Red Hat build of Node.js 20

Node.js Runtime Guide

Use Node.js 20 to develop scalable network applications that run on OpenShift and on stand-alone RHEL

Last Updated: 2024-04-02

Red Hat build of Node.js 20 Node.js Runtime Guide

Use Node.js 20 to develop scalable network applications that run on OpenShift and on stand-alone RHEL

Legal Notice

Abstract

This guide provides details on using the Node.js runtime.

Table of Contents

PREFACE

This guide covers concepts as well as practical details needed by developers to use the Node.js runtime.

CHAPTER 1. INTRODUCTION TO APPLICATION DEVELOPMENT WITH NODE.JS

This section explains the basic concepts of application development with Red Hat runtimes. It also provides an overview about the Node.js runtime.

1.1. OVERVIEW OF APPLICATION DEVELOPMENT WITH RED HAT RUNTIMES

Red Hat OpenShift is a container application platform, which provides a collection of cloud-native runtimes. You can use the runtimes to develop, build, and deploy Java or JavaScript applications on OpenShift.

Application development using Red Hat Runtimes for OpenShift includes:

- A collection of runtimes, such as, Eclipse Vert.x, Thorntail, Spring Boot, and so on, designed to run on OpenShift.
- A prescriptive approach to cloud-native development on OpenShift.

OpenShift helps you manage, secure, and automate the deployment and monitoring of your applications. You can break your business problems into smaller microservices and use OpenShift to deploy, monitor, and maintain the microservices. You can implement patterns such as circuit breaker, health check, and service discovery, in your applications.

Cloud-native development takes full advantage of cloud computing.

You can build, deploy, and manage your applications on:

OpenShift Container Platform

A private on-premise cloud by Red Hat.

Red Hat CodeReady Studio

An integrated development environment (IDE) for developing, testing, and deploying applications.

This guide provides detailed information about the Node.js runtime. For more information on other runtimes, see the relevant runtime documentation.

1.2. OVERVIEW OF NODE.JS

Node.js is based on the V8 JavaScript engine from Google and allows you to write server-side JavaScript applications. It provides an I/O model based on events and non-blocking operations that enables you to write efficient applications. Node.js also provides a large module ecosystem called npm. Check out Additional Resources for further reading on Node.js.

The Node.js runtime enables you to run Node.js applications and services on OpenShift while providing all the advantages and conveniences of the OpenShift platform such as rolling updates, continuous delivery pipelines, service discovery, and canary deployments. OpenShift also makes it easier for your applications to implement common microservice patterns such as externalized configuration, health check, circuit breaker, and failover.

Red Hat provides different supported releases of Node.js. For more information how to get support, see Getting Node.js and support from Red Hat .

1.2.1. Supported Architectures by Node.js

Node.js supports the following architectures:

- x86_64 (AMD64)
- IBM Z (s390x) in the OpenShift environment
- IBM Power Systems (ppc64le) in the OpenShift environment

1.2.2. Support for Federal Information Processing Standard (FIPS)

The Federal Information Processing Standards (FIPS) provides guidelines and requirements for improving security and interoperability across computer systems and networks. The FIPS 140-2 and 140-3 series apply to cryptographic modules at both the hardware and software levels.

The Federal Information Processing Standard (FIPS) Publication 140-2 is a computer security standard developed by the U.S. Government and industry working group to validate the quality of cryptographic modules. See the official FIPS publications at NIST Computer Security Resource Center .

Red Hat Enterprise Linux (RHEL) provides an integrated framework to enable FIPS 140-2 compliance system-wide. When operating in the FIPS mode, software packages using cryptographic libraries are self-configured according to the global policy.

To learn about compliance requirements, see the Red Hat Government Standards page.

Red Hat build of Node.js runs on a FIPS-enabled RHEL system and uses FIPS-certified libraries provided by RHEL.

1.2.2.1. Additional resources

- For more information on how to install RHEL with FIPS mode enabled, see Installing a RHEL 8 system with FIPS mode enabled.
- For more information on how to enable FIPS mode after installing RHEL, see Switching the system to FIPS mode.

1.2.2.2. Verifying that Node.js is running in FIPS mode

You can use **crypto.fips** to verify that Node.js is running in FIPS mode.

Prerequisites

- FIPS is enabled on the RHEL host.

Procedure

1. In your Node.js project, create an application file named, for example, **app.js**.
2. In the **app.js** file, enter the following details:

```
const crypto = require('crypto');
console.log(crypto.fips);
```

3. Save the **app.js** file.

Verification

- In your Node.js project, run the **app.js** file:

```
node app.js
```

If FIPS is enabled, the application prints **1** to the console. If FIPS is disabled, the application prints **0** to the console.

CHAPTER 2. DEVELOPING AND DEPLOYING A NODE.JS APPLICATION

You can create new Node.js applications and deploy them to OpenShift.

2.1. DEVELOPING A NODE.JS APPLICATION

For a basic Node.js application, you must create a JavaScript file containing Node.js methods.

Prerequisites

- **npm** installed.

Procedure

1. Create a new directory **myApp**, and navigate to it.

```
$ mkdir myApp
$ cd MyApp
```

   This is the root directory for the application.

2. Initialize your application with **npm**.
   The rest of this example assumes the entry point is **app.js**, which you are prompted to set when running **npm init**.

```
$ cd myApp
$ npm init
```

3. Create the entry point in a new file called **app.js**.

   Example app.js

```
const http = require('http');

const server = http.createServer((request, response) => {
  response.statusCode = 200;
  response.setHeader('Content-Type', 'application/json');

  const greeting = {content: 'Hello, World!'};

  response.write(JSON.stringify(greeting));
  response.end();
});

server.listen(8080, () => {
  console.log('Server running at http://localhost:8080');
});
```

4. Start your application.

```
$ node app.js
Server running at http://localhost:8080
```

5. Using **curl** or your browser, verify your application is running at **http://localhost:8080**.

```
$ curl http://localhost:8080
{"content":"Hello, World!"}
```

Additional information

- The Node.js runtime provides the core Node.js API which is documented in the Node.js API documentation.

2.2. DEPLOYING A NODE.JS APPLICATION TO OPENSHIFT

To deploy your Node.js application to OpenShift, add **nodeshift** to the application, configure the **package.json** file and then deploy using **nodeshift**.

2.2.1. Preparing Node.js application for OpenShift deployment

To prepare a Node.js application for OpenShift deployment, you must perform the following steps:

- Add **nodeshift** to the application.
- Add **openshift** and **start** entries to the **package.json** file.

Prerequisites

- **npm** installed.

Procedure

1. Add **nodeshift** to your application.

```
$ npm install nodeshift --save-dev
```

2. Add the **openshift** and **start** entries to the **scripts** section in **package.json**.

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "openshift": "nodeshift --expose --
dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12",
    "start": "node app.js",
    ...
  }
  ...
}
```

The **openshift** script uses **nodeshift** to deploy the application to OpenShift.

NOTE

Universal base images and RHEL images are available for Node.js. See the Node.js release notes for more information on image names.

3. *Optional*: Add a **files** section in **package.json**.

```
{
  "name": "myApp",
  "version": "1.0.0",
  "description": "",
  "main": "app.js",
  "scripts": {
    ...
  },
  "files": [
    "package.json",
    "app.js"
  ]
  ...
}
```

The **files** section tells **nodeshift** what files and directories to include when deploying to OpenShift. **nodeshift** uses the **node-tar** module to create a tar file based on the files and directories you list in the **files** section. This tar file is used when **nodeshift** deploys your application to OpenShift. If the **files** section is not specified, **nodeshift** will send the entire current directory, excluding:

- **node_modules/**
- **.git/**
- **tmp/**
  It is recommended that you include a **files** section in **package.json** to avoid including unnecessary files when deploying to OpenShift.

2.2.2. Deploying a Node.js application to OpenShift

You can deploy a Node.js application to OpenShift using **nodeshift**.

Prerequisites

- The **oc** CLI client installed.
- **npm** installed.
- Ensure all the ports used by your application are correctly exposed when configuring your routes.

Procedure

1. Log in to your OpenShift instance with the **oc** client.

```
$ oc login ...
```

2. Use **nodeshift** to deploy the application to OpenShift.

```
$ npm run openshift
```

2.3. DEPLOYING A NODE.JS APPLICATION TO STAND-ALONE RED HAT ENTERPRISE LINUX

You can deploy a Node.js application to stand-alone Red Hat Enterprise Linux using **npm**.

Prerequisites

- A Node.js application.
- npm 6.14.8 installed
- RHEL 7 or RHEL 8 installed.
- Node.js installed

Procedure

1. If you have specified additional dependencies in the **package.json** file of your project, ensure that you install them before running your applications.

```
$ npm install
```

2. Deploy the application from the application's root directory.

```
$ node app.js
Server running at http://localhost:8080
```

Verification steps

1. Use **curl** or your browser to verify your application is running at **http://localhost:8080**

```
$ curl http://localhost:8080
```

CHAPTER 3. DEBUGGING YOUR NODE.JS BASED APPLICATION

This section contains information about debugging your Node.js–based application and using debug logging in both local and remote deployments.

3.1. REMOTE DEBUGGING

To remotely debug an application, you need to start it in a debugging mode and attach a debugger to it.

3.1.1. Starting your application locally and attaching the native debugger

The native debugger enables you to debug your Node.js–based application using the built-in debugging client.

Prerequisites

- An application you want to debug.

Procedure

1. Start the application with the debugger enabled.
   The native debugger is automatically attached and provides a debugging prompt.

   Example application with the debugger enabled

   ```
   $ node inspect app.js
   < Debugger listening on ws://127.0.0.1:9229/12345678-aaaa-bbbb-cccc-0123456789ab
   < For help see https://nodejs.org/en/docs/inspector
   < Debugger attached.
   ...
   debug>
   ```

   If you have a different entry point for your application, you need to change the command to specify that entry point:

   ```
   $ node inspect path/to/entrypoint
   ```

   For example, when using the express generator to create your application, the entry point is set to **./bin/www** by default.

2. Use the debugger prompt to perform debugging commands.

3.1.2. Starting your application locally and attaching the V8 inspector

The V8 inspector enables you to debug your Node.js–based application using other tools, such as Chrome DevTools, that use the Chrome Debugging Protocol.

Prerequisites

- An application you want to debug.

- The V8 inspector installed, such as the one provided in the Google Chrome Browser.

Procedure

1. Start your application with the V8 inspector integration enabled.

```
$ node --inspect app.js
```

   If you have a different entry point for your application, you need to change the command to specify that entry point:

```
$ node --inspect path/to/entrypoint
```

   For example, when using the express generator to create your application, the entry point is set to **./bin/www** by default.

2. Attach the V8 inspector and perform debugging commands.
   For example, if using Google Chrome:

   a. Navigate to **chrome://inspect**.

   b. Select your application from below *Remote Target*.

   c. You can now see the source of your application and can perform debugging actions.

3.1.3. Starting your application on OpenShift in debugging mode

To debug your Node.js-based application on OpenShift remotely, you must set the **NODE_ENV** environment variable inside the container to **development** and configure port forwarding so that you can connect to your application from a remote debugger.

Prerequisites

- Your application running on OpenShift.
- The **oc** binary installed.
- The ability to execute the **oc port-forward** command in your target OpenShift environment.

Procedure

1. Using the **oc** command, list the available deployment configurations:

```
$ oc get dc
```

2. Set the **NODE_ENV** environment variable in the deployment configuration of your application to **development** to enable debugging. For example:

```
$ oc set env dc/MY_APP_NAME NODE_ENV=development
```

3. Redeploy the application if it is not set to redeploy automatically on configuration change. For example:

```
$ oc rollout latest dc/MY_APP_NAME
```

4. Configure port forwarding from your local machine to the application pod:

   a. List the currently running pods and find one containing your application:

   ```
   $ oc get pod
   NAME                    READY    STATUS     RESTARTS   AGE
   MY_APP_NAME-3-1xrsp     0/1      Running    0          6s
   ...
   ```

   b. Configure port forwarding:

   ```
   $ oc port-forward MY_APP_NAME-3-1xrsp $LOCAL_PORT_NUMBER:5858
   ```

   Here, **$LOCAL_PORT_NUMBER** is an unused port number of your choice on your local machine. Remember this number for the remote debugger configuration.

5. Attach the V8 inspector and perform debugging commands.
   For example, if using Google Chrome:

   a. Navigate to **chrome://inspect**.

   b. Click *Configure*.

   c. Add **127.0.0.1:$LOCAL_PORT_NUMBER**.

   d. Click *Done*.

   e. Select your application from below *Remote Target*.

   f. You can now see the source of your application and can perform debugging actions.

6. When you are done debugging, unset the **NODE_ENV** environment variable in your application pod. For example:

   ```
   $ oc set env dc/MY_APP_NAME NODE_ENV-
   ```

3.2. DEBUG LOGGING

Debug logging is a way to add detailed information to the application log when debugging. This allows you to:

- Keep minimal logging output during normal operation of the application for improved readability and reduced disk space usage.
- View detailed information about the inner workings of the application when resolving issues.

3.2.1. Add debug logging

This example uses the debug package, but there are also other packages available that can handle debug logging.

Prerequisites

- You have an application that you want to debug.

Procedure

1. Add the **debug** logging definition.

```
const debug = require('debug')('myexample');
```

2. Add debug statements.

```
app.use('/api/greeting', (request, response) => {
  const name = request.query ? request.query.name : undefined;
  //log name in debugging
  debug('name: '+name);
  response.send({content: `Hello, ${name || 'World'}`});
});
```

3. Add the debug module to **package.json**.

```
...
"dependencies": {
    "debug": "^3.1.0"
  }
```

Depending on your application, this module may already be included. For example, when using the express generator to create your application, the **debug** module is already added to **package.json**.

4. Install the application dependencies.

```
$ npm install
```

3.2.2. Accessing debug logs on localhost

Use the **DEBUG** environment variable when starting your application to enable debug logging.

Prerequisites

- An application with debug logging.

Procedure

1. Set the **DEBUG** environment variable when starting your application to enable debug logging.

```
$ DEBUG=myexample npm start
```

The **debug** module can use wildcards to filter debugging messages. This is set using the **DEBUG** environment variable.

2. Test your application to invoke debug logging.
For example, the following command is based on an example REST API level 0 application where debug logging is set to log the **name** variable in the **/api/greeting** method:

```
$ curl http://localhost:8080/api/greeting?name=Sarah
```

3. View your application logs to see your debug messages.

```
myexample name: Sarah +3m
```

3.2.3. Accessing Node.js debug logs on OpenShift

Use the the **DEBUG** environment variable in your application pod in OpenShift to enable debug logging.

Prerequisites

- An application with debug logging.
- The **oc** CLI client installed.

Procedure

1. Use the **oc** CLI client to log into your OpenShift instance.

```
$ oc login ...
```

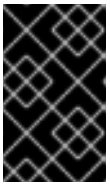
2. Deploy your application to OpenShift.

```
$ npm run openshift
```

This runs the **openshift** npm script, which wraps direct calls to nodeshift.

3. Find the name of your pod and follow the logs to watch it start.

```
$ oc get pods
....
$ oc logs -f pod/POD_NAME
```



IMPORTANT

After your pod has started, leave this command running and execute the remaining steps in a new terminal window. This allows you to *follow* the logs and see new entries made to it.

4. Test your application.
For example, the following command is based on an example REST API level 0 application where debug logging is set to log the **name** variable in the **/api/greeting** method:

```
$ oc get routes
...
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

5. Return to your pod logs and notice there are no debug logging messages in the logs.

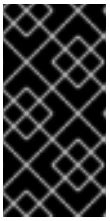
6. Set the **DEBUG** environment variable to enable debug logging.

```
$ oc get dc
...
$ oc set env dc DC_NAME DEBUG=myexample
```

7. Return to your pod logs to watch the update roll out.
   After the update has rolled out, your pod will stop and you will no longer be following the logs.

8. Find the name of your new pod and follow the logs.

```
$ oc get pods
....
$ oc logs -f pod/POD_NAME
```



IMPORTANT

After your pod has started, leave this command running and execute the remaining steps in a different terminal window. This allows you to *follow* the logs and see new entries made to it. Specifically, the logs will show your debug messages.

9. Test the application to invoke debug logging.

```
$ oc get routes
...
$ curl $APPLICATION_ROUTE/api/greeting?name=Sarah
```

10. Return to your pod logs to see the debug messages.

```
...
myexample name: Sarah +3m
```

To disable debug logging, remove the **DEBUG** environment variable from the pod:

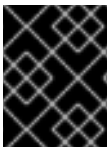
```
$ oc set env dc DC_NAME DEBUG-
```

Additional resources

More details on environment variables are available in the OpenShift documentation.

APPENDIX A. ABOUT NODESHIFT

Nodeshift is a module for running OpenShift deployments with Node.js projects.



IMPORTANT

Nodeshift assumes you have the **oc** CLI client installed, and you are logged into your OpenShift cluster. Nodeshift also uses the current project the **oc** CLI client is using.

Nodeshift uses resource files in the **.nodeshift** folder located at the root of the project to handle creating OpenShift Routes, Services and DeploymentConfigs. More details on Nodeshift are available on the Nodeshift project page .

APPENDIX B. UPDATING THE DEPLOYMENT CONFIGURATION OF AN EXAMPLE APPLICATION

The deployment configuration for an example application contains information related to deploying and running the application in OpenShift, such as route information or readiness probe location. The deployment configuration of an example application is stored in a set of YAML files. For examples that use the Fabric8 Maven Plugin, the YAML files are located in the **src/main/fabric8/** directory. For examples using Nodeshift, the YAML files are located in the **.nodeshift** directory.

IMPORTANT

The deployment configuration files used by the Fabric8 Maven Plugin and Nodeshift do not have to be full OpenShift resource definitions. Both Fabric8 Maven Plugin and Nodeshift can take the deployment configuration files and add some missing information to create a full OpenShift resource definition. The resource definitions generated by the Fabric8 Maven Plugin are available in the **target/classes/META-INF/fabric8/** directory. The resource definitions generated by Nodeshift are available in the **tmp/nodeshift/resource/** directory.

Prerequisites

- An existing example project.
- The **oc** CLI client installed.

Procedure

1. Edit an existing YAML file or create an additional YAML file with your configuration update.
   - For example, if your example already has a YAML file with a **readinessProbe** configured, you could change the **path** value to a different available path to check for readiness:

```
spec:
  template:
    spec:
      containers:
        readinessProbe:
          httpGet:
            path: /path/to/probe
            port: 8080
            scheme: HTTP
...
```

   - If a **readinessProbe** is not configured in an existing YAML file, you can also create a new YAML file in the same directory with the **readinessProbe** configuration.
2. Deploy the updated version of your example using Maven or npm.
3. Verify that your configuration updates show in the deployed version of your example.

```
$ oc export all --as-template='my-template'

apiVersion: template.openshift.io/v1
kind: Template
```

```
metadata:
  creationTimestamp: null
  name: my-template
objects:
- apiVersion: template.openshift.io/v1
  kind: DeploymentConfig
  ...
  spec:
    ...
    template:
      ...
      spec:
        containers:
          ...
          livenessProbe:
            failureThreshold: 3
            httpGet:
              path: /path/to/different/probe
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 60
            periodSeconds: 30
            successThreshold: 1
            timeoutSeconds: 1
          ...
```

Additional resources

If you updated the configuration of your application directly using the web-based console or the **oc** CLI client, export and add these changes to your YAML file. Use the **oc export all** command to show the configuration of your deployed application.

APPENDIX C. CONFIGURING A JENKINS FREESTYLE PROJECT TO DEPLOY YOUR NODE.JS APPLICATION WITH NODESHIFT

Similar to using nodeshift from your local host to deploy a Node.js application, you can configure Jenkins to use nodeshift to deploy a Node.js application.

Prerequisites

- Access to an OpenShift cluster.
- The Jenkins container image running on same OpenShift cluster.
- The Node.js plugin installed on your Jenkins server.
- A Node.js application configured to use nodeshift and the Red Hat base image.

  Example using the Red Hat base image with nodeshift

  ```
  $ nodeshift --dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12 ...
  ```

- The source of the application available in GitHub.

Procedure

1. Create a new OpenShift project for your application:
   a. Open the OpenShift Web console and log in.
   b. Click *Create Project* to create a new OpenShift project.
   c. Enter the project information and click *Create*.
2. Ensure Jenkins has access to that project.
   For example, if you configured a service account for Jenkins, ensure that account has **edit** access to the project of your application.
3. Create a new freestyle Jenkins project on your Jenkins server:
   a. Click *New Item*.
   b. Enter a name, choose *Freestyle project*, and click *OK*.
   c. Under *Source Code Management*, choose *Git* and add the GitHub url of your application.
   d. Under *Build Environment*, make sure *Provide Node & npm bin/ folder to PATH* is checked and the Node.js environment is configured.
   e. Under *Build*, choose *Add build step* and select **Execute Shell**.
   f. Add the following to the *Command* area:

   ```
   npm install -g nodeshift
   nodeshift --dockerImage=registry.access.redhat.com/rhscl/ubi8/nodejs-12 --
   namespace=MY_PROJECT
   ```

-

Substitute **MY_PROJECT** with the name of the OpenShift project for your application.

g. Click *Save*.

4. Click *Build Now* from the main page of the Jenkins project to verify your application builds and deploys to the OpenShift project for your application.
   You can also verify that your application is deployed by opening the route in the OpenShift project of the application.

Next steps

- Consider adding GITSCM polling or using the **Poll SCM** build trigger. These options enable builds to run every time a new commit is pushed to the GitHub repository.
- Consider adding nodeshift as a global package when configuring the Node.js plugin. This allows you to omit **npm install -g nodeshift** when adding your **Execute Shell** build step.
- Consider adding a build step that executes tests before deploying.

APPENDIX D. BREAKDOWN OF PACKAGE.JSON PROPERTIES

nodejs-rest-http/package.json

```
{
  "name": "nodejs-rest-http",
  "version": "4.0.0",
  "author": "Red Hat, Inc.",
  "license": "Apache-2.0",
  "scripts": {
    "pretest": "eslint --ignore-path .gitignore .",
    "test": "nyc --reporter=lcov mocha", (1)
    "prepare": "echo 'To confirm CVE compliance, run \"npm audit\"' ",
    "release": "standard-version -a",
    "openshift": "nodeshift --dockerImage=registry.access.redhat.com/ubi8/nodejs-16", (2)
    "start": "node ." (3)
  },
  "main": "./bin/www", (4)
  "standard-version": {
    "scripts": {
      "postbump": "npm run postinstall && node release.js",
      "precommit": "git add .openshiftio/application.yaml"
    }
  },
  "repository": {
    "type": "git",
    "url": "git://github.com/nodeshift-starters/nodejs-rest-http.git"
  },
  "files": [ (5)
    "package.json",
    "app.js",
    "public",
    "bin",
    "LICENSE"
  ],
  "bugs": {
    "url": "https://github.com/nodeshift-starters/nodejs-rest-http/issues"
  },
  "homepage": "https://github.com/nodeshift-starters/nodejs-rest-http",
  "devDependencies": { (6)
    "eslint": "^7.32.0",
    "eslint-config-semistandard": "^16.0.0",
    "js-yaml": "^4.1.0",
    "mocha": "^9.1.3",
    "nodeshift": "~8.6.0",
    "nyc": "~15.1.0",
    "standard-version": "^9.3.2",
    "supertest": "~6.1.6"
  },
  "dependencies": { (7)
    "body-parser": "~1.19.0",
    "debug": "^4.3.3",
    "express": "~4.17.1",
    "pino": "^7.5.1",
```

```
    "pino-debug": "^2.0.0",
    "pino-pretty": "^7.2.0"
  }
}
```

1. A **npm** script for running unit tests. Run with **npm run test**.
2. A **npm** script for deploying this application to OpenShift Container Platform. Run with **npm run openshift**.
3. A **npm** script for starting this application. Run with **npm start**.
4. The primary entrypoint for the application when run with **npm start**.
5. Specifies the files to be included in the binary that is uploaded to OpenShift Container Platform.
6. A list of development dependencies to be installed from the **npm** registry. These are used for testing and deployment to OpenShift Container Platform.
7. A list of dependencies to be installed from the **npm** registry.

APPENDIX E. ADDITIONAL NODE.JS RESOURCES

- Node.js Home Page
- npm Home Page

APPENDIX F. APPLICATION DEVELOPMENT RESOURCES

For additional information about application development with OpenShift, see:

- OpenShift Interactive Learning Portal

APPENDIX G. THE SOURCE-TO-IMAGE (S2I) BUILD PROCESS

Source-to-Image (S2I) is a build tool for generating reproducible Docker-formatted container images from online SCM repositories with application sources. With S2I builds, you can easily deliver the latest version of your application into production with shorter build times, decreased resource and network usage, improved security, and a number of other advantages. OpenShift supports multiple build strategies and input sources.

For more information, see the Source-to-Image (S2I) Build chapter of the OpenShift Container Platform documentation.

You must provide three elements to the S2I process to assemble the final container image:

- The application sources hosted in an online SCM repository, such as GitHub.
- The S2I Builder image, which serves as the foundation for the assembled image and provides the ecosystem in which your application is running.
- Optionally, you can also provide environment variables and parameters that are used by S2I scripts.

The process injects your application source and dependencies into the Builder image according to instructions specified in the S2I script, and generates a Docker-formatted container image that runs the assembled application. For more information, check the S2I build requirements, build options and how builds work sections of the OpenShift Container Platform documentation.