Looking to explore Red Hat OpenShift streamlined CI/CD workflows to run your ACE container natively on Red Hat OpenShift? In this post we show how to run a simplistic pipeline on OpenShift. You don’t have an OpenShift cluster? No worries. You can also follow this tutorial on your local workstation using Minishift (for OpenShift 3) or CodeReady (for OpenShift 4) which gives you a local cluster for development purposes.

You can download CodeReady Containers and Minishift from;

<https://developers.redhat.com/products/codeready-containers>

<https://www.okd.io/minishift/>

**Key concepts and definitions**

Before we start, we suggest you refresh your Kubernetes and OpenShift knowledge. Below we summarized some of the key concepts that is referred from OpenShift documentation. (<https://docs.openshift.com/container-platform/4.2/>)

**Image:**

An **image** holds a set of software that is ready to run, while a **container** is a running instance of a container image.

**Image registry:**

An image registry is a content server that can store and serve container images. (For example: [registry.redhat.io](http://registry.redhat.io)) A registry contains a collection of one or more image repositories, which contain one or more tagged images. Red Hat provides a registry at registry.redhat.io for subscribers. OpenShift Container Platform can also supply its own internal registry for managing custom container images.

**Image repository:**

An image repository is a collection of related container images and tags identifying them. For example, the OpenShift Jenkins images are in the repository.

**Imagestream:**

An **imagestream** provides a way of storing different versions of the same basic image. An imagestream and its associated tags provide an abstraction for referencing container images from within OpenShift Container Platform. Imagestreams do not contain actual image data, but present a single virtual view of related images, similar to an image repository.

**Project:**

A **project** allows a community of users to organize and manage their content in isolation from other communities.

**Application:**

You can deploy an OpenShift application from components that include source or binary code, images, and templates. As a result of application creation OpenShift objects that will build, deploy, and run the application will also be created.

**Template:**

Another concept unique to Openshift is the [Template](https://docs.openshift.com/container-platform/3.9/dev_guide/templates.html) object. An OpenShift Template bundles some number of objects that are related from a functional point of view.

Kubernetes is a vibrant laboratory, and so there are similar efforts in the community to group an “application”’s objects and resources in a way convenient for deployment, management, and organization. One of the other popular systems in this space is Microsoft/Deis’s [Helm](https://helm.sh/).

In terms of application promotion, **Templates** can serve as the starting point for setting up resources in a given staging environment, especially with the parameterisation capabilities. Additional post-instantiation modifications are very conceivable though when applications move through a promotion pipeline.

**Deployment Config:**

*Deployments* and *DeploymentConfigs* in OpenShift are API objects that provide two similar but different methods for fine-grained management over common user applications. They are composed of the following separate API objects:

* A DeploymentConfig or a Deployment, either of which describes the desired state of a particular component of the application as a Pod template.
* DeploymentConfigs involve one or more *ReplicationControllers*, which contain a point-in-time record of the state of a DeploymentConfig as a Pod template. Similarly, Deployments involve one or more *ReplicaSets*, a successor of ReplicationControllers.
* One or more Pods, which represent an instance of a particular version of an application.

## Why CI/CD for integration?

There are many places to read about Continuous Integration (<https://www.ibm.com/cloud/learn/continuous-integration>) and Continuous Delivery (https://www.ibm.com/cloud/learn/continuous-delivery) so we won’t cover them here. Instead, let’s take a step back and ask the question why.

CI/CD practices emerged as one of the key enablers of agility. It is essential to streamline and automate the cycle of understanding a requirement, implementing that requirement, and indeed implementing changes in the requirement to the delivered solution. Whilst CI/CD is most commonly discussed in the context of application development, it is just as relevant to integration development. In fact, considering that integration solutions are the enablers for many business-critical applications to interact with other systems effectively, it is crucial that the integration artefacts can be created with the same agility as the application components. We need to simplify and accelerate the building and deployment the integrations yet retain quality assurance and adhere to security policies.

The good news is Red Hat OpenShift and IBM App Connect together work together to make it easy to build your CI/CD pipeline for your containerized integration solutions.

A CI/CD pipeline includes many different steps, from compilation and verification of the source code to testing and then on to the creation of an executable and ultimately deployment into an environment. Ideally, we want to get to a position where all of this can be triggered from a change in a source code repository or initiated via command or user interface. By the end of this article you have learnt how to set up such a pipeline for an IBM App Connect integration.

For further reading, we can highly recommend reviewing “Accelerating Modernization with Agile Integration” (<http://www.redbooks.ibm.com/abstracts/sg248452.html?Open>)

to understand the importance of CI/CD for the adoption of agile integration and to explore different hands-on examples.

## Why Red Hat OpenShift for CI/CD?

Red Hat OpenShift includes streamlined CI/CD workflows to help teams get to production faster. Whilst Jenkins is the widespread way of approaching CI/CD, Red Hat OpenShift is evolving to offer OpenShift Pipelines, which is based on the upstream project Tekton and is available as a Developer Preview release. Thus, you have the choice of proceeding with a generic and mature solution based on Jenkins pipelines or exploring OpenShift Pipelines which is a Kubernetes-native framework aiming to better support pipeline-as-code and GitOps approaches common in cloud-native solutions in the future.

Before going into further details, there are few concepts about the OpenShift build process that have to be understood.

A **build** is the general term for the process of transforming the application’s source or binary artefacts from some input source into a resulting object which most often is a runnable container image.

A **BuildConfig** is a configuration object which defines the characteristics of a specific build process.

A **build strategy** is one of the choices specified in the BuildConfig, which decides which of the pre-defined types of build will be performed.

A build can be triggered via a number of inputs such as:

* when the source code changes (e.g. via a webhook call from a GitHub repository)
* when the base image used for the build changes
* when a user manually requests a new build (e.g. via command line or UI).

Once built, resources are published as specified by the build configuration (BuildConfig), typically to the OpenShift registry. The process is characterized by the *build strategy* specified in the BuildConfig and one or more input sources. OpenShift supports a number of build strategies:

* **Docker build** invokes the docker build command, and it expects a ***Dockerfile*** and all required artefacts in it to produce a runnable image.
* **Source-to-Image (S2I) build** uses a specially prepared base image that accepts application source code to be injected so that it can assemble a new runnable image.
* **Custom build** can run plain Docker images as a base embedded with build process logic to produce objects that are specified.
* **Pipeline build** allows you to define a CI/CD pipeline workflow for execution by the Jenkins pipeline plugin. The workflow is defined by a Jenkinsfile either embedded directly in the build configuration or supplied in a Git repository and referenced by the build configuration.

The product documentation is a great place to grasp a more in-depth understanding. (https://docs.openshift.com/container-platform/4.2/builds/understanding-image-builds.html)

## Scenario Overview

In this example we will demonstrate how to build a containerized ACE application on OpenShift using native Jenkins pipelines.

The following diagram shows the steps included in the example pipeline:

**A screenshot of a cell phone

Description automatically generated**

When a commit triggers a pipeline execution:

* OpenShift takes the BAR artefact from GitHub Repository and builds a Docker image by layering the BAR file on top of ACE image pulled from DockerHub
* The new Docker image is then pushed to local image repository with a new tag.
* The tagged image is deployed in a fresh new container in the environment.

Note that we have the option to add another stage of building the BAR file as part of the pipeline but for this example we will just focus on building the image.

We will follow along using both the web console and the command-line interface so that you can choose whichever option best fits your need.

### Create Jenkins Application

Red Hat OpenShift provides an application template for running Jenkins in a container image, which can be used to set up a basic flow for continuous testing, integration, and delivery. Perform the following steps to create a Jenkins application.

1. Log in to Red Hat OpenShift web console.

A screenshot of a cell phone

Description automatically generated

1. Click the **Create Project** button to create a new project.

A screenshot of a cell phone

Description automatically generated

Alternatively, you can create your project using CLI:

**oc project ace**

1. Click the Catalog tab and select Jenkins (Ephemeral).

A screenshot of a cell phone

Description automatically generated

We have selected jenkins-ephemeral which is practical for development and test purposes. It uses ephemeral storage meaning that on pod restart, all data is lost. If you want your Jenkins application data to survive a pod restart you should choose the jenkis which uses a persistent volume store.

1. Pop up window appears to configure Jenkins Application. The first page displays information about the template. Click Next to continue to the next page.

A screenshot of a cell phone

Description automatically generated

1. Accept the default values and click Create.

A screenshot of a cell phone

Description automatically generated

1. Review the results and click Continue to the project overview.

A screenshot of a cell phone

Description automatically generated

Alternatively, you can create your Jenkins application using CLI:

**oc new-app jenkins-ephemeral**

As a result, Jenkins deployment will be created in your project, and a pod will be running with two services created: one for the Jenkins webui and the other for the jenkins-jnlp service for Jenkins slave/agent.

A screenshot of a cell phone

Description automatically generated

1. Open a Web browser on your workstation and type the route URL provided in the Jenkins service. As OAuth integration is enabled by default you can log in to Jenkins web console using your OpenShift credentials.

Click **Log in with OpenShift**.

A screenshot of a cell phone

Description automatically generated

1. Authorize Jenkins to access to your OpenShift account.

A screenshot of a social media post

Description automatically generated

You will be redirected to Jenkins web console.

A screenshot of a social media post

Description automatically generated

### Create Docker build configuration

As mentioned earlier in order to build a docker image on OpenShift platform we can make use of BuildConfig which can be defined as yaml or json.

1. Create a new file named ace-app-bc.yaml with the following content:

kind: BuildConfig

apiVersion: build.openshift.io/v1

metadata:

name: ace-app-bc

labels:

app: ace-app

annotations:

description: Defines how to build the integration application

spec:

strategy:

type: Docker

dockerStrategy: {}

source:

type: Git

dockerfile: |-

FROM ibmcom/ace

RUN mkdir bars

COPY myintegrationapp.bar /home/aceuser/bars

RUN ace\_compile\_bars.sh

git:

uri: 'https://github.com/myrepogit/ace.git'

output:

to:

kind: ImageStreamTag

name: 'ace-app:latest'

resources: {}

successfulBuildsHistoryLimit: 5

failedBuildsHistoryLimit: 5

postCommit: {}

triggers: []

|  |  |
| --- | --- |
| **strategy section**  strategy:  type: Docker  dockerStrategy:  pullSecret:  name: builder-dockercfg-gdj2c  env:  - name: LICENSE  value: accept | The **strategy section** defines that we will employ Docker build strategy which will result in the creation of a new ACE container image. |
| **source section**  source:  type: Git  dockerfile: |-  FROM ibmcom/ace  RUN mkdir bars  COPY myintegrationapp.bar /home/aceuser/bars  RUN ace\_compile\_bars.sh  git:  uri: 'https://github.com/myrepogit/ace.git' | The **source section** defines the input content for builds to operate on. In this example we provide an inline Dockerfile definition and a Git repository which stores our BAR file.  Notice that we use a COPY instruction in the Dockerfile; to copy the file myintegrationapp.bar into /home/aceuser/bars directory of the image to be created. Where is that file coming from?  That is the BAR file fetched from Git repository. When using a Docker strategy, you can add all defined input into your container image using the ADD and COPY instructions in your Dockerfile. The files fetched from Git repository will be copied into the same directory. |
| **output section**  output:  to:  kind: ImageStreamTag  name: 'ace-app:latest' | After the container image is successfully built, it is pushed into the integrated OpenShift Container Platform registry and tagged as latest in the specified imagestream. |

1. Navigate to Add and click Import YAML.

A close up of a logo

Description automatically generated

1. Upload the BuildConfig file by dragging & dropping, selecting it, or pasting from the clipboard and click create.

A screenshot of a social media post

Description automatically generated

1. Outcome of a successful import is a BuildConfig which you can access using the OpenShift Web Console, choose Builds -> Build Configs

Alternatively, you can create the BuildConfig using CLI:

**oc create -f ace-app-bc.yaml**

### Create Jenkins Pipeline

In this stage we will create a minimalist pipeline workflow that will first build an ACE image and then deploy. So, our pipeline consists of just two steps. We will consume ace-app-bc BuildConfig to trigger the ACE image build.

1. Create a build configuration file named ace-app-pipeline.yaml with the following content:

apiVersion: build.openshift.io/v1

kind: BuildConfig

metadata:

labels:

app: ace-app

name: ace-app-pipeline

spec:

strategy:

jenkinsPipelineStrategy:

jenkinsfile: |-

pipeline script goes here.

type: JenkinsPipeline

failedBuildsHistoryLimit: 5

nodeSelector: {}

output: {}

postCommit: {}

resources: {}

runPolicy: Serial

source:

type: None

successfulBuildsHistoryLimit: 5

triggers: []

Note that jenkinsfile section is explained further down the page.

1. Navigate to the Add to Project and click Import YAML.

A close up of a logo

Description automatically generated

1. Upload the BuildConfig file by dragging & dropping, selecting it, or pasting from the clipboard and click Create.

A screenshot of a cell phone

Description automatically generated

Alternatively, you can create the pipeline using CLI:

**oc create -f ace-app-pipeline.yaml**

1. Outcome of a successful import is a BuildConfig which you can access using the OpenShift Web Console, choose Builds -> Build Configs

Let’s explore what we describe in Jenkinsfile section of the ace-app-pipeline BuildConfig;

pipeline {

agent {

**node** {

label 'nodejs'

}

}

options {

timeout(time: 20, unit: 'MINUTES')

}

stages {

stage('build') {

steps {

script {

openshift.withCluster() {

openshift.withProject() {

**echo** "Using project: ${openshift.project()}"

openshift.selector("bc", "ace-app-bc").startBuild("--wait")

}

}

}

}

}

stage('deploy') {

when {

expression {

openshift.withCluster() {

return !openshift.selector('dc', 'ace-app').exists()

}

}

}

steps {

script {

openshift.withCluster() {

openshift.withProject() {

**def** created = openshift.newApp("ace-app:latest", "--name=ace-app", "--env=LICENSE=accept")

**echo** "new-app created ${created.count()} objects named: ${created.names()}"

created.narrow('svc').expose("--name=ace-console", "--port=7600")

created.narrow('svc').expose("--name=ace-traffic", "--port=7800")

}

}

}

}

}

}

}

Stage 0: Declare Jenkins Agent

We will declare Jenkins agent (also known as slave) in which the pipeline will be executing. OpenShift provides three images that are suitable for use as Jenkins agents: **Base**, **Maven**, and **Node.js** images. We will be using Node.js agent to execute our pipeline.

agent {

**node** {

label 'nodejs'

}

}

Stage 1: Build

At this stage, openshift.selector selects the build config named ace-app which we defined in the previous step. Using that build config, it initiates a build. After the build is complete, you will see an image named ace-app tagged as latest.

stage('build') {

steps {

script {

openshift.withCluster() {

openshift.withProject() {

**echo** "Using project: ${openshift.project()}"

openshift.selector("bc", "ace-app").startBuild("--wait")

}

}

}

}

}

Stage 2: Deploy

At this stage, openshift.newApp creates a new application named ace-app using the image produced at build stage. Note that this step is executed only in first time execution of pipeline. In next pipeline builds, this stage is skipped if the ace-app already exists in the namespace. But still, a deployment will be automatically done after each build is completed successfully due to the Image Change Trigger capability of Deployment Config. Meaning, after each new image build, an automated deployment will initiate. We will talk about this in the later posts.

stage('deploy') {

when {

expression {

openshift.withCluster() {

return !openshift.selector('dc', 'ace-app').exists()

}

}

}

steps {

script {

openshift.withCluster() {

openshift.withProject() {

**def** created = openshift.newApp("ace-app:latest", "--name=ace-app", "--env=LICENSE=accept")

**echo** "new-app created ${created.count()} objects named: ${created.names()}"

created.narrow('svc').expose("--name=ace-console", "--port=7600")

created.narrow('svc').expose("--name=ace-traffic", "--port=7800")

}

}

}

}

}

### So – LETS TRY!

Start the pipeline with the OpenShift Web Console. Navigate to the Builds → Pipeline section and click **Start Pipeline**.

Alternatively, you can start the pipeline using CLI:

**oc start-build ace-pipeline**

Once the pipeline is started, you should see the following actions performed within your project:

* A job instance is created on the Jenkins server.
* A Node.js pod is launched, as our pipeline requires Node.js as Jenkins agent.
  + “Build” stage of the pipeline launches a build and upon completion a new image is pushed to imagestream.
  + “Deploy” stage of pipeline creates and exposes an application using the image produced in the previous stage. If the application already exists, the steps are skipped.
* Node.js Jenkins agent pod is deleted by default after the pipeline execution completes or is stopped.

You can visualize the pipeline execution by viewing it directly in OpenShift Web Console, choose Builds -> Builds. (or from Jenkins Web Console.)

A screenshot of a cell phone

Description automatically generated

Outcome of a successful pipeline execution is a new deployment configuration and a new pod and related service and the routes for ACE dashboard and integration service. You can access the routes using the OpenShift Web Console, choose Networking -> Routes.

A screenshot of a cell phone

Description automatically generated

Just go ahead and test your API exposed by your containerized ACE application.



**THAT’s it**

You might ask at this point how about deploying to IBM Cloud Pak for Integration. Well, in that case we just need to replace “oc new-app” command with “helm install” command in our deploy script. As we said at the beginning, this is just a warmup that you can try in your local workstation and make yourself familiar with ACE containers. We will post more examples in the future.