Contents

SCENARIO 2: A Developer Introduction to OpenShift **1**
Intro . . . 1
Let’s get started . . . 1
Setup for Exercise . . . 2
Developer Concepts . . . 2
Projects . . . 2
Containers . . . 2
Pods . . . 2
Images . . . 3
Image Streams . . . 3
Builds . . . 3
Pipelines . . . 3
Deployments . . . 3
Services . . . 3
Routes . . . 3
Templates . . . 4
Verifying the Dev Environment . . . 4
Before moving on . . . 6
Live Synchronization of Project Files . . . 6
Live synchronization of project files . . . 7
Before continuing . . . 10
Deploying the Production Environment . . . 10
Prod vs. Dev . . . 10
Create the production environment . . . 10
Promoting Apps Across Environments with Pipelines . . . 11
Pipelines . . . 11
Congratulations! . . . 12
More Reading . . . 14
Adding Pipeline Approval Steps . . . 14
Congratulations! . . . 16
Summary . . . 19

SCENARIO 2: A Developer Introduction to OpenShift

- Purpose: Learn how developing apps is easy and fun
- Difficulty: `intermediate`
- Time: `45-60 minutes`

Intro

In the previous scenario you learned how to take an existing application to the cloud with JBoss EAP and OpenShift, and you got a glimpse into the power of OpenShift for existing applications.

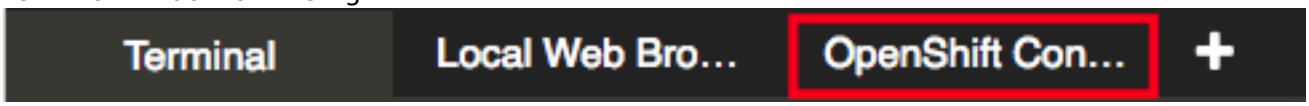
In this scenario you will go deeper into how to use the OpenShift Container Platform as a developer to build and deploy applications. We’ll focus on the core features of OpenShift as it relates to developers, and you’ll learn typical workflows for a developer (develop, build, test, deploy, and repeat).

Let’s get started

If you are not familiar with the OpenShift Container Platform, it’s worth taking a few minutes to understand the basics of the platform as well as the environment that you will be using for this workshop.

The goal of OpenShift is to provide a great experience for both Developers and System Administrators to develop, deploy, and run containerized applications. Developers should love using OpenShift because it enables them to take advantage of both containerized applications and orchestration without having the know the details. Developers are free to focus on their code instead of spending time writing Dockerfiles and running docker builds.

Both Developers and Operators communicate with the OpenShift Platform via one of the following methods:

- **Command Line Interface** - The command line tool that we will be using as part of this training is called the *oc* tool. You used this briefly in the last scenario. This tool is written in the Go programming language and is a single executable that is provided for Windows, OS X, and the Linux Operating Systems.
- **Web Console** - OpenShift also provides a feature rich Web Console that provides a friendly graphical interface for interacting with the platform. You can always access the Web Console using the link provided just above the Terminal window on the right:
- 

- **REST API** - Both the command line tool and the web console actually communicate to OpenShift via the same method, the REST API. Having a robust API allows users to create their own scripts and automation depending on their specific requirements. For detailed information about the REST API, check out the official documentation. You will not use the REST API directly in this workshop.

During this workshop, you will be using both the command line tool and the web console. However, it should be noted that there are plugins for several integrated development environments as well. For example, to use OpenShift from the Eclipse IDE, you would want to use the official JBoss Tools plugin.

Now that you know how to interact with OpenShift, let's focus on some core concepts that you as a developer will need to understand as you are building your applications!

Setup for Exercise

Run the following commands to set up your environment for this scenario and start in the right directory:

```
cd ${HOME}/projects/monolith
git pull --quiet
```

Developer Concepts

There are several concepts in OpenShift useful for developers, and in this workshop you should be familiar with them.

Projects

Projects are a top level concept to help you organize your deployments. An OpenShift project allows a community of users (or a user) to organize and manage their content in isolation from other communities. Each project has its own resources, policies (who can or cannot perform actions), and constraints (quotas and limits on resources, etc). Projects act as a wrapper around all the application services and endpoints you (or your teams) are using for your work.

Containers

The basic units of OpenShift applications are called containers (sometimes called Linux Containers). Linux container technologies are lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources.

Though you do not directly interact with the Docker CLI or service when using OpenShift Container Platform, understanding their capabilities and terminology is important for understanding their role in OpenShift Container Platform and how your applications function inside of containers.

Pods

OpenShift Container Platform leverages the Kubernetes concept of a pod, which is one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.

Pods are the rough equivalent of a machine instance (physical or virtual) to a container. Each pod is allocated its own internal IP address, therefore owning its entire port space, and containers within pods can share their local storage and networking.

Images

Containers in OpenShift are based on Docker-formatted container images. An image is a binary that includes all of the requirements for running a single container, as well as metadata describing its needs and capabilities.

You can think of it as a packaging technology. Containers only have access to resources defined in the image unless you give the container additional access when creating it. By deploying the same image in multiple containers across multiple hosts and load balancing between them, OpenShift Container Platform can provide redundancy and horizontal scaling for a service packaged into an image.

Image Streams

An image stream and its associated tags provide an abstraction for referencing Images from within OpenShift. The image stream and its tags allow you to see what images are available and ensure that you are using the specific image you need even if the image in the repository changes.

Builds

A build is the process of transforming input parameters into a resulting object. Most often, the process is used to transform input parameters or source code into a runnable image. A *BuildConfig* object is the definition of the entire build process. It can build from different sources, including a Dockerfile, a source code repository like Git, or a Jenkins Pipeline definition.

Pipelines

Pipelines allow developers to define a *Jenkins* pipeline for execution by the Jenkins pipeline plugin. The build can be started, monitored, and managed by OpenShift Container Platform in the same way as any other build type.

Pipeline workflows are defined in a Jenkinsfile, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration.

Deployments

An OpenShift Deployment describes how images are deployed to pods, and how the pods are deployed to the underlying container runtime platform. OpenShift deployments also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the replication controller.

Services

A Kubernetes service serves as an internal load balancer. It identifies a set of replicated pods in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address.

Routes

Services provide internal abstraction and load balancing within an OpenShift environment, sometimes clients (users, systems, devices, etc.) **outside** of OpenShift need to access an application. The way that external clients are able to access applications running in OpenShift is through the OpenShift routing layer. And the data object behind that is a *Route*.

The default OpenShift router (HAProxy) uses the HTTP header of the incoming request to determine where to proxy the connection. You can optionally define security, such as TLS, for the *Route*. If you want your *Services*, and, by extension, your *Pods*, to be accessible to the outside world, you need to create a *Route*.

Templates

Templates contain a collection of object definitions (BuildConfigs, DeploymentConfigs, Services, Routes, etc) that compose an entire working project. They are useful for packaging up an entire collection of runtime objects into a somewhat portable representation of a running application, including the configuration of the elements.

You will use several pre-defined templates to initialize different environments for the application. You've already used one in the previous scenario to deploy the application into a *dev* environment, and you'll use more in this scenario to provision the *production* environment as well.

Consult the OpenShift documentation for more details on these and other concepts.

Verifying the Dev Environment

In the previous lab you created a new OpenShift project called `coolstore-dev` which represents your developer personal project in which you deployed the CoolStore monolith.

1. Verify Application

Let's take a moment and review the OpenShift resources that are created for the Monolith:

- Build Config: **coolstore** build config is the configuration for building the Monolith image from the source code or WAR file
- Image Stream: **coolstore** image stream is the virtual view of all coolstore container images built and pushed to the OpenShift integrated registry.
- Deployment Config: **coolstore** deployment config deploys and redeploys the Coolstore container image whenever a new coolstore container image becomes available. Similarly, the **coolstore-postgresql** does the same for the database.
- Service: **coolstore** and **coolstore-postgresql** service is an internal load balancer which identifies a set of pods (containers) in order to proxy the connections it receives to them. Backing pods can be added to or removed from a service arbitrarily while the service remains consistently available, enabling anything that depends on the service to refer to it at a consistent address (service name or IP).
- Route: **www** route registers the service on the built-in external load-balancer and assigns a public DNS name to it so that it can be reached from outside OpenShift cluster.

You can review the above resources in the OpenShift Web Console or using the `oc get` or `oc describe` commands (`oc describe` gives more detailed info):

> You can use short synonyms for long words, like `bc` instead of `buildconfig`, and `is` for `imagestream`, `dc` for `deploymentconfig`, `svc` for service, etc.
>
> **NOTE**: Don't worry about reading and understanding the output of `oc describe`. Just make sure the command doesn't report errors!

Run these commands to inspect the elements:

```
oc get bc coolstore
```

```
oc get is coolstore
```

```
oc get dc coolstore
```

```
oc get svc coolstore
```

```
oc describe route www
```

Verify that you can access the monolith by clicking on the exposed OpenShift route at

`http://www-coolstore-dev.$ROUTE_SUFFIX` to open up the sample application in a separate browser tab.

You should also be able to see both the CoolStore monolith and its database running in separate pods:

```
oc get pods -l application=coolstore
```

The output should look like this:

```
NAME                            READY     STATUS    RESTARTS   AGE
coolstore-2-bpkkc               1/1       Running   0          4m
coolstore-postgresql-1-jpcb8    1/1       Running   0          9m
```

1. Verify Database

You can log into the running Postgres container using the following:

```
oc  rsh dc/coolstore-postgresql
```

Once logged in, use the following command to execute an SQL statement to show some content from the database:

```
psql -U $POSTGRESQL_USER $POSTGRESQL_DATABASE -c 'select name from PRODUCT_CATALOG;'
```

You should see the following:

```
          name
------------------------
 Red Fedora
 Forge Laptop Sticker
 Solid Performance Polo
 Ogio Caliber Polo
 16 oz. Vortex Tumbler
 Atari 2600 Joystick
 Pebble Smart Watch
 Oculus Rift
 Lytro Camera
(9 rows)```

Don't forget to exit the pod's shell with `exit`

With our running project on OpenShift, in the next step we'll explore how you as a developer can wor
to make changes and debug the application!



## Making Changes to Containers

In this step you will learn how to transfer files between your local machine and a running
container.

One of the properties of container images is that they are **immutable**. That is, although you can
make changes to the local container filesystem of a running image, the changes are _not permanent_.
When that container is stopped, _any changes are discarded_. When a new container is started from
the same container image, it _reverts back_ to what was originally built into the image.

Although any changes to the local container filesystem are discarded when the container is stopped,
it can sometimes be convenient to be able to upload files into a running container. One example of
where this might be done is during development and a dynamic scripting language like javascript or
static content files like html is being used. By being able to modify code in the container, you car
changes before rebuilding the image.

In addition to uploading files into a running container, you might also want to be able to download
files. During development these may be data files or log files created by the application.

## Copy files from container

As you recall from the last step, we can use `oc rsh` to execute commands inside the running pod.

For our Coolstore Monolith running with JBoss EAP, the application is installed in the `/opt/eap` di
container. Execute the `ls` command inside the container to see this:

`oc  rsh dc/coolstore ls -l /opt/eap`

You should see a listing of files in this directory **in the running container**.

> It is very important to remember where commands are executed! If you think you are in a container
destructive commands may do real harm, so be careful! In general it is not a good idea to operate in
development environment. But for doing testing and debugging it's OK.

Let's copy some files out of the running container. To copy files from a running container
```

on OpenShift, we'll use the `oc rsync` command. This command expects the name of the pod to copy fro
which can be seen with this command:

`oc get pods --selector deploymentconfig=coolstore`

The output should show you the name of the pod:

NAME READY STATUS RESTARTS AGE coolstore-2-bpkkc 1/1 Running 0 32m

The name of my running coolstore monolith pod is `coolstore-2-bpkkc` but **yours will be different**

Save the name of the pod into an environment variable called `COOLSTORE_DEV_POD_NAME` so that we can
commands:

`export COOLSTORE_DEV_POD_NAME=$(oc get pods --selector deploymentconfig=coolstore -o jsonpath='{.it

Verify the variable holds the name of your pod with:

`echo $COOLSTORE_DEV_POD_NAME`

Next, run the `oc rsync` command in your terminal window, using the new variable to refer to the nam

`oc  rsync $COOLSTORE_DEV_POD_NAME:/opt/eap/version.txt .`

The output will show that the file was downloaded:

receiving incremental file list version.txt

sent 30 bytes received 65 bytes 62,566.00 bytes/sec total size is 65 speedup is 1.00 “‘

Now you can open the file locally using this link: `version.txt` and inspect its contents.

This is useful for verifying that the contents of files in your applications are what you expect.

You can also upload files using the same `oc rsync` command but unlike when copying from the container to the local machine, there is no form for copying a single file. To copy selected files only, you will need to use the `--exclude` and `--include` options to filter what is and isn’t copied from a specified directory. We will use this in the next step.

Manually copying is cool, but what about automatic live copying on change? That’s in the next step too!

Before moving on

Let’s clean up the temp files we used. Execute:

```
rm -f version.txt hello.txt
```

Live Synchronization of Project Files

In addition to being able to manually upload or download files when you choose to, the `oc rsync` command can also be set up to perform live synchronization of files between your local computer and the container. When there is a change to a file, the changed file will be automatically copied up to the container.

This same process can also be run in the opposite direction if required, with changes made in the container being automatically copied back to your local computer.

An example of where it can be useful to have changes automatically copied from your local computer into the container is during the development of an application.

For scripted programming languages such as JavaScript, PHP, Python or Ruby, where no separate compilation phase is required you can perform live code development with your application running inside of OpenShift.

For JBoss EAP applications you can sync individual files (such as HTML/CSS/JS files), or sync entire application .WAR files. It’s more challenging to synchronize individual files as it requires that you use an *exploded* archive deployment, so the use of JBoss Developer Studio is recommended, which automates this process (see these docs for more info).

Live synchronization of project files

For this workshop, we'll Live synchronize the entire WAR file.

First, click on the Coolstore application link at

`http://www-coolstore-dev.$ROUTE_SUFFIX` to open the application in a browser tab so you can watch changes.

1. Turn on Live Sync

Turn on **Live sync** by executing this command:

```
oc  rsync deployments/ $COOLSTORE_DEV_POD_NAME:/deployments --watch --no-perms &
```

The & character at the end places the command into the background. We will kill it at the end of this step.

Now `oc` is watching the `deployments/` directory for changes to the `ROOT.war` file. Anytime that file changes, `oc` will copy it into the running container and we should see the changes immediately (or after a few seconds). This is much faster than waiting for a full re-build and re-deploy of the container image.

2. Make a change to the UI

Next, let's make a change to the app that will be obvious in the UI.

First, open `src/main/webapp/app/css/coolstore.css`, which contains the CSS stylesheet for the CoolStore app.

Add the following CSS to turn the header bar background to Red Hat red (click **Copy To Editor** to add it at the bottom):

```
.navbar-header {
    background: #CC0000
}
```

2. Rebuild application For RED background

Let's re-build the application using this command:

```
mvn package -Popenshift
```

This will update the ROOT.war file and cause the application to change.

Re-visit the app by reloading the Coolstore webpage (or clicking again on the Coolstore application link at

`http://www-coolstore-dev.$ROUTE_SUFFIX`).

You should now see the red header:

> **NOTE** If you don't see the red header, you may need to do a full reload of the webpage. On Windows/Linux press `CTRL+F5` or hold down `SHIFT` and press the Reload button, or try `CTRL+SHIFT+F5`. On Mac OS X, press `SHIFT+CMD+R`, or hold `SHIFT` while pressing the Reload button.

3. Rebuild again for BLUE background

Repeat the process, but replace the background color to be blue (click **Copy to Editor** to replace `#CC0000` with `blue`):

```
background: blue
```

Again, re-build the app:

```
mvn package -Popenshift
```

This will update the ROOT.war file again and cause the application to change.

Re-visit the app by reloading the Coolstore webpage (or clicking again on the Coolstore application link at

`http://www-coolstore-dev.$ROUTE_SUFFIX`).

It's blue! You can do this as many times as you wish, which is great for speedy development and testing.

We'll leave the blue header for the moment, but will change it back to the original color soon.

Because we used `oc rsync` to quickly re-deploy changes to the running pod, the changes will be lost if we restart the pod. Let's update the container image to contain our new blue header. Execute:

```
oc start-build coolstore --from-file=deployments/ROOT.war
```

And again, wait for it to complete by executing:

```
oc rollout status -w dc/coolstore
```

redhat

Red Hat Cool Store | Your Shopping Cart | All Orders

Red Fedora

Official Red Hat Fedora

$24.00

Forge Laptop Sticker

JBoss Community Forge Project Sticker

Figure 1: Red

redhat

Red Hat Cool Store | Your Shopping Cart | All Orders

Red Fedora

Official Red Hat Fedora

$34.99

1 Add To Cart

736 left!

Forge Laptop Sticker

JBoss Community Forge Project Sticker

$8.50

Figure 2: Blue

Before continuing

Kill the `oc rsync` processes we started earlier in the background. Execute:

```
kill %1
```

On to the next challenge!

Deploying the Production Environment

In the previous scenarios, you deployed the Coolstore monolith using an OpenShift Template into the `coolstore-dev` Project. The template created the necessary objects (BuildConfig, DeploymentConfig, ImageStreams, Services, and Routes) and gave you as a Developer a "playground" in which to run the app, make changes and debug.

In this step we are now going to setup a separate production environment and explore some best practices and techniques for developers and DevOps teams for getting code from the developer (that's YOU!) to production with less downtime and greater consistency.

Prod vs. Dev

The existing `coolstore-dev` project is used as a developer environment for building new versions of the app after code changes and deploying them to the development environment.

In a real project on OpenShift, *dev*, *test* and *production* environments would typically use different OpenShift projects and perhaps even different OpenShift clusters.

For simplicity in this scenario we will only use a *dev* and *prod* environment, and no test/QA environment.

Create the production environment

We will create and initialize the new production environment using another template in a separate OpenShift project.

1. Initialize production project environment

Execute the following `oc` command to create a new project:

```
oc new-project coolstore-prod --display-name='Coolstore Monolith - Production'
```

This will create a new OpenShift project called `coolstore-prod` from which our production application will run.

2. Add the production elements

In this case we'll use the production template to create the objects. Execute:

```
oc new-app --template=coolstore-monolith-pipeline-build
```

This will use an OpenShift Template called `coolstore-monolith-pipeline-build` to construct the production application. As you probably guessed it will also include a Jenkins Pipeline to control the production application (more on this later!)

Navigate to the Web Console to see your new app and the components using this link:

- Coolstore Prod Project Overview at

```
https://$OPENSHIFT_MASTER/console/project/coolstore-prod/overview
```

You can see the production database, and an application called *Jenkins* which OpenShift uses to manage CI/CD pipeline deployments. There is no running production app just yet. The only running app is back in the *dev* environment, where you used a binary build to run the app previously.

In the next step, we'll *promote* the app from the *dev* environment to the *production* environment using an OpenShift pipeline build. Let's get going!

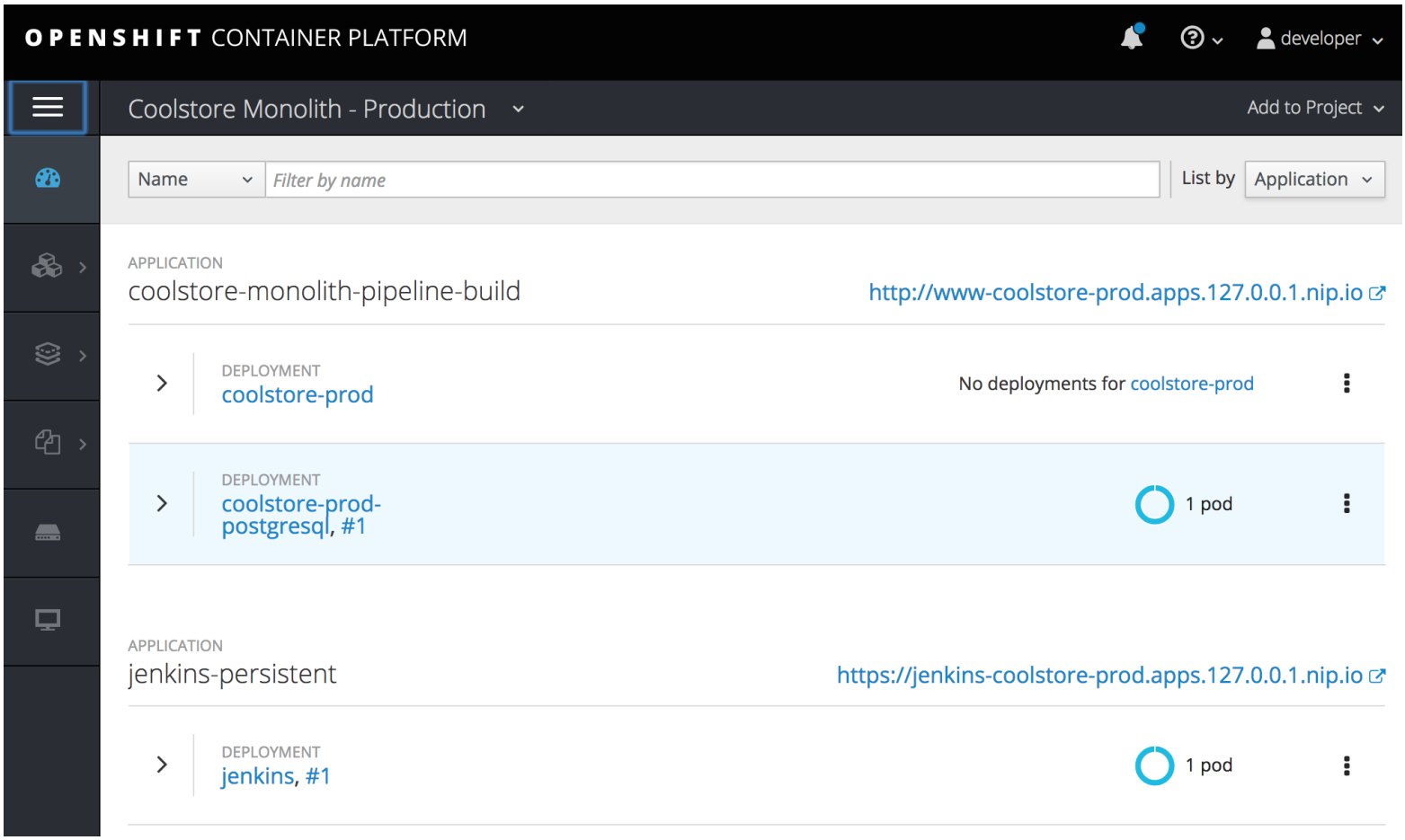


Figure 3: Prod

Promoting Apps Across Environments with Pipelines

Continuous Delivery

So far you have built and deployed the app manually to OpenShift in the *dev* environment. Although it's convenient for local development, it's an error-prone way of delivering software when extended to test and production environments.

Continuous Delivery (CD) refers to a set of practices with the intention of automating various aspects of delivery software. One of these practices is called delivery pipeline which is an automated process to define the steps a change in code or configuration has to go through in order to reach upper environments and eventually to production.

OpenShift simplifies building CI/CD Pipelines by integrating the popular Jenkins pipelines into the platform and enables defining truly complex workflows directly from within OpenShift.

The first step for any deployment pipeline is to store all code and configurations in a source code repository. In this workshop, the source code and configurations are stored in a GitHub repository we've been using at [https://github.com/RedHat-Middleware-Workshops/modernize-apps-labs]. This repository has been copied locally to your environment and you've been using it ever since!

You can see the changes you've personally made using `git --no-pager status` to show the code changes you've made using the Git command (part of the Git source code management system).

Pipelines

OpenShift has built-in support for CI/CD pipelines by allowing developers to define a Jenkins pipeline for execution by a Jenkins automation engine, which is automatically provisioned on-demand by OpenShift when needed.

The build can get started, monitored, and managed by OpenShift in the same way as any other build types e.g. S2I. Pipeline workflows are defined in a Jenkinsfile, either embedded directly in the build configuration, or supplied in a Git repository and referenced by the build configuration. They are written using the Groovy scripting language.

As part of the production environment template you used in the last step, a Pipeline build object was created. Ordinarily the pipeline would contain steps to build the project in the *dev* environment, store the resulting image in the local

repository, run the image and execute tests against it, then wait for human approval to *promote* the resulting image to other environments like test or production.

1. Inspect the Pipeline Definition

Our pipeline is somewhat simplified for the purposes of this Workshop. Inspect the contents of the pipeline using the following command:

```
oc describe bc/monolith-pipeline
```

You can see the Jenkinsfile definition of the pipeline in the output:

```
Jenkinsfile contents:
  node ('maven') {
    stage 'Build'
    sleep 5

    stage 'Run Tests in DEV'
    sleep 10

    stage 'Deploy to PROD'
    openshiftTag(sourceStream: 'coolstore', sourceTag: 'latest', namespace: 'coolstore-dev', destina
    sleep 10

    stage 'Run Tests in PROD'
    sleep 30
  }
```

Pipeline syntax allows creating complex deployment scenarios with the possibility of defining checkpoint for manual interaction and approval process using the large set of steps and plugins that Jenkins provides in order to adapt the pipeline to the process used in your team. You can see a few examples of advanced pipelines in the OpenShift GitHub Repository.

To simplify the pipeline in this workshop, we simulate the build and tests and skip any need for human input. Once the pipeline completes, it deploys the app from the *dev* environment to our *production* environment using the above `openshiftTag()` method, which simply re-tags the image you already created using a tag which will trigger deployment in the production environment.

2. Promote the dev image to production using the pipeline

You can use the *oc* command line to invoke the build pipeline, or the Web Console. Let's use the Web Console. Open the production project in the web console:

- Web Console - Coolstore Monolith Prod at

`https://$OPENSHIFT_MASTER/console/project/coolstore-prod`

Next, navigate to *Builds -> Pipelines* and click **Start Pipeline** next to the `coolstore-monolith` pipeline:

This will start the pipeline. **It will take a minute or two to start the pipeline** (future runs will not take as much time as the Jenkins infrastructure will already be warmed up). You can watch the progress of the pipeline:

Once the pipeline completes, return to the Prod Project Overview at

`https://$OPENSHIFT_MASTER/console/project/coolstore-prod` and notice that the application is now deployed and running!

View the production app **with the blue header from before** is running by clicking: CoolStore Production App at

`http://www-coolstore-prod.$ROUTE_SUFFIX` (it may take a few moments for the container to deploy fully.)

Congratulations!

You have successfully setup a development and production environment for your project and can use this workflow for future projects as well.

In the final step we'll add a human interaction element to the pipeline, so that you as a project lead can be in charge of approving changes.

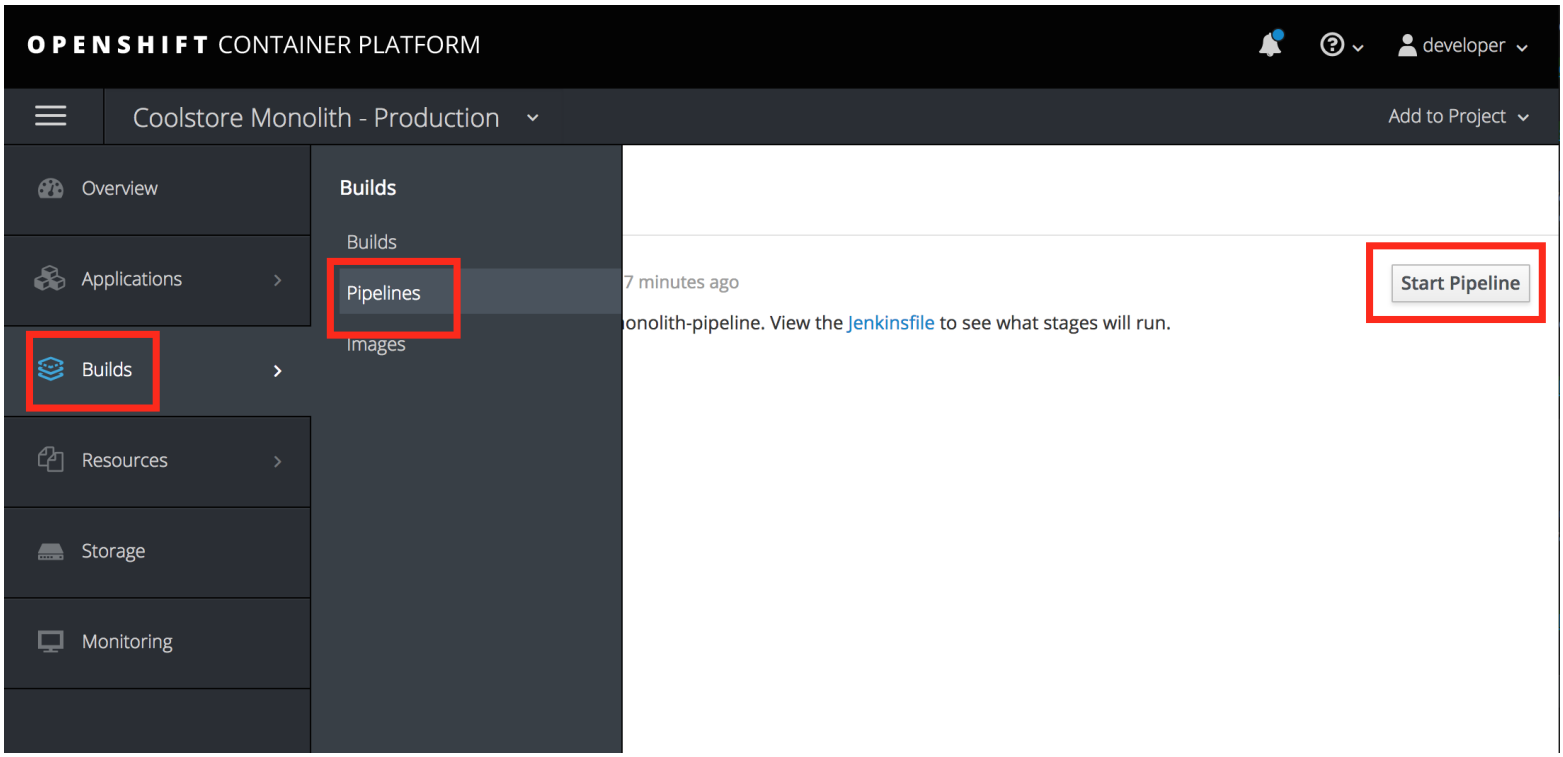


Figure 4: Prod

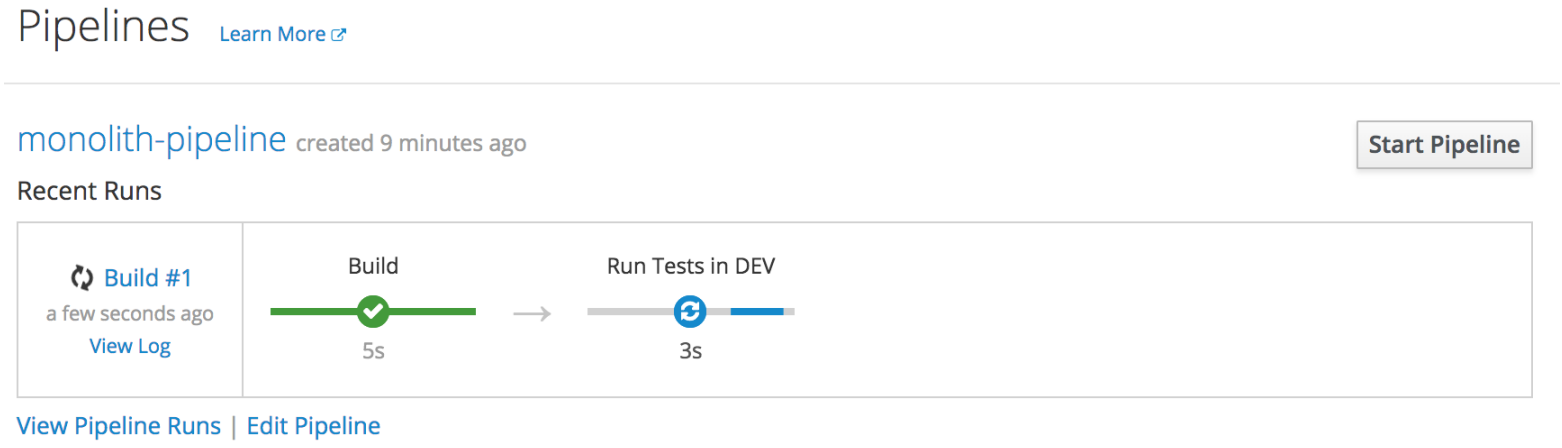


Figure 5: Prod

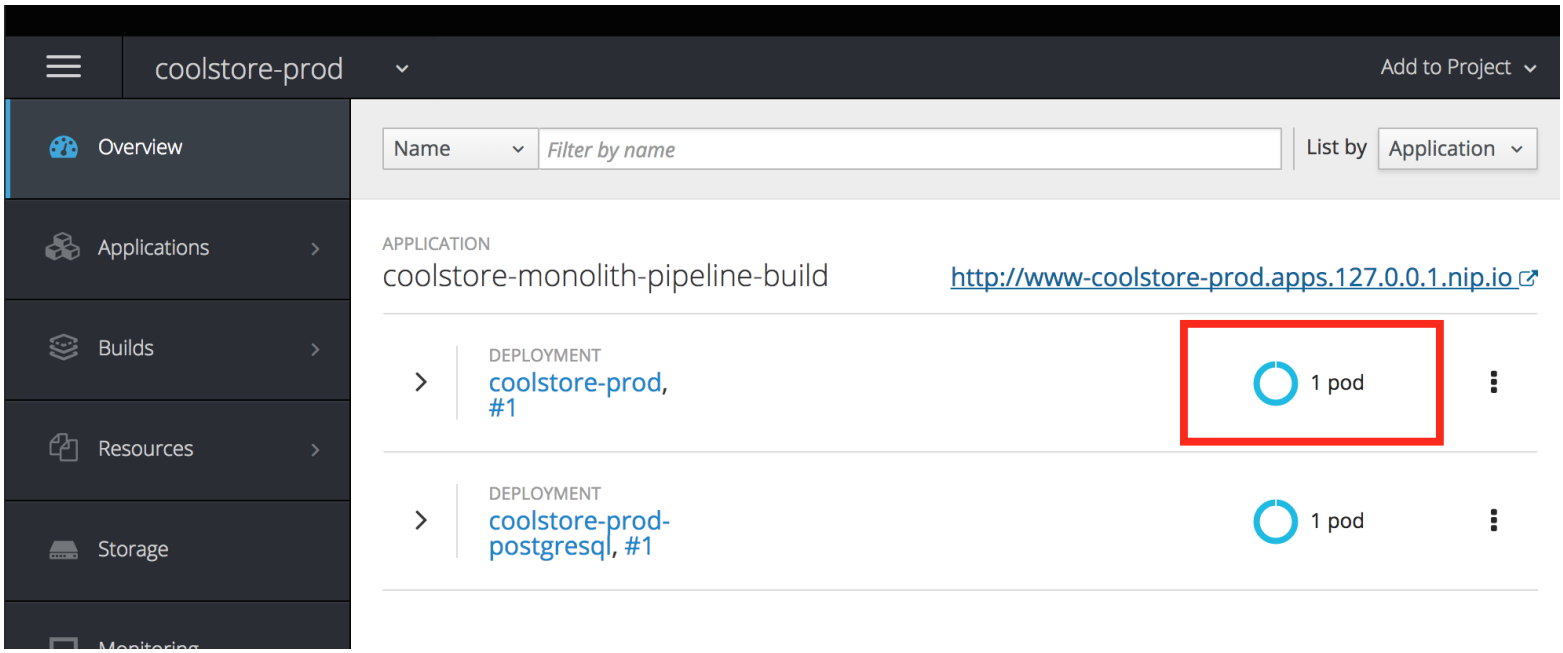


Figure 6: Prod

More Reading

- OpenShift Pipeline Documentation

Adding Pipeline Approval Steps

In previous steps you used an OpenShift Pipeline to automate the process of building and deploying changes from the dev environment to production.

In this step, we'll add a final checkpoint to the pipeline which will require you as the project lead to approve the final push to production.

1. Edit the pipeline

Ordinarily your pipeline definition would be checked into a source code management system like Git, and to change the pipeline you'd edit the *Jenkinsfile* in the source base. For this workshop we'll just edit it directly to add the necessary changes. You can edit it with the oc command but we'll use the Web Console.

Open the `monolith-pipeline` configuration page in the Web Console (you can navigate to it from *Builds -> Pipelines* but here's a quick link):

- Pipeline Config page at

`https://$OPENSHIFT_MASTER/console/project/coolstore-prod/browse/pipelines/monolith-pipeline?tab=conf`

On this page you can see the pipeline definition. Click *Actions -> Edit* to edit the pipeline:

In the pipeline definition editor, add a new stage to the pipeline, just before the `Deploy to PROD` step:

NOTE: You will need to copy and paste the below code into the right place as shown in the below image.

```
stage 'Approve Go Live'
timeout(time:30, unit:'MINUTES') {
  input message:'Go Live in Production (switch to new version)?'
}
```

Your final pipeline should look like:

Click **Save**.

2. Make a simple change to the app

With the approval step in place, let's simulate a new change from a developer who wants to change the color of the header in the coolstore back to the original (black) color.

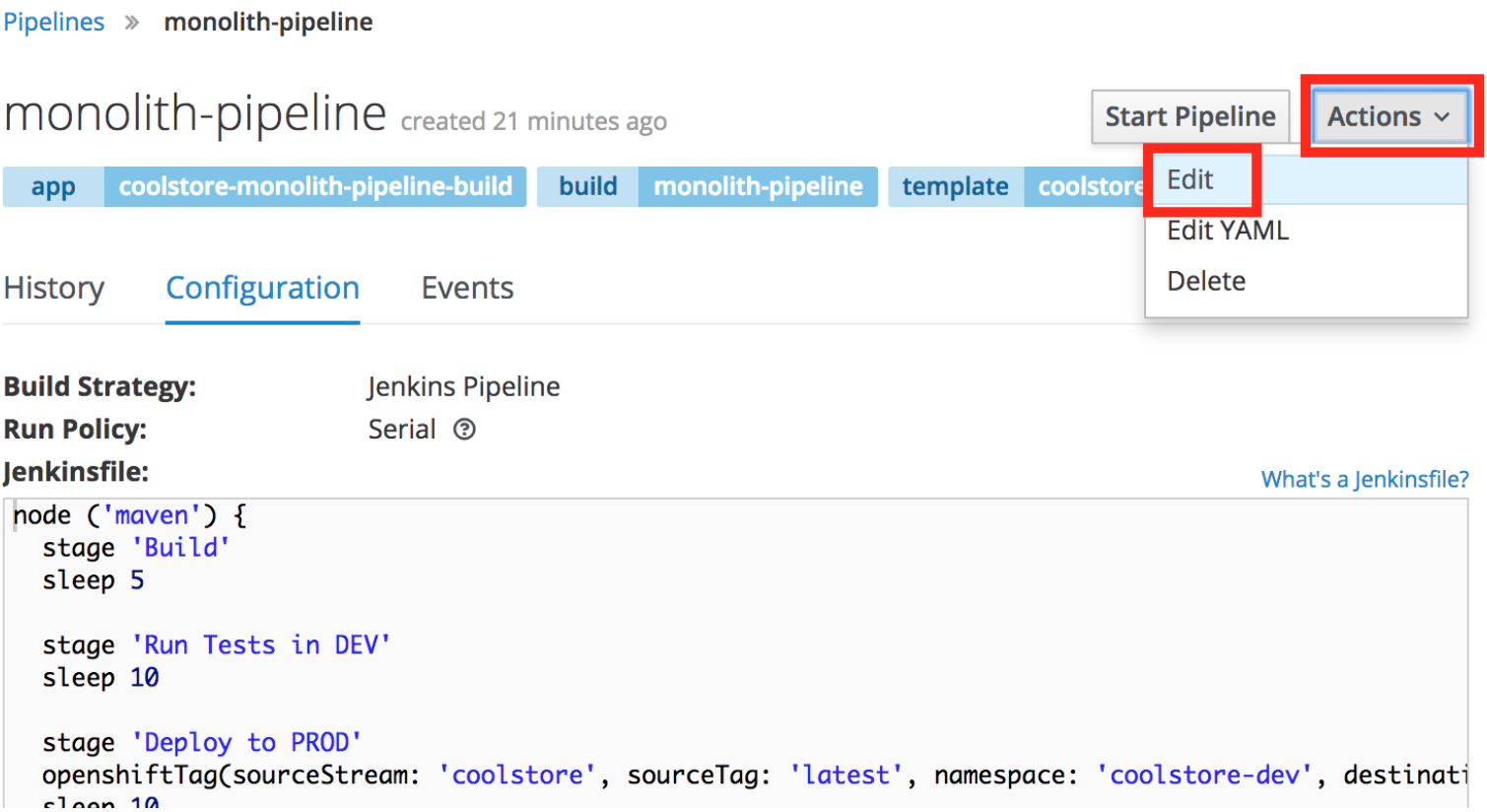


Figure 7: Prod

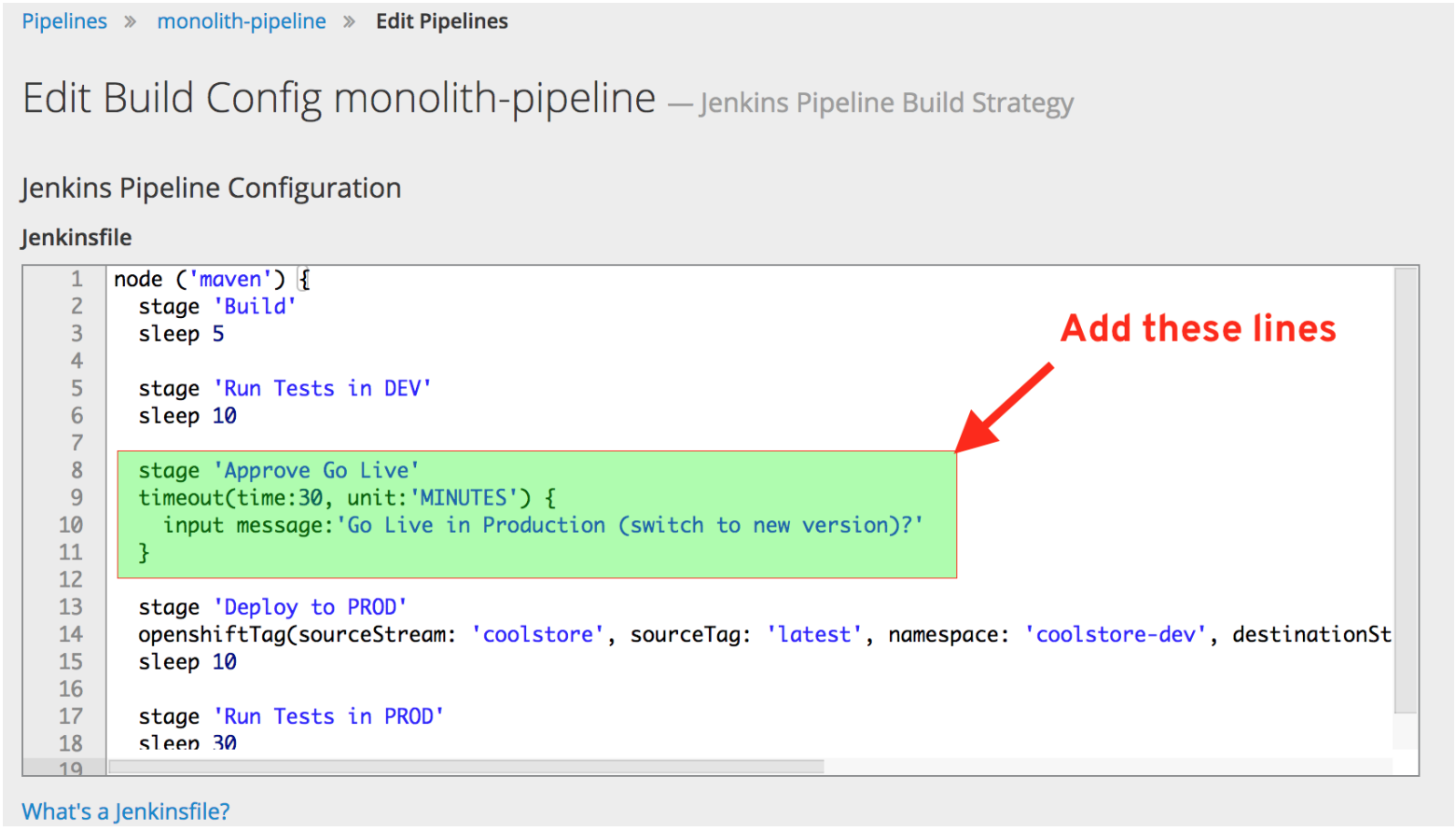


Figure 8: Prod

As a developer you can easily un-do edits you made earlier to the CSS file using the source control management system (Git). To revert your changes, execute:

```
git checkout src/main/webapp/app/css/coolstore.css
```

Next, re-build the app once more:

```
mvn clean package -Popenshift
```

And re-deploy it to the dev environment using a binary build just as we did before:

```
oc start-build -n coolstore-dev coolstore --from-file=deployments/ROOT.war
```

Now wait for it to complete the deployment:

```
oc -n coolstore-dev rollout status -w dc/coolstore
```

And verify that the original black header is visible in the dev application:

- Coolstore - Dev at

```
http://www-coolstore-dev.$ROUTE_SUFFIX
```

While the production application is still blue:

- Coolstore - Prod at

```
http://www-coolstore-prod.$ROUTE_SUFFIX
```

We're happy with this change in dev, so let's promote the new change to prod, using the new approval step!

3. Run the pipeline again

Invoke the pipeline once more by clicking **Start Pipeline** on the Pipeline Config page at

```
https://$OPENSHIFT_MASTER/console/project/coolstore-prod/browse/pipelines/monolith-pipeline
```

The same pipeline progress will be shown, however before deploying to prod, you will see a prompt in the pipeline:

Click on the link for `Input Required`. This will open a new tab and direct you to Jenkins itself, where you can login with the same credentials as OpenShift:

- Username: `developer`
- Password: `developer`

Accept the browser certificate warning and the Jenkins/OpenShift permissions, and then you'll find yourself at the approval prompt:

3. Approve the change to go live

Click **Proceed**, which will approve the change to be pushed to production. You could also have clicked **Abort** which would stop the pipeline immediately in case the change was unwanted or unapproved.

Once you click **Proceed**, you will see the log file from Jenkins showing the final progress and deployment.

Wait for the production deployment to complete:

```
oc rollout -n coolstore-prod status dc/coolstore-prod
```

Once it completes, verify that the production application has the new change (original black header):

- Coolstore - Prod at

```
http://www-coolstore-prod.$ROUTE_SUFFIX
```

Congratulations!

You have added a human approval step for all future developer changes. You now have two projects that can be visualized as:

redhat.

Red Hat Cool Store | Your Shopping Cart

Red Fedora

Official Red Hat Fedora

$34.99

1 Add To Cart

736 left!

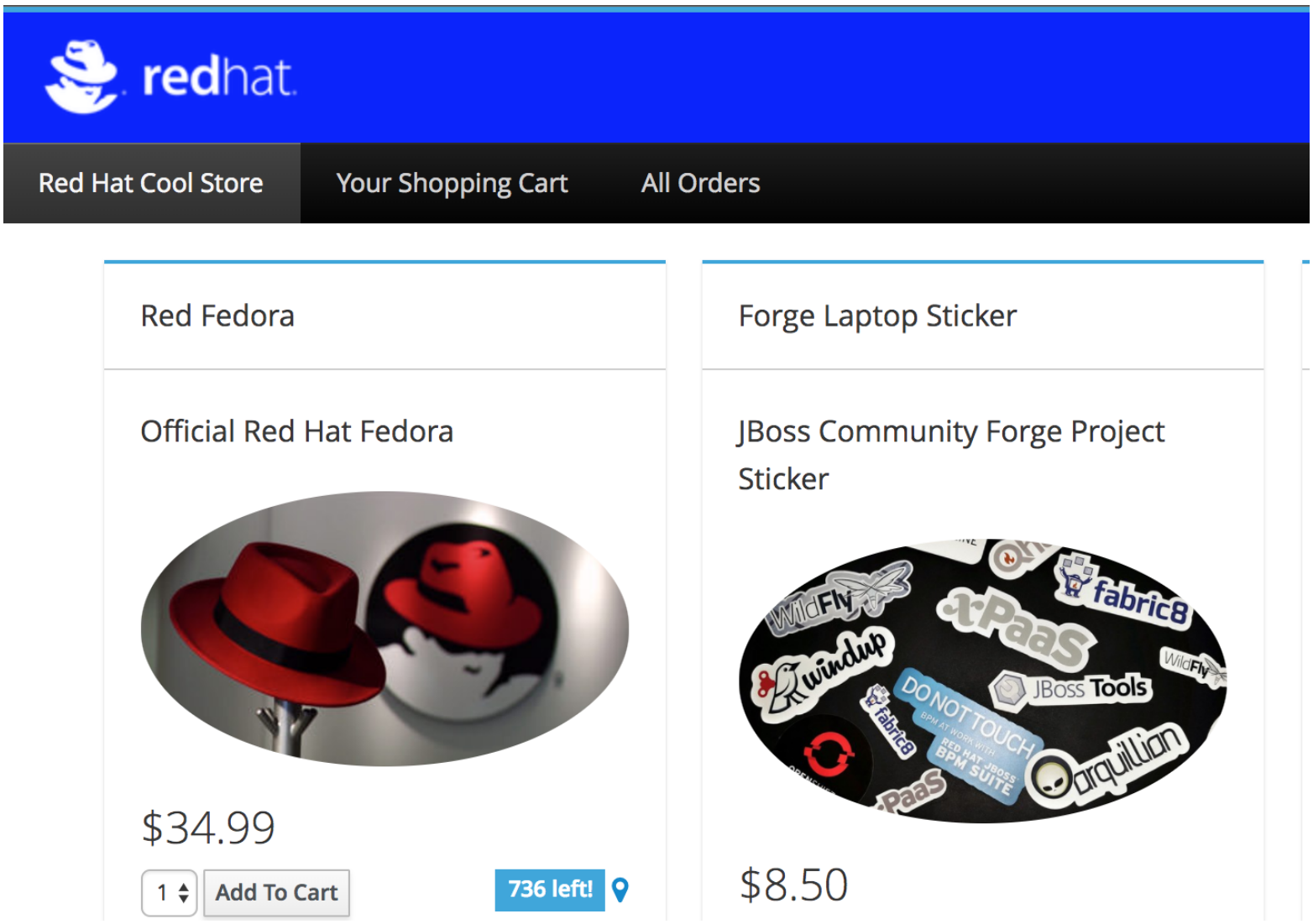


Figure 10: Prod

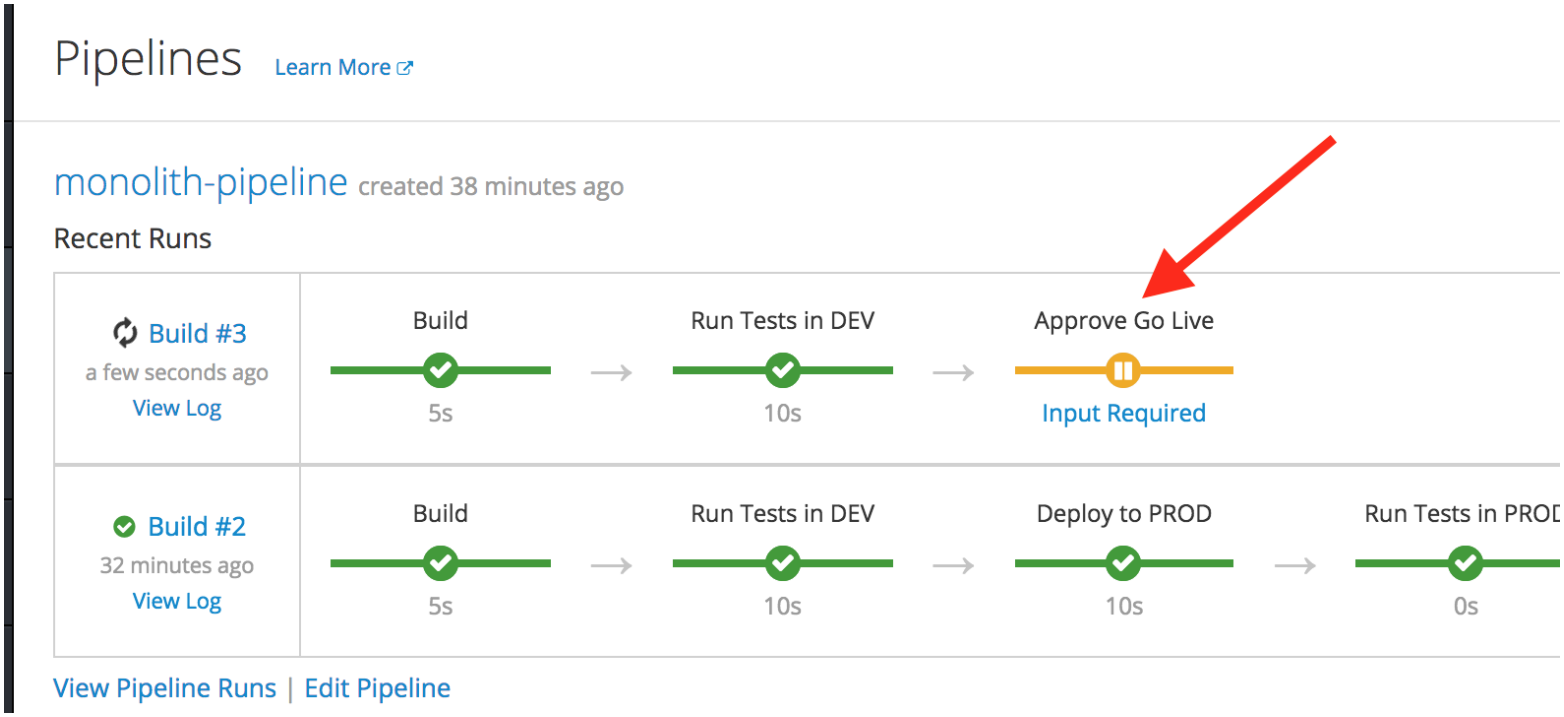


Figure 11: Prod

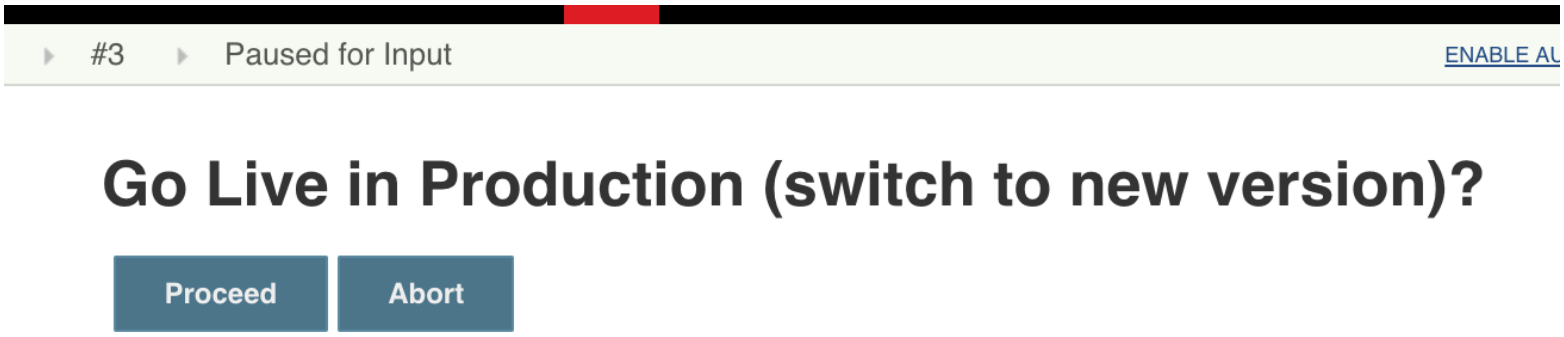


Figure 12: Prod

Summary

In this scenario you learned how to use the OpenShift Container Platform as a developer to build, and deploy applications. You also learned how OpenShift makes your life easier as a developer, architect, and DevOps engineer.

You can use these techniques in future projects to modernize your existing applications and add a lot of functionality without major re-writes.

The monolithic application we've been using so far works great, but is starting to show its age. Even small changes to one part of the app require many teams to be involved in the push to production.

In the next few scenarios we'll start to modernize our application and begin to move away from monolithic architectures and toward microservice-style architectures using Red Hat technology. Let's go!

redhat.

Red Hat Cool Store | Your Shopping Cart

Red Fedora

Official Red Hat Fedora

$34.99

1 Add To Cart

736 left!

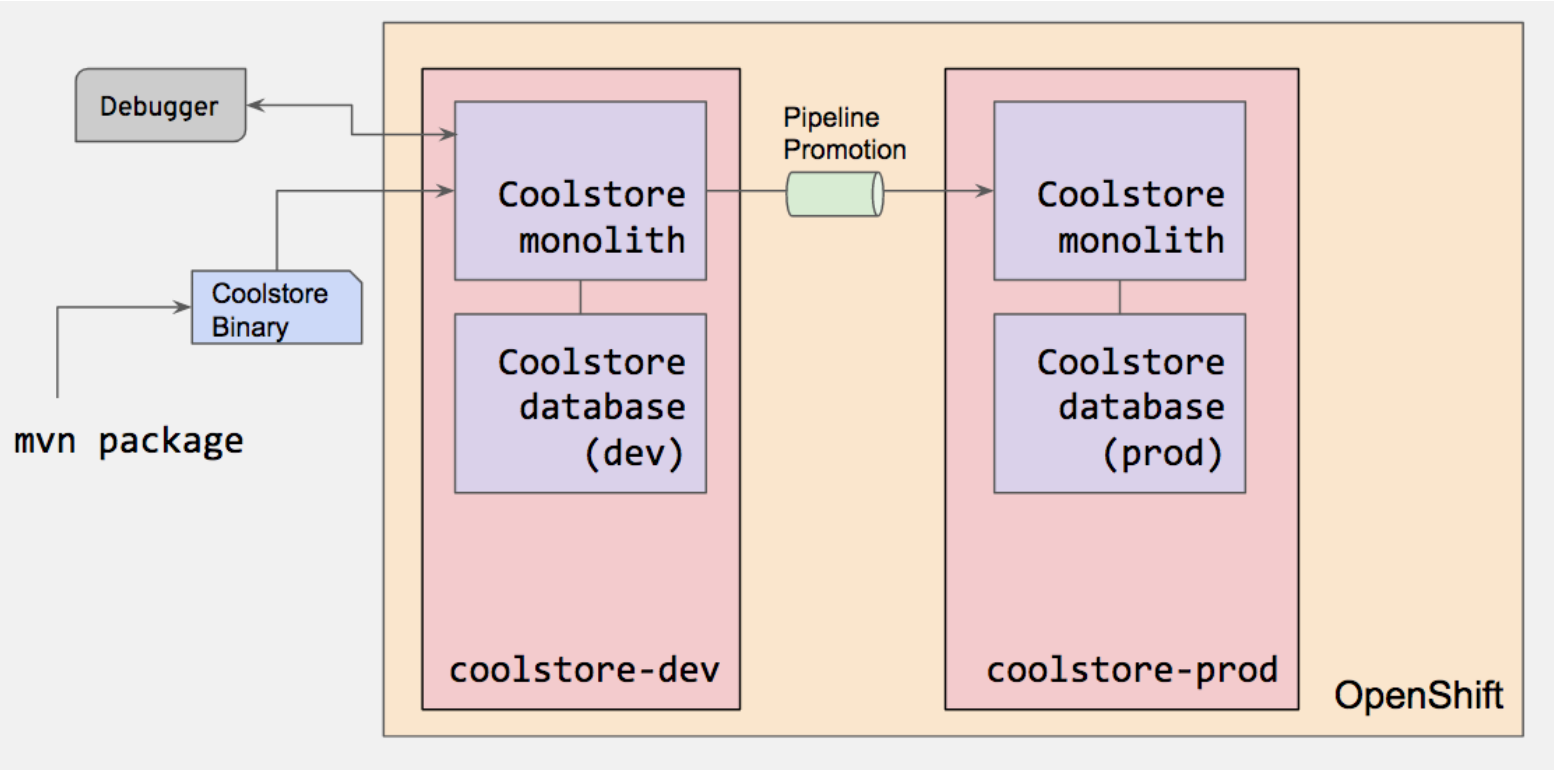


Figure 14: Prod