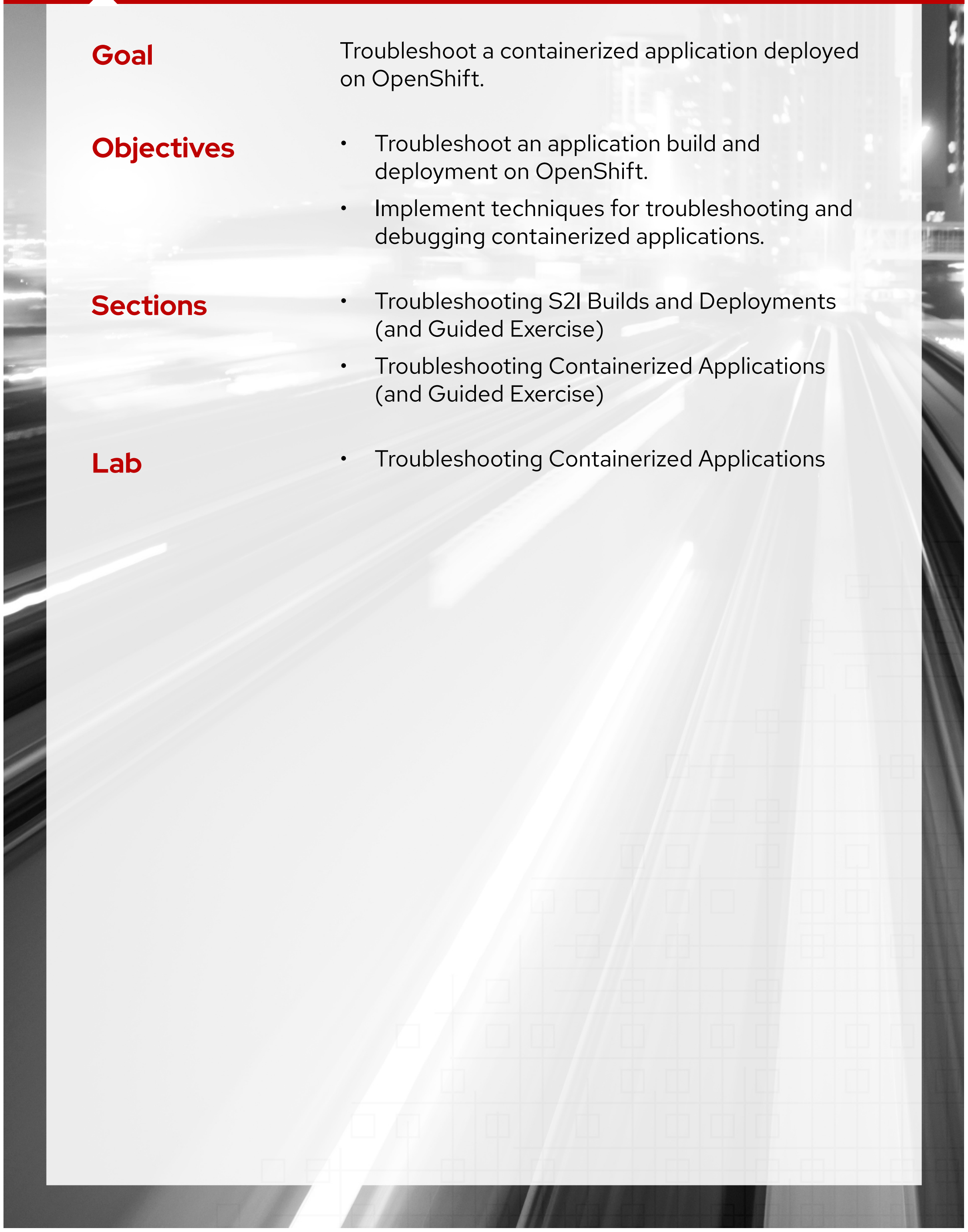
Extra Chapter

Troubleshooting Containerized Applications



Goal	Troubleshoot a containerized application deployed on OpenShift.
Objectives	• Troubleshoot an application build and deployment on OpenShift. • Implement techniques for troubleshooting and debugging containerized applications.
Sections	• Troubleshooting S2I Builds and Deployments (and Guided Exercise) • Troubleshooting Containerized Applications (and Guided Exercise)
Lab	• Troubleshooting Containerized Applications

Troubleshooting S2I Builds and Deployments

Objectives

After completing this section, you should be able to:

- Troubleshoot an application build and deployment steps on OpenShift.
- Analyze OpenShift logs to identify problems during the build and deploy process.

Introduction to the S2I Process

The *Source-to-Image (S2I)* process is a simple way to automatically create images based on the programming language of the application source code in OpenShift. While this process is often a convenient way to quickly deploy applications, problems can arise during the S2I image creation process, either by the programming language characteristics or the runtime environment that require both developers and administrators to work together.

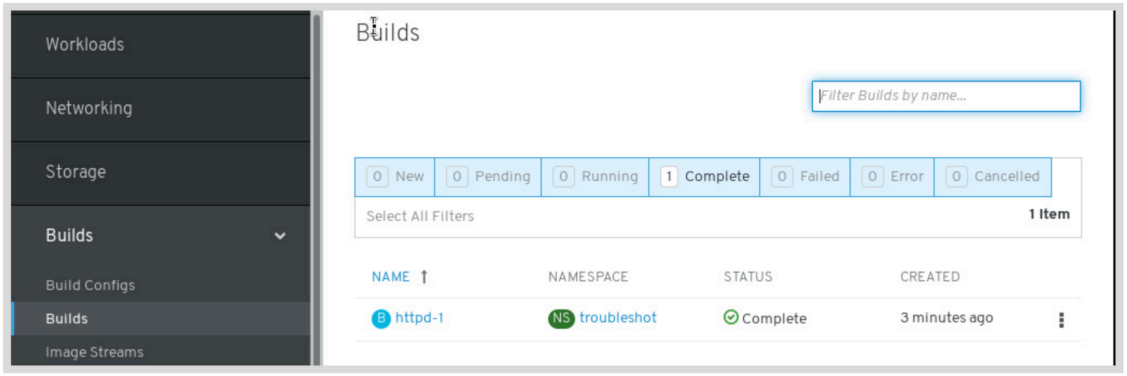
It is important to understand the basic workflow for most of the programming languages supported by OpenShift. The S2I image creation process is composed of two major steps:

- Build step: Responsible for compiling source code, downloading library dependencies, and packaging the application as a container image. Furthermore, the build step pushes the image to the OpenShift registry for the deployment step. The **`BuildConfig`** (**`BC`**) OpenShift resources drive the build step.
- Deployment step: Responsible for starting a pod and making the application available for OpenShift. This step executes after the build step, but only if the build step succeeded. The **`DeploymentConfig`** (**`DC`**) OpenShift resources drive the deployment step.

For the S2I process, each application uses its own **`BuildConfig`** and **`DeploymentConfig`** objects, the name of which matches the application name. The deployment process aborts if the build fails.

The S2I process starts each step in a separate pod. The build process creates a pod named **`<application-name>-build-<number>-<string>`**. For each build attempt, the entire build step executes and saves a log. Upon a successful build, the application starts on a separate pod named as **`<application-name>-<string>`**.

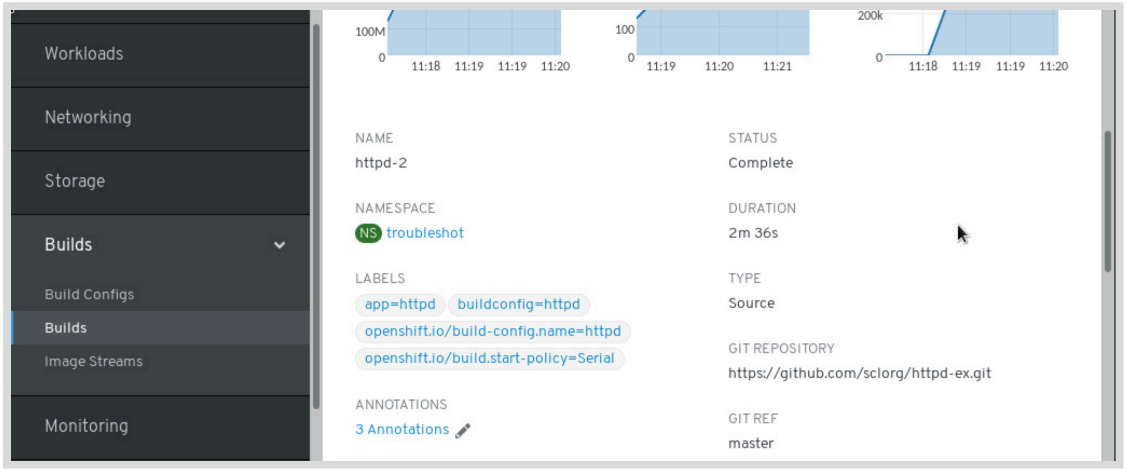
The OpenShift web console can be used to access the details for each step. To identify any build issues, the logs for a build can be evaluated and analyzed by clicking the **Builds** link from the left panel, depicted as follows.



NAME	NAMESPACE	STATUS	CREATED
B httpd-1	NS troubleshot	Complete	3 minutes ago

Figure 8.1: Build instances of a project

For each build attempt, a history of the build, tagged with a number, is provided for evaluation. Clicking on the build name leads to the details page of the build.



NAME
httpd-2

STATUS
Complete

NAMESPACE
NS troubleshot

DURATION
2m 36s

LABELS
app=httpd
buildconfig=httpd
openshift.io/build-config.name=httpd
openshift.io/build.start-policy=Serial

TYPE
Source

GIT REPOSITORY
https://github.com/sclorg/httpd-ex.git

ANNOTATIONS
3 Annotations

GIT REF
master

Figure 8.2: Detailed view of a build instance

The **Logs** tab of the build details page shows the output generated by the build execution. Those logs are handy to identify build issues.

Use the **Deployment Configs** link under **Workloads** section from the left panel to identify issues during the deployment step.

After selecting the appropriate deployment configuration, details show in the **Overview** section.

The **oc** command-line interface has several subcommands for managing the logs. Likewise in the web interface, it has a set of commands which provides information about each step. For example, to retrieve the logs from a build configuration, run the following command.

```
$ oc logs bc/<application-name>
```

If a build fails, after finding and fixing the issues, run the following command to request a new build:

```
$ oc start-build <application-name>
```

By issuing that command, OpenShift automatically spawns a new pod with the build process.

Deployment logs can be checked with the **oc** command:

```
$ oc logs dc/<application-name>
```

If the deployment is running or has failed, the command returns the logs of the process deployment process. Otherwise, the command returns the logs from the application's pod.

Describing Common Problems

Sometimes, the source code requires some customization that may not be available in containerized environments, such as database credentials, file system access, or message queue information. Those values usually take the form of internal environment variables. Developers using the S2I process may need to access this information.

The **oc logs** command provides important information about the build, deploy, and run processes of an application during the execution of a pod. The logs may indicate missing values or options that must be enabled, incorrect parameters or flags, or environment incompatibilities.

Note

Application logs must be clearly labelled to identify problems quickly without the need to learn the container internals.

Troubleshooting Permission Issues

OpenShift runs S2I containers using Red Hat Enterprise Linux as the base image, and any runtime difference may cause the S2I process to fail. Sometimes, the developer runs into permission issues, such as access denied due to the wrong permissions, or incorrect environment permissions set by administrators. S2I images enforce the use of a different user than the **root** user to access file systems and external resources. Also, Red Hat Enterprise Linux 7 enforces SELinux policies that restrict access to some file system resources, network ports, or process.

Some containers may require a specific user ID, whereas S2I is designed to run containers using a random user as per the default OpenShift security policy.

The following Dockerfile creates a Nexus container. Note the **USER** instruction indicating the **nexus** user should be used:

```
FROM ubi7/ubi:7.7
...contents omitted...
RUN chown -R nexus:nexus ${NEXUS_HOME}

USER nexus
WORKDIR ${NEXUS_HOME}

VOLUME ["/opt/nexus/sonatype-work"]
...contents omitted...
```

Trying to use the image generated by this Dockerfile without addressing volume permissions drives to errors when the container starts:

```
$ oc logs nexus-1-wzjrn
...output omitted...
... org.sonatype.nexus.util.LockFile - Failed to write lock file
...FileNotFoundException: /opt/nexus/sonatype-work/nexus.lock (Permission denied)
...output omitted...
... org.sonatype.nexus.webapp.WebappBootstrap - Failed to initialize
...lStateException: Nexus work directory already in use: /opt/nexus/sonatype-work
...output omitted...
```

To solve this issue, relax the OpenShift project security with the command **`oc adm policy`**.

```
[student@workstation ~]$ oc adm policy add-scc-to-user anyuid -z default
```

This **`oc adm policy`** command enables OpenShift executing container processes with non-root users. But the file systems used in the container must also be available for the running user. This is specially important when the container contains volume mounts.

To avoid file system permission issues, local folders used for container volume mounts must satisfy the following:

- The user executing the container processes must be the owner of the folder, or have the necessary rights. Use the **`chown`** command to update folder ownership.

- The local folder must satisfy the SELinux requirements to be used as a container volume. Assign the **`container_file_t`** group to the folder by using the **`semanage fcontext -a -t container_file_t <folder>`** command, then refresh the permissions with the **`restorecon -R <folder>`** command.

Troubleshooting Invalid Parameters

Multi-container applications may share parameters, such as login credentials. Ensure that the same values for parameters reach all containers in the application. For example, for a Python application that runs in one container, connected with another container running a database, make sure that the two containers use the same user name and password for the database. Usually, logs from the application pod provide a clear idea of these problems and how to solve them.

A good practice to centralize shared parameters is to store them in **`ConfigMaps`**. Those **`ConfigMaps`** can be injected through the **`Deployment Config`** into containers as environment variables. Injecting the same **`ConfigMap`** into different containers ensures that not only the same environment variables are available, but also the same values. See the following pod resource definition:

```
apiVersion: v1
kind: Pod
...output omitted...
spec:
  containers:
    - name: test-container
...output omitted...
  env:
        - name: ENV_1 (1)
          valueFrom:
            configMapKeyRef:
              name: configMap_name1
```

```
                key: configMap_key_1
...output omitted...
  envFrom:
        - configMapRef:
              name: configMap_name_2 2
...output omitted...
```

1. An **ENV_1** environment variable is injected into the container. Its value is the value for the **configMap_key_1** entry in the **configMap_name1** configMap.
2. All entries in **configMap_name_2** are injected into the container as environment variables with the same name and values.

Troubleshooting Volume Mount Errors

When redeploying an application that uses a persistent volume on a local file system, a pod might not be able to allocate a persistent volume claim even though the persistent volume indicates that the claim is released. To resolve the issue, delete the persistent volume claim and then the persistent volume. Then recreate the persistent volume.

```
oc delete pv <pv_name>
oc create -f <pv_resource_file>
```

Troubleshooting Obsolete Images

OpenShift pulls images from the source indicated in an image stream unless it locates a locally-cached image on the node where the pod is scheduled to run. If you push a new image to the registry with the same name and tag, you must remove the image from each node the pod is scheduled on with the command **podman rmi**.

Run the **oc adm prune** command for an automated way to remove obsolete images and other resources.

References

More information about troubleshooting images is available in the *Images* section of the OpenShift Container Platform documentation accessible at:

Creating Images
https://docs.openshift.com/container-platform/4.2/openshift_images/create-images.html

Documentation about how to consume ConfigMap to create container environment variables can be found in the *Consuming in Environment Variables* of the
Configure a Pod to use ConfigMaps
https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/#define-container-environment-variables-using-configmap-data

Guided Exercise

Troubleshooting an OpenShift Build

In this exercise, you will troubleshoot an OpenShift build and deployment process.

Outcomes

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

Before You Begin

A running OpenShift cluster.

Retrieve the lab files and verify that Docker and the OpenShift cluster are running by running the following command.

```
[student@workstation ~]$ lab troubleshoot-s2i start
```

1. Load the configuration of your classroom environment. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

2. Enter your local clone of the **DO180-apps** Git repository and checkout the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/DO180-apps
[student@workstation DO180-apps]$ git checkout master
...output omitted...
```

3. Create a new branch to save any changes you make during this exercise:

```
[student@workstation DO180-apps]$ git checkout -b troubleshoot-s2i
Switched to a new branch 'troubleshoot-s2i'
[student@workstation DO180-apps]$ git push -u origin troubleshoot-s2i
...output omitted...
* [new branch]      troubleshoot-s2i -> s2i
Branch troubleshoot-s2i set up to track remote branch troubleshoot-s2i from
 origin.
```

4. Log in to OpenShift using the configured user, password and Master API URL.

```
[student@workstation DO180-apps]$ oc login -u "${RHT_OCP4_DEV_USER}" \
> -p "${RHT_OCP4_DEV_PASSWORD}"
Login successful.

You have access to the following projects and can switch between them with 'oc
 project <projectname>':
...output omitted...
```

Create a new project named ***youruser*-nodejs**.

```
[student@workstation DO180-apps]$ oc new-project ${RHT_OCP4_DEV_USER}-nodejs
Now using project "youruser-nodejs" on server "https://
api.cluster.lab.example.com"
...output omitted...
```

5. Build a new Node.js application using the **Hello World** image located at `https://github.com/`*`yourgituser`*`/DO180-apps/` in the **nodejs-helloworld** directory.

5.1. Run the **oc new-app** command to create the Node.js application. The command is provided in the **~/DO180/labs/troubleshoot-s2i/command.txt** file.

```
[student@workstation DO180-apps]$ oc new-app --context-dir=nodejs-helloworld \
> https://github.com/${RHT_OCP4_GITHUB_USER}/DO180-apps#troubleshoot-s2i \
> -i nodejs:8 --name nodejs-hello --build-env \
> npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/npm-proxy
--> Found image a2b5ec2 ...output omitted...

    Node.js 8
...output omitted...
--> Creating resources ...
    imagestream.image.openshift.io "nodejs-hello" created
    buildconfig.build.openshift.io "nodejs-hello" created
    deploymentconfig.apps.openshift.io "nodejs-hello" created
    service "nodejs-hello" created
--> Success
    Build scheduled, use 'oc logs -f bc/nodejs-hello' to track its progress.
    Application is not exposed. You can expose services to the outside world by
 executing one or more of the commands below:
     'oc expose svc/nodejs-hello'
    Run 'oc status' to view your app.
```

The **-i** indicates the builder image to use, **nodejs:8** in this case.

The **--context-dir** option defines which folder inside the project contains the source code of the application to build.

The **--build-env** option defines an environment variable to the builder pod. In this case, it provides the **npm_config_registry** environment variable to the builder pod, so it can reach the NPM registry.

Important

In the previous command, there must be no spaces between **`registry=`** and the URL of the Nexus server.

5.2. Wait until the application finishes building by monitoring the progress with the **`oc get pods -w`** command. The pod transitions from a status of **`running`** to **`Error`**:

```
[student@workstation DO180-apps]$ oc get pods -w
NAME                    READY   STATUS     RESTARTS   AGE
nodejs-hello-1-build    1/1     Running    0          15s
nodejs-hello-1-build    0/1   Error    0      73s
^C
```

The build process fails, and therefore no application is running. Build failures are usually consequences of syntax errors in the source code or missing dependencies. The next step investigates the specific causes for this failure.

5.3. Evaluate the errors raised during the build process.

The build is triggered by the build configuration (**`bc`**) created by OpenShift when the S2I process starts. By default, the OpenShift S2I process creates a build configuration named as the name given: **`nodejs-hello`**, which is responsible for triggering the build process.

Run the **`oc`** command with the **`logs`** subcommand in a terminal window to review the output of the build process:

```
[student@workstation DO180-apps]$ oc logs bc/nodejs-hello
  Cloning "https://github.com/yourgituser/DO180-apps" ...
  Commit: f7cd8963ef353d9173c3a21dcccf402f3616840b ( Initial commit...
...output omitted...
STEP 8: RUN /usr/libexec/s2i/assemble
---> Installing application source ...
---> Installing all dependencies
npm ERR! code ETARGET
npm ERR! notarget No matching version found for express@~4.14.2
npm ERR! notarget In most cases you or one of your dependencies are requesting
npm ERR! notarget a package version that doesn't exist.
npm ERR! notarget
npm ERR! notarget It was specified as a dependency of 'nodejs-helloworld'
npm ERR! notarget

npm ERR! A complete log of this run can be found in:
npm ERR!     /opt/app-root/src/.npm/_logs/2019-10-25T12_37_56_853Z-debug.log
subprocess exited with status 1
...output omitted...
```

The log shows an error occurred during the build process. This output indicates that there is no compatible version for the **`express`** dependency. But the reason is that the format used by the express dependency is not valid.

6. Update the build process for the project.

The developer uses a nonstandard version of the Express framework that is available locally on each developer's workstation. Due to the company's standards, the version must be downloaded from the Node.js official registry and, from the developer's input, it is compatible with the 4.14.x version.

6.1. Fix the **`package.json`** file.

Use your preferred editor to open the **`~/DO180-apps/nodejs-helloworld/package.json`** file. Review the dependencies versions provided by the developers. It uses an incorrect version of the Express dependency, which is incompatible with the supported version provided by the company (**`~4.14.2`**). Update the dependency version as follows.

```
{
  "name": "nodejs-helloworld",
 ...output omitted...
  "dependencies": {
    "express": "4.14.x"
  }
}
```

Note

Notice the **`x`** in the version. It indicates that the highest version should be used, but the version must begin with **`4.14.`**.

6.2. Commit and push the changes made to the project.

From the terminal window, run the following command to commit and push the changes:

```
[student@workstation DO180-apps]$ git commit -am "Fixed Express release"
...output omitted...
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation DO180-apps]$ git push
...output omitted...
To https://github.com/yourgituser/DO180-apps
   ef6557d..73a82cd  troubleshoot-s2i -> troubleshoot-s2i
```

7. Relaunch the S2I process.

7.1. To restart the build step, execute the following command:

```
[student@workstation DO180-apps]$ oc start-build bc/nodejs-hello
build "nodejs-hello-2" started
```

The build step is restarted, and a new build pod is created. Check the log by running the **`oc logs`** command.

```
[student@workstation DO180-apps]$ oc logs -f bc/nodejs-hello
Cloning "https://github.com/yougituser/DO180-apps" ...
Commit: ea2125c1bf4681dd9b79ddf920d8d8be38cfcf3b (Fixed Express release)
...output omitted...
Pushing image ...image-registry.svc:5000/nodejs/nodejs-hello:latest...
...output omitted...
Push successful
```

The build is successful, however, this does not indicate that the application is started.

7.2. Evaluate the status of the current build process. Run the **oc get pods** command to check the status of the Node.js application.

```
[student@workstation DO180-apps]$ oc get pods
```

According to the following output, the second build completed, but the application is in error state.

```
NAME                              READY      STATUS              RESTARTS    AGE
nodejs-hello-1-build   0/1       Error             0          29m
nodejs-hello-1-rpx1d   0/1       CrashLoopBackOff  6          6m
nodejs-hello-2-build   0/1       Completed         0          7m
```

The name of the application pod (**nodejs-hello-1-rpx1d**) is generated randomly, and may differ from yours.

7.3. Review the logs generated by the application pod.

```
[student@workstation DO180-apps]$ oc logs dc/nodejs-hello
...output omitted...
npm info using npm@6.4.1
npm info using node@v8.16.0
npm ERR! missing script: start
...output omitted...
```

Note

The **oc logs dc/nodejs-hello** command dumps the logs from the deployment pod. In the case of a successful deployment, that command dumps the logs from the application pod, as previously shown.

The application fails to start because the start script declaration is missing.

8. Fix the problem by updating the application code.

8.1. Update the **package.json** file to define a startup command.

The previous output indicates that the **~/DO180-apps/nodejs-helloworld/package.json** file is missing the **start** attribute in the **scripts** field. The **start** attribute defines a command to run when the application starts. It invokes the **node** binary, which runs the app.js application.

To fix the problem, add to the **`package.json`** file the following attribute. Do not forget the comma after the bracket.

```
...
  "description": "Hello World!",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "author": "Red Hat Training",
...
```

8.2. Commit and push the changes made to the project:

```
[student@workstation DO180-apps]$ git commit -am "Added start up script"
...output omitted...
1 file changed, 3 insertions(+)
[student@workstation DO180-apps]$ git push
...output omitted...
To https://github.com/yourgituser/DO180-apps
   73a82cd..a5a0411  troubleshoot-s2i -> troubleshoot-s2i
```

Continue the deploy step from the S2I process.

8.3. Restart the build step.

```
[student@workstation DO180-apps]$ oc start-build bc/nodejs-hello
build "nodejs-hello-3" started
```

8.4. Evaluate the status of the current build process. Run the command to retrieve the status of the Node.js application. Wait for the latest build to finish.

```
[student@workstation DO180-apps]$ oc get pods -w
NAME                        READY   STATUS            RESTARTS   AGE
nodejs-hello-1-build   0/1     Error              0          66m
nodejs-hello-1-mtxsh   0/1     CrashLoopBackOff   9          23m
nodejs-hello-2-build   0/1     Completed          0          28m
nodejs-hello-3-build   1/1     Running            0          2m3s
nodejs-hello-2-deploy   0/1   Pending   0     0s
nodejs-hello-2-deploy   0/1   Pending   0     0s
nodejs-hello-2-deploy   0/1   ContainerCreating   0     0s
nodejs-hello-3-build 0/1 Completed 0 3m9s
nodejs-hello-2-deploy   1/1   Running   0     4s
nodejs-hello-2-8tsl4   0/1   Pending   0     0s
nodejs-hello-2-8tsl4   0/1   Pending   0     1s
nodejs-hello-2-8tsl4   0/1   ContainerCreating   0     1s
nodejs-hello-2-8tsl4 1/1 Running 0 50s
nodejs-hello-1-mtxsh   0/1   Terminating   9     25m
nodejs-hello-1-mtxsh   0/1   Terminating   9     25m
nodejs-hello-2-deploy 0/1 Completed 0 61s
nodejs-hello-2-deploy   0/1   Terminating   0     61s
```

```
nodejs-hello-2-deploy   0/1   Terminating   0     61s
nodejs-hello-1-mtxsh   0/1   Terminating   9     25m
nodejs-hello-1-mtxsh   0/1   Terminating   9     25m
```

According to the output, the build is successful, and the application is able to start with no errors. The output also provides insight into how the deployment pod (**`nodejs-hello-2-deploy`**) was created, and that it completed successfully and terminated. As the new application pod is available (**`nodejs-hello-2-8tsl4`**), the old one (**`nodejs-hello-1-mtxsh`**) is rolled out.

8.5. Review the logs generated by the **`nodejs-hello`** application pod.

```
[student@workstation DO180-apps]$ oc logs dc/nodejs-hello
Environment:
 DEV_MODE=false
 NODE_ENV=production
 DEBUG_PORT=5858
Launching via npm...
npm info it worked if it ends with ok
npm info using npm@2.15.1
npm info using node@v4.6.2
npm info prestart nodejs-helloworld@1.0.0
npm info start nodejs-helloworld@1.0.0

> nodejs-helloworld@1.0.0 start /opt/app-root/src
> node app.js

Example app listening on port 8080!
```

The application is now running on port 8080.

9. Test the application.

9.1. Run the **oc** command with the **`expose`** subcommand to expose the application:

```
[student@workstation DO180-apps]$ oc expose svc/nodejs-hello
route.route.openshift.io/nodejs-hello exposed
```

9.2. Retrieve the address associated with the application.

```
[student@workstation DO180-apps]$ oc get route -o yaml
apiVersion: v1
items:
- apiVersion: route.openshift.io/v1
  kind: Route
...output omitted...
  spec:
    host: nodejs-hello-nodejs.apps.cluster.lab.example.com
    port:
      targetPort: 8080-tcp
    to:
      kind: Service
      name: do180-apps
...output omitted...
```

9.3. Access the application from the **workstation** VM by using the **curl** command:

```
[student@workstation DO180-apps]$ curl -w "\n" \
> http://nodejs-hello-nodejs.apps.cluster.lab.example.com
Hello world!
```

The output demonstrates the application is up and running.

Finish

On **workstation**, run the **lab troubleshoot-s2i finish** script to complete this exercise.

```
[student@workstation DO180-apps]$ lab troubleshoot-s2i finish
```

This concludes the exercise.

Troubleshooting Containerized Applications

Objectives

After completing this section, you should be able to:

- Implement techniques for troubleshooting and debugging containerized applications.
- Use the port-forwarding feature of the OpenShift client tool.
- View container logs.
- View OpenShift cluster events.

Forwarding Ports for Troubleshooting

Occasionally developers and system administrators need special network access to a container that would not be needed by application users. For example, developers may need to use the administration console for a database or messaging service, or system administrators may make use of SSH access to a container to restart a terminated service. Such network access, in the form of network ports, are usually not exposed by the default container configurations, and tend to require specialized clients used by developers and system administrators.

Podman provides port forwarding features by using the **`-p`** option along with the **`run`** subcommand. In this case, there is no distinction between network access for regular application access and for troubleshooting. As a refresher, the following is an example of configuring port forwarding by mapping the port from the host to a database server running inside a container:

```
$ sudo podman run --name db -p 30306:3306 mysql
```

The previous command maps the host port 30306 to the port 3306 on the **`db`** container. This container is created from the **`mysql`** image, which starts a MySQL server that listens on port 3306.

OpenShift provides the **`oc port-forward`** command for forwarding a local port to a pod port. This is different than having access to a pod through a service resource:

- The port-forwarding mapping exists only in the workstation where the **`oc`** client runs, while a service maps a port for all network users.
- A service load-balances connections to potentially multiple pods, whereas a port-forwarding mapping forwards connections to a single pod.

Here is an example of the **`oc port-forward`** command:

```
$ oc port-forward db 30306 3306
```

The previous command forwards port 30306 from the developer machine to port 3306 on the **`db`** pod, where a MySQL server (inside a container) accepts network connections.

Note

When running this command, be sure to leave the terminal window running. Closing the window or canceling the process stops the port mapping.

While the **`podman run -p`** method of mapping (port-forwarding) can only be configured when the container is started, the mapping with the **`oc port-forward`** command can be created and destroyed at any time after a pod was created.

Note

Creating a service of **`NodePort`** type for a database pod would be similar to running **`podman run -p`**. However, Red Hat discourages the usage of the **`NodePort`** approach to avoid exposing the service to direct connections. Mapping with port-forwarding in OpenShift is considered a more secure alternative.

Enabling Remote Debugging with Port Forwarding

Another use for the port forwarding feature is enabling remote debugging. Many *integrated development environments (IDEs)* provide the capability to remotely debug an application.

For example, JBoss Developer Studio (JBDS) allows users to utilize the Java Debug Wire Protocol (JDWP) to communicate between a debugger (JBDS) and the Java Virtual Machine. When enabled, developers can step through each line of code as it is being executed in real time.

For JDWP to work, the Java Virtual Machine (JVM) where the application runs must be started with options enabling remote debugging. For example, WildFly and JBoss EAP users must configure these options on application server startup. The following line in the **`standalone.conf`** file enables remote debugging by opening the JDWP TCP port 8787, for a WildFly or EAP instance running in standalone mode:

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

When the server starts with the debugger listening on port 8787, a port forwarding mapping needs to be created to forward connections from a local unused TCP port to port 8787 in the EAP pod. If the developer workstation has no local JVM running with remote debugging enabled, the local port can also be 8787.

The following command assumes a WildFly pod named **`jappserver`** running a container from an image previously configured to enable remote debugging:

```
$ oc port-forward jappserver 8787:8787
```

Once the debugger is enabled and the port forwarding mapping is created, users can set breakpoints in their IDE of choice and run the debugger by pointing to the application's host name and debug port (in this instance, 8787).

Accessing Container Logs

Podman and OpenShift provide the ability to view logs in running containers and pods to facilitate troubleshooting. But neither of them is aware of application specific logs. Both expect the application to be configured to send all logging output to the standard output.

A container is simply a process tree from the host OS perspective. When Podman starts a container either directly or on the RHOCP cluster, it redirects the container standard output and standard error, saving them on disk as part of the container's ephemeral storage. This way, the container logs can be viewed using **podman** and **oc** commands, even after the container was stopped, but not removed.

To retrieve the output of a running container, use the following **podman** command.

```
$ podman logs <containerName>
```

In OpenShift, the following command returns the output for a container within a pod:

```
$ oc logs <podName> [-c <containerName>]
```

Note

The container name is optional if there is only one container, as **oc** defaults to the only running container and returns the output.

OpenShift Events

Some developers consider Podman and OpenShift logs to be too low-level, making troubleshooting difficult. Fortunately, OpenShift provides a high-level logging and auditing facility called *events*.

OpenShift events signal significant actions like starting a container or destroying a pod.

To read OpenShift events, use the **get** subcommand with the **events** resource type for the **oc** command, as follows.

```
$ oc get events
```

Events listed by the **oc** command this way are not filtered and span the whole RHOCP cluster. Using a pipe to standard UNIX filters such as **grep** can help, but OpenShift offers an alternative in order to consult cluster events. The approach is provided by the **describe** subcommand.

For example, to only retrieve the events that relate to a **mysql** pod, refer **Events** field from the output of **oc describe pod *mysql*** command.

```
$ oc describe pod mysql
...output omitted...
Events:
  FirstSeen    LastSeen    Count From          Reason          Message
  Wed, 10 ... Wed, 10 ... 1     {scheduler } scheduled       Successfully as...
...output omitted...
```

Accessing Running Containers

The **podman logs** and **oc logs** commands can be useful for viewing output sent by any container. However, the output does not necessarily display all of the available information if the application is configured to send logs to a file. Other troubleshooting scenarios may require inspecting the container environment as seen by processes inside the container, such as verifying external connectivity.

As a solution, Podman and OpenShift provide the **exec** subcommand, allowing the creation of new processes inside a running container, with the standard output and input of these processes redirected to the user terminal. The following screen display the usage of the **podman exec** command:

```
$ sudo podman exec [options] container command [arguments]
```

The general syntax for the **oc exec** command is:

```
$ oc exec [options] pod [-c container] -- command [arguments]
```

To execute a single interactive command or start a shell, add the **-it** options. The following example starts a Bash shell for the **myhttpdpod** pod:

```
$ oc exec -it myhttpdpod /bin/bash
```

You can use this command to access application logs saved to disk (as part of the container ephemeral storage). For example, to display the Apache error log from a container, run the following command:

```
$ sudo podman exec apache-container cat /var/log/httpd/error_log
```

Overriding Container Binaries

Many container images do not contain all of the troubleshooting commands users expect to find in regular OS installations, such as **telnet**, **netcat**, **ip**, or **traceroute**. Stripping the image from basic utilities or binaries allows the image to remain slim, thus, running many containers per host.

One way to temporarily access some of these missing commands is mounting the host binaries folders, such as **/bin**, **/sbin**, and **/lib**, as volumes inside the container. This is possible because the **-v** option from **podman run** command does not require matching **VOLUME** instructions to be present in the **Dockerfile** of the container image.

Note

To access these commands in OpenShift, you need to change the pod resource definition in order to define **volumeMounts** and **volumeClaims** objects. You also need to create a **hostPath** persistent volume.

The following command starts a container, and overrides the image's **/bin** folder with the one from the host. It also starts an interactive shell inside the container.

```
$ sudo podman run -it -v /bin:/bin image /bin/bash
```

> **Note**
>
> The directory of binaries to override depends on the base OS image. For example, some commands require shared libraries from the **`/lib`** directory. Some Linux distributions have different contents in **`/bin`**, **`/usr/bin`**, **`/lib`**, or **`/usr/lib`**, which would require to use the **`-v`** option for each directory.

As an alternative, you can include these utilities in the base image. To do so, add instructions in a **`Dockerfile`** build definition. For example, examine the following excerpt from a **`Dockerfile`** definition, which is a child of the **`rhel7.5`** image used throughout this course. The **`RUN`** instruction installs the tools that are commonly used for network troubleshooting.

```
FROM ubi7/ubi:7.7

RUN yum install -y \
      less \
      dig \
      ping \
      iputils && \
    yum clean all
```

When the image is built and the container is created, it will be identical to a **`rhel7.5`** container image, plus the extra available tools.

Transferring Files To and Out of Containers

When troubleshooting or managing an application, you may need to retrieve or transfer files to and from running containers, such as configuration files or log files. There are several ways to move files into and out of containers, as described in the following list.

Volume mounts

Another option for copying files from the host to a container is the usage of volume mounts. You can mount a local directory to copy data into a container. For example, the following command sets **`/conf`** host directory as the volume to use for the Apache configuration directory in the container. This provides a convenient way to manage the Apache server without having to rebuild the container image.

```
$ sudo podman run -v /conf:/etc/httpd/conf -d do180/apache
```

`podman cp`

The **`cp`** subcommand allows users to copy files both into and out of a running container. To copy a file into a container named **`todoapi`**, run the following command.

```
$ sudo podman cp standalone.conf todoapi:/opt/jboss/standalone/conf/
standalone.conf
```

To copy a file from the container to the host, flip the order of the previous command.

```
$ sudo podman cp todoapi:/opt/jboss/standalone/conf/standalone.conf .
```

The **`podman cp`** command has the advantage of working with containers that were already started, while the following alternative (volume mounts) requires changes to the command used to start a container.

`podman exec`

For containers that are already running, the **`podman exec`** command can be piped to pass files both into and out of the running container by appending commands that are executed in the container. The following example shows how to pass in and execute a SQL file inside a MySQL container:

```
$ sudo podman exec -i <container> mysql -uroot -proot < /path/on/host/db.sql <
 db.sql
```

Using the same concept, it is possible to retrieve data from a running container and place it in the host machine. A useful example of this is the usage of the **`mysqldump`** utility, which creates a backup of MySQL database from the container and places it on the host.

```
$ sudo podman exec -it <containerName> sh \
> -c 'exec mysqldump -h"$MYSQL_PORT_3306_TCP_ADDR" \
> -P"$MYSQL_PORT_3306_TCP_PORT" \
> -uroot -p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD" items'
> db_dump.sql
```

The previous command uses the container environment variables to connect to the MySQL server to execute the **`mysqldump`** command and redirects the output to a file on the host machine. It assumes that the container image provides the **`mysqldump`** utility, so there is no need to install the MySQL administration tools on the host.

The **`oc rsync`** command provides functionality similar to **`podman cp`** for containers running under OpenShift pods.

References

More information about port-forwarding is available in the *Port Forwarding* section of the OpenShift Container Platform documentation at
Architecture
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html-single/architecture/index/

More information about the CLI commands for port-forwarding are available in the *Port Forwarding* chapter of the OpenShift Container Platform documentation at
Developing Applications
https://access.redhat.com/documentation/en-us/openshift_container_platform/4.2/html-single/applications/index/

Guided Exercise

Configuring Apache Container Logs for Debugging

In this exercise, you will configure an Apache httpd container to send the logs to the **stdout**, then review Podman logs and events.

Outcomes

You should be able to configure an Apache httpd container to send debug logs to **stdout** and view them using the **podman logs** command.

Before You Begin

A running OpenShift cluster.

Retrieve the lab files and verify that Docker and the OpenShift cluster are running by running the following command.

```
[student@workstation ~]$ lab troubleshoot-container start
```

1. Configure a Apache web server to send log messages to the standard output and update the default log level.

1.1. The default log level for the Apache httpd image is **warn**. Change the default log level for the container to **debug**, and redirect log messages to **stdout** by overriding the default **httpd.conf** configuration file. To do so, create a custom image from the **workstation** VM.

Briefly review the custom **httpd.conf** file located at **/home/student/DO180/labs/troubleshoot-container/conf/httpd.conf**.

- Observe the **ErrorLog** directive in the file:

```
ErrorLog /dev/stdout
```

The directive sends the httpd error log messages to the container's standard output.

- Observe the **LogLevel** directive in the file.

```
LogLevel debug
```

The directive changes the default log level to **debug**.

- Observe the **CustomLog** directive in the file.

```
CustomLog /dev/stdout common
```

The directive redirects the httpd access log messages to the container's standard output.

2. Build a custom container to save an updated configuration file to the container.

   2.1. From the terminal window, run the following commands to build a new image.

```
[student@workstation ~]$ cd ~/DO180/labs/troubleshoot-container
[student@workstation troubleshoot-container]$ sudo podman build \
>  -t troubleshoot-container .
STEP 1: FROM redhattraining/httpd-parent
...output omitted...
--> e23d...c1de
STEP 7: COMMIT troubleshoot-container
[student@workstation troubleshoot-container]$ cd ~
```

   2.2. Verify that the image is created.

```
[student@workstation ~]$ sudo podman images
```

The new image must be available in the local storage.

```
REPOSITORY                          TAG      IMAGE ID   CREATED          SIZE
localhost/troubleshoot-container    latest   e23df...   9 seconds ago    137MB
quay.io/redhattraining/httpd-parent      latest   0eba3...   4 weeks ago     137MB
```

3. Create a new httpd container from the custom image.

```
[student@workstation ~]$ sudo podman run \
> --name troubleshoot-container -d \
> -p 10080:80 troubleshoot-container
4c8bb12815cc02f4eef0254632b7179bd5ce230d83373b49761b1ac41fc067a9
```

4. Review the container's log messages and events.

   4.1. View the debug log messages from the container using the **`podman logs`** command:

```
[student@workstation ~]$ sudo podman logs -f troubleshoot-container
... [mpm_event:notice] [pid 1:tid...] AH00489: Apache/2.4.25 (Unix) configur...
... [mpm_event:info] [pid 1:tid...] AH00490: Server built: Mar 21 2017 20:50:17
... [core:notice] [pid 1:tid...] AH00094: Command line: 'httpd -D FOREGROUND'
... [core:debug] [pid 1:tid ...): AH02639: Using SO_REUSEPORT: yes (1)
... [mpm_event:debug] [pid 6:tid ...): AH02471: start_threads: Using epoll
... [mpm_event:debug] [pid 7:tid ...): AH02471: start_threads: Using epoll
... [mpm_event:debug] [pid 8:tid ...): AH02471: start_threads: Using epoll
```

Notice the debug logs, available in the standard output.

   4.2. Open a new terminal and access the home page of the web server by using the **`curl`** command:

```
[student@workstation ~]$ curl http://127.0.0.1:10080
Hello from the httpd-parent container!
```

4.3. Review the new entries in the log. Look in the terminal running the **podman logs** command to see the new entries.

```
[student@workstation ~]$ sudo podman logs troubleshoot-container
...[authz_core:debug] ...: authorization result of Require all granted: granted
...[authz_core:debug] ...: authorization result of <RequireAny>: granted
10.88.0.1 - - [08/Mar/2019:20:30:53 +0000] "GET / HTTP/1.1" 200 45
```

4.4. Stop the Podman command with **Ctrl**+**C**.

Finish

On **workstation**, run the **lab troubleshoot-container finish** script to complete this lab.

```
[student@workstation ~]$ lab troubleshoot-container finish
```

This concludes the guided exercise.

Lab

Troubleshooting Containerized Applications

Performance Checklist

In this lab, you will troubleshoot the OpenShift build and deployment process for a Node.js application.

Outcomes

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

Before You Begin

A running OpenShift cluster.

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab troubleshoot-review start
```

1. Load the configuration of your classroom environment. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

2. Enter your local clone of the **DO180-apps** Git repository and checkout the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/DO180-apps
[student@workstation DO180-apps]$ git checkout master
...output omitted...
```

3. Create a new branch to save any changes you make during this exercise:

```
[student@workstation DO180-apps]$ git checkout -b troubleshoot-review
Switched to a new branch 'troubleshoot-review'
[student@workstation DO180-apps]$ git push -u origin troubleshoot-review
...output omitted...
* [new branch]      troubleshoot-review -> troubleshoot-review
Branch troubleshoot-review set up to track remote branch troubleshoot-review from
 origin.
```

4. Log in to OpenShift using the configured user, password and Master API URL.
5. Create a new project named ***youruser*-nodejs-app**:

6. In the **youruser-nodejs-app** OpenShift project, create a new application from the source code located **nodejs-app** directory in the Git repository at `https://github.com/`*`yourgituser`*`/DO180-apps`. Name the application **nodejs-dev**.

   Expect the build process for the application to fail. Monitor the build process and identify the build failure.

7. Update the version of the **express** dependency in the **package.json** file with a value of **4.x**. Commit and push the changes to the Git repository.

8. Rebuild the application. Verify that the application builds without errors.

9. Verify that the application is not running because of a runtime error. Review the logs and identify the problem.

10. Correct the spelling of the dependency in the first line of the **server.js** file. Commit and push changes to the application to the Git repository. Rebuild the application. After the application builds, verify that the application is running.

11. Create a route for the application and test access to the application. Expect an error message. Review the logs to identify the error.

12. Replace **process.environment** with **process.env** in the **server.js** file to fix the error. Commit and push the application changes to the Git repository. Rebuild the application. When the new application deploys, verify that application does not generate errors when you access the application URL.

Evaluation

Grade your work by running the **lab troubleshoot-review grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation nodejs-app]$ lab troubleshoot-review grade
```

Finish

From **workstation**, run the **lab troubleshoot-review finish** command to complete this lab.

```
[student@workstation nodejs-app]$ lab troubleshoot-review finish
```

This concludes the lab.

Solution

Troubleshooting Containerized Applications

Performance Checklist

In this lab, you will troubleshoot the OpenShift build and deployment process for a Node.js application.

Outcomes

You should be able to identify and solve the problems raised during the build and deployment process of a Node.js application.

Before You Begin

A running OpenShift cluster.

Open a terminal on **workstation** as the **student** user and run the following command:

```
[student@workstation ~]$ lab troubleshoot-review start
```

1. Load the configuration of your classroom environment. Run the following command to load the environment variables created in the first guided exercise:

```
[student@workstation ~]$ source /usr/local/etc/ocp4.config
```

2. Enter your local clone of the **DO180-apps** Git repository and checkout the **master** branch of the course's repository to ensure you start this exercise from a known good state:

```
[student@workstation ~]$ cd ~/DO180-apps
[student@workstation DO180-apps]$ git checkout master
...output omitted...
```

3. Create a new branch to save any changes you make during this exercise:

```
[student@workstation DO180-apps]$ git checkout -b troubleshoot-review
Switched to a new branch 'troubleshoot-review'
[student@workstation DO180-apps]$ git push -u origin troubleshoot-review
...output omitted...
* [new branch]      troubleshoot-review -> troubleshoot-review
Branch troubleshoot-review set up to track remote branch troubleshoot-review from
 origin.
```

4. Log in to OpenShift using the configured user, password and Master API URL.

```
[student@workstation DO180-apps]$ oc login -u "${RHT_OCP4_DEV_USER}" \
> -p "${RHT_OCP4_DEV_PASSWORD}" "${RHT_OCP4_MASTER_API}"
Login successful.
...output omitted...
```

5. Create a new project named ***youruser*-nodejs-app**:

```
[student@workstation ~]$ oc new-project ${RHT_OCP4_DEV_USER}-nodejs-app
Now using project "youruser-nodejs-app" on server "https://
api.cluster.lab.example.com"
...output omitted...
```

6. In the **youruser-nodejs-app** OpenShift project, create a new application from the source code located **nodejs-app** directory in the Git repository at `https://github.com/`*`yourgituser`*`/DO180-apps`. Name the application **nodejs-dev**.

   Expect the build process for the application to fail. Monitor the build process and identify the build failure.

   6.1. Run the **oc new-app** command to create the Node.js application.

```
[student@workstation ~]$ oc new-app --context-dir=nodejs-app \
> https://github.com/${RHT_OCP4_GITHUB_USER}/DO180-apps#troubleshoot-review
> -i nodejs:8 --name nodejs-dev --build-env \
> npm_config_registry=http://${RHT_OCP4_NEXUS_SERVER}/repository/npm-proxy
--> Found image a2b5ec2 ...output omitted...

  Node.js 8
...output omitted...
--> Creating resources ...
    imagestream.image.openshift.io "nodejs-dev" created
    buildconfig.build.openshift.io "nodejs-dev" created
    deploymentconfig.apps.openshift.io "nodejs-dev" created
    service "nodejs-dev" created
--> Success
    Build scheduled, use 'oc logs -f bc/nodejs-dev' to track its progress.
    Application is not exposed. You can expose services to the outside world by
 executing one or more of the commands below:
     'oc expose svc/nodejs-dev'
    Run 'oc status' to view your app.
```

   6.2. Monitor build progress with the **oc logs -f bc/nodejs-dev** command:

```
[student@workstation ~]$ oc logs -f bc/nodejs-dev
Cloning "https://github.com/yourgituser/DO180-apps" ...
...output omitted...
STEP 8: RUN /usr/libexec/s2i/assemble
---> Installing application source ...
---> Installing all dependencies
npm ERR! code ETARGET
npm ERR! notarget No matching version found for express@4.20
npm ERR! notarget In most cases you or one of your dependencies are requesting
npm ERR! notarget a package version that doesn't exist.
npm ERR! notarget
npm ERR! notarget It was specified as a dependency of 'nodejs-app'
npm ERR! notarget

npm ERR! A complete log of this run can be found in:
npm ERR!     /opt/app-root/src/.npm/_logs/2019-10-28T11_30_27_657Z-debug.log
subprocess exited with status 1
subprocess exited with status 1
error: build error: error building at STEP "RUN /usr/libexec/s2i/assemble": exit
 status 1
```

The build process fails, and therefore no application is running. The build log indicates that there is no version of the **`express`** package that matches a version specification of **`4.20.x`**.

6.3. Use the **`oc get pods`** command to confirm that the application is not deployed:

```
[student@workstation ~]$ oc get pods
NAME                   READY    STATUS    RESTARTS    AGE
nodejs-dev-1-build     0/1      Error     0           2m
```

7. Update the version of the **`express`** dependency in the **`package.json`** file with a value of **`4.x`**. Commit and push the changes to the Git repository.

7.1. Edit the **`package.json`** file in the **`nodejs-app`** subdirectory, and change the version of the **`express`** dependency to **`4.x`**. Save the file.

```
[student@workstation DO180-apps]$ cd nodejs-app
```

```
[student@workstation nodejs-app]$ sed -i s/4.20/4.x/ package.json
```

The file contains the following content:

```
[student@workstation nodejs-app]$ cat package.json
{
  "name": "nodejs-app",
  "version": "1.0.0",
  "description": "Hello World App",
  "main": "server.js",
  "author": "Red Hat Training",
  "license": "ASL",
  "dependencies": {
    "express": "4.x",
    "html-errors": "latest"
  }
}
```

7.2. Commit and push the changes made to the project.

From the terminal window, run the following command to commit and push the changes:

```
[student@workstation nodejs-app]$ git commit -am "Fixed Express release"
...output omitted...
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-app]$ git push
...output omitted...
To https://github.com/yourgituser/DO180-apps/
   ef6557d..73a82cd  troubleshoot-review -> troubleshoot-review
```

8. Rebuild the application. Verify that the application builds without errors.

8.1. Use the **`oc start-build`** command to rebuild the application.

```
[student@workstation nodejs-app]$ oc start-build bc/nodejs-dev
build.build.openshift.io/nodejs-dev-2 started
```

8.2. Use the **`oc logs`** command to monitor the build process logs:

```
[student@workstation nodejs-app]$ oc logs -f bc/nodejs-dev
Cloning "https://github.com/yourgituser/DO180-apps" ...
...output omitted...
Pushing image ...image-registry.svc:5000/nodejs-app/nodejs-dev:latest ...
...output omitted...
Push successful
```

The build succeeds if an image is pushed to the internal OpenShift registry.

9. Verify that the application is not running because of a runtime error. Review the logs and identify the problem.

9.1. Use the **`oc get pods`** command to check the status of the deployment of the application pod. Eventually, you see that the first application deployment has a status of **`CrashLoopBackoff`**.

```
[student@workstation nodejs-app]$ oc get pods
NAME                  READY   STATUS             RESTARTS   AGE
nodejs-dev-1-86gg5    0/1     CrashLoopBackOff   6          7m
nodejs-dev-1-build    0/1     Error              0          26m
nodejs-dev-2-build    0/1     Completed          0          11m
```

9.2. Use the **`oc logs -f dc/nodejs-dev`** command to follow the logs for the application deployment:

```
[student@workstation nodejs-app]$ oc logs -f dc/nodejs-dev
Environment:
  DEV_MODE=false
  NODE_ENV=production
  DEBUG_PORT=5858
...output omitted...

Error: Cannot find module 'http-error'
```

```
...output omitted...

npm info nodejs-app@1.0.0 Failed to exec start script
...output omitted...
npm ERR!
npm ERR! Failed at the nodejs-app@1.0.0 start script 'node server.js'.
npm ERR! This is most likely a problem with the nodejs-app package,
npm ERR! not with npm itself.
...output omitted...
```

The log indicates that the **server.js** file attempts to load a module named **http-error**. The **dependencies** variable in the **packages** file indicates that the module name is **html-errors**, not **http-error**.

10. Correct the spelling of the dependency in the first line of the **server.js** file. Commit and push changes to the application to the Git repository. Rebuild the application. After the application builds, verify that the application is running.

10.1. Correct the spelling of the module in the first line of the **server.js** from **http-error** to **html-errors**. Save the file.

```
[student@workstation nodejs-app]$ sed -i s/http-error/html-errors/ server.js
```

The file contains the following content:

```
[student@workstation nodejs-app]$ cat server.js
var createError = require('html-errors');

var express = require('express');
app = express();

app.get('/', function (req, res) {
  res.send('Hello World from pod: ' + process.environment.HOSTNAME + '\n')
});

app.listen(8080, function () {
  console.log('Example app listening on port 8080!');
});
```

10.2. Commit and push the changes made to the project.

```
[student@workstation nodejs-app]$ git commit -am "Fixed module typo"
...output omitted...
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-app]$ git push
...output omitted...
To https://github.com/yourgituser/DO180-apps/
   ef6557d..73a82cd  troubleshoot-review -> troubleshoot-review
```

10.3. Use the **oc start-build** command to rebuild the application.

```
[student@workstation nodejs-app]$ oc start-build bc/nodejs-dev
build "nodejs-dev-3" started
```

10.4. Use the **`oc logs`** command to monitor the build process logs:

```
[student@workstation nodejs-app]$ oc logs -f bc/nodejs-dev
Cloning "https://github.com/yourgituser/DO180-apps" ...
...output omitted...
Pushing image ...-image-registry.svc:5000/nodejs-app/nodejs-dev:latest ...
...output omitted...
Push successful
```

10.5. Use the **`oc get pods -w`** command to monitor the deployment of pods for the **`nodejs-dev`** application:

```
[student@workstation nodejs-app]$ oc get pods -w
NAME                 READY    STATUS      RESTARTS   AGE
nodejs-dev-1-build   0/1      Error       0          6h9m
nodejs-dev-2-build   0/1      Completed   0          5h55m
nodejs-dev-2-xt8q4   1/1      Running     0          4m
nodejs-dev-3-build   0/1      Completed   0          7m57s
```

After a third build, the second deployment results in a status of **`Running`**.

11. Create a route for the application and test access to the application. Expect an error message. Review the logs to identify the error.

11.1. Use the **`oc expose`** command to create a route for the **`nodejs-dev`** application:

```
[student@workstation nodejs-app]$ oc expose svc nodejs-dev
route.route.openshift.io/nodejs-dev exposed
```

11.2. Use the **`oc get route`** command to retrieve the URL of **`nodejs-dev`** route:

```
[student@workstation nodejs-app]$ oc get route
NAME          HOST/PORT                                             ...
nodejs-dev    nodejs-dev-your_user-nodejs-app.wildcard_domain    ...
```

11.3. Use the **`curl`** to access the route. Expect an error message to display.

```
[student@workstation nodejs-app]$ curl \
> nodejs-dev-${RHT_OCP4_DEV_USER}-nodejs-app.${RHT_OCP4_WILDCARD_DOMAIN}
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Error</title>
</head>
<body>
<pre>Internal Server Error</pre>
</body>
</html>
```

11.4. Review the logs for the **`nodejs-dev`** deployment configuration:

```
[student@workstation nodejs-app]$ oc logs dc/nodejs-dev
Environment:
  DEV_MODE=false
  NODE_ENV=production
  DEBUG_PORT=5858
Launching via npm...
npm info it worked if it ends with ok
npm info using npm@2.15.1
npm info using node@v4.6.2
npm info prestart nodejs-app@1.0.0
npm info start nodejs-app@1.0.0

> nodejs-app@1.0.0 start /opt/app-root/src
> node server.js

Example app listening on port 8080!
TypeError: Cannot read property 'HOSTNAME' of undefined
...output omitted...
```

The corresponding section of the **server.js** file is:

```
app.get('/', function (req, res) {
  res.send('Hello World from pod: ' + process.environment.HOSTNAME + '\n')
});
```

A **process** object in Node.js contains a reference to a **env** object, not a **environment** object.

12. Replace **process.environment** with **process.env** in the **server.js** file to fix the error. Commit and push the application changes to the Git repository. Rebuild the application. When the new application deploys, verify that application does not generate errors when you access the application URL.

    12.1. Replace **process.environment** with **process.env** in the **server.js** file to fix the error.

```
[student@workstation nodejs-app]$ sed -i \
> s/process.environment/process.env/ server.js
```

The file contains the following content:

```
[student@workstation nodejs-app]$ cat server.js
var createError = require('html-errors');

var express = require('express');
app = express();

app.get('/', function (req, res) {
  res.send('Hello World from pod: ' + process.env.HOSTNAME + '\n')
});
```

```
app.listen(8080, function () {
  console.log('Example app listening on port 8080!');
});
```

12.2. Commit and push the changes made to the project.

```
[student@workstation nodejs-app]$ git commit -am "Fixed process.env"
...output omitted...
 1 file changed, 1 insertion(+), 1 deletion(-)
[student@workstation nodejs-app]$ git push
...output omitted...
To https://github.com/yourgituser/DO180-apps/
   ef6557d..73a82cd  troubleshoot-review -> troubleshoot-review
```

12.3. Use the **oc start-build** command to rebuild the application.

```
[student@workstation nodejs-app]$ oc start-build bc/nodejs-dev
build.build.openshift.io/nodejs-dev-4 started
```

12.4. Use the **oc logs** command to monitor the build process logs:

```
[student@workstation nodejs-app]$ oc logs -f bc/nodejs-dev
Cloning "https://github.com/yourgituser/DO180-apps" ...
...output omitted...
Pushing image ...image-registry.svc:5000/nodejs-app/nodejs-dev:latest ...
...output omitted...
Push successful
```

12.5. Use the **oc get pods** command to monitor the deployment of pods for the **nodejs-dev** application:

```
[student@workstation nodejs-app]$ oc get pods
NAME                   READY   STATUS      RESTARTS   AGE
nodejs-dev-1-build     0/1     Error         0          7h
nodejs-dev-2-build     0/1     Completed     0          6h
nodejs-dev-3-build     0/1     Completed     0          1h
nodejs-dev-3-m7wvj     1/1     Running       0          46s
nodejs-dev-4-build     0/1     Completed     0          3m
```

After a fourth build, the third deployment has a status of **Running**.

12.6. Use the **curl** command to test the application. The application displays a **Hello World** message containing the host name of the application pod:

```
[student@workstation nodejs-app]$ curl \
> nodejs-dev-${RHT_OCP4_DEV_USER}-nodejs-app.${RHT_OCP4_WILDCARD_DOMAIN}
Hello World from pod: nodejs-dev-3-m7wvj
```

Evaluation

Grade your work by running the **lab troubleshoot-review grade** command from your **workstation** machine. Correct any reported failures and rerun the script until successful.

```
[student@workstation nodejs-app]$ lab troubleshoot-review grade
```

Finish

From **`workstation`**, run the **`lab troubleshoot-review finish`** command to complete this lab.

```
[student@workstation nodejs-app]$ lab troubleshoot-review finish
```

This concludes the lab.

Summary

In this chapter, you learned:

- Applications typically log activity, such as events, warnings and errors, to aid the analysis of application behavior.
- Container applications should print log data to standard output, instead of to a file, to enable easy access to logs.
- To review the logs for a container deployed locally with Podman, use the **`podman logs`** command.
- Use the **`oc logs`** command to access logs for **`BuildConfig`** and **`DeploymentConfig`** objects, as well as individual pods within an OpenShift project.
- The **`-f`** option allows you to monitor the log output in near real-time for both the **`podman logs`** and **`oc logs`** commands.
- Use the **`oc port-forward`** command to connect directly to a port on an application pod. You should only leverage this technique on non-production pods, because interactions can alter the behavior of the pod.