Beginning Jenkins Blue Ocean

Create Elegant Pipelines With Ease

Nikhil Pathania

Apress®

Beginning Jenkins Blue Ocean

Create Elegant Pipelines With Ease

Nikhil Pathania

Apress®

Beginning Jenkins Blue Ocean

Nikhil Pathania
Brande, Denmark

ISBN-13 (pbk): 978-1-4842-4157-8 ISBN-13 (electronic): 978-1-4842-4158-5
https://doi.org/10.1007/978-1-4842-4158-5

Library of Congress Control Number: 2018965935

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Nikhil Karkal
Development Editor: Matthew Moodie
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4157-8. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dedicated to the open-source community

Table of Contents

About the Author

Nikhil Pathania is currently practicing DevOps at Siemens Gamesa Renewable Energy in Denmark. He has been working in the DevOps industry for over 10 years. Nikhil started his career in software configuration management as an SCM engineer and later moved on to learn various other tools and technologies in the field of automation and DevOps.

During his career, Nikhil has architected and implemented continuous integration and continuous delivery solutions across diverse IT projects. He enjoys finding new and better ways to automate and improve manual processes.

In his spare time, Nikhil likes to read, write, and meditate. He is an avid climber and now hikes and cycles.

You can reach Nikhil on Twitter at @otrekpiko.

About the Technical Reviewer

Nitesh Agarwal is working as an SDE II with CodeNation building standardized CI/CD pipelines using the gitOps principles. He has also worked in setting up terabyte-scale ETL pipelines in the past. Nitesh has interests in the field of image processing, machine learning, and distributed systems. In his free time he likes playing guitar and watching movies & documentaries.

Acknowledgments

First and foremost, I would like to thank my beautiful wife Karishma for supporting and helping me write yet another book on Jenkins. I would like to thank Nikhil Karkal for bringing me this opportunity. And I give great thanks to Nitesh Agarwal, who provided me with valuable feedback and suggestions throughout the writing process.

A special thanks to Divya Modi, Matthew Moodie, and the entire Apress publishing team that worked hard to make this book the best possible experience for the readers.

I would also like to thank all the amazing readers for giving us their valuable feedback through Twitter, Amazon, Goodreads, Apress, and Springer websites. I read them seriously, and they help me understand you, the readers, better. So please keep giving me feedback.

And finally, a great thanks to the Jenkins Blue Ocean team and the Jenkins community for creating such fantastic software.

Introduction

Jenkins is the de facto tool to implement Continuous Delivery and the first choice among users. It is open source and has an extensive library of plugins, which makes it lucrative among its contenders. Moreover, recently with the addition of the "Pipeline as Code" feature to Jenkins combined with the plugins that support "Pipeline as Code," it has become possible to create complex Continuous Delivery pipelines of any degree.

However, the things that make Jenkins extensible also make it a sophisticated tool to learn; as a result, over the years Jenkins has gained a reputation of being a comprehensive yet difficult Continuous Delivery tool to master.

To ease the learning curve, Jenkins has come up with its Blue Ocean flavor. Jenkins Blue Ocean intends to make the process of creating Continuous Delivery pipelines easy and fun, with its intuitive user interface and its powerful Pipeline editor. As a result, Jenkins Blue Ocean has become an attractive entry point for users who wish to join the Continuous Delivery arena.

The topics discussed in this book will take the readers through all the features embedded in Jenkins Blue Ocean.

CHAPTER 1

Introduction

Before diving deep into Jenkins Blue Ocean, an introductory overview about it would be of great use. Blue Ocean is not just a renovation of the classic Jenkins UI—it's also a change in the way you compose, describe, and visualize pipelines.

In this introductory chapter, you'll learn the following:

- What is Jenkins Blue Ocean?
- What does it have to offer?
- Things to consider

Along with the introduction, it is equally important to set the expectations right. Therefore, I have included a section about the essential things to consider before you decide to move to Jenkins Blue Ocean.

At the end of this chapter, you'll get a clear picture of Jenkins Blue Ocean's capabilities. Remember the current chapter is an introduction to Jenkins Blue Ocean. The upcoming chapters expound on the topics discussed here. So, let us begin.

What Is Jenkins Blue Ocean?

Jenkins Blue Ocean, or *Blue Ocean*, is more than just a face-lift to the *Classic Jenkins*.[1] The following section answers why there is a Jenkins Blue Ocean in the first place and its purpose.

[1]The standard Jenkins.

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5_1

A Response to Stimuli

The need for an improved software delivery process has introduced various continuous practices, such as *Continuous Integration* and *Continuous Delivery*, to name a few.

These continuous practices are creating a wave of DevOps tools. As a result, a typical developer deals with at least two to five DevOps tools in everything he does.

As a consequence, demand for improved usability and user experience is inevitable. Jenkins, being a widely used CD tool, is no exception to these demands.

Jenkins is extensible and robust yet criticized for its poor user experience. An attempt to improve Jenkins's usability is visible in Jenkins version 2.0 and afterward, such as the Jenkins Setup Wizard, the well-ordered configurations inside a Jenkins Job using tabs, and more recently the new login page, to name a few. However, these few enhancements do not fill the much more significant void in Jenkins concerning the user experience.

That is why we have Jenkins Blue Ocean. Blue Ocean is altogether a more generous attempt to address the user experience issue. Beauty with brains, I must say.

Competition from the other CI/CD tools is also responsible for the conception of Jenkins Blue Ocean. Tools such as *Circle CI*, *GitLab CI*—to name a few—already had the feature to visualize pipelines.

However, Jenkins Blue Ocean, although late in the competition, has the edge over the others for introducing the *Visual Pipeline Editor*—a UI tool to simultaneously visualize and create pipelines. You'll see more of it in the upcoming sections and chapters.

A Continuous Delivery Tool for Everyone

"Continuous Delivery is no longer for the experts," claims Jenkins Blue Ocean. Let's look at the key things that give Blue Ocean an edge over the others and make it a tool for the masses.

- Jenkins Blue Ocean comes with an embedded tool called the Visual Pipeline Editor. This tool allows users to create and edit their pipeline visually using a UI interface that's accessible directly from the pipeline dashboard. Also, the Visual Pipeline Editor saves your pipeline in code inside a file (Jenkinsfile), following the Declarative Pipeline Syntax, directly to your source code repository.
- The pipeline visualization allows users to diagnose pipeline failures with ease and speed. When a pipeline fails, Blue Ocean tells you exactly where it has failed by pointing out the failed step. Also, the pipeline logs are displayed individually for every stage and step of a pipeline, so that users do not end up scrolling through a single huge log.
- Blue Ocean also creates separate pages to view your testing results and built artifacts for every pipeline run.
- The pipeline creation process in Jenkins Blue Ocean is more like a wizard. It also comes with a UI tool to create and edit pipelines. Above all, it has an interactive pipeline visualization, which makes your pipeline easy to understand.
- Those who would like to create a more complex Jenkins pipeline while staying within Jenkins Blue Ocean's framework can seek help from *script* and *Jenkins Shared Libraries*.

All these features make Jenkins Blue Ocean a continuous delivery tool for people of all skill levels. Moreover, the pipeline creation experience in Blue Ocean gets even better if the following conditions are taken care of beforehand:

- Your source code is on Git/GitHub/GitLab/BitBucket.
- Your build/test agents are set up and connected to the Jenkins Master. If you plan to spawn them on Docker, the required *Docker images*[2] and *Dockerfiles*[3] are ready.
- Your Continuous Delivery Pipeline is straightforward and free from legacy technologies.[4]
- Your Jenkins Administrator has set up the supporting DevOps tools and Jenkins Plugins required by your CD pipeline.

A Jenkins Plugin

Blue Ocean is not a standalone tool; it comes as a plugin for Jenkins. You can install Blue Ocean in the following two ways:

- As a suite of plugins on an existing Jenkins installation
- As part of Jenkins in Docker

Installing Jenkins Blue Ocean is easy for both new and existing Jenkins users. The Blue Ocean plugin works with Jenkins 2.7.x or later. Besides, there are important things to consider before using Blue Ocean if you run Jenkins behind a reverse proxy server such as Apache or Nginx.

In the upcoming chapter, you will learn to set up Blue Ocean in ways described earlier.

---

[2]A combination of a file system and parameters

[3]A text file used to define and build docker images

[4]Software or hardware that has been superseded but is difficult to replace because of its wide use

What Blue Ocean Offers?

The classic Jenkins is infamous for not being user-friendly. Jenkins Blue Ocean, on the other hand, is designed to provide the best possible user experience. Let's see what Jenkins Blue Ocean has to offer.

Pipeline Creation Wizard

Creating a pipeline using Blue Ocean is much more relaxed in many ways. Thanks to the Blue Ocean engineers, the pipeline creation process is now a simple wizard.

If you search for the term *Pipeline Creation Wizard*, you are going to find nothing. It's a name that I find appropriate for an improvement in Jenkins Blue Ocean—related to the way you create pipelines in Jenkins. Let's look at what I mean.

On the Jenkins Blue Ocean dashboard, you see an option named *New Pipeline*. See Figure 1-1.

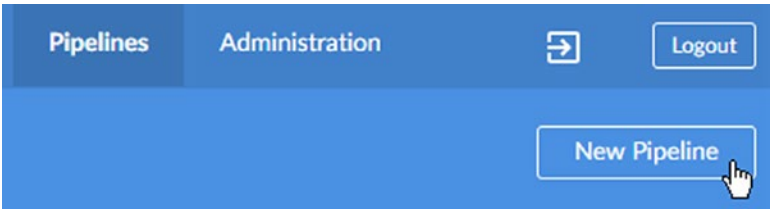

Figure 1-1. *The Pipeline creation link to initiates the Pipeline Creation Wizard*

Clicking on *New Pipeline* takes you through a Pipeline Creation Wizard. In the back end, Jenkins creates a multibranch pipeline. At first, the wizard asks you to choose the type of source code repository you use (see Figure 1-2).

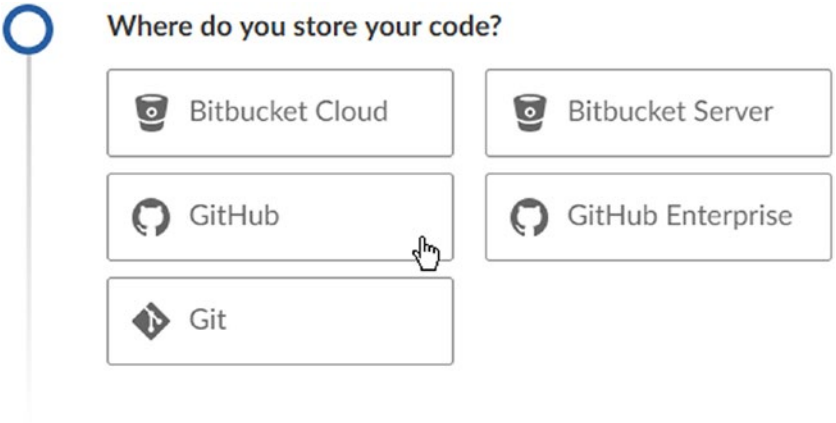

Figure 1-2. *Git-based repository managers that integrate with Blue Ocean*

When you make an appropriate selection, it asks you for authentication (see Figure 1-3). In the back end, a credential of the type *GitHub Access Token* gets created inside *Jenkins Credentials*.

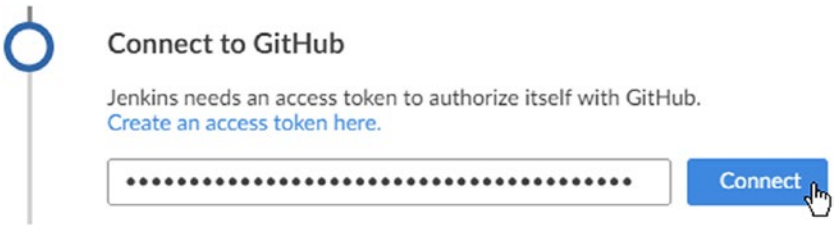

Figure 1-3. *Connecting to GitHub using an access token*

After authenticating yourself, the wizard asks you a few more questions. See Figures 1-4 and 1-5. These questions depend on the type of source code repository you choose in the first step. In the back end, Jenkins is configuring the *Source* section of the pipeline.

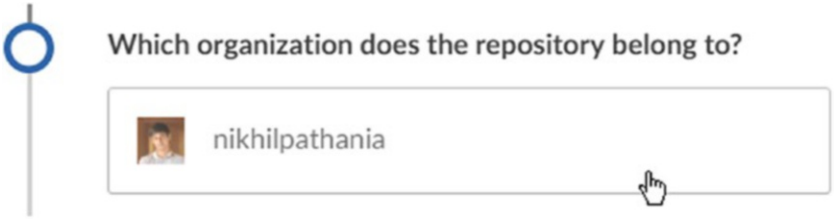

Figure 1-4. *Choosing a GitHub organization*

When you click on *Create Pipeline*, as shown in Figure 1-5, Blue Ocean searches for a Jenkinsfile inside your source code repository. You'll learn more about Jenkinsfile in the upcoming section.

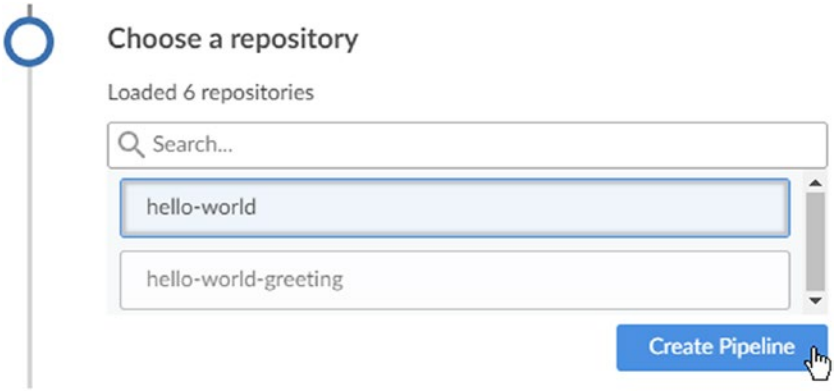

Figure 1-5. *Choosing a repository*

If it finds a Jenkinsfile, the wizard creates a pipeline in Jenkins Blue Ocean and initiates it. On the other hand, if no Jenkinsfile is found, the wizard takes you to the Pipeline Editor (see Figure 1-6). You'll see more about the Pipeline Editor in the next section.

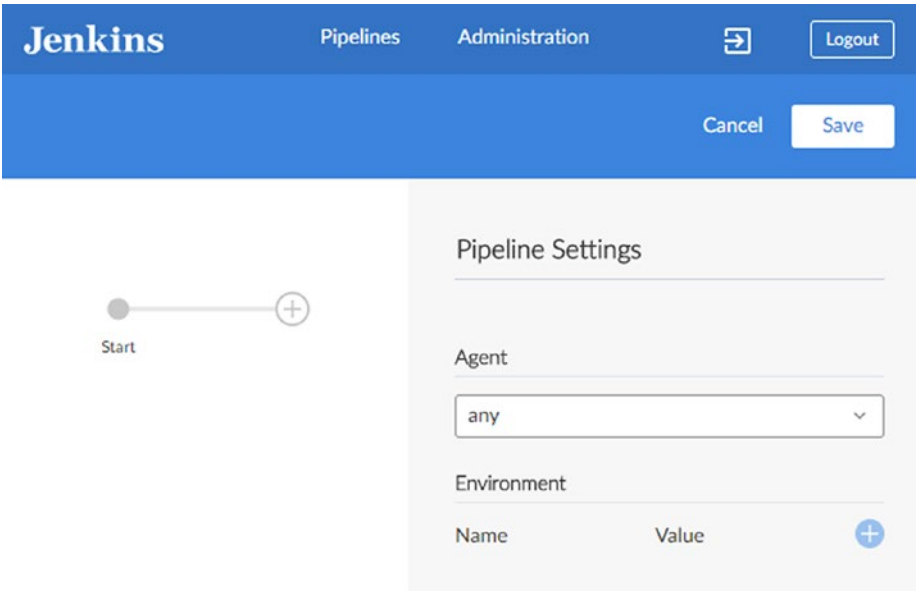

Figure 1-6. *The Visual Pipeline Editor*

When you try to create a pipeline in Jenkins Blue Ocean, you are bound to follow the aforementioned Pipeline Creation Wizard. It's restrictive in many ways, but the good part is that it does not baffle new users by flashing all possible pipeline configurations on a single page, like in the Classic Jenkins.

This new way of creating a pipeline in Jenkins Blue Ocean is quick, easy, and intuitive. Next, let's look at the Visual Pipeline Editor.

Visual Pipeline Editor

The Visual Pipeline Editor or Pipeline Editor is the most helpful feature in Jenkins Blue Ocean. It allows you to create a pipeline using a UI.

While using the Pipeline Editor, on the right side, you see options to configure the pipeline[2], and on the left side, you see a visual flow of your crude pipeline[1] (see Figure 1-7).

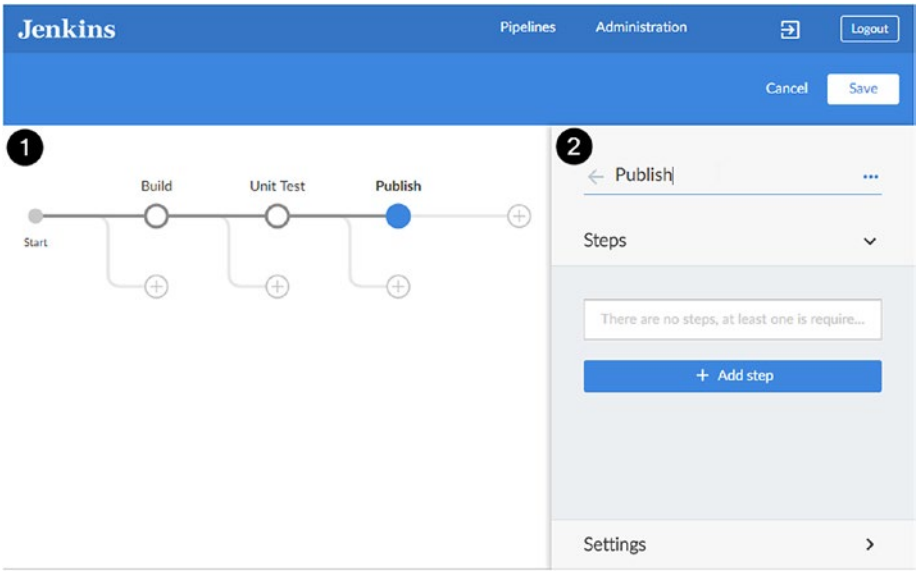

Figure 1-7. *Working with the Visual Pipeline Editor*

When you save your work, the Pipeline Editor converts it to a Jenkinsfile. You'll get a brief introduction to Jenkinsfile in the next section.

Pipeline Editor can do everything that is possible using the Declarative Pipeline Syntax. You'll learn about Declarative Pipeline Syntax in the coming section. There is also a chapter explaining Declarative Pipeline Syntax in detail.

When you install a Jenkins plugin, it makes itself available as a step inside the Pipeline Editor. Only Blue Ocean-compatible plugins are available. With every day that passes, more and more Jenkins plugins are getting Blue Ocean-ready.

In the upcoming chapters, you'll learn to create a pipeline using the Pipeline Editor.

Jenkinsfile

Jenkinsfile is a text file that contains your pipeline code. Jenkinsfile resides inside your source control repository alongside the source code. When you create a pipeline in Blue Ocean, it automatically saves your pipeline design as pipeline code, which is saved as a Jenkinsfile inside your source control repository.

There are many benefits of versioning your Jenkinsfile. Following are the direct benefits:

- Each branch of your source code repository can have its pipeline.
- It is possible to review the pipeline.
- Multiple project members can view/edit the pipeline.

A Sample Jenkinsfile Following the Declarative Pipeline Syntax:

```
pipeline {
    agent any
    stages {
       stage('Build') {
             steps {
                    sh 'mvn clean compile'
              }
       }
       stage('Unit Test') {
             steps {
                    sh 'mvn test'
                    junit '**/target/surefire-reports/TEST-*.xml'
             }
       }
       stage('Publish') {
           steps {
```

```
                sh 'mvn package'
                archive 'target/*.jar'
            }
        }
    }
}
```

Pipeline Visualization

Pipeline visualization is another nice feature that comes with Jenkins Blue Ocean. Figure 1-8 demonstrates an example of pipeline flow. As you can see, every circle represents a stage. A green circle with a right tick inside it signifies success.

Figure 1-8. *Pipeline visualization*

All the steps of a respective stage are visible below the pipeline flow. See Figure 1-9. You can expand a step to see its logs. The complete pipeline log is available in the Artifacts.

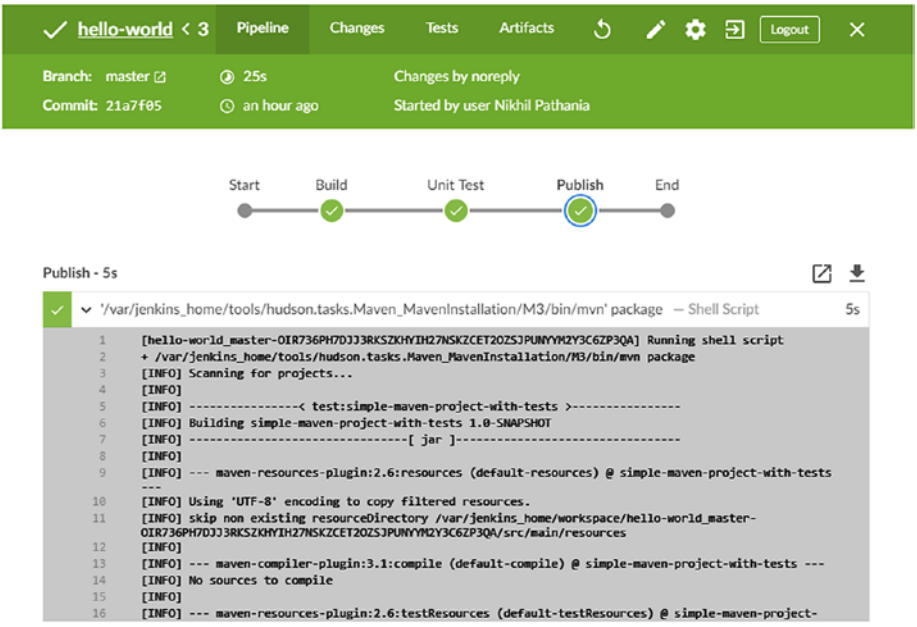

Figure 1-9. *Individual log about a step belonging to a stage of the pipeline*

Figure 1-10 demonstrates a pipeline that is in progress. You can quickly identify the stage of the pipeline that is in progress, separated nicely from the ones waiting to start.

Figure 1-10. *A pipeline stage in progress*

Figure 1-11 demonstrates a failed pipeline. You can easily distinguish the failed stage from the successful ones. It is also possible to see the exact step that failed in the pipeline (not shown).

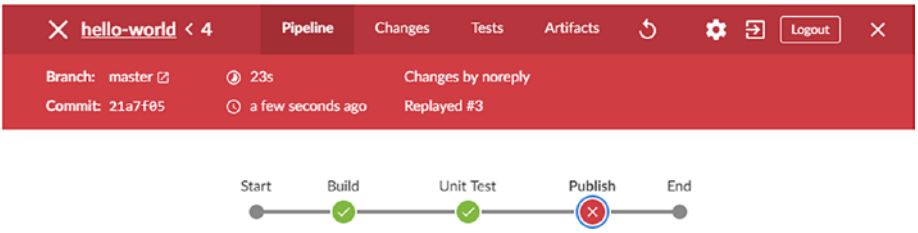

Figure 1-11. *A failed stage of a pipeline*

What you saw is a very simple pipeline flow. Blue Ocean, however, allows you to visualize complex pipeline flows that have nested parallel and sequential stages. You'll see them in the upcoming chapters.

View Changes, Tests, and Artifacts

See Figure 1-12. The Changes section shows you info about the commit that triggered the pipeline. Clicking on the commit takes you to the GitHub repository online. Also, the pipeline run status gets published on GitHub.

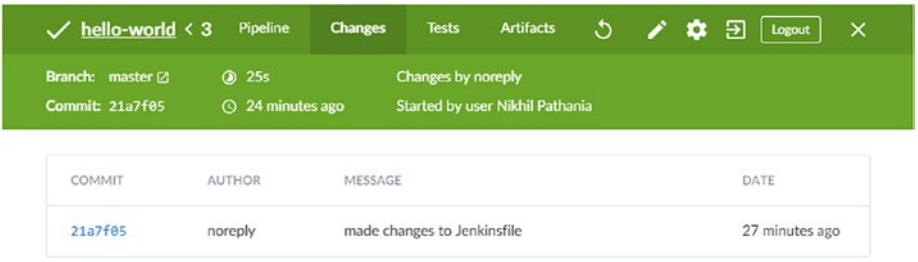

Figure 1-12. *The Changes tab from a pipeline run*

The Tests section gives you info about the test results (see Figure 1-13). You should have the necessary Jenkins Plugin installed and configured to read and display the test results.

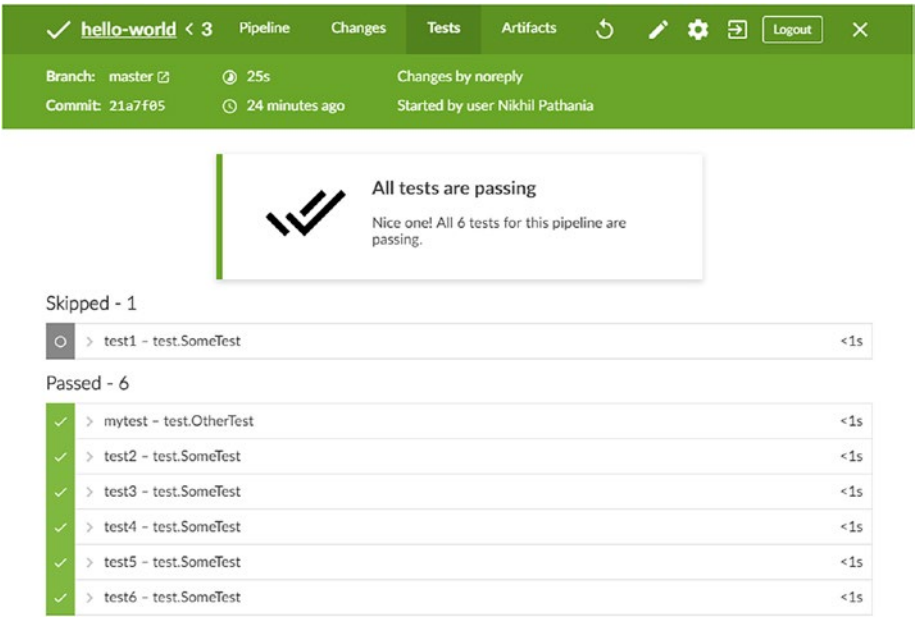

Figure 1-13. *The Tests tab from a pipeline run*

The last section is Artifacts. Here you find a complete log of the pipeline along with the artifacts that you choose to upload. Artifacts displayed here get stored on the Jenkins Server. It is not ideal to upload build-artifacts on the Jenkins server, especially the huge ones. You should use tools such as Artifactory for that purpose. Figure 1-14 shows you the Artifacts sections. The Artifact section is useful to share small reports for a quick view.

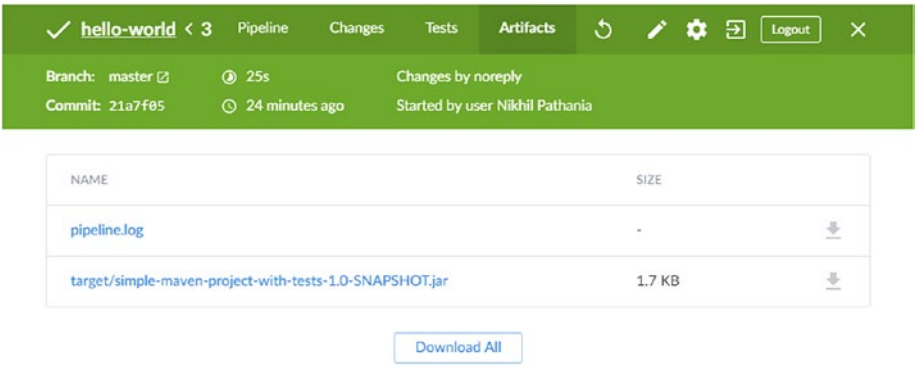

Figure 1-14. *The Artifacts tab from the pipeline run*

Pipeline Activity/Branches

For every push on your remote source control repository, a pipeline runs in Jenkins Blue Ocean. The Activity tab is the right place to see all the pipelines that are running or have run for a given project (see Figure 1-15).

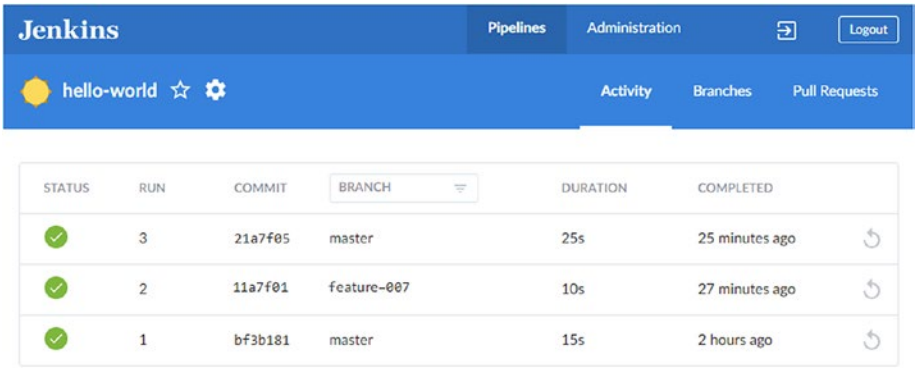

Figure 1-15. *The Activity view of the pipeline project dashboard*

It is also possible to segregate the pipelines as per branch (not shown). Figure 1-16 shows you the Branches tab. The following section lists the latest pipeline (if any) on every branch of your source control repository.

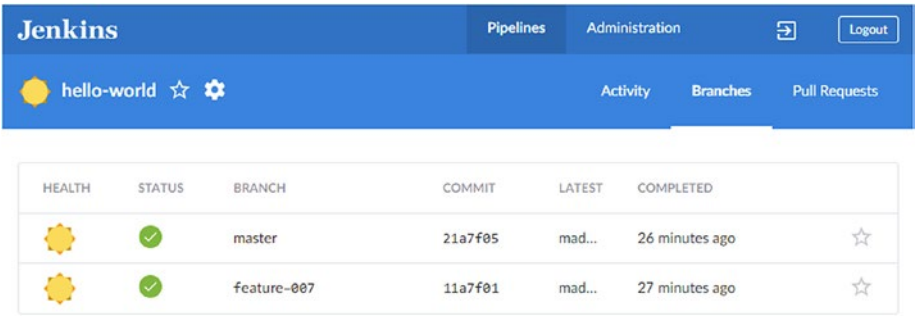

Figure 1-16. *The Branches view of the pipeline project dashboard*

Blue Ocean Dashboard

Figure 1-17 show you an example of the Jenkins Blue Ocean dashboard. The Blue Ocean dashboard lists all the projects. It is also the first page that you see when you open Jenkins Blue Ocean.

You get a health status about every project along with some statistics about the number of passing/failing branches [1]. You also have a Search tab to search for a project [2]. The *New Pipeline* button [3] allows you to create a new pipeline. The Administration link [4] takes you to the Manage Jenkins page. Button [5] takes you to the Classic Jenkins dashboard. The Logout button [6] is to log out from the current Jenkins session.

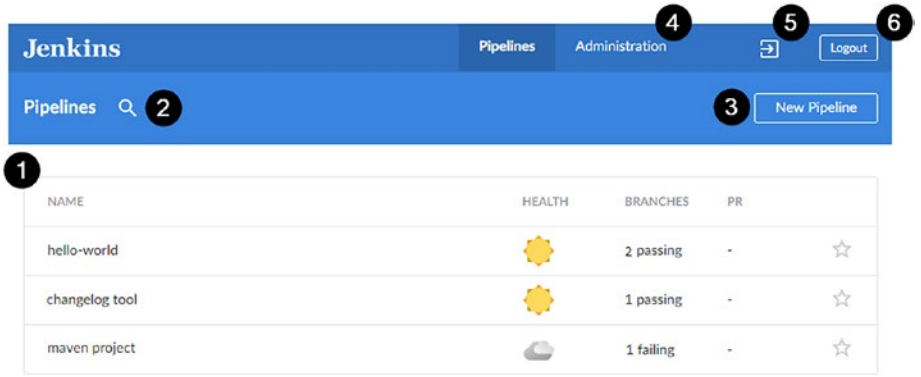

Figure 1-17. *The Jenkins Blue Ocean dashboard*

You'll see more about the visual elements of Jenkins Blue Ocean in the upcoming chapters.

Things to Consider

The following section deals with topics that are significant for you to consider before making Jenkins Blue Ocean your one-stop shop for creating pipelines.

Running Existing Jenkins Projects in Blue Ocean

At the time of writing this book, you can view and edit the following type of Jenkins Projects in Blue Ocean:

- A Pipeline Project that's created using Jenkins Blue Ocean

You can view but not edit the following types of pipelines in Blue Ocean:

- A Pipeline Project that's created using Classic Jenkins, is a multibranch pipeline, and is written in Declarative Pipeline Syntax.

- A Pipeline Project created using Classic Jenkins; it could be a multibranch pipeline, GitHub Organization Project, or a Pipeline Project. Furthermore, it follows an Imperative Pipeline Syntax (basically a Scripted Pipeline).

Do Freestyle/Multi-Configuration Projects Work in Blue Ocean?

At the time of writing this book, it is possible to view your Freestyle project in Blue Ocean, as shown in Figure 1-18.

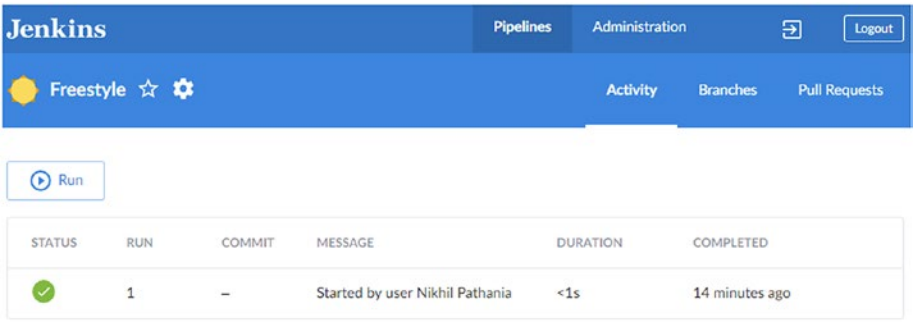

Figure 1-18. *Viewing a Freestyle project dashboard in Blue Ocean*

There are no stages or steps, and you see one single log for the complete build execution (see Figure 1-19).

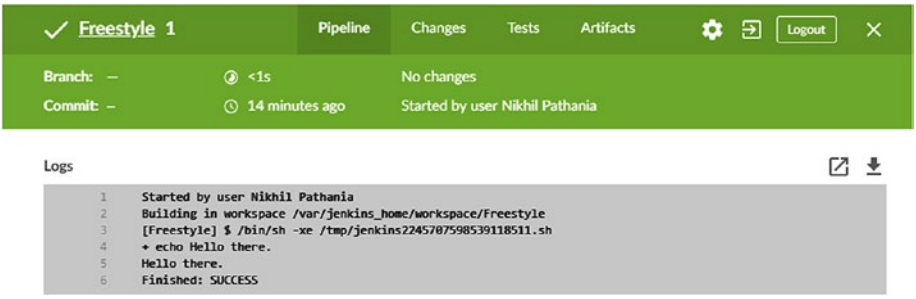

Figure 1-19. *Viewing a Freestyle project pipeline in Blue Ocean*

In addition to this, you won't be able to use the Blue Ocean features build around the pipelines. For example, you can't edit the Freestyle Project using the Pipeline Editor, and you can't use the Pipeline Visualization.

However, it may be possible for Blue Ocean to support the Freestyle Project and the other types of Jenkins jobs in the future.

Declarative Pipeline Syntax

The foundation of Jenkins pipeline is Groovy. Jenkins even has an embedded Groovy Engine. There is no limit to what an admin or a user can write using Groovy.

With Groovy as a foundation, the Jenkins engineers came up with Scripted Pipeline, also known as Pipeline Script. The Scripted Pipeline is flexible and extensible but has a steep learning curve.

In addition to that, Scripted Pipeline happens to be less structured. It comes with a higher technical debt, higher maintenance, and lower code readability.

An Example of Scripted Pipeline:

```
node {
    try {
       stage('Greet') {
            sh 'echo "Hello there"'
       }
    } finally {
        // The following runs always
        echo 'The pipeline has passed.'
    }
}
```

To make things simpler, Jenkins has come up with the Declarative Pipeline Syntax. It's structured and comes with a smooth learning curve. If you have tried the Scripted Pipeline, you may find Declarative a bit restrictive.

The Declarative Pipeline Syntax is suitable for writing simple Continuous Delivery Pipelines. However, it's also possible to make it extensible using Jenkins Shared Libraries.

An Example of Declarative Pipeline:

```
pipeline {
    agent any
    stages {
        stage('Greet') {
            steps {
                sh 'echo "Hello there"'
            }
        }
    }
    post {
        always {
            // The following runs always
            echo 'The pipeline has passed.'
        }
    }
}
```

Pipelines created using Jenkins Blue Ocean follow the Declarative Pipeline Syntax. The Pipeline Editor in Jenkins Blue Ocean reads/writes pipeline code to Jenkinsfile in the Declarative Pipeline Syntax. It's designed to make it easy to write pipeline using a text editor. You'll learn more about the elements of Declarative Pipeline Syntax in the upcoming chapter.

Sailing Through Complex Pipelines

Declarative Pipelines Syntax is restrictive and not as flexible as its counterpart. However, it is possible to write complex pipelines by using the following in your Declarative Pipelines:

- Script step
- Jenkins Shared Libraries

Script step allows you to write a portion of Scripted Pipeline inside your Declarative Pipeline. You should use this feature only when necessary.

An Example of Declarative Pipeline With a Script Step:

```
pipeline {
    agent any
    stages {
        stage('EvenorOdd') {
            steps {
                script {
                    if (currentBuild.number % 2 ==0) {
                        echo "The current build number is even."
                    }
                    else {
                          echo "The current build number is odd."
                    }
                }
            }
        }
    }
}
```

However, more prominent and complex script step should move into a Shared Library. A Jenkins Shared Library is an external source control repository containing your complex Groovy code. It acts like a function that could be used on-demand inside your Declarative Pipeline.

A Groovy Script (example.groovy)Inside Jenkins Shared Library Repository:

```
def greet(message) {
    echo "Hello ${message}, welcome to Jenkins Blue Ocean."
}
```

Jenkins Declarative Pipeline Utilizing the Shared Library:

```
@Library('Example_Shared_Library') _

pipeline {
    agent none
    stage ('Example') {
        steps {
             script {
                 example.greet 'Readers'
             }
        }
    }
}
```

You'll learn more about using Script steps and Jenkins Shared Libraries in the upcoming chapters.

Do I Still Need to Visit the Standard Jenkins Interface?

The answer is yes. To work with the Classic Jenkins Projects—such as Freestyle Project, Multi-configuration Project, Pipeline Project, to name a few—you must visit the Classic Jenkins UI.

The Classic Jenkins dashboard would also continue to remain as an entry page after you install Jenkins or when you log in to Jenkins. Even with the Jenkins Blue Ocean installed, the Classic Jenkins UI remains the default entry point when you log in to Jenkins.

The Classic Jenkins UI including the dashboard is not going to fade out any time soon. However, as more and more users move toward Jenkins Blue Ocean for creating and editing pipelines, there will be less reason to visit the Classic Jenkins UI, and managing Jenkins pipelines through the Classic Jenkins UI may become unpopular.

However, you don't need to access the Classic Jenkins UI to perform administrative tasks. The Jenkins Configuration page for the administrative activities is now directly accessible from Jenkins Blue Ocean Dashboard. See Figure 1-20 for the *Administration* link.

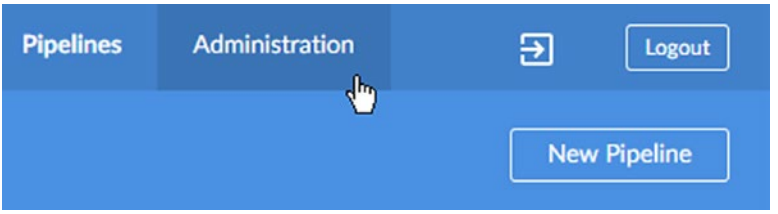

Figure 1-20. *Link to the Jenkins administration page from Blue Ocean*

Is It Wise to Move Now?

Both Jenkins Blue Ocean and Classic Jenkins co-exist happily together. Pipelines created in Jenkins Blue Ocean are fully compatible in Classic Jenkins. Installing Jenkins Blue Ocean is made easy by shipping it as a Jenkins Plugin. Having said so, new users should for sure try Blue Ocean.

However, is it wise for the existing Jenkins users to move to Jenkins Blue Ocean? Yes, it is, for the following reasons:

- Creating pipelines in Jenkins Blue Ocean is easy. The Declarative Pipeline Syntax is designed to be structured and straightforward. As a result, it is suitable for the users who prefer writing pipelines using a text editor. For the rest, there is the easy-to-use Visual Pipeline Editor.
- Jenkins Blue Ocean promotes Declarative Syntax. Pipelines written in Declarative Pipeline Syntax are easy to maintain, and from experience, maintainability is a precious thing.
- Creating a pipeline using Jenkins Blue Ocean is made easy through all its brilliant features. This eases doing business, making it the best tool for teams who expect all their developers to be well-versed in creating pipelines.
- The *Stage View* feature available in Classic Jenkins is incapable of rendering parallel pipeline stages correctly. Furthermore, there is no attempt to fix it, since Jenkins Blue Ocean already comes with a feature to visualize pipelines.
- Onboarding new users to Blue Ocean is easier than Classic Jenkins, as it's more intuitive.

- Debugging failures in Jenkins Blue Ocean is easier than Classic Jenkins. It is made possible by displaying the build log for each step separately. Also, the status of the pipeline is visible at the stage and step level. All these features allow developers to pinpoint failures with accuracy.

Apart from the above points, the Jenkins community is actively listening to its users (Jenkins as well as Blue Ocean users). Furthermore, all the feedback received goes into Blue Ocean's roadmap. Nevertheless, it's a challenge to make Jenkins Blue Ocean compatible with the full range of configurations out there.

Who Should Use Blue Ocean?

Jenkins Blue Ocean puts creating pipelines in the hands of developers who are a novice to Continuous Integration and Continuous Delivery practices. The easy-to-use Pipeline Creation Wizard in addition to the Pipeline Editor do all that is needed.

Though Blue Ocean is for everyone, it is best-suited for the following:

- Development teams who use Git/GitHub/GitLab/Bitbucket as their source control tool
- Development teams who encourage using feature branches, or shall I say GitFlow Workflow[5]
- Development teams who create and manage pipelines themselves, and where the Jenkins admin's involvement is minimal or limited to administrative tasks

[5]A strict branching model that emphasizes using a Develop and a Master branch, multiple features, releases, and Hotfix branches

- Development teams with a requirement of simple pipelines that stay under the framework of Declarative Pipeline Syntax and Jenkins Shared Libraries
- Development teams who would like to build & test their changes before and after a code review
- Development teams who do not possess legacy elements[6] anywhere in their environment

In the coming chapters, you'll learn to write simple as well as complex pipelines using Jenkins Blue Ocean.

What to Expect in the Future?

With the advent of the pipeline as a code (Scripted Pipelines in general), Freestyle Jenkins Jobs are on the brink of extinction. Similarly, with the arrival of Declarative Pipeline Syntax, fewer developers are expected to go for the Scripted Pipelines. The reason is that Declarative Pipeline Syntax is easy to understand, use, and maintain.

Scripted Pipelines, however, are not going to vanish from the scene, as there will always be a need for writing complex pipelines. Nevertheless, for a majority of people, Blue Ocean is going to turn out to be their favorite.

I do not claim to have seen the future, but I can say for sure Blue Ocean is the way to go. It is going to be the Continuous Delivery tool for the masses, and there would undoubtedly be a need for a second version of this book.

A lot of features and improvements are waiting to see their first light in Blue Ocean. Information on the most important ones is available on the Jenkins Blue Ocean Roadmap page (`https://jenkins.io/projects/blueocean/roadmap/`). On this page, you'll find a list of features with information on their current status (see Figure 1-21).

[6]Software or hardware that has been superseded but is difficult to replace because of its wide use

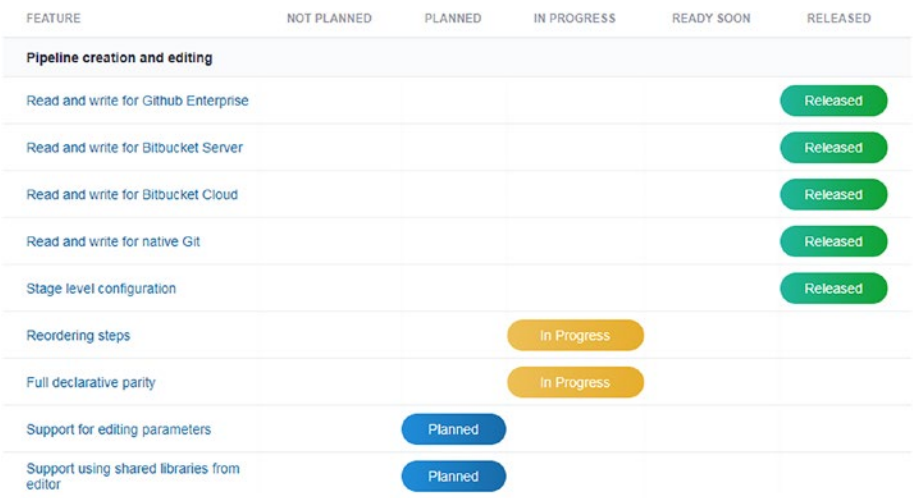

Figure 1-21. *The Jenkins Blue Ocean roadmap page*

Since Jenkins is a community-driven tool, it is possible that someday the Classic Jenkins jobs, such as the Jenkins Freestyle Project and the Jenkins Pipeline (Scripted Pipeline) , would be editable in Jenkins Blue Ocean.

On the Jenkins Plugin front, more and more plugins are expected to be available inside Jenkins Blue Ocean as steps.

The Declarative Pipeline Syntax and the Pipeline Editor, along with the other cool features of Jenkins Blue Ocean, are already making it easy for teams to adopt Continuous Delivery practices.

Summary

In this chapter, you learned why Jenkins Blue Ocean exists. It's a step toward providing a better user experience, with an aim to bring CI/CD to the masses. You took a slice of what Blue Ocean has to offer. The Pipeline Creation Wizard and the Pipeline Editor look promising. The Pipeline visualization is intuitive and interactive. In the upcoming chapters, you'll learn more about them.

A Pipeline created using Jenkins Blue Ocean follows the Declarative syntax. The Declarative syntax allows you to write structured and maintainable pipelines. You'll learn more about the Declarative Pipeline Syntax in the upcoming chapter.

With Jenkins Blue Ocean, you can create a simple CD pipeline with ease. However, it's also possible to write complex pipelines using Shared Libraries. You'll see both of these in the coming chapters.

You also saw the types of Classic Jenkins pipelines compatible with Blue Ocean and the ones that are not.

With this, I conclude the first chapter. I hope that your expectations are set right, so let us move forward. In the next chapter, you'll learn the various ways to set up Blue Ocean.

CHAPTER 2

Setting up Jenkins Blue Ocean

Installing Blue Ocean is easy. Existing Jenkins users can try it by installing the Jenkins Blue Ocean Plugin. New users and docker enthusiasts can try the docker image for Jenkins Blue Ocean. The current chapter describes both methods in detail.

In this chapter, you'll learn the following:

- Setting up Blue Ocean using docker
- Using the Jenkins Setup Wizard
- Setting up Blue Ocean on an existing Jenkins Server
- Things to consider when running Jenkins Blue Ocean behind a reverse proxy server
- Things to consider when running Jenkins Blue Ocean on Apache Tomcat Server

Sometimes, a reverse proxy server can rewrite URIs that contain encoding. As a result, accessing a Blue Ocean link from the Classic Jenkins results in a Page not found (404) error. Therefore, I have included the section *Things to Consider* at the end of the current chapter.

So let us begin.

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5_2

Setting up Blue Ocean Using Docker

In the following section, you'll learn to set up Jenkins Blue Ocean the fun way. Setting up Jenkins Blue Ocean using docker is quick and easy. I would recommend you to take a look at the section *Basic Docker Concepts* in the *Appendix* if docker is something new to you.

To follow the guidelines described in the current section, you'll need a docker host. If you don't have one, check the section *Setting up a Docker Host* in the *Appendix*. Make sure your docker host can pull images from the Docker Hub.

Download the Jenkins Blue Ocean Docker Image

For Jenkins Blue Ocean there is a particular docker image on the Docker Hub: *jenkinsci/blueocean*. The docker image *jenkinsci/blueocean* comes with Blue Ocean Plugin installed.

While pulling the Jenkins Blue Ocean docker image, you can choose between the latest and a specific version of Blue Ocean release.

Downloading the Latest Jenkins Blue Ocean (recommended)

Blue Ocean is new, with a few releases so far. Every new release of Blue Ocean comes with a new feature or a major fix. Therefore, it's recommended to use the latest version of Blue Ocean. Follow these steps:

1. Pull the latest docker image of Jenkins Blue Ocean. Use `sudo` wherever necessary.

   ```
   docker pull jenkinsci/blueocean
   ```

Output:

```
Using default tag: latest
latest: Pulling from jenkinsci/blueocean
8e3ba11ec2a2: Pull complete
311ad0da4533: Pull complete
.
.
.
022c7adf3026: Pull complete
621cbdrttf37: Pull complete
24600ace3bc4: Pull complete
Digest: sha256:e5ad8063b4cbfac7fe4f8a5afd5226f395b2ac2d71f88d30
4f2a80a40ade78a9
Status: Downloaded newer image for jenkinsci/blueocean:latest
```

2. To list the downloaded docker image, execute the following docker command:

```
docker images
```

Output:

```
REPOSITORY           TAG     IMAGE ID      CREATED        SIZE
jenkinsci/blueocean  latest  43d06f6468f0  7 hours ago    443MB
```

And this is how you download the latest version of the Jenkins Blue Ocean docker image.

Downloading a Specific Version of Jenkins Blue Ocean

Alternately, to download a specific release of Jenkins Blue Ocean, you'll need to check the list of available tags for the Jenkins Blue Ocean repository: *jenkinsci/blueocean*. Follow these steps:

1. Access the link: https://hub.docker.com/r/jenkinsci/blueocean/tags/ to see the list of available tags (see Figure 2-1).

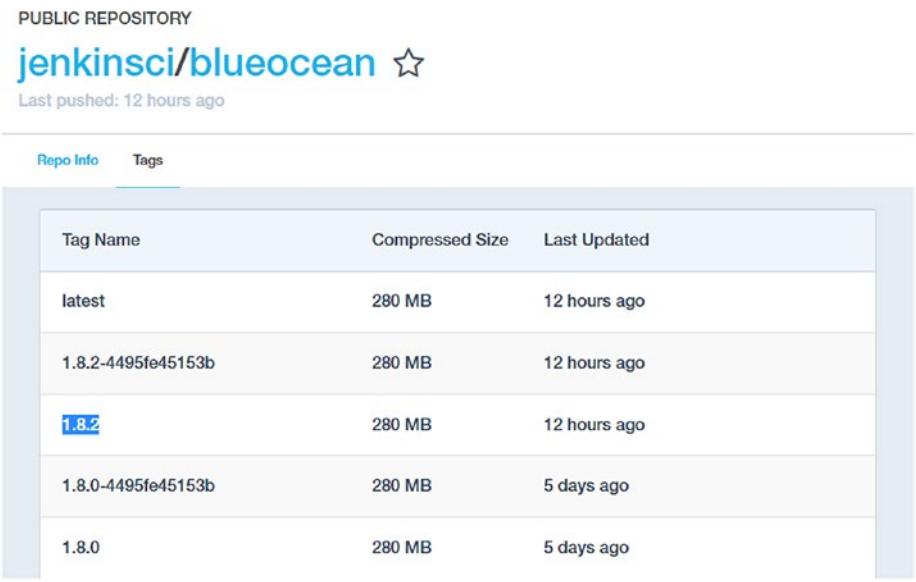

Figure 2-1. *List of available tags for the jenkinsci/blueocean repository on Docker Hub*

2. Once you have decided on a specific tag, use it in the below command to download the right release of Jenkins Blue Ocean. Use `sudo` wherever necessary.

   ```
   docker pull jenkinsci/blueocean:<Tag Name>
   ```

Example:

```
docker pull jenkinsci/blueocean:1.8.2
```

Output:

```
1.8.2: Pulling from jenkinsci/blueocean
8e3ba11ec2a2: Pull complete
311ad0da4533: Pull complete
.
.
.
de61388d6431: Pull complete
022c7adf3026: Pull complete
24600ace3bc4: Pull complete
Digest: sha256:e5ad8063b4cbfac7fe4f8a5afd5226f395b2ac2d71f88d30
4f2a80a40ade78a9
Status: Downloaded newer image for jenkinsci/blueocean:1.8.2
```

3. To list the downloaded docker image, execute the following docker command:

```
docker images
```

Output:

```
REPOSITORY            TAG     IMAGE ID      CREATED        SIZE
jenkinsci/blueocean   1.8.2   43d06f6468f0  7 hours ago   443MB
```

And this is how you download the specific version of Jenkins Blue Ocean docker image.

Spawning a Jenkins Blue Ocean Container

In the following section, you'll learn to run a container using the Jenkins Blue Ocean's docker image. We'll do this in a bit particular way, by using docker volumes. Wondering what a docker volume is? Here is the answer.

Docker containers generate and use data. When a container gets deleted, its relevant data also gets lost. To make the data persistent, we use docker volumes.

Bind mount is another way of making the data generated by a container persistent. However, using docker volumes is always a better choice over bind mounts.

Create a Docker Volume

In the following section, you'll create, list, and describe a docker volume. Follow these steps:

1. Create a docker volume named: `jenkins_home`. Use `sudo` wherever necessary.

   ```
   docker volume create jenkins_home
   ```

Output:

```
jenkins_home
```

2. List the newly created docker volume.

   ```
   docker volume ls
   ```

Output:

```
DRIVER              VOLUME NAME
local               jenkins_home
```

3. To get detailed info about your docker volume, use the `docker inspect` command.

   ```
   docker volume inspect jenkins_home
   ```

Output:

```
[
    {
        "CreatedAt": "2018-08-23T22:04:14+02:00",
        "Driver": "local",
        "Labels": {},
        "Mountpoint": "/var/lib/docker/volumes/jenkins_home/_data",
        "Name": "jenkins_home",
        "Options": {},
        "Scope": "local"
    }
]
```

Running a Jenkins Blue Ocean Container Using a Volume

You now have the required docker image and docker volume to spawn a Jenkins Blue Ocean container. Follow these steps:

1. Spawn a container using the `docker run` command.

```
docker run -d --name jenkins \
-p 8080:8080 -p 50000:50000 \
-v jenkins_home:/var/jenkins_home \
jenkinsci/blueocean
```

Output

```
54172c9b92b6c9589f11d117093912b610e9edad43ba0f74a84543ea57077237
```

The following table explains the `docker run` command used to run the Jenkins Blue Ocean container.

Option	Description
-d	Run container in background and print container ID
--name jenkins	Assign a name to the container.
-p 8080:8080	Publish the container's port: 8080 to the host port: 8080.
-p 50000:50000	Publish the container's port: 50000 to the host port: 50000.
-v jenkins_home:/var/jenkins_home	Mount the jenkins home directory inside the container: `/var/jenkins_home` to docker volume: `jenkins_home`
jenkinsci/blueocean	Docker image: `jenkinsci/blueocean` used to spawn the container.

Alternatively, to run a container using a specific release of Jenkins Blue Ocean image, execute the following docker command:

```
docker run -d --name jenkins \
-p 8080:8080 -p 50000:50000 \
-v jenkins_home:/var/jenkins_home \
jenkinsci/blueocean:<Tag Name>
```

2. Run the following command to list your running containers:

```
docker ps --format "table {{.ID}} \t {{.Image}} \t
{{.Ports}} \t {{.Names}}"
```

Output:

```
CONTAINER ID  IMAGE                PORTS                    NAMES
54172c9b92b6  jenkinsci/blueocean  0.0.0.0:8080->8080/tcp, jenkins
                                   0.0.0.0:50000->50000/tcp
```

3. Run the following command to get the running status of your containers:

```
docker ps --format "table {{.ID}} \t {{.Names }}
\t {{.Status}}"
```

Output:

```
CONTAINER ID        NAMES               STATUS
54172c9b92b6        jenkins             Up 2 hours
```

You can execute the following command to get complete info about your running container.

```
docker inspect jenkins
```

You'll now be able to access Jenkins using the following URL:
`http://<Docker Host IP>:8080/`

Running Through the Jenkins Startup Wizard

You now have a Jenkins Blue Ocean container up and running. Next, you should go through the Jenkins Setup Wizard to start using Jenkins. During the setup wizard, you'll do the following:

- Unlock Jenkins.
- Install some basic Jenkins Plugins.
- Create an admin user.
- Choose between existing and a new Jenkins URL.

Unlock Jenkins

When you access the Jenkins URL, you'll be asked to unlock it using an initial password (see Figure 2-2). You can find this initial password inside the file */var/jenkins_home/secrets/initialAdminPassword* [1]. The file is inside the Jenkins Blue Ocean container.

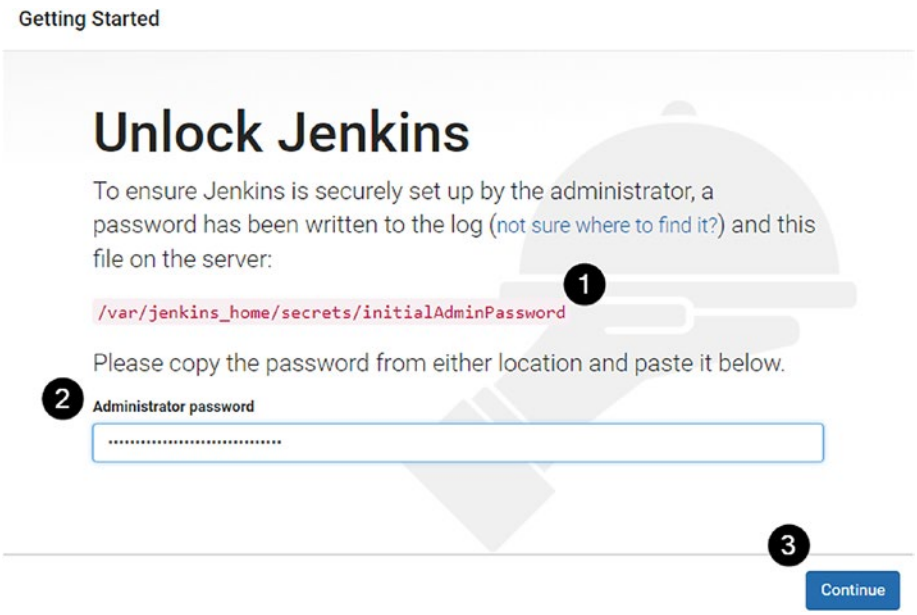

Figure 2-2. *Unlocking Jenkins*

Run the following docker command to fetch the content of *initialAdminPassword* file.

```
docker exec -it jenkins /bin/bash -c \
 "cat /var/jenkins_home/secrets/initialAdminPassword"
```

Output:

```
094aef10666b4e198fec840069c311de
```

Paste the output of the above command inside the field **Administrator password**[2]. Click on **Continue**[3] to proceed.

Installing Some Basic Plugins

Next, the wizard requests you to install some essential Jenkins plugins by offering you two choices: **Install suggested plugins**[1] and **Select plugins to install**[2] (see Figure 2-3).

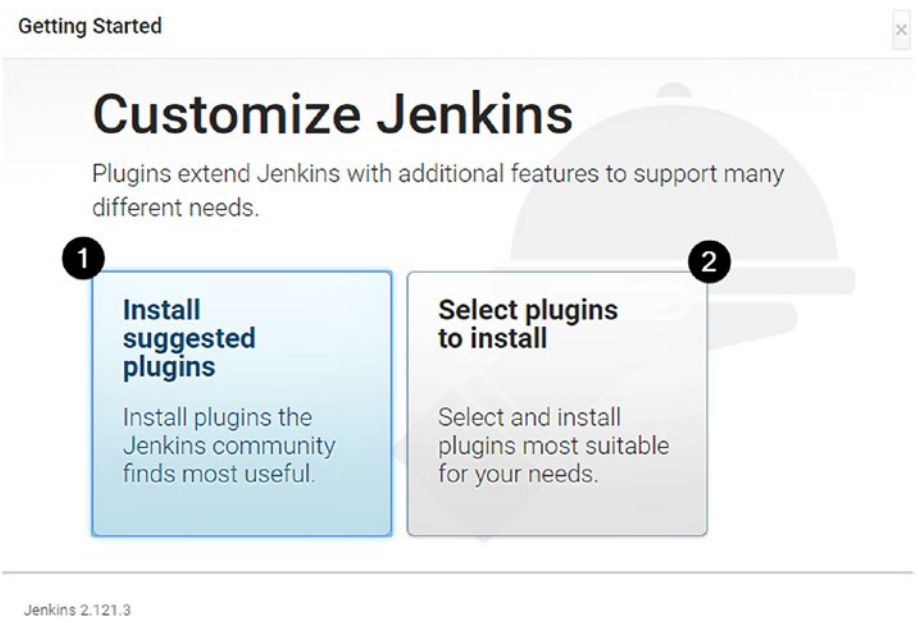

Figure 2-3. *Choosing between suggested plugins and custom plugins to install during the Jenkins Setup Wizard*

Choosing the option **Install suggested plugins** installs some essential plugins that Jenkins thinks are important. Choosing the option **Select plugins to install** takes you to another page (see Figure 2-4).

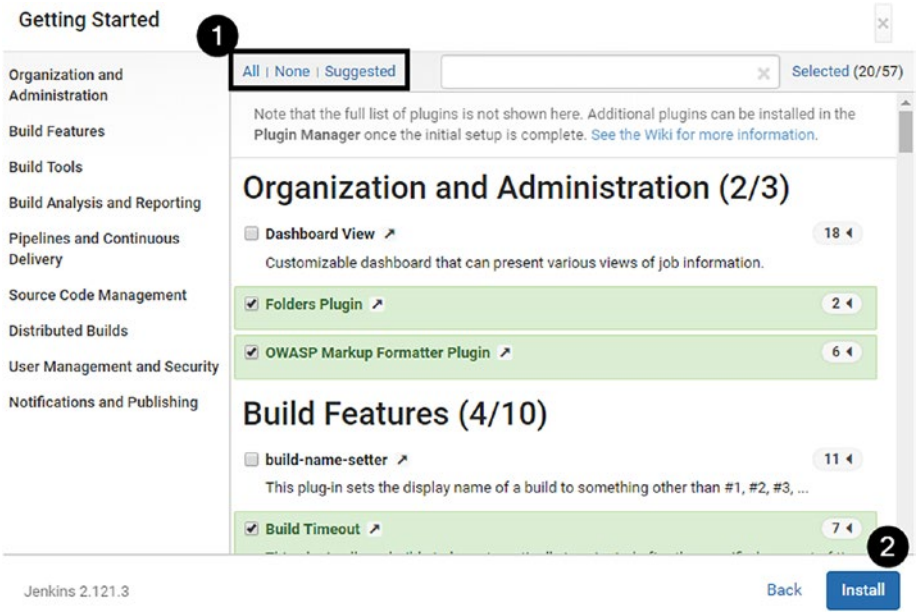

Figure 2-4. *Choosing the custom plugins to install during the Jenkins Setup Wizard*

On this new page, you can choose to install between **All**, **None,** or **Suggested** plugins [1].

After making a selection, click **Install** [2] to proceed.

Next, Jenkins wizard installs the plugin.

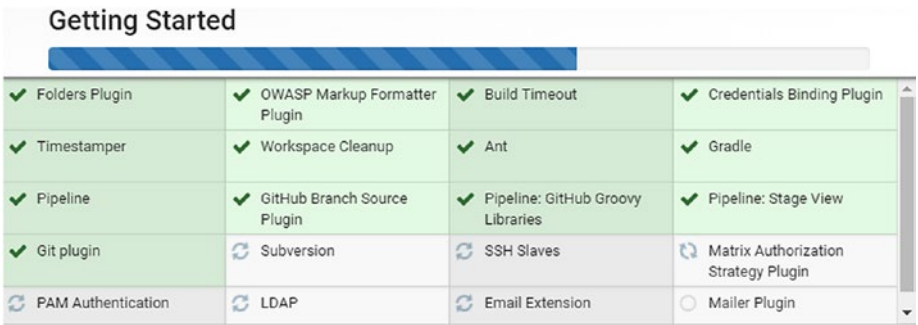

Figure 2-5. *Plugin installation in progress*

Creating an Admin User

After installing some essential Jenkins plugins, the Wizard requires you to create a new admin user (see Figure 2-6).

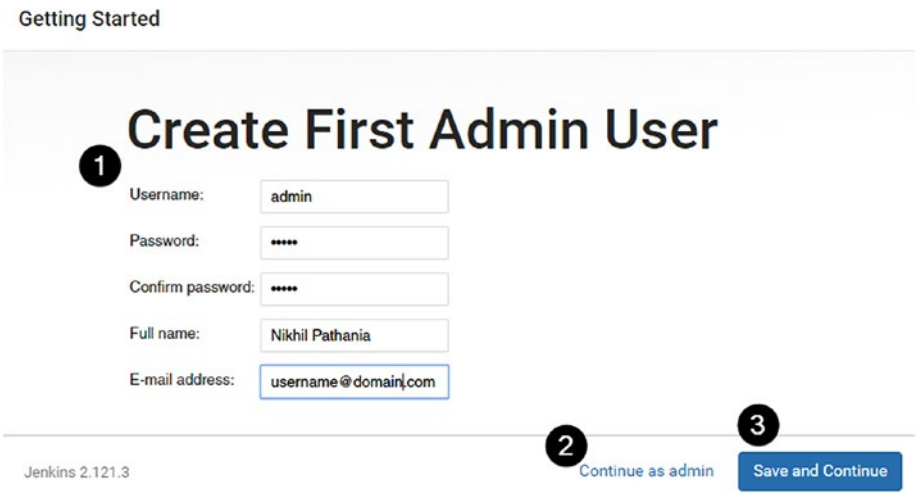

Figure 2-6. *Creating your first admin user account during the Jenkins Setup Wizard*

All the fields in Figure 2-7 are self-explanatory. You also have the option to continue with the existing admin user [2]. The existing user has the following credentials:

- Username: *admin*
- Password: *the content of /var/jenkins_home/secrets/initialAdminPassword*

When you have made your choice, click on **Save and Continue** [3].

Configuring the Jenkins URL

Next, the Wizard asks you to configure the Jenkins URL (see Figure 2-7). The field **Jenkins URL** [1] comes pre-filled with an existing URL. The pre-filled URL is where Jenkins is currently available.

For the sake of this exercise, you can leave the Jenkins URL field to its default value by clicking on either **Not now** [2] or **Save and Finish** [3].

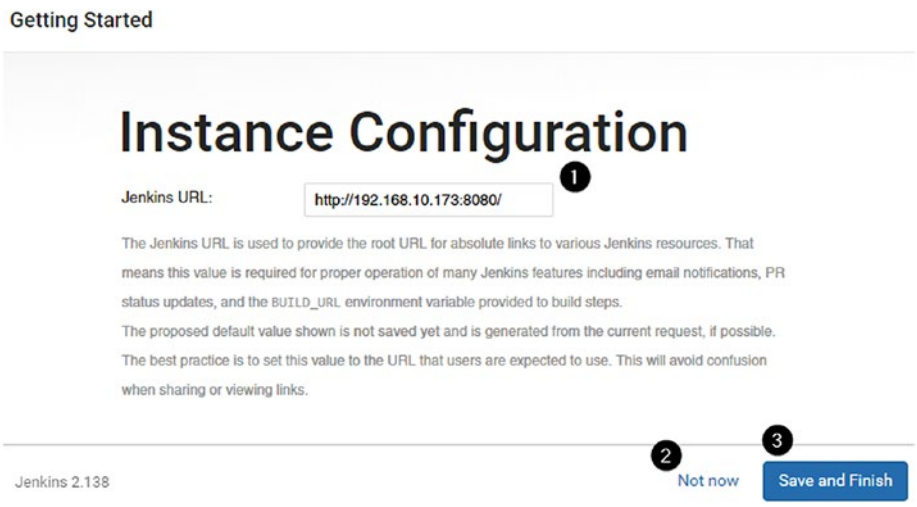

Figure 2-7. *Configuring the Jenkins URL during the Jenkins Setup Wizard*

Now, your Jenkins setup is complete. Click on **Start using Jenkins** when you are ready (see Figure 2-8).

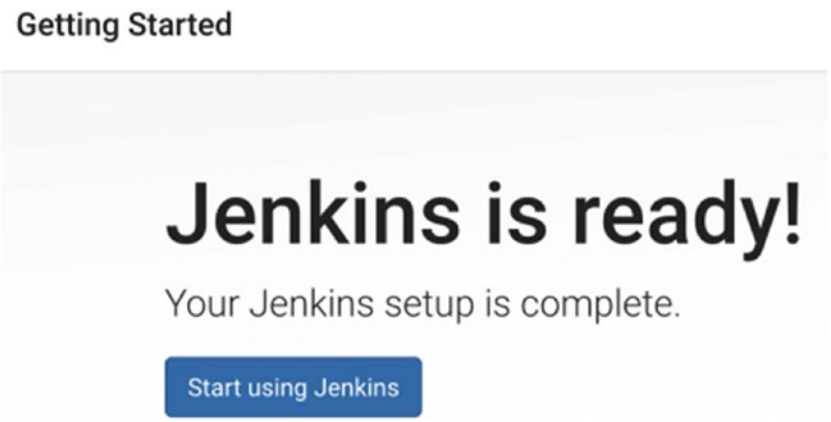

Figure 2-8. *Welcome page to start using Jenkins at the end of the Jenkins Setup Wizard*

Remember to read the section *Enabling proxy compatibility for Jenkins* in the *Appendix* chapter if you are running Jenkins on a Docker host or a Kubernetes cluster that's on a cloud platform.

Accessing the Blue Ocean Dashboard

As you have learned earlier, the classic Jenkins Dashboard is still the starting page for Jenkins. Therefore, you won't see the Jenkins Blue Ocean Dashboard right away.

To access Blue Ocean, click on the link **Open Blue Ocean** [1] from the top-left side menu on the Classic Jenkins Dashboard (see Figure 2-9).

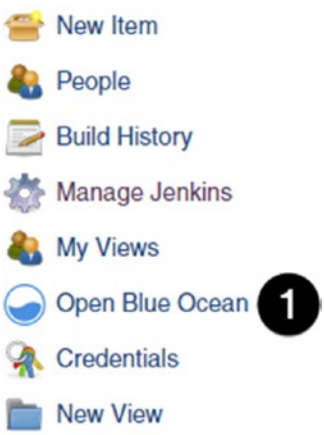

Figure 2-9. *The Jenkins Blue Ocean link on the Classic Jenkins Dashboard*

And finally, this is how the Jenkins Blue Ocean Dashboard looks (see Figure 2-10).

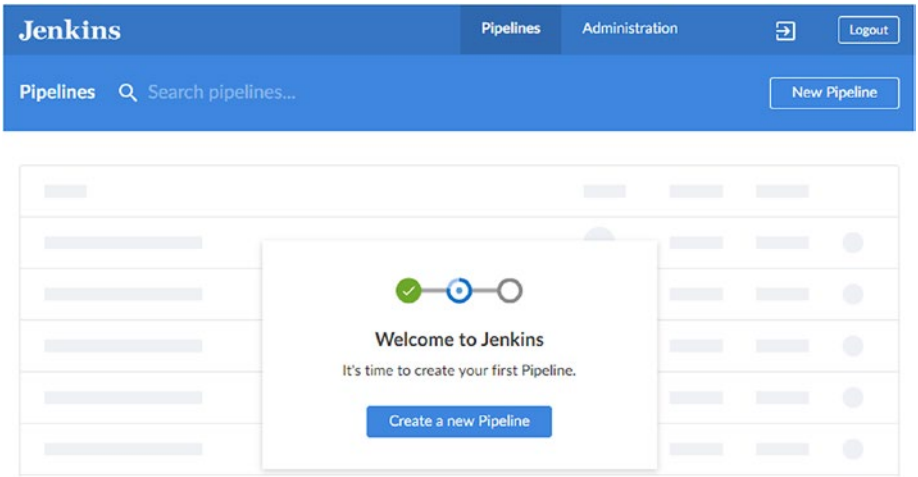

Figure 2-10. *The Jenkins Blue Ocean Dashboard*

JENKINS DISASTER RECOVERY

The `jenkins_home` directory stores all the Jenkins configuration and data. Reading/writing the content of `jenkins_home` to a docker volume allows you to recover your Jenkins server during failures.

The following exercise helps you to get a glimpse of what is possible when running the Jenkins Blue Ocean container with docker volume.

Following is the summary of what you'll be doing in the current exercise:

- Make some changes on the Jenkins server. There is no need to do it, however, because, during the Jenkins Setup Wizard, you have installed some plugins and *created a new admin user. These activities are very well* a change on the Jenkins server.
- Delete the existing running Jenkins container: `jenkins`, thus, simulating a Jenkins crash. However, keep the docker volume: `jenkins_home`.
- You'll then create a new Jenkins container and bind it to the same old docker volume: `jenkins_home`.

The exercise is considered a success if:

- All your previous changes on the Jenkins server are intact after recreating the Jenkins container.
- You see the Jenkins login page instead of the Jenkins Setup Wizard.
- You can log in using the admin user that you created previously during the Jenkins Setup Wizard.

Following are the exercise steps:

1. Stop the existing Jenkins container.

   ```
   docker stop jenkins
   ```

2. Remove the existing Jenkins container.

   ```
   docker rm jenkins
   ```

3. Run the following command to confirm deleting the Jenkins container. You should get an empty output, or, at the least, there should be nothing with the name `jenkins`.

   ```
   docker ps -a
   ```

4. Run the following docker command to spawn a new Jenkins container.

   ```
   docker run -d --name jenkins \
   -p 8080:8080 -p 50000:50000 \
   -v jenkins_home:/var/jenkins_home \
   jenkinsci/blueocean
   ```

5. Access the Jenkins server using the following URL: `http://<Docker Host IP>:8080/`.

You should see a login page instead of the Jenkins Setup Wizard. Moreover, you should be able to log in using the admin user account created earlier.

SPAWNING A CLASSIC JENKINS SERVER WITH DOCKER

In the following exercise, you'll run a Jenkins container using the standard Jenkins docker image: *jenkins/jenkins*.

1. Pull the latest docker image of Jenkins Blue Ocean. Use `sudo` wherever necessary.

   ```
   docker pull jenkins/jenkins
   ```

   (or)

   ```
   docker pull jenkins/jenkins:<Tag Name>
   ```

2. Create a docker volume.

   ```
   docker volume create jenkins_home_classic
   ```

3. You can list and inspect your new volume

   ```
   docker volume ls
   ```

   (and)

   ```
   docker volume inspect jenkins_home_classic
   ```

4. Spawn a container using the Jenkins docker image.

   ```
   docker run -d --name jenkins_classic \
   -p 8081:8080 -p 50001:50000 \
   -v jenkins_home:/var/jenkins_home \
   jenkins/jenkins
   ```

   Alternatively, you can run a container using a specific version of Jenkins image using the following command:

   ```
   docker run -d --name jenkins_classic \
   -p 8080:8080 -p 50000:50000 \
   -v jenkins_home_classic:/var/jenkins_home \
   jenkins/jenkins:<Tag Name>
   ```

5. Access the Jenkins server using the following URL: `http://<Docker Host IP>:8081/`.

This Jenkins container runs the latest release of the standard Jenkins, one that is without the Blue Ocean Plugin suite. To get Jenkins Blue Ocean on this container, you have to install the Jenkins Blue Ocean Plugin suite manually.

Setting up Blue Ocean on an Existing Jenkins Server

To get Jenkins Blue Ocean on an existing Jenkins server, you need to install the Blue Ocean Plugin suite. Let's begin.

Follow these steps to install the Blue Ocean Plugin suite:

1. From the Classic Jenkins dashboard, click on **Manage Jenkins** [1] available on the left-hand side menu. See Figure 2-11. You land on the *Manage Jenkins* page. From here, click on **Manage Plugins** [2].

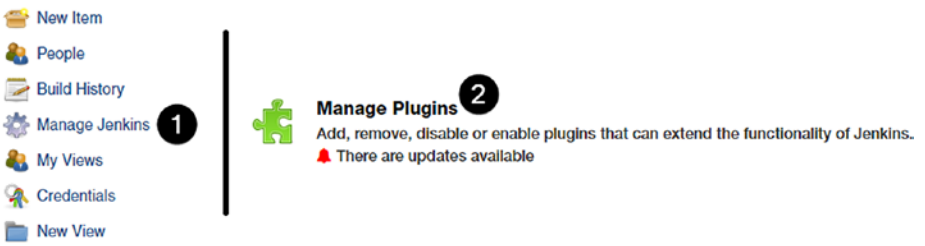

Figure 2-11. *Accessing the Manage Plugins page from the Classic Jenkins Dashboard*

2. On the Manage Plugins page, click on the **Available** tab (see Figure 2-12).

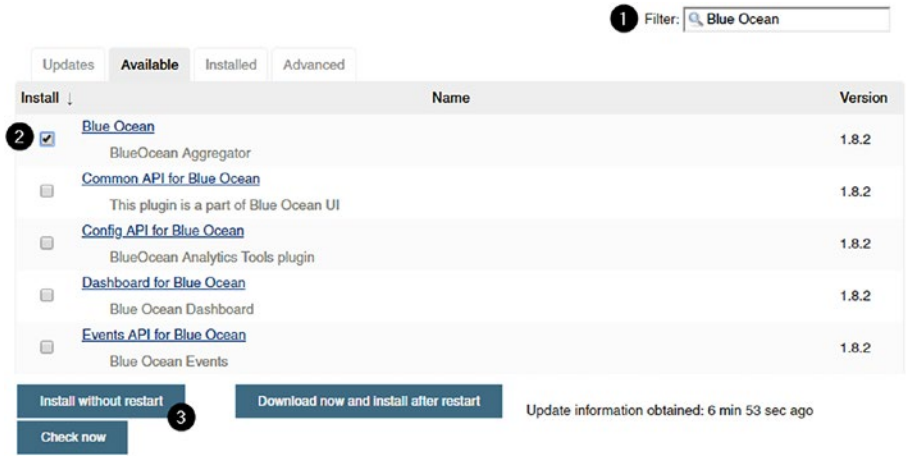

Figure 2-12. *Searching for the Blue Ocean plugin*

3. Using the field: **Filter** [1] search for *Blue Ocean*.
4. Next, From the resulting list of available plugins, choose **Blue Ocean** [2] and click on **Install without restart** [3].
5. Jenkins starts installing the Blue Ocean Plugin suite along with its dependencies (see Figure 2-13).

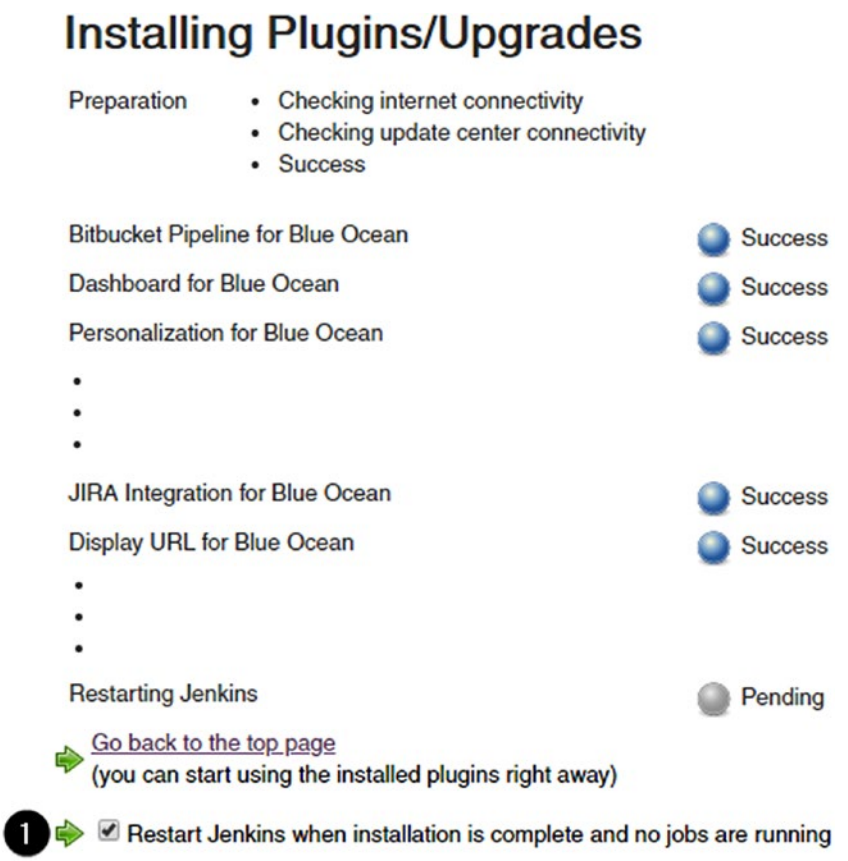

Figure 2-13. *Jenkins plugin installation in progress*

6. If possible click on Restart Jenkins when installation is complete and no jobs are running [1].
7. Otherwise restart Jenkins separately by typing the following in your browser: `<Jenkins URL>/restart`. Example: `http://192.168.10.173:8080/restart`.
8. After you restart Jenkins, log in to the dashboard (see Figure 2-14).

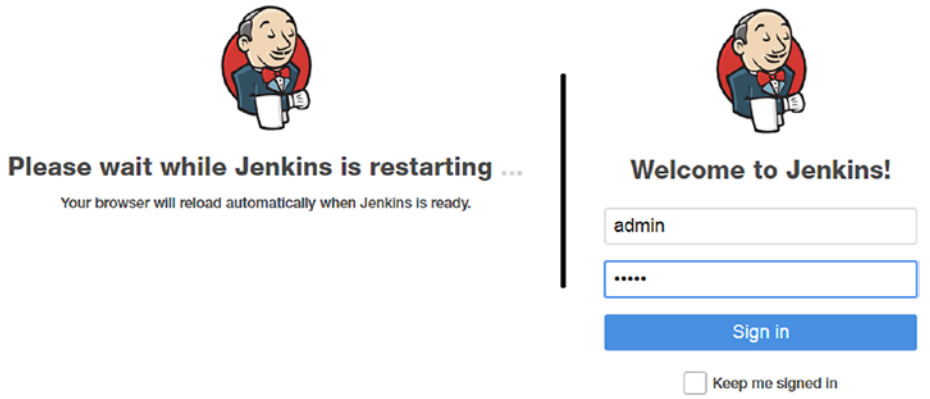

Figure 2-14. *Jenkins restarting and login page*

9. You should see the link **Open Blue Ocean** [1] on the Classic Jenkins Dashboard page (see Figure 2-15).

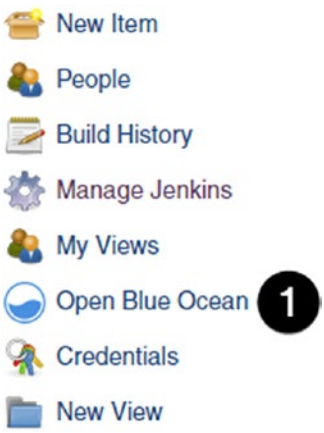

Figure 2-15. *The Jenkins Blue Ocean link on the Classic Jenkins Dashboard*

And this is how you install the Jenkins Blue Ocean Plugin suite on an existing Jenkins server.

Things to Consider

It is a common practice among users to run Jenkins behind a reverse proxy server such as Nginx or Apache. A reverse proxy server comes with many benefits. It handles sending static files to slow clients without blocking Jenkins threads. It caches static files or serves them right off the filesystem to lower the load on Jenkins. Nginx is explicitly very efficient at axing SSL and is a better choice than the default Winstone Servlet Container that comes with Jenkins.

A few others like to run Jenkins with Apache Tomcat server for other good reasons. Nevertheless, these reverse proxy servers and servlet containers can sometimes rewrite Blue Ocean URIs, which results in the following error (see Figure 2-16).

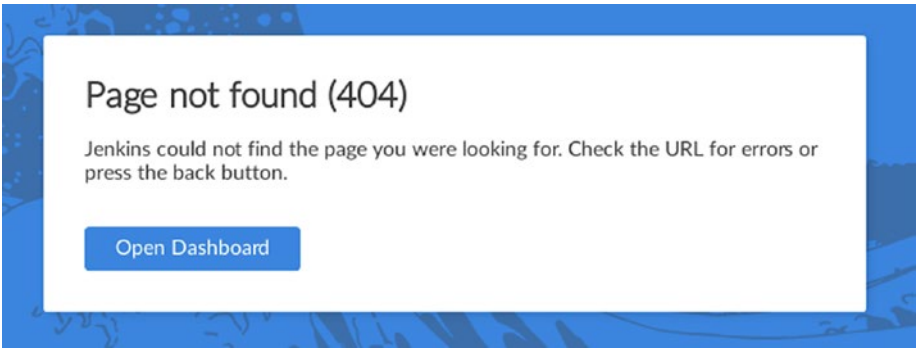

Figure 2-16. *Page not found (404) error while accessing Jenkins Blue Ocean Dashboard*

Thus, in the following section, I have included a list of settings that are a must on your reverse proxy server and servlet containers. Let's begin.

While Running Jenkins Blue Ocean Behind Apache

If you run Jenkins Blue Ocean behind an Apache reverse proxy server, make sure that you have the following configuration in place.

1. Make sure the nocanon option is present for the ProxyPass inside your Apache *.conf file

*A section from the Apache *.conf file*

```
ProxyPass         /jenkins  http://localhost:8080/ nocanon
ProxyPassReverse  /jenkins  http://localhost:8080/
ProxyRequests     Off
AllowEncodedSlashes NoDecode

# Local reverse proxy authorization override
# Most unix distribution deny proxy by default (ie /etc/
apache2/mods-enabled/proxy.conf in Ubuntu)
<Proxy http://localhost:8080/*>
  Order deny,allow
  Allow from all
</Proxy>
```

2. Make sure to have the following line: AllowEncodedSlashes NoDecode inside your Apache *.conf file.

Both the AllowEncodedSlashes NoDecode and the nocanon option to ProxyPass are essential for Blue Ocean URIs to work correctly.

These settings hold true if you run Jenkins with a reverse proxy in HTTPS.

The above configuration assumes that you are running Jenkins on port 8080 with an empty context path. Moreover, both Jenkins and Apache servers are running on the same host.

If you are running Jenkins and Apache on separate hosts, then replace `localhost/127.0.0.1` with the IP of the Jenkins host machine.

While Running Jenkins Blue Ocean Behind Nginx

If you run Jenkins Blue Ocean behind a Nginx reverse proxy server, make sure that you have the following configuration in place:

1. Make sure that you use `proxy_pass` directly to the Jenkins IP and Port.
2. If you continue to face problems with paths (especially folders) in Blue Jenkins Ocean, make sure to add the following lines inside your *.conf file.

```
If ($request_uri ~* "/blue(/.*)") {
  proxy_pass http://jenkins/blue$1;
  break;
  }
```

3. Both of the above settings are highlighted in bold in the following code. The following code is available for download separately from: https://github.com/Apress/beginning-jenkins-blue-ocean/blob/master/Ch02/default.conf.

*A typical Nginx *.conf file (complete):*

```
upstream jenkins {
  keepalive 32; # keepalive connections
  server 127.0.0.1:8080; # jenkins ip and port
}
```

```
server {
  listen        80;       # Listen on port 80 for IPv4 requests

  server_name   jenkins.example.com;

  #this is the jenkins web root directory (mentioned in the /
  etc/default/jenkins file)
  root          /var/run/jenkins/war/;

  access_log    /var/log/nginx/jenkins/access.log;
  error_log     /var/log/nginx/jenkins/error.log;
  ignore_invalid_headers off; #pass through headers from
  Jenkins which are considered invalid by Nginx server.

  location ~ "^/static/[0-9a-fA-F]{8}\/(.*)$" {
    #rewrite all static files into requests to the root
    #E.g /static/12345678/css/something.css will become /css/
    something.css
    rewrite "^/static/[0-9a-fA-F]{8}\/(.*)" /$1 last;
  }

  location /userContent {
    #have nginx handle all the static requests to the
    userContent folder files
    #note : This is the $JENKINS_HOME dir
        root /var/lib/jenkins/;
    if (!-f $request_filename){
      #this file does not exist, might be a directory or a
      /**view** url
      rewrite (.*) /$1 last;
          break;
    }
          sendfile on;
  }
```

```
  location @jenkins {
      sendfile off;
      proxy_pass         http://jenkins;
      if ($request_uri ~* "/blue(/.*)") {
        proxy_pass http://jenkins/blue$1;
        break;
      }
      proxy_redirect     default;
      proxy_http_version 1.1;

      proxy_set_header   Host              $host;
      proxy_set_header   X-Real-IP         $remote_addr;
      proxy_set_header   X-Forwarded-For   $proxy_add_x_
                                           forwarded_for;
      proxy_set_header   X-Forwarded-Proto $scheme;
      proxy_max_temp_file_size 0;

      #this is the maximum upload size
      client_max_body_size       10m;
      client_body_buffer_size    128k;
      proxy_connect_timeout      90;
      proxy_send_timeout         90;
      proxy_read_timeout         90;
      proxy_buffering            off;
      proxy_request_buffering    off; # Required for HTTP CLI
                                      commands in Jenkins > 2.54
      proxy_set_header Connection ""; # Clear for keepalive
  }

  location / {
    # Optional configuration to detect and redirect iPhones
    if ($http_user_agent ~* '(iPhone|iPod)') {
      rewrite ^/$ /view/iphone/ redirect;
```

```
    }

    try_files $uri @jenkins;
  }
}
```

This configuration assumes that you are running Jenkins on port 8080 with an empty context path. Moreover, both Jenkins and Nginx servers are running on the same host.

If you are running Jenkins and Apache on separate hosts, then replace `localhost/127.0.0.1` with the IP of the Jenkins host machine.

RUN JENKINS BLUE OCEAN BEHIND A REVERSE PROXY

Following is an exercise to run Jenkins Blue Ocean behind Nginx.

Before proceeding with the exercise make sure that you delete all existing Jenkins containers and their corresponding docker volumes.

---

1. Run a Jenkins container with a name *jenkins*. Use `sudo` wherever necessary.

   ```
   docker run -d --name jenkins \
   -v jenkins_home:/var/jenkins_home \
   jenkinsci/blueocean
   ```

In the above docker command, note that the Jenkins container ports 8080 and 50000 are left unmapped with their corresponding host ports. The reason is that we do not want to expose Jenkins to the host IP.

---

2. Download the docker image for Nginx.

```
docker pull nginx
```

3. Spawn a Docker container for Nginx. Also, link the *nginx* container to the *jenkins* container using the `--link` option.

```
docker run -d --name nginx -p 80:80 --link jenkins nginx
```

4. Get inside the Nginx container using the `docker exec` command.

```
docker exec -it nginx /bin/bash
```

5. Update the Ubuntu package lists.

```
apt-get update
```

6. Install the nano text editor.

```
apt-get install nano
```

7. Take the backup of the *default.conf* file inside */etc/nginx/conf.d/*.

```
cp etc/nginx/econf.d/default.conf etc/nginx/econf.d/
default.conf.backup
```

8. Next, replace the content of the *default.conf* file with the following.

```
upstream jenkins {
  server jenkins:8080;
}

server {

    listen 80;
    server_name jenkins.example.com;

    location / {
      proxy_pass              http://jenkins;
      proxy_set_header        Host $host;
```

```
        proxy_set_header        X-Real-IP $remote_addr;
        proxy_set_header        X-Forwarded-For $proxy_
        add_x_forwarded_for;
        proxy_set_header        X-Forwarded-Proto $scheme;
      }
  }
```

The above configuration assumes that you are running Jenkins on port 8080 with an empty context path.

---

9. Exit the Nginx container.

   ```
   exit
   ```

10. Restart the Nginx container.

    ```
    docker restart nginx
    ```

If everything goes well, Jenkins will be available on the following URL: `http://<DOCKER HOST IP>/`. However, you should fail to access Jenkins on `http://<Docker Host IP>:8080/`.

While Running Jenkins Blue Ocean with Apache Tomcat

If you use Apache Tomcat to run Jenkins, make sure that you have the following configuration inside Apache Tomcat:

```
-Dorg.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH=true
-Dorg.apache.catalina.connector.CoyoteAdapter.ALLOW_BACKSLASH=true
```

Summary

In the preceding chapter, you learned to set up Jenkins Blue Ocean with docker using volumes. You also saw the Jenkins Setup Wizard. Running Jenkins with docker is easy and helpful, and it's evident from the current chapter.

Next, you learned to set up Jenkins Blue Ocean on an existing Jenkins server.

If you wish to set up a production Blue Ocean instance using docker, it's recommended to use the *jenkins/jenkins* docker image. The *jenkins/jenkins* docker image comes without Blue Ocean, and to get Blue Ocean you have to install the Blue Ocean Plugin suite manually. Doing so gives you the freedom to choose a specific Jenkins release with a particular version of the Blue Ocean Plugin.

Lastly, you learned some critical configuration for the reverse proxy server and the Apache Tomcat server to make them work with Jenkins Blue Ocean.

If you have followed all the exercises in the current chapter, then you know the following bonus topics:

- Jenkins disaster recovery
- Spawning a Classic Jenkins server with docker
- Running Jenkins Blue Ocean behind a reverse proxy

In the next chapter, you'll learn to create and visualize pipelines in Jenkins Blue Ocean and more.

CHAPTER 3

Creating Your First Pipeline

The current chapter is a step-by-step guide that teaches you to work with Jenkins Blue Ocean. It begins with the Pipeline Creation Wizard that includes integrating your Blue Ocean pipeline projects with the various types of source code repositories, followed by creating a pipeline using the Visual Pipeline Editor.

You'll learn and practice both of these activities in detail using an example maven project, which you can fork from the following GitHub repository: https://github.com/Apress/beginning-jenkins-blue-ocean/tree/master/Ch03/example-maven-project.

Next, you learn to edit an existing pipeline using the Visual Pipeline Editor. Along with this, you'll learn about the multibranch pipeline feature in Jenkins Blue Ocean. You'll also see how to deal with pull requests in Jenkins Blue Ocean. The chapter also demonstrates using the various features in Jenkins Blue Ocean related to the Pipeline Visualization.

The current chapter is enormous and full of screenshots. It's one big exercise that you have to follow sequentially. Following is a list of topics that you'll learn in the current chapter:

- Integrate a pipeline project in Jenkins Blue Ocean with a source code repository that's on:
  - Git server

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5_3

    - GitHub server
    - Bitbucket server
    - GitLab server
- Design and create a pipeline using the Visual Pipeline Editor.
- Learn about the pipeline visualization.
- Trace and debug logs at the step, stage, and pipeline level.
- View testing results and artifacts for each pipeline run.
- Learn to edit an existing Blue Ocean pipeline using the Visual Pipeline Editor.
- Learn to view the pipelines for multiple branches of your pipeline project.
- Learn to work with pull requests in Jenkins Blue Ocean.

Along with this, you'll also learn to use docker to spawn on-demand containers that serve as Jenkins agents for your pipelines. You'll use the docker image `nikhilpathania/jenkins-ssh-agent`, *which is* available on the Docker Hub.

Prerequisites

Before you start with the real thing, it's essential to set up all the necessary prerequisites that are required to follow the steps described in the current chapter.

In the following section, you'll download the docker image required by Jenkins to spawn containers that perform builds, and you'll learn to install and configure the docker plugin in Jenkins.

Pulling the Docker Image for Jenkins Agent

You'll need a docker image that allows Jenkins to spawn on-demand docker containers that serve as Jenkins agents. For this purpose, I have created a docker image (`nikhilpathania/jenkins_ssh_agent`) that you can download from the Docker Hub.

To download the docker image, log in to your docker host and execute the following command. Use sudo wherever necessary.

```
docker pull nikhilpathania/jenkins_ssh_agent
```

The docker image `nikhilpathania/jenkins_ssh_agent` is based on Ubuntu and comes with *Git*, *Java JDK*, *Maven*, and *sshd* installed. The image also contains a user account named *jenkins*.

You can run a container using the image `nikhilpathania/jenkins_ssh_agent` by executing the following command to test it:

```
docker run -it --name jenkins_agent nikhilpathania/jenkins_ssh_
agent /bin/bash
```

Creating Credentials for the Docker Image in Jenkins

It is necessary to add credentials inside Jenkins that allow it to interact with the docker image `nikhilpathania/jenkins_ssh_agent`. Credentials stored inside Jenkins using the Credentials feature are available across all your pipelines. Nevertheless, it's also possible to add additional control over its usage, such that the credentials are available only for a particular pipeline or a user. So let's add the necessary credentials for the docker image using the below steps:

1. From the Classic Jenkins dashboard, navigate to **Credentials ➤ System ➤ Global credentials (unrestricted)**. Or directly access the following link: `http://<Jenkins URL>/credentials/store/system/domain/_/` from your browser.

2. Next, try to add a new credential by clicking on the **Add Credentials** link (see Figure 3-1).

Figure 3-1. *Adding a new credentials to Jenkins*

3. Jenkins presents you with a couple of options; configure them exactly as shown in Figure 3-2.

[1] **Kind**: This should be **Username with Password.**

[2] **Username**: This is the username to interact with the docker image: `nikhilpathania/jenkins_ssh_agent. It should be jenkins.`

[3] **Password**: This is the password for the docker image `nikhilpathania/jenkins_ssh_agent. It should be jenkins.`

[4] **ID**: You can add a meaningful name to recognize these credentials.

[5] **Description**: You can add a meaningful description for these credentials.

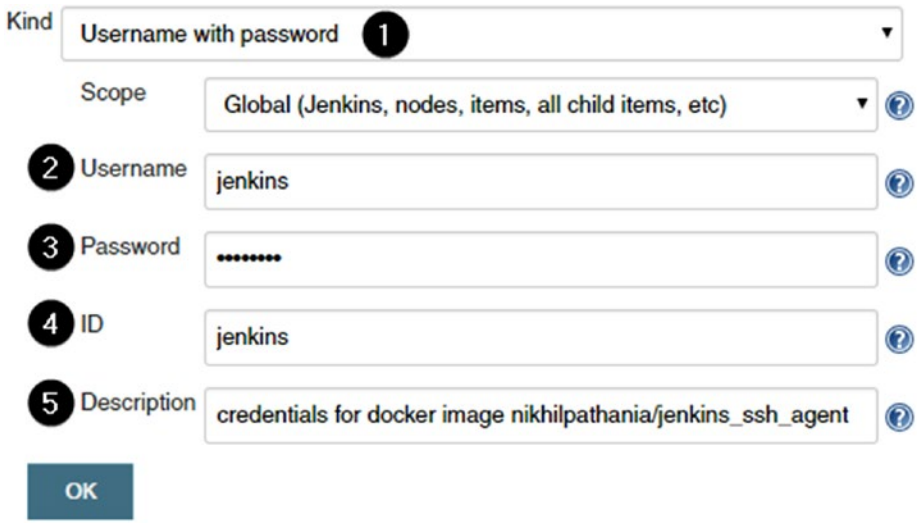

Figure 3-2. *Configuring the credential*

Once you finish configuring, click on the **OK** button. Now, you have successfully created credentials of the type **Username with Password** in Jenkins.

Installing the Docker Plugin

To spawn on-demand docker containers serving as Jenkins agents, you'll need to install the docker plugin for Jenkins. To install the docker plugin for Jenkins, follow the below steps:

1. Click on the **Manage Jenkins** link available on the Classic Jenkins Dashboard. Alternatively, you can also click on the **Administration** link from the Jenkins Blue Ocean Dashboard.
2. Next, from the **Manage Jenkins** page, click **Manage Plugins** (see Figure 3-3).

Figure 3-3. *The Jenkins Manage Plugins link*

3. On the **Manage Plugins** page, click on the **Available** tab and type *Docker* in the **Filter field** [1]. From the list of available plugins choose the **Docker plugin** [2] and click on **Install without restart** button [3] (see Figure 3-4).

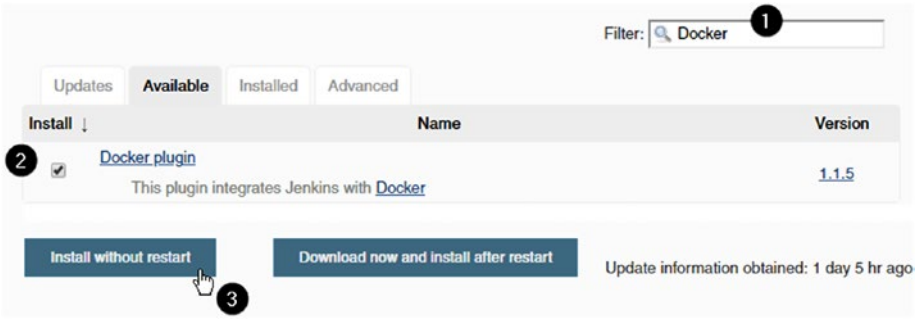

Figure 3-4. *Installing the Docker Plugin for Jenkins*

Jenkins begins the plugin installation. Once the installation finishes, you may choose to restart Jenkins by selecting the option **Restart Jenkins when installation is complete and no jobs are running,** or you can start using the installed plugin right away by clicking the option **Go back to the top page**.

Configuring the Docker Plugin

Next, let's try to configure the newly installed docker plugin. To do so, from the **Manage Jenkins** page, click on the **Configure System** link (see Figure 3-5).

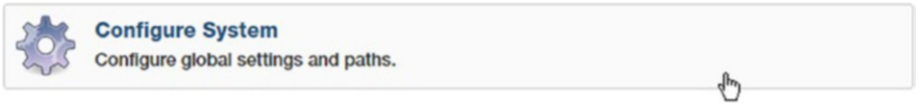

Figure 3-5. *Link to configure Jenkins global settings and paths*

On the **Configure System** page, scroll all the way down to the **Cloud** section. You'll now try to connect a docker host with your Jenkins server. To do so, click on **Add a new cloud** button and choose **Docker** (see Figure 3-6).

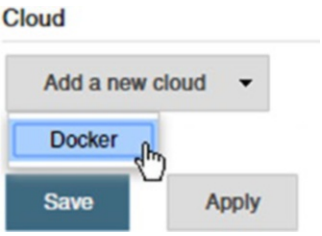

Figure 3-6. *Adding a new cloud of the type Docker*

Configuring the Docker Cloud Details

You are required to provide a unique name for your docker cloud configuration; thus add a meaningful name in the **Name** field [1]. To configure a docker host with Jenkins, click on the **Docker Cloud details...** button (see Figure 3-7).

Figure 3-7. *Adding a name to your Docker cloud*

You'll see a list of options to configure (see Figure 3-8). Following is an explanation of the configurations:

[1] **Docker Host URI**: This is the URI used by Jenkins to talk to the docker host. Paste your docker host URI in the field provided. Refer the *Appendix section: Setting up a Docker host*, to get your docker host URI.

[2] **Server Credentials**: If your docker host requires a login, you need to add the credentials to Jenkins using the **Add** button. However, do nothing if you are using a docker host that's running your Jenkins server container.

[3] **Test Connection**: Click on this to test the communication between your Jenkins server and the docker host. You should see the docker version and the API version [4] if the connection is successful.

[4] **Enabled**: A checkbox to enable/disable the current configuration.

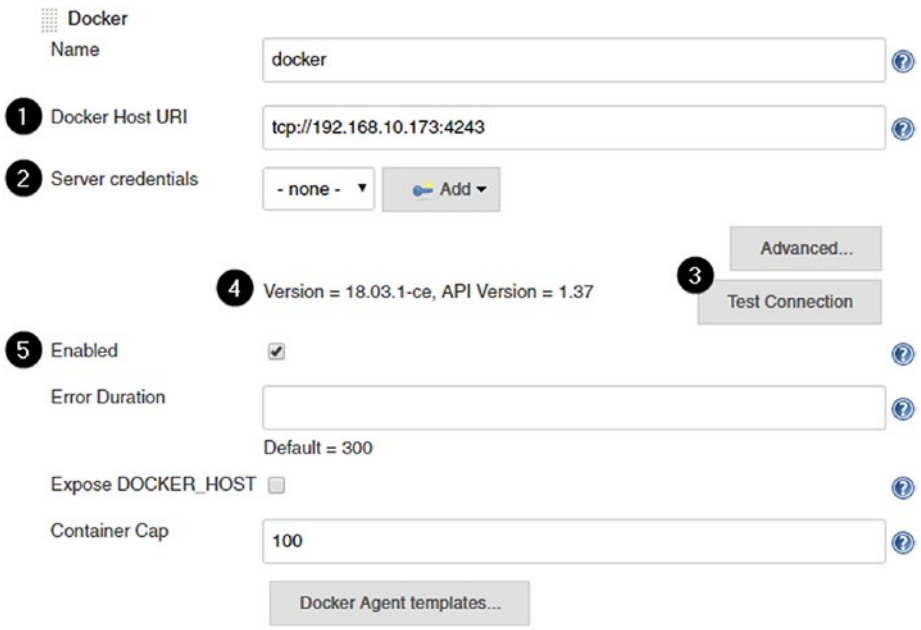

Figure 3-8. *Configuring the Docker host URI and testing the connection*

Configuring the Docker Agent Template

Next, switch back to your Jenkins' **Configure System** page, and click on the **Docker Agent templates...** button (see Figure 3-8).

Next, on the same page, you'll see, the **Add Docker Template** button (see Figure 3-9). Click on it to configure the docker image that Jenkins should use to spawn container.

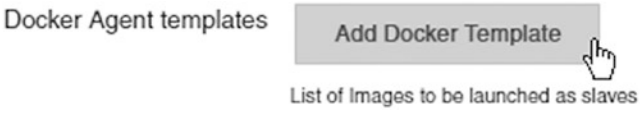

Figure 3-9. *Adding a new Docker agent template*

You'll get many options to configure. I have divided the configurations into two figures for your convenience: Figure 3-10a, and Figure 3-10b.

Under the section, Docker Agent Template, configure the options exactly as shown in Figures 3-10a and 3-10b:

[1] **Labels**: The label that you type in under the **Labels** field gets used inside your Pipeline to define agents for your stages. In this way, Jenkins knows that it has to use docker to spawn agents.

[2] **Enabled**: This checkbox is used to enable/disable the current configuration.

[3] **Docker Image**: Add the name of the docker image that should be used to spawn agents containers.

[4] **Remote File System Root**: This is the directory inside the container that holds the workspace of the Pipeline that runs inside it.

[5] **Usage**: We would like only to build pipelines that have the right agent label, in our case it is `docker`.

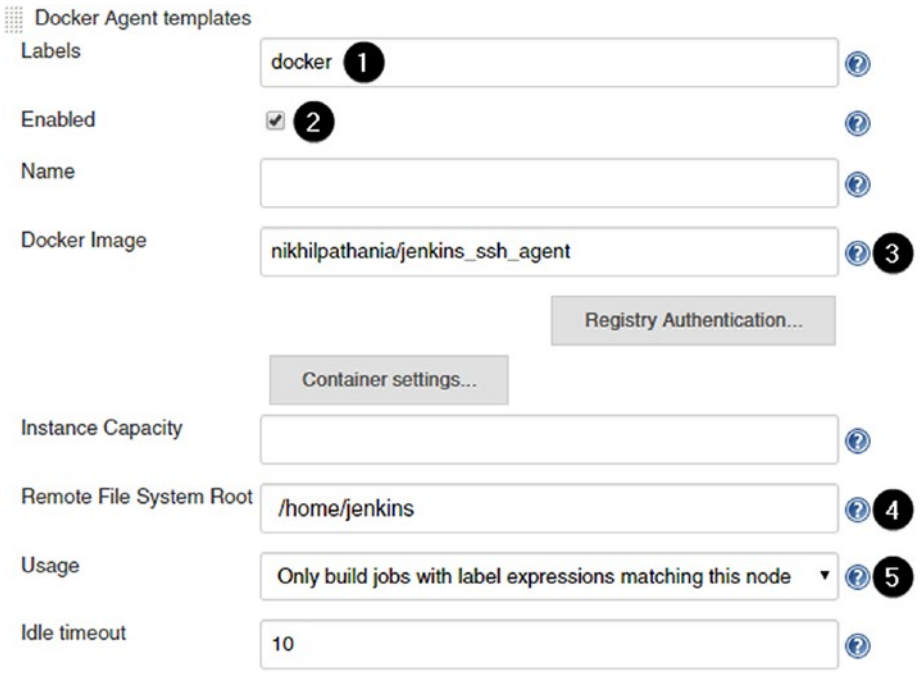

Figure 3-10a. *Configuring the Docker agent template*

[6] **Connect method**: Choose to **Connect with SSH** option to allow Jenkins to connect with the container using the SSH protocol.

[7] **SSH Key**: Choose **use configured SSH credentials** from the options to use the SSH credentials as the preferred mode of authentication.

[8] **SSH Credentials**: From the list of options choose the credentials that you have created earlier, in the section: *Creating Credentials for the Docker Image in Jenkins*.

[9] **Host Key Verification Strategy**: Choose **Non verifying Verification Strategy** to keep things simple. However, this is not the recommended setting for a production Jenkins server.

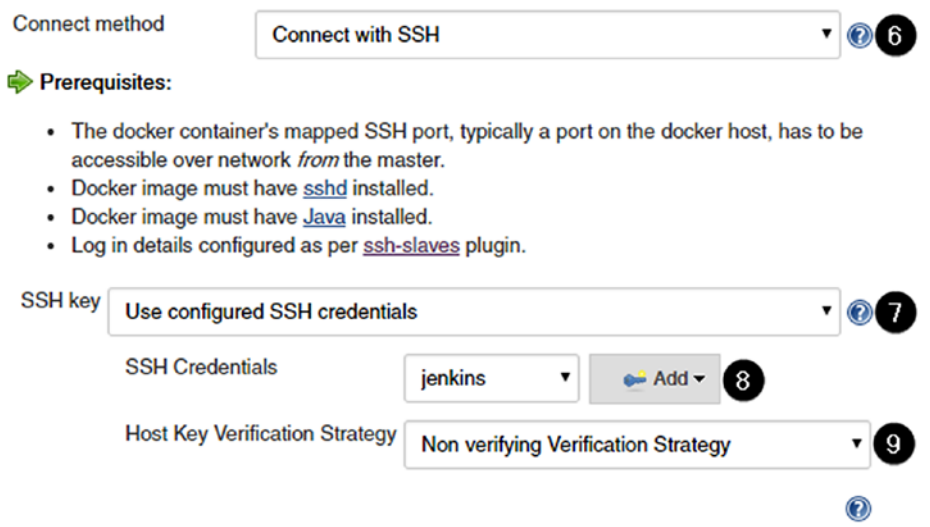

Figure 3-10b. *Defining how Jenkins should communicate with the Docker container*

Once you finish making the configurations, scroll all the way down to the bottom and click on **Apply** and then **Save** to save the settings.

What you saw is the basic Docker Plugin configuration you are required to do to allow Jenkins to spawn containers on demand during the pipeline run.

Using the Pipeline Creation Wizard

In this section and the sections that follow, you'll learn to create a Pipeline in Jenkins Blue Ocean using the Pipeline Creation Wizard.

To begin with creating a Pipeline, from the Jenkins Blue Ocean dashboard, click on the **New Pipeline** button [1] (see Figure 3-11).

Figure 3-11. *Creating a new pipeline*

The *Pipeline Creation Wizard* begins, and the first thing it asks you is, "Where do you store your code?" (see Figure 3-12).

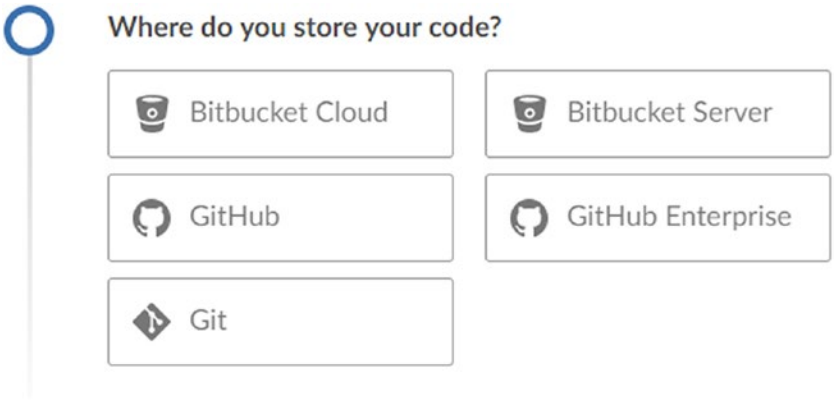

Figure 3-12. *List of source control tools*

You can choose from Bitbucket, GitHub, Git, and GitLab. For GitLab, you do not see an option. However, it's very much possible to connect to it; you'll see that shortly.

So, let's see how to connect and create a Jenkins Blue Ocean Pipeline for each of these repositories.

Integrating Blue Ocean Pipeline with a Git Repository

A few organizations still prefer using a standard, on-premises instance of a Git server. If your team is one among them, then this section shows you how to connect Blue Ocean with a repository hosted on such a Git server.

Select **Git** as an answer to "Where do you store your code?" (see Figure 3-13).

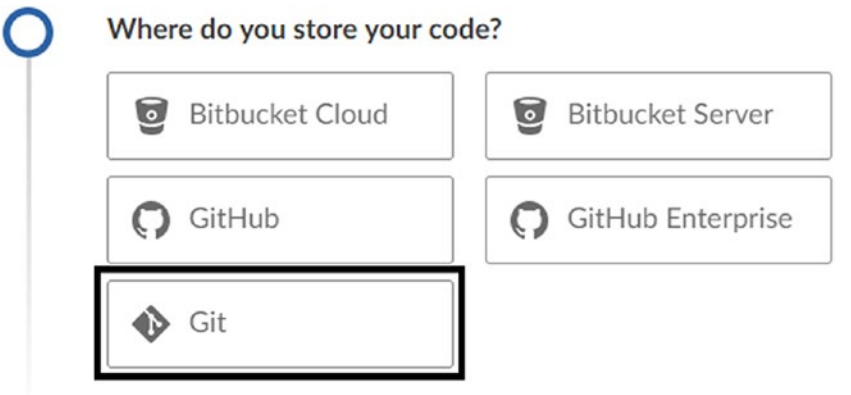

Figure 3-13. *Integrating Blue Ocean with Git*

You'll then be asked to provide a repository URL. Paste the HTTP/HTTPS or the SSH repository URL of your repository. The authentication methods, however, are different for both. Let's see them one by one.

HTTP/HTTPS Authentication Method

If the mode of authentication configured on your Git server is HTTP/HTTPS, then you can type in your repository URL (HTTP/HTTPS) under the **Repository URL** field [1] (see Figure 3-14).

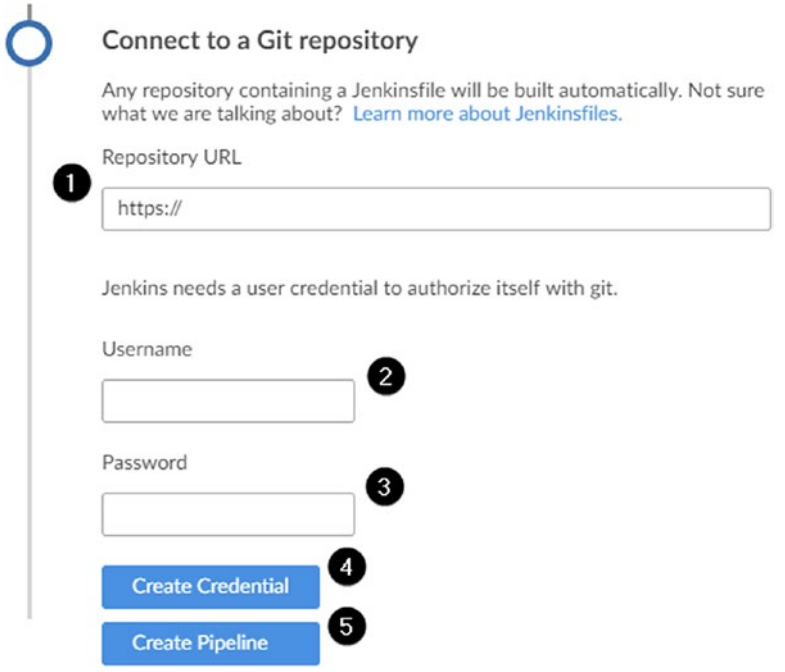

Figure 3-14. *the HTTP/HTTPS authentication method*

Add your credentials in the **Username** [2] and **Password** [3] fields, respectively, and click on the button **Create Credential** [4]. Doing so saves your credentials inside Jenkins. And, if everything goes well, you can click on the button **Create Pipeline** [5] to proceed with creating a Pipeline for your source code repository.

SSH Authentication Method

If the authentication method on your Git server is SSH, then pasting the SSH URL of your repository inside the **Repository URL** field [1] generates a **public SSH key** [2] (see Figure 3-15).

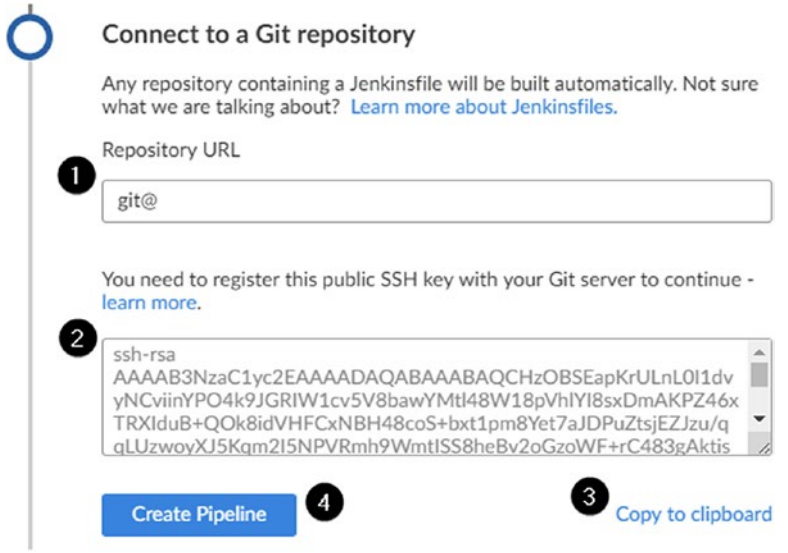

Figure 3-15. *Generating a public SSH key*

You are required to configure this public SSH key on your Git server, and following are the steps:

1. Log in to your Git Server, and create a new `.pub` file under the `/tmp` folder using your favorite text editor.

   ```
   nano /tmp/id_rsa.peter.pub
   ```

2. Copy the public SSH key generated on the Jenkins Blue Ocean window by clicking on the **Copy to clipboard** [3] link.

3. Paste the copied public SSH key to the new `.pub` file.

4. Next, append the content of the newly created `.pub` file to the git user's `authorized_keys` file that's present inside the `~/.ssh/` directory.

   ```
   cat /tmp/id_rsa.peter.pub >> ~/.ssh/authorized_keys
   ```

Once you have configured the public SSH key generated by Blue Ocean inside the Git server, you can click on the **Create Pipeline** button [4] to proceed with creating a Pipeline for your source code repository.

If running a Git server on-premises is your thing, then there are other better alternatives. GitLab is one such tool that's worth trying. It's a Git-based version control system. Setting up GitLab using one of its Omnibus package installations is a better substitute for running a bare-metal Git server.

If you wish, you can skip the next few sections and continue with *Using the Visual Pipeline Editor.*

Integrating Blue Ocean Pipeline with a GitHub Repository

Integrating Blue Ocean with GitHub is simple, and all you need is a GitHub personal access token. In the following section, you'll learn to connect Blue Ocean with a repository hosted on GitHub.

To begin, select **GitHub** as an answer to "Where do you store your code?" (see Figure 3-16).

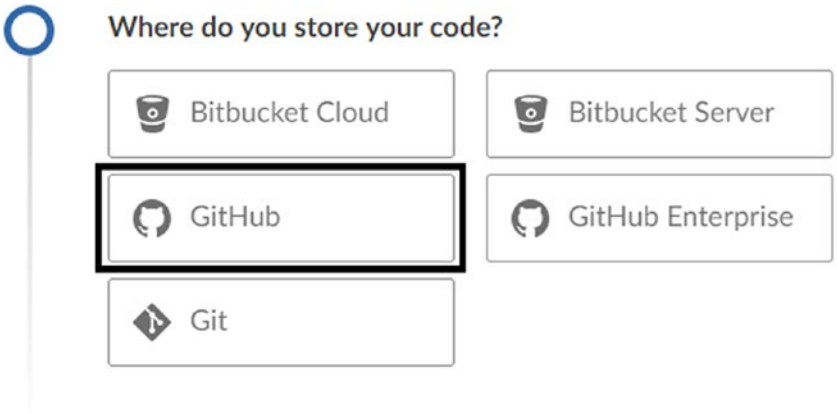

Figure 3-16. *Integrating Blue Ocean with GitHub*

Jenkins asks you to provide your GitHub access token. With GitHub, you do not paste your repository URL, as Jenkins allows you to choose between all your repositories on GitHub, right from the *Pipeline Creation Wizard*.

Find your GitHub access token and paste it into the **text field** [1] (see Figure 3-17). To connect Blue Ocean with GitHub, click on the **Connect** button [2].

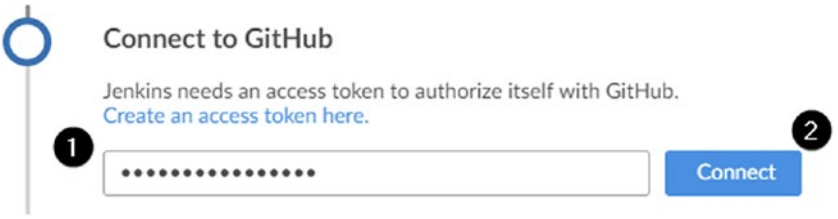

Figure 3-17. *Connecting to GitHub using token*

Getting a GitHub personal access token is simple, even if you have never used one. Click on the **Create an access token here** link, as shown in Figure 3-17. It takes you to the *Personal Access Token* page on your GitHub account, where you can generate a token to serve the purpose.

However, for new users, I recommend the more extended way of doing it. Let's see how to create a personal access token on GitHub.

Creating a Personal Access Token

Log in to your GitHub account. Next, from the menu bar, click on your user account, available on the right side. From the resultant drop-down menu, click on **Settings** (see Figure 3-18).

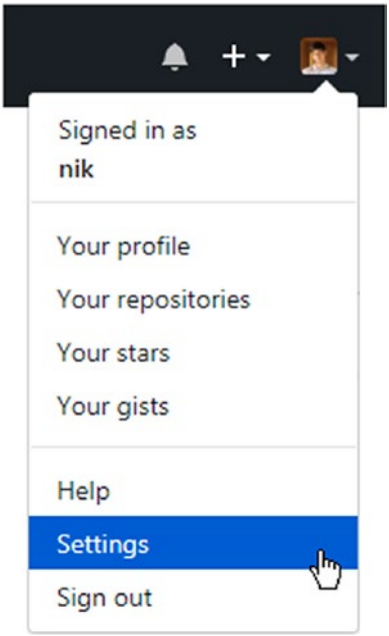

Figure 3-18. *Link to the user settings in GitHub*

This takes you to your **Personal settings** page. On this page, from the **Personal settings** sidebar, click on **Developer settings**. On the resultant page, click on **Personal access tokens** from the left sidebar menu.

If this is the first time you are creating a personal access token, you'll see something as shown in Figure 3-19.

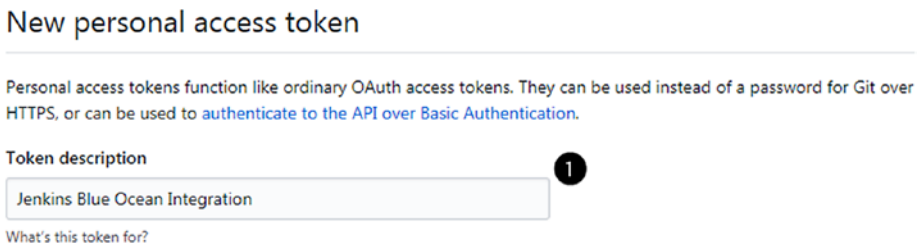

Figure 3-19. *Adding a description about your new personal access token*

Under the **Token description** field [1], add a meaningful name for your new token that you are going to create.

On the same page, under the **Select scopes** section, make sure the required scopes are defined as shown in Figure 3-20.

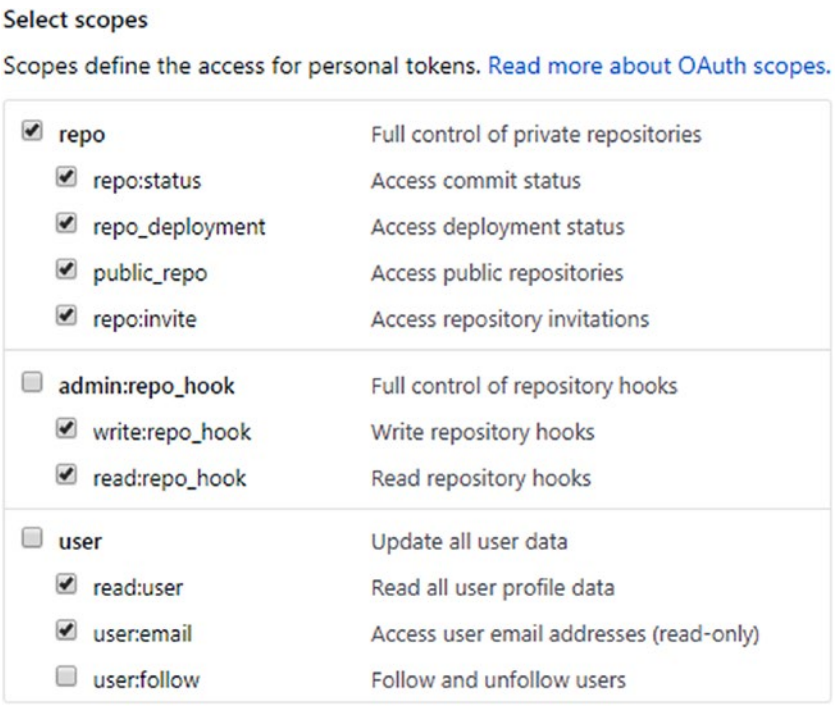

Figure 3-20. *Defining the scope of your personal access token*

The aforementioned scopes are necessary for Jenkins Blue Ocean to work correctly with GitHub. Once, you have defined the necessary scopes for your new token, click on the **Generate token** button at the bottom to generate the token (not shown in Figure 3-20).

You should now see a newly created token, as shown in Figure 3-21. Click on the **link** [1] to copy the token to the clipboard. You can also revoke the current token by clicking the **Delete** button [2]. Use the **Revoke all** button [3] to delete all tokens that you have ever created.

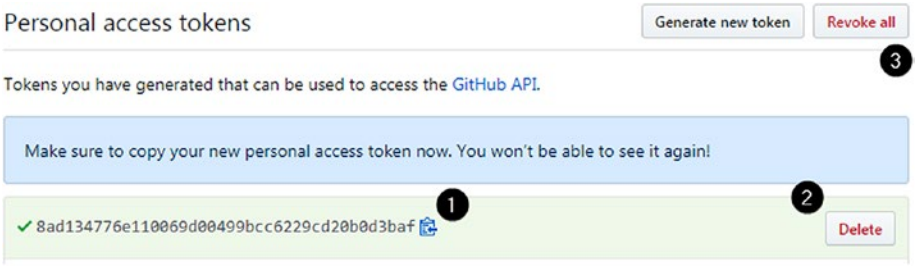

Figure 3-21. *Your newly generated personal access token*

Selecting the GitHub Repository

After posting your GitHub Personal Access token, Jenkins tries to connect to your GitHub account. It then presents to you all the organizations that it finds on your GitHub account. You'll have to choose one of them [1] (see Figure 3-22).

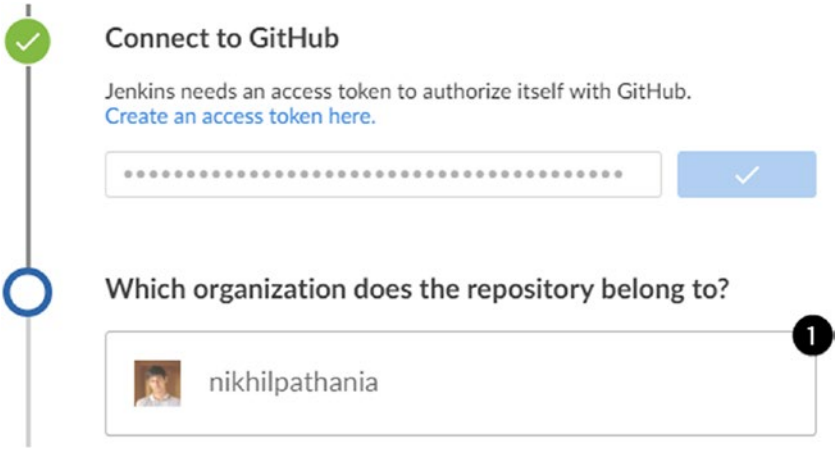

Figure 3-22. *Selecting your GitHub organization*

Once you select the organization that you want, Jenkins presents you with all the repositories that it finds inside the selected organization (see Figure 3-23).

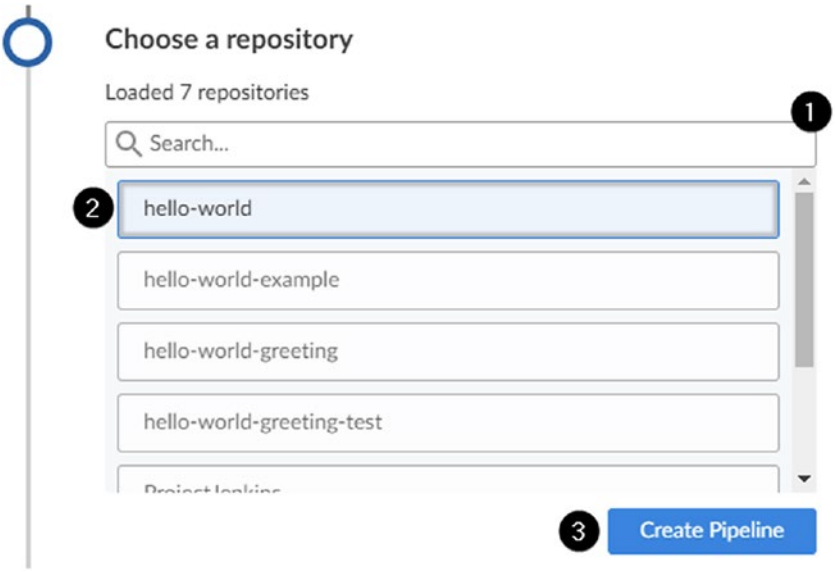

Figure 3-23. *Selecting your GitHub repository*

You can search for your desired repository using the **Search field** [1]. When you find the repository that you are looking for, select it [2]. Click on the **Create Pipeline** button [3] to proceed with creating a Pipeline.

If you wish, you can skip the next few sections and continue with *Using the Visual Pipeline Editor.*

Integrating Blue Ocean Pipeline with a Bitbucket repository

Integrating Blue Ocean with Bitbucket is as simple as it is with GitHub. In the following section, you'll learn to connect Blue Ocean with a repository hosted on Bitbucket.

To begin, select **Bitbucket Cloud** as an answer to *Where do you store your code?* (see Figure 3-24).

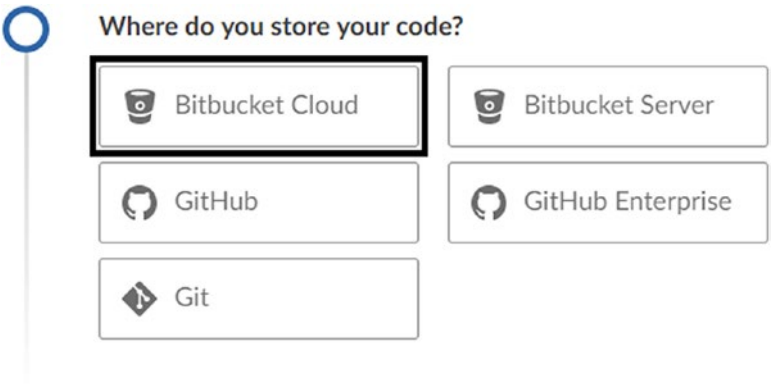

Figure 3-24. *Integrating Blue Ocean with Bitbucket*

Next, Jenkins asks you to provide your Bitbucket credentials (see Figure 3-25). Add your Bitbucket credentials under the **Username** [1] and **Password** [2] fields, respectively. To connect Blue Ocean with Bitbucket, click on the **Connect** button [3].

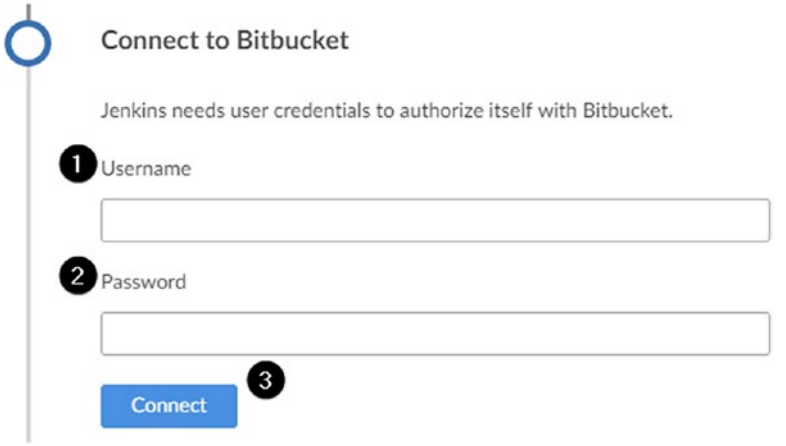

Figure 3-25. *Connecting to your Bitbucket account*

Selecting the Bitbucket Repository

When you click on Connect, Jenkins tries to connect to your Bitbucket account. It then presents to you all the teams that it finds on your Bitbucket account. You'll have to choose one of them [1] (see Figure 3-26).

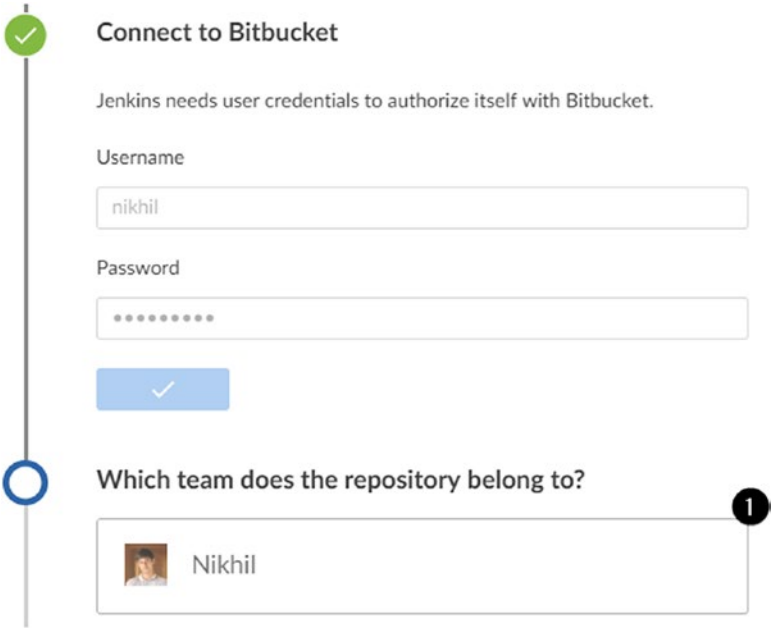

Figure 3-26. *Selecting your Bitbucket team*

Once you select the team that you want, Jenkins presents you with all the repositories that it finds inside the selected team (see Figure 3-27).

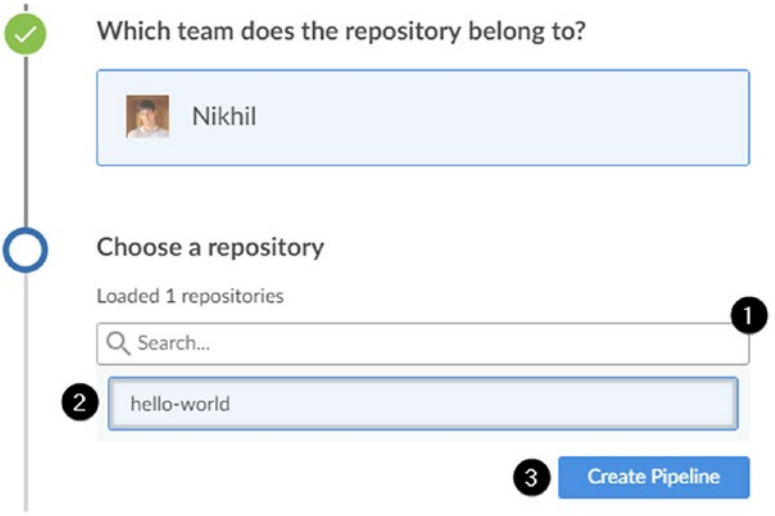

Figure 3-27. *Selecting your Bitbucket repository*

You can search for your desired repository using the **Search field** [1]. When you find the repository that you are looking for, select it [2]. Click on the **Create Pipeline** button [3] to proceed with creating a Pipeline.

If you wish, you can skip the next few sections and continue with *Using the Visual Pipeline Editor.*

Integrating Blue Ocean Pipeline with a GitLab Repository

The steps to integrate your Blue Ocean Pipeline with a GitLab repository are similar to the one that we saw in the section *Integrating Blue Ocean with a Git Server*. However, with GitLab the process is a bit easy and intuitive.

When using GitLab server (cloud or on-premise), both the authentication methods, HTTP/HTTPS and SSH, are available by default, and there is no configuration needed from you on the GitLab server. Also, you select Git as the answer to *Where do you store your code?* There is no particular option for GitLab yet in Blue Ocean (see Figure 3-28).

Select **Git** as an answer to *Where do you store your code?* (see Figure 3-28).

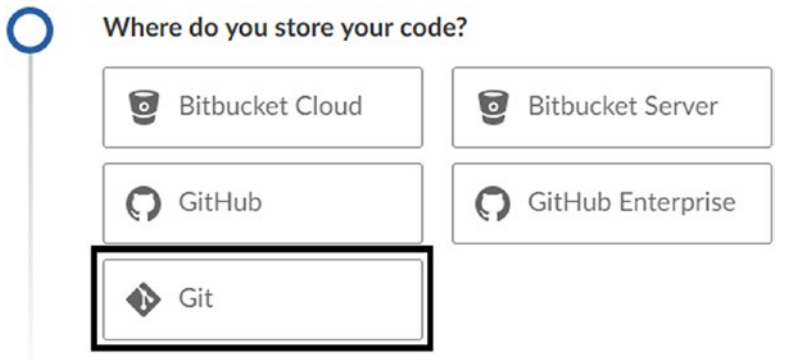

Figure 3-28. *Integrating Blue Ocean with GitLab*

Jenkins asks you to provide a repository URL. Paste the HTTP/HTTPS or the SSH repository URL of your repository. The authentication methods, however, are different for both. Let's see them one by one.

HTTP/HTTPS Authentication Method

To use the HTTP/HTTPS authentication method, simply type in your repository URL (HTTP/HTTPS) under the **Repository URL** field [1] (see Figure 3-29).

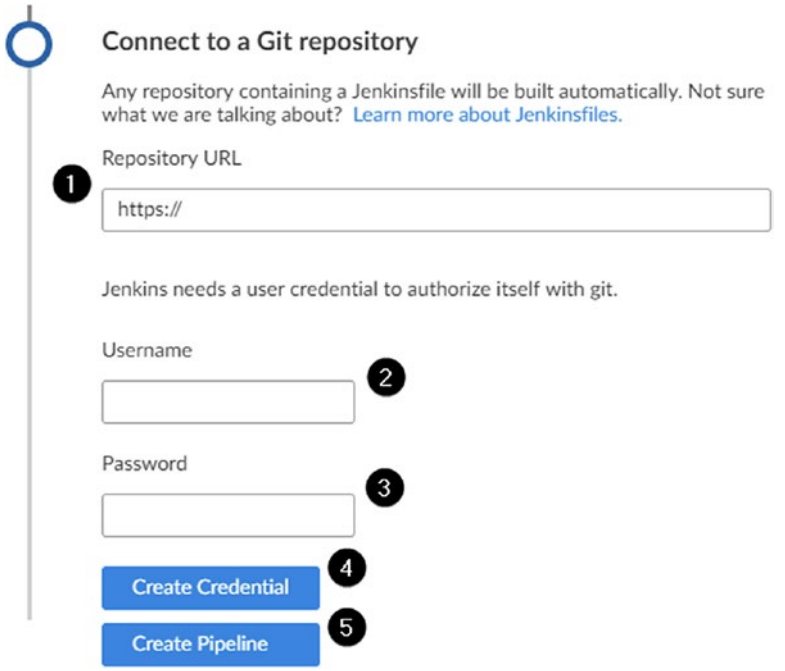

Figure 3-29. *the HTTP/HTTPS authentication method*

Add your GitLab credentials in the **Username** [2] and **Password** [3] fields, respectively, and click on the button **Create Credential** [4]. Doing so saves your credentials inside Jenkins. If everything goes well, you can click on the button **Create Pipeline** [5] to proceed with creating a Pipeline for your source code repository.

SSH Authentication Method

To use the SSH authentication method, simply type in your SSH repository URL inside the **Repository URL** field [1]. This generates a **public SSH key** [2]. See Figure 3-30.

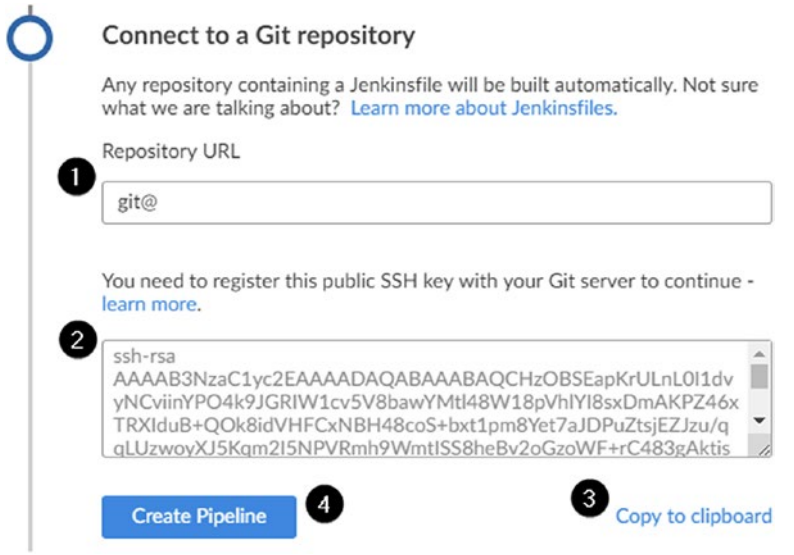

Figure 3-30. *Generating a public SSH key*

You are required to configure the public SSH key on your GitLab server. Follow these steps to do so:

1. Log in to your GitLab server.
2. From the menu bar at the top, click on your user account, and from the resultant drop-down menu, click on **Settings** (see Figure 3-31).

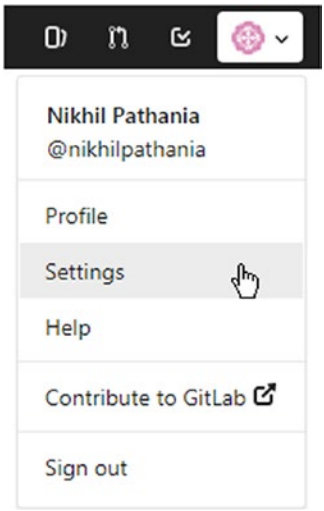

Figure 3-31. *Link to the user settings in GitLab*

3. On the resultant **User Settings** page, from the left-hand sidebar, click **SSH Keys**. Over here, you get the options to add your public SSH key to the GitLab server (see Figure 3-32).

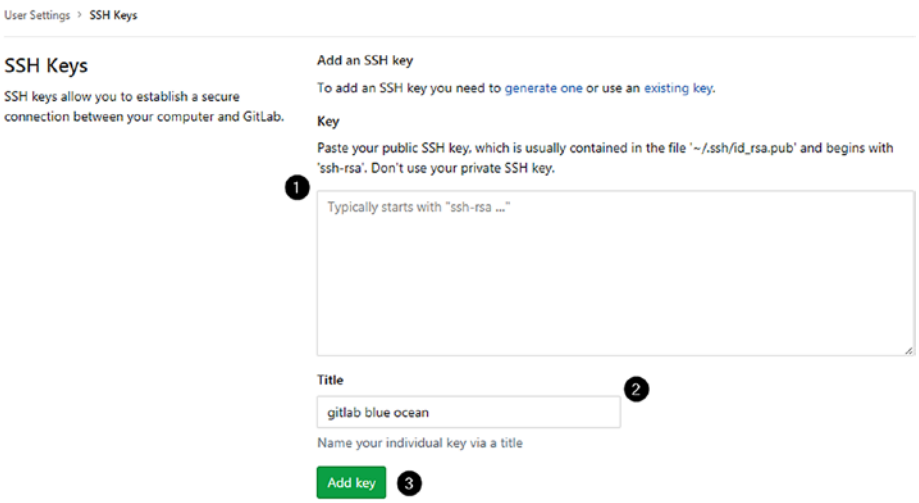

Figure 3-32. *Adding a public SSH key to GitLab*

Paste the public SSH key that you generated on the Jenkins Blue Ocean inside the **text** field [1]. Add a meaningful title to the SSH key using the **Title** field [2]. Click on the **Add key** button to save the SSH key.

Figure 3-33 demonstrates how your saved public SSH key looks.

Figure 3-33. *Your saved public SSH key on GitLab*

Once you have configured the public SSH key generated by Blue Ocean inside the GitLab server, you can click on the **Create Pipeline** button [4] (see Figure 3-30) to proceed with creating a Pipeline for your source code repository.

If you wish, you can skip the next section and continue with *Using the Visual Pipeline Editor.*

Viewing the Saved Credentials for your Repository in Jenkins

In the previous sections, during the process of connecting Jenkins Blue Ocean with various repositories, you provided the credentials of your GitHub/Git/GitLab/Bitbucket account to Jenkins.

In the back end, Jenkins saves all your credentials under your user account. To view/edit the saved credentials, navigate to the Classic Jenkins Dashboard, and on the top-right corner, click the drop-down menu available beside your **user account [1].** Then, click on **Credentials [2]** (see Figure 3-34). You can find all your saved credentials on this page.

Figure 3-34. *Viewing the saved credentials for your repository in Jenkins*

Using the Visual Pipeline Editor

In the previous sections, you learned to configure Jenkins Blue Ocean with various types of source code repositories.

That was the first part of the Pipeline Creation Wizard. The remaining part is using the Visual Pipeline Editor to design your Pipeline. And, that's exactly what you'll learn in the current section.

To let you know in advance, you'll create a Jenkins Blue Ocean Pipeline that does the following:

- Downloads the source code from the GitHub repository
- Performs a build and some testing
- Publishes the testing results under the Pipeline's Tests page
- Uploads the built artifact to Jenkins Blue Ocean

Since the Visual Pipeline Editor is a UI, in the current section, you'll find lots of screenshots, at least one for every step that you are going to perform using it. This is to help you learn how to interact with the Visual Pipeline Editor.

So let's begin with the step where you took a break, at the clicking of the **Create Pipeline** button.

Assigning a Global Agent

The Pipeline that you are going to create should have two stages, and each stage is supposed to run inside a docker container. You'll define the agents for each stage separately in the stage's settings.

Therefore, let's keep the Pipeline's global agent setting to none. To do so, select **none** from the options available for the **Agent** field (see Figure 3-35).

Figure 3-35. *Assigning a global agent*

As for today, using the Visual Pipeline Editor, you can only define agents and environment variables at the global level of your Pipeline.

Your pipeline code so far:

```
pipeline {
    agent none
}
```

Creating a Build & Test Stage

Let's create a stage by clicking the plus icon, as shown in Figure 3-36.

Figure 3-36. *Adding a Build & Test stage*

Type in the name **Build & Test** [1] for your stage (see Figure 3-37). You'll notice that the visualization on the right side changes while you make a change on the configuration panel present on the left side.

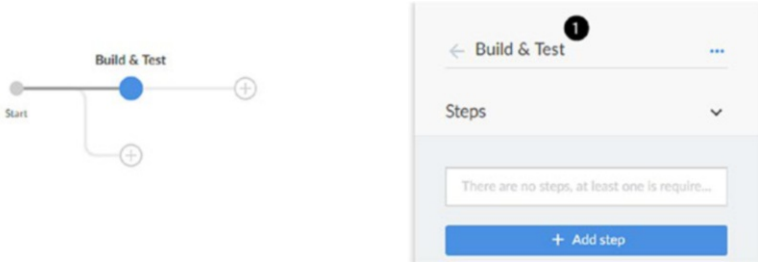

Figure 3-37. *Naming your stage*

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {

        }
    }
}
```

Adding Steps

Let's add some steps to our **Build & Test** stage. To add a step, click on the **+ Add step** button (see Figure 3-38).

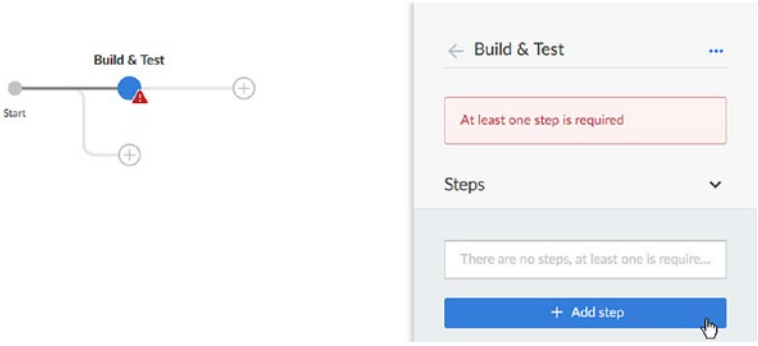

Figure 3-38. *Adding a new step*

On the configuration panel, you'll be present with a list of steps [3] that are available for Jenkins Blue Ocean. The list keeps growing, as you install more new Blue Ocean compatible plugins (see Figure 3-39).

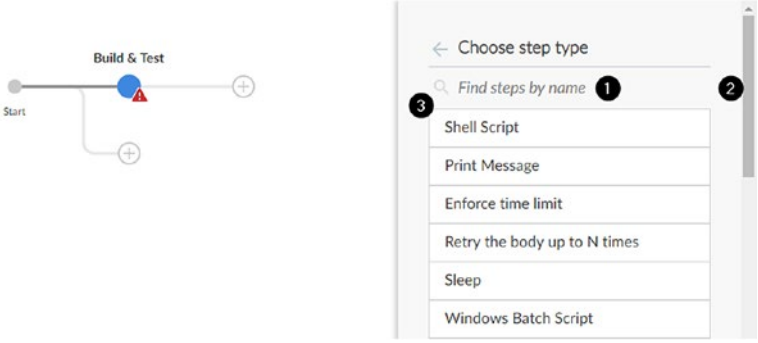

Figure 3-39. *List of available steps*

There is a search field [1] that allows you to search for a given step. Use the scroll bar [2] to scroll through the list of available steps.

Adding a Shell Script Step

Let's add a step that executes a shell script. Our source code is a Maven project, and we would like to build and test it using an mvn command, which eventually gets executed inside a shell on the Jenkins agent.

To do so, look for the step **Shell Script** [3] by searching it using the search field [1] (see Figure 3-40). Click it once you find it.

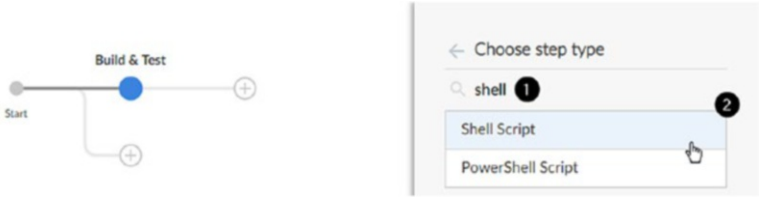

Figure 3-40. *Adding a Shell Script step*

You'll see a text field where you can paste your shell commands (see Figure 3-41). Paste the below code inside the text field [1], which is a maven command to build, test, and create a package out of your source code.

```
mvn -Dmaven.test.failure.ignore clean package
```

Figure 3-41. *Configuring a Shell Script step*

Next, click on the back arrow [2] to come out of the current configuration.

You should see a step **Shell Script** under the **Steps** section (see Figure 3-42).

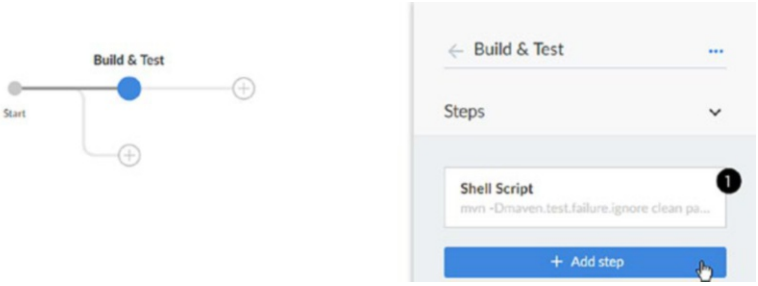

Figure 3-42. *The list of steps for a stage*

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
            }
        }
    }
}
```

Next, You'll add another step to stash the built package and the testing report generated by the maven command. To do so by clicking on the + **Add step** button see Figure 3-42.

Adding a Stash Step to Pass Artifact Between Stages

In Jenkins, you can stash artifacts of your choice to pass between stages. In the current section, you are going to create a second step inside the Build & Test stage that stashes the built package and the testing report. This stash will be used later in the subsequent stages of our Jenkins Blue Ocean Pipeline.

To do so, look for the step **Stash some files to be used later in the build** [2] by searching it using the search field [1] (see Figure 3-43). Click it once you find it.

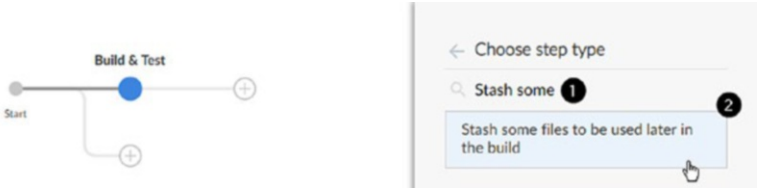

Figure 3-43. *Adding a Stash step*

The configuration for the step **Stash some files to be used later in the build** looks something like that shown in Figure 3-44.

Add the name "build-test-artifacts" for your stash using the Name* field [1], which is mandatory. Add the following to the Includes field [2]: **/target/surefire-reports/TEST-*.xml,target/*.jar.

With this configuration you are telling Jenkins to stash any .jar file (built package) from the target directory, and the TEST-*.xml file(test report) from the **/target/surefire-reports/ directory on the build agent.

Next, click on the back arrow [3] to come out of the current configuration.

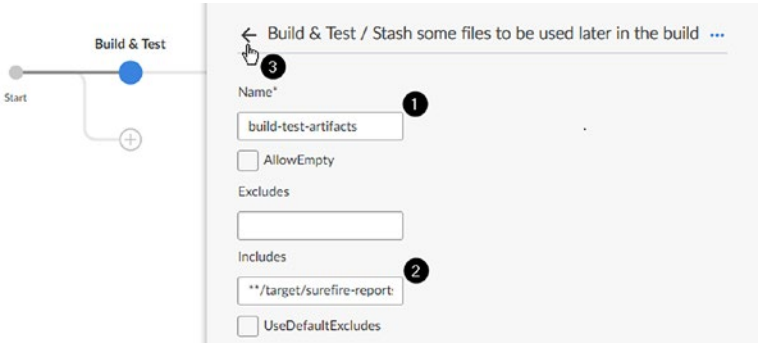

Figure 3-44. *Configuring a Stash step*

You should see a new step, **Stash some files to be used later in the build,** under the **Steps** section [1] as shown in Figure 3-45.

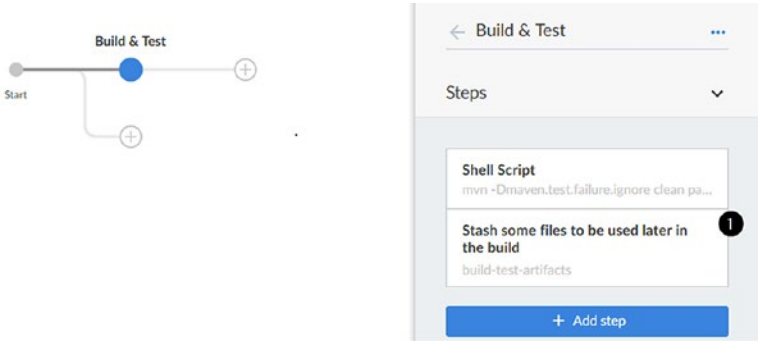

Figure 3-45. *The list of steps for a stage*

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
    }
}
```

Assigning an Agent for the Build & Test Stage

Next, you'll assign a build agent for your **Build & Test** stage. The agent is going to be a docker container that will be spawn automatically by Jenkins. Once the stage is complete, Jenkins will destroy the container.

To do so, expand the **Settings** section [1] (see Figure 3-46). You'll see a few options to configure. Choose **node** as the agent type [2], and type in **docker** for the **Label*** field [3]. Contract the **Settings** section [1] afterward.

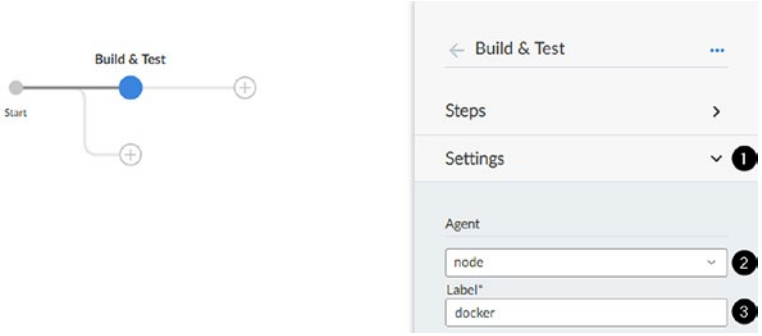

Figure 3-46. *Assigning an agent to a stage*

With the following configuration, Jenkins looks for an agent with the label **docker**. Remember the section from the current chapter, wherein you configured the Docker Plugin in Jenkins. You specified the label **docker** while configuring the **Docker Agent Template**. Refer to Figure 3-10a.

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            agent {
                node {
                    label 'docker'
```

```
                }
            }
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
    }
}
```

Creating a Report & Publish Stage

Let's add another stage named **Report & Publish** that will publish the testing results on the **Tests** page of the Pipeline and that will publish the built package on the **Artifacts** page of the Pipeline.

To do so, click on the plus icon, as shown in Figure 3-47.

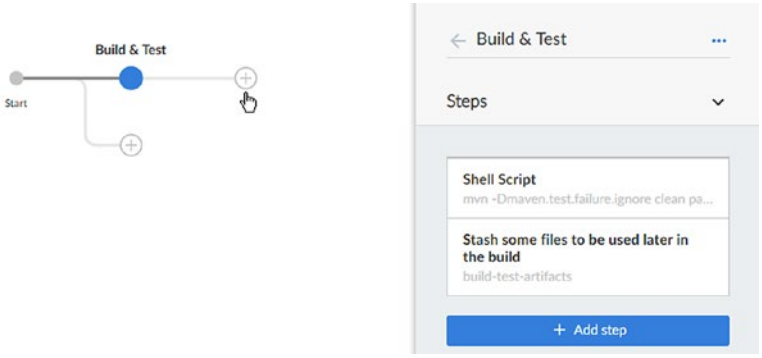

Figure 3-47. *Creating a Report & Publish stage*

Type in the name **Report & Publish** [1] for your new stage (see Figure 3-48).

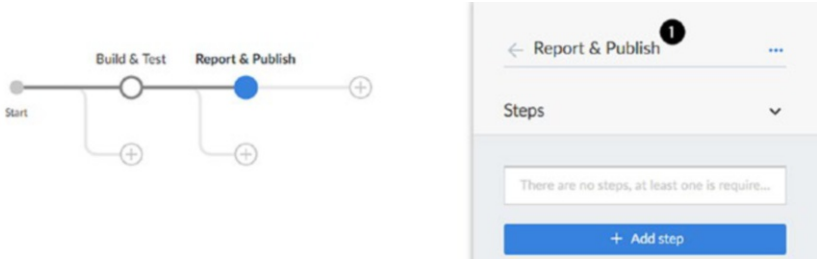

Figure 3-48. *Naming your stage*

Let's add a few steps to your new stage **Report & Publish**.

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            agent {
                node {
                    label 'docker'
                }
            }
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
        stage('Build & Test') {

        }
    }
}
```

Adding an Un-Stash Step

Before we do anything in the **Report & Publish** stage, it is first crucial to un-stash the files that were stashed in the previous stage. So let's add a step to un-stash a stash from the previous stage.

To add a step, click on the + **Add step** button (see Figure 3-49).

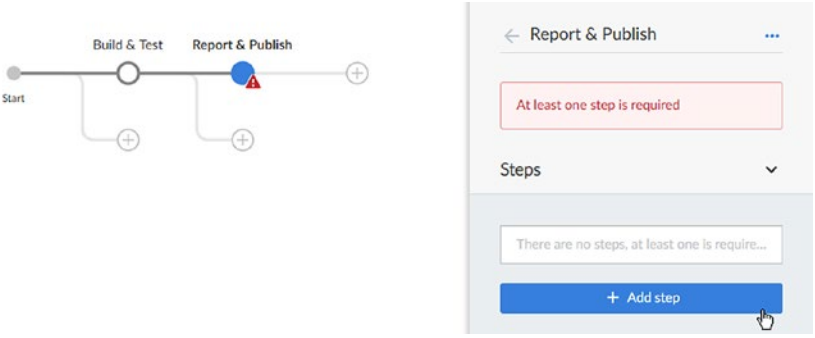

Figure 3-49. *Adding a Restore files previously stashed step*

Next, look for the step **Restore files previously stashed** [2] by searching it using the search field [1] (see Figure 3-50). Click it once you find it.

Figure 3-50. *Adding a Restore files previously stashed step*

You'll see a text field [1] **Name*** where you should paste the name of your stash precisely as it was defined during its creation (see Figure 3-51).

Figure 3-51. *Configuring the Restore files previously stashed step*

Next, click on the back arrow [2] to come out of the current configuration.

You should see a step, **Restore files previously stashed [1]**, under the **Steps** section, as shown in Figure 3-52.

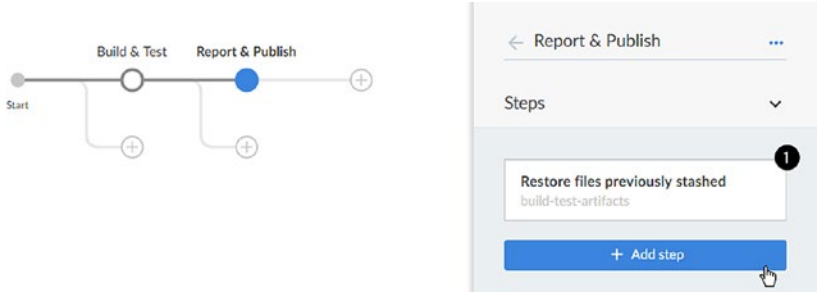

Figure 3-52. *The list of steps for a stage*

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            agent {
                node {
```

```
                    label 'docker'
                }
            }
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
        stage('Build & Test') {
            steps {
                unstash 'build-test-artifacts'
        }
    }
}
```

Report Testing Results

You now have added the necessary step to un-stash the required files from the previous stage, **Build & Test**, into the current stage.

The stash contains a JUnit test results **.xml** file that you'll publish on the pipeline's Tests page. For this, we need to add a step named **Archive Junit-formatted test results**.

To do so, click on the **+ Add step** button (see Figure 3-52). Next, look for the step **Archive Junit-formatted test results** [2] by searching it using the search field [1] (see Figure 3-53). Click it once you find it.

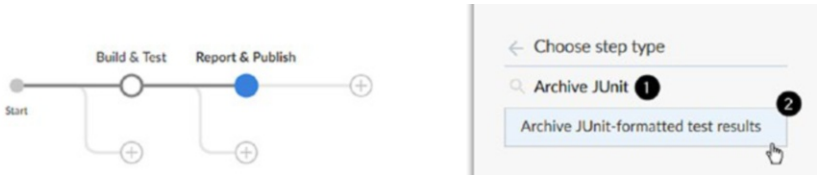

Figure 3-53. *Adding an Archive Junit-formatted test results step*

You'll be presented with a few options to configure (see Figure 3-54). Use the **TestResults*** field [1] to provide Jenkins with the path to your JUnit test result file. In our case, it is `**/target/surefire-reports/TEST-*.xml`. Leave the rest of the options to their default values, and click on the back arrow [2] to come out of the current configuration.

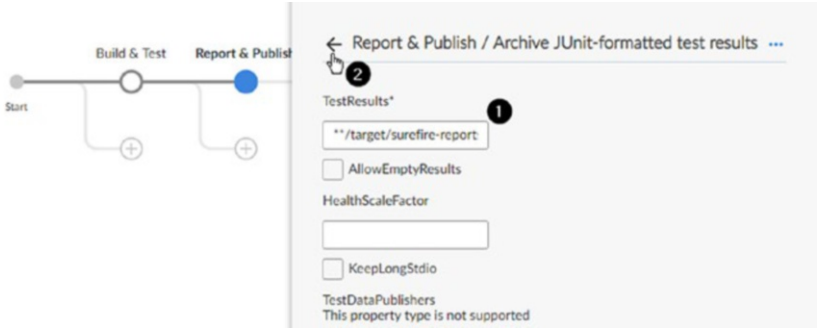

Figure 3-54. *Configuring an Archive Junit-formatted test results step*

Notice that the stashing and un-stashing activity retains the stashed file's path.

You should now see a step **Archive Junit-formatted test results [1]**, under the **Steps** section (shown in Figure 3-55).

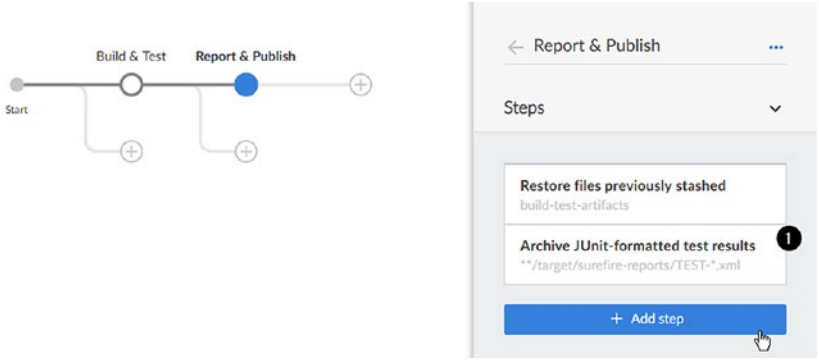

Figure 3-55. *The list of steps for a stage*

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            agent {
                node {
                    label 'docker'
                }
            }
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
        stage('Build & Test') {
            steps {
```

```
                unstash 'build-test-artifacts'
                junit '**/target/surefire-reports/TEST-*.xml'
        }
    }
}
```

Upload Artifacts to Blue Ocean

Next, let's add a step that will upload the built package to the Pipeline's Artifacts page. From the un-stashed files, you also have a **.jar** file that is the built package.

To upload it to the Pipeline Artifacts page, you'll need to use the **Archive the artifacts** step.

To do so, click on the + **Add step** button (see Figure 3-55). Next, look for the step **Archive the artifacts** [2] by searching it using the search field [1] (see Figure 3-56). Click it once you find it.

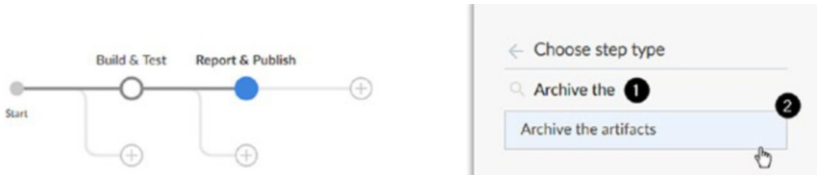

Figure 3-56. *Adding a Archive the Artifacts step*

You'll be presented with a few options to configure (see Figure 3-57). Use the **Artifacts*** field [1] to provide Jenkins with the path to your built package file. In our case, it is `target/*.jar`. Also, tick the option **OnlyIfSuccessful** [2] to upload the artifacts only if the Pipeline status is green or yellow.

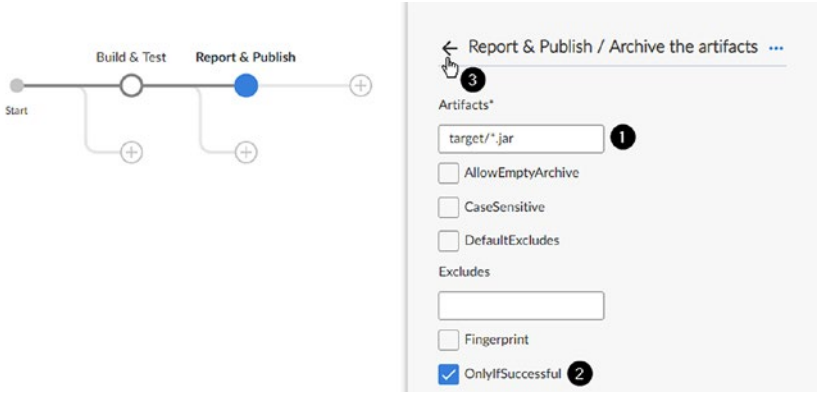

Figure 3-57. *Configuring the Archive the Artifacts step*

Leave the rest of the options to their default values, and click on the back arrow [3] to come out of the current configuration.

You should now see a step, **Archive the artifacts** [1], under the **Steps** section, as shown in Figure 3-58.

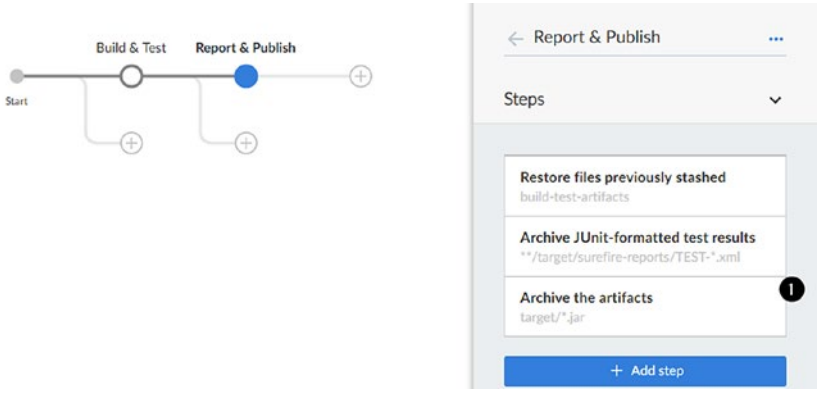

Figure 3-58. *The list of steps for a stage*

Your pipeline code so far:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            agent {
                node {
                    label 'docker'
                }
            }
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
        stage('Build & Test') {
            steps {
                unstash 'build-test-artifacts'
                junit '**/target/surefire-reports/TEST-*.xml'
                archiveArtifacts(artifacts: 'target/*.jar', \
                onlyIfSuccessful: true)
        }
    }
}
```

Assigning an Agent for the Report & Publish Stage

Next, you'll assign a build agent for your **Report & Publish** stage. The agent is going to be a docker container that will be spawn automatically by Jenkins. Once the stage is complete, Jenkins will destroy the container.

To do so, expand the **Settings** section [1] (see Figure 3-59). You'll see a few options to configure. Choose **node** as the agent type [2], and type in **docker** for the **Label*** field [3]. Contract the **Settings** section [1] afterward.

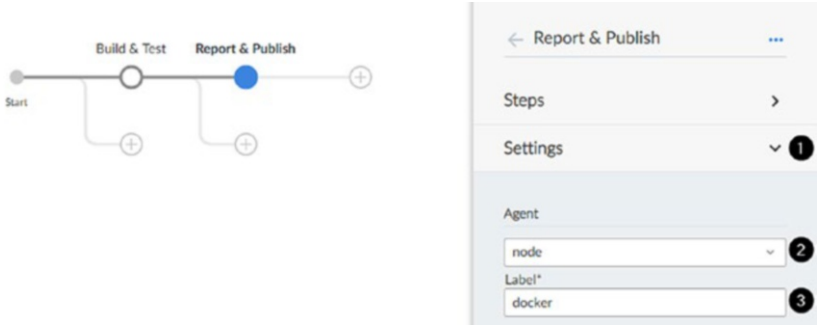

Figure 3-59. *Assigning an agent to a stage*

Your final pipeline code:

```
pipeline {
    agent none
    stages {
        stage('Build & Test') {
            agent {
                node {
                    label 'docker'
                }
            }
            steps {
                sh 'mvn -Dmaven.test.failure.ignore clean package'
```

```
                stash(name: 'build-test-artifacts', \
                includes: '**/target/surefire-reports/TEST-*.
                xml,target/*.jar')
            }
        }
        stage('Build & Test') {
            agent {
                node {
                    label 'docker'
                }
            }
            steps {
                unstash 'build-test-artifacts'
                junit '**/target/surefire-reports/TEST-*.xml'
                archiveArtifacts(artifacts: 'target/*.jar', \
                onlyIfSuccessful: true)
        }
    }
}
```

You are now done with creating the Pipeline. To save the changes, click on the **Save** button (see Figure 3-60).

Figure 3-60. *Save button to save your pipeline changes*

When you click on the **Save** button, Jenkins in the back end converts your UI configurations into a Jenkinsfile that follows the Declarative Pipeline Syntax.

Next, you'll see a pop-up window asking you to check in the new Jenkinsfile to the source code repository (see Figure 3-61). Add a meaningful comment in the **Description** field [1], which is your commit message. Select **Commit to master** [2], which means you are making a change on the master branch of your repository. Moreover, to finally make a commit, click on the **Save & Run** button, which will also eventually trigger a Pipeline in Jenkins Blue Ocean.

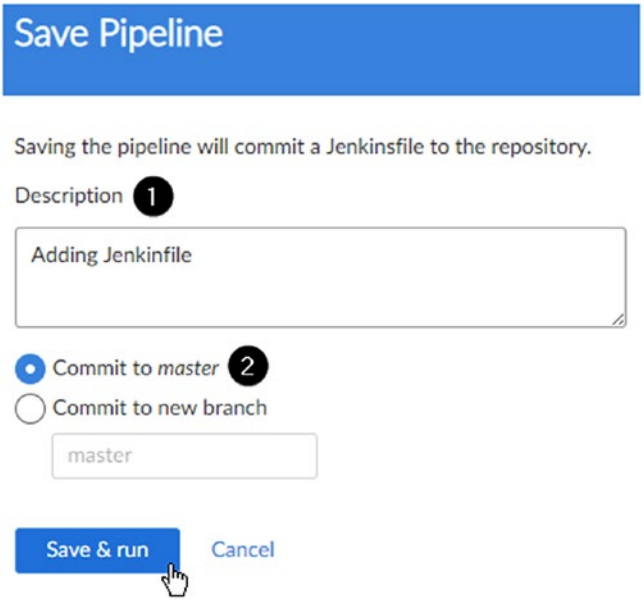

Figure 3-61. *Committing your pipeline configurations changes*

Using the Pipeline Visualization

The Pipeline Visualization feature in Jenkins Blue Ocean allows you to view a Pipeline's running status, its logs, the changes that triggered it, the test results and artifacts (if any), and, in the end, its overall status.

When the Pipeline has finished running, you can scan through the Pipeline logs at the Pipeline, stage, or step level. Compared to Classic Jenkins, where you just get a huge amount of logs, segregation of stages and steps in the Pipeline Visualization makes it much more convenient to browse through what has happened.

The current section is all about visualizing a pipeline run. Here, you'll take a closer look at the elements of Pipeline Visualization in Jenkins Blue Ocean. So let's begin.

Canceling a Running Pipeline

Let's continue from where you left off. By clicking on the **Save & Run** button, Jenkins saves the Jenkinsfile to your source code repository and also triggers a Pipeline.

You'll notice that you have directly moved to the project dashboard. See Figure 3-62, which demonstrates a running pipeline. I am running a pipeline for the GitHub repository **example-maven-project** [1].

The project's **Activity** page [6] lists all its builds (running, finished and canceled) along with their status [2], build number [3], shortened commit hash [4], a message about what's going on [5], and other useful information.

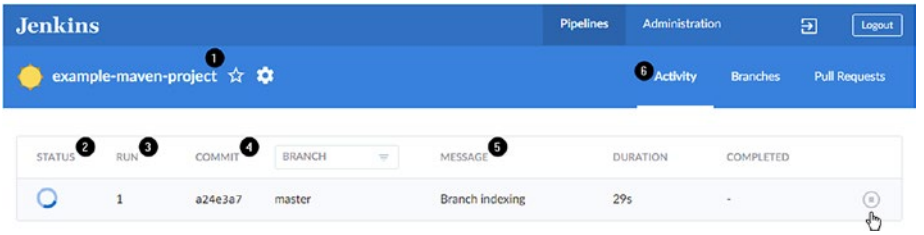

Figure 3-62. *Canceling a running pipeline in Blue Ocean*

To stop a running build (pipeline run), use the stop button, as shown in Figure 3-62.

Re-Running a Pipeline

The example-maven-project is deliberately designed to randomly fail a few tests every now and then. Unluckily my first pipeline run failed some tests, and that is why you see a yellow exclamation mark [1] (see Figure 3-63). You'll also notice that the project status has turned a bit cloudly [3].

In this case, I would like to re-run my pipeline, with a hope that the next run passes all tests. To do so, you can click on the re-run icon [3], as shown in Figure 3-63.

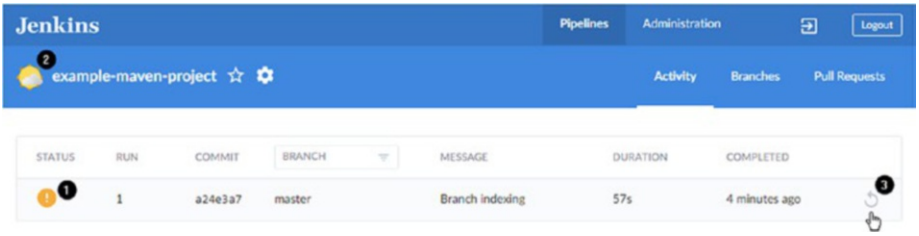

Figure 3-63. *Re-running a pipeline in Blue Ocean*

Moreover, this is how a successful pipeline looks in Jenkins Blue Ocean (see Figure 3-64). Let's take a detailed view of this successful pipeline run by clicking on it.

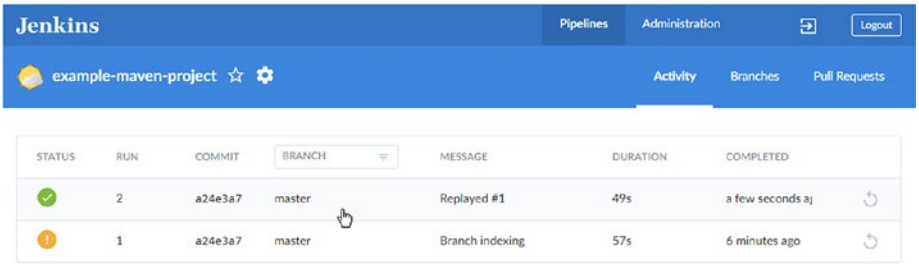

Figure 3-64. *A successful Blue Ocean pipeline*

Using the Pipeline Flow

Clicking on a particular pipeline run takes you to its dashboard. See Figure 3-65, which shows you a dashboard of my green build (pipeline run #2).

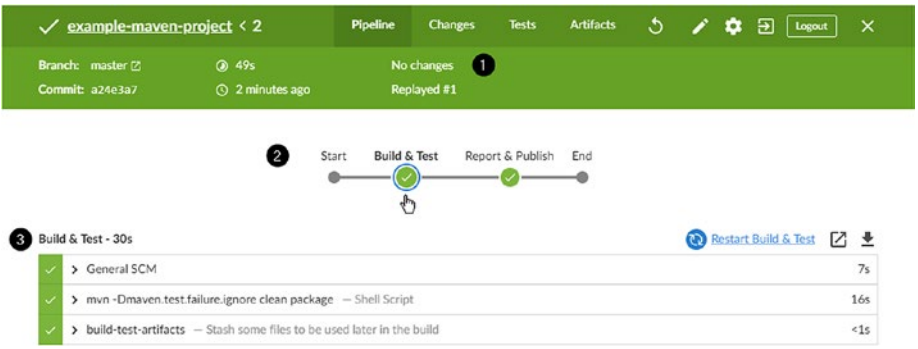

Figure 3-65. *The pipeline flow visualization*

On this page, you see little info about the pipeline run [1], such as the branch name for which the pipeline has run, the duration of the run, how long ago the pipeline ran, was it a manual replay?...etc.

Below the info panel, you'll see a nice and beautiful pipeline flow [2] that's interactive.

You can click on a stage to see it's corresponding steps [3]. You can expand every step to see its respective logs. Let's learn more about log tracing.

Tracing Logs at the Step, Stage, and Pipeline Level

If you are interested in the complete log of a particular stage, you can click on the **open in new tab** icon [1], as shown in Figure 3-66. This opens up the complete log of that particular stage in a new tab.

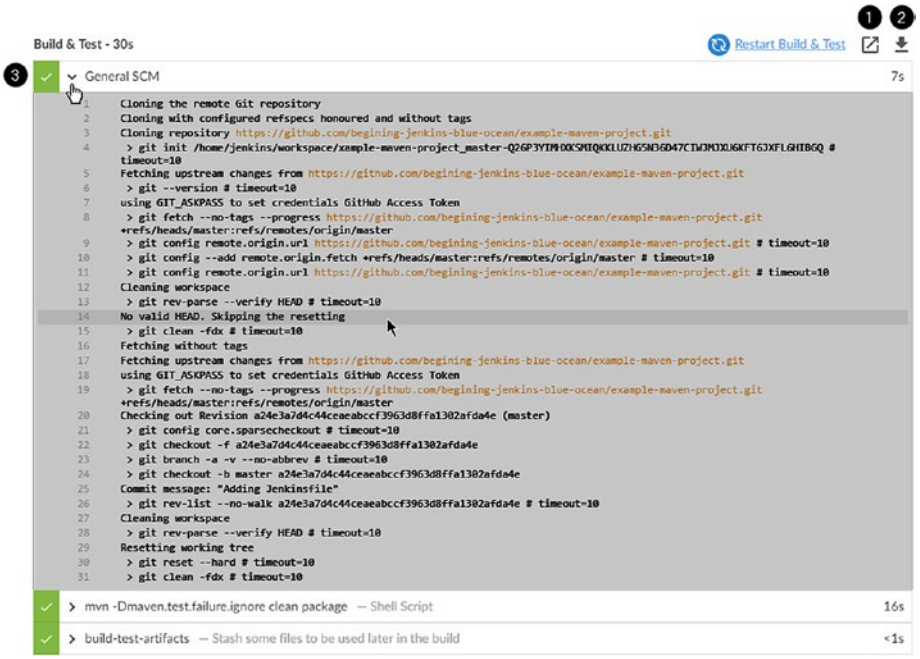

Figure 3-66. *Build log of an individual step*

To download the logs for a particular stage, click on the **download** icon [2]. You can also debug a particular step by expanding its corresponding log; do this by clicking on the **greater than arrow** [3].

To get the logs for the complete pipeline run, navigate to the **Artifacts** page and take a look at the section *Using the Artifacts View*.

Using the Tests View

The Tests page shows you the testing results for a particular pipeline run (see Figure 3-67). The Tests page only works if you have configured a step inside your pipeline to upload testing results.

As of today, Jenkins Blue Ocean supports reading/displaying testing reports that are in JUnit or Xunit format.

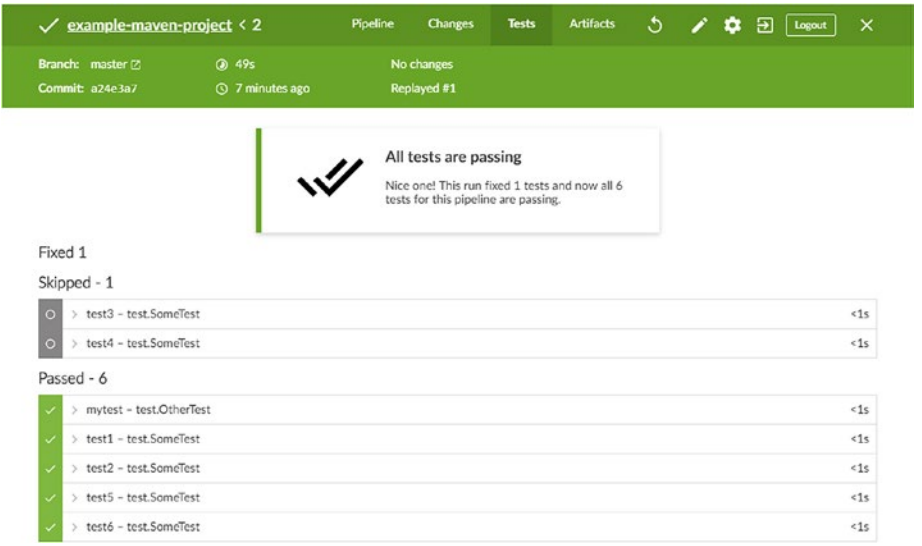

Figure 3-67. *The test result page of a pipeline run*

The pipeline shows green if all the tests pass, assuming that all the other stages of your pipeline have passed too. If some of the tests fail, the pipeline turns yellow.

Figure 3-68 shows you a pipeline run with some failed tests. You can see the total number of test failures are visible right on the Tests page tab [1]. There is also a short note on the Tests page with a big cross-mark [2], which gives you a quick summary of the testing failure.

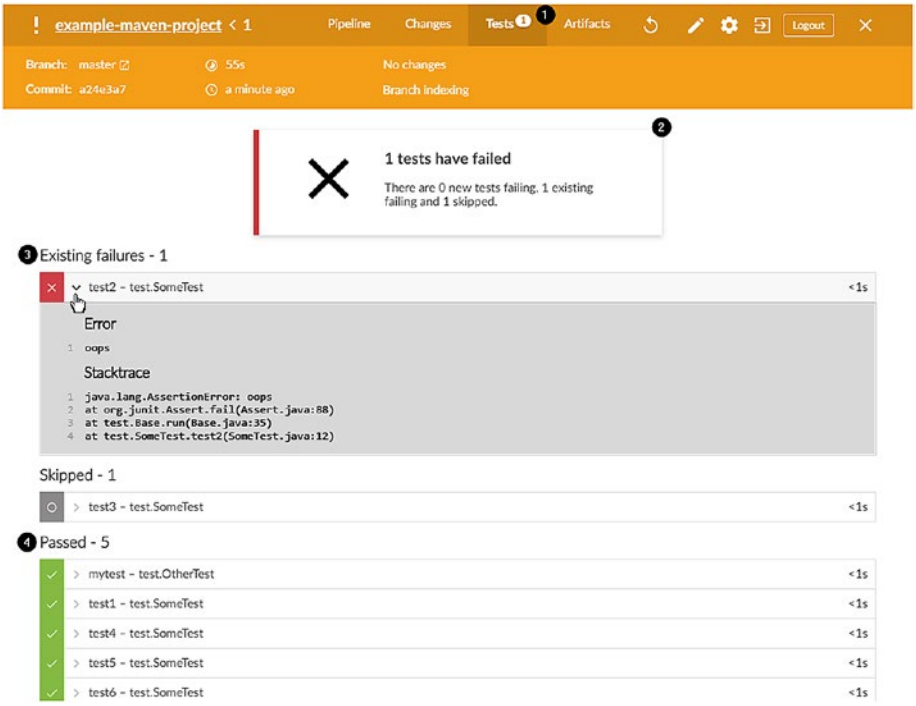

Figure 3-68. *Stack trace of a test failure*

Below the short note, you have a list of the failed tests [3], which are displayed first. Since failure is the first thing that a developer wants to see, I would say this is Intuitive. The failed tests are followed by the skipped test, which is then followed by the successful tests [4]. You can expand each test to get more details about the test failure.

All that you see on the Tests page is Jenkins working hand-in-hand with the JUnit plugin. When you install Jenkins Blue Ocean plugin suite, the JUnit plugin comes installed as part of its dependency.

Using the Artifacts View

The Artifacts page lists all the artifacts for a particular pipeline run (see Figure 3-69).

The Artifacts page contains the log for the complete pipeline run [1] by default. You can also see the built artifact [2] that is uploaded to Jenkins using the step **Archive the artifacts**.

It is, however, not recommended to upload built artifacts to the Jenkins server, as this may adversely affect the Jenkins server's disk space and its performance.

The risk could be reduced by regularly deleting old pipeline runs that are no longer needed. However, there are better options out there to help you store your built artifacts, such as Artifactory and Nexus, to name a few.

You can download all the artifacts for a pipeline run by clicking on the **Download All** button [4], or you can download a particular artifact [3] by clicking on its respective **download** icon [3].

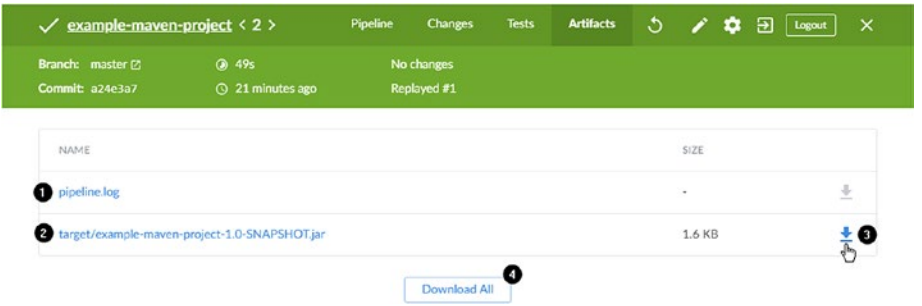

Figure 3-69. *The Artifacts page of a pipeline run*

Editing an Existing Pipeline in Blue Ocean

In the following section, you'll learn to edit an existing pipeline in Jenkins Blue Ocean. To keep things simple, let us continue with editing the same pipeline that we created at the beginning of this chapter.

To keep things exciting, and to learn other fascinating things on the way, I have carefully chosen the changes that you'll make on the existing pipeline as part of the editing process. You'll add a parallel stage to the existing Report & Publish stage of the pipeline, which publishes artifacts to the Artifactory server.

Following is the list of tasks that you'll perform in this section:

- You'll spawn an Artifactory server with Docker.
- You'll install and configure the Artifactory plugin in Jenkins.
- You'll edit the existing pipeline in Blue Ocean and add a parallel stage.
- You'll save the changes to a new branch instead of master to see the pipeline for multiple branches in Blue Ocean.

So let's begin.

Run an Artifactory Server

In the following section, you'll learn to spawn an Artifactory server with docker. Artifactory is a popular tool to manage and version control software build artifacts. The community editor of Artifactory is free. Follow the below steps to get it:

1. Log in to your docker host.
2. Execute the `docker volume create` command to create a docker volume named `artifactory_data` that you'll use with your Artifactory container. We have already seen the advantages of using a docker volume in Chapter 2. Use sudo wherever necessary.

   ```
   docker volume create --name artifactory_data
   ```

3. Download the docker image for Artifactory. Please note that we are downloading the latest version of Artifactory community edition.

   ```
   docker pull \
    docker.bintray.io/jfrog/artifactory-oss:latest
   ```

4. Spawn an Artifactory server container using the respective image and the docker volume for Artifactory.

```
docker run --name artifactory -d \
 -v artifactory_data:/var/opt/jfrog/ \
 -p 8081:8081 \
 docker.bintray.io/jfrog/artifactory-oss:latest
```

5. If everything runs well, you should be able to access your Artifactory server on the following URL: `http://<Docker_Host_IP>:8081/artifactory/webapp/#/home`

6. Try to log in to the Artifactory server using the admin credentials (i.e., username: `admin`, and password as `password`; see Figure 3-70). Also, note the default example repository in Artifactory named **example-repo-local** [1]. You'll be using it to keep things simple.

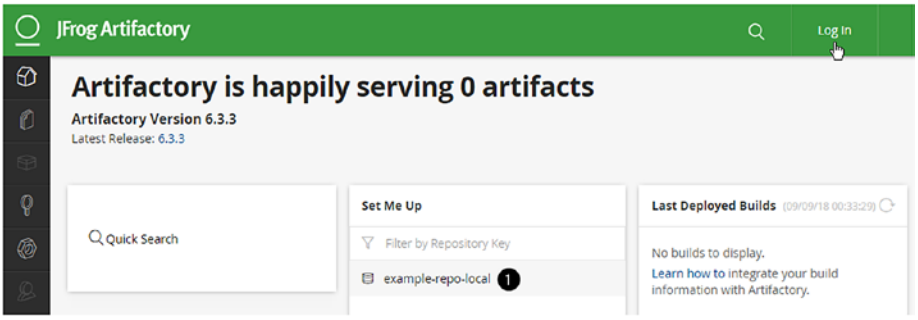

Figure 3-70. *The Artifactory server dashboard*

Installing the Artifactory Plugin for Jenkins

You have your Artifactory server up and running. Next, let us install the Artifactory plugin in Jenkins.

To do so, from the Jenkins Blue Ocean Dashboard, or the Project dashboard, click on the **Administration** link [1], as shown in Figure 3-71.

Figure 3-71. *The Jenkins Administration link*

From the **Manage Jenkins** page, click on the **Manage Plugins** link. And, on the Manage Plugins page, click on the **Available** tab. Next, use the **Filter** field to search for *Artifactory* [1]. When you find it, select it [2], and click on **Install without restart** button [3] (see Figure 3-72).

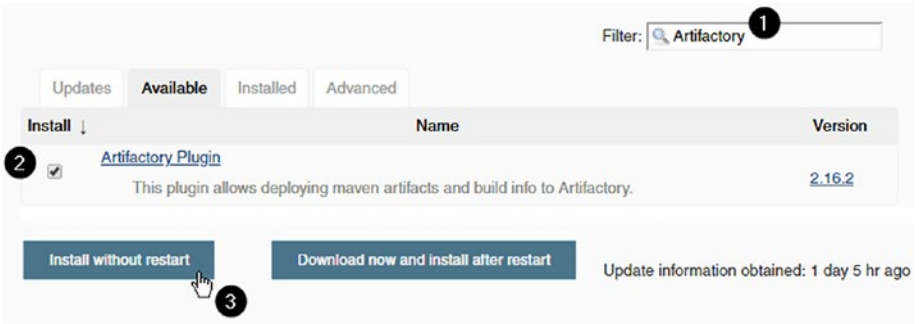

Figure 3-72. *Installing the Artifactory plugin using the Jenkins Plugin Manager*

Jenkins begins installing the plugin. Once the installation finishes, you may choose to restart Jenkins by selecting the option **Restart Jenkins when installation is complete and no jobs are running,** or you can start using the installed plugin right away by clicking the option **Go back to the top page**.

Configuring the Artifactory Plugin in Jenkins

You have the Artifactory plugin installed, now let's configure it. From the Classic Jenkins Dashboard, navigate to **Manage Jenkins ➤ Configure System**. On the Configuration page, scroll all the way to the **Artifactory** section.

Figure 3-73 shows you the desired configuration for the Artifactory server. Let's see them one by one.

[1] **Server ID**: Add a unique name to identify your Artifactory server configuration.

[2] **URL**: Add the URL for your Artifactory server.

[3] **Default Deployer Credentials**: These are the default Artifactory credentials that are used by Jenkins to upload/download artifacts from Artifactory.

[4] **Test Connection**: Click this button to test the communication between Jenkins and your Artifactory server.

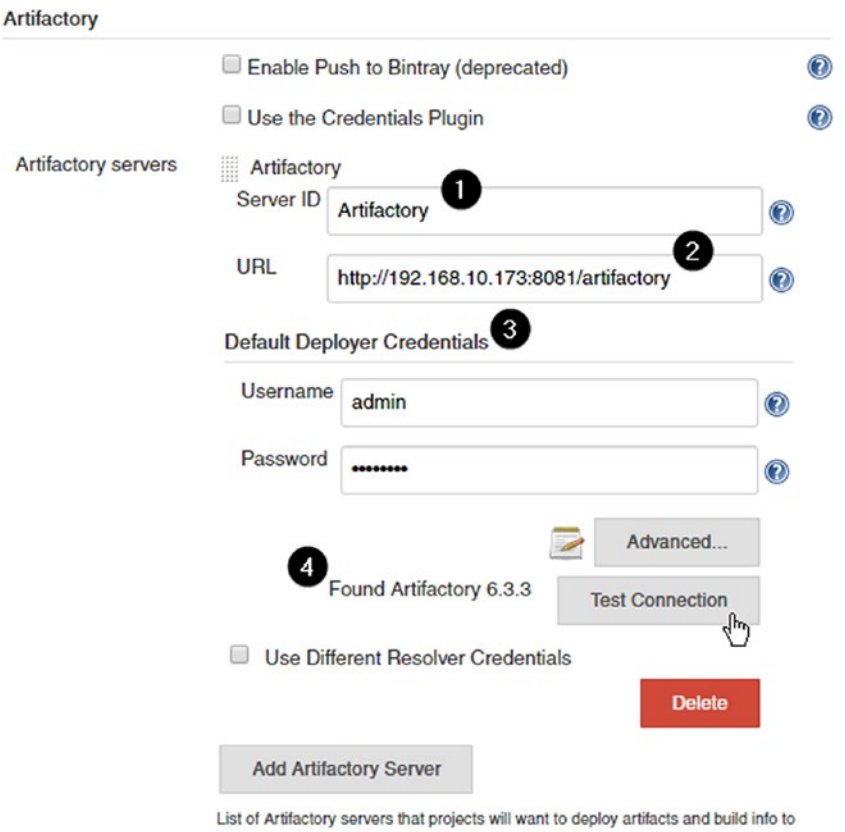

Figure 3-73. *Configuring the Artifactory plugin*

Once you finish working with your configurations, scroll to the bottom of the page and click on **Apply & Save** buttons to save the configuration to Jenkins.

You are now done performing the small prerequisites that are needed by the new pipeline change. Let's begin with editing your existing JenkinsBlue Ocean Pipeline.

Editing the Pipeline in Jenkins Blue Ocean

In the following section, you'll learn to edit your existing Jenkins Pipeline using the Visual Pipeline Editor.

To begin editing a pipeline, go to the pipeline run's dashboard and click on the edit icon [1] from the menu bar (see Figure 3-74).

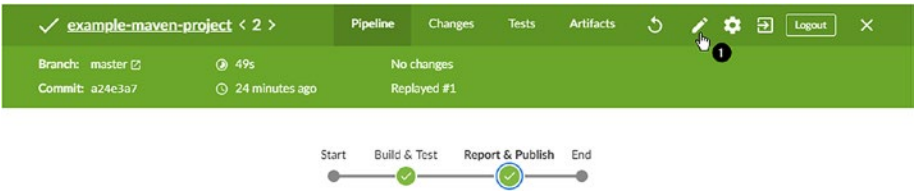

Figure 3-74. *Editing a Blue Ocean pipeline*

The Visual Pipeline Editor opens up. Let's begin creating a stage and a few steps.

Creating a Publish to Artifactory Stage (Parallel Stage)

Let's add a stage in parallel to your existing **Report & Publish** stage. To do so, click on the plus icon, as shown in Figure 3-75.

Figure 3-75. *Creating a new stage*

Name your new stage **Publish to Artifactory** using the field [1] in the configuration section (see Figure 3-76).

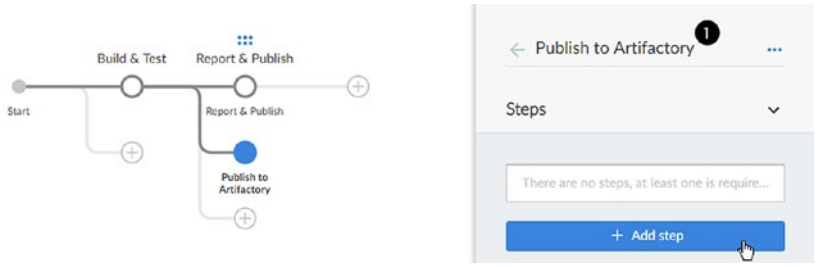

Figure 3-76. *Naming your stage*

Your new stage first downloads the stash files from the previous stage, and then it publishes the built artifacts to the Artifactory server.

Adding a Scripted Pipeline Step

You'll add a scripted pipeline step that does two things. First, it fetches the stash files from the previous stage. Second, it runs a filespec that uploads the built package to the Artifactory server.

To do so, click on the **+ Add step** button (see Figure 3-76). Then, look for the step **Run arbitrary Pipeline script** by searching it using the search field [1] (see Figure 3-40). Click it once you find it.

Figure 3-77. *Adding a Scripted Pipeline step*

You'll be presented with a text box to add a script of your choice. Add the following code to the text box. See Figure 3-78.

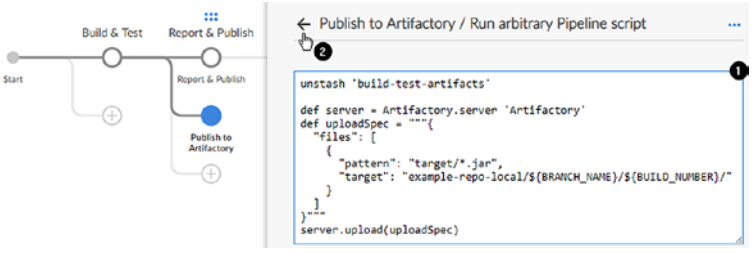

Figure 3-78. *Running an arbitrary pipeline script*

Script to Un-Stash Built Artifacts and Upload Them to the Artifactory Server

```
unstash 'build-test-artifacts'

def server = Artifactory.server 'Artifactory'
def uploadSpec = """{
  "files": [
    {
      "pattern": "target/*.jar",
      "target": "example-repo-local/${BRANCH_NAME}/${BUILD_NUMBER}/"
    }
  ]
}"""
server.upload(uploadSpec)
```

Let me explain what this code. The code line `unstash 'build-test-artifacts'` downloads the previously stashed package. The rest of the code is a Filespec that uploads the `target/*jar` file, which is our built package file, to the Artifactory server on the repository `example-repo-local`.

Notice that the target path contains Jenkins Global variables, `${BRANCH_NAME}` and `${BUILD_NUMBER}`, representing the branch name and build number, respectively. Doing so uploads the built artifacts to a unique path on Artifactory every single time a pipeline runs.

Next, click on the back arrow [2] to come out of the current configuration.

Assigning an Agent for the Publish to Artifactory Stage

Next, you'll assign a build agent for your **Publish to Artifactory** stage. The agent is going to be the docker container that gets spawned automatically by Jenkins. Once the stage is complete, Jenkins destroys the container.

To do so, expand the **Settings** section [1] (see Figure 3-79). You'll see a few options to configure. Choose **node** as the agent type [2], and type in **docker** for the **Label*** field [3]. Contract the **Settings** section [1] afterward.

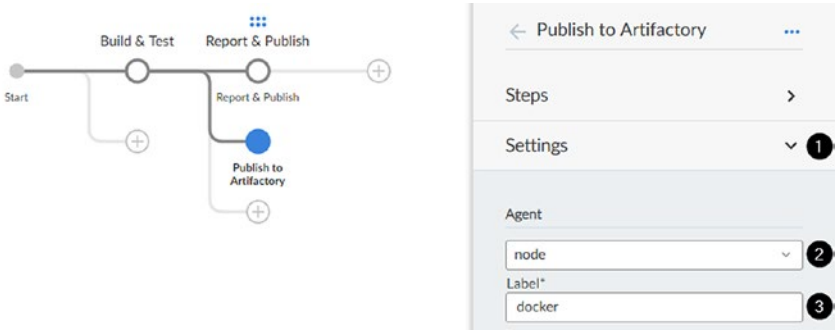

Figure 3-79. *Assigning an agent to a stage*

You're now done editing the Pipeline.

Your final pipeline code:

```
pipeline {
  agent none
  stages {
```

```
stage('Build & Test') {
  agent {
    node {
      label 'docker'
    }
  }
  steps {
    sh 'mvn -Dmaven.test.failure.ignore clean package'
    stash(name: 'build-test-artifacts', \
    includes: '**/target/surefire-reports/TEST-*.
    xml,target/*.jar')
  }
}
stage('Report & Publish') {
  parallel {
    stage('Report & Publish') {
      agent {
        node {
          label 'docker'
        }
      }
      steps {
        unstash 'build-test-artifacts'
        junit '**/target/surefire-reports/TEST-*.xml'
        archiveArtifacts(onlyIfSuccessful: true, artifacts:
        'target/*.jar')
      }
    }
    stage('Publish to Artifactory') {
      agent {
        node {
```

```
            label 'docker'
          }
        }
        steps {
          script {
            unstash 'build-test-artifacts'

            def server = Artifactory.server 'Artifactory'
            def uploadSpec = """{
              "files": [
                {
                  "pattern": "target/*.jar",
                  "target": "example-repo-local/${BRANCH_
                            NAME}/${BUILD_NUMBER}/"
                }
              ]
            }"""
            server.upload(uploadSpec)
          }
        }
      }
    }
  }
}
```

To save the changes, click on the **Save** button (see Figure 3-80).

Figure 3-80. *Save button to save your pipeline changes*

When you click on the **Save** button, Jenkins in the back end converts your UI configurations into a Jenkinsfile, as you know it.

Next, you should see a pop-up window, asking you to check-in your new Jenkinsfile to the source code repository (see Figure 3-81). Add a meaningful comment in the **Description** field [1]. Select **Commit to new branch** [2], which means you are making a change on a new branch of your repository. In the text field below, you add the name that you would like to give to your new branch. To finally make a commit, click on the **Save & Run** button, which will also eventually trigger a Pipeline in Jenkins Blue Ocean.

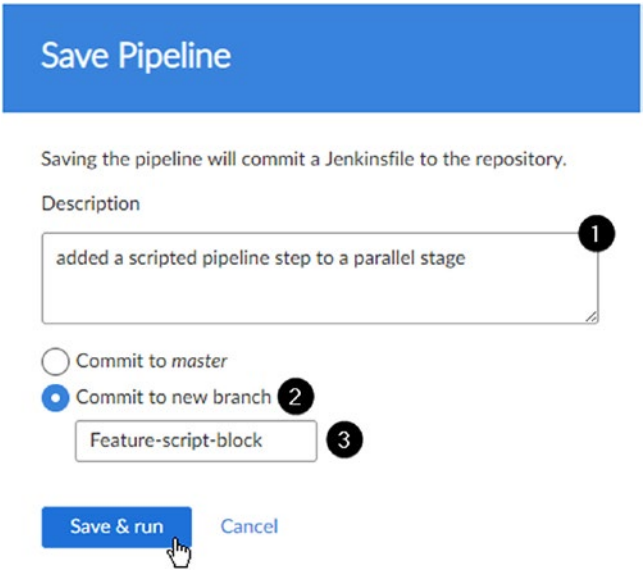

Figure 3-81. *Committing your pipeline configurations changes*

Viewing Pipelines for Multiple Branches of a Project

In the previous section, you made some changes to the Pipeline using the Visual Pipeline Editor. These changes were in the end saved to a new branch instead.

This eventually results in a pipeline in Jenkins Blue Ocean (see Figure 3-82). Your project dashboard now shows a new pipeline for the Feature-script-block branch, which might turn yellow or green.

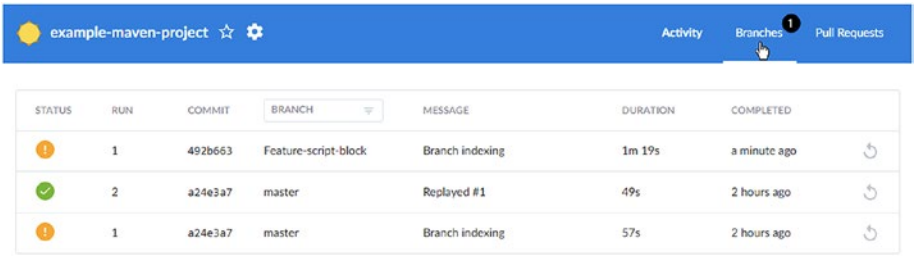

Figure 3-82. *Pipeline for a feature branch*

Let's take a look at the **Branches** tab [1] by clicking on it. You'll see two pipelines, one for the *master* branch and the other for the *Feature-script-block* branch, provided you are following through the steps described in the current chapter.

The branches tab shows the latest pipeline run for every branch of your project that contains a Jenkinsfile (see Figure 3-83).

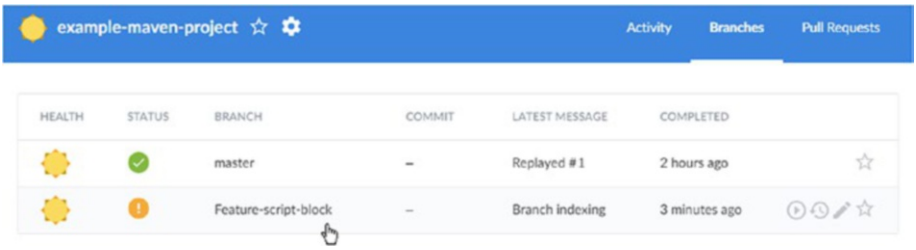

Figure 3-83. *The branches tab*

You can take a look at the pipeline flow of the new pipeline run on the Feature-script-block branch. Just click on it, and you'll see the pipeline run dashboard.

Figure 3-84. *The pipeline flow diagram*

The steps for the stage Publish to Artifactory have all passed. Let's take a look inside your Artifactory server to see if the built artifacts are present.

Log in to the Artifactory server using the admin credentials. Using the **Artifactory Repository Browser,** look inside the example-repo-local. You should see the build artifact inside a folder path that's a combination of the branch name and the pipeline run number (see Figure 3-85).

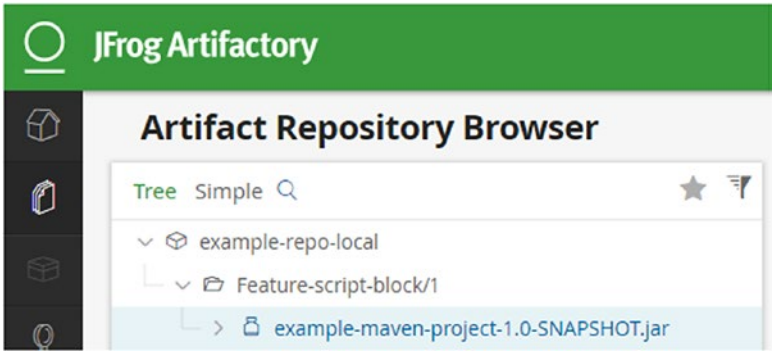

Figure 3-85. *Built artifact uploaded to Artifactory*

Running a Pipeline for a Pull Request

Jenkins Blue Ocean can detect pull requests on your Git/GitHub/Bitbucket repositories and run a pipeline with it for you. The pipeline run result (fail/pass/canceled) gets reported back to your source code repository.

The person who is responsible for accepting the pull request can then decide based on the pipeline run result whether he should merge the new changes into the destination branch or not.

If you are following the steps described in the current chapter, then at this point, you are in a position to create a pull request. Let's see how Jenkins Blue Ocean reacts to your pull request.

To create a pull request, log in to your GitHub repository. On your Repository Dashboard page, you should see a notification about GitHub proposing to you a pull request [1] (see Figure 3-86). If you don't see anything, you can always switch to the **Pull requests** tab [2] and create a new pull request.

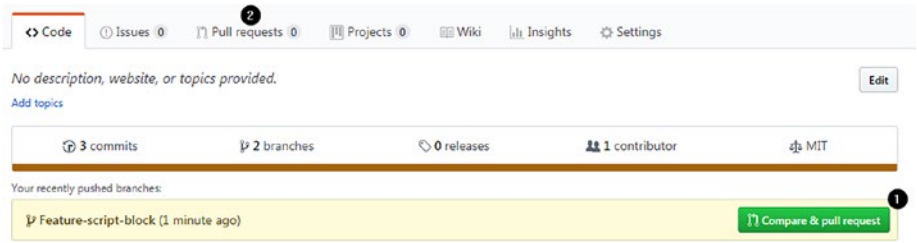

Figure 3-86. *A pull request proposal*

Figure 3-87 demonstrates opening a new pull request. As you can see, you need to choose a destination branch [1] and a source branch [2]. The source branch is the branch from where you pull the changes, and the destination branch is the branch where you merge the pulled changes.

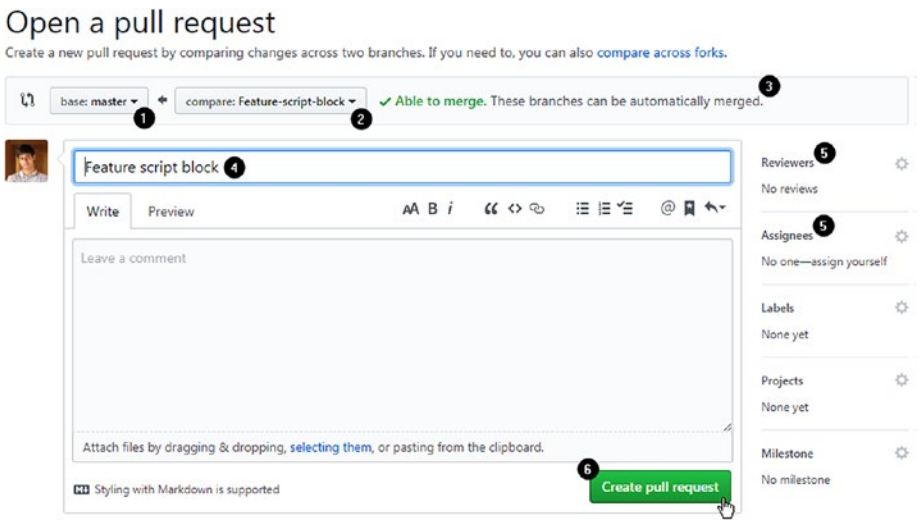

Figure 3-87. *Opening a new pull request*

GitHub also shows you a small notification [3] about the possibility of an automatic merge. In the text field [4] you can provide a short comment describing your pull request. You can also assign **Reviewers** and **Assignees** for this merge [5]. Finally, to create a pull request, click on the **Create pull request** button [6]. Doing so runs a Pipeline in Jenkins Blue Ocean for the pull request.

Switch to your Project Dashboard, and you'll see a pipeline run for the pull request (see Figure 3-88).

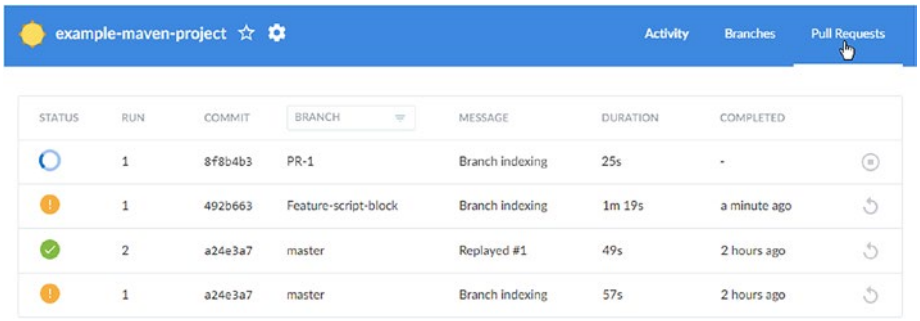

Figure 3-88. *Pipeline run for a pull request*

You can also see all pipeline runs for the pull requests using the **Pull Requests** tab (see Figure 3-89).

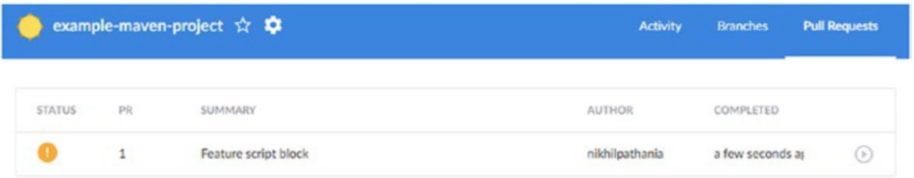

Figure 3-89. *The pull request tab*

When the pipeline run of your pull request is complete, switch back to the GitHub Pull Request page. The Assignees of this pull request should get a notification (see Figure 3-90). Since I assigned myself as the assignee, I get a request to review the pull request.

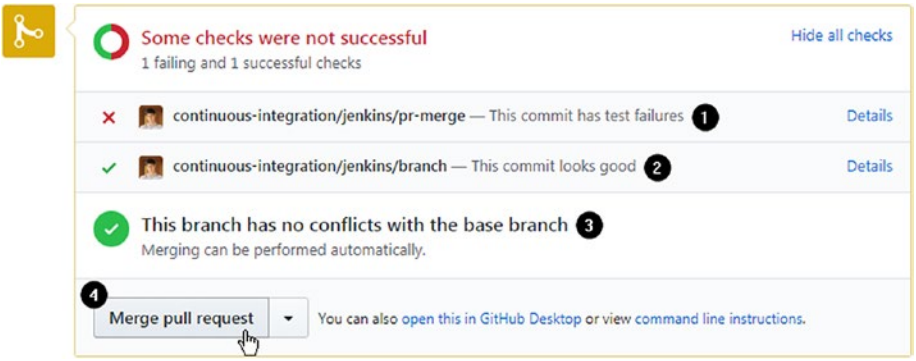

Figure 3-90. *Accept or Cancel a merge request*

You can see that GitHub in collaboration with Jenkins provided you (the assignee) with some info related to the pull request to help you decide whether or not to merge the pull request. The info includes the following:

[1] Pipeline status about the commit on the source branch that's pulled

[2] Pipeline status of the pull request

[3] Information about the possibility of an automatic merge

Clicking on the **Details** link takes you to the Pipeline Run dashboard on Jenkins Blue Ocean. Next, you can either decide to accept [4] or cancel the merge.

If you accept it, GitHub asks you again to Confirm the merge (see Figure 3-91).

Figure 3-91. *Confirming a merge request*

Once you confirm the merge, you have the option to delete the source branch. See Figure 3-92 for the **Delete branch** [1] option.

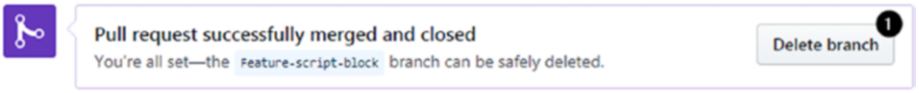

Figure 3-92. *The Delete branch option with GitHub pull request*

When you merge the pull request, Jenkins Blue Ocean immediately runs a pipeline on the master branch (see Figure 3-93).

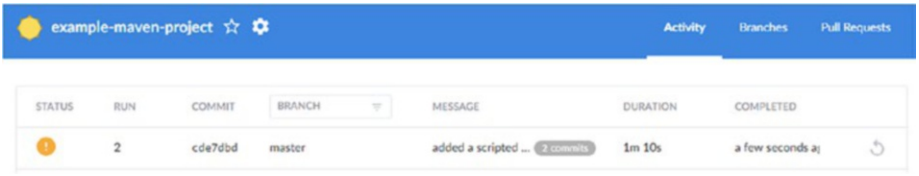

Figure 3-93. *An automated Jenkins pipeline for a successful merge request*

If you check Artifactory, you should find an artifact for every branch, including the pull request. We can improve the pipeline further not to upload built artifacts to Artifactory for a pipeline that's related to a pull request. However, let's leave this activity for the last chapter, where we learn to use more script steps and Jenkins Shared Libraries.

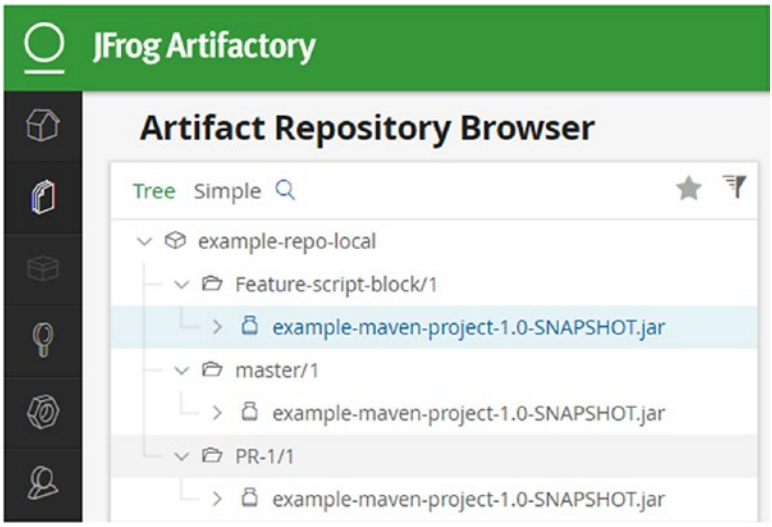

Figure 3-94. *Built artifacts from various source code branches uploaded to artifactory*

Summary

To summarize, in this chapter you learned in detail how to work with pipelines in Jenkins Blue Ocean. This includes using the Visual Pipeline Editor to create and edit pipelines; using the Pipeline Visualization to trace and debug pipeline logs at the step, stage, and pipeline levels; and viewing testing results and artifacts for every pipeline run. Along with this, you learned to deal with pull requests in Jenkins Blue Ocean.

In addition to these items, you did all this with docker (i.e., to spawn an on-demand Jenkins agent running as containers). On a side note, we also learned about running Artifactory and uploading built artifacts to it using the script step in your pipeline.

With this chapter, we end the GUI part of Jenkins Blue Ocean. To have a better understanding of the Declarative Pipeline and to gain greater control over it, you'll learn to design pipelines using the code, and the following chapters are all about it.

CHAPTER 4

Declarative Pipeline Syntax

Creating Pipelines using the *Visual Pipeline Editor* is quick and convenient. However, to get more control over your Pipelines, it is necessary to understand the underlying Pipeline code.

Before Jenkins Blue Ocean's *Visual Pipeline Editor*, Pipeline code was and is still widely written using a keyboard (coding). It is therefore essential to take a look at the components that make up the Pipeline code.

You'll learn more about the Pipeline code shortly. However, let me tell you in advance that the Pipeline code follows a declarative or an imperative coding style, and the following chapter is all about the declarative flavor. I have chosen the Declarative Pipeline Syntax because Jenkins Blue Ocean goes hand in hand with it. When you create a pipeline using the Visual Pipeline Editor, Blue Ocean saves it as a Declarative Pipeline.

The current chapter begins with a short note on the evolution of Pipeline as Code from Scripted Pipeline to Declarative Pipeline and later moves on to describe the elements of Declarative Pipeline Syntax in detail.

In this chapter you'll learn the following:

- Introduction about Pipeline as Code and Jenkinsfile
- Defining Scripted Pipeline
- Defining Declarative Pipeline
- Detailed guide about Declarative Pipeline Syntax

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5_4

The following chapter contains example codes that help you understand the various options provided by the Declarative Pipeline. Let's begin.

Caution All pipeline code described in the current chapter are templates, intended to serve as examples. The idea is to make you understand various options and features available with the Declarative Pipeline Syntax. Modify the templates using the necessary hints available with them before use.

Introduction to Pipeline as Code

Pipeline as Code, as the name stipulates, is an idea about describing your Pipeline in the form of code. While materializing the idea of Pipeline as Code, Apache Groovy (a Java-based object-oriented programming language) was chosen as its foundation. The result was *Scripted Pipeline*.

Scripted Pipeline

The Scripted Pipeline followed, and it continues to follow a syntax that is domain-specific and imperative in style. Moreover, it allows using the Groovy code inside it without any restrictions. Sometimes the Scripted Pipeline is also referred to as Job DSL.

An Example of Scripted Pipeline:

```
node('master') {
    stage('Poll') {
        // Checkout source code
    }
```

```
    stage('Build') {
        sh 'mvn clean verify -DskipITs=true';
    }
    stage('Static Code Analysis'){
        sh 'mvn clean verify sonar:sonar';
    }
    stage ('Publish to Artifactory'){
        def server = Artifactory.server 'Default Artifactory'
        def uploadSpec = """{
            "files": [
                {
                  "pattern": "target/*.war",
                  "target": "helloworld/${BUILD_NUMBER}/",
                  "props": "Unit-Tested=Yes"
                }
            ]
        }"""
        server.upload(uploadSpec)
    }
}
```

For non-programmers, here is a short description about *Imperative programming* and *Domain-Specific Language*.

Imperative programming: By definition, Imperative programming is a standard in computer programming, in which the program describes what you want to achieve along with how you want to achieve it. C, C++, Java, and Groovy are some of the Imperative programming languages.

Domain-Specific Language (DSL): A Domain-Specific Language is a language that's designed to address a specific area of problems. Apache Ant, Apache Maven, and Gradle are some of the DSLs. They are indeed distinct from a general purpose language, which is designed to work across various domains.

The Scripted Pipeline Syntax is powerful yet comes with a steep learning curve not desired by many. It's also difficult to maintain and understand.

Imagine debugging or modifying a Scripted Pipeline code written by someone else who is no longer on your team. Such situations are not desirable when you have a lot of Scripted Pipelines to maintain.

This problem pushed the Jenkins engineers to create a more comfortable to use DSL to describe Pipelines, thus giving rise to the Declarative Pipeline.

Declarative Pipeline

The Declarative Pipeline follows a syntax (Declarative Pipeline Syntax) that is domain-specific and declarative in style. Moreover, it allows using the Groovy code inside it, but with some restrictions.

The Pipelines that you create in Jenkins Blue Ocean are by default saved as Multi-Branch Declarative Pipelines.

An Example of Declarative Pipeline:

```
pipeline {
    agent {
        node {
            label 'master'
        }
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean verify -DskipITs=true'
            }
        }
```

```
        stage('Static Code Analysis') {
            steps {
                sh 'mvn clean verify sonar:sonar'
            }
        }
        stage ('Publish to Artifactory') {
            steps {
                script {
                    def server = Artifactory.server 'Default
                    Artifactory'
                    def uploadSpec = """{
                        "files": [
                            {
                                "pattern": "target/*.war",
                                "target": "helloworld/${BUILD_
                                          NUMBER}/",
                                "props": "Unit-Tested=Yes"
                            }
                        ]
                    }"""
                    server.upload(uploadSpec)
                }
            }
        }
    }
}
```

For non-programmers, here is a short description about *Declarative programming*.

Declarative programming: Declarative programming is a standard in computer programming, in which the program describes what you want to achieve, and not how you want to achieve it. It is up to the compiler to figure out how to achieve it. SQL, HTML, XML, and CSS are some of the Declarative programming languages.

The Declarative Pipeline Syntax is well-structured and comes with a smooth learning curve. It's easy to compose and maintain. However, it's restrictive in many ways with what it provides to the pipeline authors. To give you an example, in Declarative Pipeline you should always put Groovy code inside a `script {}` block.

Jenkinsfile

A Jenkinsfile is a file that contains Pipeline code. The Pipeline code can be in a Scripted Pipeline Syntax or in a Declarative Pipeline Syntax. As long as the Pipeline code is present in a file and version-controlled along with your source code inside a repository, it's a Jenkinsfile.

Pipelines that are created using Jenkins Blue Ocean are by default saved inside a Jenkinsfile, in a Declarative Pipeline Syntax, and uploaded to your source code repository.

In the Classic Jenkins, however, it's possible to save your Pipeline code directly inside the Pipeline Job.

Declarative Pipeline Syntax

In the following section, you'll learn in detail about the elements that make up the Declarative Pipeline Syntax. The topics discussed in the following section detail all that is possible using the Declarative Pipeline.

Sections

Sections in Declarative Pipeline Syntax can contain one or more Directives or Steps. Let's see all the sections one-by-one.

Agent

The `agent` section defines where the whole Pipeline, or a specific stage, runs. It is positioned right at the beginning of the `pipeline {}` block and is compulsory. Additionally, you can also use it inside the `stage` directive, but that's optional.

The `agent` section allows specific parameters to suit various use-cases. Let's see them one-by-one.

Any

Use `agent any` inside your `pipeline {}` block, or `stage` directive, to run them on any agent available in Jenkins.

A Pipeline {} Block with agent any:

```
pipeline {
    agent any
    stages {
        stage ('say hello') {
           steps {
               echo 'Hello everyone'
           }
       }
    }
}
```

None

Use agent none inside your pipeline {} block to stop any *global agent* from running your pipeline. In such a case, each stage directive inside your pipeline must have its agent section.

A Pipeline {} Block with Agent None:

```
pipeline {
    agent none
    stages {
        stage('Parallel Testing') {
            parallel {
                stage('Test with Firefox') {
                    agent {
                        label 'Windows2008_Firefox'
                    }
                    steps {
                        echo "Testing with Firefox."
                    }
                }
                stage('Test with Chrome') {
                    agent {
                        label 'Windows2008_Chrome'
                    }
                    steps {
                        echo "Testing with Chrome."
                    }
                }
            }
        }
    }
}
```

Label

Run your complete pipeline, or stage, on an agent available in Jenkins with a specific label.

An `Agent` Section with Label Parameter:

```
agent {
    label 'windows2008_Chrome'
}
```

This `agent` section runs a complete pipeline, or a stage, on a Jenkins agent with label: windows2008_Chrome.

A `Pipeline {}` Block with `Agent { Label '...' }` Section:

```
pipeline {
    agent none
    stages {
        stage('Parallel Testing') {
            parallel {
                stage('Test with Firefox') {
                    agent {
                        label 'Windows2008_Firefox'
                    }
                    steps {
                        echo "Testing with Firefox."
                    }
                }
                stage('Test with Chrome') {
                    agent {
                        label 'Windows2008_Chrome'
                    }
```

```
                    steps {
                        echo "Testing with Chrome."
                    }
                }
            }
        }
    }
}
```

Node

Run your complete pipeline, or stage, on an agent available in Jenkins with a specific label. The behavior is similar to the *Label* parameter. However, the *Node* parameter allows you to specify additional options such as *custom Workspace*. Workspace, or Jenkins Pipeline workspace, is a reserved directory on your Jenkins agent, where all your source code is downloaded and where your builds are performed.

A `Pipeline {}` Block with `Agent { Node { Label '...' } }` Section:

```
pipeline {
    agent {
        node {
            label 'ubuntu_maven'
            customWorkspace '/some/path'
        }
    }
    stages {
        stage('Build') {
             steps {
                sh 'mvn -B clean verify'
             }
        }
    }
}
```

Docker

Run your complete Pipeline, or stage, inside a docker container. The container will be dynamically spawned on a pre-configured docker host.

The *docker* option also takes an *args* parameter that allows you to pass arguments directly to the `docker run` command.

An `Agent` Section with Docker Parameter:

```
agent {
    docker {
        image 'ubuntu:trusty'
        label 'ubuntu_maven'
        args  '-v /tmp:/home/dev'
    }
}
```

Dockerfile

Run your Pipeline, or stage, inside a container built from a *dockerfile*. To use this option, the *Jenkinsfile* and the *dockerfile* should be present inside the root of your source code.

An `Agent` Section with Dockerfile Parameter:

```
agent {
    dockerfile true
}
```

If your *dockerfile* is in some path inside your source code, use the `dir` option.

```
agent {
    dockerfile {
        dir 'someSubDir'
    }
}
```

If your *Dockerfile* has a different name, you can specify your *Dockerfile* name with the `filename` option.

```
agent {
    dockerfile {
        filename 'Dockerfile.maven'
        dir 'someSubDir'
    }
}
```

You can pass additional arguments to the `docker build` command with the *additionalBuildArgs* option.

```
agent {
    // Equivalent to "docker build -f Dockerfile.maven --build-
    arg version=0.0.7 ./someSubDir/
    dockerfile {
        filename 'Dockerfile.maven'
        dir 'someSubDir'
        additionalBuildArgs  '--build-arg version=0.0.7'
    }
}
```

Post

The `post` section allows you to run additional `steps` after running a Pipeline or a stage.

A `Post` Section at the End of a Pipeline:

```
pipeline {
    agent any
    stages {
        stage('Say Hello') {
```

```
            steps {
                echo 'Hello World.'
            }
        }
        // A post section at the end of a stage
        post {
            sucess {
              echo 'This stage is successful.'
            }
        }
    }
    // A post section at the end of a pipeline
    post {
        always {
            echo 'Jenkins Blue Ocean is wonderful.'
        }
    }
}
```

You can add specific conditions to your `post` section. These conditions allow the execution of `steps` inside your `post` section depending on the completion status of your Pipeline or stage.

Condition	Description
always	Run the `steps` in the `post` section in every case.
changed	Run the `steps` in the `post` section only if the completion status of your current Pipeline or stage run is different from its previous run.
fixed	Run the `steps` in the `post` section only if your current Pipeline or stage run is successful and the previous run failed or was unstable.

(*continued*)

Condition	Description
regression	Run the `steps` in the `post` section only if your current Pipeline or stage status is a failure, unstable, or aborted, and the previous run was successful.
aborted	Run the `steps` in the `post` section only if your current Pipeline or stage has an "aborted" status.
failure	Run the `steps` in the `post` section only if your current Pipeline or stage has failed.
success	Run the `steps` in the `post` section only if your current Pipeline or stage is successful.
unstable	Run the `steps` in the `post` section only if the current Pipeline or stage run is unstable, usually caused by test failures, code violations.
cleanup	Run the `steps` in this `post` section after every other post condition gets evaluated, regardless of the Pipeline or stage run status.

Stages

The `stages` section contains a sequence of one or more `stage` directives. It's used to segregate a single or a collection of `stage` directives from the rest of the code.

The `stages` section has no parameters and gets used at least once in the pipeline. You'll learn more about `stages` in the upcoming topics *Sequential Stages* and *Parallel Stages*.

Steps

The `steps` section contains one or more steps that should run inside a `stage` directive. The `steps` section is used to segregate a single or a collection of steps from the rest of the code inside a `stage` directive.

A `Pipeline {}` Block with `Steps` Section:

```
pipeline {
    agent none
    stages {
        stage ('say hello') {
          agent {
            label 'ubuntu_maven'
            }
          // A steps section
          steps {
            // An echo step
              echo 'Hello everyone'
            // A sh step
            sh 'pwd'
          }
      }
    }
}
```

Directives

Directives are supporting actors that give direction, set conditions, and provide assistance to the steps of your pipeline in order to achieve the required purpose. In the following section we will see all the `Directives` that are available to you with the Declarative Pipeline Syntax.

Environment

The `environment` directive is used to specify a set of key value pairs that are available as environment variables for all the steps of your pipeline, or only for the steps of a specific stage, depending on where you place the `environment` directive.

A `Pipeline {}` Block with `Environment` Section:

```
pipeline {
    agent any
    // At pipeline level
    environment {
        key1 = 'value1'
        key2 = 'value2'
    }
    stages {
        stage('Example') {
            // At stage level
            environment {
                key3 = 'value3'
            }
            steps {
                sh 'echo $key1'
                sh 'echo $key2'
                sh 'echo $key3'
            }
        }
    }
}
```

Options

The `options` directive allows you to define pipeline-specific `options` from within the Pipeline.

Jenkins comes with a few core `options`. Additionally, new `options` are available through plugins. Following is a short list of some `options` that make sense in Blue Ocean.

buildDiscarder

Allows you to keep a certain number of recent builds runs.

```
// Will keep the last 10 Pipeline logs and artifacts in
Jenkins.
options {
    buildDiscarder(logRotator(numToKeepStr: '10'))
}
```

disableConcurrentBuilds

Disallows concurrent executions of the Pipeline.

To understand the usage of this option, imagine a situation. You have a Pipeline in Jenkins Blue Ocean that's triggered on every push on your source code repository. Moreover, you have a single build agent to run your Pipeline.

Now, if there are multiple pushes to the repository in a short duration of time (say three different pushes at once), then there is a pipeline for every push (say, Pipelines #1, #2, and #3). However, since we have a single build agent, Pipelines #2 and #3 wait in a queue with Pipeline #1 running. Pipelines #2 and #3 run whenever the build agent is free again.

To disable this functionality, and discard everything that's in the queue, you should use the *disableConcurrentBuilds* option. The following option is useful to perform incremental builds.

```
options {
    disableConcurrentBuilds ()
}
```

newContainerPerStage

The `newContainerPerStage` option is used inside an agent section that employs the docker or dockerfile parameters.

When specified, each stage runs in a new container on the same docker host, as opposed to all stages utilizing the same container.

This option only works with an `agent` section defined at the top-level of your `Pipeline {}` block.

preserveStashes

In Jenkins, you can pass artifacts between stages. You do it using *stash*. The `preserveStashes` option allows you to preserve stashes of the completed Pipelines. It's useful if you have to re-run a stage from a completed Pipeline.

```
// Preserve stashes from the most recent completed Pipeline.
options {
    preserveStashes()
}
```

Or

```
//  Preserve stashes from the ten most recent completed
Pipelines.
options {
    preserveStashes(10)
}
```

Retry

The `retry` option allows you to retry a Pipeline, or a stage, on failure.

Retry an Entire Pipeline Three Times, On Failure:

```
pipeline {
    agent any
    options {
        retry(3)
    }
    stages {
        stage ('say hello') {
            steps {
                echo 'Hello everyone'
            }
        }
    }
}
```

Retry a `stage` 3 times, on failure:

```
pipeline {
    agent any
    stages {
        stage ('say hello') {
            options {
                retry(3)
            }
            steps {
                echo 'Hello everyone'
            }
        }
    }
}
```

skipDefaultCheckout

The `skipDefaultCheckout` option is another handy choice. In Jenkins Declarative Pipeline, the source code by default gets checked out in every `stage` directive of a Pipeline.

Use this option to skip checking out source code in a given `stage` directive.

A `Pipeline {}` Block with skipDefaultCheckout Option:

```
pipeline {
    agent any
    stages {
        stage ('say hello') {
            options {
                skipDefaultCheckout()
            }
            steps {
                echo 'Hello everyone'
            }
        }
        stage ('say hi') {
            steps {
                echo 'Hi everyone'
            }
        }
    }
}
```

Timeout

The `timeout` option allows you to set a timeout period for your Pipeline run. If the pipeline run times out the duration defined using a `timeout`, Jenkins aborts the Pipeline.

It is possible to use the `timeout` option at the Pipeline or stage level.

A `Pipeline {}` Block with the `Timeout` Option:

```
pipeline {
    agent any
    option {
        timeout(time: 2, unit 'HOURS')
    }
    stages {
        stage ('Build') {
            steps {
                bat 'MSBuild.exe MyProject.proj /t:build'
            }
        }
    }
}
```

There are times when a pipeline that's waiting for a process runs forever, generating huge logs and bringing down Jenkins server. The `timeout` option comes handy in such cases.

Timestamps

The `timestamps` option prepend all lines of the console log with a timestamp. The following option is useful in debugging issues where you would need the time of execution of a specific command as evidence.

It is possible to use the `timestamps` option at the Pipeline or stage level.

A `Pipeline {}` Block with the `Timestamps` Option:

```
pipeline {
    agent any
    option {
        timestamps()
    }
```

```
    stages {
        stage ('Build') {
            steps {
                bat 'MSBuild.exe MyProject.proj /t:build'
            }
        }
    }
}
```

Parameters

The `parameters` directive allows a user to provide a list of specific parameters when triggering a Pipeline.

Following are the types of available parameters that I find useful.

String

Use the `string` parameter to pass a string.

A `Pipeline {}` Block with a `String` Parameter:

```
pipeline {
    agent any
    parameters {
        string(name: 'perf_test_dur', defaultValue: '9000',
        description: 'Performance testing duration')
    }
    stages {
        stage ('Performance Testing') {
            steps {
                // The string parameter is available as an
                environment variable
```

```
                echo "${params.perf_test_dur}"
                // Execute some performance testing
            }
        }
    }
}
```

Text

Use the text parameter to pass a multiline text.

A Pipeline {} Block with a Text Parameter:

```
pipeline {
    agent any
    parameters {
        text(name: 'comments', defaultValue: 'Hello',
        description: ")
    }
    stages {
        stage ('say hello') {
            steps {
            echo "${params.comments}"
            }
        }
    }
}
```

booleanParam

Use the `booleanParm` parameter to pass true or false.

A `Pipeline {}` Block with a `booleanParm` Parameter:

```
pipeline {
    agent any
    parameters {
       booleanParam(name: 'upload_artifacts', defaultValue:
       true, description: '')
    }
    stages {
        stage ('Publis Artifacts') {
            steps {
                script {
                    if (upload_artifacts == 'true') {
                        // Do something
                    }
                    else {
                        // Do something else
                    }
                }
            }
        }
    }
}
```

Choice

Use the `choice` parameter to allow you to choose from a set of values, and then pass it to the Pipeline.

A `Pipeline {}` Block with a `Choice` Parameter:

```
pipeline {
    agent any
    parameters {
        choice(name: 'testing_sites', choices: 'A\nB\nC',
        description: ")
    }
    stages {
        stage ('Testing') {
            steps {
                // Do something
                sh 'ssh ${params.testing_sites}'
            }
        }
    }
}
```

File

A `file` parameter allows you to specify a file to upload that your Pipeline might require.

A `Pipeline {}` Block with a `File` Parameter:

```
pipeline {
    agent any
    parameters {
        file(name: 'name', description: 'file to upload')
    }
    stages {
        stage ('Testing') {
```

```
            steps {
                // Do something
            }
        }
    }
}
```

Triggers

The `triggers` directive defines the various ways to trigger a Pipeline.

Pipelines created using Jenkins Blue Ocean may not require triggers, since such Pipelines are triggered using webhooks configured on the source code repository (Git/GitHub/GitLab).

Following are the two `triggers` directives that might be useful, in some scenarios, for your *Continuous Delivery* pipeline.

Cron

The `cron` trigger accepts a cron-style string to define a regular interval at which the Pipeline should be re-triggered, for example:

```
pipeline {
    agent any
    triggers {
        cron('H */4 * * 1-5')
    }
    stages {
        stage('Long running test') {
            steps {
                // Do something
            }
        }
    }
}
```

Upstream

The `upstream` trigger allows you to define a comma-separated list of jobs and a threshold. When any of the jobs from the list finishes with the defined threshold, your Pipeline gets triggered. This feature is useful if your CI and CD Pipelines are two separate Jenkins Pipelines.

A `Pipeline {}` Block with an `Upstream` Trigger Directive:

```
pipeline {
    agent any
    triggers {
        upstream(upstreamProjects: 'jobA', threshold: hudson.
        model.Result.SUCCESS) }
    }
    stages {
        stage('Say Hello') {
            steps {
                echo 'Hello.'
            }
        }
    }
}
```

Stage

A Pipeline stage contains one or more `steps` to achieve a task, for example: Build, Unit-Test, Static Code Analysis, etc. The `stage` directive contains one or more `steps` sections, an `agent` section (optional), and other stage-related directives.

In the previous sections, you have already seen many examples demonstrating the `stage` directive. It is possible to run multiple stage directives in sequence, in parallel, or a combination of both. You'll see it shortly.

Tools

A `tools` directive allows you to install a tool on your agent automatically. It can be defined inside a `pipeline {}` or a `stage {}` block.

A `Pipeline {}` Block with a `Tools` Directive:

```
pipeline {
    agent any
    tools {
        maven 'apache-maven-3.0.1'
    }
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean verify'
            }
        }
    }
}
```

For the tools directive to work, the respective tool must be pre-configured in Jenkins under **Manage Jenkins ➤ Global Tool Configuration**. However, the tools directive gets ignored if you put an `agent none` section in the `pipeline {}` block.

As of today, only the following tools work in Declarative Pipeline:

- maven
- jdk
- gradle

Input

The `input` directive allows the Pipeline to prompt for input. The `input` directive works with the `stage` directive. When used, the `stage` directive pauses and wait for your input, and when you provide the input, the stage then continues to run.

The following table shows some of the options that are available with the input directive.

Options	Description	Required
message	A message to describe the input.	Yes
id	An identifier to identify the input. Its default value is the stage name.	No
ok	Alternative text for the "ok" button. Its default value is "ok."	No
submitter	A comma-separated list of users or external group names that are allowed to submit the input. Not using the submitter option allows any user to answer the input.	No
submitterParameter	A name of an environment variable to set with the submitter name, if present.	No
parameters	A list of parameters to prompt for, along with the input. See the parameters section for more details.	No

```
pipeline {
    agent any
    stages {
        stage('Smoke Testing') {
```

```
            steps {
                // Perform smoke testing that's quick
                echo 'Performing smoke testing.'
            }
        }
        stage('Full Integration Testing') {
            input {
                message "Should we go ahead with full
                integration testing?"
                ok "Yes"
                submitter "Luke,Yoda"
                parameters {
                    string(name: 'simulators', defaultValue: '4',
                    description: 'Test farm size')
                }
            }
            steps {
                echo "Running full integration testing using
                ${simulators} simulators."
            }
        }
    }
}
```

When

The `when` directive allows your Pipeline to decide whether a `stage` directive should be run based on some conditions.

The `when` directive must contain at least one condition. If the `when` directive contains more than one condition, all the child conditions must return true for the stage to execute.

Branch

Run a `stage` directive only if the branch that's being built matches the branch pattern provided.

A `Pipeline {}` Block Containing `When` Directive with `Branch` Condition:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Performinging a Build.'
            }
        }
        stage('Performance Testing') {
            when {
                branch "release-*"
            }
            steps {
                echo 'Performing performance testing.'
            }
        }
    }
}
```

The *Performance Testing* stage in this Pipeline runs for all branches that start with a `release-`string. Example: `release-beta`, `release-1.0.0.1`.

buildingTag

The `buildingTag` option allows you to run a stage only if the Pipeline is building a tag.

A `Pipeline {}` Block Containing `When` Directive with a `buildingTag` Condition:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            when {
                buildingTag()
            }
            steps {
                echo 'Performing a Build.'
            }
        }
    }
}
```

Tag

The `tag` option allows you to run a `stage` directive only if the `TAG_NAME` variable matches the pattern provided.

If you provide an empty pattern, and your Pipeline is building some tag, then the `tag` option behaves the same as a `buildingTag()`.

A `Pipeline {}` Block Containing `When` Directive with `Tag` Condition:

```
pipeline {
    agent any
    stages {
        stage('Publish') {
```

```
            when {
                tag "release-*"
            }
            steps {
                echo 'Publishing product package.'
            }
        }
    }
}
```

To the `tag` option, you can add a `comparator` parameter to define how a given pattern must get evaluated. The options for the `comparator` parameter include `EQUALS` for simple string comparison, `GLOB` (the default) for an ANT style path glob, or`REGEXP` for regular expression matching.

A `Pipeline {}` Block Containing a `When` Directive, with `Tag` Condition and a `Comparator`:

```
pipeline {
    agent any
    stages {
        stage('Publish') {
            when {
                tag pattern: "release-\d{1}\.\d{1}.\d{1}.\d{1}-
                nightly", comparator: "REGEXP"
            }
            steps {
                echo 'Publishing product package.'
            }
        }
    }
}
```

The *Publish* stage in the above Pipeline runs if the tags being built match anything between `release-0.0.0.0-nightly` and `release-9.9.9.9-nightly`.

changelog

The `changelog` option allows you to execute a `stage` directive only if the Pipeline's SCM changelog contains a specific regular expression pattern. The SCM changelog contains information about the list of modified files, the branch of the repository where the respective files were modified, the user who made the changes, and more.

A `Pipeline {}` Block Containing a `When` Directive, with `changelog` Condition:

```
pipeline {
    agent any
    stages {
        stage('Publish') {
            when {
                changelog '\sbugfix\s'
            }
            steps {
                echo 'Publishing product package.'
            }
        }
    }
}
```

changeset

The `changeset` option allows you to execute a `stage` directive if the Pipeline's SCM changeset contains one or more files matching the given string or glob. The SCM changeset is a list of modified files relating to a commit.

A `Pipeline {}` Block Containing a `When` Directive, with `changeset` Condition:

```
pipeline {
    agent any
    stages {
        stage('Publish') {
            when {
                changeset "**/*.js"
            }
            steps {
                echo 'Publishing product package.'
            }
        }
    }
}
```

By default the matching is case-insensitive; to make case-sensitive, use the `caseSensitive` parameter.

```
when { changeset glob: "Simlulator_SAS.*", caseSensitive: true
}
```

Environment

The `environment` option allows you to execute a `stage` directive only when a specific environment variable is set to the given value. To give you an example use-case, let us assume that you have multiple geographical regions where you are required to deploy your successfully tested artifacts—let's say, apac and japac. And, for all these environments, the deployment steps are different. So, using the `when` directive with `environment` condition, you can design your pipeline as shown here.

A `Pipeline {}` Block Containing a `When` Directive, with `Environment` Condition:

```
pipeline {
    agent any
    stages {
        stage('Deploy') {
            when {
                environment name: 'target', value: 'apac'
            }
            steps {
                echo 'Performing package deployment.'
            }
        }
    }
}
```

Equals

The `equals` option allows you to execute a `stage` directive only when an expected value equals the real value.

A `Pipeline {}` Block Containing a `When` Directive, with `Equals` Condition:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            when {
                equals expected: 2, actual: currentBuild.number
            }
```

```
            steps {
                echo 'Performing a Build.'
            }
        }
    }
}
```

Expression

The `expression` option allows you to run a stage only when the specified Groovy expression holds true.

A `Pipeline {}` Block Containing a `When` Directive, with `Expression` Condition:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Performing a Build.'
            }
        }
        stage('Static Code Analysis') {
            when {
                expression { BRANCH_NAME ==~ /(feature-*)/ }
            }
            steps {
                echo 'Performing Static Code Analysis.'
            }
        }
    }
}
```

It is possible to build more complex conditional structures using the nesting conditions: `not`, `allOf`, or `anyOf`. Moreover, it's possible to nest them to any depth.

not

Use the `not` option to execute a `stage` directive when all of the nested conditions are true and must contain at least one condition.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Performing a Build.'
            }
        }
        stage('Deploy') {
            when {
                not { branch 'master' }
            }
            steps {
                echo 'Deploying artifacts.'
            }
        }
    }
}
```

allOf

Use the `allOf` option to execute a `stage` directive when all of the nested conditions are true and must contain at least one condition.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Performing a Build.'
            }
        }
        stage('Example Deploy') {
            when {
                allOf {
                    branch 'production'
                    environment name: 'region', value: 'apac'
                }
            }
            steps {
                echo 'Deploying artifacts.'
            }
        }
    }
}
```

anyOf

Use the `anyOf` option to execute a `stage` directive when at least one of the nested conditions is true and must contain at least one condition.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Performing a Build.'
            }
        }
        stage('Static Code Analysis') {
            when {
                anyOf {
                    branch 'master'
                    branch 'dev'
                }
            }
            steps {
                echo 'Performing Static Code Anlaysis.'
            }
        }
    }
}
```

Evaluate When Before Entering the Stage's Agent

By default the `when` condition for a `stage` directive gets evaluated after entering the agent defined for the stage.

To change this behavior, use the `beforeAgent` option within the `when {}` block.

```
pipeline {
    agent none
    stages {
        stage('Build') {
            agent {
                label 'ubuntu_maven'
            }
            when {
                beforeAgent true
                branch 'master'
            }
            steps {
                echo 'Performing a build.'
            }
        }
    }
}
```

Sequential Stages

Multiple `stage` directives in Declarative Pipeline can be defined in sequence. A `stage` directive can contain only one `steps {}`, `parallel {}`, or `stages {}` block.

Simple Sequential Stages

Figure 4-1 describes a simple sequential flow of multiple `stage` directives.

Figure 4-1. *Simple Sequential Stages*

Following is the Pipeline code for the Pipeline flow described in Figure 4-1.

An Example Pipeline Code with Simple Sequential Stages:

```
pipeline {
    agent {
      label 'master'
    }
    stages {
        stage('Stage 1') {
            steps {
                echo "Running Stage 1"
            }
        }
        stage('Stage 2') {
            steps {
                echo "Running Stage 2"
            }
        }
        stage('Stage 3') {
            steps {
                echo "Running Stage 3"
            }
        }
    }
}
```

The `pipeline {}` block compulsory has a `stages {}` block. All the `stage` directives that should run in the sequence are defined inside the `stages {}` block, one after the other.

Nested Sequential Stages

In Declarative Pipeline, it is possible to nest multiple `stage` directives that are in sequence inside another `stage` directive. Figure 4-2 describes multiple nested `stage` directives that are in sequence.

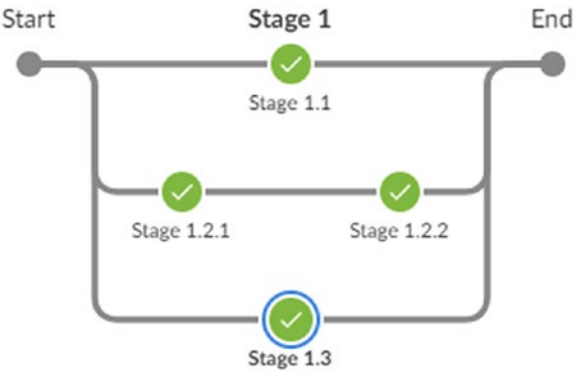

Figure 4-2. *Nested Sequential Stages*

Following is the Pipeline code for the Pipeline flow described in Figure 4-2.

An Example Pipeline Code with Nested Sequential Stages:

```
pipeline {
    agent {
      label 'master'
    }
    stages {
        stage('Stage 1') {
            parallel {
                stage('Stage 1.1') {
                    steps {
                        echo "Running Stage 1.1"
                    }
                }
```

```
                stage('Stage 1.2') {
                    stages {
                        stage('Stage 1.2.1') {
                            steps {
                                echo "Running Stage 1.2.1"
                            }
                        }
                        stage('Stage 1.2.2') {
                            steps {
                                echo "Running Stage 1.2.2"
                            }
                        }
                    }
                }
                stage('Stage 1.3') {
                    steps {
                        echo "Running Stage 1.3"
                    }
                }
            }
        }
    }
}
```

Notice the two `stage` directives that are in sequence (highlighted in bold) are not placed directly inside the parent `stage ('Stage 1.2') {...}` block. But, they are first added to a `stages {}` block, which is them placed inside the `stage ('Stage 1.2') {...}` block.

Parallel Stages

It is possible to define multiple `stage` directives in Declarative Pipeline to run in parallel with one another. A `stage` directive can contain only one `steps {}`, `parallel {}`, or `stages {}` block. Also, the nested stages cannot contain further parallel stages.

A `stage` directive that contains a `parallel {}` block cannot contain `agent` or `tools` sections since they are not applicable without the `steps {}` block.

Simple Parallel Stages

Figure 4-3 describes a simple parallel flow of a multiple `stage` directive.

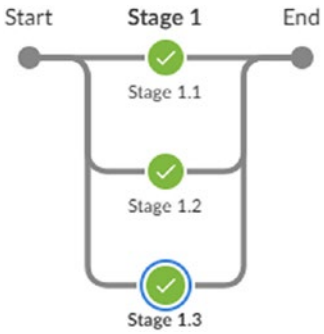

Figure 4-3. *Simple Parallel Stages*

Following is the Pipeline code for the Pipeline flow described in Figure 4-3.

An Example Pipeline Code for with Simple Parallel Stages:

```
pipeline {
    agent {
      label 'master'
    }
```

```
    stages {
        stage('Stage 1') {
            parallel {
                stage('Stage 1.1') {
                    steps {
                        echo "Running Stage 1.1"
                    }
                }
                stage('Stage 1.2') {
                    steps {
                        echo "Running Stage 1.2"
                    }
                }
                stage('Stage 1.3') {
                    steps {
                        echo "Running Stage 1.3"
                    }
                }
            }
        }
    }
}
```

The `pipeline {}` block compulsory has a `stages {}` block. All the `stage` directives that should run in parallel are defined one after the other inside a `parallel {}` block. The `parallel {}` block is then defined inside the parent `stage {}` block.

Nested Parallel Stages

In Declarative Pipeline, it is possible to nest multiple `stage` directives that are in parallel inside another `stage` directive. Figure 4-4 describes multiple nested `stage` directives that are in parallel.

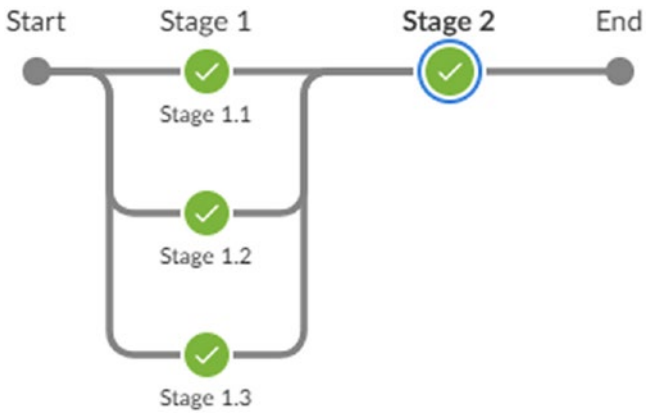

Figure 4-4. *Nested Parallel Stages*

Following is the Pipeline code for the Pipeline flow described in Figure 4-4.

An Example Pipeline Code with Nested Parallel Stages:

```
pipeline {
    agent {
      label 'master'
    }
    stages {
        stage('Stage 1') {
            parallel {
                stage('Stage 1.1') {
                    steps {
                        echo "Running Stage 1.1"
                    }
                }
                stage('Stage 1.2') {
                    steps {
                        echo "Running Stage 1.2"
                    }
                }
```

```
                stage('Stage 1.3') {
                    steps {
                        echo "Running Stage 1.3"
                    }
                }
            }
        }
        stage('Stage 2') {
            steps {
                echo "Running Stage 2"
            }
        }
    }
}
```

Steps

There are a large number of steps available for Jenkins. You can find them at https://jenkins.io/doc/pipeline/steps/.

Every new Jenkins Plugin that's compatible with the Scripted Pipeline adds a new step to the list of steps. However, not all of the steps available for Jenkins are compatible with the Declarative Pipeline. The ones that are compatible are available as a `step` inside the *Visual Pipeline Editor*.

To make the incompatible steps work with Declarative Pipeline, you need to enclose them inside the `script {}` block, also called the `script` step.

However, if you are creating a pipeline using the *Visual Pipeline Editor*, the `script` step is available as *Run Arbitrary Pipeline Script* inside the list of Steps.

Script

In simple terms, the `script {}` block allows you to execute a Scripted Pipeline inside a Declarative Pipeline.

Shown here is an example of Declarative Pipeline code with a `script` step containing an Artifactory upload filespec.

A `Pipeline {}` Block with a `Script` Step:

```
pipeline {
    agent {
      label 'master'
    }
    stages {
        stage('Build') {
            steps {
                echo "Running a Build."
            }
        }
        stage('Publish') {
            steps {
                script {
                    // Code to upload artifacts to Artifactory
                    server
                    def server = Artifactory.server
                    'my-server-id'
                    def uploadSpec = """{
                      "files": [
                        {
                          "pattern": "bazinga/*froggy*.zip",
                          "target": "bazinga-repo/froggy-files/"
                        }
                     ]
                    }"""
```

```
                    server.upload(uploadSpec)
                }
            }
        }
    }
}
```

There is no limit to the size of Scripted Pipeline you can run inside the `script {}` block. However, it is recommended to move anything bigger than a few lines to Shared Libraries. Also, any piece of code that is common across multiple pipelines must be moved to Shared Libraries.

In the final chapter, you'll learn to extend your Jenkins Blue Ocean pipelines or, shall we say, the Declarative Pipeline, using the `script` step and Shared Libraries.

Summary

The various options and features about the Declarative Pipeline Syntax discussed in this chapter are all that exist. In this chapter, you learned about the sections, directives, sequential, and parallel stages as well as steps in detail.

As I mentioned at the beginning of this chapter, Declarative Pipeline Syntax is restrictive in what it provides to the Pipeline authors. However, it's easy to understand and maintain.

There could be many problems that a Declarative Pipeline cannot directly solve, but its counterpart can. However, using the `script` step and Shared Libraries, you can fulfill the gap.

In the next chapter, you'll take a look at some tools that help you compose Declarative Pipeline code.

CHAPTER 5

Declarative Pipeline Development Tools

The following chapter is a short guide about various Editors, their plugins, and some native Jenkins tools that help you write a Declarative Pipeline code.

Auto-Completion, Syntax Highlighting, and Jenkinsfile validation are some of the features that you'll learn in the current chapter. If you are not sure what they are, here is a short description of them:

- **Auto-Completion**: a feature in which an application suggests the rest of the code a user is typing.
- **Syntax Highlighting**: a feature that displays source code in different colors and fonts according to the type of terms.
- **Jenkinsfile validation**: a feature that validated your Jenkinsfile for syntax and linting errors.

You'll also see some native Jenkins tools, such as the Snippet Generator and the Declarative Directive Generator, that help you in constructing a Declarative Pipeline code. So let's begin.

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5_5

Auto-Completion and Syntax Highlighting in Atom Editor

In the current section, you'll learn to install and use a few packages in Atom editor that will help you in writing your Jenkinsfile. You'll first learn to install and configure these packages and later learn to use them.

Installing the Package for Auto-Completion and Syntax Highlighting

To get the Auto-completion and Syntax Highlighting features in Atom editor, install the *language-jenkinsfile* package. The following are the steps:

1. Open Atom editor. I am using version 1.30.0 of Atom editor at the time of writing this book.
2. Open the settings page from **File ➤ Settings** or by typing **Ctrl+Comma**.
3. On the Settings page, click on **Install** from the left-side menu. Next, using the search field [1] look for *language-jenkinsfile*. See Figure 5-1.

Figure 5-1. *Installing the language-jenkinsfile package*

4. When you find the required package [2], click on **Install** button [3] to install.
5. The package is ready for use, and it doesn't need any further configuration.

Modifying the File config.cson

We would like the Atom editor to detect a Jenkinsfile. You'll do this by editing the file `config.cson` in Atom editor. Follow the below steps:

1. Open the file `config.cson` from **File ➤ Config...**
2. Append the following lines of code to the file `config.cson`.

```
'file-types':
'Jenkinsfile*': 'Jenkinsfile.xml'
```

Its recommended that you always name your Jenkinsfile as `Jenkinsfile` or `Jenkinsfile-<Some Text>`. This is because a Jenkinsfile does not come with an extension, and the Atom editor has to rely on searching the word `Jenkinsfile` in the file name instead.

Auto-Completion and Syntax Highlighting in Action

To see the Auto-Completion and Syntax Highlighting feature working, open a new project in Atom editor from **File ➤ Add Project Folder...** alternatively, click **Ctrl+Shift+A**.

Next, create a new file (Jenkinsfile) inside your projects root folder by typing **A**. You'll get a prompt asking you to provide a name for your new file; type `Jenkinsfile`. Make sure the project root folder is selected while you do this, or the new file might get created somewhere else.

After adding the new file (Jenkinsfile), Atom editor should automatically detect it as a Jenkinsfile. To begin with, type in some Declarative sections, and you should get the auto-completion suggestions (see Figure 5-2).

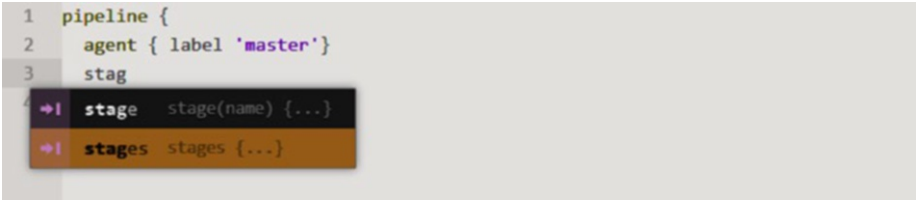

Figure 5-2. *Auto-Completion and Syntax Highlighting in Atom editor*

Learn more about the *language-jenkinsfile* package for Atom editor at `https://github.com/BastienAr/language-jenkinsfile`.

Syntax Highlighting and Jenkinsfile Validation in Visual Studio Code

In the current section, you'll learn to install and use a few extensions for Visual Studio Code that help you in writing your Jenkinsfile. You'll first learn to install and configure these extensions and later learn to use them.

Installing the Extension for Syntax Highlighting

To get the Syntax Highlighting features in Visual Studio Code, install the *Declarative Jenkinsfile Support* extension. The following are the steps:

1. Open the Visual Studio Code Editor. I am using version 1.27.2 of Visual Studio Code editor at the time of writing this book.
2. Open the Extensions page from **File ➤ Preferences ➤ Extensions** alternatively, type **Ctrl+Shift+X**.
3. Next, using the field **Search Extensions in Marketplace**, look for *Declarative Jenkinsfile Support* [1]. Once you find it [2], click on the **Install** button [3] to install. Once installed, click on the **Reload** button [4] to load the extension (see Figure 5-3).

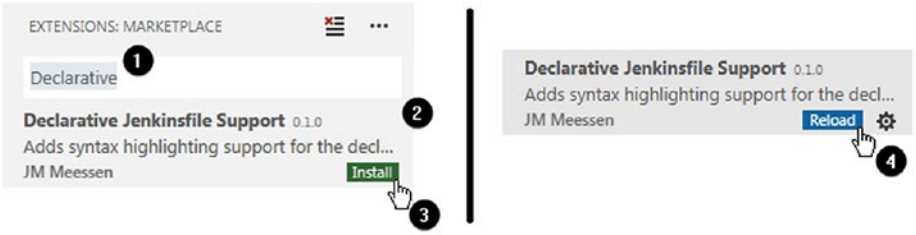

Figure 5-3. *Installing the Declarative Jenkinsfile Support extension*

4. The extension is ready for use, and it doesn't need any further configuration.

Installing the Extension for Jenkinsfile Validation

Similarly, to get the Jenkinsfile validation features in Visual Studio Code, install the *Jenkins Pipeline Linter Connector* extension. Following are the steps:

1. Open the Extensions page from **File ➤ Preferences ➤ Extensions** alternatively, type **Ctrl+Shift+X**.
2. Next, using the field **Search Extensions in Marketplace**, look for *Jenkins Pipeline Linter Connector* [1]. Once you find it [2], click on the **Install** button [3] to install. Once installed, click on the **Reload** button [4] to load the extension (see Figure 5-4).

Figure 5-4. *Installing the Jenkins Pipeline Linter Connector extension*

3. The following extension, however, needs some configuration before you start using it. Let's see that in the next section.

Modifying the File settings.json

The Jenkinsfile validation process in any editor works by connecting to a Jenkins server. For the Jenkins Pipeline Linter Connector extension to work, you are required to add your Jenkins server details inside the file `setting.json` in Visual Studio Code.

1. Open Visual Studio Code.
2. Open the settings page from **File ➤ Preferences ➤ Settings;** alternatively, type **Ctrl+Comma**.
3. On the settings page, click on **More Actions...** [1], and choose **Open settings.json** (see Figure 5-5).

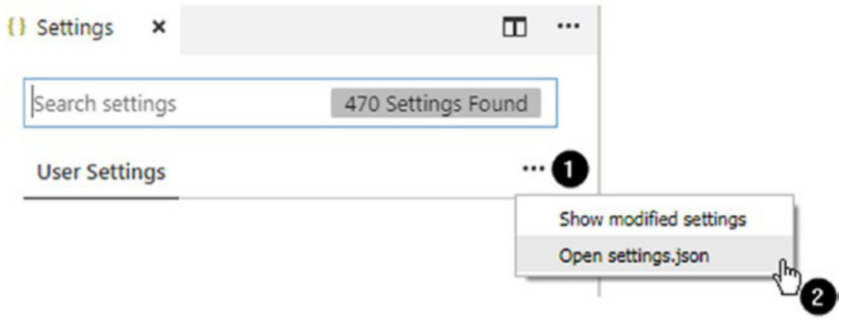

Figure 5-5. *Opening the file settings.json*

4. On the resultant page, you'll see **DEFAULT USER SETTINGS** on your left side and **USER SETTINGS** on the right side. Place the following line of code to the **USER SETTINGS** inside the {}. These settings allow Visual Studio Code to connect with a Jenkins server to validate your Jenkinsfile.

```
"jenkins.pipeline.linter.connector.url": \
"<Jenkins Server URL>/pipeline-model-converter/
validate",
"jenkins.pipeline.linter.connector.user": "<Jenkins
Username>",
"jenkins.pipeline.linter.connector.pass": "<Jenkins
Password or API Key>",
```

```
"jenkins.pipeline.linter.connector.crumbUrl": \
"<Jenkins Server URL>/crumbIssuer/api/xml?xpath= \
concat(//crumbRequestField,%22:%22,//crumb)",
```

5. To the same **USER SETTINGS,** add the following line of code inside the {}. This will allow Visual Studio Code to associate any given file named `Jenkinsfile` or `Jenkinsfile-<Some Text>` as *JenkinsfileDeclarativePipeline*.

```
"files.associations": {
     "Jenkinsfile*": "declarative"
 }
```

Syntax Highlighting and Jenkinsfile Validation in Action

To see the Syntax Highlighting and Jenkinsfile Validation feature working, open a new folder in Visual Studio Code from **File ➤ Open Folder...;** alternatively, click **Ctrl+K** or **Ctrl+O**.

Next, create a new file (Jenkinsfile) inside your project's root folder. You'll get a prompt asking you to provide a name for your new file; type `Jenkinsfile`.

After adding the new file (Jenkinsfile), Visual Studio Code should automatically detect it as a Jenkinsfile. To begin with, type in some Declarative Pipeline code, and you should see the syntax highlighting.

To validate your Jenkinsfile, deliberately make a mistake; for example, delete a closing or opening bracket. Save the file by typing Ctrl+S. Then, type **Shift+Alt+V** to validate (see Figure 5-6).

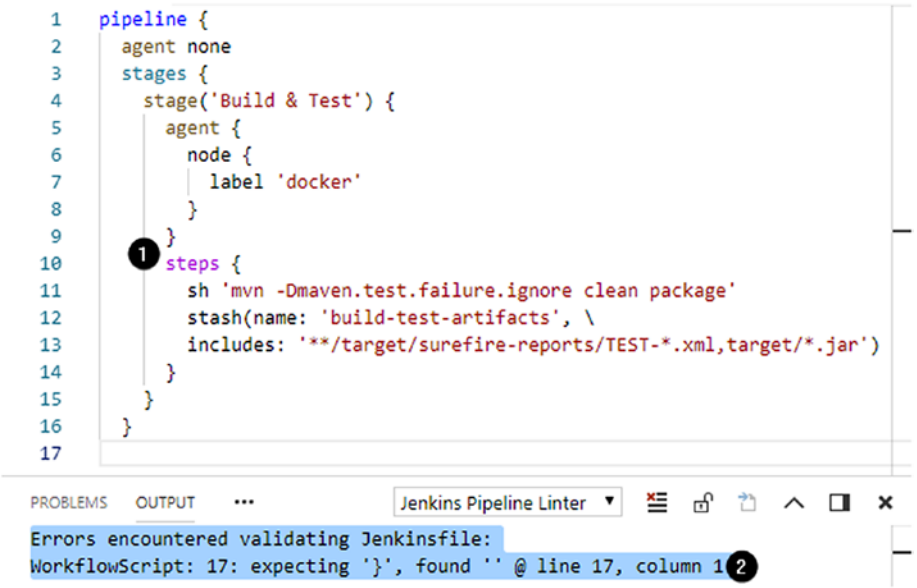

Figure 5-6. *Syntax Highlighting and Jenkinsfile validation in Visual Studio Code*

Learn more about the *Declarative Jenkinsfile Support* extension for Visual Studio Code at https://github.com/jmMeessen/vsc-jenkins-declarative.

Learn more about the *Jenkins Pipeline Linter Connector* extension for Visual Studio Code at https://github.com/janjoerke/vscode-jenkins-pipeline-linter-connector.

Auto-Completion, Syntax Highlighting, and Jenkinsfile Validation in Eclipse IDE

In the current section, you'll learn to install and use a plugin for Eclipse IDE that helps you in writing your Jenkinsfile. You'll first learn to install and configure this plugin and later learn to use it.

Installing the Plugin to Auto-Completion, Syntax Highlighting, and Jenkinsfile Validation

To get the Auto-Completion, Syntax Highlighting, and Jenkinsfile Validation features in Eclipse IDE, install the *Jenkins Editor* plugin. Following are the steps:

1. Open Eclipse IDE. I am using Photon Release (4.8.0) of Eclipse IDE at the time of this writing.
2. To install the *Jenkins Editor* plugin, open the *Eclipse Marketplace* from **Help ➤ Eclipse Marketplace**.
3. Search for Jenkins Editor using the field **Find:** [1]. Once you find it [2], click on the **Install** button [3] to install (see Figure 5-7).

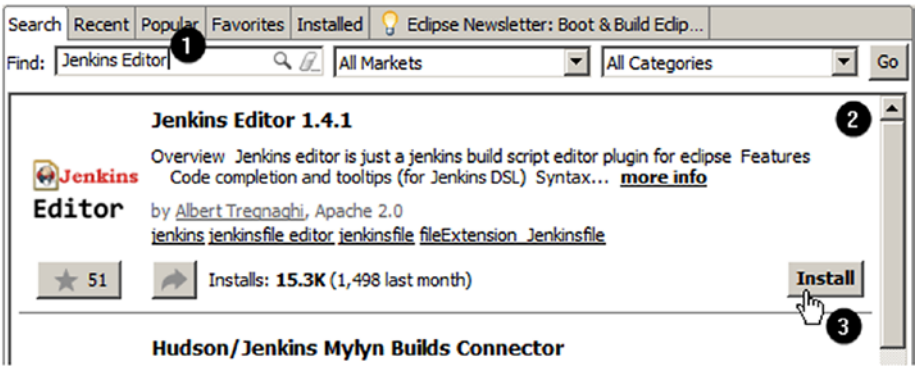

Figure 5-7. *Syntax Highlighting and Jenkinsfile validation in Visual Studio Code*

4. You'll be asked to review the licenses. Accept the terms and conditions and click on the **Finish** button.

Modifying the Jenkins Editor Plugin Settings

The Jenkinsfile validation process in any editor works by connecting to a Jenkins server. For the *Jenkins Editor* plugin to work, you are required to add your Jenkins server details inside the file Jenkins Editor plugin settings. Follow the below steps:

1. Open Eclipse IDE, and navigate to **Window ➤ Preferences ➤ Jenkins Editor**, and configure the settings as shown in Figure 5-8. All the settings are self-explanatory.

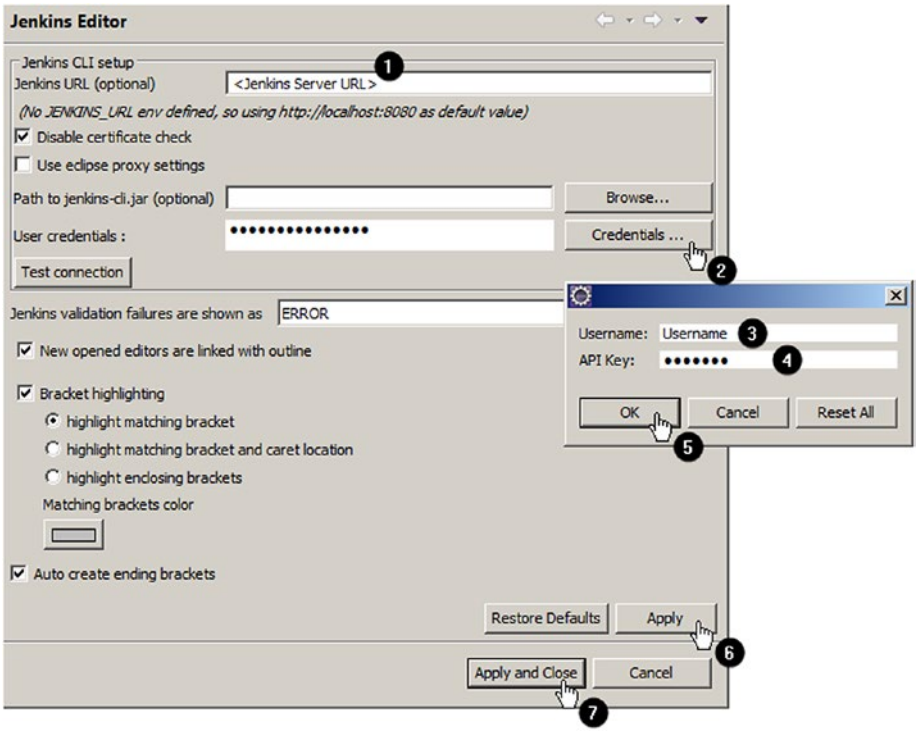

Figure 5-8. *Jenkins Editor plugin settings in Eclipse IDE*

2. Before saving the settings, you can click on the Test connection button to test if Eclipse IDE can communicate with your Jenkins server.

Auto-Completion, Syntax Highlighting, and Jenkinsfile Validation in Action

Let's see the Auto-Completion, Syntax Highlighting, Jenkinsfile Validation feature in action. Open an existing project using Eclipse IDE from **File ➤ Open Projects from File System...**.

Next, create a new file (Jenkinsfile) inside your project's root folder. To do so, right-click on your project's root and select **New ➤ File**. You'll get a prompt asking you to provide a name for your new file; type `Jenkinsfile`, and click the **Finish** button.

After adding the new file (Jenkinsfile), Eclipse IDE should automatically detect it as a Jenkinsfile. To begin with, click anywhere inside the empty file and type **Ctrl+Shift** or **Ctrl+P**. You should see the Auto-completion suggestions [2]. You'll also see a short description of the code snippet that you choose [3] (see Figure 5-9).

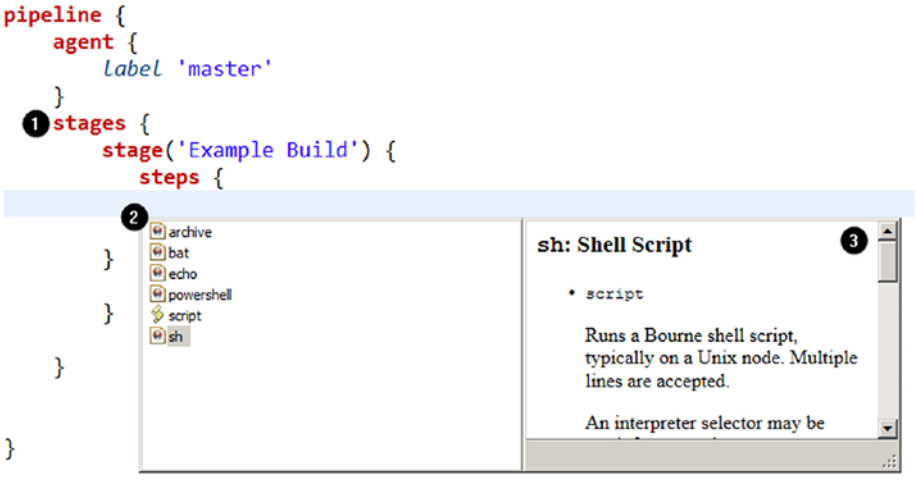

Figure 5-9. *Auto-Completion, Syntax Highlighting in Eclipse IDE*

Now, try to construct some Declarative pipeline code, and you should see the syntax highlighting [1].

To validate your Jenkinsfile, deliberately make a mistake; for example, delete a closing or opening bracket. Save the file by typing Ctrl+S. You should immediately see a red cross-mark at the respective line number where there is an error (see Figure 5-10). Hover your mouse on the red cross [1], and it should display the error message [2].

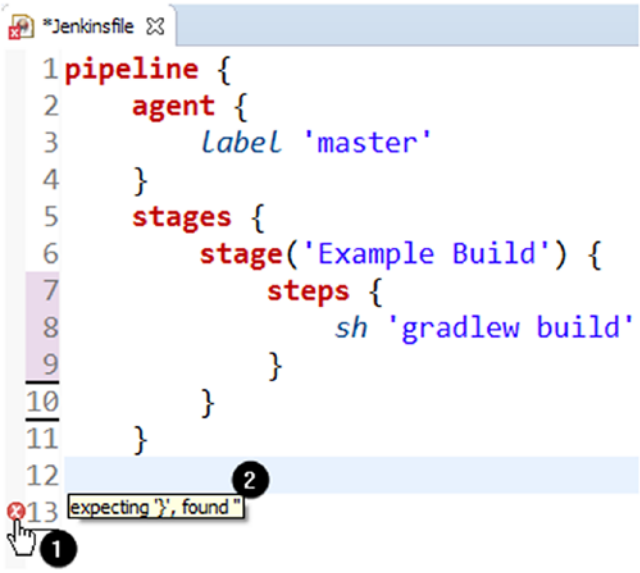

Figure 5-10. *Jenkinsfile Validation in Eclipse IDE*

Learn more about the *Jenkins Editor* plugin for Eclipse IDE at https://github.com/de-jcup/eclipse-jenkins-editor.

Declarative Directive Generator in Jenkins

The Directive Generator allows you to generate code for a Declarative Pipeline directive, such as agent, stage, when, and so on (see Figure 5-11).

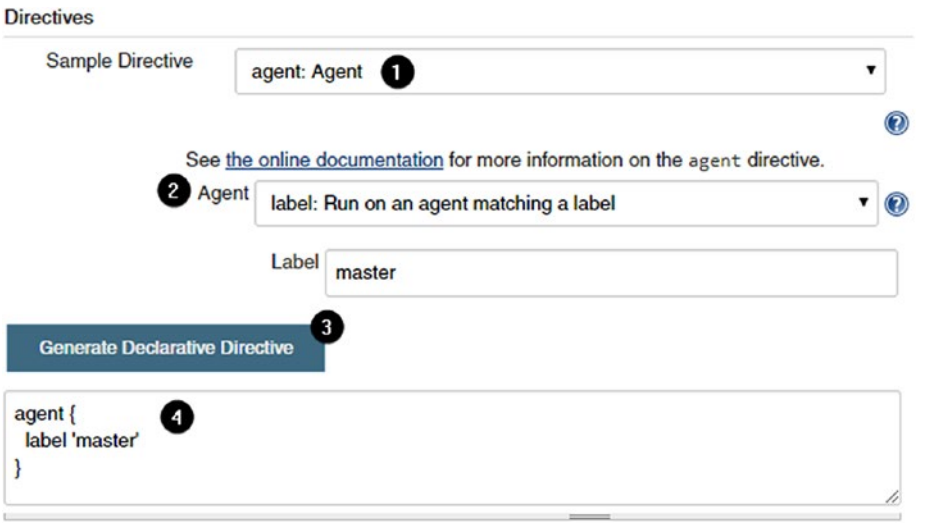

Figure 5-11. *The Declarative Directive Generator*

For this, you are required to choose the directive that you're interested in from the **Sample Directive** drop-down list [1]. Doing so results in a new form that gets displayed below it [2]. The new form contains lots of settings to configure depending on the directive chosen.

Once you've filled out all the options for your directive, click the **Generate Declarative Directive** button [3]. The Pipeline code appears in the box here [4]. You can later copy this generated code into your Jenkinsfile.

To access the Declarative Directive Generator, use the following URL: `<Jenkins Server URL>/job/<Any Jenkins Job Name>/pipeline-syntax/`, alternatively from the Jenkins Blue Ocean Dashboard click on any of your projects. Then, from the Project Dashboard, click on the **Configure** icon [1] (see Figure 5-12). A new tab opens up with the project's configuration page. From here, click on the project's drop-down menu options [2] and select **Pipeline Syntax** [3].

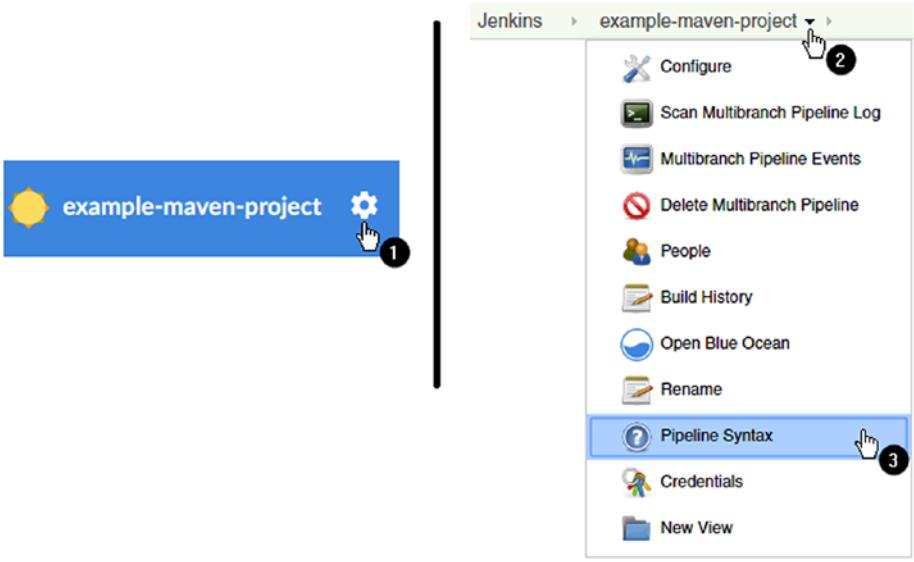

Figure 5-12. *Accessing the Declarative Directive Generator*

Figure 5-13 shows the list of all the Sections and Directives available using the Declarative Directive Generator.

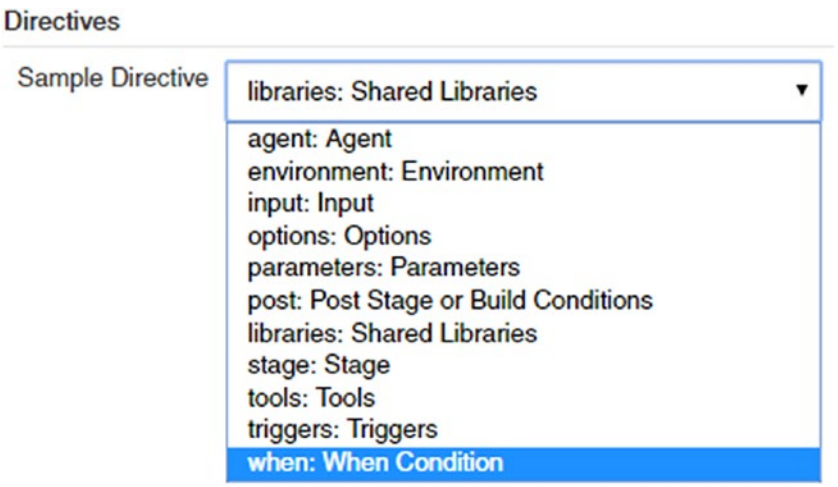

Figure 5-13. *The list of sections and directives available with Declarative Directive Generator*

Snippet Generator in Jenkins

The Snippet Generator allows you to generate code for Scripted Pipeline elements. The list of available options is directly proportional to the installed plugins that support Scripted Pipeline (see Figure 5-14).

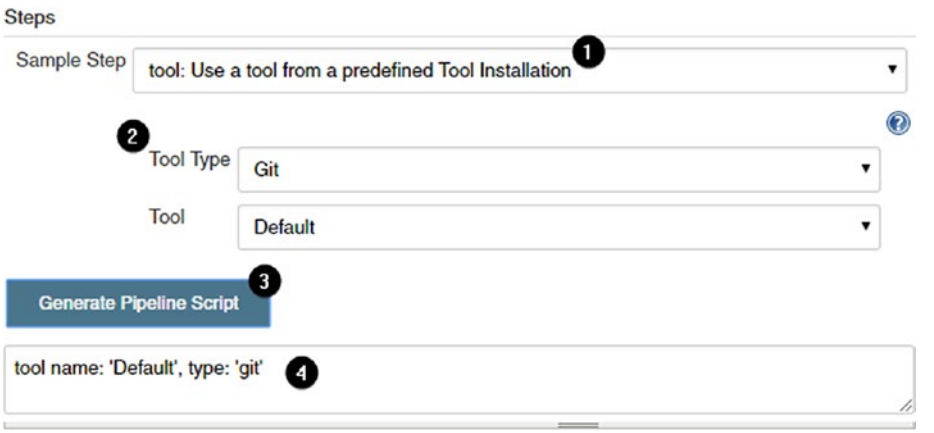

Figure 5-14. *The Declarative Directive Generator*

For this, you are required to choose the step that you're interested in from the **Sample Step** drop-down list [1]. Doing so results in a new form that gets displayed below it [2]. The new form contains lots of settings to configure depending on the step chosen.

Once you've filled out all the options for your step, click the **Generate Pipeline Script** button [3]. The Pipeline code appears in the box here [4]. You can later copy this generated code into your Jenkinsfile inside the script {} block. Remember, the code generated using the Snippet Generator always gets inside the `script {}` block of your Declarative Pipeline. If you're using a Scripted Pipeline, then it's a different case.

JENKINSFILE VALIDATION USING COMMAND-LINE

In the previous sections you saw various text editors that help you with Auto-Completion, Syntax Highlighting, and Jenkinsfile validation.

But, what if you don't use any of the tools described in the current chapter? I am not sure about Auto-Completion and Syntax Highlighting, but, there is for sure a command-line option to validate your Jenkinsfile. All you need is a Jenkins server.

The thing is, when you start composing a Jenkinsfile, it is quite natural to make mistakes with the pipeline syntax. And it's quite frustrating to see your pipeline failing due to silly typos. Therefore, it's always a good idea to check your Jenkinsfile for linting errors. In the current exercise you will validate your Jenkinsfile using *curl* and *HTTP POST* request.

1. curl comes installed by default on a Linux machine. But, if you are working from a Windows machine, then you need to install it. Download and Install curl from `https://curl.haxx.se/windows/`.
2. Execute the command `curl --version` to see if its installed.

3. Make a note of your Jenkins URL.
4. Also, get a *Jenkins Crumb*. According to Jenkins, *Crumbs* are hashes that include information that uniquely identifies an agent that sends a request, along with a guarded secret so that the crumb value cannot be forged by a third party.
5. To get a Jenkins crumb, execute the following command:

```
curl "<Jenkins server URL>/crumbIssuer/api/xml? \
xpath=concat(//crumbRequestField,\":\",//crumb)"
```

6. Now, with your `<Jenkins Server URL>`, your `<Path to your Jenkinsfile>`, and your `<Jenkins Crumb>` construct and execute the following command:

```
curl -X POST -H <Jenkins Crumb> -F \
"jenkinsfile=<<Path to your Jenkinsfile>" \
<Jenkins Server URL>/pipeline-model-converter/validate
```

Example:

```
curl -X POST -H jaskdsad93ewewdqdqdk -F \
 "jenkinsfile=</home/dev/Jenkinsfile" \
 http://jenkinsurl.com/pipeline-model-converter/validate
```

Example output of this command:

```
Jenkinsfile successfully validated.
```

If your linter output returned `Jenkinsfile successfully validated`, then it's all good. Otherwise, you will get a detailed explanation about the typo in your Jenkinsfile.

Summary

If you like writing your Jenkinsfile in Declarative Syntax, then you must use the development tools described in this chapter.

Sometimes to a developer it's convenient to write a Jenkinsfile in text rather than using the Visual Pipeline Editor, and in such cases, the editors described in the current chapter along with their respective plugins come in handy.

If your favorite text editor does not support Auto-Completion, Syntax Highlighting, and Validation of Jenkinsfile, then you can always use the native Jenkins tools, such as the Declarative Directive Generator and the Snippet Generator.

In the next chapter, you'll learn more about the Snippet Generator and also writing some advance pipelines using the Jenkins Shared Libraries.

CHAPTER 6

Working with Shared Libraries

The concept of having pipelines as a code brings convenience and handiness in maintaining and designing CI/CD pipelines. I need not speak about its advantages, because by now they are already apparent to you. It all came with Jenkins 2.0 and helped everyone write complex and flexible CI/CD pipelines with ease.

It is a significant improvement over the massive, GUI-based CI/CD pipelines comprised of multiple freestyle pipeline jobs. Blue Ocean further made things better by allowing users to design CI/CD pipeline with ease using its so-called Visual Pipeline Editor. The Declarative Pipeline Syntax used by Blue Ocean also needs some applause.

So most of us now who follow CI/CD in our organizations have got a few pipelines projects running; some of us may even have hundreds of pipelines projects. So what's next? How do you avoid repeating the same common code across different pipelines? Shared Libraries is the answer, and the current chapter is all about it.

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5_6

In this you'll learn about the following:

- Why Use Shared Libraries?
- How Do Shared Libraries Work?
- Retrieving Shared Libraries
- Using Shared Libraries Inside Your Pipeline
- Creating Shared Libraries

Using Shared Libraries, you can take additional advantage of having pipeline as a code. The idea behind Shared Libraries is to whip out the re-usable code carefully and store them as separate pipeline libraries that can be re-used by multiple pipelines on demand. In the following section, you'll learn in detail about Jenkins Shared Libraries. So let's begin.

Why Use Shared Libraries?

Let's say there are multiple CI/CD pipelines projects in your organization, all of which have a static code analysis stage in them (using SonarQube).

Instead of writing pipeline code (Declarative/Scripted Pipeline) for static code analysis for each pipeline, it makes more sense to separate the common code out intelligently, store it as a piece of pipeline library in a common place (that acts more like a function), and summon it on demand in all your CI/CD pipelines projects. In this way, it becomes possible to re-use a piece of code and avoid redundancy in your CI/CD infrastructure.

In the SonarQube example, I have assumed that we use the project configuration file (sonar-*project.properties*), one for every pipeline project to describe their static code analysis configuration.

There are many such examples that you can imagine. Anything that you feel could be re-used should be put into a Shared Library. Over time, you'll have a collection of these reusable functions in your library. Shared Libraries make your code more readable and shareable.

How Shared Libraries Work?

Following are the four simple steps that describe how Shared Libraries work with your Pipeline.

Step 1: First, you initiate a Git Repository meant only to store Shared Libraries files.

Step 2: You then create a few Groovy scripts and add them to your Git Repository. In the upcoming section, *Creating Jenkins Shared Libraries*, you'll see how to write Groovy scripts that serve as Shared Libraries.

Step 3: Next, you configure your Shared Library's repository details inside Jenkins. In the upcoming section, *Retrieving Shared Libraries*, you'll see all the possible ways of configuring a Shared Library into Jenkins.

Step 4: Finally, you summon the Shared Library into your Pipeline. In the upcoming section, *Calling Shared Libraries Inside your Pipeline*, you'll see all possible ways of calling your Shared Library from your Pipeline.

The Directory Structure for Shared Libraries

The Jenkins Shared Library should follow the directory structure as described in Figure 6-1.

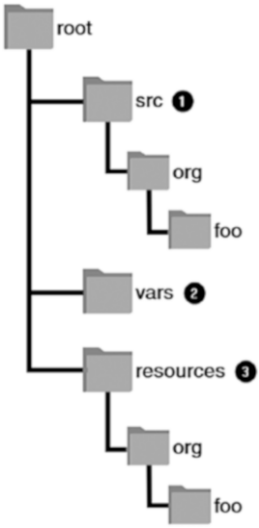

Figure 6-1. *The directory structure for a Jenkins Shared Library*

Let's learn about the directory structure.

The **src** directory [1] looks more like a standard Java source directory structure. This directory is added to the classpath when executing Pipelines. The available classes get loaded with an import statement.

The **vars** directory [2] can contain multiple Groovy files that define global variables. These global variables are accessible from your Pipeline using the `script {}` block. Similarly, the **vars** directory can hold your custom steps, where each of these steps get defined inside a Groovy script. The basename of each Groovy file should be a Groovy (~ Java) identifier that follows camelCase.[1]

When a Pipeline that uses a Jenkins Shared Library, comprising such elements, finishes running, the steps from the **vars** directory get listed under `<jenkins-url>/pipeline-syntax/globals` page. It's also possible

[1]Camel case: formalized as camelCase, also known as camel caps, is the practice of writing composite words or phrases in a way that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation.

to add a corresponding `.txt` file inside the library's **vars** directory, presenting additional documentation about it. In this way, you can provide extra info to the other members of your team about the step.

A **resources** directory [3] allows you to store additional non-Groovy/Java files that get loaded using the `libraryResource` step. The stuff from the **resources** directory is used from an external library.

Retrieving Shared Libraries

In the following section, you'll learn multiple ways to retrieve Shared Libraries. The first one is using the pre-configured settings inside Jenkins, and the other is through directly retrieving it from the Pipeline during runtime. Let's see both the methods in detail.

Retrieving Shared Libraries Using Pre-Configured Settings in Jenkins

In the following method, you tell Jenkins the name, location, credentials, and other parameters for retrieving a Shared Library. You can either do this at a global level by making necessary configurations inside the Jenkins global settings or at the Folder/Pipeline Project level. Nevertheless, the settings and configuration options remain the same at both levels.

The difference lies in their scope. Shared Libraries defined inside Jenkins global settings are available across all pipelines. On the other hand, Shared Libraries defined at the Folder/Pipeline Project level are available only to the pipelines inside the Folder or only to the pipelines belonging to a particular Pipeline Project, respectively. Let's see how to retrieve Shared Libraries using the Global settings in Jenkins.

1. From the Jenkins Blue Ocean Dashboard, click on the **Administration** link from the top menu bar. Once, you are on the **Manage Jenkins** page, click on the **Configure System link**.
2. On the **Configure System** page, scroll all the way down until you see the section **Global Pipeline Libraries**.
3. Click on the **Add** button to add a new library (see Figure 6-2). You can add as many Shared Libraries as you want.

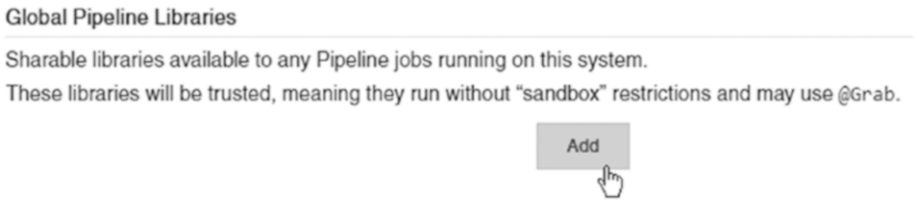

Figure 6-2. *Adding Global Pipeline Libraries*

4. After clicking the **Add** button, you'll see a few settings that you must configure (see Figure 6-3). Let's understand them one by one.

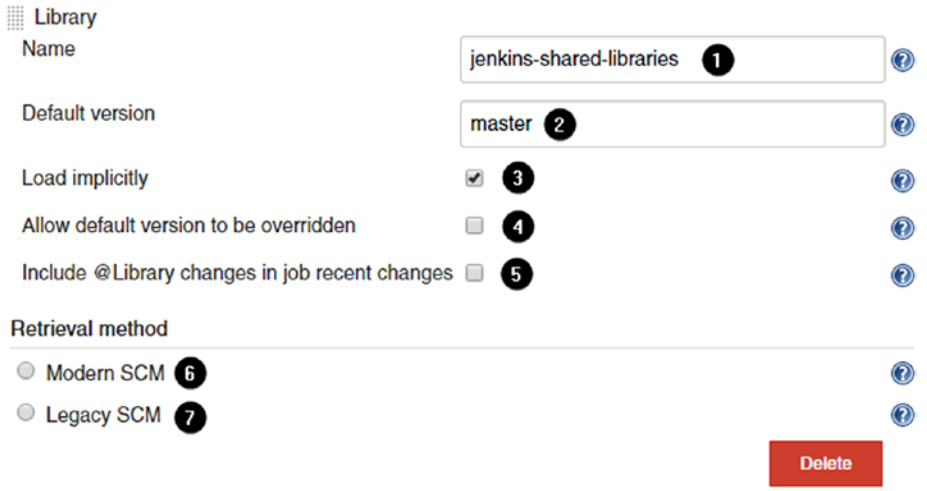

Figure 6-3. *Global Pipeline Libraries basic settings*

The **Name** field [1] is an identifier you pick for this library, to be used in the `@Library` annotation later inside your pipeline.

The **Default version** [2] field allows you to specify a branch name, tag, and commit hash of the Shared Libraries repository. In Figure 6-3, I have specified `master` (branch). As a result, my pipeline during runtime should load the latest version of my Shared Library from the `master` branch.

An environment variable `library.THIS_NAME.version` is set to the version loaded for a build (whether that comes from the **Default version** [2], or from an annotation after the @ separator).

The **Load implicitly** [3] setting is a critical thing that must be selected when using Jenkins Blue Ocean and otherwise. If checked, scripts inevitably have access to your Shared Library without needing to request it via `@Library`.

The **Allow default version to be overridden** option [4]; if checked, it allows you to select a custom version of your Shared Library by using `@someversion` in the `@Library` annotation. By default, you're restricted to use the version that you specify using the **Default version** field.

The **Include @Library changes in job recent changes** option [5] allows you to include changes on the Shared Library, if any, in the changesets of your build. You can also override this setting during the Pipeline runtime by specifying the following inside your Jenkinsfile: `@Library(value="name@version", changelog=true|false).`

There are two retrieval methods: **Modern SCM** [6] and **Legacy SCM** [7]. Both of them allow you to connect to a Git/Mercurial/Subversion repositories to retrieve your Shared Library. The difference lies in the prevalidation of versions. When you are using Legacy SCM option, no prevalidation of versions is available, and you must manually configure the SCM to refer to `${library.THISLIBNAME.version}`. Also, the **Legacy SCM** option does not allow you to connect to GitHub.

Figure 6-4 demonstrates using the **Modern SCM** retrieval method that connects to a Git Repository to retrieve a Shared Library.

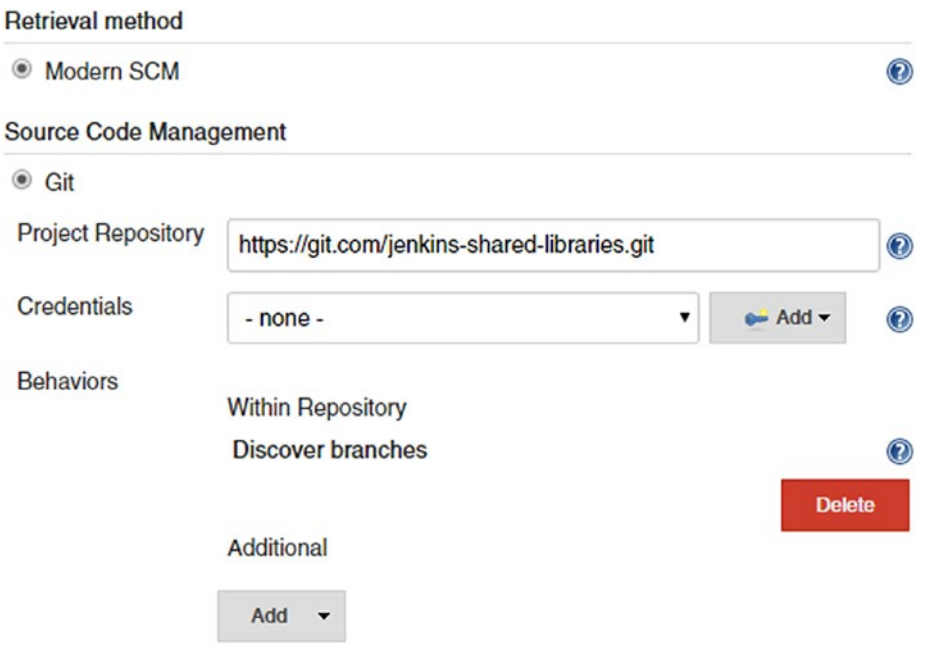

Figure 6-4. *Global Pipeline Libraries retrieval methods*

So, this is how you retrieve Shared Libraries using Jenkins global settings. I am going to skip the other method that is about retrieving Shared Libraries at the Folder/Pipeline Project level, because in the current book our focus is on using Jenkins Blue Ocean and not Classic Jenkins.

Retrieving Shared Libraries Directly During the Pipeline Runtime

In the previous section, you learned to retrieve a Shared Library inside Jenkins global settings. However, it's also possible to do the same dynamically during the Pipeline runtime. In this method, there is no need to predefine the Shared Library in Jenkins. Following is an example:

```
library identifier: '<custom name for the Shared
Library>@<version>', retriever: modernSCM(
  [$class: 'GitSCMSource',
   remote: '<Git repository URL>',
   credentialsId: '<Credential ID for the above remote
repository>'])
```

Example of Configuration from Figures 6-3 & 6-4

```
library identifier: 'jenkins-shared-libraries@master',
retriever: modernSCM(
  [$class: 'GitSCMSource',
   remote: 'https://git.com/jenkins-shared-libraries.git',
   credentialsId: 'none'])
```

You can further refer to the **Pipeline Syntax** to know the precise syntax for your SCM. Note that you must specify the library version.

Calling Shared Libraries Inside Your Pipeline

In the following section, you'll learn to summon Shared Libraries in your Pipeline. You do this by using the `@Library` annotation inside your Jenkinsfile.

Following is an example of a Jenkinsfile that's calling a function `test()` from the `test.groovy` script, which is inside the Shared Library: `jenkins-shared-libraries`.

Example of Jenkinsfile Using a Shared Library

```
/* Using a version specifier, such as branch, tag, etc */

@Library('jenkins-shared-libraries') _

test()
```

Example of a Shared Library for the Above Jenkinsfile jenkins-shared-libraries/vars/test.groovy

```
def call() {
  pipeline {
    stages {
      stage('Build') {
        steps {
          echo "Building."
        }
      }
      stage('Test') {
        steps {
            echo "Testing."
          }
      }
```

```
      stage('Publish') {
        steps {
            echo "Publishing."
        }
      }
    }
  }
}
```

If you wish to load a specific version of the Shared Library, then use the following annotation.

```
@Library('<Shared Library Name>@<version>') _
```

In this code, in place of `<version>`, you can add a branch name, a tag, or a commit hash.

Example:

```
@Library('jenkins-shared-libraries@master') _
```

For you to use a specific version of Jenkins Shared Library, it's crucial to select the option **Allow default version to be overridden**, see Figure 6-3.

Shared Libraries that are marked as **Load implicitly** (see Figure 6-3) are automatically available for your Pipelines. You need not use the `@Library` annotation inside your Pipeline code.

Example of Jenkinsfile Without @Library Annotation

```
/* Using a function from Shared Libraries without @Library
annotation */

test()
```

Creating Shared Libraries

Creating Shared Libraries is easy. If you know how to write a Groovy script, then you know how to write Shared Libraries since, at the ground level, any Groovy code can serve as a legitimate library for use in your Pipeline. Following is an example:

Example of Shared Library jenkins-shared-libraries/vars/sayhello.groovy

```
def call() {
  echo 'Hello Everyone.'
}
```

Using Global Variables with Shared Libraries

Groovy scripts coming from the **vars** directory are incorporated on-demand as individual elements. This makes it possible to define multiple methods in a single Groovy file.

Example of a Shared Library jenkins-shared-libraries/vars/log.groovy

```
def info(message) {
    echo "INFO: ${message}"
}

def warning(message) {
    echo "WARNING: ${message}"
}
```

Example of Jenkinsfile (Scripted Pipeline) with @Library annotation

```
@Library('jenkins-shared-libraries') _

log.info 'Starting.'
log.warning 'Nothing to do!'
```

Example of Jenkinsfile (Declarative Pipeline) with @Library annotation

```
@Library('jenkins-shared-libraries') _

pipeline {
    agent none
    stage ('Example') {
        steps {
             script {
                 log.info 'Starting.'
                 log.warning 'Nothing to do!'
             }
        }
    }
}
```

Using Custom Steps with Shared Libraries

In the following section, you'll learn to write custom steps with Shared Libraries. Shown here is a Groovy script for sending automatic build status e-mails.

Example of Shared Library jenkins-shared-libraries/vars/email.groovy

```
import hudson.model.Result
import org.jenkinsci.plugins.workflow.support.steps.build.
RunWrapper

def call(RunWrapper currentBuild, List<String> emailList) {
    if (!emailList) {
        return
    }
```

```
    def currentResult = currentBuild.currentResult
    def previousResult = currentBuild.getPreviousBuild()?.
    getResult()

    def buildIsFixed =
        currentResult == Result.SUCCESS.toString() &&
        currentResult != previousResult &&
        previousResult != null

    def badResult =
        currentResult in [Result.UNSTABLE.toString(), Result.
        FAILURE.toString()]

    if (buildIsFixed || badResult) {
        emailext (
            recipientProviders: [[$class:
            "RequesterRecipientProvider"]],
            to: emailList.join(", "),
            subject: "\$DEFAULT_SUBJECT",
            body: "\$DEFAULT_CONTENT"
        )
    }
}
```

This function from the Shared Library can be used inside a Pipeline as shown here.

Example of Jenkinsfile (Declarative Pipeline) with @Library annotation

```
pipeline {
    agent { label "master" }

    libraries {
        lib('jenkins-shared-libraries')
    }
```

```
    stages {
        stage("echo") {
            steps {
               echo "You are using Shared Libraries."
            }
        }
    }
    post {
        always {
            script {
                email(currentBuild, ['user@organization.com'])
            }
        }
    }
}"""
```

REUSABLE PIPELINE CODE

The following exercise assumes a hypothetical use case that may be well-suited for many of you. In this exercise, we assume a modularized software project. When I say modularized, I mean a large software product that's made up of multiple small components.

Every individual component of our software project must build, test, and publish individually. Moreover, we would like to take advantage of Shared Libraries.

So, in the current exercise, we'll construct a Shared Library that holds a common pipeline code for our CI.

1. Initiate a new empty Git Repository. This new repo is going to serve as your Shared Library.

   ```
   mkdir reusable-pipeline-library

   cd reusable-pipeline-library

   git init
   ```

2. Create the necessary directory structure for your Shared Library. See the section *The Directory Structure for Shared Library*. You are required to create only the **vars** directory.

   ```
   mkdir vars
   ```

3. Now, inside the **vars** directory, create a new Groovy file named `pipeline.groovy` using your favorite text editor and paste the following code inside it. You can also download the file directly from: https://github.com/Apress/beginning-jenkins-blue-ocean/tree/master/Ch06/pipeline.groovy.

   ```
   def call() {
   pipeline {
     agent none
     stages {
       stage('Build & Test') {
         agent {
           node {
             label 'docker'
           }
         }
         steps {
           sh 'mvn -Dmaven.test.failure.ignore clean package'
           stash(name: 'build-test-artifacts', \
           includes: '**/target/surefire-reports/TEST-*.
           xml,target/*.jar')
         }
       }
   ```

```
stage('Report & Publish') {
  parallel {
    stage('Report & Publish') {
      agent {
        node {
          label 'docker'
        }
      }
      steps {
        unstash 'build-test-artifacts'
        junit '**/target/surefire-reports/TEST-*.xml'
        archiveArtifacts(onlyIfSuccessful: true,
        artifacts: 'target/*.jar')
      }
    }
    stage('Publish to Artifactory') {
      agent {
        node {
          label 'docker'
        }
      }
      steps {
        script {
          unstash 'build-test-artifacts'

          def server = Artifactory.server 'Artifactory'
          def uploadSpec = """{
            "files": [
              {
                "pattern": "target/*.jar",
                "target": "example-repo-local/ \
                ${JOB_NAME}/${BRANCH_NAME}/${BUILD_
                NUMBER}/"
              }
```

```
                    ]
                }"""
                server.upload(uploadSpec)
            }

        }
    }
  }
 }
}
}
```

4. Save the changes made to the `pipeline.groovy` file.

5. Execute the following command to add the new files to Git:

   ```
   git add .
   ```

6. Commit your changes.

   ```
   git commit -m "Added initial files to the Shared
   Library."
   ```

7. Next, push your changes to the remote Git Repository on GitHub or wherever it delights you.

   ```
   git remote add origin <Remote Git Repository URL>

   git push -u origin master
   ```

8. You now have your Shared Library repository on a remote Git Repository (I assume its GitHub).

9. Next, you'll make the necessary configurations inside Jenkins to retrieve your new Shared Library. To do so, follow the section *Retrieving Shared Libraries Using Pre-Configured Settings in Jenkins*. Give a unique name to your Shared Library configuration inside Jenkins global settings.

10. If you have successfully made the necessary settings inside Jenkins, then let's move forward. For this exercise to work, let's create three maven projects named `component-1`, `component-2`, and `component-3`.

11. To do so, create three separate Git Repositories. Follow this command:

```
mkdir component-1 component-2 component-3

cd component-1
git init

cd ../component-2
git init

cd ../component-3
git init
```

12. Now download the source code from the following GitHub Repository folder: https://github.com/Apress/beginning-jenkins-blue-ocean/tree/master/Ch06/example-maven-project, and add it to all the three component repositories that you created in step 11.

13. Next, you'll add a Jenkinsfile to all three component repositories. To do so, create a new file using your favorite text editor and add the following content inside it. Or you can download it directly from: https://github.com/Apress/beginning-jenkins-blue-ocean/blob/master/Ch06/Jenkinsfile.

```
@Library('my-shared-library') _
call()
```

14. I assume that you have named your Shared Library configuration inside Jenkins as `my-shared-library`. However, feel free to replace `my-shared-library` with whatever name you have given to your Shared Library configuration inside Jenkins.

15. Next, execute the following commands inside all the three component repositories:

    ```
    git add .
    ```

    ```
    git commit -m "Added initial files to the source code repository."
    ```

16. Next, push the changes for all three component repositories to their respective remote Git Repositories on GitHub or wherever it delights you.

    ```
    git remote add origin <Remote Git Repository URL>
    ```

    ```
    git push -u origin master
    ```

17. Now, you have three remote Git Repositories for the three example components projects (I assume it's on GitHub).

18. For this exercise to work, make sure you have an Artifactory server up and running. It should also contain a local repository named `example-repo-local` inside it. For more information refer to the sections *Run an Artifactory Server*, *Installing the Artifactory Plugin for Jenkins*, and *Configuring the Artifactory Plugin in Jenkins* from Chapter 3.

19. At this point, you should have the following things ready with you:

    - A Shared Library hosted on a Git server that contains the required Groovy script inside the `vars` directory.
    - Three component repositories hosted on a Git server that contain the example maven source code and the necessary Jenkinsfile inside them.
    - An Artifactory server, hosting a local generic repository named `example-repo-local` with all the necessary configuration and settings inside your Jenkins server.

20. Next, open the Jenkins Blue Ocean Dashboard and create three new pipelines, one for every component, using the *Pipeline Creation Wizard*.

You should see all the three pipelines projects running, with some green and some yellow (depending on the test results). You now have a common pipeline code for all your components source code that's coming from the shared library.

The maven source code used in this exercise is puny and does not represent a real-world project. However, the idea behind this exercise is to demonstrate how Shared Libraries can be used to centralize all the common Pipeline code in the form of libraries that could be used across multiple Jenkins Blue Ocean Pipelines.

Summary

The idea behind this chapter was to introduce you to the concept of Shared Libraries in Jenkins. In this chapter you learned the basic concepts of Shared Libraries. You also learned to use it through an exercise at the end of this chapter.

The exercise *REUSABLE PIPELINE CODE* was one simple hypothetical use-case. Nevertheless, the things that you can do with Shared Libraries is limited only by your imagination.

With this, we end our book. I hope the things discussed in this book serve you well.

Appendix

Setting up a Docker Host

In the following section, you'll set up a Docker host on Ubuntu. You'll install the stable community edition (CE) of Docker. The Docker host that you'll set up using the steps described in this section should be used only for practicing the exercises and examples in the current book. The following steps should not be used to set up a production Docker host.

Prerequisites

Before we begin, make sure you have the following things ready:

- You'll need a machine with at least 4 GB of memory (the more, the better) and a multi-core processor.
- Depending on how you manage the infrastructure in your team, the machine could be an instance on a cloud platform (such as AWS or Digital Ocean) or a bare metal machine, or it could be a VM (on Wmware vSphere, Virtual Box, or any other server virtualization software).
- The machine should have Ubuntu 16.04 or higher installed on it.
- Check for admin privileges; the installation might ask for admin credentials.

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5

Set up the Repository

In the following section, you'll first set up the repository to use before installing Docker. Follow the below steps:

1. Update the apt package index:

   ```
   sudo apt-get update
   ```

2. Install packages to allow apt to use a repository over HTTPS:

   ```
   sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
   ```

3. Add docker's official GPG key:

   ```
   curl -fsSL \
    https://download.docker.com/linux/ubuntu/gpg \
    | sudo apt-key add -
   ```

4. Verify that you now have the key with the fingerprint "9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88" by searching for the last eight characters of the fingerprint.

   ```
   sudo apt-key fingerprint 0EBFCD88
   ```

   output

   ```
   pub   4096R/0EBFCD88 2017-02-22
   Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81
   803C 0EBF CD88
   uid   Docker Release (CE deb) <docker@docker.com>
   sub   4096R/F273FCD8 2017-02-22
   ```

5. Set up the stable repository. The following command automatically detects your Ubuntu OS distribution and sets up the appropriate Docker repository.

```
sudo add-apt-repository "deb [arch=amd64] \
 https://download.docker.com/linux/ubuntu \
 $(lsb_release -cs) stable"
```

Install Docker

Next, let's install the community edition of Docker.

1. Update the apt package index.

```
sudo apt-get update
```

2. Install the *latest version* of Docker CE.

```
sudo apt-get install docker-ce
```

3. Verify that Docker CE is installed correctly by running the hello-world image.

```
sudo docker run hello-world
```

Output

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
d1725b59e92d: Pull complete
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799
f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest
```

```
Hello from Docker!
.
.
.
```

Enabling Docker Remote API (Critical)

The Docker remote API allows external applications to communicate with the Docker server using REST APIs. Jenkins (through the Docker Plugin) uses the docker remote API to communicate with a docker host.

To enable the Docker remote API on your Docker host, you'll need to modify Docker's configuration file. Depending on your OS version and the way you have installed Docker on your machine, you might need to choose the right configuration file to modify. Shown here are two methods that work on Ubuntu. Try them one by one.

Modifying the docker.conf File

Follow these steps to modify the `docker.conf` file:

1. Log in to your docker server; make sure you have sudo privileges.
2. Execute the following command to edit the file `docker.conf`:

   ```
   sudo nano /etc/init/docker.conf
   ```

3. Inside the `docker.conf` file, go to the line containing `"DOCKER_OPTS="`.

You'll find `"DOCKER_OPTS="` variable at multiple places inside the `docker.conf` file. Use the `DOCKER_OPTS=` that is available under the `pre-start script` or `script` section.

---

4. Set the value of DOCKER_OPTS as shown here. Do not copy and paste; type it all in a single line.

   ```
   DOCKER_OPTS='-H tcp://0.0.0.0:4243 -H unix:///var/run/
   docker.sock'
   ```

The above setting binds the Docker server to the UNIX socket as well on TCP port 4243.

"0.0.0.0" makes Docker engine accept connections from anywhere. If you would like your Docker server to accept connections only from your Jenkins server, then replace "0.0.0.0" with your Jenkins Server IP.

---

5. Restart the Docker server using the following command:

   ```
   sudo service docker restart
   ```

6. To check if the configuration has worked, execute the following command. It lists all the images currently present on your Docker server.

   ```
   curl -X GET http://<Docker Server IP>:4243/images/json
   ```

7. If this command does not return a meaningful output, try the next method.

Modifying the docker.service File

Follow these steps to modify the `docker.service` file:

1. Execute the following command to edit the `docker.service` file.

   ```
   sudo nano /lib/systemd/system/docker.service
   ```

2. Inside the `docker.service` file, go to the line containing `ExecStart=` and set its value as shown here. Do not copy and paste; type it all in a single line.

   ```
   ExecStart=/usr/bin/dockerd -H fd:// -H tcp://0.0.0.0:4243
   ```

The above setting binds the docker server to the UNIX socket as well on TCP port `4243`.

`"0.0.0.0"` makes the Docker engine accept connections from anywhere. If you would like your Docker server to accept connections only from your Jenkins server, then replace `"0.0.0.0"` with your Jenkins Server IP.

---

3. Execute the following command to make the Docker demon notice the modified configuration:

   ```
   sudo systemctl daemon-reload
   ```

4. Restart the Docker server using the below command:

   ```
   sudo service docker restart
   ```

5. To check if the configuration has worked, execute the following command. It lists all the images currently present on your Docker server.

   ```
   curl -X GET http://<Docker Server IP>:4243/images/json
   ```

Enabling Proxy Compatibility for Jenkins

Sometimes HTTP/HTTPS proxies filter out information used by the default crumb issuer. If, by any chance, you receive a 403 response when submitting a form to Jenkins, checking this option may help. Using this option makes the nonce value easier to forge.

Enabling proxy compatibility may help to solve this issue. Follow these simple steps to enable proxy compatibility:

1. From the Jenkins Blue Ocean Dashboard, click on **Administration** link.
2. On the Manage Jenkins page, click on **Configure Global Security.**
3. On the **Configure Global** Security page navigate **to** the section **CSRF Protection**, and select the option **Enable proxy compatibility [1]** (see Figure A-1).

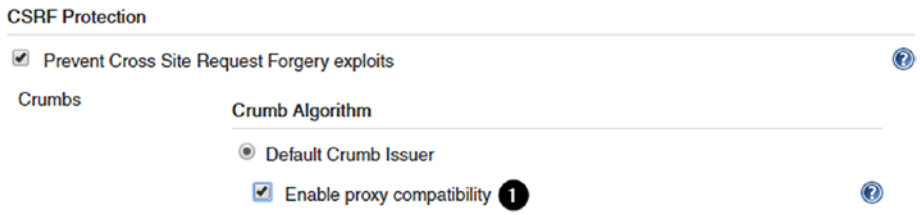

Figure A-1. *CSRF Protection settings in Jenkins*

Index

N. Pathania, *Beginning Jenkins Blue Ocean*, https://doi.org/10.1007/978-1-4842-4158-5

E

F

G, H

I

J, K

L

M

N, O

P, Q

R

S

T

U

V

W, X, Y, Z