2 Pipeline as code with Jenkins

This chapter covers

- How pipeline as code works with Jenkins
- An overview of Jenkinsfile structure and syntax
- Introduction to Blue Ocean, the new Jenkins user experience
- Declarative versus scripted Jenkins pipelines
- Integration of a GitFlow model within Jenkins projects
- Tips for productivity and efficiency while writing Jenkinsfiles for complex CI/CD pipelines

There's no doubt that cloud computing has had a major impact on the way companies build, scale, and maintain technology products. The ability to click a few buttons to provision machines, databases, and other infrastructure has led to an increase in developer productivity we've never seen before.

While it was easy to spin up simple cloud architectures, mistakes can easily be made while provisioning complex ones. Human error will always be present, especially when you can launch cloud infrastructure by clicking buttons on the cloud provider's web console.

The only way to avoid these kinds of errors is through automation, and infrastructure as code (IaC) is helping engineers automatically launch cloud environments quickly and without mistakes. The growth of DevOps and the adoption of its practices have led to more tooling that can implement the IaC paradigm to a larger degree.

In the past, setting up CI/CD workflow has been a manual process. It was commonly done via defining a series of individual jobs for the various pipeline tasks. Each job was configured via web forms—filling in text boxes, selecting entries from dropdown lists, and so forth. And then the series of jobs were strung together, each triggering the next, into a pipeline.

Jenkins somewhat lagged in this area until the release of Jenkins 2. Although widely used and a primary workflow tool for creating CI/CD pipelines, this way of creating and connecting Jenkins jobs to form a pipeline was challenging. It did not meet the definition of IaC. Job configurations were stored only as Extensible Markup Language (XML) files within the Jenkins configuration area. This meant that the files were not easily readable or directly modifiable. And the Jenkins application itself provided the user's primary view and access to them.

> **NOTE** Jenkins 2 is the name we are generally applying to newer versions that support the pipeline-as-code functionality, as well as other features.

Because it's an important part of each project, the pipeline configuration should be managed as code and rolled out automatically. This also allows us to manage the pipeline itself, applying the same standards that apply to application code. That's where pipeline as code comes into play.

2.1 Introducing the Jenkinsfile

Pipeline as code (PaC) describes a set of features that allow Jenkins users to define pipelined job processes with code, stored and versioned in a source repository. These features allow Jenkins to discover, manage, and run jobs for multiple source repositories and branches—eliminating the need for manual job creation and management.

PaC helps you automate the CI/CD workflows in a repeatable, consistent manner, which has many benefits:

- *Speed*—You can quickly and easily write a CI/CD workflow for sandbox, staging, and production environments, which can help you deliver your product on time.
- *Consistency*—PaC completely standardizes the setup of CI/CD, so there's a reduced possibility of any human errors or deviations.
- *Risk management*—Because the pipeline can be version-controlled, PaC allows every change to your CI/CD workflow to be documented, logged, tracked, and tested just like application code. Hence, you can revert to a working version in case of failure.
- *Efficiency*—It minimizes the introduction of human errors and helps your application's deployment run more smoothly.

The bottom line is simple: adopting the PaC paradigm will create a culture that generates better software, and will save you a lot of money, time, and headaches trying to implement complex CI/CD workflows through UIs and web forms. So how does PaC work with Jenkins?

To use PaC with Jenkins, projects must contain a file named Jenkinsfile in the code repository top-level folder. This template file contains a set of instructions, or steps, called *stages* that will be executed on Jenkins every time the development team pushes a new feature to the code repository. Because Jenkinsfile is living along with the source code, we can always pull, edit, and push the Jenkinsfile within source control, just as we would for any other file. We can also do things like code reviews on the pipeline script.

Jenkinsfile uses a domain-specific language (DSL) based on the Groovy programming language to define the entire CI/CD workflow. Figure 2.1 is an example of a classic CI/CD workflow.

Figure 2.1 CI/CD workflow

Those phases can be described in a Jenkinsfile by using the `stage` keyword. A *stage* is a block that contains a series of steps. It can be used to visualize the pipeline process. The following listing is an example of a simple Jenkinsfile for figure 2.1.

Listing 2.1 Jenkinsfile stages

```
node('workers'){
    try {
        stage('Checkout'){
            checkout scm
        }

        stage('Quality Test'){
            echo "Running quality tests"
        }

        stage('Unit Test'){
            echo "Running unit tests"
        }

        stage('Security Test'){
            echo "Running security checks"
        }

        stage('Build'){
            echo "Building artifact"
        }

        stage('Push'){
           echo "Storing artifact"
        }
```

```
        stage('Deploy'){
            echo "Deploying artifact"
        }

        stage('Acceptance Tests'){
            echo "Running post-integrations tests"
        }
    } catch(err){
        echo "Handling errors"
    } finally{
       echo "Cleaning up"
    }
}
```

We'll dive deep into the syntax in the next chapter, but for now, let's focus on what the stages are doing:

- *Checkout*—Pulls the latest changes from the source code repository, which can be GitHub, Bitbucket, Mercurial, or any SCM.
- *Quality tests*—Contains instructions on how to execute static code analysis to measure code quality, and identify bugs, vulnerabilities, and code smell. It can be automated by integrating external tools like SonarQube to fix code-quality violations and reduce technical debt.
- *Unit tests*—In this stage, unit tests are executed. If tests are successful, a code coverage report will be generated that can be consumed by Jenkins plugins to show a visual overview of the project's health and keep track of the code coverage metrics as your project grows. Code coverage can be an indication of how much your application code is executed during your tests, and can give some indication as to how well your team is applying good testing practices such test-driven development (TDD) or behavior-driven development (BDD).
- *Security tests*—Responsible for identifying project dependencies and checks if any known, publicly disclosed vulnerabilities exist. A security report will be published with the total number of findings grouped by severity (critical, high, medium, or low). A well-known open source Jenkins plugin is OWASP Dependency-Check (http://mng.bz/MvR7).
- *Build*—In this phase, the needed dependencies will be installed, the source code will be compiled, and an artifact will be built (Docker image, zip file, Maven JAR, and so forth).
- *Push*—The artifact built in the previous stage will be versioned and stored in a remote repository.
- *Deploy*—In this stage, the artifact will be deployed to a sandbox/testing environment for quality assurance or to production after the user has approved the deployment.
- *Acceptance tests*—After the changes are deployed, a series of smoke and validation tests will be executed against the deployed application to verify that the application is running as expected. The tests can be simple health checks with cURL commands or sophisticated API calls.

If any of these stages throws an exception or error, the pipeline build's status will be set to fail. This default behavior can be overridden by using `try-catch` blocks. The `finally` block can be used to clean up the Jenkins workspace (temporary files or build packages) or to execute post-script commands such as sending Slack notifications to alert the development team about the build status.

> **NOTE** Don't worry if you don't completely understand the steps of the Jenkinsfile in listing 2.1. You will get an in-depth explanation of how to implement each stage in chapters 7, 8, and 9.

One of the things that makes Jenkins a leader when it comes to CI tools is the ecosystem behind it. You can customize your Jenkins instance with free open source plugins. A must-have plugin is Pipeline Stage View (https://plugins.jenkins.io/pipeline-rest-api), shown in figure 2.2. It allows you to have a visualization of your pipeline stages. This plugin is handy when you have complex build pipelines and want to track the progress of each stage.

The pipeline output is organized as a matrix, with each row representing a run of the job, and each column mapped to a defined stage in the pipeline. When you run some builds, the stage view will appear with Checkout, Quality Test, Unit Test, Security Test, Build, Push, and Deploy columns, and one row per build showing the status of those stages. When hovering over a stage cell, you can click the Logs button to see log messages printed in that stage.

> **NOTE** Part 3 of this book covers how to create a Jenkins job and define a pipeline like the one in figure 2.2.

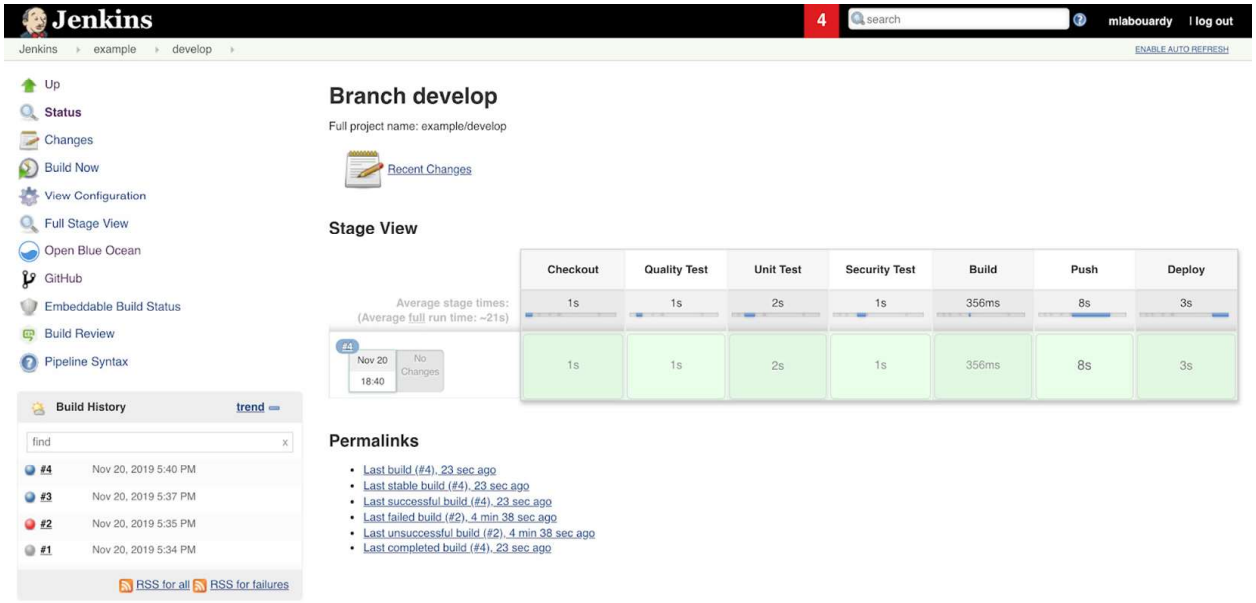

Figure 2.2 Jenkins Pipeline Stage View

You can take this UI further and install the Blue Ocean plugin (https://plugins.jenkins.io/blueocean/) to have a fast and intuitive comprehension of the CI/CD stages, as shown in figure 2.3. This plugin requires Jenkins version 2.7 or later.

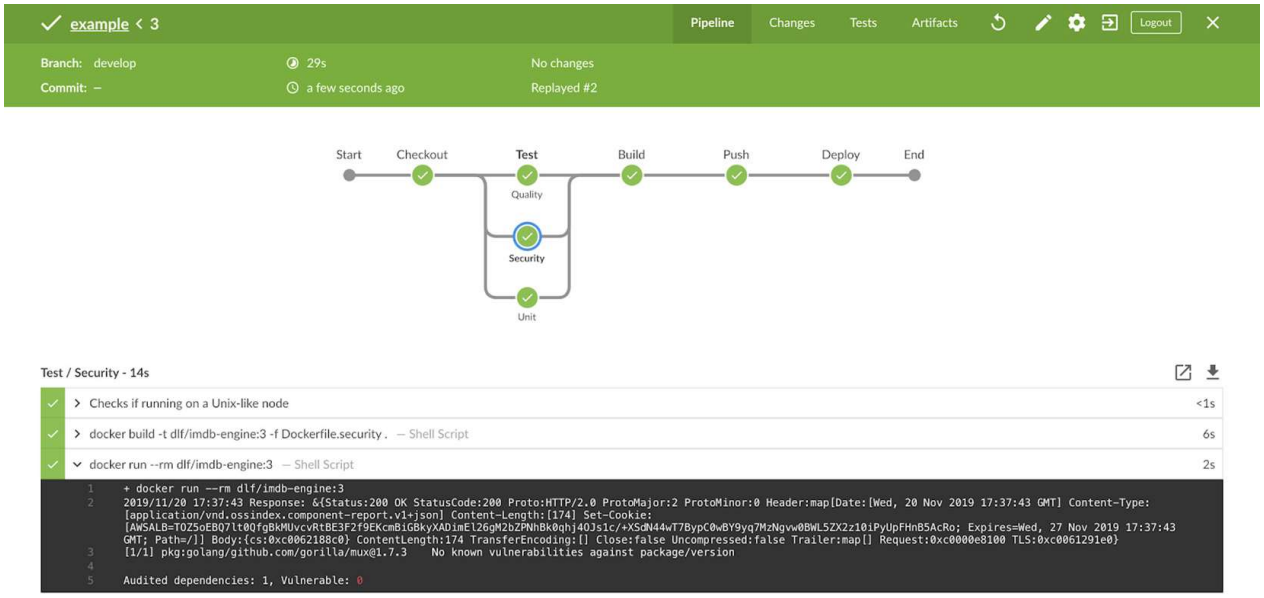

Figure 2.3 Blue Ocean plugin's detailed view of the pipeline

> **NOTE** Chapter 5 covers how to install and configure the Jenkins Blue Ocean plugin.

2.1.1 Blue Ocean plugin

You can also troubleshoot pipeline failure by clicking the stage in red to easily identify the problem without going through thousands of output logs.

One of the big concerns while choosing Jenkins is the user interface, which many users consider outdated, unintuitive, and hard to navigate when you have many projects. That's why the Jenkins core team launched Blue Ocean in April 2017 for a new, modern Jenkins user experience.

Blue Ocean is a new user experience for Jenkins, based on a modern design that allows users to graphically create, personalize, visualize, and diagnose CD pipelines. It comes bundled with the Jenkins Pipeline plugin or as a separate plugin (www.jenkins.io/doc/book/blueocean/getting-started/).

> **NOTE** The Jenkins Classic UI exists side-by-side at its usual place at JENKINS_URL/jenkins. The Blue Ocean plugin is available by appending `/blue` to the end of the Jenkins server URL.

Anyone in your team can create a CI/CD pipeline with just several clicks. Blue Ocean has seamless integration with Git and GitHub. It prompts you for credentials to access

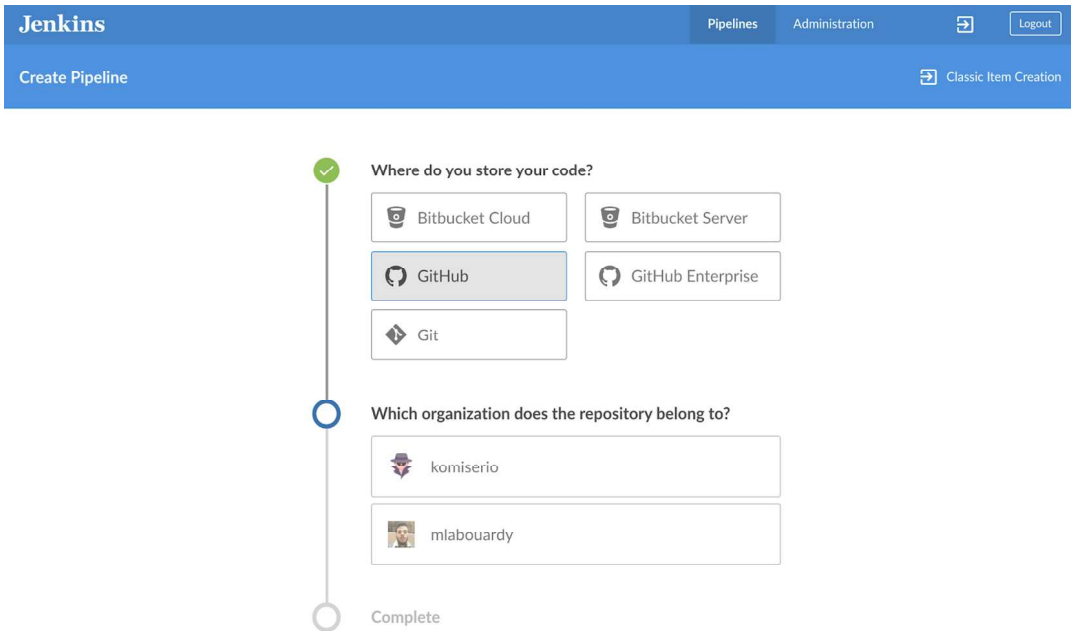

Figure 2.4 New pipeline in Blue Ocean mode

your repositories on the Git server in order to create pipelines based on those repositories (figure 2.4).

You can also create a complete CI/CD pipeline from start to finish by using the intuitive and visual pipeline editor (figure 2.5). It's a great way to write pipeline prototypes and debug pipeline stages before generating a working Jenkinsfile.

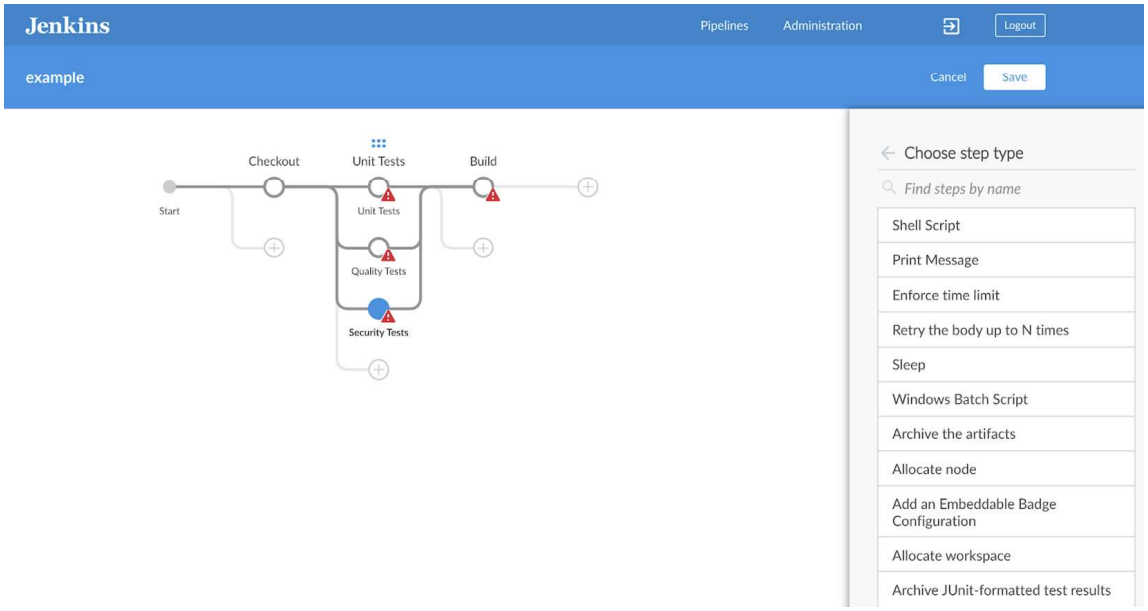

Figure 2.5 Defining stages with pipeline editor

Any pipeline created with the visual editor can also be edited in your favorite text editor, bringing all the benefits of PaC. Figure 2.6 shows an example of the pipeline script generated by pressing Ctrl-S for Windows users and Command-S for macOS users.

You can now copy the content and paste it in a new file called Jenkinsfile in your code repository, alongside the source code. Alternatively, you can upload the file directly from the Blue Ocean editor by supplying an appropriate description and the target Git branch (figure 2.7).

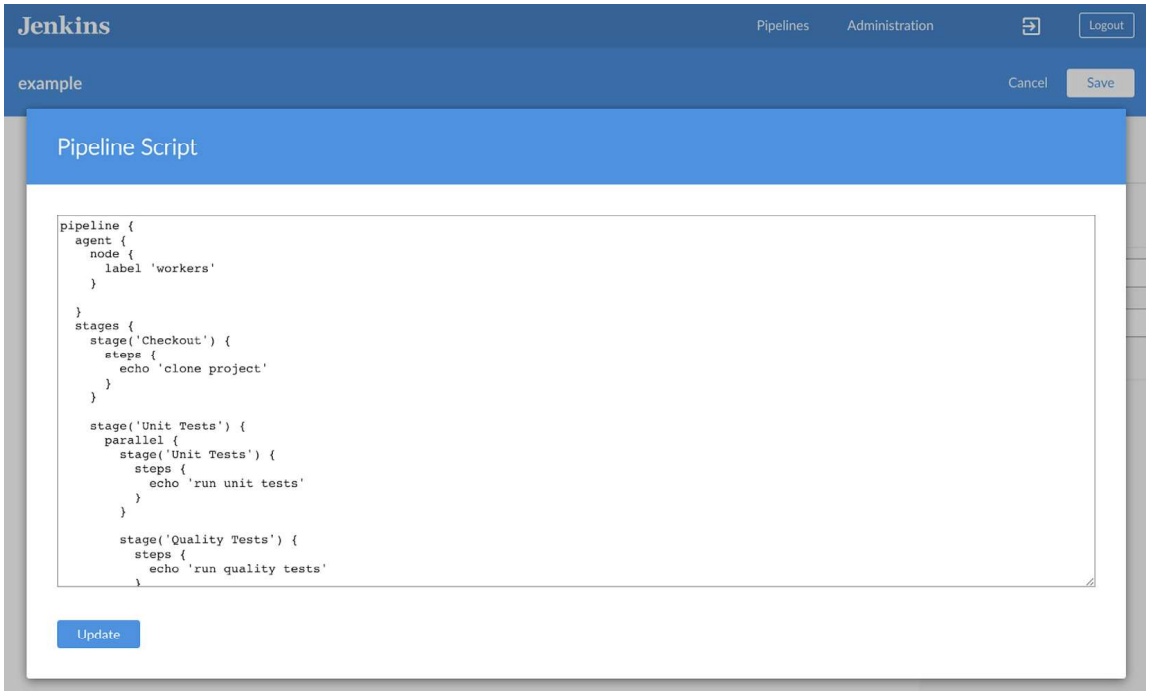

Figure 2.6 Jenkinsfile generated from the pipeline editor

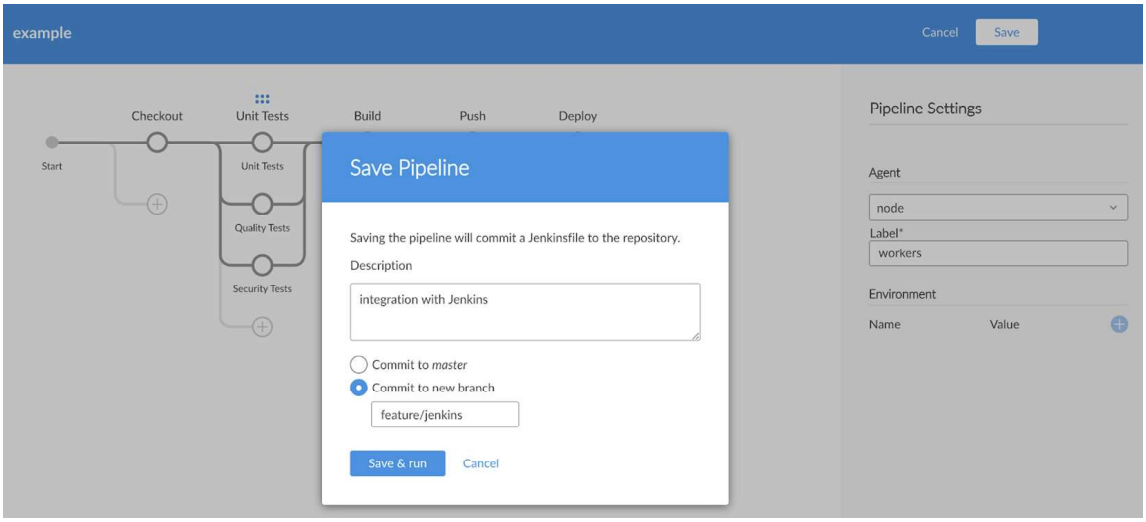

Figure 2.7 Committing the Jenkinsfile to the Git repository

Once the file is committed, the pipeline will be triggered, and the stages defined in the pipeline will be executed.

Keep in mind that Blue Ocean doesn't support all features of Jenkins such as administration, nodes management, or credential settings. However, you can always switch back to the classic Jenkins UI by clicking the exit icon at the top right of the Blue Ocean navigation bar.

> **NOTE** This is just a sneak peek of Blue Ocean's main features. In chapter 7, we will dig deeper into each feature.

Now that you're familiar with how a Jenkinsfile works, let's see how to write your own pipeline as code with Jenkins. Jenkins 2 allows two styles of structure and syntax for building out workflows. These are referred to as scripted and declarative pipelines.

2.1.2 Scripted pipeline

A *scripted pipeline* is a traditional way of writing pipeline code. In this pipeline, the Jenkinsfile is written on the Jenkins UI instance. The pipeline steps are wrapped in a node block (denoted by the opening and closing braces). Here, a node refers to a *Jenkins agent* (formerly referred to as a *slave instance*).

The node gets mapped to the Jenkins cluster by using a label. A *label* is simply an identifier that has been added when configuring the node in Jenkins via the Manage Nodes section, as shown in figure 2.8.

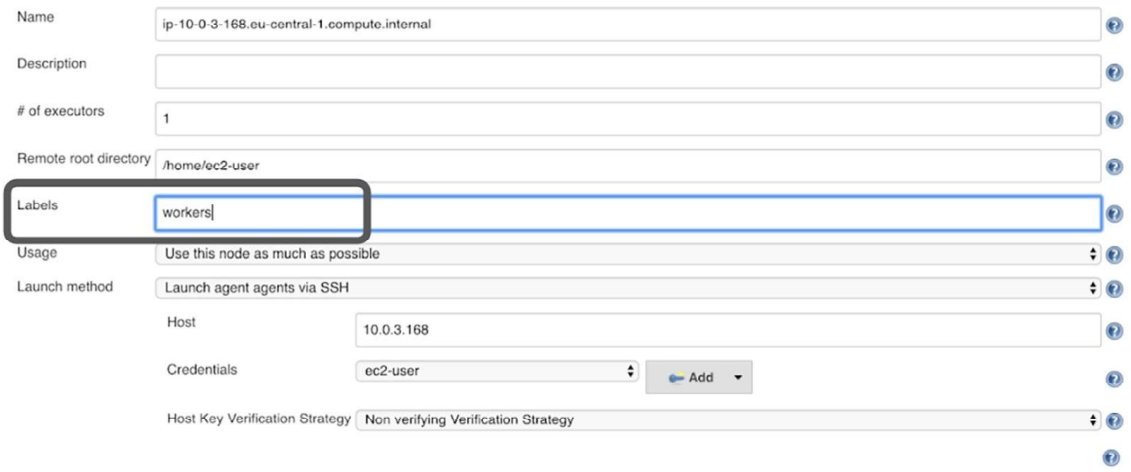

Figure 2.8 Assigning labels to Jenkins workers

> **NOTE** The next chapter covers how the Jenkins distributed mode works and how node agents can be used to offload work from Jenkins.

The steps inside the node block can include and make use of any valid Groovy code. The pipeline can be defined by creating a new pipeline project and typing the code in the Pipeline Editor section, as shown in figure 2.9.

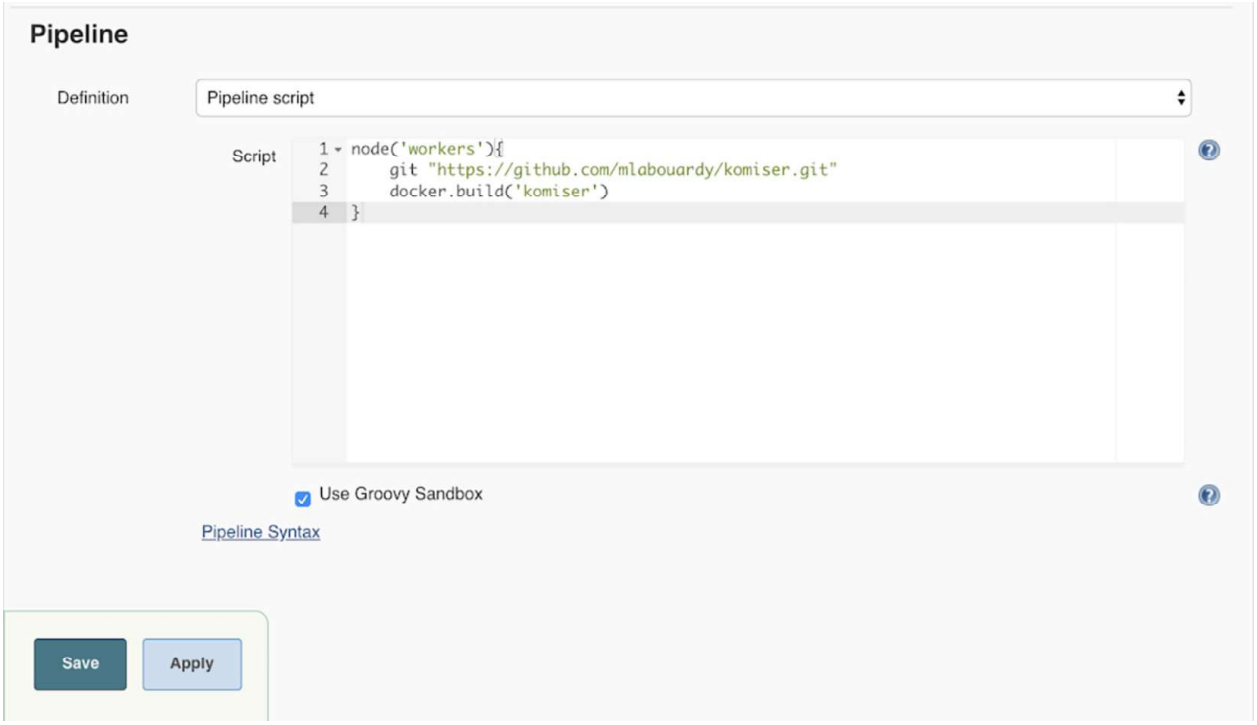

Figure 2.9 Using an inline Jenkinsfile with Pipeline scripts

Although this simple node block is technically valid syntax, Jenkins pipelines generally have a further level of granularity—stages. A *stage* is a way to divide the pipeline into logical functional units. It also serves to group steps and Groovy code together to create targeted functionality. Figure 2.10 shows an example of the preceding pipeline using stages.

Figure 2.10 Using the `stage` keyword to define logical units

The pipeline has two stages:

- *Checkout*—For cloning the project GitHub repository
- *Build*—For building the project Docker image

How much of the pipeline's logic goes into a particular stage is up to the developer. However, the general practice is to create stages that mimic the separate pieces of a traditional pipeline.

The scripted pipeline uses stricter Groovy-based syntaxes because it was the first pipeline to be built on the Groovy foundation. Since this Groovy script was not typically desirable to all users, the declarative pipeline was introduced to offer a simpler and more optioned Groovy syntax.

NOTE Chapter 14 covers how to write a shared Jenkins library with custom Groovy scripts for code modularity.

2.1.3 Declarative pipeline

A *declarative pipeline* is a relatively new feature (introduced in Pipeline 2.5, https://plugins.jenkins.io/workflow-aggregator) that supports the PaC approach. It makes the pipeline code easier to read and write for new Jenkins users.

This code is written in a Jenkinsfile that can be checked into a version-control system (VCS) such as SVN or an SCM system such as GitHub, GitLab, Bitbucket, or others. Figure 2.11 is an example of a Jenkinsfile located at the root folder of a GitHub repository.

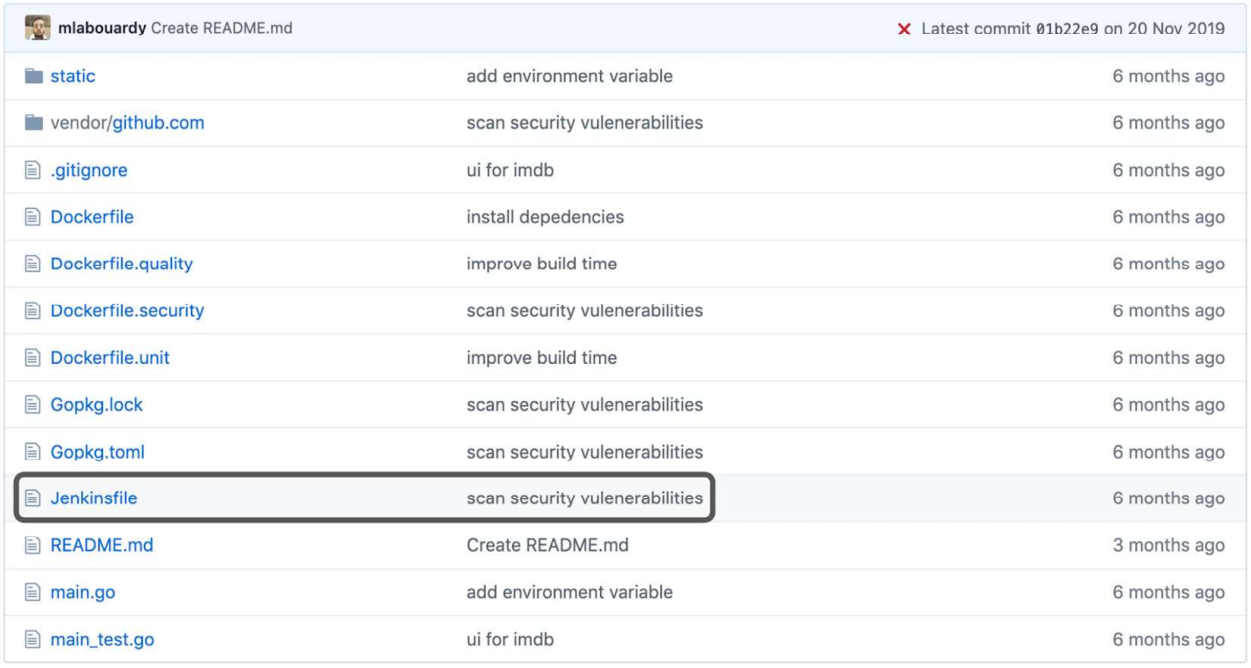

Figure 2.11 A Jenkinsfile stored in a source-control repository

In declarative syntax, you cannot use Groovy code such as variables, loops, or conditions. You are restricted to the structured sections/blocks and the DSL (Jenkins domain-specific language) steps.

Figure 2.12 shows the differences between scripted and declarative pipelines. Declarative pipelines are restricted and have well-defined structures (for example, all DSL statements must be enclosed in a `steps` directive).

Scripted

```
node('node label'){
stage('id #1'){
//DSL statements
}
stage('id #2'){
//DSL statements
}

// OR

//DSL statements without stage block

// OR

//Loops, conditions, variables, etc
def variable = value

if(variable){
//DSL statements
}else{
//DSL statements
}

def list = []
for(int i=0;i<list.size();i++){
//DSL statements
}
}
```

Declarative

```
pipeline{
agent{
label 'node label'
}

environment{
ENV_VARIABLE_A = 'v
}

stages{
stage('id #1'){
agent{
label 'node label 1
}
environment{}
steps{
//DSL statements
}
}
stage('id #2'){
agent{
label 'node label 2
}
environment{}
steps{
//DSL statements
}
}
}

post {
always {
//DSL statements
}
success {
//DSL statements
}
failure {
//DSL statements
}
}
}
```

Figure 2.12 Differences between scripted and declarative pipelines

Declarative pipelines provide a more restrictive syntax, as each pipeline must use these predefined block attributes or sections:

- `agent`
- `environment`
- `post`
- `stages`
- `steps`

The `agent` section defines the worker or machine where the pipeline will be executed. This section must be defined at the top level inside the pipeline block or overridden at the stage level. The agent can be any of the following:

- Jenkins worker or node (refer to chapter 3 for distributed builds on Jenkins)
- Docker container based on a Docker image or a custom Dockerfile (covered in chapter 9)
- Pod deployed on a Kubernetes cluster (covered in chapter 14)

For example, you can define the pipeline to run on a custom Docker container, as shown in the following listing.

Listing 2.2 Declarative pipeline agents definition

```
pipeline{
    agent {
        node {
            label 'workers'
        }

        dockerfile {
            filename 'Dockerfile'
            label 'workers'
        }

        kubernetes {
            label 'workers'
            yaml """
            kind: Pod
            metadata:
            name: jenkins-worker
            spec:
            containers:
            - name: nodejs
              image: node:lts
              tty: true
            """
        }
    }
}
```

NOTE Refer to the official documentation for more information about the agent syntax: www.jenkins.io/doc/book/pipeline/syntax/.

The `environment` section contains a set of environment variables needed to run the pipeline steps. The variables can be defined as sequences of key-value pairs. These will be available for all steps if the environment block is defined at the pipeline top level; otherwise, the variables can be stage-specific. You can also reference credential variables by using a helper method `credentials()`, which takes as a parameter the ID of the target credential, as shown in the following listing.

Listing 2.3 Environment variables definition

```
pipeline{
    environment {
        REGISTRY_CREDENTIALS= credentials('DOCKER_REGISTRY')
        REGISTRY_URL = 'https://registry.domain.com'
    }

    stages {
        stage('Push'){
            steps{
                sh 'docker login $REGISTRY_URL --username
 $REGISTRY_CREDENTIALS_USR --password $REGISTRY_CREDENTIALS_PSW'
            }
        }
    }
}
```

The Docker registry username and password are accessible automatically by referencing the `REGISTRY_CREDENTIALS_USR` and `REGISTRY_CREDENTIALS_PSW` environment variables. Those credentials are then passed to the `docker login` command to authenticate with the Docker Registry before pushing a Docker image.

The `post` section contains commands or scripts that will be run upon the completion of a pipeline or stage run, depending on the location of this section within the pipeline. However, conventionally the `post` section should be placed at the end of the pipeline. Examples of commands that can be used within the `post` section are those that provide Slack notifications, clean up the job workspace, and execute post-scripts based on the build status. The pipeline build status can be fetched by using either the `currentBuild.result` variable or the post-condition blocks `always`, `success`, `unstable`, `failure`, and so forth.

The following listing is an example Slack notification. The instructions wrapped by the `always` directive will run no matter the status of the build and will not interfere with the final status.

Listing 2.4 Post build actions in a declarative pipeline

```
pipeline{
    post {
        always {
            echo 'Cleaning up workspace'
        }
```

```
        success {
            slackSend (color: 'GREEN', message: \
                "${env.JOB_NAME} Successful build")
        }
        failure {
           slackSend (color: 'RED', message: "${env.JOB_NAME} Failed build")
        }
    }
}
```

This code references the `env.JOB_NAME` variable, which contains the Jenkins job name.

> **NOTE** Chapter 10 has a dedicated section on how to implement Slack notifications with Jenkins.

The `stages` section is the core of the pipeline. This section defines what is to be done at a high level. It contains a sequence of more stage directives for each discrete part of the CI/CD workflow.

Finally, the `steps` section contains a series of more steps to be executed in a given stage directive. The following listing defines a `Test` stage with instructions to run unit tests and generate code coverage reports.

Listing 2.5 Running automated tests within a pipeline

```
pipeline{
    agent any
    stages {
        stage('Test'){
            steps {
                sh 'npm run test'
                sh 'npm run coverage'
            }
        }
    }
}
```

These are the most used directives and sections while writing a declarative pipeline. Additional directives will be covered throughout this book. For an overview of all available blocks, refer to Pipeline Syntax documentation (www.jenkins.io/doc/book/pipeline/syntax/#stages).

Both declarative and scripted styles can be used to build CI/CD pipelines in either the web UI or with a Jenkinsfile. However, it's generally considered a best practice to create a Jenkinsfile and check it into the source-control repository to have a single source of truth and be able to track all changes (auditing) that your pipeline went through.

> **NOTE** In chapters 7 through 11, you will learn how to write a scripted pipeline from scratch for various application architectures and how to convert a Jenkinsfile from a scripted to a declarative format.

2.2 Understanding multibranch pipelines

When you're building your application, you must separate your deployment environments to test new changes without impacting your production. Therefore, having multiple environments for your application makes sense. To be able to achieve that, you need to structure your code repository to use multiple branches, with each branch representing an environment. For instance, the master branch corresponds to the current production code.

While it's easier nowadays to replicate multiple infrastructure environments with the adoption of cloud computing and IaC tools, you still need to configure your CI tools for each target branch.

Fortunately, when using a Jenkinsfile, your pipeline definition lives with the code source of the application going through the pipeline. Jenkins will automatically scan through each branch in the application code repository and check whether the branch has a Jenkinsfile. If it does, Jenkins will automatically create and configure a subproject within the multibranch pipeline project to run the pipeline for that branch. This eliminates the need for manual pipeline creation and management.

Figure 2.13 shows the jobs in a multibranch pipeline project after executing against the Jenkinsfiles and source repositories. Jenkins automatically scans the designated repository and creates appropriate projects for each branch in the repository that contains a Jenkinsfile.

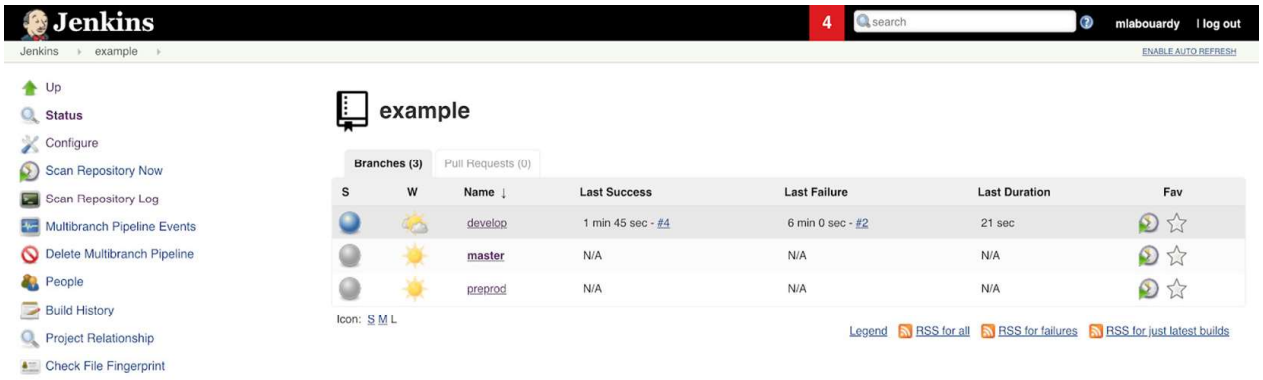

Figure 2.13 Jenkins automatically creates a job for each branch with a Jenkinsfile.

In figure 2.13, Jenkins will trigger a build whenever a new code change occurs on any of the develop, preprod, or master branches. In addition, each branch might have different pipeline stages. For example, you might perform a complete CI/CD pipeline for the master branch and only a CI pipeline for the develop branch (see figure 2.14). You can do this with the help of a multibranch pipeline project.

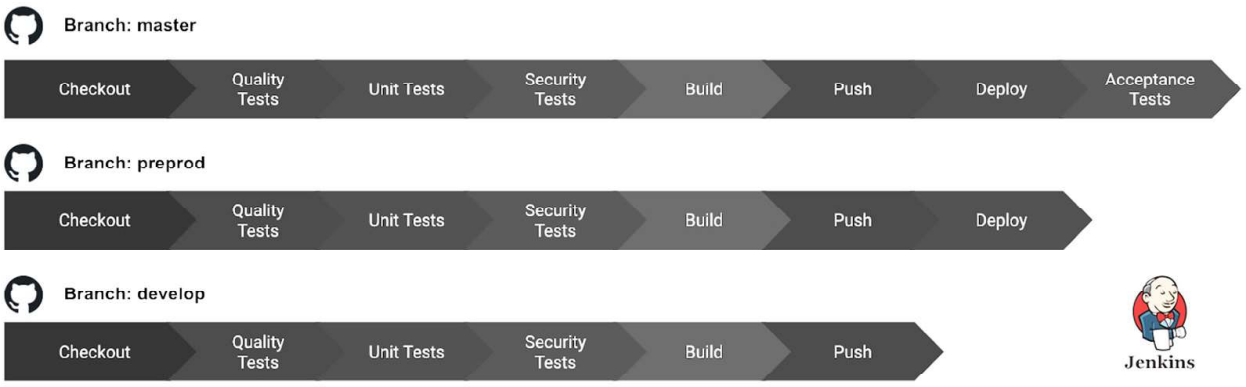

Figure 2.14 Each Git branch can have its own Jenkinsfile stages.

A multibranch pipeline can also be used to validate pull requests before merging them to target branches. You can configure Jenkins to launch pre-integration tests against the application's code and block the pull request merge if the tests failed, as in figure 2.15.

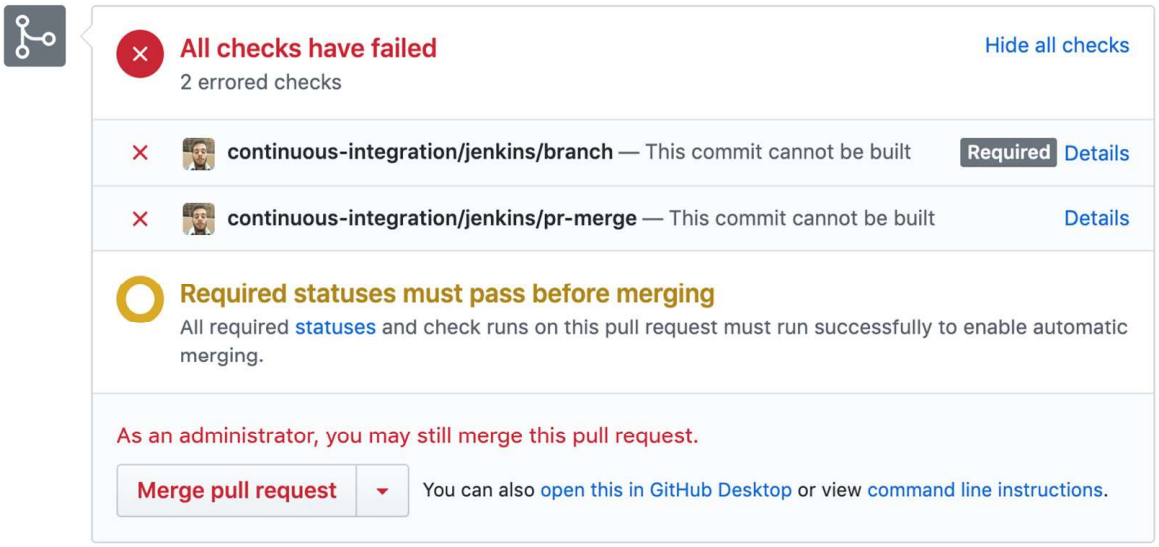

Figure 2.15 Jenkins build status in GitHub pull request

NOTE Chapter 9 covers using multibranch pipelines to validate pull/change requests.

Now that you're familiar with the basics of the Jenkins multibranch pipeline, you must follow Git branching guidelines to have a common vision and methodology within the development team. So which Git branching strategies should you use for your development cycle?

2.3 Exploring the GitFlow branch model

A couple of Git branching strategies exist. The most interesting and used one is GitFlow. It consists of the following essential branches:

- *Master*—A branch that corresponds to the current production code. You can't commit directly except for hotfixes. Git tags can be used to tag all commits in the master branch with a version number (for instance, you can use the semantic versioning convention detailed at https://semver.org/).
- *Preprod*—A release branch, a mirror of production. It can be used to test all new features developed on the develop branch before merging them to the master branch.
- *Develop*—A development integration branch containing the latest integrated development code.
- *Feature/X*—An individual feature branch being developed. Each new feature resides in its own branch, and it's generally created from the latest develop branch.
- *Hotfix/X*—When you need to solve something in production code, you can use the hotfix branch and open a pull request for the master branch. This branch is based on the master branch.

NOTE A complete example demonstrating the use of GitFlow with the Jenkins multibranch pipeline project is given in chapters 7 through 11.

The overall flow of GitFlow within Jenkins can be summarized as follows:

- A develop branch is created from the master branch.
- A preprod branch is created from the develop branch.
- A developer creates a new feature branch based on the development branch. When a feature is completed, a pull request is created.
- Jenkins automatically runs pre-integration tests in this individual feature. If the tests are successful, Jenkins marks the commits as successful. The development team will then review the changes and merge the pull request of the new feature branch to the develop branch and delete the feature branch.
- A build will be triggered on the develop branch, and the changes will be deployed to the sandbox/development environment.
- A pull request is created to merge the develop branch into the preprod branch.
- When the develop branch is merged to the preprod branch, the pipeline will be triggered to deploy the new features to the staging environment upon the completion of the pipeline.
- Once the release is being validated, the preprod branch will be merged to master, and changes will be deployed to the production environment after user approval.
- If an issue in production is detected, a hot branch is created from the master branch. Once the hotfix is complete, it will be merged to both the develop and master branches.

NOTE You can use the GitFlow wrapper around the Git command line (available on multiple operating systems) to create a project blueprint with all needed branches.

Figure 2.16 summarizes how GitFlow works.

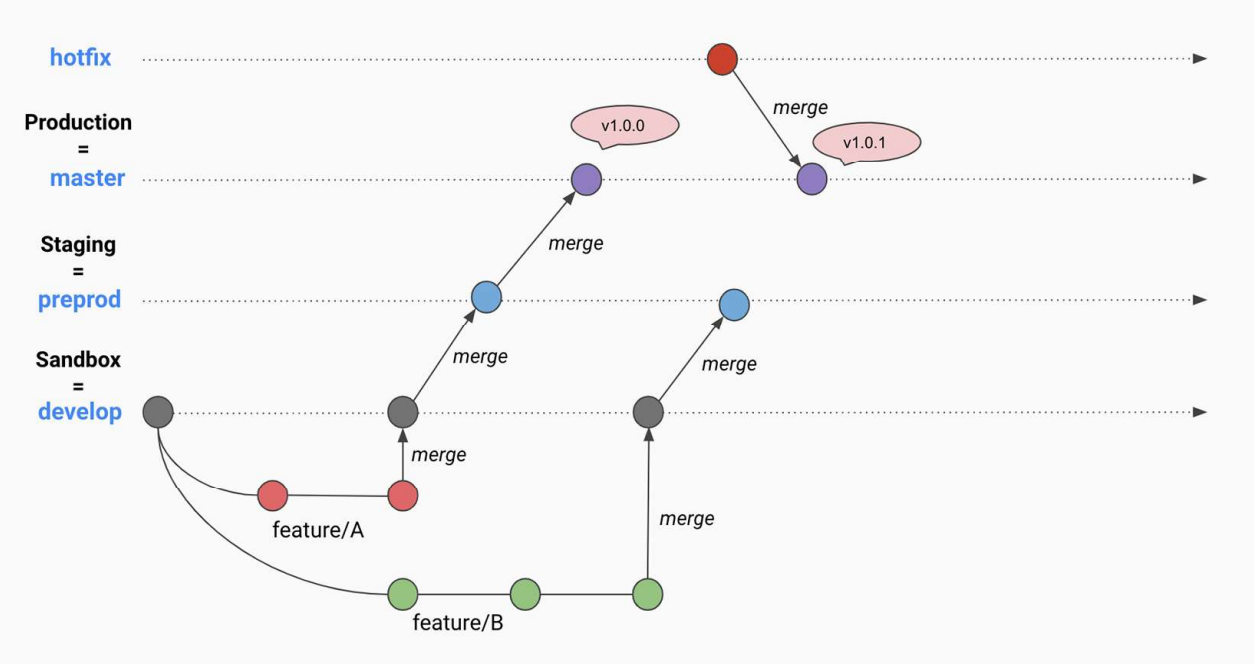

Figure 2.16 Overview of GitFlow branches

GitFlow does not solve all problems with branching. But it offers you a more logical branch structure and a great workflow organization model when working within a big team. In addition, many feature branches are developed concurrently, which makes parallel development easy. For smaller projects (and smaller teams), GitFlow can be overkill. Hence, in upcoming chapters, we will usually use three main branches:

- *Master* branch, to store the official release history and the source code of an application running in a production environment
- *Preprod* branch, to store new integrated features running in the staging environment and ready to be merged to the master branch
- *Develop* branch, for the latest delivered development changes and mirror of the application running in a sandbox environment

2.4 Test-driven development with Jenkins

Using Jenkinsfiles has one potential downside: it can be more challenging to discover problems up-front when you are working in the external file and not in the environment of the Jenkins server. One approach to dealing with this is developing the code

within the Jenkins server as a pipeline project first. Then, you can convert it to a Jenkinsfile afterward.

You can also use Blue Ocean mode as a playground, as seen earlier in this chapter, to set up a Jenkinsfile from scratch with a modern and intuitive pipeline editor. Another approach to test a new pipeline is a declarative pipeline linter application that you can run against Jenkinsfiles, outside Jenkins, to detect problems early.

2.4.1 The Jenkins Replay button

Sometimes, when working on Jenkins jobs, you might find yourself stuck in this cycle of committing the Jenkinsfile, pushing it, and running the job over and over again. It can be a time-consuming and tedious workflow, especially if your build time is inherently long. Plus, your Git history will get filled with junk commits (unnecessary debugging commits).

What if you could work on your Jenkinsfile in a "sandbox" and test the Jenkinsfile live on the system? A neat little feature allows you to modify the Jenkins file and rerun the job. You can do it over and over until you are happy with the results and then commit the working Jenkinsfile without breaking anything.

Now, this is a little easier. If you have a Pipeline build that did not proceed exactly as you expected, you can use the Replay button in the build's sidebar, shown in figure 2.17.

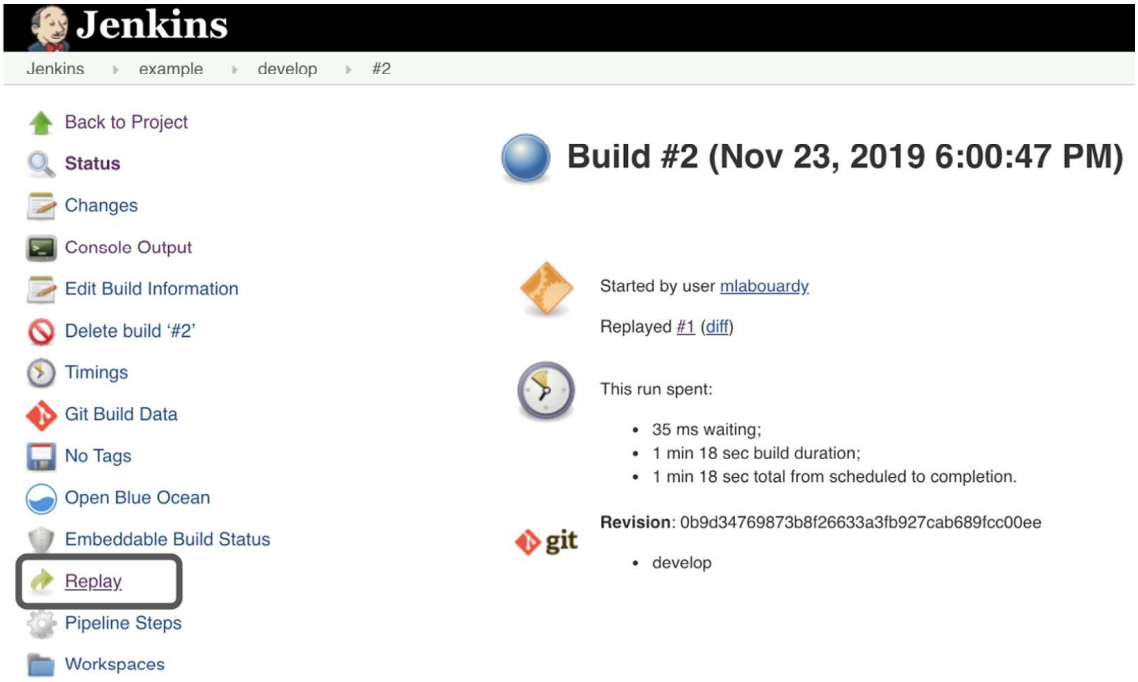

Figure 2.17 Rerunning the build with a Replay button

It is somewhat similar to the Rebuild button but allows you to edit the Jenkinsfile content just before running the job. Therefore, you can use the built-in Jenkinsfile block in the UI (figure 2.18), to test your pipelines out there before committing them to source control like GitHub.

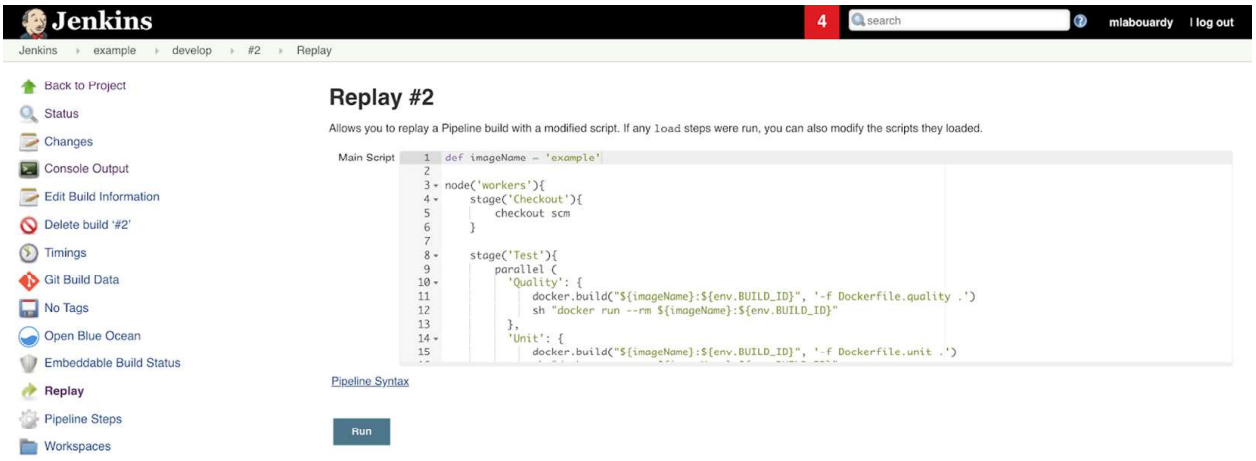

Figure 2.18 Updating the Jenkinsfile before replaying the pipeline

You can change your pipeline's code and click the Run button and rerun the job. Once you are satisfied with the changes, you update the Jenkinsfile with the applied changes and commit them to your SCM.

The Replay button feature allows for quick modifications and execution of an existing pipeline without changing the pipeline configuration or creating a new commit. It's ideal for rapid iteration and prototyping of a pipeline.

2.4.2 Command-line pipeline linter

For advanced users, you can use the Jenkins RESTful API to validate the Jenkinsfile syntax by issuing an HTTP/HTTPS POST request with the parameters shown in figure 2.19.

> **NOTE** To get the API endpoint working on a Jenkins server with cross-site request forgery (CSRF) protection enabled, you need to request a crumb issuer and include it in the Authorization header in the issued HTTP request. To generate this crumb, you need to request the following URL: JENKINS_URL/jenkins/crumbIssuer/api/json.

Figure 2.19 is an example of how to use the Jenkins Linter API to validate Jenkinsfile syntax. We're using Postman in this example, and the Jenkinsfile form data has been loaded from the developer machine.

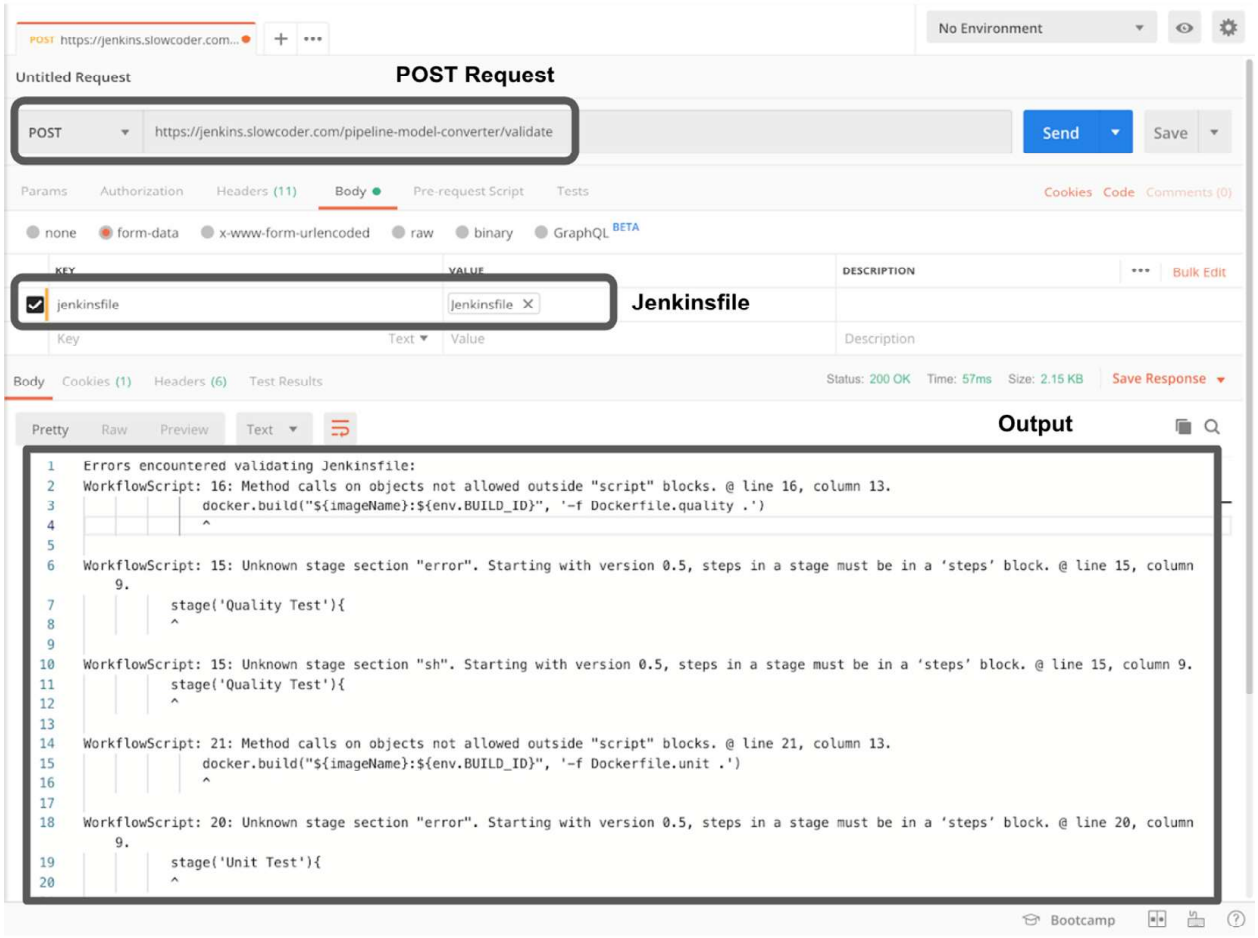

Figure 2.19 Example of using Jenkins Linter API

The API response will return both errors and warnings, which can save time during the development and allows you to follow best practices while writing a Jenkinsfile.

Specifying the real password is still supported, but it is not recommended because of the risk of revealing the password, and the human tendency to reuse the same password in different places. Another way of validating the Jenkinsfile is to run the following command from the terminal session (cURL is available for most operating systems):

```
curl -X POST -L --user USERNAME:TOKEN JENKINS_URL/pipeline-model-converter/
     validate
-F "jenkinsfile=<Jenkinsfile"
```

NOTE Chapter 7 covers another way of creating a Jenkins API token from the Jenkins web dashboard.

The Jenkins command-line interface (CLI), www.jenkins.io/doc/book/managing/cli/, can also be used with the `declarative-lint` option to lint a declarative pipeline from the command line before actually running it. You can issue this command to lint a Jenkinsfile via the CLI with SSH:

```
ssh -p $JENKINS_SSHD_PORT $JENKINS_HOSTNAME declarative-linter < Jenkinsfile
```

Replace the `JENKINS_HOSTNAME` and `JENKINS_SSHD_PORT` variables based on the URL and port where you are running Jenkins. You can also use localhost as a URL if you are running Jenkins on your machine.

2.4.3 IDE integrations

The Jenkins CLI or API does a great job of reducing the turnaround times when writing a Jenkinsfile, but its usage has its own inconveniences. You need tools like SSH to make a connection to your Jenkins server, and you need to remember the correct command to validate your Jenkinsfile.

Fortunately, you can install extensions on your favorite integrated development environment (IDE) to automate the validation process. For instance, on Visual Studio Code (VSCode), you can install Jenkins Validation Linter from the marketplace. This extension, shown in figure 2.20, validates Jenkinsfiles by sending them to the Pipeline Linter endpoint of a Jenkins server.

> **NOTE** Similar extensions and packages are available to validate a Jenkinsfile for Eclipse, Atom, and Sublime Text.

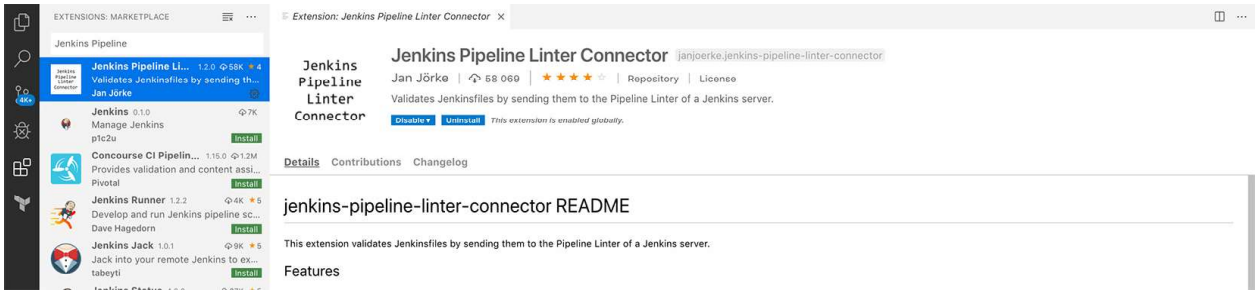

Figure 2.20 Jenkins Pipeline Linter extension for VSCode

Once the extension is installed, you must provide Jenkins server settings, including the server URL (with the following format: JENKINS_URL/pipeline_model_converter/validate) and credentials (Jenkins username and password, or token if CSRF protection is enabled) by clicking Preferences from the top navigation bar, and selecting Settings, as shown in figure 2.21.

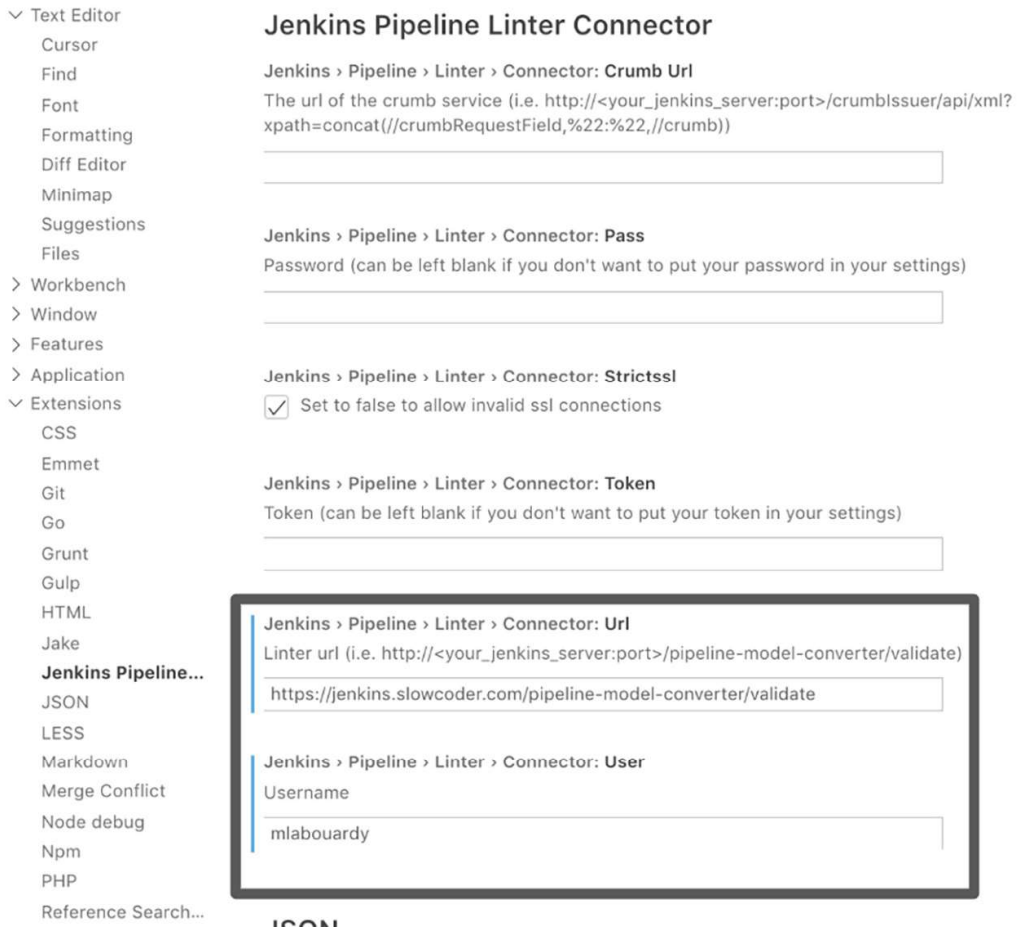

Figure 2.21 Jenkins Pipeline Linter configuration

Once settings are configured, you can type the `Validate Jenkinsfile` command on the command palette search bar (keyword shortcut ⇧⌘P), as shown in figure 2.22.

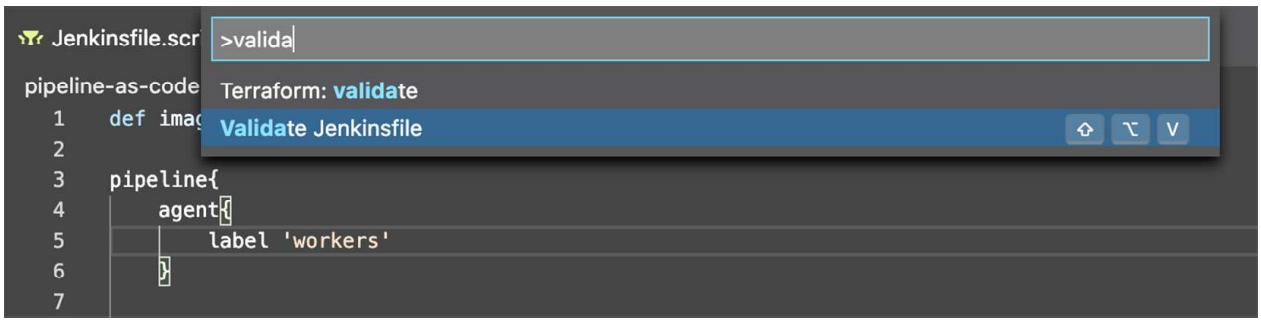

Figure 2.22 VSCode command palette

The linter will report the pipeline validation results in the terminal, as shown in figure 2.23.

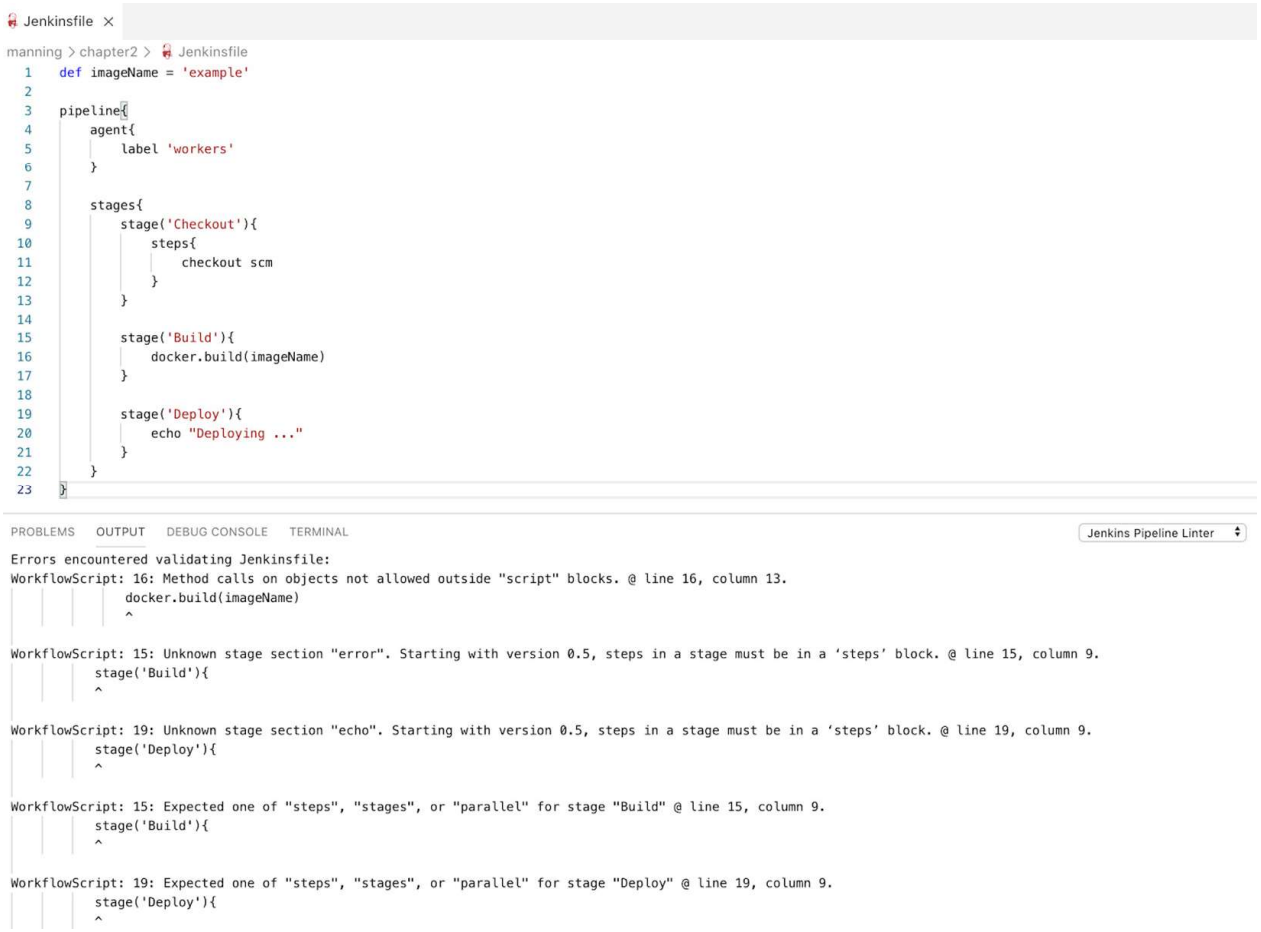

Figure 2.23 Example of Jenkins Linter's output

NOTE In chapter 8, you will learn how to write unit tests for CI pipelines and use the Jenkins Pipeline Unit (https://github.com/jenkinsci/JenkinsPipelineUnit) testing framework to mock the pipeline executor locally.

Summary

- Infrastructure as code influenced CI/CD tools to embrace the pipeline-as-code concepts.
- A Jenkinsfile uses Groovy syntax and utilizes shared Jenkins libraries to customize a CI/CD workflow.
- Declarative pipelines encourage a declarative programming model. Scripted pipelines follow a more imperative programming model.

- The Blue Ocean editor can facilitate a quick and easy setup of a new Jenkins pipeline with minimal hassle.
- A feature branch workflow facilitates pull requests and more efficient collaboration.
- GitFlow offers a dedicated channel for hotfixes to production without interrupting the rest of the workflow or waiting for the next release cycle.
- The Jenkins UI, Replay button, and code linters can be used to test new pipelines before committing them to source control, enabling you to avoid a bunch of unnecessary debugging commits.