samples for PV supported by NFS are provided.

Creating a PV-PVC Pair

The technology supporting the Kubernetes PV-PVC is not dictated by UIM cloud native. Samples have been provided for NFS and BV, and can either be used as is, or as a reference for other implementations.

To create a PV-PVC pair supported by NFS:

1. Edit the sample PV and PVC yaml files and update entries with enclosing brackets

   > **Note**
   >
   > PVCs need to be ReadWriteMany.

   ```
   vi $COMMON_CNTK/samples/nfs/pv.yaml
   vi $COMMON_CNTK/samples/nfs/pvc.yaml
   ```

2. Create the Kubernetes PV and PVC.

   ```
   kubectl create -f $COMMON_CNTK/samples/nfs/pv.yaml
   kubectl create -f $COMMON_CNTK/samples/nfs/pvc.yaml
   ```

3. Set up the storage volume. The storage volume type is `emptydir` by default.

   ```
   storageVolume:
      enabled: true
      pvc: sr-nfs-pvc
   ```

```
#  pvc: storage-pvc #Specify this only if case type is PVC
isBlockVolume: false # set this to true if BlockVolume is used
```

Deleting the pod that has storage volume disabled deletes the corresponding logs. To retain the logs:

- Uncomment **storageVolume**.
- Set the **storageVolume.enabled** to **true**.
- Specify the name of the pvc created.

```
# The storage volume must specify the PVC to be used for persistent
storage.
storageVolume:
  enabled: true
  pvc: sr-nfs-pvc #Specify this only if case type is PVC
```

After the instance is created, you should see the following directories in your PV mount point, if you have enabled logs:

```
[oracle@localhost project-instance]$ dir
server, UIM, uim-dbinstaller
```

To create a PV-PVC pair supported by BV:

1. Edit the sample PV and PVC yaml files and update entries with enclosing brackets:

   ```
   vi $COMMON_CNTK/samples/bv/pv.yaml
   vi $COMMON_CNTK/samples/bv/pvc.yaml
   ```

2. Create the Kubernetes PV and PVC as follows:

   ```
   kubectl create -f $COMMON_CNTK/samples/bv/pv.yaml
   kubectl create -f $COMMON_CNTK/samples/bv/pvc.yaml
   ```

3. Repeat step 1 and 2 to create PV-PVCs required for all the servers such as introspector, admin, db-installer, and for each managed server.

   > **Note**
   >
   > Do not provide `<server-name>` in the prefix for db-installer PV-PVC.

4. To use Block Volume:

   a. Set `StorageVolume.enabled` to **true**.

   b. Uncomment `#pvc: storage-pvc` and replace `storage-pvc` with the appropriate suffix of all the PVCs, which is same as the name of db-installer PVC.

   c. Set `StorageVolume.isBlockVolume` to **true**.

   ```
   storageVolume:
   enabled: true # Acceptable values are pvc and emptydir
   pvc: <project>-<storage-endpoint>-bv-pvc #this is equal to suffix of pvc
   ```

```
and equal to pvc used by db-installer
isBlockVolume: true # set this to true if BlockVolume is used
```

5. Change permissions of **blockVolume** using **initContainer**. By default, **initContainerImage** is commented. Uncomment it and mention the corresponding image name that you want to use.

```
#uncomment this to use initContainer for introspector,admin,db-
installer,ms pods and change permission of mount volume
initContainerImage: "container-registry.oracle.com/os/oraclelinux:8-slim"
```

Managing Logs

UIM cloud native generates traditional textual logs. By default, these log files are generated in the managed server pod, but can be re-directed to a Persistent Volume Claim (PVC) supported by the underlying technology that you choose. See "Setting Up Persistent Storage" for details.

By default, logging is enabled. When persistent storage is enabled, logs are automatically re-directed to the Persistent Volume. The storage volume type is `disabled` by default.

```
storageVolume:
  enabled: false # Acceptable values are pvc and emptydir
  pvc: storage-pvc #Specify this only if case type is PVC
```

Deleting the pod that has storage volume type as `disabled` deletes the corresponding logs. To retain the logs:

- Uncomment **storageVolume**.
- Set the **storageVolume.enabled** to **true**.
- Specify the name of the pvc created.

```
# The storage volume must specify the PVC to be used for persistent storage.
storageVolume:
enabled: true # Acceptable values are pvc and emptydir
pvc: storage-pvc #Specify this only if case type is PVC
```

- The UIM application logs can be found at: *pv-directory*/*project-instance*/*UIM*/logs
- The UIM WebLogic server logs can be found at: *pv-directory*/*project-instance*/server
- The UIM DB Installer logs can be found at: *pv_directory*/*project-instance*/*uim-dbinstaller*/logs

Viewing Logs using Fluentd and OpenSearch Dashboard

You can view and analyze the UIM cloud native logs using Fluentd and OpenSearch dashboard.

The logs are generated as follows:

1. Fluentd collects the text logs that are generated during cloud native deployments and sends them to OpenSearch.
2. OpenSearch collects all types of logs and converts them into a common format so that OpenSearch dashboard can read and display the data.

3. OpenSearch dashboard reads the data and presents it in a simplified view.

Setting up OpenSearch Dashboard and Fluentd

To set up OpenSearch Dashboard and Fluentd:

1. Set up OpenSearch and OpenSearch dashboard. See "Setting Up OpenSearch" in *Unified Inventory and Topology Deployment Guide* for more information.
2. Update the following in **app-uim.yaml** to enable the sidecar injection:

```
sidecar:
  enabled: true
  containers:
    - template: "fluentd-container"
      volumeTemplate: "fluentd-configmap-volume"
      containerFiles:
        - fluentd-config-map.yaml
        - _fluentd-sidecar-container.tpl
```

3. Update the values for **FLUENT_OPENSEARCH_HOST**, **FLUENT_OPENSEARCH_PORT**, **OPENSEARCH_USER**, and **OPENSEARCH_PASSWORD** in $COMMON_CNTK/samples/uim/customExtensions/sidecar-fluentd/_fluentd-sidecar-container.tpl.
4. (Optional) Update the FluentD ConfigMap file in the **customExtensions** folder to add customizations for selecting or adding any required logs.
5. In the Kubernetes pod, create an instance with sidecar injection as follows:

```
$COMMON_CNTK/scripts/create-applications.sh -p <project_name> -i
<instance_name> - s $SPEC_PATH -m <path to customExtentions dir/sidecar-
fluentd> -a uim
```

To access the logs on OpenSearch dashboard, create an Index Pattern as follows:

1. Click on the **Three Bars** icon.
2. Under OpenSearch Dashboards, select **Discover**.
3. Create a new Index Pattern using `index <project>-<instance>`.
   The logs can be accessed on the **Discover** page under `<project>-<instance> index`.

Enabling GC Logs

You can monitor the Java garbage collection data by using GC logs. By default, these GC logs are disabled and you can enable them to view the logs at **/logMount/<domain>/servers/<server-name>**.

To enable the GC logs, update **<applications-base.yaml>** or **<app-uim.yaml>** from **$SPEC_PATH/project/instance** as follows:

1. Under **gcLogs** make `enabled` as **true**.
2. To configure the maximum size of each file and limit for number of files, set `fileSize` and `noOfFiles` inside **gcLogs**.

```
gcLogs:
  enabled: true
```

```
        fileSize: 10M
        noOfFiles: 10
```

WebLogic Diagnostic Logs

In UIM cloud native, the WebLogic diagnostic logs are stored (by default) at **/u01/oracle/user_projects/domains/domain/servers/ms1/logs/**, inside the pod. To retain these logs after a pod goes down, redirect them to a storage volume by uncommenting the specified property in **app-uim.yaml**. Optionally, you can also specify an additional logger class.

```
uim:
  log:
    handlerLevel: "ERROR"
  #  # This is to optionally control logging level for specific classes.
Uncomment to add the entries.
  #  # 'class' will have full ClassName e.g. com.mslv.oms.poller.EventPoller
  #  # 'level' will have same possible values as above e.g. ERROR:1
  #  #loggers:
  #  #  - class:
  #  #    level:
```

Managing UIM Cloud Native Metrics

Authentication is enabled for UIM metrics. Perform the following steps to create Kubernetes secret for the metrics URL. After you create the secret, configure the metrics scrape job in Prometheus operator for the UIM cloud native by following the instructions mentioned in "Configuring Metrics For Services" in *Unified Inventory and Topology Deployment Guide*.

Creating Secret for UIM Metrics Authentication

To enable **Basic Authentication** for Prometheus scrape metrics job from UIM service, you should be creating a Kubernetes secret with user credentials as follows:

1. Create a Kubernetes secret named `<project>-<instance>-uim-metrics-user`.
2. Create this secret in the same namespace where **Prometheus Operator** is deployed.
3. Ensure that the secret includes **username** and **password** that you use to access the UIM homepage.
4. Ensure that the specified username is part of the **uim-metrics-users** group, which is configured during the creation of the **Embedded LDAP user secret**.

See "Creating Users in Embedded LDAP" for more information.

Configuring Prometheus for UIM Cloud Native Metrics

The following job configuration has to be added to Prometheus configuration, replace the username & password for the UIM metric endpoint:

```
- job_name: 'uimcn'
  kubernetes_sd_configs:
  - role: pod
  relabel_configs:
  - source_labels:
```

```
['__meta_kubernetes_pod_annotationpresent_uimcn_metricspath']
    action: 'keep'
    regex: 'true'
  - source_labels: [__meta_kubernetes_pod_annotation_uimcn_metricspath]
    action: replace
    target_label: __metrics_path__
    regex: (.+)
  - source_labels: ['__meta_kubernetes_pod_annotation_prometheus_io_scrape']
    action: 'drop'
    regex: 'false'
  - source_labels: [__address__,
__meta_kubernetes_pod_annotation_uimcn_metricsport]
    action: replace
    regex: ([^:]+)(?::\d+)?;(\d+)
    replacement: $1:$2
    target_label: __address__
  #- action: labelmap
  #  regex: __meta_kubernetes_pod_label_(.+)
  - source_labels: ['__meta_kubernetes_pod_label_weblogic_serverName']
    action: replace
    target_lable: server_name
  - source_labels: ['__meta_kubernetes_pod_label_weblogic_clusterName']
    action: replace
    target_label: cluster_name
  - source_labels: [__meta_kubernetes_pod_name]
    action: replace
    target_label: pod_name
  - source_labels: [__meta_kubernetes_namespace]
    action: replace
    target_label: namespace
  basic_auth:
    username: <METRICS_UESR_NAME>
    password: <PASSWORD>
```

> **Note**
>
> UIM cloud native has been tested with Prometheus and Grafana installed and configured using the Helm chart **prometheus-community/kube-prometheus-stack** available at: https://prometheus-community.github.io/helm-charts.

Viewing UIM Cloud Native Metrics Without Using Prometheus

The metrics URL is enabled with **BASIC** authentication, provide the credentials in the dialog box. The user must have **uim-metrics-group** associated. The UIM cloud native metrics can be viewed at: :

`http://instance.project.domain_Name:LoadBalancer_Port/Inventory/metrics`

By default, `hostSuffix` is set to **uim.org** and can be modified in **applications-base.yaml**. This only provides metrics of the managed server that is serving the request. It does not provide consolidated metrics for the entire cluster. Only Prometheus Query and Grafana dashboards can provide consolidated metrics.

Viewing UIM Cloud Native Metrics in Grafana

UIM cloud native metrics scraped by Prometheus can be made available for further processing and visualization. The UIM cloud native toolkit comes with sample Grafana dashboards to get you started with visualizations.

Import the dashboard JSON files from **$COMMON_CNTK/samples/uim/grafana** into your Grafana environment.

The sample dashboards are:

- UIM by Services: Provides a view of UIM cloud native metrics for one or more instances in the selected managed server.

Exposed UIM Service Metrics

The following UIM metrics are exposed via Prometheus APIs.

> **Note**
>
> - All metrics are per managed server. Prometheus Query Language can be used to combine or aggregate metrics across all managed servers.
> - All metric values are short-lived and indicate the number of requests in a particular state since the managed server was last restarted.
> - When a managed server restarts, all the metrics are reset to **0**.

Interaction Metrics

The following table lists interaction metrics exposed via Prometheus APIs.

Table 8-4 Interaction Metrics Exposed via Prometheus APIs

Name	Type	Help Text	Notes
uim_sfws_capture_requests	Summary	Summary that tracks the duration of `sfws` capture requests.	This metric is observed for the **CaptureInteraction** request. The action can be **CREATE** or **CHANGE**.
uim_sfws_process_requests	Summary	Summary that tracks the duration of `sfws` process requests.	This metric is observed for the **ProcessInteraction** request. The action can be **PROCESS**.
uim_sfws_update_requests	Summary	Summary that tracks the duration of `sfws` update requests.	This metric is observed for the **UpdateInteraction** request. The action can be **APPROVE**, **ISSUE**, **CANCEL**, **COMPLETE** or **CHANGE**.

Table 8-4 (Cont.) Interaction Metrics Exposed via Prometheus APIs

Name	Type	Help Text	Notes
uim_sfws_requests	Summary	Summary that tracks the duration of `sfws` requests.	This metric is observed for the capture, process, and update interaction requests.

Labels For All Interaction Metrics

The following table lists labels for all interaction metrics.

Table 8-5 Labels for All Metrics

Label Name	Sample Value
action	The values can be **CREATE**, **CHANGE**, **APPROVE**, **CANCEL**, **COMPLETE**, and **CANCEL**.

Service Metrics

The following metrics are captured for completion of a business interaction.

Table 8-6 Service Metrics Captured for Completion of a Business Interaction

Name	Type	Help Text	Summary
uim_services_processed	Counter	Counter that tracks the number of services processed.	This metric is observed for suspend, resume, complete, and cancel of a service.

Labels for all Service Metrics

A task metric has all the labels that a service metric has.

Table 8-7 Labels for All Service Metrics

Label	Sample Value	Notes	Source of Label
spec	VoipServiceSpec	The service specification name.	UIM Metric Label Name/Value
status	IN_SERVICE	The service status. The values can be **IN_SERVICE**, **SUSPEND**, **DISCONNECT**, and **CANCELLED**.	UIM Metric Label Name/Value

Generic Labels for all Metrics

Following are the generic labels for all metrics:

Table 8-8 Generic Labels for all Metrics

Label Name	Sample Value	Source of the Label
server_name	ms1	Prometheus Kubernetes SD

Table 8-8 (Cont.) Generic Labels for all Metrics

Label Name	Sample Value	Source of the Label
job	cmcn	Prometheus Kubernetes SD
namespace	sr	Prometheus Kubernetes SD
pod_name	ms1	WebLogic Operator Pod Label
weblogic_cluseterName	uimcluster	WebLogic Operator Pod Label
weblogic_clusterRestartVersion	v1	WebLogic Operator Pod Label
weblogic_createdByOperator	true	WebLogic Operator Pod Label
weblogic_domainName	domain	WebLogic Operator Pod Label
weblogic_domainRestartVersion	v1	WebLogic Operator Pod Label
weblogic_domainUID	quicksr	WebLogic Operator Pod Label

Managing WebLogic Monitoring Exporter (WME) Metrics

UIM cloud native provides a sample Grafana dashboard that you can use to visualize WebLogic metrics available from a Prometheus data source.

You use the WebLogic Monitoring Exporter (WME) tool to expose WebLogic server metrics. WebLogic Monitoring Exporter is part of the WebLogic Kubernetes Toolkit. It is an open source project, based at: https://github.com/oracle/weblogic-monitoring-exporter. You can include WME in your UIM cloud native images. Once a UIM cloud native image with WME is generated, creating a UIM cloud native instance with that image automatically deploys a WME WAR file to the WebLogic server instances. While WME metrics are available through WME Restful Management API endpoints, UIM cloud native relies on Prometheus to scrape and expose these metrics. This version of UIM supports WME 1.3.0. See WME documentation for details on configuration and exposed metrics.

The following topics provide a sample integration:

- Generating the WME WAR File
- Deploying the WME WAR File
- Configuring the Prometheus Scrape Job for WME Metrics
- Viewing WebLogic Monitoring Exporter Metrics in Grafana

Generating the WME WAR File

To generate the WME WAR file, run the following commands, which update the **wls-exporter.war** WAR file with the **exporter-config.yaml** configuration file.

```
mkdir -p ~/wme
cd ~/wme

curl -x $http_proxy -L https://github.com/oracle/weblogic-monitoring-exporter/
releases/download/v1.3.0/wls-exporter.war -o wls-exporter.war
curl -x $http_proxy https://raw.githubusercontent.com/oracle/weblogic-
monitoring-exporter/v1.3.0/samples/kubernetes/end2end/dashboard/exporter-
config.yaml -o config.yml

jar -uvf wls-exporter.war config.yml
```

Deploying the WME WAR File

After the WME WAR file is generated and updated, you can deploy it as a custom application archive.

For details about deploying entities, see "Deploying Entities to a UIM WebLogic Domain".

You can use the following sample to deploy the WME WAR file to the admin server and the managed servers in a cluster:

```
appDeployments:
    Application:
        'wls-exporter':
            SourcePath: 'wlsdeploy/applications/wls-exporter.war'
            ModuleType: war
            StagingMode: nostage
            PlanStagingMode: nostage
            Target: '@@PROP:ADMIN_NAME@@ , @@PROP:CLUSTER_NAME@@'
```

Configuring the Prometheus Scrape Job for WME Metrics

The following job configuration has to be added to Prometheus configuration:

> **Note**
>
> In the `basic_auth` section, specify the WebLogic username and password.

```
- job_name: 'basewls'
    kubernetes_sd_configs:
    - role: pod
    relabel_configs:
    - source_labels: ['__meta_kubernetes_pod_annotation_prometheus_io_scrape']
      action: 'keep'
      regex: 'true'
    - source_labels: [__meta_kubernetes_pod_label_weblogic_createdByOperator]
      action: 'keep'
      regex: 'true'
    - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
      action: replace
      target_label: __metrics_path__
      regex: (.+)
    - source_labels: [__address__,
__meta_kubernetes_pod_annotation_prometheus_io_port]
      action: replace
      regex: ([^:]+)(?::\d+)?;(\d+)
      replacement: $1:$2
      target_label: __address__
    - action: labelmap
      regex: __meta_kubernetes_pod_label_(.+)
    - source_labels: [__meta_kubernetes_pod_name]
      action: replace
      target_label: pod_name
```

```
- source_labels: [__meta_kubernetes_namespace]
  action: replace
  target_label: namespace
basic_auth:
  username: weblogic_username
  password: weblogic_password
```

Viewing WebLogic Monitoring Exporter Metrics in Grafana

WebLogic Monitoring Exporter metrics scraped by Prometheus can be made available for further processing and visualization. The UIM cloud native toolkit comes with sample Grafana dashboards to get you started with visualizations. The WebLogic server dashboard provides a view of WebLogic Monitoring Exporter metrics for one or more managed servers for a given instance in the selected project namespace.

Import the sample dashboard **weblogic_dashboar.json** file from https://github.com/oracle/weblogic-monitoring-exporter/blob/master/samples/kubernetes/end2end/dashboard into your Grafana environment by selecting Prometheus as the data source.

9

Integrating UIM

Typical usage of UIM involves the UIM application coordinating activities across multiple peer systems. Several systems interact with UIM for various purposes. This chapter examines the considerations involved in integrating UIM cloud native instances into a larger solution ecosystem.

This chapter describes the following topics and tasks:

- Integration with UIM cloud native
- Configuring SAF
- Applying the WebLogic patch for external systems
- Configuring SAF for External Systems
- Setting up Secure Communication with SSL

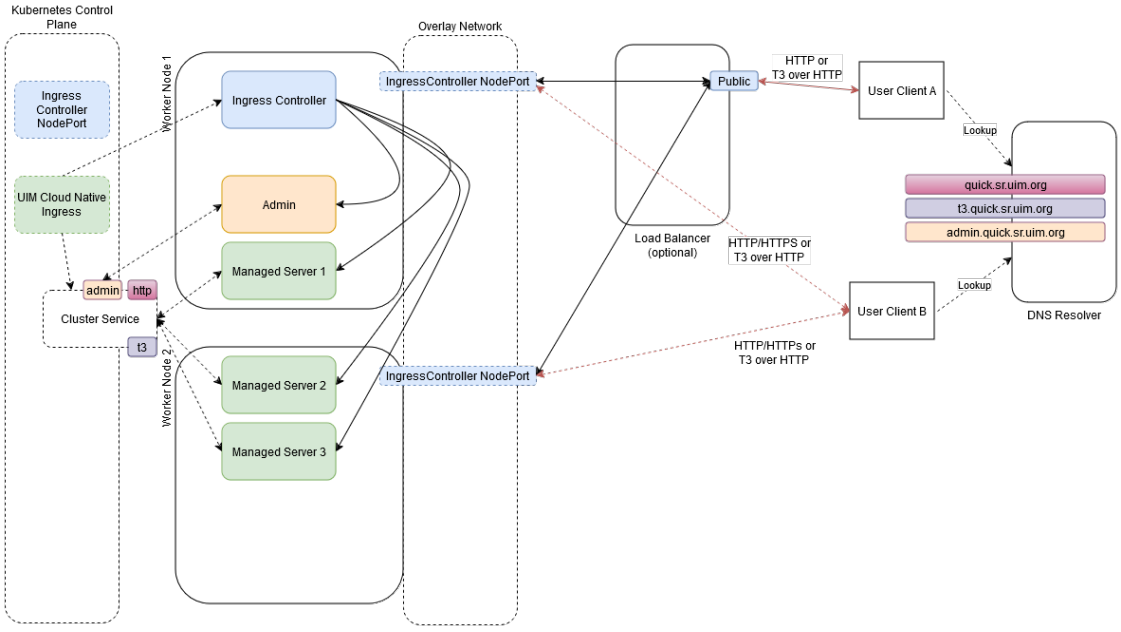
Integrating with UIM Cloud Native

Functionally, the integration requirements of UIM do not change when UIM is running in a cloud native environment. All of the categories of integrations that are applicable to traditional UIM instances are applicable and must be supported for UIM cloud native.

Connectivity Between the Building Blocks

The following diagram illustrates the connectivity between the building blocks in a UIM cloud native environment using an example:

Figure 9-1 Integration Across Building Blocks in UIM Cloud Native Environment



Invoking the UIM cloud native Helm chart creates a new UIM instance. In the above illustration, the name of the instance is "quick" and the name of the project is "sr". The instance consists of the WebLogic cluster that has one Admin Server and three Managed Servers and a Kubernetes Cluster Service.

The Cluster Service contains endpoints for both HTTP and T3 traffic. The ingress creation script creates the UIM cloud native Ingress object. The Ingress object has metadata to trigger the Generic ingress controller as a sample. Ingress controller responds by creating new front-ends with the configured "hostnames" for the cluster (**quick.sr.uim.org** and **t3.quick.sr.uim.org** in the illustration) and the admin server (**admin.quick.sr.uim.org**) and links them up to new back-end constructs. Each back-end routes to each member of the Cluster Service (**MS1**, **MS2**, and **MS3** in the example) or to the Admin Server. The **quick.sr.uim.org** front-end is linked to the back-end pointing to the HTTP endpoint of each managed server, while the **t3.quick.sr.uim.org** front-end links to the back-end pointing to the T3 endpoint of each managed server.

The prior installation of Ingress Contoller has already exposed HAProxy itself through a selected port number on each worker node.

Inbound HTTP Requests

A UIM instance is exposed outside of the Kubernetes cluster for HTTP access via an Ingress Controller and potentially a Load Balancer.

Because the Ingress Controller port is common to all UIM cloud native instances in the cluster, Ingress Controller must be able to distinguish between the incoming messages headed for different instances. It does this by differentiating on the basis of the "hostname" mentioned in the HTTP messages. This means that a client (User Client B in the illustration) must believe it is talking to the "host" **quick.sr.uim.org** when it sends HTTP messages to Port on the access IP. This might be the Master node IP, or IP address of one of the worker nodes, depending on your cluster setup. The "DNS Resolver" provides this mapping.

In this mode of communication, there are concerns around resiliency and load distribution. For example, If the DNS Resolver always points to the IP address of Worker Node 1 when asked to resolve **quick.sr.uim.org**, then that Worker node ends up taking all the inbound traffic for the instance. If the DNS Resolver is configured to respond to any ***.sr.uim.org** requests with that IP, then that worker node ends up taking all the inbound traffic for all the instances. Since this latter configuration in the DNS Resolver is desired, to minimize per-instance touches, the setup creates a bottleneck on Worker node 1. If Worker node 1 were to fail, the DNS Resolver would have to be updated to point ***.sr.uim.org** to Worker node 2. This leads to an interruption of access and requires intervention. The recommended pattern to avoid these concerns is for the DNS Resolver to be populated with all the applicable IP addresses as resolution targets (in our example, it would be populated with the IPs of both Worker node 1 and node 2), and have the Resolver return a random selection from that list.

An alternate mode of communication is to introduce a load balancer configured to balance incoming traffic to the Ingress Controller ports on all the worker nodes. The DNS Resolver is still required, and the entry for ***.sr.uim.org** points to the load balancer. Your load balancer documentation describes how to achieve resiliency and load management. With this setup, a user (User Client A in our example) sends a message to **quick.sr.uim.org**, which actually resolves to the load balancer - for instance, **http://sr.quick.uim.org:8080/Inventory/faces/login.jspx**. Here, **8080** is the public port of the load balancer. The load balancer sends this to Ingress Controller, which routes the message, based on the "hostname" targeted by the message to the HTTP channel of the UIM cloud native instance.

By adding the hostname resolution such that **admin.quick.sr.uim.org** also resolves to the Kubernetes cluster access IP (or Load Balancer IP), User Client B can access the WebLogic

console via **http://admin.quick.sr.uim.org/console** and the credentials specified while setting up the "wlsadmin" secret for this instance.

> **Note**
>
> Access to the WebLogic Admin console is provided for review and debugging use only. Do not use the console to change the system state or configuration. These are maintained independently in the WebLogic Operator, based on the specifications provided when the instance was created or last updated by the UIM cloud native toolkit. As a result, any such manual changes (whether using the console or using WLST or other such mechanisms) are liable to be overwritten without notice by the Operator. The only way to change state or configuration is through the tools and scripts provided in the toolkit.

Inbound JMS Requests

JMS messages use the T3 protocol. Since Ingress Controllers and Load Balancers do not understand T3 for routing purposes, UIM cloud native requires all incoming JMS traffic to be "T3 over HTTP". Hence, the messages are still HTTP, but contain a T3 message as payload. UIM cloud native requires the clients to target the "t3 hostname" of the instance - **t3.quick.sr.uim.org**, in the example. This "t3 hostname" should behave identically as the regular "hostname" in terms of the DNS Resolver and the Load Balancer. Ingress Controller however not only identifies the instance this message is meant for (quick.sr) but also that it targets the T3 channel of instance.

The "T3 over HTTP" requirement applies for all inbound JMS messages - whether generated by direct or foreign JMS API calls or generated by SAF. The procedure in SAF QuickStart explains the setup required by the message producer or SAF agent to achieve this encapsulation. If SAF is used, the fact that T3 is riding over HTTP does not affect the semantics of JMS. All the features such as reliable delivery, priority, and TTL, continue to be respected by the system. See "Applying the WebLogic Patch for External Systems" for more information.

A UIM instance can be configured for secure access, which includes exposing the T3 endpoint outside the Kubernetes cluster for HTTPS access. See "Configuring Secure Incoming Access with SSL" for details on enabling SSL.

Inbound JMS Requests Within the Same Kubernetes Cluster

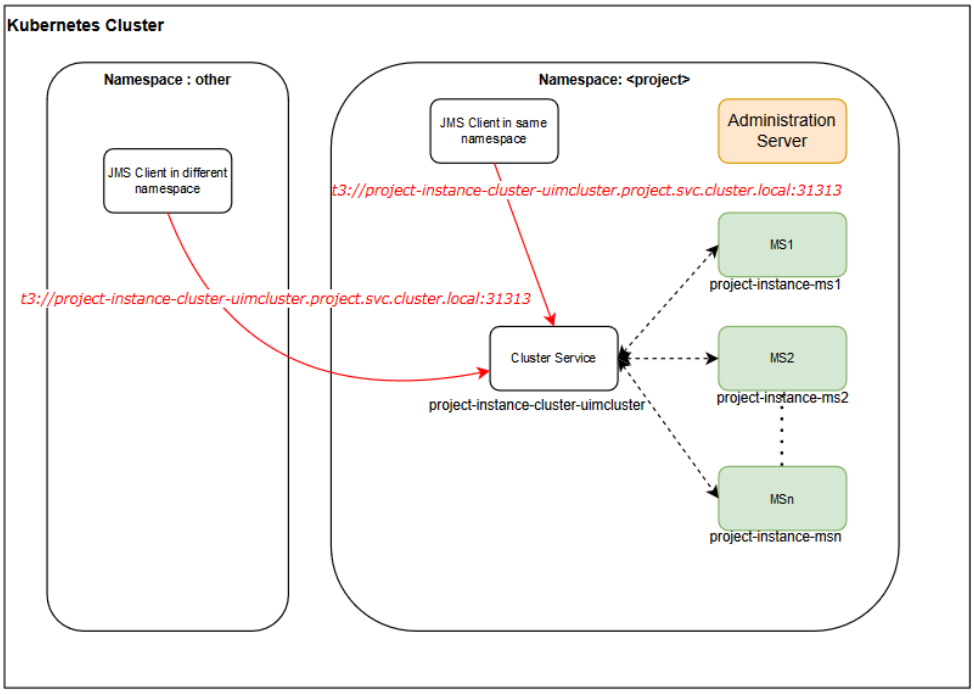
There can be situations where UIM cloud native needs to be accessed from within the same Kubernetes cluster where it is deployed. For example, in a Service and Network Orchestration (SNO) an upstream application (OSM) and downstream application UIM could be deployed in the same Kubernetes cluster. For such requirements, there is no need for the request to be routed via an Ingress Controller or a load balancer and resolved via a DNS Resolver.

UIM cloud native exposes a T3 channel exclusively for such connections and can be accessed via **t3://*project-instance*-cluster-uimcluster.project.svc.cluster.local:31313**.

This saves the various network hops typically involved in routing a request from an external client to UIM cloud native deployed in a Kubernetes cluster. The following diagram illustrates inbound JMS requests within the same Kubernetes cluster using an example. For the example, the URL is **t3://sr-quick-cluster-uimcluster.sr.svc.cluster.local:31313**.

> **Note**
>
> The protocol is T3 as there is no need for wrapping in HTTP; the port is different.

Figure 9-2 Inbound JMS Integration in a Kubernetes Cluster



If SSL is enabled for domains, communication between the domains within the Kubernetes cluster is not secured because the ingress is not involved. See "Setting Up Secure Communication with SSL" for further details.

Outbound HTTP Requests

No specific action is required to ensure the HTTP messages from UIM cloud native instance reach out of the Kubernetes Cluster.

When a domain inside a Kubernetes cluster sends REST API or Web Service requests over HTTP to a domain that is outside the cluster that is enabled with SSL, then you should set up some required configuration. For instructions, see "Configuring Access to External SSL-Enabled Systems".

Outbound JMS Connectivity

JMS messages originating from the UIM cloud native instance such as requests to peer systems always end up on local queues. The UIM cloud native Helm chart allows for the specification of SAF connections to remote systems in order to get these messages to their destinations. Custom Templates can be used to create SAF connections in UIM. This allows for a canonical expression of the SAF connectivity requirements, which are uniquely fulfilled by

each project by pointing to the appropriate upstream, downstream, peer systems or emulators, and so on.

When a domain inside a Kubernetes cluster sends JMS messages to a domain that is outside the cluster that is SSL-enabled, then see "Configuring Access to External SSL-Enabled Systems" for instructions on setting up some required configuration.

Configuring SAF

UIM cloud native requires SAF to send messages to external systems through JMS. The SAF configuration in UIM cloud native is configured at **app-uim** specification level. The **app-uim** specification can be used to define all the SAF connections that any UIM cloud native instance must make. Each of these SAF connections must be given a specific remote endpoint. See "Adding a Store-and-Forward-Agent and SAF Resources" for more information on configuring SAF templates.

Configuring the app-uim Specification

The **app-uim** specification lists out all the SAF connections and endpoint for each of these SAF connections that are required. These are listed under the **safDestinationConfig** element of the **app-uim** specification. The following sample shows a basic SAF specification that describes the need to interact with **external_system_identifier** through SAF. The **app-uim** specification contains the T3 URL of the external system along with the name of a Kubernetes secret that provides the credentials required to interact with that system. The T3 URL can be specified using any of the standard mechanisms supported by WebLogic. The Kubernetes secret must contain the fields **username** and **password**, carrying credentials which have permissions to include JMS messages into the remote system. It specifies that the project accesses two queues on that remote system: **remote_queue_1** and **remote_queue_2**. These queues can be addressed using the JNDI prefix **prefix_1** on the system. Further, **remote_queue_1** is also mapped locally as **local_queue_1**. The mapping depends on the addressing system coded into the UIM cartridge's external sender automation plugins. UIM cloud native supports both local names and remote names for SAF destinations.

If the external system is a UIM cloud native instance deployed in the same Kubernetes cluster, use the T3 URL as described "Inbound JMS Requests Within the Same Kubernetes Cluster".

If SSL is enabled for the external system, use the T3 URL as described in "Configuring Access to External SSL-Enabled Systems".

```
safDestinationConfig:
  - name: external_system_identifier
    t3Url: t3_url
    secretName: secret_t3_user_pass
    destinations:
      - jndiPrefix: prefix_1
        queues:
         - queue:
            remoteJndi: remote_queue_1
            localJndi: local_queue_1
         - queue:
            remoteJndi: remote_queue_2
```

If the queues of an external system are spread across more than one JNDI prefix, the **jndiPrefix** element can be repeated as many times as necessary. In this example, **prefix_1** applies to **remote_queue_1** and **remote_queue_2**, while **prefix_2** applies to remote_queue_3.

The following sample shows SAF **app-uim** specification with multiple JNDIs:

```
safDestinationConfig:
    - name: external_system_identifier
      t3Url: t3_url
      secretName: secret_t3_user_pass
      destinations:
        - jndiPrefix: prefix_1
          queues:
            - queue:
                remoteJndi: remote_queue_1
                localJndi: local_queue_1
            - queue:
                remoteJndi: remote_queue_2
        - jndiPrefix: prefix_2
          queues:
            - queue:
                remoteJndi: remote_queue_3
```

It is possible for an external system to not use a JNDI prefix, which is configured by leaving the value empty for jndiPrefix. However, at most, one of the jndiPrefix entries in a destinations list can be empty, as the jndiPrefixes in this list have to be unique. If there are more than one external system that the project's solution cartridges interact with via SAF, these can be named and listed as follows:

```
safDestinationConfig:
    - name: external_system_identifier_1
      t3Url: t3_url
      secretName: secret_t3_user_pass
      destinations:
        - jndiPrefix: prefix_1
          queues:
            - queue:
                remoteJndi: remote_queue_1
    - name: external_system_identifier_2
      t3Url: t3_url
      secretName: secret_t3_user_pass
      destinations:
        - jndiPrefix: prefix_2
          queues:
            - queue:
                remoteJndi: remote_queue_2
```

> **Note**
>
> Using the provided configuration, UIM cloud native automatically computes names for some entities required for completing the SAF setup. You may find such entities when you log into WebLogic Administration Console for troubleshooting purposes and are not to be confused.

Configuring Domain Trust

For details about global trust, see "Enabling Global Trust" in *Oracle Fusion Middleware Administering Security for Oracle WebLogic Server*.

Because the shared password provides access to all domains that participate in the trust, strict password management is critical. Trust should be enabled when SAF is configured as it is needed for inter-domain communication using distributed destinations. In a Kubernetes cluster where the pods are transient, it is possible that a SAF sender will not know where it can forward messages unless domain trust is configured.

If trust is not configured when using SAF, you may experience unstable SAF behavior when your environment has pods that are growing, shrinking, or restarting.

To enable domain trust, in your instance specification file, for `domainTrust`, change the default value to **true**:

```
domainTrust:
 enabled: true
```

If you are enabling domain trust, then you must create a Kubernetes secret (exactly as specified) to store the shared trust password by running the following command:

> **Note**
>
> This step is not required if you are not enabling domain trust in the **app-uim** specification.

```
kubectl create secret generic -n project project-instance-global-trust-
credentials --from-literal=password=pwd
```

The same password must be used in all domains that connect to this one through SAF.

Applying the WebLogic Patch for External Systems

When an external system is configured with a SAF sender towards UIM cloud native, using HTTP tunneling, a patch is required to ensure the SAF sender can connect to the UIM cloud native instance. This is regardless of whether the connection resolves to an ingress controller or to a load balancer. Each such external system that communicates with UIM through SAF must have the WebLogic patch **30656708** installed and configured, by adding `-Dweblogic.rjvm.allowUnknownHost=true` to the WebLogic startup parameters.

For environments where it is not possible to apply and configure this patch, a workaround is available. On each host running a Managed Server of the external system, add the following entries to the **/etc/hosts** file:

```
0.0.0.0 project-instance-ms1
0.0.0.0 project-instance-ms2
0.0.0.0 project-instance-ms3
0.0.0.0 project-instance-ms4
0.0.0.0 project-instance-ms5
0.0.0.0 project-instance-ms6
0.0.0.0 project-instance-ms7
0.0.0.0 project-instance-ms8
0.0.0.0 project-instance-ms9
```

```
0.0.0.0 project-instance-ms10
0.0.0.0 project-instance-ms11
0.0.0.0 project-instance-ms12
0.0.0.0 project-instance-ms13
0.0.0.0 project-instance-ms14
0.0.0.0 project-instance-ms15
0.0.0.0 project-instance-ms16
0.0.0.0 project-instance-ms17
0.0.0.0 project-instance-ms18
```

You should add these entries for all the UIM cloud native instances that the external system interacts with. Set the IP address to **0.0.0.0**. All the managed servers possible in the UIM cloud native instance must be listed regardless of how many are actually configured in the instance specification.

Configuring SAF on External Systems

To create SAF and JMS configuration on your external systems to communicate with the UIM cloud native instance, use the configuration samples provided as part of the SAF sample as your guide.

It is important to retain the "Per-JVM" and "Exactly-Once" flags as provided in the sample.

All connection factories must have the "Per-JVM" flag, as must SAF foreign destinations.

Each external queue that is configured to use SAF must have its QoS set to "Exactly-Once".

Enabling Domain Trust

To enable domain trust, in your domain configuration, under **Advanced**, edit the **Credential** and **ConfirmCredential** fields with the same password you used to create the global trust secret in UIM cloud native.

Setting Up Secure Communication with SSL

When UIM cloud native is involved in secure communication with other systems, either as the server or as the client, you should additionally configure SSL/TLS. The configuration may involve the WebLogic domain, the ingress controller or the URL of remote endpoints, but it always involves participating in an SSL handshake with the other system. The procedures for setting up SSL use self-signed certificates for demonstration purposes. However, replace the steps as necessary to use signed certificates.

If a UIM cloud native domain is in the role of the client and the server, where secure communications are coming in as well as going out, then both of the following procedures need to be performed:

- Configuring Secure Incoming Access with SSL
- Configuring Access to External SSL-enabled Systems

Configuring Secure Incoming Access with SSL

This section demonstrates how to secure incoming access to UIM cloud native. In the TERMINATE strategy, SSL termination happens at the ingress. The traffic coming in from external clients must use one of the HTTPS endpoints. When SSL terminates at the ingress, it

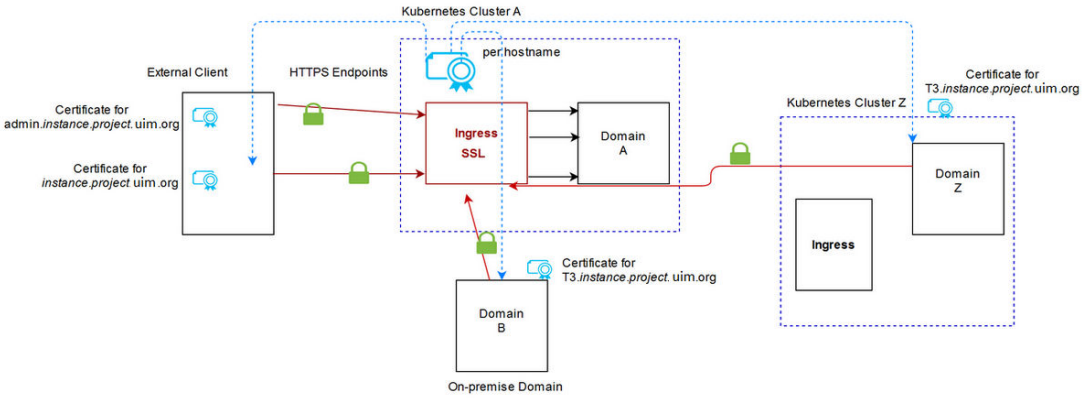
implies that communication within the cluster, such as SAF between the UIM cloud native instances, is not secured.

The Common cloud native toolkit provides the sample configuration for HAProxy ingress. If you use Voyager or other ingress, you can look at the **$COMMON_CNTK/charts/uim-ingress/charts/uim-ingress/templates/generic-ingress.yaml** file to understand the configuration that is applied.

Generating SSL Certificates for Incoming Access

The following illustration shows when certificates are generated.

Figure 9-3 Generating SSL Certificates



When UIM cloud native dictates secure communication, then it is responsible for generating the SSL certificates. These must be provided to the appropriate client. When a UIM cloud native instance in a different Kubernetes cluster acts as the external client (Domain Z in the illustration), it loads the T3 certificate from Domain A as described in "Configuring Access to External SSL-Enabled Systems".

Setting Up UIM Cloud Native for Incoming Access

The ingress controller routes unique hostnames to different backend services. You can see this if you look at the ingress controller YAML file (obtained by running **kubectl get ingress -n** *project ingress_name* **-o yaml**):

```
Kind: Rule
Match: Host(`instance.project.uim.org`)
Services:
  Name: project-instance-cluster-uimcluster
  Port: 8502
  Sticky:
    Cookie:
      Http Only: true
Kind: Rule
Match: Host(`t3.instance.project.uim.org`)
Services:
  Name: project-instance-cluster-uimcluster
  Port: 30303
  Sticky:
    Cookie:
```

```
      Http Only: true
Kind: Rule
Match: Host(`admin.instance.project.uim.org`)
Services:
  Name: project-instance-admin
  Port: 8501
  Sticky:
    Cookie:
      Http Only: true
```

To set up UIM cloud native for incoming access:

1. Use Common certificate and key created while deploying ATA application to create **commoncert.pem** and **commonkey.pem**. See "About Unified Inventory and Topology" in *Unified Inventory and Topology Deployment Guide* for more information:

   ```
   # Create a directory to copy your common keys and certificates. This is
   for sample only. Proper management policies should be used to store
   private keys.
   mkdir $SPEC_PATH/ssl
   copy commoncert.pem and commonkey.pem to $SPEC_PATH/ssl location
   # Create secrets to hold each of the certificates. The secret name must be
   in the format below. Do not change the secret names

   kubectl create secret -n project tls project-instance-uim-tls-cert --
   key $SPEC_PATH/ssl/commonkey.pem --cert $SPEC_PATH/ssl/commoncert.pem
   kubectl create secret -n project tls project-instance-admin-tls-cert --
   key $SPEC_PATH/ssl/commonkey.pem --cert $SPEC_PATH/ssl/commoncert.pem
   kubectl create secret -n project tls project-instance-t3-tls-cert --
   key $SPEC_PATH/ssl/commonkey.pem --cert $SPEC_PATH/ssl/commoncert.pem
   ```

2. Edit the **applications-base.yaml** specification and set `tls.enabled` to `true`

   ```
   tls:
     enabled: true
   ```

3. Create Ingress as follows:

   ```
   $COMMON_CNTK/scripts/create-ingress.sh -i instance -p project -
   s $SPEC_PATH -a uim
   ```

4. After running **create-ingress.sh**, you can validate the configuration by describing the ingress controller for your instance:

   ```
   kubectl get ingress -n project

   NAME                                 AGE
   project-instance-ingress-admin-tls   22h
   project-instance-ingress-t3-tls      22h
   project-instance-ingress-uim-tls     22h
   ```

5. Create your instance as usual.

Configuring Incoming HTTP and JMS Requests for External Clients

This section describes how to configure incoming HTTP and JMS requests for external clients.

> **Note**
>
> Remember to have your DNS resolution set up on any remote hosts that will connect to the UIM cloud native instance.

Incoming HTTPS Requests

External Web clients that are connecting to UIM cloud native must be configured to accept the certificates from UIM cloud native. They will then connect using the HTTPS endpoint and port **30443**.

Incoming JMS Requests

For external servers that are connected to UIM cloud native through SAF, the certificate for the t3 endpoint needs to be copied to the host where the external domain is running.

If your external WebLogic configuration uses "CustomIdentityAndJavaSTandardTrust", then you can follow these instructions exactly to upload the certificate to the Java Standard Trust. If, however, you are using a CustomTrust, then you must upload the certificate into the custom trust keystore.

The keytool is found in the **bin** directory of your jdk installation. The alias should uniquely describe the environment where this certificate is from.

```
./keytool -importcert -v -trustcacerts -alias alias -file /path-to-copied-t3-
certificate/t3.crt -keystore /path-to-jdk/jre/lib/security/cacerts -storepass
default_password

# For example
./keytool -importcert -v -trustcacerts -alias uimcn -file /scratch/t3.crt -
keystore /path-to-jdk/jre/lib/security/cacerts -storepass default_password
```

Update the SAF remote endpoint (on the external UIM instance) to use HTTPS and 30443 port (still t3 hostname).

From the SAF sample provided with the toolkit, the external system would configure the following remote endpoint URL:

```
https://t3.quick.sr.uim.org:30443/ResponseQueue
```

Configuring Access to External SSL-Enabled Systems

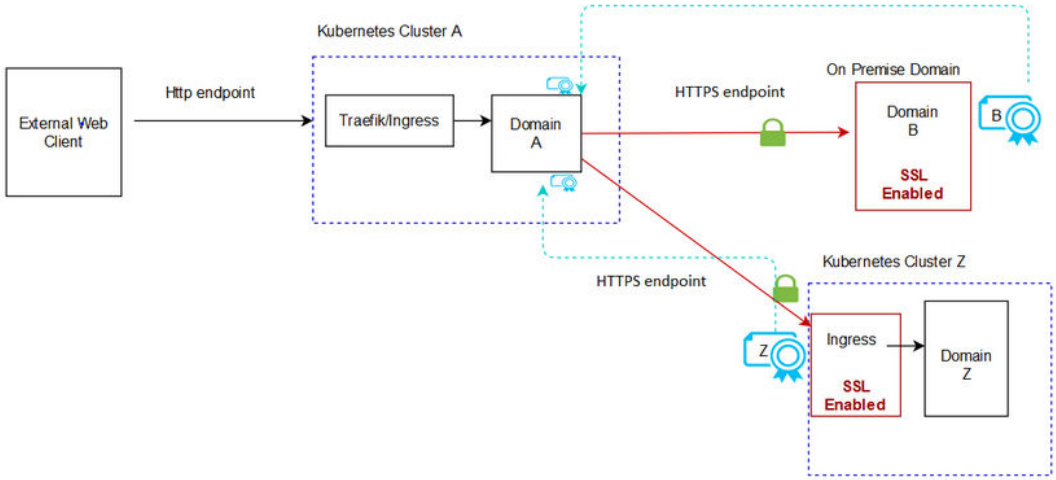
In order for UIM cloud native to participate successfully in a handshake with an external server for SAF integration, the SSL certificates from the external domain must be made available to the UIM cloud native setup. See "Enabling SSL on an External WebLogic Domain" for details about how you could do this for an on-premise WebLogic domain. If you have an external system that is already configured for SSL and working properly, you can skip this procedure and proceed to "Setting Up UIM Cloud Native for Outgoing Access".

Loading Certificates for Outgoing Access

In outgoing SSL, the certificates come from the external domain, whether on-premise or in another Kubernetes cluster. These certificates are then loaded into the UIM cloud native trust.

The following illustration shows information about loading certificates into UIM cloud native setup.

Figure 9-4 SSL Certificates for Outgoing Requests



Enabling SSL on an External WebLogic Domain

These instructions are specific to enabling SSL on a WebLogic domain that is external to the Kubernetes cluster where UIM cloud native is running.

To enable SSL on an external WebLogic domain:

1. Create the certificates. Perform the following steps on the Linux host that has the on-premise WebLogic domain:
   a. Use the Java keytool to generate public and private keys for the server. When the tool asks for your username, use the FQDN for your server.

      ```
      path-to-jdk/bin/keytool -genkeypair -keyalg RSA -keysize 1024 -alias alias -keystore keystore file -keypass private key password -storepass keystore password -validity 360
      ```

   b. Export the public key. This certificate will then be used in the UIM cloud native setup.

      ```
      path-to-jdk/bin/keytool -exportcert -rfc -alias alias -storepass password -keystore keystore -file certificate
      ```

2. Configure WebLogic server for SSL. Follow steps 3 to 17 (skip step 7) in "Set up SSL" in *Oracle Fusion Middleware Administration Console Online Help*.

3. Validate that SSL is configured properly on this server by importing the certificate to a trust store. For this example, the Java trust store is used.

```
path-to-jdk/bin/keytool -importcert -trustcacerts -alias alias -file
certificate -keystore path-to-jdk/jre/lib/security/cacerts -storepass
default_password
```

4. Verify that t3s over the specified port is working by connecting using WLST. Navigate to the directory where the WLST scripts are located.

```
# Set the environment variables. Some shells don't set the variables
correctly so be sure to check that they are set afterward
path-to-FMW/Oracle/Middleware/Oracle_Home/oracle_common/common/bin/
setWlsEnv.sh

# ensure CLASSPATH and PATH are set
echo $CLASSPATH

java -
Dweblogic.security.JavaStandardTrustKeyStorePassPhrase=default_password
weblogic.WLST

# once wlst starts, connect using t3s
wls:offline> connect('<admin user>','<admin password>','t3s://
<server>:<port>')

# If successful you will see the prompt
wls:>domain_name/serverConfig>

#when finished disconnect
disconnect()
```

Setting Up UIM Cloud Native for Outgoing Access

To set up UIM cloud native for outgoing access:

1. Set up custom trust using the following steps:

   a. Load the certificate from your remote server into a trust store and make it available to the UIM cloud native instance.
   Use the Java keytool to create a jks file (truststore) that holds the certificate from your SSL server:

```
keytool -importcert -v -alias alias -file /path-to/certificate.cer -
keystore /path-to/truststore.jks -storepass password
```

> **Note**
>
> Repeat this step to add as many trusted certificates as required.

   b. Create the **commonTrust** secret to store the created **truststore.jks** file.

```
$COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -
s $SPEC_PATH create commonTrust
```

```
#Provide Common TrustStore Details:
Truststore Path: ./truststore.jks #provide truststore path
Truststore Passphrase: ***** #provide truststore passphrase

#verify secret created
secret/project-instance-common-truststore created
```

2. It is mandatory to Set up custom identity using the following steps when trust is passed:
   a. Create the keystore from **commoncert.pem** and **commonkey.pem**.

```
 openssl pkcs12 -export -in <path-to-certs>/commoncert.pem -inkey <path-
to-certs>/commonkey.pem -out keyStore.p12 -name "identity"

 keytool -importkeystore -srckeystore keyStore.p12 -srcstoretype PKCS12
-destkeystore identity.jks -deststoretype JKS
```

   b. Create the secret.

```
kubectl create secret generic secretName -n project --from-
file=secretName.jks=</path-to/identity.jks> --from-
literal=passphrase=passphrase

# verify
kubectl describe secret -n project secretName
```

   c. Edit the **app-uim.yaml** file at location **$SPEC_PATH/project/instance**, uncomment, and provide following properties:

```
uim:
  identity:
    name: secretName .
    alias: identity #alias used to store key in keystore
```

3. Configure SAF by updating the SAF connection configuration in the UIM cloud native **app-uim** specification file to reflect t3s and the SSL port:

```
safConnectionConfig:
  - name: simple
    t3Url: t3s://remote_server:7002
    secretName: simplesecret
```

4. Create the UIM cloud native instance as usual.

Adding Additional Certificates to an Existing Trust

You can add additional certificates to an existing trust while a UIM cloud native instance is up and running.

To add additional certificates to an existing trust:

1. Set up UIM cloud native for outgoing access. See "Configuring Access to External SSL-Enabled Systems" for instructions.

2. Copy the certificates from your remote server and load them into the existing *truststore*.**jks** file you had created:

   ```
   keytool -importcert -v -alias alias -file /path-to/certificate.cer -
   keystore /path-to/truststore.jks -storepass password
   ```

3. Re-create your Kubernetes secret using the same name as you did previously:

   ```
   $COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -
   s $SPEC_PATH create commonTrust
   #Provide updated TrustStore Details:
   Truststore Path: ./truststore.jks #provide updated truststore path
   Truststore Passphrase: ***** #provide truststore passphrase
   ```

4. Restart the instance to force WebLogic Operator to re-evaluate:

   ```
   $COMMON_CNTK/scripts/restart-applications.sh -p project -i instance -
   s $SPEC_PATH -a uim -r all
   ```

Debugging SSL

To debug SSL, do the following:

- Verify Hostname
- Enable SSL logging

Verifying Hostname

When the keystore is generated for the on-premise server, if FQDN is not specified, then you may have to disable hostname verification. This is not secure and should only be done in development environments.

To do so, add the following Java option to the managed server in the **app-uim** specification:

```
managedServers:

  project:
    #JAVA_OPTIONS for all managed servers at project level
    java_options: "-Dweblogic.security.SSL.ignoreHostnameVerification=true"
```

Enabling SSL Logging

When trying to establish the handshake between servers, it is important to enable SSL specific logging.

Add the following Java options to your managed server in the **app-uim** specification. This should be done for your external server as well.

```
managedServers:

  project:
    #JAVA_OPTIONS for all managed servers at project level
    java_options: "-Dweblogic.StdoutDebugEnabled=true -Dssl.debug=true -
Dweblogic.security.SSL.verbose=true -Dweblogic.debug.DebugSecuritySSL=true -
Djavax.net.debug=ssl"
```

Using Wild Card SSL Certificates

UIM cloud native supports wildcard certificates. You can generate wildCard Certificates with the **hostSuffix** value provided in the **applications-base.yaml** spec files. The default is `uim.org`.

To use Wild Card certificates:

1. To create a self-signed wild card certificate, run the following command:

   ```
   openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout $COMMON_CNTK/
   certs/wildcardkey.pem -out $COMMON_CNTK/certs/wildcardcert.pem -subj "/
   CN=*.uim.org" -extensions san -config <(echo '[req]'; echo
   'distinguished_name=req';
   echo '[san]';echo 'subjectAltName=@alt_names'; \echo '[alt_names]'; \
   echo 'DNS.1=*.uim.org'; \
   )
   ```

2. Change the `subDomainNameSeperator` value from period (.) to hyphen (-) so that incoming hostnames match the wild card DNS pattern and update the **$SPEC_PATH/project/instance/applications-base.yaml** file as follows:

   ```
   #Uncomment and provide the value of subDomainNameSeparator, default is "."
   #Value can be changed as "-" to match wild-card pattern of ssl
   certificates.
   #Example hostnames for "-" admin-quick-sr.uim.org, quick-sr.uim.org, t3-
   quick-sr.uim.org
   subDomainNameSeparator: "-"
   ```

3. For the above configured settings, use the following hostnames to access UIM application for `project:sr`, `instance:quick` and `loadBalancerDomainName: uim.org`:

   ```
   uim-admin hostname: admin-quick-sr.uim.org
   uim hostname:  quick-sr.uim.org
   uim-t3 hostname: t3-quick-sr.uim.org
   ```

10

Running the SAF Sample for UIM Cloud Native

It is highly recommended that you explore UIM cloud native support of SAF using a predefined set of configurations and instructions. This activity not only serves to quickly identify any environment issues but also provides the experience in setting up the connectivity for your own projects.

This chapter describes how to run the SAF sample for UIM cloud native.

The SAF sample for UIM cloud native consists of the following components:

- The **SAFSample** cartridge that is ready to be deployed. This cartridge implements a flow that consists of sending a JMS message to a remote system and receiving a JMS message in response.
- Configuration fragments for a project and an instance. These can be added to your **app-uim** specification and contain all the SAF connection specifications as well as endpoint identification.
- A simple emulator that is available as a JAR file, along with instructions and configuration samples. This emulator can be set up on a WebLogic system outside the Kubernetes cluster and functions as a "remote system" in the SAF communication. The emulator simply echos the message given to it.

The SAF sample described in this section uses "sr" as *project* and "quick" as *instance* names.

Prerequisites for running the SAF sample

For the SAF sample, you need the following:

- A Linux host capable of running WebLogic Server 12.2.1.4 outside of the Kubernetes cluster.
- Traffic should be routable between the Kubernetes cluster and this host.
- If you are not using a centralized DNS resolution server, edit the /**etc/hosts** file of the Linux host to add resolution for your UIM cloud native instance. For example, use **<k8s-access-ip> quick.sr.uim.org t3.quick.sr.uim.org admin.quick.sr.uim.org**.

For further details, see "Planning and Validating Your Cloud Environment".

Running the SAF sample involves the following tasks:

- Preparing WebLogic System to Run the Emulator
- Deploying the Emulator on the WebLogic System
- Preparing the UIM Cloud Native Instance
- Deploying the SAF Sample Cartridge
- Validating the SAF Endpoints

Preparing WebLogic System to Run the Emulator

Install WebLogic 12.2.1.4 on the prepared Linux host. The specific patchset does not matter as long as it contains the patch referenced in "Applying the WebLogic Patch for External Systems".

To prepare the WebLogic system to run the emulator:

1. Create a WebLogic domain while accepting all defaults.

   > **Note**
   >
   > Do not enable JRF or any other Fusion Middleware capabilities for this sample. WebLogic domain should be standalone installation with development mode.

2. Stop WebLogic and find the domain home.
3. Edit `<domain-home>/config/config.xml` with the configuration fragment xml file in the COMMON_CNTK, at `samples/uim/saf-sample/remote-weblogic-resources/config/config_fragment.xml`.
4. From the configuration fragment file, copy the following contents to the **config.xml** file:
   a. Add `<jms-system-resource>` for SAF Module at the end, just before `</domain>`.
   b. Add `<saf-agent>` at the end, just before `</domain>`.

   This creates a JMS Module and a SAF agent. The SAF agent will eventually get used in sending emulator responses back to the UIM cloud native instance.
5. From the **$COMMON_CNTK** folder, copy the `samples/uim/saf-sample/remote-weblogic-resources/config/jms/uim-saf-module-jms.xml` file to `<domain-home>/config/jms`.
   This creates SAF entities.
6. Update SAF configuration for connecting to UIM cloud native instance. This instance does not need to be up at this point, but you do need the details such as project name, instance name, weblogic username, and weblogic password.
   a. Edit the **<domain-home>/config/jms/uim-saf-module-jms.xml** file and update the following fields:

   > **Note**
   >
   > The password is entered as plain text, and gets auto-encrypted during WLS startup.

```
<saf-login-context>
    <loginURL>{uim_cn_t3_url}</loginURL>
    <username>{uim_cn_weblogic_username}</username>
    <password-encrypted>{uim_cn_weblogic_password}</password-encrypted>
</saf-login-context>
```

Where `uim_cn_t3_url` is **http://t3.*instance*.*project*.uim.org:30505 or http://t3.*instance*.*project*.uim.org:80**.

b. Start WebLogic. At this point, if you see errors from SAF or JMS about your UIM cloud native instance, you can ignore the errors. These errors go away once the UIM cloud native instance is up and configured for SAF sample.

> **Note**
>
> You can also manually set up SAF agent and SAF entities through WebLogic Server Administration console.

Deploying the Emulator on the WebLogic System

To deploy the emulator on the WebLogic system:

1. Find the **samples/uim/saf-sample/remote-weblogic-resources/emulator-resources/uim-emulator-mdb-0.0.1-SNAPSHOT.jar** emulator **MDB** jar file in **$COMMON_CNTK**.
2. Open the remote WebLogic console.
3. In the **Install** folder of **Deployments**, upload the emulator MDB jar file under **Upload**.
4. Complete the deployment using the defaults and ensure that the MDB file is shown with State "Active" and Health "OK".

Preparing the UIM Cloud Native Instance

To prepare the UIM cloud native instance for the SAF sample:

1. Create customized image with **$COMMON_CNTK/samples/uim/saf-sample/uim-cn-resources/cartridge-resources/SAFSample cartridge**. See "Adding Solution Cartridge Customizations" for more information.
2. Enable the extension mechanism by setting the custom flag to **true** in the **app-uim** specification.
3. Copy **$COMMON_CNTK/samples/uim/customExtensions/_custom-domain-model.tpl** to your **customExtPath** directory.
4. From **$COMMON_CNTK**, copy the contents from **samples/uim/saf-sample/uim-cn-resources/_custom-jms-support.tpl** to **$COMMON_CNTK/samples/uim/customExtensions/_custom-jms-support.tpl**.
5. Copy **$COMMON_CNTK/samples/uim/customExtensions/_custom-jms-support.tpl** to your **customExtPath** directory.
6. Set the `jms` flag in the **app-uim** specification to **true**.
7. From **$COMMON_CNTK**, copy the contents from **samples/uim/saf-sample/uim-cn-resources/_custom-saf-support.tpl** to **$COMMON_CNTK/samples/uim/customExtensions/_custom-saf-support.tpl**.
8. Copy **$COMMON_CNTK/samples/uim/customExtensions/_custom-saf-support.tpl** to your **customExtPath** directory.
9. Set the `saf` flag in the **app-uim** specification to **true**.
   The sample settings are as follows:

```
custom:
  enabled: true
```

```
  application: false
  jdbc: false
  jms: true
  saf: true
  #wdtFiles: {} # This empty declaration should be removed if adding items
here.
  wdtFiles:
   - _custom-domain-model.tpl
  # - _custom-jdbc-support.tpl
   - _custom-jms-support.tpl
   - _custom-saf-support.tpl
```

10. Create a secret for storing the remote server credentials.
11. Replace the user name and password with the values for remote WebLogic credentials as follows:

    ```
    kubectl create secret generic secret_name -n project --from-literal
    username=remote_domain_weblogic_username --from-literal
    password=remote_domain_weblogic_password
    ```

12. Configure the SAF URL and SAF Queue (`RequestQueue`). The cartridge deployed for this sample uses this SAF Queue (`RequestQueue`) to send messages to external Weblogic domain.
13. Replace `safDestinationConfig: {}` in **app-uim.yaml** with the following:

    ```
    safDestinationConfig:
     - name: sample
       t3Url: "t3://{remote_weblogic_hostname}:{remote_weblogic_port}"
       secretName: samplesecret
       destinations:
        - jndiPrefix:
          queues:
           - queue:
              localJndi: RequestQueue
              remoteJndi: RequestQueue
    ```

    Replace the value of `{remote_weblogic_hostname}` and `{remote_weblogic_port}` with the hostname and port where remote WebLogic is installed.

    > **Note**
    >
    > If `safDestinationConfig` already exists in your **app-uim.yaml**, do not create a new element; append `SAFSample` to the end of the existing list of items in `safDestinationConfig`.

14. Create UIM cloud native instance as follows:

    ```
    $COMMON_CNTK/scripts/create-applications.sh -p sr -i quick -s $SPEC_PATH -
    m customExtPath -a uim
    ```

Both the SAF endpoints, one on the remote WebLogic and one on this UIM cloud native instance, become active.

Deploying the SAF Sample Cartridge

Deploy the SAF sample cartridge from **$COMMON_CNTK/samples/uim/saf-sample/uim-cn-resources/cartridge-resources/** to UIM cloud native instance using Design Studio or CMT. The deployment does not have any dependency on base cartridges. See "Deploying Cartridges" for information on deploying cartridges using Design Studio or CMT.

Validating the SAF Endpoints

To validate the SAF endpoints:

1. On the remote WebLogic, login to the WebLogic console and perform the following:
   a. Navigate to **Store and Forward Agents**, **Monitoring**, and then to **Remote Endpoints**. You can see a remote endpoint **uim-saf-module!uim_saf_destination! uim_saf_queue@uim_saf_agent** with the URL pointing to your UIM cloud native instance.
   b. Navigate to **Deployments**. You can see the emulator MDB shown with **State** "Active" and **Health** "OK".
2. On the UIM cloud native instance, login to the WebLogic console.
3. Navigate to **Store and Forward Agents**, **Monitoring**, and then to **Remote Endpoints**. You can see a remote endpoint **uim_sample_saf_module! uim_sample_saf_destinations_0!saf_queue_0@uim_saf_agent@ms1** with the URL pointing to your remote WebLogic.

Performing a Test

You can perform a test as follows:

1. Navigate to **Administration**, **Execute Rule**, **Select SAFSample**, and then to **Process**.
2. Navigate to **Store and Forward Agents**, **uim_saf_agent**, **Monitoring**, and then to **Statistics**.
3. Validate messages received count increased in remote WebLogic console.
4. Validate that the following message appears in **custom_jms_module/ResponseQueue**:

```
Hello!
TimeStamp: 2021.08.25.07.44.01
```

11

Upgrading the UIM Cloud Native Environment

This chapter describes the tasks you perform in order to apply a change or upgrade to a component in the cloud native environment.

Creating a detailed upgrade plan can be a complex process. It is useful to start by mapping your use case to an upgrade path. These upgrade paths identify a set of sequenced activities that align to a CD stage. Once you know the activity sequence, you can then look for the detailed steps involved in each to come up with the comprehensive set of steps to be performed.

Upgrade paths consist of activities that fall into the following two main categories:

- Operational Procedures
- Component Upgrade Procedures

Operational Procedures

There are many different operational procedures and all of these affect the operating state of UIM. UIM cloud native provides the mechanism to change the operational state as described in "Running Operational Procedures".

The flowcharts in this chapter use the following image to depict an operational procedure:

Component Upgrade Procedures

These are the actual set of steps to perform a component upgrade and can be one of the following types:

- **UIM Cloud Native Procedures**: UIM cloud native owns the component and therefore the upgrade procedure for that component. UIM cloud native provides the mechanism to perform the upgrade via the scripts that are bundled with the Common cloud native toolkit. An example of this is a change to a value in a UIM cloud native specification file (shape, project, and instance).

  The flowcharts in this chapter use the following image to depict a UIM cloud native owned procedure.

- **External Procedures**: These procedures are for components that are part of the UIM cloud native operating environment, but are out of the control of UIM cloud native. UIM cloud native does not determine how to apply the upgrade, but provides recommendations on the operational state of UIM accompanying the upgrade.
  An example would be updating the operating system on a worker node.

The flowcharts in this chapter use the following image to depict an external upgrade procedure.

- Miscellaneous upgrade procedures: There are some procedures that require special handling and are not captured in any of the upgrade paths. These are described in "Miscellaneous Upgrade Procedures".

Rolling Restart

Occasionally, you may need to restart UIM managed servers in a rolling fashion, one at a time. This does not result in downtime, but only reduced capacity for a limited period. A rolling restart can be triggered by invoking the **restart-applications.sh** script. This script can restart the whole instance in a rolling fashion, or only the admin server or all the managed servers in a rolling fashion. Some operations may automatically trigger rolling restart. These include image updates, tuning parameter changes, and so on pushed through the **upgrade-applications.sh** script.

Identifying Your Upgrade Path

In order to prepare your detailed plan for an upgrade, you need to be able to map your upgrade use case to an upgrade path. Some common use cases are detailed in the following charts. If your use case is not listed, see "Upgrade Path Flow Chart", which guides you through the decision making process to prepare a specific upgrade path.

Table 11-1 Common Upgrade Paths

Upgrade Type	Component	Upgrade Path	Requires Changing Image?
Cartridge Management	Deploy new cartridge version with Ruleset code (Where the ruleset is referring Java files)	Online change, application upgrade, cartridge deployment	Yes
Cartridge Management	Redeploy a cartridge against an existing cartridge version with Ruleset code (Where the ruleset is referring Java files)	Online change, application upgrade, cartridge deployment	Yes
Cartridge Management	Deploy new cartridge version without Ruleset code	Online change, online cartridge deployment	No
Cartridge Management	Redeploy a cartridge against an existing cartridge version without Ruleset code	Online change, online cartridge deployment	No
Configuration and Tuning	UIM cluster size (scaling up or down)	Online change, application upgrade	No
Configuration and Tuning	Java parameters (memory, GC, and so on)	Online change, application upgrade	No
Configuration and Tuning	WebLogic domain configuration (WDT such as JMS Queue configuration)	Online change, application upgrade	No
Configuration and Tuning	UIM configuration parameters (custom-extensions.properties)	Online change, application upgrade	No

Table 11-1 (Cont.) Common Upgrade Paths

Upgrade Type	Component	Upgrade Path	Requires Changing Image?
Database Storage Management	DB Purges	Offline Change, PDB upgrade	No
Security parameters	WebLogic Password change (poms cache coordination)	Miscellaneous upgrade procedures	No
Security parameters	UIM Schema Password Change	Miscellaneous upgrade procedures	No
Software Upgrade and Patching	UIM release or patch upgrade with Database change	Offline change, PDB upgrade, application upgrade	Yes
Software Upgrade and Patching	Fusion MiddleWare upgrade	Online change, application upgrade (some exceptions needing offline change)	Yes
Software Upgrade and Patching	UIM patch upgrade without Database change	Online Change, application upgrade (some exceptions needing offline change)	Yes
Software Upgrade and Patching	Fusion MiddleWare overlay patches (for example, PSU or one-off patch)	Online Change, application upgrade (some exceptions needing offline change)	Yes
Software Upgrade and Patching	Java upgrade	Online Change, application upgrade	Yes
Software Upgrade and Patching	Linux	Online Change, application upgrade	Yes
Software Upgrade and Patching	Custom code or third-party tool (custom image)	Online Change, application upgrade (some exceptions needing offline change)	Yes
Software Upgrade and Patching	Common cloud native toolkit	The release dictates the constraints.	No
Shared infrastructure	Operating system or hardware on worker node	Online change, external procedure	No
Shared infrastructure	Docker	Online change, external procedure	No
Shared infrastructure	WebLogic Operator minor upgrade (backward compatible)	Online change, external procedure	No
Shared infrastructure	WebLogic Operator major upgrade (non-backward compatible)	Online change, external procedure	No

Once you understand the activities in your upgrade path, you can begin to map out the sequence of activities that you need to perform.

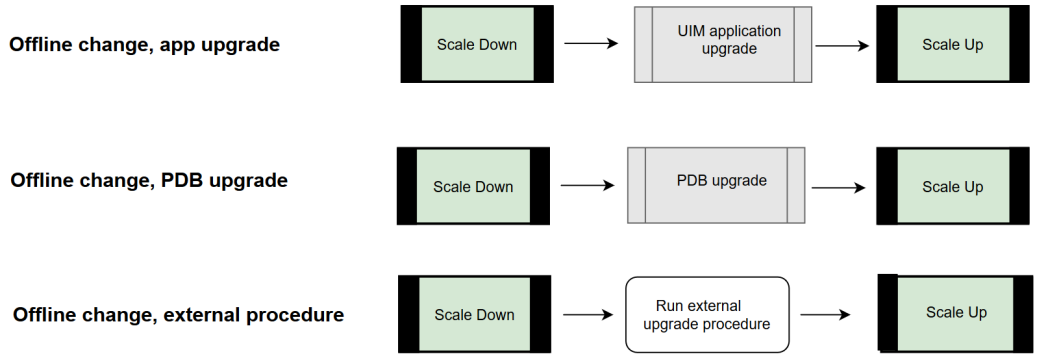
Offline Change Upgrade Paths

Offline changes are defined as those requiring UIM to be shutdown before the change can be applied.

All offline upgrades must start with a Scale Down procedure and end with a Scale Up procedure. You can find the explicit steps to perform these activities in Running Operational Procedures.

Once the cluster has been scaled down, you will need to perform either an external procedure (referencing documentation for the component) or follow a UIM cloud native owned procedure. See "UIM Cloud Native Upgrade Procedures" for details.

Figure 11-1 Offline Change Upgrade Paths



As an example, if your use case is to perform DB purges, then the upgrade path is "Offline Change, DB Purge procedure". The actual steps involve the following:

- Scale Down
  - Edit the shape specification file to set cluster size to **0**.
  - Run **upgrade-applications.sh**.
- PDB Upgrade
  - Edit the **app-uim** specification file to include purge command.
  - Run **install-database.sh** with the command appropriate for the purge use case.
- Scale Up
  - Edit the shape specification file to return cluster size to original (1-18).
  - Run **upgrade-applications.sh**.

Online Change Upgrade

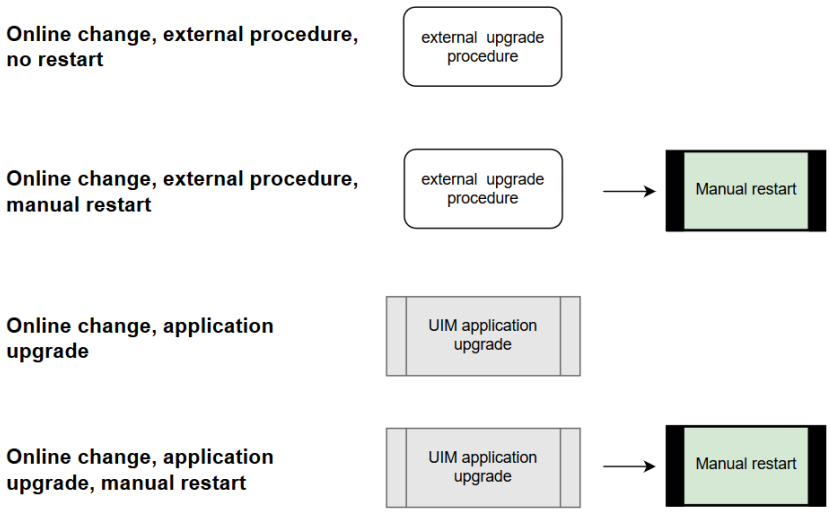
Online changes are changes for which UIM can remain running while the component upgrade is performed. There is, therefore, no operational procedure at the start of the flow, but some paths include a rolling restart after the upgrade procedure is performed.

The component upgrade will either be an external procedure (referencing documentation for the component) or follow a UIM cloud native owned procedure described in "UIM Cloud Native Upgrade Procedures".

If explicit post-upgrade operational activities are required, you can find details in "Running Operational Procedures".

The following flowchart illustrates online change upgrade paths.

Figure 11-2 Online Change Upgrade Paths



Exceptions and Unsupported Tasks

Exceptions

The following require shutdown:

- Some UIM patches
- Some Oracle Fusion Middleware overlay patches
- Oracle Fusion Middleware version upgrades

Unsupported Tasks

Adding, modifying, and deleting users or groups from embedded LDAP are not supported through an upgrade procedure. To make changes to users and groups, the instance must be deleted and re-create.

UIM Cloud Native Upgrade Procedures

The UIM cloud native owned upgrade procedures are:

- Pre-upgrade tasks
- In-Place upgrade instructions
- PDB upgrade
- UIM application upgrade
- Online cartridge deployment

Change or upgrade procedures that are dictated by UIM cloud native are applied using the scripts and the configuration provided in the toolkit.

Pre-Upgrade Tasks

This section includes the pre-upgrade tasks you should perform before upgrading UIM cloud native.

Pre-Upgrade Tasks for Release 8.0.0.0.0 or Later

Upgrading UIM cloud native to a new release includes FMW 14c major tech stack change. Because of this, the usual upgrade procedure does not work, and you should follow the steps mentioned in the subsequent sections. However, you can still build the images as usual for the new release. While upgrading schema and instance, follow the steps mentioned in the subsequent sections.

Upgrading RCU Schema

For a new release, use an RCU schema that is compatible with FMW 14c. To perform this, you may use the following approaches:

- Create a new RCU schema
- Upgrade the existing RCU schema

Creating a New RCU Schema

This is the recommended approach for UIM cloud native. However, it removes any custom roles and policies created through the EM Console, which should be reconfigured later. In this approach, new RCU schemas are created, and during the UIM 8.0.0 upgrade, the application directs to these new schemas.

To create a new RCU schema:

1. Take a backup for the existing OPSS wallet file secret and delete the secret:

   ```
   kubectl get secrets -n $PROJECT $PROJECT-$INSTANCE-opss-walletfile-secret -
   o yaml > $PROJECT-$INSTANCE-opss-walletfile-secret.yaml

   kubectl delete secret -n $PROJECT $PROJECT-$INSTANCE-opss-walletfile-secret
   ```

2. Create an RCU secret with a new RCU prefix name:

   ```
   #Provide new rcu prefix when prompted

   $COMMON_CNTK/scripts/manage-app-credentials.sh -p $PROJECT -i $INSTANCE -
   s $SPEC_PATH -a uim create rcudb
   ```

3. Create new RCU schema:

   ```
   $COMMON_CNTK/scripts/install-database.sh -p $PROJECT -i $INSTANCE -
   s $SPEC_PATH -a uim -c 2
   ```

Upgrading an Existing RCU Schema

If you want to preserve the existing roles and policies data, you must manually copy the current domain and mount it to a container running the FMW 14c image to perform the RCU schema upgrade.

To upgrade an existing RCU schema:

1. Copy domain home outside the **ms1** pod:

```
#domain will be copied outside in directory old_domain

kubectl cp -n $PROJECT $PROJECT-$INSTANCE-ms1:/u01/oracle/user_projects/
domains/domain ./old_domain
```

2. Provide the domain and RCU details in the **response.txt** file by replacing the <*placeholders*> with the appropriate values in the **response.txt** file.
3. Create a container using the UIM cloud native DB installer new release image and mount both the existing domain directory and the **response.txt** file to it:

```
podman run -it -v </path/on/host/to/old_domain>:/tmp/old_domain -v </
path/on/host/to/response.txt>:/tmp/response.txt --name rcu-container <uim-
cn-dbinstaller-image:tag>
```

4. Validate and Run RCU schema upgrades as follows:

```
#Run below commands inside podman container created above.
#switch to upgrade directory
cd $ORACLE_HOME/oracle_common/upgrade/bin
#check rediness for upgrade, verify the output is not having any error.
./ua -readiness -response /tmp/response.txt -logDir /tmp
#If above readiness is successful, run below command to upgrade rcu
./ua -response /tmp/response.txt -logDir /tmp
```

To download the **response.txt** file and for more information about RCU upgrade, see https://oracle.github.io/weblogic-kubernetes-operator/managing-domains/major-weblogic-version-upgrade/upgrade-14210/#upgrade-the-jrf-database.

Upgrading UIM Schema

Upgrade the UIM schema as usual. Before running the upgrade script, ensure that the **$SPEC_PATH/$PROJECT/$INSTANCE/database.yaml** specification file is configured for the new release:

```
$COMMON_CNTK/scripts/install-database.sh -p $PROJECT -i $INSTANCE -
s $SPEC_PATH -a uim -c 3
```

Upgrading UIM Instance

To upgrade UIM CN to a new release:

1. If commonTrust secret is created, providing the Identity keystore in **app-uim.yaml,** is mandatory. For more information, see "Setting Up UIM Cloud Native for Outgoing Access".
2. Delete and recreate the domain.

> **Note**
>
> Due to the major version change from FMW 12c to 14c, direct domain upgrades using the **upgrade-applications.sh** script are not supported.

3. Before running the application creation script, verify that the files **app-uim.yaml** and **applications-base.yaml** located in the **$SPEC_PATH/$PROJECT/$INSTANCE** directory are configured for the new release:

```
# Delete application
$COMMON_CNTK/scripts/delete-applications.sh -p $PROJECT -i $INSTANCE -
s $SPEC_PATH -a uim
#Create application using new COMMON_CNTK
$COMMON_CNTK/scripts/create-applications.sh -p $PROJECT -i $INSTANCE -
s $SPEC_PATH -a uim
```

> **Note**
>
> If there are any customizations, provide them in the above command by using the **-m** option.

In-Place Upgrade

To perform In-Place upgrade:

1. Build the UIM images. For more information, see "Creating the UIM Cloud Native Images".
2. Make sure you have **assembled specification** and the updated **database.yaml**, **app-uim.yaml**, and **applications-base.yaml** files with the corresponding details from the old release.
3. In **app-uim.yaml** and **database.yaml**, update the images build for the new release.
4. Update the configuration files at **$SPEC_PATH/project/instance/config/uim**. For more information, see "Customizing UIM Configuration Properties".
5. Upgrade the UIM schema:

```
$COMMON_CNTK/scripts/install-database.sh -p $PROJECT -i $INSTANCE -
s $SPEC_PATH -a uim -c 3
```

6. Upgrade the UIM cloud native instance and verify the application version:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p $PROJECT -i $INSTANCE -
s $SPEC_PATH -a uim
```

Performing UIM Schema Upgrade

Changes impacting the PDB can be found in any of the specification files - app-uim.yaml, database.yaml, or shape.

Examples include updating the UIM DB Installer image.

To perform a UIM schema upgrade:

1. Make the necessary modifications in your specification files.
2. Invoke **$COMMON_CNTK/scripts/install-database.sh** with the command appropriate for your use case.
   To see a list of options, invoke with **-h**.

UIM Application Upgrade

Changes impacting the UIM application can be found in any of the specification files - project, instance or shape.

Examples include changing an existing value, changing the UIM image or supplying something new such as a secret or a new WDT extension.

To perform UIM application upgrade:

1. Make the necessary modifications in your specification files.
2. Invoke **$COMMON_CNTK/scripts/upgrade-applications.sh** to push out the changes you just made to the running instance. This also triggers introspection for upgrade paths where introspection is required.
3. In upgrade paths where a manual restart is required, restart the instance. See "Restarting the Instance" for details.

Online Cartridge Deployment

The Online deployment mode supports deployment of new cartridges and depends on the type of the cartridge. The cartridges are classified as follows:

- Simple cartridge (such as entity specifications, Groovy, or Drools code)
- Custom Extension cartridge (Java code, configuration files, images, custom applications, Java libraries, Aspects, and localization)

For Simple Cartridges, deployment can be performed without any upgrade path.

For Custom Extension Cartridges, perform the deployment as follows:

1. Build customized image.
2. Make the necessary modifications in your **app-uim** specification to modify the image name.
3. Upgrade the instance.
4. Deploy cartridges.

Upgrades to Infrastructure

From the point of view of UIM instances, upgrades to the cloud infrastructure fall into two categories: rolling upgrades and one-time upgrades.

> **Note**
>
> All infrastructure upgrades must continue to meet the supported types and versions listed in the UIM documentation's certification statement.

Rolling upgrades are where, with proper high-availability planning (like anti-affinity rules), the instance as a whole remains available as parts of it undergo temporary outages. Examples of this are Kubernetes worker node OS upgrades, Kubernetes version upgrades and Docker version upgrades.

One-time upgrades affect a given instance all at once. The instance as a whole suffers either an operational outage or a control outage. Examples of this are WebLogic Operator upgrade and perhaps Ingress Controller upgrade.

Kubernetes and Docker Infrastructure Upgrades

Follow standard Kubernetes and Docker practices to upgrade these components. The impact at any point should be limited to one node - Master (Kubernetes and OS) or worker (Kubernetes, OS, and Docker). If a worker node is going to be upgraded, drain and cordon the node first. This will result in all pods moving away to other worker nodes. This is assuming your cluster has the capacity for this - you may have to temporarily add a worker node or two. For UIM instances, any pods on the cordoned worker will suffer an outage until they come up on other workers. However, their messages and orders are redistributed to remaining managed server pods and processing continues at a reduced capacity until the affected pods relocate and initialize. As each worker undergoes this process in turn, pods continue to terminate and start up elsewhere, but as long as the instance has pods in both affected and unaffected nodes, it will continue to process orders.

WebLogic Operator Upgrade

To upgrade the WebLogic Operator, follow the Operator documentation. As long as the current version can be upgraded to target version through helm upgrade command, a phased cutover can be performed. In this, you will perform a helm upgrade to the new version of the Operator into the same namespace. RBAC will not be updated in the namespace. You do not have to register the namespace again as upgraded WebLogic operator should be monitoring using the same `LableSelector`.

```
#export WebLogic operator namespace
export WLSKO_NS=wlsko

#update WebLogic helm repo
helm repo add weblogic-operator https://oracle.github.io/weblogic-kubernetes-
operator/charts --force-update
#upgrade WebLogic operator, check compatibility matrix for supported versions
helm upgrade weblogic-operator \
           weblogic-operator/weblogic-operator \
          --version <version-to-be-upgraded>\
          --namespace $WLSKO_NS
```

All instances with the transitioned project are impacted by this operation. However, there is no order processing outage during the transition. There is a control outage - where no changes can be pushed to the instances (**upgrade-applications.sh**, **delete-applications.sh** or **restart-applications.sh**). Also, during the control outage, the termination of a pod does not immediately trigger healing. However, once the transition of the project is complete, the new Operator will react to any changed state (whether in the cluster, like pod termination, or in pushed changes, like instance upgrades) and run the required actions.

Ingress Controller Upgrade

Follow the documentation of your chosen Ingress Controller to perform an upgrade. Depending on the Ingress Controller used and its deployment in your Kubernetes environment, the UIM instances it serves may see a wide set of impacts, ranging from no impact at all (if the Ingress Controller supports a clustered approach and can be upgraded that way) to a complete outage.

To take the sample of HAProxy Ingress that the Common cloud native toolkit uses as an Ingress Controller illustration. See UIM Cloud Native Deployment Software Compatibility for the corresponding Ingress version.

```
$ helm repo update haproxytech

$ helm upgrade haproxy-kubernetes-ingress haproxytech/kubernetes-ingress --
namespace haproxy --version <version>
```

During this transition, there will be an outage in terms of the outside world interacting with UIM. Any data that flows through the ingress will be blocked until the new HAProxy takes over. This includes GUI traffic, order injection, API queries, and SAF responses from external systems. This outage will affect all the instances in the namespace being transitioned.

Miscellaneous Upgrade Procedures

This section describes miscellaneous upgrade scenarios.

Network File System (NFS)

If an instance is created successfully, but a change to the NFS configuration is required, then the change cannot be made to a running UIM instance. In this case, the procedure is as follows:

1. Perform a fast delete. See "Running Operational Procedures" for details.
2. Update the `nfs` details in the instance specification.
3. Start the instance.

Security Parameters

To set the security parameters:

1. Perform a fast delete. See "Running Operational Procedures" for details.
2. Update the secrets for WebLogic, PDB credentials, or UIM Schema credentials.
3. Start the instance.

Running Operational Procedures

This section describes the tasks you perform on the UIM server in response to a planned upgrade to the UIM cloud native environment. You must consider if the change in the environment fundamentally affects UIM processing to the extent that UIM should not run when the upgrade is applied or UIM can run during the upgrade but must be restarted to properly process the change.

The operational procedures are performed using the UIM cloud native specification files and scripts.

The operational procedures you perform for upgrading your cloud environment are:

- Trigger introspection
- Scaling down the cluster
- Scaling up the cluster
- Restarting the cluster

- Fast delete
- Shutting down the cluster
- Starting up the cluster

Triggering Introspection

When any of the specification files have changed, invoke the **upgrade-applications.sh** script to trigger the operator's introspector to examine the change and apply it to the running instance.

Scaling Down the Cluster

The scaling down procedure described here is only in the context of the upgrade flow diagram. Hence, scaling down is down to 0 managed servers. A generalized scaling can change the cluster size down to a value between 0 and 18 (both inclusive) in any desired increment or decrement.

To scale down the cluster, edit the shape specification and change the `clusterSize` parameter to `0`. This terminates all the managed server pods, but leaves the admin server up and running.

Apply the change to the running Helm release by running the upgrade script:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Scaling Up the Cluster

The scaling up procedure described here is only in the context of the upgrade flow diagram. Hence, scaling up is up to the initial cluster size. A generalized scaling can change the cluster size up to a value between 0 and 18 (both inclusive) in any desired increment or decrement.

To scale up the cluster, edit the shape specification and change the value of the `clusterSize` parameter to its original value to return the cluster to its previous operational state.

Apply the change to the running Helm release by running the upgrade script:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Restarting the Instance

The Common cloud native toolkit provides a script (**restart-applications.sh**) that you can use to perform different flavors of restarts on a running instance of UIM cloud native.

Following is the usage of the **restart-applications.sh** script

```
restart-applications.sh parameters
      -p <projectName>   : mandatory
      -i <instanceName>  : mandatory
      -s <applications-specPath>      : mandatory; locations of specification
files
```

```
      -a <applicationName> : optional;  if not provided values in
applications-base.yaml will be considered.
      -m <customExtPath> : optional; only for uim; locations of custom
extension files
      -r <restart>       : optional; Mandatory if ata applicationName is
provided.
   restart allowed values for uim:
      all: Restart the whole instance (rolling restart)
      admin: Restart the WebLogic Admin Server only
      ms: Restart all the Managed Servers (rolling restart)
```

For example, to restart a complete cluster, run the following command:

```
$COMMON_CNTK/scripts/restart-applications.sh -p project -i instance -
s $SPEC_PATH -r all
```

Fast Delete

When the entire domain, including the admin server, needs to be taken offline, then the full shutdown and full startup procedures follow. This can be used to perform a "fast delete" or "dehydration" of the domain, instead of a full **delete-applications** operation where you may have to be concerned about the secrets and other pre-requisites being deleted. To quickly restore the domain, simply perform the startup procedure.

Shutting Down the Cluster

To shut down the cluster, edit the **app-uim** specification and add or modify the value of the `serverStartPolicy` parameter to **Never**. This terminates all the pods.

```
# Operational control parameters
# scope - domain or cluster
serverStartPolicy: Never
```

Apply the change to the running Helm release by running the upgrade script:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Starting Up the Cluster

To start up the cluster, edit the **app-uim** specification and comment out or modify the value of the `serverStartPolicy` parameter to **IfNeeded**. This starts up all the pods.

```
# Operational control parameters
# scope - domain or cluster
serverStartPolicy: IfNeeded
```

Apply the change to the running Helm release by running the upgrade script:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

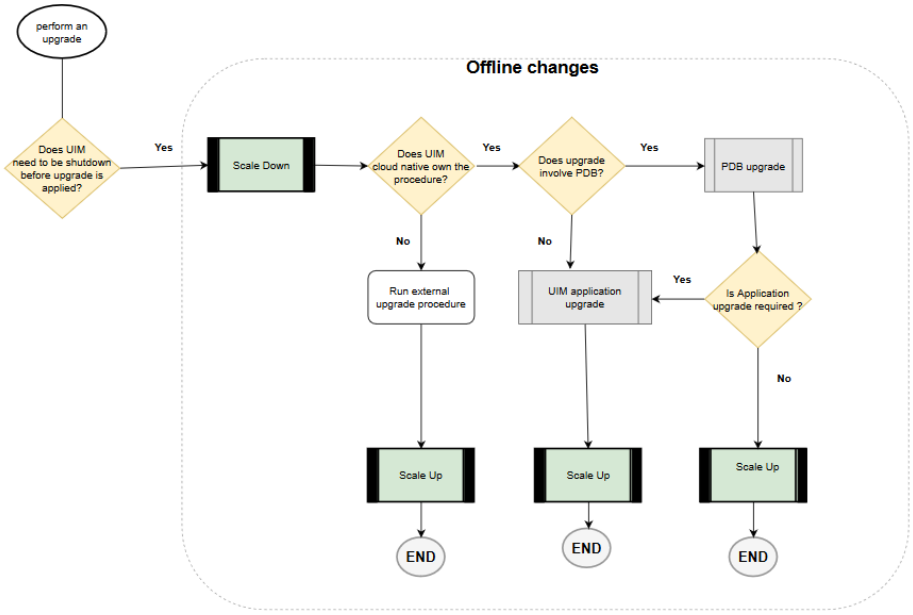
Upgrade Path Flow Chart

When comparing and contrasting the different flows, identifying common steps or divergences, it can be useful to have a combined view of the flowcharts along with the main decision points. This can be useful when trying to automate parts of the process.

The first decision to make is whether UIM can be running when you apply the change. Typically, UIM needs to be shutdown for PDB impacting scenarios and the exceptions listed in the "Exceptions and Unsupported Tasks" section.

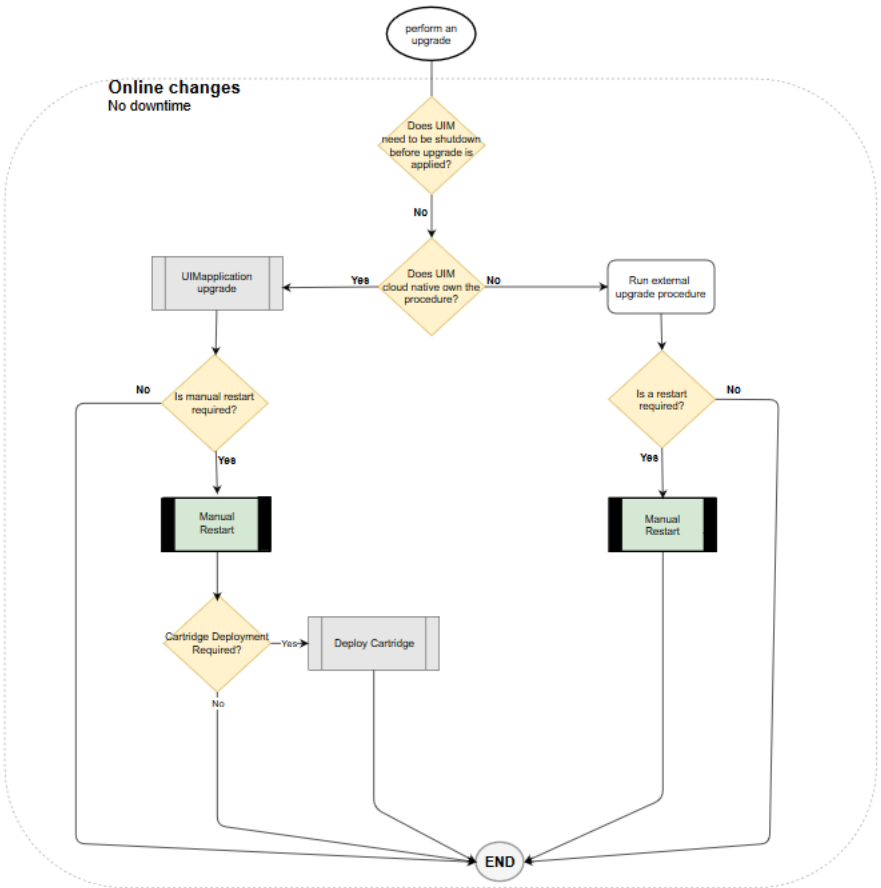
The following flowchart illustrates the flow for offline upgrades and various scenarios.

Figure 11-3 Upgrade Path Flow for Offline Changes



The following flowchart illustrates the flow for online upgrades and various scenarios.

Figure 11-4 Upgrade Path Flow for Online Changes



12

Moving to UIM Cloud Native from a Traditional Deployment

You can move to a UIM cloud native deployment from your existing UIM traditional deployment. This chapter describes tasks that are necessary for moving from a traditional UIM deployment to a UIM cloud native deployment.

Supported Releases

You can move to UIM cloud native from all supported traditional UIM releases.

About the Move Process

The move to UIM cloud native involves offline preparation as well as maintenance outage. This section outlines the general process as well as the details of the steps involved in the move to UIM cloud native. However, there are various places where choices have to be made. It is recommended that a specific procedure be put together after taking into account these choices in your deployment context.

The UIM cloud native application layer runs on different hardware locations (within a Kubernetes cluster) than the UIM traditional application layer.

The process of moving to UIM cloud native involves the following sets of activities:

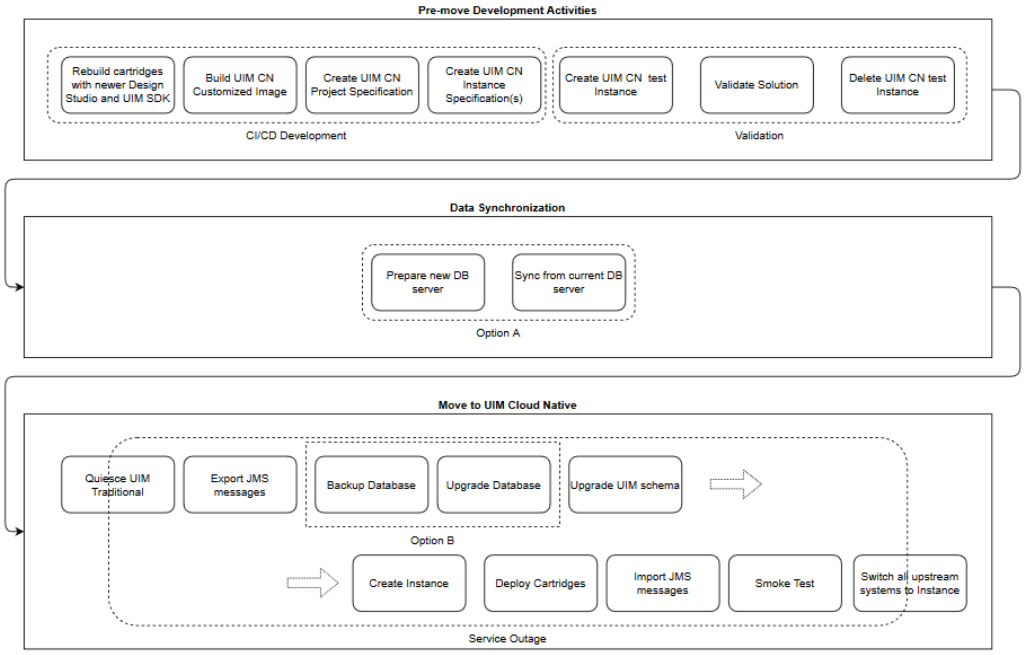
- Pre-move development activities, which include the following tasks:
  - Rebuilding cartridges using Design Studio and UIM SDK (solution task)
  - Building UIM cloud native images (cloud native task)
  - Assembling specifications and configuring them for UIM cloud native (cloud native and solution task)
  - Creating a UIM cloud native instance for testing (cloud native task)
  - Validating your solution cartridges (solution task)
  - Deleting the test UIM cloud native instance (cloud native task)
  - Finalizing your specifications (cloud native and solution task)
- Data synchronization activities, which include the following tasks:
  - Preparing a new database server (database task)
  - Synchronizing the current database server (database task)
- Tasks for moving to UIM cloud native, which include the following:
  - Quiescing the UIM traditional instance (solution task)
  - Backing up the database (database task)
  - Upgrading the database (database task)
  - Upgrading the UIM schema (database task)

- Creating a UIM cloud native instance (cloud native task)
- Deploying cartridges (solution task)
- Importing JMS messages (WebLogic Server administration task)
- Performing a smoke test (solution task)
- Switching all upstream systems (solution task)

The following diagram illustrates these activities.

> **Note**
>
> In the diagram, the short form of "UIM CN" stands for "UIM cloud native".

Figure 12-1 Move to UIM Cloud Native Process



Pre-move Development Activities

In preparation to move your traditional UIM instance into a UIM cloud native environment, you must do the following activities:

1. If your UIM cartridges were built against a UIM deployment that is older than 7.5.0, use Design Studio to rebuild them with the UIM SDK of the target release. Select the Design Studio version based on its compatibility matrix.
2. Build UIM cloud native images. This task includes creating the UIM Docker image and the DB Installer Docker image by using the UIM cloud native download packages. See "Creating the UIM Cloud Native Images" for details.

3. Analyze your solution and configure a **applications-base** and **app-uim** specifications for your UIM cloud native instance. This specification includes details of JMS queues and topics, as well as SAF connections and SAF endpoint details.
4. (Optional) Create a test instance, pointing to a test PDB. You can later change this specification to point to the migrated database. When creating the specification, choose your cloud native production shape - prodsmall, prod, prodlarge. Alternatively, create a custom production shape by copying and modifying one of the shapes. See "Creating Custom Shapes" for details about custom shapes. See "Configuring the Specification Files" for details. If your solution requires model extensions or custom files, create the additional YAML files for those as well.
5. Create a UIM cloud native test instance and test your specifications. To do this, bring up the UIM cloud native instance as described in "Creating a Basic UIM Instance" (create instance secrets, install the RCU schema, install the UIM schema, bring up UIM, and create ingress, deploy your cartridges).
6. Validate the solution.
7. Shut down your test instance and remove the associated secrets, PDB, and ingress.
8. Finalize your specifications for the move by picking up any changes from your test activity and re-create instance secrets to use the migrated database. Change the project and instance specification to:
    a. In **app-uim** specification, point to the migrated database location once it is known.
    b. In **app-uim** specification, switch SAF endpoints to the actual components, instead of emulators.
    c. In shape and **app-uim.yaml** specifications, arrange for the same number of managed servers in your UIM cloud native instance.

       UIM cloud native requires the use of standard sizing for managed servers. This is represented as a set of "shapes". As a result, it is possible that your UIM cloud native instance needs a different number of managed servers to handle your workload as compared to your UIM traditional instance. For the purpose of this migration activity, it is recommended to start with the same number of managed servers, perform the import and smoke tests, and then scale (scale-up or scale-down) the UIM cloud native instance to the desired size.

       If it is not possible to arrange for the same number of managed servers in your UIM cloud native instance as there are in your UIM traditional instance, it is recommended that you get as close as you can. You can then import the JMS messages from the leftover managed servers into one of the UIM cloud native managed servers. For example, consider a UIM traditional instance with four managed servers (**ms1**, **ms2**, **ms3**, and **ms4**). The analysis may show that you only need two managed servers (**cn-ms1** and **cn-ms2**) of prod shape in your UIM cloud native instance. You can import all JMS messages from **ms1** into **cn-ms1**, and from **ms2** into **cn-ms2**. And then, import the remaining messages from **ms3** to **cn-ms1** and from **ms4** to **cn-ms2**. Try to spread the load as evenly as possible.

Moving to a UIM Cloud Native Deployment

Moving to a UIM cloud native deployment from a UIM traditional deployment requires performing the following tasks:

1. Quiesce the UIM traditional instance. See "Quiescing the Traditional Instance of UIM" for more information.

2. Export JMS messages. See "Exporting and Importing JMS Messages" for more information.
3. Take a back up and upgrade the database. See "Upgrading the Database" for more information.
4. Upgrade the UIM schema. See "Upgrading the UIM Schema" for more information.
5. Use the existing RCU schema See "Reusing RCU" section of "Recreating an Instance" for more information.
6. Create the UIM cloud native instance. See "Creating Your Own UIM Cloud Native Instance" for more information.
7. Deploy cartridges. See "Deploying Cartridges" for more information.
8. Import JMS messages. See "Importing JMS Messages" for more information.
9. Perform a smoke test. See "Performing a Smoke Test" for more information. Once the UIM cloud native instance passes smoke test and is optionally resized to the desired target value, shut down the UIM traditional instance fully.
10. Switch all upstream systems to the UIM cloud native instance. See "Switching Integration with Upstream Systems" for more information.

Quiescing the Traditional Instance of UIM

At the start of the maintenance window, the UIM traditional instance must be quiesced. This involves stopping database jobs, stopping all upstream and peer systems from sending messages (for example, http/s, JMS, and SAF) to UIM, and ensuring all human users are logged out. It also involves pausing the JMS queues so that no messages get queued or dequeued. The result is that UIM is up and running, but completely idle.

Exporting and Importing JMS Messages

Irrespective of the persistence mechanism you use (file-based or JDBC) in your UIM traditional instance, you must still export and import outstanding messages as described in this section. If file-based persistence is used, this procedure accomplishes a switch to JDBC-based persistence. On the other hand, if JDBC-based persistence is already in use, this procedure brings the setup (in WebLogic and in the database) in line with UIM cloud native requirements.

Overall, this procedure consists of exporting the JMS messages to disk, switching over to the UIM cloud native instance, and importing the JMS messages from disk. Due to the configuration in the UIM cloud native instance, the imported messages will get populated into the appropriate database tables of the UIM cloud native instance rather than their original location. The time taken for the export and import depends on the number of messages that are in the persistent store to begin with.

See the following topics for further details:

- Exporting JMS Messages
- Importing JMS Messages

Exporting JMS Messages

Before exporting JMS messages, validate that your UIM traditional instance has the WebLogic patch 31169032 (or its equivalent for your WebLogic version) installed. This patch is required to properly export UIM JMS messages. If it is not installed, follow the standard WebLogic patch procedures to procure and install the patch.

To export JMS messages:

1. Login to the WebLogic Console and navigate to the list of UIM queues.
2. For each queue, open its Monitoring tab to get the list of current destinations for the queue. The Monitoring tab shows as many destinations as the number of managed servers.
3. Select each destination and click **Show Messages**. If there are any messages pending in this destination of this queue, click the **Export** button to export all the messages to a file. Use the queue name and destination in the filename for ease of tracing.
4. If you have defined other JMS Modules as part of your solution, repeat steps 2 and 3 for each of those modules.

Importing JMS Messages

Before importing JMS messages, ensure that your UIM cloud native instance is up and running, but quiesced (queues paused and database jobs stopped). It is recommended that your UIM cloud native instance has the same number of managed servers as your UIM traditional instance.

To import JMS messages:

1. Transfer all the exported files into the Admin Server pod using the **kubectl cp** command.
2. Log in to the WebLogic Console and navigate to JMS Modules where the UIM queues are listed.
3. For each queue for which you have an export file, open its Monitoring tab.
4. For each destination on this queue for which you have an export file, find the same destination in the list
5. Select the destination and click **Show Messages**. Click **Import** to specify the filename and import the messages.

Upgrading the Database

To upgrade the database, you perform the following tasks:

- Upgrading the UIM Schema
- Switching Integration with Upstream Systems

Upgrading the Database Server

You may need to upgrade the database server itself to the version supported by the UIM cloud native release you are moving to. To identify the required version of the database server and to determine if you need a database server upgrade, see "UIM Software Compatibility" in *UIM Compatibility Matrix*.

If you do need a database server upgrade, choose one of the following options:

- **Option A: Create an additional database server**: Create a second database server of the target database version (with required patches), seeded with an RMAN backup of the UIM traditional database. Enable Oracle Active DataGuard to continuously synchronize data from the UIM traditional database to this new database. Use this new database for the UIM cloud native instance. For further details, see *Mixed Oracle Version support with Data Guard Redo Transport Services* (Doc ID 785347.1) knowledge article on My Oracle Support. The exact mechanisms to be used are subject to circumstances such as resource availability, size of data, timing, and so on but the goal is to have a second database server

running the target database version but always containing the data from the UIM traditional database.
This option has the following advantages:

  - Allows switching from a standalone database to a Container DB and Pluggable DB model that is required for UIM cloud native, without impacting other users of the existing database.
  - Reduces the duration of a service outage since you can avoid having to backup the database and upgrade it as part of the maintenance window.
  - Preserves the UIM traditional database unchanged reducing the risk and cost associated with reverting to UIM traditional instance, if that becomes necessary.

- **Option B: Retain the existing DB server**: You can retain the existing database server, upgrading it in-place to the target database version and patches.

If you choose option A, the upgrade process must pause after the export of JMS messages, and ensure the Active DataGuard sync is complete (if there are pending redo logs). Then, before proceeding, the sync must be turned off and the new database must be brought online fully.

Preparing the Required Database Entities for UIM Cloud Native

To meet the UIM cloud native pre-requisites, you will have to re-use existing RCU Schema. See "*Reusing the RCU*" section of "Recreating an Instance" for more information.

To ensure a clean start for UIM cloud native managed servers, delete the leftover LLR tables. When UIM cloud native managed servers start, these tables are recreated with the required data automatically.

To delete the LLR tables:

1. Connect to the database using the UIM cloud native user credentials.
2. Get the list of tables to delete:

```
select 'drop table '||tname||' cascade constraints PURGE;' from tab where
tname like ('WL_LLR_%');
```

3. For the tables listed, run the commands for dropping a table.

Upgrading the UIM Schema

To upgrade the UIM schema and cartridges, do the following:

- **Migrate the UIM schema**: To migrate the schema of your UIM traditional instance into a schema that is compatible with UIM cloud native, run the UIM cloud native DB Installer with command 3.
- **Upgrade the UIM Schema to the target version**: If you are running a version of UIM traditional instance that is older than the target UIM cloud native version, use the UIM cloud native DB Installer with command 3 to upgrade the UIM schema to the correct version.

Switching Integration with Upstream Systems

After you shut down the UIM traditional instance fully, do the following:

- Ensure that the UIM cloud native instance has its JMS and SAF objects unpaused and its DB jobs restarted.
- Configure the upstream and peer systems to resume sending messages. See "Integrating UIM" for more details.

Reverting to Your UIM Traditional Deployment

During the move to UIM cloud native, if there is a need to revert to your UIM traditional deployment, the exact sequence of steps that you need to perform depend on the options you have chosen while moving to UIM cloud native.

In general, the UIM traditional deployment application layer should be undisturbed through the upgrade process. If Option A was followed for upgrading the database, the UIM traditional instance can simply be started up again, still pointing to its database.

If however, Option B was followed for upgrading the database, the following steps are required before the UIM traditional instance can be spun up:

- Revert the database server version to the earlier version (if a database server upgrade was performed as part of the switch to UIM cloud native)
- Restore the database contents from the backup taken as part of Option B for upgrading the database.

Cleaning Up

Once the UIM cloud native instance is deemed operational, you can release the resources used for the UIM traditional application layer.

If Option A was adopted for the database, then you can delete the database used for UIM traditional instance and release its resources as well. If Option B was followed and your UIM traditional instance was using JDBC persistent stores, the tables corresponding to these are now defunct and you can delete these as well.

13

Debugging and Troubleshooting

This chapter provides information about debugging and troubleshooting issues that you may face while setting up UIM cloud native environment and creating UIM cloud native instances.

This chapter describes information about the following:

- Setting Up Java Flight Recorder (JFR)
- Troubleshooting Issues with Ingress Controller, UIM UI, and WebLogic Administration Console
- Common Error Scenarios
- Known Issues

Setting Up Java Flight Recorder (JFR)

The Java Flight Recorder (JFR) is a tool that collects diagnostic data about running Java applications. UIM cloud native leverages JFR. See Java Platform, Standard Edition Java Flight Recorder Runtime Guide for details about JFR.

You can change the JFR settings provided with the toolkit by updating the appropriate values in the **$SPEC_PATH/project/instance/app-uim.yaml** specification.

To analyze the output produced by the JFR, use Java Mission Control. See Java Platform, Standard Edition Java Mission Control User's Guide for details about Java Mission Control.

JFR is turned off by default in all managed servers. You can enable this feature by setting the `enabled` flag to **true**.

You can customize how much data is maintained, by changing the `max_age` parameter in the instance specification:

```
# Java Flight Recorder (JFR) Settings
jfr:
 enabled: true
 max_age: 4h
```

Data that is generated by the JFR is saved in the container in **/logMount**/*$(DOMAIN_UID)/server/$(SERVER_NAME)*/**performance**.

Persisting JFR Data

JFR data can be persisted outside of the container by re-directing it to persistent storage through the use of a PV-PVC. See "Setting Up Persistent Storage" for details.

Once the storage has been set up, enable the `storageVolume`. The default storage volume is `false`. Once enabled, JFR data is re-directed automatically.

```
storageVolume:
 enable: false
 pvc: sr-nfs-pvc #name of pvc
```

Deleting the pod that has storage volume as `disabled` deletes the corresponding logs. To retain the logs:

- Set the **storageVolume.enable** to **true**.
- Provide **storageVolume.pvc**.
- Specify the name of the pvc created.

```
storageVolume:
 enable: true
 pvc: storage-pvc
```

Troubleshooting Issues with Ingress Controller, UIM UI, and WebLogic Administration Console

This section describes how to troubleshoot issues with access to the UIM UI clients, WLST, and WebLogic Administration Console.

It is assumed that HAProxy is the Ingress controller being used and the domain name suffix is `uim.org`. You can modify the instructions to suit any other domain name suffix that you may have chosen.

The following table lists the URLs for accessing the UIM UI clients and the WebLogic Administration Console, when the Oracle Cloud Infrastructure load balancer is used and not used:

Table 13-1 URLs for Accessing UIM Clients

Client	If Not Using Oracle Cloud Infrastructure Load Balancer	If Using Oracle Cloud Infrastructure Load Balancer
UIM Web Client	http://*instance*.*project*.uim.org:30505/Inventory	http://*instance*.*project*.uim.org:80/Inventory
WLST	http://t3.*instance*.*project*.uim.org:30505	http://t3.*instance*.*project*.uim.org:80
WebLogic Admin Console	http://admin.*instance*.*project*.uim.org:30505/console	http://admin.*instance*.*project*.uim.org:80/console

Error: Http 503 Service Unavailable (for UIM UI Clients)

This error occurs if the managed servers are not running.

To resolve this issue:

1. Check the status of the managed servers and ensure that at least one managed server is up and running:

   ```
   kubectl -n project get pods
   ```

2. Log into WebLogic Admin Console and navigate to the Deployments section and check if the State column for **oracle.communications.inventory** shows **Active**. The value in the Targets column indicates the name of the cluster.

If the application is not Active, check the managed server logs and see if there are any errors. For example, it is likely that the UIM DB Connection pool could not be created. The following could be the reasons for this:

- DB connectivity could not be established due to reasons such as password expired, account locked, and so on.
- DB Schema heath check policy failed.

There could be other reasons for the application not becoming Active.

Resolution: To resolve this issue, address the errors that prevent the application from becoming Active. Depending on the nature of the corrective action you take, you may have to perform the following procedures as required:

- Restart the instance, by running **restart-applications.sh**
- Delete and create a new instance, by running **delete-applications.sh** followed by **create-applications.sh**

Security Warning in Mozilla Firefox

If you use Mozilla Firefox to connect to a UIM cloud native instance via HTTP, your connection may fail with a security warning. You may notice that the URL you entered automatically change to `https://`. This can happen even if HTTPS is disabled for the UIM instance. If HTTPS is enabled, it only happens if you are using a self-signed (or otherwise untrusted) certificate.

If you wish to continue with the connection to the UIM instance using HTTP, in the configuration settings for your Firefox browser (URL: "about:config"), set the **network.stricttransportsecurity.preloadlist** parameter to FALSE.

Error: Http 404 Page not found

This is the most common problem that you may encounter.

To resolve this issue:

1. Check the Domain Name System (DNS) configuration.

> **Note**
>
> These steps apply for local DNS resolution via the **hosts** file. For any other DNS resolution, such as corporate DNS, follow the corresponding steps.

The hosts configuration file is located at:

- On Windows: **C:\Windows\System32\drivers\etc\hosts**
- On Linux: **/etc/hosts**

Check if the following entry exists in the hosts configuration file of the client machine from where you are trying to connect to UIM:

- Local installation of Kubernetes without Oracle Cloud Infrastructure load balancer:

```
Kubernetes_Cluster_Master_IP  instance.project.uim.org
t3.instance.project.uim.org  admin.instance.project.uim.org
```

- If Oracle Cloud Infrastructure load balancer is used:

  ```
  Load_balancer_IP  instance.project.uim.org  t3.instance.project.uim.org
  admin.instance.project.uim.org
  ```

2. Resolve the DNS configuration.
3. Check the browser settings and ensure that ***.uim.org** is added to the No proxy list, if your proxy cannot route to it.
4. Check if the Ingress Controller pod is running. If not, install or update the Ingress Controller:

   ```
   kubectl get pod -n haproxy


   NAME READY STATUS RESTARTS AGE
   haproxy-operator-ingress-haproxy-** 1/1 Running 0 128m
   ```

5. Check if Ingress Controller service is running:

   ```
   kubectl -n haproxy get svc
   NAME        TYPE        CLUSTER-IP    EXTERNAL- IP    PORT(S)        AGE
   oci-lb-service-haproxy    LoadBalancer    10.96.136.31
   100.77.18.141    80:31115/TCP    20d    <---- Is expected in
   OCI environment only -->
   haproxy-operator    NodePort    10.98.176.16
   <none>    443:30543/TCP,80:30505/TCP    141m
   ```

6. Check if Ingress Object is created, by running the following command:

   ```
   $ kubectl get ingress -n $PROJECT

   NAME                        CLASS     HOSTS
   project-instance-ingress    haproxy   instance.project.uim.org
   ```

7. Verify if the CLASS matches the configured `className` for **Ingress Controller**. If not, update **applications-base.yaml** with the appropriate class name. The default is **haproxy**.If Ingress is not created, create Ingress by running the following command:

   ```
   $COMMON_CNTK/scripts/create-ingress.sh -p project -i instance -
   s $SPEC_PATH -a uim
   ```

Debugging Ingress Controller Access Logs

To increase the log level and debug access logs of Ingress Controller, refer the corresponding ingress controller documentation. For HAProxy ingress controller, see https://www.haproxy.com/documentation/kubernetes-ingress/administration/logging/#view-access-logs.

Cleaning Up the Installation of Ingress Controller

> **Note**
>
> Clean up is not usually required. It should be performed as a desperate measure only.
>
> **Warning**: The access to UIM instances is configured with Ingress Controller will be interrupted if it is uninstalled using the process mentioned here.

If Ingress Controller is installed with helm charts, it can be cleaned up by uninstalling that helm chart.

To clean up the HAProxy Helm chart, run the following command:

```
helm uninstall haproxy-kubernetes-ingress -n haproxy
```

Cleaning up of Ingress Controller does not impact actively running UIM instances. However, they cannot be accessed during that time. Once the Ingress Controller chart is re-installed with the same `ingressClass`, UIM instances can be accessed again.

Setting up Logs

As described earlier in this guide, UIM and WebLogic logs can be stored in the individual pods or in a location provided via a Kubernetes Persistent Volume. The PV approach is strongly recommended, both to allow for proper preservation of logs (as pods are ephemeral) and to avoid straining the in-pod storage in Kubernetes.

Within the pod, logs are available at: **/u01/oracle/user_projects/domains/domain/servers/ms1/logs**.

> **Note**
>
> Replace **ms1** with the appropriate managed server or with "admin".

When a PV is configured, **stdout** logs are available at the following path starting from the root of the PV storage:

*project-instance***/server/<servername>/logs**

When a PV is configured, **main** logs are available at the following path starting from the root of the PV storage:

*project-instance***/server/introspector/logs**

The following logs are available in the location (within the pod or in PV) based on the specification:

- admin.log: The **main** log file of the admin server.
- admin.out: `stdout` from the admin server.
- admin_nodemanager.log: The **main** log from `nodemanager` on the admin server.
- admin_nodemanager.out: `stdout` from `nodemanager` on the admin server.
- admin_access.log: The log of http/https access to the admin server.
- ms1.log: The **main** log file of the ms1 managed server.
- ms1.out: `stdout` from the ms1 managed server.

- ms1_nodemanager.log: The **main** log from `nodemanager` on the ms1 managed server.
- ms1_nodemanager.out: `stdout` from `nodemanager` on the ms1 managed server.
- ms1_access.log: The log of http/https access to the ms1 managed server.

All logs in the above list for ms1 are repeated for each running managed server, with the logs being named for their originating managed server in each case.

In addition to these logs:

- Each JMS Server configured will have its log file with the name **<server>_ms<x>-jms_messages.log** (for example: **uim_jms_server_ms2-jms_messages.log**). By default, the JMS queue or topic logs are disabled. These logs temporarily can be enabled from weblogic console.
- When custom templates for the SAF agent are configured, it will have log file with the name **<server>_ms<x>-jms_messages.log** (for example: **uim_saf_agent_ms1-jms_messages.log**).

UIM Cloud Native and Oracle Enterprise Manager

UIM cloud native instances contain a deployment of the Oracle Enterprise Manager application, reachable at the admin server URL with the path "/em". However, the use of Enterprise Manager in this Kubernetes context is not supported. Do not use the Enterprise Manager to monitor UIM. Use standard Kubernetes pod-based monitoring and UIM cloud native logs and metrics to monitor UIM.

Recovering a UIM Cloud Native Database Schema

When the UIM DB Installer fails during an installation, it exits with an error message. You must then find and resolve the issue that caused the failure. You can re-run the DB Installer after the issue (for example, space issue or permissions issue) is rectified. You do not have to rollback the DB.

> **Note**
>
> Remember to uninstall the failed DB Installer helm chart before rerunning it. Contact Oracle Support for further assistance.

It is recommended that you always run the DB Installer with the logs directed to a Persistent Volume so that you can examine the log for errors. The log file is located at: *filestore*/*project-instance*/**uim-dbinstaller**/logs/DbVersionController.log.

When you install the Oracle Database schema for the first time and if the database schema installation fails, do the following:

1. Delete the new schema or use a new schema user name for the subsequent installation.
2. Restart the installation of the database schema from the beginning.

To recover a schema upgrade failure, do the following:

1. Fix the issue. Use the information in the log or error messages to fix the issue before you restart the upgrade process. For information about troubleshooting log or error messages, see *UIM Cloud Native System Administrator's Guide*.

Common Problems and Solutions

This section describes some common problems that you may experience because you have run a script or a command erroneously or you have not properly followed the recommended procedures and guidelines regarding setting up your cloud environment, components, tools, and services in your environment. This section provides possible solutions for such problems.

Domain Introspection Pod Does Not Start

Upon running **create-applications.sh** or **upgrade-applications.sh**, no change is observed. Running `kubectl get pods -n project --watch` shows that the introspector pod did not start at all.

The following are the potential causes and mitigations for this issue:

- WebLogic Kubernetes Operator (WKO) is not up or not healthy: Confirm if WKO is up by running `kubectl get pods -n $WLSKO_NS`. There should be one WKO pod in the `RUNNING` state. If there is a pod, check its logs. If a pod is not there, check if WKO is uninstalled. You may need to terminate an unhealthy pod or reinstall WKO.
- Project is not registered with WKO.
  Run the following command:

  ```
  helm get manifest -n $WLSKO_NS weblogic-operator | grep -i
  domainNamespaceLabelSelector

  kubectl get ns <project> -o yaml
  ```

  Your project namespace should include the label that appeared after `domainNamespaceLabelSelector` in the first command. If it is not listed, run `$COMMON_CNTK/scripts/register-namespace.sh -p <project> -t wlsko -l <label shown>`.

  For more details on registering wlsko with namespace, see "Registering the Namespace"

  Other causes are infrastructure related issues such as worker capacity and user RBAC.

In case the introspector does not start, the Operator may not monitoring your namespace or your namespace that is not tagged to right label which operator is monitoring. See https://oracle.github.io/weblogic-kubernetes-operator/managing-operators/common-mistakes/#forgetting-to-configure-the-operator-to-monitor-a-namespace for more information on Operator monitoring

Domain Introspection Pod Status

While the introspection is running, you can check the status of the introspection pod by running the following command:

```
kubectl get pods -n namespace

## healthy status looks like this

NAME                                        READY   STATUS    RESTARTS   AGE
project-instance-introspect-hzh9t   1/1     Running   0          3s
```

The READY field is showing 1/1, which indicates that the pod status is healthy.

If there is an issue accessing the image specified in the instance specification, then it shows the following:

```
NAME                                             READY   STATUS
RESTARTS   AGE
project-instance-introspect-r2d6j   0/1     ErrImagePull   0          5s

### OR

NAME                                             READY   STATUS
RESTARTS   AGE
project-instance-introspect-r2d6j   0/1     ImagePullBackOff   0          45s
```

This shows that the introspection pod status is not healthy. If the image can be pulled, it is possible that it took a long time to pull the image.

To resolve this issue, verify that the image name and the tag and that it is accessible from the repository by the pod.

You can also try the following:

- Increase the value of `podStartupDeadlineSeconds` in the instance specification. Start with a very high timeout value and then monitor the average time it takes, because it depends on the speed with which the images are downloaded and how busy your cluster is. Once you have a good idea of the average time, you can reduce the timeout values accordingly to a value that includes the average time and some buffer.
- Pull the container image manually on all Kubernetes nodes where the UIM cloud native pods can be started up.

Domain Introspection Errors Out

Some times, the domain introspector pod runs, but ends with an error.

To resolve this issue, run the following command and look for the causes:

```
kubectl logs introspector_pod -n project
```

The following are the possible causes for this issue:

- RCU Schema is pre-existing: If the logs shows the following, then RCU schema could be pre-existing:

  ```
  WLSDPLY-12409: createDomain failed to create the domain: Failed to write
  domain to /u01/oracle/user_projects/domains/domain: wlst.writeDomain(/u01/
  oracle/user_projects/domains/domain) failed : Error writing domain:
  64254: Error occurred in "OPSS Processing" phase execution
  64254: Encountered error:
  oracle.security.opss.tools.lifecycle.LifecycleException: Error during
  configuring DB security store. Exception
  oracle.security.opss.tools.lifecycle.LifecycleException: The schema
  FMW1_OPSS is already in use for security store(s).  Please create a new
  schema..
  64254: Check log for more detail.
  ```

This could happen because the database was reused or cloned from a UIM cloud native instance. If this is so, and you wish to reuse the RCU schema as well, provide the required secrets. For details, see "Reusing the Database State".
If you do not have the secrets required to reuse the RCU instance, you must use the UIM cloud native DB Installer to create a new RCU schema in the DB. Use this new schema in your `rcudb` secret. If you have login details for the old RCU users in your `rcudb` secret, you can use the UIM cloud native DB Installer to delete and re-create the RCU schema in place. Either of these options gives you a clean slate for your next attempt.

Finally, it is possible that this was a clean RCU schema but the introspector ran into an issue after RCU data population but before it could generate the wallet secret (opssWF). If this is the case, debug the introspector failure and then use the UIM cloud native DB Installer to delete and re-create the RCU schema in place before the next attempt.

- Fusion Middleware cannot access the RCU: If the introspector logs show the following error, then it means that Fusion Middleware could not access the schema inside the RCU DB.

```
WLSDPLY-12409: createDomain failed to create the domain: Failed to get FMW
infrastructure database defaults from the service table: Failed to get the
database defaults: Got exception when auto configuring the schema
component(s) with data obtained from shadow table:
Failed to build JDBC Connection object:
```

Typically, this happens when wrong values are entered while creating secrets for this deployment. Less often, the cause is a corrupted RCU DB or an invalid one. Re-create your secrets, verifying the credentials and drop and re-create the RCU DB.

Recovery After Introspection Error

If the introspection fails during instance creation, once you have gathered the required information and have a solution, delete the instance and then re-run the instance creation script with the fixed specification, model extension, or other environmental failure cause.

If the introspection fails while upgrading a running instance, then do the following:

1. Make the change to fix the introspection failure. Trigger an instance upgrade. If this results in successful introspection, the recovery process stops here.
2. If the instance upgrade in step 1 fails to trigger a fresh introspection, then do the following:
   a. Rollback to the last good Helm release by first running the `helm history -n` *`project project-instance`* command to identify the version in the output that matches the running instance (that is, before the upgrade that led to introspection failure). The timestamp on each version helps you identify the version. Once you know the "good" version, rollback to that version by running: `helm rollback -n` *`project project-instance version`*. Monitor the pods in the instance to ensure only the Admin server and the appropriate number of Managed Server pods are running.
   b. Upgrade the instance with the fixed specification.

Instance Deletion Errors with Timeout

You use the **delete-applications.sh** script to delete an instance that is no longer required. The script attempts to do this in a graceful manner and is configured to wait up to 10 minutes for any running pods to shut down. If the pods remain after this time, the script times out and exits with an error after showing the details of the leftover pods.

The leftover pods can be UIM pods (Admin Server, Managed Server) or the DBInstaller pod.

For the leftover UIM pods, see the WKO logs to identify why cleanup has not run. Delete the pods manually if necessary, using the **kubectl delete** commands.

For the leftover DBInstaller pod, this happens only if **install-database.sh** is interrupted or if it failed in its last run. This should have been identified and handled at that time itself. However, to complete the cleanup, run `helm ls -n` *project* to find the failed DBInstaller release, and then invoke `helm uninstall -n` *project release*. Monitor the pods in the project namespace until the DBInstaller pod disappears.

Changing the WebLogic Kubernetes Operator Log Level

Some situations may require analysis of the WKO logs. These logs can be certain kinds of introspection failures or unexpected behavior from the operator. The default log level for the Operator is **INFO**.

For information about changing the log level for debugging, see the documentation at: https://oracle.github.io/weblogic-kubernetes-operator/managing-operators/troubleshooting/#operator-and-conversion-webhook-logging-level.

Deleting and Re-creating the WLS Operator

You may need to delete a WLS operator and re-create it. You do this when you want to use a new version of the operator where upgrade is not possible, or when the installation is corrupted.

When the corresponding operator is removed, the existing UIM cloud native instances continue to function. However, any updates cannot be processed (when you run **upgrade-applications.sh**) or respond to the Kubernetes events such as the termination of a pod.

Go through WKO troubleshooting to avoid errors while installing the operator. To uninstall the operator, see Uninstall the Operator

Register namespaces using **RegisterNamespace** and **UnregisterNamespace** scripts from CNTK . You can install the operator following the instructions from WKO Documentation and then register all the projects again, one after the other as mentioned in Registering the Namespace.

one by following

Normally, the **remove-operator.sh** script fails if there are UIM cloud native projects registered with the operator. But you can use the **-f** flag to force the removal. When this is done, the script prints out the list of registered projects and reminds you to manually re-register them (by running **register-namespace.sh**) after reinstalling the operator.

You can install the operator as usual and then register all the projects again, one by one by running **register-namespace.sh -p *project* -t wlsko**.

Lost or Missing opssWF and opssWP Contents

For a UIM instance to successfully connect to a previously initialized set of DB schemas, it needs to have the opssWF (OPSS Wallet File) and opssWP (OPSS Wallet-file Password) secrets in place. The **$COMMON_CNTK/scripts/manage-app-credentials.sh** script can be used to set these up if they are not already present.

If these secrets or their contents are lost, you can delete and recreate the RCU schemas (using **$COMMON_CNTK/scripts/install-database.sh** with command code 2). This deletes data (such as some user preferences, MDS, and so on) stored in the RCU schemas and requires redeployment cartridges. On the other hand, if there is a WebLogic domain currently running against that DB (or its clone), the "exportEncryptionKey" offline WLST command can be run to dump out the "ewallet.p12" file. This command also takes a new encryption password. See "Oracle Fusion MiddleWare WLST Command Reference for Infrastructure

Security" for details. The contents of the resulting ewallet.p12 file can be used to recreate the opssWF secret, and the encryption password can be used to recreate the opssWP secret. This method is also suitable when a DB (or the clone of a DB) from a traditional UIM installation needs to be brought into UIM cloud native.

Clock Skew or Delay

When submitting JMS message over the Web Service queue, you might see the following in the SOAP response:

```
Security token failed to validate.
weblogic.xml.crypto.wss.SecurityTokenValidateResult@5f1aec15[status: false][msg
UNT Error:Message older than allowed MessageAge]
```

Oracle recommends synchronizing the time across all machines that are involved in communication. See "Synchronizing Time Across Servers" for more details. Implement Network Time Protocol (NTP) across the hosts involved, including the Kubernetes cluster hosts.

It is also possible to temporarily fix this through configuration by adding the following properties to **java_options** in the **app-uim** specification for each managed **server**.**managedServers**: project:

```
#JAVA_OPTIONS for all managed servers at project level java_options:
-Dweblogic.wsee.security.clock.skew=72000000
-Dweblogic.wsee.security.delay.max=72000000
```

Cartridge Deployment Error

When the secret for encrypted WebLogic password <*project*>-<*instance*>-*weblogic-encrypted-credentials* is incorrect, you may find the following errors during cartridge deployment:

```
[deployCartridge] Deployment of ora_uim_baseextpts (7.4.2.0.0) failed due to :
[deployCartridge] Exception: EJB Exception: ; nested exception is:
[deployCartridge]       java.lang.NoClassDefFoundError: org/springframework/
context/ApplicationContext
```

To resolve this issue:

1. Delete the secret: <*project*>-<*instance*>-*weblogic-encrypted-credentials*.
2. Generate the WebLogic encrypted password as follows:

   ```
   $ $COMMON_CNTK/scripts/install-database.sh -p project -i instance -
   s $SPEC_PATH -a uim -c 8
   ```

3. Restart the Managed server as follows:

   ```
   $ $COMMON_CNTK/scripts/restart-applications.sh -p project -i instance -
   s $SPEC_PATH -a uim -r ms
   ```

Show the encrypted merged model json file for Model in Image

Weblogic operator has scripts to show the encrypted merged model json file for model in image.

`~/weblogic-kubernetes-operator/operator/integration-tests/bash/show_merged_model.sh -h` will provide all the required parameters to use this script.

Sample usage:

```
show_merged_model.sh -i model-in-image:v1 -n sample-domain1-ns -p weblogic -d
domain1
```

Upgrading WebLogic Operator

To upgrade the WebLogic Operator, you have the following approaches:

- **Operator Upgrade:** Follow the Operator documentation for a standard upgrade process. Ensure that the target version is compatible with the current version within your Kubernetes cluster.
  Here are some points you need to consider before using this approach:
  - You do not need any additional Kubernetes resources.
  - You do not need to register namespace again.
  - You cannot test with a canary namespace.
  - It is very challenging to to roll back to the previous version.
- **PhasedCutover Approach:** To install the new WKO (WebLogic Kubernetes Operator), create a new namespace with a fresh label selector. Transition UIM namespaces by removing the old label and adding the new label to each respective namespace. Once all namespaces have successfully transitioned and are stable, proceed to uninstall the old WKO.
  Here are some points you need to consider before using this approach:
  - You can test with a canary namespace before full deployment.
  - You can perform a phased cutover while accommodating program timelines.
  - This approach supports an easy backout option by reverting the label change on UIM namespaces.
  - This approach requires modification of all UIM namespaces to use the new WKO.
  - Extra Kubernetes resources are in use until the old WKO is uninstalled.

Known Issues

This section describes known issues that you may come across, their causes, and the resolutions.

SituationalConfig NullPointerException

In the managed server logs, you might notice a stacktrace that indicates a NullPointerException in situational config.

This exception can be safely ignored.

Connectivity Issues During Cluster Re-size

When the cluster size changes, whether from the termination and re-creation of a pod, through an explicit upgrade to the cluster size, or due to a rolling restart, transient errors are logged as the system adjusts.

These transient errors can usually be ignored and stop after the cluster has stabilized with the correct number of Managed Servers in the Ready state.

If the error messages were to persist after a Ready state is achieved, then looking for secondary symptoms of a real problem would be appropriate. Such connectivity errors could result in orders that were inexplicably stuck or were otherwise processing abnormally.

While not an exhaustive list, some examples of these transient errors you may see in a managed server log are:

- An MDB is unable to connect to a JMS destination. The specific MDB and JMS destination can vary, such as:
  - `The Message-Driven EJB inventoryQueueListener is unable to connect to the JMS destination inventoryWSQueue.`
  - `The Message-Driven EJB ActivityListenerBean is unable to connect to the JMS destination inventoryActivityQueue.`
- Failed to Initialize JNDI context. Connection refused; No available router to destination. This type of error is seen in an instance where SAF is configured.
- Failed to process events for event type[AutomationEvents].
- Consumer destination was closed.

Upgrade Instance failed with spec.persistentvolumesource: Forbidden: is immutable after creation.

You may come across the following error when you run the commands for upgrading the UIM Helm chart:

```
Error: UPGRADE FAILED: cannot patch "<project>-<instance>-nfs-pv" with kind
PersistentVolume: PersistentVolume "<project>-<instance>-nfs-pv" is invalid:
spec.persistentvolumesource:
Forbidden: is immutable after creation
Error in upgrading UIM helm chart
```

Once created, the Persistent Volume Claim cannot be changed.

To resolve this issue:

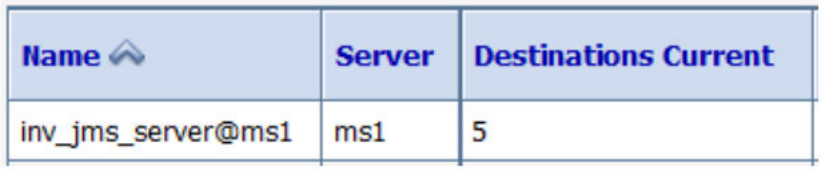
1. Disable NFS by setting the `nfs.enabled` parameter to **false** and run the upgrade-instance script. This removes the PV from the instance.
2. Enable it again by changing `nfs.enabled:` to **true** with the new values of NFS and then run upgrade-instance.

JMS Servers for Managed Servers are Reassigned to Remaining Managed Servers

When scaling down, the JMS servers for managed servers that do not exist are getting reassigned to remaining managed servers.

For example, for a JMS server when there is only 1 managed server running, you can see the server details as follows, in the WebLogic console:

Figure 13-1 Server Details for a JMS Server with One Managed Server

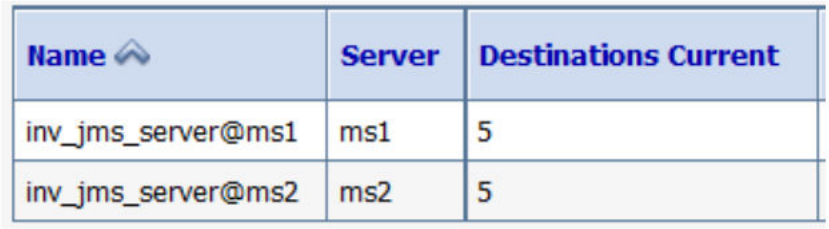


Name	Server	Destinations Current
inv_jms_server@ms1	ms1	5

Notice that `uim_jms_server@ms1` is targeting `ms1`.

When scaled to 2 Managed Servers, the WebLogic console shows the following server details:

Figure 13-2 Server Details of WebLogic Console with Two Managed Servers

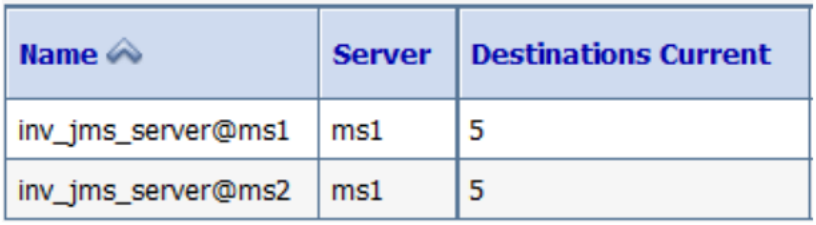


Name	Server	Destinations Current
inv_jms_server@ms1	ms1	5
inv_jms_server@ms2	ms2	5

Notice that `uim_jms_server@ms1` is targeting `ms1` and `uim_jms_server@ms2` is targeting `ms2`.

After scaling back to 1 managed server, the WebLogic console shows the following server details:

Figure 13-3 Server Details of WebLogic Console with One Managed Server



Name	Server	Destinations Current
inv_jms_server@ms1	ms1	5
inv_jms_server@ms2	ms1	5

Notice that the JMS Server `uim_jms_server@ms2` is not deleted and is targeting ms1.

This is completely expected behavior. This is a WebLogic feature and not to be mistaken for any inconsistency in the functionality.

14

Differences Between UIM Cloud Native and UIM Traditional Deployments

If you are moving from a traditional deployment of UIM to a cloud native deployment, this section describes the differences between UIM cloud native and UIM traditional.

- **Embedded LDAP and Open LDAP**

  You no longer need to create human users using the embedded LDAP capabilities of WebLogic Server.

  By default, UIM uses the WebLogic embedded LDAP as the authentication provider and all UIM fixed set of users are created in embedded LDAP during the creation of the instance. The Common cloud native toolkit provides a sample configuration that uses OpenLDAP to demonstrate how to integrate with external LDAP server for human users.

  A sample script for populating users to OpenLDAP can be found at: **$COMMON_CNTK/samples/credentials/manage-app-credentials.sh**. See "**Creating Users in OpenLDAP**" for more details.

- **WebLogic Domain Configuration**

  In a traditional deployment of UIM, the WebLogic domain configuration is done using WLST or the WebLogic Admin Console. In UIM cloud native, domain configuration is done by providing WDT metadata in the instance creation process. See "Extending the WebLogic Server Deploy Tooling (WDT) Model" for details.

  Do not perform WebLogic administrative activities such as changing the configuration, shutting down and restarting the server directly on the WebLogic Server cluster of the UIM cloud native instance. The same applies to the activities done using WebLogic Server Admin Console, WLST invocation, or any mechanism, other than those supplied by the specification files for updating and upgrading the UIM cloud native instance.

- **Incoming SAF and Outgoing SAF**

  For incoming SAF agents, the originator must use T3 over HTTP tunneling.

  Outgoing SAF mechanism has not changed.

- **UIM Solution Cartridges**

  Deploy the solution cartridges in UIM cloud native as follows.

  1. Build the customized image with solution cartridges.
  2. Deploy the cartridge using CMT or Design Studio on UIM cloud native running instance.
     In UIM Cloud Native, the cartridge management variables **wladmin.host.name** and **wladmin.host.port** are not required for deploying cartridges unlike in Traditional Deployments.

  See "Deploying Cartridges" for more information.

- **UIM Shared Storage File System**: Dependency on shared file system is removed in UIM cloud native. Persistent Volume Mounts are used only for logging purpose.

- **Custom WebServices**: The Custom Webservices are packaged in **custom.ear** in the traditional environment and are packaged in Inventory application in the cloud native. See "Customizing Images" to package the Custom WebServices.
- **UIM Application Roles**: The assignment of UIM application roles to UIM users can be achieved using sample script provided in **$COMMON_CNTK/samples/credentials/assign-role.sh**. The EM console can also be used like in traditional environment.
- **UIM User Interfaces**: All UIM user interfaces are still available with both UIM traditional and UIM cloud native deployments. The UIs can be accessed using the default hostname: *instance*.*project*.**uim.org** and port 30505, which is the default but configurable and the path that is necessary for the specific UI. For example, to access the Inventory UI, use:

  ```
  http://instance.project.uim.org:30505/Inventory
  ```

- **UIM API**: Accessing UIM through the traditional APIs such as the Web Services API and the REST API has not changed.
- **UIM System Configuration Properties**: UIM system configuration parameters can be controlled using the **system-config.properties** file. This configuration is still available in the UIM cloud native, but is managed differently. See "Customizing UIM Configuration Properties" for more details.

A

Migrating from Traefik Ingress Controller to Annotations-Based Generic Ingress Controller

This appendix describes how to migrate from Traefik Ingress Controller to annotations-based generic Ingress controller.

Prerequisites

Here are the prerequisites you need:

- Install annotation-based ingress controller.

Installing Generic Ingress Controller

To install generic ingress controller:

1. You can use any annotation-based ingress controller that supports standard Kubernetes ingress API. The samples for **haproxy ingressController** are provided.
2. For installation of HAProxy, the sample values are provided under $COMMON_CNTK/samples/charts/haproxy. For more information, see "About Load Balancing and Ingress Controller".

Migrating to Generic Ingress Controller for UIM CN

To migrate to a generic ingress controller:

1. Delete UIM CN Ingress:

   ```
   $COMMON_CNTK/scripts/delete-ingress.sh -p project -i instance -
   s $SPEC_PATH -a uim
   ```

2. Go to **$SPEC_PATH** of upgraded UIM release.
3. Update the **applications-base.yaml** specification file with **ingressController** as `GENERIC`:

   ```
   ingressController: "GENERIC"
   ```

4. Uncomment and provide the ingress annotations according to your ingress controller. The samples for HAProxy are provided. Make sure that you provide the corresponding **className** field, which is required to choose your ingress controller based on the **ingressClassName** value:

   ```
   ingress:
     className: "haproxy"
     annotations:
        haproxy.org/cookie-persistence: "uimhaproxycookie"
   ```

5. Update the **loadbalancerport** value in **applications-base.yaml** specification file to your ingess controller **loadbalancer** or **NodePort** port.

6. Based on the SSL-enablement, update annotations in **applications-base.yaml** specification file.
7. Update the **custom-config.properties** file for ATA details with the updated port number.
8. Create UIM Ingress:

   ```
   $COMMON_CNTK/scripts/create-ingress.sh -p project -i instance -
   s $SPEC_PATH -a uim
   ```

9. Upgrade UIM instance to reflect port changes:

   ```
   $COMMON_CNTK/scripts/upgrade-instance.sh -p project -i instance -
   s $SPEC_PATH -a uim
   ```

10. Verify if the application can be accessed using your ingressController port.

B

Managing Certificate Expiry

The utility scripts to analyze certificates used by UIM cloud native environment are provided. You can renew the expired certificates using this script. You must follow the prerequisites and postrequisites for this script.

Here are the guidelines for using the utility script:

- In case of SSL TERMINATE as ingress for UIM CN, you should run this script with the corresponding arguments and renew or verify the expiry of these certificates.
- This script supports renewal of certificates for any egress communication. If your IDP certificate is expired, you can replace or add a new certificate to the truststore of UIM CN using this script.

Prerequisites

Here are the prerequistes for managing the certificate expiry:

- You should have new SSL certificates that should be imported.
- UIM CN must be running over SSL Terminate at ingress.

Renewing Ingress Certificates

To renew ingress certificates:

1. Run the following command to verify ingress certificates:

   ```
   $COMMON_CNTK/scripts/manage-certificates.sh -p project -i instance -c
   verify -t ingress
   ```

   This command displays validity for all ingress certificates for UIM CN.
2. Run the following command to renew the ingress certificates:

   ```
   $COMMON_CNTK/scripts/manage-certificates.sh -p project -i instance -c
   import -t ingress
   ```

   This command prompts for the certificate and key inputs. Provide the new certificates so that all ingress certificates are renewed.

Importing Egress Certificates

To import egress certificates:

1. Run the following command to verify egress certificates:

   ```
   $COMMON_CNTK/scripts/manage-certificates.sh -p project -i instance -c
   verify -t egress
   ```

   This command displays validity for all egress certificates from the truststore of all services.

2. Run the following command to import egress certificates:

```
$COMMON_CNTK/scripts/manage-certificates.sh -p project -i instance -c
import -t egress
```

This command prompts for the certificate and alias name as inputs.

> **Note**
>
> If the provided alias name already exists, the older certificates will be overridden by the new certificate. If you want to retain your old certificate, provide a new alias name.

Postrequisites

Here are the postrequisites:

- Restart the application if you have imported egress certificates for the application.
- After renewal of ingress certificates, ensure that you have imported the new certificates into the client trust.

C

Migrating UIM_CNTK to COMMON_CNTK

From the UIM 8.0.0.0 release, UIM_CNTK is merged with COMMON_CNTK. For all operations on the UIM cloud native instance, use COMMON-CNTK.

The reasons for merging UIM cloud native toolkit with Common cloud native toolkit are as follows:

1. To converge all deployment scripts in a single place, that is in COMMON_CNTK.
2. To simplify the deployment process, UI-based deployment and configuration management tool (DCMT) are provided. These are delivered with Common cloud native toolkit.
3. Common configurations such as Ingress Controller, SSL configuration, and so on, can be managed at a single location instead of duplicating them in both UIM_CNTK and COMMON_CNTK.
4. To reduce the maintenance overhead of both artifacts.

Changes Due to Migration

This section lists the changes due to this migration.

Changes in Artifacts

The changes in artifacts are as follows:

- UIM_CNTK is now removed.
- Use COMMON_CNTK for all actions on UIM cloud native instance such as create, delete, upgrade, schema create, schema delete, schema upgrade, and so on.

Changes in Specification Files

The changes in specification files are as follows:

- The contents of **project.yaml** and **instance.yaml** are now available in **applications-base.yaml**, **app-uim.yaml** and **database.yaml**.
- **$SPEC_PATH/project.yaml** and **$SPEC_PATH/instance.yaml** are now placed at **$SPEC_PATH/project/instance/ <applications-base.yaml, app-uim.yaml, database.yaml>**.
- The locations of sample, logging, shape, and configuration files are changed as follows:

Table C-1 Changes in Specification Files

File/Directory	Old Location	New Location
loggingconfig.xml	uim-cntk/charts/uim/config/logging	$SPEC_PATH/project/instance/config/uim/logging
• custom-config.properties, • affinity-config.properties	uim-cntk/charts/uim/config/system-config	$SPEC_PATH/project/instance/config/uim /system-config

Table C-1 (Cont.) Changes in Specification Files

File/Directory	Old Location	New Location
topologyMapping	uim-cntk/charts/uim/topologyMapping	$SPEC_PATH/project/instance/config/uim /topologyMapping
All samples file	uim-cntk/samples	common-cntk/samples
All Shape files	uim-cntk/charts/uim/shapes	$SPEC_PATH/project/instance/shapes/<shape>/uim.yaml

> **Note**
>
> Run the **assemble-specification** script to use COMMON_CNTK. See "Assembling the Specifications" for information.

Changes in WLSKO Helper Operations

All the commands used to perform any operation on registration of namespace are now impacted because of this migration. Refer the following old and new commands to understand better.

Register Namespace

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/register-namespace.sh -p project -t wlsko -l
wlsko=enabled
```

- **New**:

```
$COMMON_CNTK/scripts/register-namespace.sh -p project -t wlsko -l
wlsko=enabled
```

Unregister Namespace

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/unregister-namespace.sh -p project -t wlsko -l
wlsko=enabled
```

- **New**:

```
$COMMON_CNTK/scripts/unregister-namespace.sh -p project -t wlsko -l
wlsko=enabled
```

Changes in Secrets

The command to create, delete, or update secrets is changed as follows.

Make sure you explicitly provide the following:

- Provide application name with -a uim option.
- Update **applications-base.yaml** by uncommenting applications name as **uim** and comment all other values.

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/manage-instance-credentials.sh -p project -i instance
<create/update/delete/verify> wlsadmin,opssWP,wlsRTE,rcudb,uimdb
```

- **New**:

```
$COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -
s $SPEC_PATH -a uim <create/update/delete/verify>
wlsadmin,opssWP,wlsRTE,rcudb,uimdb
```

> **Note**
>
> - In case of SSL communications, create one more **commonTrust** secret.
> - Do not mention the **trust.name** in the specification files. If the secret is available, it will be mounted to pods.

The following secret is common for all services with same namespace and instance name:

```
$COMMON_CNTK/scripts/manage-app-credentials.sh -p project -i instance -
s $SPEC_PATH create commonTrust
```

Changes in Embedded LDAP

All utility scripts and related files to manage LDAP users are moved under COMMON_CNTK/samples/credentials directory. To create an embedded LDAP user, perform the following command changes and update the uim.inventoryUsers with users list field in the **app-uim.yaml** specification file.

The changes are as follows:

- **Old**:

```
$UIM_CNTK/samples/credentials/manage-uim-credentials.sh  -p project -i
instance -c <create/update/delete/verify> -f <text file>
```

- **New**:

```
$COMMON_CNTK/samples/credentials/manage-uim-credentials.sh  -p project -i
instance -c <create/update/delete/verify> -f <text file>
```

Changes in Schema Operations

All schema operations will use **install-database.sh** script. This script reads the **database.yaml** present at **$SPEC_PATH/project/instance/database.yaml**.

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/install-uimdb.sh -p project -i instance -s $SPEC_PATH -c
<1/2/…./9>
```

- **New**:

```
$COMMON_CNTK/scripts/install-database.sh  -p project -i instance -
s $SPEC_PATH -a uim  -c <1/2/…./9>
```

Changes in Instance Operations

Instance operations commands copy the configuration files from **$SPEC_PATH/project/instance/config** directory to the corresponding locations in $COMMON_CNTK before running the command. Therefore, only the application configurations that are present at **$SPEC_PATH/project/instance/config** are applied. Any direct changes under the **$COMMON_CNTK/charts** directory overrides the previous ones.

Creation

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/create-instance.sh -p project -i instance -s $SPEC_PATH
```

- **New**:

```
$COMMON_CNTK/scripts/create-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Deletion

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/delete-instance.sh -p project -i instance
```

- **New**:

```
$COMMON_CNTK/scripts/delete-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Upgrade

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/upgrade-instance.sh -p project -i instance -s $SPEC_PATH
```

- **New**:

```
$COMMON_CNTK/scripts/upgrade-applications.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Ingress

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/create-ingress.sh -p project -i instance -s $SPEC_PATH
```

- **New**:

```
$COMMON_CNTK/scripts/create-ingress.sh -p project -i instance -
s $SPEC_PATH -a uim
```

Changes in Customizations

Use COMMON_CNTK for all customizations. The scripts and samples related to customizations such as **dump-merged-mode.sh**, **customExtensions**, are moved to the corresponding **COMMON_CNTK/scripts** and **COMMON_CNTK/samples/uim** directories.

The changes are as follows:

- **Old**:

```
$UIM_CNTK/scripts/create-instance.sh -p project -i instance -s $SPEC_PATH -
m <CUSTOMIZATION_LOCATION>
```

- **New**:

```
$COMMON_CNTK/scripts/create-instance.sh -p project -i instance -
s $SPEC_PATH -m <CUSTOMIZATION_LOCATION> -a uim
```

> **Note**
>
> In the customization WDT files, if you refer to any values from the **applications** specification files, Oracle recommends you place these values under **uim.custom** tag and not anywhere else.

Changes in Post-Deployment Operations

This section includes the changes in post-deployment operations.

Assign Roles

The changes are as follows:

- **Old**:

  ```
  $UIM_CNTK/samples/credentials/assign-role.sh -p project -I instance -f uim-user-roles.txt
  ```

- **New**:

  ```
  $COMMON_CNTK/samples/credentials/assign-role.sh -p project -I instance -f uim-user-roles.txt
  ```

Migrating from the Existing Files

This section provides the details about migration from the existing files.

Mapping the Existing Specification Files to New

Update **database.yaml**, **applications-base.yaml**, and **app-uim.yaml** at **$SPEC_PATH/project/instance** by copying the relevant configurations from your existing $SPEC_PATH **project.yaml** and **instance.yaml** files.

Following are the examples of mappings:

Table C-2 Examples of Specification File Mappings

UIM CNTK - $SPEC_PATH	COMMON CNTK - $SPEC_PATH

`#project.yaml`
 	
`image: "uim-cn-base:latest"`
	`#app-uim.yaml`
	
	`uim:`
	`  image:`
	`  name: "uim-cn-base”`
	`  tag: “latest"`

Table C-2 (Cont.) Examples of Specification File Mappings

UIM CNTK - $SPEC_PATH	COMMON CNTK - $SPEC_PATH

`# project.yaml`
 	
`dbinstaller:`
`  image: "uim-cn-db-installer:latest"`
	`#database.yaml`
	
	`uim-dbinstaller:`
	`    dbinstaller:`
	`        image:`
	`           name: "uim-cn-db-installer”`
	`            tag: “latest"`

`#project.yaml`
 	
`custom:`
`    enabled: true`
`    application: false`
`    jdbc: false`
`    jms: false`
`    saf: false`
`    wdtFiles:`
`     - _custom-domain-model.tpl`
	`#app-uim.yaml`
	
	`uim:`
	`    custom:`
	`        enabled: true`
	`        application: false`
	`        jdbc: false`
	`        jms: false`
	`        saf: false`
	`        wdtFiles:`
	`       - _custom-domain-model.tpl`

`#instance.yaml`
 	
`db:`
`    datasourcesPrimary:`
`        port: 1521`
`        rcuPort: 1521`
`    defaultTablespace: "SYSTEM"`
`    tempTablespace: "TEMP"`
 	
`    purge:`
`        enabled: false`
`        purgeCommand: ./ldPurge.sh report -ldid 450003,450005`
	`#database.yaml`
	
	`uim-dbinstaller:`
	`   db:`
	`        datasourcesPrimary:`
	`            port: 1521`
	`            rcuPort: 1521`
	`        defaultTablespace: "SYSTEM"`
	`        tempTablespace: "TEMP"`
	
	`        purge:`
	`           enabled: false`
	`           purgeCommand: ./ldPurge.sh report -ldid 450003,450005`

Table C-2 (Cont.) Examples of Specification File Mappings

UIM CNTK - $SPEC_PATH	COMMON CNTK - $SPEC_PATH

`#instance.yaml`
 	
`ssl:`
`  incoming: true`
`  strategy: TERMINATE`
`   trust:`
`    name: truststore`
	`#applications-base.yaml`
	
	`tls:`
	`  enabled: true`
	
	`(trust it will read from commonTrust secret, make sure you create it with all certificates using manage-credentials.sh script)`

Copying the Configuration Files

Copy the configurations such as logging, system-config, topologyMapping, siaMapping, and so on to the specification path location as follows:

- From the existing: `$UIM_CNTK/charts/uim/config/*`
- To a new: `$SPEC_PATH/project/instance/config/uim/*`

Performing the Operations

Use the updated commands from "Changes in Schema Operations" and "Changes in Instance Operations" to perform any operation on schema or instance using COMMON_CNTK.