CHAPTER 6

IMPLEMENTING CONTINUOUS INTEGRATION

Overview	
Goal	Establish continuous integration with Jenkins.
Objectives	• List the benefits of implementing CI. • List the tools and techniques needed to implement CI. • Configure Jenkins to build the application triggered by a Git commit. • List the tools available to implement the techniques and processes supported by CI. • Integrate the Jenkins build process into the S2I mode.
Sections	• Introduction to Continuous Integration (CI) (and Guided Exercise) • Implementing CI for the Bookstore Application (and Guided Exercise) • Integrating Continuous Integration (CI) into OpenShift Enterprise (and Guided Exercise)
Lab	• Implementing CI for the Node.js application

Introduction to Continuous Integration (CI)

Objectives

After completing this section, students should be able to:

- List the benefits of implementing CI.
- List the tools and techniques needed to implement CI.

CI as a DevOps practice

Continuous integration is a software development practice that:

1. *Verifies build integrity*: Check if the source code can be pulled from the repository and it can be built for deploying purposes. The build process may include compiling, packaging, and configuring the software.
2. *Validates the test results*: Run all the tests created by developers in a production-like environment to verify if the source code was not broken due to a side-effect from the commit.
3. *Checks the integration among multiple systems*: Using integration tests, all systems integration used by the software will be validated.
4. *Identifies any problem*: Alert all affected teams about the problem to fix it promptly.

This practice can be executed manually, but due to the lack of traceability for manual operations, the CI process is usually managed by a tool. In a DevOps culture, CI is mandatory and executed by a tool that runs automation scripts created by developers to eliminate all human intervention during the CI process.

Note

There are many CI tools available for usage: **`https://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software`**.

CI can be successfully implemented if some practices are addressed by the development team:

- *Test-driven development (TDD)*: It is a software development paradigm in which a developer must create tests before coding to guarantee the source code logic is correct.
- *Constant check in*: Check in to a source code management (SCM) tool both test and application source code constantly.
- *Request constant feedback*: Use the test and build tool feedbacks to commit new code or fix the existing source code.

Usually the problematic feedbacks should be addressed by the developer that broke the test or the build process.

TDD

Test-driven development is an agile development practice adopted by many software development processes in order to guarantee software consistency and integrity. Many frameworks and tools were developed to support TDD in Java and most of them uses JUnit as the base framework.

Note

All the languages supported by OpenShift Enterprise (OSE) have a testing framework to support TDD practice. A comprehensive list of them can be obtained from: **`https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks`**.

Fortunately, most unit test frameworks can be customized to support advanced use cases such as integration tests or acceptance tests. JUnit provides a large number of extensions that can be used for multiple test conditions.

Red Hat JBoss has implemented an in-container test framework called Arquillian that supports most of the JavaEE containers and implements a simple, but powerful, dependency management mechanism.

Note

Arquillian web site:**`http://arquillian.org/`**.

Arquillian provides APIs to:

- *Identify library dependencies*: Identifies all the external libraries needed in order to deploy the application working with **`pom.xml`** as a dependency guideline and Tattletale APIs.
- *Package the contents*: Packages all the artifacts needed by the Java EE application for deployment process using Shrinkwrap APIs.
- *Deploy the contents*: Connects to a running Java EE container and deploy the artifacts using configuration files.
- *Test the application*: Runs JUnit tests configured with Arquillian annotations to access the application for testing purposes.

Due to the pluggable nature of Arquillian, some extensions were created to simplify the creation of additional tests, focused on different aspects of a software such as acceptance and sanity testing.

In order to create acceptance tests, Arquillian Drone trigger the testing environment and deploy the application. In order to execute the test, Drone uses Selenium testing tool APIs to mimic the usage of an actual person accessing a web browser.

Jenkins

Jenkins is a continuous integration tool developed in Java by Kohsuke Kawaguchi. It provides:

- *Web application*: to monitor the build process of each application. Due to its flexible nature, it also can be used to get information using REST.

- *Builder engine*: to checkout source codes from a remote source code management (SCM) tool and trigger the build process using Maven, Ant, or any existing Java build tool.
- *Persistence storage*: to store all the builds executed by Jenkins and allow a simpler release management policy.
- *Extensible by plugins*: to expand Jenkins capabilities; for example introducing support to other build tools, implementing reports, and managing software metrics.
- *A scheduler*: to execute on demand build requests.

Installation

Jenkins is provided in two formats:

1. *Web application:* It is provided as a Java web application that can be deployed as it is to any Java EE container.
2. *Docker container:* Jenkins is also provided as a Docker container and it can also be integrated with OpenShift as well.

Note

Jenkins web site [https://www.jenkins-ci.org]

Installed as a web application

Jenkins can be deployed as a web application in any Java EE container. In a JBoss EAP instance, the war file can be deployed at **`JBOSS_HOME/standalone/deployments`** directory and it can be accessed via a web browser using the address **`http://localhost:8080/jenkins`**.

Note

For other Java EE container deployment procedures, please check **`https://wiki.jenkins-ci.org/display/JENKINS/Containers`**.

In addition to the web application deployment, Jenkins can also be started as a Java application. It can be started using the following command:

```
java -jar jenkins.war
```

Under this circumstance, Jenkins can be accessed via a web browser using the address: **`http://localhost:8080`**.

Note

For further parameter that can be used by Jenkins as a Java application, please refer to: **`https://wiki.jenkins-ci.org/display/JENKINS/Starting+and+Accessing+Jenkins`**.

Installed as a Docker container

Jenkins also provides a container image available for download from the public registry. Since this course requires Jenkins with OpenShift support, a custom image was developed by the OpenShift team. In order to pull it from the public registry, run:

```
# docker pull openshift/jenkins-16-centos7
```

Note

Due to the restricted environment for this classroom, the container images are stored at a private docker registry at **`workstation.podX.example.com:5000`**.

To start Jenkins as a container image, run the following command:

```
# docker run -e JENKINS_PASSWORD=<adminPassword> jenkins-16-centos7
```

Note

All contents from the build are stored internally to the container image. To store them into a local storage, use the parameter **`-v <localDirectoryName>:/var/docker_image`**.

The web console

Jenkins provides a simple web console to manage the build processes.

Figure 6.1: Jenkins welcome page

For Jenkins initial setup, the **Manage Jenkins** link provides important configuration that should be defined in order to allow any build process.

Figure 6.2: The manage page from Jenkins

In order to customize essential configuration parameter, click the **Configure System** link. This will allow any administrator to refer to a Maven installation or an Ant installation. Scroll-down to the **Maven** section and click the **Add Maven** button to configure Maven. In an Internet-enabled environment, Jenkins enables an automatic installation process by checking the **Install automatically** checkbox and selecting the version to be downloaded.

Guided Exercise: Exploring the Tests for the Bookstore Application

In this lab, you will evaluate the tests from the bookstore application.

Resources	
Files:	NA
Application URL:	NA

Outcome
The student will be able to evaluate the tests available from the bookstore application.

Before you begin
JBoss EAP should be configured to run using JBoss Developer Studio (JBDS).

1. Cloning the Source Code From a Git Repository

Note

The following steps should be executed if the bookstore application was not imported to JBDS.

1.1. Open a new Terminal window from the workstation VM (**Applications** > **Utilities** > **Terminal**) and execute the following command:

```
[student@workstation ~]$ cd ~/DO290/labs
[student@workstation labs]$ git clone http://workstation.podX.example.com/
bookstore.git
```

Replace *X* with your student number.

1.2. Open JBDS by double-clicking the icon found at the desktop. Select the directory **/home/student/DO290/workspace** as the workspace and click the **Ok** button.

1.3. From the JBDS menu, select **File** > **Import** and choose the **Maven** > **Existing Maven Projects** and click the **Next** button. Click the **Browse** button and select the directory **/home/student/DO290/labs/bookstore**. Click the **Finish** button to import the project.

Note

A popup may open due to problems in **Auto share git projects**. You may safely dismiss the message by clicking the **Ok** button. The side effect of this problem is that JBDS integration with Git will not work.

2. Start JBoss EAP
   2.1. From the **Servers** tab, right-click the JBoss EAP 6 server and select the **Start** option to start the embedded JBoss instance. Wait until JBoss EAP is completely started.

   2.2. Deploy the application by right-clicking the bookstore project from the **Project Explorer** tab and select the **Run As** > **Run on Server** option. Expand the localhost node and select the JBoss EAP instance. Click the **Finish** button.

3. Run an integration test from bookstore.

   3.1. Open the **Java Resources** > **src/test/java** folder from the **Project Explorer** tab. Expand the **`com.redhat.training.service`** package and open the **`CatalogServiceTest`** class. This class will test the services provided by the **`CatalogService`** class from the **Java Resources** > **src/main/java** folder.

   3.2. Right-click the **`CatalogServiceTest`** class and select the **Run As** > **JUnit Test** item. The test will trigger a deployment process that can be observed from the **Console** tab. Another tab called **JUnit** will be opened and a green bar will be shown. This means that the tests from the **`CatalogServiceTest`** run successfully.

4. Run an acceptance test.

   4.1. Open the **Java Resources** > **src/test/java** folder from the **Project Explorer** tab. Expand the **`com.redhat.training.ui`** package and open the **`AddToCartUITest`** class. This class will test the process to add items to the shopping cart.

   4.2. Right-click the **`AddToCartUITest`** class and select the **Run As** > **JUnit Test** item. The test will trigger a deployment process that can be observed from the **Console** tab. Another tab called **JUnit** will be opened and a green bar will be shown. This means that the tests from the **`AddToCartUITest`** run successfully.

5. Delete the project from JBDS by right-clicking the project from the **Project Explorer** tab and selecting the **Delete** option. Select the **Delete project contents on disk (cannot be undone)** checkbox and click **OK**.

6. Stop the JBoss EAP instance. In the **Servers** tab from JBDS, right-click the JBoss EAP and select the **Stop** option to shut down EAP.

   This concludes this guided exercise.

Implementing CI for the Bookstore Application

Objectives

After completing this section, students should be able to configure Jenkins to build the application triggered by a Git commit.

Define the build job

Once Jenkins is configured with the tools needed to build application, applications may be added for building. To create a new application, the developer should point to a Source Code Management tool. By default, Jenkins supports CVS and Subversion as SCM tools to download source codes. Fortunately, Jenkins can be extended using plugins and additional SCMs tools can be used, such as Git. Since this course relies on Git, the Git plugin must be installed to support the continuous integration process explained previously. To install plugins, Jenkins provides a link called **Manage Plugins** accessible via the **Manage** Jenkins menu.

In addition to the web console, a CLI is also provided by Jenkins to install these plugins. It can be downloaded from the **Manage Jenkins** > **Jenkins CLI** option. To add a new build process to Jenkins, the administrator must click the **New Item** and select the project type that will be built by Jenkins (Maven, Ant, custom, etc). Each project will have its own name.

Note

Jenkins must have a build tool available to execute the build. For a traditional environment, that tool may be downloaded by and automatically configured inside Jenkins. Due to extensible nature of Jenkins, plugins may be installed to support other build tools from other languages, such as npm, rake, or pip.

Jenkins can pull source code from multiple SCM tools. To select the SCM responsible to store the source code from the application, a set of radio buttons are available in the **Source Code Management** section. Since OSE only supports Git as the SCM, the URL with the Git repository must be passed as a parameter to the **Repository URL** text field. Depending on the nature of the project, a branch may be built and this can be passed as a parameter to the **Branch to build** text field.

Due to the nature of Jenkins, the builds can be schedule to be executed at any time. Since the CI process requires that a build must happen right after a commit from a SCM tool, a **Poll SCM** option is provided to monitor when a build should happen. Once a build is triggered, Jenkins must wait until the build is completed. Normally a cron-like syntax is used to define when a new build should be executed right after the build execution request due to the last commit. A value like `H/10 * * * *` can be used to try another build after ten minutes.

Depending on the complexity of a build, Jenkins support post job execution, where a script or another build job may be started. This is useful when the application build is complex and requires a specific execution order to finish the build process. Normally, to work correctly, a multi-project configuration wizard must be used, since it provides a flexible set of configuration needed to start a complex build workflow.

Finally, to store these changes, click the **Save** button found at the bottom of the wizard.

Note

In order to force a build, select the project from the welcome page and click the **Build Now** menu item.

Guided Exercise: Triggering a Build

In this lab, you will clone the Git repository with the bookstore application, install and configure Jenkins to checkout from a Git repository, and trigger a build.

Resources	
Files:	**/home/student/DO290/installers/**
Application URL:	NA

Outcomes
You will be able to customize Jenkins to use Git as an alternative SCM tool, configure it to run Maven, and implement continuous integration process locally.

Before you begin
JBoss EAP should be integrated to JBDS.

The source code should be cloned from the Git repository and imported to JBDS.

Note

If you have not cloned the bookstore project, run the following commands from the Terminal window, replacing *X* with the station number:

```
[student@workstation ~]$ cd ~/DO290/labs
[student@workstation labs]$ git clone http://workstation.podX.example.com/
bookstore.git
```

Note

If you have not imported the project, open JBDS by double-clicking the icon found at the desktop. Select the directory **/home/student/DO290/workspace** as the workspace and click the **Ok** button. From the JBDS menu, select **File** > **Import** and choose the **Maven** > **Existing Maven Projects** and click the **Next** button. Click the **Browse** button and select the directory **/home/student/DO290/labs/bookstore**. Click the **Finish** button to import the project.

Note

A popup may open due to problems in **Auto share git projects**. You may safely dismiss the message by clicking the **Ok** button.The side effect of this problem is that JBDS integration with Git will not work.

1. Install Jenkins
   Copy the **jenkins.war** file from the **/home/student/DO290/installers** directory to the JBoss EAP 6.4 deployment directory:

```
[student@workstation ~]$ cp ~/DO290/installers/jenkins.war \
```

```
~/DO290/opt/jboss-eap-6.4/standalone/deployments
```

1.1. Using the same Terminal window, run the following commands to start EAP:

```
[student@workstation ~]$ cd ~/DO290/opt/jboss-eap-6.4/bin
[student@workstation bin]$ ./standalone.sh
```

This will deploy Jenkins to EAP 6.4. Wait until the following text is presented at the Terminal window.

```
[hudson.WebAppMain] (Jenkins initialization thread) Jenkins is fully up and
 running
```

Note

This line appears may not be obvious at first glance. Please, scroll up the Terminal window to look for this output.

2. Configure Jenkins
Open the welcome page from Jenkins from a web browser: **`http://localhost:8080/jenkins`**.

Select the **Manage Jenkins** link found at the left side from the welcome page. Click the **Manage Plugins** link and open the **Advanced** tab to upload the set of plugins available at **`/home/student/DO290/installers/jenkins-plugins.zip`**. To unzip them open a Terminal window and run the following command:

```
[student@workstation ~]$ unzip ~/DO290/installers/jenkins-plugins.zip \
   -d ~/DO290/installers/
```

The plugins must be uploaded in the following order:

1. scm-api.hpi
2. git-client.hpi
3. git.hpi

2.1. Restart Jenkins to install the new plugins. Hit **`Ctrl+C`** in the window where JBoss is running. From the same Terminal window, run the following command:

```
[student@workstation bin]$ ./standalone.sh
```

3. Configure the build tool
Select **Manage Jenkins** > **Configure System** . Click the **Add Maven** button. Name it **`maven3.0.5`**, uncheck the **Install automatically** checkbox and change the **MAVEN_HOME** parameter to **`/home/student/DO290/opt/apache-maven-3.0.5`**. Click the **Save** button to save your changes.

4. Configure a build
From the web browser, select the **New item** icon found at the welcome page from Jenkins. use the following values to the project configuration page:
   - Item name:**bookstore**
   - Project type: **Maven project**

   Click the **OK** button.

   Select the Git option from the **Source Code Management** section. Use as the **Repository URL** **http://workstation.podX.example.com/bookstore.git**. From the **Build Trigger** section, select the **Poll SCM** option and type the following string at the **Schedule** text area: ***/1 * * * ***. Also configure the Build section to use the goal **-U clean package**, to clean, download all the dependencies, run the tests, and package the application. Click the **Save** button to configure the build.

5. Commit changes to the repository
Using the **Project Explorer** tab, open the bookstore project, expand the **src** > **main** > **webapp** > **error.xhtml** and change the text **Check the server log for details** to **Please contact your system administrator** from that file. Open a Terminal window and run the following command to commit the changes:

```
[student@workstation ~]$ cd ~/D0290/labs/bookstore
[student@workstation bookstore]$ git add .
[student@workstation bookstore]$ git commit -m "Updated error.xhtml file to update
 the messages"
[student@workstation bookstore]$ git push
```

   This will trigger another build from Jenkins. From the dashboard page, select the bookstore project, and look for the **Build History** status bar, and click the **Console Output** view to check the build process.

6. Clean up the environment
From the Terminal window where JBoss EAP was started, hit **Ctrl+C**.

   Delete the project from JBDS by right-clicking the project from the **Project Explorer** tab and selecting the **Delete** option. Select the **Delete project contents on disk (cannot be undone)** checkbox and click **OK**.

   This concludes this guided exercise.

Integrating Continuous Integration (CI) into OpenShift Enterprise

Objectives

After completing this section, students should be able to integrate the Jenkins build process into the S2I mode.

CI and the Source-to-image (S2I) model

OpenShift Enterprise 3 implements the S2I process to create, deploy, and run new applications pulled from a Git repository. This facility takes a application source code as an input and produces a new image that runs the assembled application as output.

S2I is the major strategy used for building applications in OSE. The main reasons for using sources builds are:

- **User efficiency**: Developers will not perform arbitrary `yum install` opertation during their application build because S2I prevents this type of operation. The development iteration will be faster since the builder is faster.
- **Patchability**: S2I allows for rebuilding all the applications consistently if a base image needs a patch due to a security issue. For example, if a security issue is found in a PHP base image, the update of this image will update all applications that use this image as base.
- **Speed**: With S2I, the assemble process can perform a large number of complex operations without creating a new layer at each step, resulting in a fast process.
- **Ecosystem**: S2I encourages a shared ecosystem of images where best practices can be leveraged for your applications.

In a CI workflow, this facility may be used to create a container image after every commit with the latest source code and the updated image with eventual patches included. Ideally, Jenkins would be able to identify any commits made to a Git repository, run the build, and package the application. Another build job may be used to deploy the package, creating an image build by the S2I process to OpenShift.

Fortunately, the OpenShift development team has created a container image with Jenkins that implements the workflow described previously. Due the limitations of the classroom environment, an alternative version with some customizations to allow a better deployment management process was developed for this class.

Note

Container image provided by the OpenShift team: **`https://github.com/openshift/jenkins`**.

Due to its dependencies, the custom Jenkins image is already stored at the container image repository used by this course.

Modifying the build configuration for S2I

In order to implement CI using OpenShift:

- Create an OSE project to store the Jenkins application.
- Change service account privileges to allow monitoring from the build process.
- Introduce a build config to pass parameters to the S2I assemble script.
- Deploy the Jenkins container image as an application using a template.
- Configure a persistent storage to allow Jenkins application storing all the build dependencies and artifacts.
- Customize an external build to trigger the build process for the application and generate an image with the resulting artifact to OSE.
- Introduce to the application the S2I assemble scripts that will be used to customize the deployment process.

Create an OSE project

To create a project in OpenShift run the following command:

```
$ oc new-project <project_name>
```

Change service account privileges

An **oc** client is bundled to the customized Jenkins container to monitor the OSE internal registry. Subsequently, a service account for the project should be able to execute queries on the OSE3 master API and to push images to the internal OSE registry. To do so, an admin role must be added to this service account.

Note

The ID of the default service account for the project will take the form of:
system:serviceaccount:<your_project_id>:default

In addition to that, Jenkins requires access to the Git repository and internally the Jenkins user must be able to access the Git repository. To achieve this goal, we must change the security parameters to allow Jenkins to access Git. Unfortunately this access can be executed only by the administrator. This customization must be executed on the master node as the system:admin user in OSE and run the following command:

```
# oc edit scc restricted
```

Edit the line **runAsUser** and change to the type to **RunAsAny**.

Introduce a build config

The default templates provides lots of facilities to build and deploy applications, Unfortunately the build process monitoring is primitive and does not allow the developer with basic OSE administration capabilities to get information about the build process. In order to simplify this management, an instance of Jenkins may be deployed in OSE. Even though there is an official

version of Jenkins provided as a container image, it requires some customization to with OSE, such as:

- *Git repository support*: by default, Jenkins supports Subversion and CVS as the major source code management tools. The Git plugin must be installed to provide mechanisms to get the source code from the same source from OSE.

- *Deployment mechanisms*: Jenkins must be able to run the source code tests and build the artifact that will be deployed at OSE. A build job must be created to get the source code and run the build process.

- *Block the build process in OSE*: By default OSE rebuilds the application. The application must be customized to block the build execution by OSE. In order to achieve this goal, a source-to-image (S2I) script must be added to the original application (`.sti/bin/assemble`).

- *Trigger the S2I process manually*: From Jenkins, a job must be created to trigger the build process. Due to the nature of OSE, a webhook may be used to start the build process. Since the application was already build, the image must merge the build artifact.

The following build config file provides some details about the customization needed:

```
{
    "kind": "List",
    "apiVersion": "v1beta3",
    "metadata": {},
    "items": [
  <Configuration for HTTP 8080 service>
  <Configuration for HTTPS 8443 service>,
  <Configuration for Ping service>,
  <Configuration for the Database 3306 port>,
  <HTTP Route>,
  <HTTPS Route>,
  <EAP ImageStream>,
  <Build Config for the Bookstore app>,
        {
            "apiVersion": "v1",
            "kind": "BuildConfig",
            "metadata": {
                "labels": {
                    "application": "bookstore",
                    "template": "eap6-mysql-sti"
                },
                "name": "bookstore-jenkins"
            },
            "spec": {
                "output": {
                    "to": {
                        "kind": "ImageStreamTag",
                        "name": "bookstore:latest"
                    }
                },
                "source": {
                    "contextDir": "",
                    "git": {
                        "ref": "master",
                        "uri": "https://workstation.podX.example.com/bookstore.git"
                    },
                    "type": "Git"
                },
                "strategy": {
                    "sourceStrategy": {
```

```
                    "from": {
                        "kind": "ImageStreamTag",
                        "name": "jboss-eap6-openshift:6.4",
                        "namespace": "openshift"
                    },
                    "env": [
                      {
                            "Name": "DISABLE_ASSET_COMPILATION",
                            "Value": "true"
                      }, {
                            "Name": "APPLICATION_ARTIFACT_URL",
                            "Value": "http://jenkins.cloudappsX.example.com/job/
bookstore/lastSuccessfulBuild/artifact/target/bookstore.war" ❷
                      }, {
                            "Name": "DOWNLOAD_USER_CREDENTIALS",
                            "Value": "admin:password"❸
    }
  ]
                },
                "type": "Source"
            },
             <Trigger config>
            }
        },
        {
            "apiVersion": "v1",
            "kind": "DeploymentConfig",
            "metadata": {
                "labels": {
                    "application": "bookstore",
                    "template": "eap6-mysql-sti"
                },
                "name": "bookstore"
            },
            "spec": {
                "replicas": 0,
                "selector": {
                    "deploymentConfig": "bookstore"
                },
                "strategy": {
                    "type": "Recreate"
                },
                "template": {
                    "metadata": {
                        "labels": {
                            "application": "bookstore",
                            "deploymentConfig": "bookstore"
                        },
                        "name": "bookstore"
                    },
                    "spec": {
                        "containers": [
                            {
                        <EAP Container configuration>
                                "readinessProbe": {
                                    "exec": {
                                        "command": [
                                            "/bin/bash",
                                            "-c",
                                            "/opt/eap/bin/readinessProbe.sh"❹
                                        ]
                                    }
                                },
```

```
                                "volumeMounts": [
                                    {
                                        "mountPath": "/etc/eap-secret-volume", ❺
                                        "name": "eap-keystore-volume",
                                        "readOnly": true
                                    }
                                ]
                            }
                        ],
                        <Secrets config>
                    }
                },
                <Triggers>
            }
        },
        <Configuration omitted>
}
```

❶ The URL where the Git repo with the source code is available for clone/pull.

❷ The URL where the latest build generated by Jenkins is stored.

❸ The credentials used by a user to access Jenkins.

❹ Script used to monitor the build process.

❺ The persistent volume used by Jenkins to store data.

Deploy the Jenkins container using a template

A custom template will be used to download the Jenkins custom image available at the workstation.podX.example.com registry, create the Jenkins application, and passing some parameters that will be needed by the build process. The template file is as it follows:

```
{
  "kind": "Template",
  "apiVersion": "v1",
  "metadata": {
    "name": "jenkins-ephemeral",
    }
  },
 <Contents omitted>
   {
      "kind": "Route",
      "apiVersion": "v1",
      "metadata": {
        "name": "jenkins",
        "creationTimestamp": null
      },
      "spec": {
        "host": "jenkins.cloudappsX.example.com", ❶
        "to": {
          "kind": "Service",
          "name": "${JENKINS_SERVICE_NAME}"
        }
      }
    },
    {
      "kind": "DeploymentConfig",
     <Deployment Config omitted>
        "spec": {
          "containers": [
            {
              "name": "jenkins",
```

```
                "image": "openshift/jenkins-ose-dev",❷
                "env": [
                  {
                    "name": "JENKINS_PASSWORD",
                    "value": "${JENKINS_PASSWORD}"
                  }
                ],
                "resources": {},
                "volumeMounts": [
                  {
                    "name": "${JENKINS_SERVICE_NAME}-data",
                    "mountPath": "/var/lib/jenkins"
                  }
                ],
                "terminationMessagePath": "/dev/termination-log",
                "imagePullPolicy": "IfNotPresent",
                "capabilities": {},
                "securityContext": {
                  "capabilities": {},
                  "privileged": false
                }
              }
            ]
    <Configuration omitted>
    }
  ],
  "parameters": [
    {
      "name": "JENKINS_SERVICE_NAME",
      "description": "Jenkins service name",
      "value": "jenkins"
    },
    {
      "name": "JENKINS_PASSWORD",❸
      "description": "Password for the Jenkins user",
      "generate": "expression",
      "value": "password"
    }
  ],
  "labels": {
    "template": "jenkins-ephemeral-template"
  }
}
```

❶ The address where Jenkins will be available for external access.

❷ Place where the OpenShift should pull the image from a Docker registry.

❸ The admin password used to access Jenkins. If none is provided, the default password will be **password**.

Implementing secrets

Secrets provide a mechanism to hold sensitive information such as passwords, OpenShift client config files, keys used by HTTPS protocol, etc. Secrets decouple sensitive content from the pods that use it and can be mounted into containers using a volume plug-in or used by the system to perform actions on behalf of a pod. Secret data:

- Can be referenced independently from its definition.
- Never comes to rest on the node. Volumes are backed by temporary file-storage facilities (tmpfs).

- Can be shared within a namespace.

Configure a persistent storage
Optionally, a persistent storage may be create to leave all the dependencies and files used by Jenkins to build applications locally for reuse. In order to configure a persistent storage, a network file storage (NFS) should be configured by the administrator in the master machine and make it mountable by all the nodes.

Note

The NFS creation is beyond the scope of this course.

To make it available for a project in OpenShift a persistent volume must be created pointing to the mounting point of a NFS share. This can be configured using a JSON file similar to:

```
{
  "apiVersion": "v1",
  "kind": "PersistentVolume",
  "metadata": {
    "name": "smallvolume"
  },
  "spec": {
  "capacity": {
  "storage": "2Gi"
  },
  "accessModes": [ "ReadWriteMany" ],
    "nfs": {
      "path": "/var/export/vol"❶,
      "server": "nas.pod0.example.com"❷
    }
  }
}
```

❶ The directory in which the NFS is mounted.

❷ The server host name where the NFS is available.

To allocate space from a PV for an application, a persistent volume claim (PVC) should be added:

```
$ oc volume dc/jenkins --add --overwrite \
    -t persistentVolumeClaim --claim-name=<persistent_volume_claim_name> \
    --name=<persistent_volume_name>
```

To associate the PVC to an application, the developer must refer to that PVC during the application creation. Since the Jenkins deployment config refers to a PVC, the previous command will overwrite it.

Customize an external build process to trigger the image builder
The application build job in Jenkins will be able to generate the artifact. However, it will not automatically trigger the S2I process in OSE. Fortunately Jenkins provides a facility to trigger another build if a A script should be used to trigger the application deployment in OSE. This script will be used by Jenkins to trigger the build and deploy the application using a facility from OSE called webhook. The idea behind a webhook is to call a URL from OSE to trigger the build process. To identify the URL from any application deployed in OpenShift, a command must be executed from the command line:

```
$ oc describe bc <app-name>
```

the output is presented as follows:

```
..OUTPUT OMITTED...
Webhook Generic: https://master.podX.example.com:8443/oapi/v1/namespaces/custom/
buildconfigs/myapp/webhooks/sZ6qqLxI0eFUib1-x-oz/generic
Webhook GitHub:        https://master.podX.example.com:8443/oapi/v1/namespaces/custom/
buildconfigs/myapp/webhooks/m8NeRZ-Q-lH44TRCUteN/github
..OUTPUT OMITTED...
```

The build job from Jenkins can trigger the build process by calling curl with some parameters to trigger the build:

```
$ curl -i -H "Accept: application/json" -H "X-HTTP-Method-Override: PUT" -X POST -k
 <generic-trigger-url>
```

CI Workflow

Ideally, the CI workflow a tool such as Jenkins will:

- *Checkout or clone a repository*: using a SCM where the source code is available, such as Git or SVN, the CI tool will get the latest updates.

- *Execute the build process*: A tool such as Maven, Rake, or PIP can be used to build, compile, and test the application. In a closed environment such as the one provided by a container image, starting up a server to test the application may potentially create port conflicts that can break the build process. To allow a better integration, the best approach would be create a container image that will be used to deploy the application for testing purposes. However, depending on the complexity of the container, it will be better to deploy it externally, running as a segregated server instance. For example, in a Java EE application, where a complex set of services must be started, the easier approach is to start a standalone server that will get all the deployments requested by Jenkins for testing purposes.

Note

JBoss EAP 6 can be used to execute these integration tests. In order to allow the remote deployment, it is important to enable the access to the management console and the public web applications. In a standalone installation, JBoss can be started using the command line **`bin/standalone.sh -bmanagement <IPPort> -b <IPAddress>`**

- *Provide externally the resulting package for external consumption:*The CI tool will provide links to download the generated artifact. In OpenShift, the application build should be triggered by a webhook to start the build process in S2I. However, S2I, by default, triggers the build process again to generate the image. To avoid the build of a S2I rebuilding the source code, a S2I customization must be used to deactivate the build process and use the generated artifact to build the new container image.

Demonstration: Integrating Jenkins builds with OpenShift Enterprise

In this example, please read these steps while your instructor demonstrates how to integrate Jenkins builds with OpenShift Enterprise.

1. *Login OSE*: The **oc** client bundled in the Jenkins container will access OSE via the default service account. From the **workstation** VM run the **oc login** command using the following credentials:
   - Username: **student**
   - Password: **redhat**
   - Server: **https://master.pod0.example.com:8443**

2. *Create a project*: from the workstation VM, open a Terminal window and run the following command:

```
[student@workstation ~]$ oc new-project bookstore-ci-demo
```

3. *Review the contents of the Jenkins OSE template* : Open the **/home/student/DO290/labs/bookstore-ci-demo/jenkins-template.json** data file using a text editor. Modify the **hostname** parameter on the **Route** object to **jenkins.cloudappsX.example.com** and run the following command from the workstation VM.

```
[student@workstation ~]$ oc new-app \
    /home/student/DO290/labs/bookstore-ci-demo/jenkins-template.json
```

   Verify if Jenkins was deployed by running:

```
[student@workstation ~]$ watch oc get pods
```

   Wait until that the status of the Jenkins pod is **Running**.

4. *Check if Jenkins is running*: Open a web browser from the workstation host and open the address **http://jenkins.cloudappsX.example.com** to verify if the Jenkins is running. Login to Jenkins using **admin** as username and **password** as password. Replace the *X* with the station number.

5. *Start JBoss EAP 6*: This instance will be responsible for running the integration and acceptance tests from the bookstore application. To start it, the management console must be activated to accept external requests. The application must be accessible externally to allow Jenkins to run the acceptance tests. Run the following commands to start JBoss EAP:

```
[student@workstation ~]$ cd ~/DO290/opt/jboss-eap-6.4/bin
[student@workstation bin]$ ./standalone.sh -bmanagement 172.25.X.9 -b 172.25.X.9
```

   Replace the *X* with the station number.

6. *Clone the git repository*: From the Terminal window, run the following command:

```
[student@workstation ~]$ cd ~/DO290/labs
[student@workstation ~]$ git clone http://workstation.podX.example.com/bookstore-
jenkins.git
```

7. *Customize the testing parameters for the bookstore application*: The bookstore application uses Arquillian to run the integration tests. To deploy the application remotely, the **`src/test/resources/arquillian.xml`** file must be edited to point to the full qualified domain name (FQDN) to the workstation and use the port 9999. Additionally, to allow a remote access the credentials must be configured as well. To achieve these goals edit the **`arquillian.xml`** file from the bookstore as it follows:

```
<arquillian xmlns="http://jboss.org/schema/arquillian"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://jboss.org/schema/arquillian
        http://jboss.org/schema/arquillian/arquillian_1_0.xsd">

    <engine>
        <property name="deploymentExportPath">target/deployments</property>
    </engine>
    <extension qualifier="webdriver">
        <property name="browser">phantomjs</property>
<!--          <property name="browser">firefox</property> -->
    </extension>

    <container qualifier="jbossas" default="true">
        <configuration>

            <!-- Change the following to the ip address of your JBoss EAP server -->
            <property name="managementAddress">workstation.podX.example.com</
property>(1)
            <property name="managementPort">9999</property>

            <property name="username">admin</property>(2)

            <property name="password">JBoss@RedHat123</property>(3)
        </configuration>
    </container>

</arquillian>
```

(1) The FQDN where the previously started JBoss EAP is running.

(2) Username to connect to the JBoss EAP 6 administrative console

(3) The password from the JBoss EAP 6 administrative console.

For the actual environment, the FQDN must be updated to the actual JBoss EAP instance.

8. *Edit the .sti/bin/assemble script*: This script will be used by OSE to customize the image creation and deployment process. In order to avoid the Maven build execution (that was previously triggered by Jenkins), edit the **`/home/student/DO290/labs/bookstore-jenkins/.sti/bin/assemble`** file with a text editor and update the following line:

```
APPLICATION_ARTIFACT_URL=http://jenkins.cloudappsX.example.com/job/bookstore/
lastSuccessfulBuild/artifact/target/bookstore.war
```

Replace the *X* with the station number.

Evaluate the contents of the file. It skips the Maven build process and downloads the war file generated by Jenkins and appends to the JBoss container image.

9. *Commit changes to the SCM tool*: using the command line, the recently updated file must be committed and pushed to the remote Git repository. To achieve this goal run the following command:

```
[student@workstation ~]$ cd ~/D0290/git/bookstore-jenkins
[student@workstation bookstore-jenkins]$ git add .
[student@workstation bookstore-jenkins]$ git commit -m 'Support testing'
[student@workstation bookstore-jenkins]$ git push
```

10. *Deploy the secrets*: In order to allow secure communication, a JKS file must be provided to OSE for the bookstore application. The file is already provided at the **/home/student/D0290/labs/bookstore-ci-demo/eap-app-secret.json**. Run the following command to deploy the secrets:

```
[student@workstation bookstore-jenkins]$ oc create -f /home/student/D0290/labs/
bookstore-ci-demo/eap-app-secret.json
```

11. *Deploy the bookstore application*: Open the **/home/student/D0290/labs/bookstore-ci-demo/bookstore.json** and edit the file to change all the references to **workstation.pod*X*.example.com** and **cloudapps*X*.example.com**. Using the **bookstore.json** file, create the bookstore application.

```
[student@workstation ~]$ oc create -f /home/student/D0290/labs/bookstore-ci-demo/
bookstore.json
```

12. *Access the Jenkins web console*: Open in a browser the following URL:**http://jenkins.cloudapps*X*.example.com**. Authenticate into the Jenkins container using the following credentials:
    - Username: **admin**
    - Password: **password**

13. *Customize the bookstore Jenkins build job*: The bookstore is already customized to support the build process. Select the **bookstore** build and edit the following parameters:
    - OSE_NEXUS_HOST: **http://workstation.pod*X*.example.com:8081/nexus/content/groups/RedHatTraining**
    - Git Repository:**http://workstation.podX.example.com/bookstore-jenkins.git**
    - Goals: **-U clean package -DOSE_NEXUS_HOST=$OSE_NEXUS_HOST**

    Click the **Save** button.

14. *Customize the bookstore-invoke-ose3 Jenkins build job*:The bookstore is already customized to support the build process. Select the **bookstore-invoke-ose3** build and edit the parameter:
    - WEBHOOK_URL

The value can be obtained running the following command:

```
[student@workstation ~]$ oc describe bc bookstore
```

Look for the **Generic Webhook URL**.

15. Start the build process using the bookstore application. Open the bookstore build job and click the **Enable** button. And click the **Build with Parameters** option from the left menu to start the build.

16. Remove the project from OSE using the following command:

```
[student@workstation ~]$ oc delete project bookstore-ci-demo
```

17. Stop the JBoss EAP instance by hitting **Ctrl+C** from the Terminal window where JBoss EAP was started.

18. Delete the bookstore project by running the folllowing command:

```
[student@workstation ~]$ rm -rf ~/DO290/labs/bookstore-jenkins
```

Guided Exercise: Building the Bookstore Application in OpenShift Enterprise with Jenkins

In this lab, you will build the bookstore application in OSE with Jenkins

Resources	
Files	`/home/student/DO290/labs/bookstore-ci`
Application URL	`http://jenkins.cloudappsX.example.com`

Outcome
The student will be able to trigger a deployment using Jenkins' image in OSE.

Before you begin
OpenShift must be installed and running.

OpenShift client must be installed at the workstation machine.

The previous demo should be executed to enable Jenkins.

1. Login using **oc** command
   From the workstation VM, open a Terminal window and run the following command:

```
[student@workstation ~]$ oc login -u student -p redhat
```

2. Create the bookstore-ci project
   From the same Terminal window, run:

```
[student@workstation ~]$ oc new-project bookstore-ci
```

3. Deploy Jenkins to the project
   Jenkins must be deployed on OSE to manage the build process that will be responsible to create the war file. In order to achieve this goal a customized template. Open it and update as follows:

   3.1. Change all the references about Git repositories to refer to your workstation VM full qualified domain name, using your station number, instead of *X*.

   3.2. Update the **APPLICATION_ARTIFACT_URL** to refer to the address where your Jenkins instance is running (**`jenkins.cloudappsX.example.com`**)

```
[student@workstation ~]$ oc new-app \
    ~/DO290/labs/bookstore-ci/jenkins-template.json
```

4. Configure the bookstore application objects
   Modify the **`~/DO290/labs/bookstore-ci/bookstore.json`**, replacing the *X* from the addresses to the station number.

4.1. Alter the **host** field of the **bookstore-http-route** from **bookstore.cloudappsX.example.com**.

4.2. Alter the **master** field from the second **BuildConfig** kind to **http://workstation.podX.example.com/bookstore-jenkins.git**.

4.3. Alter the place where the bookstore application will be cloned from **http://workstation.podX.example.com/bookstore-jenkins.git**.

4.4. Modify the value of the **APPLICATION_ARTIFACT_URL** build config environment variable to point to the Jenkins **http://jenkins.cloudappsX.example.com**.

5. Create the secrets needed by this project running the following command:

```
[student@workstation ~]$ oc create -f \
    ~/DO290/labs/bookstore-ci/eap-app-secret.json
```

6. Create new **bookstore** related OSE objects:

```
[student@workstation ~]$ oc create -f \
    ~/DO290/labs/bookstore-ci/bookstore.json
```

7. EAP will be responsible for running the integration and acceptance tests from the bookstore application. To start it, the management console must be activated to accept external requests. Run the following commands to start JBoss EAP:

```
[student@workstation ~]$ cd ~/DO290/opt/jboss-eap-6.4/bin
[student@workstation bin]$ ./standalone.sh -bmanagement 172.25.X.9 -b 172.25.X.9
```

8. Checkout the customized bookstore application. From the Terminal window, run the following command:

```
[student@workstation ~]$ cd ~/DO290/labs
[student@workstation ~]$ git clone http://workstation.podX.example.com/bookstore-
jenkins.git
```

The bookstore application uses Arquillian to run the integration tests. To deploy the application remotely, the **~/DO290/labs/bookstore-jenkins/src/test/resources/arquillian.xml** file must be edited to point to the full qualified domain name (FQDN) to the workstation and use the port 9999. Additionally, to allow a remote access the credentials must be configured as well. To achieve these goals edit the **arquillian.xml** file. The username to access the JBoss EAP instance management console is **admin** and the password is **JBoss@RedHat123**.

9. The **~/DO290/labs/bookstore-jenkins/.sti/bin/assemble** script will be used by OSE to customize the image creation and deployment process. In order to avoid the Maven build execution (that was previously triggered by Jenkins), edit the **/home/student/DO290/labs/bookstore-jenkins/.sti/bin/assemble** file with a text editor and update the following line:

```
APPLICATION_ARTIFACT_URL=http://jenkins.cloudappsX.example.com/job/bookstore/
lastSuccessfulBuild/artifact/target/bookstore.war
```

Replace the *X* with the station number.

10. Edit the **~/DO290/labs/bookstore-jenkins/src/main/resources/com/redhat/training/i18n/bookstore.properties** file to break a test. The reason for this is that a new deploy can not happen if any test fail. Change from:

```
template.greeting.username=Username
```

to:

```
template.greeting.username=User
```

11. *Commit changes to the SCM tool*: using the command line, the recently updated file must be committed and pushed to the remote Git repository. To achieve this goal run the following command:

```
[student@workstation ~]$ cd ~/DO290/labs/bookstore-jenkins
[student@workstation bookstore-jenkins]$ git add .
[student@workstation bookstore-jenkins]$ git commit -m 'Updated to include IP
 address'
[student@workstation bookstore-jenkins]$ git push
```

12. *Access the Jenkins web console*: Open in a browser the following URL:**http://jenkins.cloudappsX.example.com**. Authenticate into the Jenkins container using the following credentials:
    - Username: **admin**
    - Password: **password**

13. *Customize the bookstore Jenkins build job*: The bookstore is already customized to support the build process. Select the **bookstore** build and edit the following parameters:
    - OSE_NEXUS_HOST: **http://workstation.podX.example.com:8081/nexus/content/groups/RedHatTraining**
    - Git Repository:**http://workstation.podX.example.com/bookstore-jenkins.git**
    - Goals: **-U clean package -DOSE_NEXUS_HOST=$OSE_NEXUS_HOST**

    Click the **Save** button.

14. *Customize the bookstore-invoke-ose3 Jenkins build job*:The bookstore is already customized to support the build process. Select the **bookstore-invoke-ose3** build, click the Configure option, and edit the parameter **WEBHOOK_URL**.

    The value can be obtained running the following command and look for the Generic Webhook line:

```
[student@workstation ~]$ oc describe bc bookstore
```

15. Start the build process using the bookstore application. Open the bookstore build job and click the **Enable** button. And click the **Build with Parameters** option from the left menu to start the build.

16. Navigate to the **bookstore** home page in Jenkins, click **Build** on the left, and in the **Build History** section on the left of the page, click the latest build. On the left of the build page, click **Console Output** and monitor the progress of the OSE build of your **bookstore** application.

    You will know that the build fails when you see the following log messages at the bottom of the Jenkins Console Output:

```
Tests run: 26, Failures: 1, Errors: 0, Skipped: 0
...
Finished: FAILURE
```

17. Edit the **~/DO290/labs/bookstore-jenkins/src/main/resources/com/redhat/training/i18n/bookstore.properties** file to fix the test.

    Change from:

```
template.greeting.username=User
```

    to:

```
template.greeting.username=Username
```

18. Commit changes to the SCM tool:

```
[student@workstation ~]$ cd ~/DO290/labs/bookstore-jenkins
[student@workstation bookstore-jenkins]$ git add .
[student@workstation bookstore-jenkins]$ git commit -m 'Fixed the test'
[student@workstation bookstore-jenkins]$ git push
```

19. Start a new build in Jenkins. Open the bookstore build job and click the **Build with Parameters** option from the left menu to start the build.

20. Navigate to the **bookstore** home page in Jenkins, click **Build** on the left, and in the **Build History** section on the left of the page, click the latest build. On the left of the build page, click **Console Output** and monitor the progress of the OSE build of your **bookstore** application.

    You will know that the build is successful when you see the following log messages at the bottom of the Jenkins Console Output:

```
Finished: SUCCESS
```

21. A new build should start in OpenShift:

```
[student@workstation ~]$ oc get builds
NAME                 TYPE      STATUS     POD
bookstore-1          Source    Complete   bookstore-1-build
bookstore-2          Source    Running    bookstore-2-build
bookstore-jenkins-1  Source    Complete   bookstore-jenkins-1-build
```

22. Wait while the application is deployed.

```
[student@workstation ~]$ oc get pods
NAME                         READY     STATUS      RESTARTS   AGE
bookstore-1-build            0/1       ExitCode:0  0          13m
bookstore-2-build            0/1       ExitCode:0  0          6m
bookstore-3-clxoi            1/1       Running     0          4m
bookstore-jenkins-1-build    0/1       ExitCode:0  0          13m
bookstore-mysql-1-jcrkv      1/1       Running     0          13m
jenkins-1-u7or8              1/1       Running     0          15m
```

23. Open a web browser (**Applications** > **Internet** > **Firefox**) on the **workstation.pod*X*.example.com** and open the following URL: **http://bookstore.cloudapps*X*.example.com**.

24. Hit **Ctrl+C** at the JBoss command prompt to stop the EAP server.

25. Remove the project running the following command:

```
[student@workstation ~]$ oc delete project bookstore-ci
```

Lab: Implementing CI for the Node.js application

In this lab, you will customize a NodeJS application to be configured in Jenkins, execute tests and deploy it as a image container.

Resources	
Files	**/home/student/DO290/labs/nodejs-ci**
Application URL	**https://jenkins.cloudapps*X*.example.com**

Outcome
You should be able to deploy a Jenkins container image in OpenShift to install a Node.js application.

Before you begin
- Verify the OpenShift Enterprise Client is installed.
- OpenShift Enterprise is installed.
- The user student should be logged in and will be responsible for running the following lab.

1. Create hello-nodejs OSE objects
2. Jenkins Build Projects
   Included in the Jenkins container are a couple of Jenkins jobs related to the **hello-nodejs** application. The **hello-nodejs** and **hello-nodejs-invoke-ose3** Jenkins jobs consist of tasks and parameters that build, test and deploy our **hello-nodejs** application.

   2.1. In the **Configure** page of the **hello-nodejs-invoke-ose3** Jenkins job, scroll to the list of build parameters and change it:

   - WEBHOOK_URL: **<URL>**

   Change the parameter to the URL from the **Generic Webhook** from the following command:

   ```
   [student@workstation ~]$ oc describe bc hello-nodejs
   ```

   2.2. In the Jenkins main page, choose the **hello-nodejs** build job. Click the **Configure** link to change parameters from the Jenkins build.

   Update the **Source Code Management** to point to the correct Git repository: **http://workstation.pod*X*.example.com/ose-dev3-helloworld-nodejs.git**. Replace *X* with the student's number.

   In the **Build** section, update the **Execute shell** text areas to include the parameter **--registry=http://workstation.pod*X*.example.com:8081/nexus/content/groups/nodejs** right after the **npm install** and **npm test**.

Note

For an Internet enabled environment the **`--registry`** parameter may be dropped.

2.3. Once you have tailored the values of your build parameters, click **Save** (at the bottom of the page).

2.4. By default, the **hello-nodejs** Jenkins job is disabled. It can be enabled by navigating to the homepage of the **hello-nodejs** job and clicking **Enable**.

3. Trigger a **`hello-nodejs`** Build
To trigger a build of our **`hello-nodejs`** Jenkins job, we can either wait for 5 minutes (as per the job's **`Build Trigger`**) or manually initiate the build. At this time, initiate the build as follows:

3.1. Navigate to the **`hello-nodejs`** home page in Jenkins.

On the left, click **Build**.

In the **Build History** section on the left of the page, click on the latest build. to open the build page.

On the left of the build page, click **Console Output**

From the console output, monitor the progress of the OSE build of your **`hello-nodejs`** application.

The build will be finished when you see the following log messages at the bottom of the Jenkins **Console Output**:

```
Finished: SUCCESS
```

Note

Check the **Console Output** of both the **`hello-nodejs`** and **`hello-nodejs-invoke-ose3`** Jenkins jobs.

3.2. Test the application. Open a web browser and access the following address: **`hello-nodejs.cloudappsX.example.com`**

This concludes the lab.

Solution

In this lab, you will customize a NodeJS application to be configured in Jenkins, execute tests and deploy it as a image container.

Resources	
Files	**/home/student/DO290/labs/nodejs-ci**
Application URL	**https://jenkins.cloudappsX.example.com**

Outcome
You should be able to deploy a Jenkins container image in OpenShift to install a Node.js application.

Before you begin
- Verify the OpenShift Enterprise Client is installed.
- OpenShift Enterprise is installed.
- The user student should be logged in and will be responsible for running the following lab.

1. Create hello-nodejs OSE objects
   1.1. From the workstation VM, open a Terminal window and connect to OpenShift as **student** user:

```
[student@workstation ~]$ oc login -u student -p redhat
```

   1.2. Create a new project:

```
[student@workstation ~]$ oc new-project hello-nodejs
```

   1.3. Open the **jenkins-template.json** file to update the **jenkins.cloudappsX.example.com** host to use the student's number. Deploy Jenkins with the updated file. Replace *X* by your student number:

```
[student@workstation ~]$ oc new-app /home/student/DO290/labs/nodejs-ci/jenkins-
template.json
```

   1.4. Open a web browser and open the following address: **http://jenkins.cloudappsX.example.com** to verify if Jenkins is running. The login is **admin** and the password is **password**.

   1.5. Using the out-of-the-box **nodejs imagestream**, create OSE objects to host the **hello-hodejs** application. Replace *X* with your student number:

```
[student@workstation ~]$ oc new-app http://workstation.podX.example.com/ose-
dev3-helloworld-nodejs.git --name=hello-nodejs
```

   1.6. Add a route to the **hello-hodejs** service. Replace *X* with your student number:

```
[student@workstation ~]$ oc expose service hello-nodejs --hostname=hello-
nodejs.cloudappsX.example.com
```

2. Jenkins Build Projects
Included in the Jenkins container are a couple of Jenkins jobs related to the **hello-nodejs** application. The **hello-nodejs** and **hello-nodejs-invoke-ose3** Jenkins jobs consist of tasks and parameters that build, test and deploy our **hello-nodejs** application.

   2.1. In the **Configure** page of the **`hello-nodejs-invoke-ose3`** Jenkins job, scroll to the list of build parameters and change it:

   - WEBHOOK_URL: **<URL>**

   Change the parameter to the URL from the **`Generic Webhook`** from the following command:

```
[student@workstation ~]$ oc describe bc hello-nodejs
```

   2.2. In the Jenkins main page, choose the **hello-nodejs** build job. Click the **Configure** link to change parameters from the Jenkins build.

   Update the **Source Code Management** to point to the correct Git repository: **`http://workstation.pod`***X*`.example.com/ose-dev3-helloworld-nodejs.git`**. Replace *X* with the student's number.

   In the **Build** section, update the **Execute shell** text areas to include the parameter **`--registry=http://workstation.pod`***X*`.example.com:8081/nexus/content/groups/nodejs`** right after the **`npm install`** and **`npm test`**.

Note

For an Internet enabled environment the **`--registry`** parameter may be dropped.

   2.3. Once you have tailored the values of your build parameters, click **Save** (at the bottom of the page).

   2.4. By default, the **hello-nodejs** Jenkins job is disabled. It can be enabled by navigating to the homepage of the **hello-nodejs** job and clicking **Enable**.

3. Trigger a **`hello-nodejs`** Build
To trigger a build of our **`hello-nodejs`** Jenkins job, we can either wait for 5 minutes (as per the job's **`Build Trigger`**) or manually initiate the build. At this time, initiate the build as follows:

   3.1. Navigate to the **`hello-nodejs`** home page in Jenkins.

   On the left, click **Build**.

   In the **Build History** section on the left of the page, click on the latest build. to open the build page.

   On the left of the build page, click **Console Output**

From the console output, monitor the progress of the OSE build of your **hello-nodejs** application.

The build will be finished when you see the following log messages at the bottom of the Jenkins **Console Output**:

```
Finished: SUCCESS
```

Note

Check the **Console Output** of both the **hello-nodejs** and **hello-nodejs-invoke-ose3** Jenkins jobs.

3.2. Test the application. Open a web browser and access the following address: **hello-nodejs.cloudapps*X*.example.com**

This concludes the lab.

Summary

In this chapter, you learned:

- Continuous integration can be used to improve the feedback and quality of any application developed.
- DevOps practices are highly based on feedback provided by tools (not people) in a highly automated environment.
- OpenShift provides an integrated environment for deployment purposes and can be used to deploy Jenkins, as a container.
- Jenkins deployment in OpenShift requires some customization to trigger deployment of new artifacts.

CHAPTER 7

CREATING COMPLEX DEPLOYMENTS

Overview	
Goal	Create deployments that utilize clustering, load balancing, and customized node scheduling.
Objectives	• Scale an application up and down by changing its number of replicas. • Deploy the bookstore application in a cluster. • Configure the OSE router for high availability (HA). • Use labels and selectors to control to which nodes pods are deployed on.
Sections	• Scaling An Application • Clustering the Bookstore Application (and Guided Exercise) • Configuring the OSE Router (and Quiz) • Controlling Node Scheduling (and Quiz)

Scaling an Application

Objectives

After completing this section, students should be able to scale an application up and down by changing its number of replicas.

OSE DeploymentConfigs and Kubernetes ReplicationControllers

Although OpenShift Enterprise (OSE) can create standalone application pods, when using the web console or the **oc new-app** command, it wraps a pod definition inside a **DeploymentConfig** (dc) resource. A dc provides advanced features like rolling deployments for updating the application and elasticity to deal with variable user demand.

An OSE dc creates and manages at runtime one or more Kubernetes **ReplicationController** (rc) resources. An rc manages the life cycle of a group of pods created from the same definition, that is, it creates **replicas** of an application pod. It monitors all pods in the group and creates new ones, or destroy existing ones, to meet the configured number of replicas.

The **DeploymentConfig** adds to the rc the ability to detect changes to a container image, refreshing currently running pods to use the updated image. It can do that using a few different strategies, such as **rolling upgrade**: By that strategy, the dc creates a new rc instance for pods created from the new image, while keeping the old rc instance along to manage existing pods created from the old image. The dc then increases the number of replicas for the new rc, while decreasing the number of replicas of the old rc. This way, pods using the new image progressively replace pods using the old image until all running pods are using the new image, without disrupting active application users.

Actually a dc does not create the rc (and thus the pods) directly. It does that by creating a **Deployment** (deployment) resource. A deployment creates a rc and a rc creates pods. Having each deployment as a distinct resource allows OSE to keep information about particular deployments: whether they worked or failed, when they started, how long each took, and to roll back or roll forward to different releases of the application.

A dc is closely related to a **Service** (svc) resource, which is also created by the web console or the **oc new-app** command. A svc provides a group of pods, supposed to have been created from the same definition, with a single IP address, and load-balances network connections among them. Pods are supposed to connect to services and not directly to other pods.

The web console project overview page and the **oc status** command have a focus in showing runtime information about an application and will not show **DeploymentConfig** and **ReplicationController**. Instead they show services and deployments, which are the objects having runtime information.

The following figures illustrates the web console overview page for the **hello-openshift** application, when created from source code in a Git repository.

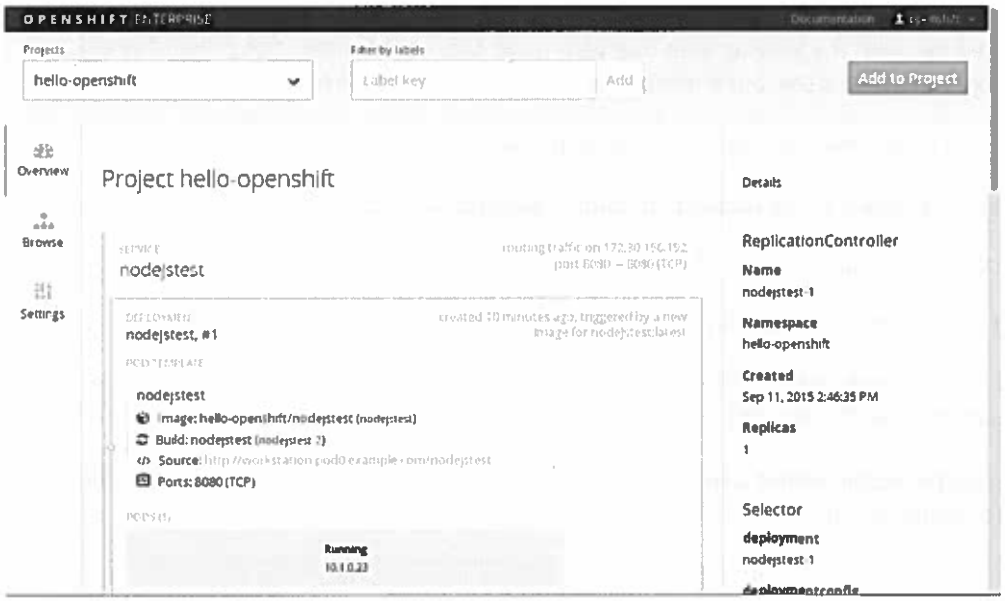


Figure 7.1: A project overview page, showing services, deployments, and pods

If a deployment is selected, like in the following figure, the details panel show information about the rc created by the deployment, like the desired number of replicas (or pods). The details pane also show information about the dc that generated the deployment, but this information is cropped in the previous figure.

The web console project overview page shows deployments as children of services because pods created from a deployment are load-balanced by a service, but in fact there is no direct relationship between either the service and the rc that created the pods, or the service and the dc that created the rc.

Note

The web console project overview page also shows information about builds that create the images used by the rc to create pods. **`Build`** (build) resources, and their corresponding **`BuildConfig`** (bc), resources are an essential component of the OSE Source-to-Image (S2I) feature and will be detailed in the following chapters of this course.

A similar view is provided by the **`oc status`** command. A sample output follows:

```
In project hello-openshift

service/nodejstest - 172.30.156.192:8080
  dc/nodejstest deploys istag/nodejstest:latest <-
    builds http://workstation.pod0.example.com/nodejstest with openshift/nodejs:latest
 through bc/nodejstest
    #1 deployed 9 minutes ago - 1 pod

To see more, use 'oc describe <resource>/<name>'.
You can use 'oc get all' to see a list of other objects.
```

This time the **DeploymentConfig** (dc) resource is shown, bellow the service and above the build. Just like with the web console overview page, this nesting does not reflect the actual structure of the resources, but tries to give a runtime view of the resources:

- Requests from other applications come to the service.
- The service forwards the requests to pods created by the dc.
- The dc starts a build.
- The result of the build was deployed, creating pods.

The pods themselves are not shown by the **oc status** command. To see the pods from the command line, use the **oc get pods** command.

To see specific details about a **DeploymentConfig**, check the output of **oc describe dc**, as in the following listing:

```
Name:  nodejstest
Created: 29 minutes ago
Labels:  app=nodejstest
Latest Version: 1
Triggers: Config, Image(nodejstest@latest, auto=true)
Strategy: Rolling
Template:
  Selector: deploymentconfig=nodejstest
  Replicas: 1
  Containers:
  NAME  IMAGE                ENV
  nodejstest 172.30.26.87:5000/hello-openshift/
nodejstest@sha256:96959ee4bad9875e0e372d07232b3dd8eeca6074b3cd66f564193ab96e3db708
Deployment #1 (latest):
 Name:  nodejstest-1
 Created: 23 minutes ago
 Status:  Complete
 Replicas: 1 current / 1 desired
 Selector: deployment=nodejstest-1,deploymentconfig=nodejstest
 Labels:  app=nodejstest,openshift.io/deployment-config.name=nodejstest
 Pods Status: 1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen    LastSeen   Count From  SubobjectPath Reason  Message
  Fri, 11 Sep 2015 14:41:01 -0400 Fri, 11 Sep 2015 14:44:40 -0400 3 {deployer }
 failedCreate Couldn't create initial deployment: DeploymentConfig "nodejstest" is
 invalid: triggers[1].imageChange.tag: invalid value 'latest': no image recorded for
 hello-openshift/nodejstest:latest
```

Insight

Errors on first attempt to deploy an application built using S2I, as shown by the previous listing, are common. They happen because the deployment resource is created together with the build resource, and so the deployment tries to use an image that does not exist yet, and has to wait for the build to complete.

The **template** attribute is the wrapped **ReplicationController**, which in turn wraps a **Pod** resource definition. The **Replicas** attribute specifies how many pods are expected to be running.

Following the **DeploymentConfig** attributes, **oc describe dc** shows each deployment triggered by the **DeploymentConfig**. All pods currently running (and their respective **ReplicationController**) were created by a specific deployment, so it is the deployment resource that tells how many pods are actually running, if there are some pods waiting to be started, or if there are failed pods that could not start for any reason.

Insight

Some developers may prefer seeing the complete resource definitions shown by **oc get -o json** instead of the summary from **oc describe**.

Adjusting the number of replicas

OSE and Kubernetes refer to **scaling** as the action of changing the number of replicas in a **ReplicationController**. An application can be **scaled up** by increasing the number of replicas and **scaled down** by decreasing that number. The rc responds to scaling by either creating more pods or destroying some of the currently running pods.

An external application can monitor the current load and scale the application up and down according to some predefined policy, thus implementing the common elastic cloud application pattern.

Note

Autoscaling is a feature expected for a future OpenShift Enterprise 3.x release.

The web console and **oc new-app** command create **ReplicationController** resources with a single replica, that is, with the **replicas** attribute assigned to one. While it is possible to change this value directly on the **DeploymentConfig** resource using the **oc edit** command, the preferred way to scale an application is by using the **oc scale** command on the **ReplicationController** resource.

Comparison

In **OSE 2**, an application had to be defined as **scalable** during its creation to be able to scale up and down. If an OSE 2 application was created without this, it had to be recreated from scratch as an scalable application before it was possible to load-balance requests to multiple instances of the same application. But in **OSE 3**, all applications created from the web console or from **oc new-app** command are scalable, because all of them have a **DeploymentConfig** that creates a **ReplicationController**.

For example, to scale application **myapp** to have three replicas, use the following command:

```
$ oc scale --replicas=3 dc myapp
```

Usually the **oc scale** command targets the application dc resource, but it can target any rc resource instead. This is useful when having multiple rc active for the same application, like in a rolling upgrades scenario.

High availability (HA) for containerized applications

The fact that OpenShift Enterprise allows any application to run as a scalable application, creating multiple replicas of the application pods, **does not** mean any application automatically becomes scalable and fault-tolerant because it is running containerized inside OSE. The application itself has to be able to work correctly with multiple instances of itself.

Traditional web applications manage **user state** using some kind of HTTP session abstraction that is managed by the web application runtime or framework, usually based on HTTP cookies. More modern Web 2.0 applications are designed to be **stateless**, meaning there is no HTTP session abstraction in the server side. The user state is either managed by the web browser, using JavaScript objects, or retrieved from a database for each request, using an HTTP cookie or other client ID kept by the web browser. Most microservices fall in the same approach as Web 2.0 applications. The use of lightweight NoSQL databases or in-memory data stores makes those approaches much more palatable than for traditional web applications that assumed the only viable data store was a heavyweight relational database.

Clustering and HA are the same, whatever the approach used by an application or web framework, with or without OpenShift Enterprise. A developer has to check what are the application requirements for multiple instances to work, and if the container image implements the necessary requirements. OSE provides only the load balancer and the ability to restart failed application pods.

Each application layer probably has different requirements and implementation, and because of that, OSE created distinct services and replication controllers for each layer. For example, a web application can be stateless, and scaled to multiple replicas, but the relational database layer is configured with a single replica. Another option is having the database layer configured to perform master-slave replication, which probably means a service and replication controller for the master, and another for the slaves, where only the slaves get more than one replica, and the application connects to the slave database service for queries, but to the master database service for updates. Of course, nothing prevents the developer and the database images to employ more sophisticated strategies like sharding.

Clustering the Bookstore Application

Objectives

After completing this section, students should be able to deploy the bookstore application in a cluster.

Enabling clustering in EAP within a container

OpenShift Enterprise (OSE) can run any application as a scalable application. That means OSE will create multiple pods (replicas) for the application and load-balance requests among them, but the application itself (or its base image) has to provide for having multiple instances running concurrently without either conflicts or concurrency issues.

The JBoss EAP 6 images provided by Red Hat as part of the subscriber private registry are enabled for clustering out of the box. A Java developer can use them as provided, with no further changes or customizations.

Insight

The EAP container images come preconfigured and ready for clustering using the Infinispan and JGroups subsystem from a standard EAP installation. But they ship with a distinct JGroups configuration that interacts with OSE to find cluster members without needing either UDP multicast or a TCP ping initial node list. They find cluster members by querying the OSE service to get all pod IP addresses.

As per the Java EE standards, a clustered Java EE application needs to include the **`<distributable>`** element in its **`web.xml`** deployment descriptor. If this element is present, JBoss EAP replicates HTTP session objects among all cluster members, that is, across all pods from the same service if running under OSE.

A clustered Java EE application is also expected to:

- Have all its session attributes as **`Serializable`** objects.
- Not put too many objects as session attributes.
- Not use excessively large objects as session attributes
- Not saving any user state outside session attributes; for example, not using **`static`** instance attributes or singletons.

Not following those best practices may mean the application works, but not with the expected performance and scalability.

Demonstration: Verifying the application is clustered

In this example, please follow along with these steps while your instructor demonstrates how to verify a Java EE application is clustered and working inside an OSE instance.

1. Clone the **`session-replication`** application from the classroom Git server. Replace *X* by your student number.

```
$ cd /home/student
$ git clone http://workstation.podX.example.com/session-replication.git
```

This creates a folder named **session-replication** in the **student**'s home directory. Enter this folder to continue.

2. Inspect the application web deployment descriptor in file **src/main/webapp/WEB-INF/web.xml**. It should contain the **<distributable/>** element right after the **<webapp>** element.

3. Inspect the application welcome page in file **src/main/webapp/index.jsp**.

   It displays a simple counter that stores the current value as an HTTP session attribute. It also shows:
   - The session ID cookie value, before the current counter value.
   - The server host name, obtained from **jboss.host.name** Java **SystemProperty**, after the current counter value.

   Those values allows checking which container served the request, and which is the HTTP session maintained by JBoss EAP on behalf of the application.

4. The easiest way to deploy this application to OpenShift is by using the **eap6-basic-sti** template.

   Using this template from the command line is a three-step procedure, implemented by the shell script in file **process-template-sessrep.sh** at folder **/home/student/DO290/labs/clustering-demo**.

   Review this script, but do NOT execute it: the generated resource definition file is already available in the same folder and it will be used in **step 7**.

5. Log into OSE as user **student** and create a new project for this demo.

```
$ oc login -u student -p redhat
$ oc new-project sessrep
```

6. Edit the **/home/student/DO290/labs/clustering-demo/processed-template-sessrep.json** to change the X from all the occurrences of **workstation.podX.example.com** and **cloudappsX.example.com** in this file to the student's number.

7. Run the provided resource definition file to create the application.

```
$ oc create -f \
      /home/student/DO290/labs/clustering-demo/processed-template-sessrep.json
```

8. Wait until the build starts. Use the following command to check the build status:

```
$ watch oc status
```

9. To check the build status, use the following command:

```
$ oc build-logs sessrep-1
```

10. Open the application in a web browser. Use the following command to get the route host name for the URL.

```
$ oc get route
```

The URL is **`http://sessrep.cloudappsX.example.com`**. Replace *X* with the student's number.

The following figure shows what the application looks like.

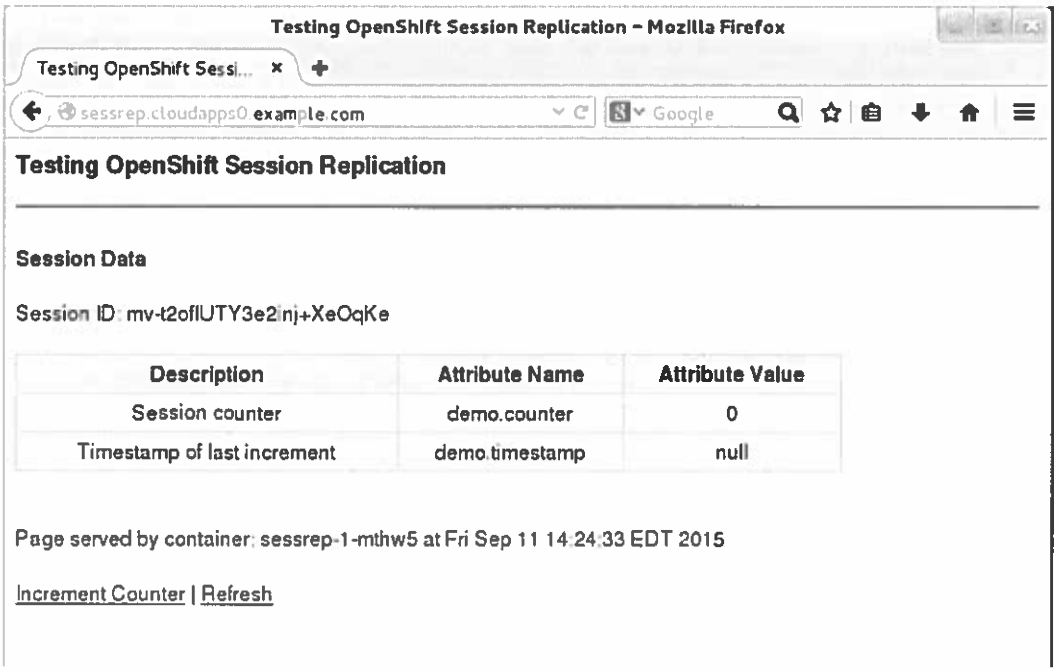


Figure 7.2: Session-replication application

11. Click the **`Increment Counter`** link several times and notice the value of attribute **`Session counter`** increments by one each time. Take note also of the container name displayed. In the figure, it is ***`sessrep-1-mthw5`***.

12. Inspect the deployment configuration created for the application. Notice it specifies a single (one) replica for both current and desired.

```
$ oc describe dc sessrep
```

13. Scale the application to have two pods.

```
$ oc scale --replicas=2 dc sessrep
```

The change to the deployment configuration should be propagated to the current deployment replication controller and after a few seconds, a second application pod should be created. Use the following command to check the new pod is running.

```
$ oc get pod
```

14. Check the logs from one of the pods, looking for a "New cluster view" entry.

```
$ oc logs sessrep-1-mthw5 | grep "New cluster view"
```

The expected output is similar to:

```
14:18:04,979 INFO  [org.jboss.as.clustering] (Incoming-2,shared=tcp) JBAS010225:
 New cluster view for partition web (id: 1, delta: 1, merge: true) : [68c0e631-
b752-7cb0-6be2-7327213afd5e/web, dbc88f2d-ce53-3e91-9993-baa5623fdcea/web]
```

It should show two cluster member UUIDs, proving both pods are replicating HTTP sessions to one another.

You may also get a message about a MERGED view instead of a New cluster view, for example:

```
$ oc logs sessrep-1-mthw5 | grep "MERGED cluster view"
13:11:10,207 INFO [org.infinispan.remoting.transport.jgroups.JGroupsTransport]
 (Incoming-2,shared=tcp) ISPN000093: Received new, MERGED cluster view: MergeView::
[482eda46-e4d5-7345-6873-44d05f94b1f5/web|1] [482eda46-e4d5-7345-6873-44d05f94b1f5/
web, 1931ba94-0beb-c126-0414-6531992fb018/web], subgroups=[482eda46-
e4d5-7345-6873-44d05f94b1f5/web|0] [482eda46-e4d5-7345-6873-44d05f94b1f5/web],
 [1931ba94-0beb-c126-0414-6531992fb018/web|0] [1931ba94-0beb-c126-0414-6531992fb018/
web
```

Make sure you there are two distinct UUIDs on the message.

Note

It may take a few seconds until the new pod joins the EAP cluster.

15. Back in the browser, increment the counter a few more times.

Check the name of the container that handles each request. It should stay the same, proving the OSE router implements sticky sessions correctly.

16. Delete the pod handling all requests. It is the same as the container name displayed by the application.

```
$ oc delete pod sessrep-1-mthw5
```

17. Return to the browser and increment the counter again.

The session ID value shown by the application should stay the same, and the counter should increment as expected. But the container name should change, proving the HTTP session failed over to the newer pod.

Note

If you do not see the container name change, repeat this step until it changes. Destroying a pod may take some time, and while this is being processed EAP may still reply to requests.

18. In a short amount of time, the OSE instance should note there is a single application pod, but the replication controller expects having two pods. The replication controller will create a replacement pod. Verify this happened using the following command.

```
$ oc get pod
```

19. Clean up: delete the demo project.

```
$ oc delete project sessrep
```

This concludes the demonstration.

Guided Exercise: Clustering the Bookstore Application

In this lab, you will change the bookstore application to run as a distributed Java EE application and test if it can survive pod failures without losing state.

Resources	
Files	**/home/student/DO290/labs/clustering-bookstore**
Application URL	NA

Outcome(s)
Bookstore application running with multiple pods that form an EAP web cluster.

Before you begin
OpenShift Enterprise (OSE) instance installed with router and registry, developer user (openshift) with access to the OSE instance and access to the classroom Git server.

1. Deploy The Nonclustered Bookstore Application To Initialize The Database
   1.1. Log into OSE as the developer user **student** and create a new project for this lab.

```
[student@workstation ~]$ oc login -u student -p redhat
[student@workstation ~]$ oc new-project bookstore
```

   1.2. The easiest way to deploy the bookstore application to OpenShift is by using the **eap6-mysql-sti** template. It already includes a MySQL database pod, but using ephemeral storage. This template also needs a **Secret** in the project to provide SSL certificates for HTTPS.

   Create the secret from the provided JSON file:

```
[student@workstation ~]$ oc create -f \
      /home/student/DO290/labs/clustering-bookstore/secret.json
```

   1.3. Creating a Java EE application from a template using the command line is a three-step procedure. Review the commands for each step in the shell script at **/home/student/DO290/labs/clustering-bookstore/process-template-bookstore.sh** but do NOT execute the script: the generated resource definition file is already available in the same folder and it will be used during the next step.

   1.4. Copy the provided JSON resource definition file and open using your favorite text editor.

```
[student@workstation ~]$ cp \
      /home/student/DO290/labs/clustering-bookstore/processed-template-
bookstore.json .
[student@workstation ~]$ gedit processed-template-bookstore.json
```

   Change all instances of **pod*X*** and **cloudapps*X*** to use your student number and save the file.

1.5. Create the application using the edited resource definition file.

```
[student@workstation ~]$ oc create -f \
    processed-template-bookstore.json
```

1.6. Wait until the build completes and there are two pods running: one for the bookstore application, and the other for the MySQL database server. Using the **oc get pods** command, the expected output is similar to:

```
NAME                      READY     STATUS      RESTARTS   AGE
bookstore-1-build         0/1       ExitCode:0  0          9m
bookstore-1-te62l         1/1       Running     0          52s
bookstore-mysql-1-dw85k   1/1       Running     0          9m
```

1.7. Wait until EAP startup finishes. Check the application pod logs using the following command:

```
[student@workstation ~]$ oc logs bookstore-1-te62l | egrep "JBoss EAP .*
 started"
```

The expected output is similar to:

```
15:10:17,957 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874: JBoss
 EAP 6.4.1.GA (AS 7.5.1.Final-redhat-3) started in 144615ms - Started 452 of 527
 services (132 services are lazy, passive or on-demand)
```

Note

The bookstore application initialization may take a few minutes and this impacts the EAP start up time.

2. Change The Bookstore Application To Be Distributed

2.1. Clone the bookstore application and enter the local source repository folder. Replace *X* with your student number.

```
[student@workstation ~]$ git clone http://workstation.podX.example.com/
bookstore-cluster.git
[student@workstation ~]$ cd bookstore-cluster
```

2.2. Edit the web deployment descriptor **web.xml** at **src/main/webapp/WEB-INF** and add the **<distributable/>** element right after the **<display-name>** element. The web deployment descriptor should end as in the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/
ns/javaee"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/
ns/javaee/web-app_3_0.xsd"
 version="3.0">
```

```
  <display-name>Bookstore</display-name>

  <distributable/>

  <session-config>
   <session-timeout>30</session-timeout>
  </session-config>
...
```

2.3. The bookstore application is configured to initialize a clean database at each start, but this will NOT work for a clustered application: later pods will erase all data inserted or updated by previous pods. You may end up with a corrupted database schema.

Edit the JPA configuration file **persistence.xml** at **src/main/resources/META-INF** and change the property with value **"create-drop"** to **"none"**. This way the application will expect the database to be already initialized, and this was done by the first deployment. The JPA configuration file should end up as in the following listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
 xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/
xml/ns/persistence/persistence_2_0.xsd">
 <persistence-unit name="bookstore" transaction-type="JTA">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <jta-data-source>java:/jboss/datasources/mysql</jta-data-source>
  <properties>
   <property name="hibernate.dialect"
 value="org.hibernate.dialect.MySQLDialect" />
   <property name="hibernate.hbm2ddl.auto" value="none" />
   <property name="hibernate.show_sql" value="false" />
   <property name="hibernate.format_sql" value="true" />
  </properties>
 </persistence-unit>
 </persistence>
```

Note

Also make sure the **hibernate.show_sql** property has value **false** as this speeds up EAP startup and the application itself considerably.

2.4. Changing the Hibernate configuration on the previous step prevented application tables to be re-created, but we also need to prevent test data from being inserted. Edit the properties file **runtime.properties** at **src/main/resources** so the only entry is set to false:

```
preloaddata=false
```

2.5. Commit local changes and push them to the remote Git server:

```
[student@workstation bookstore-cluster]$ git commit -a -m "made the application
 distributable"
```

```
[student@workstation bookstore-cluster]$ git push -u origin master
```

2.6. Start a new application build.

```
[student@workstation bookstore-cluster]$ oc start-build bookstore
```

Note

The build process may take a long time. Please be patient. You may want to follow the build using the **oc build-logs** command.

2.7. Wait until the new build completes and there is a new application pod running. The database pod should not have been affected. Using the **oc get pods** command, the expected output is similar to:

```
NAME                      READY     STATUS      RESTARTS   AGE
bookstore-1-build         0/1       ExitCode:0  0          24m
bookstore-2-build         0/1       ExitCode:0  0          8m
bookstore-2-lugs9         1/1       Running     0          3m
bookstore-mysql-1-dw85k   1/1       Running     0          24m
```

2.8. Wait until the new EAP pod startup finished. Use the instructions from **step 1.7** but remember to use the new pod name.

3. Identify The Pod Serving Requests

3.1. Get the route host name to access the bookstore application.

```
[student@workstation bookstore-cluster]$ oc get route
```

The expected output is:

```
NAME                    HOST/PORT                                PATH      SERVICE
        LABELS                                                TLS TERMINATION
bookstore-http-route    bookstore.cloudappsX.example.com                   bookstore
        application=bookstore,template=eap6-mysql-sti
bookstore-https-route   bookstore.cloudappsX.example.com                   secure-
bookstore   application=bookstore,template=eap6-mysql-sti   passthrough
```

3.2. Open the Firefox web browser and type the application host name for route **bookstore-http-route**. The URL should be as follows. Replace *X* by your student number:

http://bookstore.cloudappsX.example.com

The expected result is the bookstore application welcome page: The Shoppe.

3.3. List the application cookies to find the pod that is handling the user session.

The OSE router adds a cookie named **OPENSHIFT_*route_name*_SERVERID** to each request. That cookie value is the destination pod IP address.

The Firefox web browser included in RHEL7 has the web developer toolbar, accessed by pressing **Shift+F2**. This toggles a command prompt at the bottom of the browser window. Type there the **cookie list** command to get a pop-up and take note of the **OPENSHIFT_*route_name*_SERVERID** and the **JSESSIONID** cookie values. The following figure illustrates this step:

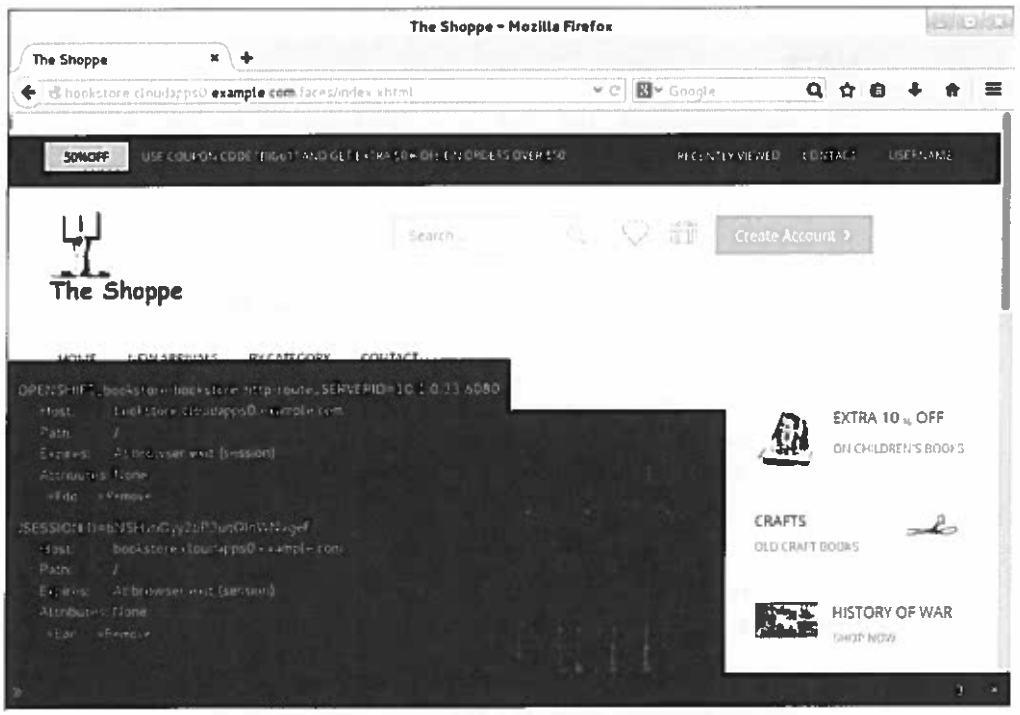


Figure 7.3: Firefox web developer toolbar listing cookies

3.4. Verify the application pod has the IP obtained from the previous step.

Use the **oc get pods** command to find the application pod name and then **oc describe** on this pod to get its internal IP address. It should be the same as shown by the cookie list on Firefox, as we still have a single application pod.

```
[student@workstation bookstore-cluster]$ oc describe pod bookstore-2-lugs9 |
 grep IP
IP:    10.1.0.33
```

Take note of the pod name; it will be needed during the next steps.

Note

The pod name was already displayed in **step 2.6**; when it was checked that a new application pod was started.

4. Test Application Fail Over
   4.1. Scale the application to have a second pod.

```
[student@workstation bookstore-cluster]$ oc scale --replicas=2 dc bookstore
```

Wait until the second pod is running before proceeding to the next step. Use the **oc get pod** command. The following listing shows the expected result, with two pods running:

```
NAME                     READY     STATUS      RESTARTS   AGE
bookstore-1-build        0/1       ExitCode:0  0          52m
bookstore-2-build        0/1       ExitCode:0  0          36m
bookstore-2-lugs9        1/1       Running     0          31m
bookstore-2-oyxe0        1/1       Running     0          1m
bookstore-mysql-1-dw85k  1/1       Running     0          52m
```

4.2. Wait for the second pod EAP instance to finish start up. Use instructions from **step 1.7** but using the second pod name.

4.3. Wait for both pods to find each other and form a cluster. Search for "cluster view" messages in both, using the **oc logs** command, for example:

```
[student@workstation bookstore-cluster]$ oc logs bookstore-2-lugs9 | grep
 "cluster view"
```

Check the output for either a **new cluster view** message similar to:

```
14:18:04,979 INFO  [org.jboss.as.clustering] (Incoming-2,shared=tcp) JBAS010225:
 New cluster view for partition web (id: 1, delta: 1, merge: true) : [68c0e631-
b752-7cb0-6be2-7327213afd5e/web, dbc88f2d-ce53-3e91-9993-baa5623fdcea/web]
```

Or for a **MERGED cluster view** similar to:

```
13:11:10,207 INFO [org.infinispan.remoting.transport.jgroups.JGroupsTransport]
 (Incoming-2,shared=tcp) ISPN000093: Received new, MERGED cluster view:
 MergeView::[482eda46-e4d5-7345-6873-44d05f94b1f5/web|1] [482eda46-
e4d5-7345-6873-44d05f94b1f5/web, 1931ba94-0beb-c126-0414-6531992fb018/
web], subgroups=[482eda46-e4d5-7345-6873-44d05f94b1f5/web|0] [482eda46-
e4d5-7345-6873-44d05f94b1f5/web], [1931ba94-0beb-c126-0414-6531992fb018/web|0]
 [1931ba94-0beb-c126-0414-6531992fb018/web
```

You may find more than one message from both kinds. Consider only the latest one. It should display TWO distinct UUID values. If not, wait for a few moments and check the logs again. Repeat until you are sure a cluster is formed.

Important

It may take a few seconds, after the second application pod is running, to join the EAP cluster. Be patient.

4.4. Add a book to the shopping cart, so the user session has state to be preserved.

Return to the browser window, and hover the pointer over BY CATEGORY to show a menu. Click Comics in the menu, below the SHOP heading.

Click a book cover to get the book details and then click **Buy**. After that, the application page should be similar to the following figure:

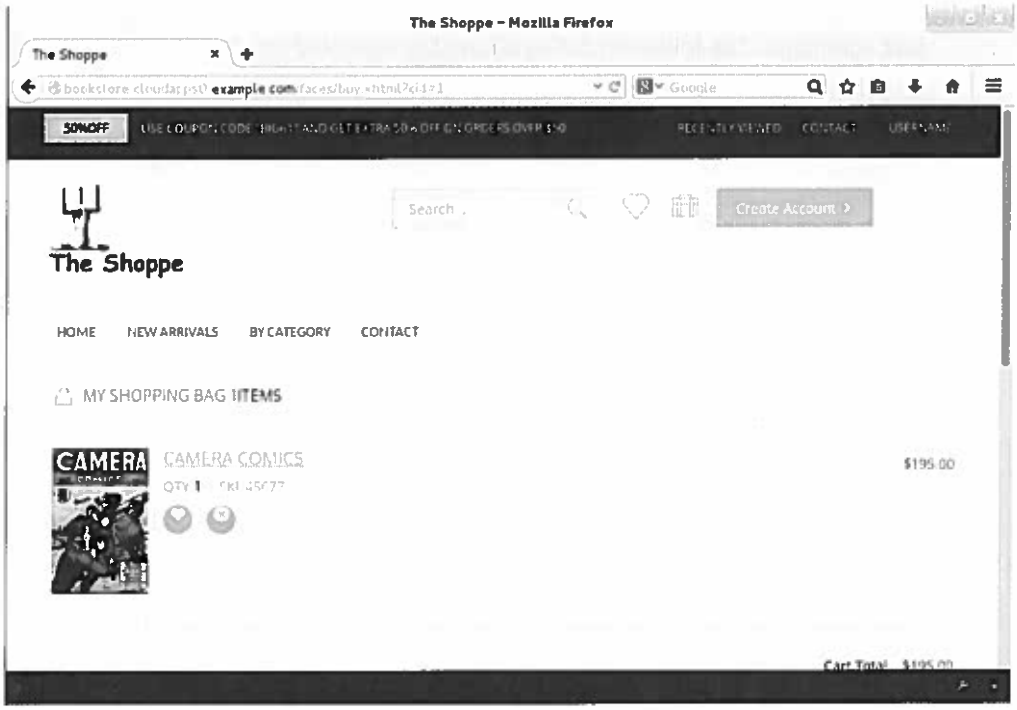


Figure 7.4: Shopping cart with one comic

Note

If you get an error claiming the session expired, simply click the **The Shoppe** link at the upper-left corner and start over. This may happen because of the time starting an EAP pod takes in the classroom. If this happens, check again the cookies using instructions from **step 3.3** but most of the time the application will still be served by the same pod (the firs one) and use the same session cookie.

4.5. Delete the first pod, or whatever pod is serving the web browser.

```
[student@workstation bookstore-cluster]$ oc delete pod bookstore-2-lugs9
```

4.6. Wait until the pod is deleted. Use the command **oc get pods** and wait until the deleted pod disappears from the output. Then in a few moments OSE will create a replacement pod. This can also be verified with the **oc get pods** commands.

Note

When the EAP pod is destroyed, it will perform an orderly shutdown and may take some time. Not to mention the Bookstore app perform Ajax calls and EAP will handle those before shutting down.

But sometimes deleting the EAP pod can go very quickly and the replacement pod also start very soon. Beware you do not confuse the new replacement pod with the old one you just deleted.

4.7. Verify the user session is now handled by the new pod.

Return to the browser window and click the **Home** link to go back to the application welcome page. Use the Firefox web developer toolbox to get the cookie list, as in **step 3.3**. The following figure illustrates the expected output:

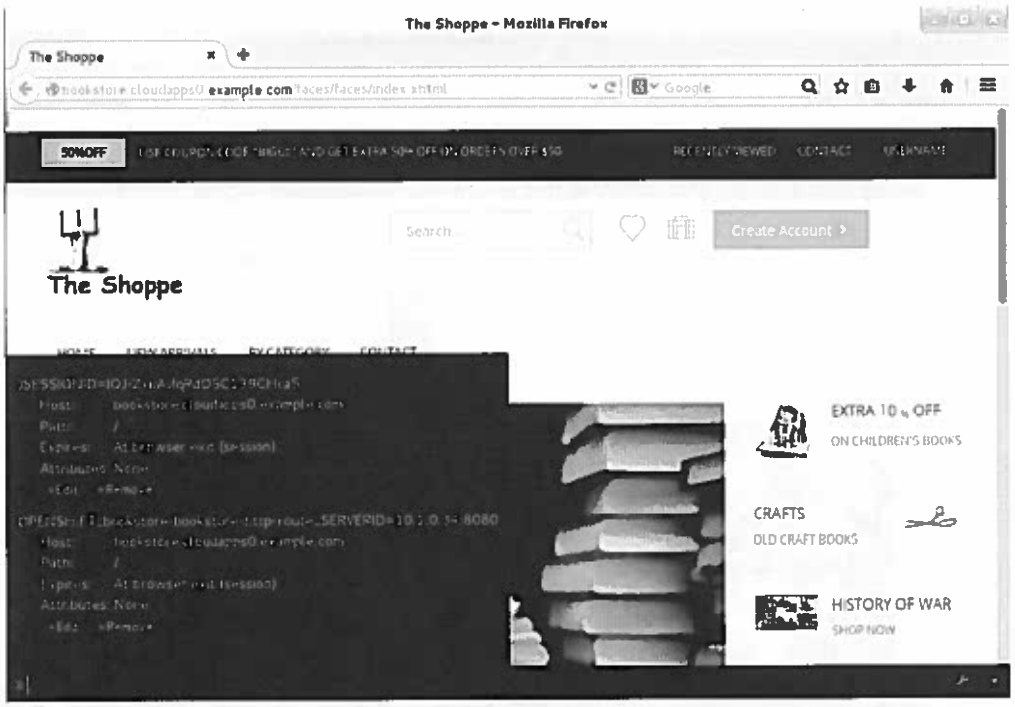


Figure 7.5: Cookies show this request was served by another pod

Note

The first time a pod runs the Bookstore application it may take some time to reply to the first click. This happens because JSF does some initialization on-demand and this impacts the firs access.

4.8. Verify the IP address of the server that handled the last request changed to the IP address of the new pod, created in **step 4.2**. To do that, use the instructions from **step 3.4**. The following listing illustrates the expected output:

```
[student@workstation bookstore-cluster]$ oc get pod
NAME                    READY     STATUS      RESTARTS   AGE
bookstore-1-build       0/1       ExitCode:0  0          1h
bookstore-2-build       0/1       ExitCode:0  0          45m
bookstore-2-ekuy0       1/1       Running     0          7m
bookstore-2-oyxe0       1/1       Running     0          10m
bookstore-mysql-1-dw85k 1/1       Running     0          1h
[student@workstation bookstore-cluster]$  oc describe pod bookstore-2-oyxe0 |
 grep IP
IP:    10.1.0.34
```

4.9. Verify the shopping cart still has the book added in step 4.1.

Hover the mouse over the bag icon and verify the shopping cart shows the correct book cover, as in the following figure:

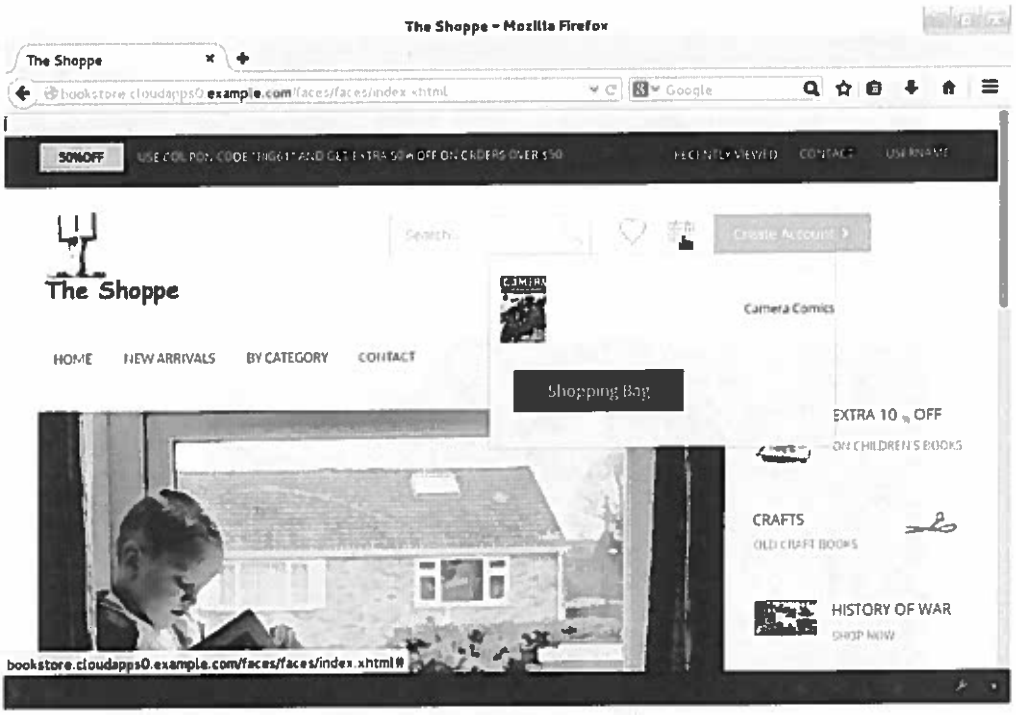


Figure 7.6: Proof the shopping cart contents were NOT lost on fail over

This proves the user session was not lost when the pod serving the user was deleted.

5. Clean Up
Delete the bookstore project.

```
[student@workstation bookstore-cluster]$ oc delete project bookstore
```

This concludes the exercise.

Configuring the OSE Router

Objective

After completing this section, students should be able to configure the OSE router for high availability (HA).

Configuring the OSE router for HA

All external accesses to an OSE application go through the **OSE router** that is based on **HAProxy**. The application's externally visible host name (that is, the application route host name) must resolve to the public IP of the node that runs the OSE router. Controlling which node runs the router can be done, as for any other OSE application, by using node labels and deployment config selectors matching those labels. More details on pod placement into nodes are shown later in this chapter, in the section about scheduling.

If the OSE router is not running, no application can be accessed, even if their nodes and pods are fine. So the OSE instance administrator needs to configure a high availability (HA) router service if there are any applications expecting running with HA.

The OSE router is stateless, so it would be just a matter of running many replicas and configuring all selected node IP addresses in a round-robin DNS setup, that is, multiple A records for the same wild-card domain. It looks very simple, but not enough. If a node becomes unavailable, a plain round-robins DNS will still direct traffic to that node's IP address.

This issue does not happen with normal application pods because they are not exposed directly using the node IP address. Application pods can be destroyed and recreated in any available node because other pods access them through the service IP, and external users access the application through the router IP.

To solve this router issue, OSE provides an **IP fail-over service**, based in **Keepalived**. This service manages a set of virtual IP addresses (VIP) assigned to the router service. The VIP addresses are assigned to the round-robin DNS wild-card domain. The number or router replicas has to be smaller than the number of nodes that match the router selector, because if a node fails, there needs to be another node available to schedule the failed router pod and assign the VIP of the failed node.

Using the OSE IP fail-over service, all VIPs assigned to the router will always be assigned to a node running a working router pod.

To deploy an HA router, follow these steps:

1. Configure DNS to resolve the wild-card domain to the VIP addresses reserved for the router.
2. Label the nodes that will run the OSE router. For example:

```
# oc label nodes openshift-node-{1,2,3} router=cloudapps
```

3. Create the router specifying a `--selector` option that matches the label from the previous step and the desired number of replicas for load balancing and high availability.

```
# oadm router cloudapps --replicas=2 --selector=router=cloudapps
```

4. Create the IP fail-over service using the **oadm** command and specifying the same selector as the router, but with a number of replica VIP addresses equal to the number of nodes that matches the selector.

```
# oadm ipfailover cloudapps --replicas=3 --selector=router=cloudapps \
--virtual-ips="10.245.2.101-103" \
--watch-port=80
```

The previous example commands create a router service named **cloudapps** and an IP fail-over service also named **cloudapps** (names do not need to match). Three nodes are assigned to run two replicas of the router pod, so three VIP addresses are configured.

The IP fail-over service will assign VIP addresses only to nodes that run a router, so there will be nodes with no VIP address that run an **ipfailover** pod, and nodes that have possibly more than one VIP address that run both a **cloudapps** router pod and an **ipfailover** pod.

This example assumes the OSE instance administrator expects at most one node to fail, and the physical datacenter setup should back that expectation; for example, having the nodes on distinct sites, or at least on different racks connected to distinct power supplies and backbone switches.

Quiz: Configuring the OSE Router

Match the following items to their counterparts in the table.

HAProxy	Keepalived	VIPs	applications	nodes	router

Description	Name
Open source project that provides the OSE router service implementation	
Open source project that provides the OSE IP fail-over service implementation	
The number of replicas for the router service should match the number of	
The selector in both the router and IP fail-over service definitions should match the labels on	
The OSE IP fail-over service is necessary to provide HA for	
The OSE router service helps providing HA for	

Solution

Match the following items to their counterparts in the table.

Description	Name
Open source project that provides the OSE router service implementation	HAProxy
Open source project that provides the OSE IP fail-over service implementation	Keepalived
The number of replicas for the router service should match the number of	VIPs
The selector in both the router and IP fail-over service definitions should match the labels on	nodes
The OSE IP fail-over service is necessary to provide HA for	router
The OSE router service helps providing HA for	applications

Controlling Pod Scheduling

Objective

After completing this section, students should be able to use labels and selectors to control to which nodes pods are deployed on.

Default scheduler algorithm

The scheduler is essentially the OpenShift master. When a pod needs to be created, the master must figure out which node is the best to create the pod. This is called scheduling. The default scheduling algorithm selects a node to host the pod in a three-step operation:

1. Filter the nodes.

   The current OSE instance node list is filtered by running each of the nodes through the list of filter functions called **predicates**. The end result is a (probably shorter) list of node candidates to run the pod.

2. Prioritize the filtered list of nodes.

   Each node in the candidate list passes through a series of **priority** functions that assign it a score between 0 and 10, with higher values indicating the node is a better fit to host the pod. The score provided by each priority function is multiplied by a weight assigned to the function and then all scores are added.

3. Select the best fit node.

   The candidate list is sorted based on the scores and the node with the highest score is selected to host the pod. If multiple nodes have the same high score, then one of them is selected at random.

Insight

If the predicate functions collectively filter out all nodes, leaving as the result from the first step an empty candidate node list, the pod will be left in the **pending** state. The pod creation will NOT fail. The pod is created, but is left waiting for a new node to be added to the OSE instance that may match all the scheduling filters.

The OSE master configuration file **`/etc/openshift/master/master-config.yaml`** defines the scheduler configuration file, which by default is **`/etc/openshift/master/scheduler.json`**. This file provides the ordered list of predicates and priority functions to be used by the scheduler when selecting the node to run a new pod. The following listing shows the default scheduler configuration file created by the OSE installer:

```
{
  "predicates": [
    {"name": "MatchNodeSelector"},
    {"name": "PodFitsResources"},
    {"name": "PodFitsPorts"},
    {"name": "NoDiskConflict"},
    {"name": "Region", "argument": {"serviceAffinity" : {"labels" : ["region"]}}}
  ],"priorities": [
```

```
    {"name": "LeastRequestedPriority", "weight": 1},
    {"name": "ServiceSpreadingPriority", "weight": 1},
    {"name": "Zone", "weight" : 2, "argument": {"serviceAntiAffinity" : {"label":
 "zone"}}}
  ]
}
```

Each predicate function is specified by a **name** and an optional **argument** attribute, while each priority function also takes a **weight** attribute. The **argument** attribute distinguishes between two kinds of functions:

- Static functions are identified by the **name** attribute and take no arguments.
- Dynamic functions are identified by the first key on the **argument** attribute. That key value refers to the pod metadata attributes, which are used as arguments to the function.

Note

The full set of predicate and priority functions is describe in the OSE documentation.

To read the previous listing, first consider that the candidate node list starts with the full set of nodes currently known by the OSE master. The default scheduler configuration then filters nodes so the only remaining nodes are nodes where all the following conditions are true:

1. Match the pod node selector.

   The pod node selector can be empty so all nodes match.

2. Have enough resources to run the pod.

   If neither the pod nor its project defines default resource limits, all nodes are left in the list.

3. Have the requested host port available.

   A privileged pod definition may request a host TCP or UDP port. This is how the OSE router pod works.

4. Have the requested host volume available.

   A privileged pod definition may request direct access to the host node file system. This is NOT recommended for most production environments.

5. Has a matching **region** label.

Then the resulting candidate node list is subjected by the default scheduler configuration to the following priority scores:

1. Nodes with fewer pods running score higher.
2. Nodes that do NOT host pods from the same service score higher.
3. Nodes with different **zone** labels score higher.

OpenShift resource labels -- more specifically pod and node labels -- play a crucial role in the default scheduler configuration. The usage is similar to a service resource usage of a selector to match pod labels and thus find the pods to be load-balanced by the service.

Node selectors and labels

The **NodeSelector** attribute is a part of a pod definition. It matches labels from node resources. This way, an administrator can label nodes to implement any desired topology, and a developer can configure pods to run only on nodes positioned in a desired way inside that topology.

It is possible to get the labels on nodes using the **oc get nodes** command:

```
$ oc get nodes
```

The expected result is:

```
NAME                          LABELS                                          STATUS
master.podX.example.com       kubernetes.io/hostname=master.pod0.example.com
 Ready,SchedulingDisabled
node.podX.example.com         kubernetes.io/hostname=node.pod0.example.com    Ready
```

The only label assigned by the OSE installer is the **kubernetes.io/hostname** label, whose value is the node's FQDN. This allow precise control over which node runs a pod, but usually a developer does not want or need such precise control.

Unschedulable nodes

A node resource definition includes the **unschedulable** attribute. Setting this attribute to true makes sure no pod will be scheduled to the node. It is left out of the candidate node list even before evaluating predicate functions. This attribute, if set as true, shows on the **oc get nodes** command output as the **SchedulingDisabled** status.

The OSE installer will, by default, configure the master a node in the OSE instance, but will also configure the master node to NOT receive any pod. That is the reason **oc get nodes** show the master node with **SchedulingDisabled** status.

Insight

The OSE master needs to be configured as a node because some OSE services, like the internal registry, the router, build pods, and deploy pods, need access to the OSE API and use the SDN for that.

To change a node's **unschedulable** attribute, use the **oc edit** command; for example:

```
$ oc edit node master.podX.example.com
```

Delete the following line:

```
unschedulable: true
```

After saving this node, the master is set to receive pods. This is usually done, alongside adding new labels to the master node, to make the master node run the OSE internal registry and router pods, leaving other nodes only for user application pods. The next section provides more details on how to accomplish this.

Using zones and regions

A common pattern for designing cloud topologies is using the zones and regions concepts:

- A **zone** represents a set of nodes that share a common infrastructure, such as rack, power source, network switch, or even an entire datacenter. It is expected nodes from the same zone may fail together.

- A **region** represents a set of nodes reserved for a specific usage, such as a single customer, a critical set of application services, or that provides some kind of specialized hardware. Usually there are nodes from the same region in different zones, so the specific use case represented by the region has higher resiliency to infrastructure failures represented by each zone.

The default OSE scheduler configuration file at **`/etc/openshift/master/scheduler.json`** was designed with the zones and regions concepts in mind. The following partial listing shows that a **zone** label is used as a **`serviceAffinity`** predicate function argument, and that a **region** label is used as a **`serviceAntiAffinity`** priority function: argument:

```
{
  "predicates": [
    ... other predicates omitted ...
    {"name": "Region", "argument": {"serviceAffinity" : {"labels" : ["region"]}}}
  ],"priorities": [
    ... other priorities omitted ...
    {"name": "Zone", "weight" : 2, "argument": {"serviceAntiAffinity" : {"label":
 "zone"}}}
  ]
}
```

The **`serviceAffinity`** predicate function makes pods from the same service run in the same region to reduce network latency and to account for the fact all pods from the same service serve the same use case. But the **`serviceAntiAffinity`** priority function tries to spread pods from the same service into distinct zones, so the service is not down because of a single zone outage.

Even if the scheduler already expects the use of regions and zones, the OSE installer does not assigns nodes to any region or zone. This can be corrected using the **`oc label`** command, for example:

```
$ oc label node master.paascloud.example.com region=infra zone=default
$ oc label node node2.paascloud.example.com region=primary zone=east
$ oc label node node3.paascloud.example.com region=primary zone=west
```

This example configures a three-node OpenShift instance divided up into two regions (**`infra`** and **`primary`**). The nodes are also divided into three regions (**`default`**, **`east`**, and **`west`**). The **`oc get nodes`** output would be:

```
NAME                              LABELS
                STATUS
master.paascloud.example.com    kubernetes.io/
hostname=master.pod0.example.com,region=infra,zone=default   Ready
node2.paascloud.example.com     kubernetes.io/
hostname=node.pod0.example.com,region=primary,zone=east      Ready
node2.paascloud.example.com     kubernetes.io/
hostname=node.pod0.example.com,region=primary,zone=west      Ready
```

The **infra** region in this example would be reserved for OSE infrastructure services such as the router and the internal registry. With a single node in this region, there would be no high availability for OSE itself. But the **primary** region, intended for running application pods, has two nodes and so applications can survive the failure of one of them.

Insight

Application pods continue to run in the event the OSE master is not available. It simply will not be possible to create new pods, scale applications, or perform other changes to the environment.

The two nodes from the **primary** region have to be in distinct zones, else all pods from the same service could be scheduled to the same node, and if this node fails, the entire application would be unavailable.

Edit the default node selector

It is possible to configure OpenShift to land user pods in a specific region by setting a **defaultNodeSelector** in the OSE master configuration file at **/etc/openshift/master/master-config.yaml**. Change the following:

```
projectConfig:
 defaultNodeSelector: "region=primary"
```

Note

After changing the master configuration file, restart the **openshift-master systemd** service unit. This will make both **defaultNodeSelector** and scheduling policy changes take effect:

```
$ systemctl restart openshift-master
```

Tweak the default project

The **default** project is a special one where the infrastructure-related resources will be put. This project does not have a node selector by default.

To add a node selector to the project and make OSE infrastructure pods for the internal registry and router run in the **infra** region, edit the project resource definition using the following command as an OSE instance administrator:

```
$ oc edit -o json namespace default
```

In the **annotations** object, add this line:

```
"openshift.io/node-selector": "region=infra"
```

Save and exit the editor. There is no need to restart the OSE master after this change.

Note

Other projects could also be configured to have a node selector that will be applied to all the project pods.

Another way to make the OSE services land in the **infra** region is adding a **--selector** option to the **oadm** command, for example:

```
$ oadm registry \
    --credentials=/etc/openshift/master/openshift-registry.kubeconfig \
    --images='registry.access.redhat.com/openshift3/ose-docker-registry:${version}' \
 --selector='region=infra'
```

Quiz: Controlling Pod Scheduling

Choose the correct answer to the following questions according to the cluster:

Name	Labels	Status
Master	• kubernetes.io/ hostname=master.example.com • region=infra • zone=default	Ready,SchedulingDisabled
Node1	• kubernetes.io/ hostname=node1.example.com • region=primary • zone=east	Ready
Node2	• kubernetes.io/ hostname=node2.example.com • region=primary • zone=south	Ready
Node3	• kubernetes.io/ hostname=node3.example.com • region=primary • zone=north	NotReady

1. If no labels are defined in the **`DeploymentConfig`**, which node(s) are available for the new pod?

   a. All nodes.
   b. Node1, Node2, and Node3.
   c. Node1, Node2
   d. No possible nodes to choose.

2. The following command is used to create a new internal registry:

```
$ oadm registry  --credentials=/etc/openshift/master/openshift-registry.kubeconfig \
    --images='registry.access.redhat.com/openshift3/ose-docker-registry:v3.0.0.1' \
 --selector='region=infra'
```

   Which node(s) are available for the new registry?

   a. Master.
   b. Node1, Node2, and Node3.
   c. All nodes.

d. No possible nodes to choose.

3. The following **DeploymentConfig** is used to create a new application:

```
{
    "kind": "DeploymentConfig",
    "apiVersion": "v1",
    "metadata": {
        "name": "version",
        "creationTimestamp": null,
        "labels": {
            "app": "version",
   "openshift.io/node-selector:" "region=primary"
        }
    },
... Output Omitted ...
```

Which node(s) are available for the new pod?

a. Node1 and Node2.
b. Node1, Node2, and Node3.
c. All nodes.
d. No possible nodes to choose.

Solution

Choose the correct answer to the following questions according to the cluster:

Name	Labels	Status
Master	• kubernetes.io/ hostname=master.example.com • region=infra • zone=default	Ready,SchedulingDisabled
Node1	• kubernetes.io/ hostname=node1.example.com • region=primary • zone=east	Ready
Node2	• kubernetes.io/ hostname=node2.example.com • region=primary • zone=south	Ready
Node3	• kubernetes.io/ hostname=node3.example.com • region=primary • zone=north	NotReady

1. If no labels are defined in the **`DeploymentConfig`**, which node(s) are available for the new pod?

   a. All nodes.
   b. Node1, Node2, and Node3.
   c. **Node1, Node2**
   d. No possible nodes to choose.

2. The following command is used to create a new internal registry:

```
$ oadm registry  --credentials=/etc/openshift/master/openshift-registry.kubeconfig \
    --images='registry.access.redhat.com/openshift3/ose-docker-registry:v3.0.0.1' \
 --selector='region=infra'
```

   Which node(s) are available for the new registry?

   a. Master.
   b. Node1, Node2, and Node3.
   c. All nodes.
   d. **No possible nodes to choose.**

3. The following **DeploymentConfig** is used to create a new application:

```
{
    "kind": "DeploymentConfig",
    "apiVersion": "v1",
    "metadata": {
        "name": "version",
        "creationTimestamp": null,
        "labels": {
            "app": "version",
   "openshift.io/node-selector:" "region=primary"
        }
    },
... Output Omitted ...
```

Which node(s) are available for the new pod?

a. **Node1 and Node2.**
b. Node1, Node2, and Node3.
c. All nodes.
d. No possible nodes to choose.

Summary

In this chapter, you learned:

- How to configure the number of replicas for a pod.
- OpenShift Enterprise (OSE) and applications roles to implement clustering and high availability (HA).
- How to configure a clustered Java EE application.
- How to configure HA for the OSE router.
- How to use labels and selectors to control pod placement into nodes.

CHAPTER 8

TROUBLESHOOTING APPLICATIONS

Overview	
Goal	Use debugging and logs to troubleshoot problems with application performance or deployment issues.
Objectives	• Configure and execute a remote debugging session with the containerized version of the bookstore application. • Locate and review OpenShift Enterprise logs to determine causes of issues with applications.
Sections	• Debugging the Bookstore Application (and Guided Exercise) • Reviewing OpenShift Enterprise Logs (and Guided Exercise)
Lab	• Troubleshooting an Application

Debugging the Bookstore Application

Objectives

After completing this section, students should be able to configure and execute a remote debugging session with the containerized version of the bookstore application.

Running the EAP 6 container in debug mode

Running JBoss EAP in an OpenShift Enterprise (OSE) container has several differences compared to using a regular instance of JBoss EAP:

- The JBoss EAP Management Console is not available to manage OSE JBoss EAP images.
- The JBoss EAP management CLI is only bound locally. This means that the management CLI can only be used from within the pod that hosts the EAP image.
- Domain mode is not supported because OpenShift controls the creation and distribution of applications in the containers.
- The default root page is disabled. A deployable can be set to the root context by setting the file name to **`ROOT.war`**.

Because the EAP image lacks the traditional Management Console that is normally available in EAP 6, it is recommended to manage and debug the server using the Management CLI. The Management Console is not available because the images are intended to be immutable. Any modification to the configuration of an EAP instance will be overwritten as soon as the pod is restarted. The CLI is exposed only for the sake of troubleshooting purposes and for debugging. To access the EAP Management CLI, execute the following command:

```
$ oc exec -p -it <pod_name> -- /opt/eap/bin/jboss-cli.sh
```

Note

Any configuration changes made using the JBoss EAP Management CLI on a running container will be lost when the container restarts.

In addition to viewing the OpenShift logs, troubleshooting a JBoss EAP container can be achieved by viewing the JBoss EAP logs that are written to the container's console:

```
$ oc logs -f <pod_name>
```

The Java Debug Wire Protocol (JDWP) is the protocol used for communication between a debugger and the Java virtual machine (JVM) that allows for a debugger tool (such as that found in JBoss Developer Studio) to connect to it remotely and step through the source code as it is being executed in real time. One common way to enable debug mode in EAP 6 is to edit the **`/opt/eap/bin/standalone.conf`** file by uncommenting the following line:

```
JAVA_OPTS="$JAVA_OPTS -
agentlib:jdwp=transport=dt_socket,address=8787,server=y,suspend=n"
```

If activated in EAP 6, JDWP will open the TCP port 8787 from the environment it is running on, according to the parameter **address** previously listed.

The OSE xPaaS images for JBoss EAP contains an out of the box mechanism to modify the **standalone.conf** file. An environment variable called **DEBUG** is available to activate the JDWP. This variable can be added to the **env** attribute of EAP pods, for example by changing the pod template inside the **DeploymentConfig** resource of any of the EAP templates:

```
{
    "kind": "Template",
    "apiVersion": "v1",
    "metadata": {
        "name": "eap6-basic-sti",
...
        {
            "kind": "DeploymentConfig",
...
                    "spec": {
                        "containers": [
                            {
...
                                "env": [
                                    {
                                        "name": "DEBUG",
                                        "value": "true"
                                    },
                                    {
                                        "name": "OPENSHIFT_DNS_PING_SERVICE_NAME",
                                        "value": "${APPLICATION_NAME}-ping"
                                    },
...
```

Defining a debugging session in JBoss Developer Studio

Remotely debugging an application within JBoss Developer Studio can be a quick way to diagnose problems within a specific environment, such as running in a pod. Now that JBoss EAP JVM is started (with JDWP listener enabled), the listener needs to be exposed so that it is accessible to clients (such as the JDWP debugger client in JBDS). The approach for making the debugger accessible is via OpenShift's port-forwarding capability. The end state is an network tunnel created between TCP port 8787 of the EAP pod and a local TCP port on our local workstation. To enable remote debugging in JBoss Developer Studio, follow these steps:

1. Set up a **port-forwarding** tunnel on the workstation machine to forward connections from a local unused TCP port to the 8787 port in the EAP pod. If there is no local JVM running in debug port the *localPort* value can be 8787.

```
$ oc port-forward -p pod_name -p <appName> localPort:8787
```

Note

Leave this terminal window running. Cancelling this process or closing the window will close the tunnel and stop the port forwarding.

2. In JBoss Developer Studio, click **Run** > **Debug Configurations...**.
3. Select **Remote Java Application** from the left window and click the **New Launch Configuration** button.
4. Set **project** to "bookstore," **host** to `127.0.0.1`, and port to *localPort*.
5. Click **Debug** and the debugger will connect to the EAP server on *localPort* and the debugger will begin running.

If any break points are set, JBoss Developer Studio will automatically switch to the **Debugging** perspective as soon as a break point is hit by the server.

Guided Exercise: Debugging the Bookstore Application

In this lab, you will configure the JBoss container image to start in debug mode and remotely debug the bookstore application.

Resources	
Files	**/home/student/DO290/labs/remote-debugging**
Application URL	NA

Outcomes
You should be able to:
- Run the JBoss EAP container in debug mode.
- Debug bookstore application remotely using JBDS.
- Troubleshoot a real failure in the bookstore application.

Before you begin
- Verify the OpenShift Enterprise Client is installed.

1. Create a Bookstore App in OpenShift Enterprise
   The Java Debug Wire Protocol (JDWP) is the protocol used for communication between a debugger and the Java virtual machine (JVM). Let's now use the OSE S2I workflow to start an EAP container (with JDWP enabled) that runs our **bookstore** application.

   1.1. Use the provided script on the workstation to create the persistent volume:

   ```
   [student@workstation ~]$  bash /home/student/DO290/labs/remote-debugging/
   createpv.sh
   ```

   1.2. At a terminal window of the workstation VM, run the following command to create a new project:

   ```
   [student@workstation ~]$ oc login -u student -p redhat
   [student@workstation ~]$ oc new-project debug-bookstore
   ```

   1.3. The template being used requires that you create a secret. Inspect and create the following secret:

   ```
   [student@workstation ~]$ oc create -f \
       /home/student/DO290/labs/remote-debugging/secret.json
   ```

   1.4. Navigate to: **/home/student/DO290/labs/remote-debugging**, open the file **bookstore.json** and replace every instance *X* in **http://workstation.pod*X*.example.com/bookstore.git** and **bookstore.cloudapps*X*.example.com**.

1.5. An environment variable is responsible for enabling the JDWP inside the JBoss Application Server. Edit the **/home/student/DO290/labs/remote-debugging/bookstore.json** and in **DeploymentConfig** section, add the DEBUG variable:

```
...OUPUT OMMITED...
"env": [
   {
      "name": "DEBUG",
      "value": "true"
   },
   {
      "name": "DB_SERVICE_PREFIX_MAPPING",
      "value": "bookstore-mysql=DB"
   },
   ...OUPUT OMMITED...
```

Insight

The **bookstore.json** was generated by processing the **eap6-mysql-persistent-sti** standard OSE template.

1.6. Execute the following to create a bookstore app managed by OSE:

```
[student@workstation ~]$ oc create -f \
  /home/student/DO290/labs/remote-debugging/bookstore.json
```

1.7. Monitor the build logs using the following command:

```
[student@workstation ~]$ oc build-logs bookstore-1
```

1.8. Once the build is complete, wait until the deployment begin:

```
[student@workstation ~]$ oc get pods

NAME                 READY  STATUS      RESTARTS  AGE
bookstore-1-build    0/1    ExitCode:0  0         5m
bookstore-1-deploy   1/1    Running     0         18s
bookstore-1-n56t1    0/1    Running     0         10s
```

1.9. Tail the application log file of your running bookstore container.

Toward the top of the application log file, note the following entry indicating that the JWDP listener is now enabled in the JVM:

```
[student@workstation ~]$ oc logs bookstore-1-n56t1
```

```
...OUTPUT OMMITED...
Listening for transport dt socket at address: 8787
...OUTPUT OMMITED...
```

2. Create The **`port-forward`** Tunnel
   2.1. Determine the name of the bookstore pod:

```
[student@workstation ~]$ oc get pod
bookstore-1-n56t1         1/1       Running     0          1h
```

   2.2. Using the pod name, create the port-forwarding tunnel:

```
[student@workstation ~]$ oc port-forward -p bookstore-1-n56t1 8787:8787
```

   2.3. Leave this terminal open and running in order to keep the tunnel open.

3. Clone locally the bookstore app
   Open a Terminal window and execute the following commands. Replace *X* with your student number:

```
[student@workstation ~]$ cd ~/DO290/labs
[student@workstation labs]$ git clone http://workstation.podX.example.com/
bookstore.git
```

4. Import the project to JBDS
   Open JBDS, and import the project selecting from the menu File > **Import** and choose the **Maven > Existing Maven Projects**. Click **Next**. Click the **Browse** button and select **`/home/student/DO290/labs/bookstore`**. Click **Finish** to import the project to JBDS.

Note

A popup may open due to problems in **Auto share git projects**. You may safely dismiss the message by clicking the **Ok** button. The side effect of this problem is that JBDS integration with Git will not work.

5. Remote Debug Connections in JBDS
   Now that the JDWP listener port from our JBoss EAP container has been forwarded (via an SSH tunnel) to our local workstation, it is now possible to connect a local JDWP client. JBoss Developer Studio provides a JDWP client that will be leveraged in this lab.

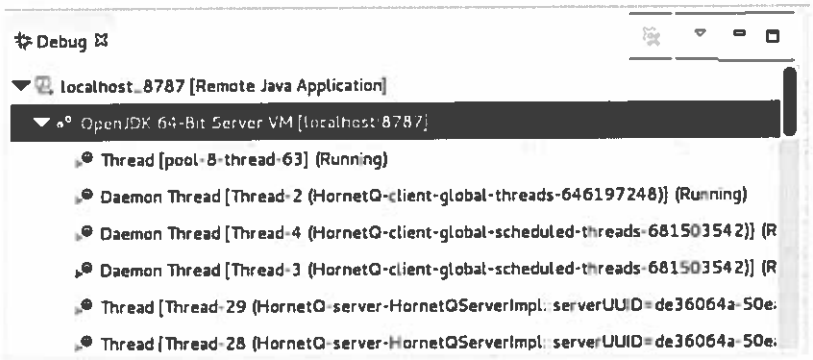
   5.1. Open the **Debug** perspective of JBDS by navigating to: **Window > Perspective > Open Perspective > Debug**.

   5.2. Create a new remote debug connection. In JBDS, navigate to **Run > Debug Configurations ...**. In the **Debug Configurations** pop-up, right-click **Remote Java Application** and select **New**. Populate the **Debug Configuration** fields as follows:
   - Project: Select the bookstore application, clicking the **Browse...** button.

- Name: **`localhost_8787`**
- Host: **`localhost`**
- Port: **8787**

Click the **Apply** and **Debug** buttons.

5.3. The **Debug** panel should now display the stack of remote Java EAP threads controlled by this JWDP client.



JBDS is now able to connect to and control your remote **`bookstore`** application deployed to OpenShift Enterprise.

6. Set a Breakpoint in the Bookstore App
Let's imagine (a common scenario) where you as an application developer are attempting to inspect the result set of a database query. Our **`bookstore`** application contains a **`CatalogService`** with a function used to **`searchForItems`**. Let's apply a break point to this function and inspect the exact state of the result set in real time as it returns from the **`bookstore`** database.

6.1. In JBDS, return to the **JBoss** perspective and switch to the **Project Explorer** panel.

6.2. In the **bookstore** project, navigate to: **src** > **main** > **java** > **com** > **redhat** > **training** > **service** and open the Java source file called **`CatalogService.java`**.

6.3. Scroll down to about line 80 and notice the function with the following signature:

```
List<CatalogItem> searchForItems(String criteria)
```

CatalogService.java

```
73
74          TypedQuery<String> query = mgr.createQuery(
75                  "select distinct i.category from CatalogItem i", String.class);
76          return query.getResultList();
77      }
78
79      public List<CatalogItem> searchForItems(String criteria) {
80          try {
81              FullTextEntityManager fem = Search.getFullTextEntityManager(mgr);
82
83              MultiFieldQueryParser mfqp = new MultiFieldQueryParser(
84                      Version.LUCENE_36, new String[] { "category",
85                              "description", "title", "author" },
86                      new StandardAnalyzer(Version.LUCENE_36));
87              mfqp.setDefaultOperator(QueryParser.Operator.AND);
88              Query query = mfqp.parse(criteria);
89              javax.persistence.Query fquery = fem.createFullTextQuery(query,
90                      CatalogItem.class);
91              @SuppressWarnings("unchecked")
92              List<CatalogItem> items = (List<CatalogItem>) fquery
93                      .getResultList();
94              return items;
95          } catch (ParseException e) {
96              System.out.println(e);
97              return new ArrayList<CatalogItem>();
98          }
99
100     }
```

6.4. In the left edge of this source code panel, double-click the line that includes the following code:

```
FullTextEntityManager fem = Search.getFullTextEntityManager(mgr);
```

A small blue circle should appear indicating that a **breakpoint** has been set on this specific line of code.

6.5. Switch back to the **Debug** perspective of JBDS.

Notice that the **Breakpoints** tab now includes your single break point.

Congratulations! You have set a breakpoint in the **bookstore** source code. Once this **`searchForItems`** function is invoked, the JBDS debugger will stop the remote JVM at this break point. You will then be able to step through the code and inspect the variable state along the way.

7. Trigger **break point**

Now that a *break point* has been set in the **`searchForItems( ...)`** function of the **`CatalogService`** of the **`bookstore`** source code, let's trigger it.

7.1. Open your browser and navigate to the home page of your **`bookstore`** application using the following URL: **`http://bookstore.cloudappsX.example.com`**.

7.2. Using the search box at the top of the home page, execute a full text search using the string : **`Potter`** (which happens to be the last name of an author of several children's books catalogued by our **`bookstore`** application).

7.3. Within a second or two, the **Debug** perspective of JBDS should break at exactly the point in the **`searchForItems`** function where we set the break point. Correspondingly, JBDS will attempt to display the **`CatalogService`** source code and highlight the line where remote execution has stopped.

We are almost ready to step through the **`CatalogService`** code.

7.4. You should see the source code line with our break point highlighted.

The remote JVM has been interrupted during execution at this exact line in our **`CatalogService`** source code.

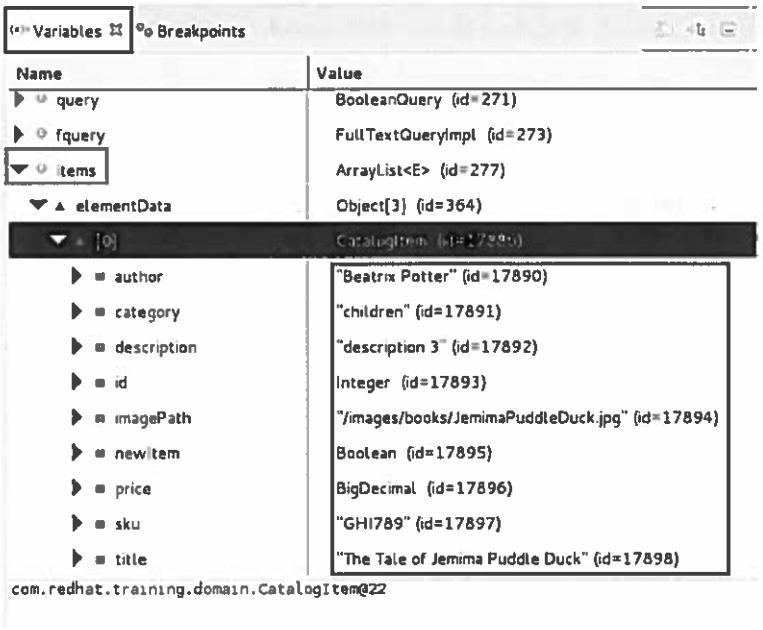
7.5. Let's get a first-hand glimpse at the result set returned by executing the full-text search on the **`CatalogService`**.

1. Press **F6** to step through the source code until line 94 (**`return items;`**) is reached.

```
79    public List<CatalogItem> searchForItems(String criteria) {
80        try {
81            FullTextEntityManager fem = Search.getFullTextEntityManager(mgr);
82
83            MultiFieldQueryParser mfqp = new MultiFieldQueryParser(
84                    Version.LUCENE_36, new String[] { "category",
85                            "description", "title", "author" },
86                    new StandardAnalyzer(Version.LUCENE_36));
87            mfqp.setDefaultOperator(QueryParser.Operator.AND);
88            Query query = mfqp.parse(criteria);
89            javax.persistence.Query fquery = fem.createFullTextQuery(query,
90                    CatalogItem.class);
91            @SuppressWarnings("unchecked")
92            List<CatalogItem> items = (List<CatalogItem>) fquery
93                    .getResultList();
94            return items;
95        } catch (ParseException e) {
96            System.out.println(e);
97            return new ArrayList<CatalogItem>();
98        }
99
100   }
```

2. Switch to the **Variables** panel in the **Debug** perspective.

3. Select **items** (which is the data structure that temporarily stores the returned result set) and then drill down into: **elementData > [0]**.



You are now viewing the state of all variables for this application in real time. Congratulations on having stepped through your **bookstore** application deployed in OpenShift Enterprise using the JDWP.

7.6. Switch back to the panel in the **Debug** perspective displaying the suspended thread stack. Right-click and select **Disconnect**. Your remote JBoss EAP JVM managed by OpenShift Enterprise now executes freely again.

8. Troubleshoot **bookstore** Catalog Service

8.1. Break Catalog Service

Introduce a functional error in the **CatalogService**. We will then step through the **CatalogService** (using the remote debugging capabilities we've previously set up) to isolate the problem. In the **JBoss** perspective of JBDS, open the **CatalogService.java** and navigate to line 93. Add the following just after line 93:

```
List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
items = badDataStructure;
```

```
92          List<CatalogItem> items = (List<CatalogItem>) fquery
93                  .getResultList();
94          List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
95          items = badDataStructure;
96          return items;
```

Save the change.

Commit the change and push the commit to your **bookstore** repository in Git:

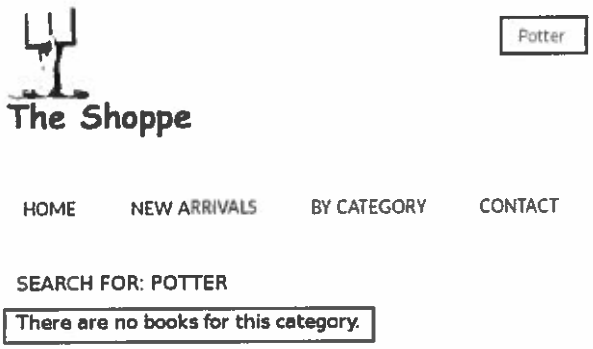
```
[student@workstation ~]$ cd D0290/labs/bookstore
[student@workstation bookstore]$ git add .
[student@workstation bookstore]$ git commit -m "Included a bug"
[student@workstation bookstore]$ git push
```

Trigger a new **build** of your **bookstore** application in OSE:

```
[student@workstation bookstore]$ oc start-build bookstore
bookstore-2
```

Wait until the new build has completed and a new **bookstore** pod has started.

Navigate to the **bookstore** home page using your browser and (as previously done) search for the keyword **Potter**.



As expected (because we introduced a functional error in our **CatalogService**), the result set from our book catalog search is empty.

8.2. Debug Catalog Service

Imagine that you, a developer, were not aware of the root cause of why bookstore catalog searches appear to be returning an empty result set since the last commit of the **bookstore** source code.

Subsequently, the remote debugger is your best friend and you will want to step through the **CatalogService** to isolate the problem. As previously done, create a port-forward tunnel on the debugger port (8787) between your latest remote **bookstore** container and your local workstation. In JBDS, switch to the **Debug** perspective and restart the **localhost_8787** debug connection. Execute the same bookstore catalog search in the **bookstore** home page using the keyword **Potter**. Once the debugger suspends execution at the previously set break point in the **CatalogService**, begin to step through the code (**F6**). At lines 92 and 93, notice that the original **items** data structure is populated. However, the **items** data structure is set to an empty **List<CatalogItem>** data structure in the next line.

8.3. Repair Catalog Service

Comment out the offending code on lines 94 and 95:

```
// List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
// items = badDataStructure;
```

Save the change.

Commit the change and push the commit to your **bookstore** repository in Git by running:

```
[student@workstation bookstore]$ git add .
[student@workstation bookstore]$ git commit -m "Fixing bugs"
```

```
[student@workstation bookstore]$ git push
```

Trigger a new *build* of your **bookstore** application in OSE:

```
[student@workstation bookstore]$ oc start-build bookstore
bookstore-3
```

Wait until the new build has completed and a new **bookstore** pod has started. Navigate to the **bookstore** home page using your browser and (as previously done) search for the keyword **Potter**.This time around, the result set should be three children books.

9. Remove all the elements from the project

   9.1. Remove from JBDS the project **bookstore** by right-clicking the project in the **Project Explorer** tab and choosing **Delete**. Click the **Ok** button to remove the project from the workspace.

   9.2. From a Terminal window, run the following command:

```
[student@workstation ~]$ oc delete project debug-bookstore
[student@workstation ~]$ rm -rf ~/D0290/labs/bookstore
```

This concludes the exercise.

Reviewing OpenShift Enterprise Logs

Objectives

After completing this section, students should be able to locate and review OpenShift Enterprise logs to determine causes of issues with applications.

Master and node logs

Looking at the higher-level logs at the master and node level is often helpful for debugging issues that may occur at the administrative level of the OpenShift environment. After SSHing into either a node or master, the logs are viewable by either reading the **/var/log/messages** file or by using the **journalctl** command. The contents of the **/var/log/messages** file are similar to that output by the **journalctl** command; however, **journalctl** provides some additional features that make reading and debugging with the logs easier. The **journalctl** command provides the following benefits:

- The priority of entries is marked visually.
- The time stamps are converted for the local time zone of the system.
- All logged data is shown, including rotated logs.

To run **journalctl** on a node, first SSH as root onto the machine and run the following command:

```
$ journalctl
```

In many instances, only the most recent entries in the log are relevant. To reduce the output of **journalctl**, use the **-n** option that lists only the specified number of most recent log entries:

```
$ journalctl -n <number>
```

Replace *<number>* with the number of lines to be shown.

To view the full metadata about all entries, use the verbose flag when using **journalctl**.

```
$ journalctl -o verbose
```

Note

To read more about **journalctl** and about some of the other features, visit the *RHEL 7 Documentation.* [https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/System_Administrators_Guide/s1-Using_the_Journal.html]

Adjustments to the **openshift-master** log level can be made by editing the following line of **/etc/sysconfig/openshift-master** from the master VM:

```
OPTIONS=--loglevel=4
```

Note

Restart the **openshift-master** service to make the changes valid.

```
$ sudo -i systemctl restart openshift-master.service
```

In a similar manner, the operating system logs pertaining to the **openshift-node** service (on each OSE 3 node) can be monitored as well. In a node machine, the **journalctl** command provides filtered information using the following syntax:

```
# journalctl -f -u openshift-node
```

CLI commands to view logs

The OpenShift Enterprise client can retrieve the log output for a specific pod or container. To track the logging for the JBoss EAP pod, use the following command **oc logs**.

The **-f** flag follows the log. To find a list of the pods and copy the name of the desired pod:

```
$ oc get pods
```

Execute the **oc logs** command to point to the pod.

```
$ oc logs -f bookstore-1-abcde
```

While using the **oc logs** command can be useful for viewing any output that is logged to the pod or container's console, it does not necessarily display all of the available debug information if further information is being logged to a file. In JBoss EAP 6, for example, the configurable log levels may output more detailed log information into the log file as opposed to the console. The **server.log** file is accessible for standalone server instances at **<EAP-HOME>/standalone/log/server.log**. To access the EAP logs on the EAP container, use the following syntax to execute a remote command in a container:

```
$ oc exec <pod> -it /bin/bash
```

This command opens an interactive Bash terminal within the specified pod. It is now possible to navigate to the JBoss EAP standalone logs and view and search the files.

Build logs

Another resource to facilitate troubleshooting issues is using the build logs. Use the following commands to access the build logs:

Get the list of available builds:

```
$ oc get builds
```

Access the log of the build by name:

```
$ oc build-logs <buildName>
```

Application logs

Most language runtimes allow for adjustment of their logging level. In this section of the lab, the log level of the JBoss EAP container running in OpenShift Enterprise will be increased from **INFO** to **TRACE**. In particular, the verbosity of the logging involving the **infinispan subsystem** may be increased.

Note

Infinispan is a community data grid tool leveraged by JBoss EAP for clustering and load balancing of JEE functionality such as stateful session beans (SFSB).

To run these changes in a EAP instance running on OSE, the administrator must access the EAP container instance using the following command:

```
$ oc exec -p <project> -c <application> -i -t -- bash -il
```

And execute commands using the JBoss EAP CLI, such as:

```
$ /opt/eap/bin/jboss-cli.sh --connect \
  --command="/subsystem=logging/
logger=org.infinispan/:add(category=org.infinispan,level=TRACE,use-parent-
handlers=true)"
```

Note

JBoss administration skills are beyond the scope of this course.

Guided Exercise: Troubleshooting with Logs

In this lab, you will explore the logging capabilities in OpenShift Enterprise.

Resources	
Files	**/home/student/DO290/labs/troub-logs**
Application URL	NA

Outcomes
You should be able to:
- Increase the log level of application logs.
- Increase log level of **oc** client.
- Monitor OSE master and node logs.

Before you begin...
- Verify the OpenShift Enterprise Client is installed.

1. Increase Application Log Level
   1.1. At a terminal window of the workstation VM, run the following command to create a new project:

```
[student@workstation ~]$ oc login -u student -p redhat
[student@workstation ~]$ oc new-project session-replication
```

   1.2. Navigate to **/home/student/DO290/labs/troub-logs**, open the file **session-replication.json** and replace every instance X in **http://workstation.pod*X*.example.com/session-replication.git** and **session-replication.cloudapps*X*.example.com**.

   1.3. Create a **session-replication** JEE application managed by OSE:

```
[student@workstation ~]$ oc create -f \
  /home/student/DO290/labs/troub-logs/session-replication.json
```

   1.4. Once the new **session-replication** app has been built, deployed, and started, determine the name of the pod running the app:

```
[student@workstation ~]$ oc get pod
NAME                          READY   REASON    RESTARTS   AGE
session-replication-1-yd5nk   0/1     Running   0          21s
```

Note

The session replication name may change depending on your environment.

1.5. Once the pod and container names are determined, launch a shell in the **session-replication** container:

```
[student@workstation ~]$ oc exec session-replication-1-yd5nk \
    -c session-replication -i -t -- bash -il
```

1.6. Once at the shell of the **session-replication** container, execute the following JBoss command-line interface (CLI) commands to modify the log level of the **org.infinispan** logger:

```
$ /opt/eap/bin/jboss-cli.sh --connect \
--command="/subsystem=logging/
logger=org.infinispan/:add(category=org.infinispan,level=TRACE,use-parent-
handlers=true)"
```

```
$ /opt/eap/bin/jboss-cli.sh --connect --command="reload"
```

1.7. Exit from the container shell (which returns you back to the shell of your local workstation).

1.8. Now that the embedded JVM has been reloaded, JBoss will begin to log Infinispan cluster messages at the **TRACE** level.

From the shell of your workstation where the **oc** utility is installed, access the application log file:

```
[student@workstation ~]$ oc logs -f session-replication-1-yd5nk session-
replication
```

1.9. Review the log file and notice TRACE-level logging of Infinispan functionality:

```
19:08:59,879 TRACE [org.infinispan.configuration.cache.StateTransferConfigurationBuilder]
19:08:59,879 TRACE [org.infinispan.configuration.cache.StateTransferConfigurationBuilder]
19:08:59,902 TRACE [org.infinispan.configuration.cache.StateTransferConfigurationBuilder]
19:08:59,902 TRACE [org.infinispan.configuration.cache.StateTransferConfigurationBuilder]
```

2. Increase Log Level of **oc** Client

Occasionally, you might have a need to gain more details regarding the exact network communication that occurs between the **oc** client on your local workstation and the OSE master API. Increasing the log level of the **oc** client can provide those details.

To get a feel of this capability, execute the following commands and spend a bit of time studying the HTTP request and response:

Log out from the actual user connected to OpenShift:

```
[student@workstation ~]$ oc logout
```

Log in as the **student** user and printing the response from the request.

```
[student@workstation ~]$ oc login -u student --insecure-skip-tls-verify --v=10
```

Print the project information using a raw view.

```
[student@workstation ~]$ oc project --v=10
```

Print the pods information using a raw view.

```
[student@workstation ~]$ oc get pods --v=10
```

Notice that the HTTP method (ie; GET, POST, PUT, DELETE, etc) and the URL of each **oc** invocation is printed out. Also printed is the full HTTP request headers along with the JSON response body.

3. Clean up
As this project and its resources will not he re-used later during this course, remove everything to keep things tidy.

```
[student@workstation ~]$ oc delete project session-replication
```

This concludes the exercise.

Lab: Troubleshooting an Application

In this lab, you will configure the JBoss container image to run in debug mode and remotely debug the bookstore application.

Resources	
Files	`/home/student/DO290/labs/remote-debugging`
Application URL	NA

Outcomes
You should be able to:
- Debug the bookstore application remotely using JBDS.
- Troubleshoot a real failure in the bookstore application.

Before you begin
- Verify the OpenShift Enterprise Client is installed.

1. Create the bookstore application in OSE.
2. Create a port-forward tunnel for the **bookstore** application.
3. Clone locally the bookstore app
4. Import the project to JBDS
5. Turn on the remote debugger in JBDS.
6. Set a break point in the Bookstore App at line 81 in the **`CatalogService.java`** file inside the **`searchForItems(String criteria)`** function.
7. Trigger the break point by searching for **`Potter`** in the **bookstore** application in your browser and inspect the contents of **`elementData`** in debug mode.
8. Troubleshoot **bookstore** Catalog Service
   8.1. Break the catalog service
   8.2. Trigger the break point again and see where the bug in the code is by stepping through the debugger.
   8.3. Fix the code and rebuild the bookstore application.

```
// List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
// items = badDataStructure;
```

9. Remove all the elements from the project
   9.1. Remove from JBDS the project **bookstore** by right-clicking the project in the **Project Explorer** tab and choosing **Delete**. Click the **Ok** button to remove the project from the workspace.
   9.2. From a Terminal window, run the following command:

```
[student@workstation ~]$ oc delete project lab-bookstore
[student@workstation ~]$ rm -rf ~/DO290/labs/bookstore
```

This concludes the exercise.

Solution

In this lab, you will configure the JBoss container image to run in debug mode and remotely debug the bookstore application.

Resources	
Files	**/home/student/DO290/labs/remote-debugging**
Application URL	NA

Outcomes

You should be able to:

- Debug the bookstore application remotely using JBDS.
- Troubleshoot a real failure in the bookstore application.

Before you begin

- Verify the OpenShift Enterprise Client is installed.

1. Create the bookstore application in OSE.

   1.1. Use the provided script to create the persistent volume:

   ```
   [root@master ~]# bash /home/student/DO290/labs/remote-debugging/createpv.sh
   ```

   1.2. At a terminal window of the workstation VM, run the following command to create a new project:

   ```
   [student@workstation ~]$ oc login -u student -p redhat
   [student@workstation ~]$ oc new-project lab-bookstore
   ```

   1.3. The template being used requires that you create a secret. Inspect and create the following secret:

   ```
   [student@workstation ~]$ oc create -f \
       /home/student/DO290/labs/remote-debugging/secret.json
   ```

   1.4. Navigate to: **/home/student/DO290/labs/remote-debugging**, open the file **bookstore.json** and replace every instance *X* in **http://workstation.pod*X*.example.com/bookstore.git** and **bookstore.cloudapps*X*.example.com**.

   1.5. An environment variable is responsible for enabling the JDWP inside the JBoss Application Server. Edit the **/home/student/DO290/labs/remote-debugging/bookstore.json** and in **DeploymentConfig** section, add the DEBUG variable:

   ```
   ...OUPUT OMMITED...
   "env": [
      {
         "name": "DEBUG",
         "value": "true"
      },
      {
   ```

```
      "name": "DB_SERVICE_PREFIX_MAPPING",
      "value": "bookstore-mysql=DB"
    },
...OUPUT OMMITED...
```

1.6. Execute the following to create a bookstore app managed by OSE:

```
[student@workstation ~]$ oc create -f \
    /home/student/DO290/labs/remote-debugging/bookstore.json
```

1.7. Monitor the build logs.

1.8. Once the build is complete, wait until the deployment begin:

```
[student@workstation ~]$ oc get pods
NAME                READY  STATUS      RESTARTS  AGE
bookstore-1-build   0/1    ExitCode:0  0         5m
bookstore-1-deploy  1/1    Running     0         18s
bookstore-1-n56t1   0/1    Running     0         10s
```

1.9. Tail the application log file of your running bookstore container.

Toward the top of the application log file, note the following entry indicating that the JDWP listener is now enabled in the JVM:

```
[student@workstation ~]$ oc logs bookstore-1-n56t1
```

```
...OUTPUT OMMITED...
Listening for transport dt socket at address: 8787
...OUTPUT OMMITED...
```

2. Create a port-forward tunnel for the **bookstore** application.

2.1. Determine the name of the bookstore pod:

```
[student@workstation ~]$ oc get pod
bookstore-1-n56t1         1/1       Running      0          1h
```

2.2. Using the pod name, create a tunnel:

```
[student@workstation]$ oc port-forward -p bookstore-1-n56t1 8787:8787
```

2.3. Leave this terminal open and running in order to keep the tunnel open.

3. Clone locally the bookstore app
Open a Terminal window and execute the following commands. Replace *X* with your student number:

```
[student@workstation ~]$ cd ~/DO290/labs
[student@workstation labs]$ git clone http://workstation.podX.example.com/
bookstore.git
```

4. Import the project to JBDS
   Open JBDS, and import the project selecting from the menu **File > Import** and choose the **Maven > Existing Maven Projects**. Click **Next**. Click the **Browse** button and select **`/home/student/DO290/labs/bookstore`**. Click **Finish** to import the project to JBDS.

   ## Solution

   A popup may open due to problems in **Auto share git projects**. You may safely dismiss the message by clicking the **Ok** button. The side effect of this problem is that JBDS integration with Git will not work.

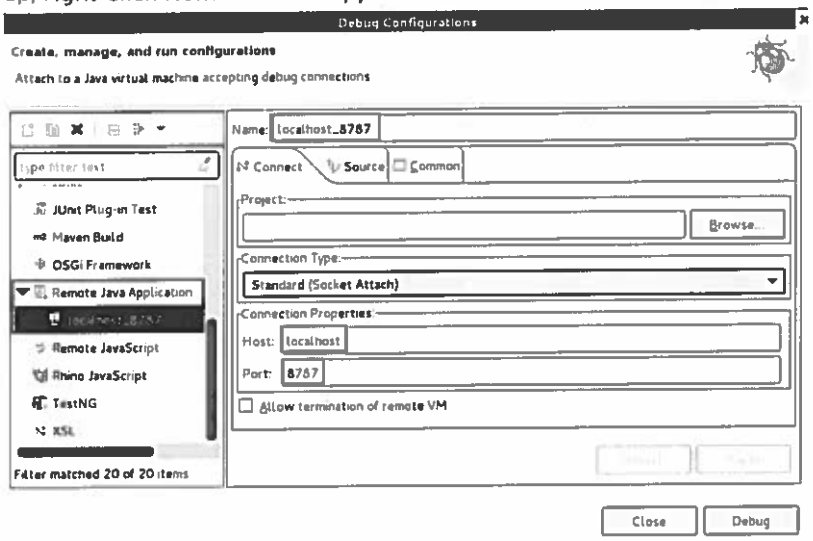
5. Turn on the remote debugger in JBDS.

   5.1. Open the **Debug** perspective of JBDS by navigating to **Window > Show Perspective > Open Perspective > Debug**.

   5.2. Create a new remote debug connection.

   In JBDS, navigate to **Run > Debug Configurations ...**. In the **Debug Configurations** pop-up, right-click **Remote Java Application** and select **New**.

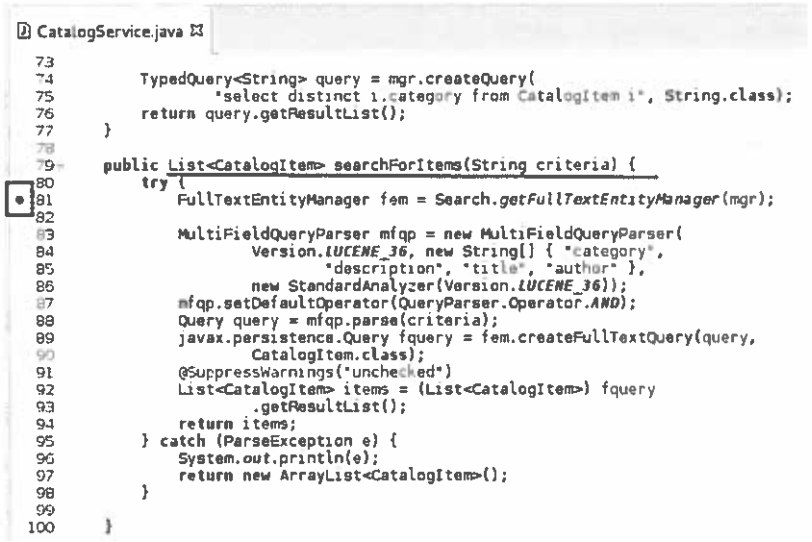
6. Set a break point in the Bookstore App at line 81 in the **`CatalogService.java`** file inside the **`searchForItems(String criteria)`** function.

   6.1. In JBDS, return to the **JBoss** perspective and switch to the **`Project Explorer`** panel.

   6.2. In the **bookstore** project, navigate to **src > main > java > com > redhat > training > service** and open the Java source file called **`CatalogService.java`**.

6.3. Scroll down to about line 80 and notice the function with the following signature:

```
List<CatalogItem> searchForItems(String criteria)
```



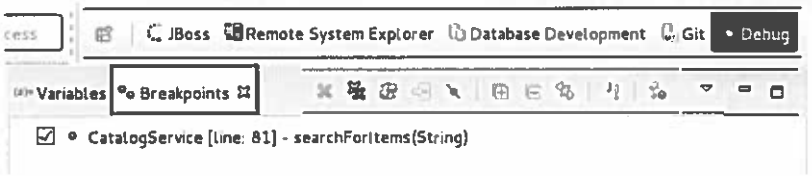
6.4. In the left edge of this source code panel, double-click the line that includes the following code:

```
FullTextEntityManager fem = Search.getFullTextEntityManager(mgr);
```

A small blue circle should appear indicating that a **breakpoint** has been set on this specific line of code.

6.5. Switch back to the **Debug** perspective of JBDS.

Notice that the **Breakpoints** tab now includes your single break point.



7. Trigger the break point by searching for **`Potter`** in the **bookstore** application in your browser and inspect the contents of **`elementData`** in debug mode.

7.1. Open your browser and navigate to the home page of your **`bookstore`** application.

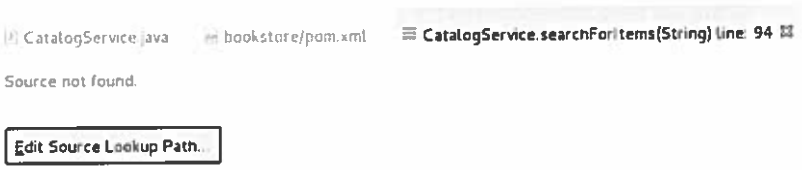
7.2. Using the search box at the top of the home page, execute a full text search using the **`String Potter`** (which happens to be the last name of an author of several children's books catalogued by our **`bookstore`** application).

7.3. Within a second or two, the **Debug** perspective of JBDS should break at exactly the point in the **`searchForItems`** function where we set the break point. Correspondingly, JBDS will attempt to display the **`CatalogService`** source code and highlight the line where remote execution has stopped.

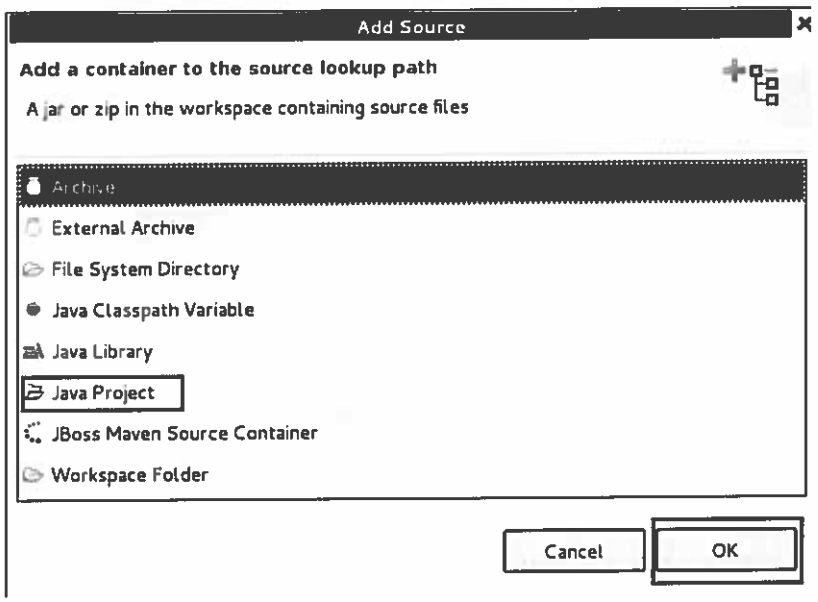
We are almost ready to step through the **`CatalogService`** code.

7.4. It is likely that the source lookup path to the **bookstore** source code is not yet set up in your JBDS.

Once the break point is triggered, you may see the following in the **Debug** perspective:



If you do see this message, execute the following: Click **Edit Source Lookup Path**, in the **Edit Source Lookup Path** pop-up, click **Add**. In the **Add Source** pop-up, click **Java Project** and press **OK**.



A new pop-up should appear, entitled **Project Selection**.

Click **`Select All`** (to ensure the **`bookstore`** project is selected) and then click **OK**.

7.5. Once the **Source Lookup** path to the **bookstore** project is set, you should see the source code line with our break point highlighted.

The remote JVM has been interrupted during execution at this exact line in our **CatalogService** source code.

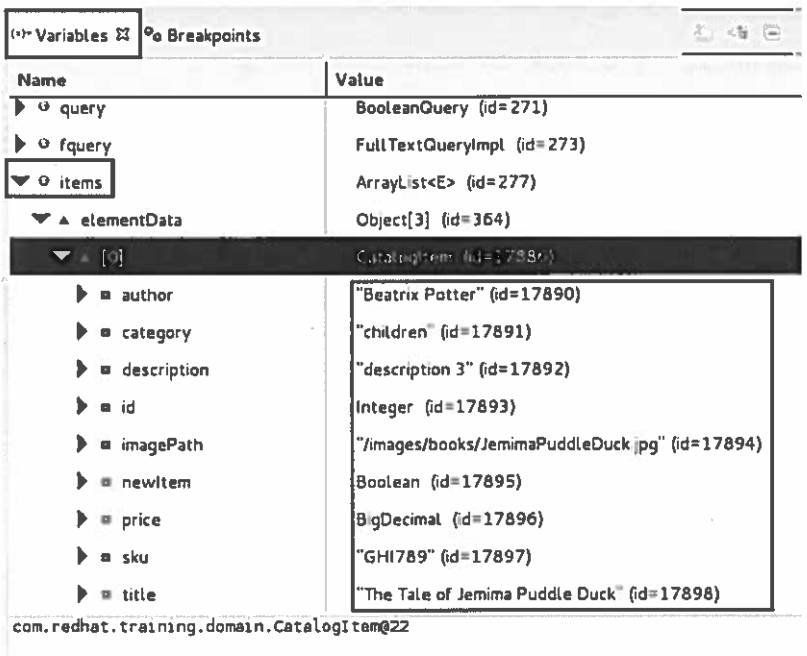
7.6. Let's get a first-hand glimpse at the result set returned by executing the full-text search on the **CatalogService**.

Press **F6** to step through the source code until line 94 (**return items;**) is reached.

```
79-    public List<CatalogItem> searchForItems(String criteria) {
80         try {
81             FullTextEntityManager fem = Search.getFullTextEntityManager(mgr);
82
83             MultiFieldQueryParser mfqp = new MultiFieldQueryParser(
84                     Version.LUCENE_36, new String[] { "category",
85                             "description", "title", "author" },
86                     new StandardAnalyzer(Version.LUCENE_36));
87             mfqp.setDefaultOperator(QueryParser.Operator.AND);
88             Query query = mfqp.parse(criteria);
89             javax.persistence.Query fquery = fem.createFullTextQuery(query,
90                     CatalogItem.class);
91             @SuppressWarnings("unchecked")
92             List<CatalogItem> items = (List<CatalogItem>) fquery
93                     .getResultList();
94             return items;
95         } catch (ParseException e) {
96             System.out.println(e);
97             return new ArrayList<CatalogItem>();
98         }
99
00     }
```

Switch to the **Variables** panel in the **Debug** perspective.

Select **items** (which is the data structure that temporarily stores the returned result set) and then drill down into: **elementData > [0]**.



You are now viewing the state of all variables for this application in real time. Congratulations on having stepped through your **bookstore** application deployed in OpenShift Enterprise using the JDWP.

7.7. Switch back to the panel in the **Debug** perspective displaying the suspended thread stack. Right-click and select **Disconnect**. Your remote JBoss EAP JVM managed by OpenShift Enterprise now executes freely again.

8. Troubleshoot **bookstore** Catalog Service

8.1. Break the catalog service

Break the catalog service by inserting the following lines in **CatalogService.java** and try to search for "Potter" in the application again:

```
92              List<CatalogItem> items = (List<CatalogItem>) fquery
93                      .getResultList();
94              List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
95              items = badDataStructure;
96              return items;
```

In the **JBoss** perspective of JBDS, open the **CatalogService.java** and navigate to line 93 and add the following just after line 93:

```
List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
items = badDataStructure;
```

```
92              List<CatalogItem> items = (List<CatalogItem>) fquery
93                      .getResultList();
94              List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
95              items = badDataStructure;
96              return items;
```

Save the change.

Commit the change and push the commit to your **bookstore** repository.

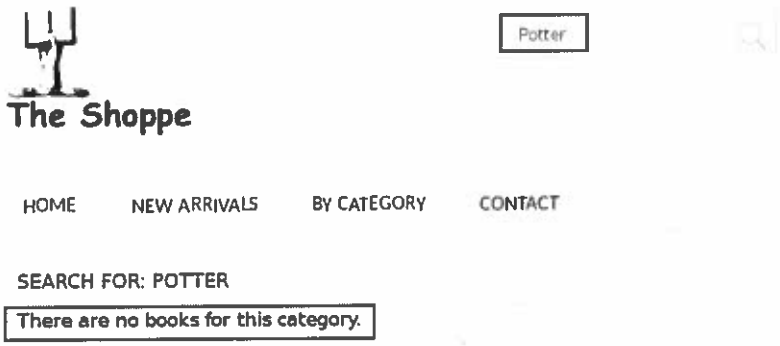
```
[student@workstation bookstore]$ git add .
[student@workstation bookstore]$ git commit -m "Including bugs"
[student@workstation bookstore]$ git push
```

Trigger a new **build** of your **bookstore** application in OSE.

```
$ oc start-build bookstore
bookstore-2
```

Wait until the new build has completed and a new **bookstore** pod has started.

Navigate to the **bookstore** home page using your browser and (as previously done) search for the keyword **Potter**.



As expected (because we introduced a functional error in our **CatalogService**), the result set from our book catalog search is empty.

8.2. Trigger the break point again and see where the bug in the code is by stepping through the debugger.

As previous, create a port-forward tunnel on the debugger port (8787) between your latest remote **bookstore** container and your local workstation.

In JBDS, switch to the **Debug** perspective and restart the **localhost_8787** debug connection.

Execute the same bookstore catalog search in the **bookstore** home page using the keyword **Potter**.

Once the debugger suspends execution at the previously set break point in the **CatalogService**, begin to step through the code (**F6**).

At lines 92 & 93, notice that the original **items** data structure is populated.

However, the **items** data structure is set to an empty **List<CatalogItem>** data structure in the next line.

8.3. Fix the code and rebuild the bookstore application.

Comment out the offending code on lines 94 & 95:

```
// List<CatalogItem> badDataStructure = new ArrayList<CatalogItem>();
// items = badDataStructure;
```

Save the change.

Commit the change and push the commit to your **bookstore** Git repository:

```
[student@workstation bookstore]$ git add .
[student@workstation bookstore]$ git commit -m "Fixing bugs"
[student@workstation bookstore]$ git push
```

Trigger a new **build** of your **bookstore** application in OSE.

```
[student@workstation ~]$ oc start-build bookstore
```

Wait until the new build has completed and a new **bookstore** pod has started.

Navigate to the **bookstore** home page using your browser and (as previously done) search for the keyword **Potter**.

This time around, the result set should be three children's books.

9. Remove all the elements from the project

   9.1. Remove from JBDS the project **bookstore** by right-clicking the project in the **Project Explorer** tab and choosing **Delete**. Click the **Ok** button to remove the project from the workspace.

   9.2. From a Terminal window, run the following command:

```
[student@workstation ~]$ oc delete project lab-bookstore
[student@workstation ~]$ rm -rf ~/DO290/labs/bookstore
```

This concludes the exercise.

Summary

In this chapter, you learned:

- Enabling debug mode in EAP 6 requires enabling the Java Debug Wire Protocol (JDWP) in the **`standalone.conf`** file.
- To remotely debug an EAP application, customize from the EAP template, a parameter called DEBUG as true, open a port-forwarding tunnel (**`oc port-forward -p <podName> <port>:podPort`**) and connect to the local server at the specified debug port within JBoss Developer Studio.
- The **`journalctl`** command is useful for debugging at the administrator level, the **`oc logs`** command is useful for debugging pods and containers, and **`oc build-logs`** provides access to the build logs.

CHAPTER 9

CUSTOMIZING OPENSHIFT ENTERPRISE

Overview	
Goal	Create custom S2I builder images and templates.
Objectives	• Create a custom S2I builder image. • Deploy a custom S2I builder image. • Create and deploy a custom template.
Sections	• Creating a Custom S2I Builder Image (and Guided Exercise) • Deploying Custom S2I Builders (and Guided Exercise) • Creating a Custom Template (and Guided Exercise)
Lab	• Creating and deploying a custom S2I builder

Creating a Custom S2I Builder Image

Objectives

After completing this section, students should be able to create a custom S2I builder image.

Introduction

OpenShift Enterprise (OSE) runs applications as pods, which each contains one or more running containers. The containers are in turn created from container images pulled from a image registry. The norm for a developer is to use the OSE Source-to-Image (S2I) process to build images from application source code; the S2I process also runs as a pod, using a builder image that has specific knowledge about the application runtime environment.

If OSE does not provides a builder image for a particular runtime environment, the developer needs to create an S2I builder image for a programming language, web framework, or application server. Maybe the builder image already exists in a third-party image registry; in this case, the developer needs to create an image stream pointing to the builder image so it can be used by OpenShift.

The developer experience can be enhanced by creating customized templates that automate creating deployment configurations, build configurations, services, etc., to use the custom S2I builder image. Custom templates would make it easier using those images for other developers.

Design and create the container

A container image should contain an application binary and its dependencies, but not include any extra binaries or libraries. For example, common UNIX services like SSH should not be part of most container images.

Container images can be created manually, from scratch; after all, they are basically tar files containing the complete container file system, plus a few metadata files. A few base Linux distribution images have to be created this way, but most container images created outside an OSE installation are created from **Dockerfiles**, which are text files with a few entries describing what to add to a base image to produce a new container image.

S2I can be used, in most cases, as a better alternative to Dockerfiles because S2I:

- Generates container images with fewer layers.
- Can reuse artifacts from previous builds.
- Will not risk the production systems to root privilege escalation.

In fact, S2I does not need to perform compilation or builds from application source code: it can also be used to inject application binaries, configuration files, and data files into a preexisting image and package the result as a new container image.

But for creating custom builder images, the S2I process usually will not be enough because OSE does not allow S2I builds to perform operations like installing new RPM packages into a base image. So Dockerfiles will be needed to create builder images most of the time.

OSE can build and deploy container images from Dockerfiles without exposing the user to Docker internals. If a Dockerfile and supporting files are available from a Git repository, the **oc new-app**

command is able to fetch the Dockerfile and run the build as an OSE pod, then push the resulting image to the internal registry, where it can be used to create application pods.

Note

The container images generated by some Dockerfiles available from the general Docker community may not run unchanged in OSE because they require running under specific user ID or even as root. OSE, by default, runs containers under an exclusive random UID so one container cannot interact directly with other containers or its files.

Demonstration: Creating a container from its Dockerfile

In this example, please follow along with these steps while your instructor demonstrates how to create an application from a Dockerfile using OSE.

1. Clone the **rhel7-httpd** application from the classroom Git server. Replace *X* by your student number.

```
[student@workstation ~]$ git clone http://workstation.podX.example.com/rhel7-
httpd.git
```

   This creates a folder named **rhel7-httpd** in the **student**'s home directory. Enter this folder to continue.

2. Inspect the Dockerfile. It contains instructions to install Apache Httpd over a minimal RHEL7 image, create a sample welcome page, and start the web server inside the container. The following listing shows the Dockerfile.

   Edit the Dockerfile to replace *X* with your student number:

```
# Sample Dockerfile to build an Apache Web Server container image
FROM workstation.podX.example.com:5000/rhel7
USER default
MAINTAINER John Doe &lt;jdoe@example.com>
EXPOSE 8080
USER root
COPY ./training.repo /etc/yum.repos.d/training.repo
RUN yum -y install httpd && yum clean all -y
RUN sed -i 's/Listen 80/Listen 8080/' /etc/httpd/conf/httpd.conf
RUN echo "Sample welcome page" > /var/www/html/index.html
RUN chmod -R a+rwx /run/httpd /etc/httpd/logs
# OSE ignores uid. Image should work with any, that's why rwx above
USER 1001
# should change apache config to log to stdout
CMD /bin/bash -c 'echo Starting web server... ; /usr/sbin/httpd -DFOREGROUND || echo
 Apache Httpd startup failed: $?'
```

3. Push the changes to the remote Git repository. From the **/home/student/rhel7-httpd** directory, run the following commands:

```
[student@workstation rhel7-httpd]$ git add .
[student@workstation rhel7-httpd]$ git commit -m 'registry changed'
```

```
[student@workstation rhel7-httpd]$ git push
```

4. Log in as the developer user **openshift** and create a new project to run the Dockerfile build.

```
[student@workstation rhel7-httpd]$ oc login -u student -p redhat
[student@workstation rhel7-httpd]$ oc new-project rhel7-httpd
```

5. Create a new application from the Git repository containing the Dockerfile. Replace *X* by your student number.

```
[student@workstation rhel7-httpd]$ oc new-app http://workstation.podX.example.com/
rhel7-httpd.git --name=webserver
```

6. Wait until the Dockerfile build finishes and the web server pod is created. You can follow the build progress using the **oc build-logs webserver-1** command and, after the build finishes, verify the web server pod is running using the **oc get pods** command. The expected output is something similar to:

```
NAME                READY     STATUS        RESTARTS   AGE
webserver-1-build   0/1       ExitCode:0    0          4m
webserver-1-inp1i   1/1       Running       0          2m
```

7. Create a service and a route so the web server pod can be accessed from the developer's workstation.

```
[student@workstation rhel7-httpd]$ oc expose pod webserver-1-inp1i
[student@workstation rhel7-httpd]$ oc expose svc webserver
```

8. Verify the web server responds to requests with the sample welcome page. Replace *X* by your student number.

```
[student@workstation rhel7-httpd]$ curl http://webserver.rhel7-
httpd.cloudappsX.example.com/
```

Insight

When the **oc expose svc** command is invoked without the **--hostname** option, it generates a host name concatenating the service name, the project name, and the default wild-card domain configured in the OSE master.

9. Clean up; delete the project.

```
[student@workstation rhel7-httpd]$ oc delete project rhel7-httpd
```

This concludes the demonstration.

Create a S2I builder image

OSE builder images are supposed to collaborate with the OSE **s2i** utility to perform the following tasks:

1. Fetch (pull) application source code from a Git repository.
2. Build application binaries from the fetched source code and optionally artifacts saved from a previous build.
3. Optionally save the build artifacts so they can be reused by a new build.
4. Inject the application binaries into a new container image.
5. Publish (push) the new container image into the OSE internal registry.

The entire workflow is coordinated by the **s2i** utility, which creates a pod running the builder image and invokes scripts as needed from inside the pod. The **s2i** utility itself is language-agnostic; the scripts can be written in any language supported by the builder image as long as they are executable by the shell.

The **s2i** utility performs all runtime agnostic steps, like fetching sources from a Git repository and uploading the generated application image to the OSE internal registry. The scripts inside the builder image perform the runtime-dependent steps, like building application binaries and saving build artifacts for reuse.

The builder image has also to provide the application runtime, dependencies, and any other language-specific tooling. For example, a Java EE build image would provide a JRE, Maven and an application server, alongside scripts to run Maven to build and package the Java EE application, start the application server, and deploy the application.

The **s2i** utility can be run standalone in any Linux system that has Docker installed to create and test build images outside an OSE instance. Inside an OSE instance, it runs as a privileged pod created by a build configuration from the **openshift/origin-sti-builder** container image. This pod creates the build pod that runs the builder image specific for an application runtime.

The following table describes the scripts provided by the builder image.

Script	Description	Required
assemble	The **assemble** script builds the application artifacts from sources and places them into appropriate directories inside the image.	Yes
run	The **run** script executes your application.	Yes
save-artifacts	The **save-artifacts** script gathers all dependencies and other artifacts that can speed up repeated builds into a tar file.	No
usage	The usage script allows you to inform the user how to properly use your image.	No
test/run	The **test/run** script allows for creating a simple process to check if the image is working correctly.	No

The builder image has to define the **io.openshift.s2i.scripts-url** label with the directory name where those scripts are to be found; for example, by adding the following directive to its Dockerfile:

```
LABEL io.openshift.s2i.scripts-url=image:///usr/local/sti
```

Besides implementing the S2I workflow, the **s2i** utility also provides some developer features. It can create a skeleton builder image project, containing starter S2I scripts and a starter Dockerfile. For example, to create a project named **myproject** that creates image **mybuilder**, use the **s2i create** command:

```
$ sti create mybuilder myproject
```

The starter S2I scripts will be in a hidden folder named **myproject/.sti** and the starter Dockerfile will be in the project root folder. The generated **Dockerfile** file uses as its base the image **openshift/base-centos7**, available from the Docker Hub. A builder image is NOT required to use this base image; any other one could be used.

Best Practice

Customers with Red Hat subscriptions probably want to change the base image to **openshift/base-rhel7**. This image Dockerfile is available from the following GitHub project and has to be built locally:

https://github.com/openshift/sti-base

Another option would be the **rhel7** image available from the Red Hat subscriber private registry.

Inside the project folder, add any files and scripts needed by the builder image. When all files are in place, run the Dockerfile using the **docker build** command, having the project root folder as the current directory:

```
$ docker build -t mybuilder .
```

Insight

This Docker build could be run inside OSE to integrate the builder image build process to a Continuous Integration (CI) or a Continuous Delivery (CD) pipeline.

To test the new builder image outside of OpenShift, use the **s2i build** command, passing three arguments:

1. A test application source repository URL.
2. The builder image name, which has to be available from the local Docker daemon.
3. The container image name to be created.

For example, to use the **mybuilder** builder image to create the **testapp** application image from sources at the Git repository **http://mygitserver/testapprepo**, use the following command:

```
$ s2i build http://mygitserver/testapprepo mybuilder testapp
```

After the builder image is tested, it can be tagged and uploaded (pushed) to an image registry so it is available to an OSE instance.

Insight

The **docker** command can only be run as **root** from a RHEL, CentOS, or Fedora system. Developer users are advised to use the **sudo** command to get the necessary **root** access. The **s2i** utility makes Docker API calls, so it also needs to be run as **root**.

Guided Exercise: Create a Custom Apache Httpd Builder Image

In this lab, you will create a S2I builder image that creates static web sites hosted by an Apache Httpd server.

Resources	
Files	**/home/student/DO290/labs/apache-builder**
	/home/student/DO290/installers
Application URL	NA

Outcome(s)
A container image compatible with OpenShift Enterprise (OSE) Source-to-Image (S2I) builds.

Before you begin
Have a local Docker installation and **sudo** root access.

1. Install The s2i Utility
   The **s2i** utility is currently not packaged as part of the RHEL7 or OSE3 repositories, so it need to be installed from source code.

   1.1. Extract the **s2i** binary distribution from the tar file into the **student** user home folder.

   ```
   [student@workstation ~]$ tar xzvf \
     ~/DO290/installers/source-to-image-v1.0.2-00d1cb3-linux-amd64.tar.gz
   ```

   1.2. Move the **s2i** utility binary file to the local shared programs folder and delete its alias **sti**.

   ```
   [student@workstation ~]$ sudo mv s2i /usr/local/bin
   [student@workstation ~]$ rm sti
   ```

   1.3. Verify the **s2i** utility is available on the user **PATH**.

   ```
   [student@workstation ~]$ s2i version
   ```

   The expected output is:

   ```
   s2i v1.0.2
   ```

2. Build The base-rhel7 Container Image
   The **base-rhel7** container image is the base image for all OSE3 builder images. It contains common files and scripts to use the **Red Hat Software Collections Library** (SCL) that allow using newer releases of application runtimes and databases, even if they were not packaged originally as part of a RHEL release series.

   2.1. Go back to the student user home folder and fetch the **sti-base** repository from the classroom Git server. Remember to replace *X* with your student number.

```
[student@workstation ~]$ git clone \
   http://workstation.podX.example.com/sti-base.git
```

2.2. Inspect the **Dockerfile.rhel7** file that builds the **base-rhel7** container image. It was modified to fit the classroom environment.

```
[student@workstation ~]$ cd sti-base
[student@workstation sti-base]$ less Dockerfile.rhel7
```

2.3. Fetch the **Dockerfile.rhel7** base image **rhel7** from the classroom private image repository. Remember to replace *X* with your student number.

```
[student@workstation sti-base]$ sudo docker pull
 workstation.podX.example.com:5000/rhel7
```

2.4. Build the **base-rhel7** container image using Docker.

```
[student@workstation sti-base]$ sudo docker build -t openshift/base-rhel7 \
  -f Dockerfile.rhel7 .
```

Note

Do not forget the current directory (.) as the last command argument.

3. Build The httpd-s2i Builder Image

The **httpd-s2i** builder image takes static HTML files as source code and adds them to an Apache Httpd server. The web server is installed from SCL packages instead of from regular RHEL7 packages.

Note

All files created or changed during this exercise are available from the **/home/student/DO290/labs/apache-builder** folder.

3.1. Go back to the student user home folder and use the **s2i** utility to create a skeleton directory for the new builder image.

```
[student@workstation sti-base]$ cd
[student@workstation ~]$ s2i create httpd-s2i httpd-s2i
```

3.2. Enter the newly created folder and create a subdirectory for holding other scripts used by the Dockerfile:

```
[student@workstation ~]$ cd httpd-s2i
[student@workstation httpd-s2i]$ mkdir -p contrib/etc
[student@workstation httpd-s2i]$ cd contrib/etc
```

3.3. Copy the file **scl_enable** to the actual directory using the following command:

```
[student@workstation etc]$ cp ~/DO290/labs/apache-builder/scl_enable .
```

This allows running the **httpd** binary from SCL packages instead of from regular RHEL7 packages.

3.4. Copy the file **httpdconf.sed** to the actual directory using the following command:

```
[student@workstation etc]$ cp ~/DO290/labs/apache-builder/httpdconf.sed .
```

This changes the Apache Httpd configuration to work inside a container.

3.5. Go to the hidden **.s2i** folder in the project root to add S2I scripts.

```
[student@workstation etc]$ cd ../../.s2i/bin
```

3.6. Review the assemble script generated by the **s2i** utility.

```
[student@workstation bin]$ less assemble
```

It extracts the application sources to the container current directory, which will be set up by the **Dockerfile** as the Apache **DocumentRoot** folder. As these builder sources are static HTML files, there is no need to do anything else.

3.7. Copy the **run** script to the actual directory using the following command:

```
[student@workstation bin]$ cp ~/DO290/labs/apache-builder/run .
```

It simply starts the Apache Httpd server in the foreground.

Best Practice

Containerized applications should not be started in the background.

3.8. Go back to the project root and change the **Dockerfile** file generated by **s2i** to build the new container image.

```
[student@workstation bin]$ cd ../../
[student@workstation httpd-s2i]$ cp ~/DO290/labs/apache-builder/Dockerfile .
```

3.9. Copy the **training.repo** file from **/home/student/DO290/labs/apache-builder** to the s2i project:

```
[student@workstation httpd-s2i]$ cp \
    ~/DO290/labs/apache-builder/training.repo .
```

3.10. Build the **httpd-s2i** container image using Docker.

```
[student@workstation httpd-s2i]$ sudo docker build -t httpd-s2i .
```

Note

Do not forget the current directory (.) as the last command argument.

4. Test S2I Builds Using the httpd-sti Builder Image
Use the **s2i** utility to create a container image using the new builder image. This new image is an application container image including HTML source files fetched from a local Git repository.

4.1. Create a welcome page as the **index.html** file in the **test/test-app** folder. The **index.html** file should have the following contents.

```
<html>
<head>
        <title>HTML Home Page</title>
</head>
<body>
<h1>This is the home page</h1>
</body>
</html>
```

4.2. Initialize the **test/test-app** folder as a local Git repository.

```
[student@workstation httpd-s2i]$ cd test/test-app
[student@workstation test-app]$ git init .
[student@workstation test-app]$ git add .
[student@workstation test-app]$ git commit -m "test HTML application"
```

4.3. Go back to the project root and use the **s2i** utility to create an application image using the local Git repository as the source repository.

```
[student@workstation test-app]$ cd ../..
[student@workstation httpd-s2i]$ sudo /usr/local/bin/s2i \
build file:///home/student/httpd-s2i/test/test-app/ \
httpd-s2i:latest httpd-s2i-test
```

Note

The error message **An error occurred when pulling httpd-s2i:latest: unable to get httpd-s2i:latest. Attempting to use local image** can be safely ignored. It is expected to happen as s2i build process starts by trying to fetch an older version of the destination image.

4.4. Verify the new application image **httpd-s2i-test** is available on the local Docker daemon, as well as the builder image **httpd-s2i** and the base image **openshift/base-rhel7**.

```
[student@workstation httpd-s2i]$ sudo docker images | grep httpd-s2i
```

The expected output is similar to:

```
REPOSITORY                                                      TAG
      IMAGE ID            CREATED             VIRTUAL SIZE
httpd-s2i-test                                                  latest
      8976c9f0eb45        59 seconds ago      386.4 MB
httpd-s2i                                                       latest
      46a63ed6ae43        8 minutes ago       386.4 MB
```

4.5. Create a new container using the new application image.

```
[student@workstation httpd-s2i]$ sudo docker run -p 8080:8080 httpd-s2i-test
```

Note

The **-p** option from the **docker run** command redirects port 8080 from the local host to port 8080 in the container.

4.6. Open another terminal, and access the web server running inside the new container:

```
[student@workstation ~]$ curl http://127.0.0.1:8080
```

The expected output is the same as the welcome page created as the **test/test-app/index.html** file.

4.7. Go back to the first terminal and stop the test container using **Ctrl+C**

5. Make The New Builder Image Available To OSE
Publish the new S2I builder image to the classroom private registry, so it can be accessed by OSE.

5.1. Tag and upload (push) the new builder image to the classroom private image registry so it becomes available to the OSE instance. Remember to replace *X* with your student number.

```
[student@workstation ~]$ sudo docker tag httpd-s2i:latest \
 workstation.podX.example.com:5000/httpd-s2i:latest
[student@workstation ~]$ sudo docker push \
    workstation.podX.example.com:5000/httpd-s2i:latest
```

5.2. Optional: save a backup of the builder image in a **tar** file.

```
[student@workstation ~]$ sudo docker save -o httpd-s2i.tar httpd-s2i
```

```
[student@workstation ~]$ gzip -v httpd-s2i.tar
```

6. Clean up

6.1. Delete the local clones of the **sti-base** and **httpd-s2i** Git repositories.

```
[student@workstation ~]$ rm -rf sti-base httpd-s2i
```

6.2. Delete all local containers.

```
[student@workstation ~]$ sudo docker rm $(sudo docker ps -qa)
```

6.3. Delete all local container images created or fetched by this lab.

```
[student@workstation ~]$ sudo docker rmi -f $(sudo docker images | egrep 'httpd|
rhel7' | awk '{print $3}')
```

Note

Error messages such as **Error: failed to remove images** can be safely ignored. They are a by-product of the way **docker build** works.

6.4. Alternatively, delete all local container images, if there are none you wish to preserve.

```
[student@workstation ~]$ sudo docker rmi -f $(sudo docker images -q)
```

Deploying Custom S2I Builders

Objective

After completing this section, students should be able to deploy a custom S2I builder image.

Adding the new builder to OSE

After a new S2I builder image is built, tested, and pushed to an image registry, it only becomes visible to OSE users after an image stream is created. This image stream allows automating application update processes that rebuild and redeploy applications based on the S2I builder image whenever the image is changed; for example, to include an updated release of a dependency library.

The image stream resource definition has to include an **spec.tags.annotations** object attribute with **supports** and **tags** attributes. Those annotations are used by the web console to search candidate builder images when creating a new application from source code.

For example, inspect a predefined S2I builder image stream:

```
$ oc export -o json -n openshift is php
{
    "kind": "ImageStream",
    ... apiVersion and metadata attributes omitted ...
    "spec": {
        "tags": [
            {
                "name": "5.5",
                "annotations": {
                    "description": "Build and run PHP 5.5 applications",
                    "iconClass": "icon-php",
                    "supports": "php:5.5,php",
                    "tags": "builder,php",
                    "version": "5.5"
                },
                "from": {
                    "kind": "ImageStreamTag",
                    "name": "latest"
                }
            },
            ... other image stream spec attributes omitted ...
        ]
    }
}
```

The previous listing shows the **php** image stream from project **openshift** is an S2I builder image because it contains the **builder** tag, and that it supports **php** version **5.5**.

After an image stream resource is created with the correct annotations, the builder image is shown by the OSE web console whenever it detects the source code matches the image stream annotations. If for any reason the web console cannot detect which kind of application is in the source code repository, or if it offers an inadequate builder image, the user can ask to see all builder images and choose the desired image.

The image stream resource definition for a new S2I builder image can be added to any project using the **oc create** command, but this way it will be visible only for applications created inside

the same project. An OSE instance administrator can use the **-n openshift** option to the **oc create** command to create the image stream resource inside the **openshift** project, and all users will be able to use the referenced S2I builder image from any project.

Guided Exercise: Deploying a Custom S2I Builder Image

In this lab, you will deploy an application created from a prebuilt container image.

Resources	
Files	**/home/student/DO290/labs/custom-s2i-builder**
Application URL	NA

Outcome
An application pod built from source, using a custom Source-to-Image (S2I) builder image.

Before you begin
Have an OSE instance installed with router and the **httpd-s2i** builder image pushed to the classroom internal registry. If you do not have that image, run the **/home/student/DO290/labs/custom-s2i-builder/publish-httpd-s2i.sh** script using **sudo**.

1. Verify Prerequisites
   Check the **httpd-s2i** builder image is available in the classroom private image registry.

   ```
   [student@workstation ~]$ sudo docker search httpd-s2i
   ```

   If it is not, please check this exercise's prerequisites.

2. Create an Image Stream Referring to the Builder Image
   Custom S2I builder images will only be visible to OSE web console users if there is an image stream pointing to the image and that image stream has the correct annotations.

   2.1. Log in as the developer user **student** and create a new project to test the custom S2I builder image.

   ```
   [student@workstation ~]$ oc login -u student -p redhat
   [student@workstation ~]$ oc new-project s2i-builder
   ```

   2.2. Review the resource definition for the image stream in the **httpd-s2i-is.json** file, for example, using the **gedit** text editor to have JSON syntax highlighting.

   ```
   [student@workstation ~]$ gedit /home/student/DO290/labs/custom-s2i-builder/
   httpd-s2i-is.json
   ```

   Check it contains **tags**, **supports**, and **version** annotations. Use the following partial listing as a reference:

   ```
   ...
           "tags": [
               {
                   "name": "2.4",
                   "annotations": {
   ```

```
                    "description": "Build and run static web apps served by
 httpd 2.4",
                    "supports": "httpd,httpd:2.4",
                    "tags": "builder,httpd",
                    "version": "2.4"

                },
...
```

2.3. While reviewing the **httpd-s2i-is.json** file, change the image stream **dockerImageRepository**, replacing *X* with your student number. Use the following partial listing as a reference.

```
...
    "spec": {
        "dockerImageRepository": "workstation.podX.example.com:5000/httpd-s2i",
        "tags": [
...
```

2.4. Create an image stream resource pointing to the custom builder image using the provided JSON file.

```
[student@workstation ~]$ oc create -f \
    /home/student/DO290/labs/custom-s2i-builder/httpd-s2i-is.json
```

3. Create An Application Using The Web Console.
Create a pod from source code built using the custom builder image.

3.1. Open a web browser and enter the following URL to access the OSE web console, replacing *X* with your student number.

https://master.pod*X*.example.com:8443/

3.2. Log in as the developer user **student** using password **redhat** and enter project **s2i-builder**.

3.3. In the project overview page, click the **Add to project** button.

3.4. Use the following URL as the application source code Git repository, replacing *X* with your student number, and click the **Next** button.

http://workstation.pod*X*.example.com/sample-site.git

3.5. Click the box with the **httpd-s2i** builder. Use the following figure as reference.

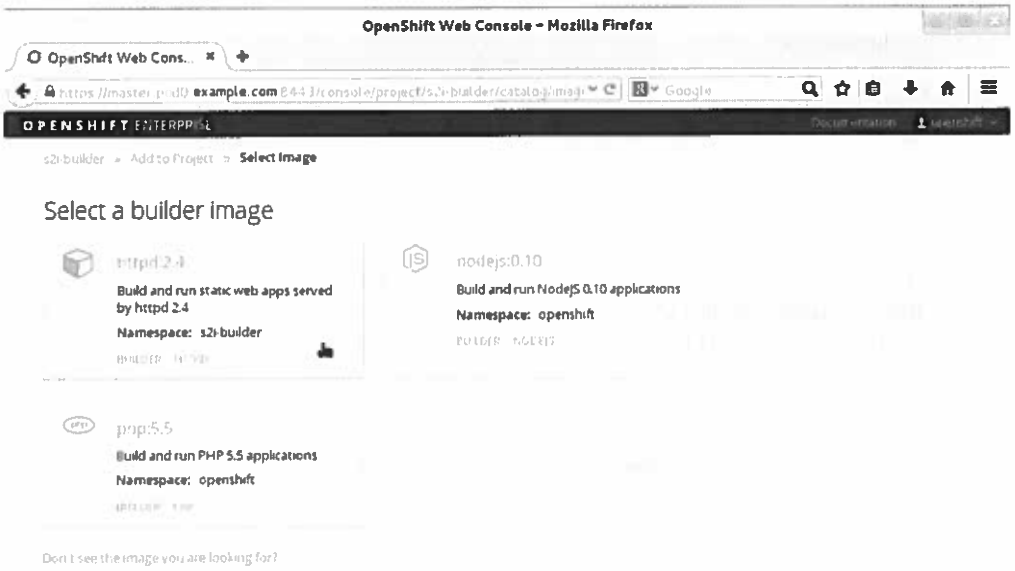


Figure 9.1: httpd-s2i builder

3.6. Scroll down the page and click the **Create** button.

3.7. In a few moments a build should start. Wait until it finishes and the application pod is running. Use the following figure as a reference.

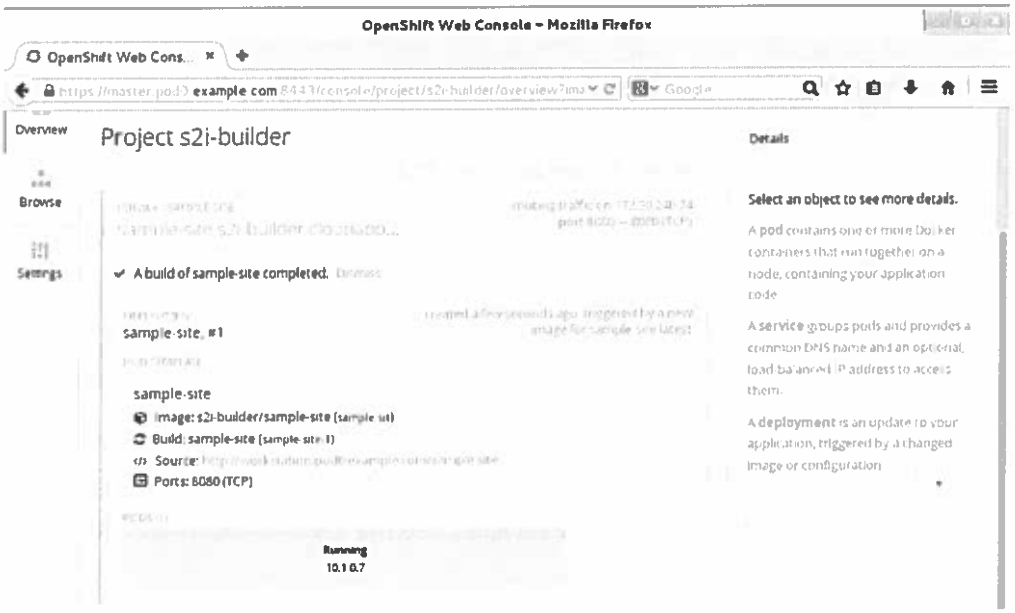


Figure 9.2: sample-site application built and deployed

4. Test The Application
The web console already created a route to allow access to the application.

Open another web browser tab and access the route URL, which should be as follows, replacing *X* with your student number.

http://sample-site.s2i-builder.cloudapps*X*.example.com

The result should be the sample web site welcome page.

5. Clean Up
Delete the **s2i-builder** project.

```
[student@workstation ~]$ oc delete project s2i-builder
```

This concludes the exercise.

Creating a Custom Template

Objectives

After completing this section, students should be able to create and deploy a custom template.

Create the template definition

A template is simply a resource definition file that describes a list containing multiple OSE resources. It can describe everything from a single application pod to a complex application composed from multiple tiers or multiple collaborating microservices.

To create a template from scratch, a developer needs to know the syntax for each individual resource definition: pod, service, deployment configuration, image stream, etc. But there is a simpler alternative: the **oc new-app** command accepts option **-o json** to not create anything, but generate the resource description using JSON syntax. The generated resource definition can be changed to become a template definition.

For example, the following command generates the **hello.json** file with resource definitions for a single pod application created from a predefined container image:

```
$ oc new-app -o json openshift/hello-openshift > hello.json
```

The output is something like:

```
{
    "kind": "List",
    "apiVersion": "v1",
    "metadata": {},
    "items": [
        {
            "kind": "ImageStream",
            ... image stream attributes omitted ...
        },
        {
            "kind": "DeploymentConfig",
            ... deployment config attributes omitted ...
            "spec": {
                ... image stream triggers and strategy omitted ...
                "replicas": 1,
                "selector": {
                    "deploymentconfig": "hello"
                },
                "template": {
                    "metadata": {
                        "creationTimestamp": null,
                        "labels": {
                            "deploymentconfig": "hello"
                        }
                    },
                    "spec": {
                        "containers": [
                            {
                                "name": "hello-openshift",
                                "image": "station.example.com:5000/openshift/hello-
openshift",
                                "resources": {}
```

```
                        }
                    ]
                }
            }
        },
        "status": {}
    }
  ]
}
```

To turn this from a simple resource listing into a template resource definition:

- Change the **kind** attribute from **List** to **Template**.
- Add a **name** attribute and value to the **metadata** object, so the template has a name users can refer to.
- Add an **annotations** attribute to the **metadata** object, containing a **description** attribute for the template so users know what the template is supposed to do.
- Rename the **items** array attribute to **objects**.

The **single-image** template definition, based on the previous listing, would be something like:

```
{
    "kind": "Template",
    "apiVersion": "v1",
    "metadata": {
        "name": "single-image",
        "creationTimestamp": null,
        "annotations": {
            "description": "Application template for creating applications from pre-
build container images."
        }
    },
    "objects": [
        {
            "kind": "ImageStream",
            ... image stream attributes omitted ...
        },
        {
            "kind": "DeploymentConfig",
            ... deployment config attributes omitted ...
            "spec": {
                ... image stream triggers and strategy omitted ...
                "replicas": 1,
                "selector": {
                    "deploymentconfig": "${APP_NAME}"
                },
                "template": {
                    "metadata": {
                        "creationTimestamp": null,
                        "labels": {
                            "deploymentconfig": "${APP_NAME}"
                        }
                    },
                    "spec": {
                        "containers": [
                            {
                                "name": "${APP_NAME}",
                                "image": "${IMAGE_NAME}",
                                "resources": {}
```

```
                        }
                    ]
                }
            }
        },
        "status": {}
    }
  ]
}
```

A template that always creates the same resources, using fixed values for names and other attribute values, is not that useful. The true power of templates comes from the addition of **parameters**. They provide text values that can be referenced multiple times inside the template. Any attribute value inside a template can reference a parameter value.

To add parameters to a template, follow these steps:

- At the end of the template definition (just before the closing curly bracket), add a **`parameters`** array attribute.

  Each array element is an object containing attributes **`name`**, **`description`**, and **`value`** (optional).

- Replace fixed attribute values in the template with parameter value references denoted by **`${parameter_name}`**.

Most prebuild images define **environment variables** whose values are ideal candidates for parameter values. For example, the **`mysql-55-rhel7`** container image defines variables for the database name, database user, etc.

For example, after adding parameters **`APP_NAME`** and **`IMAGE_NAME`** to the previous listing, the result would be:

```
{
    "kind": "Template",
    "apiVersion": "v1",
    "metadata": {
        "name": "single-image",
        "creationTimestamp": null,
        "annotations": {
            "description": "Application template for creating applications from pre-
build container images."
        }
    },
    "objects": [
        {
            "kind": "ImageStream",
            ... image stream attributes omitted ...
        },
        {
            "kind": "DeploymentConfig",
            ... deployment config attributes omitted ...
            "spec": {
                ... image stream triggers and strategy omitted ...
                "replicas": 1,
                "selector": {
                    "deploymentconfig": "${APP_NAME}"
                },
                "template": {
```

```
                "metadata": {
                    "creationTimestamp": null,
                    "labels": {
                        "deploymentconfig": "${APP_NAME}"
                    }
                },
                "spec": {
                    "containers": [
                        {
                            "name": "${APP_NAME}",
                            "image": "${IMAGE_NAME}",
                            "resources": {}
                        }
                    ]
                }
            }
        },
        "status": {}
    }
],
"parameters": [
    {
        "name": "APP_NAME",
        "description": "The application name, used for naming pods and services",
        "value": "hello"
    },
    {
        "name": "IMAGE_NAME",
        "description": "The prebuilt image used to create this application pods",
        "value": "openshift/hello-openshift"
    }
]
}
```

With the addition of parameters, a resource definition file that generates an application from a fixed container image was changed into a template that can generate an application from any pre built container image. This is a simple use case that the **oc new-app** command can handle itself, but serves as a basis for creating templates for more complex use cases.

A template parameter can also have **generator** and **from** attributes. If these attributes are found instead of the **value** attribute, OSE generates as the default parameter value a random string that matches the regular expression given by the **from** attribute value. A common use case for this feature is generating passwords. For example, the **eap6-basic-sti** template includes the following parameter definition:

```
    {
        "name": "HORNETQ_CLUSTER_PASSWORD",
        "description": "HornetQ cluster admin password",
        "generate": "expression",
        "from": "[a-zA-Z0-9]{8}"
    },
```

A real-world template would probably add at least a service and a route definition to make the application accessible to either other pods or to external users. Many OSE client commands accept the **-o json** option. For example, the following command generates a service definition snippet that could be copied to a template definition:

```
$ oc expose -o json pod hello-1-2vyz5 --port=8080
```

A common strategy for building template definitions starts by manually creating each resource needed by the application using commands like **oc new-app**, **oc expose**, etc., and testing if the resources work as expected. After all tests are OK, use the **oc export** command with the **-o json** option to export the existing resource definitions, then the resource definitions are merged into a template definition file and parameters are added to the template. Finally the resource definition itself is tested in another project.

Best Practice

JSON syntax errors are not easy to identify and OSE is very picky about them, refusing JSON files most browsers would accept as valid. The **jsonlint -s** command from the **python-demjson** package, available from EPEL, will identify syntax issues in a JSON resource definition file.

Define to OSE in openshift namespace

A JSON template definition file can be imported into any project using the **oc create** command and then used to create new applications from either the web console or the **oc new-app** command. OSE also provides a predefined project named **openshift** where the standard templates are available and visible to all users, in all projects. This is where the instant apps and other templates shown in the web console come from.

An OSE instance administrator can add new templates to the **openshift** project using the **oc create** command with option **-n**; for example:

```
# oc create -f single-image.json -n openshift
```

If the previous command omits the **-n openshift** option, it creates the template in the current project, which can be useful for testing a template without making it available for all users, or to create multiple instances of the same application inside a single project.

Any user can list available templates using the**oc get templates** command:

```
$ oc get templates -n openshift
```

The result listing shows each template description, alongside the number of parameters and resources defined by the template.

To list all parameters defined by a template, use the **oc process** command. For example, any user could list all parameters from the **mysql-persistent** template using the following command:

```
$ oc process --parameters=true -n openshift mysql-persistent
```

Each parameter is shown with its description, default value (if defined), and generator expression (if defined).

To create an application from a template, either use the web console or use the **oc new-app** command, for example:

```
$ oc new-app --template=single-image \
```

```
    -p APP_NAME=myhello,IMAGE_NAME=openshift/hello-openshift
```

The previous command works if the **single_image** template is part of the current project and also if it is part of the **openshift** project; no need to use the **-n openshift** option.

A developer may want to perform additional customization over template resources before creating them. In this case, the template definition can be exported to a JSON file and then the modified template can be created inside another project. Another option would be to process the template directly from the JSON file; for example:

```
$ oc export -o json \
    -n openshift mysql-ephemeral > mysql-ephemeral.json
... change the mysql-ephemeral.json file ...
$ oc process -f mysql-ephemeral.json \
    -v MYSQL_DATABASE=testdb,MYSQL_USE=testuser,MYSQL_PASSWORD=secret > testdb.json
$ oc create -f testdb.json
```

Note

The **oc process** command uses the **-v** option to provide parameter values, while the **oc new-app** command uses the **-p** option for the same thing.

Test the template with the web console

The OpenShift web console project overview page has an **Add to Project** button to create new resources from templates. Templates from both the current project and from the **openshift** project are shown as options. Initially, only templates tagged as **instant apps** are shown. After listing all instant app templates, the web console provides the **Show All Templates** button that lists all available templates, no matter the tagging.

It is very comfortable using templates from the web console because after selecting a template, the web console shows all template parameters with their respective descriptions and allows changing its values in a wizard-like interface. But from the **oc new-app** command, all parameter values have to be provided as a single **-p** option argument, separated by commas. It is very hard to spot typos on such a command line.

Using a template as a starter application

OpenShift **instant apps** are templates designed to function as starter applications: they are supposed to provide not only all needed resources ready to run, with sensible default values, but also some sample source code. For this reason, an instant app will usually point to a public GitHub repository providing an example application ready to run.

Of course, creating an unchanged instant app is not that interesting for a developer, because it will just run the example application from the template. But a developer can get the example application Git repository URL from the template, clone it, and then provide a cloned repository as a template parameter. This way, the example application can be modified to suit the developer's needs and the modified application build can run inside OSE.

To tag a template as an instant app, add a **tags** attribute to the **annotation** object inside the template definition. For example, the **cakephp-example** predefined instant app template has a few tags:

```
$ oc export template cakephp-example -n openshift -o json | head -n 12
{
    "kind": "Template",
    "apiVersion": "v1",
    "metadata": {
        "name": "cakephp-example",
        "creationTimestamp": null,
        "annotations": {
            "description": "An example CakePHP application with no database",
            "iconClass": "icon-php",
            "tags": "instant-app,php,cakephp"
        }
    },
```

The **tags** attribute contains a comma-separated list of tag values, which are simple strings. Some of those values are used internally by OSE; for example, to choose a suitable template to build an application created from source code. If the **oc new-app** is invoked with only an application source repository URL, and it detects those sources are from a PHP application, it will search for a template tagged as **php**. If it also detects the application was built using the Cake PHP web application framework, it will search for templates tagged as **cakephp**.

Demonstration: Using a template as a starter application

In this example, please follow along with these steps while your instructor demonstrates how to create an application from a template that is based on a custom S2I builder image.

1. This demo uses the project and image stream created in the Guided Exercise about creating a custom builder image. If you do not have those, edit the script **publish-httpd-s2i.sh**:

   ```
   $ gedit /home/student/DO290/labs/custom-template-demo/publish-httpd-s2i.sh
   ```

   Update the X variable from 0 to the student's number and save it.

2. Edit the **httpd-s2i-is.json** file to point to the local docker registry.

   ```
   $ gedit /home/student/DO290/labs/custom-template-demo/httpd-s2i-is.json
   ```

   In the **dockerImageRepository** spec, change the *X* from the address **workstation.pod*X*.example.com:5000/httpd-s2i** with the student's number, and save the file. This will allow OSE to access the docker registry to download the image.

3. Run the script to push the image to the local Docker registry:

   ```
   $ sudo bash /home/student/DO290/labs/custom-template-demo/publish-httpd-s2i.sh
   ```

4. If needed, log into the OSE client as the developer user **student** and enter the **s2i-builder** project.

   ```
   $ oc login -u student -p redhat
   $ oc project s2i-builder
   ```

5. Check the **httpd-s2i** container image is available in the classroom private image registry.

```
$ sudo docker search httpd-s2i
```

6. Check that the **httpd-s2i** image stream points to the image found in the previous step.

```
$ oc export is httpd -o json
```

Verify the image stream definition includes the annotations that makes this a S2I builder.

7. Review the custom template that uses the image stream to provide an instant app for a static web site.

```
$ gedit /home/student/DO290/labs/custom-template-demo/httpd-s2i-template.json
```

Note

Instead of **gedit**, the **less** command or any other text editor can be used. But **gedit** provides syntax highlighting for JSON files.

Verify the template definition includes the annotations that make this template an OSE instant app. Update the *X* from the **httpd-s2i-template.json** file from the **workstation.pod*X*.example.com** and the **cloudapps*X*.example.com** with the student's number.

8. Create the template using the provided JSON file.

```
$ oc create -f \
    /home/student/DO290/labs/custom-template-demo/httpd-s2i-template.json
```

9. Open a web browser and enter the OSE web console at **http://master.pod0.example.com:8443/**.

10. Log into the web console as the developer user **student** using password **redhat** and enter the **s2i-builder** project.

11. In the project overview page, click the **Add to project** button.

12. Scroll down to the **Instant Apps** listing to find the box for the **httpd-s2i** template. Click it. Use the following figure as reference.

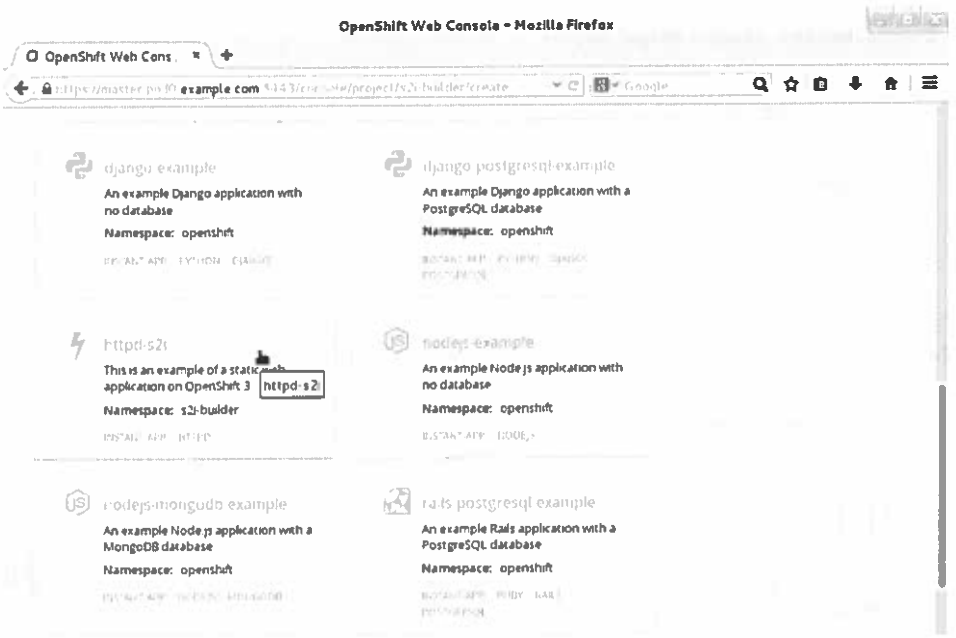


Figure 9.3: httpd-s2i instant app

13. Click the **Edit parameters** link and type "mysite" in both the **APPLICATION_NAME** and the **APPLICATION_HOSTNAME** fields. Do not touch any other field.

14. Scroll down to find the **Create** button. Click it.

15. The web console should display a message stating all template resources were created, then after a few moments change to indicate a build is running.

16. Wait until the build finishes. Use the following figure as a reference.

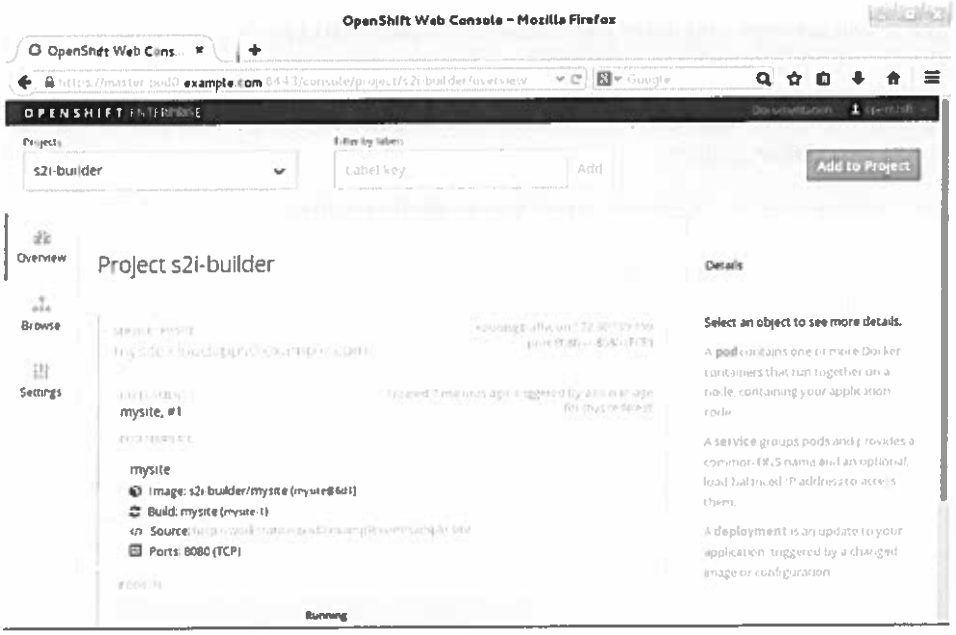


Figure 9.4: mysite application build and deployed

17. Click the link to **mysite.cloudapps0.example.com**. The expected result is the "Sample Web Site" welcome page.

18. Clean up the environment by running:

```
$ oc delete project sti-builder
```

This concludes the demonstration.

Guided Exercise: Create a Custom Template

In this lab, you will create a template to create applications from static web pages.

Resources	
Files	**/home/student/DO290/labs/custom-template** (workstation VM)
	/root/DO290/labs/custom-template (master VM)
Application URL	NA

Outcome(s)
A custom template, based on a custom S2I builder image, available to all users, and an application created from that custom template.

Before you begin
Have an OSE instance installed with router and registry and the **httpd-s2i** builder image pushed to the classroom internal registry.

If you do not have that image, run the **/home/student/DO290/labs/custom-template/publish-httpd-s2i.sh** script using **sudo**.

1. Create The Custom Template as a Global Template
   Create the image stream and the template that uses the custom builder image in the **openshift** project so they are available to all users in all projects.

   1.1. Open a SSH session to the **master** VM and log in as **root** using password **redhat**.

   ```
   [student@workstation ~]$ ssh root@master.podX.example.com
   ```

   Replace the *X* with the student's number.

   1.2. On the **master** VM, log into OSE as the instance administrator user. All remaining **1.x** steps are done on the **master** VM.

   ```
   [root@master ~]# oc login -u system:admin
   ```

   1.3. Verify the **httpd-s2i** image is available from the classroom internal registry.

   ```
   [root@master ~]# docker search httpd-s2i
   ```

   If it is not, please check this exercise's prerequisites.

   1.4. Edit the image stream resource definition provided as the **httpd-s2i-is.json** file in the **/root/DO290/labs/custom-template** folder.
   - Change the **dockerImageRepository** attribute value, replacing ***X*** with your student number.
   - Verify the same attribute refers to the **httpd-s2i** image.

   Use the following partial listing as a reference.

```
...
    "spec": {
        "dockerImageRepository": "workstation.podX.example.com:5000/httpd-s2i",
        "tags": [
...
```

1.5. Create the image stream from the provided resource definition file in the **openshift** project.

```
[root@master ~]# oc create -n openshift -f \
    /root/DO290/labs/custom-template/httpd-s2i-is.json
```

Best Practice

Although a template can include an image stream resource definition, it is recommended that the image stream be created outside the template. This makes easier to change the image repository used to get the images without having to change potentially many templates that could refer to the same image.

1.6. Edit the template resource definition provided as the **httpd-s2i-template.json** file in the **/root/DO290/labs/custom-template** folder.

- Change the embedded route **host** attribute value, replacing ***X*** with your student number.
- Verify the embedded build configuration refers to the **httpd** image stream in the **openshift** project.
- Change the **GIT_URI** parameter default value replacing ***X*** with your student number.

Use the following partial listing as a reference.

```
...
        {
            "kind": "Route",
...
            "spec": {
                "host": "${APPLICATION_HOSTNAME}.cloudappsX.example.com",
...
        {
            "kind": "BuildConfig",
...
                    "sourceStrategy": {
                        "from": {
                            "kind": "ImageStreamTag",
                            "name": "httpd:latest",
                            "namespace": "openshift"
...
    ],
    "parameters": [
...
        {
            "name": "GIT_URI",
            "description": "GIT URI",
```

```
            "value": "http://workstation.podX.example.com/sample-site.git"
        },
...
```

1.7. Create the template from the provided resource definition file in the **openshift** project.

```
[root@master ~]# oc create -n openshift -f \
    /root/DO290/labs/custom-template/httpd-s2i-template.json
```

1.8. Terminate the SSH session.

```
[root@master ~]# logout
```

You should be back to the **student** VM shell prompt.

2. Create a New Application Using the Custom Template
Use the **oc new-app** command to create a new application using the custom template, but using as source code the static website on Git repository **http://workstation.pod*X*.example.com/mysite.git**, replacing the *X* with the student's number.

2.1. Log in as the developer user **student** and create a new project for the static web site application.

```
[student@workstation ~]$ oc login -u student -p redhat
[student@workstation ~]$ oc new-project mysite
```

2.2. Verify the **httpd** image stream exists in the **openshift** project.

```
[student@workstation ~]$ oc get is -n openshift | grep httpd
```

The expected output is similar to:

```
httpd                                 workstation.podX.example.com:5000/httpd-s2i
                              2.4,latest    10 seconds ago
```

2.3. Verify the **httpdp-s2i** template exists in the **openshift** project.

```
[student@workstation ~]$ oc get template -n openshift | grep httpd
```

The expected output is similar to:

```
httpd-s2i                                  This is an example of a static web
 application on OpenShift 3                        7 (3 blank)       5
```

2.4. Inspect the template parameters.

```
[student@workstation ~]$ oc process --parameters=true -n openshift httpd-s2i
```

The expected output follows. Only the first parameters are shown because the others will be left as is:

```
NAME                     DESCRIPTION
                GENERATOR           VALUE
APPLICATION_NAME         Application NAME

APPLICATION_HOSTNAME     Application hostname

GIT_URI                  GIT URI
                                    http://workstation.podX.example.com/sample-
site.git
...
```

2.5. Copy the provided application creation script **new-app-mysite.sh**.

```
[student@workstation ~]$ cp /home/student/DO290/labs/custom-template/new-app-
mysite.sh .
```

Important

Do not forget the current directory (.) as the last **cp** argument.

2.6. Edit the copy so the parameter values (in option **-p**) are as follows, replacing the *X* with the student's number:

- **APP_NAME=mysite**
- **APP_HOSTNAME=mysite**
- **GIT_URI=http://workstation.pod*X*.example.com/mysite.git**

  Replace *X* with your student number.

Use the following listing as a reference.

```
#!/bin/bash

# do not add white space to this command
oc new-app --template=httpd-s2i -p \
APPLICATION_NAME=mysite,\
APPLICATION_HOSTNAME=mysite,\
GIT_URI=http://workstation.podX.example.com/mysite.git
```

2.7. Use the edited copy of the script to create the application using the custom template.

```
[student@workstation ~]$ bash new-app-mysite.sh
```

2.8. A build should start after a few moments. Wait until the build finishes and an application pod is running, using the **oc status** and **oc get pods** commands.

```
[student@workstation ~]$ oc status ; oc get pods
```

Wait until the output is similar to:

```
In project mysite

service/mysite - 172.30.175.63:8080
  dc/mysite deploys istag/mysite:latest <-
    builds http://workstation.podX.example.com/mysite#master with openshift/
httpd:latest through bc/mysite
    #1 deployed 32 seconds ago - 1 pod

To see more, use 'oc describe <resource>/<name>'.
You can use 'oc get all' to see a list of other objects.
NAME             READY     STATUS       RESTARTS   AGE
mysite-1-build   0/1       ExitCode:0   0          3m
mysite-1-euk9q   1/1       Running      0          27s
```

One build pod should be finished with **ExitCode:0** and one application pod should be running and ready.

2.9. Open a web browser and navigate to the **http://mysite.cloudappsX.example.com** URL, replacing the X with your student number. The expected result is a Hello, world HTML page.

3. Modify The Application And Deploy Changes
Change the application HTML page and deploy the changes to OSE.

3.1. Clone the application Git repository to the student workstation. Replace *X* with your student number.

```
[student@workstation ~]$ git clone http://workstation.podX.example.com/
mysite.git
```

3.2. Enter the local repository folder and change the **index.html** file inside the application, adding your name. Use the following listing as a reference.

```
<html>
<body>
<h1>My Web Site</h1>
<hr/>
Hello, Student!
</body>
</html>
```

3.3. Push changes to the classroom Git server.

```
[student@workstation mysite]$ git commit -a -m "changed welcome page"
[student@workstation mysite]$ git push
```

3.4. Start a new application build, and wait until it finishes.

```
[student@workstation mysite]$ oc start-build mysite --follow
```

3.5. Wait until a new application pod is running, using the instructions from **step 2.7**. The output should be similar to:

```
In project mysite

service/mysite - 172.30.175.63:8080
  dc/mysite deploys istag/mysite:latest <-
    builds http://workstation.pod0.example.com/mysite#master with openshift/
httpd:latest through bc/mysite
    #2 deployed about a minute ago - 1 pod
    #1 deployed 5 minutes ago

To see more, use 'oc describe <resource>/<name>'.
You can use 'oc get all' to see a list of other objects.
NAME              READY     STATUS      RESTARTS   AGE
mysite-1-build    0/1       ExitCode:0  0          8m
mysite-2-build    0/1       ExitCode:0  0          1m
mysite-2-lf2z4    1/1       Running     0          1m
```

The second build should be finished with **`ExitCode:0`** and a single application pod should be running and ready.

3.6. Refresh the application welcome page in the web browser. It should now display your name.

4. Clean up

Delete all artifacts created by this lab to keep things tidy.

4.1. Delete the project created in step **2**.

```
[student@workstation mysite]$ cd
[student@workstation ~]$ rm -rf mysite
[student@workstation ~]$ oc delete project mysite
```

4.2. Open a new SSH session to the master VM and delete the image stream and template created in step **1**. Replace the *X* with the student's number. The password for the **ssh** command is **redhat**.

```
[student@workstation ~]$ ssh root@master.podX.example.com
[root@master ~]# oc login -u system:admin
[root@master ~]# oc delete template httpd-s2i -n openshift
[root@master ~]# oc delete is httpd -n openshift
[root@master ~]# logout
```

This concludes the guided exercise.

Lab: Creating and Deploying a Custom S2I Builder

In this lab, you will create an S2I builder image and a template to add an OSE instant app that creates static web sites.

Resources	
Files	`/home/student/DO290/labs/custom-containers`
Application URL	`http://customsite.cloudapps0.example.com`

Outcome(s)
A custom template, based in a custom S2I builder image, available to all users, and an application created from that custom template.

Custom S2I builder image
The **`httpd-s2i`** custom S2I builder image takes static HTML pages as source code and includes an Apache Httpd web server.

- Is based on the **`base-rhel7`** container image, whose Dockerfile is available at:

  `http://workstation.pod`*X*`.example.com:5000/sti-base`

- The **`httpd-s2i`** container image Dockerfile is at the following location:

  `http://workstation.pod`*X*`.example.com:5000/httpd-s2i`

- The **`httpd-s2i`** container image should be pushed to the private registry at:

  `workstation.pod`*X*`.example.com:5000`

Custom template
The custom template adds an OSE instant app to create static web sites.

- The **`httpd-s2i`** image is the same used by the application builder image.
- The template does not reference the builder image directly, but instead references an image stream that references the builder image.
- A JSON resource definition file for the builder image stream can be found as **`httpd-s2i-is.json`** in the labs folder.
- A sample web site to use as a starter application is available at:

  `http://workstation.pod`*X*`.example.com:5000/sample-site`

- A JSON resource definition file for the template can be found as **`httpd-s2i-template.json`** in the labs folder.

Before you begin
Have a local Docker installation with **sudo** access, the **`s2i`** utility installed (or access to the binaries), OpenShift Enterprise (OSE) installed, an OSE developer user, and Git client available.

1. Install The **s2i** Utility
   The **s2i** utility is currently not packaged as part of the RHEL7 or OSE3 repositories, so it need to be installed from the binary distribution.

2. Build The base-rhel7 Container Image
   The **base-rhel7** container image is the base image for all OSE3 builder images. It contains common files and scripts to use the **Red Hat Software Collections Library** (SCL) that allow using newer releases of application runtimes and databases, even if they were not packaged originally as part of a RHEL release series.

3. Build The httpd-s2i Builder Image
   The **httpd-s2i** builder image takes static HTML files as source code and adds them to an Apache Httpd server. The web server is installed from SCL packages instead of from regular RHEL7 packages.

4. Make The New Builder Image Available To OSE
   Push the new S2I builder image to the classroom private registry and create the image stream in the **openshift** project.

5. Create The Custom Template as a Global Template
   Create the template that uses the custom builder image in the **openshift** project so they are available to all users in all projects.

6. Create A New Application Using The Custom Template
   Use the web console to create a new application from the **httpd-s2i** template, which should be shown as an instant app.

   This concludes the exercise.

Solution

In this lab, you will create an S2I builder image and a template to add an OSE instant app that creates static web sites.

Resources	
Files	**/home/student/DO290/labs/custom-containers**
Application URL	**http://customsite.cloudapps0.example.com**

Outcome(s)
A custom template, based in a custom S2I builder image, available to all users, and an application created from that custom template.

Custom S2I builder image
The **httpd-s2i** custom S2I builder image takes static HTML pages as source code and includes an Apache Httpd web server.

- Is based on the **base-rhel7** container image, whose Dockerfile is available at:

  http://workstation.pod*X*.example.com:5000/sti-base

- The **httpd-s2i** container image Dockerfile is at the following location:

  http://workstation.pod*X*.example.com:5000/httpd-s2i

- The **httpd-s2i** container image should be pushed to the private registry at:

  workstation.pod*X*.example.com:5000

Custom template
The custom template adds an OSE instant app to create static web sites.

- The **httpd-s2i** image is the same used by the application builder image.
- The template does not reference the builder image directly, but instead references an image stream that references the builder image.
- A JSON resource definition file for the builder image stream can be found as **httpd-s2i-is.json** in the labs folder.
- A sample web site to use as a starter application is available at:

  http://workstation.pod*X*.example.com:5000/sample-site

- A JSON resource definition file for the template can be found as **httpd-s2i-template.json** in the labs folder.

Before you begin
Have a local Docker installation with **sudo** access, the **s2i** utility installed (or access to the binaries), OpenShift Enterprise (OSE) installed, an OSE developer user, and Git client available.

1. Install The **s2i** Utility
   The **s2i** utility is currently not packaged as part of the RHEL7 or OSE3 repositories, so it need to be installed from the binary distribution.

   1.1. Extract the **s2i** binary distribution from the tar file.

```
[student@workstation ~]$ tar xzf \
  /home/student/DO290/installers/source-to-image-v1.0.2-00d1cb3-linux-
amd64.tar.gz
```

1.2. Move the **s2i** and utility binary file to the local shared programs folder and delete its alias **sti**.

```
[student@workstation ~]$ sudo mv s2i /usr/local/bin
[student@workstation ~]$ rm sti
```

1.3. Go back to the student user home folder and verify the **s2i** utility is available on the user **PATH**.

```
[student@workstation ~]$ s2i version
```

The expected output is:

```
s2i v1.0.2
```

2. Build The base-rhel7 Container Image
The **base-rhel7** container image is the base image for all OSE3 builder images. It contains common files and scripts to use the **Red Hat Software Collections Library** (SCL) that allow using newer releases of application runtimes and databases, even if they were not packaged originally as part of a RHEL release series.

2.1. Go back to the student user home folder and fetch the **sti-base** repository from the classroom Git server. Remember to replace *X* with your student number.

```
[student@workstation ~]$ git clone \
  http://workstation.podX.example.com/sti-base.git
```

2.2. Inspect the **Dockerfile.rhel7** file that builds the **base-rhel7** container image. It was modified to fit the classroom environment.

```
[student@workstation ~]$ cd sti-base
[student@workstation sti-base]$ less Dockerfile.rhel7
```

2.3. Fetch the **Dockerfile.rhel7** base image **rhel7** from the classroom private image repository. Remember to replace *X* with your student number.

```
[student@workstation sti-base]$ sudo docker \
  pull workstation.podX.example.com:5000/rhel7
```

2.4. Build the **base-rhel7** container image using Docker.

```
[student@workstation sti-base]$ sudo docker build \
 -t openshift/base-rhel7 -f Dockerfile.rhel7 .
```

Note

Do not forget the current directory (.) as the last command argument.

3. Build The httpd-s2i Builder Image
The **httpd-s2i** builder image takes static HTML files as source code and adds them to an Apache Httpd server. The web server is installed from SCL packages instead of from regular RHEL7 packages.

   3.1. Go back to the student user home folder and fetch the **httpd-s2i** repository from the classroom Git server. Remember to replace *X* with your student number.

```
[student@workstation sti-base]$ cd
[student@workstation ~]$ git clone http://workstation.podX.example.com/httpd-
s2i.git
```

   3.2. Inspect the **Dockerfile** file that builds the **httpd-s2i** container image. It already fits the classroom environment.

```
[student@workstation ~]$ cd httpd-s2i
[student@workstation httpd-s2i]$ less Dockerfile
```

   Verify it already includes the S2I required label **io.openshift.tags="builder,httpd"**.

   3.3. Verify the project already includes S2I scripts in the **.s2i** hidden folder in the project root.

```
[student@workstation httpd-s2i]$ ls .s2i/bin
```

   3.4. Build the **httpd-s2i** container image using Docker.

```
[student@workstation httpd-s2i]$ sudo docker build -t httpd-s2i .
```

Note

Do not forget the current directory (.) as the last command argument.

4. Make The New Builder Image Available To OSE
Push the new S2I builder image to the classroom private registry and create the image stream in the **openshift** project.

   4.1. Tag and upload (push) the new builder image to the classroom private image registry so it becomes available to the OSE instance. Remember to replace *X* with your student number.

```
[student@workstation ~]$ sudo docker tag httpd-s2i:latest
 workstation.podX.example.com:5000/httpd-s2i:latest
[student@workstation ~]$ sudo docker push workstation.podX.example.com:5000/
httpd-s2i:latest
```

4.2. Open an SSH session to the **master** VM and log in as **root** using password **redhat**, replacing the *X* with the student's number:

```
[student@workstation ~]$ ssh root@master.podX.example.com
```

4.3. On the **master** VM, log into OSE as the instance administrator user.

```
[root@master ~]# oc login -u system:admin
```

4.4. Edit the image stream resource definition provided as the **httpd-s2i-is.json** file in the **/root/DO290/labs/custom-containers** folder.

- Change the **dockerImageRepository** attribute value, replacing ***X*** with your student number.
- Verify the the **dockerImageRepository** attribute value refers to the **httpd-s2i** image.

Use the following partial listing as a reference.

```
...
    "spec": {
        "dockerImageRepository": "workstation.podX.example.com:5000/httpd-s2i",
        "tags": [
...
```

4.5. Verify the image stream already includes the annotations required to be used as an S2I builder.

```
...
                "annotations": {
                    "description": "Build and run static web apps served by
 httpd 2.4",
                    "supports": "httpd,httpd:2.4",
                    "tags": "builder,httpd",
                    "version": "2.4"
...
```

4.6. Create the image stream from the provided resource definition file in the **openshift** project.

```
[root@master ~]# oc create -n openshift -f \
    /root/DO290/labs/custom-containers/httpd-s2i-is.json
```

5. Create The Custom Template as a Global Template
Create the template that uses the custom builder image in the **openshift** project so they are available to all users in all projects.

5.1. Edit the template resource definition provided as the **httpd-s2i-template.json** file in the **/root/DO290/labs/custom-containers** folder.

- Change the embedded route **host** attribute value, replacing ***X*** with your student number.
- Verify the embedded build configuration refers to the **httpd** image stream in the **openshift** project.
- Change the **GIT_URI** parameter default value replacing ***X*** with your student number.

Use the following partial listing as a reference.

```
...
        {
            "kind": "Route",
...
            "spec": {
                "host": "${APPLICATION_HOSTNAME}.cloudappsX.example.com",
...
        {
            "kind": "BuildConfig",
...
                    "sourceStrategy": {
                        "from": {
                            "kind": "ImageStreamTag",
                            "name": "httpd:latest",
                            "namespace": "openshift"
...
    ],
    "parameters": [
...
        {
            "name": "GIT_URI",
            "description": "GIT URI",
            "value": "http://workstation.podX.example.com/sample-site"
        },
...
```

5.2. Verify the template already includes the annotations required to be used as an instant app. Use the following partial listing as reference:

```
...
        "annotations": {
            "description": "This is an example of a static web application on
 OpenShift 3",
            "tags": "instant-app,httpd"
...
```

5.3. Create the template from the provided resource definition file in the **openshift** project.

```
[root@master ~]# oc create -n openshift -f \
     /root/DO290/labs/custom-containers/httpd-s2i-template.json
```

5.4. Terminate the SSH session.

```
[root@master ~]# logout
```

You should be back to the **student** VM shell prompt.

6. Create A New Application Using The Custom Template
Use the web console to create a new application from the **httpd-s2i** template, which should be shown as an instant app.

6.1. Open a web browser and enter the OSE web console at:

https://master.pod*X*.example.com:8443/.

Replacing the *X* with the student's number.

6.2. Log into the web console the developer user **student** using password **redhat** and create a new project named **custom-container**.

6.3. In the project overview page, click the **Add to project** button.

6.4. Scroll down to the **Instant Apps** listing to find the box for the **httpd-s2i** template. Click it.

6.5. Click the **Edit parameters** link and type **customsite** in both the **APPLICATION_NAME** and the **APPLICATION_HOSTNAME** fields. Do not touch any other field.

6.6. Scroll down to find the **Create** button. Click it.

The web console should display a message stating all template resources were created, then after a few moments change to indicate a build is running.

6.7. Wait until the build finishes and an application pod is running.

6.8. Click the link to:

customsite.cloudapps*X*.example.com

The expected result is the "Sample Web Site" welcome page.

This concludes the exercise.

Summary

In this chapter, you learned:

- How to create applications from Dockerfiles: (Create a Dockerfile, commit to a Git repository and deploy it using OSE **`oc new-app <gitRepository>`**).
- How to create and use custom S2I builder images .
- To read a template example as a JSON file with parameters, upload the template to OSE (using **`oc create -f <jsonFile>`**), and use custom templates and instant apps from the web console.

CHAPTER 10

COMPREHENSIVE REVIEW OF OPENSHIFT ENTERPRISE DEVELOPMENT

Overview	
Goal	Practice the skills learned during this course.
Objectives	• Deploy a multi-pod application to OpenShift Enterprise. • Implement Jenkins CI for the application. • Integrate Jenkins with OSE builds
Labs	• Comprehensive Review: Creating an Application with a Template • Comprehensive Review: Implementing CI for the Application

Comprehensive Review: Creating an Application with a Template

In this comprehensive review, you will deploy a Java application using templates. The **member** application was developed using Java connecting to a MySQL database through Hibernate.

Resources	
Files	**/home/student/DO290/labs/review-deploy** (workstation VM)
Application URL	NA

Outcome(s)
You should be able to:

- Create secrets.
- Deploy an application using templates.

Before you begin
OSE installed with router and registry, and NFS server with persistent volume configured on the master.

1. On the workstation, create a new project called **member**.
2. Provide the Template Prerequisites.

   The **eap6-mysql-persistent-sti** standard OSE template needs a persistent volume to be available and a secret to provide SSL certificates.
3. Create the application using the **eap6-mysql-persistent-sti** template.
4. Verify if the build has started.

   Check the build logs.
5. Verify the JBoss startup log. Find the pod name and inspect its logs.
6. Test the Application.
7. Clean Up.

   This project will not be reused for the next review lab, so delete it to keep things tidy and your environment light.

   This concludes the review lab.

Solution

In this comprehensive review, you will deploy a Java application using templates. The **member** application was developed using Java connecting to a MySQL database through Hibernate.

Resources	
Files	**/home/student/DO290/labs/review-deploy** (workstation VM)
Application URL	NA

Outcome(s)
You should be able to:

- Create secrets.
- Deploy an application using templates.

Before you begin
OSE installed with router and registry, and NFS server with persistent volume configured on the master.

1. On the workstation, create a new project called **member**.

   1.1. Log in as the developer user:

   ```
   [student@workstation ~]$ oc login -u student -p redhat
   ```

   ### Note

   If this is your first login to OSE (ex: you reset the VMs before starting this lab) you will need to add the OSE master URL **https://master.pod0.example.com:8443/** to the previous command and answer **y** to the prompt about using insecure connections.

   1.2. Create the project:

   ```
   [student@workstation ~]$ oc new-project member
   ```

2. Provide the Template Prerequisites.

   The **eap6-mysql-persistent-sti** standard OSE template needs a persistent volume to be available and a secret to provide SSL certificates.

   2.1. Inspect the secret definition:

   ```
   [student@workstation ~]$ less /home/student/DO290/labs/review-deploy/secret.json
   ```

   The file contents are:

   ```
   {
       "kind": "List",
   ```

```
        "apiVersion": "v1",
        "metadata": {},
        "items": [
            {
                "kind": "ServiceAccount",
                "apiVersion": "v1",
                "metadata": {
                    "name": "eap-service-account"
                },
                "secrets": [
                    {
                        "name": "eap-app-secret-member"
                    }
                ]
            },
            {
                "kind": "Secret",
                "apiVersion": "v1",
                "metadata": {
                    "annotations": {
                        "description": "Default secret file with name 'jboss' and
 password 'mykeystorepass'"
                    },
                    "name": "eap-app-secret-member"
                },
                "data": {
                    "keystore.jks": "<reallyLargeToken>"
                }
            }
        ]
}
```

Take a note about three important items:

- Secret name: **eap-app-secret-member**.
- Keystore alias: **jboss**.
- Keystore password: **mykeystorepass**.

2.2. Create the secret using the provided JSON resource definition file:

```
[student@workstation ~]$ oc create -f \
/home/student/DO290/labs/review-deploy/secret.json
```

2.3. Create the persistent volume (PV).

```
[student@workstation ~]$ bash /home/student/DO290/labs/review-deploy/createpv.sh
```

3. Create the application using the **eap6-mysql-persistent-sti** template.

3.1. List the parameters available from **eap6-mysql-persistent-sti**:

```
[student@workstation ~]$ oc process \
  --parameters eap6-mysql-persistent-sti -n openshift
```

Note

The parameters with prefix **EAP_HTTPS** refer to the secret definition inspected in **step 2.1**.

3.2. Export the **eap6-mysql-persistent-sti** template:

```
[student@workstation ~]$ oc -o json export \
    template eap6-mysql-persistent-sti -n openshift > \
    /home/student/DO290/labs/review-deploy/eap6-mysql-persistent-sti.json
```

3.3. Edit the file **/home/student/DO290/labs/review-deploy/template.sh** and replace the *X* from the **APPLICATION_HOSTNAME** and **GIT_URI** parameters. Then execute the script to create a resource definition file:

```
[student@workstation ~]$ bash /home/student/DO290/labs/review-deploy/template.sh
```

Note

The parameters related to the secret definition and the database pod already have correct values and do not need to be changed.

3.4. Deploy the application using the JSON resource definition file created by the script during the previous step:

```
[student@workstation ~]$ oc create -f \
  /home/student/DO290/labs/review-deploy/processedtemplate.json
```

The expected output is:

```
services/member
services/secure-member
services/member-ping
services/member-mysql
routes/member-http-route
routes/member-https-route
imagestreams/member
buildconfigs/member
deploymentconfigs/member
deploymentconfigs/member-mysql
persistentvolumeclaims/member-mysql-claim
```

4. Verify if the build has started.

```
[student@workstation ~]$ watch oc get builds
NAME       TYPE      STATUS    POD
member-1   Source    Running   member-1-build
```

Check the build logs.

```
[student@workstation ~]$ oc build-logs member-1
```

Note

The build can take some time to start.

5. Verify the JBoss startup log. Find the pod name and inspect its logs.

```
[root@workstation ~]$ oc get pods
NAME                READY     REASON          RESTARTS   AGE
member-1-build      0/1       ExitCode:0      0          20m
member-1-m6ynh      0/1       Running         0          37s
member-mysql-1-28zyg   1/1    Running         0          20m
```

```
[student@workstation ~]$ oc logs member-1-m6ynh
```

Make sure the EAP server startup finished without errors before moving to the next step.

6. Test the Application.

Open a web browser (**Applications** > **Internet** > **Firefox**) on workstation host and open the following URL: **http://member.cloudappsX.example.com**. Replace *X* with your student number. Test the member application.

7. Clean Up.

This project will not be reused for the next review lab, so delete it to keep things tidy and your environment light.

7.1. Delete the **member** project:

```
[student@workstation ~]$ oc delete project member
```

7.2. Delete the PV on the master using the provided Bash script:

```
[student@workstation ~]$ bash /home/student/DO290/labs/review-deploy/deletepv.sh
```

This concludes the review lab.

Comprehensive Review: Implementing CI for the Application

In this comprehensive review, you will build the member application with Jenkins and make it deploy the generated WAR file to OSE inside an EAP pod.

Resources	
Files	**`/home/student/DO290/labs/review-jenkins`**
Application URL	**`http://jenkins.cloudappsX.example.com`**

Outcome
The student will be able to trigger a deployment using Jenkins running inside OSE.

Before you begin
OpenShift must be installed and running.

The OpenShift client must be installed at the workstation machine.

1. Create a New OSE Project
   From the workstation VM, open a Terminal window, log into OSE, and create project **`member-ci`**.

2. Deploy Jenkins
   Deploy Jenkins using the customized container image **`openshift/jenkins-ose-dev`** available from the classroom private registry at **`workstation.pod0.example.com:5000`**.

3. Create a Jenkins Project to Build the Member Application
   Usually a Jenkins project would perform unit and integration tests, but we will skip those to keep this lab shorter. Create a Jenkins project named **`member-war`** just to build the member application using Maven and expose the generated WAR file for reuse.

4. Verify Jenkins Can Build the Member Application
   Before moving on, make sure the **`member-war`** Maven Jenkins project created during the previous step is configured correctly to build the member application from Git using Maven.

5. Customize the Member Application S2I Scripts
   Use the provided **`assemble`** script that keeps the OSE S2I builder from building the application sources again, but fetches the WAR file generated by Jenkins and deploys it into EAP inside the pod.

6. Deploy the Member Application to OSE
   Deploy the member application from the WAR file generated by OSE using a customized build configuration derived from the standard EAP templates. The build pod needs to know the URL to download the WAR artifact from Jenkins. This information is provided by the **`APPLICATION_ARTIFACT_URL`** environment variable.

7. Verify the Member Application Is Running
   Now we should have the member application pod running alongside the Jenkins pod as part of the same OSE project.

8. Create a Jenkins Project to Update the Application Inside OSE
   Now we want Jenkins to trigger new deploys for the application inside OSE to enable building a simple Continuous Integration (CI)/Continuous Delivery (CD) pipeline. To do this, create a new Jenkins project named **member-ose** that uses an OSE web hook to trigger a new OSE build.

9. Verify Jenkins Can Trigger an OSE Build
   Before moving on, make sure the **member-ose** Freeform Jenkins project created during the previous step is configured correctly to trigger a build in the OSE **member-ci** project.

10. Connect Both Jenkins Projects to Complete The CI/CD Pipeline
    To have a full though simple CI/CD pipeline, we need to connect both Jenkins projects so when the member application is rebuilt by the **member-war** Maven Jenkins project, a new build is triggered in the **member-ose** jenkins project and, as a consequence, a new OSE build is started.

11. Verify Building the Member Application in Jenkins Creates New Pods in OSE
    Now test everything together; building the member application from its sources should deploy new application pods in OSE.

    This concludes the review.

Solution

In this comprehensive review, you will build the member application with Jenkins and make it deploy the generated WAR file to OSE inside an EAP pod.

Resources	
Files	**/home/student/DO290/labs/review-jenkins**
Application URL	**http://jenkins.cloudappsX.example.com**

Outcome
The student will be able to trigger a deployment using Jenkins running inside OSE.

Before you begin
OpenShift must be installed and running.

The OpenShift client must be installed at the workstation machine.

1. Create a New OSE Project
   From the workstation VM, open a Terminal window, log into OSE, and create project **member-ci**.

   1.1. Log in as the developer user created by the custom OSE install script:

   ```
   [student@workstation ~]$ oc login -u student -p redhat
   ```

   Note

   If this is your first login to OSE (ex: you reset the VMs before starting this lab), you will need to add the OSE master URL **https://master.pod0.example.com:8443/** to the previous command and answer **y** to the prompt about using insecure connections.

   1.2. Create the new project in OSE.

   ```
   [student@workstation ~]$ oc new-project member-ci
   ```

2. Deploy Jenkins
   Deploy Jenkins using the customized container image **openshift/jenkins-ose-dev** available from the classroom private registry at **workstation.pod0.example.com:5000**.

   2.1. Deploy the Jenkins image using the **oc new-app** command. Replace *X* with your student number:

   ```
   [student@workstation ~]$ oc new-app \
       workstation.podX.example.com:5000/openshift/jenkins-ose-dev --name=jenkins
   ```

   2.2. Create a route to allow external access to the Jenkins pod. Replace *X* with your student number:

```
[student@workstation ~]$ oc expose svc jenkins \
  --hostname=jenkins.cloudappsX.example.com
```

2.3. Make sure Jenkins finished starting before moving to the next step. Find the name of the Jenkins pod and check its logs:

```
[student@workstation ~]$ oc get pods
NAME              READY     STATUS    RESTARTS   AGE
jenkins-1-ykuaz   1/1       Running   0          23s
[student@workstation ~]$ oc logs jenkins-1-ykuaz | tail -n 2
Oct 03, 2015 8:10:36 PM hudson.WebAppMain$3 run
INFO: Jenkins is fully up and running
```

Repeat those steps until the logs show the **`Jenkins is fully up and running`** message.

3. Create a Jenkins Project to Build the Member Application
Usually a Jenkins project would perform unit and integration tests, but we will skip those to keep this lab shorter. Create a Jenkins project named **`member-war`** just to build the member application using Maven and expose the generated WAR file for reuse.

3.1. Open Jenkins in a web browser.

Use the **`http://jenkins.cloudappsX.example.com`** URL. Log in using user **`admin`** and password **`password`**.

3.2. Configure Maven inside Jenkins.

Click **Manage Jenkins** and then **Click Configure System**. Scroll down to **Maven** and click **Add Maven**. Type **`local maven`** in the **Name** field. Clear **Install automatically** and type **`/opt/apache-maven-3.0.5`** in the MAVEN_HOME field.

Scroll down and click **Save**.

3.3. Create the Jenkins project.

Click **New Item**, type **`member-war`** as the project name, and select the **Maven project** radio box. Click **Ok**.

3.4. Configure the **`member-war`** project to fetch the member application from the classroom Git server.

Select the **Source Code Management** option **Git** and type as the **Repository URL** the URL to the member application in the classroom Git server: **`http://workstation.podX.example.com/member.git`**. Replace *X* with your student number.

3.5. Make sure this build runs only when requested.

Scroll down to **Build Triggers** and clear all boxes. Usually a Jenkins project would be configured with either a schedule or a trigger to start builds, but this lab will use only manual builds to keep things simple.

3.6. Configure the **`member-war`** project to build using Maven.

Scroll down to **Build** and type **`clean package`** in the **Goals and options** field.

3.7. Configure the **`member-war`** project to keep the WAR file.

Scroll down to **Post-build Actions**, click **Add post-build action** and, select **Archive the artifacts**. Type **`target/*.war`** in the **Files to archive** field.

3.8. Save the **`member-war`** project.

Click the **Save** button to save the **`member-war`** project. You will be left in the **Maven project member-war** page.

4. Verify Jenkins Can Build the Member Application
Before moving on, make sure the **`member-war`** Maven Jenkins project created during the previous step is configured correctly to build the member application from Git using Maven.

4.1. Start a Jenkins build.

You should be in the **Maven project member-war** page. If not, click the **Jenkins** link in the top-left corner to see the Jenkins project listing and then click the **member-war** link.

Click **Build Now**. In a short time, a progress bar should be seen in the **Build History** box.

4.2. Verify the Jenkins build logs.

Click the build number such as **#1** in the **Build History** box and then click the **Console Output** link. This shows the build logs where you should see output from the **`git clone`** and **`maven clean package`** commands started by Jenkins.

Wait until the **`Finished: SUCCESS`** message is shown in the **Console Output** page. You may need to scroll down the page.

4.3. Verify the WAR file was kept available.

This file will be needed to create the EAP pod so the Maven build is not done again by OSE.

After the build finishes successfully, scroll up the **Console Output** page and click the **Back to Project** link. You should see the **member.war** link below **Last Successful Artifacts**. Click the link to **member.war**. The web browser should offer to download the file. Cancel the download.

Insight

Note that OSE builds, such as the one from the previous **Comprehensive Review Lab**, generate the application artifact as **ROOT.war**. This happens because the Maven project POM file (**pom.xml** in the project source root) contains an **openshift** Maven profile that is used by OSE builds, and this profile changes the name of the generated artifact.

The Maven project POM file could be changed so normal Maven builds (outside OSE) also generates **ROOT.war** but this was not done to highlight there are ways to make OSE builds behave different from Jenkins builds.

5. Customize the Member Application S2I Scripts
Use the provided **assemble** script that keeps the OSE S2I builder from building the application sources again, but fetches the WAR file generated by Jenkins and deploys it into EAP inside the pod.

   5.1. Clone the member application to a local Git repo. Replace *X* with your student number.

```
[student@workstation ~]$ git clone http://workstation.podX.example.com/
member.git
```

   5.2. Copy the custom **assemble** script to folder **.sti/bin** inside the cloned Git repo.

```
[student@workstation ~]$ mkdir -p member/.sti/bin
[student@workstation ~]$ cp /home/student/DO290/labs/review-jenkins/assemble
 member/.sti/bin
```

   5.3. Commit changes and push them to the classroom Git server.

```
[student@workstation ~]$ cd member
[student@workstation member]$ git add .sti
[student@workstation member]$ git commit -a -m "added custom s2i assemble
 script"
[student@workstation member]$ git push
[student@workstation member]$ cd
```

Note

The last **cd** command, without arguments, returns the user to his home folder.

6. Deploy the Member Application to OSE
Deploy the member application from the WAR file generated by OSE using a customized build configuration derived from the standard EAP templates. The build pod needs to know the URL to download the WAR artifact from Jenkins. This information is provided by the **APPLICATION_ARTIFACT_URL** environment variable.

6.1. The member application will be deployed using the **oc new-app** command and the standard **eap6-mysql-sti** template. This template requires a secret to provide SSL certificates for the EAP HTTPS connector. Create the secret using the provided resource definition file:

```
[student@workstation ~]$ oc create -f \
    /home/student/DO290/labs/review-jenkins/secret.json
```

Note

This lab will NOT use persistent storage to keep it simpler and focus on integrating Jenkins with OSE.

6.2. Using the template requires a very long command line, but this lab provides a Bash shell script with that command.

Edit the script **deploy-member.sh** in the lab folder to replace *X* with your student number. Following are the script contents:

```
#!/bin/bash
oc new-app -o json --template=eap6-mysql-sti -p \
GIT_URI=http://workstation.podX.example.com/member.git,\
APPLICATION_NAME=member,\
APPLICATION_HOSTNAME=member.cloudappsX.example.com,\
DB_JNDI=java:jboss/datasources/memberDS,\
DB_DATABASE=member,\
DB_USERNAME=member,\
DB_PASSWORD=member,\
EAP_HTTPS_SECRET=eap-app-secret-member,\
EAP_HTTPS_KEYSTORE=keystore.jks,\
EAP_HTTPS_NAME=jboss,\
EAP_HTTPS_PASSWORD=mykeystorepass \
> member.json
```

6.3. Run the edited script:

```
[student@workstation ~]$ bash /home/student/DO290/labs/review-jenkins/deploy-
member.sh
```

6.4. Edit the generated **member.json** resource definition file, in the student user home folder, to add the environment variable needed by the custom **assemble** script.

Find the line with **"kind": "BuildConfig"** and then scroll down to find the line with **"from": {**. Note this line opens a block using an angle bracket (**{**). Find the closing angle bracket (**}**) and add immediately after it a comma (**,**) and three environment variables needed by the custom **assemble** script. Use the following listing as a reference:

```
        {
            "kind": "BuildConfig",
...
```

```
                "strategy": {
                    "type": "Source",
                    "sourceStrategy": {
                        "from": {
                            "kind": "ImageStreamTag",
                            "namespace": "openshift",
                            "name": "jboss-eap6-openshift:6.4"
                        },
                        "env": [
                            {
                                "Name": "DISABLE_ASSET_COMPILATION",
                                "Value": "true"
                            },
                            {
                                "Name": "APPLICATION_ARTIFACT_URL",
                                "Value": "http://jenkins.cloudappsX.example.com/
job/member-war/lastSuccessfulBuild/artifact/target/member.war"
                            },
                            {
                                "Name": "DOWNLOAD_USER_CREDENTIALS",
                                "Value": "admin:password"
                            }
                        ]
                    }
                },
...
```

Note

The labs folder has an example of how the resource definition file should look after the change as file **member-edited-sample.json**. You can use it to copy-and-paste the environment variables or edit and use this file directly in place of **member.json**.

6.5. Create the member application resources from the generated resource configuration file:

```
[student@workstation ~]$ oc create -f member.json
```

6.6. After a few moments, a build should start. Use the **oc get builds** command while waiting for this. Then use the following command to follow the OSE build logs and check no Maven build is running inside the build pod, just a download for the WAR file from Jenkins.

```
[student@workstation ~]$ oc build-logs member-1 --follow
```

6.7. **If you made any mistake** during the previous steps, such as forgetting to create the secret or making typos while editing the **deploy-member.sh** script, use the following command to delete all resources from the member application but without deleting the Jenkins pod, then go back to step 5.

```
[student@workstation ~]$ oc delete all -l template=eap6-mysql-sti
```

7. Verify the Member Application Is Running
Now we should have the member application pod running alongside the Jenkins pod as part of the same OSE project.

7.1. Verify there are an application pod, a database pod, and a Jenkins pod running:

```
[student@workstation ~]$ oc get pods
```

The expected output is similar to:

```
NAME                READY     STATUS      RESTARTS   AGE
jenkins-1-ykuaz     1/1       Running     0          1h
member-1-4drmh      1/1       Running     0          7m
member-1-build      0/1       ExitCode:0  0          8m
member-mysql-1-w1kla 1/1      Running     0          8m
```

7.2. Verify that JBoss EAP started without problems by checking the application pod logs:

```
[student@workstation ~]$ oc logs member-1-4drmh | tail -n 3
```

The expected output is similar to:

```
17:42:10,945 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015961: Http
 management interface listening on http://0.0.0.0:9990/management
17:42:10,945 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015951: Admin
 console listening on http://0.0.0.0:9990
17:42:10,945 INFO  [org.jboss.as] (Controller Boot Thread) JBAS015874: JBoss
 EAP 6.4.1.GA (AS 7.5.1.Final-redhat-3) started in 21085ms - Started 308 of 399
 services (129 services are lazy, passive or on-demand)
```

7.3. Find the host name of the route created by the template for the application:

```
[student@workstation ~]$ oc get route member-http-route
```

The expected output is similar to:

```
NAME                HOST/PORT                          PATH      SERVICE   LABELS
                                                TLS TERMINATION
member-http-route   member.cloudappsX.example.com                 member
 application=member,template=eap6-mysql-sti
```

7.4. Access the application using a web browser and verify it works. The URL follows, replacing *X* with your student number:

http://member.cloudappsX.example.com/member

Insight

OSE builds publish the Java EE application in the root web context, but the Jenkins build left the application on the **/member** web context. See note on **step 4.3** for a more detailed explanation.

7.5. Get the member OSE build configuration web hook URL.

This will be needed by Jenkins to start new OSE builds. Use the following command:

```
[student@workstation ~]$ oc describe bc member | grep -a Webhook
```

The expected output is similar to:

```
Webhook GitHub:   https://master.podX.example.com:8443/oapi/v1/namespaces/
member-ci/buildconfigs/member/webhooks/IgYtll6P/github
Webhook Generic:  https://master.podX.example.com:8443/oapi/v1/namespaces/
member-ci/buildconfigs/member/webhooks/3upgsICs/generic
```

Copy the full **Webhook Generic** URL to the clipboard so you can use it during the next step. Part of this URL was randomly generated by OSE when the build configuration object was created, so yours may be different.

8. Create a Jenkins Project to Update the Application Inside OSE
Now we want Jenkins to trigger new deploys for the application inside OSE to enable building a simple Continuous Integration (CI)/Continuous Delivery (CD) pipeline. To do this, create a new Jenkins project named **member-ose** that uses an OSE web hook to trigger a new OSE build.

8.1. Create another Jenkins project.

Click the **Jenkins** link in the top-left corner to see the Jenkins project listing.

Click **New Item**, type **member-ose** as the project name, and select the **Freestyle project** radio box. Click **Ok**.

8.2. Configure a Jenkins **Project Parameter**.

To make the Jenkins configuration generic and reusable for different applications, we will configure the OSE webhook URL as a parameter.

Check the **This build is parameterized** box, click the **Add Parameter** button, and select the **String Parameter** option.

Type **WEBHOOK_URL** in the **Name** field and paste the full **Webhook Generic** URL from **step 7.5** into the **Default Value** field.

8.3. Configure the **member-ose** project to trigger the webhook.

Scroll down to **Build** section, click the **Add build step** button, and select **Execute shell**. Type the following in the **Command** field:

```
curl -i -H "Accept: application/json" -H "X-HTTP-Method-Override: PUT" -X POST -
k $WEBHOOK_URL
```

Important

Make sure there are no line breaks in the **Command** field.

Click the **Save** button to save the **member-ose** project.

You should be left in the **Project maven-ose** page.

9. Verify Jenkins Can Trigger an OSE Build
Before moving on, make sure the **member-ose Freeform Jenkins project** created during the previous step is configured correctly to trigger a build in the OSE **member-ci** project.

   9.1. Start a build.

   You should be in the **Project member-ose** page. If not, click the **Jenkins** link in the top-left corner to see the Jenkins project listing and then click the **member-ose** link.

   Click **Build with Parameters** and then the **Build** button. In a short time, a progress bar should be seen in the **Build History** box.

   9.2. Verify the Jenkins build logs.

   Click the latest build number (such as **#1**) in the **Build History** box and then click the **Console Output** link. This shows the build logs where you should see output from the **curl** commands started by Jenkins.

   Wait until the **Finished: SUCCESS** message is shown in the **Console Output** page. You may need to scroll down the page.

   9.3. Verify an OSE build was started.

   Go back to the workstation VM prompt and use the following command:

```
[student@workstation ~]$ oc get build
```

   The expected output shows a new build was started. Repeat the command until the build is complete:

```
NAME       TYPE      STATUS     POD
member-1   Source    Complete   member-1-build
member-2   Source    Complete   member-2-build
```

   You can also use the **oc describe dc member** command to see a new deployment was started as the result of the new build. You also can, but is not required, to test the application still works as expected.

Insight

if you test the application again, you'll notice all data you may have inserted in **step 7.4** is lost. This is NOT due to the fact the database pod is using ephemeral storage, as the database pod was NOT recreated by the build. Thus is due to the member application itself, which is configured for a development scenario, and recreates the database whenever it is restarted.

10. Connect Both Jenkins Projects to Complete The CI/CD Pipeline
To have a full though simple CI/CD pipeline, we need to connect both Jenkins projects so when the member application is rebuilt by the **member-war** Maven Jenkins project, a new build is triggered in the **member-ose** jenkins project and, as a consequence, a new OSE build is started.

    10.1. Click the **Jenkins** link in the top-left corner to see the Jenkins project listing, click the **member-war** link, and then click the **Configure** link.

    10.2. Scroll down to **Post-build Actions**, click the **Add post-build action** button, and select **Build other projects**.

    Type **member-ose** in the **Projects to build** field.

    Click the **Save** button.

11. Verify Building the Member Application in Jenkins Creates New Pods in OSE
Now test everything together; building the member application from its sources should deploy new application pods in OSE.

    11.1. Verify the current running pod names. Take note of the current **member-#-xxxxx** pod name:

```
[student@workstation ~]$ oc get pods
NAME                 READY     STATUS       RESTARTS   AGE
jenkins-1-ykuaz      1/1       Running      0          2h
member-1-build       0/1       ExitCode:0   0          1h
member-2-build       0/1       ExitCode:0   0          21m
member-2-cxit8       1/1       Running      0          20m
member-mysql-1-w1kla 1/1       Running      0          1h
```

Note

There is no problem if you have more **member-#-build** pods as long as none of them are running. There is also no problem if your **member-#-xxxxx** pod has a number greater than two.

    11.2. Start another build.

    You should be in the **Project member-war** page. If not, click the **Jenkins** link in the top-left corner to see the Jenkins project listing and then click the **member-war** link.

Click **Build Now**. In a short time, a progress bar should be seen in the **Build History** box.

11.3. Verify the new Maven build logs in Jenkins.

Click the latest build number (such as **#2**) in the **Build History** box and then click the **Console Output** link. This shows the build logs where you should see output from the **`git clone`** and **`maven clean package`** commands started by Jenkins.

Wait until the **`Finished: SUCCESS`** message is shown in the **Console Output** page. You may need to scroll down the page.

Note

The **Console Output** page should now show the message **`Triggering a new build of member-ose`** near the end.

11.4. Verify another OSE build was started and a new application pod was created, replacing the old one.

Go back to the workstation VM prompt and use the following command:

```
[student@workstation ~]$ oc get build
```

The expected output shows a new build was started. Repeat the command until the new build completes:

```
NAME       TYPE      STATUS     POD
member-1   Source    Complete   member-1-build
member-2   Source    Complete   member-2-build
member-3   Source    Complete   member-3-build
```

Verify the current running pod names. The current **`member-#-xxxxx`** pod should have a name different than the one you saw in **step 11.1**:

```
[student@workstation ~]$ oc get pods
NAME                  READY     STATUS       RESTARTS   AGE
jenkins-1-ykuaz       1/1       Running      0          3h
member-1-build        0/1       ExitCode:0   0          1h
member-2-build        0/1       ExitCode:0   0          31m
member-3-build        0/1       ExitCode:0   0          2m
member-3-th4iy        1/1       Running      0          1m
member-mysql-1-w1kla  1/1       Running      0          1h
```

You can, but are not required to, test the application again. Just remember it is expected all database data you inserted in previous tests will be lost.

This concludes the review.

Summary

In this chapter, you learned:

- To use an existing project available at a Git repository with a template.
- To deploy Jenkins and customize its parameters to support a chained build execution.
- Run the build to trigger an application build and deploy from Jenkins.