Containerization Guide
Oracle Banking Liquidity Management
Release 14.5.0.0.0

Part Number F41688-01
May 2021

ORACLE
FINANCIAL SERVICES

Table of Contents

1. PREFACE 1-1
1.1 INTRODUCTION 1-1
1.2 AUDIENCE 1-1
1.3 DOCUMENTATION ACCESSIBILITY 1-1
1.4 RELATED DOCUMENTS 1-1
2. TECHNOLOGIES 2-1
2.1 DOCKER 2-1
2.1.1 Images and Containers 2-1
2.2 KUBERNETES (K8) 2-1
3. CONTAINERIZATION 3-1
3.1 DOCKER REGISTRY 3-1
3.2 DATABASE 3-1
3.3 BUILDING IMAGE 3-1
3.3.1 Image Creation 3-2
3.3.2 Updating Image for Patch/Customization 3-3
4. OBMA PRODUCTS DEPLOYMENT APPROACHES 4-1
4.1 INSTALLER 4-1
4.2 CONTAINERIZATION OF THE SERVICES USING TOMCAT 4-1
4.2.1 Using Jib Plugin and Tomcat Image 4-1
4.2.2 Pipeline Integration in Jenkins 4-3
4.2.3 Using War Artifacts Delivered in OSDC 4-4
4.2.4 Pipeline Integration in Jenkins 4-6
4.3 CONTAINERIZATION OF SERVICES USING WEBLOGIC 4-8
4.3.1 Using Pre-Built Weblogic Images 4-8
4.3.2 Running Weblogic Containers Using Weblogic Kubernetes Operator 4-9
4.4 DEPLOYING SERVICES WITHOUT DOCKER IMAGES 4-10
4.4.1 Deploying Applications to Tomcat without Docker Images 4-10
4.4.2 Deploying Applications to Weblogic without Docker Images 4-10
4.5 DEPLOYING SERVICES ON PRIVATE CLOUD USING DOCKER IMAGES 4-11

1. Preface

1.1 Introduction

This document provides information on how to deploy Oracle Banking Microservices Architecture (OBMA) products by creating a Docker image and deploying it in a Docker container or inside a Kubernetes (K8) cluster.

1.2 Audience

This document is intended for WebLogic admin or ops-web team who are responsible for installing OFSS Banking Products. The user of the guide should have pre-acquired skills in the below technologies to perform the steps mentioned in this guide:

- Docker
- Kubernetes
- Jenkins

1.3 Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/us/corporate/accessibility/index.html.

1.4 Related documents

For more information, refer to the following documents:

- Product Installation Guide

2. Technologies

2.1 Docker

2.1.1 Images and Containers

An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image.

A container is a standard unit of software that packages up code and all its dependencies. Hence, the application runs quickly and reliably from one environment to another. A Docker Container Image is a lightweight, standalone, executable package of software that includes everything needed to run an application - code, runtime, system tools, system libraries, and settings.

Container images become containers at runtime, and in case of Docker containers, the images become containers when they run on the engine. Containers are available for both Linux and Windows-based applications. The containerized software will always run the same code, regardless of the infrastructure. The container isolates software from its environment and ensures that it works uniformly despite differences for instance between Development, Staging, and Production.

2.2 Kubernetes (K8)

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery.

3. Containerization

3.1 Docker Registry

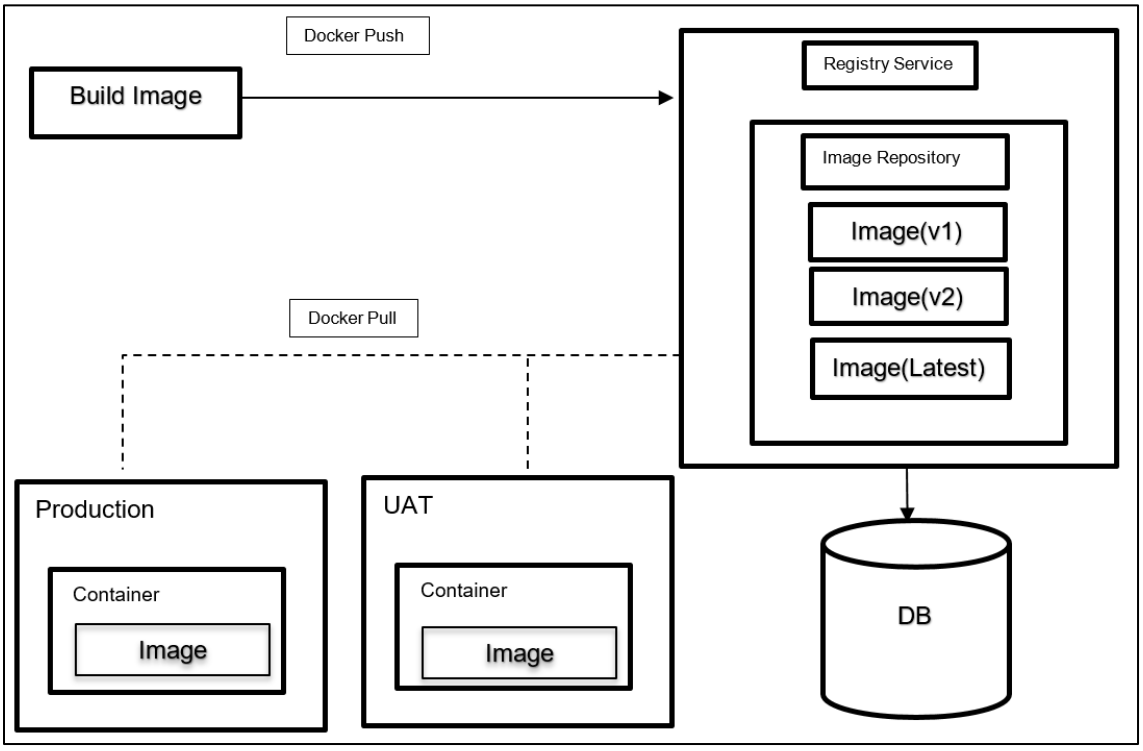
A Docker registry is a service for storing and retrieving Docker images. A registry contains a collection of one or more Docker image repositories. Each image repository contains one or more tagged images. Docker provides its own registry, the Docker Hub, but you may also use private or third-party registries. You need to register yourself in Oracle Container Registry to access images located in this registry.

3.2 Database

Database is not included inside Docker and database feature for High availability should be used.

3.3 Building Image

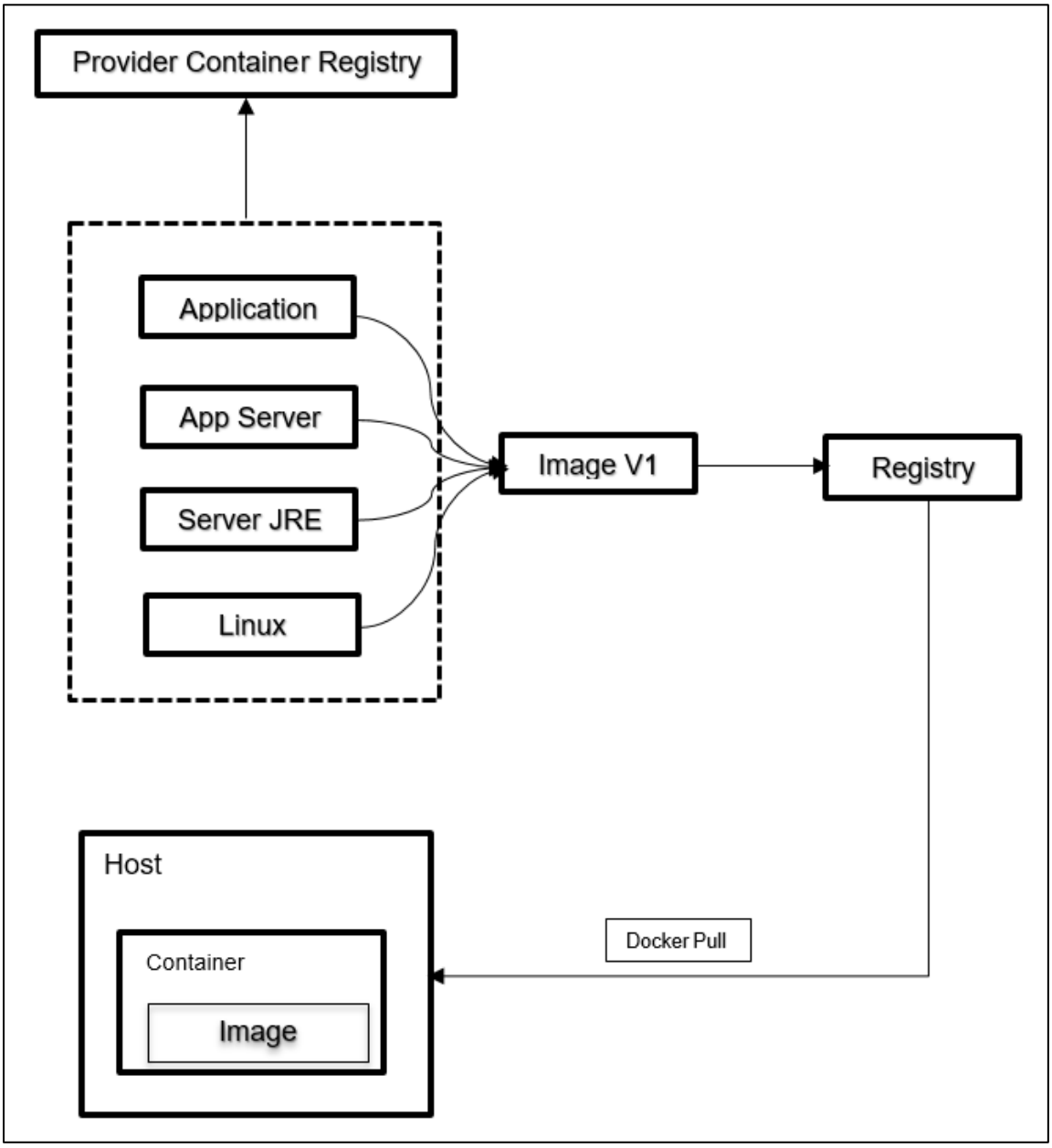
The process of building an image is shown below:



3.3.1 Image Creation

Image is layered consisting of following components:

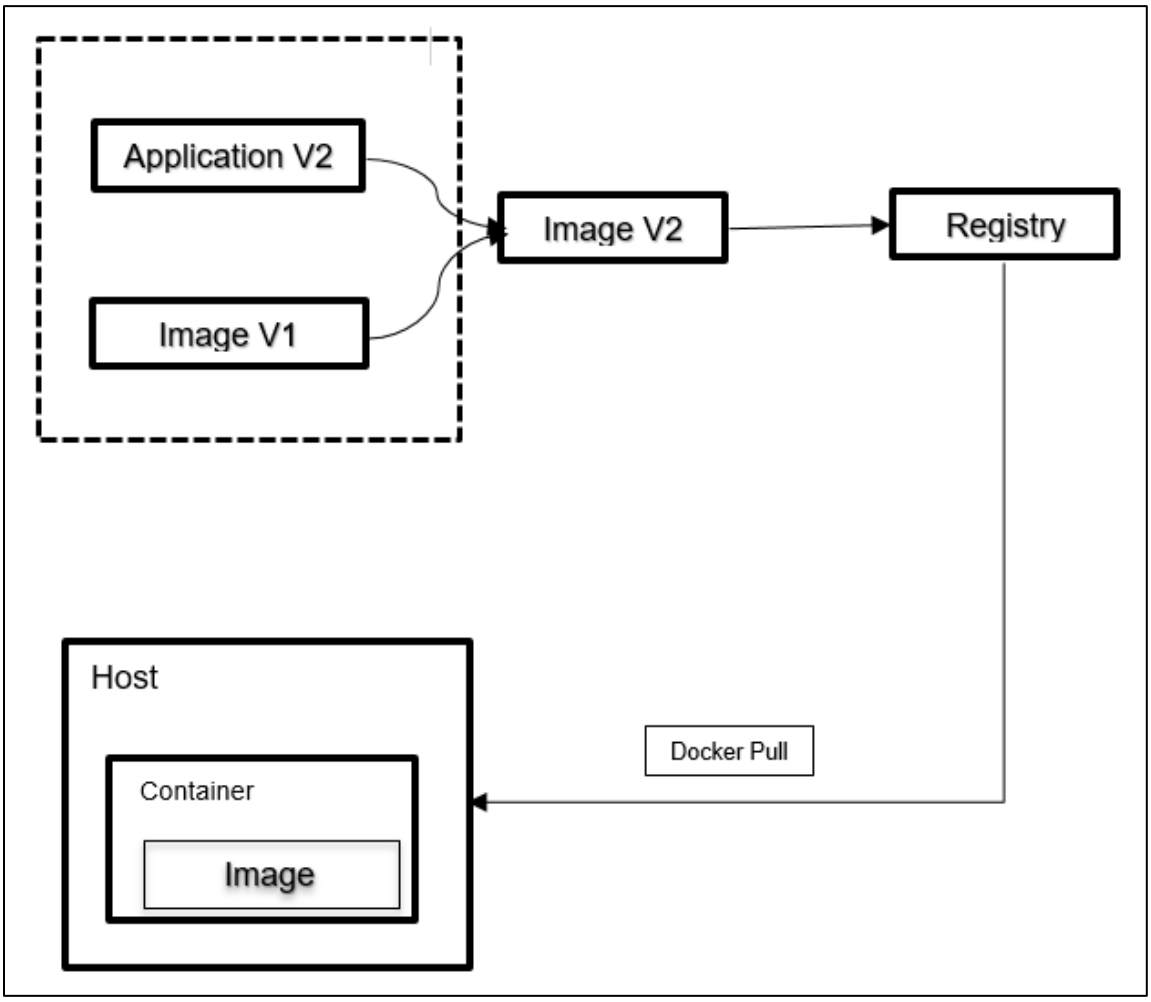
- Operating System (OS) Linux from provider Container Registry
- Java Runtime from provider Container Registry
- Application Server from provider Container Registry
- Applications from OBMA product installer



3.3.2 Updating Image for Patch/Customization

Modified Image is layered consisting of following components:

NOTE: Image needs to be updated from local Container Registry.



4. OBMA Products Deployment Approaches

4.1 Installer

Oracle banking OBMA Installer is available on OSDC for download, installer does the following:

- It Generates Deployment Artefacts (web archives) which can be deployed into the Application Server.
- The database scripts for both DDL and DML are executed with the web archives.

4.2 Containerization of the Services using Tomcat

This chapter describes the various methods that can be used to containerize a service in the OBMA Product stack.

4.2.1 Using Jib Plugin and Tomcat Image

This approach uses the google Jib plugin to integrate creation of Docker images along with gradle build. This approach can be used to create Docker images of components where the consulting or customer teams have access to code generated using OBMA Extensibility framework.

1. Update to *build.gradle* to include Jib plugin.
   id 'com.google.cloud.tools.jib' version '2.6.0'

2. Add Jib task in *build.gradle*.

```
jib {
    from {
        image = 'tomcat:<tag>'
    }
    to {
        auth{
            //it is ideal to use credHelper value here instead of
            username/passwd if it is configured using the below line
            // credHelper = '<credHelper_value>', else username/password to the
            // registry can be used
           username = 'docker_registry_username'
           password = 'docker_registry_passwork'
        }
        image = <docker_registry_name>/<image_name:image_version>'
    }
    container {
        appRoot = '/usr/local/tomcat/webapps/ROOT'
    }
}

tasks {
    build {
        dependsOn(tasks.jib)
    }
}
```

3. Run Gradle build using the below command:

   $ gradlew clean build

4. Test the Docker image as follows:

   a) Login to Registry using Docker *login 'registry_name'*.

   b) Pull the image from repository using:
   docker pull <docker_registry_name>/<image_name:image_version>.

   c) Run the docker image using *docker run -d -p 80:8080*:
   <docker_registry_name>/<image_name:image_version>.

   d) To pass *env* variable to your service to start use the below options:

   - *docker run -d -p <port> -v <Host Path>:/opt/logs/ <image> -e ds_jndi_1=jdbc/PLATO ds_db_host_1=<DBHOST> ds_db_port_1=<DB_HOST_PORT> ds_db_serviceid_1=<SID> ds_db_username_1=<USERNAME> ds_db_password_1=<PASSWORD>*
   - *docker run -d -v <Host Path>:/opt/logs/ --env-file <file> <image>*

4.2.2 Pipeline Integration in Jenkins

The Docker image creation using *Jib* plugin can be automated in the Continuous integration (CI) pipeline. The CI Pipeline is usually used to run automated tasks that build a source code at preconfigure intervals to enable automation of build for an application. The Jenkins pipeline can then be enhanced to support automated deployment of the Docker images in Continuous Deployment (CD) pipeline.

4.2.2.1 Pre-requisites

The pre-requisites for pipeline integration in Jenkins are as follows:

1. Jenkins installation
2. Docker engine installation on Jenkins VM
3. Network connectivity between docker registry and Jenkins VM
4. Gradle plugin for Jenkins installation
5. CredHelper Configuration

4.2.2.2 Automated Build – CI

The gradle build step should be added as a stage in Jenkins to trigger automated build for a service. This will also result in the image getting created and pushed to the mentioned Docker registry.

```
stage('Build and Test') {
        steps {
           dir (".") {
              sh './gradlew clean build --refresh-dependencies'
           }
        }
     }
```

4.2.2.3 Automated Deployment – CD

The codes used to auto-deploy an image on a VM with a vanilla Docker installation is detailed in the below Jenkins file.

```
def docker_image_name = "<image_name:image_version>"
def remote = [:]
remote.name = "dkx"
remote.host = "<docker_hostname>"
remote.allowAnyHosts = true
remote.user = "<username>"
remote.password = "<passwd>"

stage('Login to remote box') {
 steps {
    withCredentials([usernamePassword(credentialsId: 'sshUserAcct',
passwordVariable: 'password', usernameVariable: 'userName')]) {
        sshCommand remote: remote, command: 'docker pull
<docker_registry_name>/' + docker_image_name
        sshCommand remote: remote, command: 'docker run -d -p 80:8080
<docker_registry_name>/' + docker_image_name
    }
  }
}
```

4.2.3 Using War Artifacts Delivered in OSDC

This approach should be used if a consulting or partner installation team does not have access to source code of a service and wants to containerize the product applications. This approach uses the war files shipped under the product installer in OSDC portal. The below section details individual steps to be followed to create the Docker images for each service.

4.2.3.1 Prerequisites

The pre-requisites are as follows:

1. Docker Engine should be up and running on the VM to perform the following operations
2. Proxy setting in */etc/environment* file must be updated using root permissions.

4.2.3.2 Create a sample docker file to run a service in tomcat

Create a sample Docker file as follows:

1. Create a separate directory structure for each service.

   $mkdir <service_name>/docker

   $cd <service_name>/docker

2. Copy the service's war file from the installer to the path *<service_name>/docker*

   $cp <service_name>.war <service_name>/docker/.

3. Create a Docker file in the docker directory for the service.

   $vi Dockerfile

   NOTE: Services Dockerfile should have "tomcat:<tag_name>" as a base image.

   FROM tomcat:<tag>

4. Pass the appropriate build arguments to docker.

```
ARG application_context=<application context name>
ARG war_file_name=<microservices war file name>
ARG shutdown_port=<tomcat server shutdown port value>
ARG http_port=<tomcat server http port value>
ARG redirect_port=<tomcat server redirect port value>
ARG ajp_port=<tomcat server redirect port value>
```

application_context - This value represents the context root of the Plato application. This value must be passed as an argument in the docker file.
war_file_name - This value represents the name of the war file of the application that is present in the local system where the docker image is being built.
shutdown_port - Represents the shutdown port in the Tomcat server.
http_port - Represents the HTTP port on which the application will be available for accessing via REST API.
redirect_port - Represents the redirect port in the Tomcat server.
ajp_port - Represents the AJP port in the Tomcat server.
Note***: The port values are not mandatory to give in case the docker image is getting built for deployment in Kubernetes but it is mandatory in case of docker-compose because container port values should unique for the same. The port values are not passed/mentioned in the Dockerfile then the default will be used for while building the image.

5. Expose the container *http_port*.

   EXPOSE <http_port>

6. The completed service's Docker file is shown below.

```
FROM tomcat:<tag>

ARG application_context=plato-discovery-service
ARG war_file_name=plato-discovery-service-1.0.3.war
ARG shutdown_port=5008
ARG http_port=5005
ARG redirect_port=5007
ARG ajp_port=5006

COPY plato-discovery-service-1.0.3.war /usr/local/tomcat/webapps/
EXPOSE 5005

CMD ["catalina.sh", "run"]
```

4.2.3.3 Test the Docker image

Perform the following steps to test the Docker image:

1. Run the Docker image using the below option:
   - *docker run -d -p 80:8080 <docker_registry_name>/<image_name:image_version>*
2. To pass *env* variable to your service use the below options:
   - *docker run -d -p <port> -v <Host Path>:/opt/logs/ <image> -e ds_jndi_1=jdbc/PLATO ds_db_host_1=<DBHOST> ds_db_port_1=<DB_HOST_PORT> ds_db_serviceid_1=<SID> ds_db_username_1=<USERNAME> ds_db_password_1=<PASSWORD>*
   - *docker run -d -v <Host Path>:/opt/logs/ --env-file <file> <image>*

4.2.4 Pipeline Integration in Jenkins

The Docker image creation can be automated in the CI pipeline. The CI Pipeline is used to run automated tasks that build a source code at pre-configure intervals to enable automation of build for an application. The Jenkins pipeline can be enhanced to support automated deployment of the Docker images in CD pipeline.

4.2.4.1 Pre-requisites

The pre-requisites for pipeline integration in Jenkins are as follows:

1. Jenkins installation
2. Docker engine installation on Jenkins VM
3. Network connectivity between Docker registry and Jenkins VM
4. Gradle plugin for Jenkins installation
5. CredHelper Configuration

4.2.4.2 Automated Build – CI

The gradle build step should be added as a stage in Jenkins to trigger automated build for a service. This will result in the image getting created, and pushed to the mentioned Docker registry.

```
stage('Build and Publish Docker Image') {
    steps {
        script {
            /* provide the Dockerfile and the context of the build which is the
directory which contains the Dockerfile */
            def image = docker.build( docker_image_name, "-f " + dockerfile_path
+ "/Dockerfile " + dockerfile_path)

            /* once the image is complete, this runs the image and you can verify
if the image is correct by adding tests */
            image.inside {
                sh 'echo "Put Tests for your new image here"'
            }

            /* Replace the docker repo with your repo and the login with a cre-
dential in your Jenkins that has permission to push to your docker repo */
            docker.withRegistry('https://' + docker_registry + '/v2/',
docker_registry_login) {
                image.push(docker_image_version)
            }
        }
    }
}
```

4.2.4.3 Automated Deployment – CD

The codes used to auto-deploy an image on a VM with a vanilla Docker installation is detailed in the below Jenkins file.

```
def docker_image_name = "<image_name:image_version>"
def remote = [:]
remote.name = "dkx"
remote.host = "<docker_hostname>"
remote.allowAnyHosts = true
remote.user = "<username>"
remote.password = "<passwd>"

stage('Login to remote box') {
 steps {
    withCredentials([usernamePassword(credentialsId: 'sshUserAcct',
passwordVariable: 'password', usernameVariable: 'userName')]) {
        sshCommand remote: remote, command: 'docker pull
<docker_registry_name>/' + docker_image_name
        sshCommand remote: remote, command: 'docker run -d -p 80:8080
<docker_registry_name>/' + docker_image_name
    }
  }
}
```

4.3 Containerization of Services Using Weblogic

In this section, the various options to build service containers using Oracle Weblogic images are described.

4.3.1 Using Pre-Built Weblogic Images

This section details the steps to deploy the services on a Weblogic Server running in a Docker container.

4.3.1.1 Prerequisites

The pre-requisites are as follows:

1. Verify proxy settings on the VM where Weblogic image need to run.
2. Log in to the Oracle Container Registry portal, and accept the license agreements before pulling the Docker images.
3. Sudo access to the VM to run commands as root.

   Create a file domain.properties with the below content:
   username=””
   password=””

4.3.1.2 Pull the weblogic Docker image

Run the following command:

docker pull container-registry.oracle.com/middleware/weblogic:12.2.1.4

4.3.1.3 Run the weblogic image

Run the following command:

docker run -d -p 7002:7001 -p 9004:9002 -v $PWD:/u01/oracle/properties container-registry.oracle.com/middleware/weblogic:12.2.1.3

4.3.1.4 Deploy the application

Access the console at *<hostname>:9004/console* with admin credentials, and deploy the service.

4.3.1.5 Creating domains in weblogic and deploying applications

Deploy the application in custom domains. For information on deploying applications, refer to the below documentation:

https://github.com/oracle/docker-images/tree/main/OracleWebLogic/samples/12213-domain-home-in-image

4.3.2 Running Weblogic Containers Using Weblogic Kubernetes Operator

4.3.2.1 Prerequisites

The pre-requisites are as follows:

1. Docker engine installation
2. Kubernetes cluster
3. Access to Kubernetes operator

4.3.2.2 Install and manage weblogic domains using Kubernetes operator

An operator is an application-specific controller that extends Kubernetes to create, configure, and manage instances of complex applications. The Oracle WebLogic Server Kubernetes Operator simplifies the management and operation of WebLogic domains and deployments. A detailed guide to installation and management of weblogic domains is in the below link:

https://oracle.github.io/weblogic-kubernetes-operator/

4.4 Deploying Services without Docker Images

This section details steps to be followed to deploy product services without using Docker images.

4.4.1 Deploying Applications to Tomcat without Docker Images

4.4.1.1 Prerequisites

The pre-requisites are as follows:

1. Tomcat installation
2. Jenkins installation

4.4.1.2 Manual deployment

1. Download and place the individual war files for services in a common directory.
2. Follow the steps in the below link to deploy the individual service wars:

   https://tomcat.apache.org/tomcat-10.0-doc/deployer-howto.html

4.4.1.3 Deployment using scripts

Alternatively, the war files can be configured to be deployed using a Jenkins pipeline. The *deploy to container* plugin should be used for configuration.

4.4.2 Deploying Applications to Weblogic without Docker Images

4.4.2.1 Prerequisites

The pre-requisites are as follows:

1. Weblogic installation
2. Jenkins installation

4.4.2.2 Manual deployment

Perform the following steps:

1. Download and place the individual war files for services in a common directory
2. Follow the steps in the below link to deploy the individual service wars.
   https://docs.oracle.com/cd/E19424-01/820-4807/war-weblogic/index.html

4.4.2.3 Deployment using Jenkins

Alternatively, the war files can be configured to be deployed using a Jenkins pipeline. The Deploy WebLogic should be used for configuration. It is recommended to see if the version of the plugin has any vulnerabilities.

4.5 Deploying Services on Private Cloud using Docker Images

When deploying the services on Docker image in private cloud, it is important to build custom images of Weblogic and Tomcat using *openjdk 8* unless the appropriate license requirements are met with the built jdk versions in Weblogic.

For Weblogic use the below steps to build the base weblogic images:
https://github.com/oracle/docker-images/tree/main/OracleWebLogic

For Tomcat use the below steps to build the base tomcat images:
https://hub.docker.com/_/tomcat

The pre-built OpenJDK 8 image is available in this link.

Containerization Guide

May 2021

Version 14.5.0.0.0

Oracle Financial Services Software Limited
Oracle Park
Off Western Express Highway
Goregaon (East)
Mumbai, Maharashtra 400 063
India

Worldwide Inquiries:
Phone: +91 22 6718 3000
Fax: +91 22 6718 3001
https://www.oracle.com/industries/financial-services/index.html