Mitesh Soni

Jenkins Essentials

Second Edition

Setting the stage for a DevOps culture

Packt

Jenkins Essentials

Second Edition

Setting the stage for a DevOps culture

Mitesh Soni

BIRMINGHAM - MUMBAI

Jenkins Essentials

Second Edition

First published: July 2015

Second edition: June 2017

Production reference: 1280617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78847-106-0

www.packtpub.com

Credits

Author
Mitesh Soni

Reviewers
Stefan Lapers
Devin Young
Javier Delgado

Commissioning Editor
Vijin Boricha

Acquisition Editor
Divya Poojari

Content Development Editor
Deepti Thore

Technical Editor
Sneha Hanchate

Copy Editor
Saafis Editing

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Mariammal Chettiyar

Graphics
Tania Dutta

Production Coordinator
Nilesh Mohite

About the Author

Mitesh Soni is an avid learner with 10 years' experience in the IT industry. He is an SCJP, SCWCD, VCP, IBM Urbancode, and IBM Bluemix certified professional. He loves DevOps and cloud computing and he also has an interest in programming in Java. He finds design patterns fascinating. He believes "a picture is worth a thousand words."

He occasionally contributes to `etutorialsworld.com`. He loves to play with kids, fiddle with his camera, and take photographs at Indroda Park. He is addicted to taking pictures without knowing many technical details. He lives in the capital of Mahatma Gandhi's home state.

Mitesh has authored the following books with Packt:

- DevOps Bootcamp
- Implementing DevOps with Microsoft Azure
- DevOps for Web Development
- Jenkins Essentials
- Learning Chef

"I've missed more than 9,000 shots in my career. I've lost almost 300 games. 26 times, I've been trusted to take the game-winning shot and missed. I've failed over and over and over again in my life. And that is why I succeed."—Michael Jordan.

I've always thanked a lot of people who have been instrumental in contributing to my life's journey up to now, but I guess it's time to really acknowledge that One person who has been with me as long as I can remember.

With this book, I would like to thank the one and only invisible yet omnipresent Almighty. We share a mutual love and hate relationship and I really value it. You were always there equally during my good and bad times and without you, I wouldn't have made it this far!

Last but not the least, I want to thank all who taught me how to love myself, first!

About the Reviewers

Stefan Lapers started his career almost 20 years ago as an IT support engineer and quickly grew into Linux/Unix system engineering and software development.

Over the years he accumulated experience deploying and maintaining hosted application solutions while working for great customers such as MTV, TMF, and many more. In more recent years, he was involved in multiple development projects and their delivery as a service on the internet.

In his spare time, he enjoys his family and building/flying RC helicopters.

Devin Young graduated with a BS in sports management from Ohio State University and somehow wound up working as a software engineer shortly afterwards. He specializes in DevOps and site reliability and is particularly fond of automation and real-time applications.

He grew up as a competitive jump roper, leading him to create the mobile app RopeRacer, which was launched on iOS in March 2015. The app has become a success in the world of jump rope and is now used in tournaments around the United States.

Devin is currently an MBA candidate at the TCU Neeley School of Business.

Javier Delgado is an automation fanatic, continuous tasks (inspection, testing, delivery), evangelist, and perpetual new-knowledge addict.

He works as DevOps and Jenkins internal information well at Stratio Big Data Inc.

www.PacktPub.com

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`https://www.packtpub.com/mapt`

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at `https://www.amazon.com/dp/1788471067`.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

I would like to dedicate this book to lot of people who gave me a ray of hope amidst darkness. I would like to dedicate this book to Shreyansh (Shreyu – my sister Jigisha's baby boy) who showed me the power of innocence and smiles, Vinay Kher for his blessing, my parents who are always there silently praying for me, Simba (Priyanka Agashe) for supporting and encouraging me all the time and forcing me to believe in myself Indian Army and all brave soldiers in uniform for protecting us.

Table of Contents

Preface

Jenkins has been used specifically for Continuous Integration over the years. Continuous Integration systems are a vital part of Agile teams because they help enforce the principles of Agile development.

Jenkins, a continuous build tool, enables Agile teams to focus on work and innovations by automating the build, artifact management, and deployment processes, rather than worrying about manual processes. However, the release of Jenkins 2.0 and later versions are focused on Continuous Delivery.

Jenkins is an open source automation server. Continuous Integration is a significant part of DevOps culture and hence many open source and commercial tools for Continuous Delivery utilizes Jenkins for a complete product.

DevOps is a buzzword in 2015 and for coming years as per Market trends by various research firms. Continuous Integration is a significant part of DevOps culture and hence the trend to use Jenkins will increase in future. If Continuous Integration is the base, then Continuous Delivery is the topping. Jenkins supports and focus more on end-to-end automation of application life cycle management system.

What this book covers

`Chapter 1`, *Exploring Jenkins*, describes in detail the basics of Continuous Integration and an overview of Jenkins. It describes the recent growth of importance of Continuous Integration as a practice to cultivate DevOps culture. This chapter also describes the installation and configuration of Jenkins. We are going to take a whistle-stop tour through some of the key features of Jenkins, and plugin installation as well.

`Chapter 2`, *Installation and Configuration of Code Repositories and Build Tools*, describes in detail how to prepare the runtime environment for application life cycle management and configure it with Jenkins – the open source continuous integration tool. It will cover how to integrate Eclipse and Jenkins, so builds can be run from Eclipse as well.

`Chapter 3`, *Managing Code Quality and Notifications*, will cover how to integrate static code analysis behavior into Jenkins. Code quality is an extremely vital feature that impacts on applications' effectiveness and by integrating it with Sonar, users get insights into problematic portions of code. This chapter also covers email notifications on build status.

`Chapter 4`, *Continuous Integration with Jenkins*, describes in detail how to create and configure build jobs in Java, how to run build jobs, unit test cases using Ant, and Maven build tools. It covers all aspects of running a build to create a distribution file or war file for deployment.

`Chapter 5`, *Continuous Delivery - Implementing Automated Deployment*, provides insights into how functional testing and load testing can be performed and how they can be integrated with Jenkins to adopt the Continuous Testing practices of DevOps culture.

`Chapter 6`, *Continuous Testing - Functional and Load Testing with Jenkins*, takes one step forward in the DevOps pipeline by deploying artifacts to local or remote application servers. It gives insights into automated deployment and Continuous Delivery processes, and also covers how to deploy applications to a public cloud platform using Jenkins.

`Chapter 7`, *Build Pipeline and Pipeline as a Code*, will cover how to orchestrate build job to execute them in a specific sequence. We will cover the Build Pipeline plugin and the Pipeline as a Code feature that is available in Jenkins 2 and later.

`Chapter 8`, *Managing and Monitoring Jenkins*, gives insight into the management of Jenkins nodes and monitoring them with Java Melody, in order to provide details on the utilization of resources. It also covers how to monitor build jobs configured for Java-or .NET-related applications, and managing those configurations by keeping backups of them.

`Chapter 9`, *Security in Jenkins*, will cover security management options available in Jenkins. This will help to perform user management, authentication, and authorization, including matrix-based security and role-based access.

What you need for this book

This book is for beginners. This book assumes that you are familiar with at least the Java programming language. Knowledge of core Java and JEE is essential if you are to use this book to gain better insights. Having a strong understanding of programming logic will provide you with the background to be productive with Jenkins, while using plugins or writing commands for the shell.

As the application development life cycle will cover a lot of tools, it is essential to have some knowledge of repositories such as svn, git, and so on, IDE tools such as Eclipse, and build tools such as Ant and Maven.

Knowledge of code analysis tools will make jobs easier to configur and integrate, however, it is not required in order to complete the exercises presented in this book. Most of the configuration steps are mentioned clearly. SonarQuve version 6.3 is used for code analysis.

You will be walked through the steps required to install Jenkins on a Windows and Linux-based host. In order to be immediately successful, you will need administrative access to a host that runs a modern version of Windows and Linux; Windows 10 will be used for demonstration purposes. If you are a more experienced reader, then a recent release of almost any distribution will work just as well (but you may be required to do a little bit of extra work that is not outlined in the book).

Additionally, you will need access to the internet to download plugins that you do not already have, as well as an installation of Jenkins. Any normal hardware configuration is good enough, such as 4 GB RAM and 500 GB hard drive.

Who this book is for

This book assumes that you are familiar with at least java programming language. Knowledge of core java and JEE is essential considering this book to gain better insight. Having a strong understanding of program logic will provide you with the background to be productive with Jenkins while using plugins of writing commands for shell.

As application development life cycle will cover lot of tools in general; it is essential to have some knowledge of repositories such as svn, git etc; IDE tools such as Eclipse; build tools such as ant and maven.

Knowledge of code analysis tools will make job easier in configuration and integration, however it is not extremely vital to perform exercises given in the book. Most of the configuration steps are mentioned clearly.

Additionally, you will need access to the Internet to download plugins that you do not already have, as well as an installation of the Jenkins.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

A block of code is set as follows:

```
#run the model
model <- OneR(train_data, frisked ~ ., verbose = TRUE)
#summarize the model
summary(model)
#run the sql function from the SparkR package
SparkR::sql("SELECT sample_bin , count(*)
\FROM out_tbl group by sample_bin")
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#note we are specifing the SparkR filter, not the dplyr filer
head(SparkR::filter(out_sd1,out_sd1$sample_bin==1),1000)
```

Any command-line, (including commands at the R console) input or output is written as follows:

```
> summary(xchurn)
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Jenkins-Essentials-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `https://www.packtpub.com/sites/default/files/downloads/JenkinsEssentialsSecondEdition_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Exploring Jenkins

Jenkins is an open source automation server (after Jenkins 2.0 was released) written in Java. It was one of the most popular **Continuous Integration** (**CI**) tools used to build and test different kinds of projects. Now, it is also used for **Continuous Delivery** (**CD**) after Jenkins 2.0. This chapter describes in detail the basics of CI and overview of Jenkins 2. It describes the importance of CI and CD as a practice to cultivate DevOps culture in recent times.

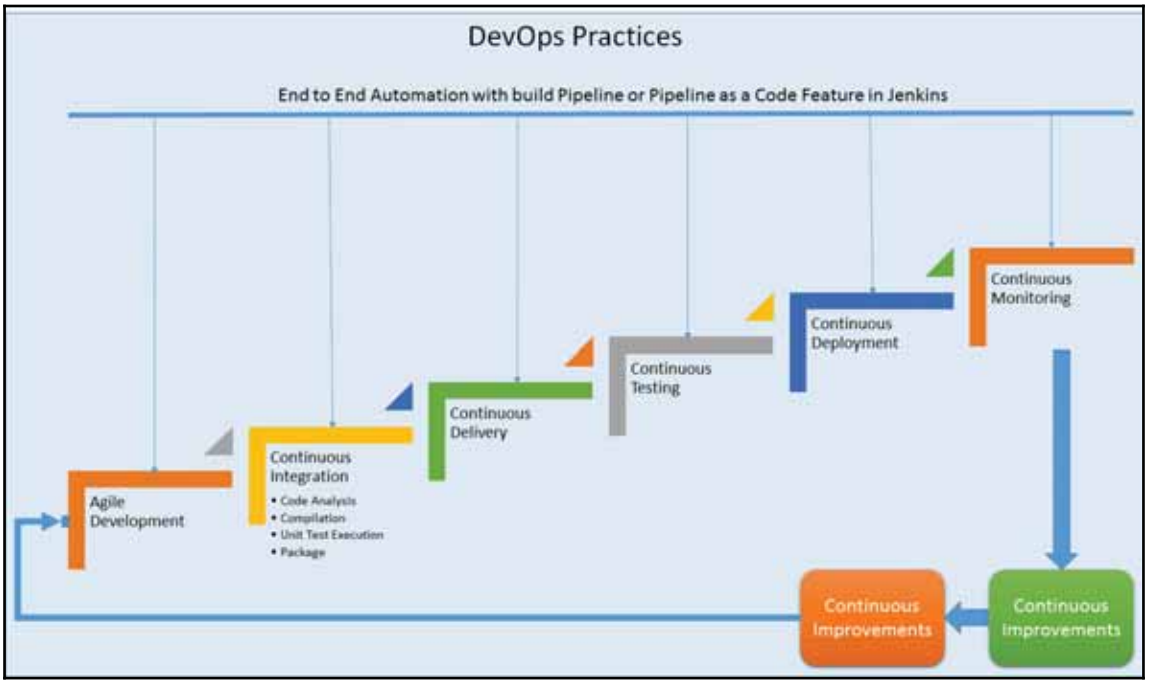
This chapter also describes installation and configuration of Jenkins 2. We are going to take a jumpstart tour through some of the key features of Jenkins and plugin installations as well.

To be precise, we will discuss the following topics in this chapter:

- Introduction of Jenkins 2 and its features
- Installation of Jenkins 2
- Jumpstart tour of Jenkins dashboard
- Configuration settings in Jenkins
- Overview of CICD pipeline

Lets get started! On your marks, get set, go!

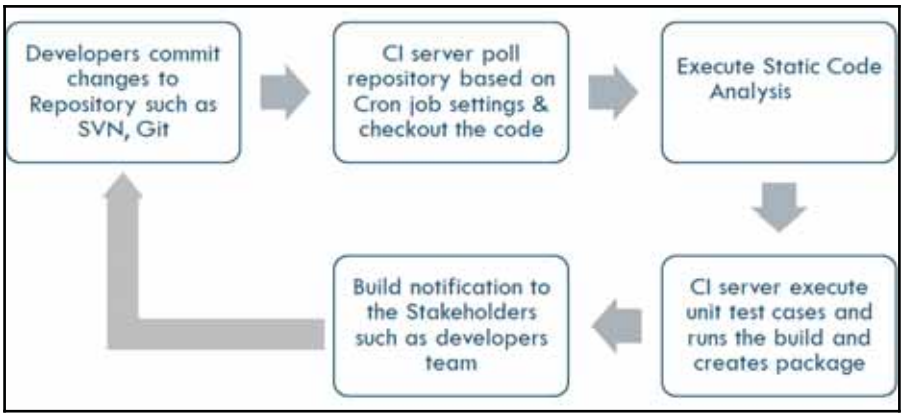
In this chapter, we will see what Jenkins 2 is and the new features introduced regarding CI as a part of our DevOps journey. We will cover the following steps to complete our DevOps journey. Each chapter is a stepping stone to reach the next one. It is always good to have an incremental approach so we can measure our success and feel the pain points as well to realize the value of this journey:



At the end of this chapter, we will know essential things about Jenkins 2 and how it is a game changer in terms of CD. It is no longer a CI server. It is on its way to becoming a mature product in the category of automation servers by focusing on Continuous Delivery after Jenkins 2.0 is released.

Introduction of Jenkins 2

Let's first understand what CI is. CI is one of the most popular application development practices in recent times. Developers integrate bug fixes, new feature development, or innovative functionality in a code repository. The CI tool verifies the integration process with an automated build and test to detect issues with current sources of an application and provide quick feedback:



Jenkins is a simple, extensible, and user friendly open source tool that provides continuous integration services for application development. Jenkins supports SCM tools such as Git, Subversion, Star Team, and CVS, AccuRev. Jenkins can build Apache Ant and Apache Maven-based projects.

The concept of plugins makes Jenkins more attractive, easy to learn, and easy to use. There are various categories of plugins available, such as the following:

- Source code management
- Slave launchers and controllers
- Build triggers
- Build tools
- Build notifiers
- Build reports
- Other post-build actions
- External site/tool integrations
- UI plugins
- Authentication and user management
- Android development
- iOS development
- .NET development
- Ruby development
- Library plugins

Jenkins defines interfaces or abstract classes that model a facet of a build system. Interfaces or abstract classes agree on what needs to be implemented, and Jenkins uses plugins to extend those implementations.

With Jenkins 2, the focus is also on CD where the application is deployed in the specific environment using an automated approach. Jenkins 2 is a clear signal regarding the focus on both CI and CD best practices of DevOps culture and not on CI only.

Features

Jenkins is one of the most popular automation servers in the market, and the reasons for its popularity are some of the following features:

- Easy installation on different operating systems, Arch Linux, FreeBSD, Gentoo, MacOS X, openBSD, openSUSE, RedHAT/Fedora/CentOS, Ubuntu/Debian, Windows, and it is also available for Docker and as generic Java packages too.
- Easy upgrades: Jenkins has very speedy release cycles (long-term support and weekly releases).
- Simple and easy to use user interface in Jenkins 2.x -.
- Set of suggested plugins at the time of installation.
- Improved new item page.
- Improved job configuration page with easy navigation.
- Jenkins 2 supports pipelines as code that uses **domain-specific language** (**DSL**) to model application delivery pipelines as code; we can utilize the pipelines as code and keep them in a repository and maintain versions in a similar way to source code.
- Easily extensible with the use of third-party plugins: there are over 400 plugins.
- Easy to configure the setup environment in the user interface. It is also possible to customize user interface as you wish.
- Master slave architecture supports distributed builds to reduce the load on CI servers.
- Build scheduling based on cron expressions.
- Shell and Windows command execution that makes any command-line tool integration in the pipeline very easy.
- Notification support related to build status.

Installation of Jenkins 2

Let's start with Jenkins 2.x installation. Go to `https://jenkins.io/` and click on the **Download** button to download packages for installation of Jenkins:

At `https://jenkins.io/download/`, we get two sections. One is for **Long-term Support (LTS)** that is selected after every 12 weeks from regular releases as the stable release from that duration.

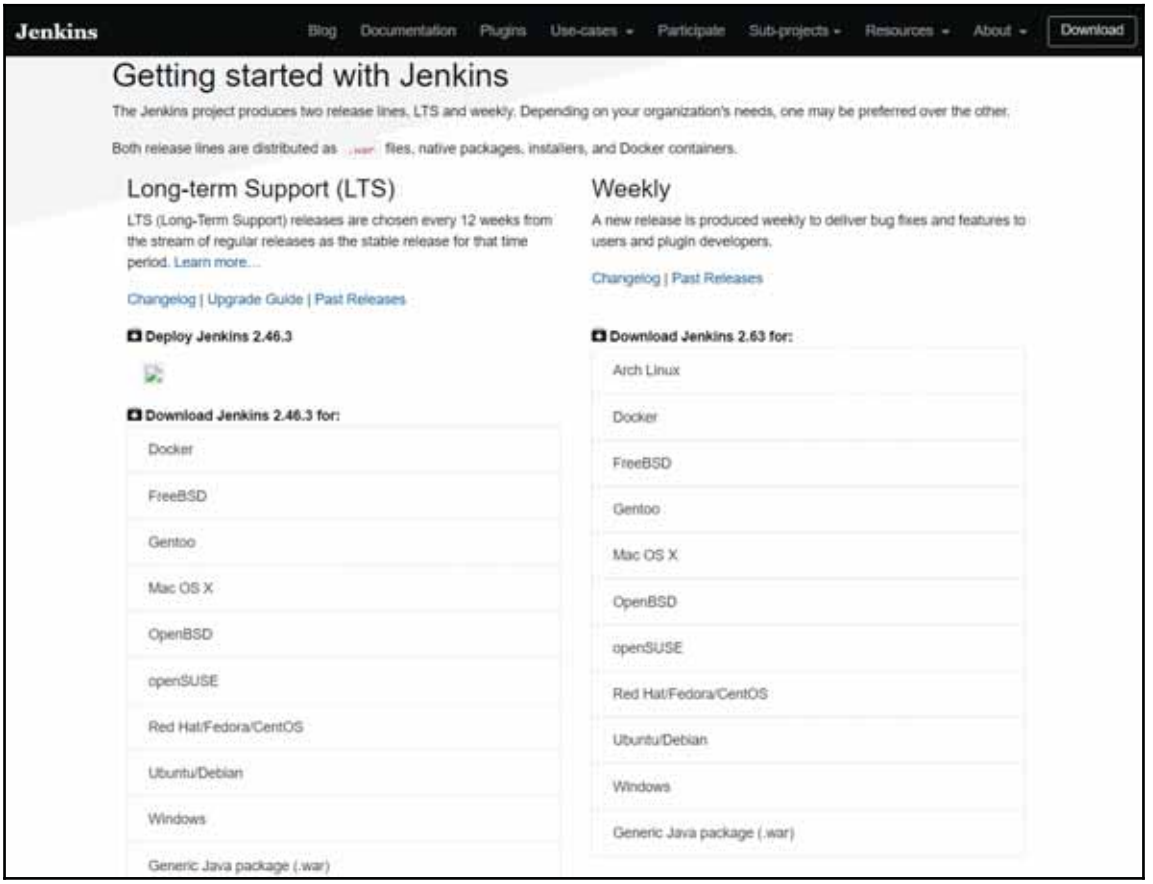
Another section is **Weekly** release, which has bug fixes and is made available to users and developers.

We will use the **Generic Java Package (.war)** file in our installation of Jenkins as shown in the following screenshot.

The reason for selecting the `.war` file for the Jenkins installation is its ease of use across operating systems. Jenkins is written in Java, and in any case, to execute Jenkins we need the latest Java version installed on our system. For Jenkins Installation, Java 8 is recommended. It is recommended to have 1-GB of memory.

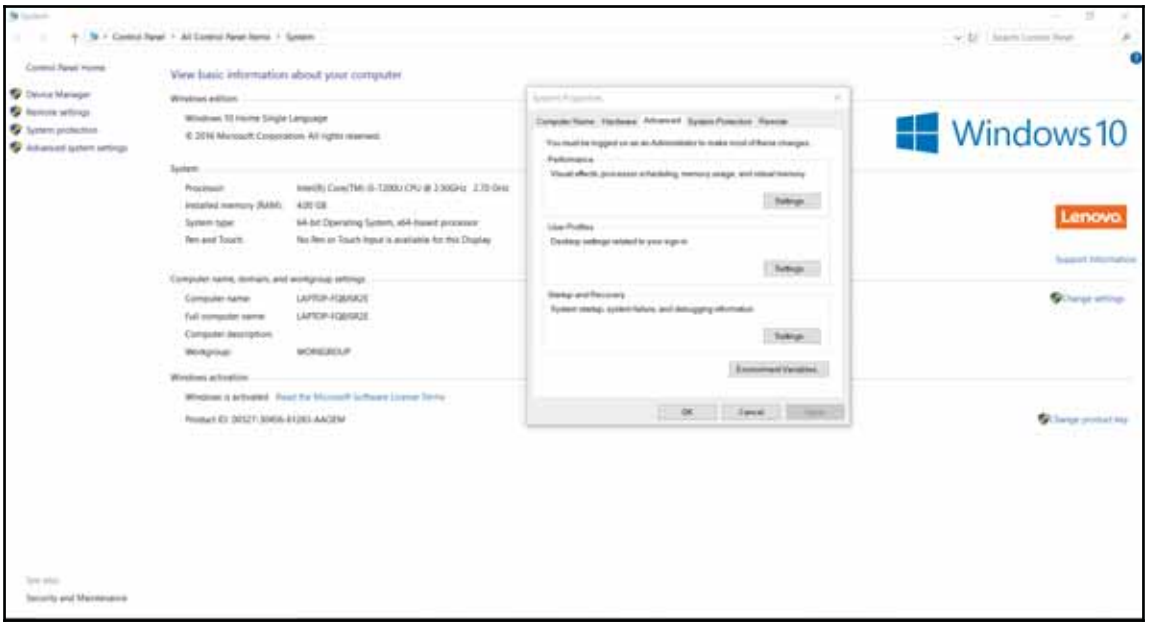
Verify the Java installation by using the `java -version` command in the command prompt or terminal based on the operating system.

Download **Generic Java Package (.war)** from `https://jenkins.io/download/` on the local system as follows:

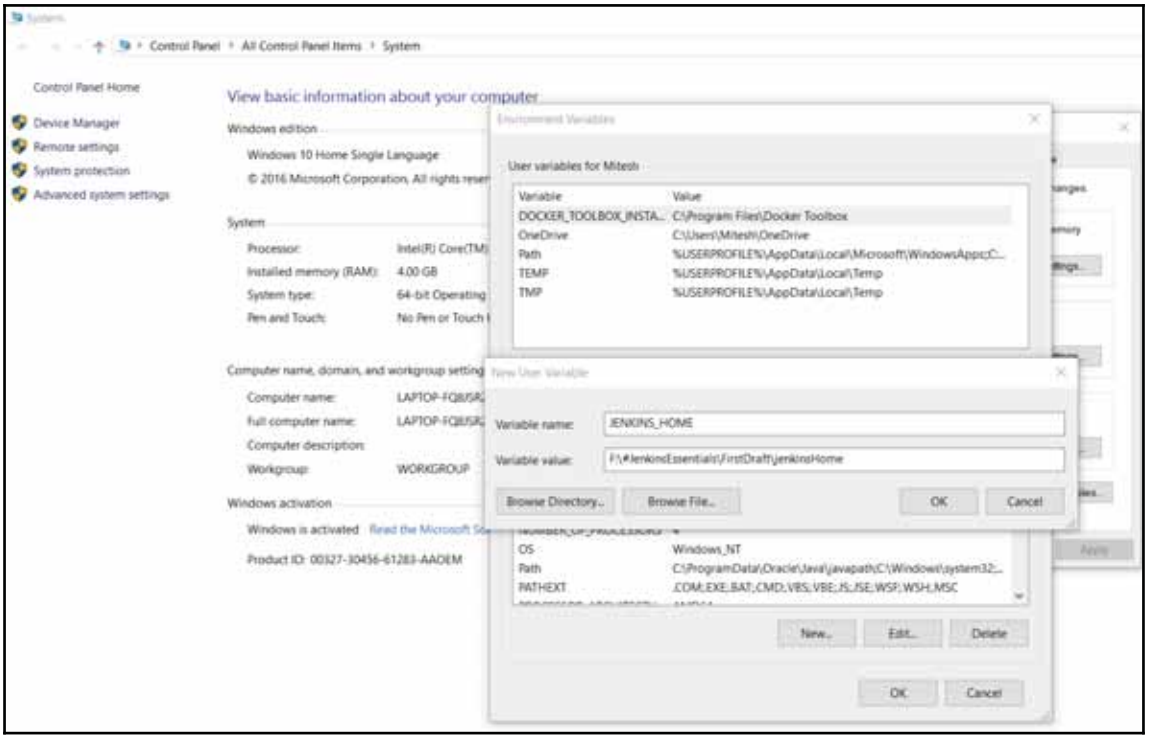


Before we start Jenkins, we will set the `JENKINS_HOME` environment variable. When we install and configure Jenkins, all the files reside in the `jenkins` directory by default. We often need to change the default location for our convenience. We can do that by setting the `JENKINS_HOME` environment variable. Follow the following steps:

1. To set the `JENKINS_HOME` environment variable in Windows 10, go to **Control Panel** | **All Control Panel Items | System** and click on **Advanced system settings** | **Advanced** | **Environment Variables...** Please see the manual for other operating systems to set **Environment Variables...**:

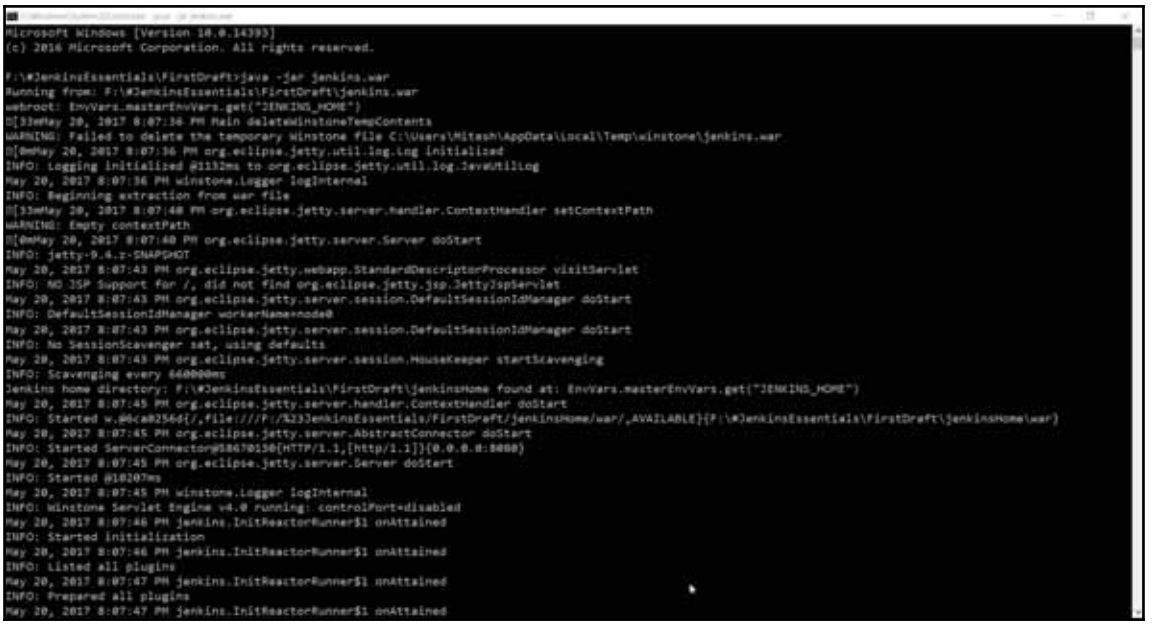


2. Click on **New**. Enter the variable name and location and click **OK**:

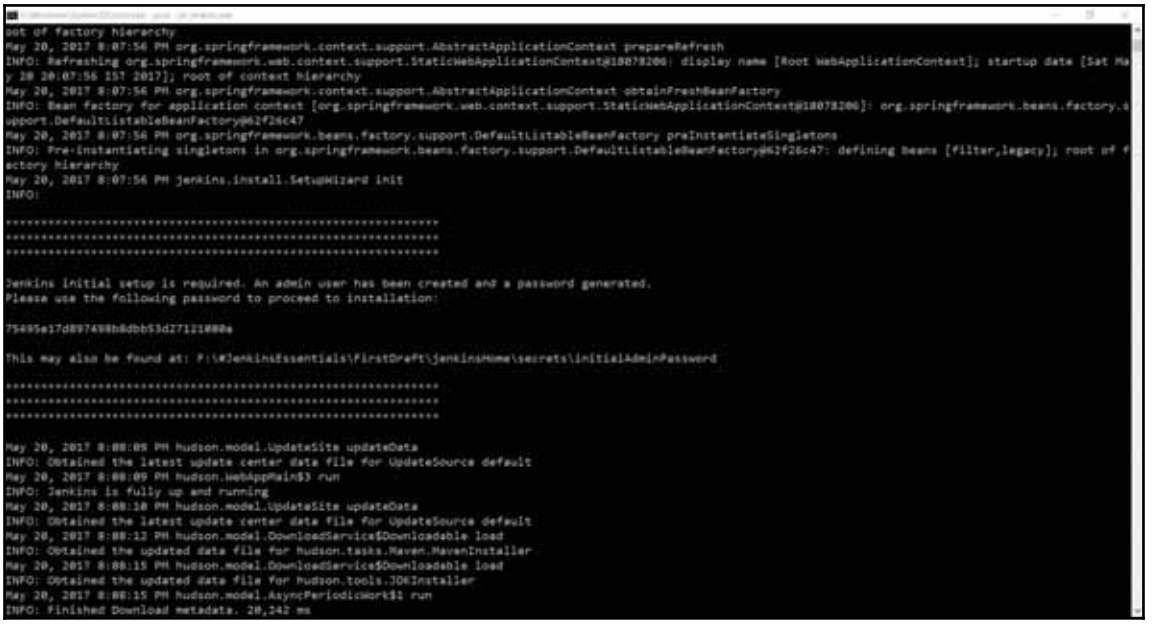


3. Now our **JENKINS_HOME** is set so Jenkins will use that directory to store all configuration files.
4. Open the command prompt or terminal (depending on your operating system) and execute the following command:

```
Java -jar Jenkins.war
```

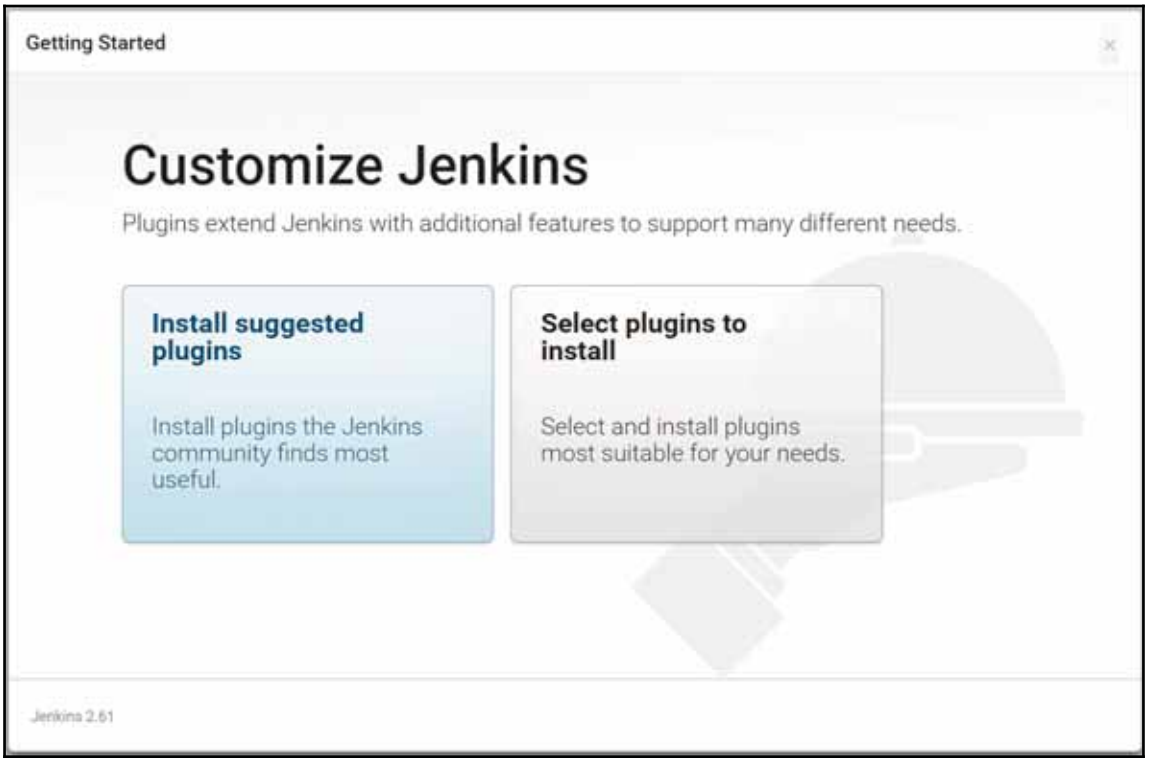


5. This is a fresh installation of Jenkins, so initial setup is required. Note the password:

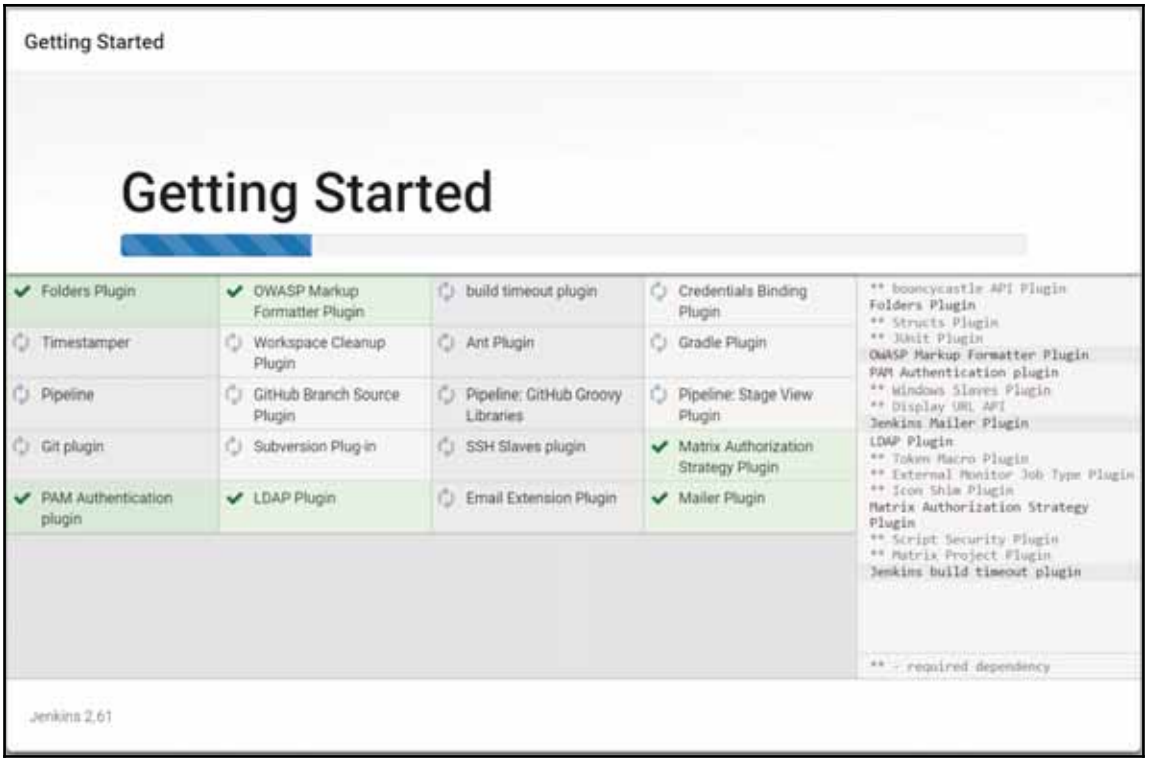


6. Once Jenkins is fully up and running, visit `http://localhost:8080` and it will open the **Getting Started** page to **Unlock Jenkins**. Give the password we copied from the terminal or go to the location given in the dialog box. Copy the password from there and click on **Continue**:

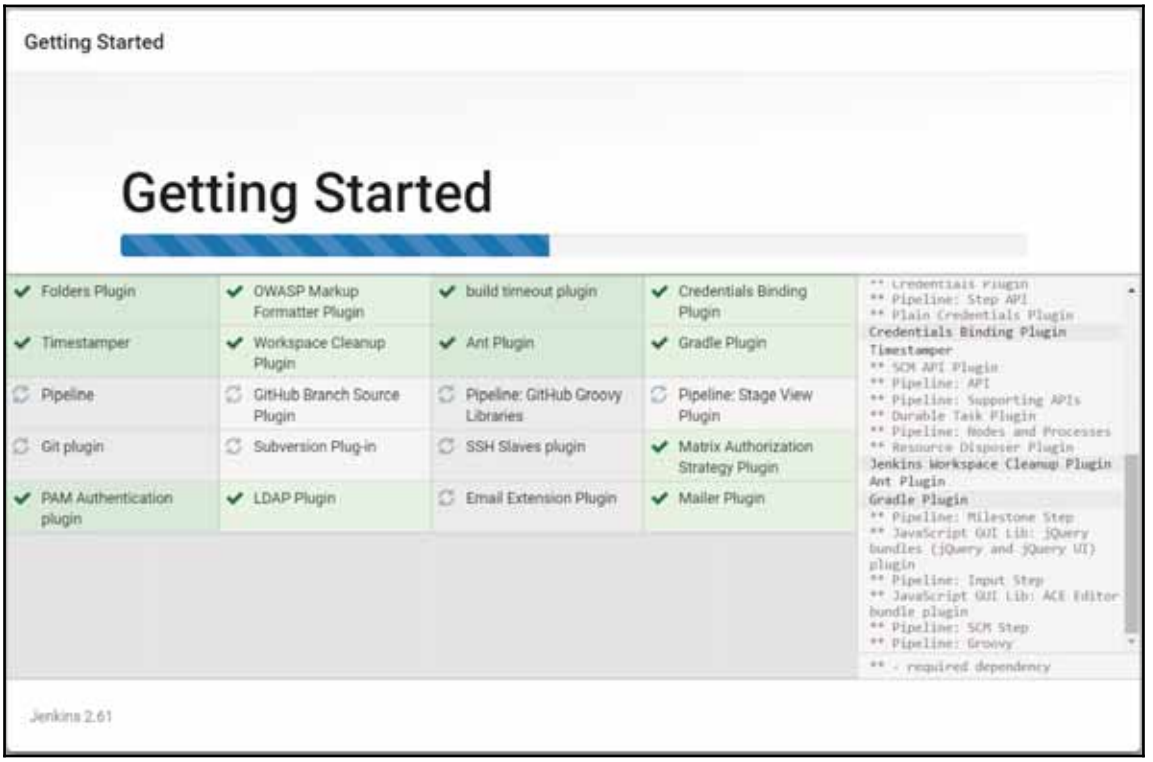
7. Click on **Install** to see the suggested plugins. If we are behind the proxy, then another dialog box will pop up before this page to provide proxy details:



8. Wait while all the plugins are installed properly:

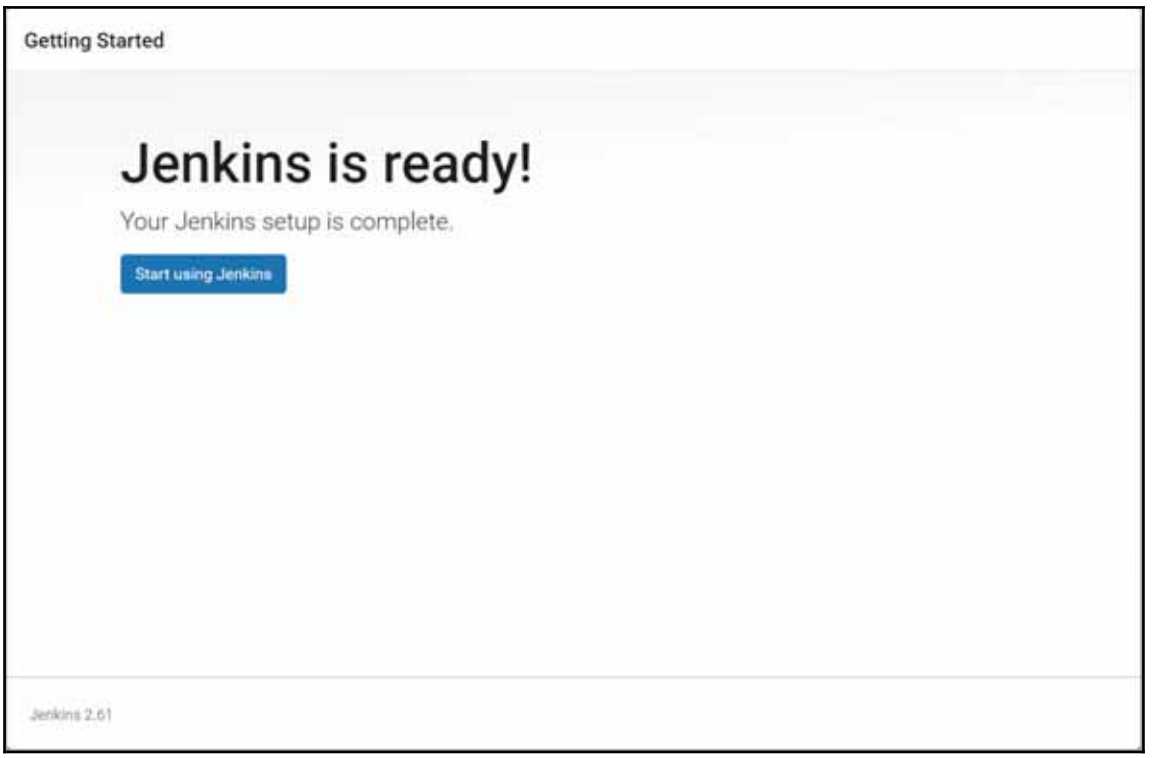


9. Verify the green tick boxes for all the plugins that have been installed successfully:



10. Once all plugins are installed successfully, create the first admin user and click on **Save and Finish**:

11. Click on **Start using Jenkins**:



In the next section, we will see details of the Jenkins dashboard.

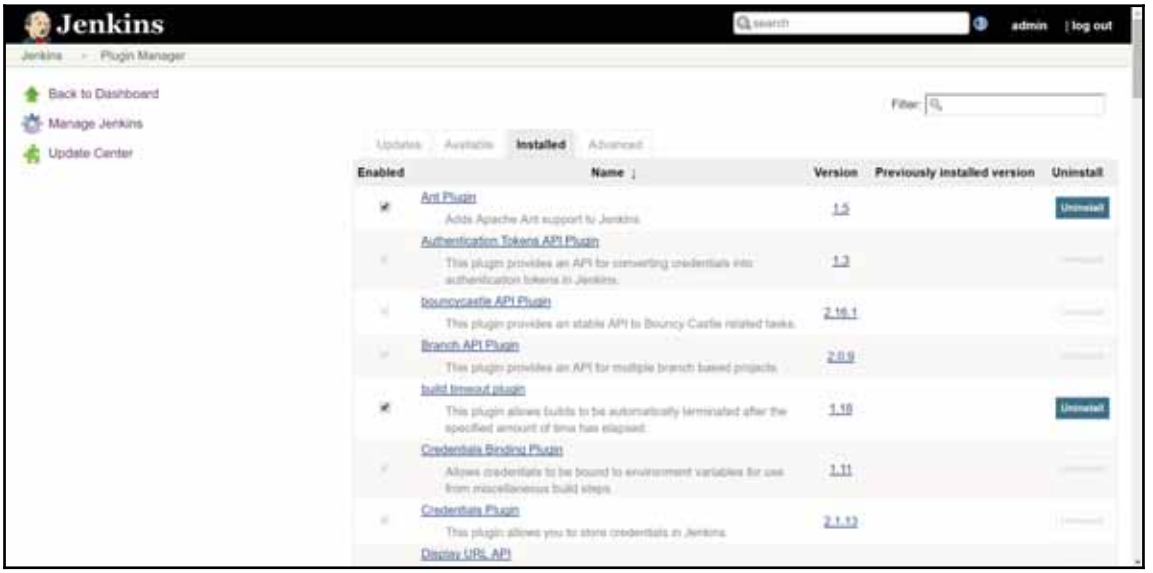
Jumpstart tour of the Jenkins dashboard

The Jenkins dashboard is the place where all the operations related to CI and CD can be managed:

Click on the **Manage Jenkins** link on the Jenkins dashboard. Here, we can configure **Jenkins**, **Security**, **Global Tools**, **Plugins**, **Users**, and more:

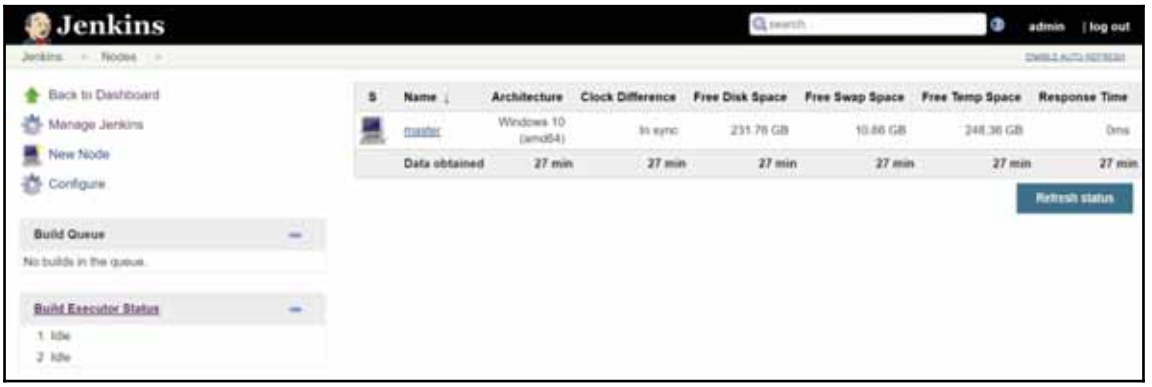
Click on **Manage Plugins**. You will see the following tabs:

- The **Updates** tab provides details on updates available on the installed plugins.
- The **Available** tab provides a list of plugins that are not installed yet.
- The **Installed** tab provides a list of plugins that are already installed.
- The **Advanced** tab contains sections to configure proxies so we can download plugins even after we are behind the proxy. It also provides sections to upload HPI files for plugins in case we have already downloaded the plugin from the internet:



In the **Manage Jenkins** section, click on **Manage Nodes**.

By default, the system on which Jenkins is installed is a master node. This is the section that can be utilized to create the master agent architecture that we will cover later in the book:

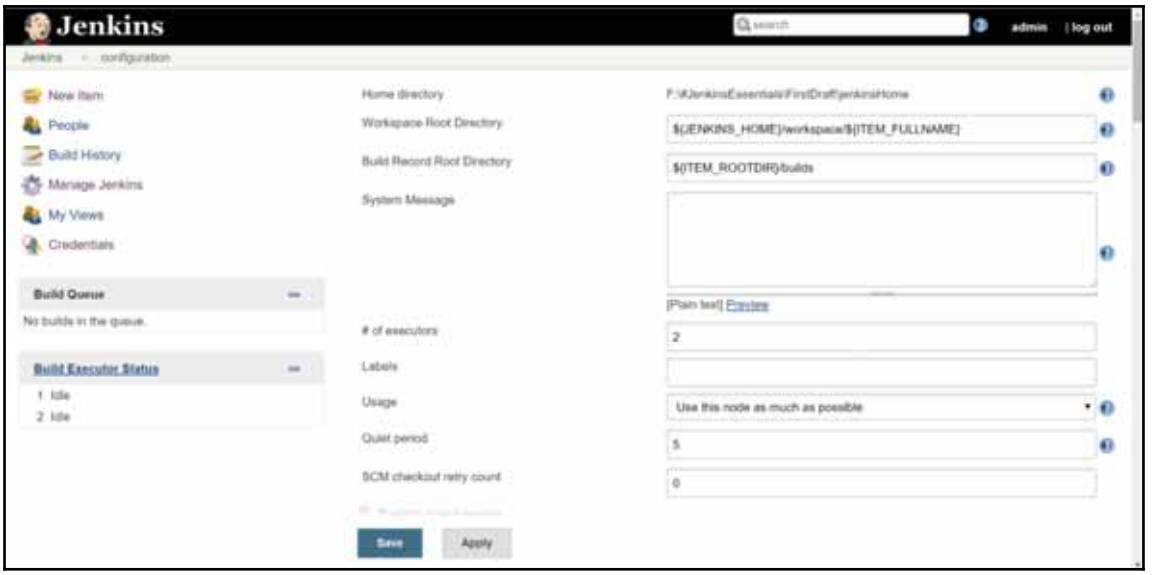


In the next section, we will cover different kinds of configuration available in the **Manage Jenkins** section.

Configuration settings in Jenkins

In **Manage Jenkins**, click on the **Configure System** link.

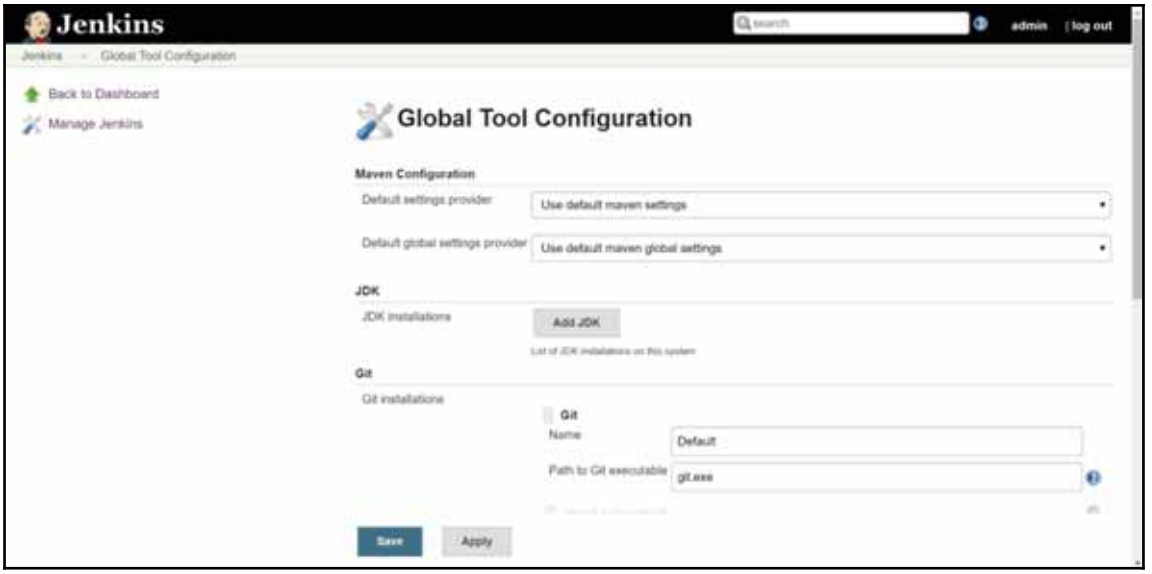
Here, we can get information about the `JENKINS_HOME` directory, the workspace root directory, and so on. We can set the Jenkins URL as well in this section:



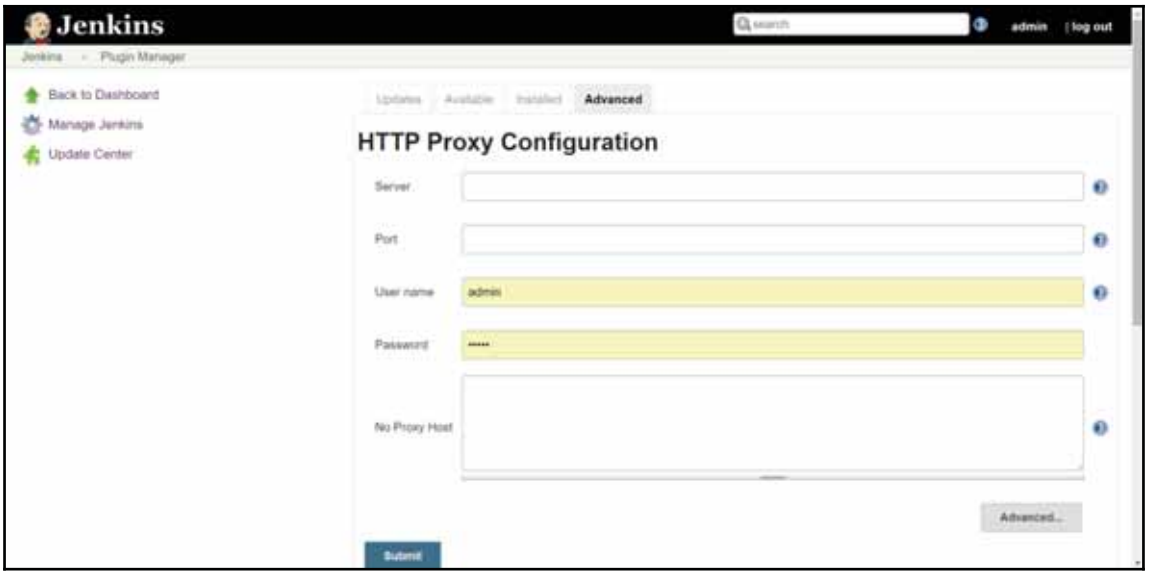
In **Manage Jenkins**, click on **Configure Global Security** to see the security settings available in Jenkins. We will cover role-based access, matrix-based project security, and other features in later parts of this book:

In **Manage Jenkins**, click on the **Global Tool Configuration** link to provide details related to all tools available on the system that can be utilized to perform certain tasks. Depending on plugins related to specific tools, that section will appear on this page.

Another important thing to mention is that we can configure multiple versions of the same tool. For example, we can configure Java 6, Java 7, and Java 8. It is highly likely that different projects require different versions of Java for their execution. In such cases, we can configure multiple JDKs here and utilize specific JDKs in specific build jobs:



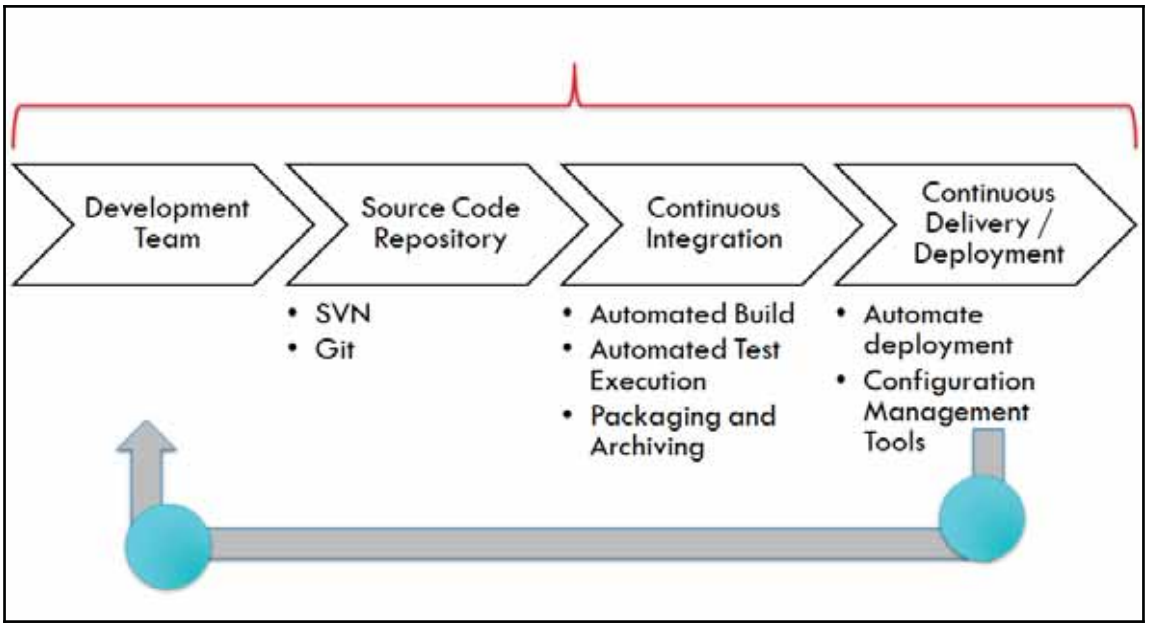
As discussed earlier in this chapter, the **Manage Plugins** section has an **Advanced** tab that allows us to configure proxy details.



This was all about the basic installation and configuration of Jenkins 2.x, and now we will cover what we are going to achieve in this book.

Overview of the CI/CD pipeline

The application development life cycle is traditionally a lengthy manual process. In addition, it requires effective collaboration between development and operations teams. The CI/CD pipeline is a demonstration of automation involved in the application development lifecycle that contains automated build execution, automated test execution, notifications to stakeholders, and deployment in different runtime environments. Effectively, a deployment pipeline is a combination of continuous integration and continuous delivery, and hence a part of DevOps practices. The following figure depicts the pipeline process. Depending on the culture of organization and the available tools, the flow and tools may differ:



Members of the development team check code into a source code repository. Continuous integration products such as Jenkins are configured to poll changes from the code repository. Changes in the repository are downloaded to the local workspace and Jenkins triggers a build process that is assisted by Ant or Maven or Gradle or any build script. Automated test execution, unit testing, static code analysis, reporting, and notification of successful or failed build processes are also parts of the Continuous Integration process.

Once the build is successful, it can be deployed to different runtime environments such as testing, pre-production, production, and so on. Deploying a WAR file in terms of a JEE application is normally the final stage in the deployment pipeline. However, after the deployment of this package into a pre-production environment, functional and security testing can be performed.

One of the biggest benefits of the pipeline is a faster feedback cycle. Identification of issues in the application in early stages and no dependency on manual effort make this entire end-to-end process more effective.

In the following chapters, we will see how Jenkins can be used to implement CI and CD practices in modernizing the culture of an organization.

Summary

Congratulations! We have reached the end of this chapter. So far, we have covered the basics of CI and have introduced Jenkins and its features. We have also completed the installation of Jenkins using generic package files. We also completed a quick tour of features available in the Jenkins dashboard. In addition to this, we have discussed the CI/CD pipeline and its importance in cultivating DevOps culture.

Now that we are able to use our automation server, Jenkins, we can begin creating a job and verify how Jenkins works. Before that, we will see how different configurations can be done in Jenkins in the next chapter.

2

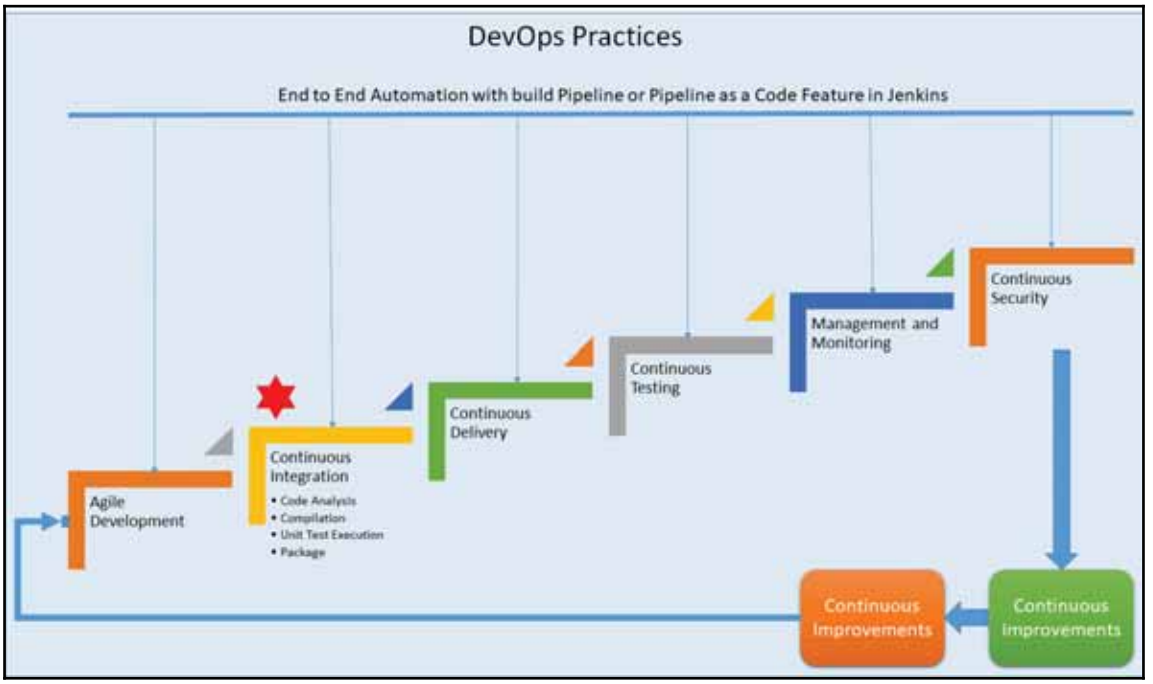
Installation and Configuration of Code Repository and Build Tools

We have seen the CI/CD pipeline in the previous chapter, where source code repositories and automated build were discussed in detail. SVN, Git, CVS, and StarTeam are some of the popular code repositories that manage changes to code, artifacts, or documents while Ant and Maven are popular build automation tools for Java applications.

This chapter describes in detail how to prepare an environment for application life cycle management and configure it with Jenkins—an open source **Continuous Integration** (**CI**) tool. It will cover how to integrate Eclipse and Jenkins so builds can be run from Eclipse as well. These are the major points that we will cover in this chapter:

- Overview of Jenkins
- Installing Java and configuring environment variables
- Installing and configuring Ant
- Installing Maven
- Configuring Ant, Maven, and JDK in Jenkins
- Overview of GitHub
- Creating a new build job in Jenkins with GitHub
- Eclipse and Jenkins integration

In this chapter, we will cover the configuration in Jenkins as part of CI best practice and as a part of our DevOps journey:



At the end of this chapter, we will know how to configure Jenkins and how to integrate different tools in Jenkins.

Overview of Jenkins

We have seen in `Chapter 1`, *Exploring Jenkins*, that the **Manage Jenkins** link on the dashboard is used to configure systems. Click on the **Global Tool Configuration** link to configure Java, Ant, Maven, and other third-party products' related information.

In this book, we will try to make things general and not operating system-specific. We have used Windows 10 for most of the sections in this book for CICD implementation, but it can be implemented on any operating system. We will specify some operating system-specific requirements if needed. Normally, path style changes and installation procedure changes, but the rest are the same irrespective of OS.

Installing Java and configuring the environment variables

In this section, we will cover installing Jenkins on both Windows and CentOS operating systems.

Installing Java on Windows 10

Go to `http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html` and download the Java installer for either the 32-bit or 64-bit operating system and follow the wizard to install it.

Execute the `java` and `javac` commands from the command window to verify the installation.

Installing Java on CentOS

If Java is not already installed on the system, then you can install it as follows:

1. Find Java-related packages in the CentOS repository and locate the appropriate package to install by using the following code:

```
[root@localhost ~]# yum search java
Loaded plugins: fastestmirror, refresh-packagekit, security
.
.
ant-javamail.x86_64 : Optional javamail tasks for ant
eclipse-mylyn-java.x86_64 : Mylyn Bridge:  Java Development
.
.
java-1.5.0-gcj.x86_64 : JPackage runtime compatibility layer for
GCJ
```

```
java-1.5.0-gcj-devel.x86_64 : JPackage development compatibility
layer for GCJ
java-1.5.0-gcj-javadoc.x86_64 : API documentation for libgcj
java-1.6.0-openjdk.x86_64 : OpenJDK Runtime Environment
java-1.6.0-openjdk-devel.x86_64 : OpenJDK Development Environment
java-1.6.0-openjdk-javadoc.x86_64 : OpenJDK API Documentation
java-1.7.0-openjdk.x86_64 : OpenJDK Runtime Environment
jcommon-serializer.x86_64 : JFree Java General Serialization
Framework
.
.
```

2. Now install the Java package in the local repositories by executing the `yum install` command as follows:

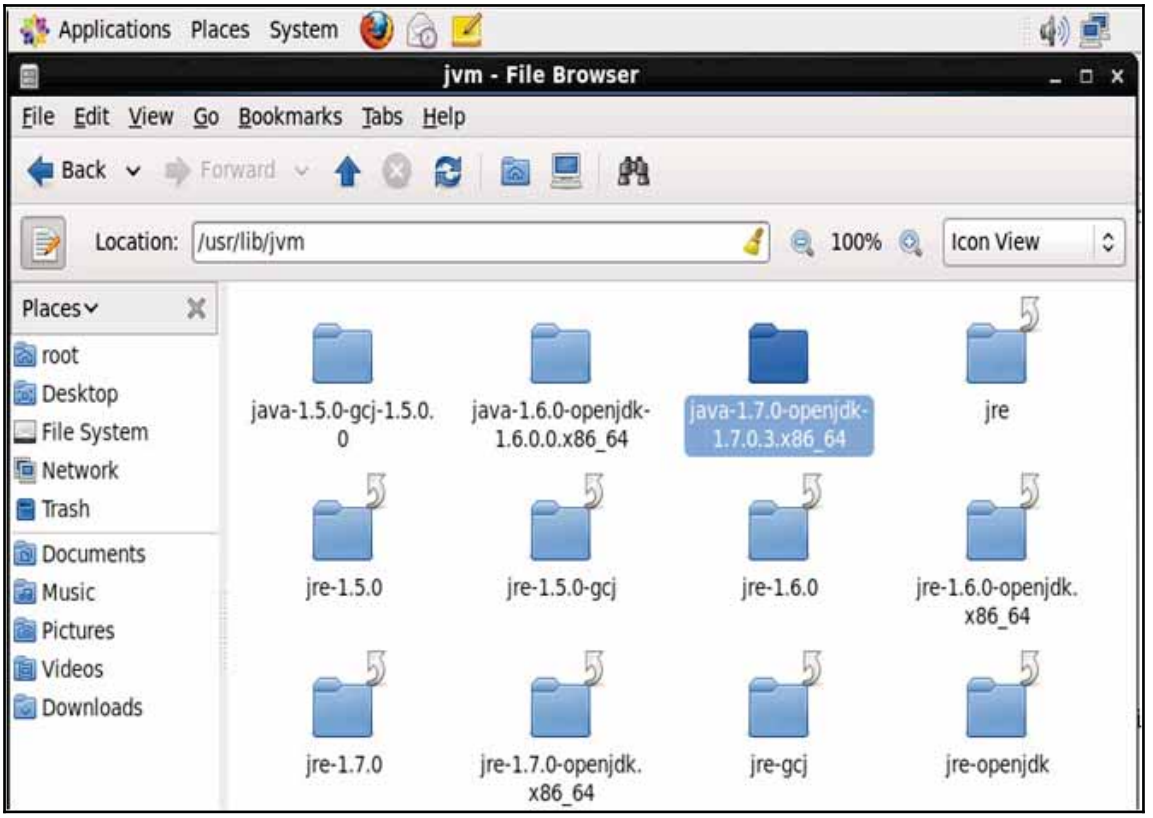
```
[root@localhost ~]# yum install java-1.7.0-openjdk.x86_64
Loaded plugins: fastestmirror, refresh-packagekit, security
Loading mirror speeds from cached hostfile
Setting up Install Process
Resolving Dependencies
--> Running transaction check
---> Package java-1.7.0-openjdk.x86_64 1:1.7.0.3-2.1.el6.7 will be
installed
--> Finished Dependency Resolution

Dependencies Resolved
.
.
Install       1 Package(s)

Total download size: 25 M
Installed size: 89 M
Is this ok [y/N]: y
Downloading Packages:
java-1.7.0-openjdk-1.7.0.3-2.1.el6.7.x86_64.rpm
|  25 MB     00:00
Running rpm_check_debug
Running Transaction Test
Transaction Test Succeeded
Running Transaction
  Installing : 1:java-1.7.0-openjdk-1.7.0.3-2.1.el6.7.x86_64
  1/1
  Verifying  : 1:java-1.7.0-openjdk-1.7.0.3-2.1.el6.7.x86_64
  1/1

Installed:
  java-1.7.0-openjdk.x86_64 1:1.7.0.3-2.1.el6.7
Complete!
```

Java is now installed successfully from the local repository.

Configuring environment variables

The following are the steps to configure environment variables:

1. Set the `JAVA_HOME` and `JRE_HOME` variables.
2. Go to `/root`.
3. Press *Ctrl* + *H* to list hidden files.
4. Find `.bash_profile` and edit it by appending the Java path, as shown in the following screenshot:



Installing and configuring Ant

Ant is a build tool. Download Ant from `https://ant.apache.org/bindownload.cgi` and unzip it.

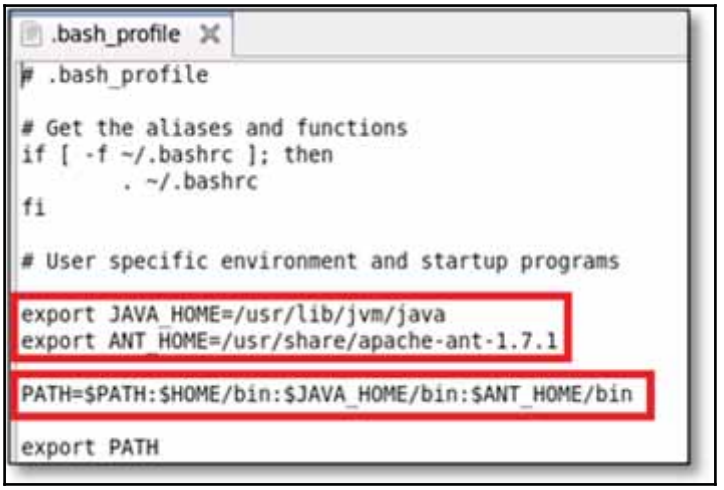
Configuring Ant in Windows

The following are the steps to configure Ant in CentOS:

1. Go to **Control Panel** | **All Control Panel Items** | **System and click on Advanced system settings** | **Advanced** | **Environment Variables...**.
2. Click on **New**.
3. Set the variable name as `ANT_HOME` and location and click **OK**.

Configuring Ant in CentOS

Set the `ANT_HOME` and `JAVA_HOME` environment variables:



There is an option available in Jenkins to install Ant or Maven automatically. We will see this in the Jenkins configuration section.

Installing Maven

Maven is a build tool. Download the Maven binary zip from `https://maven.apache.org/download.cgi` and extract it to the local system where Jenkins is installed:

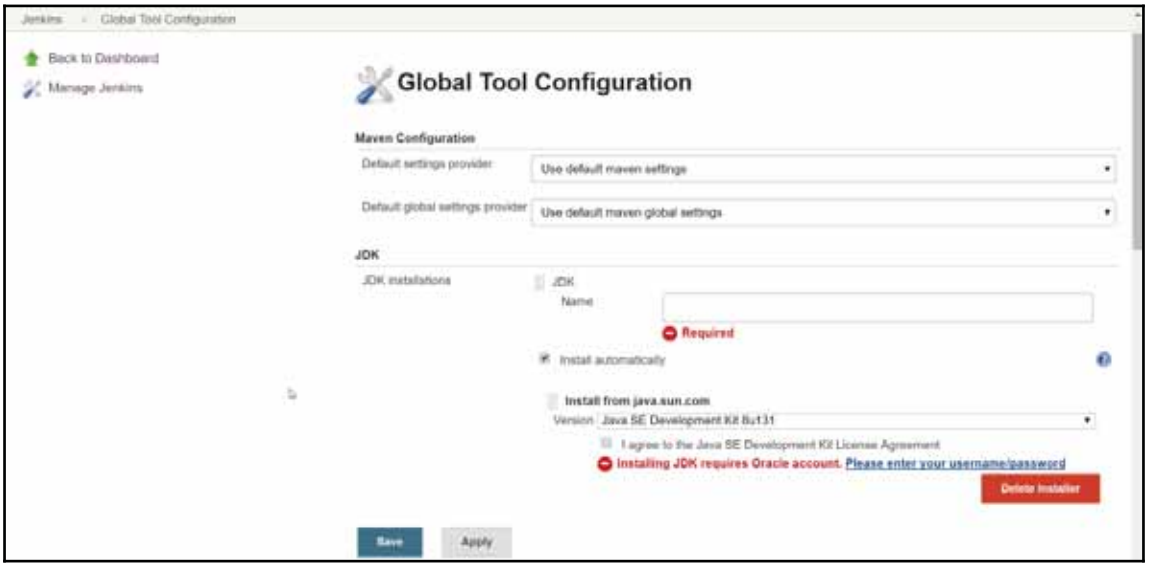
Once we have downloaded Java, Ant, and Maven, our next task is to configure them.

Configuring Ant, Maven, and JDK in Jenkins

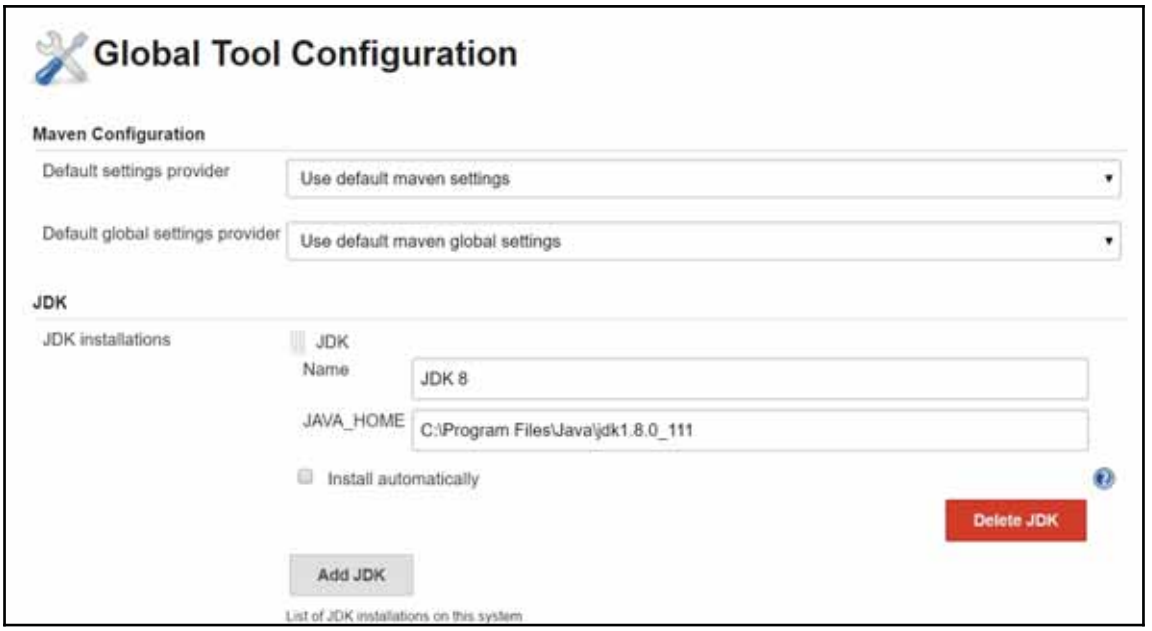
The following are the steps to configure Ant, Maven, and JDK in Jenkins:

1. Open the Jenkins dashboard in a browser with the URL `http://<ip_address>:8080`. Go to the **Manage Jenkins** section and click on **Global Tool Configuration**.

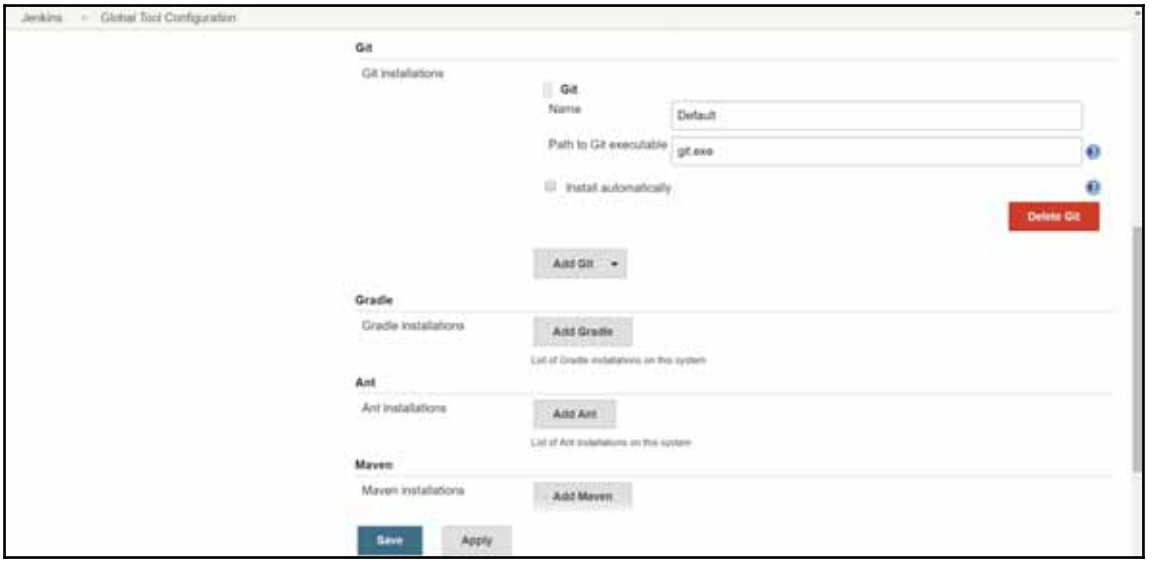
2. Configure Java based on the installation, as shown in the following screenshot. We can install it automatically:



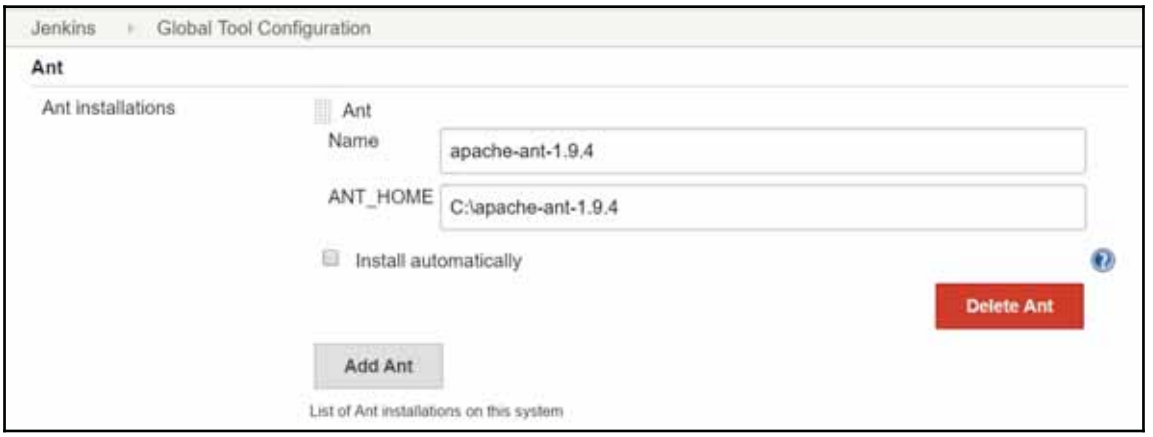
3. If Java is already installed, then uncheck the checkbox of **Install automatically** and give the `JAVA_HOME` path. If Jenkins, Ant, Maven, and Java are installed on CentOS, the path style will be different than this:



4. Download Git installer for Windows and install it on the system. Keep the settings as they are in Jenkins after you click on **Add Git**. If Jenkins, Ant, Maven, and Java are installed on CentOS, the path style will be different than this:



5. Click on **Add Ant** and provide a **Name** and `ANT_HOME` location. The value that we give in the **Name** box will be used in build job to identify the Ant version we want to use. This is similar practice for any tool that we will use. If Jenkins, Ant, Maven, and Java are installed on CentOS, the path style will be different than this:

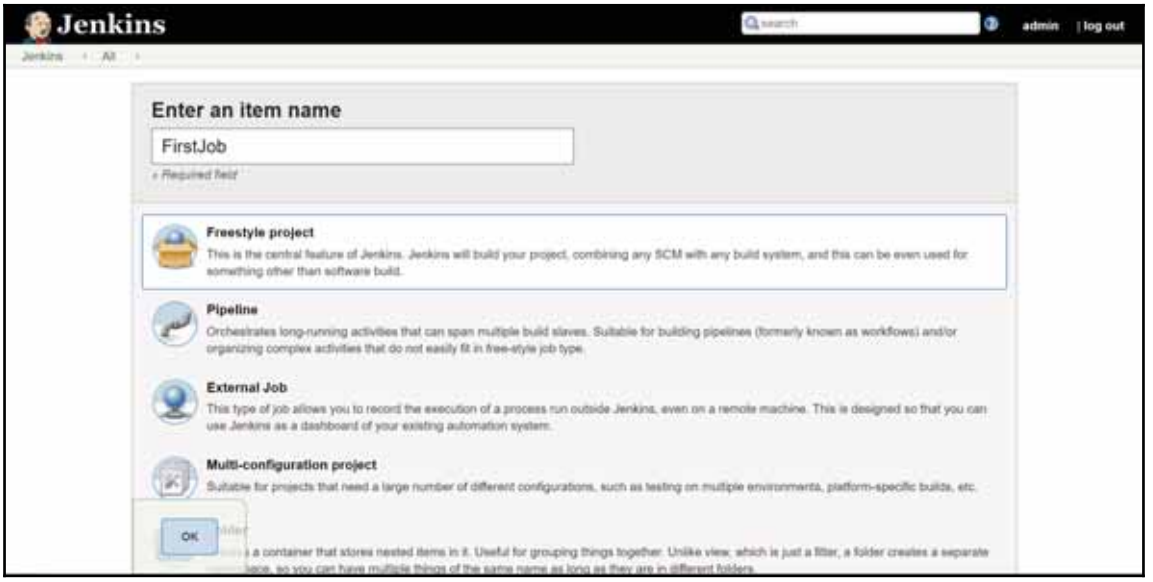


6. Click on **Add Ant** and provide a **Name** and `MAVEN_HOME` location. If Jenkins, Ant, Maven, and Java are installed on CentOS, the path style will be different than this:

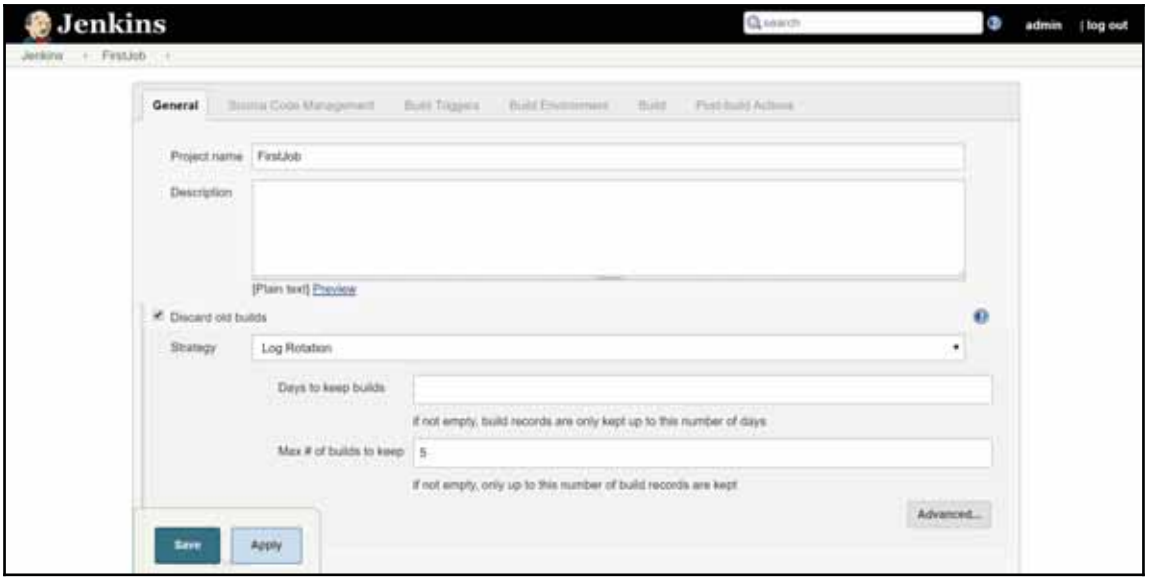
First job in Jenkins

Let's create a job in Jenkins that provides details about IP addresses and other configuration details. Basically, we will execute the `ipconfig` command. If you are using Linux OS, you can execute `ifconfig`. The following are the steps to create your first job in Jenkins:

1. On the Jenkins dashboard, click on **New Item**.
2. Enter the item's name.
3. Select **Freestyle project**.
4. Click **Ok**:

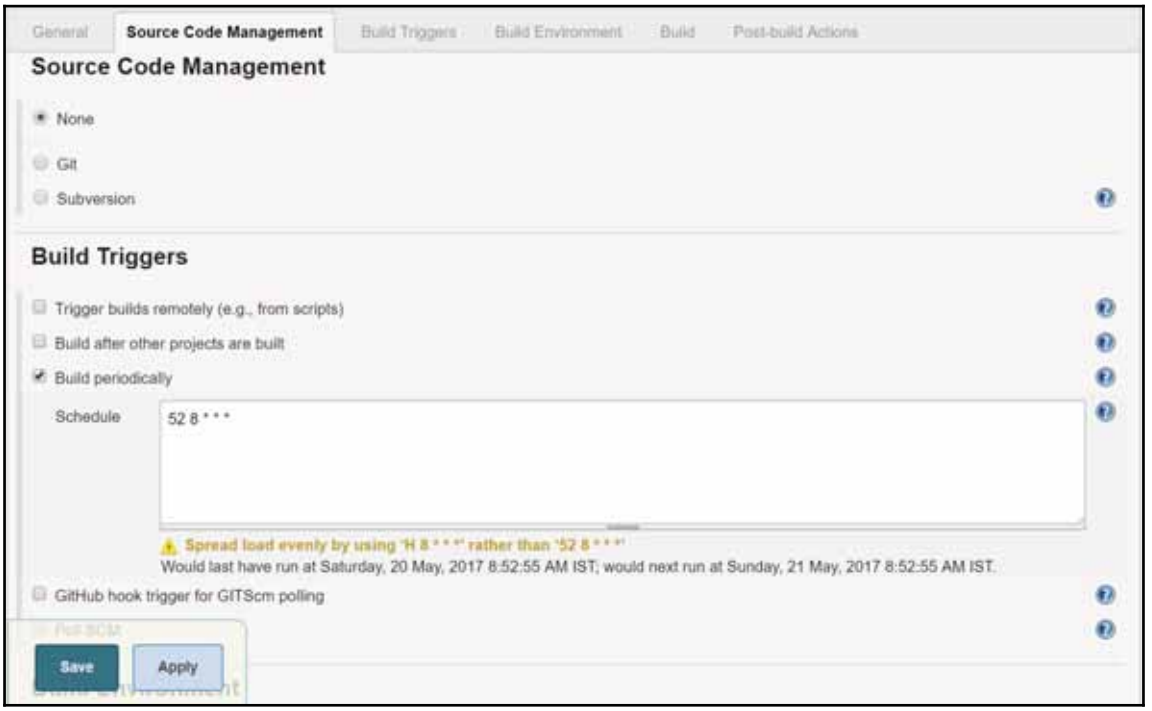


5. Give a **Project name**:

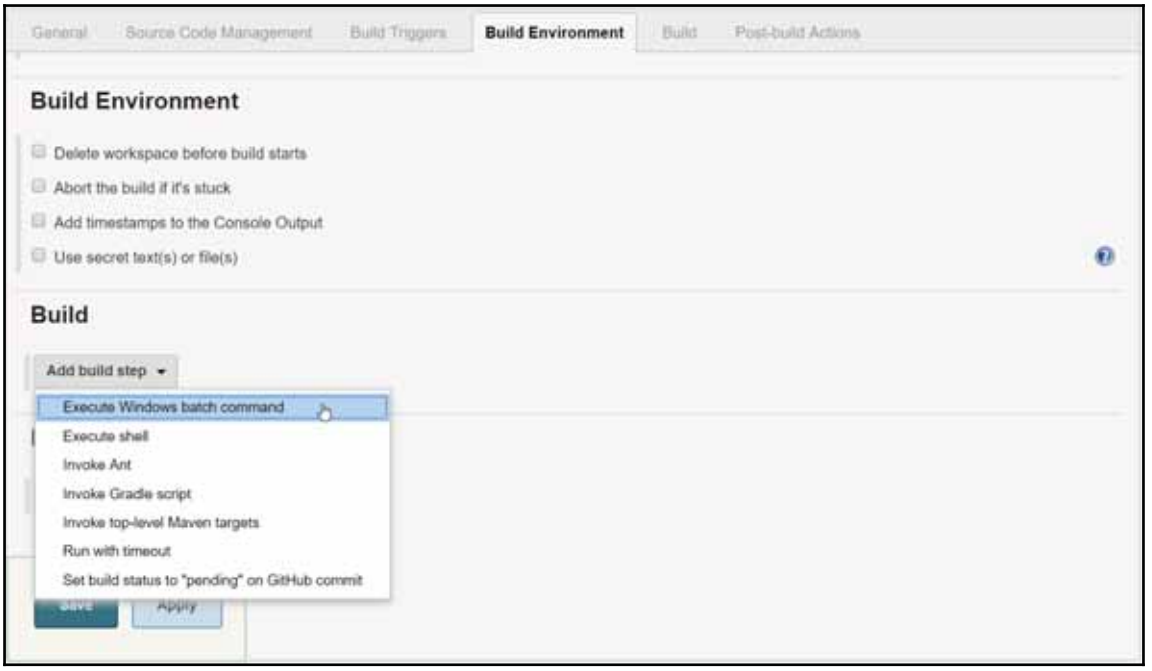


6. Select **None** in **Source Code Management**.

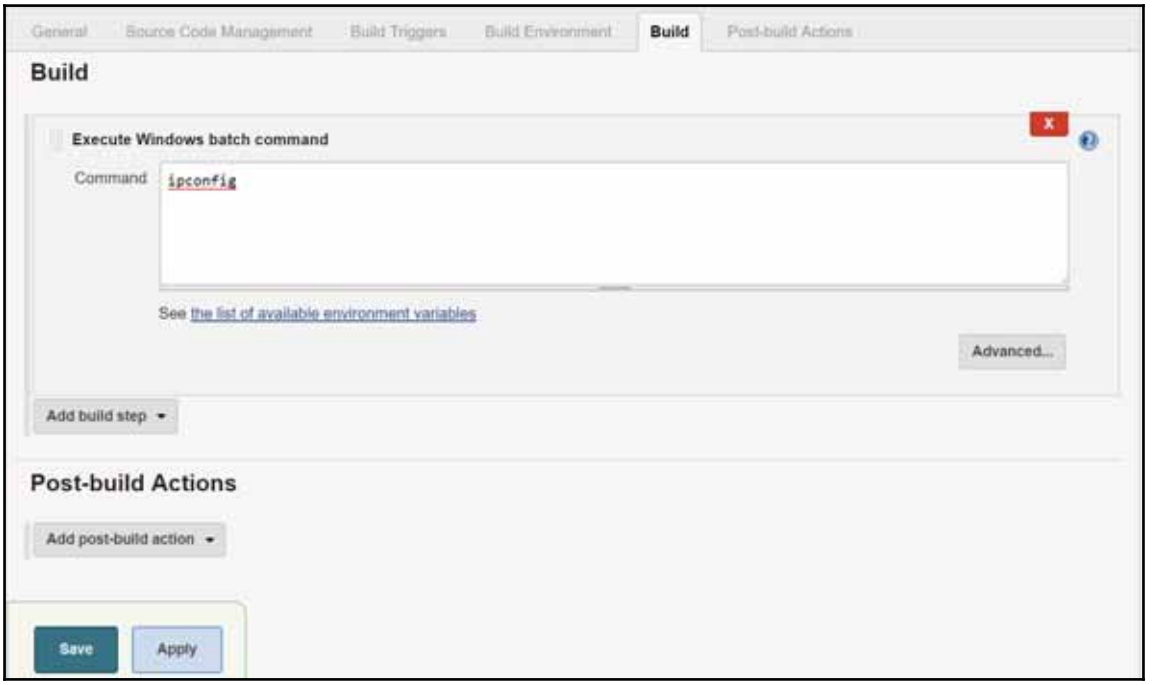
7. In the **Build Triggers** section, select **Build periodically** and give cron syntax in **Schedule**. It will always run at 8:52 AM in the morning:



8. In **Build**, click on **Execute Windows batch command**. To execute on CentOS or other flavours of Linux, select **Execute Shell**, and the command will be `ifconfig`:

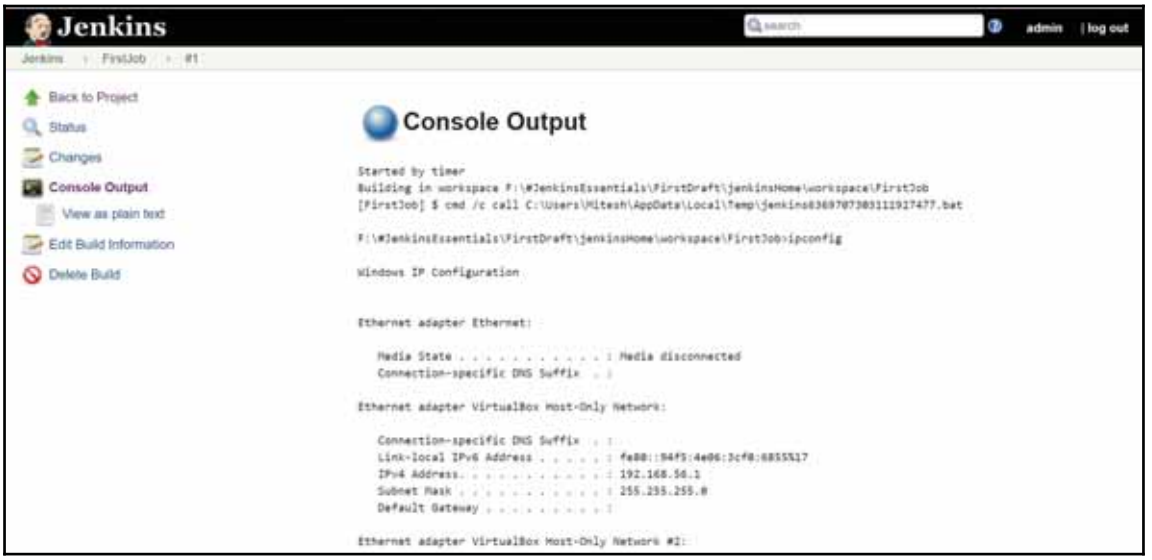


9. In the textbox, write the `ipconfig` command to get the IP address of the system.
10. Click on **Save**:

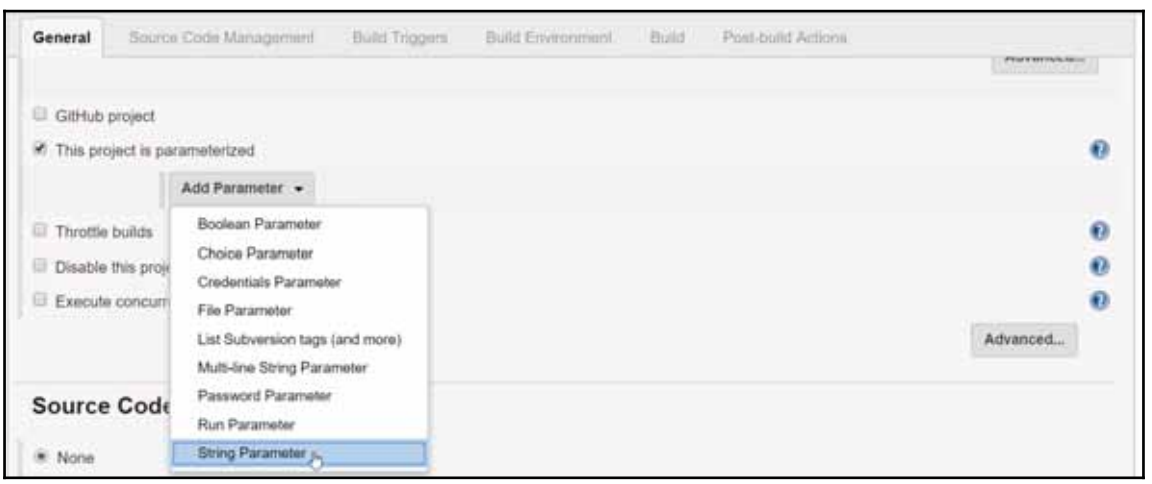


11. Click on **Build now**:

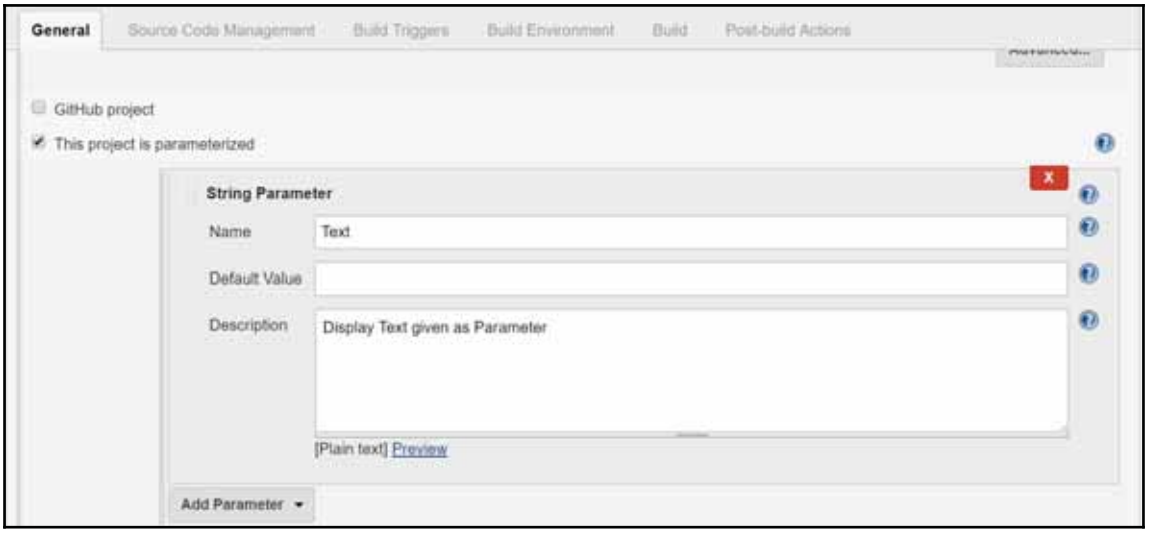
12. Observe the **Build History** and click on the blue ball to go to **Console Output**. Bingo! We have created our first job in Jenkins:



13. To add a parameter in the Jenkins build, click on **This project is parameterized**. Click on **Add Parameter**. Select **String Parameter**:



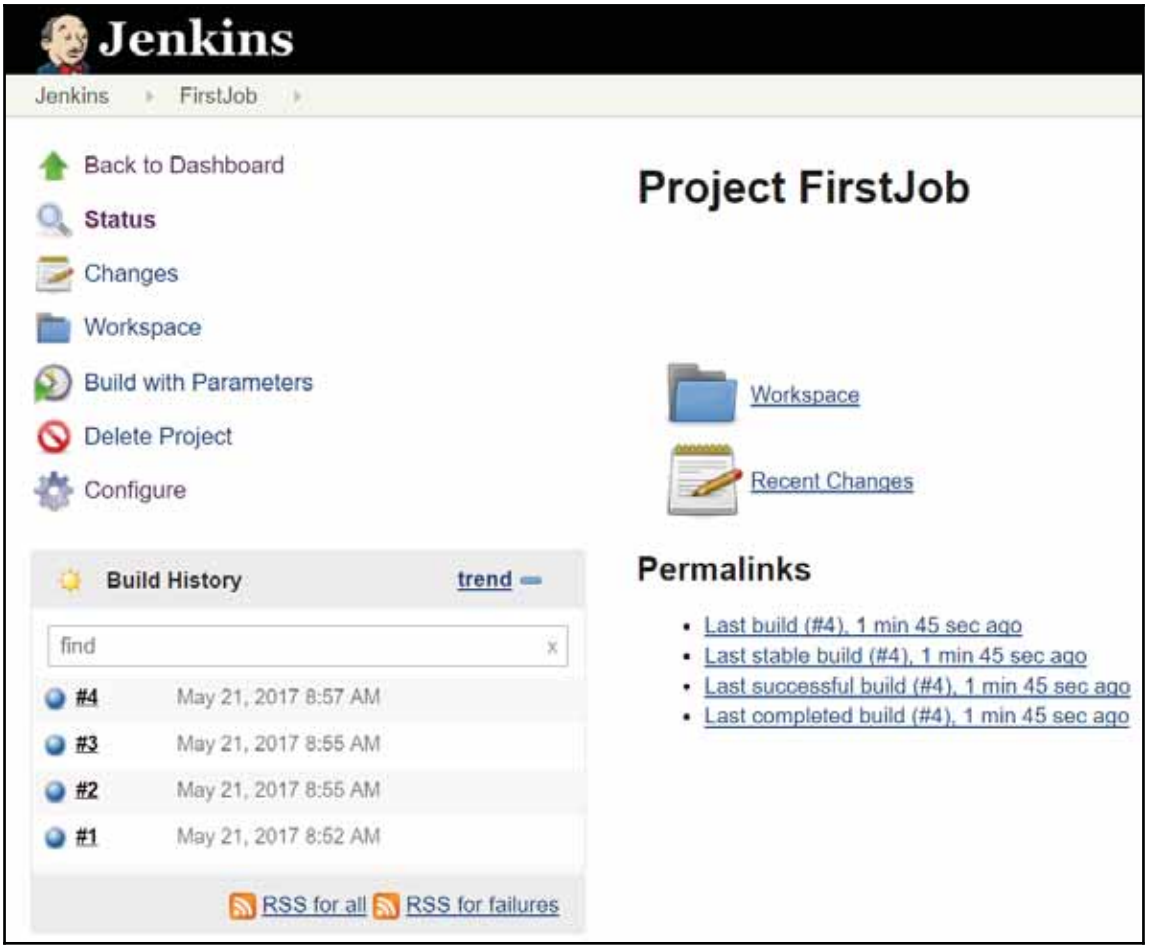
14. Provide a name and description. Click on **Save**:



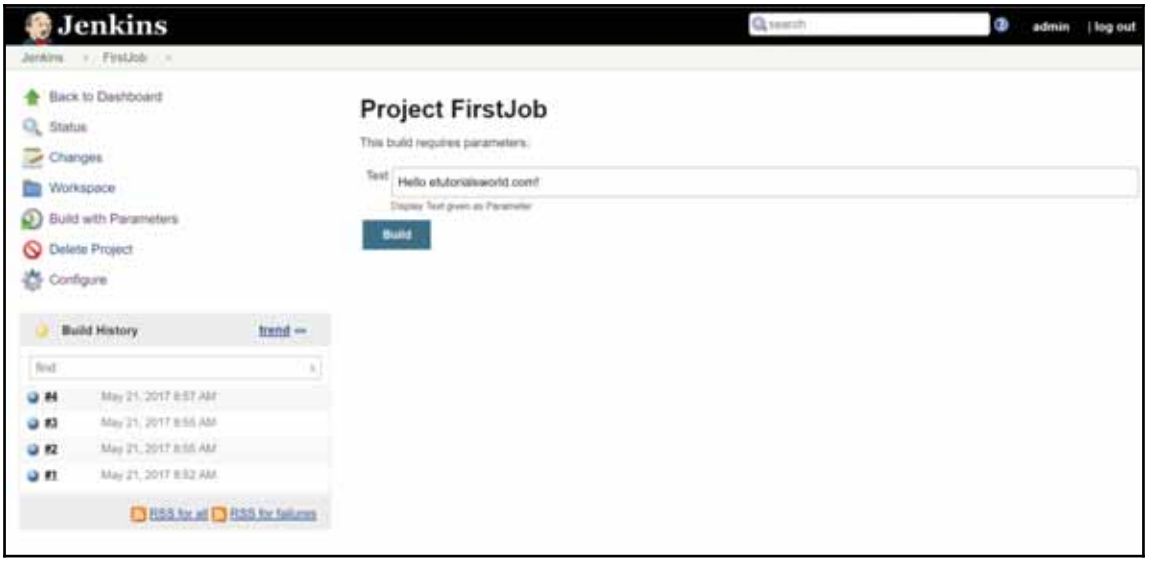
15. In the **Build** step, write the following commands in the **Execute Windows Batch Command** box:

```
echo %JOB_NAME%
echo %Text%
```

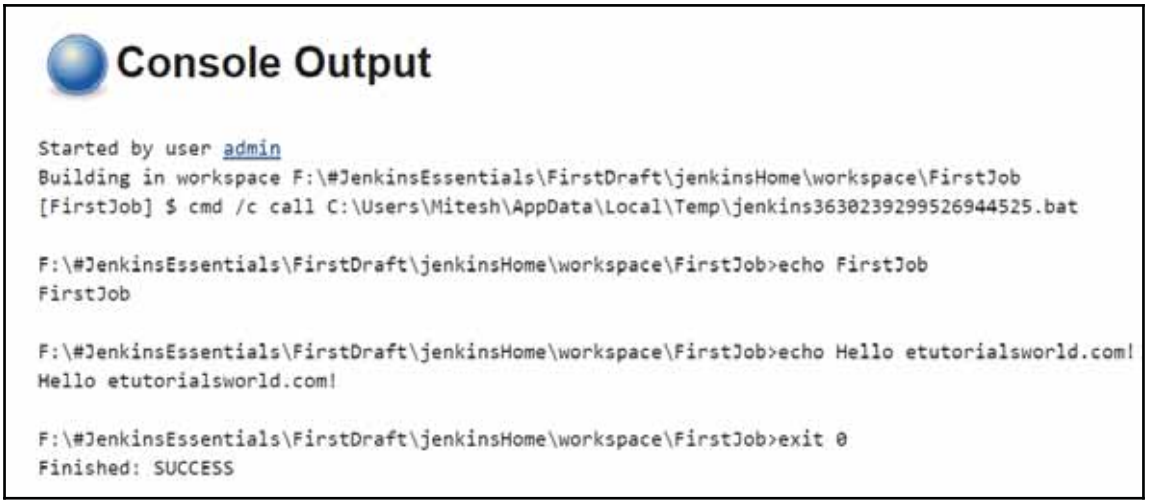
16. Click on **Build with Parameters**:



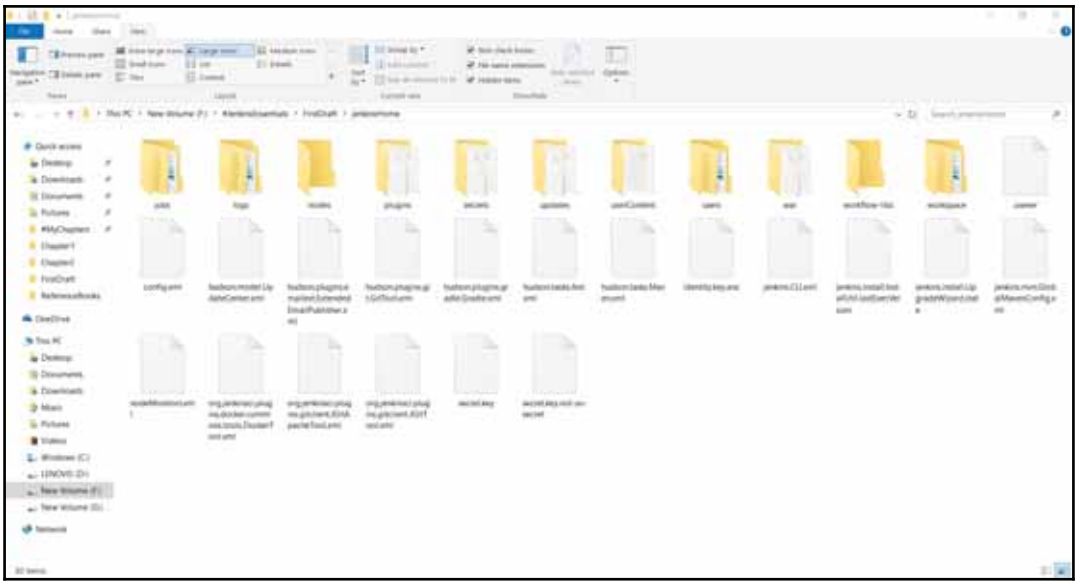
17. It will ask for the parameter. Provide some text and click on **Build**:



18. Verify the **Console Output**:

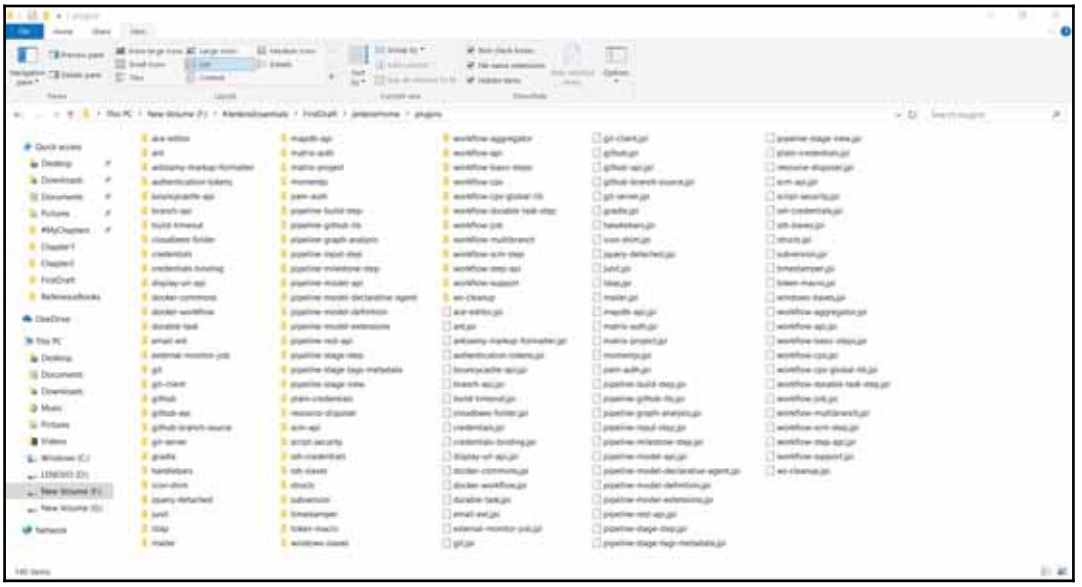


19. Go to `JENKINS_HOME` and try to find out what each directory contains:



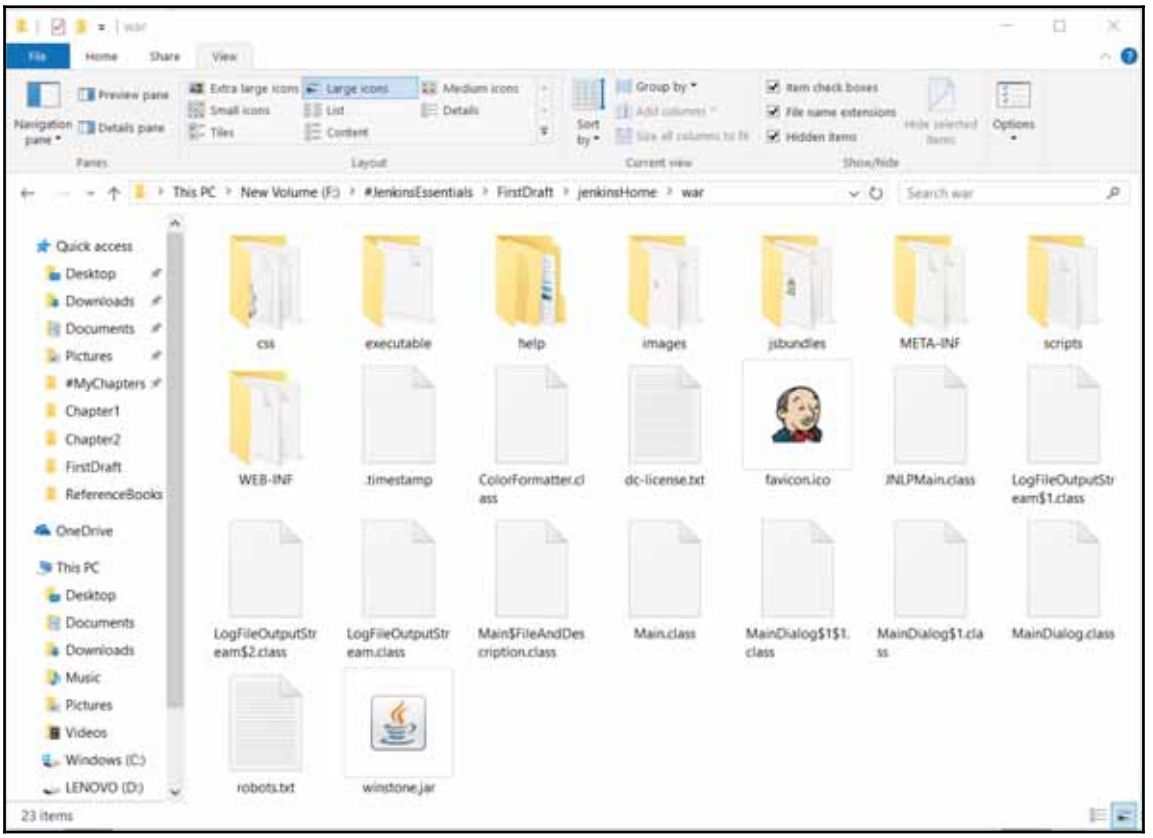
All the Directories available in JENKINS_HOME folder

20. In the plugins directory, all installed plugins are available:



All the Plugins available in JENKINS_HOME directory

21. The `war` directory contains the actual files that are used in the Jenkins application:

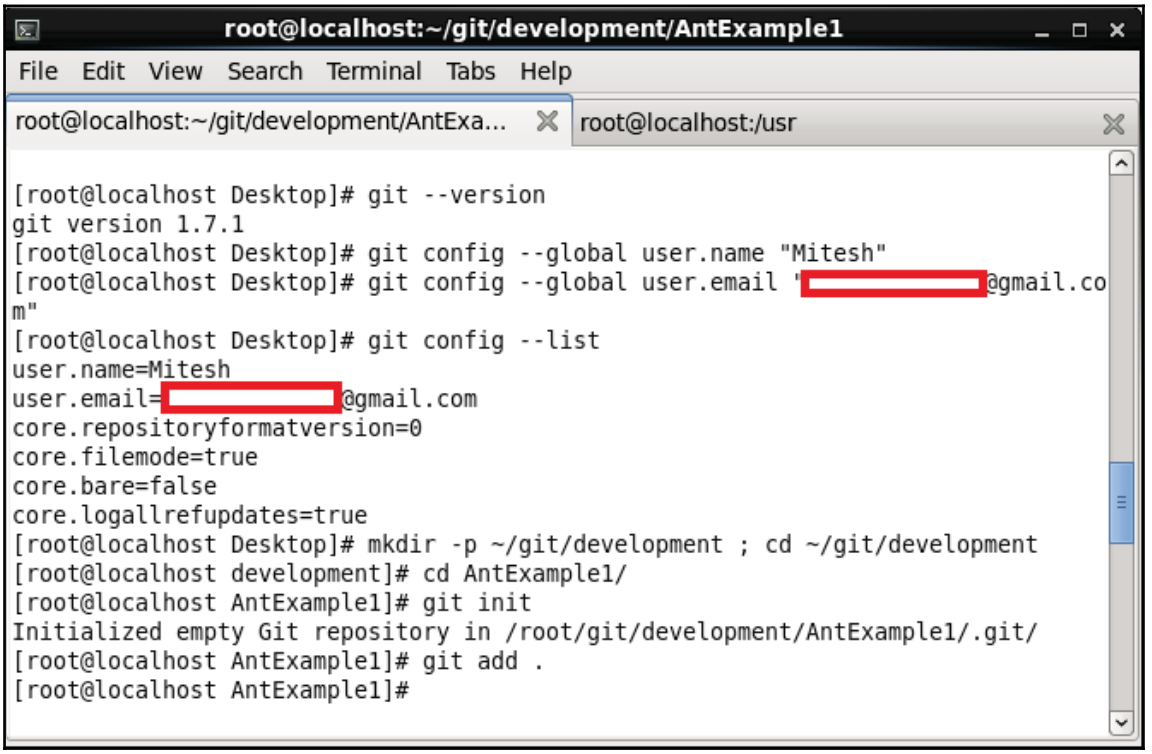


Installing and configuring the Git repository on CentOS

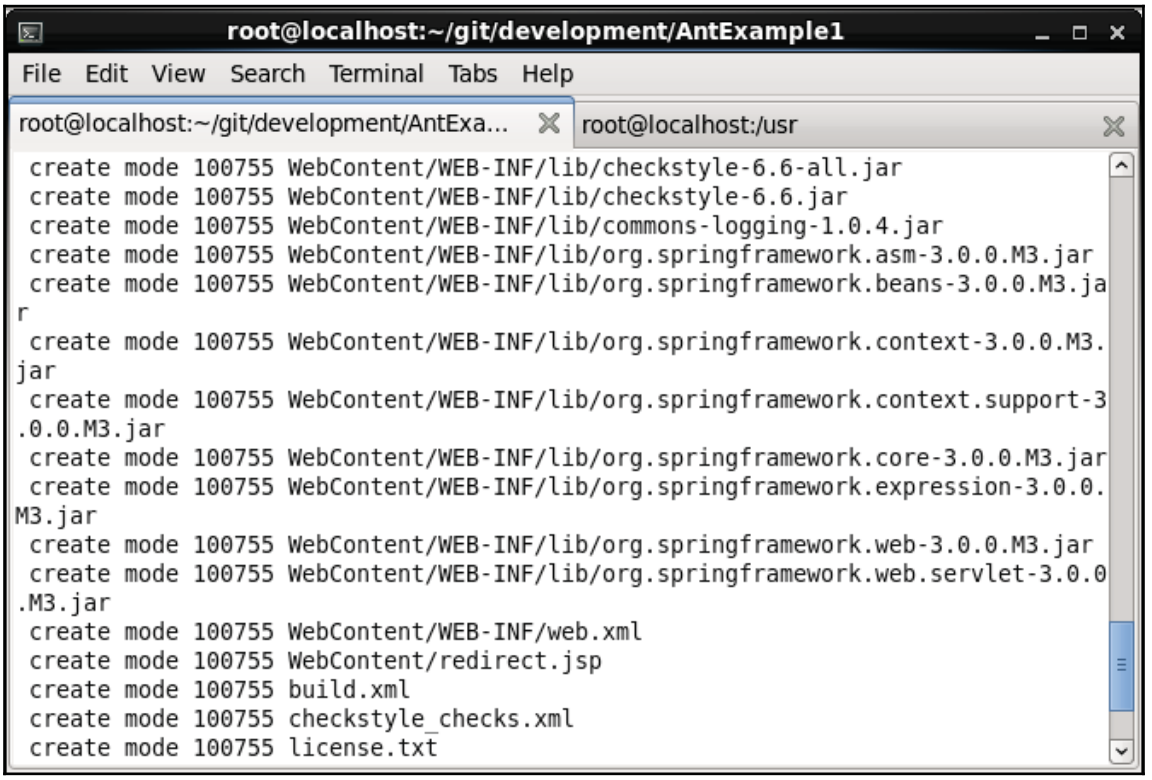
Git is a free and open source distributed version control system. In this section, we will try to install and configure Git:

1. Open a terminal in the CentOS-based system and execute the `yum install git` command in the terminal.
2. Once it is successfully installed, verify the version with the `git --version` command.

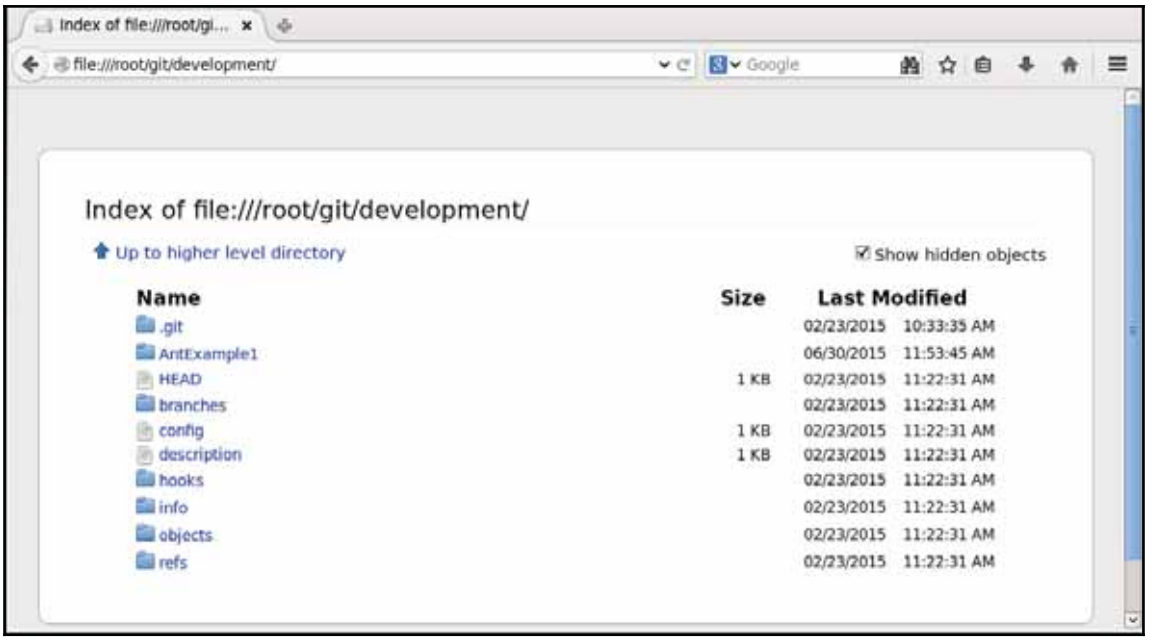
3. Submit information about the user with the use of the `git config` command so that commit messages will be generated with the correct information attached.
4. Provide the name and email address to embed into commits.
5. To create a workspace environment, create a directory called `git` in the home directory and then create a subdirectory inside of that called `development`.
6. Use `mkdir -p ~/git/development ; cd ~/git/development` in the terminal.
7. Copy the `AntExample1` directory into the development folder.
8. Convert an existing project into a workspace environment by using the `git init` command.
9. Once the repository is initialized, add files and folders:



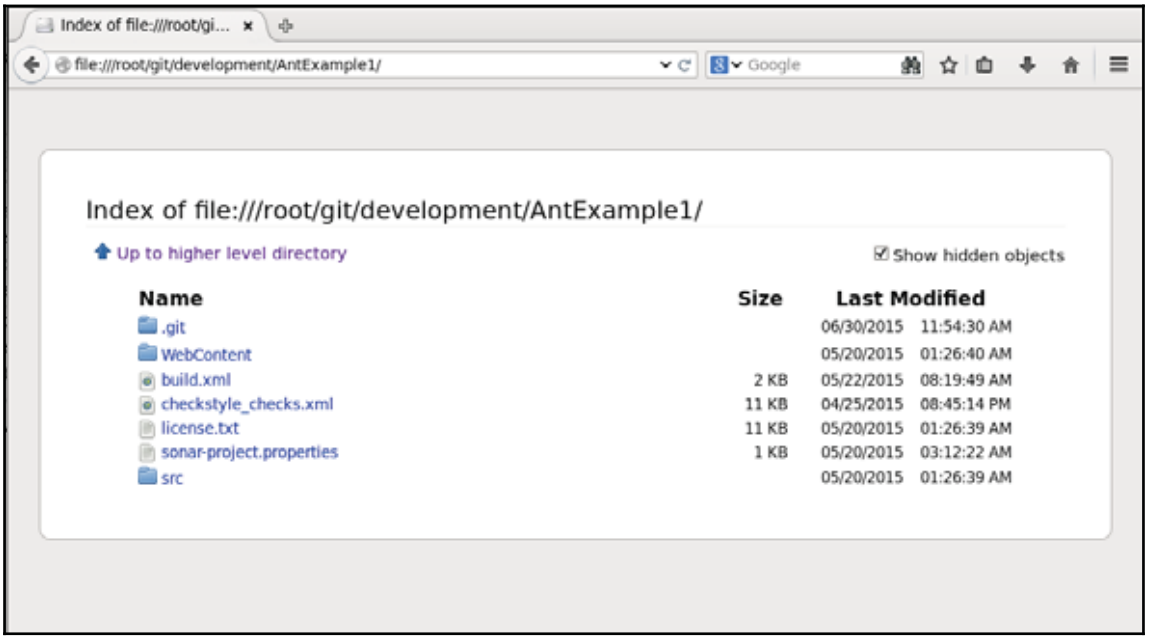
10. Commit by executing `git commit –m "Initial Commit" –a`:



11. Verify the Git repository:



12. Verify the project in the Git repository:



In the next section, we will cover how to use Git and GitHub to check out the source code and execute the build.

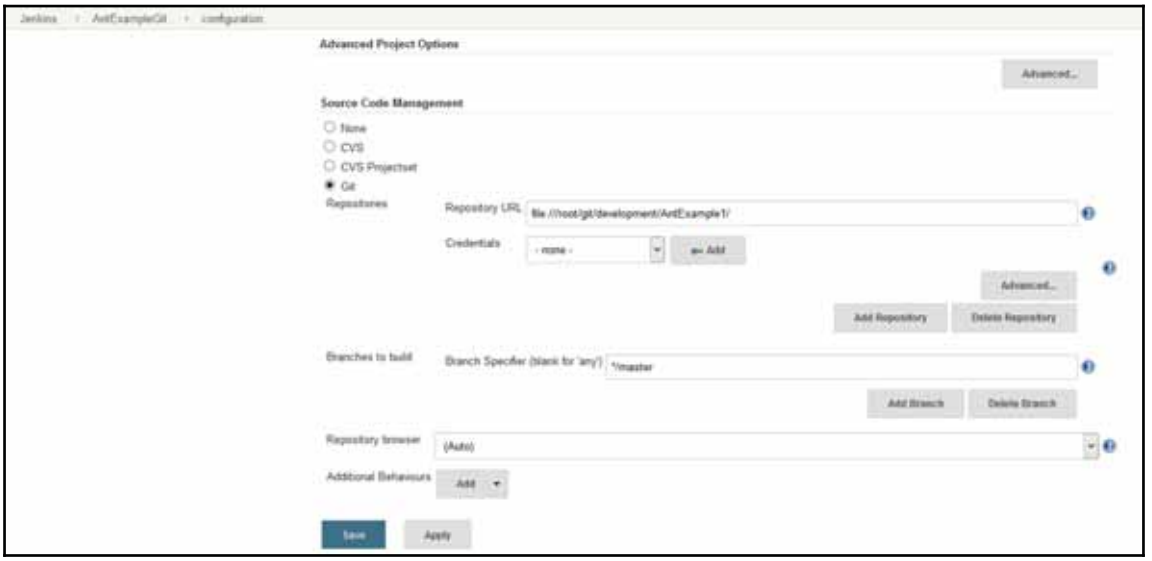
Creating a new build job in Jenkins with Git and GitHub

The following are the steps required to create a new build job in Jenkins with Git and GitHub:

1. On the Jenkins dashboard, click on **Manage Jenkins** and select **Manage Plugins**. Click on the **Available** tab and write `Github Plugin` in the search box.
2. Click the checkbox and click on the **Download now and install after restart** button.
3. Restart Jenkins:

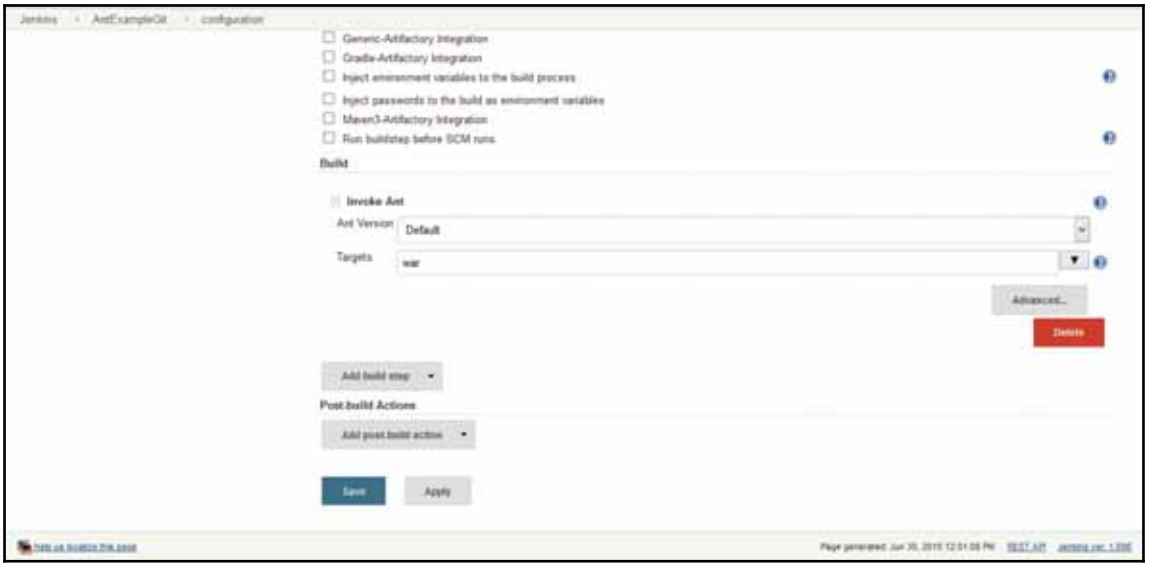
4. Create a new **Freestyle project**. Provide an item name and click on **OK**:

5. Configure **Git** in the **Source Code Management** section:

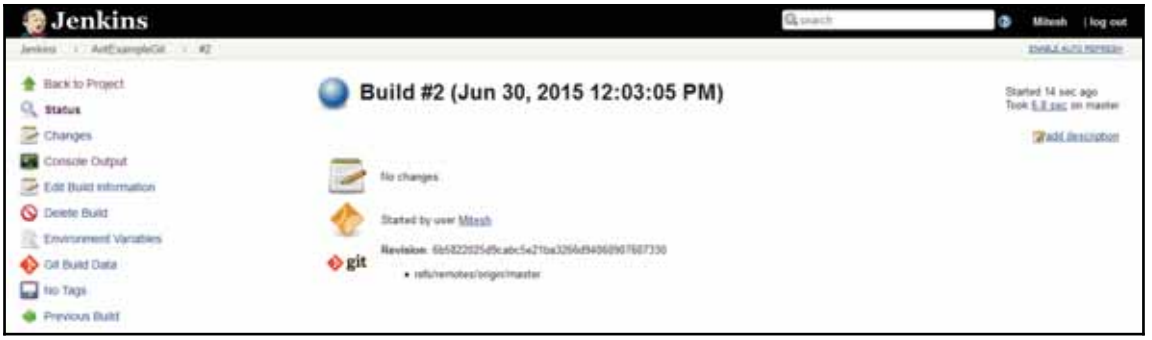


6. In **Repository URL**, we can provide a GitHub URL, which will work fine for publicly accessible projects. We can specify branch too:

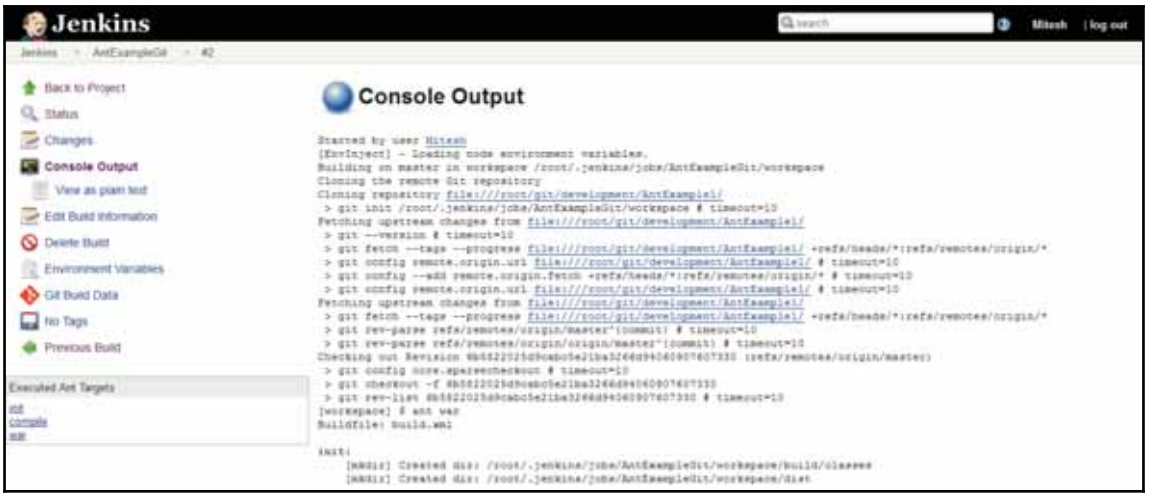
7. Add an **Invoke Ant** build step by clicking on **Add build step**:



8. Execute the build:



9. Click on the **Console Output** to see the progress of the build:



10. Once the build is successful, verify the workspace in the build job.

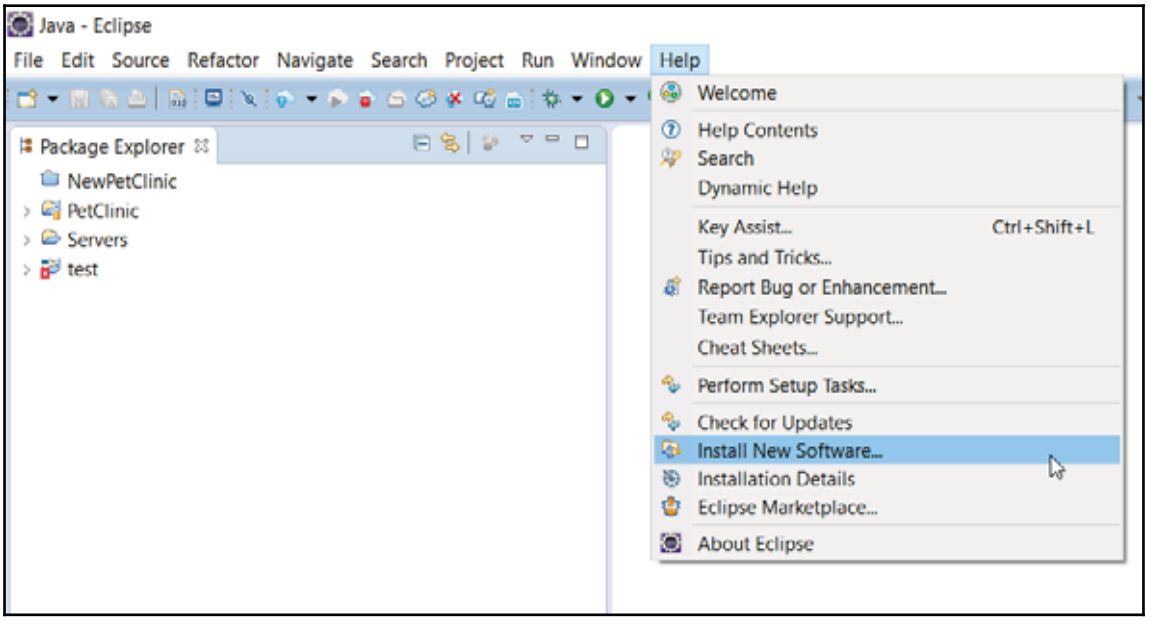
Done!

Eclipse and Jenkins integration

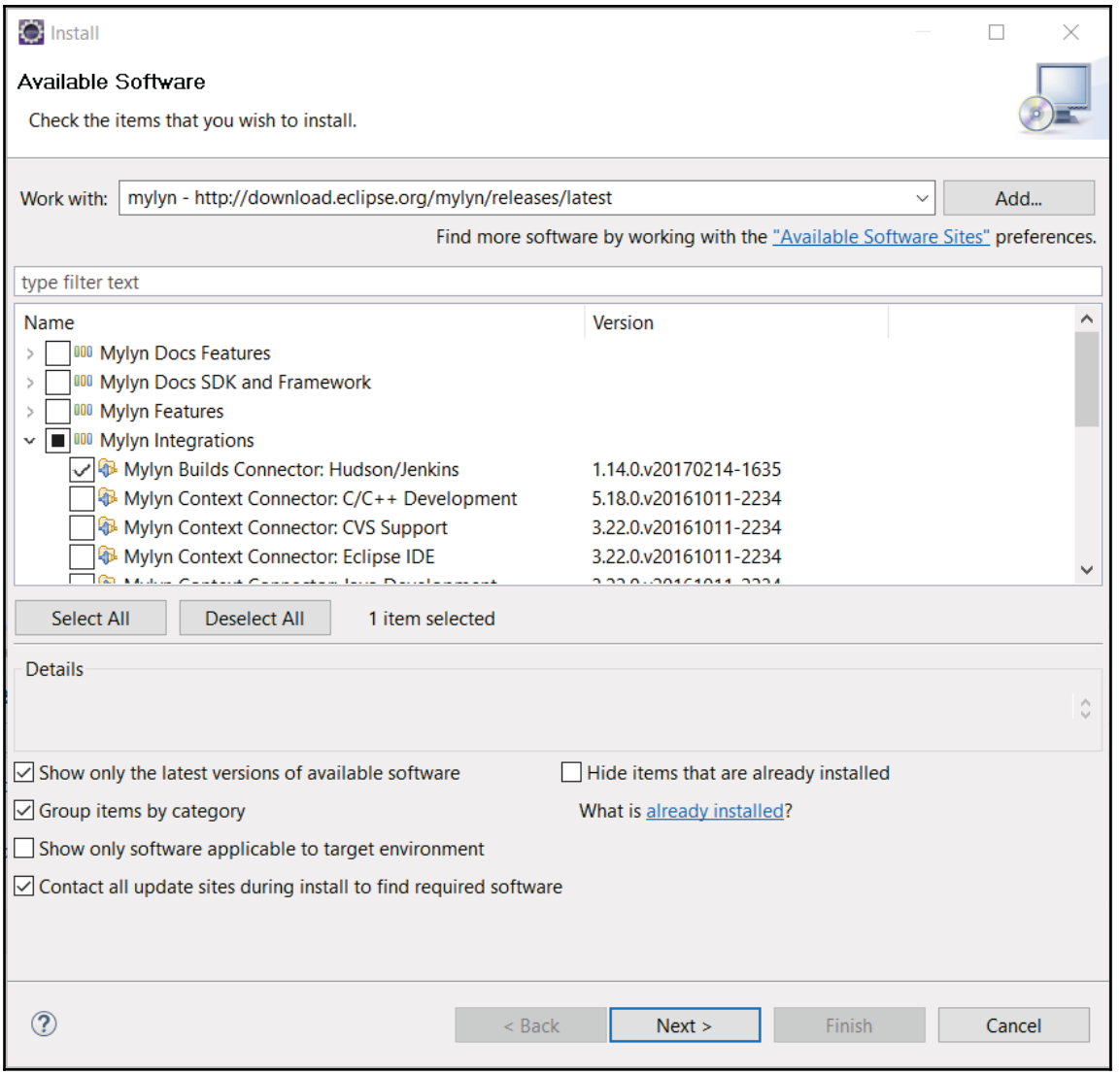
Can we execute a Jenkins job from Eclipse?

Yes, by following these steps:

1. Go to **Help** | **Install New Software...**:

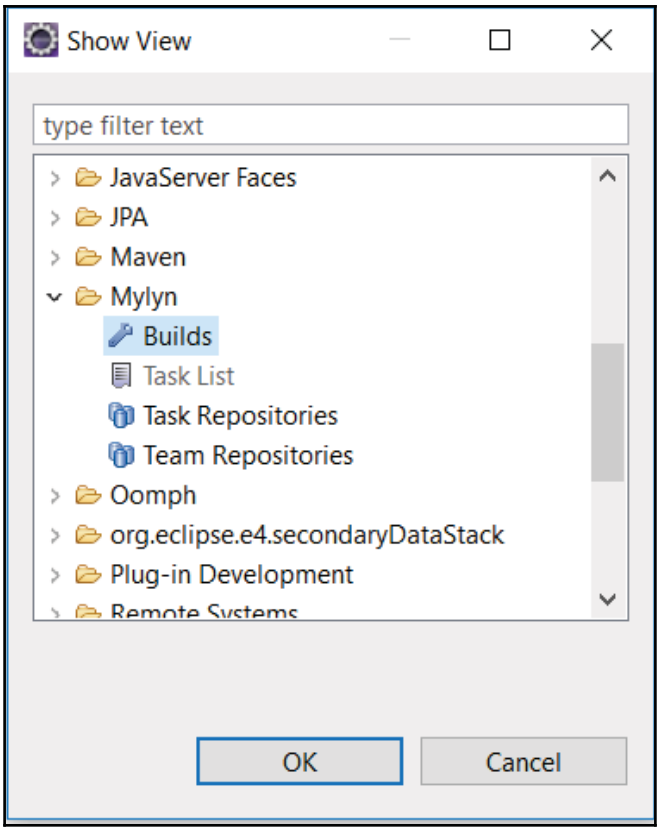


2. Add a site for `Mylyn` and click on **Next**:

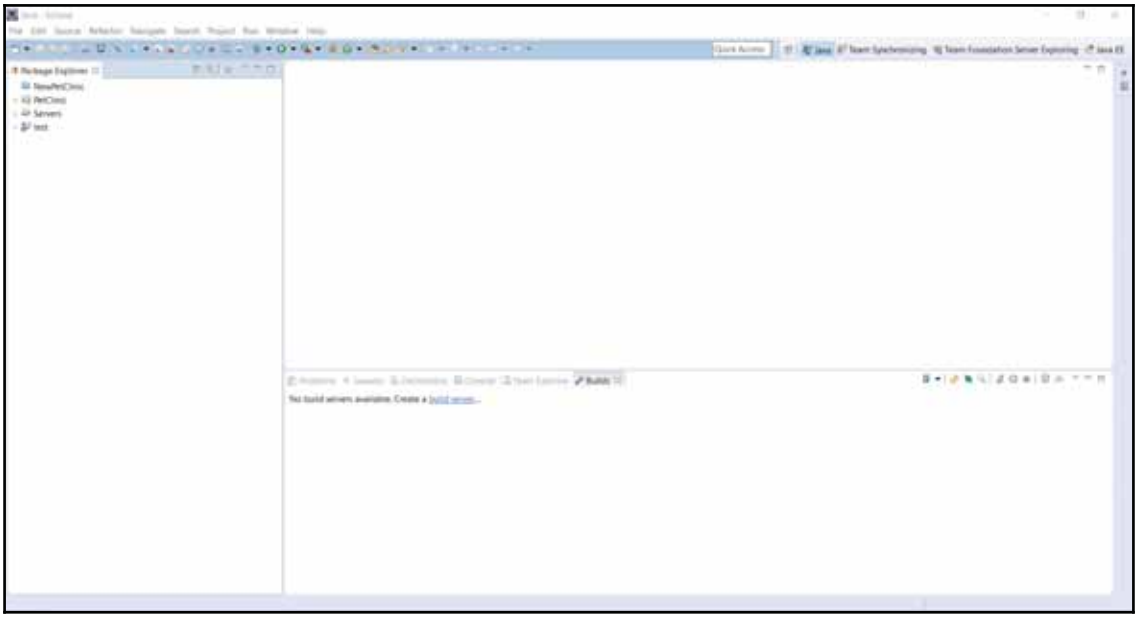


3. Review the items to be installed and click on **Next**.
4. Accept the terms of the license agreement.
5. Click on **Finish**.
6. It will start installing the **Mylyn** package. Once it is finished, restart Eclipse.

7. In the Windows menu, click on **Views**.
8. Select **Mylyn** and click on **Builds**.
9. Click **OK**:

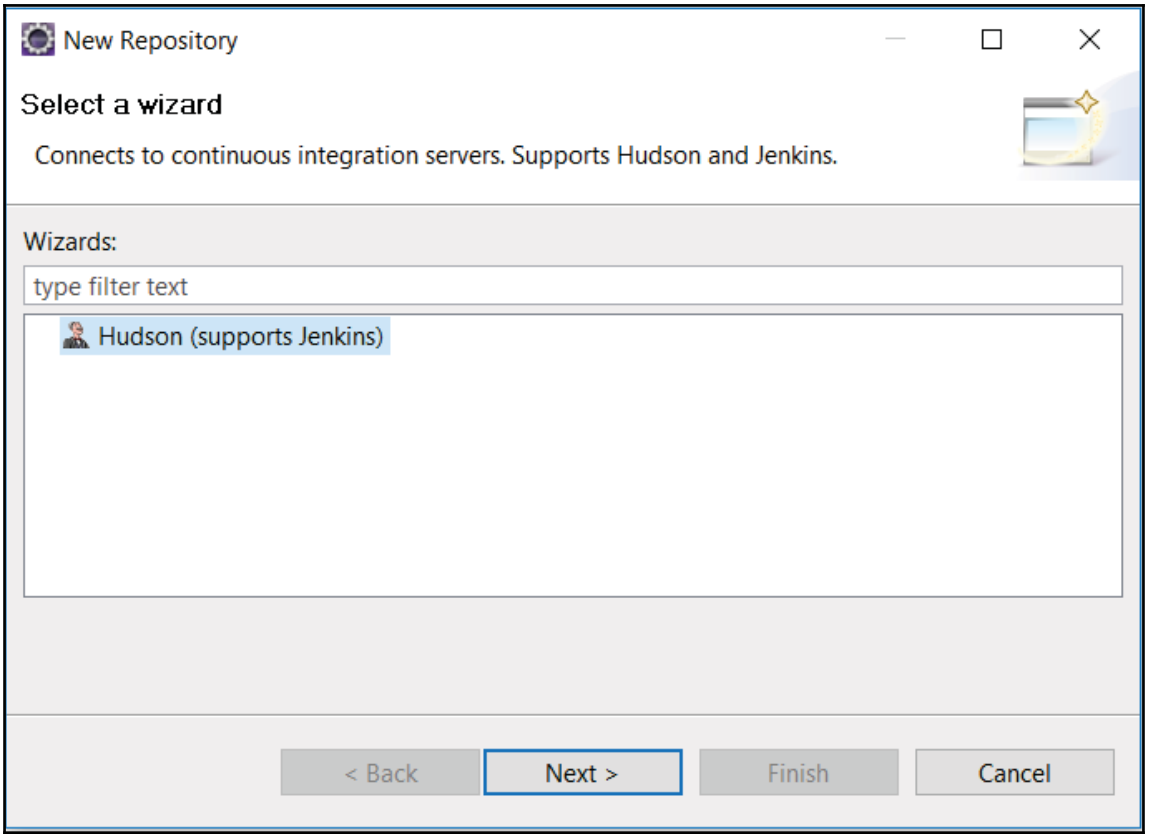


10. In the **Builds** section, click on the **build server** link:

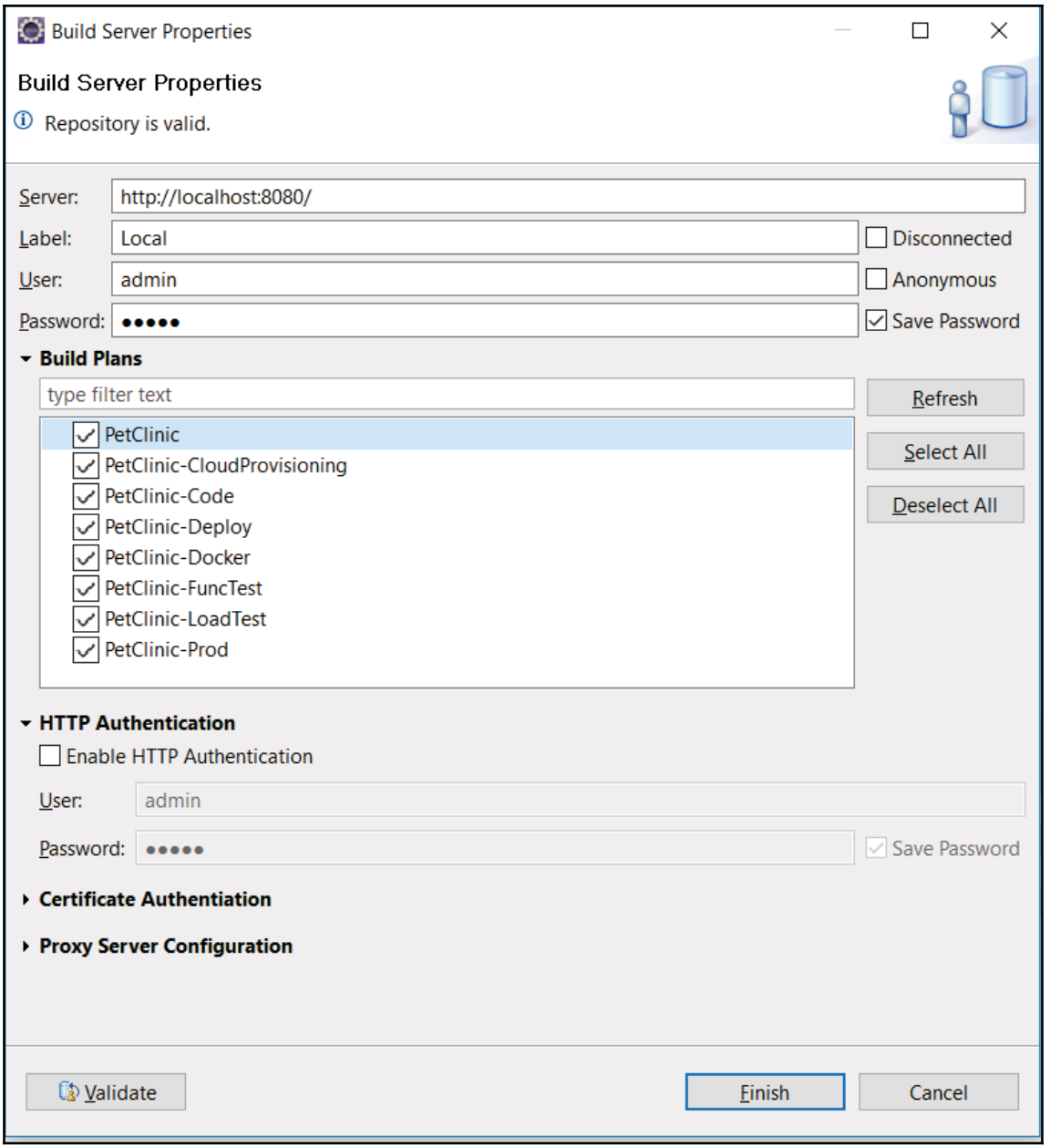


Build Section is Eclipse IDE with No build server configured

11. Select **Hudson (supports Jenkins)** and click on **Next**:

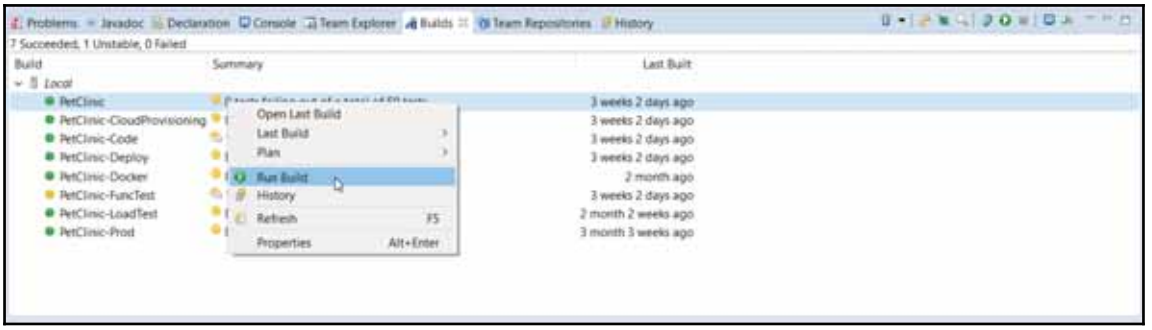


12. Provide Jenkins **Server**, **User**, and Password details. Click **Finish**:



13. Find the list of jobs in the **Builds** section.

14. Select any job and click on **Run Build** to execute it from Eclipse:



Try other options as an exercise.

Summary

Hooray! We have reached the end of this chapter. We have covered how to prepare an environment for CI by configuring Java, Ant, and Maven. We have also seen how to configure repositories and build tools in Jenkins. Finally, we have also covered how to integrate Integrated Development Environments with Jenkins so we can execute build jobs from Eclipse itself.

We have also created our first job, installed Git on a local machine, and created a job to access that Git repository in order to access the source code.

In the next chapter, we will configure sample applications for CI.

3

Managing Code Quality and Notifications

So far we have seen how to set up an environment to use Jenkins for Continuous Integration and we have also configured build tools in Jenkins. Integration of Eclipse with Jenkins was also covered and that will help developers to easily execute Jenkins jobs from the IDE.

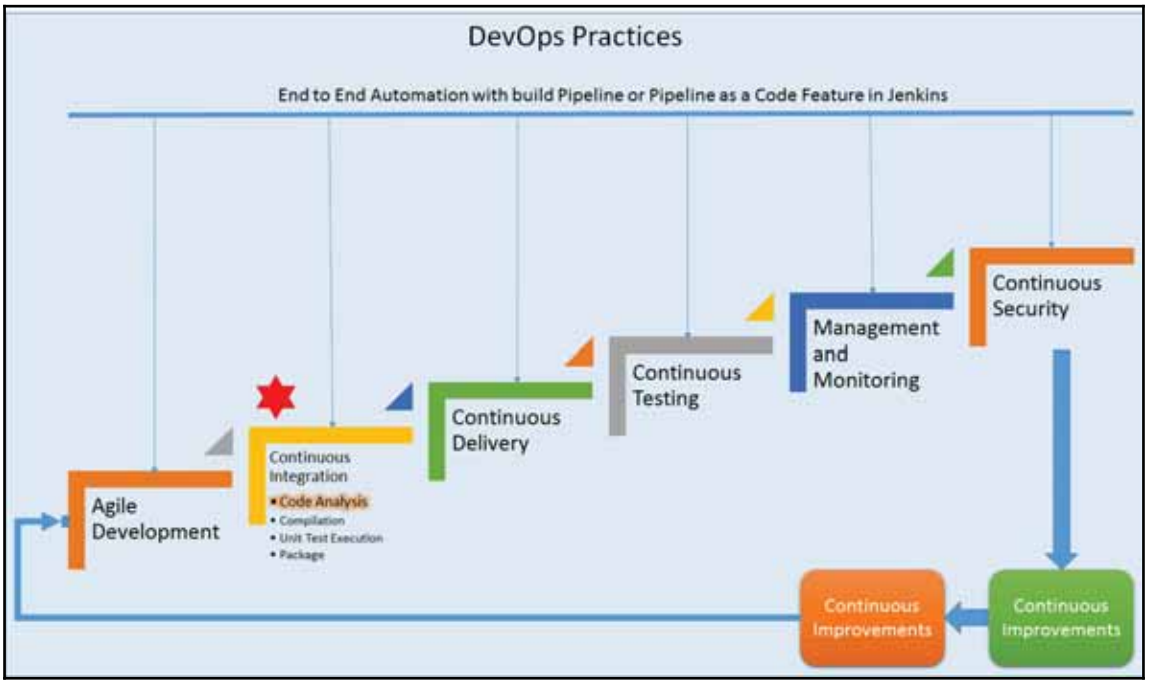
We will start our journey with Continuous Code Quality—static code analysis-- and it will be followed by Continuous Integration, Continuous Delivery, Continuous Testing, Continuous Deployment, Continuous Monitoring, and Continuous Security. For static code analysis, we will use **SonarQube** to analyze a spring-based Java project.

SonarQube is an open source quality management platform for maintaining Continuous Code Quality.

In this chapter, we will cover the following topics:

- Jenkins 2.x integration with Sonar 6.3
- Quality Gate plugin
- Email notifications on build status

In this chapter, we will cover static code analysis as part of Continuous Integration practice as a part of our DevOps journey:

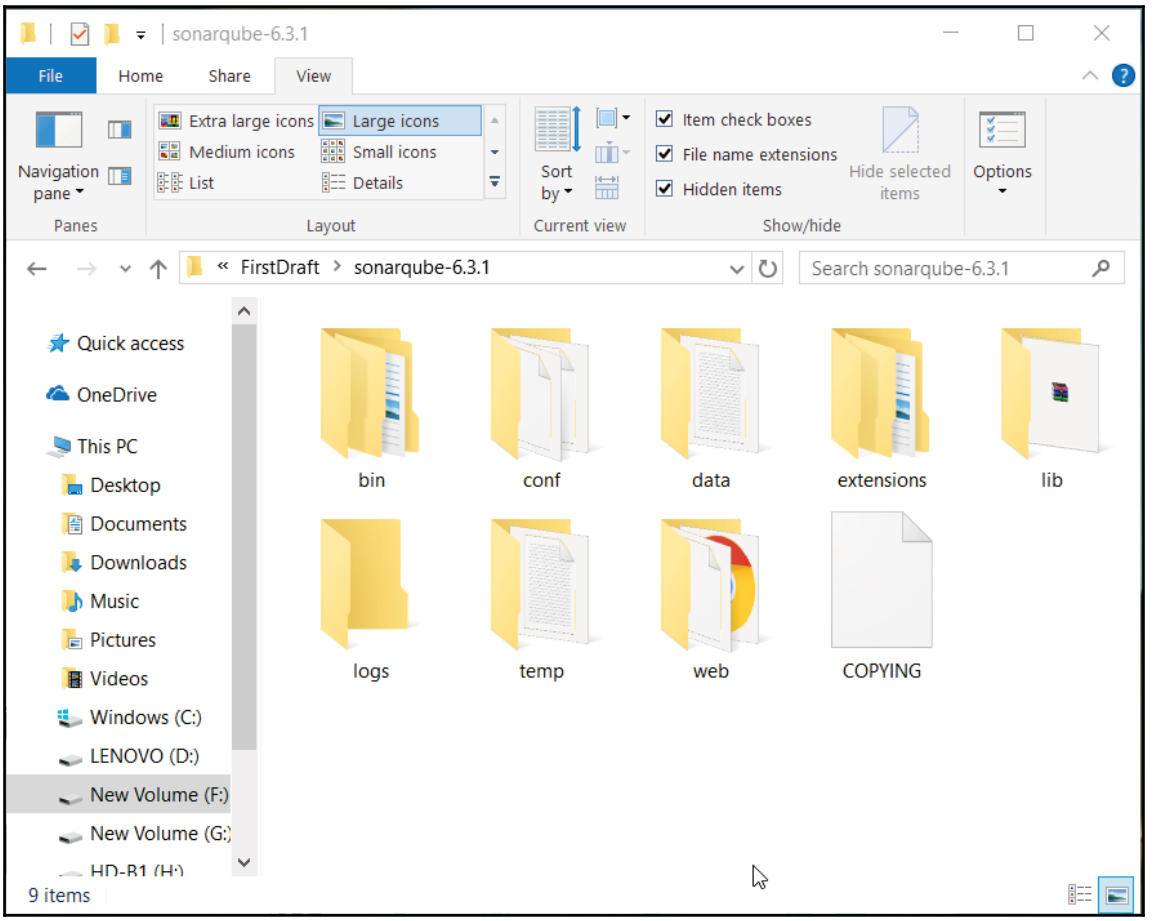


At the end of this chapter, we will know how to configure a SonarQube server with Jenkins and perform static code analysis.

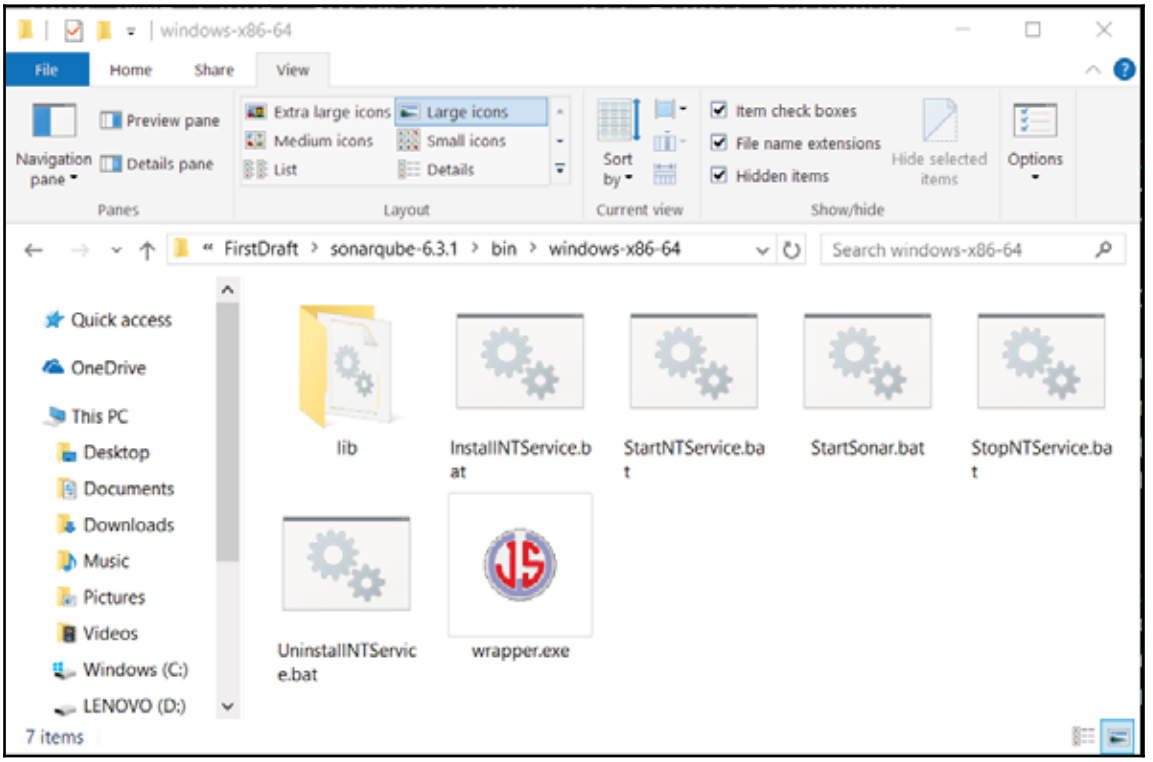
Jenkins 2.x integration with Sonar 6.3

In this chapter, we will use SonarQube 6.3 for static code analysis. Go to `https://www.sonarqube.org/downloads/` and download the latest version available:

1. Extract files:

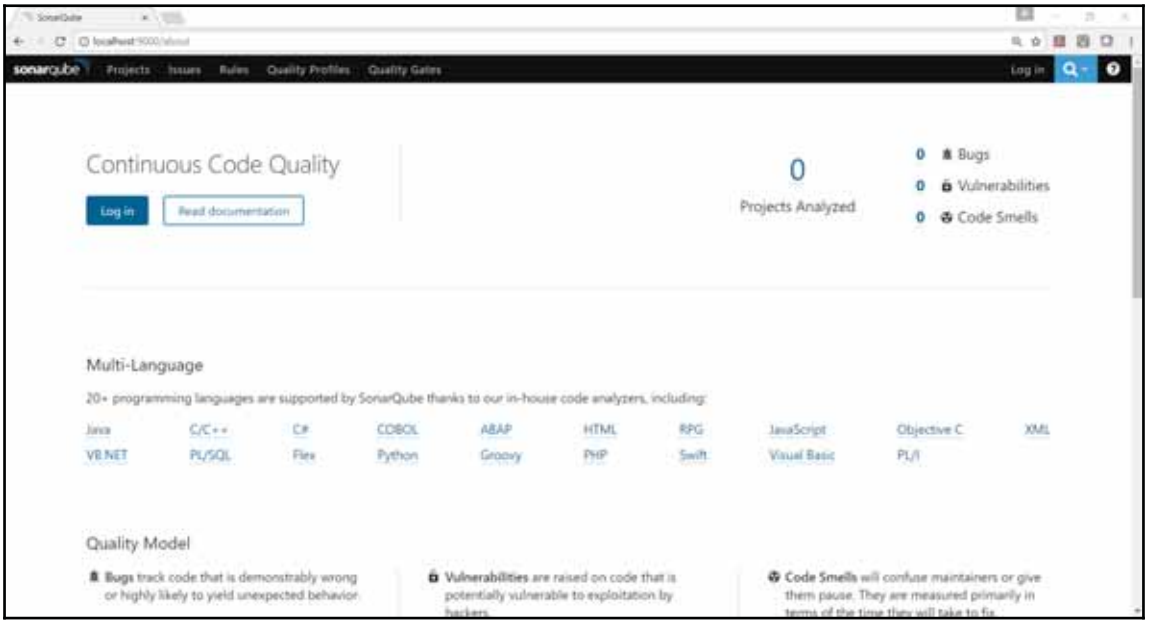


2. Go to the `bin` directory and, based on the operating system and platform of the operating system, go to a specific directory:

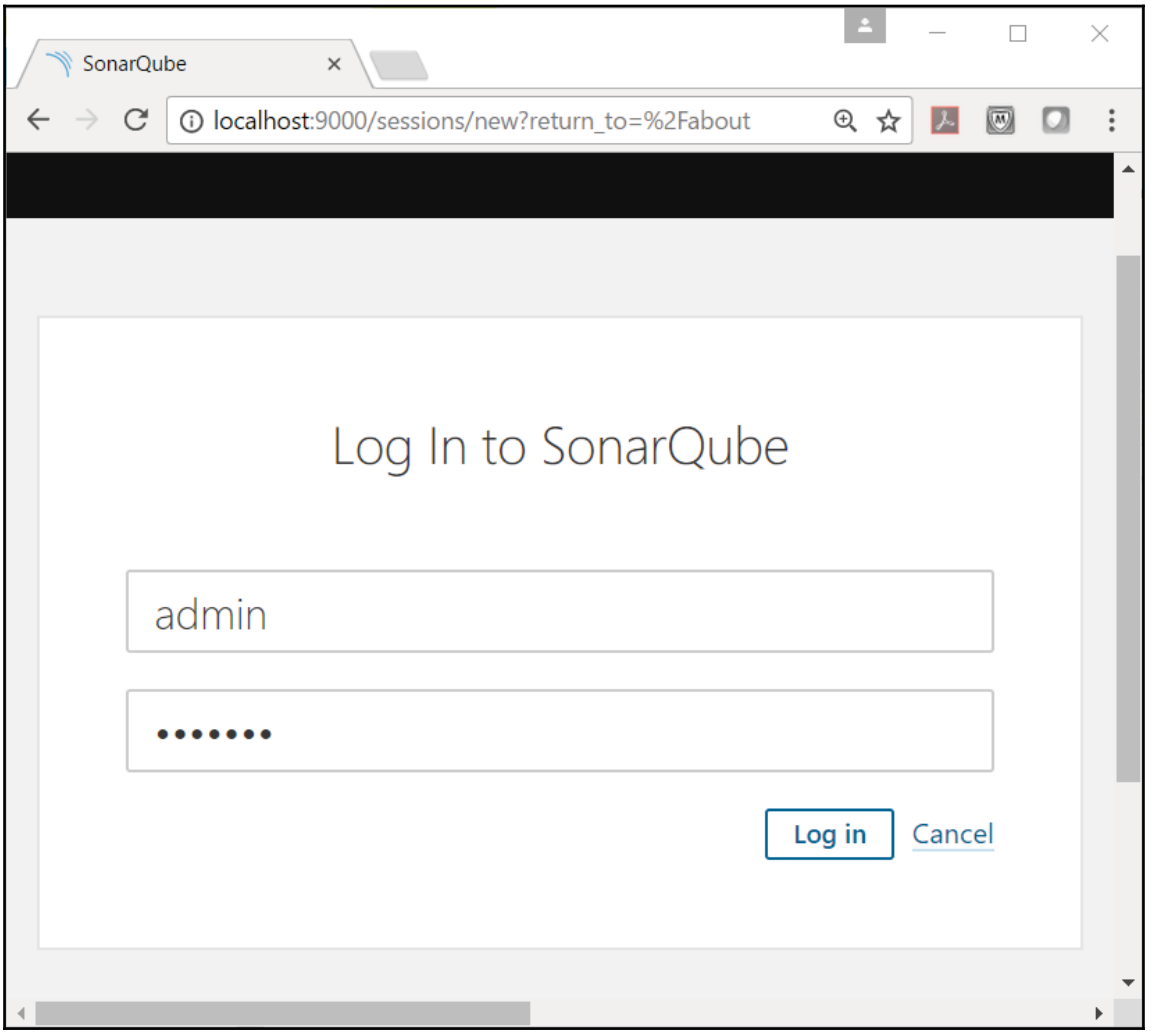


3. Execute `StartSonar.bat` in the command window. On Linux or MacOS execute the `.sh` file in the respective folder.

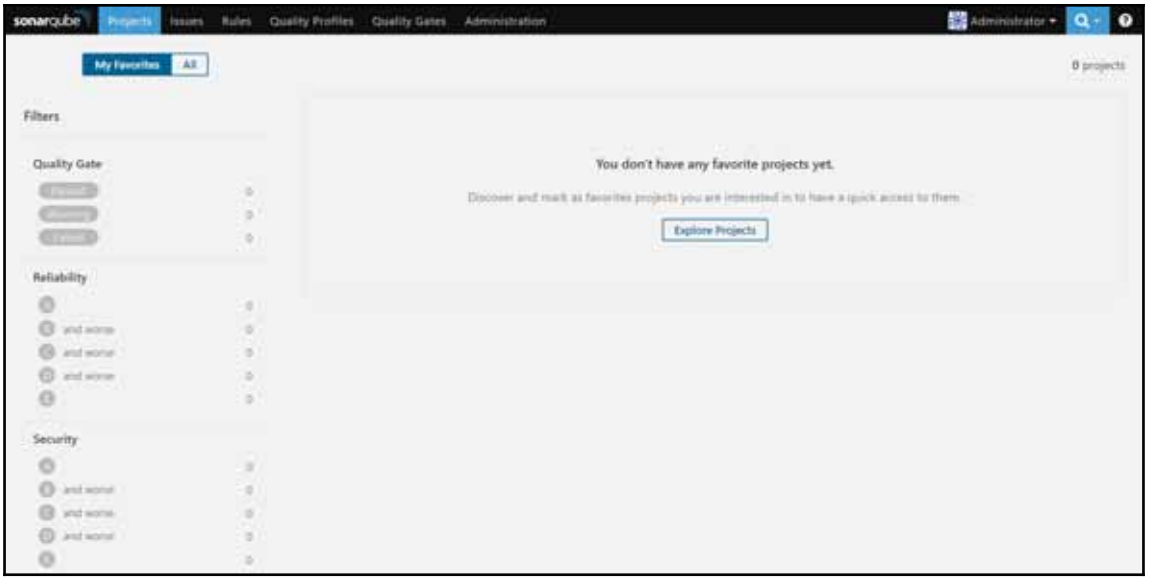
4. Once SonarQube is up and running, open `http//:localhost:9000` in a browser to visit the SonarQube dashboard:



5. Click on **Login** and give a default username and password --admin and default to-- log in as an administrator:

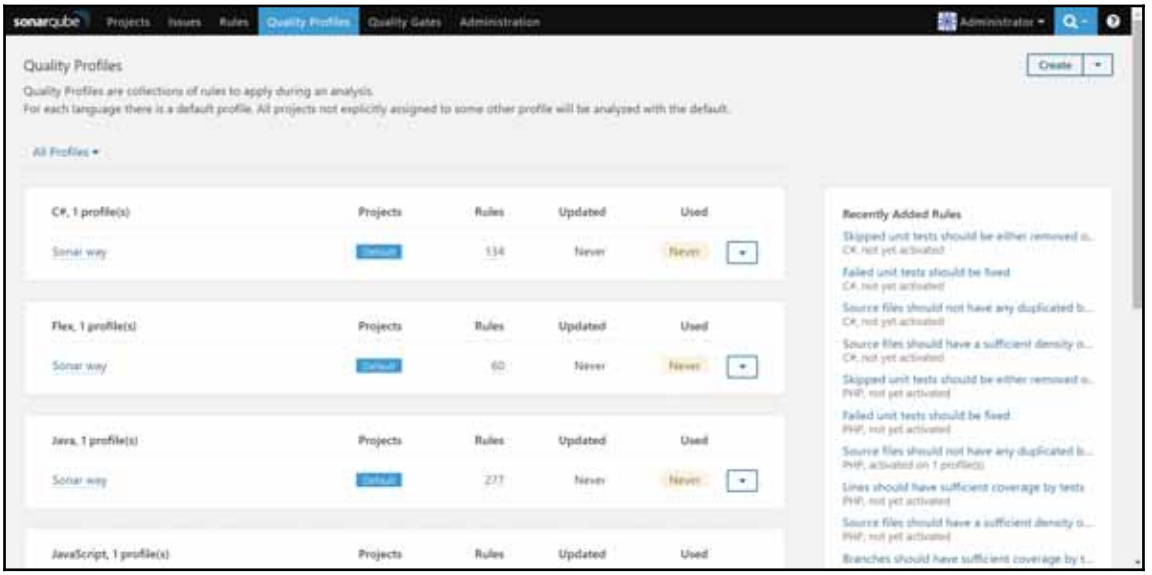


6. As of now, there is no project available in the SonarQube dashboard:

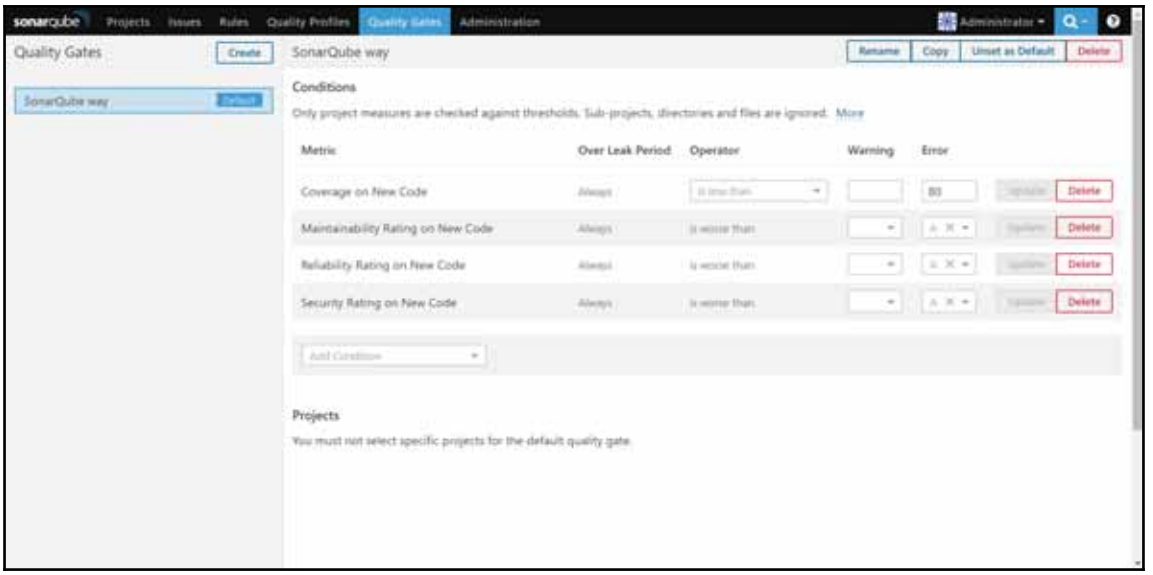


7. Click on the **Quality Profiles** tab to get details on the default quality profiles available in SonarQube.

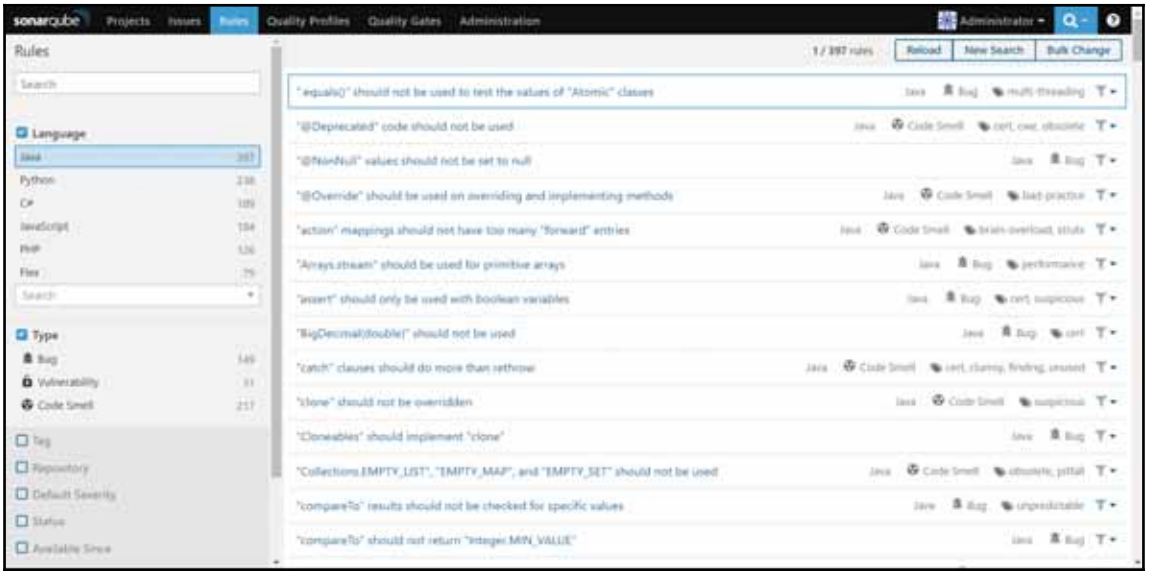
8. **Quality Profiles** are the heart of SonarQube; they are nothing but sets of rules specific to a language. If not mentioned explicitly, all the projects are analyzed with default profiles. However, it is ideal to have a profile for each project so specific rules can be set or deactivated. Each language has a default profile named **Sonar way**:



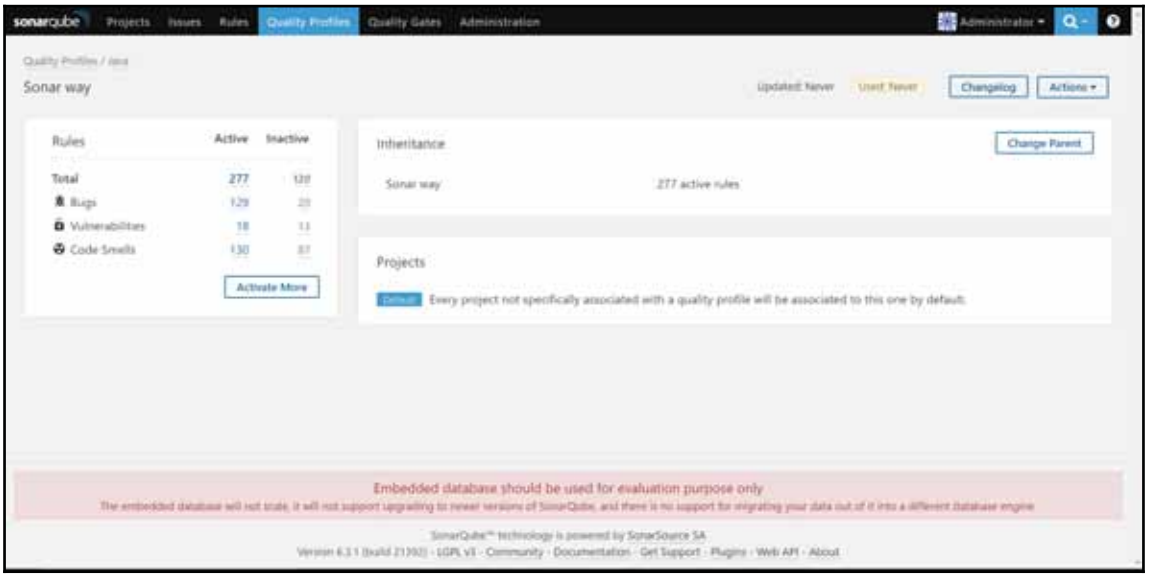
9. A Quality Gate is used to enforce policy in organization for static code analysis. The SonarQube way is the default **Quality Gate**:



10. Click on the **Rules** tab to get more details about the existing rules available in profiles:

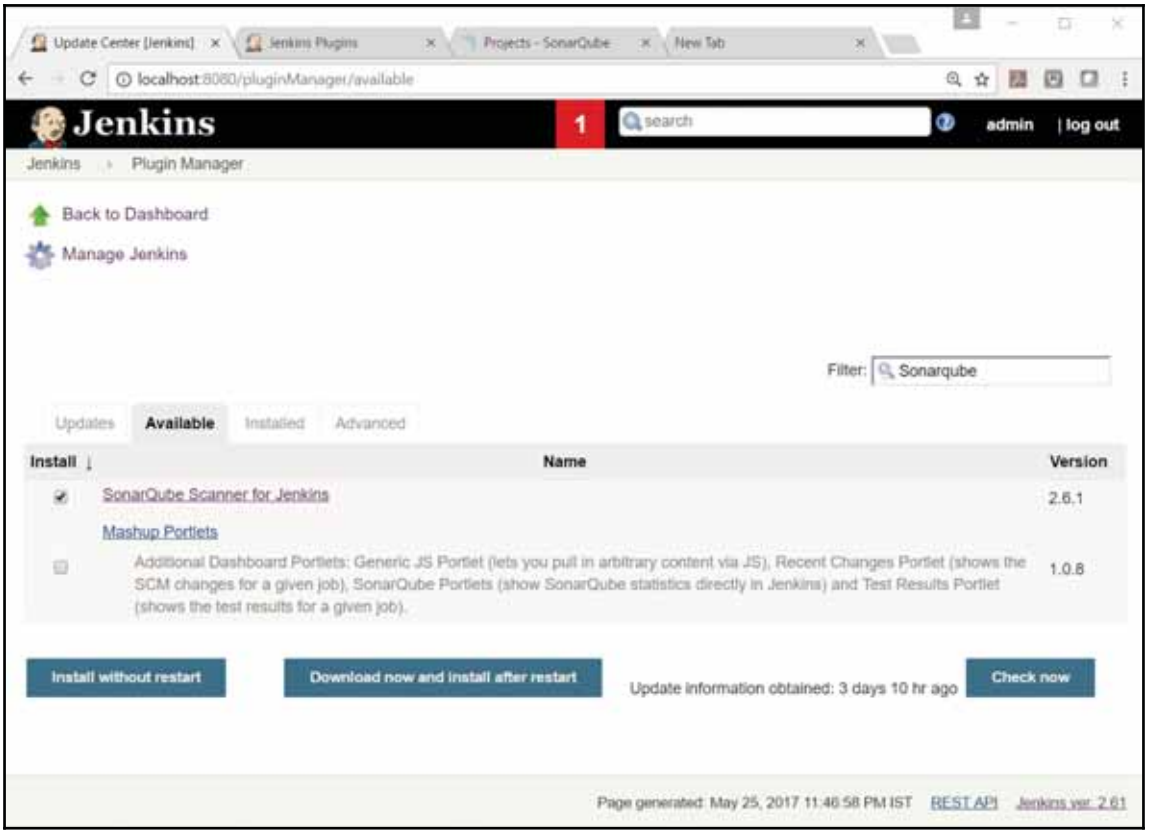


11. Go to **Quality Profiles** and select the **Sonar** way default profile. Observe total **Active** and **Inactive** rules:

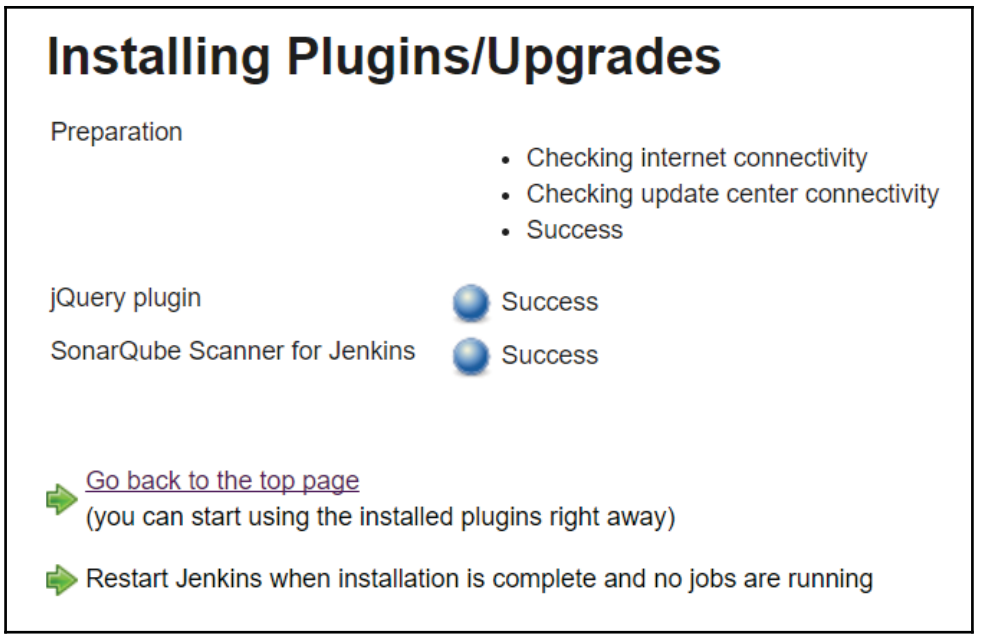


12. Go to the Jenkins dashboard and click on **Manage Jenkins**. Go to **Manage Plugins** and in the **Available** tab find the SonarQube plugin.

13. Click on **Install without restart**:

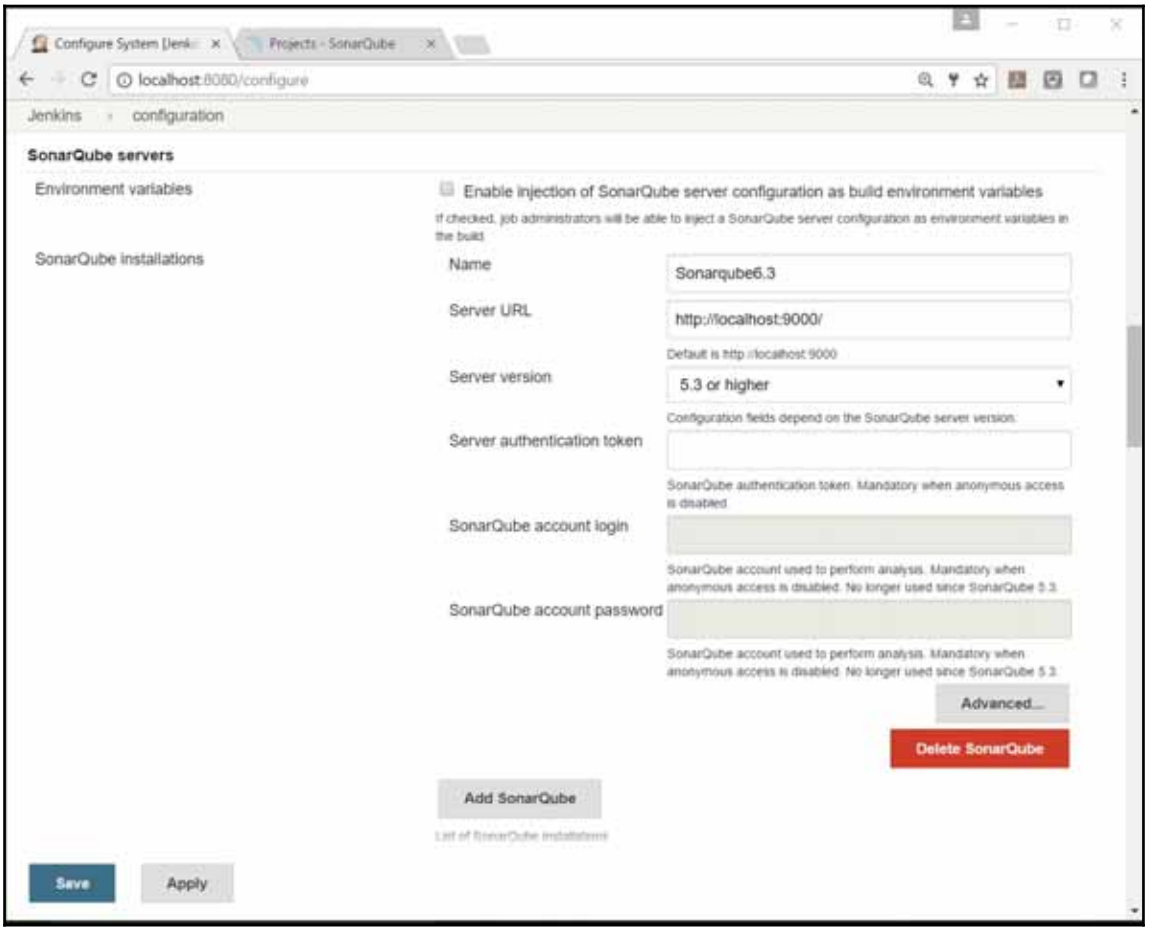


14. Verify when installation is successful:

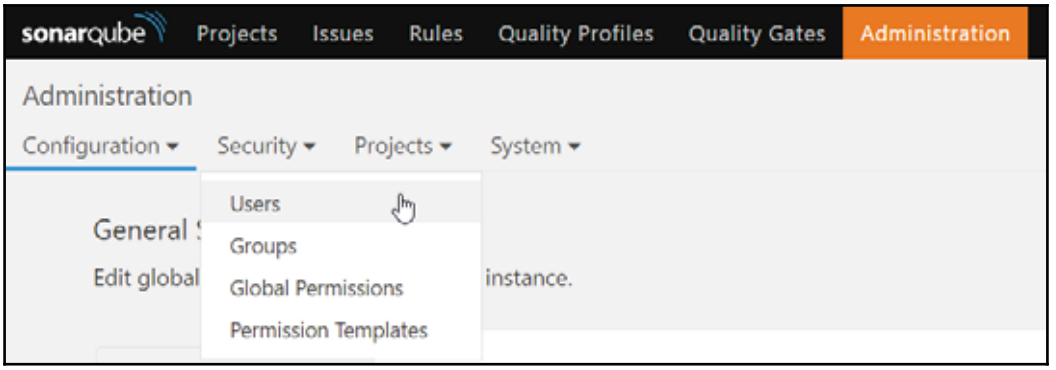


15. Go to the Jenkins dashboard and click on **Manage Jenkins**.
16. Click on **Configure system** and find the SonarQube section.
17. Click on **Add SonarQube**.

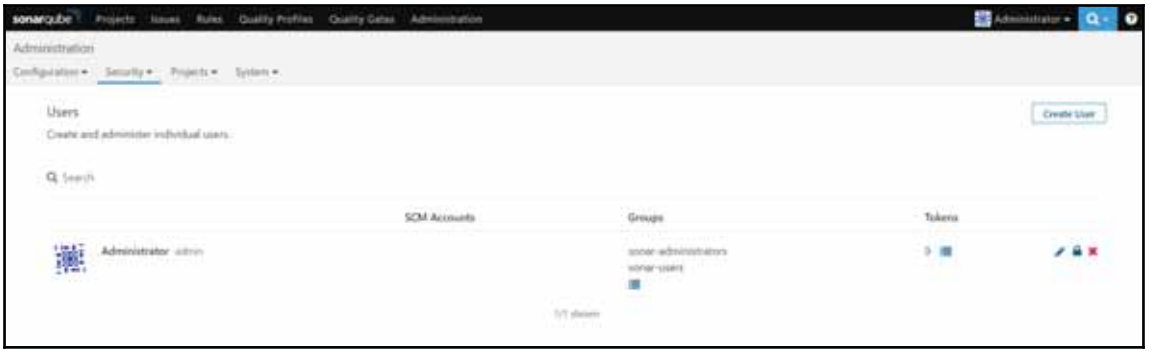
18. Provide name, URL, and version. It also asks for a **Server Authentication token**. We can get it from the SonarQube server dashboard:



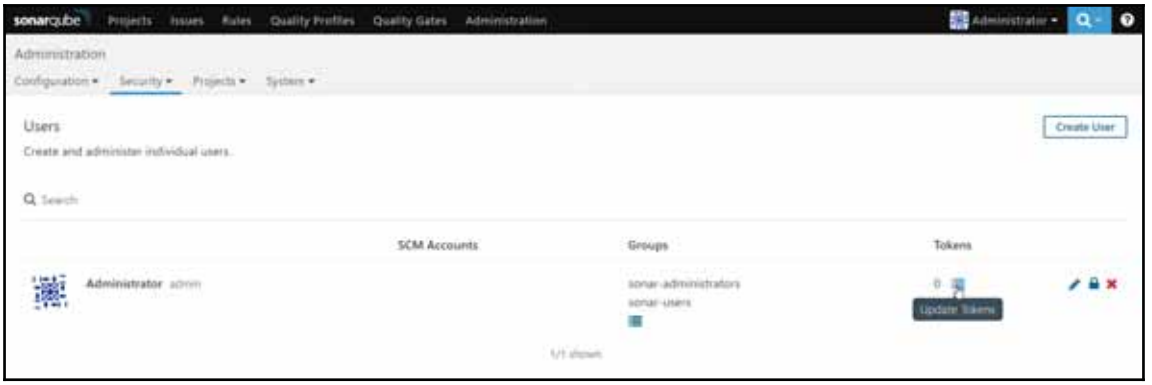
19. Click on the **Administration** tab. In the **Security** menu, click on **Users**:



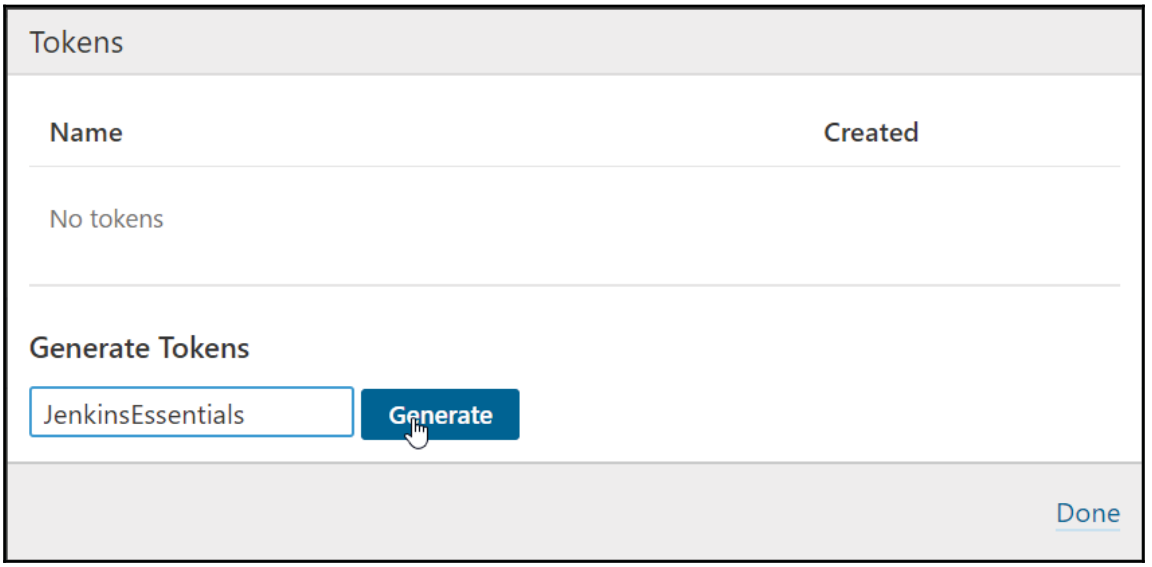
20. Observe that there is a **0** token for **Administrator**:



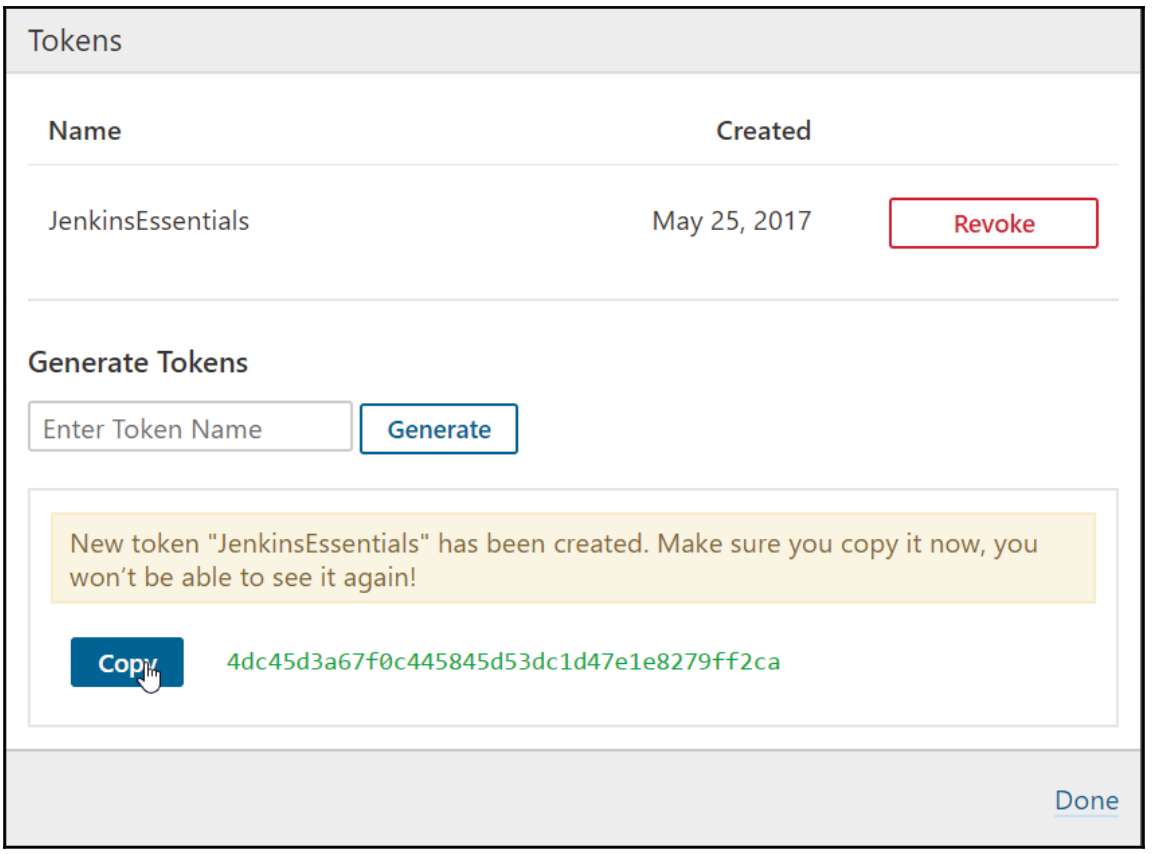
21. Click on **Tokens**:



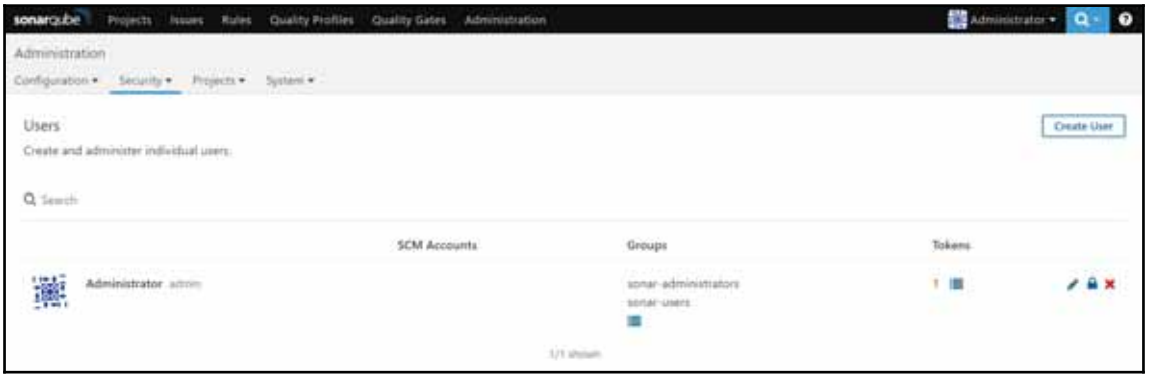
22. Give a name in the **Generate Tokens** section and click on **Generate**:



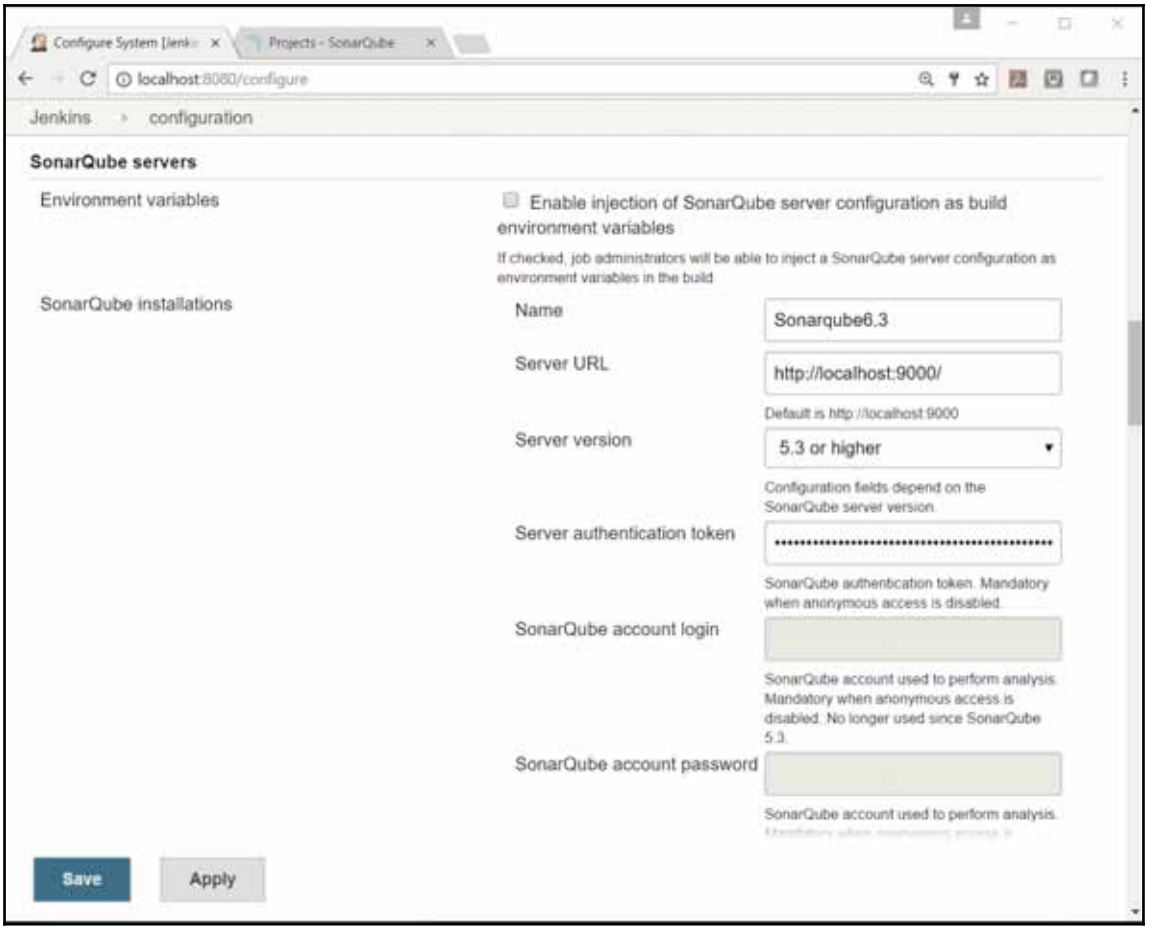
23. Copy the newly created **Token**. Click on **Done**:



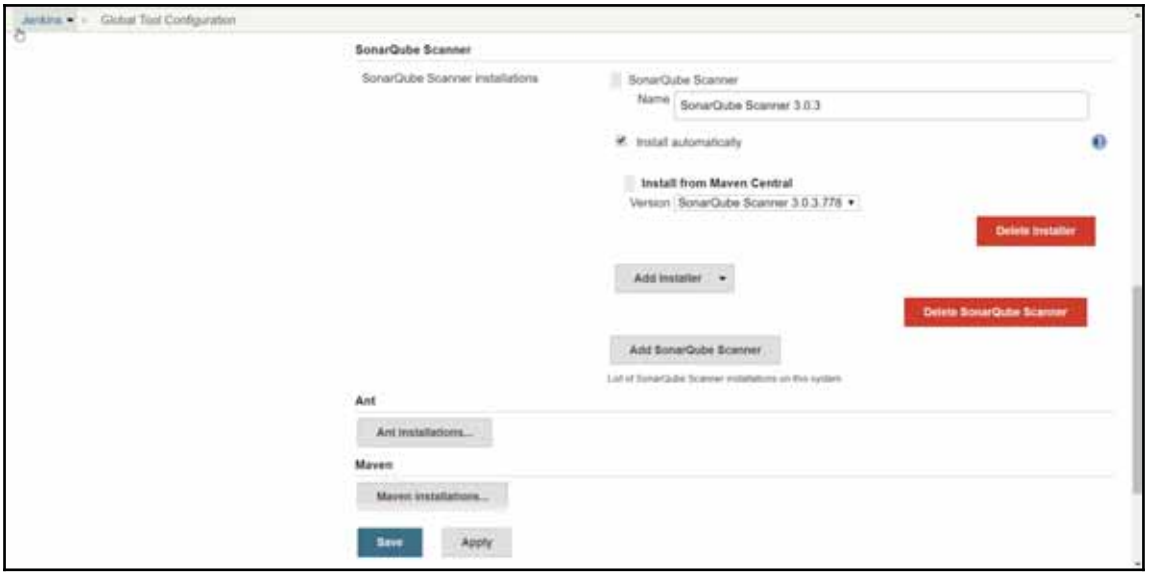
24. Verify the number of **Tokens** for the **Administrator** user:



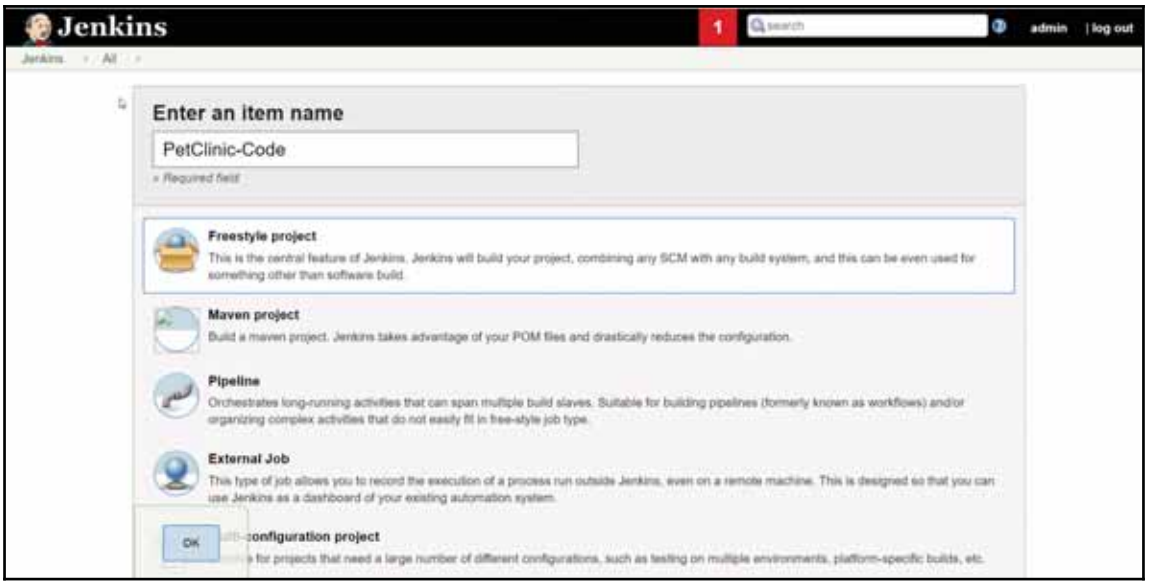
25. Paste the token value in Jenkins and **Save**:



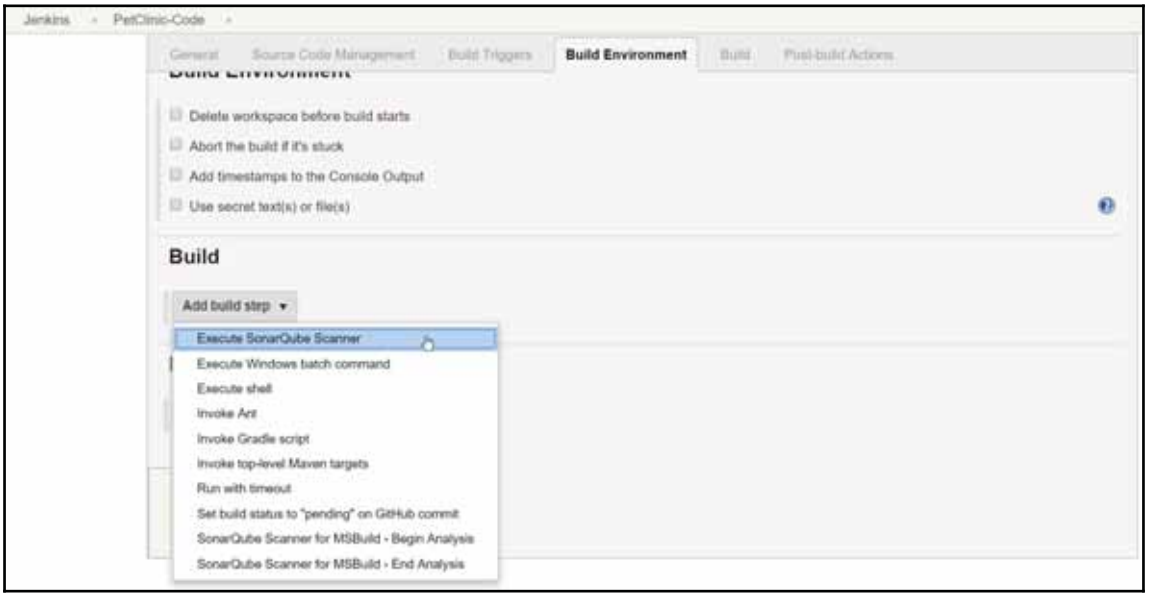
26. Go to **Global Tool Configuration** and configure **SonarQube Scanner**:



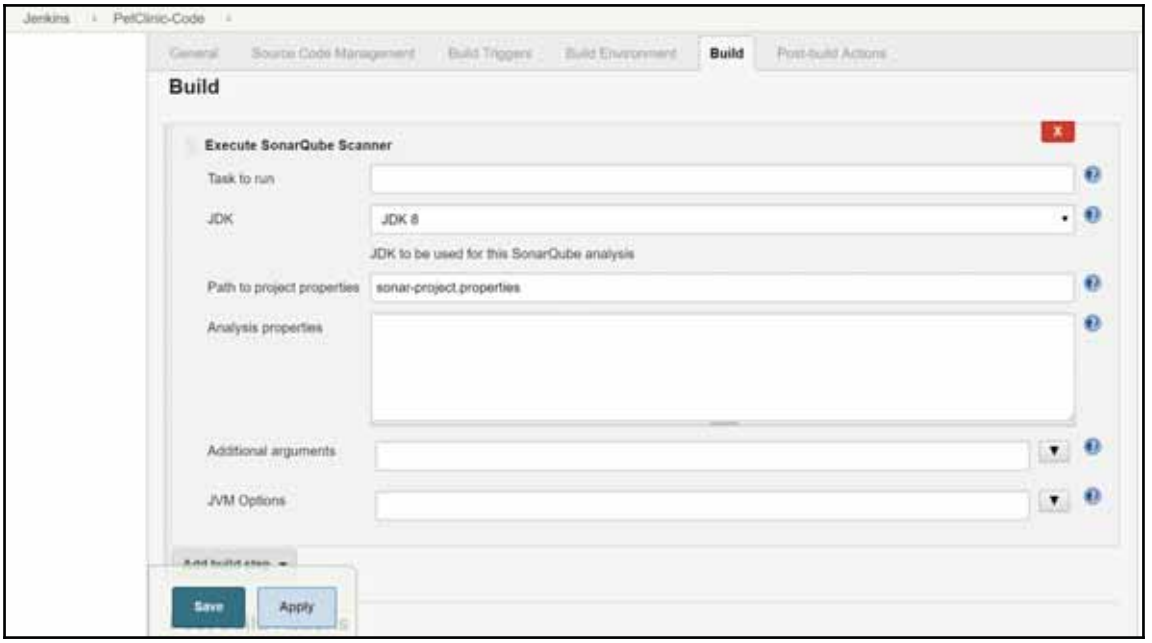
27. Go to the Jenkins dashboard and click on **New Item**.
28. Give a name, `PetClinic-Code`, and select **Freestyle project**. We are going to perform static code analysis on the sample application using SonarQube here:



29. Provide a repository URL in the **Source Code Management** section.
30. In the **Build,** section select **Execute SonarQube Scanner**:



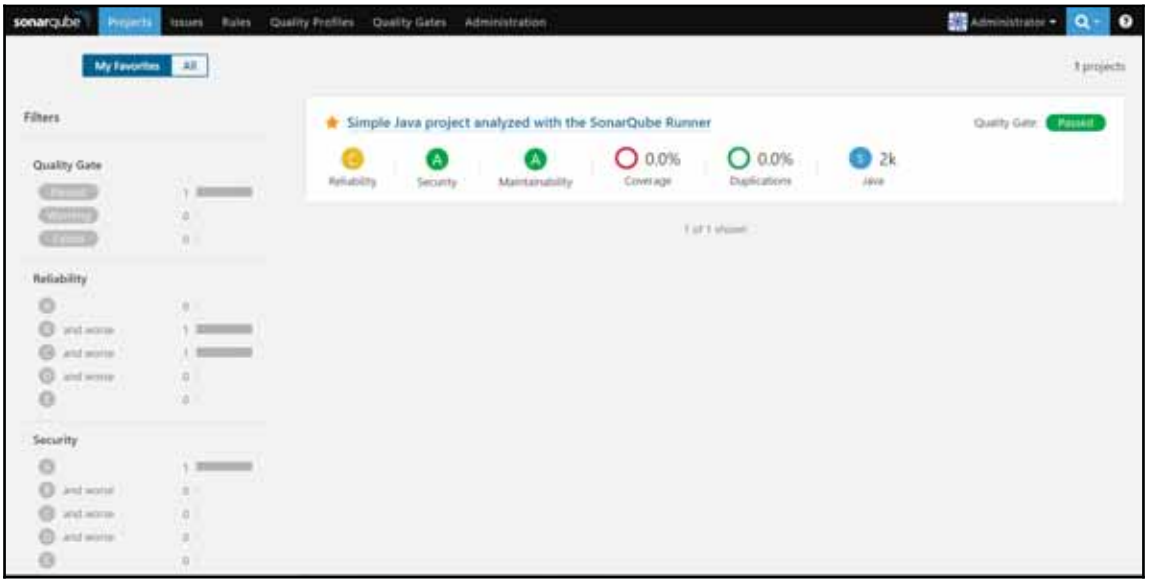
31. Select **JDK** and provide a **Path to project properties**:



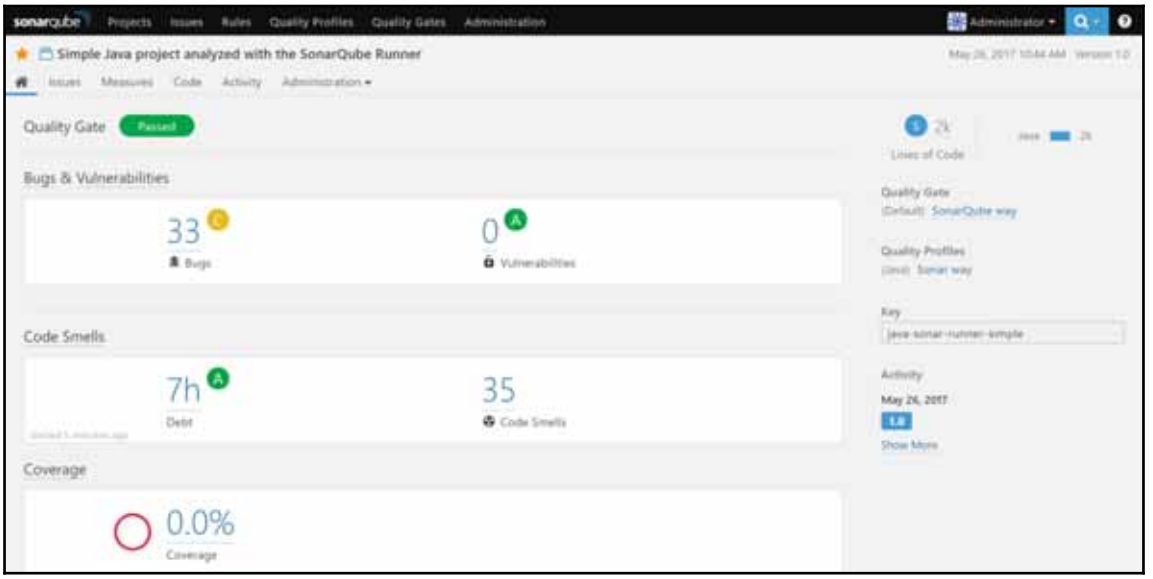
32. `sonar-project.properties` contains the following details. `sonar.source` is the main property for static code analysis. We inform SonarQube which directory needs to be analyzed. We can add same content in the analysis properties to achieve same results and not require `sonar-project.properties`:

```
# Required metadata
sonar.projectKey=java-sonar-runner-simple
sonar.projectName=Simple Java project analyzed with the SonarQube Runner
sonar.projectVersion=1.0

# Comma-separated paths to directories with sources (required)
sonar.sources=src

# Language
sonar.language=java

# Encoding of the source files
sonar.sourceEncoding=UTF-8
```

33. If language property is not mentioned, then SonarQube is intelligent enough to detect the language available in the source files. The same properties can be given in the **Analysis Properties** textbox.
34. Click on **Save** and then click on **Build Now**.
35. Once the Jenkins job is executed successfully, go to SonarQube and verify.
36. Click on the **Project**:



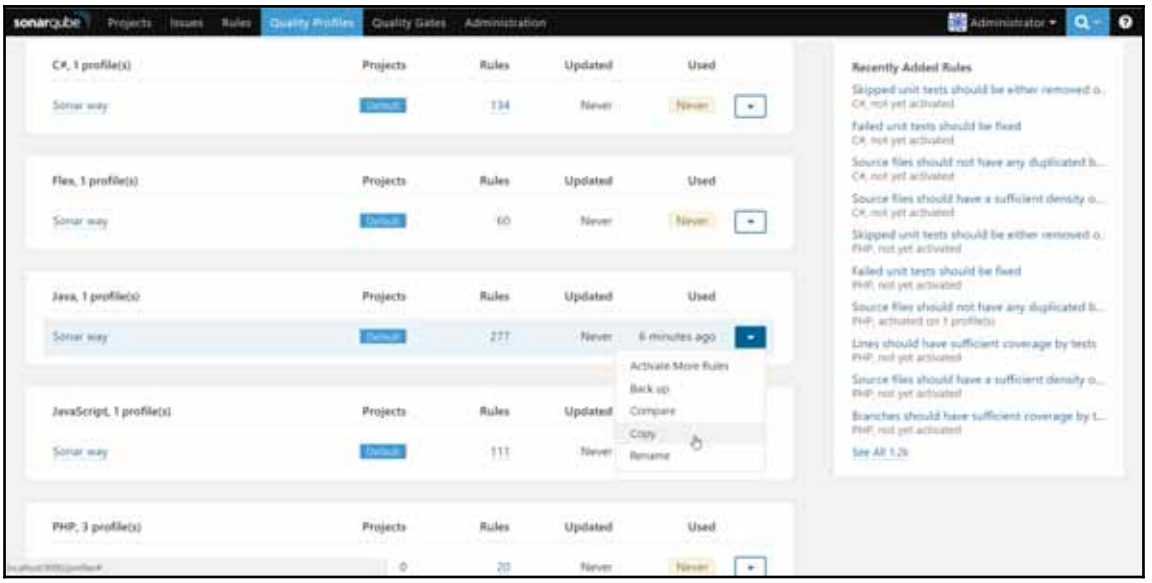
37. It gives details on **Bugs**, **Vulnerability**, and **Code Smells**. Verify each tab and content available in it as a self-exercise:



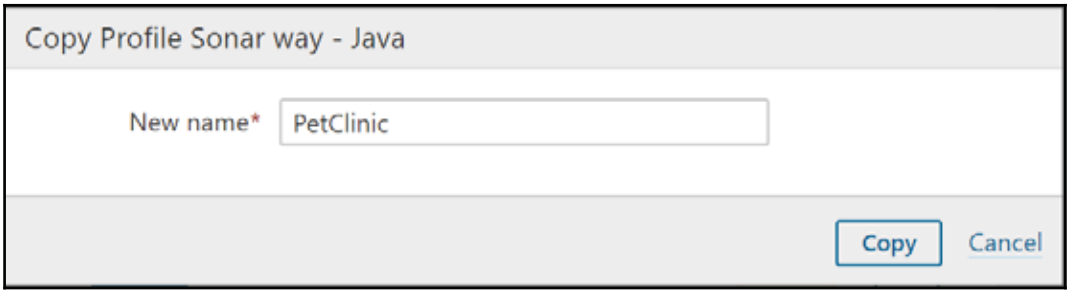
So we have successfully done static code analysis of a sample application using Jenkins.

Now let's create a new **Quality Profile** and assign the project so every time static code analysis is performed, a default profile is not used, but a custom profile is utilized:

1. Go to **Quality Profiles;** in the Java section, copy the default profile:

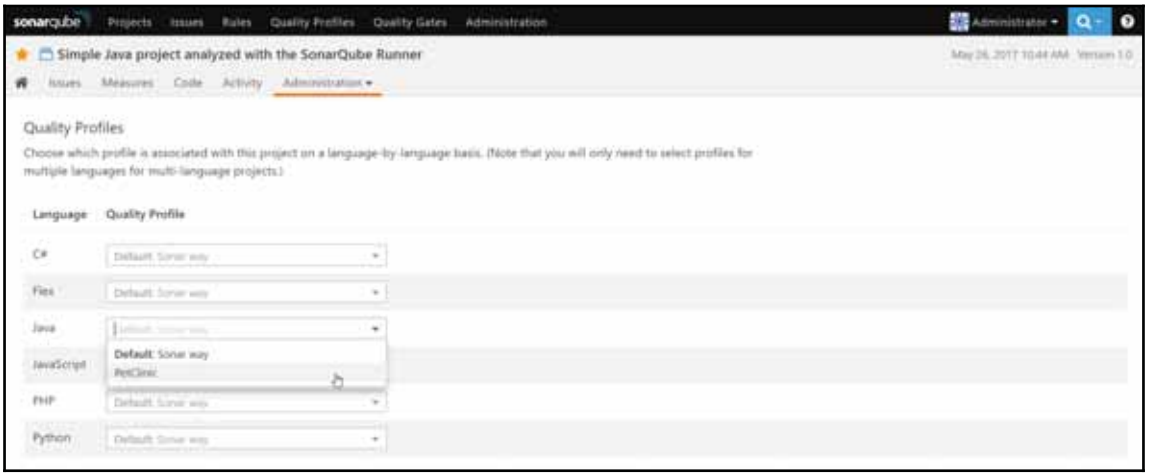


2. Give a specific name to it and click on **Copy**:



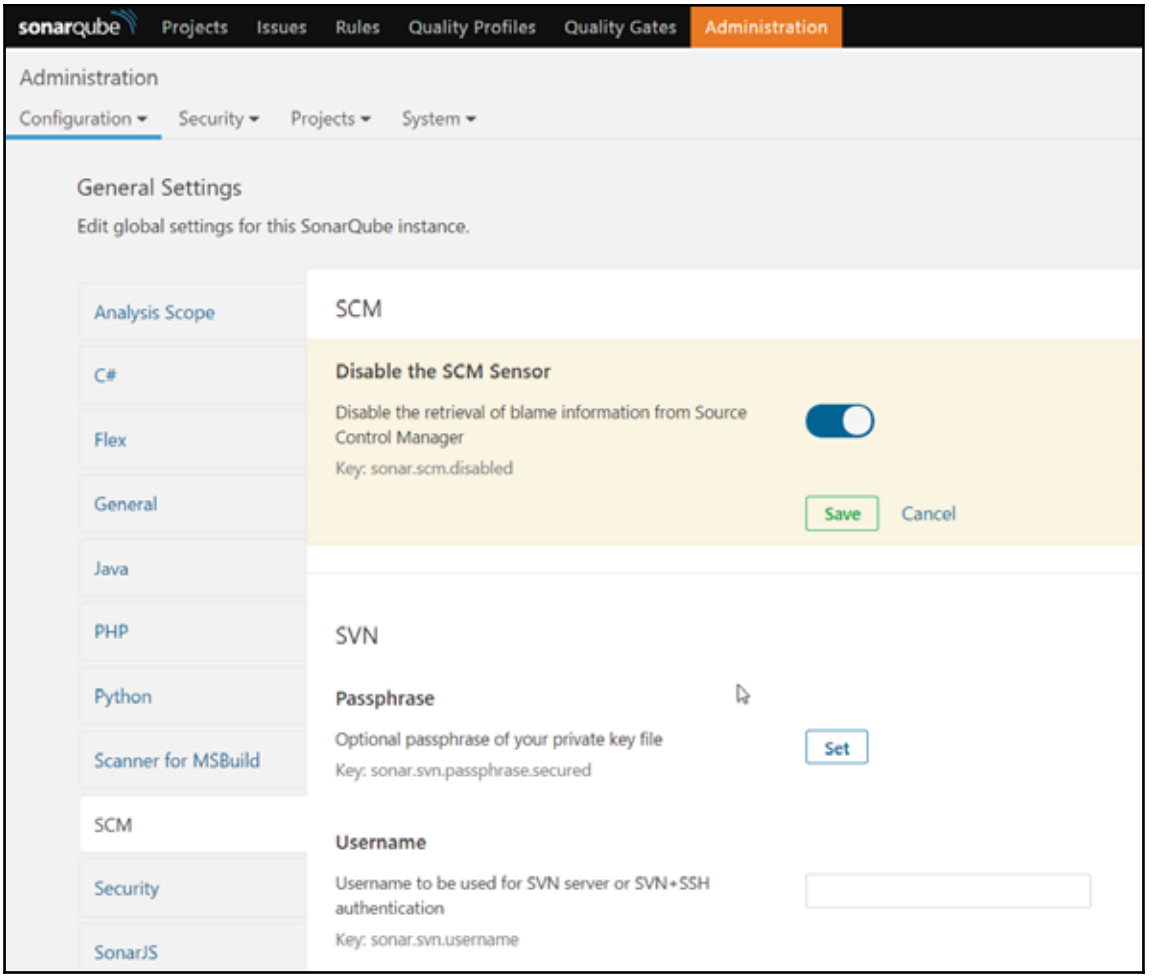
3. We can specify projects for a specific quality profile by clicking on **Change Projects** as well:

4. We can also specify **Quality Profile** by clicking a specific project. Go to **Administration,** select **Quality Profiles,** and select the custom profile created for Java:



5. Just for troubleshooting, if we come across a Jenkins job failure due to SCM blame, then we need to fix that by configuring it in SonarQube.

6. Go to SonaQube and disable the SCM sensor:

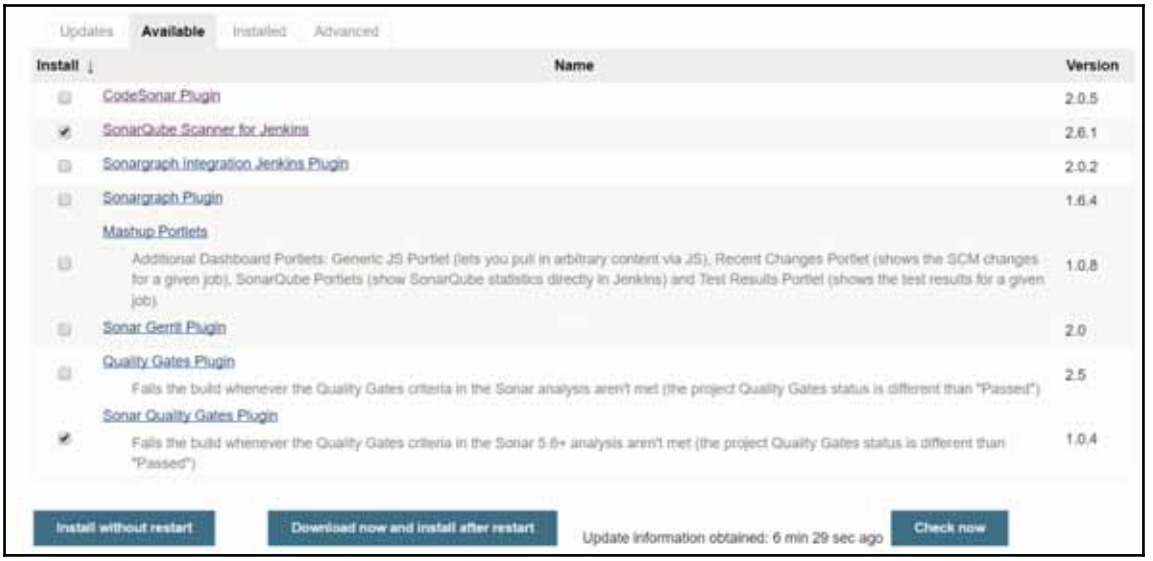


In the next section, we will cover the Quality Gate plugin.

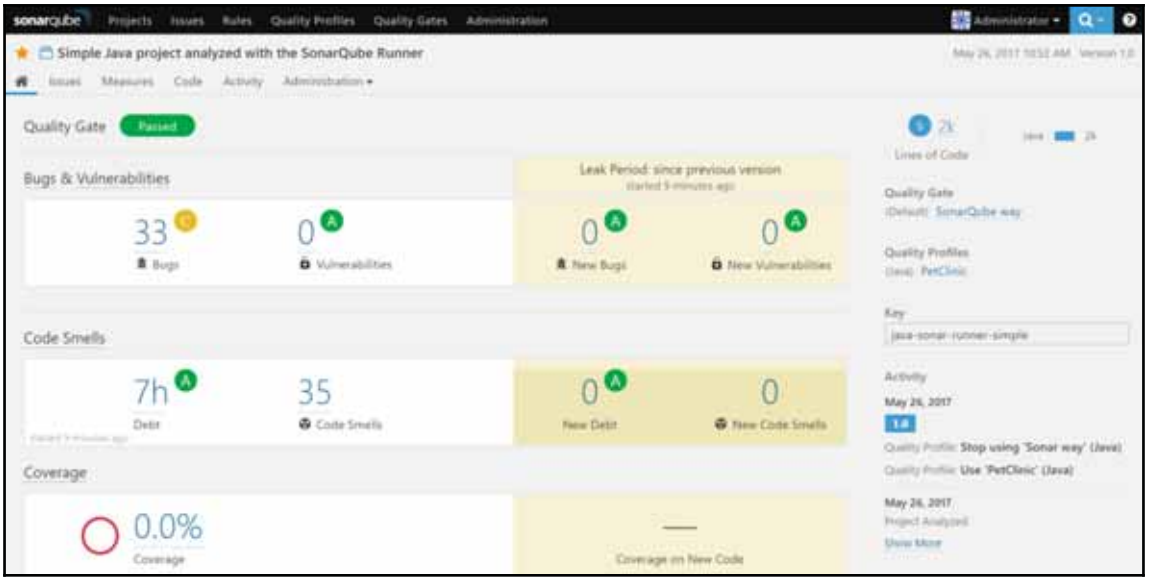
Quality Gate plugin

The `Quality Gate` plugin is useful if we want to fail the Jenkins job based on the result of Quality Gate:

1. Install the **Sonar Quality Gates Plugin** in Jenkins:

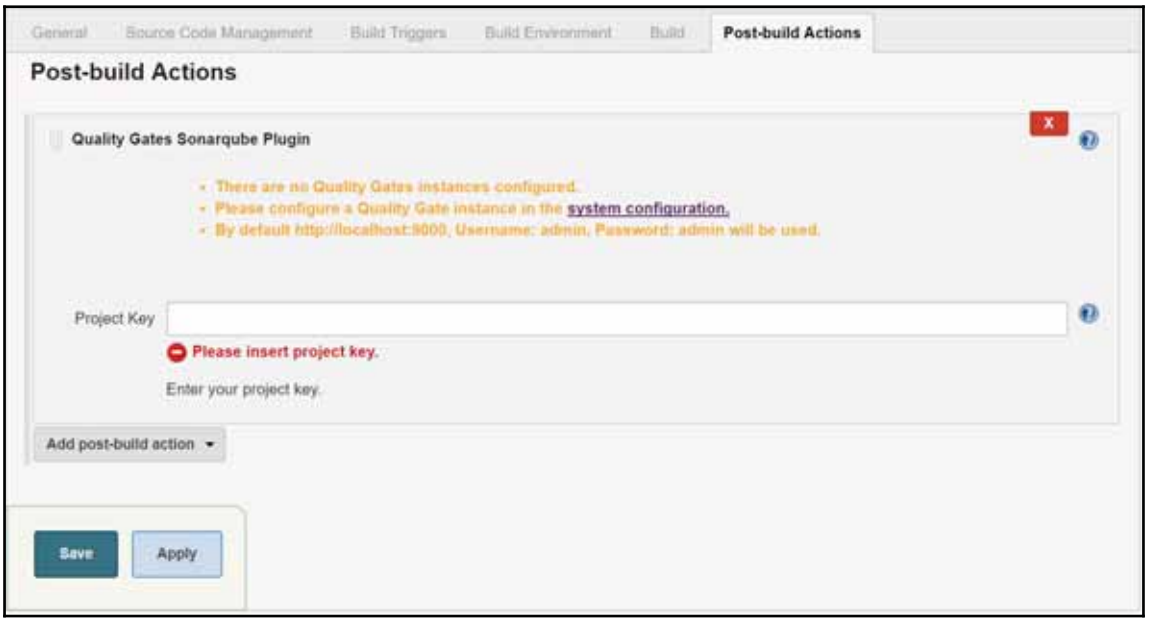


2. As of now, Quality Gate is passed for our sample application:

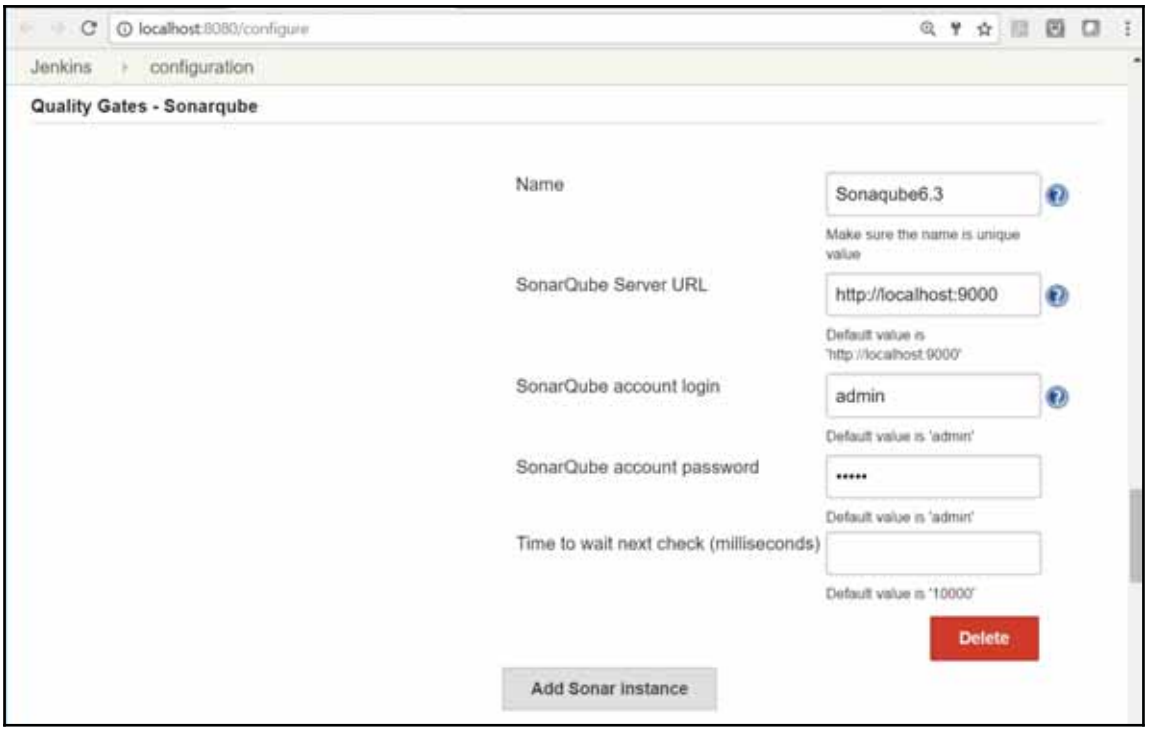


3. Go to the **Quality Gates** tab and add a condition where if issues are greater than 10, then it should give an error.

4. In the same **PetClinic-Code** build job, as a **Quality Gates SonarQube** plugin action from **Add post-build action**. It asks for the Quality Gates configuration in the Jenkins configuration:



5. Go to **Manage Jenkins,** click on **Configure system,** and configure Sonar instance for **Quality Gates**:



6. We already have `sonar-project.properties` in the application. Note the project key:

```
# Required metadata
sonar.projectKey=java-sonar-runner-simple
sonar.projectName=Simple Java project analyzed with the SonarQube Runner
sonar.projectVersion=1.0

# Comma-separated paths to directories with sources (required)
sonar.sources=src

# Language
sonar.language=java

# Encoding of the source files
sonar.sourceEncoding=UTF-8
```

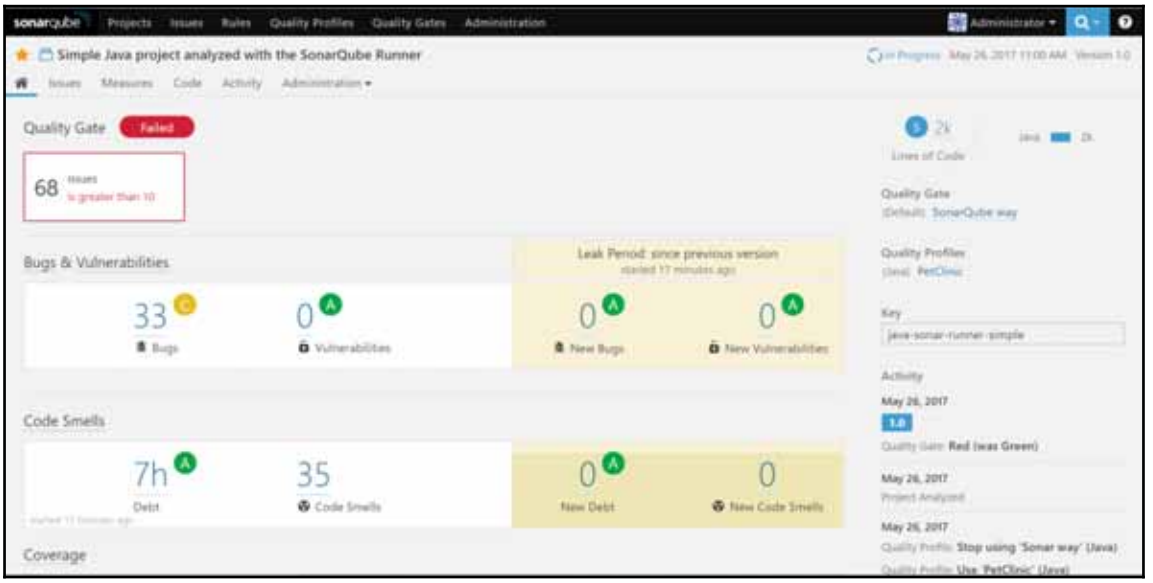
7. In the Jenkins job, enter the same **Project Key** and click **Save**:

8. Click on **Build now** to execute a Jenkins build job.

9. The Jenkins job has failed. Go through the console output and the reason will be Quality Gate failure:

```
INFO: ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashboard/index/java-sonar-runner-simple
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted
analysis report
INFO: More about the report processing at http://localhost:9000/api/ce/task?id=AVxDQYIS-pm0HP4V18aO
INFO: Task total time: 43.474 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
INFO: ------------------------------------------------------------------------
INFO: Total time: 1:01.431s
INFO: Final Memory: 46M/134M
INFO: ------------------------------------------------------------------------
Has build IN_PROGRESS with id: AVxDQYIS-pm0HP4V18aO - waiting 10000 to execute next check.
Has build IN_PROGRESS with id: AVxDQYIS-pm0HP4V18aO - waiting 10000 to execute next check.
Has build IN_PROGRESS with id: AVxDQYIS-pm0HP4V18aO - waiting 10000 to execute next check.
ERROR: Build step failed with exception
org.quality.gates.sonar.api.MaxExecutionTimeException: Max time to wait sonar job!
        at
org.quality.gates.sonar.api.QualityGatesProvider.getAPIResultsForQualityGates(QualityGatesProvider.java:82)
        at org.quality.gates.jenkins.plugin.BuildDecision.getStatus(BuildDecision.java:22)
        at org.quality.gates.jenkins.plugin.QGPublisher.perform(QGPublisher.java:84)
        at hudson.tasks.BuildStepMonitor$1.perform(BuildStepMonitor.java:20)
        at hudson.model.AbstractBuild$AbstractBuildExecution.perform(AbstractBuild.java:735)
        at hudson.model.AbstractBuild$AbstractBuildExecution.performAllBuildSteps(AbstractBuild.java:676)
        at hudson.model.Build$BuildExecution.post2(Build.java:186)
        at hudson.model.AbstractBuild$AbstractBuildExecution.post(AbstractBuild.java:621)
        at hudson.model.Run.execute(Run.java:1760)
        at hudson.model.FreeStyleBuild.run(FreeStyleBuild.java:43)
        at hudson.model.ResourceController.execute(ResourceController.java:97)
        at hudson.model.Executor.run(Executor.java:405)
Build step 'Quality Gates Sonarqube Plugin' marked build as failure
Finished: FAILURE
```

10. Go to the SonarQube dashboard and verify the reason for **Failure** and **Quality Profile** too:



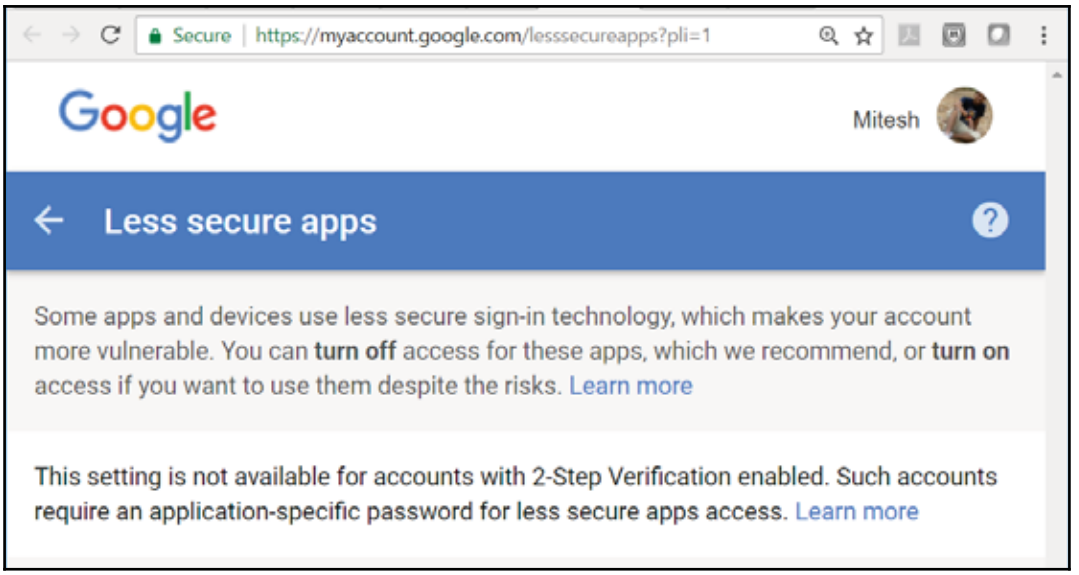
This is how static code analysis can be configured using Jenkins and SonarQube integration.

Email notifications on build status

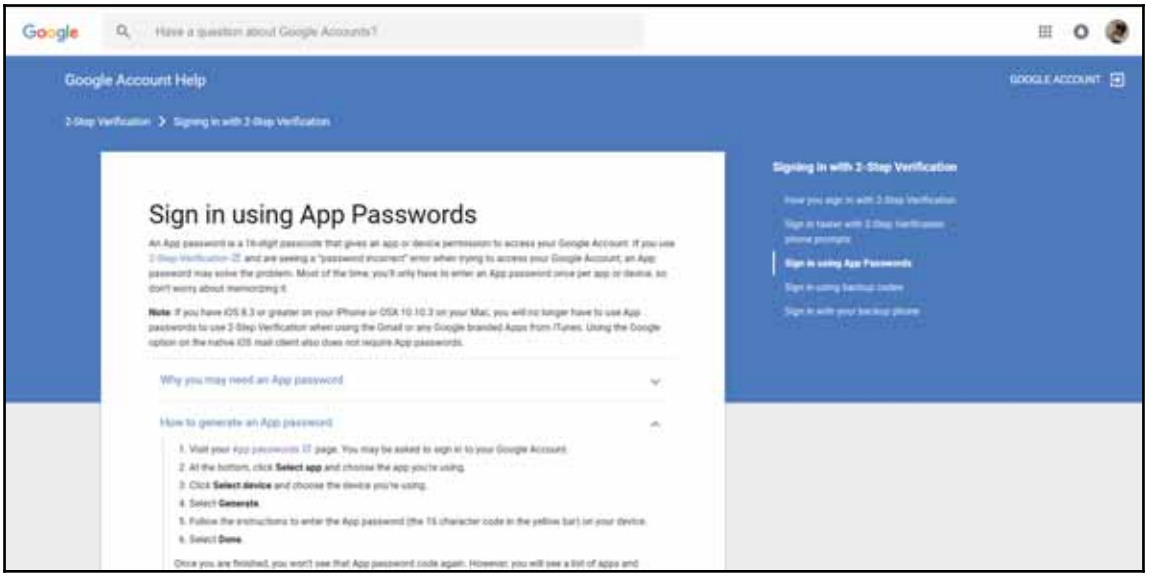
What if we want to send a notification of the build status from Jenkins? We can configure mail notifications in such scenarios.

If a Gmail account is configured with two-factor authentication, then the following process can help to setup notification systems:

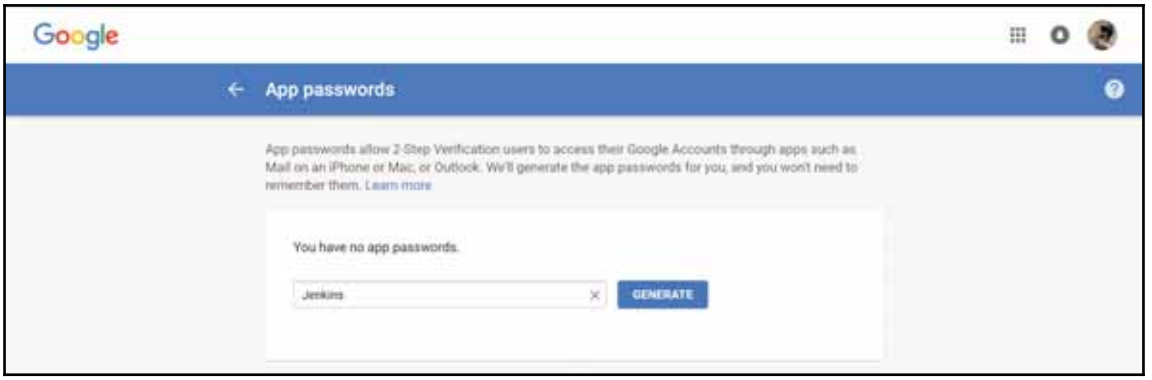
1. Go to **Less secure apps** in the Google account.
2. Click on **Learn More** in the second paragraph where two-factor authentication is mentioned:



3. Click on the **App Password** page:



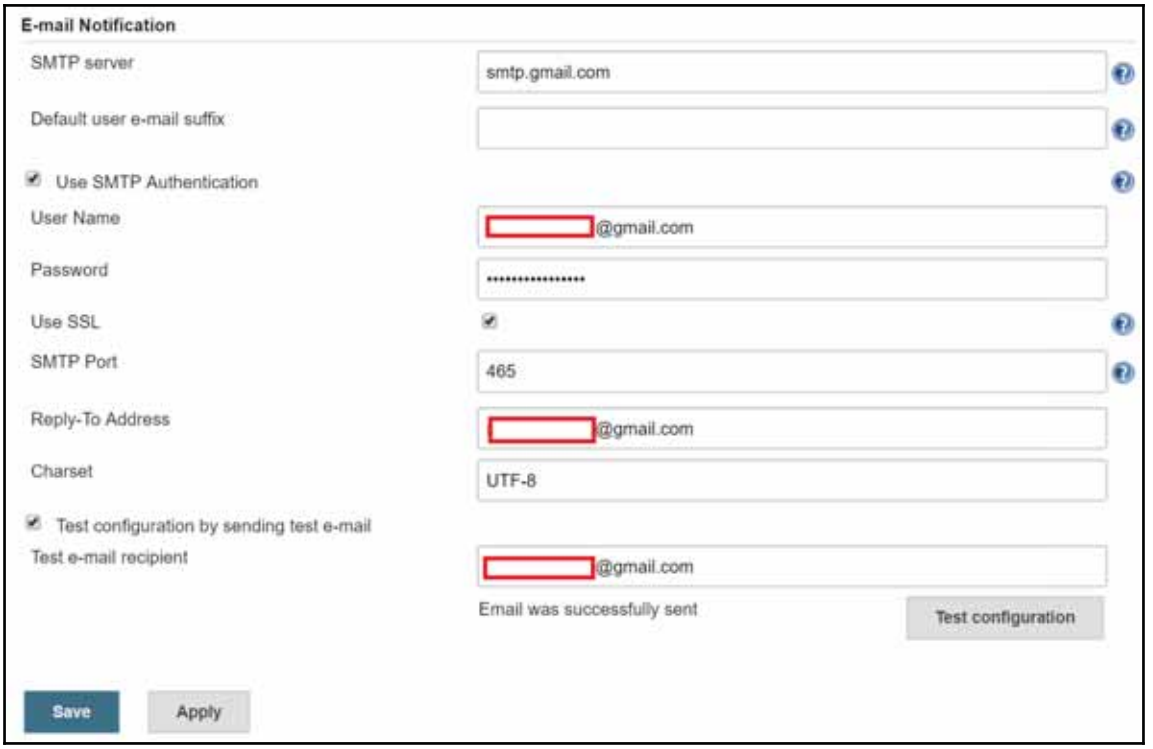
4. Provide a name and click on **Generate**:



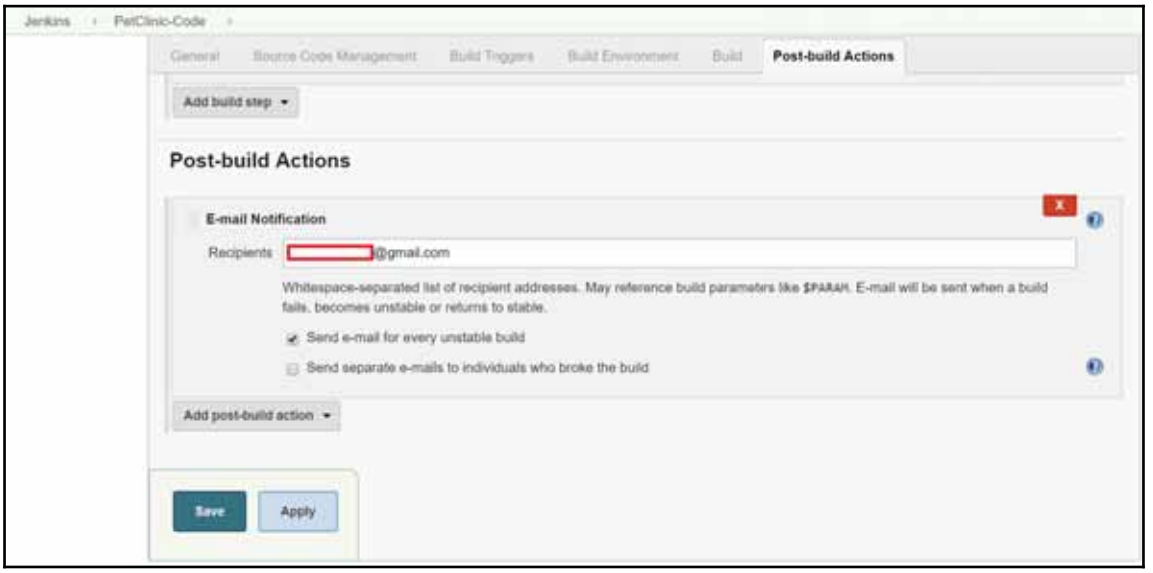
5. Copy the password:

6. Go to **Manage Jenkins** and click on **Configure system**.
7. Go to the **Email notification** section.

8. Provide all required details and in the password field, copy the recently generated password for the Jenkins app.
9. Click on **Test configuration**:



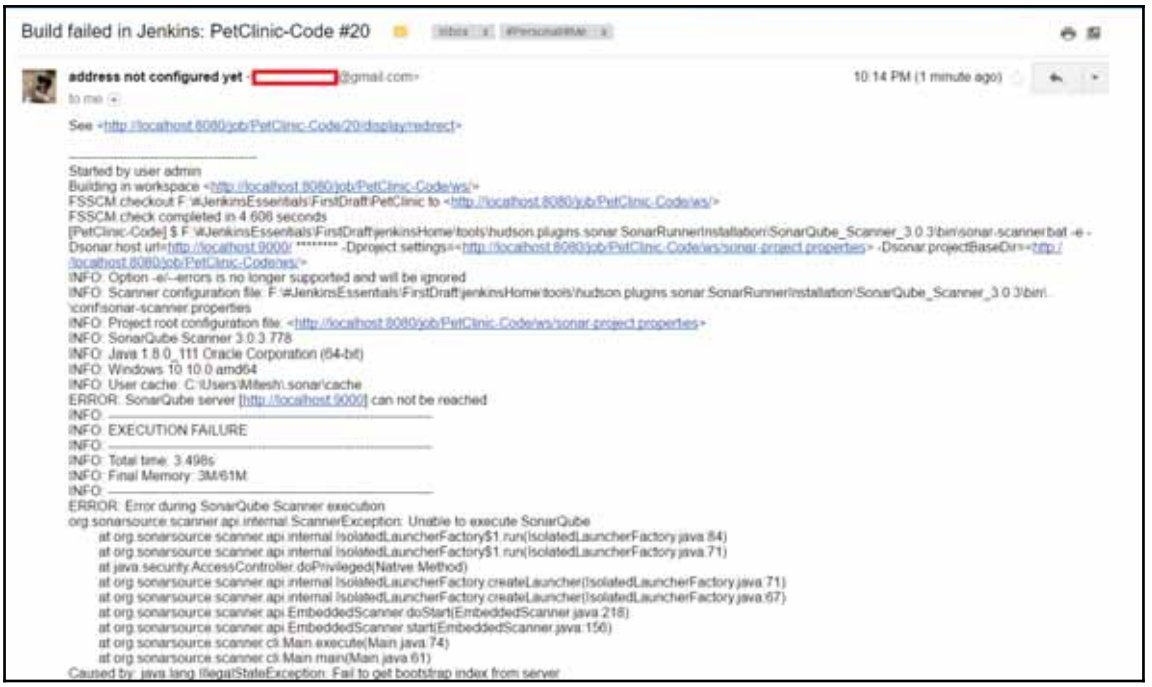
10. Go to the **PetClinic-Code** job and select **E-mail** notification from the **Post build action**:



11. Execute the Jenkins job and if it fails, go to the console output and verify the output:

```
ERROR:
ERROR: Re-run SonarQube Scanner using the -X switch to enable full debug logging.
ERROR: SonarQube scanner exited with non-zero code: 1
Sending e-mails to: [redacted]@gmail.com
Finished: FAILURE
```

12. Go to your inbox and verify the mail sent by Jenkins:



Done!

Summary

Here we are again at the section of a chapter that gives us a sense of achievement about knowing something more. In this chapter, we have covered how to configure SonarQube for static code analysis using Jenkins. We have also created a custom profile in SonarQube.

We used the Quality Gate plugin to reflect the state of quality gate of SonarQube in Jenkins.

In the last section, we configured email notifications for an unstable build where Gmail accounts use two-factor authentication.

In the next chapter, we will cover how to perform Continuous Integration using Jenkins.

4

Continuous Integration with Jenkins

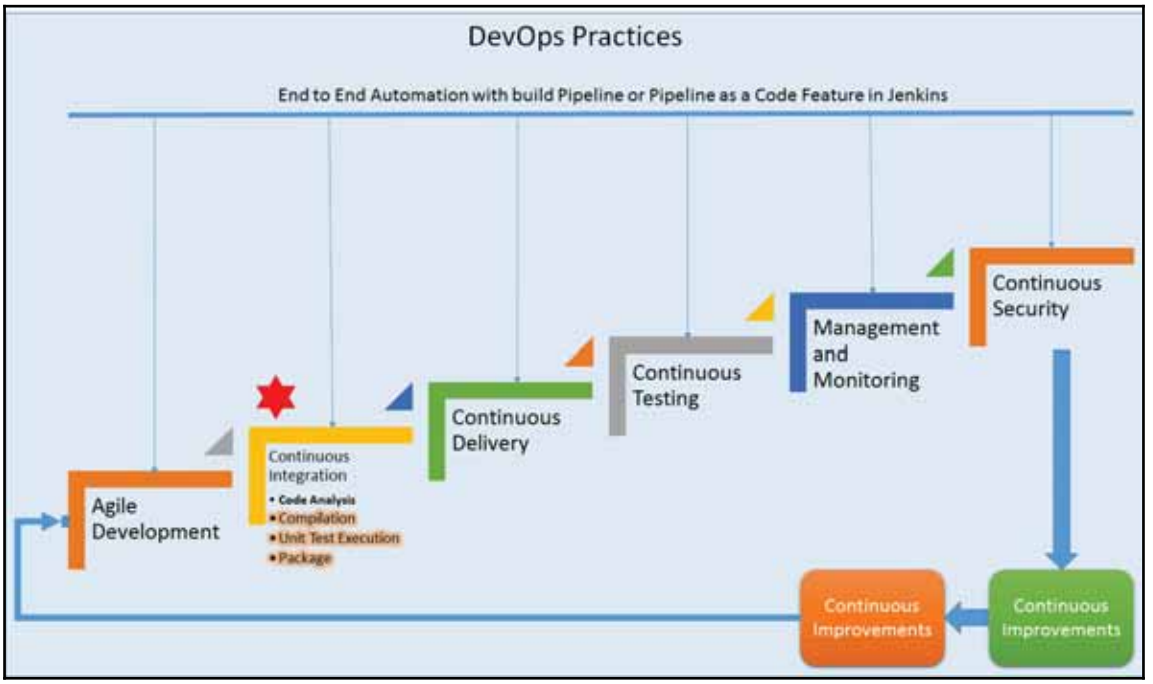
Continuous Integration is one of the most important DevOps practices and serves as a base to implement DevOps culture in any organization. It is all about committing code into shared repositories such as Git or SVN multiple times, based on feature completion or bug fixes, and then verifying it with static code analysis using SonarQube, automated builds (using Ant, Maven, or Gradle), executing unit test cases, and creating a package.

There are many tools that can be utilized for Continuous Integration. Jenkins is one of the most popular open source tools that can be utilized for multiple programming languages in which applications can be built. VSTS and Atlassian Bamboo are some other tools or services that can be utilized for Continuous Integration.

This chapter describes in detail how to create and configure build jobs for Java, and how to run build jobs and unit test cases using Ant and Maven build tools. It covers all aspects of running a build to create a distribution file or war file for deployment. We will focus on the following topics in this chapter :

- The Dashboard View plugin
- Creating and configuring a build job for a Java application with Ant
- Creating and configuring a build job for a Java application with Maven
- Build execution with test cases

In this chapter, we will cover the main parts of Continuous Integration practices as a part of our DevOps journey.



At the end of this chapter, we will know how to configure Ant, and Java-based projects in Jenkins so we can compile source files, execute unit test cases, and create a package file.

Dashboard View plugin

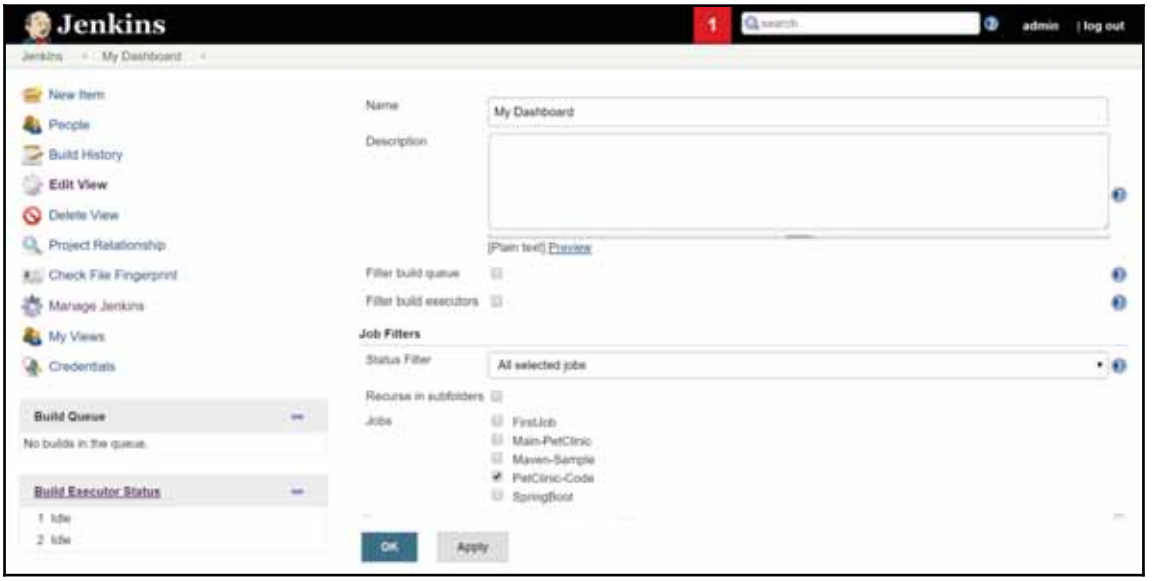
Before creating and configuring build jobs for Java applications, we will install the Dashboard View plugin for better management of builds and to display results of builds and tests.

This plugin provides a portal-like view for Jenkins build jobs. Download it from `https://wiki.jenkins-ci.org/display/JENKINS/Dashboard+View`. It will be beneficial in showing results and trends. In addition, it also allows users to arrange display items in an effective manner. On the Jenkins dashboard, go to the **Manage Jenkins** link, click on **Manage Plugins,** and install the **Dashboard View** plugin. Verify the successful installation by clicking on the **Installed** tab.

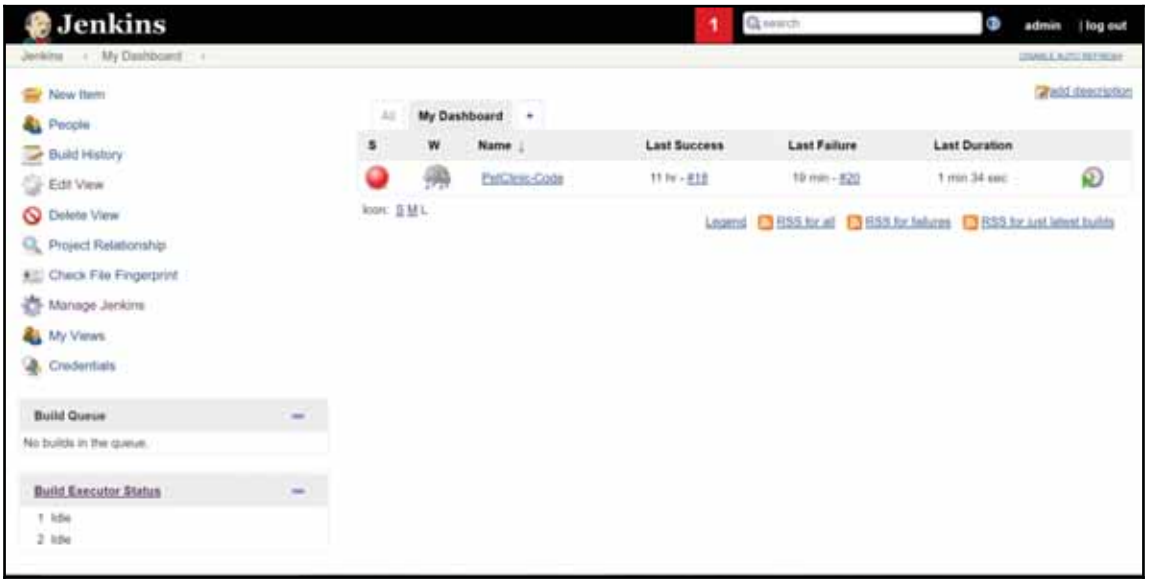
Now go to the Jenkins dashboard and click on the plus sign available on the tab.

Provide a **View name**, select **Dashboard**, and click on **OK**:

Once the **Dashboard** view is created, we can configure it by selecting **Jobs** and customizing it as shown in the following screenshot:



This is what our dashboard looks like. We can configure multiple projects and multiple portlets can be set in different sections available on dashboard:



Often the **Dashboard View** plugin is used to organize and arrange jobs with specific details, which is essential to know for an administrator or users in the context of automation results.

Configuring different portlets related to test results in the **Dashboard View**.

In `Chapter 2`, *Installation and Configuration of Code Repository and Build Tools*, we installed and configured Java and Ant. We will now integrate a sample Ant project with Jenkins and understand how things work.

Creating and configuring a build job for a Java application with Ant

We always say that tools are not important, but it is always a good idea to have some understanding of tools so we can perform operations and troubleshoot in an easy manner.

Ant uses the `build.xml` file to execute different tasks that lead to the creation of a `package` file.

We will use a sample project for Ant that is available at `https://github.com/mitesh51/AntExample`.

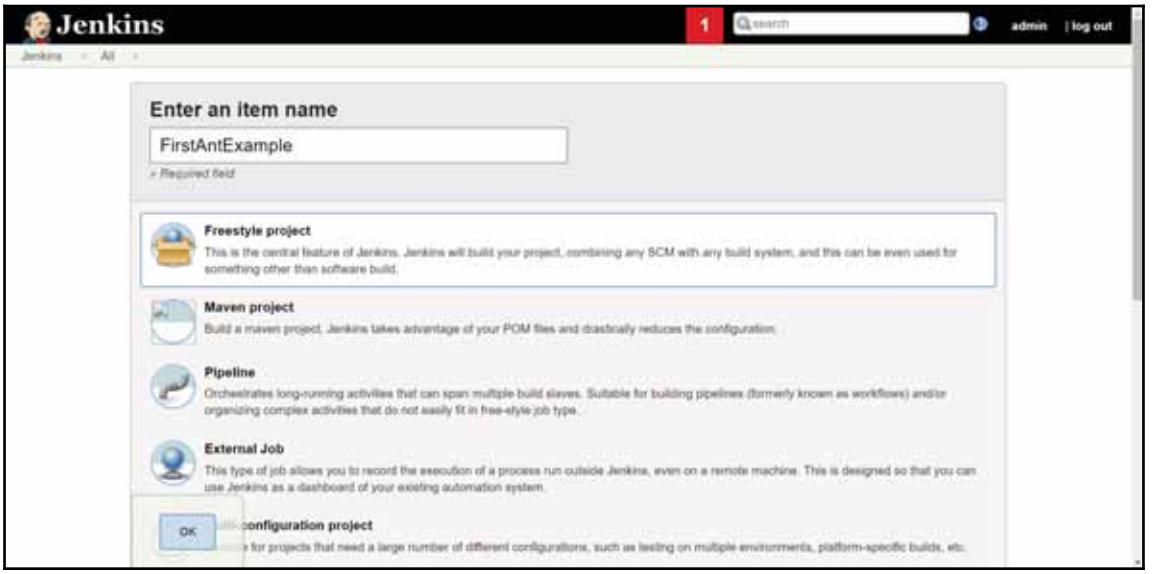
Its `build.xml` file contains the following details:

```
<?xml version="1.0" ?>
<project name="AntExample1" default="war">

<path id="compile.classpath">
<fileset dir="WebContent/WEB-INF/lib">
<include name="*.jar"/>
</fileset>
</path>

<target name="init">
<mkdir dir="build/classes"/>
<mkdir dir="dist" />
</target>

<target name="compile" depends="init">
<javac destdir="build/classes" debug="true" srcdir="src">
<classpath refid="compile.classpath"/>
</javac>
</target>

<target name="war" depends="compile">
<war destfile="dist/AntExample.war"
 webxml="WebContent/WEB-INF/web.xml">
<fileset dir="WebContent"/>
<lib dir="WebContent/WEB-INF/lib"/>
<classes dir="build/classes"/>
</war>
</target>

<target name="clean">
<delete dir="dist" />
<delete dir="build" />
</target>

</project>
```

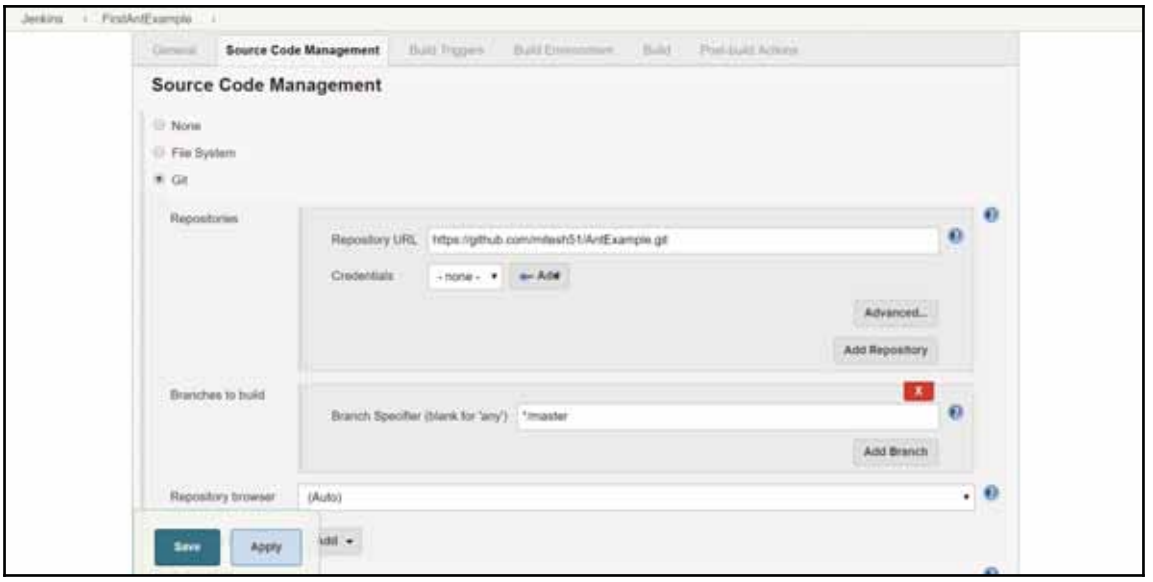
We will use a war target from this build file to create a package that we can deploy in a Local or Remote Tomcat server.

We will now create a **Freestyle project** in Jenkins for our first Ant project integration in Jenkins:

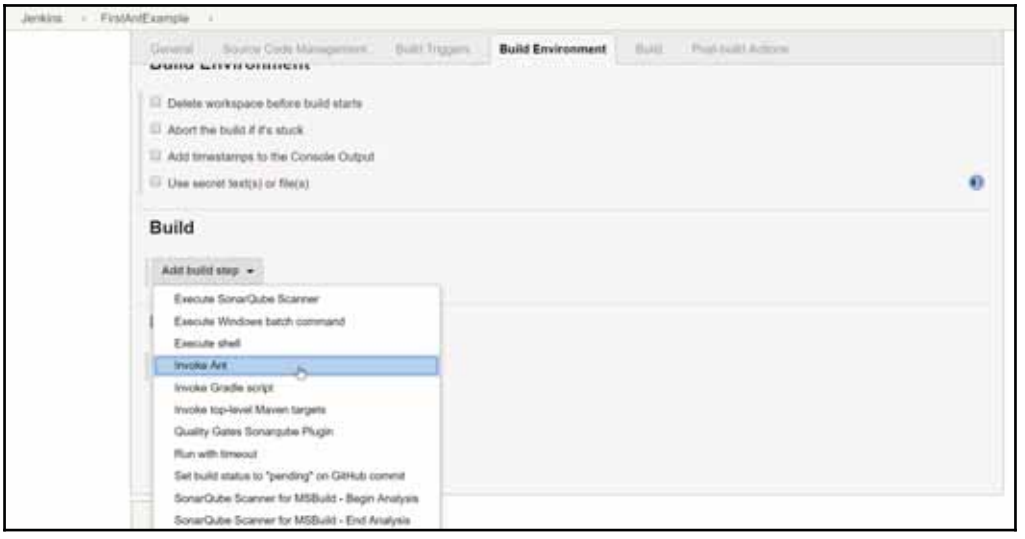
1. Go to the **Jenkins** dashboard and click on **New item**.
2. Enter an item name and select **Freestyle** project.
3. Click **OK**.



4. Once a project is created in Jenkins, it will open it in edit mode. Here we can configure all things related to automation that we want to perform in different sections.
5. Go to the **Source Code Management** section and select **Git**.
6. Provide a **Repository URL**. In our case, it is available on GitHub.

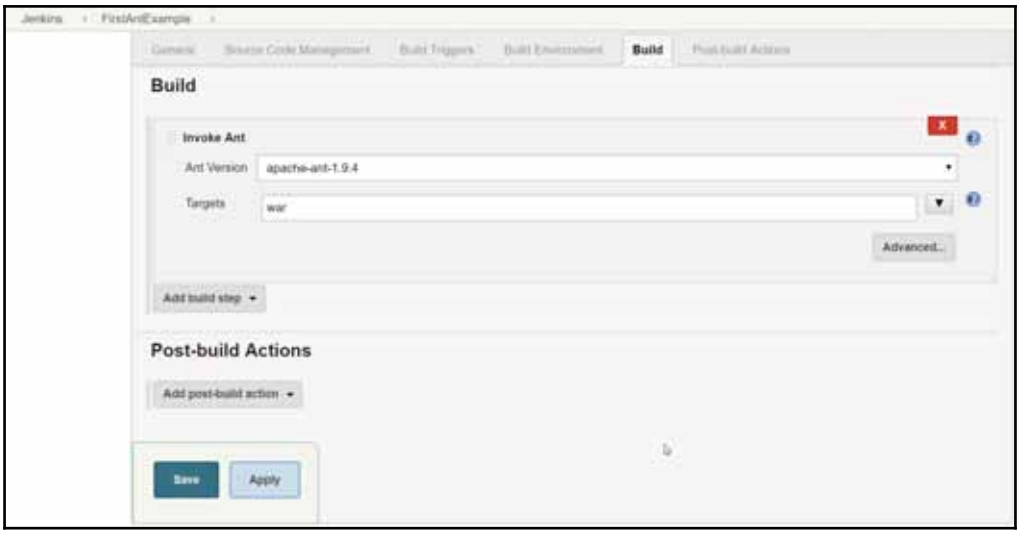


7. Click on the **Build Environment** section.
8. Click on **Add build step** and select **Invoke Ant**.

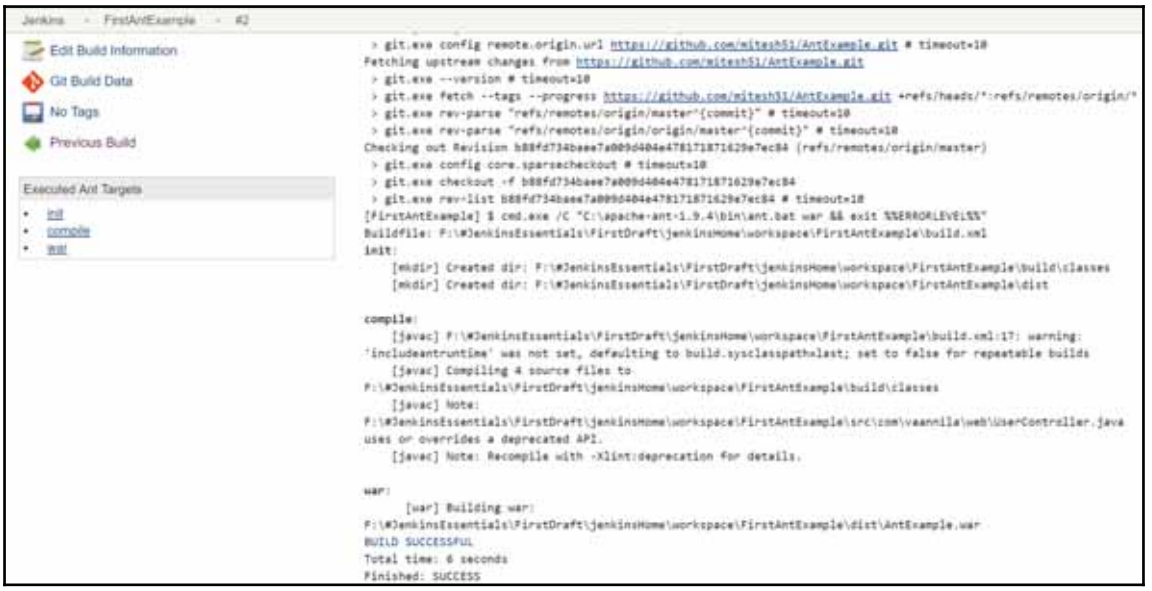


9. We have configured **Apache Ant** in Global Tool Configuration.
10. Select **Ant configured** in Jenkins from the list box.

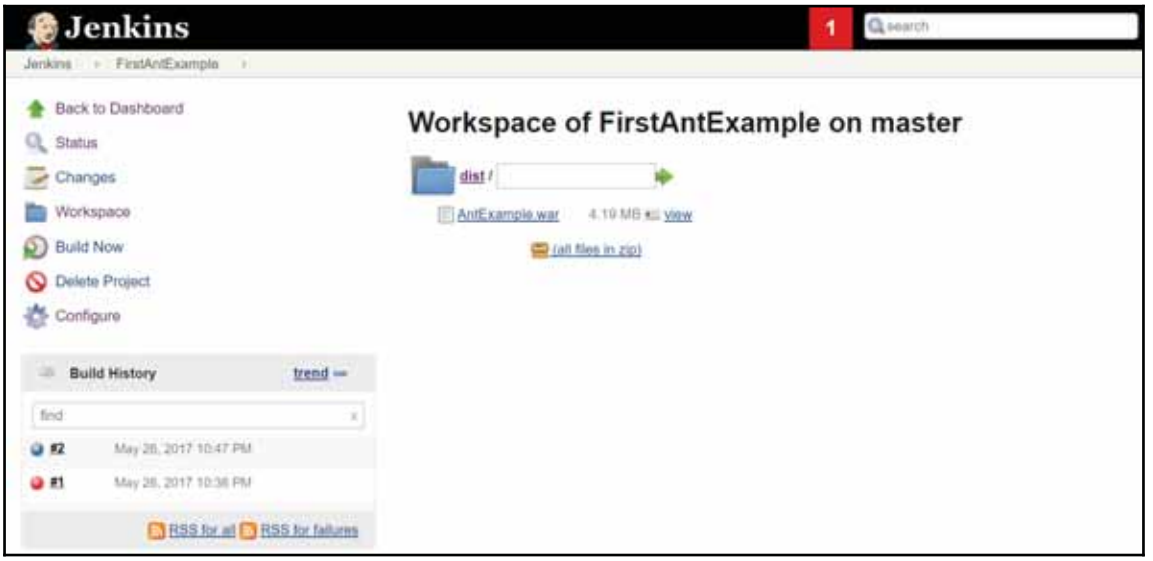
11. Select the target we want to execute from the `build.xml` file and click on **Save**.



12. Click on **Build now**.
13. Go to the Console output of a currently executed build.



14. Observe the log. All targets available in `build.xml` will be executed based on dependencies mentioned in it.
15. Verify that the `war` file has been created.
16. Go to **Workspace** and find the `war` file in the `dist` directory:



So we have now seen how to configure an Ant-based Java project in Jenkins and create a `package` file.

Creating and configuring a build job for a Java application with Maven

Apache Maven is a build automation tool specifically used for Java-based projects to automate the creation of an application build by compiling source code, running automated unit tests, and packaging binary code.

It is based on the **Project Object Model** (**POM**). We will use a PetClinic Maven-based project available at `https://github.com/mitesh51/spring-petclinic`.

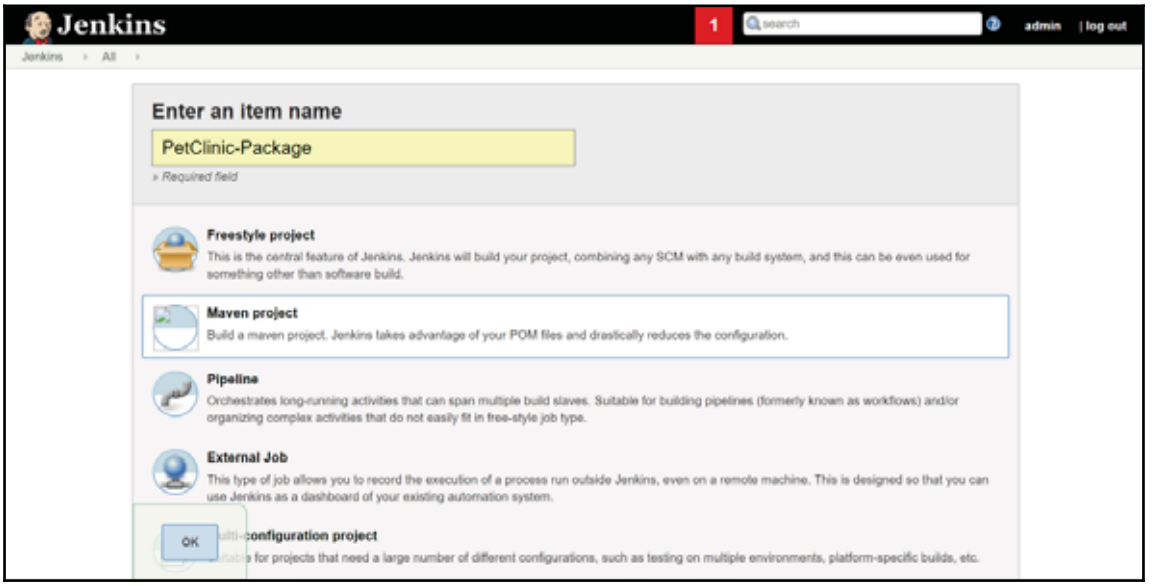
It has a `pom.xml` that is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://maven.apache.org/POM/4.0.0"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.springframework.samples</groupId>
<artifactId>spring-petclinic</artifactId>
<version>4.2.5-SNAPSHOT</version>
<name>petclinic</name>
<packaging>war</packaging>
<properties>
<!-- Generic properties -->
<java.version>1.7</java.version>
<project.build.sourceEncoding>UTF-
 8</project.build.sourceEncoding><project.reporting.outputEncoding>UTF-
 8</project.reporting.outputEncoding>
<!-- Spring -->
<spring-io-platform.version>2.0.3.RELEASE</spring-io-
 platform.version><spring-data-jdbc.version>1.1.0.RELEASE</spring-data-
 jdbc.version>
<!-- Java EE / Java SE dependencies -->
<tomcat.version>7.0.59</tomcat.version>
<!-- Test -->
<assertj.version>2.2.0</assertj.version>
<!-- Dates -->
<jodatime-hibernate.version>1.3</jodatime-
 hibernate.version>
<jodatime-jsptags.version>1.1.1</jodatime-jsptags.version>
<jadira-usertype-core.version>3.2.0.GA</jadira-usertype-
 core.version>
<!-- Others -->
<mysql-driver.version>5.1.36</mysql-driver.version>
<!-- Web dependencies -->
<dandelion.version>1.1.1</dandelion.version>
<dandelion.datatables.version>1.1.0
 </dandelion.datatables.version>
<cobertura.version>2.7</cobertura.version>
</properties>
<dependencyManagement>
<!-- Import the maven Spring IO Platform Bill Of Materials
 (BOM) -->
<dependencies>
<dependency>
<groupId>io.spring.platform</groupId>
<artifactId>platform-bom</artifactId>
```

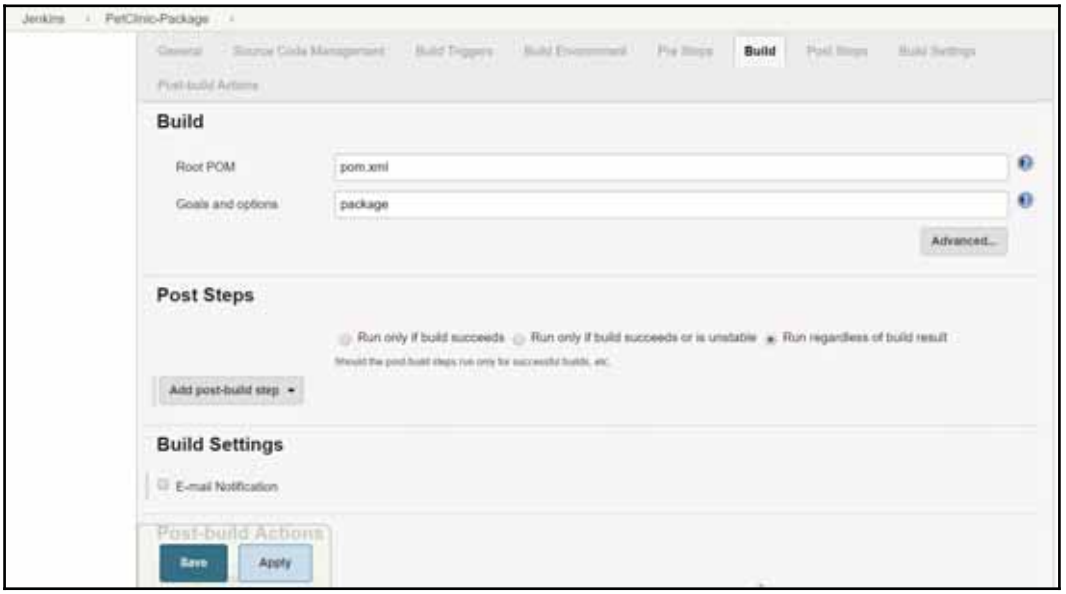
```
<version>${spring-io-platform.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
<url>demopetclinic</url>
</project>
```

In `Chapter 2`, *Installation and Configuration of Code Repository and Build Tools*, we have already configured Java and Maven. We need to utilize both in our Maven project in Jenkins:

1. Go to the **Jenkins** dashboard and click on **New item**. Enter an item name and select **Maven project**. Click **OK**:



2. In the **Build** section, give the location of **Root POM**. There are defined goals for Maven-based projects. Give `package` as a **Goal:**



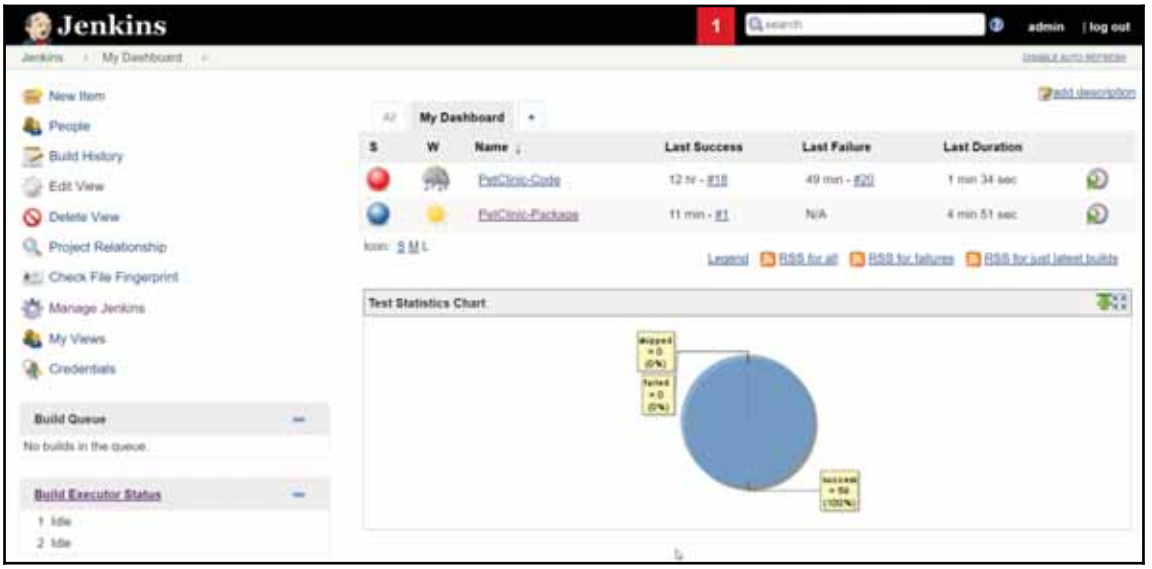
3. Click on **Save**.
4. If we are behind the proxy then we need to provide proxy details in Maven so it can use that to download dependencies.
5. Go to the `MAVEN_HOME` directory.
6. Go to the `conf` directory and open `settings.xml`.

   Uncomment the following block:

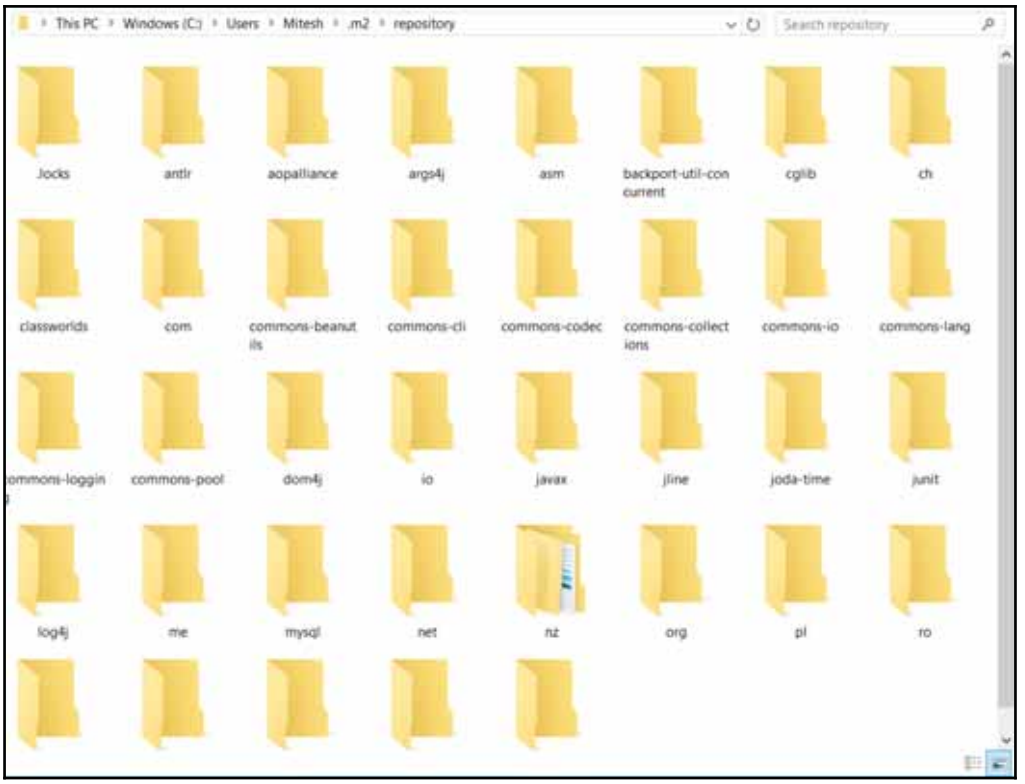
```
<proxies>
<proxy>
<id>optional</id>
<active>true</active>
<protocol>http</protocol>
<username>testuser</username>
<password>sdbjsdhbhjw</password>
<host>proxy.***.com</host>
<port>8080</port>
<nonProxyHosts>localhost*</nonProxyHosts>
</proxy>
</proxies>
```

7. Click on **Build Now**.
8. Go to **Console Output** to verify the log. As it is a Maven project, it will download dependencies mentioned in the `pom.xml` file from the internet.
9. Verify the test results in the log and also verify that the `.war` file has been created.

10. Go to the customized dashboard that we have created and verify whether any results have come in or not:



11. To verify the JAR files that are downloaded from the Maven repository, go to the `Users` directory and select the specific user that we have logged in as. Find the `.m2` directory and all the JAR files will be available there in the case of Windows operating systems:



12. To check the `package` file created with Jenkins and Maven integration, go to the `JENKINS_HOME/workspace/jobname/target` directory.
13. We have successfully created a `war` file that can be deployed on the server. It can be a web server or an application server. We will use the Tomcat server for application deployment.

Summary

As we promised in the beginning of the chapter, we have covered how to integrate Ant and Maven based applications in Jenkins in detail. We have also provided details on the **Dashboard View** plugin.

In the next chapter, we will cover deployment of the `war` file that we have created using Continuous Integration. We will deploy application packages in different public clouds such as AWS and Microsoft Azure in Infrastructure as a Service and Platform as a Service.

5

Continuous Delivery - Implementing Automated Deployment

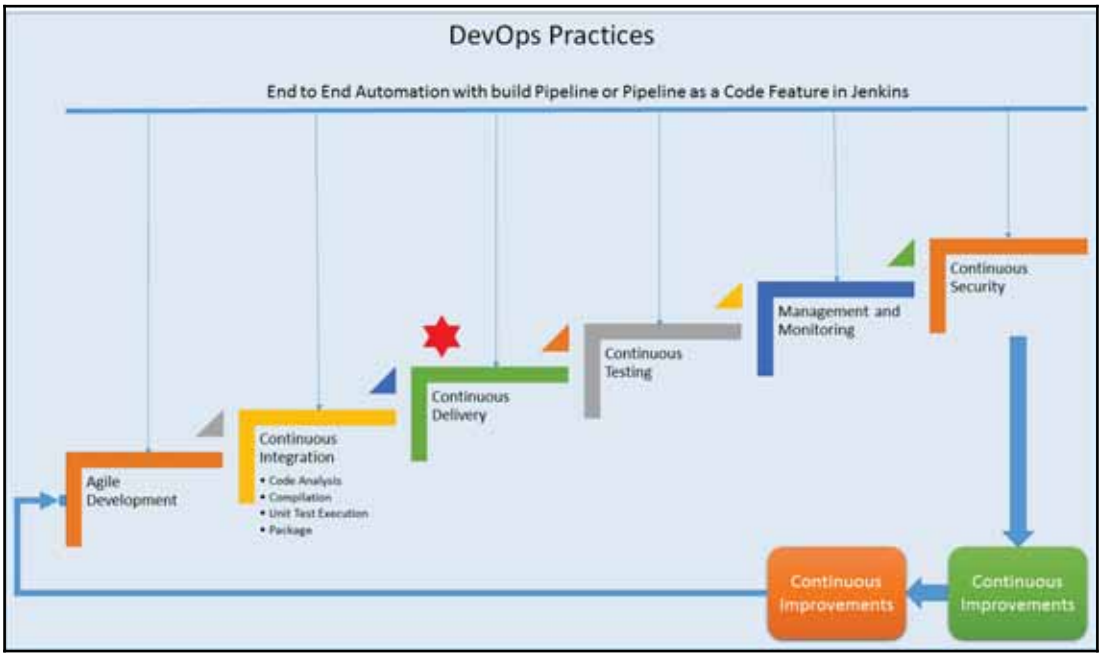
Once we have a package ready for deployment after Continuous Integration, our next step should be deployment of that package in to a web or application server.

We can deploy WAR files manually, or with commands (batch file or shell script), or with Jenkins plugins, or any third-party tool that can be integrated with Jenkins. In our case, we will use Jenkins plugins for application deployment into runtime environments, which can be local or remote.

This chapter will take one step forward in the DevOps pipeline by deploying artifacts in local or remote application servers. It will give insight into automated deployment and continuous delivery processes and it will also cover how to deploy applications on public cloud platforms using Jenkins. In this chapter, we will cover following topics:

- An overview of Continuous Delivery and Continuous Deployment
- Installing Tomcat
- Deploying a war file from Jenkins to Tomcat
- Deploying a war file from Jenkins to AWS Elastic Beanstalk
- Deploying a war file from Jenkins to Microsoft Azure App Services

In this chapter, we will cover the main parts of Continuous Delivery practice as a part of our DevOps journey:



At the end of this chapter, we will know how to deploy an application to web or application servers in AWS and Microsoft Azure.

An overview of Continuous Delivery and Continuous Deployment

Continuous Delivery (**CD**) is a DevOps practice that is used to deploy an application quickly to a high quality, with an automated approach, in non-production environments. In Continuous Delivery, an application package is always production ready.

Continuous Deployment is a DevOps practice that is used to deploy an application quickly to a high quality, with an automated approach, in a production environment.

Automated approaches to deploying application packages in production and non-production won't change. The approval process may be set up in cases of the deployment of application packages in the production environment though.

In the following sections, we will deploy war files into different environments using different approaches.

Installing Tomcat

Apache Tomcat is an open source server that can be utilized to deploy Java-based web applications. Apache Tomcat implements several Java EE specifications including Java Servlet, JavaServer Pages, Java EL, and WebSocket.

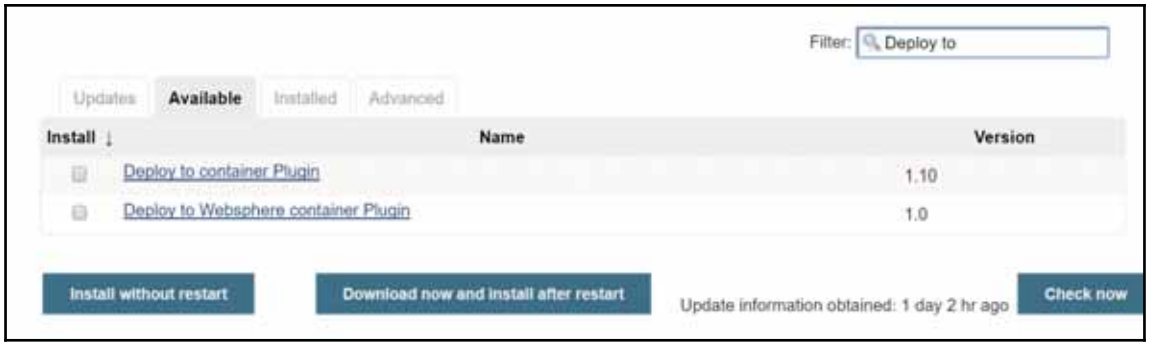
Tomcat installation is very simple. Download it from `http://tomcat.apache.org/`, based on the requirement.

Download the installable files and extract them. Go to the folder and find the bin directory. Based on the operating system, run `startup.bat` or `startup.sh` in the command window or Terminal.

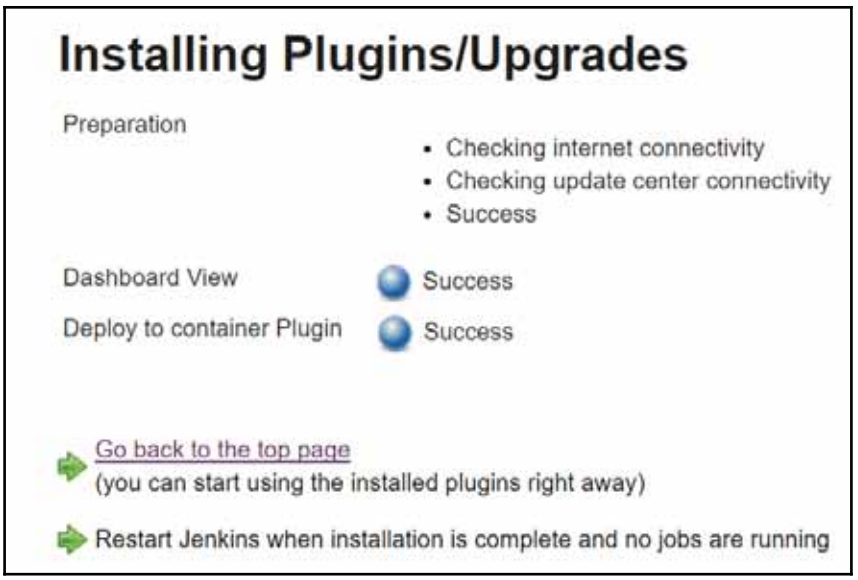
Deploying a war file from Jenkins to Tomcat

For application deployment, we can utilize multiple ways to deploy an application in a web server or application server. We can use batch script or shell script to copy the package file created after a Continuous Integration process, or we can use a Jenkins plugin to deploy an application:

1. Go to **Manage Jenkins** | **Manage Plugins** and install **Deploy to container Plugin**:



2. Wait until the plugin is installed successfully:

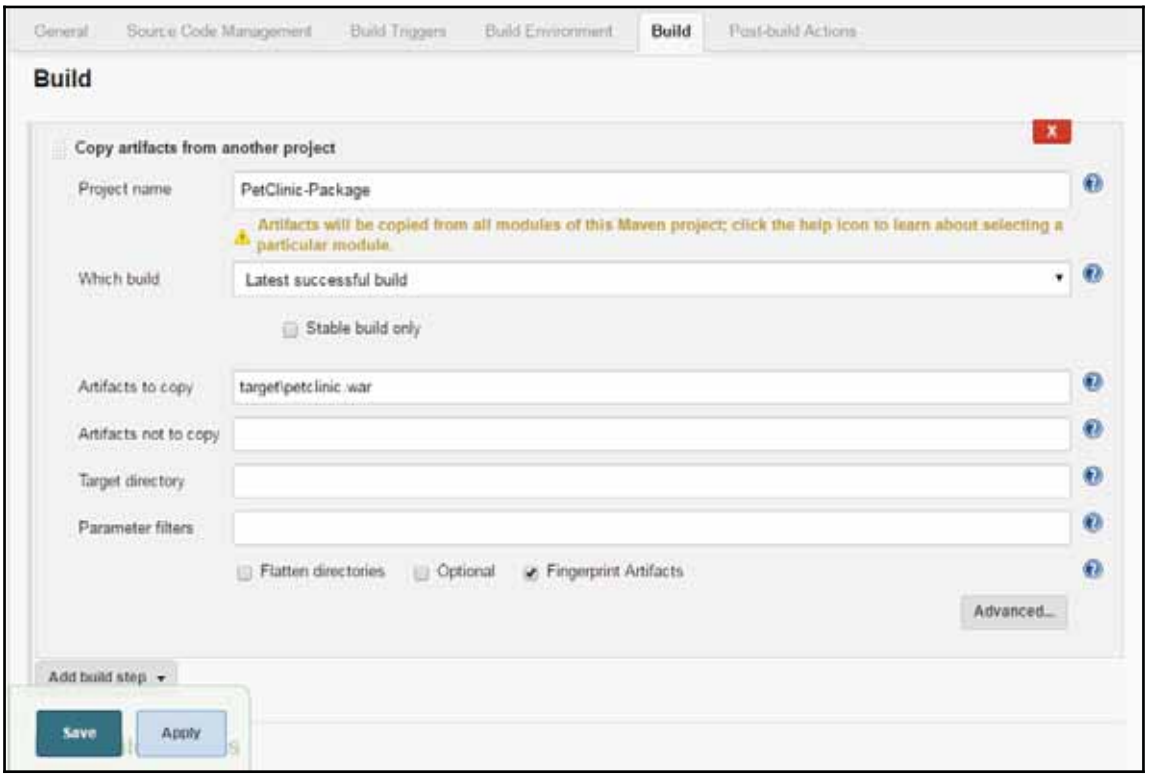


3. To allow deployment using the Jenkins plugin, go to the Tomcat installation directory and `open conf\tomcat-users.xml`.
4. Create a new role and new user as follows:

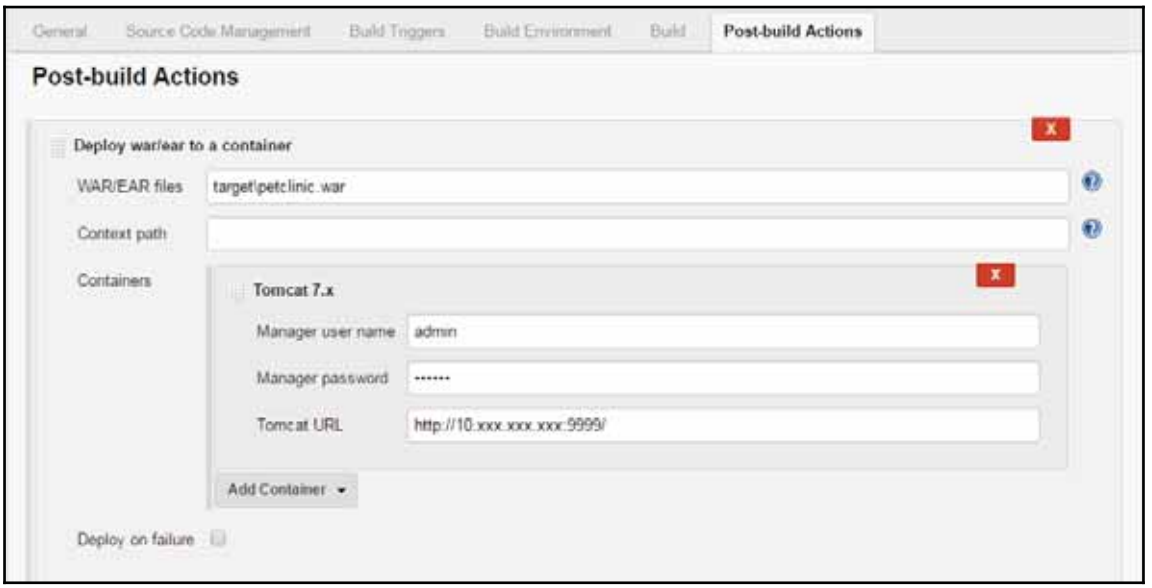
```
<?xml version='1.0' encoding='utf-8'?>
<!--
<tomcat-users>
<!--
<!--
  NOTE:  The sample user and role entries below are intended for use with the
  examples web application. They are wrapped in a comment and thus are ignored
  when reading this file. If you wish to configure these users for use with the
  examples web application, do not forget to remove the <!.. ..> that surrounds
  them. You will also need to set the passwords to something appropriate.
-->
  <role rolename="manager-script"/>
  <user username="admin" password="admin@123" roles="manager-script"/>
</tomcat-users>
```

5. Restart Tomcat.
6. Create a new **Freestyle build** in Jenkins named `PetClinic-Deploy`.

7. What we will do here is copy the artifact created from the `PetClinic-Package` job and deploy it in Tomcat. Install the Copy Artifact plugin to perform this action. Give the project a name and path from which we need to copy the WAR file:



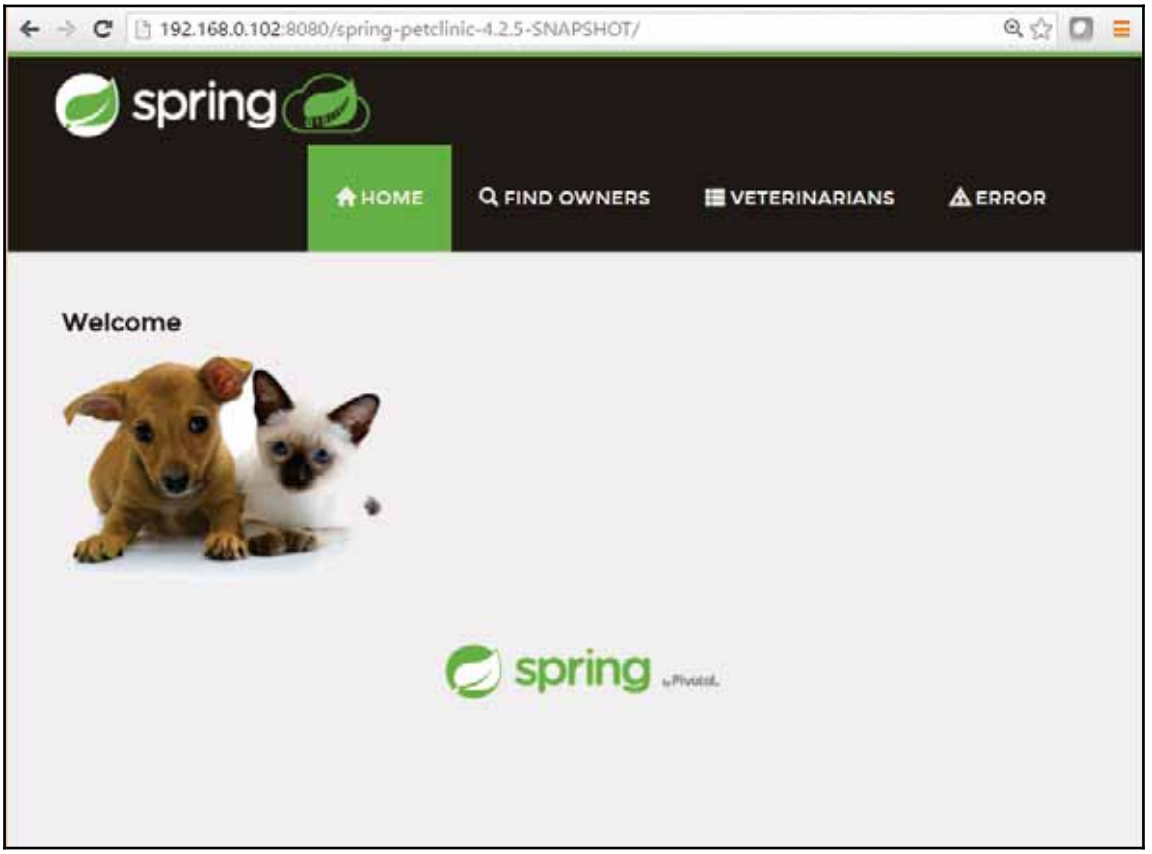
8. Give a path to the WAR file for deployment using the Jenkins plugin. Select **Deploy war/ear to a container** from Post build actions. Click on **Add Container** and select the latest version of Tomcat. Give a Tomcat URL. Give the username and password we defined in `tomcat-users.xml`:



9. Execute the build by clicking on **Build now**. Verify the logs for application deployment:

```
Results :

Tests run: 59, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-war-plugin:2.3:war (default-war) @ spring-petclinic ---
[INFO] Packaging webapp
[INFO] Assembling webapp [spring-petclinic] in [d:\jenkins\workspace\PetClinic-Test\target\spring-
petclinic-4.2.5-SNAPSHOT]
[INFO] Processing war project
[INFO] Copying webapp resources [d:\jenkins\workspace\PetClinic-Test\src\main\webapp]
[INFO] Webapp assembled in [1669 msecs]
[INFO] Building war: d:\jenkins\workspace\PetClinic-Test\target\spring-petclinic-4.2.5-SNAPSHOT.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 28.772 s
[INFO] Finished at: 2016-07-06T22:59:37+05:30
[INFO] Final Memory: 29M/261M
[INFO] ------------------------------------------------------------------------
Deploying d:\jenkins\workspace\PetClinic-Test\target\spring-petclinic-4.2.5-SNAPSHOT.war to container
Tomcat 7.x Remote
  [d:\jenkins\workspace\PetClinic-Test\target\spring-petclinic-4.2.5-SNAPSHOT.war] is not deployed.
Doing a fresh deployment.
  Deploying [d:\jenkins\workspace\PetClinic-Test\target\spring-petclinic-4.2.5-SNAPSHOT.war]
Finished: SUCCESS
```

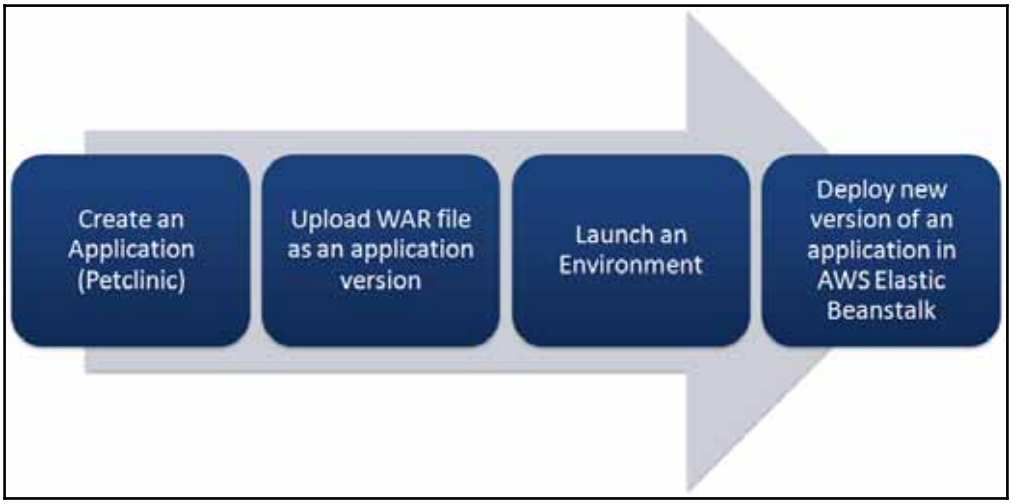
10. Go to a browser and visit the application with the Tomcat URL and the context of an application:



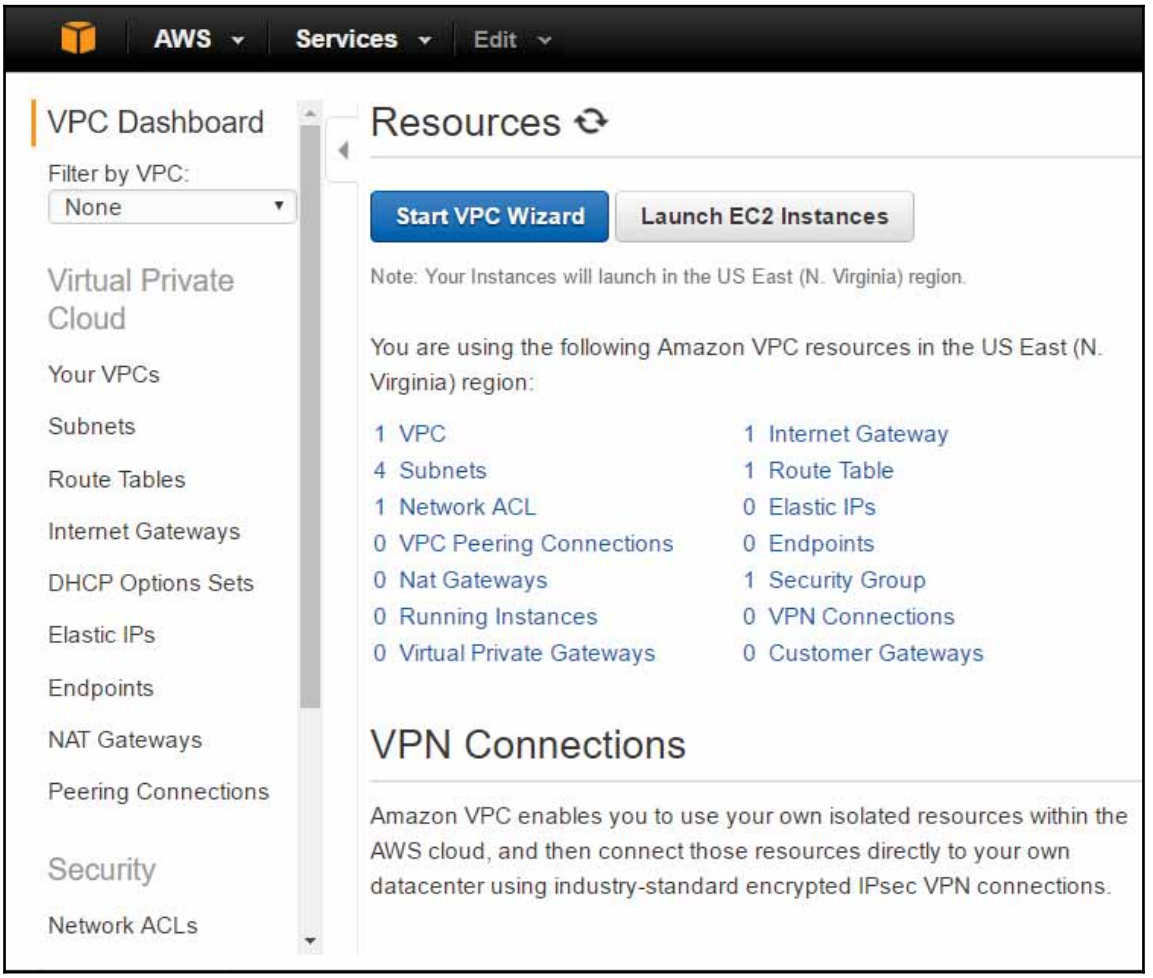
In the next section, we will deploy the PetClinic application in Tomcat that resides in the AWS EC2 instance.

Deploying a WAR file from Jenkins to AWS Elastic Beanstalk

AWS Elastic Beanstalk is a **Platform as a Service** (**PaaS**). We will use it to deploy the PetClinic application. These are the steps to deploy an application on AWS Elastic Beanstalk:

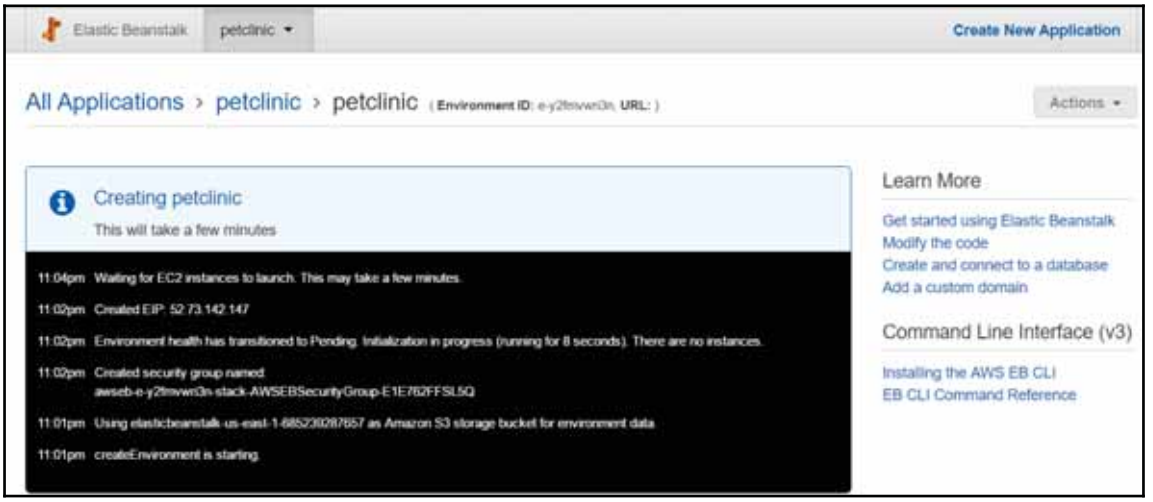


Let's create a sample application in Elastic Beanstalk to understand how Elastic Beanstalk works and then we will use the Jenkins plugin to deploy an application into it:



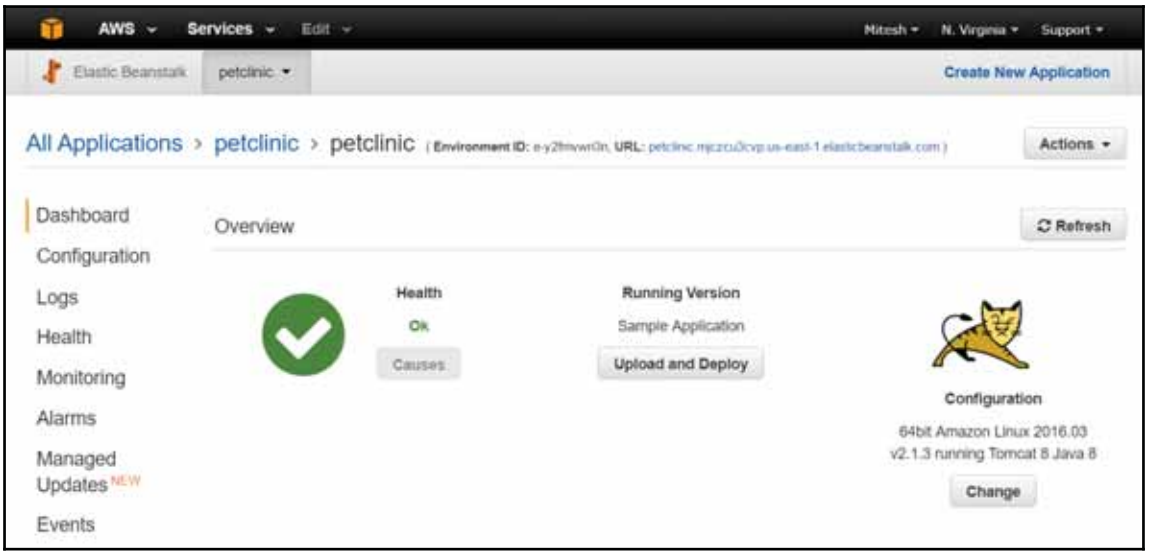
1. Click on **Services** in the AWS management console and select **AWS Elastic Beanstalk**. Create a new application named `petclinic`. Select **Tomcat** as the **Platform** and select the **Sample application** radio button:

2. Verify the sequence of events for the creation of a sample application:

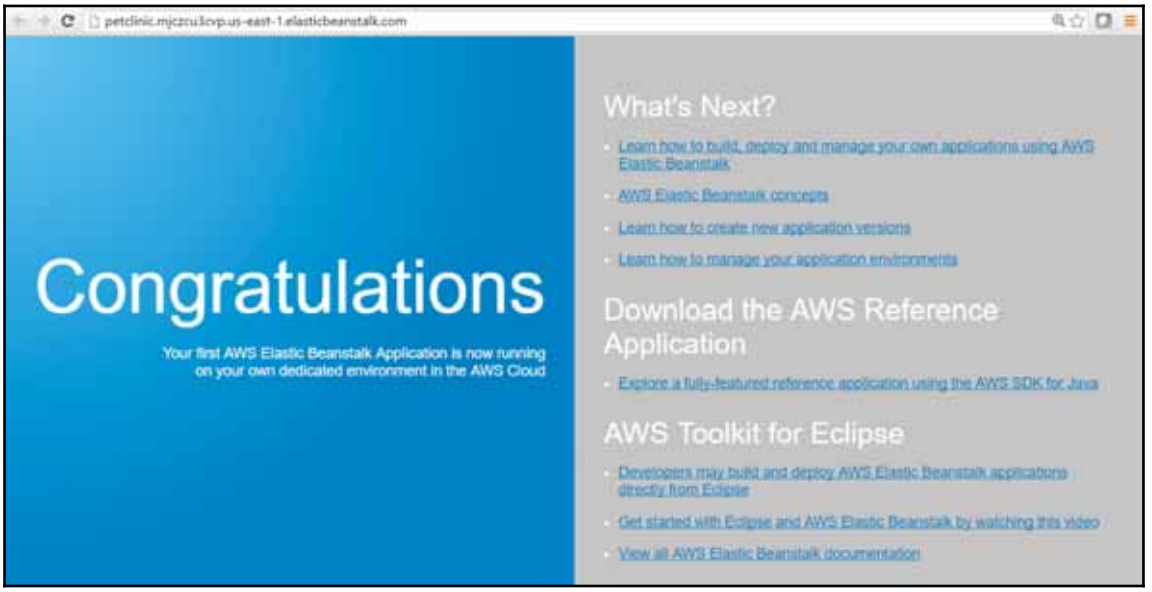


3. It will take some time, and once the environment has been created, it will be highlighted in green:

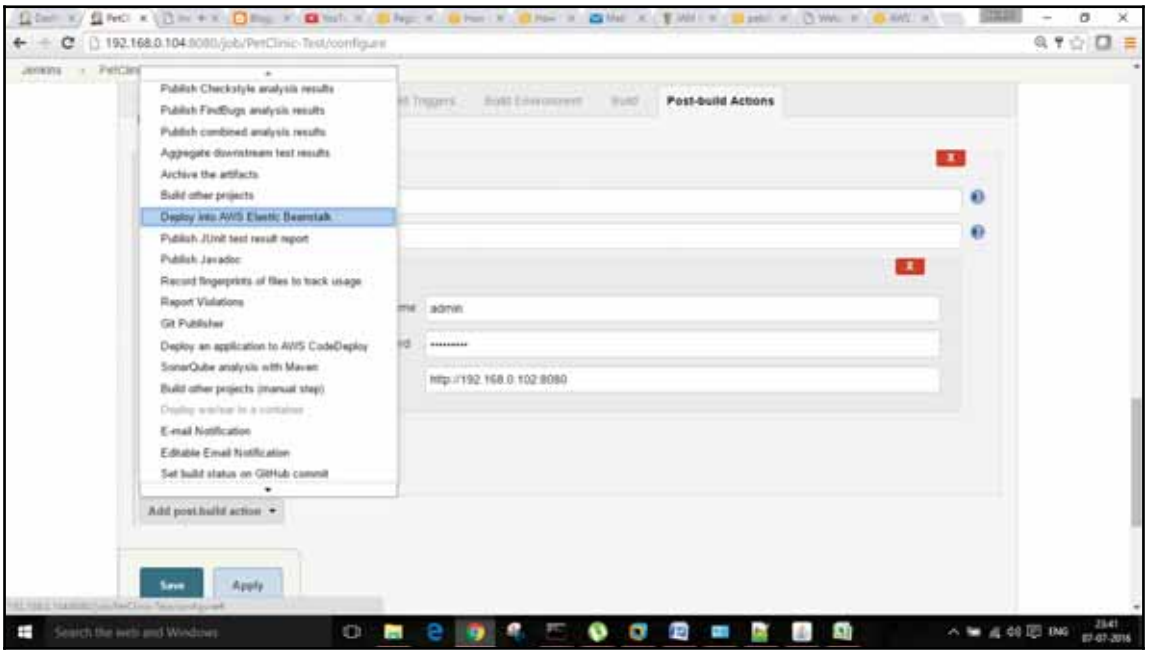
4. Click on the **petclinic** environment and verify that you can see **Health** and **Running Version** in the dashboard:



5. Verify that you can see the environment ID and URL. Click on the URL and verify that you can see the default page:

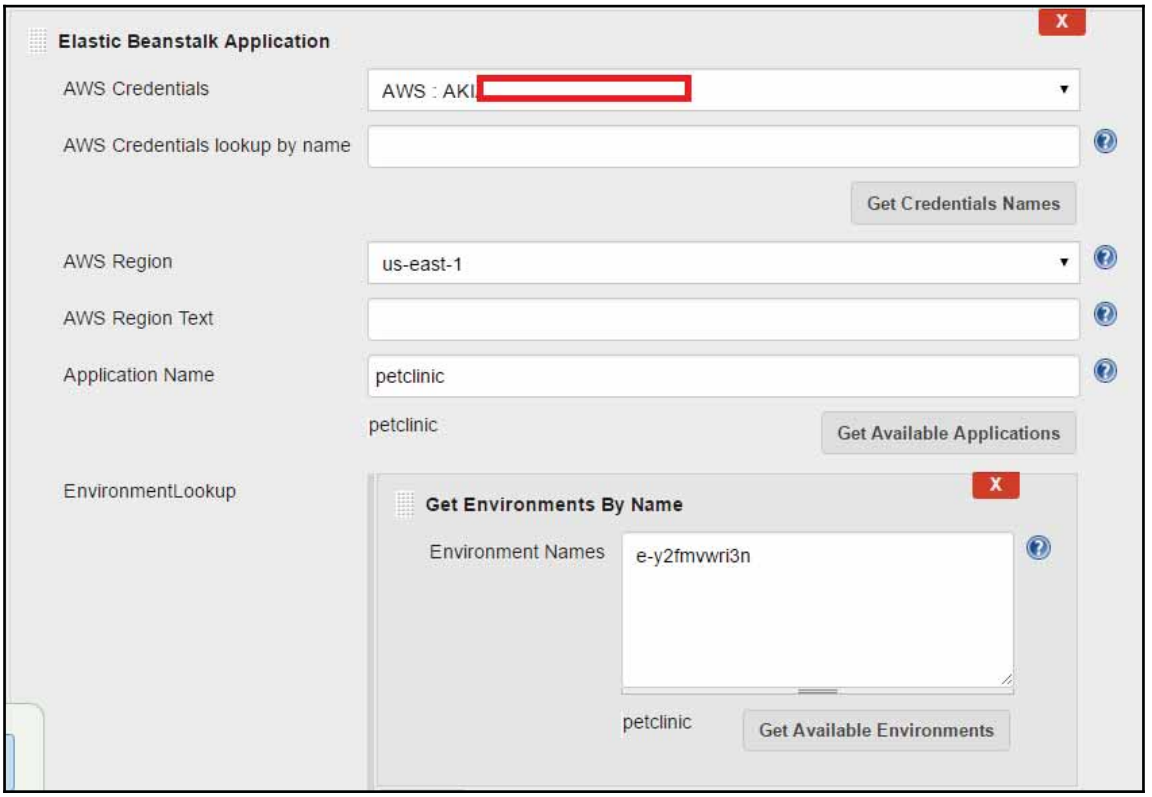


6. Install the AWS Elastic Beanstalk Publisher plugin. For more details, visit `https://wiki.jenkins-ci.org/display/JENKINS/AWS+Beanstalk+Publisher+Plugin`:



7. A new section will come up in **Post-build Actions** for Elastic Beanstalk.
8. Click on the Jenkins dashboard and select **Credentials**; add your AWS credentials.
9. Go to your Jenkins build and select AWS Credential, which is set in the global configuration.
10. Select **AWS Region** from the list and click on **Get Available Applications**. As we have created a sample application, it will show up like this.

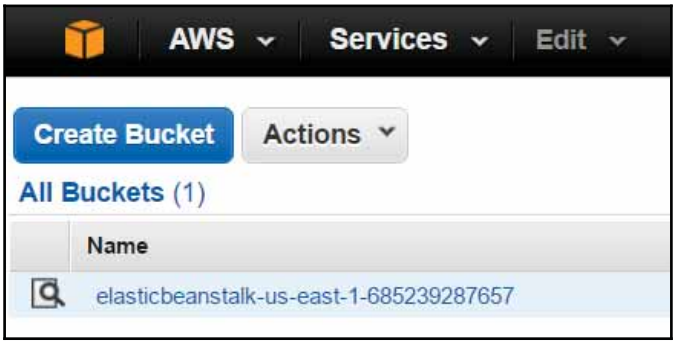
11. In **Environment Lookup**, provide an environment ID in the G**et Environments By Name** box and click on **Get Available Environments**:



12. Save the configuration and click on **Build now**.

Now let's verify the AWS management console to check whether the WAR file is being copied in Amazon S3 or not:

1. Go to S3 Services and check the available buckets:

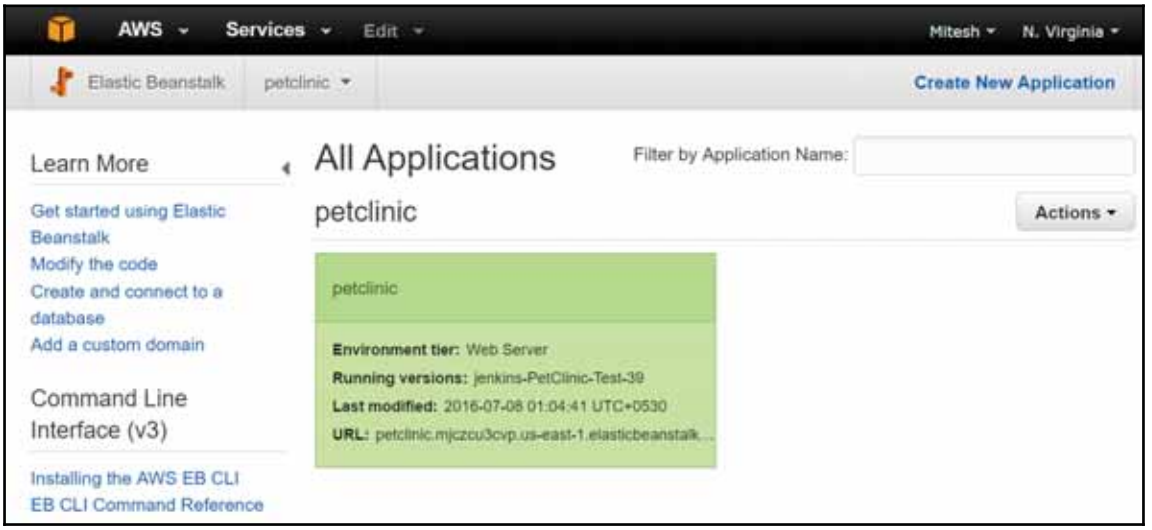


2. Since the WAR file is large, it will take a while to upload to Amazon S3. Once it is uploaded, it will be available in the Amazon S3 bucket.
3. Verify the build job's execution status in Jenkins. Some sections of the expected output are that:

    - The test case execution and WAR file creation are successful
    - The build is successful

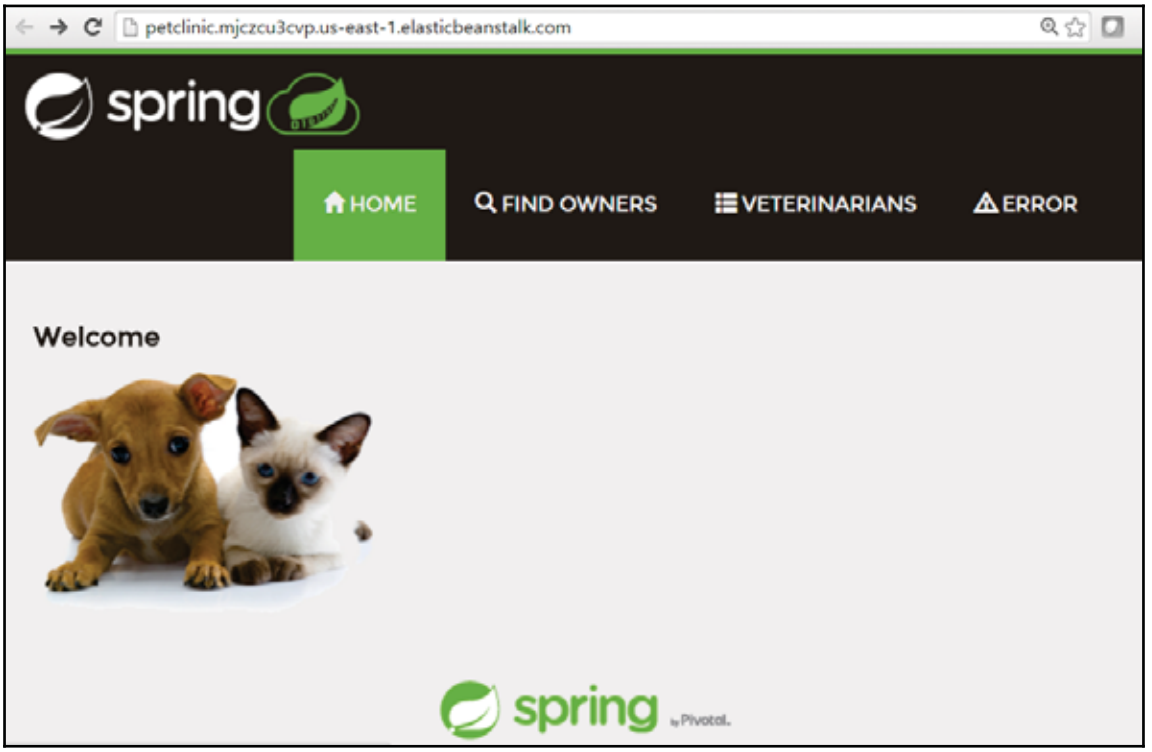
4. Now check the AWS management console:

5. Go to **Services**, click on **AWS Elastic Beanstalk**, and verify the environment. The previous version was **Sample Application**. Now, the version is updated as given in **Version Label Format** in the Jenkins build job configuration:



6. Go to the dashboard and verify **Health** and **Running Version** again.

7. Once everything has been verified, click on the URL for the environment, and our PetClinic application should now be live:

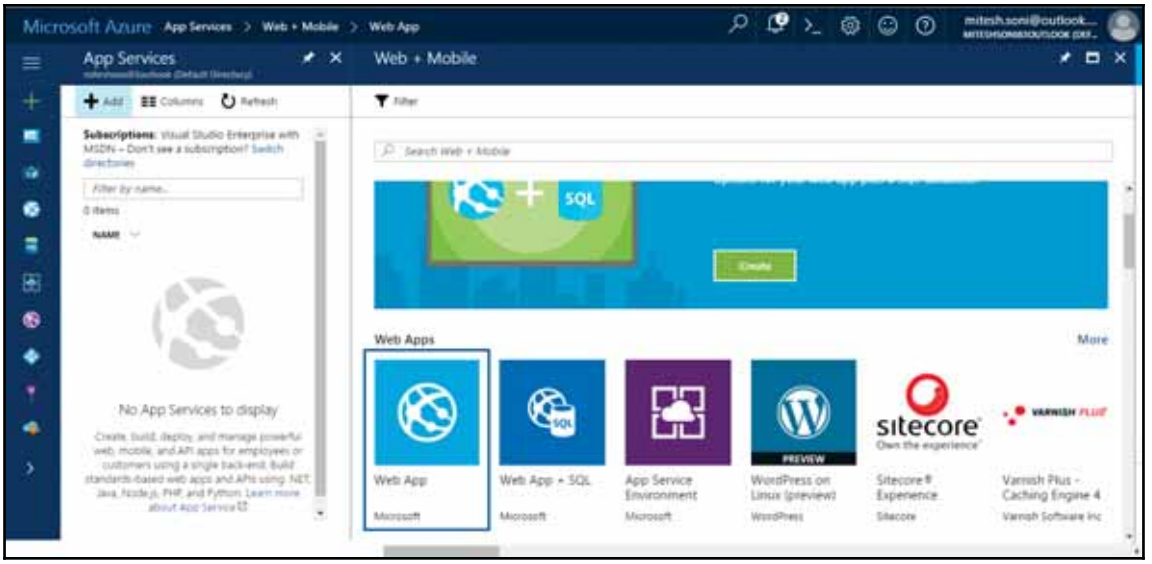


Once the application deployment is successful, terminate the environment. We have thus successfully deployed our application on Elastic Beanstalk.

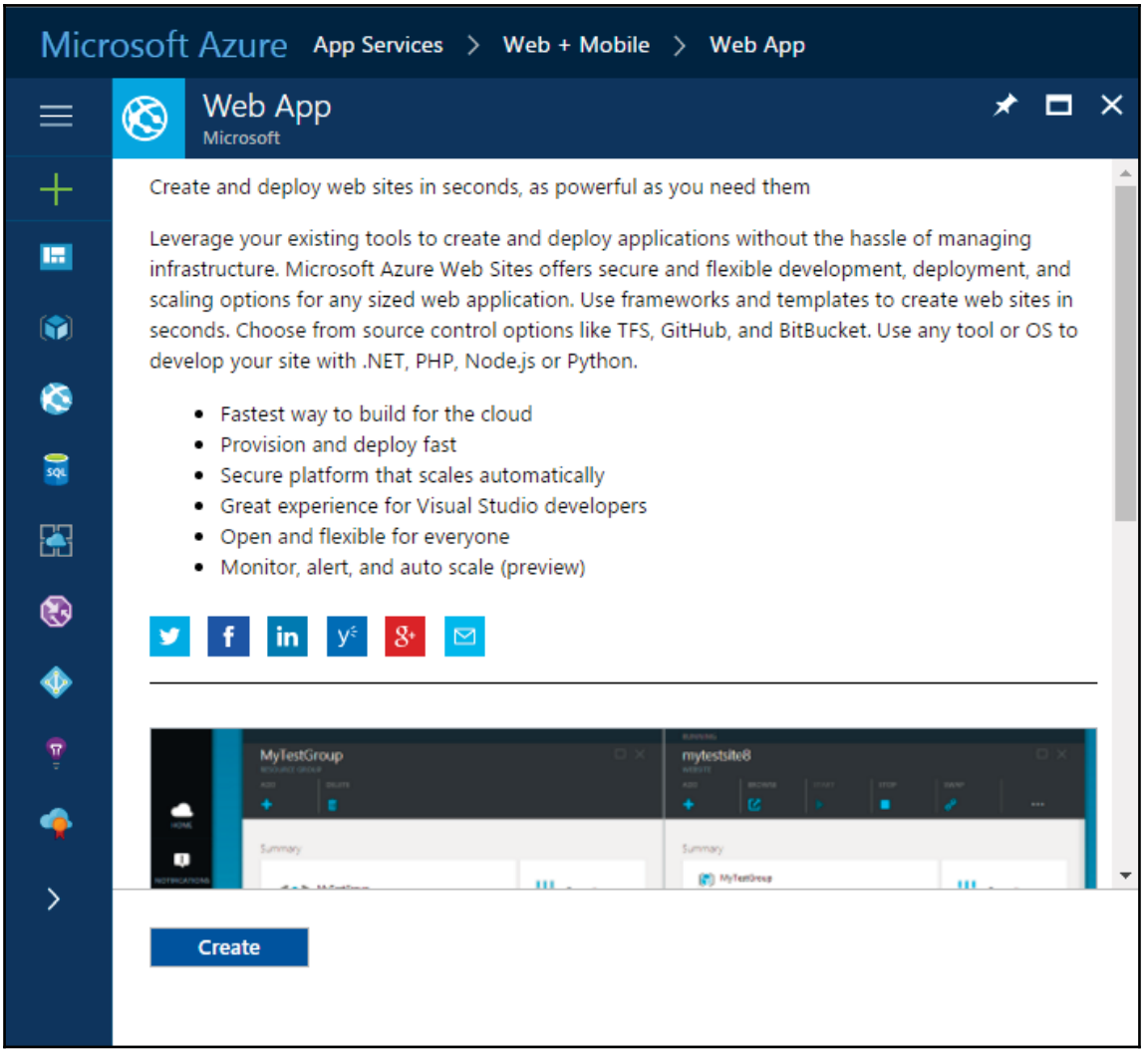
Deploying a war file from Jenkins to Microsoft Azure App Services

Microsoft Azure App Services is a PaaS. In this section, we will look at the Azure Web App and how we can deploy our PetClinic application:

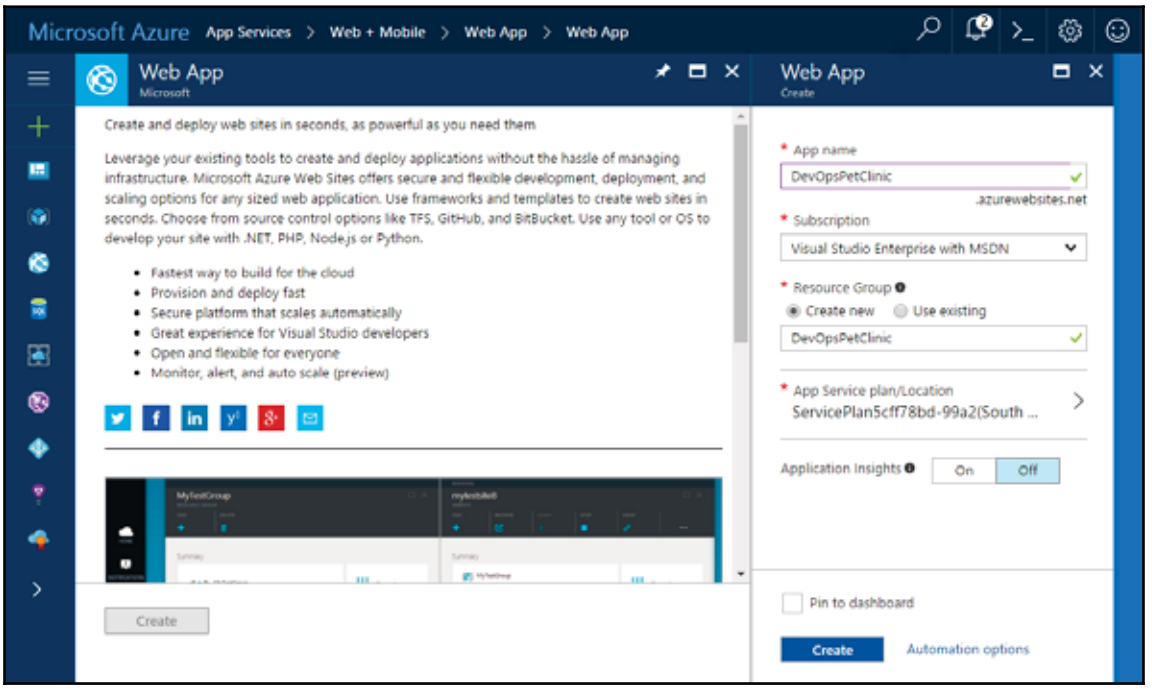
1. We need to have a Microsoft Azure subscription. Go to App Services and click on **Add**:



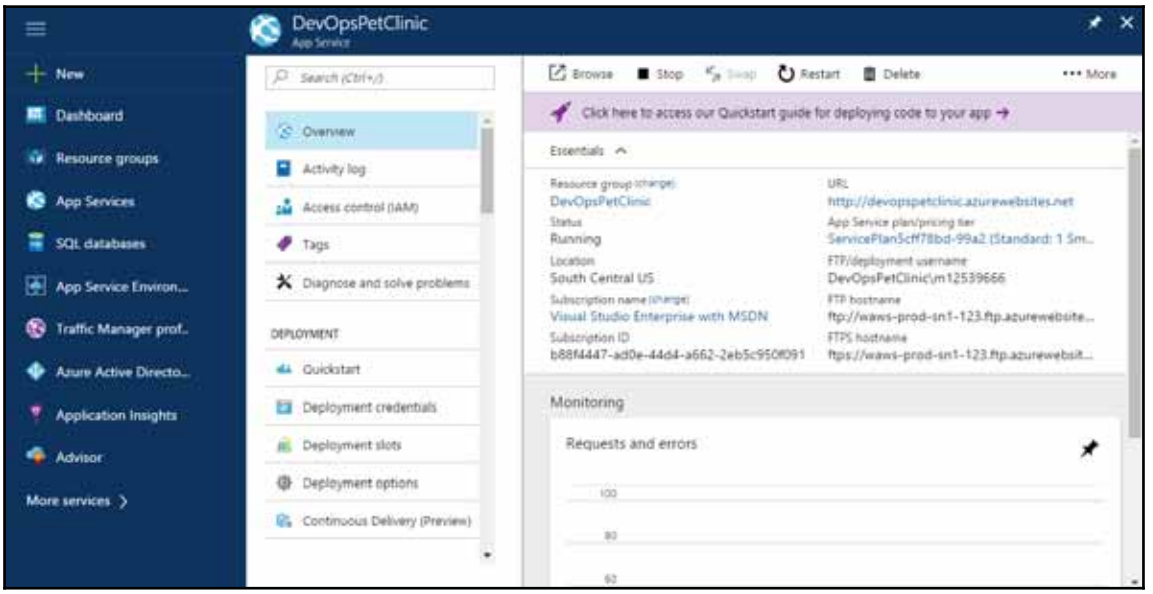
2. Click on **Create**:



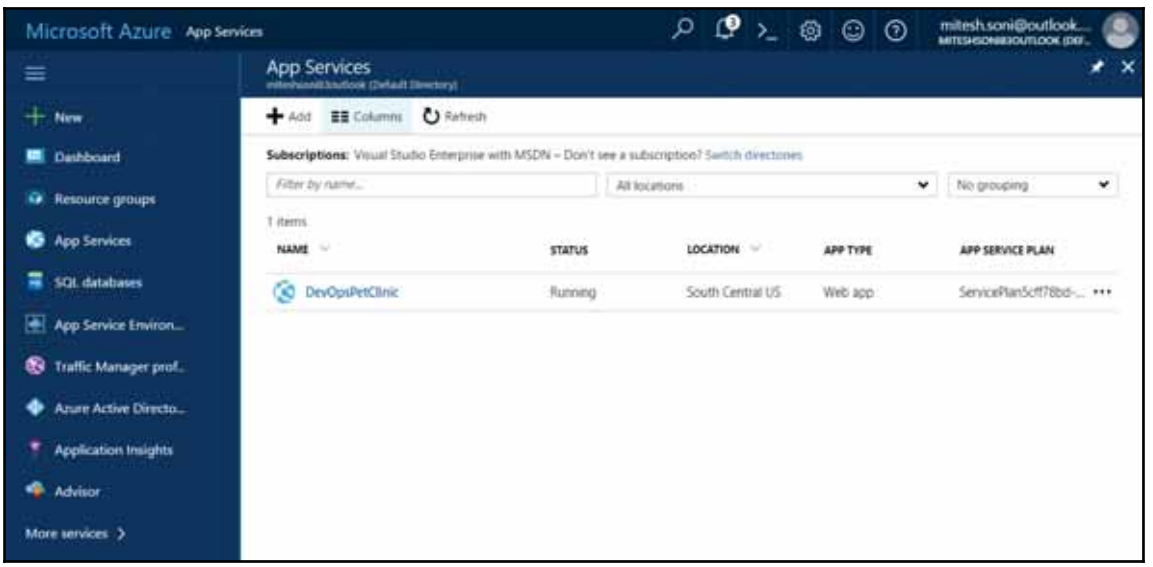
3. Go to the Microsoft Azure portal at `https://portal.azure.com`. Click on **App Services** and then on **Add**. Provide values for **App name**, **Subscription**, **Resource Group**, and **App Service Plan/Location**. Click on **Create**:



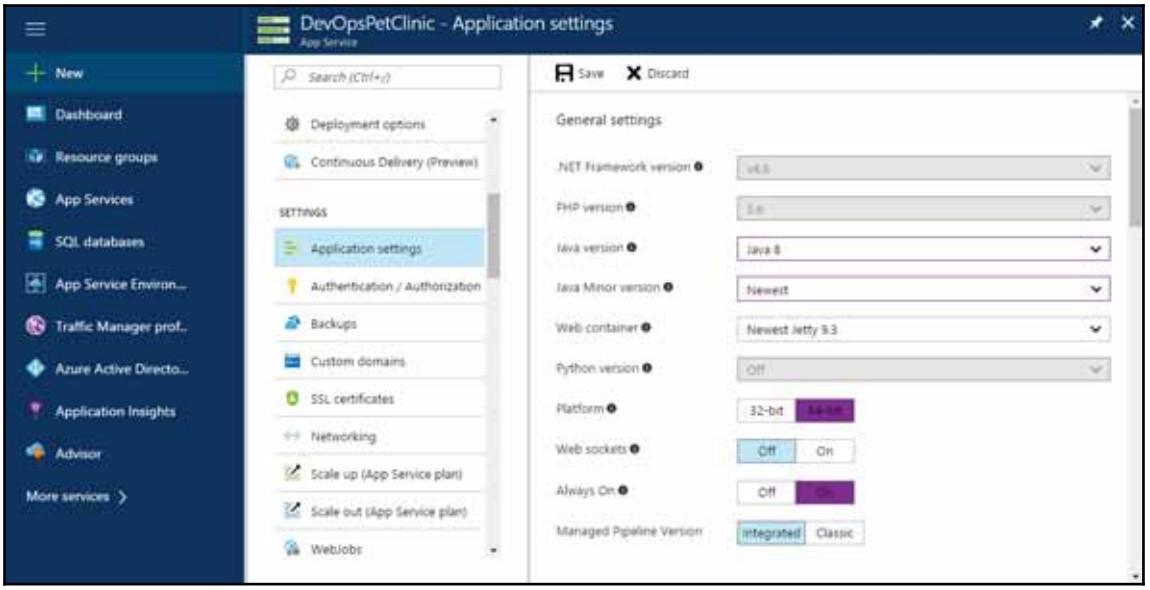
4. Once the Azure Web App is created, see whether it shows up in the Azure portal. Click on **DevOpsPetClinic** for details related to the **URL**, **Status**, **Location**, and so on:



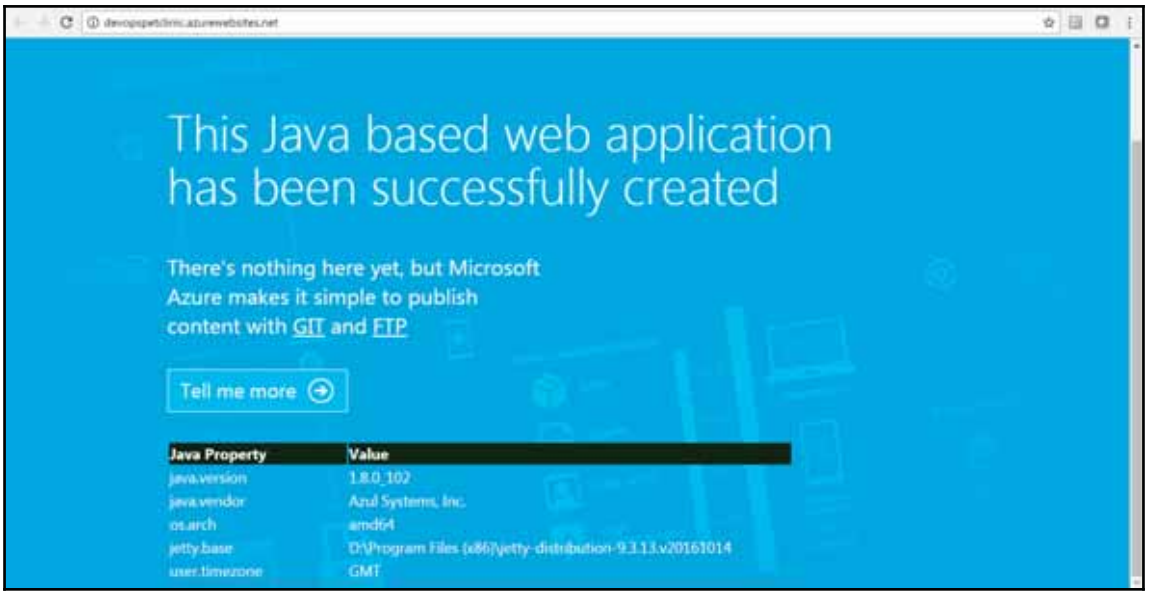
5. Verify you can see this in the **App Services** section too:



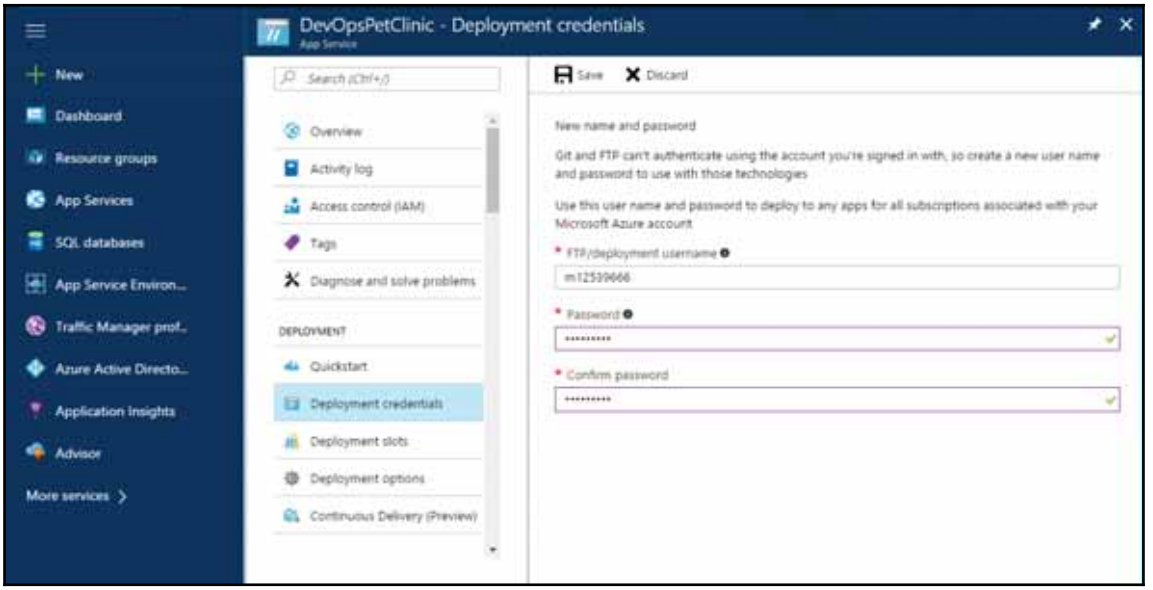
6. Click on **All Settings**, go to the **GENERAL** section, and click on **Application settings** to configure the Azure Web App for Java web application hosting. Select the **Java version**, **Java Minor version**, **Web container,** and **Platform**, and then click on **Always On**:



7. Visit the URL of an Azure Web App from your browser and verify that it is ready to host our sample Spring application, PetClinic:

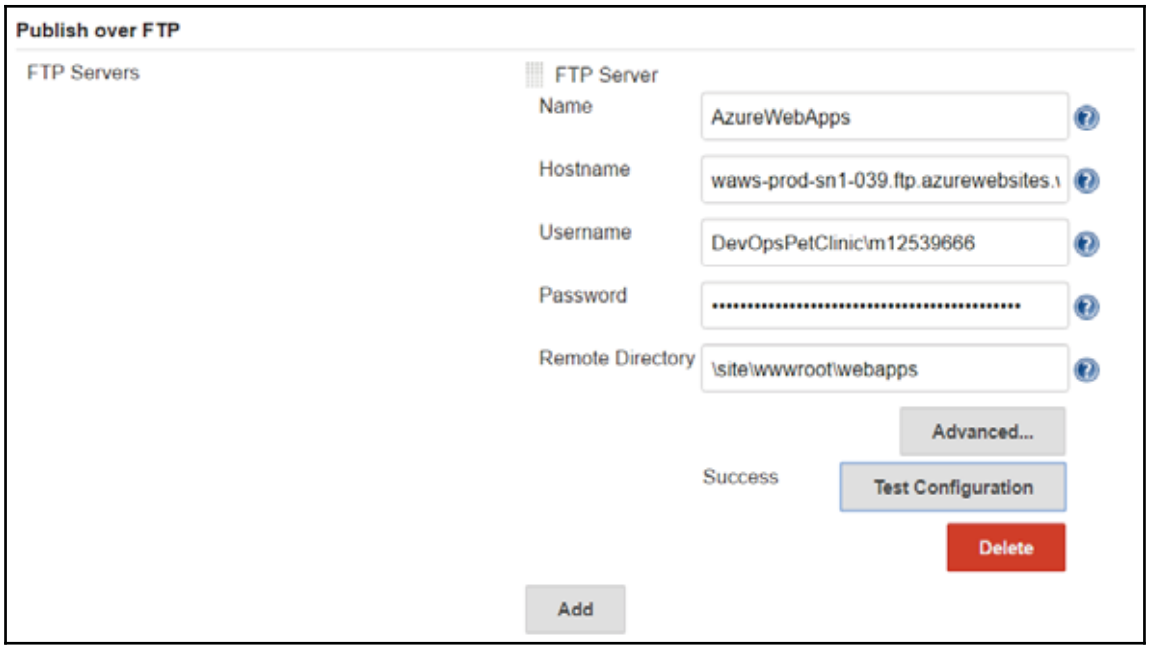


8. Click on **All Settings** and go to **Deployment credentials** in the **PUBLISHING** section. Provide a username and password, and save your changes:



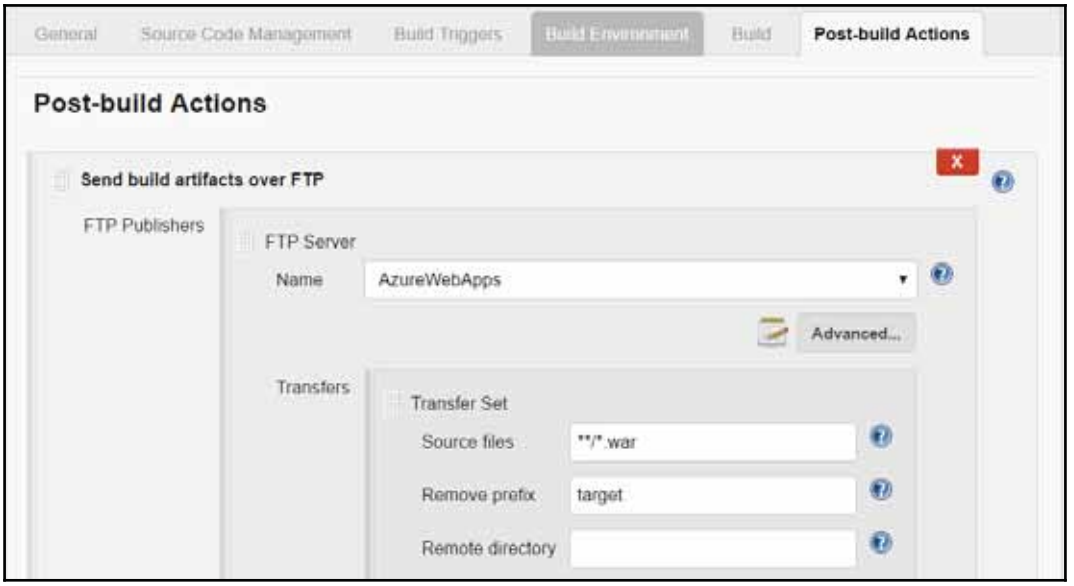
Let's install the Publish Over FTP plugin in Jenkins. We will use the Azure Web App's FTP details to publish the PetClinic WAR file. Let's go to the Jenkins dashboard:

1. Click on **New** Item and select **Freestyle** project.
2. In Jenkins, go to **Manage Jenkins** and click on **Configure** | **Configure FTP settings**. Provide a **Hostname**, **Username**, and **Password**, which are available in the Azure portal.
3. Go to `www.devopspetclinic.scm.azurewebsites.net` and download the Kudu console. Navigate to the different options and find the site directory and `webapps` directory.
4. Click on **Test Configuration** and, once you get a **Success** message, you are ready to deploy the PetClinic application:

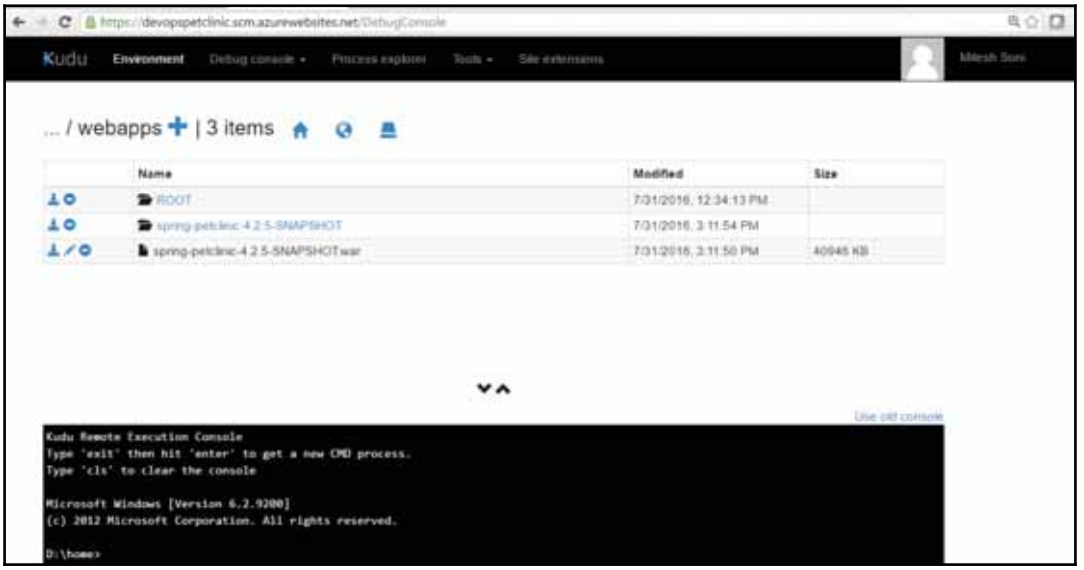


5. In the build job we created, go to the **Build** section and configure **Copy artifacts from another project**. We will copy the WAR file to a specific location on a virtual machine.
6. In **Post-build Actions**, click on **Send build artifacts over FTP**. Select the **FTP Server Name** configured in Jenkins. Configure **Source files** and the **Remove** prefix accordingly for deployment of an Azure Web App.

7. Select **Verbose output** in the console:



8. Click on **Build** now and see what happens behind the scenes.
9. Go to the Kudu console, click on **DebugConsole**, and go to **Powershell**. Go to **site** | **wwwroot** | **webapps**. Check whether the WAR file has been copied:



Now we have an application deployed on Azure Web Apps.

Summary

In this chapter, we have covered definitions of Continuous Delivery and Continuous Deployment. We have seen different approaches to application package deployment, such as application deployment in a local Tomcat server, and a Tomcat server available in Infrastructure as a Service (Amazon EC2), and Platform as a Service (AWS Elastic Beanstalk, Microsoft Azure App Services).

In the next chapter, we will discuss in detail how to perform different types of testing, such as functional testing using Selenium and load testing with Apache JMeter, to implement continuous testing.

6

Continuous Testing - Functional and Load Testing with Jenkins

Continuous Testing is one of the most important DevOps practices available for the End to End Automation of application lifecycle management.

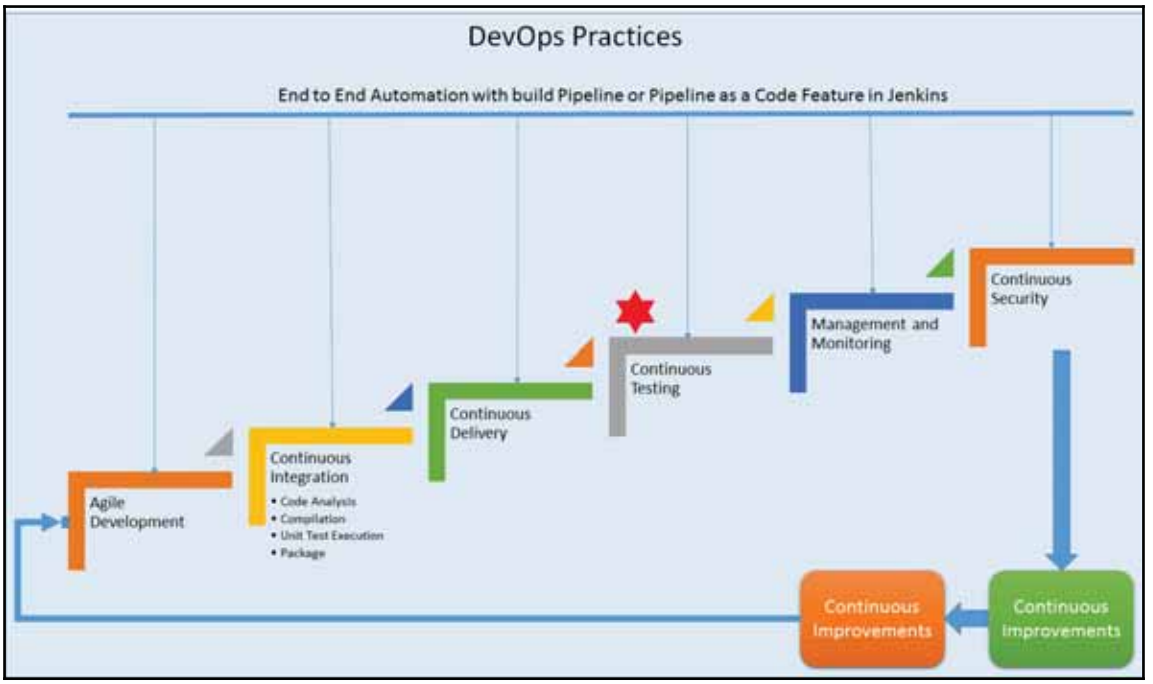
It not only considers automation, but it also includes aspects such as culture change and tools. It is essential to integrate automated tests into application lifecycle management early, to test quickly and in a timely manner, and to repeat the test execution process efficiently.

This chapter will give insights into how functional testing and load testing can be performed and how they can be integrated with Jenkins to adopt Continuous Testing practices as part of a DevOps culture.

This is not the whole picture of Continuous Testing, but it will certainly give a glimpse of how to use Continuous Testing DevOps practices to change the existing culture using automated tests. In this chapter, we will cover following topics:

- Functional testing with Selenium
- Jenkins and Selenium integration
- Load testing with Apache JMeter
- Jenkins and Apache JMeter integration

In this chapter, we will cover Continuous Testing practices as a part of our DevOps journey:



At the end of this chapter, we will know how to perform functional testing and load testing on the deployed application.

Functional testing with Selenium

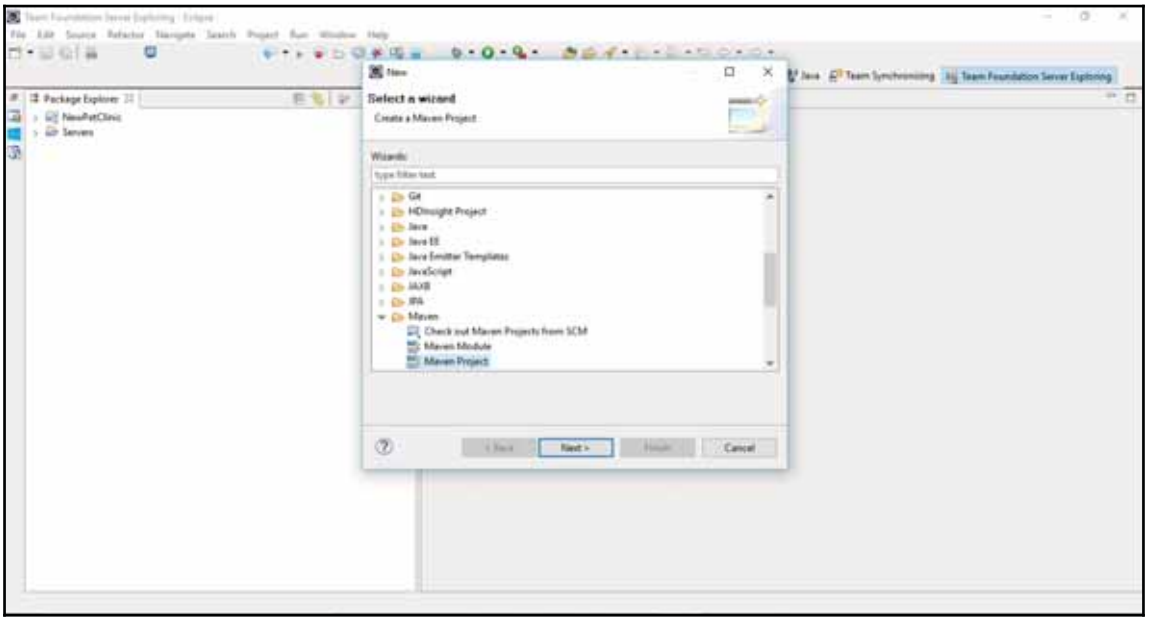
In this chapter, we will use Selenium and Eclipse for a functional test case execution. We have already deployed the application in the Tomcat, so we can perform functional tests and load tests on that deployment.

Let's go step by step through creating a sample functional test case and then executing it using Jenkins.

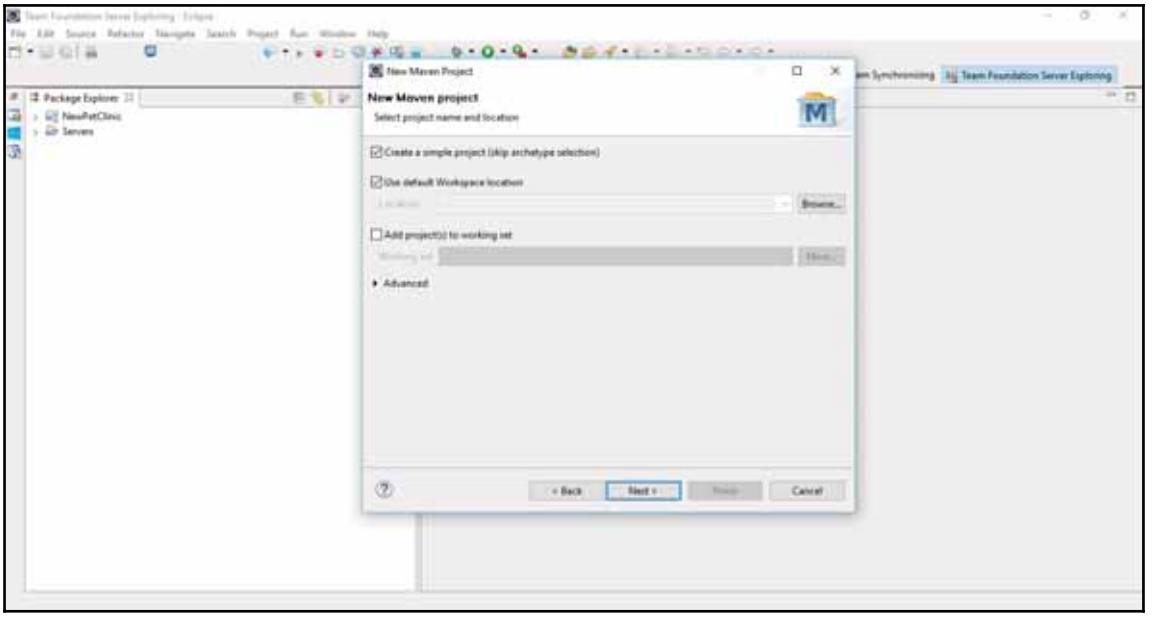
The PetClinic project is a Maven-based Spring application, and we will create a test case using Eclipse and Maven. We will utilize the `m2eclipse` plugin in Eclipse.

We have installed Eclipse Java EE IDE for Web Developers, Version: Mars.2 Release (4.5.2), Build ID: 20160218-0600, so lets start!

1. Go to the Eclipse marketplace and install the **Maven Integration for Eclipse** plugin.
2. Create a **Maven project** using a wizard in the Eclipse IDE:



3. Select **Create a simple project (skip archetype selection)** and click on **Next**:

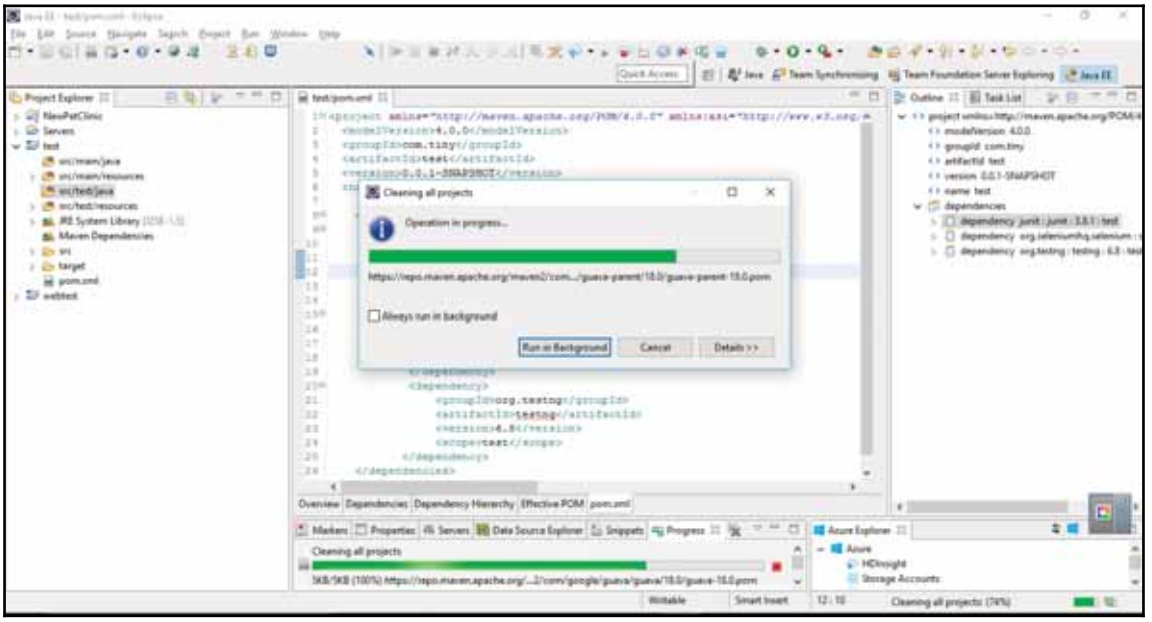


4. Go through the wizard and create a project. It will take some time to create a project in Eclipse. Provide **Artifact**, **Version**, **Packaging**, **Name**, and **Description**. Click on **Finish**.
5. Wait until the Maven project is created and configured. Make sure that Maven is installed and configured properly. If Maven is behind a proxy, configure the proxy details in `conf.xml`, available in the Maven directory.
6. In `Pom.xml`, we need to add Maven, Selenium, TestNG, and JUnit dependencies in the `<project>` node. The following is a modified `Pom.xml`:

```
<project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.tiny</groupId>
<artifactId>test</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>test</name>
<build>
<plugins>
```

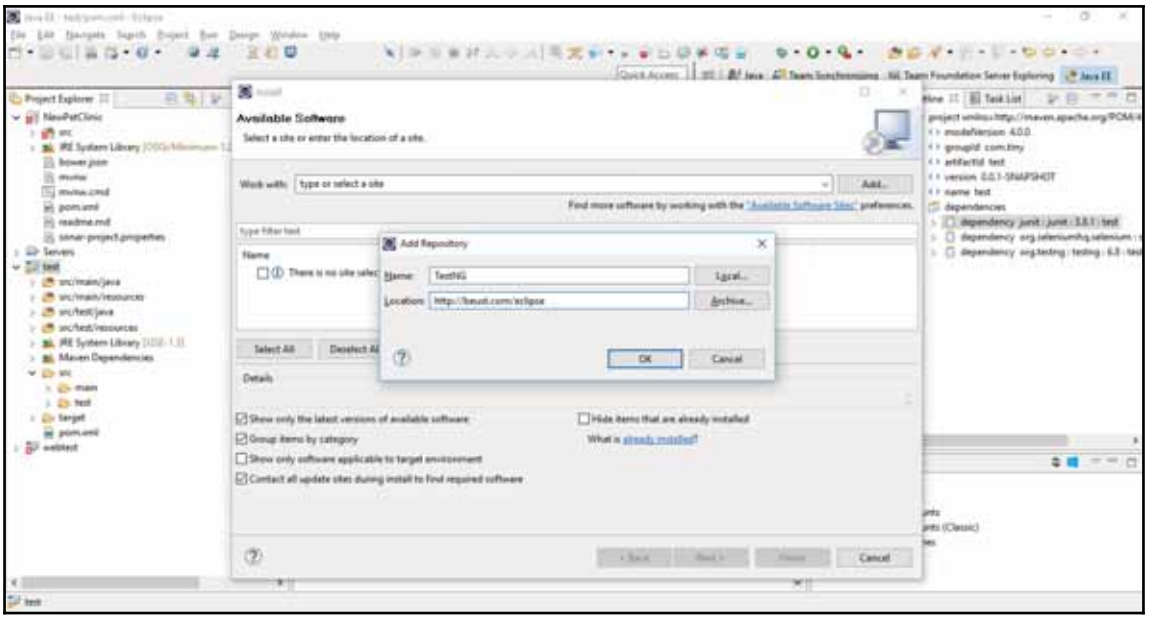
```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.6.1</version>
<configuration>
<source>1.8</source>
<target>1.8</target>
</configuration>
</plugin>
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-surefire-plugin</artifactId>
<version>2.19.1</version>
<configuration>
<suiteXmlFiles>
<suiteXmlFile>testng.xml</suiteXmlFile>
</suiteXmlFiles>
</configuration>
</plugin>
</plugins>
</build>
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>3.0.1</version>
</dependency>
<dependency>
<groupId>org.testng</groupId>
<artifactId>testng</artifactId>
<version>6.8</version>
<scope>test</scope>
</dependency>
</dependencies>
</project>
```

7. Save `pom.xml` after adding these changes and build the project again from the **Project** menu. It will download new dependencies:

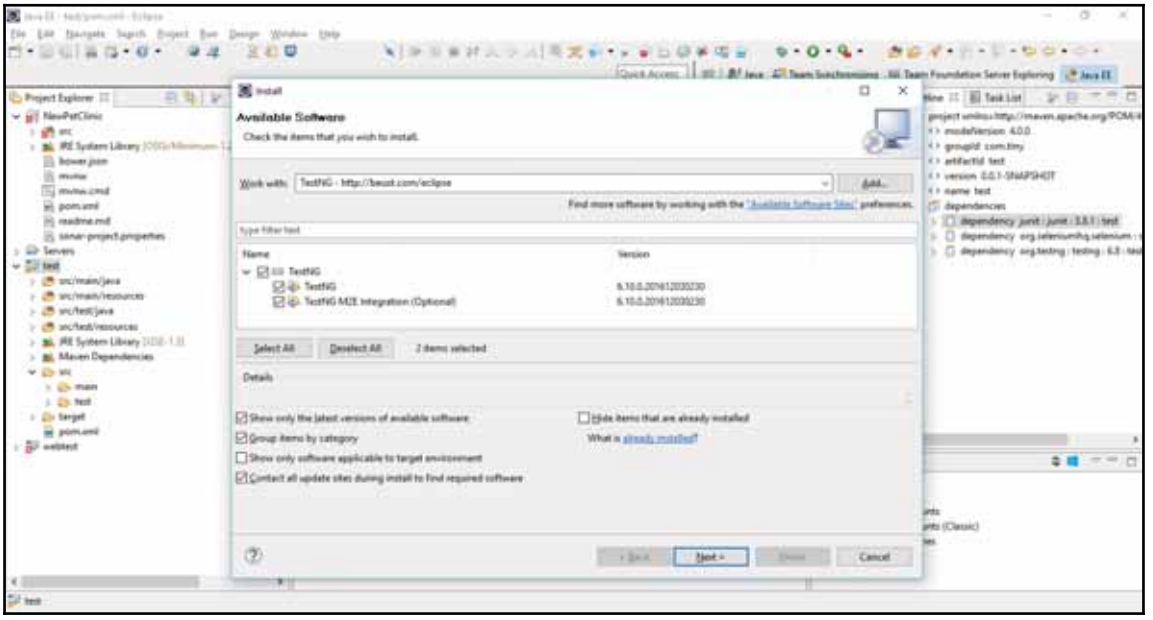


Create a Maven Project in Eclipse IDE

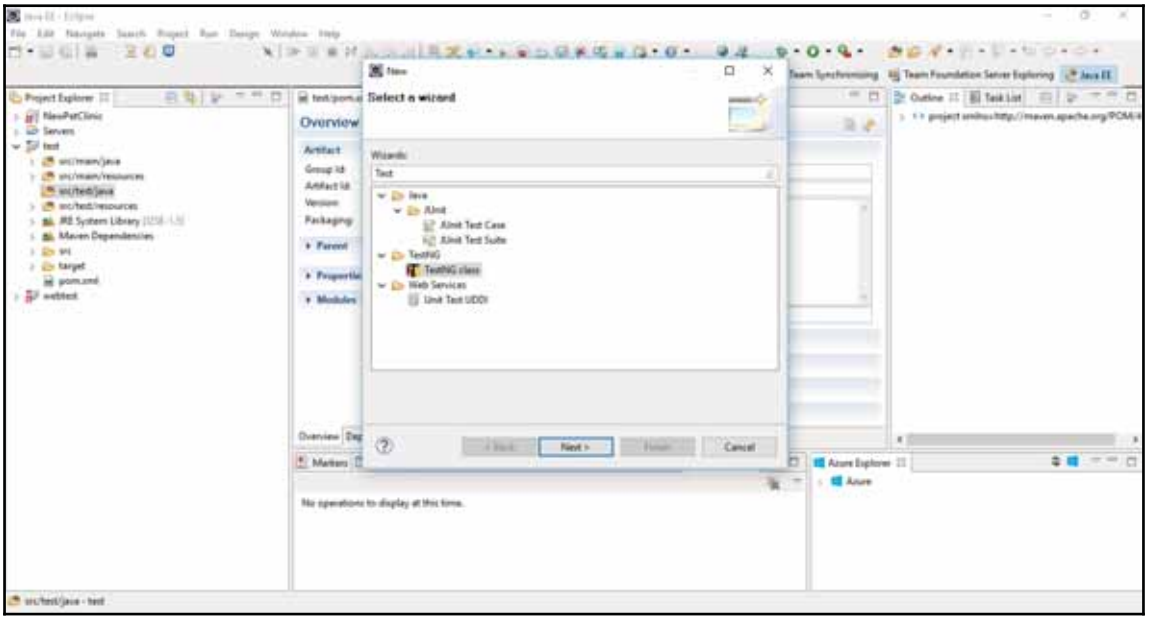
8. Click on the **Details** button of the dialog box to verify the operation in progress.
9. The next task is to write the `TestNG` class. Install the `TestNG` plugin. Go to **Help** and click on **Install New Software** and **Add Repository**:



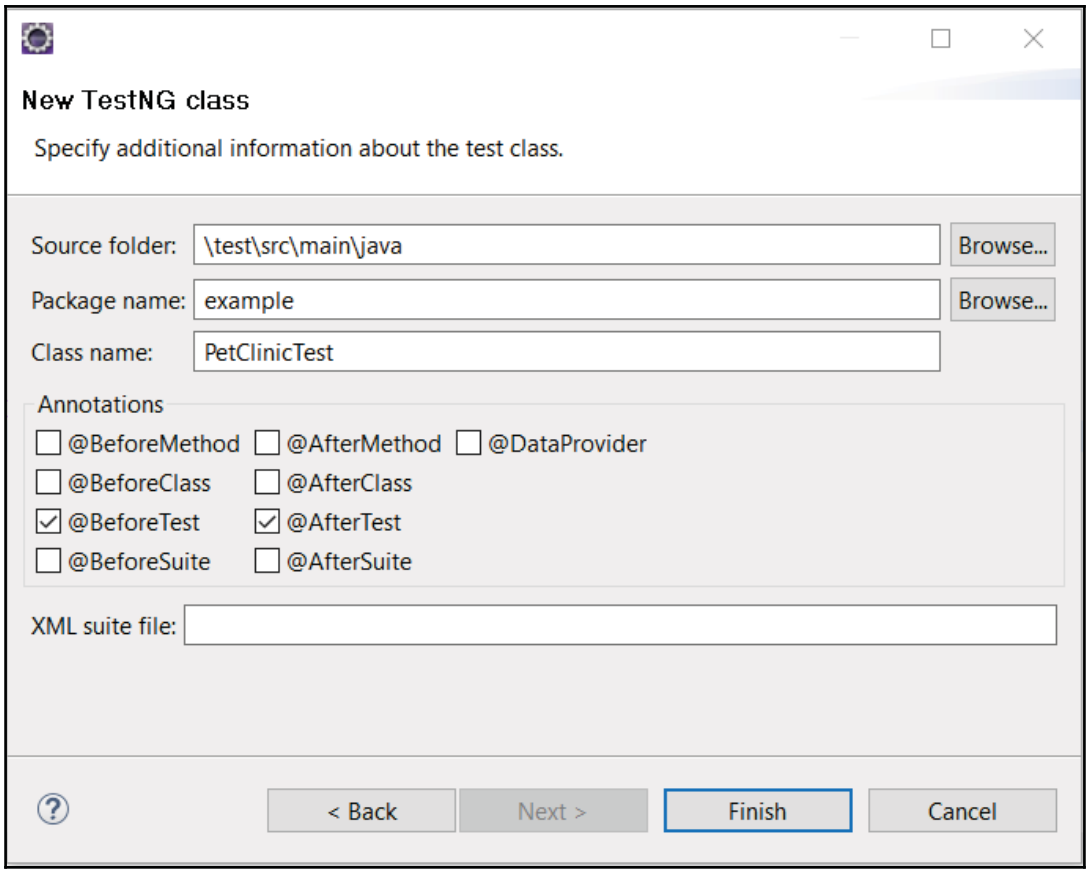
10. Select the items we need to install:



11. Review all the items that need to be installed and click on **Next**.
12. Accept the license and click on **Finish**.
13. Verify the installation progress in Eclipse.
14. Now let's create a **TestNG class**:

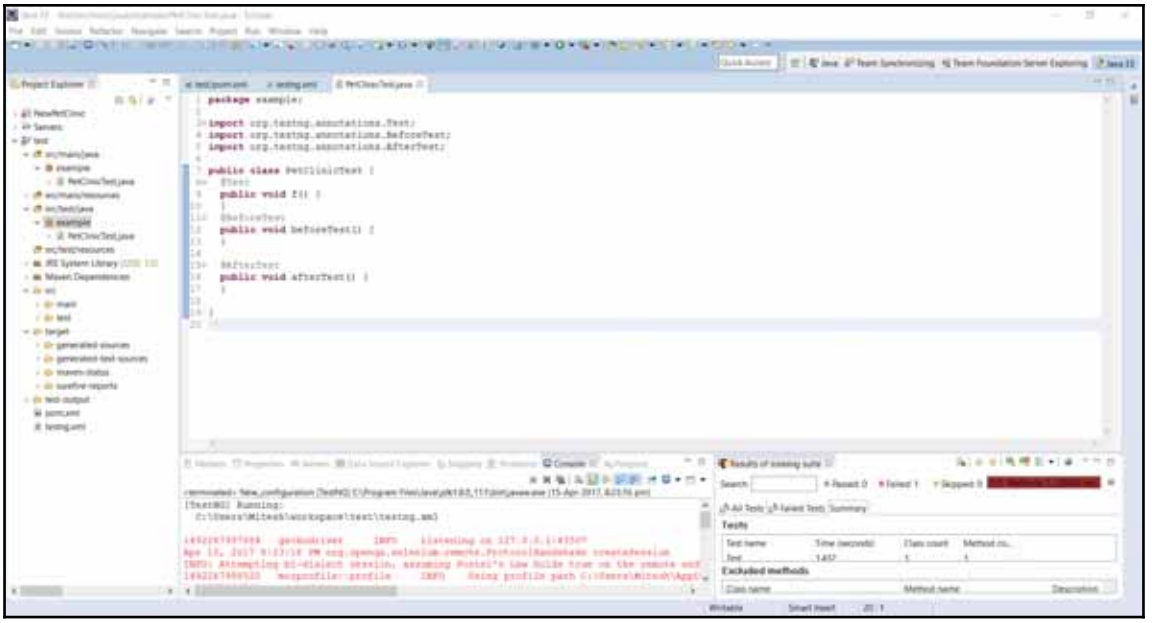


15. Provide a class name:

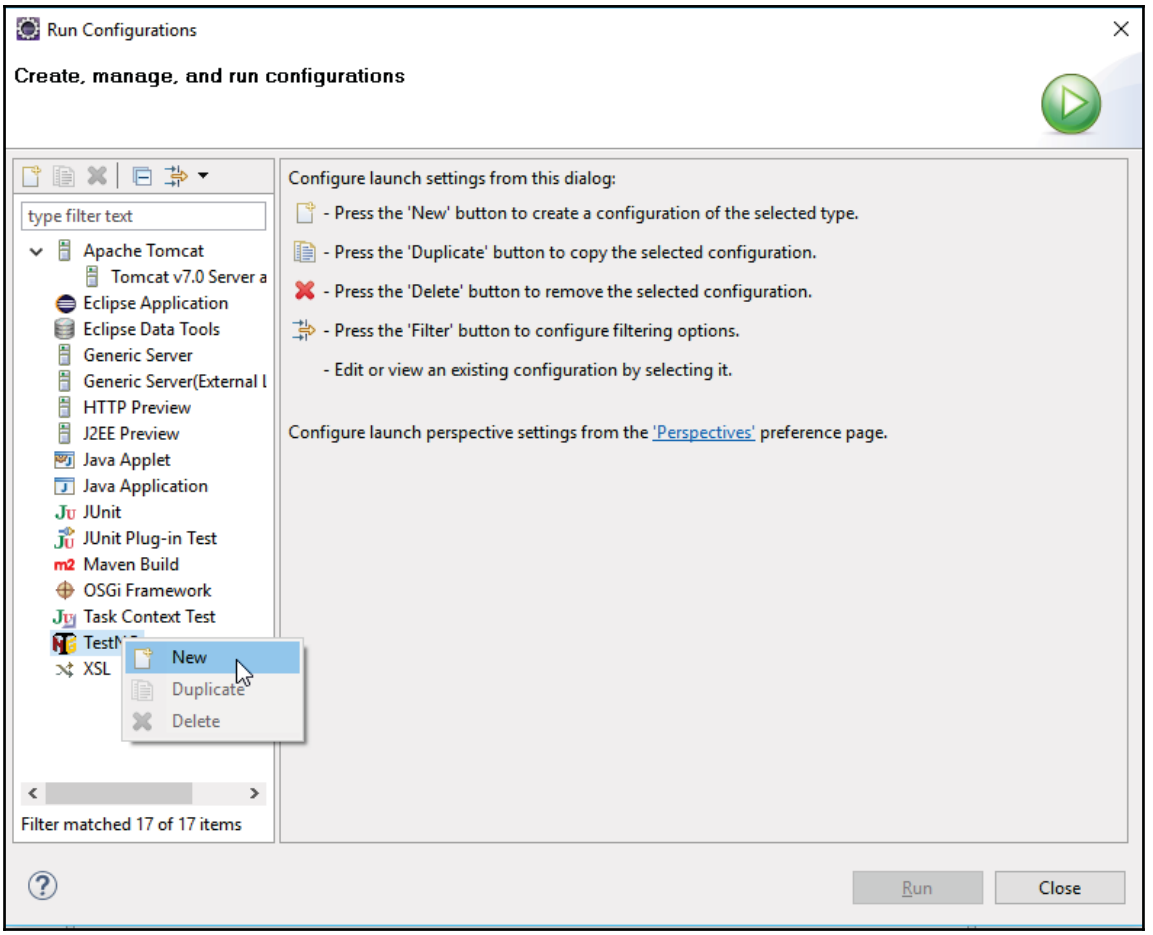


16. Give a package name and click on **Finish**.

17. The newly created class will look like the following screenshot:

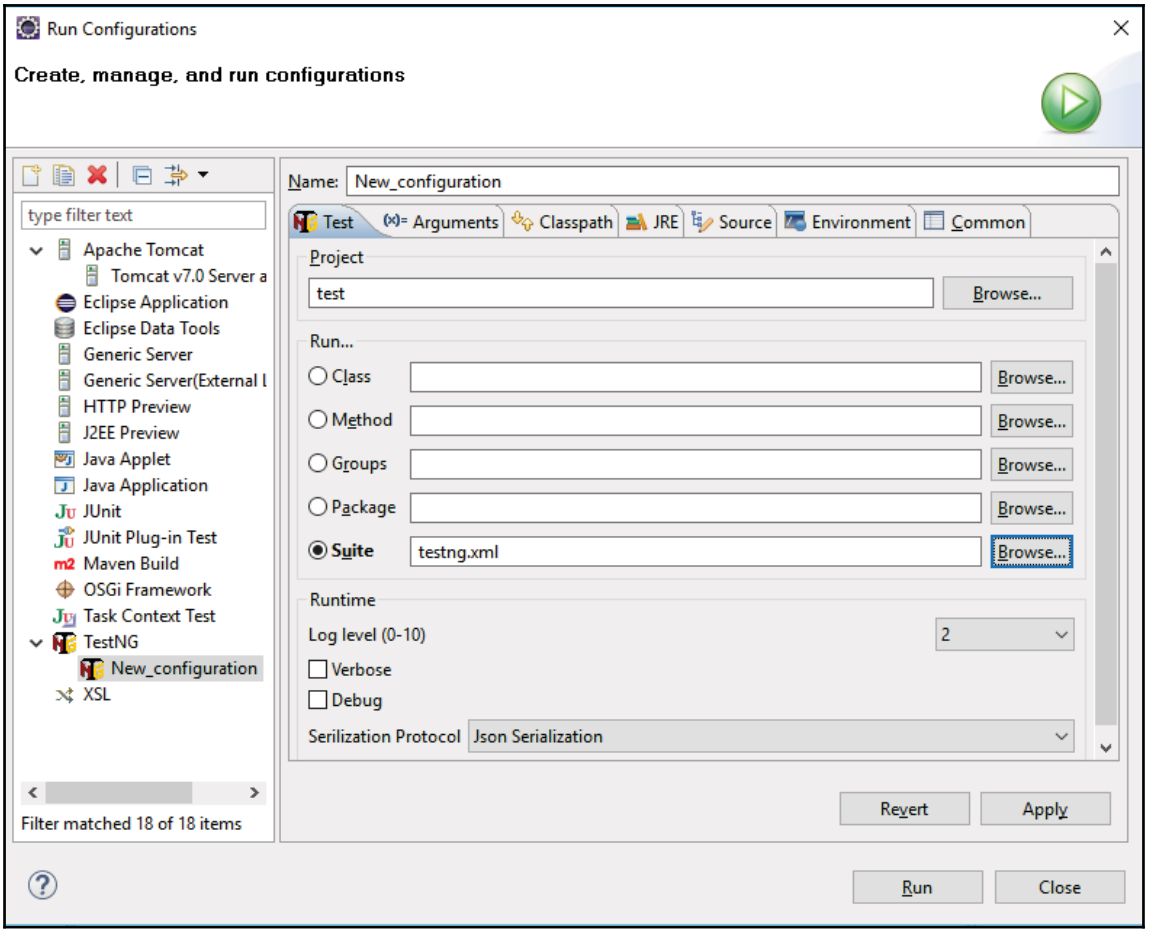


18. Right-click on the test file and click on **TestNG**, convert to TestNG.
19. This will create a `testing.xml` file that has details about the test suite.
20. Right-click on project and click on **Run Configurations**.
21. Right-click on **TestNG** and click on **New**:



22. Provide the project name and select `testing.xml` in the suite.
23. Click **OK** and **Apply**.

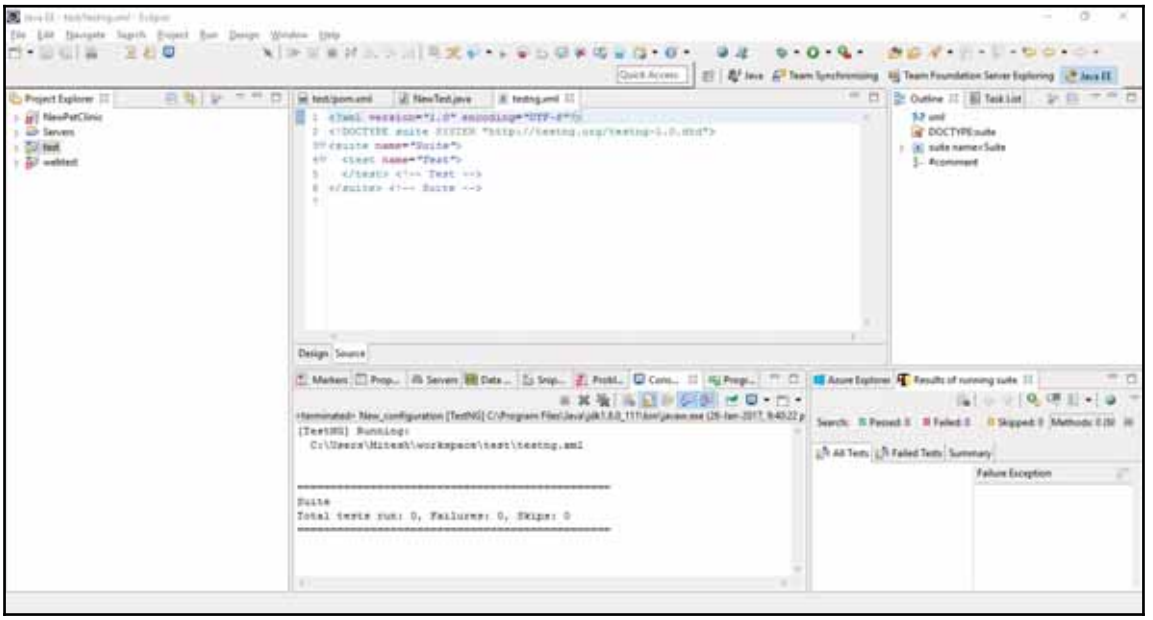
24. Click on **Run**:



25. If Windows Firewall blocks it, then click on **Allow Access**.
26. There is no configuration available in `testing.xml` for execution, so even if Maven execution runs successfully, no suite will be executed.

27. Generate the `TestNG` class under the `test` folder. Select location, `suite name`, and `class name`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite">
<test name="Test">
<classes>
<class name="example.PetClinicTest"/>
</classes>
</test><!-- Test -->
</suite><!-- Suite -->
```

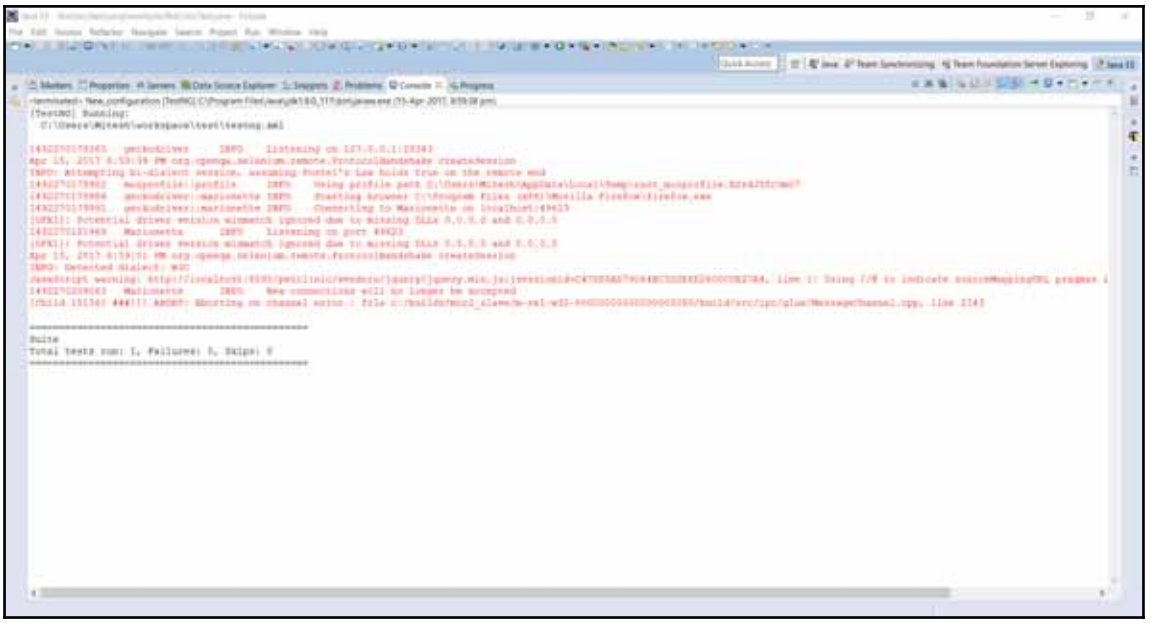


28. Go to `https://github.com/mozilla/geckodriver/releases`and download a version.
29. Extract the file available in the downloaded ZIP file, based on the system configuration you have. In our case, we have downloaded `geckodriver-v0.13.0-win64`.
30. Click on it and verify the driver details.

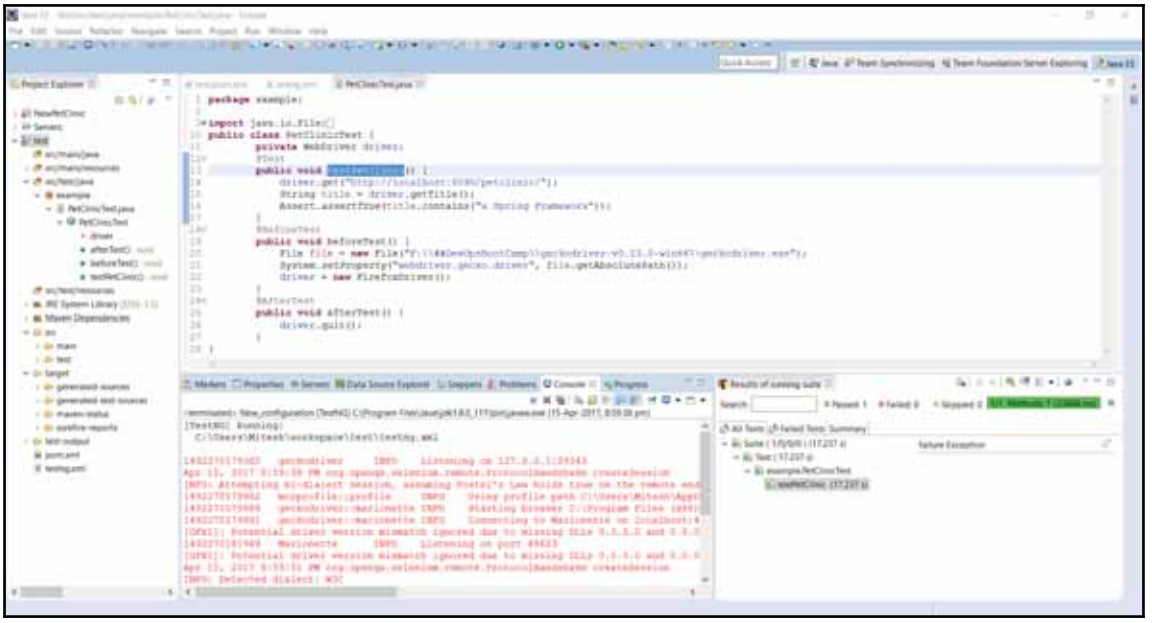
31. Let's write some code as well. It will check whether the title of the web page contains a specific string or not. The result or the outcome of the following code is based on the title of the page. If it contains a given string, then the test case will pass; otherwise, it will fail. Here is an example package:

```
import java.io.File;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.testng.Assert;
import org.testng.annotations.Test;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.AfterTest;
public class PetClinicTest {
  private WebDriver driver;
    @Test
    public void testPetClinic() {
//Change the URL based on the location where Tomcat is installed
and application is deployed
driver.get("http://localhost:8090/petclinic/");
      String title = driver.getTitle();
Assert.assertTrue(title.contains("a Spring
 Frameworkk"));
    }
    @BeforeTest
    public void beforeTest() {
      File file = new
// We have used Firefox for testing; change this driver based on
requirements and location too
 File("F:\\##JenkinsEssentials\\geckodriver-v0.13.0-
  win64\\geckodriver.exe");
 System.setProperty("webdriver.gecko.driver",
 file.getAbsolutePath());
      driver = new FirefoxDriver();
    }
    @AfterTest
    public void afterTest() {
driver.quit();
    }
}
```

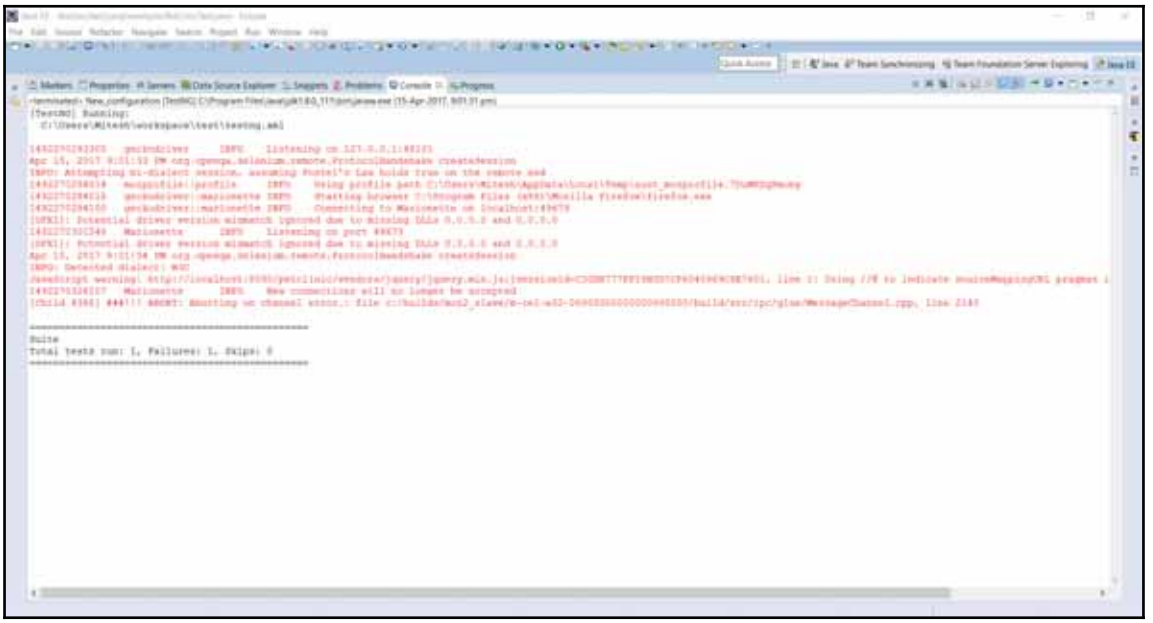
32. Let's run the Maven test again from Eclipse.
33. The following is the output when the test case is executed successfully:



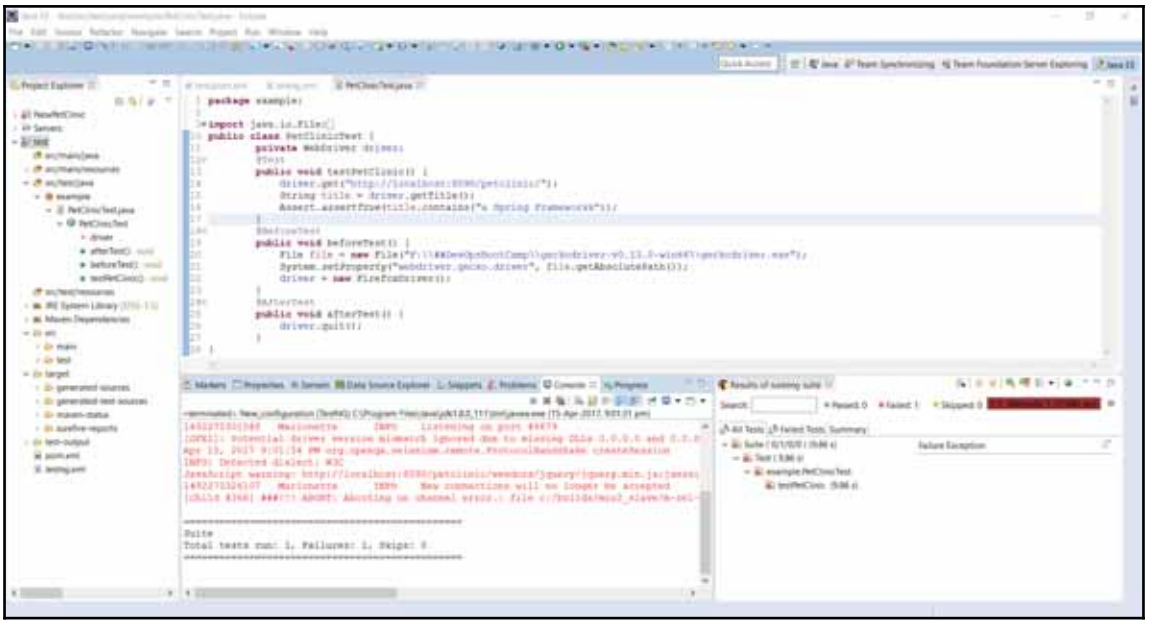
34. Check the **All Tests** tab in the **Results** of the running suite section in Eclipse. We can see successful execution here:



35. Check the **Failed Tests** tab in the **Results** of the running suite section in Eclipse.
36. Check the **Summary** tab in the **Results** of the running suite section in Eclipse in the successful scenario.
37. In the code, change the text available for title comparison so the test case fails.
38. Verify the output in Console:



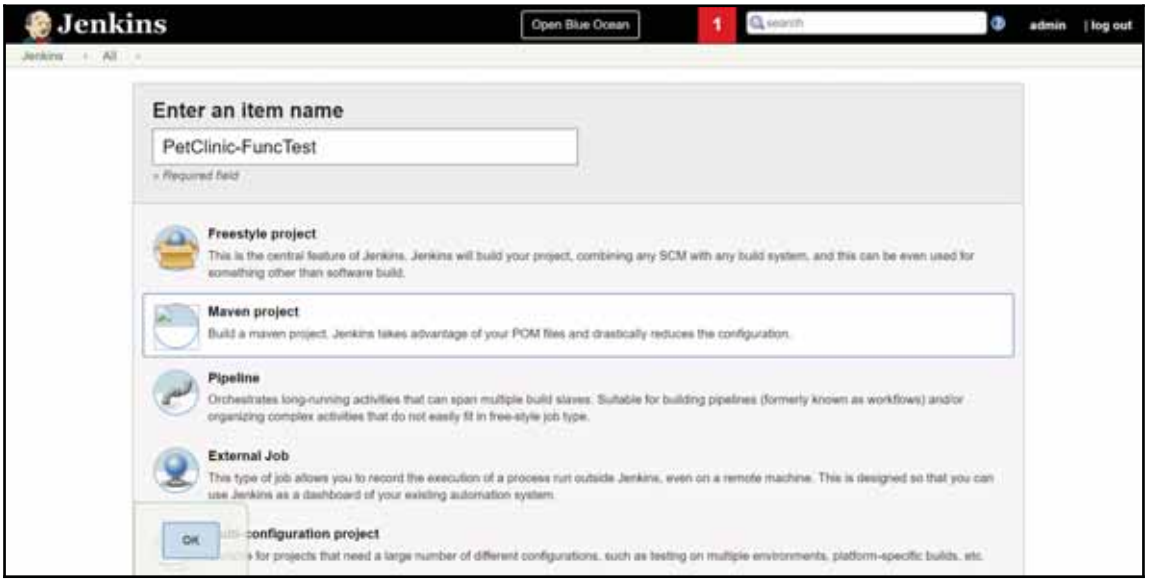
39. Check the **All Tests** tab in the **Results** of running suite section in Eclipse and notice the failure icon.



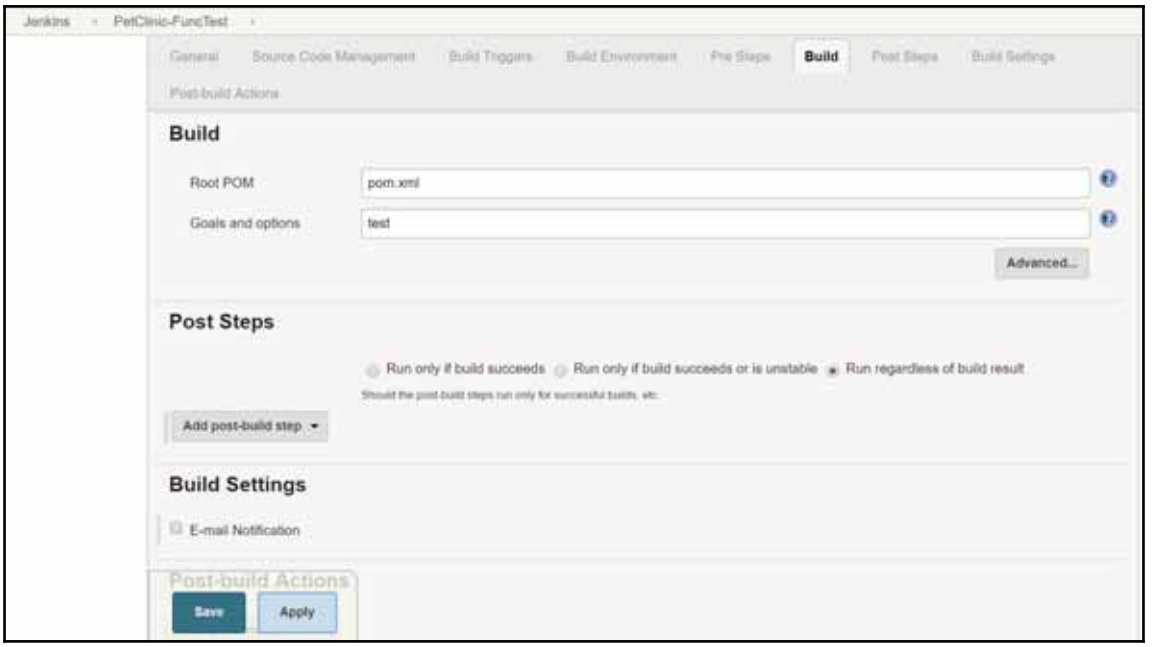
40. Observe the **Failed Tests** tab in the **Results** of the running suite section in Eclipse.
41. Click on `testPetClinic` and verify the **Failure Exception**.
42. Check the **Summary** tab in the **Results** of the running suite section in Eclipse.

So, we have created a sample test case based on Selenium to verify the title of the PetClinic home page.

Now let's try to execute the same thing from Jenkins:



1. Check in the **Test Project** in **Repository**. Create a `PetClinic-FuncTest freestyle` job in Jenkins.
2. In the **Build** section, provide the **Root POM** location and **Goals and options** to execute:



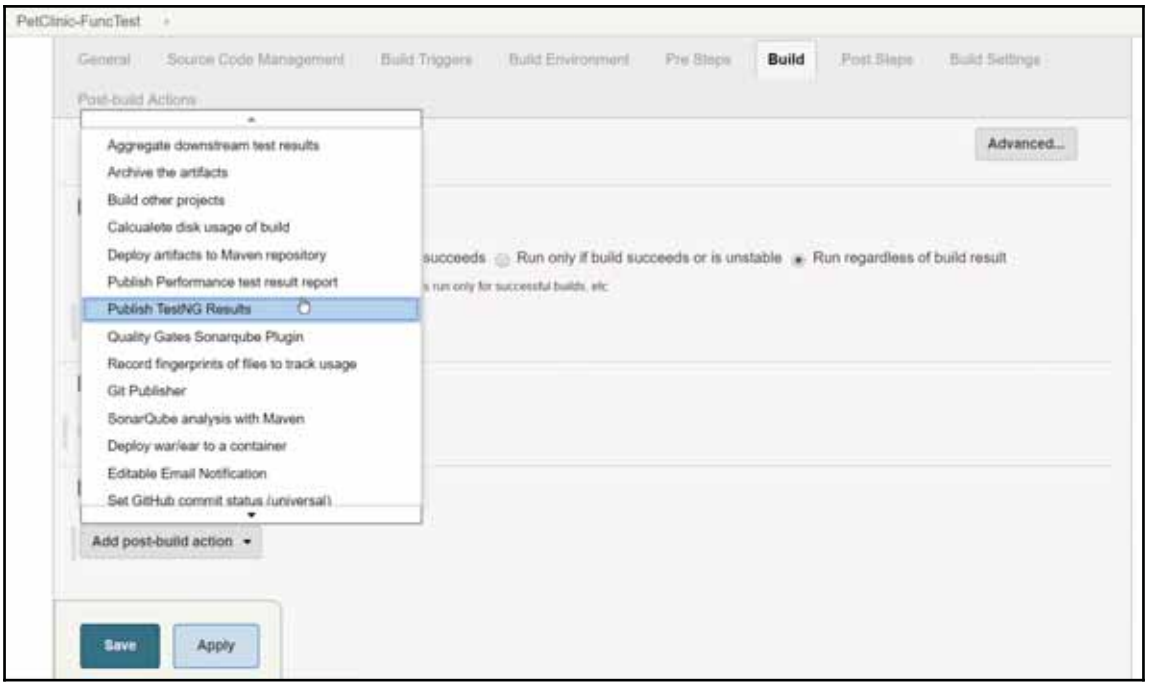
3. Save the build job and click on **Build now**.
4. Verify the execution of the build job in the Console output.

5. This will open a Mozilla Firefox window and open the URL that is given in the code. This requires our PetClinic application to be deployed on a web server and be running without any issues:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running TestSuite
1496574254522   geckodriver     INFO    Listening on 127.0.0.1:6486
Jun 04, 2017 4:34:14 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Attempting bi-dialect session, assuming Postel's Law holds true on the remote end
1496574255134   mozprofile::profile     INFO    Using profile path
C:\Users\Mitesh\AppData\Local\Temp\rust_mozprofile.t3uOSiy56Onn
1496574255138   geckodriver::marionette INFO    Starting browser C:\Program Files (x86)\Mozilla Firefox\firefox.exe
1496574255175   geckodriver::marionette INFO    Connecting to Marionette on localhost:60430
[GFX1]: Potential driver version mismatch ignored due to missing DLLs 0.0.0.0 and 0.0.0.0
[GFX1]: Potential driver version mismatch ignored due to missing DLLs 0.0.0.0 and 0.0.0.0
1496574288578   Marionette      INFO    Listening on port 60430
Jun 04, 2017 4:34:49 PM org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: W3C
JavaScript warning:
http://localhost:8090/petclinic/vendors/jquery/jquery.min.js;jsessionid=884F4251137D820A5723530BAA688915, line 1:
Using //@ to indicate sourceMappingURL pragmas is deprecated. Use //# instead
>>>>>>PetClinic :: a Spring Framework demonstration
1496574305189   Marionette      INFO    New connections will no longer be accepted
Jun 04, 2017 4:35:12 PM org.openqa.selenium.os.UnixProcess destroy
SEVERE: Unable to kill process with PID 10376
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 69.499 sec - in TestSuite
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

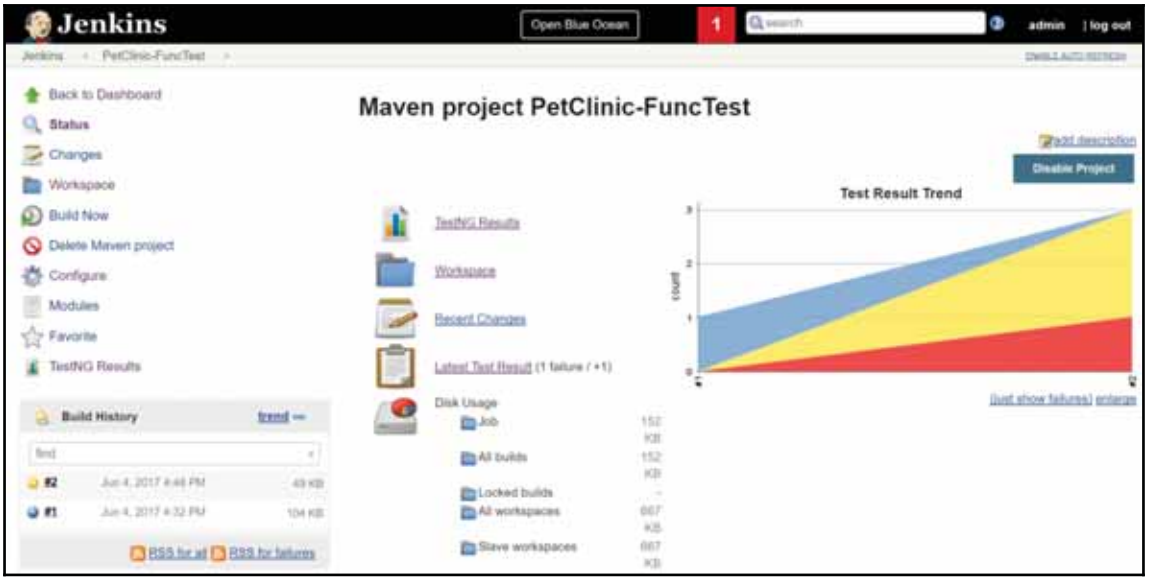
6. Install **TestNG Results Plugin**:

7. Go to **Post build Actions** and select **Publish TestNG Results**:

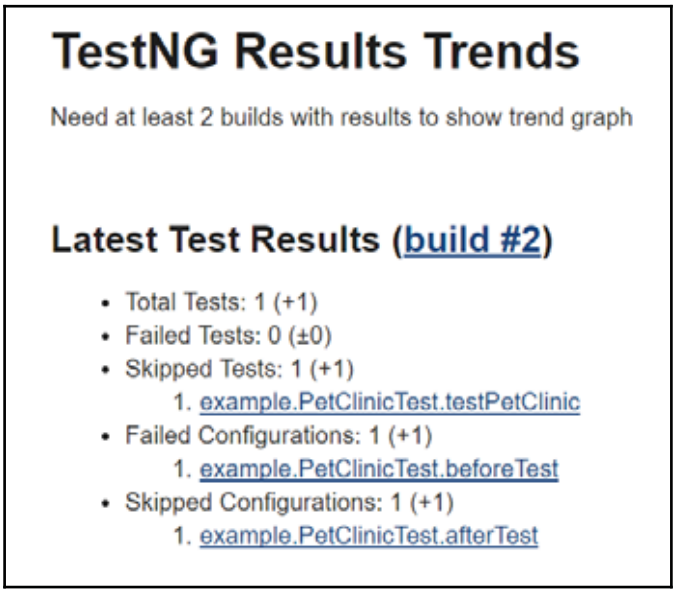


8. Provide **TestNG XML report pattern**:

9. Click on **Build now**:



10. Go to the **Project** dashboard and verify the graphs for TestNG results:



We have seen how to execute Selenium-based test cases in Jenkins. In the next section, we will see how to execute a load test using Jenkins.

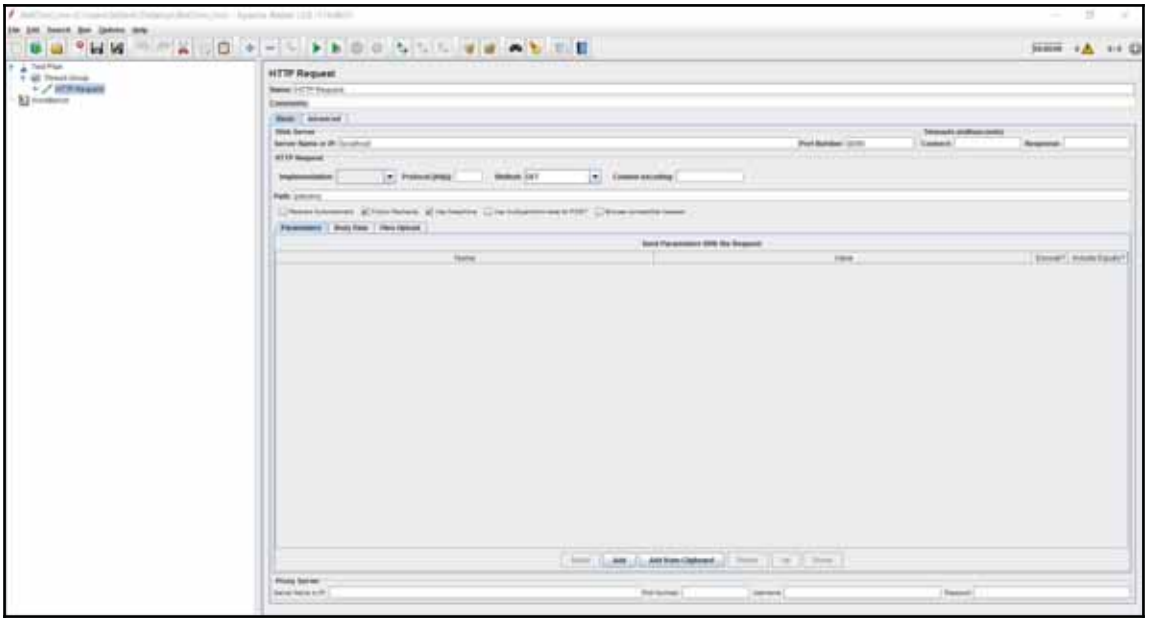
Load testing with Apache JMeter

Apache JMeter is an open source Apache project. It is a pure Java application. Apache JMeter is used to load test, in order to analyze and measure the performance of services.

Download Apache JMeter from `http://jmeter.apache.org/download_jmeter.cgi`. Extract the files and go to the bin directory. Execute `jmeter.bat` or `jmeter.sh`.

1. Open the Apache JMeter console. **Create a Test Plan**.
2. Right-click on the **Test Plan** and click on **Add**; select **Threads (Users)**.
3. Select **Thread Group**.
4. Provide **Thread Group name**.
5. In **Thread Group** properties, provide **Number of Threads**, **Ramp-up Period**, and **Loop Count**.
6. Right-click on **Thread Group**. Click on **Add**. Click on **Sampler**. Click on **HTTP Request**.
7. In HTTP Request, provide **Server Name or IP**. In our case, it will be localhost or an IP address.
8. Give the **Port Number** where your web server is running.

9. Select the **Get method** and provide a path to the load test:

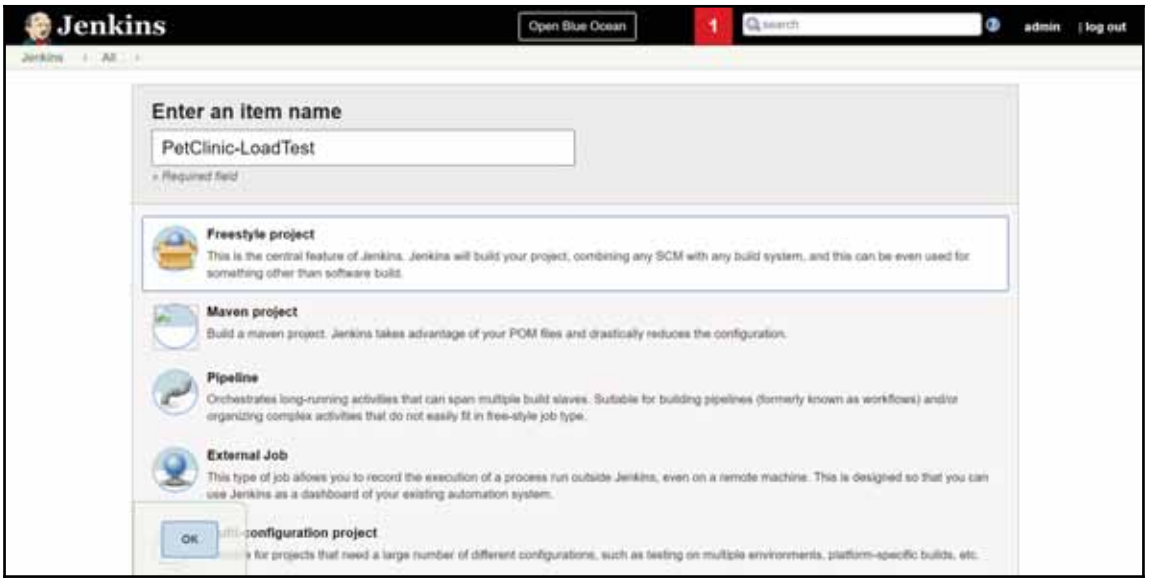


Create an HTTPRequest in APache JMeter to configure Server Name, Port number, and method details

10. Save the `.jmx` file.

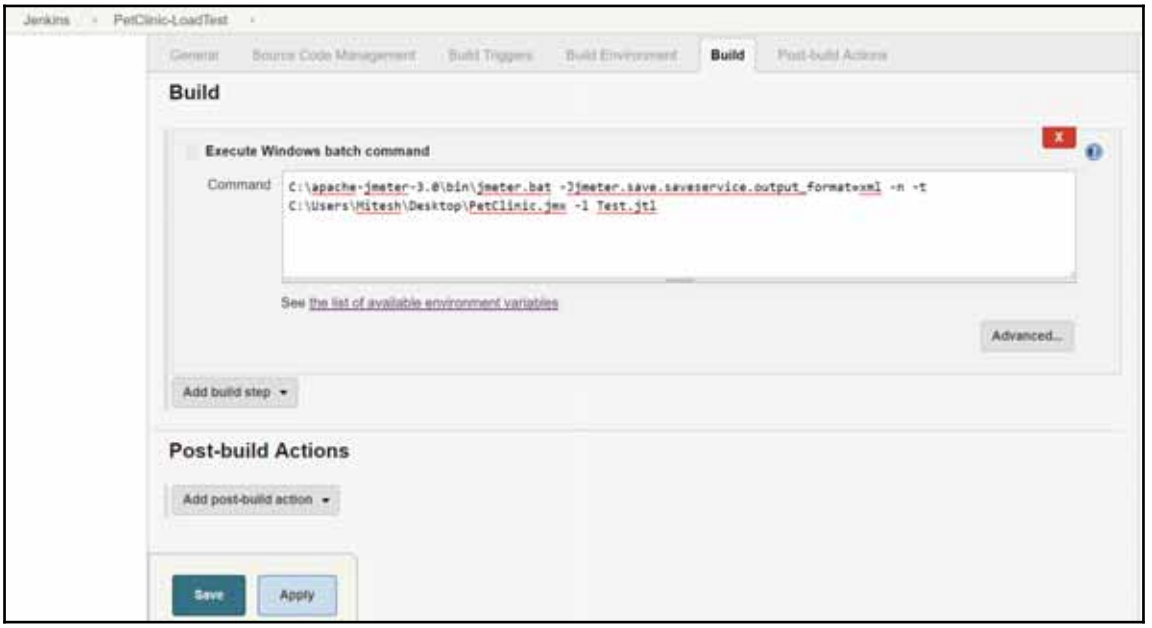
Now let's create a Jenkins job:

1. Create a freestyle job in Jenkins:

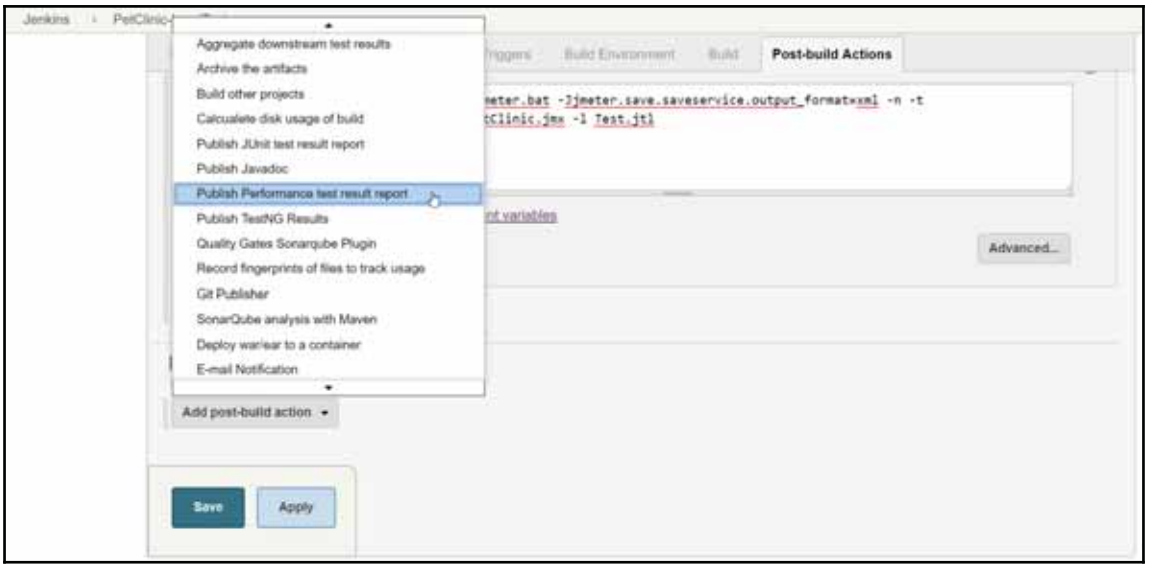


2. Add the Build step Execute Windows batch command. Add the following command. Replace the location of `jmeter.bat` based on the installation directory, and the location of the `.jmx` file too:

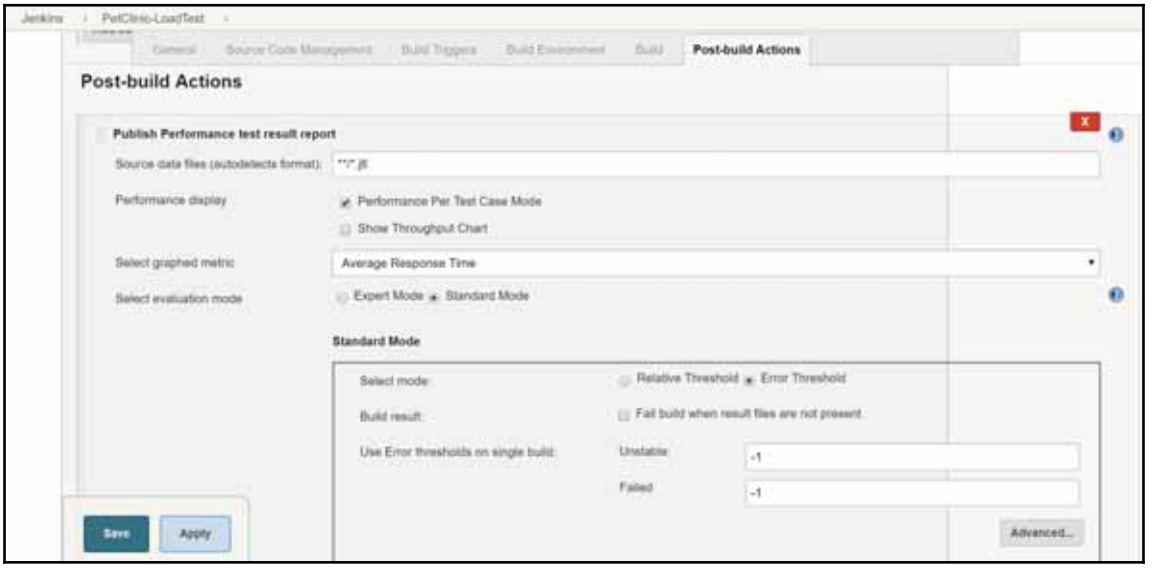
```
C:\apache-jmeter-3.0\bin\jmeter.bat -
Jjmeter.save.saveservice.output_format=xml -n -t
C:\Users\Mitesh\Desktop\PetClinic.jmx -l Test.jtl
```



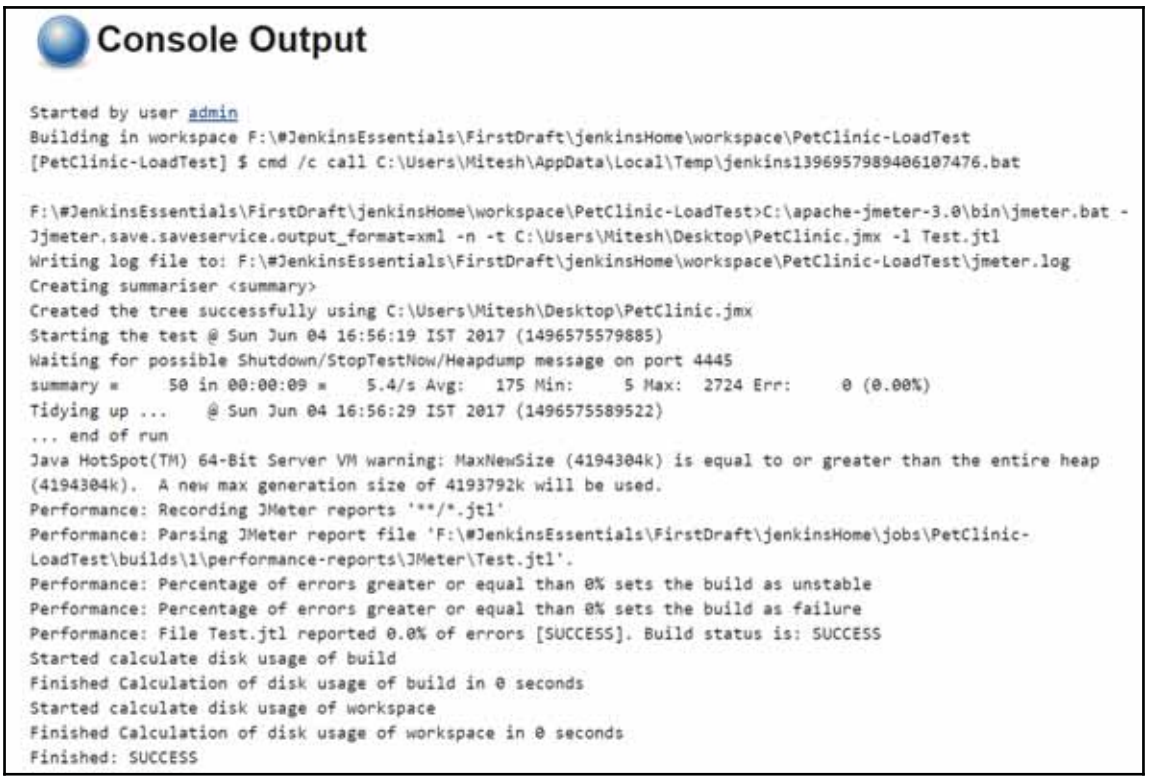
3. Add **Post-build Actions**:



4. Publish **Performance test result report** add `**/*.jtl` file.



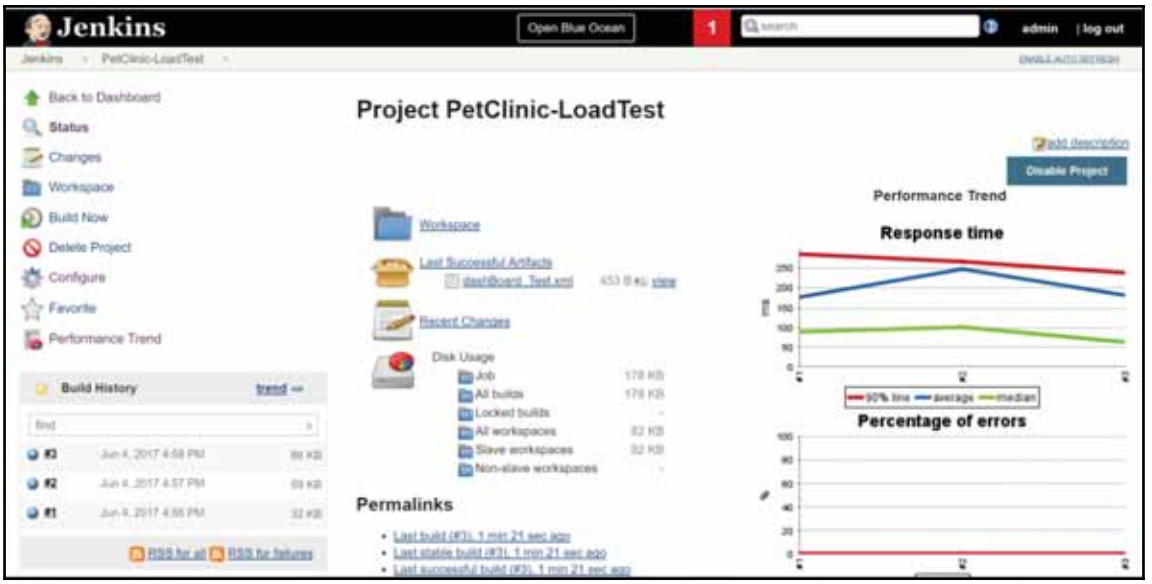
5. Click on **Build now**:



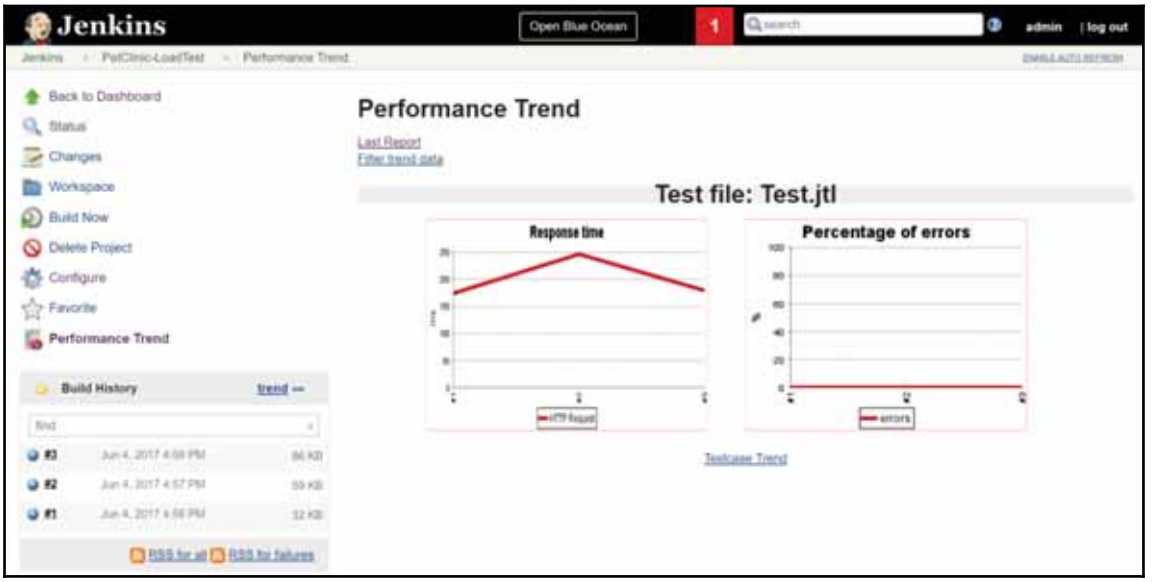
Console Output

```
Started by user admin
Building in workspace F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\PetClinic-LoadTest
[PetClinic-LoadTest] $ cmd /c call C:\Users\Mitesh\AppData\Local\Temp\jenkins1396957989406107476.bat

F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\PetClinic-LoadTest>C:\apache-jmeter-3.0\bin\jmeter.bat -
Jjmeter.save.saveservice.output_format=xml -n -t C:\Users\Mitesh\Desktop\PetClinic.jmx -l Test.jtl
Writing log file to: F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\PetClinic-LoadTest\jmeter.log
Creating summariser <summary>
Created the tree successfully using C:\Users\Mitesh\Desktop\PetClinic.jmx
Starting the test @ Sun Jun 04 16:56:19 IST 2017 (1496575579885)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary =     50 in 00:00:09 =    5.4/s Avg:   175 Min:     5 Max:  2724 Err:     0 (0.00%)
Tidying up ...    @ Sun Jun 04 16:56:29 IST 2017 (1496575589522)
... end of run
Java HotSpot(TM) 64-Bit Server VM warning: MaxNewSize (4194304k) is equal to or greater than the entire heap
(4194304k).  A new max generation size of 4193792k will be used.
Performance: Recording JMeter reports '**/*.jtl'
Performance: Parsing JMeter report file 'F:\#JenkinsEssentials\FirstDraft\jenkinsHome\jobs\PetClinic-
LoadTest\builds\1\performance-reports\JMeter\Test.jtl'.
Performance: Percentage of errors greater or equal than 0% sets the build as unstable
Performance: Percentage of errors greater or equal than 0% sets the build as failure
Performance: File Test.jtl reported 0.0% of errors [SUCCESS]. Build status is: SUCCESS
Started calculate disk usage of build
Finished Calculation of disk usage of build in 0 seconds
Started calculate disk usage of workspace
Finished Calculation of disk usage of workspace in 0 seconds
Finished: SUCCESS
```

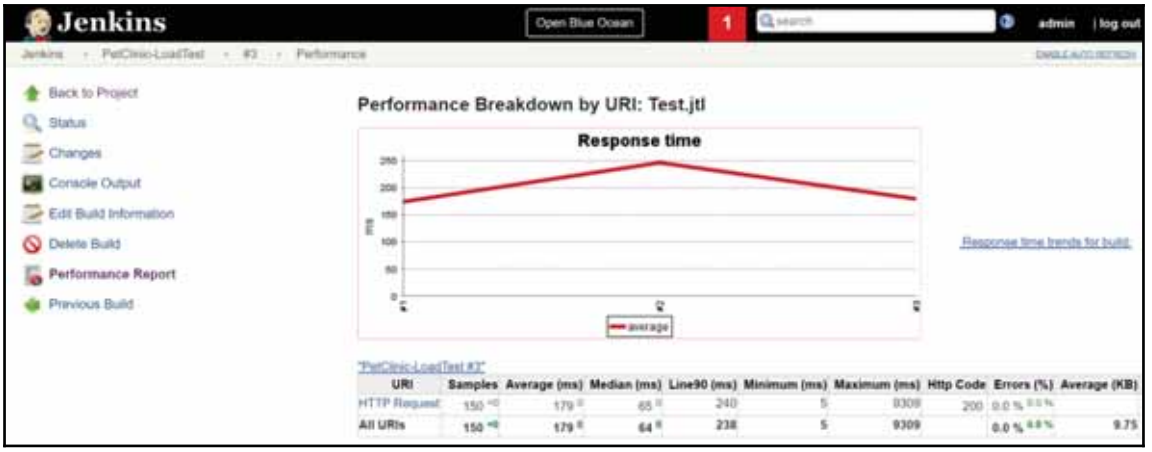
6. Verify **Performance Trend** on the Project dashboard by clicking on the Test results graph.
7. Click on **Performance Trend**:



8. Verify performance breakdown for **Response Time** and **Percentage of errors**:



9. Click on **Last Report** and get more details on the load test results:



Done!

Summary

Finally, we are at the end of the chapter. We have performed functional testing using Selenium and then integrated it with Jenkins. We have also performed load testing using Apache JMeter and then integrated it with Jenkins.

This is useful when we want to achieve automated testing in the pipeline for functional testing and load testing. We can set notifications and other configurations based on the culture of an organization too.

In the next chapter, we will cover how to orchestrate all the build jobs we have created up to now, to create a pipeline. We will create a pipeline using the Build Pipeline plugin and the Jenkins 2 Pipeline feature.

7

Build Pipeline and Pipeline as a Code

Up to now, we have covered all specific tasks that are individual and can work as a stepping stone to performing other steps, such as static code analysis, Continuous Integration, Continuous Delivery, and Continuous Testing.

What if we need to fix the sequence of execution of all such tasks with or without manual intervention?

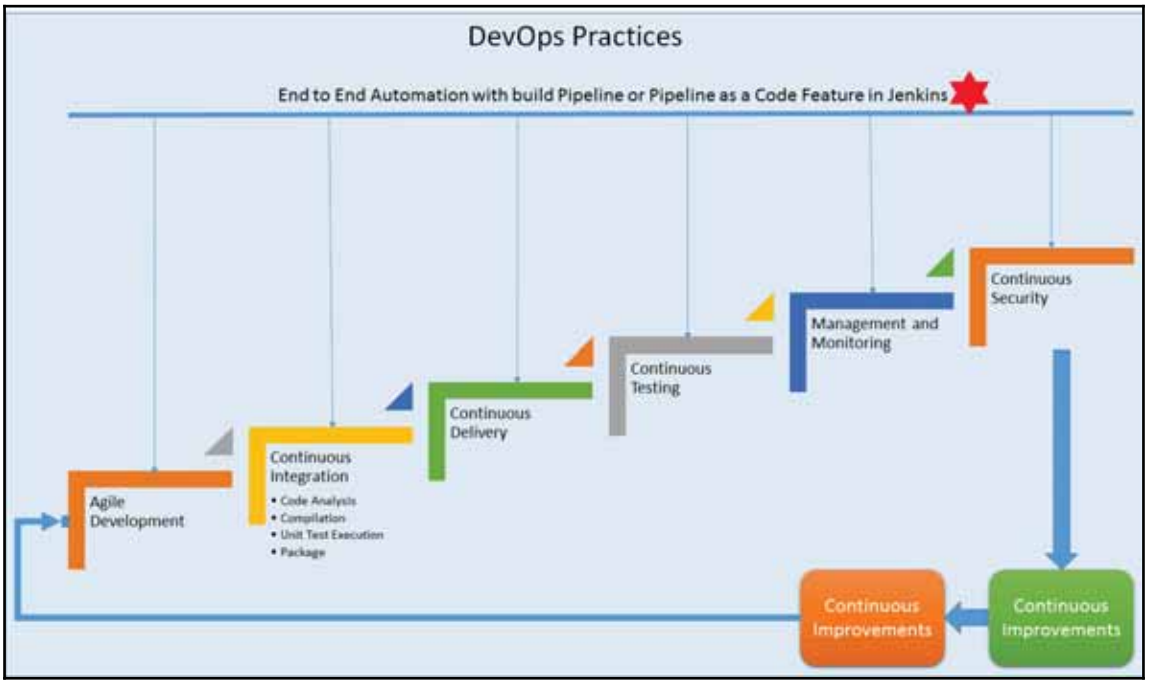
What if we want to create a pipeline where one successful execution of Job can lead to another execution of Job?

This is where we will utilize the Build Pipeline plugin and the Pipeline as a Code feature available in Jenkins 2 for the orchestration of the end-to-end automation of Application Life Cycle Management. We have executed a pipeline and all build jobs on a Windows system. Based on your operating system, there might be some changes that you may need to do and we have mentioned these as comments at specific places in the script or code.

This chapter will cover how to orchestrate a build job to execute it in a specific sequence along with other build jobs. We will cover the Build Pipeline plugin and the Pipeline as a Code feature that is available in Jenkins 2 and later. The following are the major topics that we will cover in this chapter:

- Build Pipeline
- Upstream and downstream jobs
- Overview of Pipeline as a Code
- Pipeline as a Code: implementation
- Promoted builds

In this chapter, we will cover end-to-end automation with the Build Pipeline plugin and the Pipeline as a Code feature as a part of our DevOps journey:



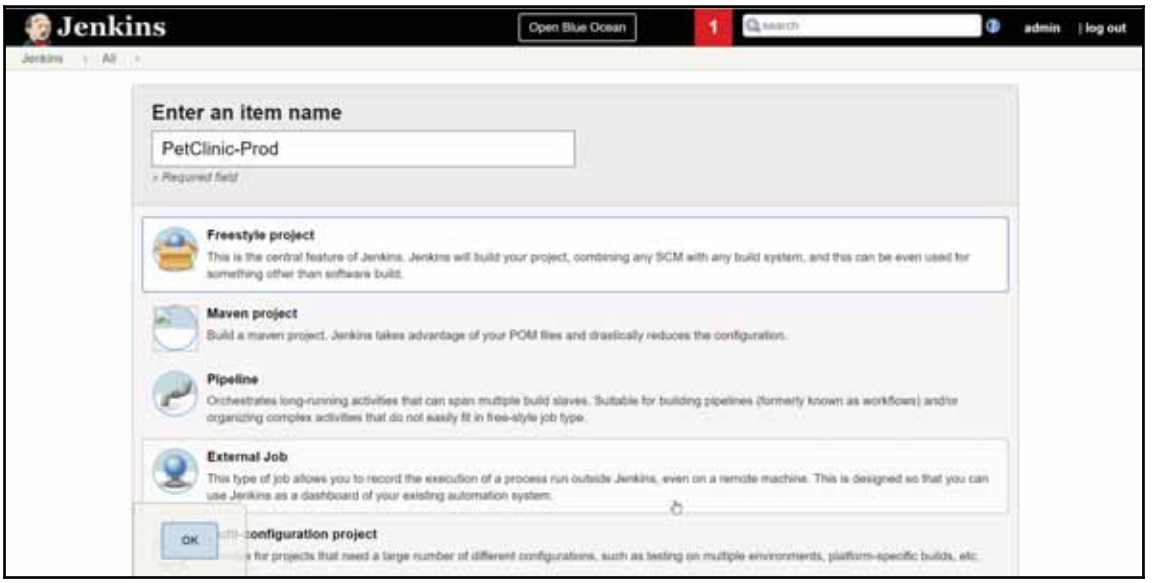
At the end of this chapter, we will know how to create and configure a pipeline using the plugin, and also using Groovy syntax.

Build Pipeline

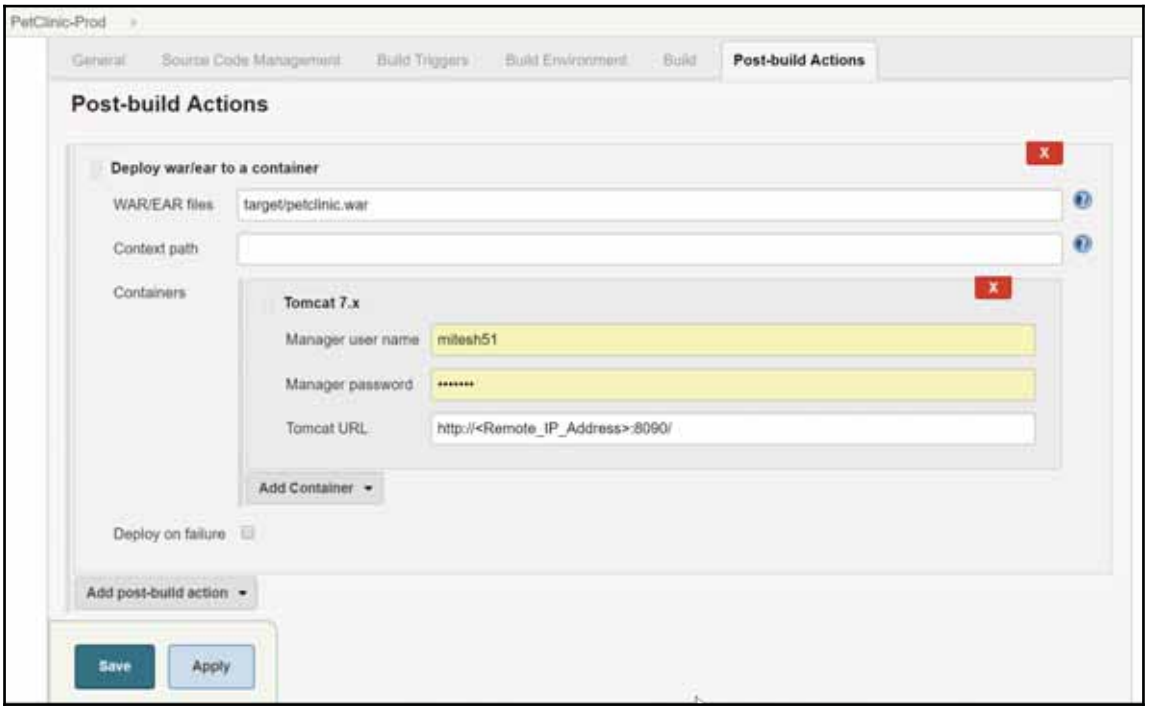
Continuous Integration and Continuous Delivery have become popular practices for application development. The Build Pipeline plugin provides a pipeline view of upstream and downstream connected jobs that typically form a build pipeline, with the ability to define manual triggers or an approval process. We can create a chain of jobs by orchestrating version promotion through different quality gates, before we deploy it in production.

Before starting with the Build Pipeline plugin, let's create a job to deploy into a production environment. We will use the Deploy to Container plugin for application deployment in remote Tomcat.

1. Click on **New Item** in the Jenkins Dashboard and give it a name. Click on **OK**:



2. Configure a post-build action similar to what we configured in the **PetClinic-Deploy** job:



3. Click on **Save** and **Apply**.

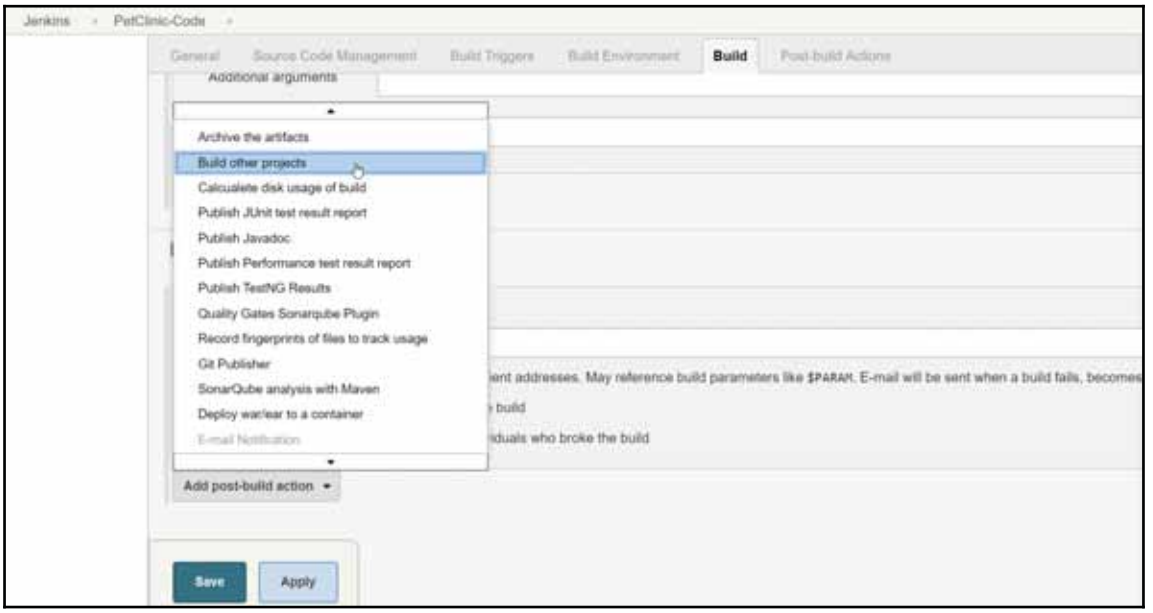
In the next section, we will configure upstream and downstream relationships between all the jobs we have created so far.

Upstream and downstream jobs

An upstream job is a configured project that triggers a project as part of its execution. A downstream job is a configured project that is triggered as part of the execution of the pipeline.

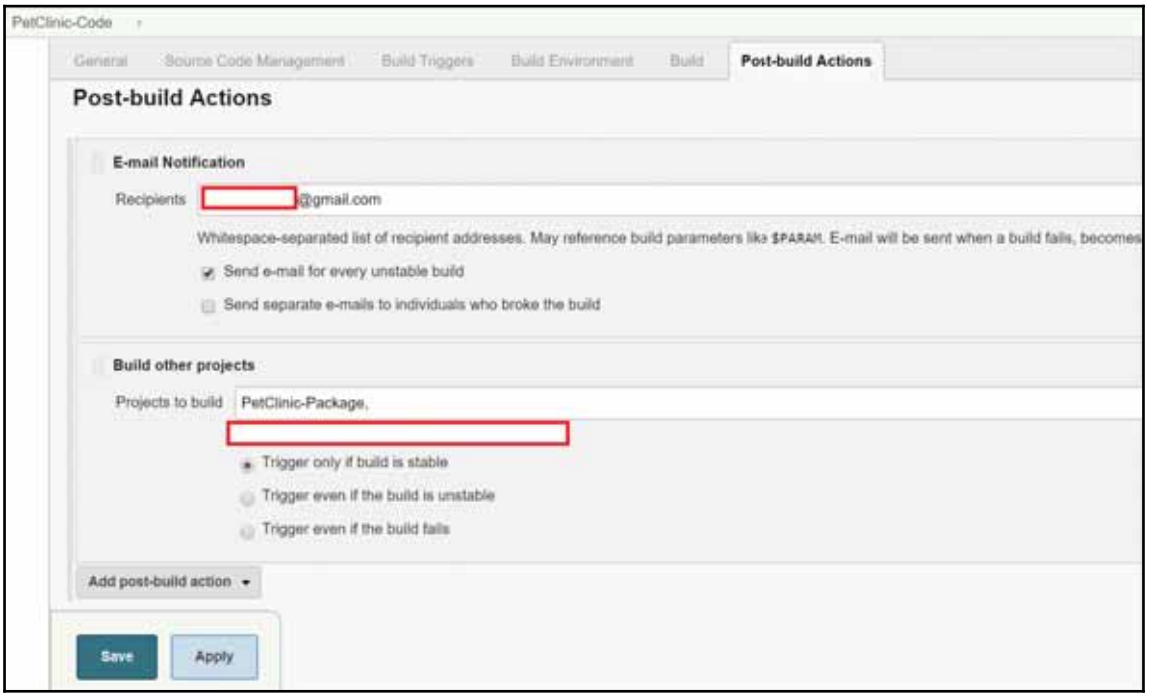
Let's start with the first job we created for static code analysis. Go to the **Post-build Action** section in the configuration of the **PetClinic-Code** build job.

1. Select **Build other projects** from the available options:

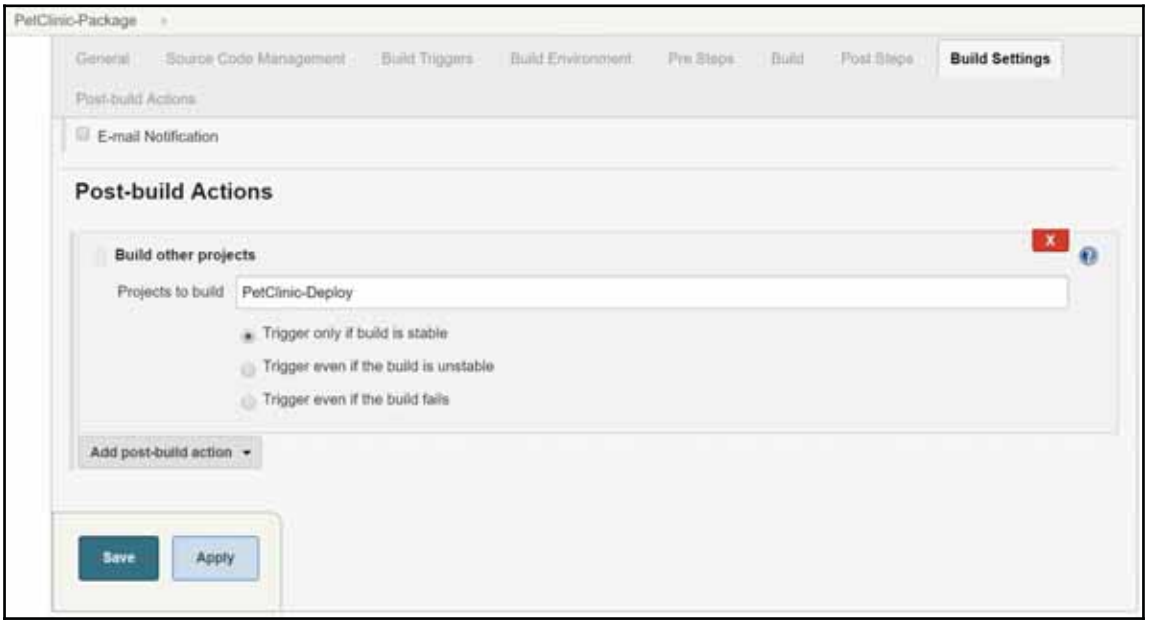


2. We would like to create a package after static code analysis is done, so we will select **PetClinic-Package** where CI is configured for the compilation of source code, unit test execution, and package file creation.
3. Click **Save**.

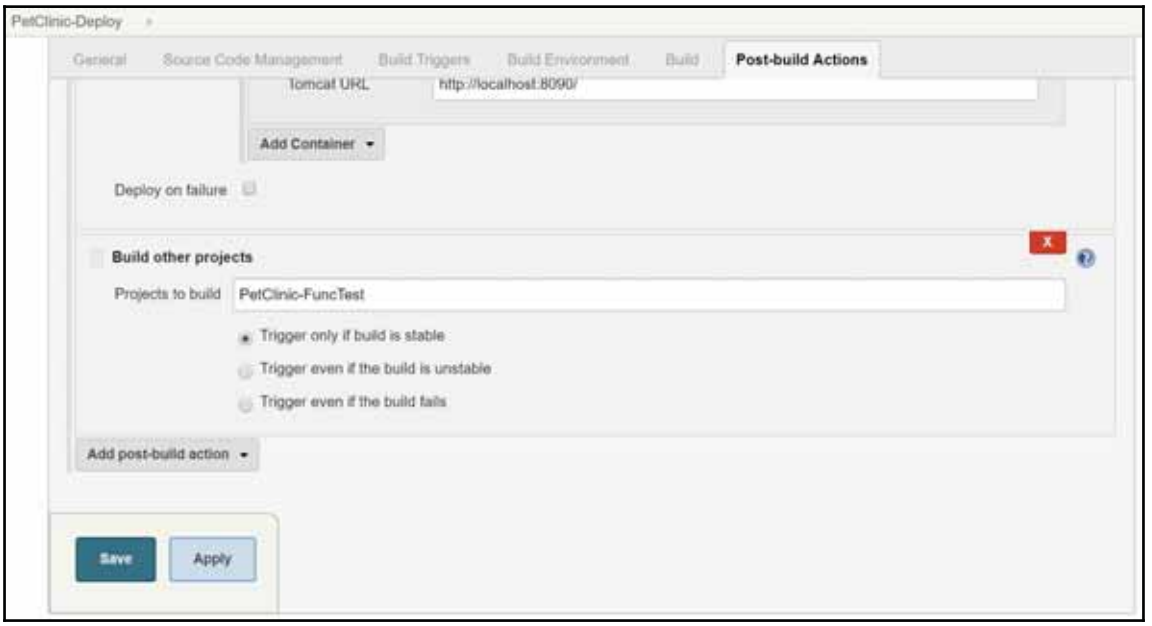
4. For **PetClinic-Code**, **PetClinic-Package** is a downstream job, while for **PetClinic-Package**, **PetClinic-Code** is an upstream job:



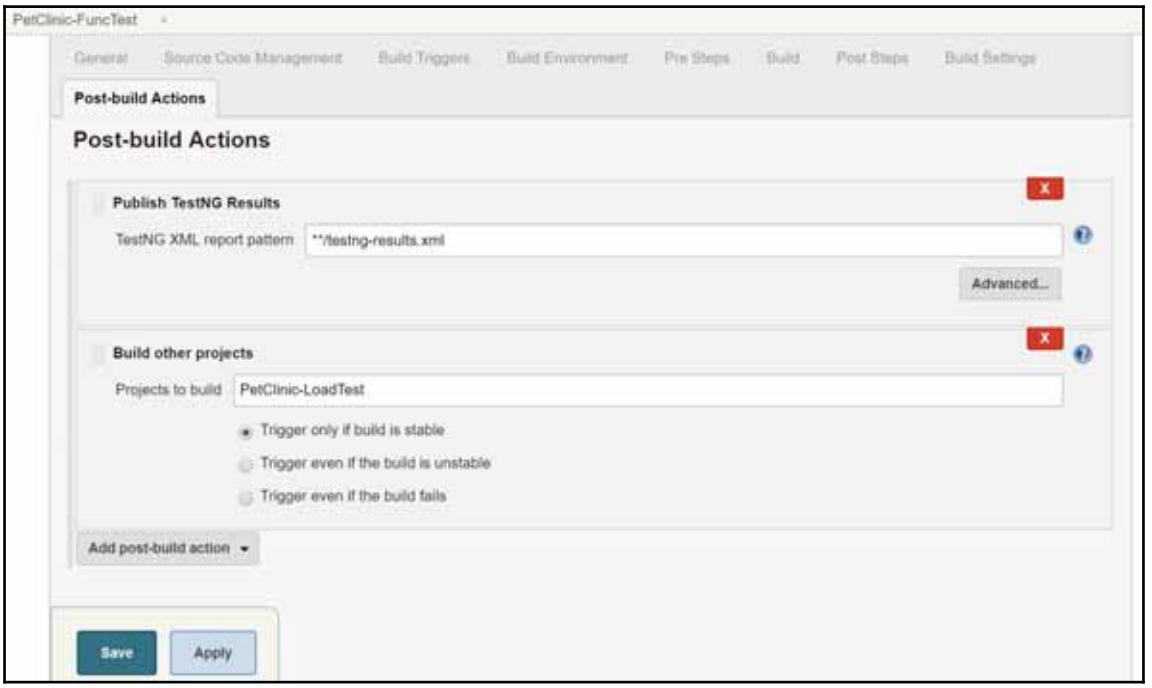
5. Once our package is ready, we would like to deploy it, so from the **PetClinic-Package** job, we will configure **PetClinic-Deploy** as a downstream job in **Post-build Actions**:



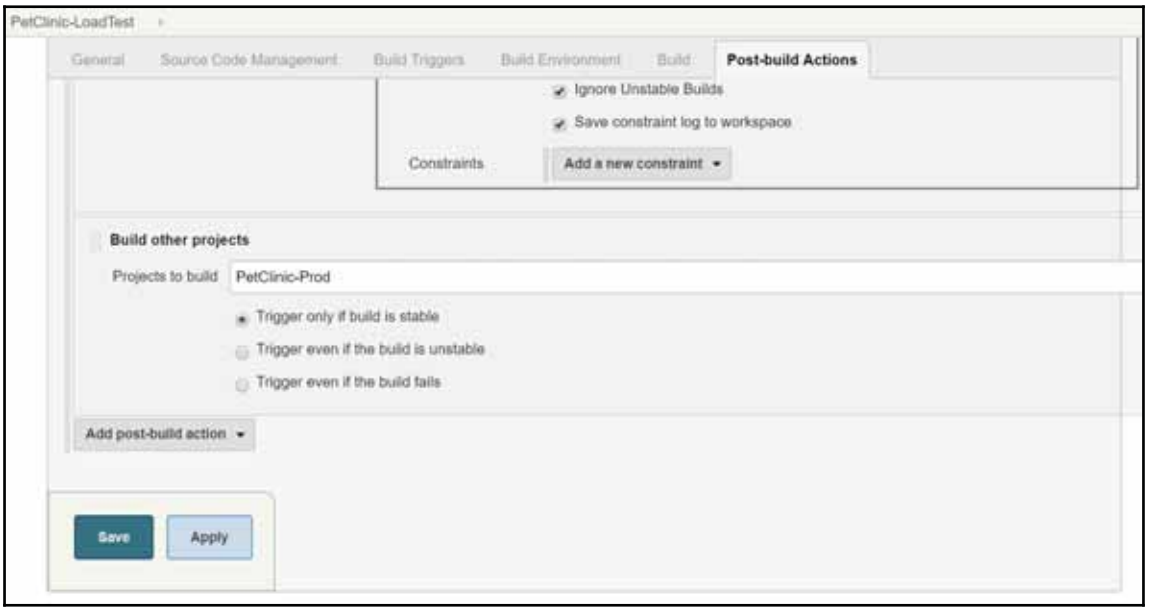
6. Once our application is deployed to the server, we would like to perform functional test cases, so from the PetClinic-Deploy job, we will configure **PetClinic-FuncTest** as a downstream job in **Post-build Actions**:



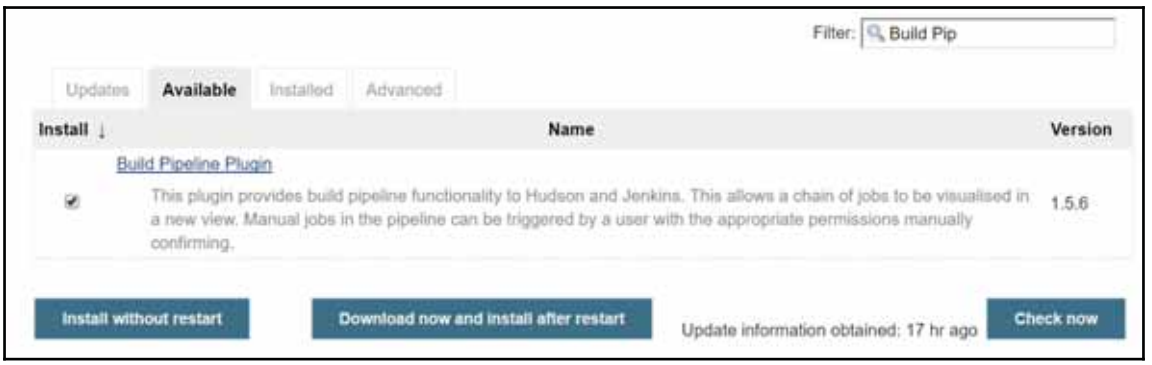
7. Once the functional test cases are executed successfully, we would like to perform load testing so from the PetClinic-FuncTest job, we will configure **PetClinic-LoadTest** as a downstream job in **Post-build Actions**:



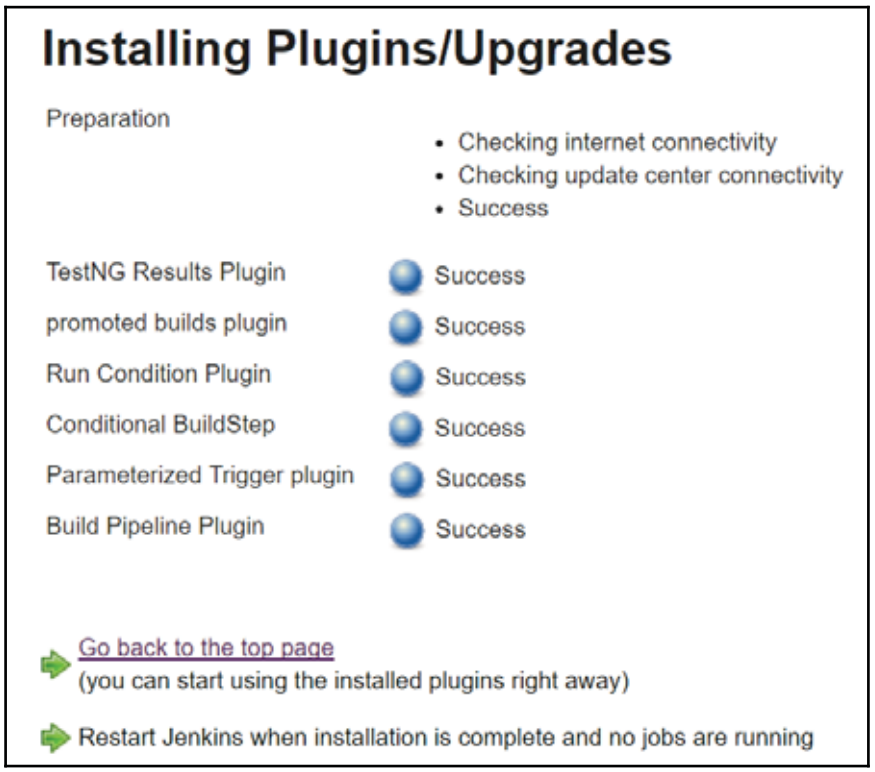
8. Once load testing is completed, we would like to deploy the application in the prod environment, so from the **PetClinic-LoadTest** job, we will configure **PetClinic-Prod** as a downstream job in **Post-build Actions**:



9. Install the plugin from **Manage Jenkins** | **Manage Plugins**:

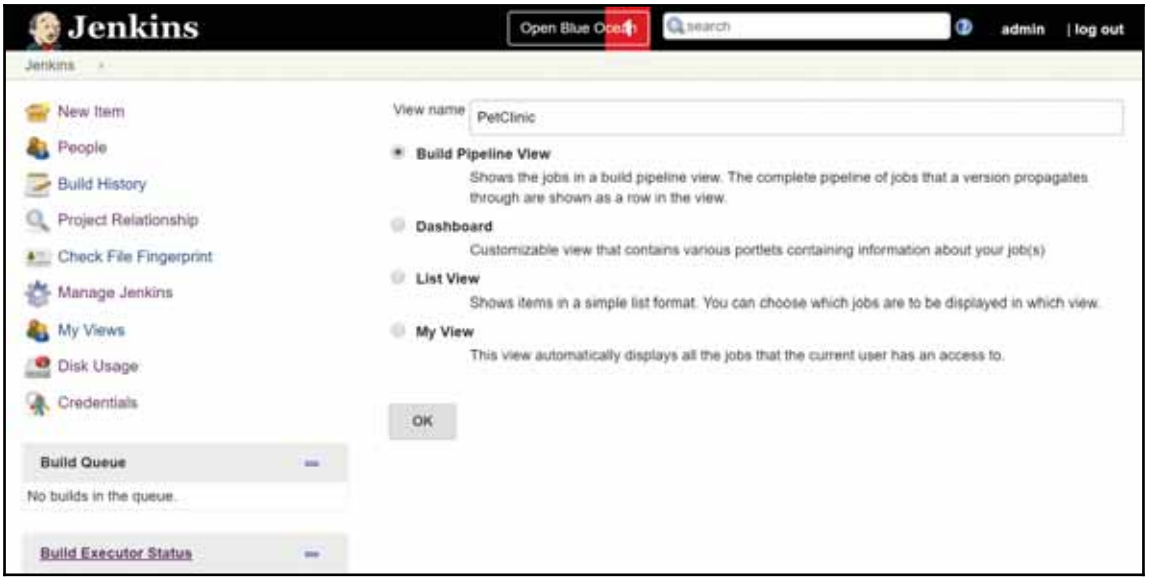


10. Verify the successful installation of the `Build Pipeline` plugin:

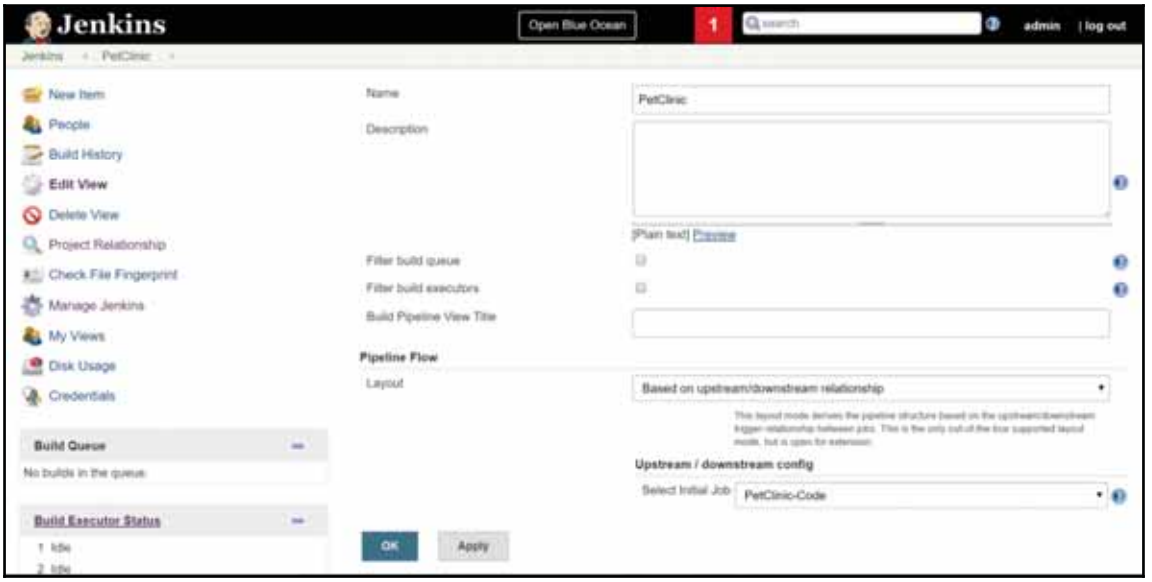


11. Go to the Jenkins dashboard and click on the plus sign on the tabs available.
12. Provide **View name** and select **Build Pipeline View**.

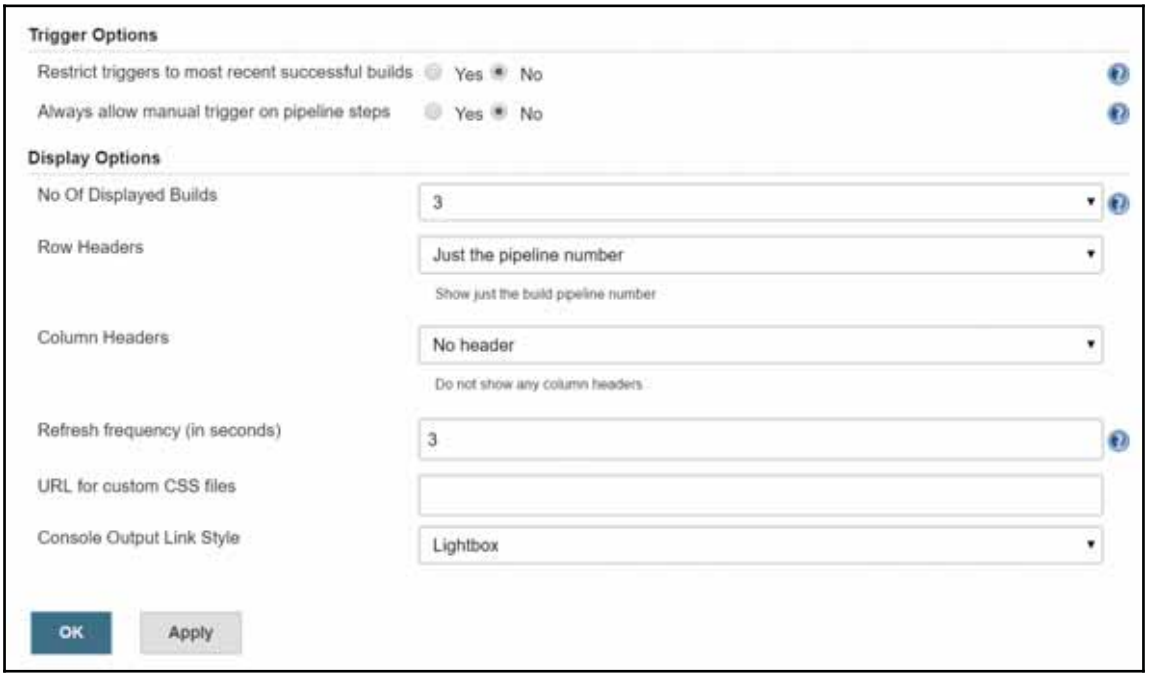
13. Click on **Save**:



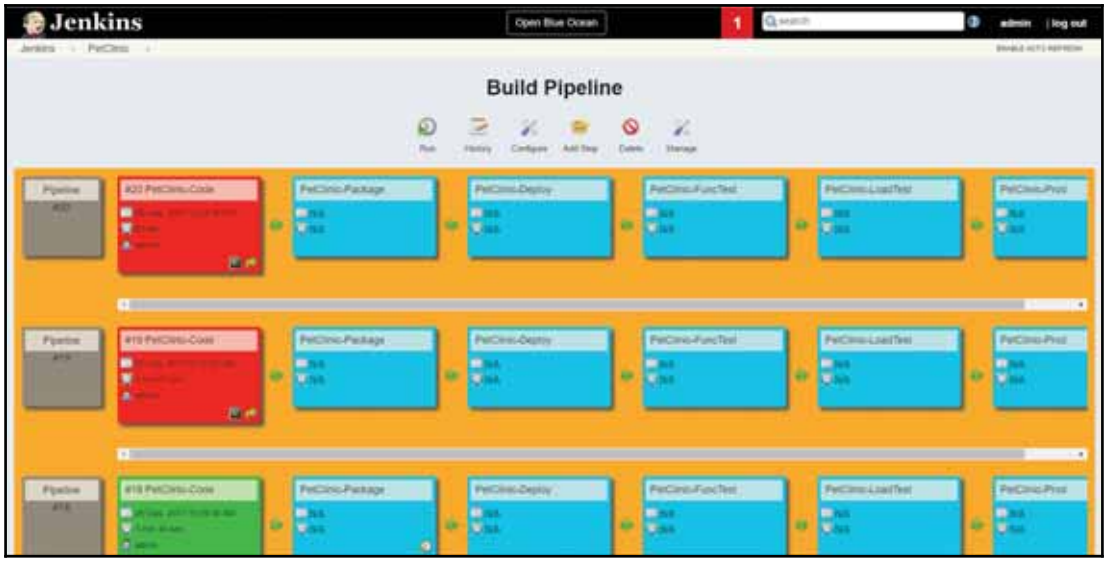
14. Verify the layout is configured as **Based on upstream/downstream relationship**.
15. We want to execute **PetClinic-Code** as a first job, so select it in **Select Initial Job**:



16. Select the rest of the configuration as per the requirement.
17. Change the number of displayed builds from **1** to **3** so it will display the last three build pipelines executed.
18. Click on **OK**:

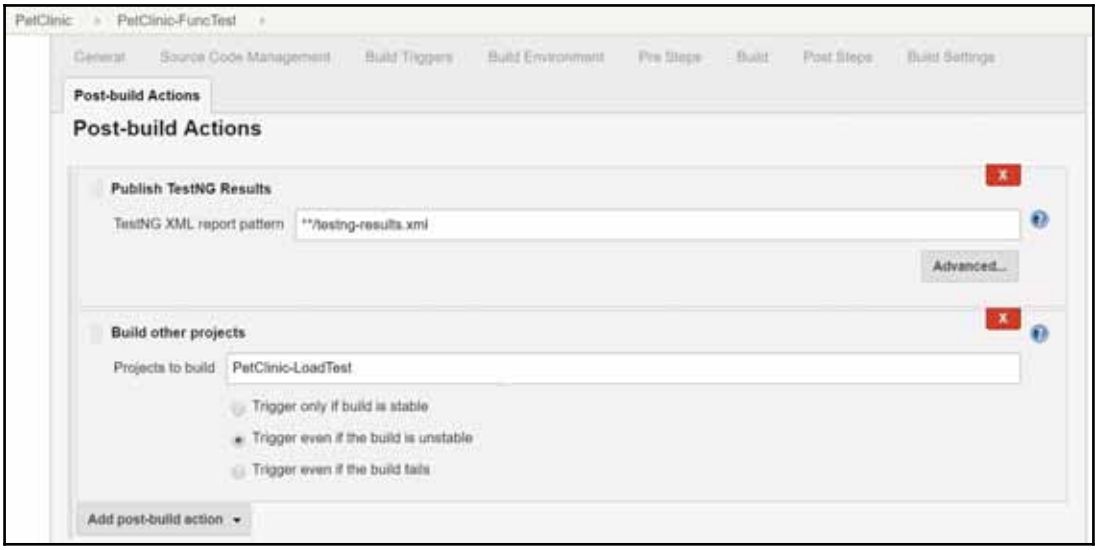


19. Check the **Build Pipeline** view in the Jenkins dashboard:

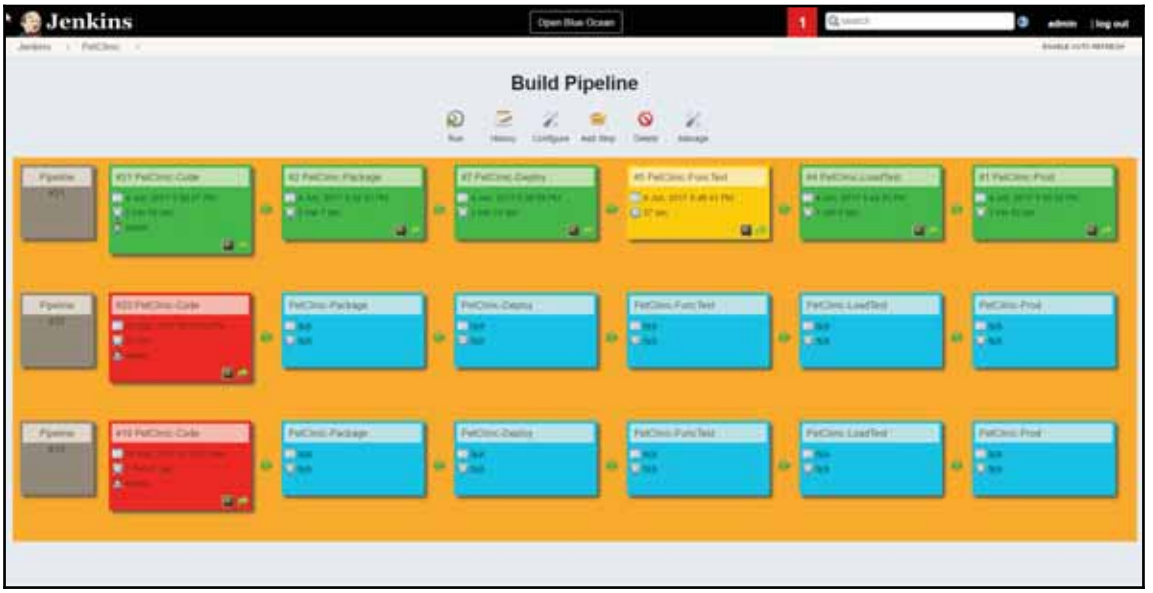


Build Pipeline with configured upstream and downstream job

20. What if we want to continue execution of the pipeline even if a build is unstable? In such a case, we need to select the option **Trigger even if the build is unstable** in **Post-build Actions**, as shown in the following screenshot:



21. Click on **Run** in the **Build Pipeline** view. Wait until the complete pipeline has been executed:



Successful Execution of Build Pipeline with configured upstream and downstream job

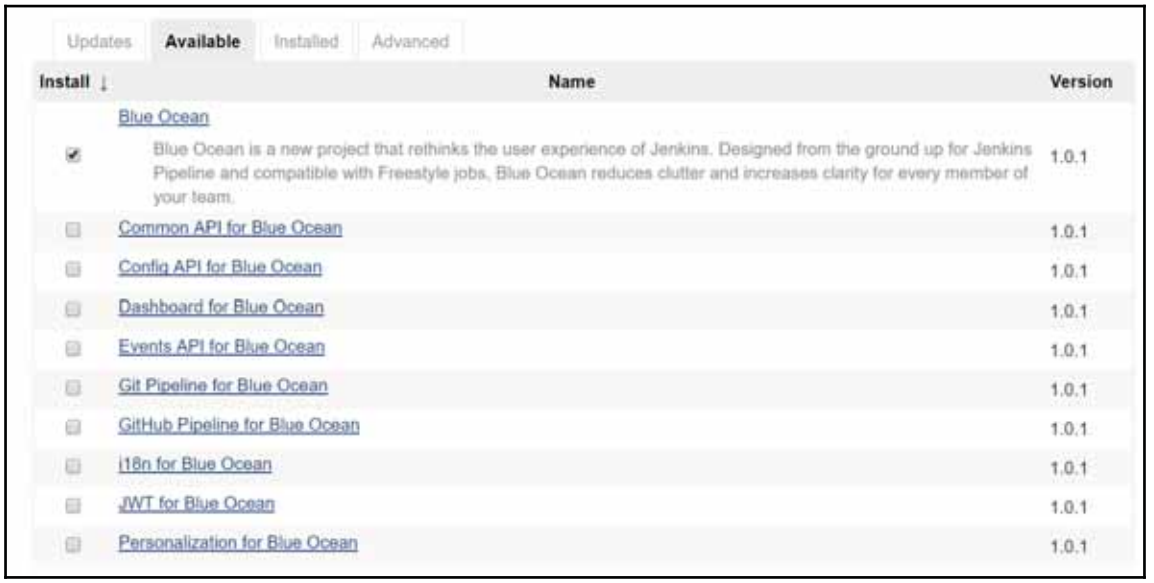
Done. This is how we can create a sequence of execution for different build jobs and achieve end-to-end automation for application lifecycle management. In the next section, we will achieve similar things using the Pipeline as a Code feature.

Overview of pipeline as a code

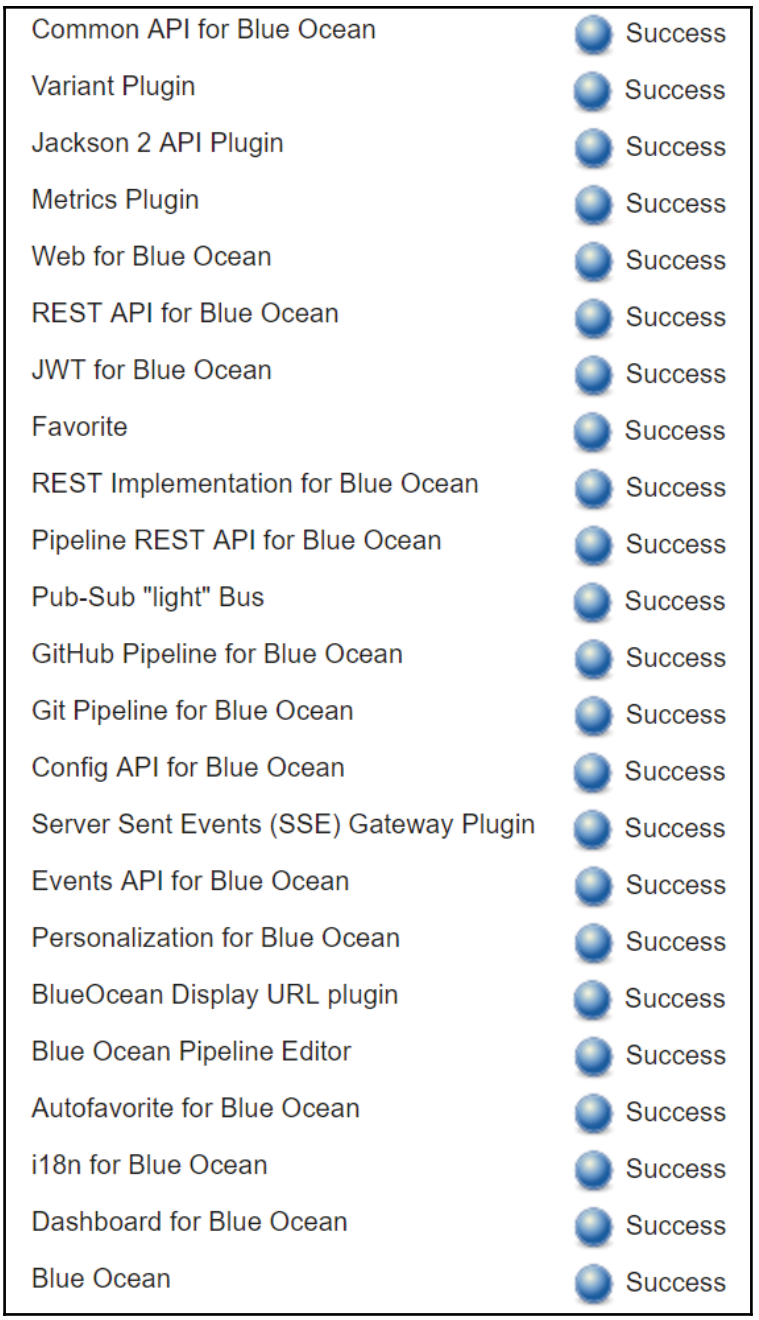
The Jenkins Pipeline feature supports Continuous Delivery pipelines and Continuous Deployment into Jenkins using Pipeline DSL. In Pipeline, we model all related tasks to decide the sequence of execution. We will perform the same tasks we performed with the Build Pipeline plugin.

Pipeline as a code - implementation

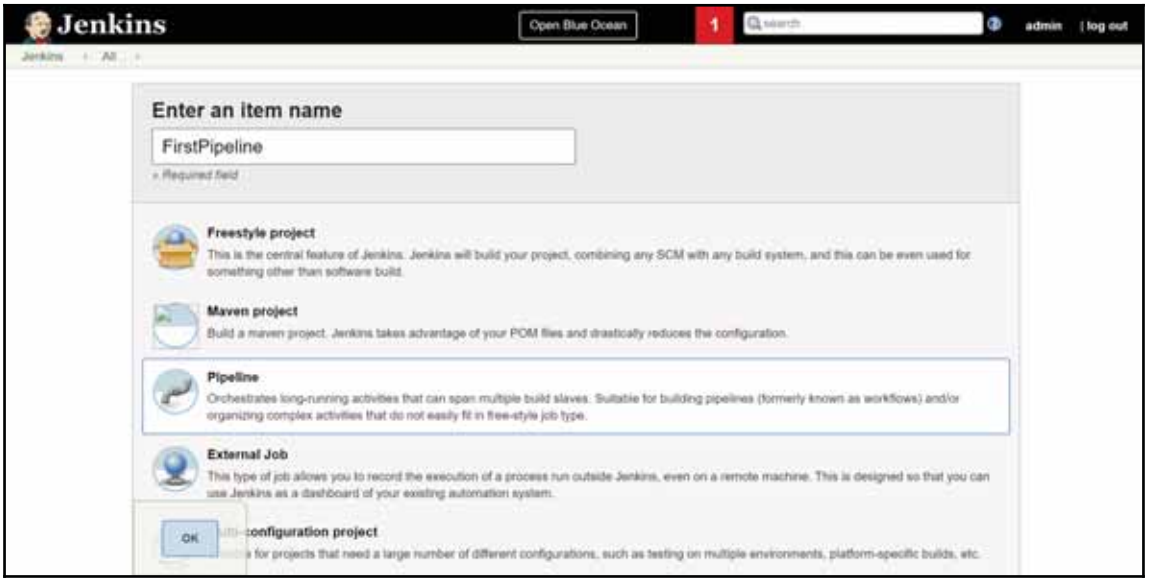
Blue Ocean is a new user interface for Jenkins. The idea behind introducing Blue Ocean is to make Jenkins and Continuous Delivery approachable to all team members. We will use Blue Ocean later in the chapter, but we will install it now:



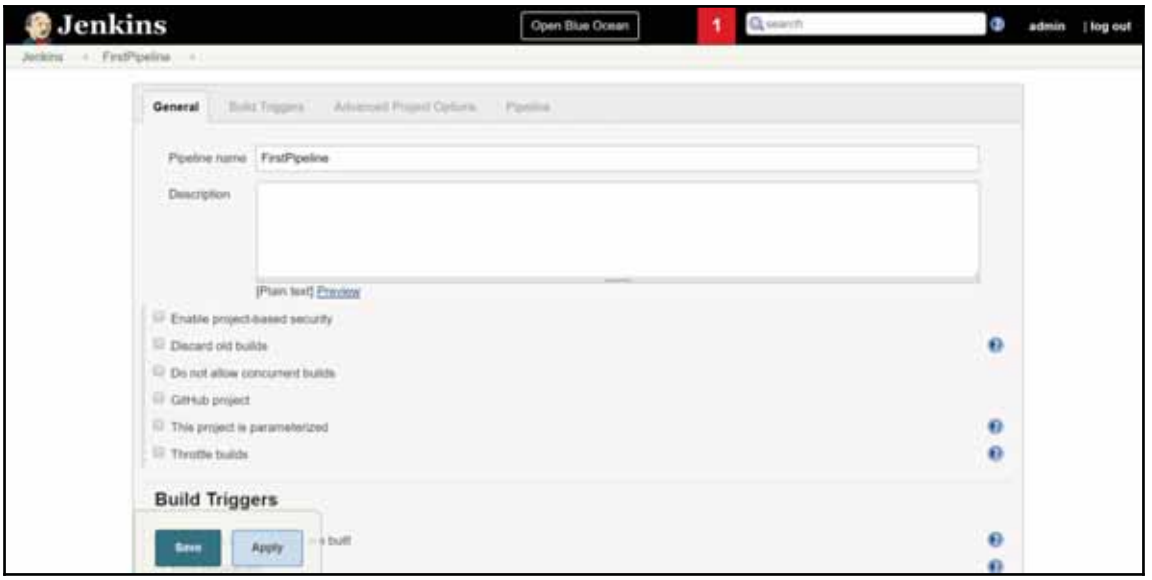
1. Verify the successful installation of the plugin in the Jenkins plugin section:



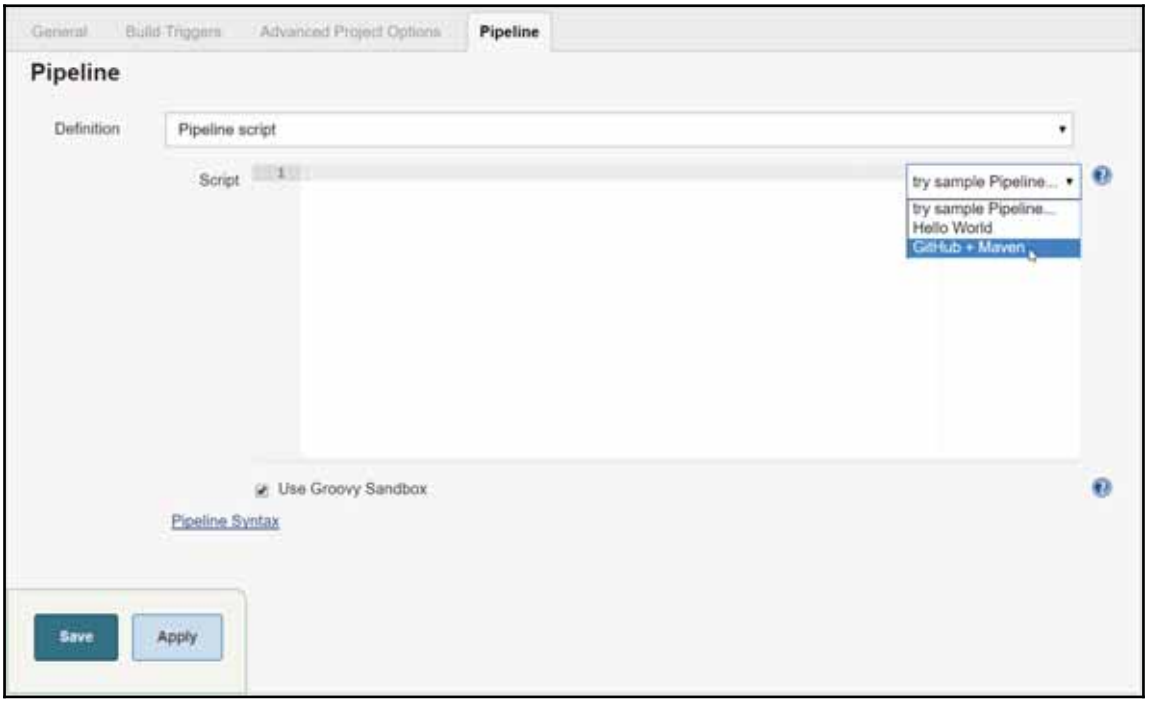
2. Now we will create our first pipeline in Jenkins.
3. Click on **New Item**. Enter an item name and select **Pipeline**.
4. Click **OK**:



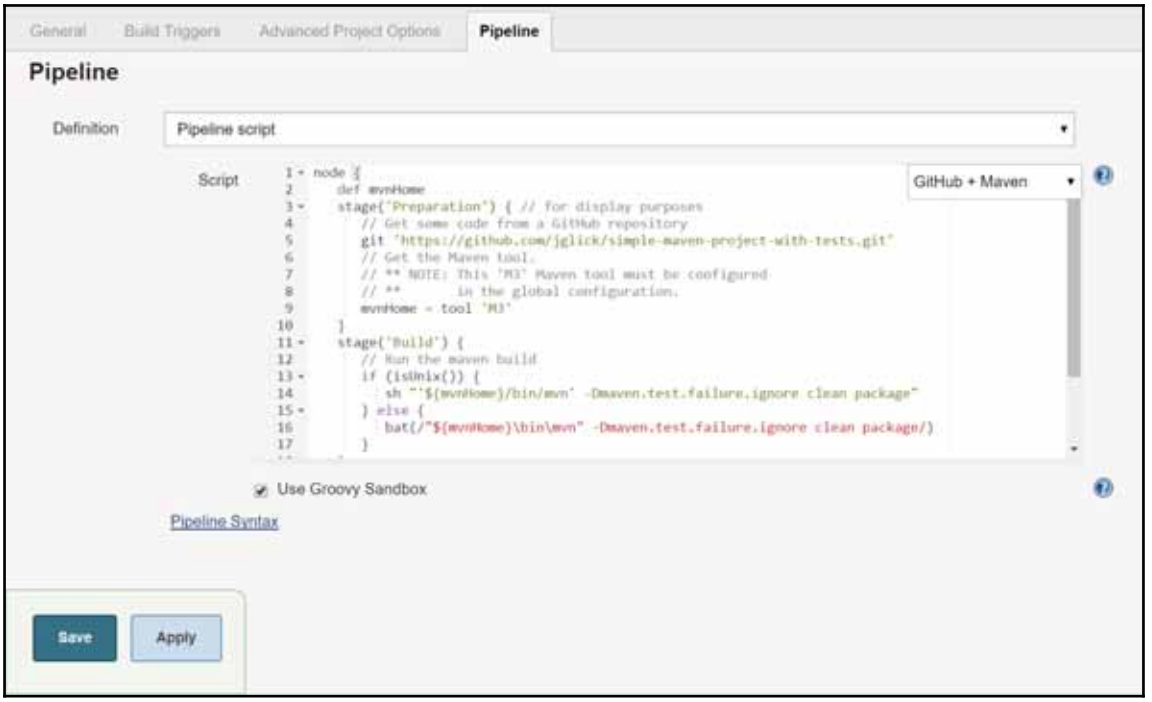
5. This will open the configuration of a newly created pipeline job:



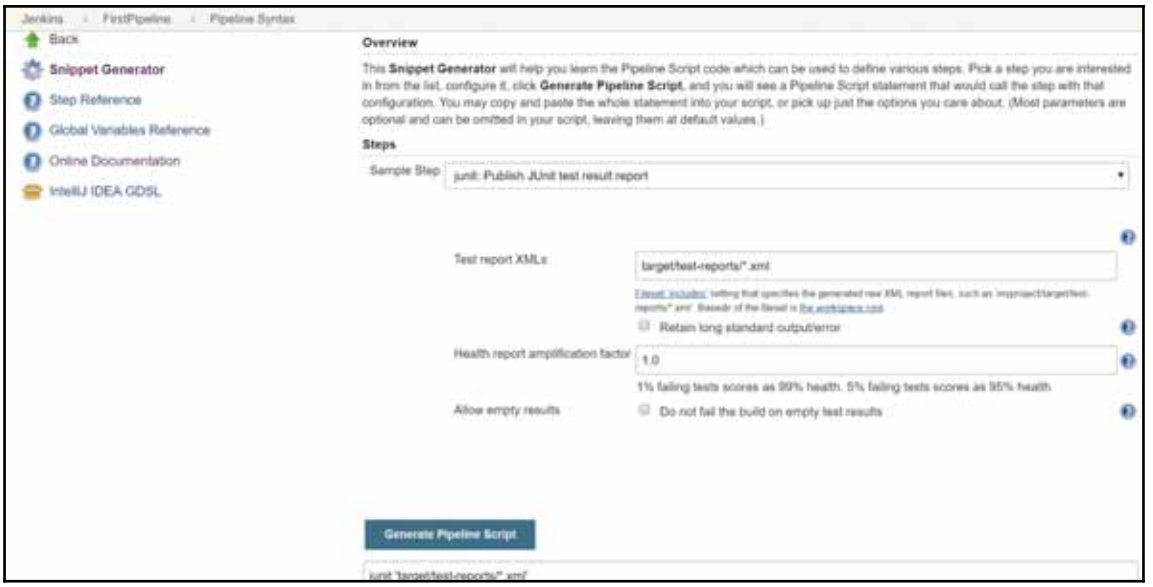
6. Go to the **Pipeline** section:



7. In the try sample **Pipeline** dropdown, select **GitHub + Maven**. It will automatically generate the syntax for the sample code. Make sure that `mvnHome` has a proper value, as per the path given for Maven in your system:



8. Click on **Save** and **Execute** the build to verify it.
9. However, we will create our own pipeline with the same sequence we tried with Build Pipeline.
10. Click on **Pipeline syntax** to generate the syntax for specific tasks we want to execute.
11. We can select the steps and configure the required things, and then click on **Generate Pipeline Script** to get the syntax that we can directly utilize in our pipeline:



12. Before creating our first script for a pipeline, let's understand some important terms:

- `Node` defines the node created in the context of Jenkins' master/agent architecture. It executes the step the moment the executor is available on the node. It creates a workspace or a directory for the pipeline to keep files. The following is the sample syntax:

```
node { // execute the pipeline on Master node } node('windows') {
 // execute the pipeline on node labelled as Windows }
```

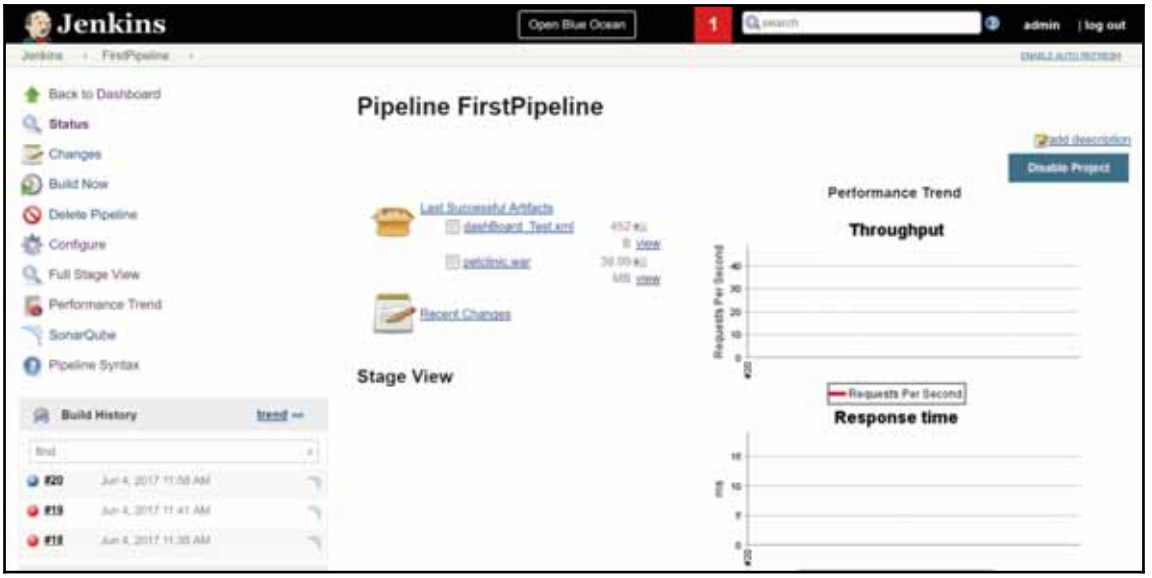
- **Stage** is a step that can be considered as a logically separate step such as init, build, test, deploy, and so on.
- **Step** or a **Build Step** is a task that can be executed to perform some activity such as copy `artifact`, `archieveartifact`, check out code from GitHub, and define environment variable.

13. Here is the script for Pipeline:

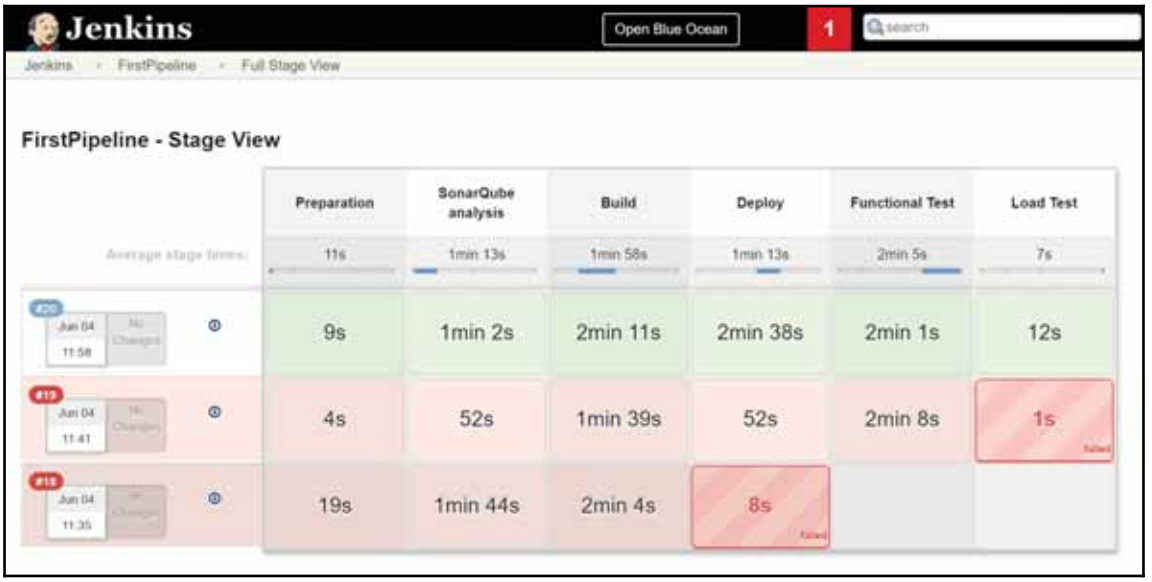
```
node {
   def mvnHome
   stage('Preparation') { // for display purposes
      // Get PetClinic code from a GitHub repository
      git 'https://github.com/mitesh51/spring-
 petclinic.git'
     // Get the Maven tool.
     // ** NOTE: This 'apache-maven-3.3.1' Maven tool must be
     configured in the global configuration.
     mvnHome = tool 'apache-maven-3.3.1'
     }
    stage('SonarQube analysis') {
     // requires SonarQube Scanner 3.0+
    def scannerHome = tool 'SonarQube Scanner 3.0.3';
 // Sonarqube6.3 must be configured in the Jenkins
 Configuration -> Add SonarWube server
         withSonarQubeEnv('Sonarqube6.3') {
 //provide all required properties for Sonar execution
            bat "${scannerHome}/bin/sonar-scanner -
 Dsonar.host.url=http://localhost:9000/ -
 Dsonar.login=1335c62cbfceab5
 954a5101ab7477cc974f58d56 -
 Dsonar.projectVersion=1.0
 -Dsonar.projectKey=petclinicKey -
 Dsonar.sources=src"
         }
    } stage('Build') {
         // Run the maven build based on the Operating system
         if (isUnix()) {
sh "'${mvnHome}/bin/mvn' -
 Dmaven.test.failure.ignore clean package"
   // Publish JUnit Report
       junit '**/target/surefire-reports/TEST-*.xml'
       } else {
          bat(/"${mvnHome}\bin\mvn" clean package/)
  // Publish JUnit Report
      junit '**/target/surefire-reports/TEST-*.xml'

      }
   }
   stage('Deploy') {
  // Archive the artifact
      archive 'target/*.war'
  // Execute the PetClinic-Deploy build to deploy war file into
  tomcat
  // Copy Artifact from this Pipeline Project into   PetClinic-
```

```
   Deploy using Copy Artifact plugin
       build 'PetClinic-Deploy'
      }
      stage('Functional Test'){
   // Checkout the code from Github to execute Functional test
 git 'https://github.com/mitesh51/petclinic-func.git'
 // Go to GitHub Directory and Fork it ... Change the URL in
 petclinic-func/src/test/java/example/NewTest.java
//driver.get("http://localhost:8090/petclinic/");
 //In the same file Change location of Gecko driver, we have used
 Firefox here on Windows...File file = new
 File("C:\\Users\\Mitesh\\Downloads\\geckodriver-v0.13.0-
 win64\\geckodriver.exe");
 // Run the maven build with test goal to execute functional test
    bat(/"${mvnHome}\bin\mvn" test/) }
 // This stage can be optional based on the requirements stage('Load
 Test'){
 // Execute command to perform load testing with the use of Apache
JMeter; In our case we are using JMeter that is already installed on
Windows hence the bat file is used. Make sure to change this
location based
on the Apache JMeter installation directory available in your
system.
          bat "C:/apache-jmeter-3.0/bin/jmeter.bat -
  Jjmeter.save.saveservice.output_format=xml -n -t
  C:/Users/Mitesh/Desktop/PetClinic.jmx -l Test.jtl"
  // Publish Apache JMeter results
          perfReport errorFailedThreshold: 50,
  errorUnstableThreshold: 30, ignoreFailedBuilds:
  true, ignoreUnstableBuilds: true,
  persistConstraintLog: true, sourceDataFiles:
  'Test.jtl'
  }
  //Done!
  }
```

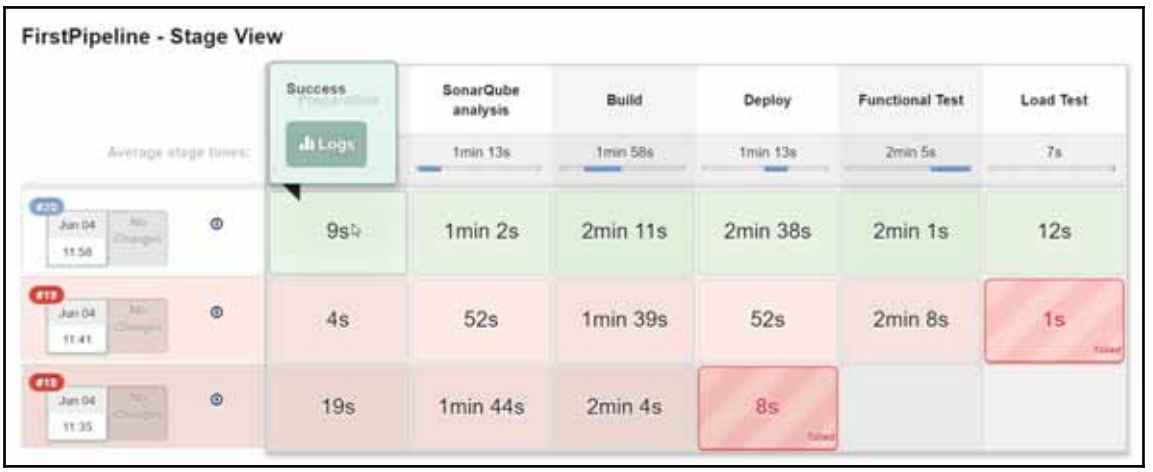
14. Click on **Build Now** for pipeline execution:



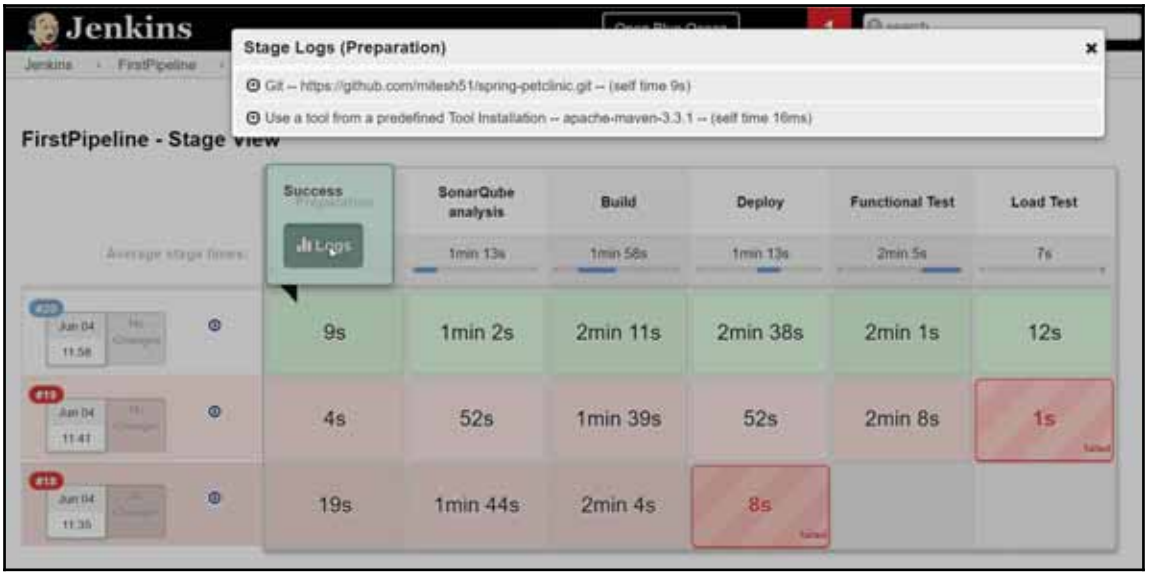
15. Verify the stage view of the pipeline we have created by clicking on **Full Stage View** on the Jenkins dashboard:



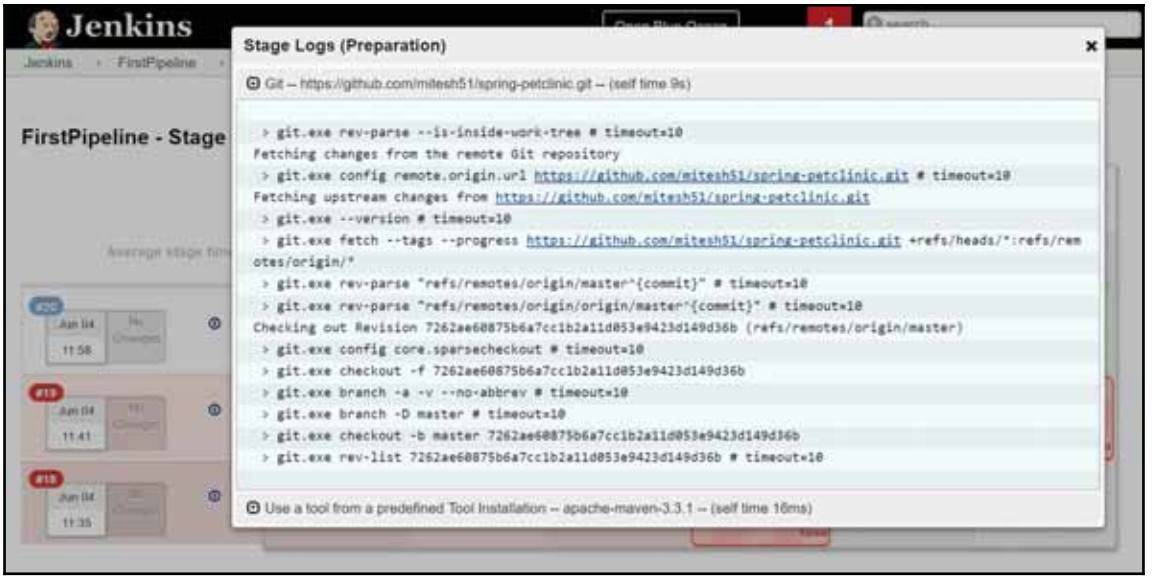
16. Mouse over the specific stage and click on **Logs**:



17. We can see and verify the stage logs directly from the stage view:

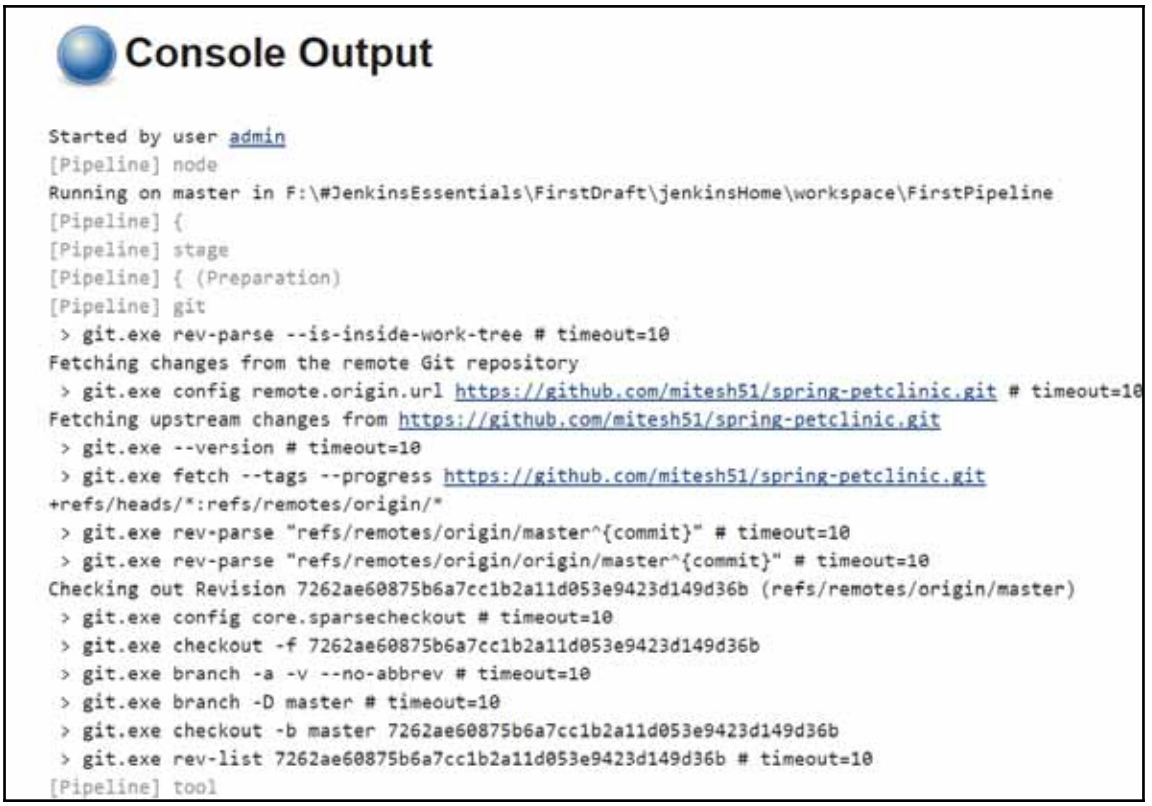


18. Click on the dropdown to get more details on the log:



19. Let's go to individual stage logs in the Console output:

    1. Look at the log for **Preparation Stage**:



Console Output

```
Started by user admin
[Pipeline] node
Running on master in F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline
[Pipeline] {
[Pipeline] stage
[Pipeline] { (Preparation)
[Pipeline] git
 > git.exe rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git.exe config remote.origin.url https://github.com/mitesh51/spring-petclinic.git # timeout=10
Fetching upstream changes from https://github.com/mitesh51/spring-petclinic.git
 > git.exe --version # timeout=10
 > git.exe fetch --tags --progress https://github.com/mitesh51/spring-petclinic.git
+refs/heads/*:refs/remotes/origin/*
 > git.exe rev-parse "refs/remotes/origin/master^{commit}" # timeout=10
 > git.exe rev-parse "refs/remotes/origin/origin/master^{commit}" # timeout=10
Checking out Revision 7262ae60875b6a7cc1b2a11d053e9423d149d36b (refs/remotes/origin/master)
 > git.exe config core.sparsecheckout # timeout=10
 > git.exe checkout -f 7262ae60875b6a7cc1b2a11d053e9423d149d36b
 > git.exe branch -a -v --no-abbrev # timeout=10
 > git.exe branch -D master # timeout=10
 > git.exe checkout -b master 7262ae60875b6a7cc1b2a11d053e9423d149d36b
 > git.exe rev-list 7262ae60875b6a7cc1b2a11d053e9423d149d36b # timeout=10
[Pipeline] tool
```

2. Look at the log for Sonarqube analysis stage:

```
[Pipeline] stage
[Pipeline] { (SonarQube analysis)
[Pipeline] tool
[Pipeline] wrap
Injecting SonarQube environment variables using the configuration: Sonarqube6.3
[Pipeline] {
[Pipeline] bat
[FirstPipeline] Running batch script

F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline>F:\#JenkinsEssentials\FirstDraft\jenkinsHome\t
ools\hudson.plugins.sonar.SonarRunnerInstallation\SonarQube_Scanner_3.0.3/bin/sonar-scanner -
Dsonar.host.url=http://localhost:9000/ -Dsonar.login****** -Dsonar.projectVersion=1.0 -
Dsonar.projectKey=petclinicKey -Dsonar.sources=src
INFO: Scanner configuration file:
F:\#JenkinsEssentials\FirstDraft\jenkinsHome\tools\hudson.plugins.sonar.SonarRunnerInstallation\SonarQube_Scanner_3
.0.3\bin\..\conf\sonar-scanner.properties
INFO: Project root configuration file: F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline\sonar-
project.properties
INFO: SonarQube Scanner 3.0.3.778
INFO: Java 1.8.0_111 Oracle Corporation (64-bit)
INFO: Windows 10 10.0 amd64
INFO: User cache: C:\Users\Mitesh\.sonar\cache
INFO: Load global settings
INFO: Load global settings (done) | time=3968ms
INFO: User cache: C:\Users\Mitesh\.sonar\cache
INFO: Load plugins index
INFO: Load plugins index (done) | time=138ms
INFO: SonarQube server 6.3.1
```

3. Look at the log for **Build Stage**:

```
[Pipeline] stage
[Pipeline] { (Build)
[Pipeline] isUnix
[Pipeline] bat
[FirstPipeline] Running batch script

F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline>"C:\apache-maven-3.3.1\bin\mvn" clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building petclinic 4.2.5-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ spring-petclinic ---
[INFO] Deleting F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline\target
[INFO]
[INFO] --- cobertura-maven-plugin:2.7:clean (default) @ spring-petclinic ---
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ spring-petclinic ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 18 resources
[INFO]
[INFO] --- maven-compiler-plugin:3.0:compile (default-compile) @ spring-petclinic ---
[INFO] Changes detected - recompiling the module!
[INFO] Compiling 45 source files to
F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline\target\classes
[parsing started
RegularFileObject[F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline\src\main\java\org\springfram
ework\samples\petclinic\web\PetValidator.java]]
[parsing completed 219ms]
```

4. Look at the log for **Deploy and Functional Test Stage**:

```
[INFO] Building war: F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline\target\petclinic.war
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 02:03 min
[INFO] Finished at: 2017-06-04T12:01:43+05:30
[INFO] Final Memory: 27M/88M
[INFO] ------------------------------------------------------------------------
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Deploy)
[Pipeline] step
Recording test results
[Pipeline] archive
[Pipeline] build (Building PetClinic-Deploy)
Scheduling project: PetClinic-Deploy
Starting building: PetClinic-Deploy #6
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Functional Test)
[Pipeline] git
 > git.exe rev-parse --is-inside-work-tree # timeout=10
Fetching changes from the remote Git repository
 > git.exe config remote.origin.url https://github.com/mitesh51/petclinic-func.git # timeout=10
Fetching upstream changes from https://github.com/mitesh51/petclinic-func.git
 > git.exe --version # timeout=10
 > git.exe fetch --tags --progress https://github.com/mitesh51/petclinic-func.git
+refs/heads/*:refs/remotes/origin/*
```

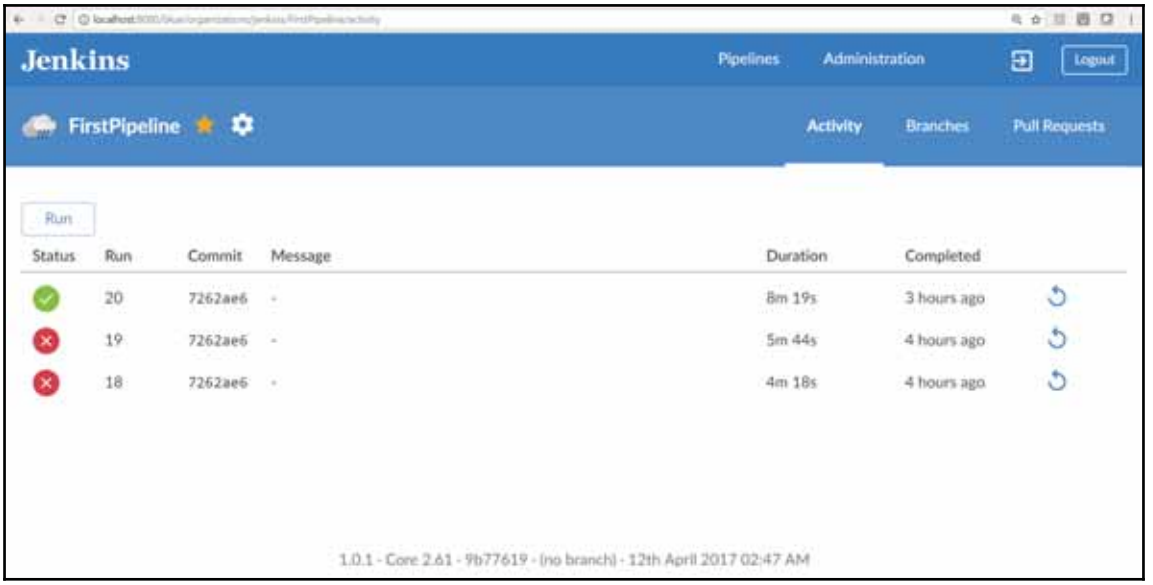
5. Look at the log for **Load Test Stage**:

```
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 01:42 min
[INFO] Finished at: 2017-06-04T12:06:24+05:30
[INFO] Final Memory: 18M/80M
[INFO] ------------------------------------------------------------------------
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Load Test)
[Pipeline] bat
[FirstPipeline] Running batch script

F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline>C:/apache-jmeter-3.0/bin/jmeter.bat -
Jjmeter.save.saveservice.output_format=xml -n -t C:/Users/Mitesh/Desktop/PetClinic.jmx -l Test.jtl
Writing log file to: F:\#JenkinsEssentials\FirstDraft\jenkinsHome\workspace\FirstPipeline\jmeter.log
Creating summariser <summary>
Created the tree successfully using C:/Users/Mitesh/Desktop/PetClinic.jmx
Starting the test @ Sun Jun 04 12:06:32 IST 2017 (1496558192833)
Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445
summary +      1 in 00:00:01 =    1.0/s Avg:   270 Min:   270 Max:   270 Err:     0 (0.00%) Active: 1 Started: 1
Finished: 0
summary +     49 in 00:00:01 =   61.1/s Avg:    13 Min:     4 Max:   157 Err:     0 (0.00%) Active: 0 Started: 1
Finished: 1
summary =     50 in 00:00:02 =   27.0/s Avg:    18 Min:     4 Max:   270 Err:     0 (0.00%)
```

20. On the project dashboard, look at the stage view at the bottom:

21. Now let's see how our pipeline looks in the Blue Ocean User Interface. Go to the **FirstPipeline** pipeline job that we have created. Click on the **Blue Ocean** link in the top bar on the Jenkins dashboard.

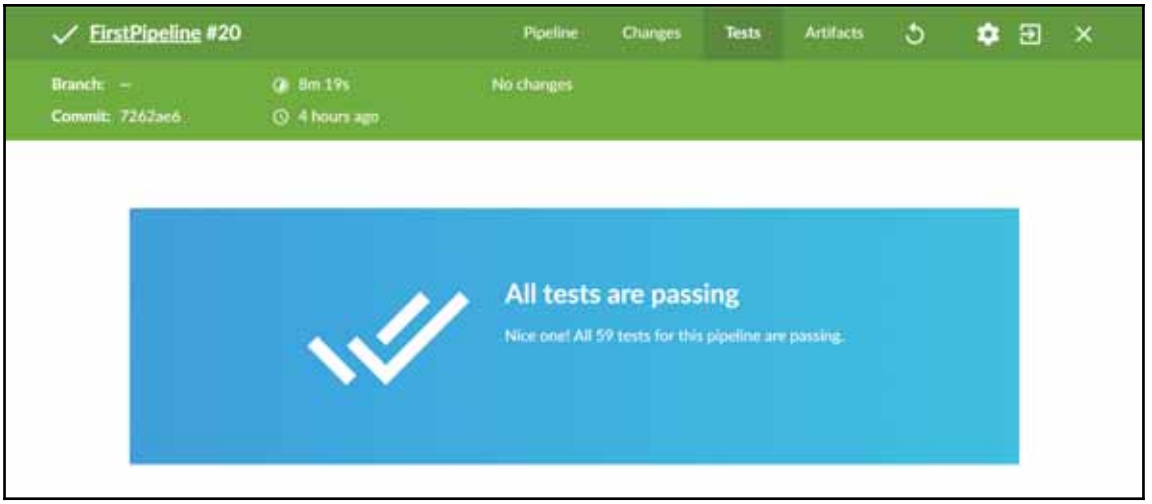
22. Click on successful pipeline **20**:



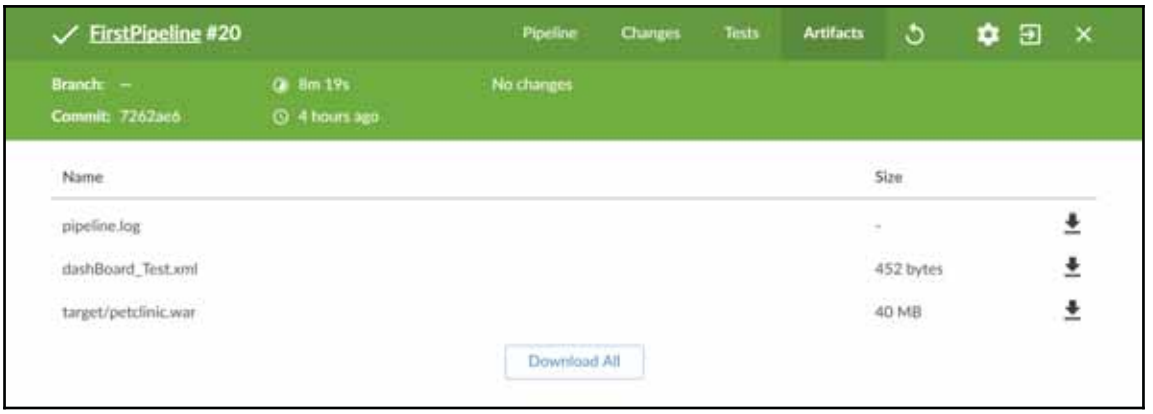
23. It will give details on the execution status of each stage in the **Blue Ocean** dashboard. Logs are available on the same page:

24. Select any stage and check the logs for the stage on the same page:

25. Click on the **Tests** link on the top bar to verify the status of the Junit test cases executed in the pipeline:



26. Click on the **Artifacts** link on the top bar to verify all the artifacts available in this pipeline:



In the next section, we will cover one important plugin, the Promoted builds plugin.

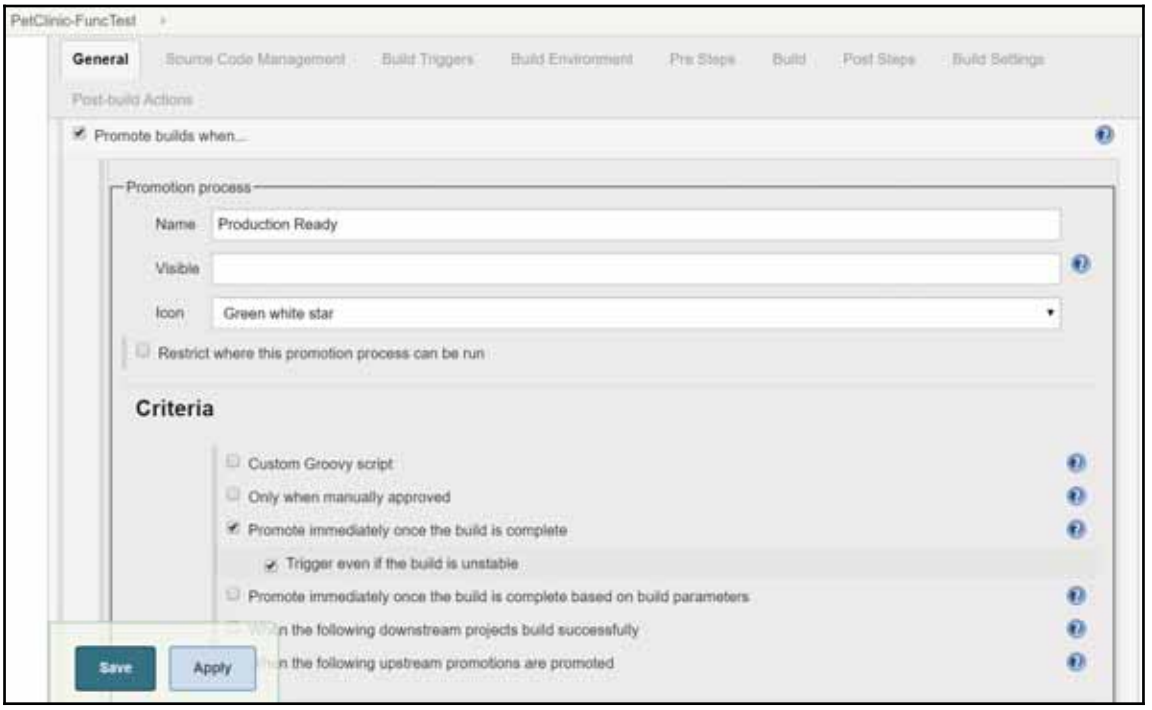
Promoted builds

The Promoted builds plugin allows us to tag the builds based on specific stages. This promotion can be manual or automated. We can identify promoted builds based on the star available on the project dashboard or the star available in Build History.

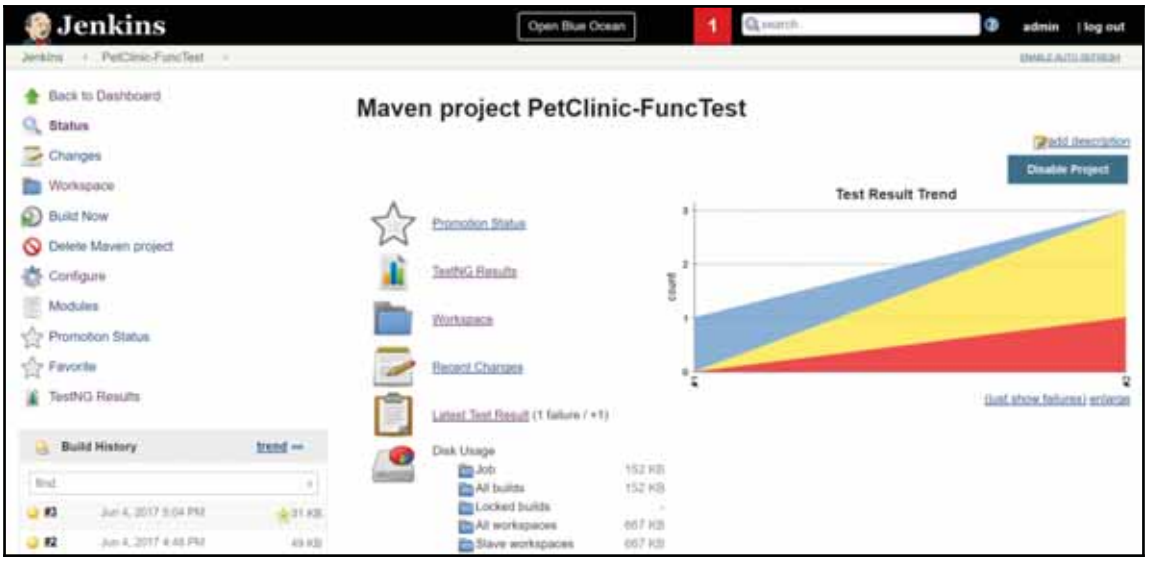
1. Go to **Manage Jenkins** and click on **Manage Plugins**.
2. Select **promoted builds plugin** and click on **Install without restart**:

3. Go to the **PetClinic-FuncTest** build and open its configuration.
4. In the **General** section, click on **Promote builds when...**

5. Provide a name and select a star you want to associate build if the criteria is passed in the icon list box in the promotion process section.
6. Select **Promote immediately when build is** complete, as shown in following screenshot:



7. We can also select **Only when manually approved** and then we can give the Email ID of the approver.
8. Click on **Build Now** and observe the Jenkins dashboard. Look out for the green star in **Build History** when the build is executed successfully:



The promoted builds feature can be utilized efficiently to assign a quality rating to the outcome of the build, so it can be utilized with confidence.

Summary

We have covered one of the most important concepts in this book; the orchestration of build jobs that performs various important tasks. We have configured end-to-end automation using the Build Pipeline plugin and also using the Pipeline as a Code feature available in Jenkins 2 and later.

We have utilized the Promoted builds plugin to assign quality tags to build jobs as well.

In the next chapter, we will see how to manage and monitor Jenkins and resources efficiently, using features available in Jenkins and also with the use of existing plugins.

8

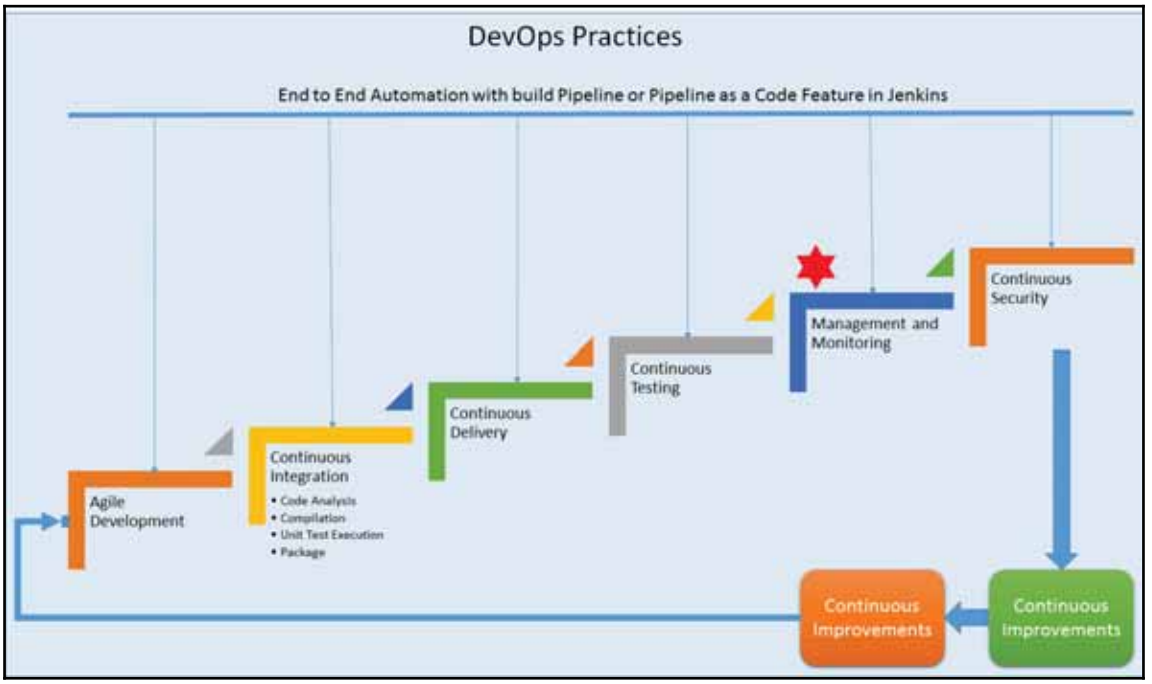
Managing and Monitoring Jenkins

The management and monitoring of Jenkins is essential, as it is at the core of our automation vision. We can utilize existing Jenkins features or plugins to manage and monitor Jenkins and its jobs effectively.

This chapter gives insight into the management of Jenkins nodes and monitoring them with Java melody to provide details on the utilization of resources. It also covers how to monitor build jobs configured for Java or .NET-related applications, and managing those configurations by keeping backups of them. This chapter describes the basic security configuration available in Jenkins in detail, for better access control and authorization. We will cover the following topics in this chapter:

- Managing Jenkins Master and Agent nodes
- Jenkins Monitoring with Java Melody
- Managing job-specific configurations - backup and restore
- Managing disk usage
- Build job-specific monitoring with the Build Monitor plugin
- The Audit Trail plugin- overview and usage
- The Workspace Cleanup plugin
- The pre-scm-build step plugin
- The conditional build step plugin
- The EnvInject plugin

In this chapter, we will cover the management and monitoring of Jenkins as a part of our DevOps journey:



At the end of this chapter, we will know how to configure Agent nodes for distributed architecture and be able to use various other plugins and functionalities to manage and monitor Jenkins effectively.

Managing Jenkins master and slave nodes

Jenkins supports a **Master/Agent** architecture. In a Master/Agent architecture, we can install Jenkins on the master and then utilize other agents for distributing the load.

We can delegate Jenkins jobs to agents for execution. This way we can support multiple executions using different resources.

There are specific scenarios where a Master/Agent architecture is extremely useful, such as the following:

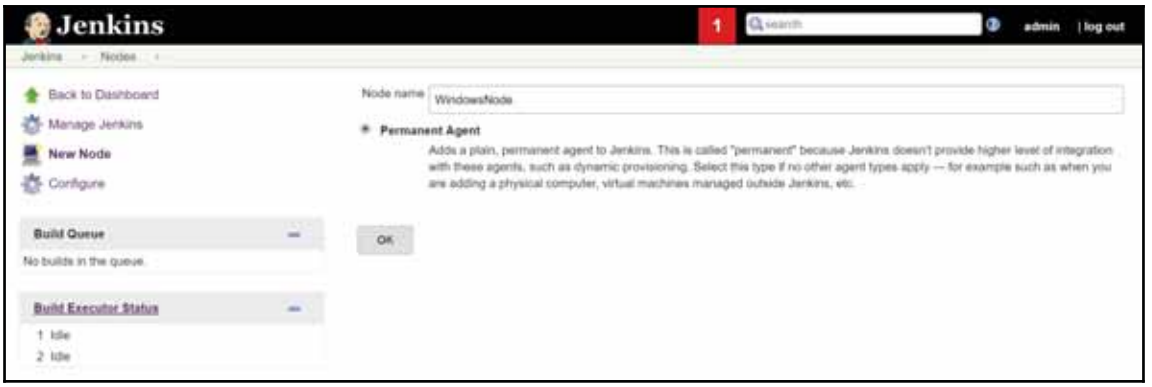
- A Jenkins machine has limited capacity. Even with higher capacity, there will be a time when it can't fulfil all requests. By distributing the load between Agent nodes, we can free system resources where Jenkins is installed.
- Different jobs require different kinds of resources, and they are restricted to specific machines only. In such cases, we can only utilize that machine -- it is not possible to configure it on the Jenkins system -- so it is better to utilize that machine as an agent.
- If different operating systems are required or some tools work only in specific OSes, then we can utilize those tools by making a system agent on which they are installed.
- To avoid a single point of failure caused by installing each and every tool on the Jenkins machine.

The important thing here is we don't install Jenkins on Agent nodes at all. We only install Jenkins on the Master and utilize that Jenkins for orchestrating the Master/Agent architecture.

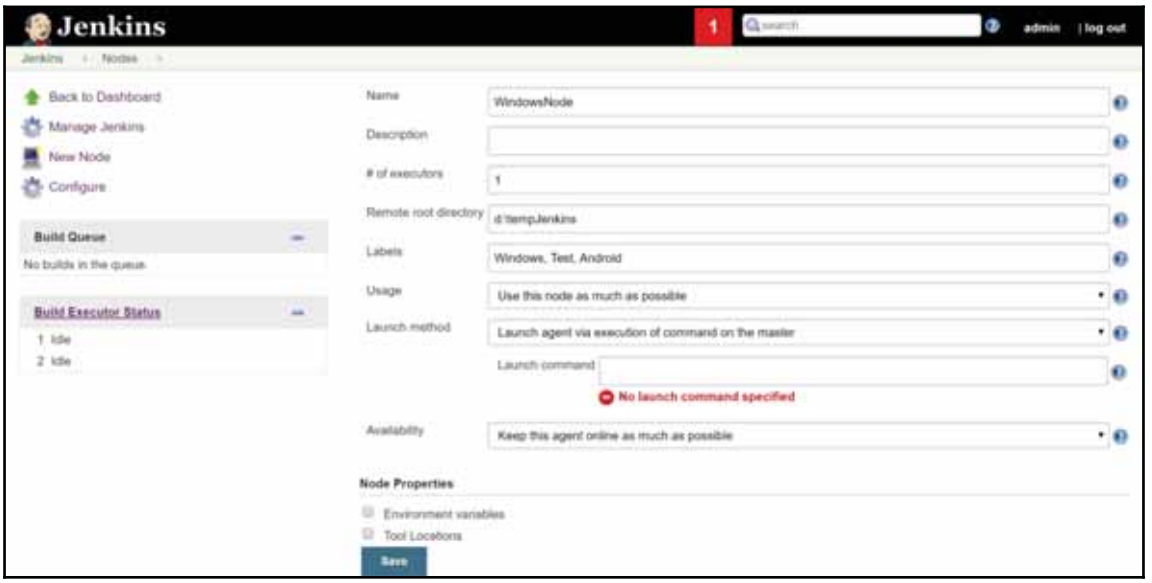
Just to note, whichever system we install Jenkins on becomes master. Verify that by navigating to **Manage Jenkins**|**Manage Nodes**.

Navigate to Manage Jenkins|Manage Nodes and click on **New Node**.

Give the node a name and select **Permanent Agent**; click **OK**:



Provide details as required and click on **Save**:



The following table describes all fields available for Agent configuration in detail:

# of executors	Here we can specify the maximum number of concurrent builds that Jenkins may execute on this agent node. Agents must have at least one executor for Job execution, and in case we don't want any execution, then we can configure this setting to 0.
Remote root directory	An agent node requires a directory dedicated to Jenkins. Specify the path to this directory on the agent. Use an absolute path only. All job configurations, build logs, and artifacts are stored on the master. Workspace is available on the Agent Node.
Labels	Labels or tags can be utilized to group multiple Agent Nodes into a logical collection. The best thing is that we can use this mechanism as a pool of resources to execute build jobs in Jenkins. For instance, we have multiple Agents where the test infrastructure is set up. We have two to three projects whose automated testing is done by the QA team. In such a scenario, we can provide the same "Test" label, or tag to all Agent nodes where the test infrastructure is available and then assign the same "Test" label to those projects. It will execute the build jobs on any one of the Test agents with the Test label, but not one without it.
Usage	This setting controls how Jenkins schedules builds on a specific Agent node. **Use this node as much as possible:** This is the default setting where most of the time this agent node will be utilized for Job execution. **Only build jobs with label expressions matching this node:** With this setting, Jenkins will execute builds on this agent node when that project (or Jenkins build job) is configured to execute on this node with label expression.
Launch method	This setting controls how Jenkins starts the specific agent node. Launch agent via Java Web Start: This allows an agent to be launched using Java Web Start. Launch the agent via the execution of a command on the master by remotely executing a process on another machine, such as via SSH or RSH. Launch slave agents via SSH by sending commands over a secure SSH connection. Let Jenkins control this Windows slave as a Windows service as it starts a Windows slave by a remote management facility built into Windows.

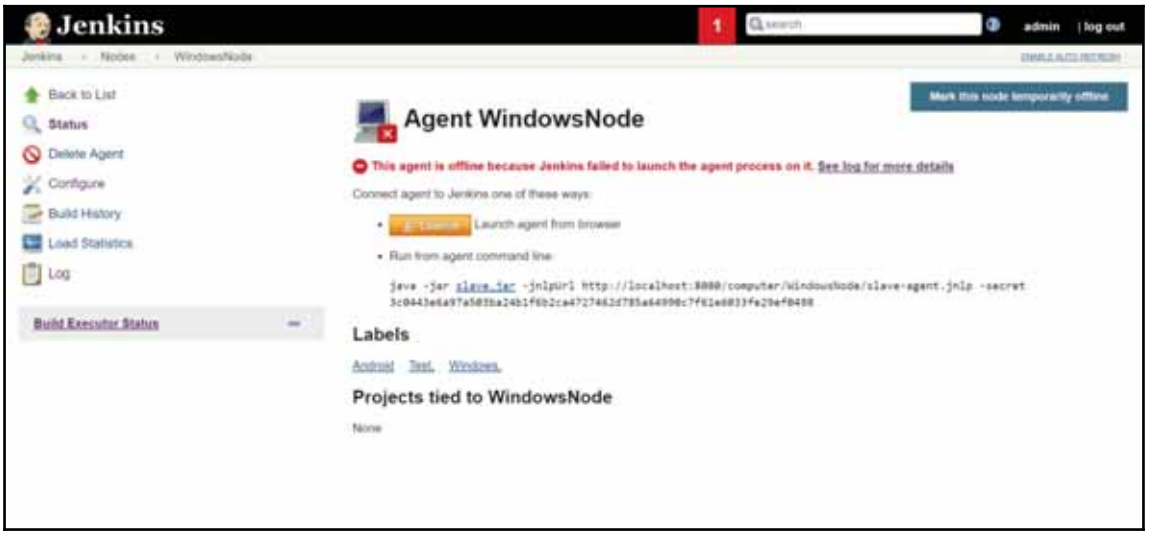
Availability	This setting controls when Jenkins starts and stops this agent. Keep this agent online as much as possible. Take this agent online and offline at specific times. Take this agent online when in demand, and offline when idle.
Environment variables	We can define Agent node-specific environment variables here.
Tool Locations	We can define Agent node-specific tools locations here.

We can see the newly created agent in the node list as disconnected. Click on it:

See the details available on the Agent page. We are not able to start it:

Go to **Manage Jenkins** | **Configure Global Security** | **Enable security**. In **TCP port for JNLP agents setting**, select **Random** and click on **Save**:

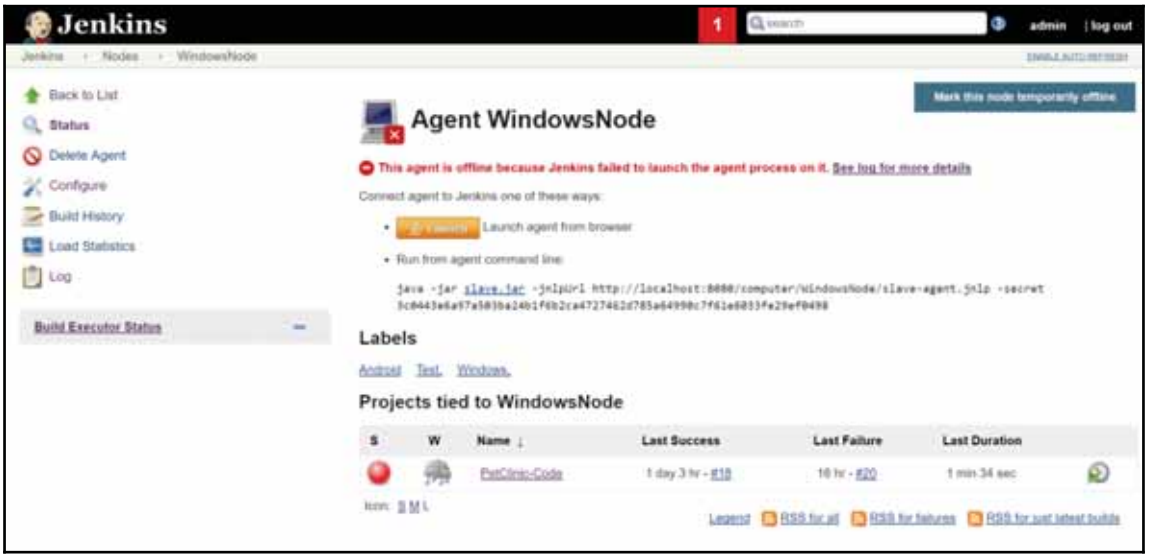
Go to the Agent node again and execute the given command from the Agent node's terminal or command line; then it should be connected:



To configure a job for a specific node, go to **Build job** and click on **Configure**.

In the **General** section, select **Restrict where this project can be run** and provide the label of the Agent node that we have created:

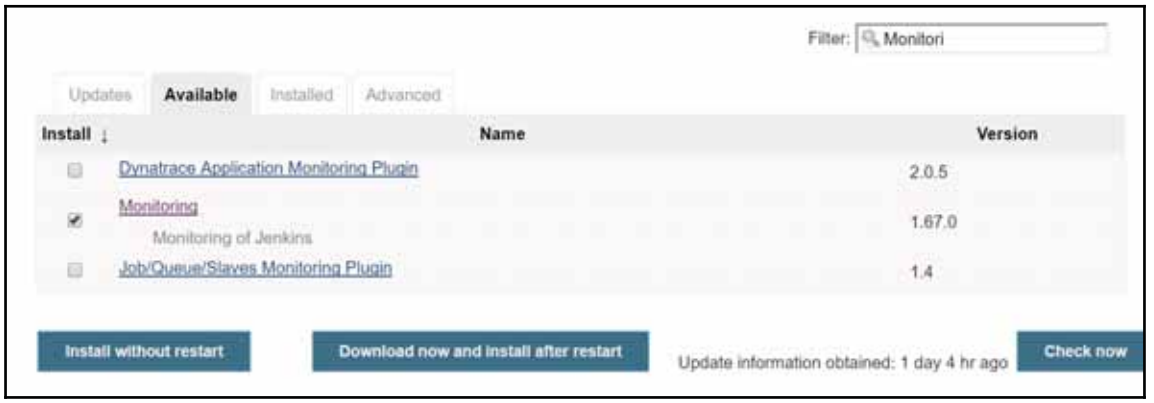
Go to **Manage Nodes** and select the Agent Node to verify the **Build job** associated with it:



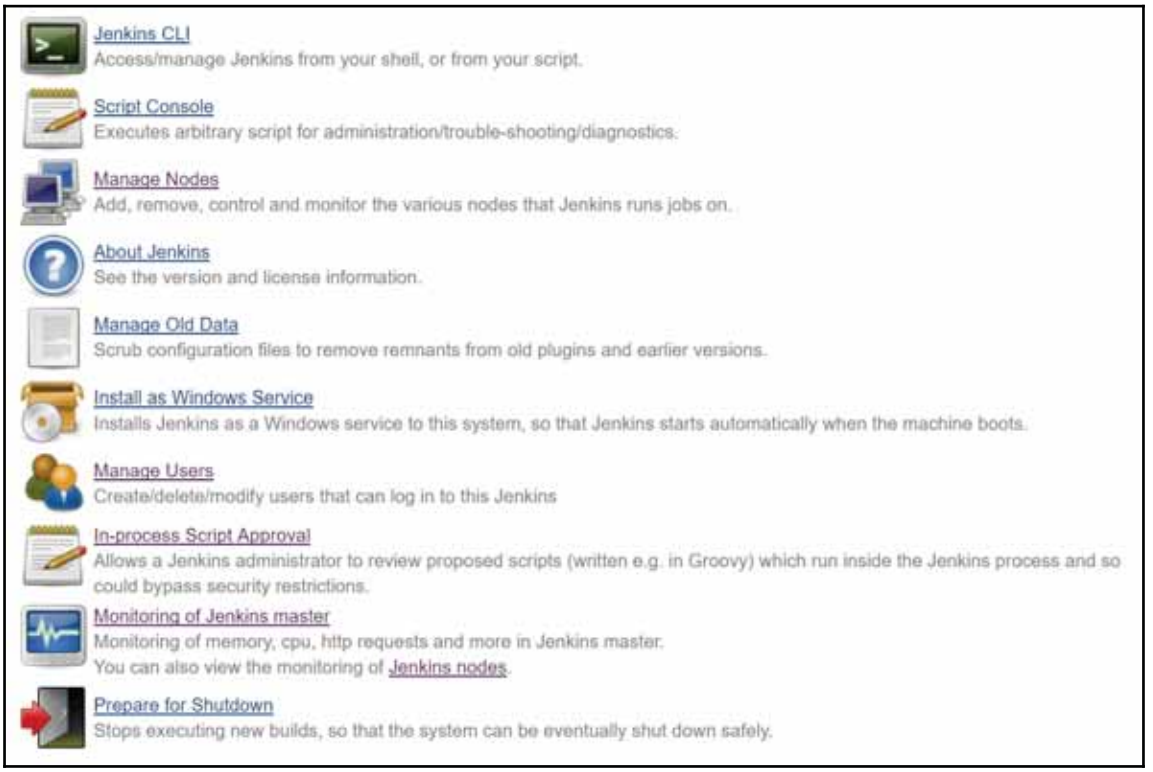
This is how we can create a Master/Agent architecture and distribute the load across agents with only one master available.

Monitoring Jenkins with JavaMelody

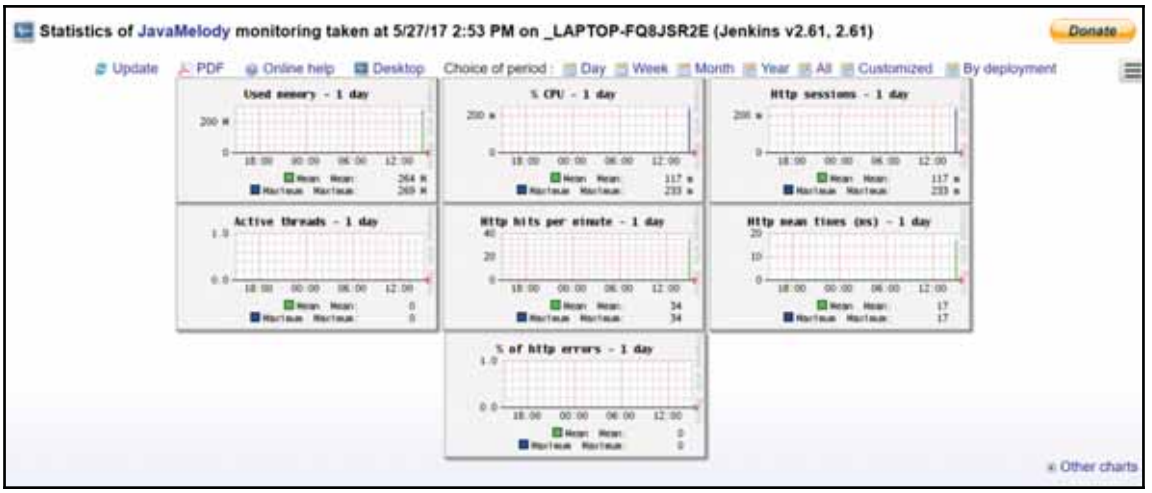
The `Monitoring` plugin provides monitoring for Jenkins with **JavaMelody**. It provides charts for CPU, memory, system load average, HTTP response time, and so on. It also provides details of HTTP sessions, errors and logs, actions for GC, heap dump, invalidate session(s), and so on. Install the **Monitoring** plugin from the Jenkins dashboard:



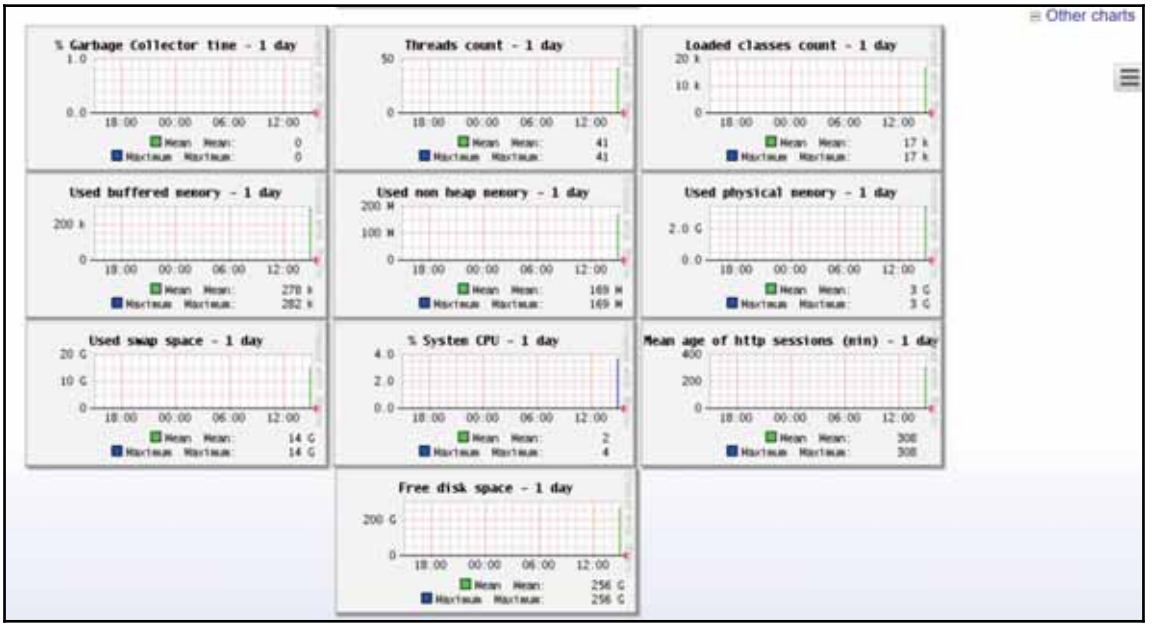
1. On the Jenkins dashboard, click on **Manage Jenkins**. Click on **Monitoring of Jenkins master**, as shown in the following screenshot:



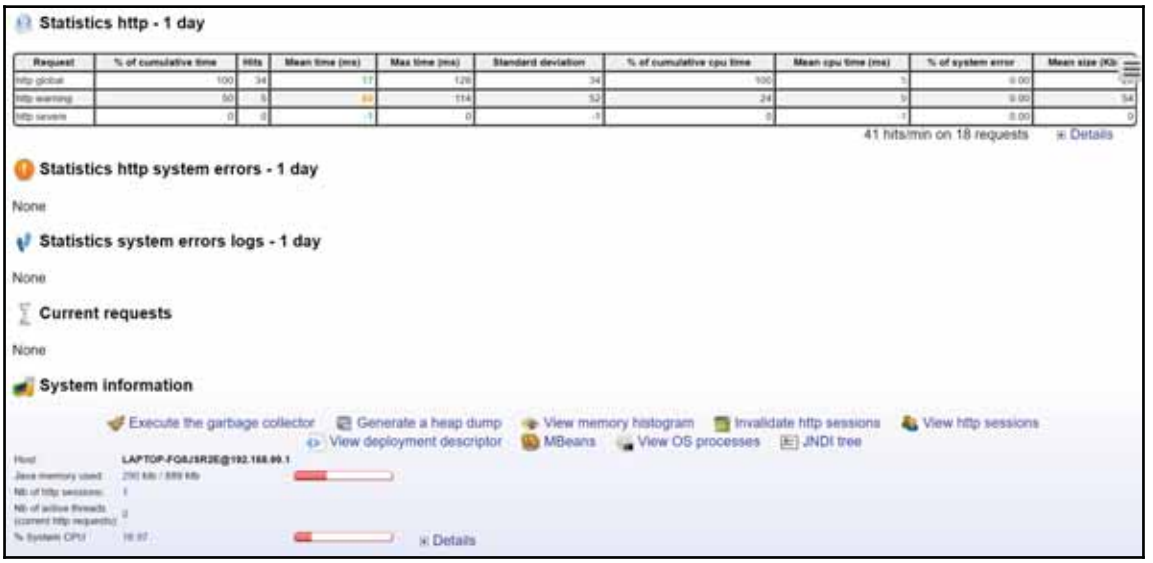
2. It will open the statistics for Jenkins instance monitoring, as shown in the following screenshot. Look at all the statistics:



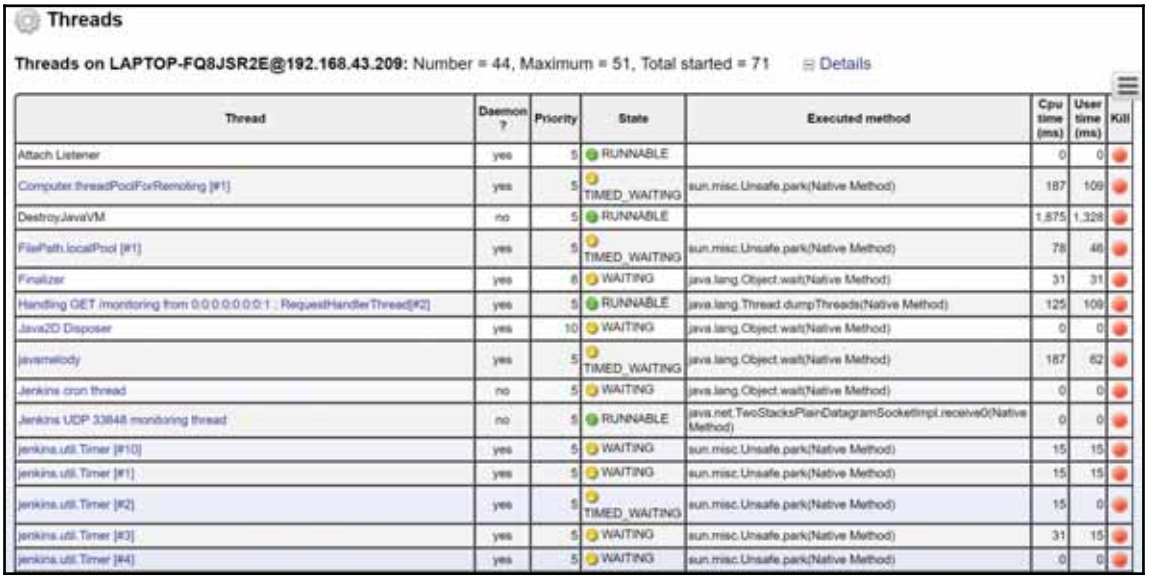
3. Click on **Other charts** and see other details related to **Garbage Collector time**, **Threadcount**, and so on:



4. Scroll down the page and find the statistics for the system errors logs. To get more information, click on the **Details** link of any section. HTTP statistics are as shown in the following screenshot:



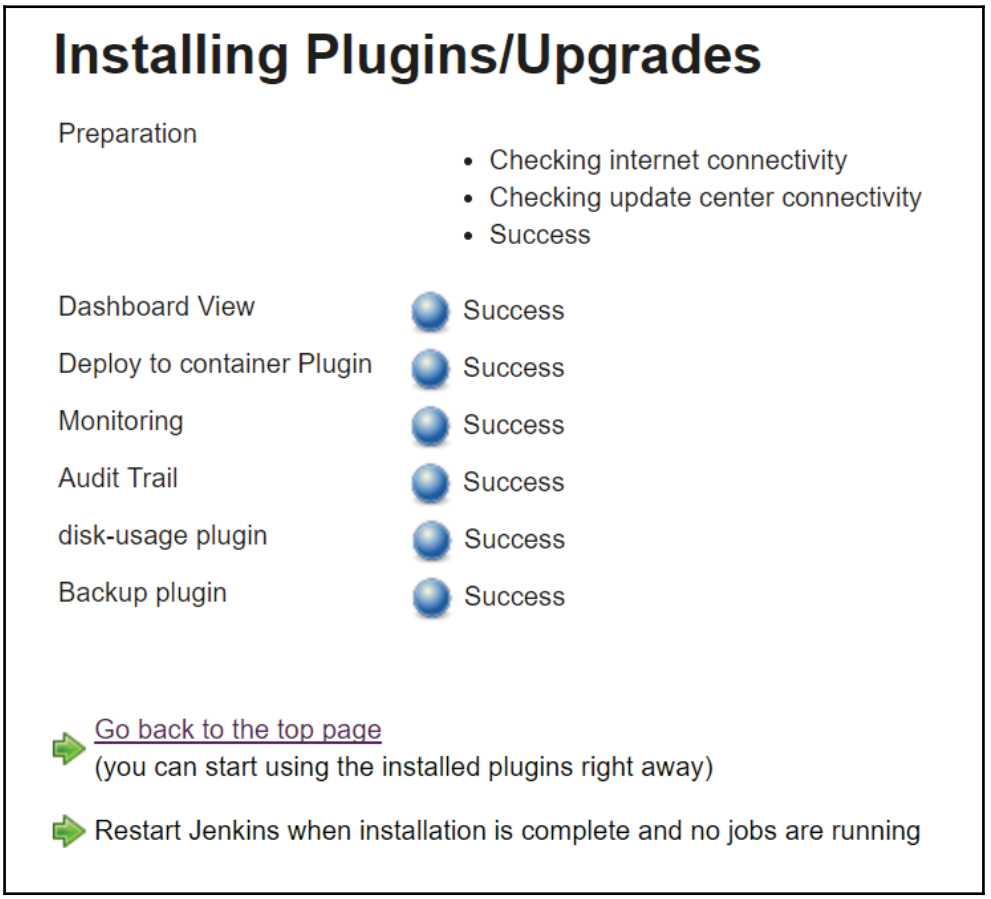
5. Check the details available on **Threads** as well:



In the next section, we will cover details on backing up and restoring `JENKINS_HOME`.

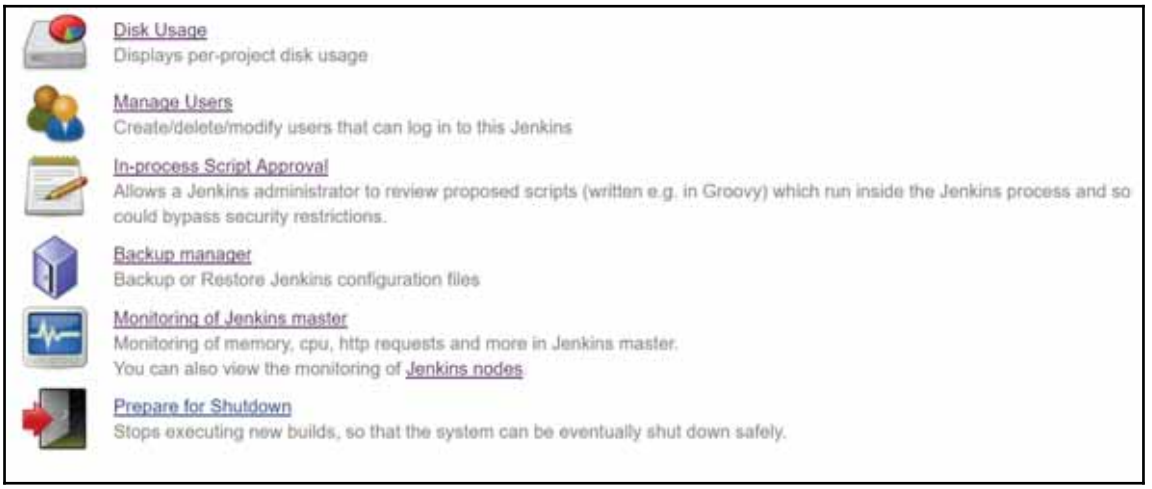
Managing job-specific configurations - backup and restore

Let's install all the important plugins required for the following sections at once:

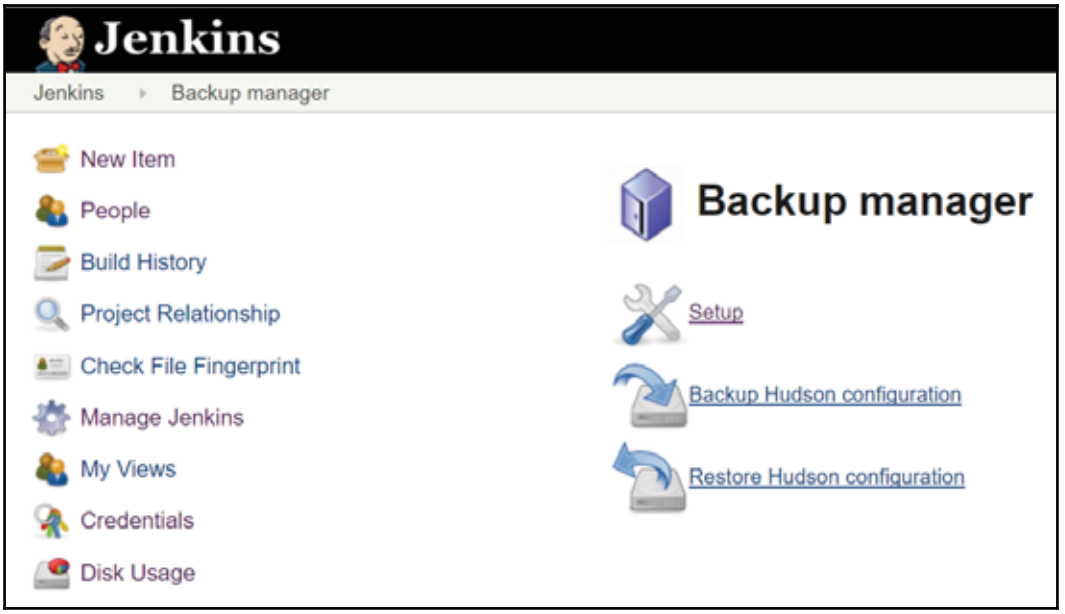


The `Backup` plugin allows us to take a backup of `JENKINS_HOME` and restore it.

1. Go to **Manage Jenkins** and click on **Backup Manager**:



2. Click on **Setup**:



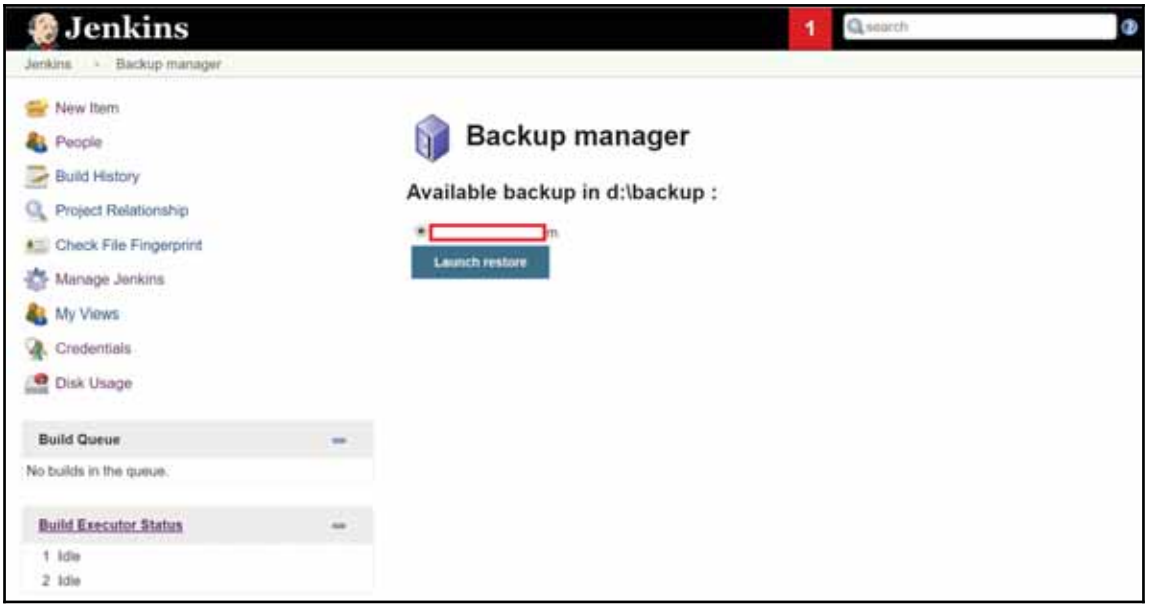
3. Configure **Backup directory**, **Format**, **File name template**, and so on:

4. Click on **Backup Hudson configuration**:

5. Once the backup has been completed **successfully**, verify the logs:

```
Jenkins is going to shut down
[ INFO] Backup started at [05/27/17 15:01:27]
[ INFO] Setting hudson in shutdown mode to avoid files corruptions.
[ INFO] Waiting all jobs end...
[ INFO] Number of running jobs detected : 0
[ INFO] All jobs finished.
[ INFO] Full backup file name : d:\backup\YYYYDDMM_hhmm
[ INFO] Saved files : 12482
[ INFO] Number of errors : 0
[ INFO] Cancel hudson shutdown mode
[ INFO] Backup end at [05/27/17 15:03:51]
[ INFO] [143.935s]
```

6. Go to **Backup Manager** and click on **Restore Hudson** configuration:

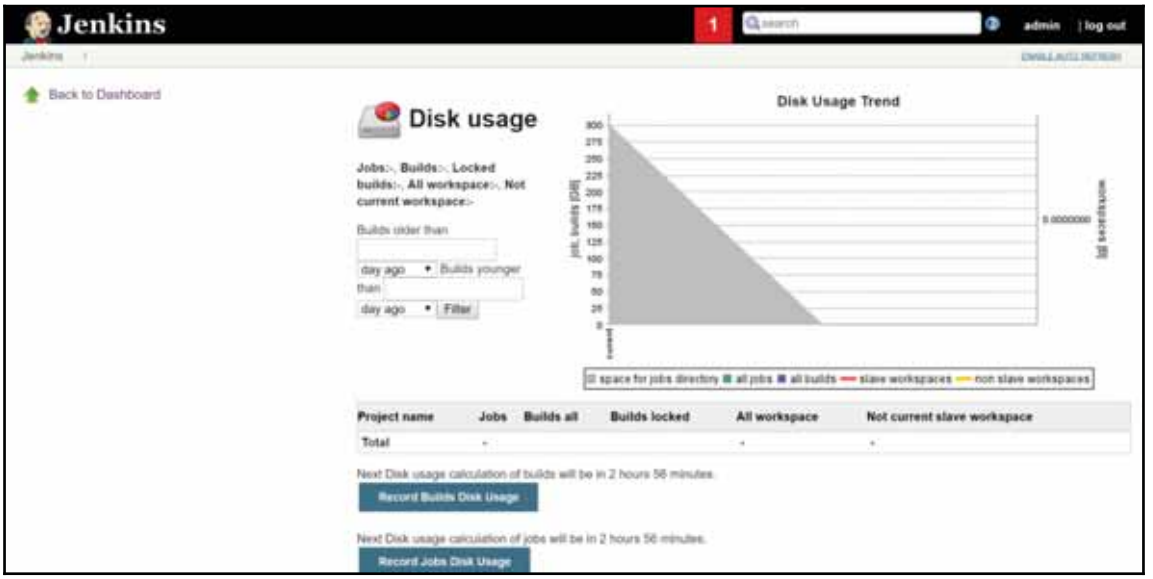


In the next section, we will see the disk usage plugin.

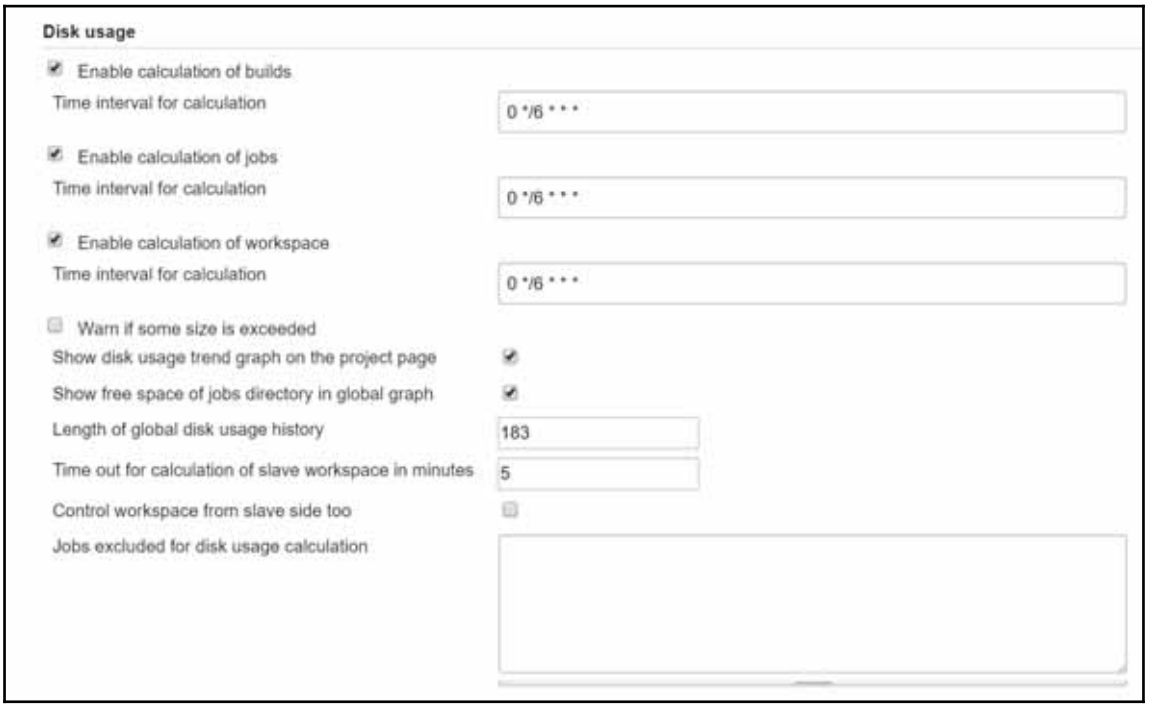
Managing disk usage

This plugin gives details on disk usage for the system where Jenkins is installed.

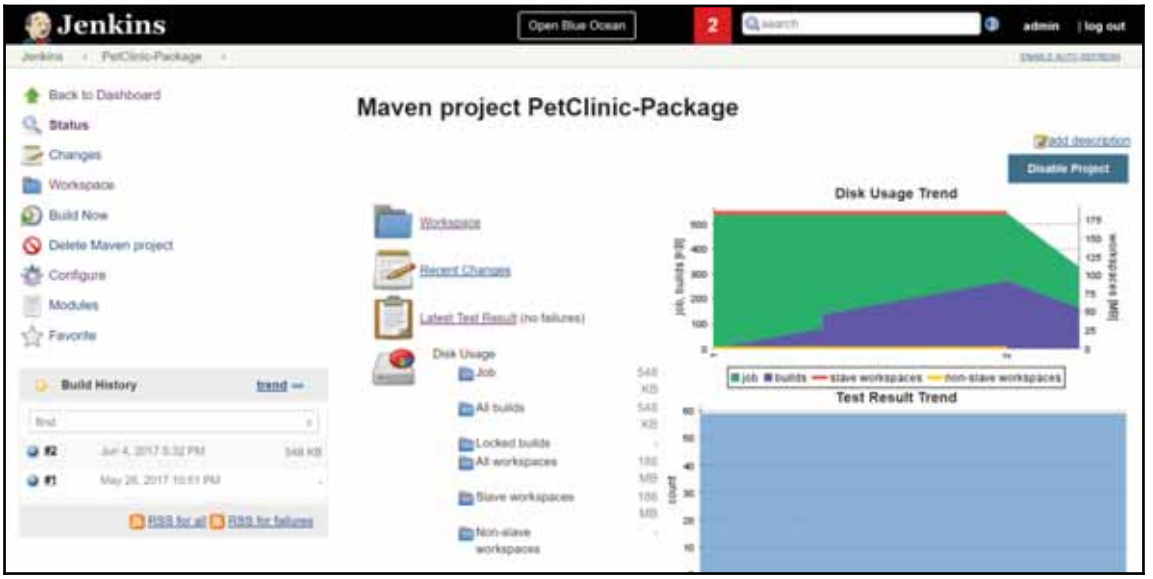
1. Go to **Dashboard** | **Manage Jenkins** | **Disk Usage**:



2. Go to **Manage Jenkins** and click on **Configure System**. Go to the **disk usage** section and click on **Show disk usage trend graph on the project page**:



3. Go to the specific project and see whether the disk usage trend chart is available or not:



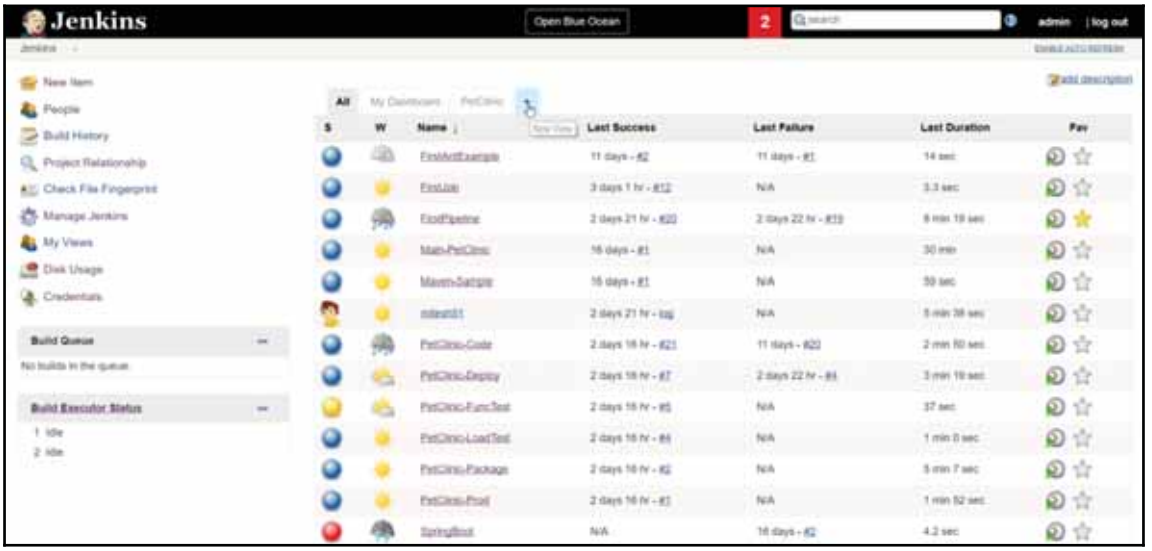
In the next section, we will use the Build Monitor View plugin to keep track of the status and progress of different builds.

Build job-specific monitoring with the Build Monitor plugin

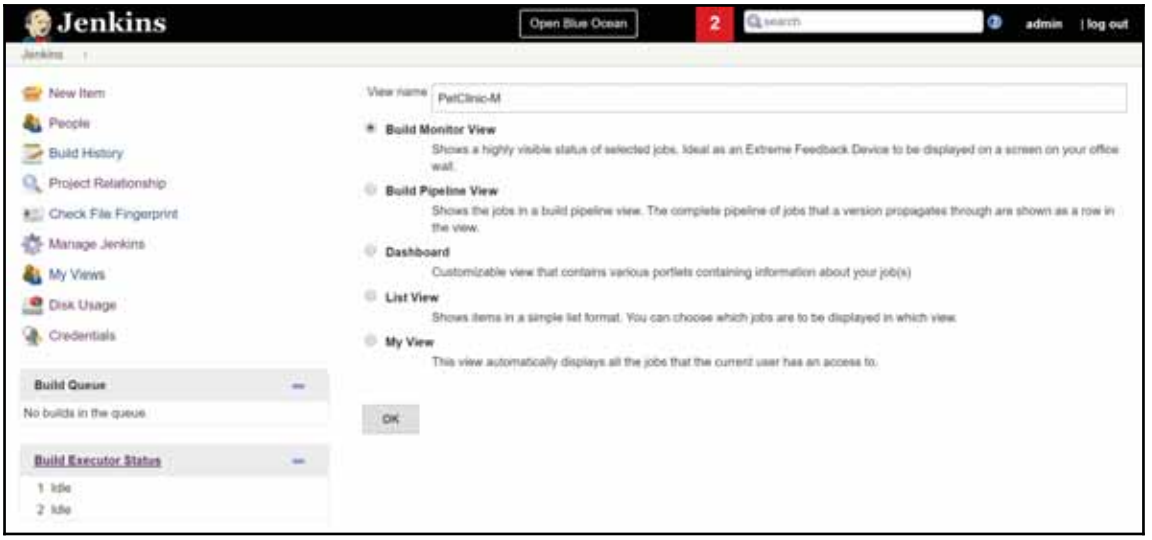
The `Build Monitor` plugin provides a visualization of the status and progress of selected Jenkins jobs. It displays an updated view automatically every couple of seconds, using AJAX. It can easily accommodate different computer screen sizes as well:

1. Go to the **Manage Jenkins** | **Manage Plugins** | **Available** tab. Install the **Build Monitor View** plugin:

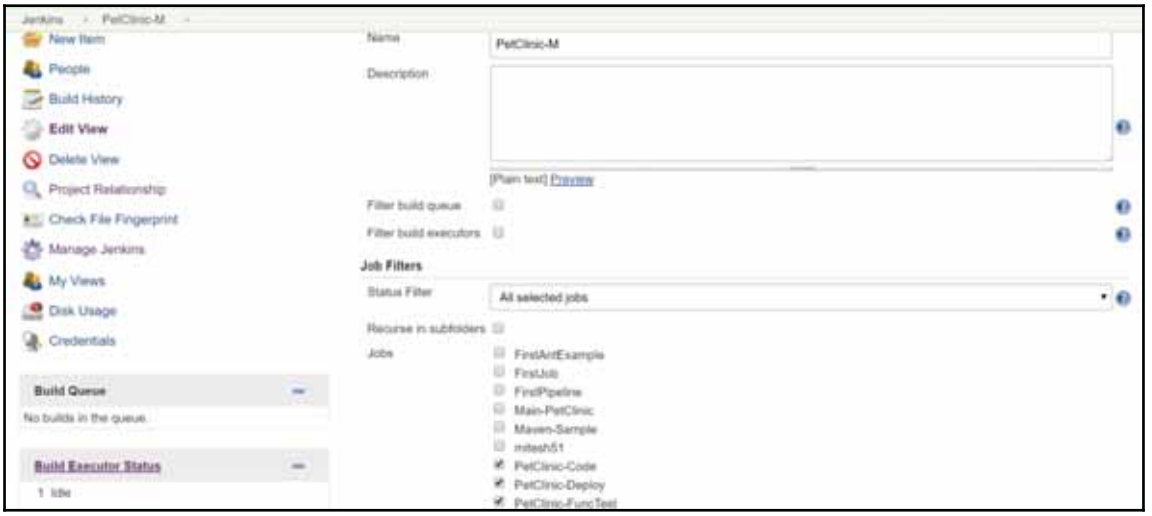
2. Go to the Jenkins dashboard. Click on **New View**:



3. Provide a **View name** and select **Build Monitor View**. Click **OK**:



4. Select the jobs to be displayed in the newly created **Build Monitor View**.
5. Click **Save**:

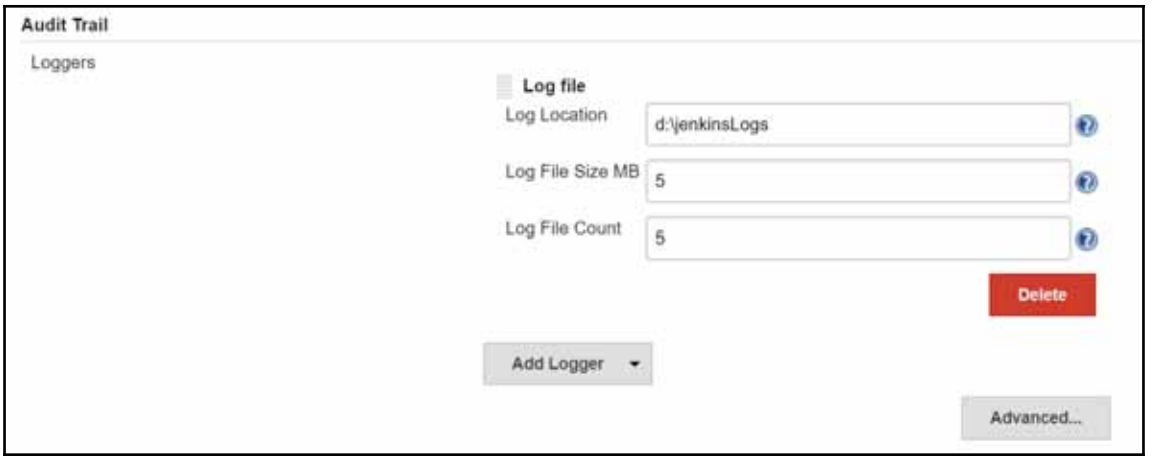


6. Verify the status of the **Build Monitor View** in the Jenkins dashboard:

In the next section, we will discuss the **Audit trail** plugin in brief.

Audit Trail plugin-overview and usage

Keep a log of who executed specific Jenkins operations, such as configuring jobs and so on. On the Jenkins configuration page, we need to configure the log file location and settings, such as file size and number of rotating log files:

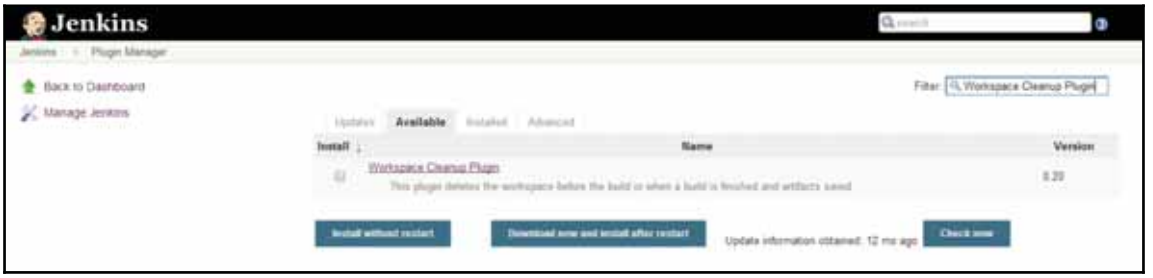


In the next section, we will discuss the workspace cleanup plugin in brief.

Workspace Cleanup plugin

The **Workspace Cleanup** plugin is used to delete the workspace from Jenkins before the build, or when a build is finished and artifacts saved. If we want to start a Jenkins build with a clean workspace, or we want to clean a particular directory before each build, then we can effectively use this plugin. Different options are available for deleting workspaces.

Install the plugin from the Jenkins dashboard:



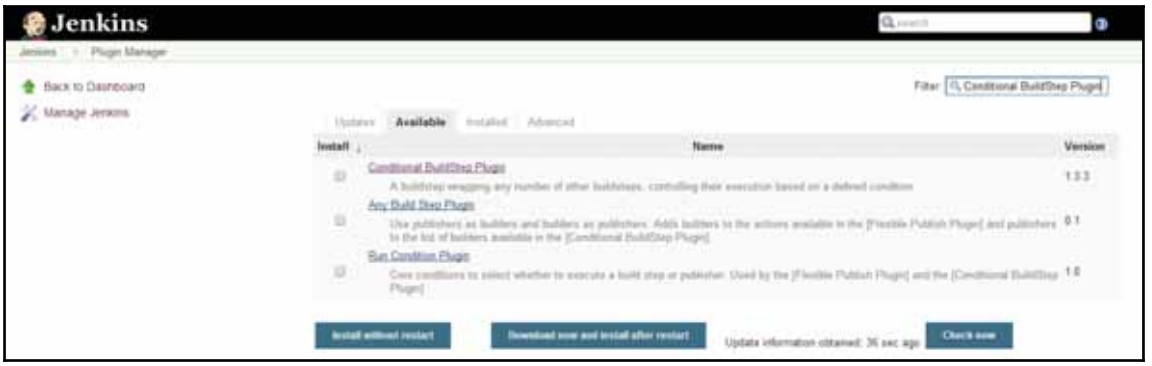
We can apply patterns for files to be deleted based on the status of the build job. We can add post-build actions for workspace deletion:

For more details on the **Workspace Cleanup** plugin, visit `https://plugins.jenkins.io/ws-cleanup`.

Conditional Build Step plugin

The Conditional Build Step plugin allows us to wrap any number of other build steps, controlling their execution based on a defined condition.

Install the plugin from the Jenkins dashboard:

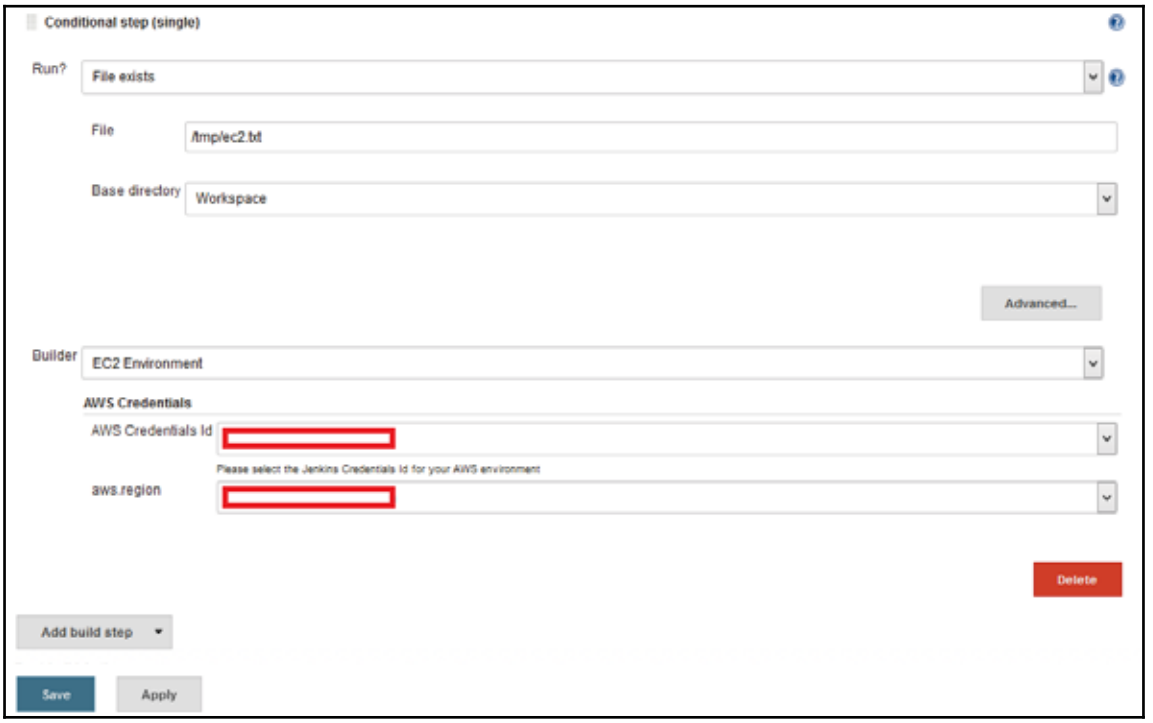


This plugin defines a few core run conditions, such as:

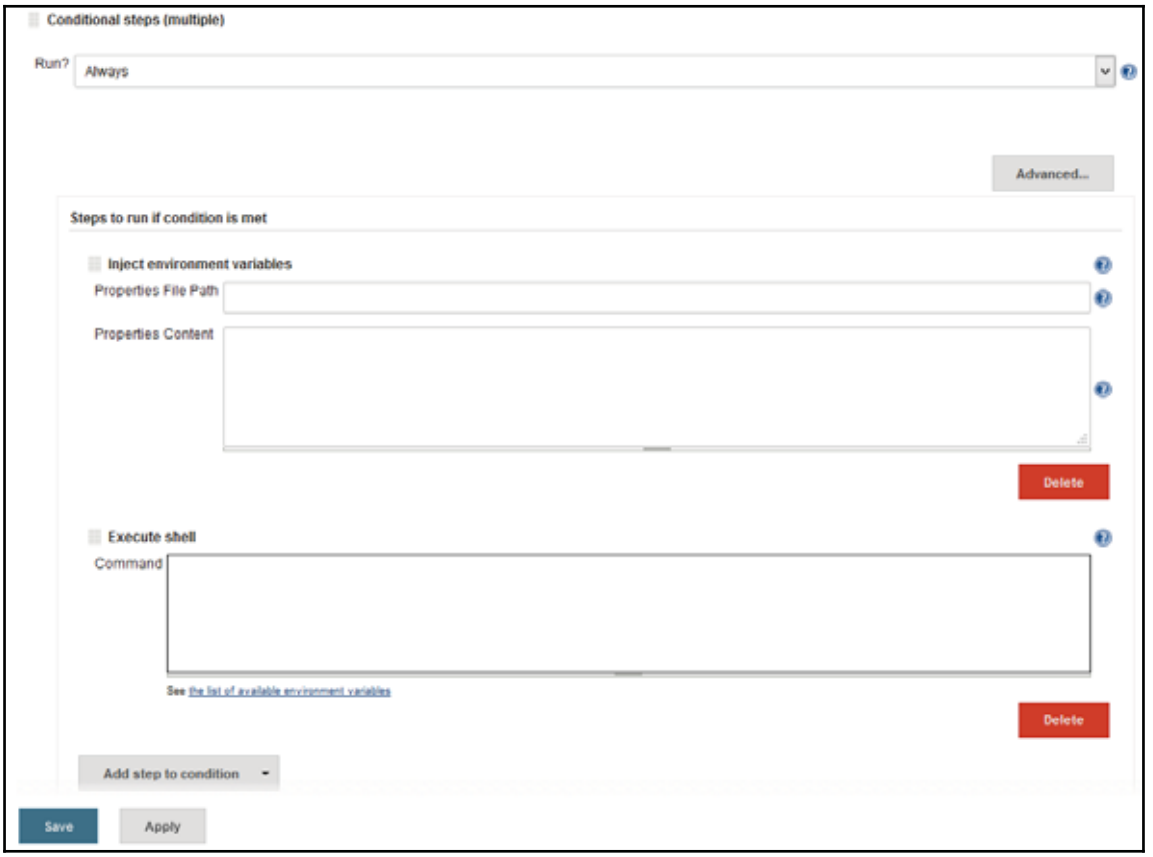
- **Always/Never**: To disable a build step from the job configuration
- **Boolean condition**: To execute the step if a token expands to a representation of true
- **Current status**: To execute the build step if the current build status is within the configured/specific range
- **File exists/Files match**: To execute the step if a file exists or matches a pattern
- **Strings match**: To execute if the two strings are the same
- **Numerical comparison**: To execute the build step depending on the result of comparing two numbers
- **Regular expression match**: To execute the build step depending on the matching of regular expression
- **Provide a regular expression and a label**: To execute the build step if the expression matches the label

- **Time/Day of week**:To execute the build job during a specified period of the day, or day of the week
    - **And/Or/Not**: Logical operations to enable the combining and sense inversion of run conditions
    - **Build Cause**: To execute the build step depending on the cause of the build, such as triggered by timer, user, scm-change, and so on
    - **Script Condition**: Utilize a shell script to decide whether a step should be skipped
    - **Windows Batch Condition**: Utilize Windows Batch to decide whether a step should be skipped

Select **Conditional step (single)** from **Add build step**:



Select **Conditional steps (multiple)** from **Add build** step. We can add multiple steps to a condition in this conditional step:

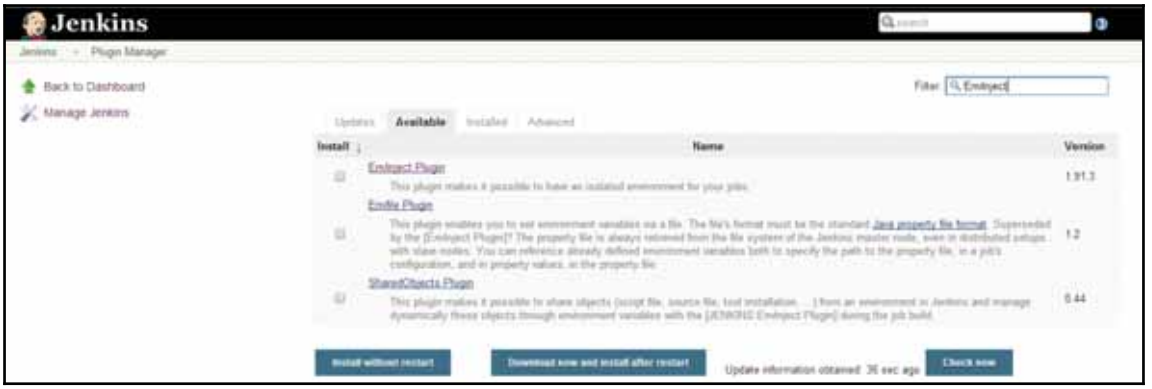


For more details on Conditional Build Step plugin, visit `https://wiki.jenkins-ci.org/display/JENKINS/Conditional+BuildStep+Plugin`.

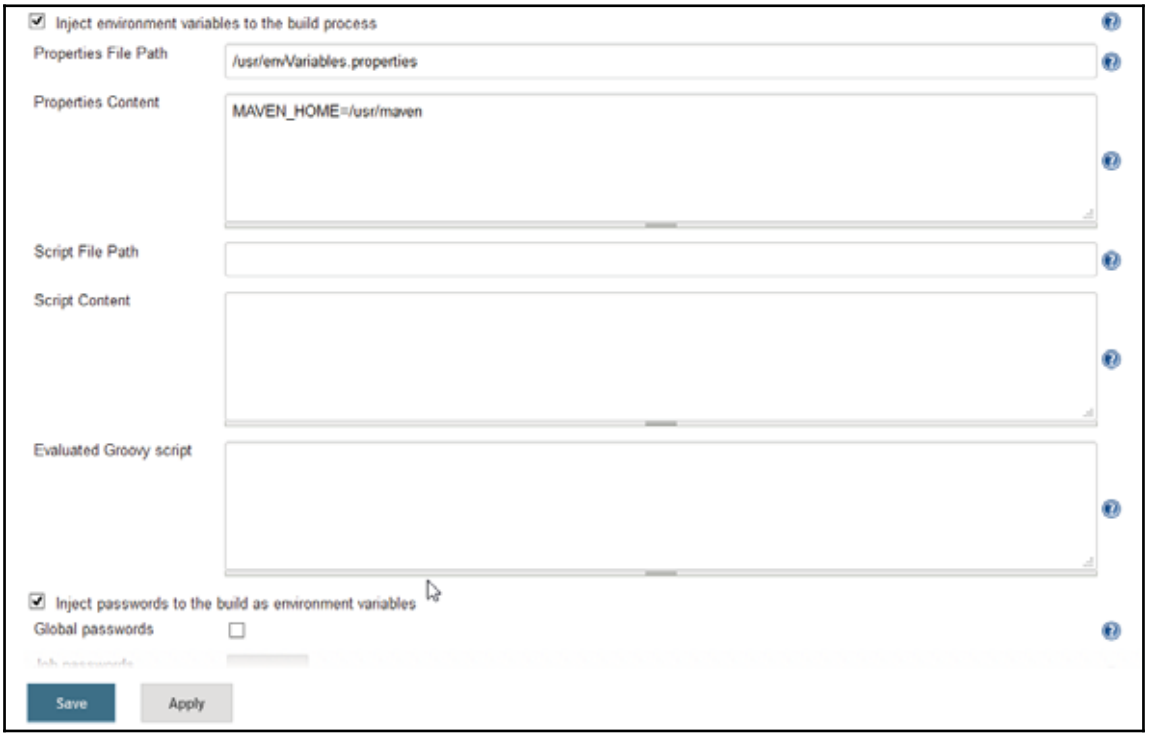
EnvInject plugin

We know that different environments such as dev, test, production, and so on require different configurations.

Install the plugin from the Jenkins dashboard:



The `EnvInject` plugin provides a facility to have an isolated environment for different build jobs. The `EnvInject` plugin injects environment variables at node startup, before and/or after a SCM checkout for a run, as a build step for a run, and so on. **Select Inject environment variables to the build process** specific to the build job:



For more details on EnvInject plugin, visit
`https://wiki.jenkins-ci.org/display/JENKINS/EnvInject+Plugin`.

Summary

In this chapter, we have seen how to configure a Master/Agent architecture to distribute the workload and to avoid a single point of failure; however, Jenkins does not have a high availability story yet and if the master goes down then it will still be a single point failure.

We have also seen different plugins that can enhance the monitoring and management of Jenkins, as well as plugins that can be utilized to extend the functionality of Jenkins. All these plugins and the Master/Agent architecture help to make automation more effective and broaden the scope of different minor innovations that can be done in automating different activities.

In the next chapter, we will see how to configure security in Jenkins. We will focus on user management, role-based access, and project-based access using Jenkins.

9
Security in Jenkins

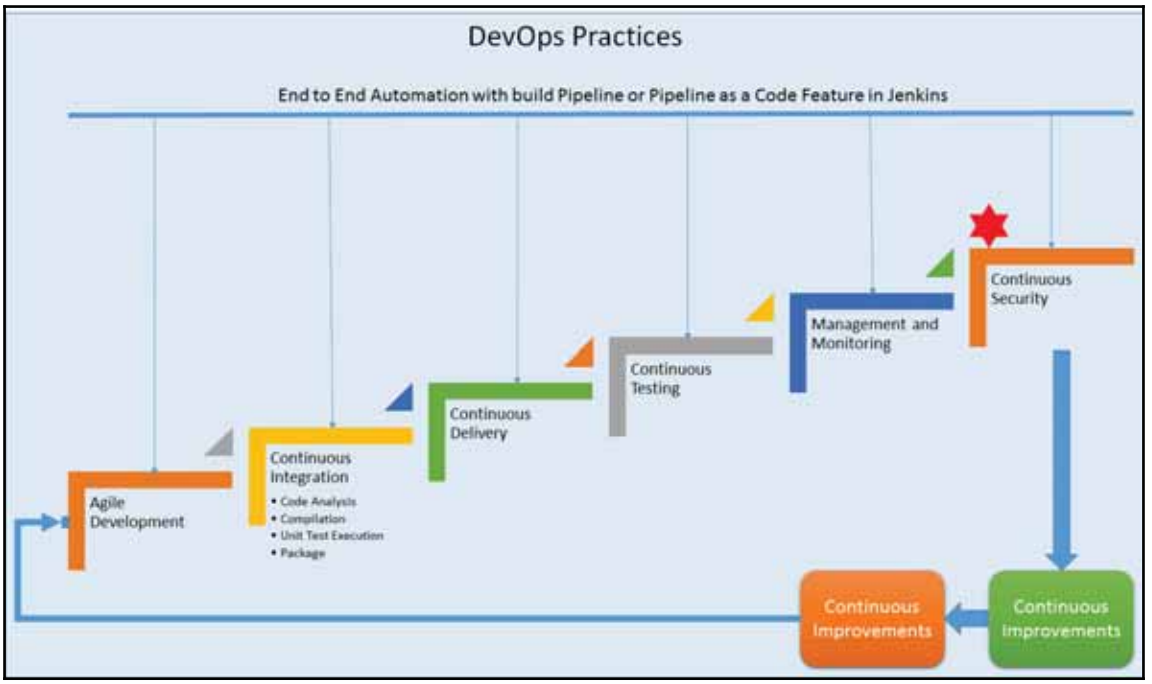
Up to now we have seen static code analysis, Continuous Integration, Continuous Delivery/Deployment, Continuous Testing, the orchestration of build jobs using the build pipeline plugin and pipeline as a code, and the management and monitoring of Jenkins resources.

This chapter will cover the security management options available in Jenkins.

It will help to perform user management, authentication, and authorization, including matrix-based security and role-based access. We will cover the following major topics in this chapter:

- User management
- Role-based security
- Project-based security

In this chapter, we will cover continuous security practices as a part of our DevOps journey:



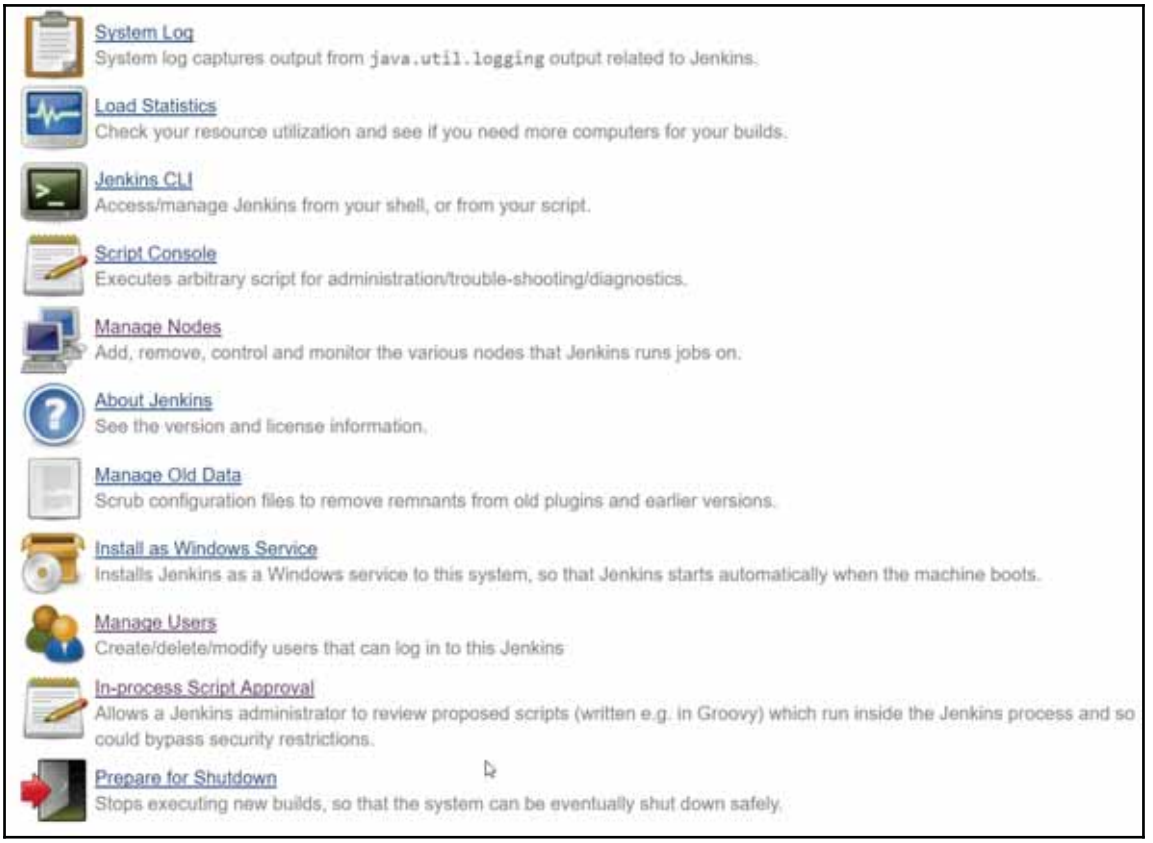
Thread details available on Jenkins Master using Jenkins Monitoring Plugin

At the end of this chapter, we will know how to configure role-based and project-based security in Jenkins, as well as user management.

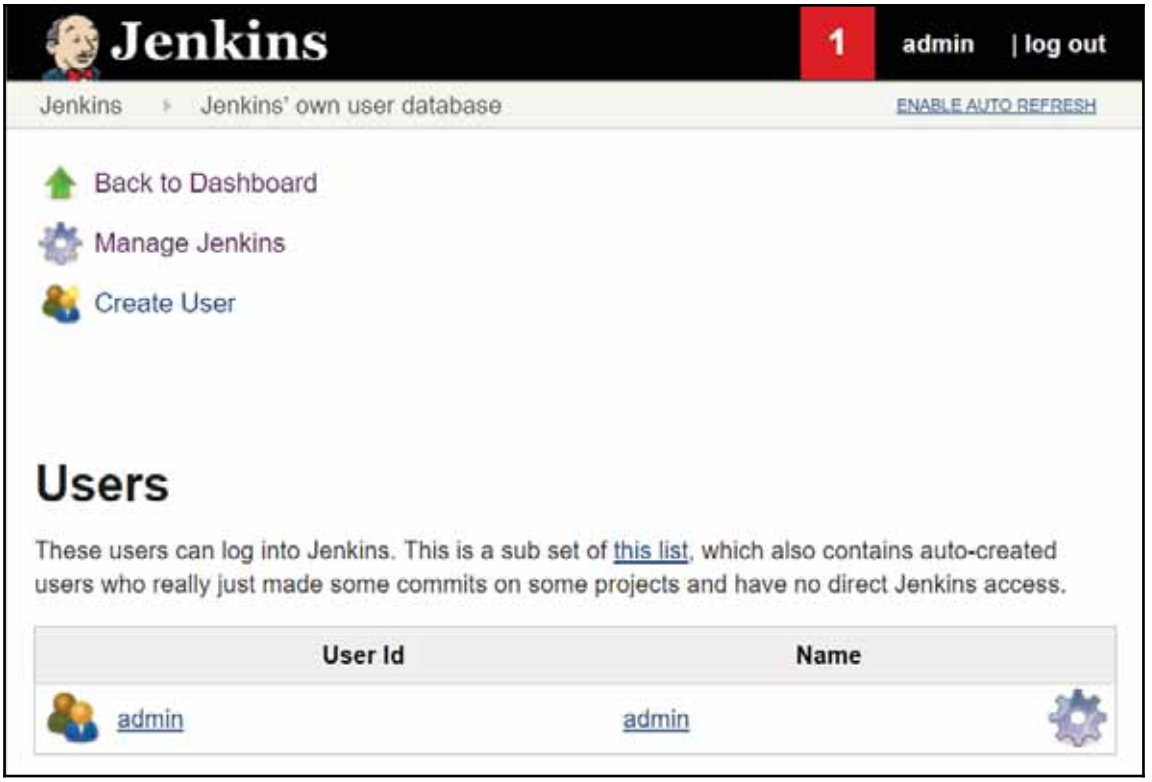
User management

In this section, we will cover how to manage multiple users. With user management, we can provide access to Jenkins for multiple users and provide them role-based or project-based access when it is required.

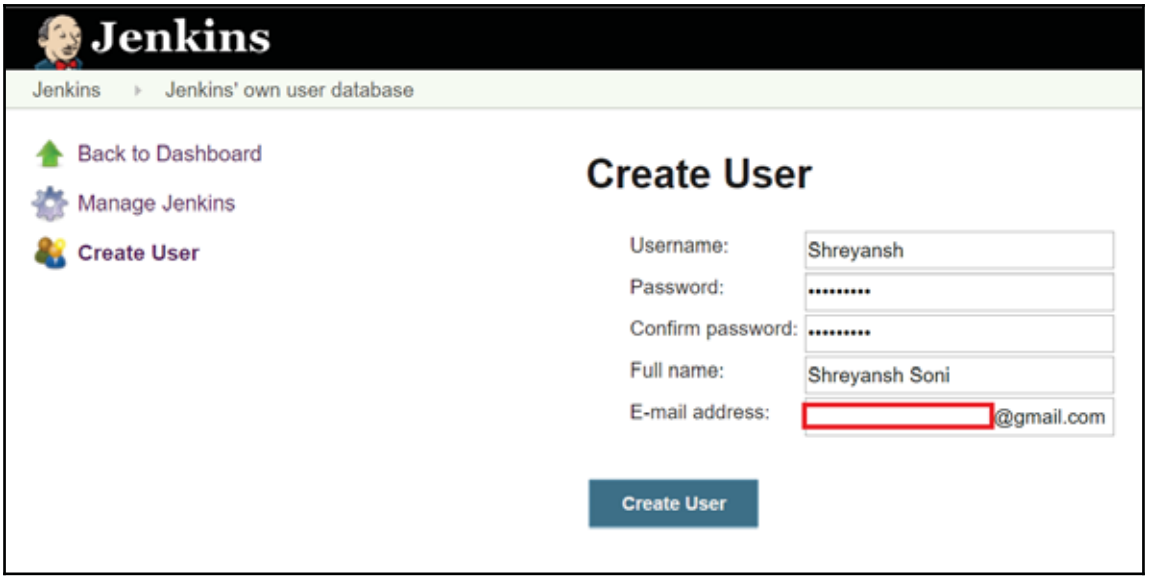
1. Go to **Manage Jenkins** and click on **Manage Users**:



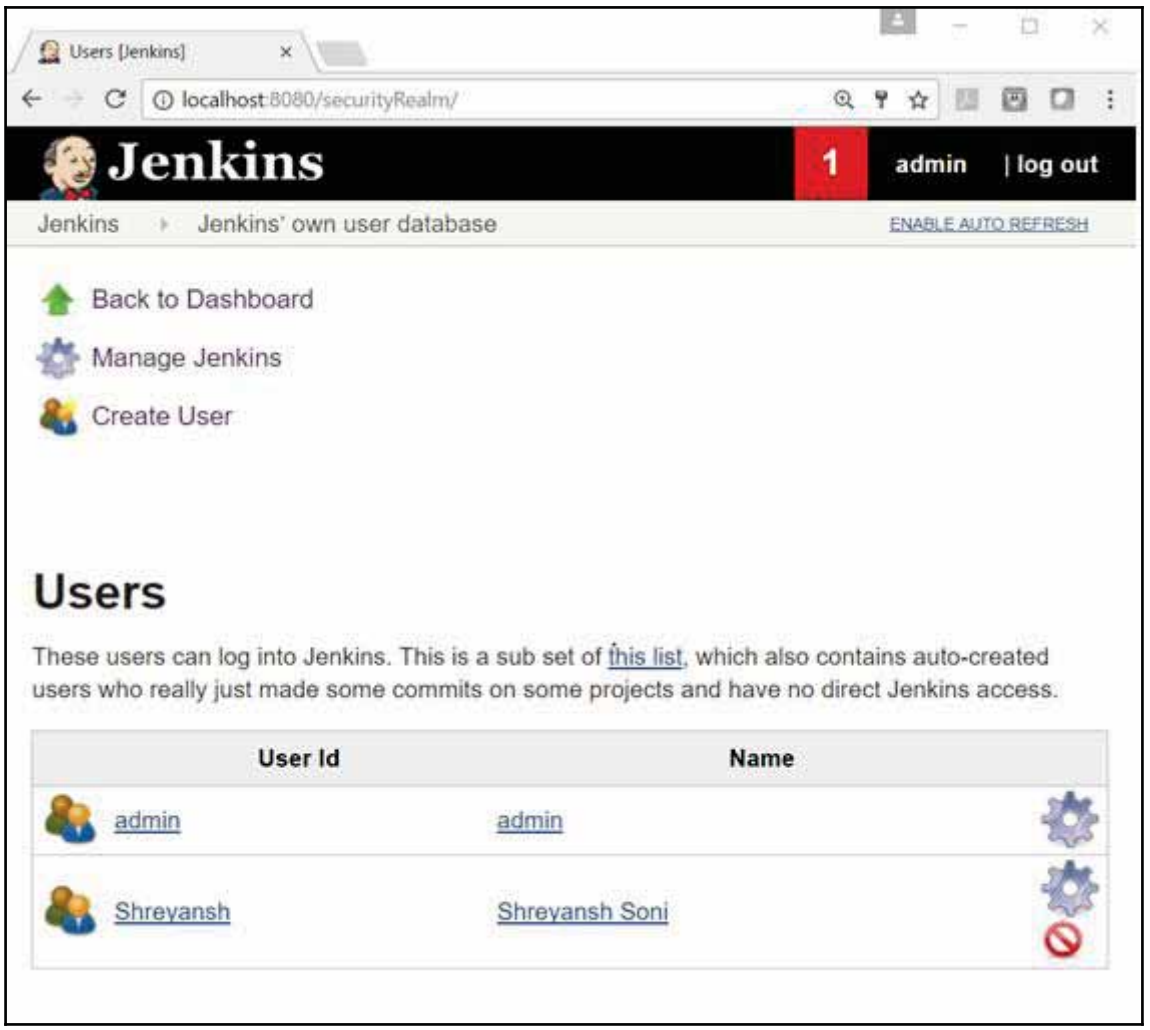
2. Check the existing admin user available in Jenkins:



3. Click on the **Create User** link and provide details:

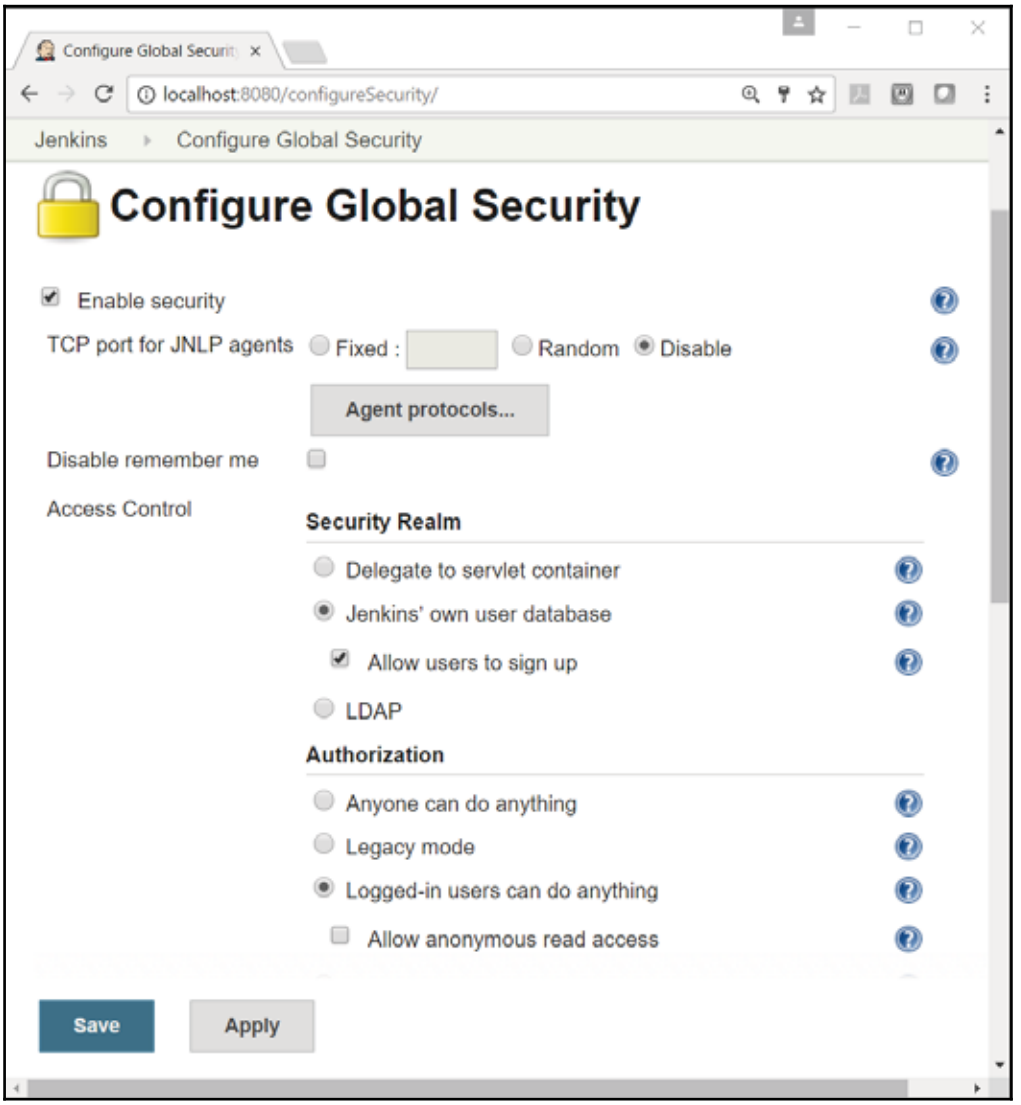


4. Check the list of users in **Manage Jenkins**| **Manage Users**:



5. To allow sign up and access to only logged in users, go to **Manage Jenkins** | **Configure Global Security**.

6. In the **Access Control** section, click on **Jenkins' own user database** and select **Allow users to sign up**:



So, this is how we can create users and allow users to sign up to access Jenkins.

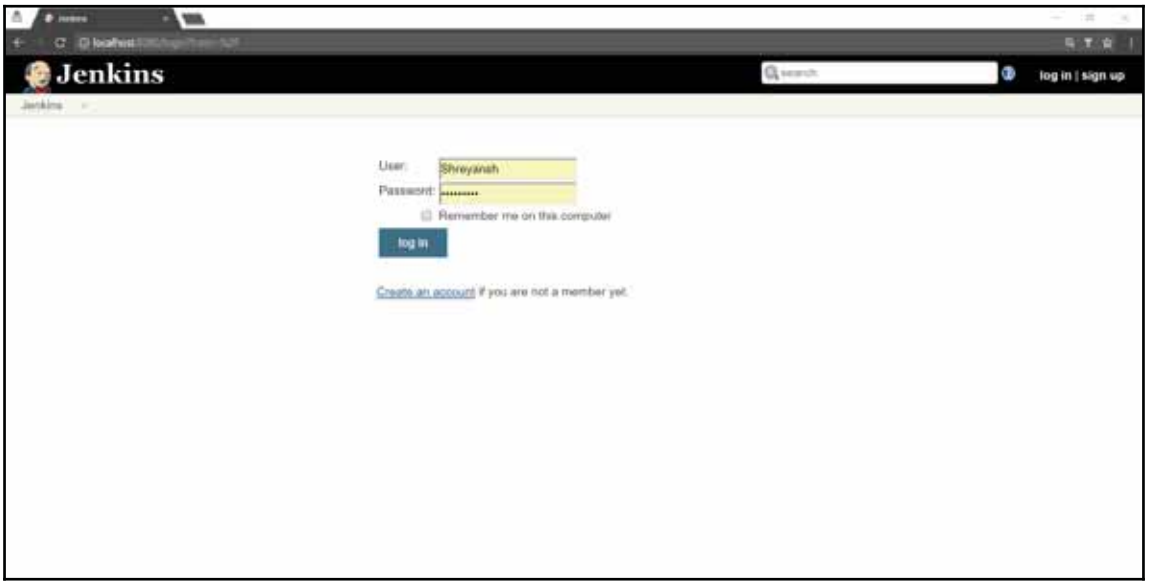
Role-based security

In the **Authorization** section, we can configure matrix-based security so we can configure who can do what. We can configure the predefined roles available in Jenkins.

1. Select **Matrix-based security** and type a name in the User/group to add box. Make sure that you give access to Admin before saving it, or the Jenkins account will be locked out:

2. Type the name of our newly created user in the **User/group to add** text box, click on **Add**, and provide all the required rights. We can do the same things for different users.
3. Click on **Save**:

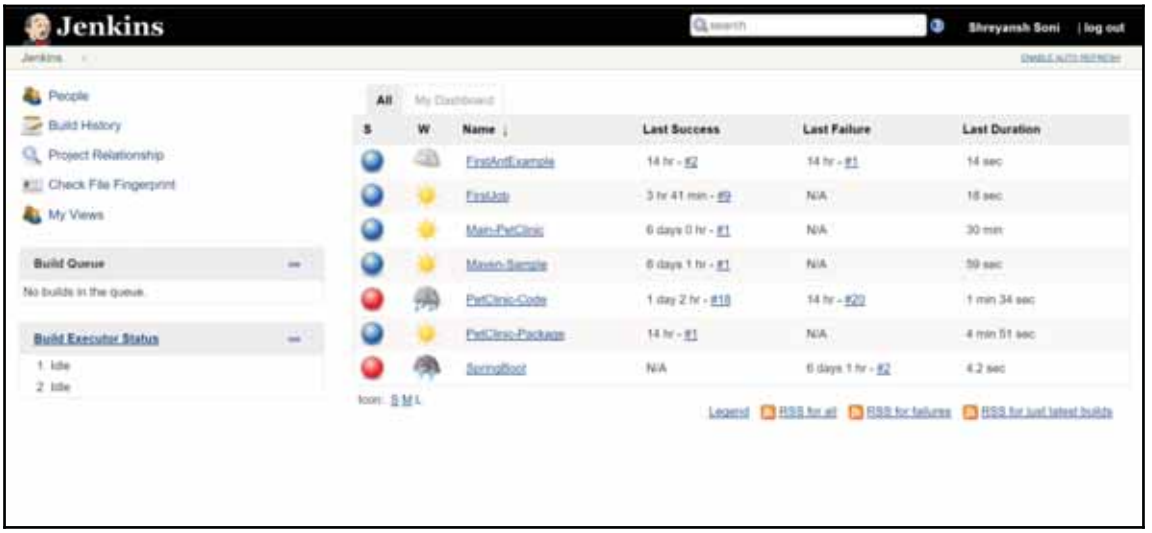
4. To verify access has been granted, open a new incognito window in your browser and log in with the username and password of the newly created user:



5. Verify that limited access is available to the new user, and that the **New Item** and **Manage Jenkins** links are not available:

6. Now go to **Manage Jenkins** | **Global Security Configuration**. Allow **Read** rights in the **Job** category for the user **Shreyansh**.
7. Click on **Save**:

8. Go to the incognito window that we opened before and refresh the page. Now we have read access to the **Jobs** available in Jenkins:



9. We can see the jobs, but we can't execute them as rights are not available:

This is how we can manage users and authorization in Jenkins. In the next section, we will see how to give project-based access.

Project-based security

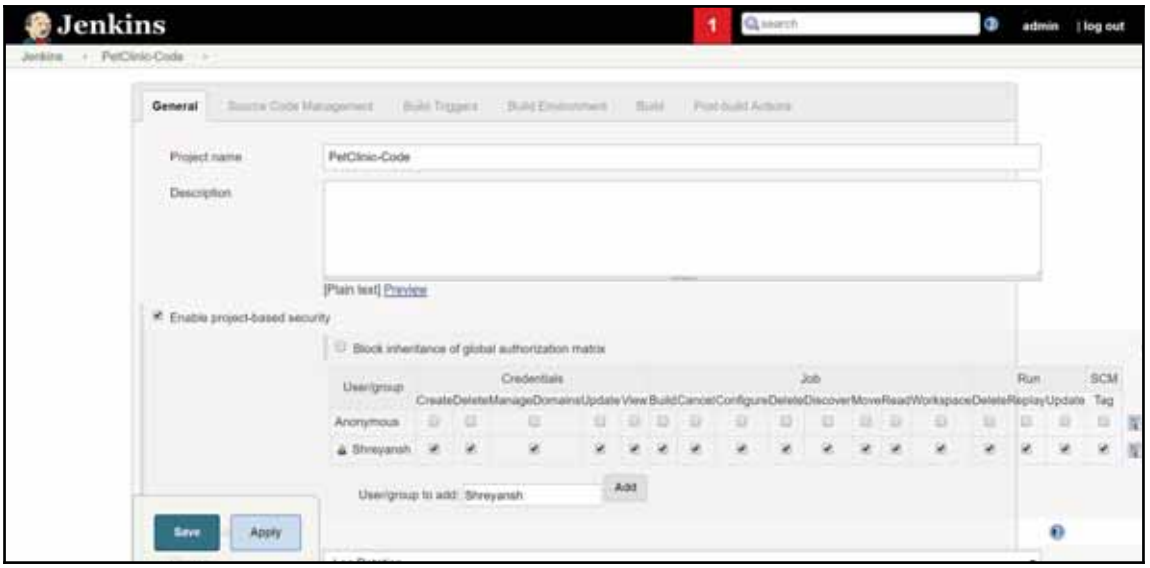
Project-based Matrix Authorization Strategy is an extension to **Matrix-based security**. It allows an access control list matrix to be defined for each project. This feature is very useful where we want to give access to specific jobs to specific users, so the security of Jenkins is not compromised.

1. Go to Manage **Jenkins** | **Global Security Configuration**. In the **Authorization** section, select **Project-based Matrix Authorization Strategy**.
2. Give admin all rights and **Save**:

3. Go to the incognito window where we logged in using the credentials for Shreyansh.
4. Refresh the page and you will get **Access Denied**. The reason is we haven't given any rights to **Shreyansh** in **Project-based Matrix Authorization Strategy**:

5. We need to provide overall read rights so **Shreyansh** can access the Jenkins dashboard:

6. Now, go to the individual build job as an admin and select **Enable project-based security** in the job configuration page.
7. Add **Sreyansh** as a **User** and click on **Save**:



8. Now, go to the incognito window where **Shreyansh** is logged in and refresh the page. We can see one job that we have configured to give access to **Shreyansh**:

9. Click on **Build** and verify all the rights are available to the user **Shreyansh**:

We have finished user management, role-based access, and project-based access in Jenkins as a part of securing Jenkins.

Summary

Up to now, we have covered static code analysis, Continuous Integration, Continuous Delivery/Deployment, Continuous Testing, the orchestration of build jobs using the build pipeline plugin and pipeline as a Code, the management and monitoring of Jenkins resources, security in Jenkins in the form of user management, role-based access, and project-based access forusers.

By covering all these topics, we have ensured that we cover almost all major aspects of application lifecycle management. It is not that only Jenkins can deliver what we have achieved up to now. It is not about tools only. It is about people, processes, and tools in DevOps implementations. Another basic but very important thing is to remember that DevOps is not a tool, technology, model or framework; *DevOps is a CULTURE*.

Index

E

F

G

J

K

L

M

P

Q

R

S

T

U

W