Community Experience Distilled

Learning Continuous Integration with Jenkins

A beginner's guide to implementing Continuous Integration and Continuous Delivery using Jenkins

Nikhil Pathania

[PACKT] PUBLISHING open source* community experience distilled

Learning Continuous Integration with Jenkins

A beginner's guide to implementing Continuous Integration and Continuous Delivery using Jenkins

Nikhil Pathania

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Learning Continuous Integration with Jenkins

First published: May 2016

Production reference: 1260516

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

ISBN 978-1-78528-483-0

www.packtpub.com

Credits

Author
Nikhil Pathania

Reviewer
Thomas Dao

Commissioning Editor
Sarah Crofton

Acquisition Editor
Nikhil Karkal

Content Development Editors
Sumeet Sawant
Preeti Singh

Technical Editor
Siddhi Rane

Copy Editors
Roshni Banerjee
Rashmi Sawant

Project Coordinator
Shweta H Birwatkar

Proofreader
Safis Editing

Indexer
Hemangini Bari

Graphics
Abhinash Sahu

Illustrations
Nikhil Pathania

Production Coordinator
Melwyn Dsa

Cover Work
Melwyn Dsa

About the Author

Nikhil Pathania is a DevOps consultant at HCL Technologies Bengaluru, India. He started his career in the domain of software configuration management as an SCM Engineer and later moved on to various other tools and technologies in the field of automation and DevOps. In his career, he has architectured and implemented Continuous Integration and Continuous Delivery solutions across diverse IT projects. He enjoys finding new and better ways to automate and improve manual processes.

Before HCL Technologies, he worked extensively with retail giant Tesco and Wipro Technologies.

First and foremost my beautiful wife, Karishma, without whose love and support this book would not have been possible.

I would like to thank Nikhil Karkal for bringing me this wonderful opportunity to write a book on Jenkins and for helping me in the preliminary stages of the book.

A great thanks to Thomas Dao, who provided me with valuable feedback throughout the writing process.

Most importantly, a special thanks to the following people who worked hard to make this book the best possible experience for the readers: Siddhi Rane, Preeti Singh, Sumeet Sawant, and the whole Packt Publishing technical team working in the backend.

And finally, a great thanks to the Jenkins community for creating such a wonderful software.

About the Reviewer

Thomas Dao has worn many hats in IT from Unix administration, build/release engineering, DevOps engineering, Android development, and now a dad to his bundle of joy, Carina. He also enjoys being the organizer of the Eastside Android Developers GDG meetup group. He can be reached at `tom@tomseattle.com`.

www.PacktPub.com

eBooks, discount offers, and more

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 customercare@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.

PACKTLIB™

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

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

Table of Contents

Preface

In the past few years, the agile model of software development has seen a considerable amount of growth around the world. There is a huge demand for a software delivery solution that is fast and flexible to frequent amendments, a specially in the e-commerce sector. As a result, Continuous Integration and Continuous Delivery methodologies are gaining popularity.

Whether small or big, all types of project are gaining benefits, such as early issue detection, avoiding bad code into production, and faster delivery, which lead to an increase in productivity.

This book, *Learning Continuous Integration with Jenkins,* serves as a step-by-step guide to setting up Continuous Integration, Continuous Delivery, and Continuous Deployment systems using hands-on examples. The book is 20% theory and 80% practical. The book starts by explaining the concepts of Continuous Integration and its significance in the agile world with a complete chapter dedicated to it. Users then learn how to configure and set up Jenkins. The first three chapters prepare the readers for the next important chapters that deal with setting up of Continuous Integration, Continuous Delivery, and Continuous Deployment.

What this book covers

Chapter 1, *Concepts of Continuous Integration*, has an account of how some of the most popular and widely used software development methodologies gave rise to Continuous Integration. It is followed by an in-depth explanation of the various requirements and best practices of Continuous Integration.

Chapter 2, *Setting up Jenkins*, is a step-by-step guide that is all about installing Jenkins across various platforms and particularly on the Apache Tomcat server.

Chapter 3, *Configuring Jenkins*, is an overview of how Jenkins looks and feels with an in-depth explanation of its important constituents. It is followed by a step-by-step guide to accomplishing some of the basic Jenkins administration tasks.

Chapter 4, *Continuous Integration Using Jenkins – Part I*, is a step-by-step guide that takes you through a Continuous Integration Design and the means to achieve it using Jenkins, in collaboration with some other DevOps tools.

Chapter 5, *Continuous Integration Using Jenkins – Part II*, is a continuation of the previous chapter.

Chapter 6, *Continuous Delivery Using Jenkins*, is a step-by-step guide that takes you through a Continuous Delivery Design and the means to achieve it using Jenkins, in collaboration with some other DevOps tools.

Chapter 7, *Continuous Deployment Using Jenkins*, explains the difference between Continuous Delivery and Continuous Deployment. It is followed by a step-by-step guide that takes you through a Continuous Deployment Design and the means to achieve it using Jenkins, in collaboration with some other DevOps tools.

Chapter 8, *Jenkins Best Practices*, is a step-by-step guide to accomplishing distributed builds using the Jenkins master-slave architecture. It is followed by some practical examples that depict some of the Jenkins best practices.

What you need for this book

To set up the Jenkins server, you will need a machine with the following configurations.

Operating systems:

- Windows 7/8/9/10
- Ubuntu 14 and above

Software tools (minimum version):

- 7Zip 15.09 beta
- Apache JMeter 2.13
- Apache Tomcat server 8.0.26
- Artifactory 4.3.2 (maximum version for the build breaker plugin to work)
- Atlassian SourceTree 1.6.25
- Git 2.6.3
- Java JDK 1.8.0

- Java JRE 1.8.0
- Jenkins 1.635
- Maven 3.3.9
- Selenium for Eclipse 2.51
- SonarQube 5.1.2
- TestNG for Eclipse 6.8
- Eclipse Mars.1

Hardware requirements:

- A machine with a minimum 1 GB of memory and a multi-core processor

Who this book is for

This book is aimed at readers with little or no previous experience with agile or Continuous Integration. It serves as a great starting point for everyone who is new to the field of DevOps and would like to leverage the benefits of Continuous Integration and Continuous Delivery in order to increase productivity and reduce delivery time.

Build and release engineers, deployment engineers, DevOps engineers, SCM (Software Configuration Management) engineers, developers, testers, and project managers all can benefit from this book.

The readers who are already using Jenkins for Continuous Integration can learn to take their project to the next level, which is Continuous Delivery. This book discusses Continuous Integration, Continuous Delivery, and Continuous Deployment using a Java-based project. Nevertheless, the concepts are still applicable if you are using other technology setups, such as Ruby on Rails or .NET. In addition to that, the Jenkins concepts, installation, best practices, and administration, remain the same irrespective of the technology stack you use.

Conventions

In this book, you will find a number of text styles 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: "You have `make` and `omake`, and also `clearmake` if you are using IBM Rational ClearCase as the version control tool."

A block of code is set as follows:

```
# Print a message.
  print "Hello, World!\n";
  print "Good Morning!\n";
```

Any command-line input or output is written as follows:

```
cd /etc/sysconfig/
vi jenkins
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Click on the **Install as Windows Service** link."

[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**.

You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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/Learning-Continuous-Integration-with-Jenkins`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

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
Concepts of Continuous Integration

Software technology has evolved much like life on Earth. In the beginning, websites were static and programming languages were primitive, just like those simple multicellular organisms in the ancient oceans. In those times, software solutions were intended only for a few large organizations. Then, in the early 90s, the popularity of the Internet led to a rapid growth in various new programming languages and web technologies. And all of a sudden, there was a Cambrian-like explosion in the domain of information technology that brought up diversity in software technologies and tools. The growth of the Internet, powered by dynamic websites running on HTML languages, changed the way information was displayed and retrieved.

This continues to date. In recent years, there has been an immense demand for software solutions in many big and small organizations. Every business wants to venture its product online, either through websites or apps. This huge need for economical software solutions has led to the growth of various new software development methodologies that make software development and its distribution quick and easy. An example of this is the **extreme programming** (**XP**), which attempted to simplify many areas of software development.

Large software systems in the past relied heavily on documented methodologies, such as the **waterfall model**. Even today, many organizations across the world continue to do so. However, as software engineering continues to evolve, there is a shift in the way software solutions are being developed and the world is going agile.

Understanding the concepts of **Continuous Integration** is our prime focus in the current chapter. However, to understand Continuous Integration, it is first important to understand the prevailing software engineering practices that gave birth to it. Therefore, we will first have an overview of various software development processes, their concepts, and implications. To start with, we will first glance through the **agile software development process**. Under this topic, we will learn about the popular software development process, the waterfall model, and its advantages and disadvantages when compared to the agile model. Then, we will jump to the **Scrum framework** of software development. This will help us to answer how Continuous Integration came into existence and why it is needed. Next, we will move to the concepts and best practices of Continuous Integration and see how this helps projects to get agile. Lastly, we will talk about all the necessary methods that help us realize the concepts and best practices of Continuous Integration.

The agile software development process

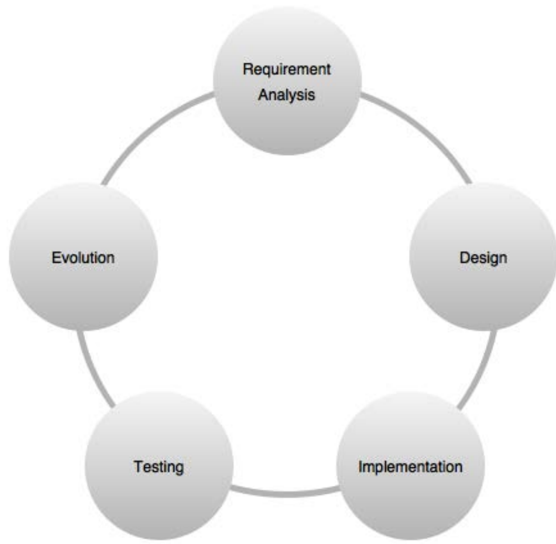
The name agile rightly suggests *quick and easy*. Agile is a collection of software development methodologies in which software is developed through collaboration among self-organized teams. Agile software development promotes adaptive planning. The principles behind agile are incremental, quick, and flexible software development.

For most of us who are not familiar with the software development process itself, let's first understand what the software development process or software development life cycle is.

> Software development process, software development methodology, and software development life cycle have the same meaning.

Software development life cycle

Software development life cycle, also sometimes referred to as **SDLC** in brief, is the process of planning, developing, testing, and deploying software. Teams follow a sequence of phases, and each phase uses the outcome of the previous phase, as shown in the following diagram:



Let's understand these phases in detail.

Requirement analysis

First, there is a requirement analysis phase: here, the business teams, mostly comprising business analysts, perform a requirement analysis of the business needs. The requirements can be internal to the organization or external from a customer. This analysis includes finding the nature and scope of the problem. With the gathered information, there is a proposal either to improve the system or to create a new one. The project cost is also decided and benefits are laid out. Then, the project goals are defined.

Design

The second phase is the design phase. Here, the system architects and the system designers formulate the desired features of the software solution and create a project plan. This may include process diagrams, overall interfaces, layout designs, and a huge set of documentation.

Implementation

The third phase is the implementation phase. Here, the project manager creates and assigns tasks to the developers. The developers develop the code depending on the tasks and goals defined in the design phase. This phase may last from a few months to a year, depending on the project.

Testing

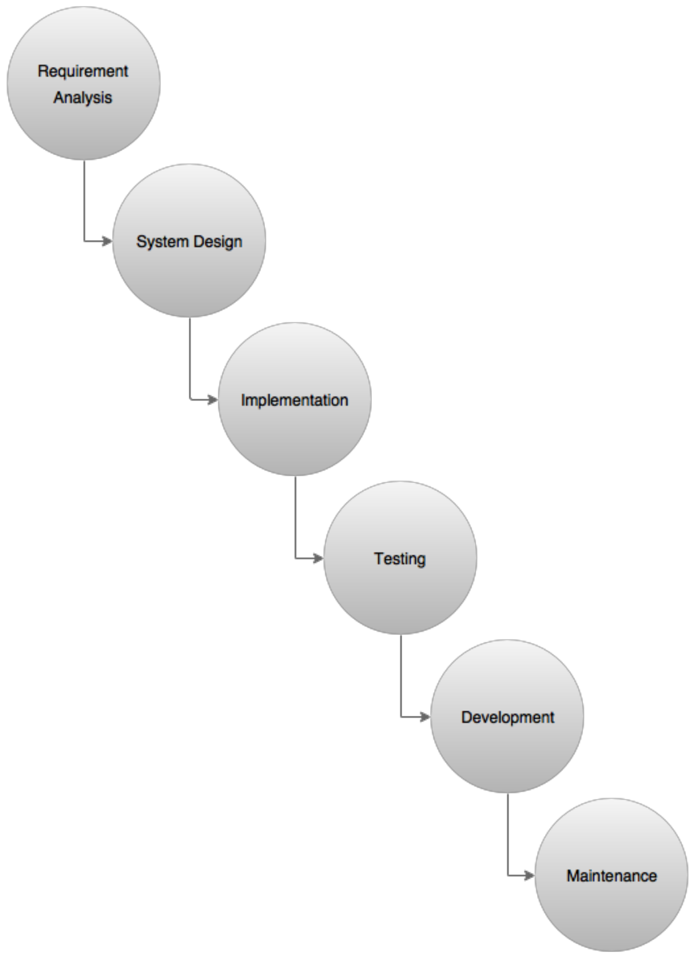
The fourth phase is the testing phase. Once all the decided features are developed, the testing team takes over. For the next few months, there is a thorough testing of all the features. Every module of the software is brought into one place and tested. Defects are raised if any bugs or errors erupt while testing. In the event of failures, the development team quickly actions on it. The thoroughly tested code is then deployed into the production environment.

Evolution

The last phase is the evolution phase or the maintenance phase. Feedback from the users/customers is analyzed, developed, tested, and published in the form of patches or upgrades.

The waterfall model of software development

One of the most famous and widely used software development processes is the waterfall model. The waterfall model is a sequential software development process. It was derived from the manufacturing industry. One can see a highly structured flow of processes that run in one direction. In those times, there were no software development methodologies, and the only thing the developers could have imagined was the production line process, which was simple to adapt for software development. The following diagram illustrates the sequence steps in the waterfall model:



The waterfall approach is simple to understand. The steps involved are similar to the ones discussed for the software development life cycle.

First, there is the requirement analysis phase followed by the designing phase. There is considerable time spent on the analysis and the designing part. And once it's over, there are no further additions or deletions. In short, once the development begins, there is no modification allowed in the design.

Then, comes the implementation phase where the actual development takes place. The development cycle can range from 3 months to 6 months. During this time, the testing team is usually free. Once the development cycle is completed, a whole week's time is planned for performing the integration and release of the source code in the testing environment. During this time, many integration issues pop up and are fixed at the earliest opportunity.

Once the testing starts, it goes on for another three months or more, depending on the software solution. After the testing completes successfully, the source code is deployed in the production environment. For this, again a day or two is planned to carry out the deployment. There is a possibility that some deployment issues may pop up.

After this, the software solution goes live. The teams get feedback and may also anticipate issues. The last phase is the maintenance phase. In this phase, the development team works on the development, testing, and release of software updates and patches, depending on the feedback and bugs raised by the customers.

There is no doubt that the waterfall model has worked remarkably well for decades. However, flaws did exist, but they were simply ignored for a long time because, way back then, software projects had an ample amount of time and resources to get the job done.

However, looking at the way software technologies have changed in the past few years we can easily say that this model won't suit the requirements of the current world.

Disadvantages of the waterfall model

The disadvantages of the waterfall model are:

- Working software is produced only at the end of the software development life cycle, which lasts for a year or so in most of the projects.
- There is a huge amount of uncertainty.
- This model is not suitable for projects based on object-oriented programming languages, such as Java or .NET.
- This model is not suitable for projects where changes in the requirements are frequent. For example, e-commerce websites.
- Integration is done after the complete development phase is over. As a result, teams come to know about the integration issues at a very later stage.
- There is no backward traceability.
- It's difficult to measure progress within stages.

Who needs the waterfall model?

By looking at the disadvantages of the waterfall model, we can say that it's mostly suitable for projects where:

- The requirements are well-documented and fixed.
- There is enough funding available to maintain a management team, testing team, development team, build and release team, deployment team, and so on.
- The technology is fixed and not dynamic.
- There are no ambiguous requirements. And most importantly, they don't pop up during any other phase apart from the requirement analysis phase.

Agile to the rescue

The agile software development process is an alternative to the traditional software development processes, as discussed earlier. The following are the 12 principles on which the agile model is based:

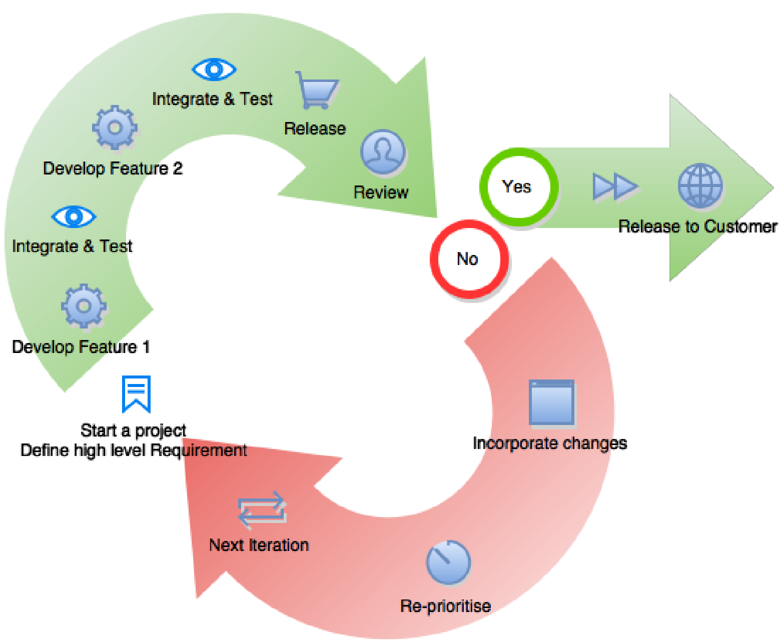
- Customer satisfaction by early and continuous delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (in weeks rather than months)
- Close daily cooperation between business people and developers
- Projects are built around motivated individuals, who should be trusted
- Face-to-face conversation is the best form of communication (co-location)
- Working software is the principal measure of progress
- Sustainable development that is able to maintain a constant pace
- Continuous attention to technical excellence and good design
- Simplicity—the art of maximizing the amount of work not done—is essential
- Self-organizing teams
- Regular adaptation to changing circumstances

> The 12 agile principles are taken from `http://www.agilemanifesto.org`.

The 12 principles of agile software development clearly indicate the expectation of the current software industry and its advantages over the waterfall model.

How does the agile software development process work?

In the agile software development process, the whole software is broken into many features or modules. These features or modules are delivered in iterations. Each iteration lasts for 3 weeks and involves cross-functional teams that work simultaneously in various areas, such as planning, requirements analysis, design, coding, unit testing, and acceptance testing. As a result, there is no single person sitting idle at any given point of time whereas in the waterfall model, while the development team is busy developing the software, the testing team, the production support team and everyone else is either idle or underutilized.



You can see, in the preceding diagram, that there is no time spent on the requirement analysis or design. Instead, a very high-level plan is prepared, just enough to outline the scope of the project.

The team then goes through a series of iterations. Iterations can be classified as time frames, each lasting for a month, or even a week in some mature projects. In this duration, a project team develops and tests features. The goal is to develop, test, and release a feature in a single iteration. At the end of the iteration, the feature goes for a demo. If the clients like it, then the feature goes live. If it gets rejected, the feature is taken as a backlog, reprioritized and again worked upon in the consecutive iteration.

There is also a possibility for parallel development and testing. In a single iteration, you can develop and test more than one feature in parallel.

Let's take a look at some of the advantages of the agile software development process:

- **Functionality can be developed and demonstrated rapidly**: In an agile process, the software project is divided on the basis of features and each feature can be called as a backlog. The idea is to develop a single or a set of features right from its conceptualization until its deployment, in a week or a month. This puts at least a feature or two on the customer's plate, which they can start using.
- **Resource requirement is less**: In agile, there is no separate development team and testing team. There is neither a build or release team or deployment team. In agile, a single project team contains around eight members, and each individual in the team is capable of doing everything. There is no distinction among the team members.
- **Promotes teamwork and cross training**: As mentioned earlier, since there is a small team of about eight members, the team members in turn switch their roles and learn about each other's experience.
- **Suitable for projects where requirements change frequently**: In the agile model of software development, the complete software is divided into features and each feature is developed and delivered in a short span of time. Hence, changing the feature, or even completely discarding it, doesn't affect the whole project.
- **Minimalistic documentation**: This methodology primarily focuses on delivering working software quickly rather than creating huge documents. Documentation exists, but it's limited to the overall functionality.
- **Little or no planning required**: Since features are developed one after the other in a short duration of time, hence, there is no need for extensive planning.
- **Parallel development**: An iteration consists of one or more features that develop in a sequence or even in parallel.

The Scrum framework

One of the widely-used agile software development methodologies is the Scrum framework. Scrum is a framework used to develop and sustain complex products that are based on the agile software development process. It is more than a process; it's a framework with certain roles, tasks, and teams. Scrum was written by Ken Schwaber and Jeff Sutherland; together they created the **Scrum guide**.

In a Scrum framework, the development team decides on how a feature needs to be developed. This is because the team knows best how to solve the problem they are presented with. I assume that most of the readers are happy after reading this line.

Scrum relies on a self-organizing and cross-functional team. The Scrum team is self-organizing; hence, there is no team leader who decides which person will do which task or how a problem will be solved. In Scrum, a team is cross-functional, which means everyone takes a feature from an idea to implementation.

Important terms used in the Scrum framework

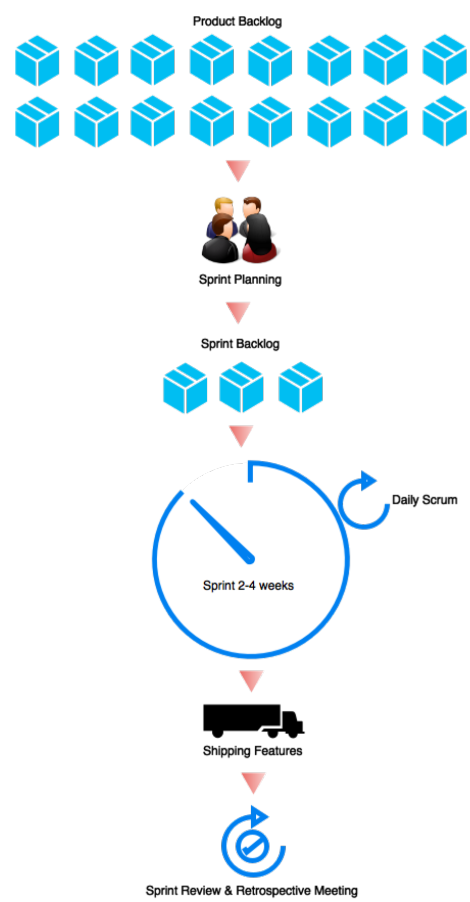
The following are the important terms used in the Scrum framework:

- **Sprint**: Sprint is a time box during which a usable and potentially releasable product increment is created. A new sprint starts immediately after the conclusion of the previous sprint. A sprint may last for 2 weeks to 1 month, depending on the projects' command over Scrum.
- **Product backlog**: The product backlog is a list of all the required features in a software solution. This list is dynamic, that is, every now and then the customers or team members add or delete items to the product backlog.
- **Sprint backlog**: The sprint backlog is the set of product backlog items selected for the sprint.
- **Increment**: The increment is the sum of all the product backlog items completed during a sprint and the value of the increments of all the previous sprints.
- **The development team**: The development team does the work of delivering a releasable set of features named increment at the end of each sprint. Only members of the development team create the increment. Development teams are empowered by the organization to organize and manage their own work. The resulting synergy optimizes the development team's overall efficiency and effectiveness.

- **The product owner**: The product owner is a mediator between the Scrum team and everyone else. He is the face of the Scrum team and interacts with customers, infrastructure teams, admin teams, and everyone involved in the Scrum, and so on.
- **The Scrum Master**: The Scrum Master is responsible for ensuring that Scrum is understood and enacted. Scrum Masters do this by ensuring that the Scrum team follows Scrum theory, practices, and rules.

How does Scrum work?

The product owner, the Scrum master, and the Scrum team together follow a set of stringent procedures to quickly deliver the software features. The following diagram explains the Scrum development process:



Let's take a look at some of the important aspects of the Scrum software development process, which the team goes through.

Sprint planning

Sprint planning is an opportunity for the Scrum team to plan the features in the current sprint cycle. The plan is mainly created by the developers. Once the plan is created, it is explained to the Scrum master and the product owner. The sprint planning is a time-boxed activity, and it is usually around 8 hours in total for a 1-month sprint cycle. It is the responsibility of the Scrum Master to ensure that everyone participates in the sprint planning activity, and he is also the one to keep it within the time box.

In the meeting, the development team takes into consideration the following items:

- Number of product backlogs to be worked on (both new and the old ones coming from the last sprint)
- The teams' performance in the last sprint
- Projected capacity of the development team

Sprint cycle

During the sprint cycle, the developers simply work on completing the backlogs decided in the sprint planning. The duration of a sprint may last from two weeks to one month, depending on the number of backlogs.

Daily scrum meeting

This activity happens on a daily basis. During the scrum meeting, the development team discusses what was accomplished yesterday and what will be accomplished today. They also discuss the things that are stopping them from achieving their goal. The development team does not attend any other meetings or discussions apart from the Scrum meeting.

Monitoring sprint progress

The daily scrum is a good opportunity for a team to measure the progress of the project. The team can track the total work that is remaining, and using it, they can estimate the likelihood of achieving the sprint goal.

The sprint review

The sprint review is like a demo to the customers regarding what has been accomplished and what they were unable to accomplish. The development team demonstrates the features that have been accomplished and answers the questions based on the increment. The product owner updates the product backlog list status till date. The product backlog list may be updated, depending on the product performance or usage in the market. The sprint review is a four-hour activity in total for a one month sprint.

Sprint retrospective

In this meeting, the team discusses the things that went well and the things that need improvement. The team then decides the points on which it has to improve to perform better in the upcoming sprint. This meeting usually occurs after the sprint review and before the sprint planning.

Continuous Integration

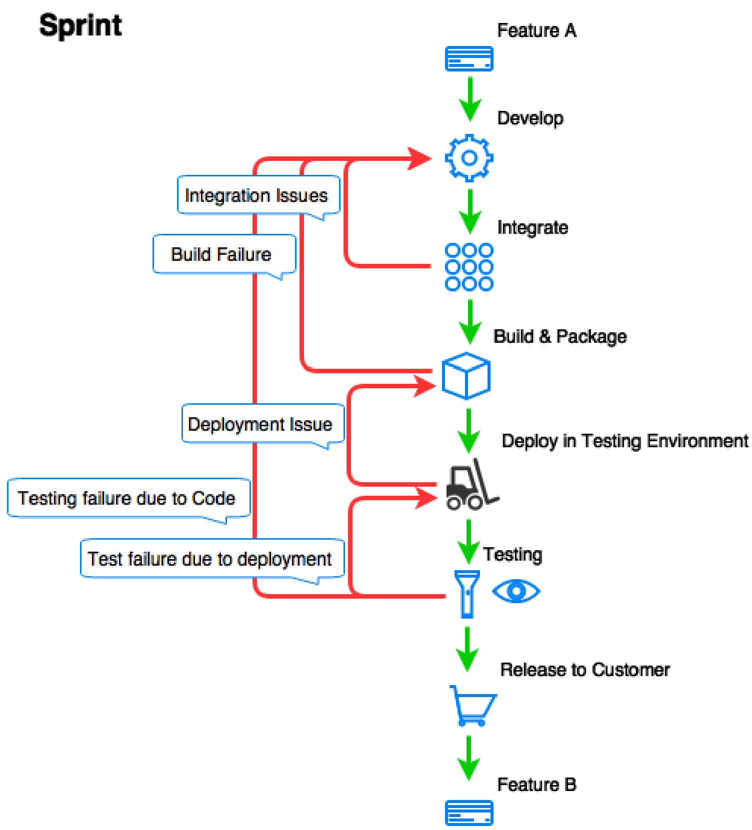
Continuous Integration is a software development practice where developers frequently integrate their work with the project's **integration branch** and create a build.

Integration is the act of submitting your personal work (modified code) to the common work area (the potential software solution). This is technically done by merging your personal work (personal branch) with the common work area (Integration branch). Continuous Integration is necessary to bring out issues that are encountered during the integration as early as possible.

This can be understood from the following diagram, which depicts various issues encountered during a software development lifecycle. I have considered a practical scenario wherein I have chosen the Scrum development model, and for the sake of simplicity, all the meeting phases are excluded. Out of all the issues depicted in the following diagram, the following ones are detected early when Continuous Integration is in place:

- Build failure (the one before integration)
- Integration issues
- Build failure (the one after integration)

In the event of the preceding issues, the developer has to modify the code in order to fix it. A build failure can occur either due to an improper code or due to a human error while doing a build (assuming that the tasks are done manually). An integration issue can occur if the developers do not rebase their local copy of code frequently with the code on the Integration branch.



In the preceding diagram, I have considered only a single testing environment for simplicity. However, in reality, there can be as many as three to four testing environments.

An example to understand Continuous Integration

To understand Continuous Integration better, let's take the previous example a bit forward, this time at a more granular level.

In any development team, there are a number of developers working on a set of files at any given point of time. Imagine that the software code is placed at a centralized location using a version control system. And developer "A" creates a branch for himself to work on a code file that prints some lines. Let's say the code when compiled and executed, prints "Hello, World".

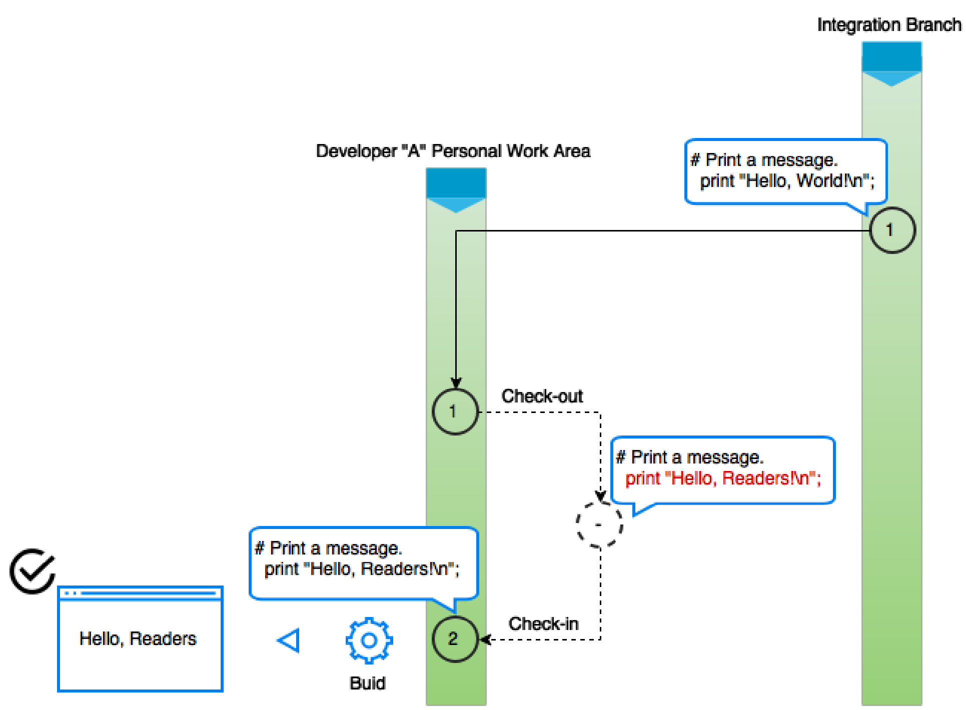
```
# Print a message.
  Print "Hello, World\n";
```

After creating a branch, developer "A" checks out the file and modifies the following code:

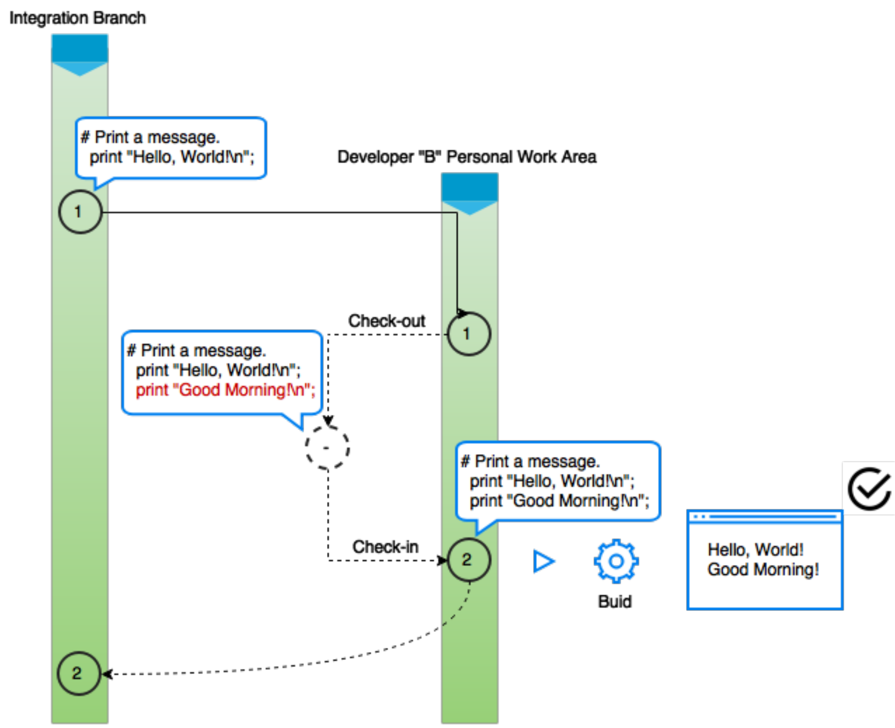
```
# Print a message.
  Print "Hello, Readers\n";
```

He then checks in the file, and after check-in, he performs a build. The code is compiled, and the unit testing results show positive.

Nevertheless, if the unit tests were to fail, the developer would have returned to the code, checked for errors, modified the code, built it again and again until the compilation and unit test show positive. This following diagram depicts the scenario that we discussed so far.



Assume that our developer "A" gets busy with some other task and simply forgets to deliver his code to the Integration branch or he plans to do it later. While the developer is busy working in isolation, he is completely unaware of the various changes happening to the same code file on the Integration branch. There is a possibility that some other developer, say developer "B," has also created a private branch for himself and is working on the same file.

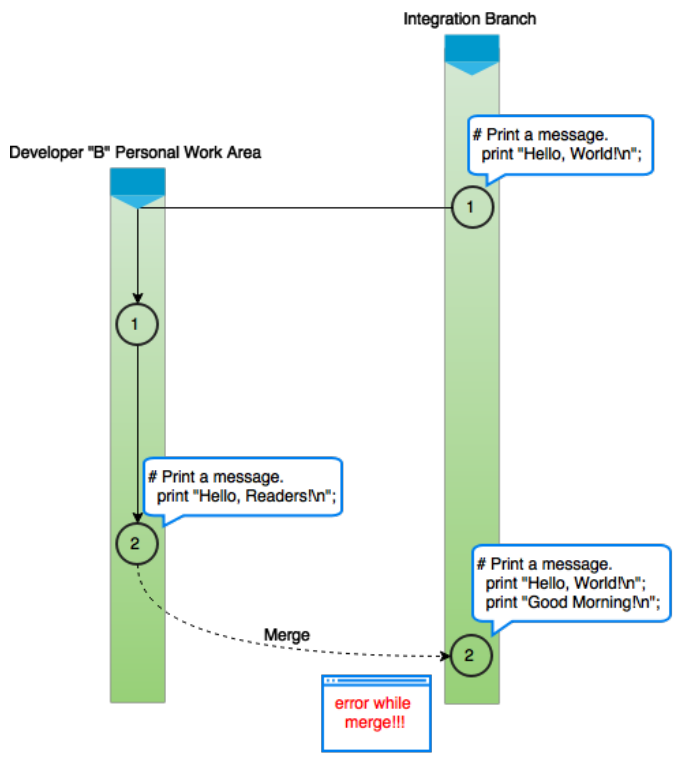


In the preceding diagram, we can see how developer "B" has changed the same file by adding the following line of code:

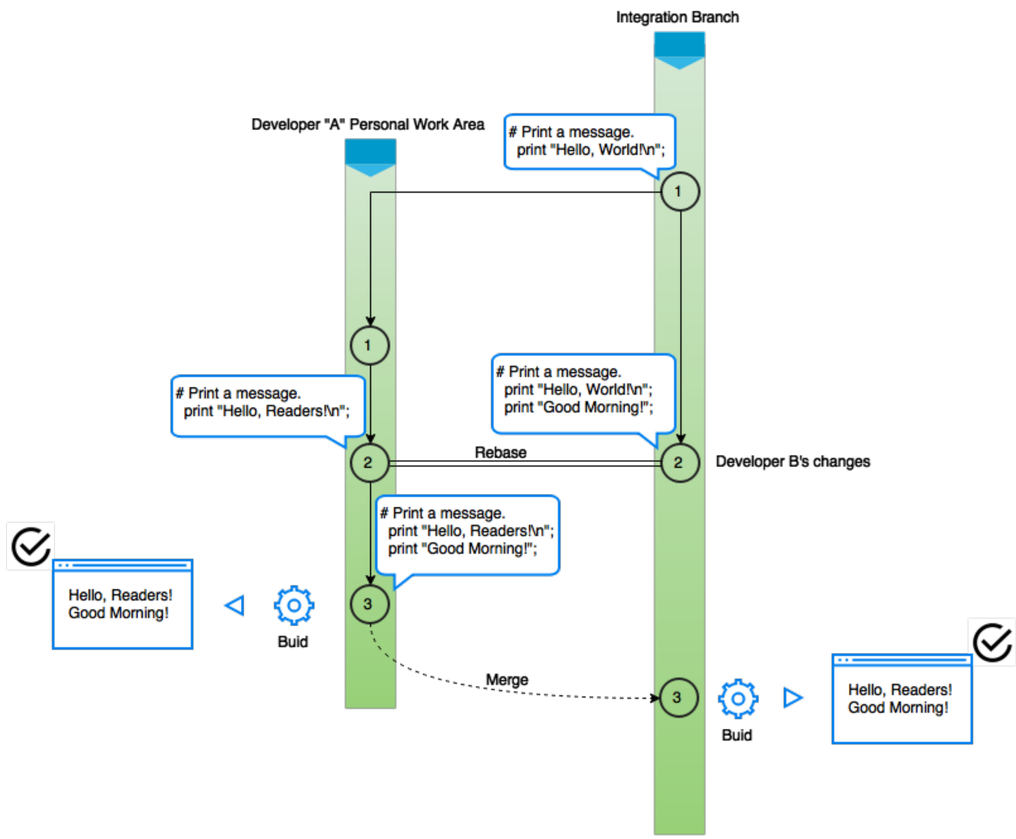
```
# Print a message.
  print "Hello, World!\n";
  print "Good Morning!\n";
```

After the modification, developer "B" compiles and unit tests the code, and then, he integrates the code on the Integration branch, thus creating a new version "2".

Now after a week of time, at the end of the sprint, the developer "A" realizes that he has not integrated his code into the Integration branch. He quickly makes an attempt to, but to his surprise, he finds merge issues (in most cases, the merge is successful, but the code on the Integration branch fails to compile due to an integration issue).



To resolve this, he does a rebase with the Integration branch (he updates his private work area with that of the **Integration Branch**) and again tries to merge, as shown in the following diagram:



What do we make out of this? If developer "A" had immediately rebased and integrated his changes with the changes on the Integration branch (Continuous Integration), then he would have known about the merge issues far in advance and not at the end of the sprint. Therefore, developers should integrate their code frequently with the code on the Integration branch.

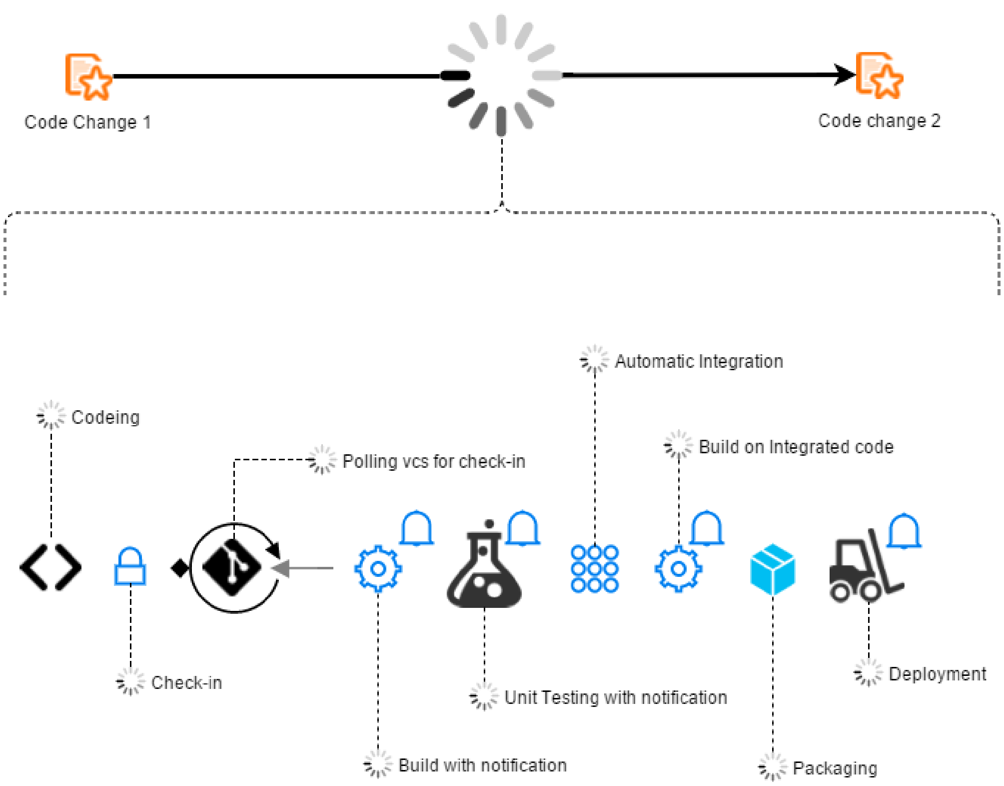
Since you're integrating frequently, there is significantly less back-tracking to discover where things went wrong.

If you don't follow a continuous approach, you'll have longer periods between integrations. This makes it exponentially more difficult to find and fix problems. Such integration problems can easily knock a project off schedule or can even cause it to fail altogether.

Agile runs on Continuous Integration

The agile software development process mainly focuses on faster delivery, and Continuous Integration helps it in achieving that speed. Yet, how does Continuous Integration do it? Let's understand this using a simple case.

Developing a feature may involve a lot of code changes, and between every code change, there can be a number of tasks, such as checking in the code, polling the version control system for changes, building the code, unit testing, integration, building on integrated code, packaging, and deployment. In a Continuous Integration environment, all these steps are made fast and error-free using automation. Adding notifications to it makes things even faster. The sooner the team members are aware of a build, integration, or deployment failure, the quicker they can act upon it. The following diagram clearly depicts all the steps involved in code changes:



In this way, the team quickly moves from feature to feature. We can safely conclude that the "agility" of an agile software development is made possible through Continuous Integration.

Types of project that benefit from Continuous Integration

The amount of code written for the embedded systems present inside a car is more than that present inside a fighter jet. In today's world, embedded software is inside every product, modern or traditional. Cars, TVs, refrigerators, wrist watches, and bikes all have little or more software dependent features. Consumer products are becoming smarter day by day. Nowadays, we can see a product being marketed more using its smart and intelligent features than its hardware capability. For example, an air conditioner is marketed by its wireless control features, TVs are being marketed by their smart features, such as embedded web browsers, and so on.

The need to market new products has increased the complexity of products. This increase in software complexity has brought agile software development and Continuous Integration methodologies into the limelight. Though, there were times when agile software development was used by a team of not more than 30-40 people, working on a simple project. Almost all types of projects benefit from Continuous Integration. Mostly the web-based projects, for example, e-commerce websites and mobile phone apps.

Continuous Integration, automation, and agile are mostly thought to be used in projects that are based on Java, .NET, and Ruby on Rails. The only place where you will see it's not used are the legacy systems. However, even they are going agile. Projects based on SAS, Mainframe, and Perl are all now using Continuous Integration in some ways.

The best practices of Continuous Integration

Simply having a Continuous Integration tool doesn't mean Continuous Integration is achieved. A considerable amount of time needs to be spent in the configuration of configuring Integration tool.

A tool such as Jenkins works in collaboration with many other tools to achieve Continuous Integration. Let's take a look at some of the best practices of Continuous Integration.

Developers should work in their private workspace

In a Continuous Integration world, working in a private work area is always advisable. The reason is simple: isolation. One can do anything on their private branch or to simply say with their private copy of the code. Once branched, the private copy remains isolated from the changes happening on the mainline branch. And in this way, developers get the freedom to experiment with their code and try new stuff.

If the code on developer A's branch fails due to some reason, it will never affect the code present on the branches belonging to the other developers. Working in a private workspace either through branching or through cloning repos is a great way to organize your code.

For example, let's assume that a bug fix requires changes to be made to the `A.java`, `B.java`, and `C.java` files. So, a developer takes the latest version of the files and starts working on them. The files after modification are let's say version 56 of the `A.java` file, version 20 of the `B.java` file, and version 98 of the `C.java` file. The developer then creates a package out of those latest files and performs a build and then performs a test. The build and testing run successfully and everything is good.

Now consider a situation where after several months, another bug requires the same changes. The developer will usually search for the respective files with particular versions that contain the code fix. However, these files with the respective versions might have been lost in the huge oceans of versions by now.

Instead, it would have been better if the file changes were brought to a separate branch long back (with the branch name reflecting the defect number). In this way, it would have been easy to reproduce the fix using the defect number to track the code containing the required fix.

Rebase frequently from the mainline

We all know about the time dilation phenomena (relativity). It is explained with a beautiful example called the **twin paradox,** which is easy to understand but hard to digest. I have modified the example a little bit to suit our current topic. The example goes like this; imagine three developers: developers A, B, and C. Each developer is sent into space in his own spacecraft that travels at the speed of light. All are given atomic clocks that show exactly the same time. Developer B is supposed to travel to planet Mars to sync the date and time on a computer, which is on Mars. Developer C is supposed to travel to Pluto for a server installation and to sync the clock with that of Earth.

Developer A has to stay on Earth to monitor the communication between the server that is present on Earth with the servers on Mars and Pluto. So, all start at morning 6 AM one fine day.

After a while, developers B and C finish their jobs and return to Earth. On meeting each other, to their surprise, they find their clocks measuring a different time (of course, they find each other aged differently). They all are totally confused as to how this happened. Then, developer A confirms that all the three servers that are on Earth, Mars, and Pluto, respectively are not in sync.

Then, developer A recalls that while all the three atomic clocks were in sync back then on Earth, they forgot to consider the time dilation factor. If they would have included it keeping in mind the speed and distance of travel, the out-of-sync issue could have been avoided.

This is the same situation with developers who clone the Integration branch and work on their private branch, each one indulging in their own assignment and at their own speed. At the time of merging, each one will definitely find their code different from the others and the Integration branch, and will end up with something called as **Merge Hell**. The question is how do we fix it? The answer is **frequent rebase**.

In the previous example (developers with the task of syncing clocks on computers located across the solar system), the cause of the issue was to neglect the "time dilation factor". In the latter example (developers working on their individual branch), the cause of the issue was neglecting the frequent rebase. **Rebase** is nothing but updating your private branch with the latest version on the Integration branch.

While working on a private repository or a private branch surely has its advantages; it also has the potential to cause lots of merge issues. In a software development project containing 10 to 20 developers, each developer working by creating a private clone of the main repository completely changes the way the main repository looked over time.

In an environment where code is frequently merged and frequently rebased, such situations are rare. This is the advantage of using continuous integration. We integrate continuously and frequently.

The other situations where rebasing frequently helps are:

- You branched out from a wrong version of the integration branch, and now you have realized that it should have been version 55 and not 66.
- You might want to know the merge issues that occur when including code developed on some other branch belonging to a different developer.

- Also, too much merging messes up the history. So rather than frequently merging, it's better to rebase frequently and merge less. This trick also works in avoiding merge issues.
- While frequent rebase means less frequent merges on the Integration branch, which, in turn, means less number of versions on the Integration branch and more on the private, there is an advantage. This makes the integration clear and easy to follow.

Check-in frequently

While rebase should be frequent, so should check-in, at least once a day on his/her working branch. Checking in once a week or more is dangerous. The one whole week of code that is not checked-in runs the risk of merge issues. And these can be tedious to resolve. By committing or merging once a day, conflicts are quickly discovered and can be resolved instantly.

Frequent build

Continuous Integration tools need to make sure that every commit or merge is built to see the impact of the change on the system. This can be achieved by constantly polling the Integration branch for changes. And if changes are found, build and test them. Afterwards quickly share the results with the team. Also, builds can run nightly. The idea is to get instant feedback on the changes they have made.

Automate the testing as much as possible

While a continuous build can give instant feedback on build failures, continuous testing, on the other hand, can help in quickly deciding whether the build is ready to go to the production. We should try to include as many test cases as we can, but this again increases the complexity of the Continuous Integration system. Tests that are difficult to automate are the ones that reflect the real-world scenarios closely. There is a huge amount of scripting involved and so the cost of maintaining it rises. However, the more automated testing we have, the better and sooner we get to know the results.

Don't check-in when the build is broken

How can we do that? The answer is simple; before checking in your code, perform a build on your local machine, and if the build breaks, do not proceed with the check-in operation. There is another way of doing it. The version control system can be programmed to immediately trigger a build using the Continuous Integration tool, and if the tool returns positive results, only then the code is checked-in. Version control tools, such as **TFS** have a built in feature called a **gated check-in mechanism** that does the same.

There are other things that can be added to the gated check-in mechanism. For example, you can add a step to perform a static code analysis on the code. This again can be achieved by integrating the version control system with the Continuous Integration tool, which again is integrated with the tool that performs a static code analysis. In the upcoming chapters, we will see how this can be achieved using Jenkins in collaboration with SonarQube.

Automate the deployment

In many organizations, there is a separate team to perform deployments. The process is as follows. Once the developer has successfully created a build, he raises a ticket or composes a mail asking for a deployment in the respective testing environment. The deployment team then checks with the testing team if the environment is free. In other words, can the testing work be halted for a few hours to accommodate a deployment? After a brief discussion, a certain time slot is decided and the package is deployed.

The deployment is mostly manual and there are many manual checks that take the time. Therefore, for a small piece of code to go to the testing environment, the developer has to wait a whole day. And if for some reasons, the manual deployment fails due to a human error or due to some technical issues, it takes a whole day in some cases for the code to get into the testing area.

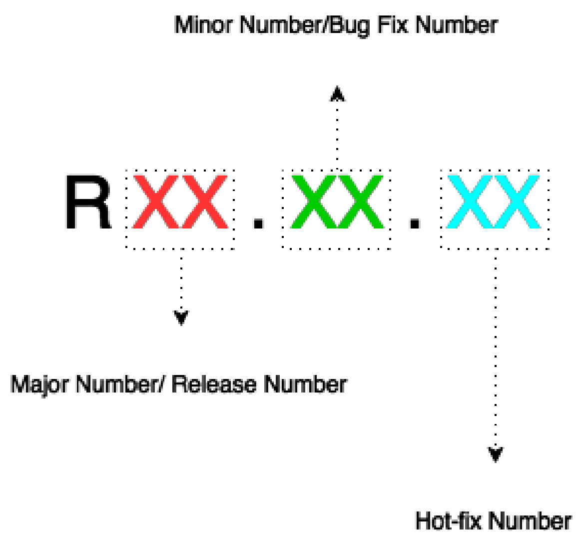
This is a painful thing for a developer. Nevertheless, this can be avoided by carefully automating the deployment process. The moment a developer tries to check-in the code, it goes through an automated compilation check, then it goes through an automated code analysis, and then it's checked-in to the Integration branch. Here the code is again picked along with the latest code on the Integration branch and then built. After a successful build, the code is automatically packaged and deployed in the testing environment.

Have a labeling strategy for releases

In my experience, some of the best practices of Continuous Integration are the same as those of software configuration management. For example, labels and baselines. While both are similar technically, they are not the same from the usage perspective. Labeling is the task of applying a tag to a particular version of a file or a set of files. We take the same concept a little bit further. For example, what if I apply a label to particular versions of all the files? Then, it would simply describe a state of the whole system. A version of the whole collective system. This is called a baseline. And why it is important?

Labels or baselines have many advantages. Imagine that a particular version of your private code fixed a production issue, say "defect number 1234". You can label that version on your private code as the defect number for later use. Labels can also be used to mark sprints, releases, and hotfixes.

The one that is used widely is shown in the following image:



Here, the first two digits are the release numbers. For example, 00 can be beta, 01 can be alpha, and 02 can represent the commercial release. The next two digits are the bug fix numbers. Let's say release 02.00.00 is in production and few bugs or improvements arise, then the developer who is working on fixing those issues can name his branch or label his code as 02.01.00.

Similarly, consider another scenario, where the release version in production is 03.02.00, and all of a sudden something fails and the issue needs to be fixed immediately. Then, the release containing the fix can be labeled as 03.02.01, which says that this was a hotfix on 03.02.00.

Instant notifications

They say communication is incomplete without feedback. Imagine a Continuous Integration system that has an automated build and deployment solution, a state-of-the-art automated testing platform, a good branching strategy, and everything else. However, it does not have a notification system that automatically emails or messages the status of a build. What if a nightly build fails and the developers are unaware of it?

What if you check-in code and leave early, without waiting for the automated build and deployment to complete? And the next day you find that the build failed due to a simple issue, which occurred just 10 minutes after you departed from the office.

If by some chance, you would have been informed through an SMS popping upon your mobile phone, then you could have fixed the issue.

Therefore, instant notifications are important. All the Continuous Integration tools have it, including Jenkins. It is good to have notifications of build failures, deployment failures, and testing results. We will see in the upcoming chapters how this can be achieved using Jenkins and the various options Jenkins provides to make life easy.

How to achieve Continuous Integration

Implementing Continuous Integration involves using various DevOps tools. Ideally, a DevOps engineer is responsible for implementing Continuous Integration. But, who is a DevOps engineer? And what is DevOps?

Development operations

DevOps stands for development operations, and the people who manage these operations are called DevOps engineers. All the following mentioned tasks fall under development operations:

- Build and release management
- Deployment management
- Version control system administration

- Software configuration management
- All sorts of automation
- Implementing continuous integration
- Implementing continuous testing
- Implementing continuous delivery
- Implementing continuous deployment
- Cloud management and virtualization

I assume that the preceding tasks need no explanation. A DevOps engineer accomplishes the previously mentioned tasks using a set of tools; these tools are loosely called DevOps tools (Continuous Integration tools, agile tools, team collaboration tools, defect tracking tools, continuous delivery tools, cloud management tools, and so on).

A DevOps engineer has the capability to install and configure the DevOps tools to facilitate development operations. Hence, the name DevOps. Let's see some of the important DevOps activities pertaining to Continuous Integration.

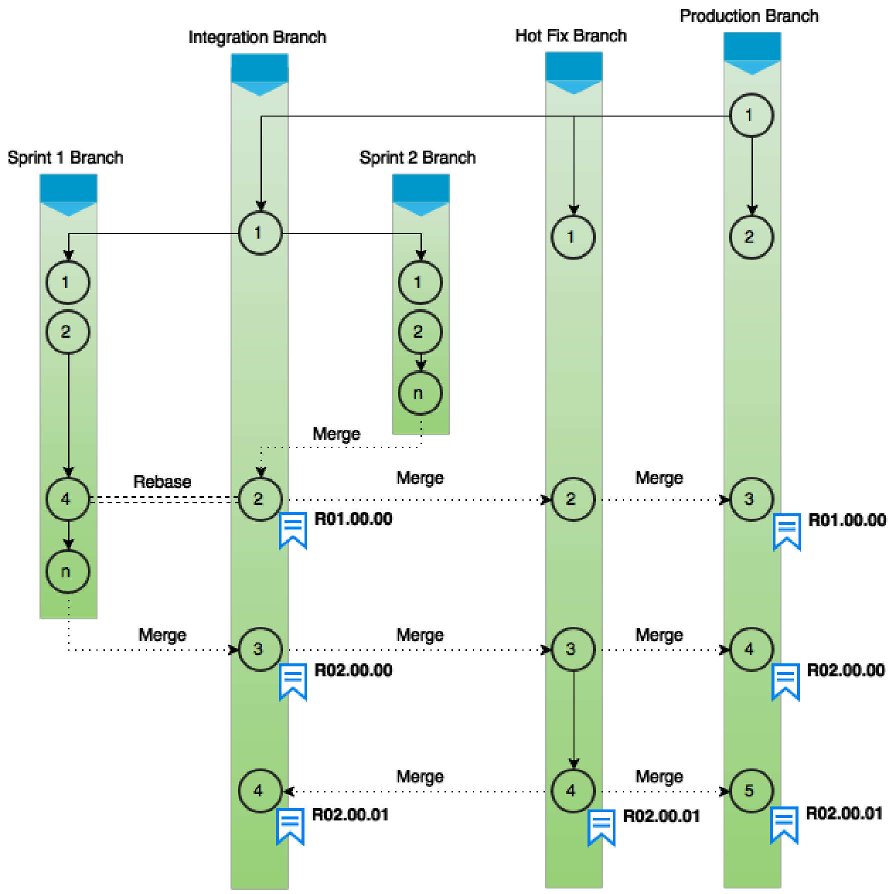
Use a version control system

This is the most basic and the most important requirement to implement Continuous Integration. A version control system, or sometimes it's also called a **revision control system**, is a tool used to manage your code history. It can be centralized or distributed. Two of the famously centralized version control systems are SVN and IBM Rational ClearCase. In the distributed segment, we have tools such as Git. Ideally, everything that is required to build software must be version controlled. A version control tool offers many features, such as labeling, branching, and so on.

When using a version control system, keep the branching to the minimum. Few companies have only one main branch and all the development activities happening on that. Nevertheless, most companies follow some branching strategies. This is because there is always a possibility that part of a team may work on a release and others may work on another release. At other times, there is a need to support older release versions. Such scenarios always lead companies to use multiple branches.

For example, imagine a project that has an Integration branch, a release branch, a hotfix branch, and a production branch. The development team will work on the release branch. They check-out and check-in code on the release branch. There can be more than one release branch where development is running in parallel. Let's say these are sprint 1 and sprint 2.

Once sprint 2 is near completion (assuming that all the local builds on the sprint 2 branch were successful), it is merged to the Integration branch. Automated builds run when there is something checked-in on the Integration branch, and the code is then packaged and deployed in the testing environments. If the testing passes with flying colors and the business is ready to move the release to production, then automated systems take the code and merge it with the production branch.



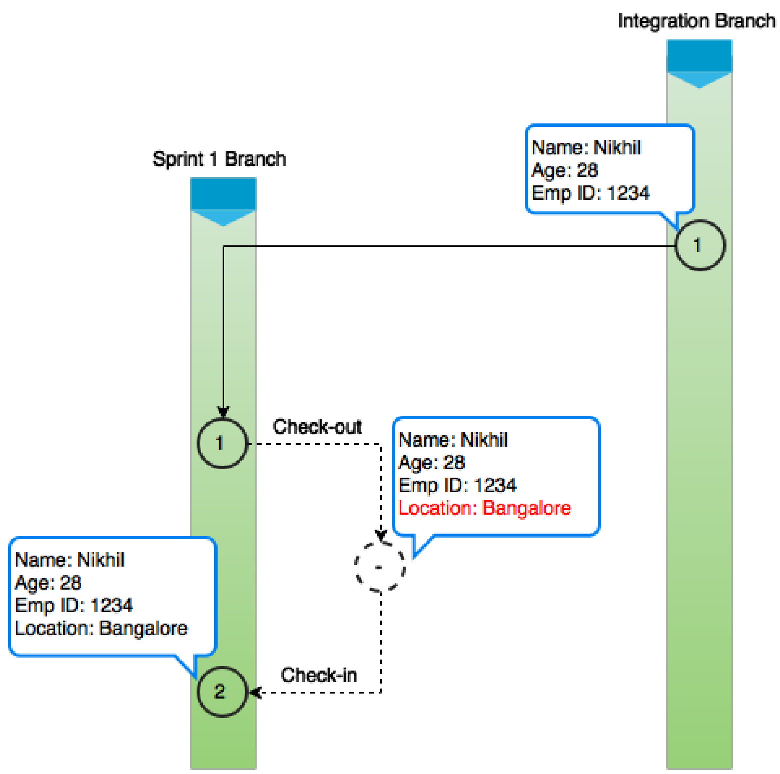
Typical branching strategies

From here, the code is then deployed in production. The reason for maintaining a separate branch for production comes from the desire to maintain a neat code with less number of versions. The production branch is always in sync with the hotfix branch. Any instant fix required on the production code is developed on the hotfix branch. The hotfix changes are then merged to the production as well as the Integration branch. The moment sprint 1 is ready, it is first rebased with the Integration branch and then merged into it. And it follows the same steps thereafter.

An example to understand VCS

Let's say I add a file named `Profile.txt` to the version control with some initial details, such as the name, age, and employee ID.

To modify the file, I have to check out the file. This is more like reserving the file for edit. Why reserve? In a development environment, a single file may be used by many developers. Hence, in order to facilitate an organized use, we have the option to reserve a file using the check-out operation. Let's assume that I do a check-out on the file and do some modifications by adding another line.



After the modification, I perform a check-in operation. The new version contains the newly added line. Similarly, every time you or someone else modifies a file, a new version gets created.

Types of version control system

We have already seen that a version control system is a tool used to record changes made to a file or set of files over time. The advantage is that you can recall specific versions of your file or a set of files. Almost every type of file can be version controlled. It's always good to use a **Version Control System** (**VCS**) and almost everyone uses it nowadays. You can revert an entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover.

Looking back at the history of version control tools, we can observe that they can be divided into three categories:

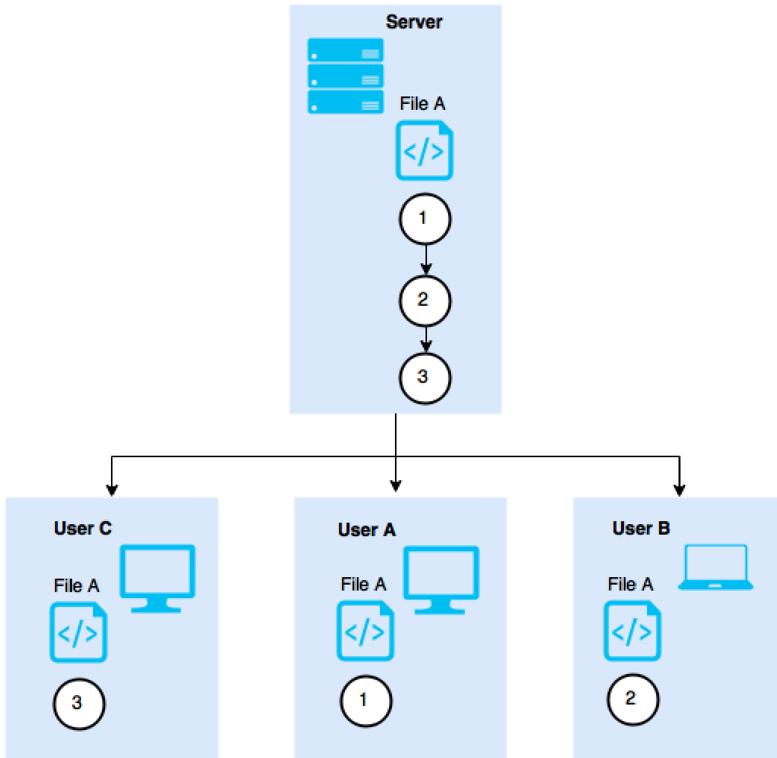
- Local version control systems
- Centralized version control systems
- Distributed version control systems

Centralized version control systems

Initially, when VCS came into existence some 40 years ago, they were mostly personal, like the one that comes with Microsoft Office Word, wherein you can version control a file you are working on. The reason was that in those times software development activity was minuscule in magnitude and was mostly done by individuals. But, with the arrival of large software development teams working in collaboration, the need for a centralized VCS was sensed. Hence, came VCS tools, such as Clear Case and Perforce. Some of the advantages of a centralized VCS are as follows:

- All the code resides on a centralized server. Hence, it's easy to administrate and provides a greater degree of control.
- These new VCS also bring with them some new features, such as labeling, branching, and baselining to name a few, which help people collaborate better.
- In a centralized VCS, the developers should always be connected to the network. As a result, the VCS at any given point of time always represents the updated code.

The following diagram illustrates a centralized VCS:

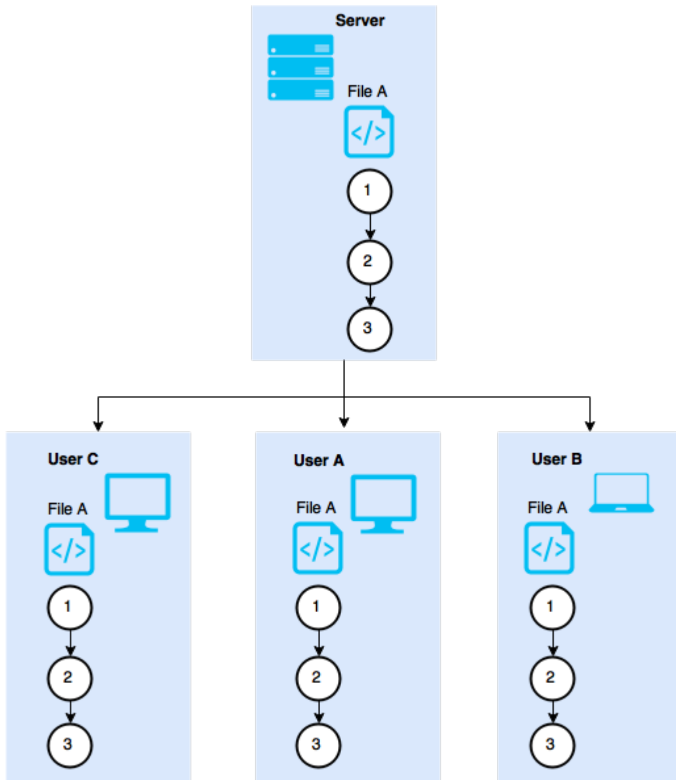


A centralized version control system

Distributed version control systems

Another type of VCS is the distributed VCS. Here, there is a central repository containing all the software solution code. Instead of creating a branch, the developers completely clone the central repository on their local machine and then create a branch out of the local clone repository. Once they are done with their work, the developer first merges their branch with the Integration branch, and then syncs the local clone repository with the central repository.

You can argue that this is a combination of a local VCS plus a central VCS. An example of a distributed VCS is Git.



A distributed version control system

Use repository tools

As part of the software development life cycle, the source code is continuously built into binary artifacts using Continuous Integration. Therefore, there should be a place to store these built packages for later use. The answer is to use a repository tool. But, what is a repository tool?

A repository tool is a version control system for binary files. Do not confuse this with the version control system discussed in the previous sections. The former is responsible for versioning the source code and the lateral for binary files, such as `.rar`, `.war`, `.exe`, `.msi`, and so on.

As soon as a build is created and passes all the checks, it should be uploaded to the repository tool. From there, the developers and testers can manually pick them, deploy them, and test them, or if the automated deployment is in place, then the build is automatically deployed in the respective test environment. So, what's the advantage of using a build repository?

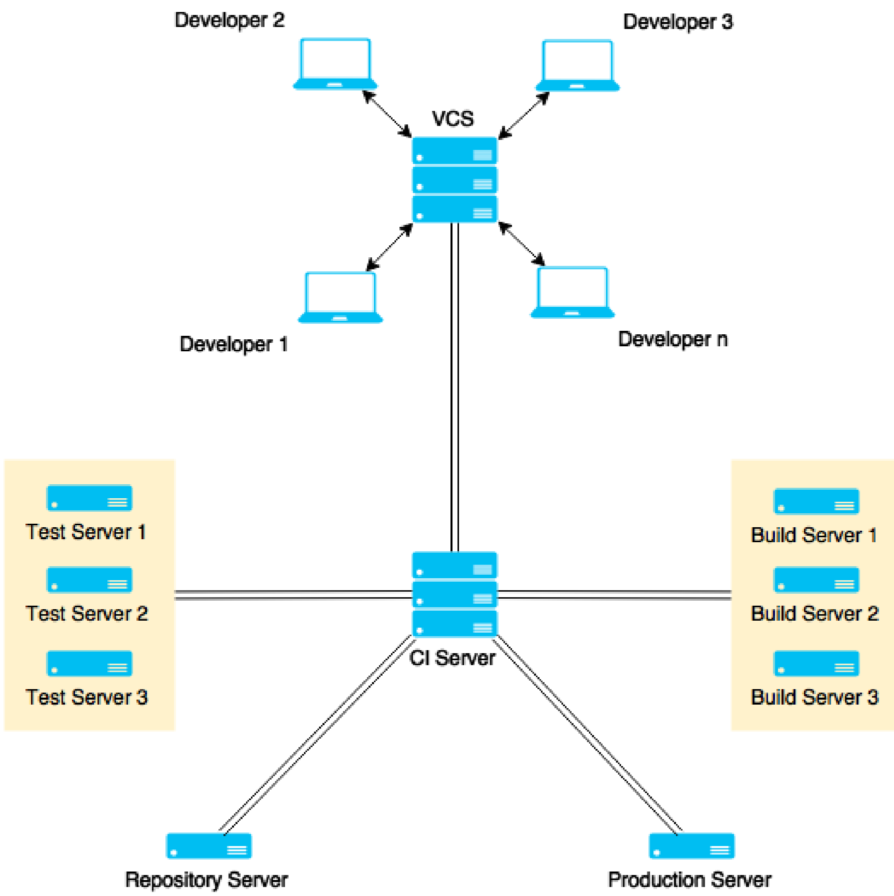
A repository tool does the following:

- Every time a build gets generated, it is stored in a repository tool. There are many advantages of storing the build artifacts. One of the most important advantages is that the build artifacts are located in a centralized location from where they can be accessed when needed.
- It can store third-party binary plugins, modules that are required by the build tools. Hence, the build tool need not download the plugins every time a build runs. The repository tool is connected to the online source and keeps updating the plugin repository.
- It records what, when, and who created a build package.
- It creates a staging area to manage releases better. This also helps in speeding up the Continuous Integration process.
- In a Continuous Integration environment, each build generates a package and the frequency at which the build and packaging happen is high. As a result, there is a huge pile of packages. Using a repository tool makes it possible to store all the packages in one place. In this way, developers get the liberty to choose what to promote and what not to promote in higher environments.

Use a Continuous Integration tool

What is a Continuous Integration tool? It is nothing more than an orchestrator. A continuous integration tool is at the center of the Continuous Integration system and is connected to the version control system tool, build tool, repository tool, testing and production environments, quality analysis tool, test automation tool, and so on. All it does is an orchestration of all these tools, as shown in the next image.

There are many Continuous Integration tools: Jenkins, Build Forge, Bamboo, and Team city to name a few.



Basically, Continuous Integration tools consist of various pipelines. Each pipeline has its own purpose. There are pipelines used to take care of Continuous Integration. Some take care of testing, some take care of deployments, and so on. Technically, a pipeline is a flow of jobs. Each job is a set of tasks that run sequentially. Scripting is an integral part of a Continuous Integration tool that performs various kinds of tasks. The tasks may be as simple as copying a folder/file from one location to another, or it can be a complex Perl script used to monitor a machine for file modification.

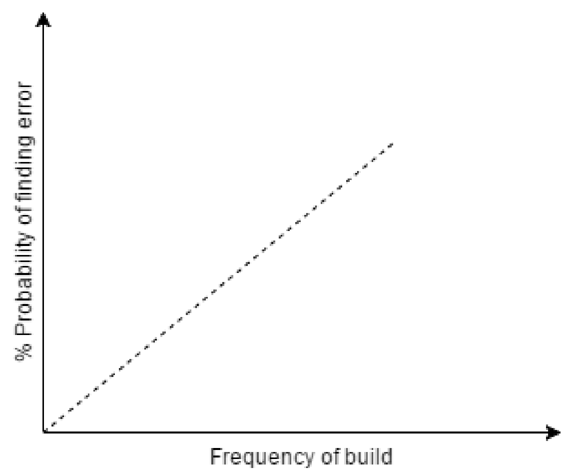
Creating a self-triggered build

The next important thing is the self-triggered automated build. Build automation is simply a series of automated steps that compile the code and generate executables. The build automation can take help of build tools, such as Ant and Maven. Self-triggered automated builds are the most important parts of a Continuous Integration system. There are two main factors that call for an automated build mechanism:

- Speed
- Catching integration or code issues as early as possible

There are projects where 100 to 200 builds happen per day. In such cases, speed is an important factor. If the builds are automated, then it can save a lot of time. Things become even more interesting if the triggering of the build is made self-driven without any manual intervention. An auto-triggered build on very code change further saves time.

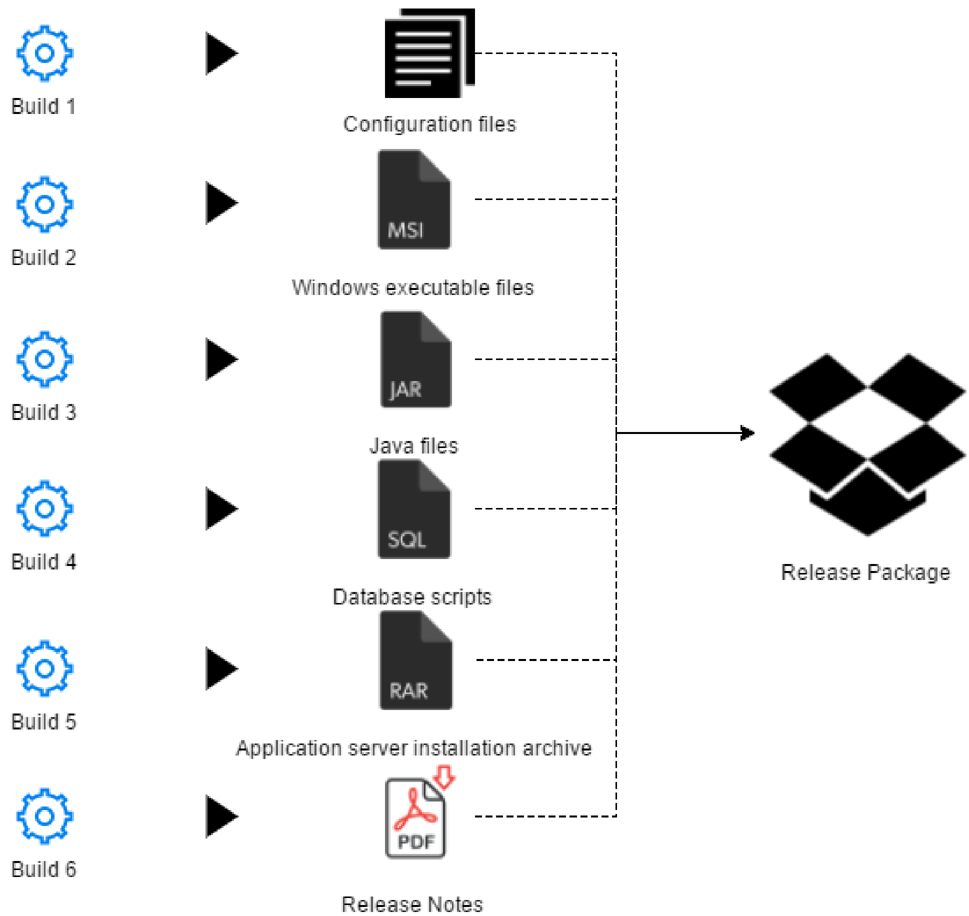
When builds are frequent and fast, the probability of finding errors (a build error, compilation error, and integration error) is also greater and faster.



Automate the packaging

There is a possibility that a build may have many components. Let's take, for example, a build that has a `.rar` file as an output. Along with this, it has some Unix configuration files, release notes, some executables, and also some database changes. All these different components need to be together. The task of creating a single archive or a single media out of many components is called packaging.

This again can be automated using the Continuous Integration tools and can save a lot of time.



Using build tools

IT projects can be on various platforms, such as Java, .NET, Ruby on Rails, C, and C++ to name a few. Also, in a few places, you may see a collection of technologies. No matter what, every programming language, excluding the scripting languages, has compilers that compile the code. Ant and Maven are the most common build tools used for projects based on Java. For the .NET lovers, there is MSBuild and TFS build. Coming to the Unix and Linux world, you have `make` and `omake`, and also `clearmake` in case you are using IBM Rational ClearCase as the version control tool. Let's see the important ones.

Maven

Maven is a build tool used mostly to compile Java code. It uses Java libraries and Maven plugins in order to compile the code. The code to be built is described using an XML file that contains information about the project being built, dependencies, and so on.

Maven can be easily integrated into Continuous Integration tools, such as Jenkins, using plugins.

MSBuild

MSBuild is a tool used to build Visual Studio projects. MSBuild is bundled with Visual Studio. MSBuild is a functional replacement for `nmake`. MSBuild works on project files, which have the XML syntax, similar to that of Apache Ant. Its fundamental structure and operation are similar to that of the Unix `make` utility. The user defines what will be the input (the various source codes), and the output (usually, a `.exe` or `.msi`). But, the utility itself decides what to do and the order in which to do it.

Automating the deployments

Consider an example, where the automated packaging has produced a package that contains `.war` files, database scripts, and some Unix configuration files. Now, the task here is to deploy all the three artifacts into their respective environments. The `.war` files must be deployed in the application server. The Unix configuration files should sit on the respective Unix machine, and lastly, the database scripts should be executed in the database server. The deployment of such packages containing multiple components is usually done manually in almost every organization that does not have automation in place. The manual deployment is slow and prone to human errors. This is where the automated deployment mechanism is helpful.

Automated deployment goes hand in hand with the automated build process. The previous scenario can be achieved using an automated build and deployment solution that builds each component in parallel, packages them, and then deploys them in parallel. Using tools such as Jenkins, this is possible. However, there are some challenges, which are as follows:

- There is a considerable amount of scripting required to orchestrate build packaging and deployment of a release containing multiple components. These scripts by themselves are huge code to maintain that require time and resources.

- In most of the cases, deployment is not as simple as placing files in a directory. For example, there are situations where the deployment activity is preceded by steps to configure the environment.

> The field of managing the configuration on multiple machines is called **configuration management**. There are tools, such as Chef and Puppet, to do this.

Automating the testing

Testing is an important part of a software development life cycle. In order to maintain quality software, it is necessary that the software solution goes through various test scenarios. Giving less importance to testing can result in customer dissatisfaction and a delayed product.

Since testing is a manual, time-consuming, and repetitive task, automating the testing process can significantly increase the speed of software delivery. However, automating the testing process is a bit more difficult than automating the build, release, and deployment processes. It usually takes a lot of efforts to automate nearly all the test cases used in a project. It is an activity that matures over time.

Hence, when we begin to automate the testing, we need to take a few factors into consideration. Test cases that are of great value and easy to automate must be considered first. For example, automate the testing where the steps are the same, but they run every time with different data. You can also automate the testing where a software functionality is being tested on various platforms. In addition, automate the testing that involves a software application running on different configurations.

Previously, the world was mostly dominated by the desktop applications. Automating the testing of a GUI-based system was quite difficult. This called for scripting languages where the manual mouse and keyboard entries were scripted and executed to test the GUI application. Nevertheless, today the software world is completely dominated by the web and mobile-based applications, which are easy to test through an automated approach using a test automation tool.

Once the code is built, packaged, and deployed, testing should run automatically to validate the software. Traditionally, the process followed is to have an environment for SIT, UAT, PT, and Pre-Production. First, the release goes through SIT, which stands for System Integration Test. Here, testing is performed on an integrated code to check its functionality all together. If pass, the code is deployed in the next environment, that is, UAT where it goes through a user acceptance test, and then similarly, it can lastly be deployed in PT where it goes through the performance test. Thus, in this way, the testing is prioritized.

It is not always possible to automate all of the testing. But, the idea is to automate whatever testing is possible. The previous method discussed requires the need to have many environments and also a number of automated deployments into various environments. To avoid this, we can go for another method where there is only one environment where the build is deployed, and then, the basic tests are run and after that, long running tests are triggered manually.

Use static code analysis

Static code analysis, also commonly called **white-box testing**, is a form of software testing that looks for the structural qualities of the code. For example, it reveals how robust or maintainable the code is. Static code analysis is performed without actually executing programs. It is different from the functional testing, which looks into the functional aspects of software and is dynamic.

Static code analysis is the evaluation of software's inner structures. For example, is there a piece of code used repetitively? Does the code contain lots of commented lines? How complex is the code? Using the metrics defined by a user, an analysis report can be generated that shows the code quality in terms of maintainability. It doesn't question the code functionality.

Some of the static code analysis tools, such as SonarQube come with a dashboard, which shows various metrics and statistics of each run. Usually, as part of Continuous Integration, the static code analysis is triggered every time a build runs. As discussed in the previous sections, static code analysis can also be included before a developer tries to check-in his code. Hence, code of low quality can be prevented right at the initial stage.

Static code analysis support many languages, such as Java, C/C++, Objective-C, C#, PHP, Flex, Groovy, JavaScript, Python, PL/SQL, COBOL, and so on.

Automate using scripting languages

One of the most important parts, or shall we say the backbone of Continuous Integration are the scripting languages. Using these, we can reach where no tool reaches. In my own experience, there are many projects where build tools, such as Maven, Ant, and the others don't work. For example, the SAS Enterprise application has a GUI interface to create packages and perform code promotions from environment to environment. It also offers a few APIs to do the same through the command line. If one has to automate the packaging and code promotion process in a project that is based on SAS, then one ought to use the scripting languages.

Perl

One of my favorites, Perl is an open source scripting language. It is mainly used for text manipulation. The main reasons for its popularity are as follows:

- It comes free and preinstalled with any Linux and Unix OS
- It's also freely available for Windows
- It is simple and fast to script using Perl
- It works both on Windows, Linux, and Unix platforms

Though it was meant to be just a scripting language for processing files, nevertheless it has seen a wide range of usages in the areas of system administration, build, release and deployment automation, and much more. One of the other reasons for its popularity is the impressive collection of third-party modules.

I would like to expand on the advantages of the multiple platform capabilities of Perl. There are situations where you will have Jenkins servers on a Windows machine, and the destination machines (where the code needs to be deployed) will be Linux machines. This is where Perl helps; a single script written on the Jenkins Master will run on both the Jenkins Master and the Jenkins Slaves.

However, there are various other popular scripting languages that you can use, such as Ruby, Python, and Shell to name a few.

Test in a production-like environment

Ideally testing such as SIT, UAT, and PT to name a few, is performed in an environment that is different from the production. Hence, there is every possibility that the code that has passed these quality checks may fail in production. Therefore, it's advisable to perform an end-to-end testing on the code in a production-like environment, commonly referred to as a pre-production environment. In this way, we can be best assured that the code won't fail in production.

However, there is a challenge to this. For example, consider an application that runs on various web browsers both on mobiles and PCs. To test such an application effectively, we would need to simulate the entire production environment used by the end users. These call for multiple build configurations and complex deployments, which are manual. Continuous Integration systems need to take care of this; on a click of a button, various environments should be created each reflecting the environment used by the customers. And then, this should be followed by deployment and testing thereafter.

Backward traceability

If something fails, there should be an ability to see when, who, and what caused the failure. This is called as backward traceability. How do we achieve it? Let's see:

- By introducing automated notifications after each build. The moment a build is completed, the Continuous Integration tools automatically respond to the development team with the report card.
- As seen in the Scrum methodology, the software is developed in pieces called backlogs. Whenever a developer checks in the code, they need to apply a label on the checked-in code. This label can be the backlog number. Hence, when the build or a deployment fails, it can be traced back to the code that caused it using the backlog number.
- Labeling each build also helps in tracking back the failure.

Using a defect tracking tool

Defect tracking tools are a means to track and manage bugs, issues, tasks, and so on. Earlier projects were mostly using Excel sheets to track their defects. However, as the magnitude of the projects increased in terms of the number of test cycles and the number of developers, it became absolutely important to use a defect tracking tool. Two of the most popular defect tracking tools are Atlassian JIRA and Bugzilla.

The quality analysis market has seen the emergence of various bug tracking systems or defect management tools over the years.

A defect tracking tools offers the following features:

- It allows you to raise or create defects and tasks that have got various fields to define the defect or the task.
- It allows you to assign the defect to the concerned team or an individual responsible for the change.
- It progresses through the life cycle stages workflow.

- It provides you with the feature to comment on a defect or a task, watch the progress of the defect, and so on.
- It provides metrics. For example, how many tickets were raised in a month? How much time was spent on resolving the issues? All these metrics are of significant importance to the business.
- It allows you to attach a defect to a particular release or build for better traceability.

The previously mentioned features are a must for a bug tracking system. There may be many other features that a defect tracking tool may offer, such as voting, estimated time to resolve, and so on.

Continuous Integration benefits

The way a software is developed always affects the business. The code quality, the design, time spent in development and planning of features, all affect the promises that a company has made to its clients.

Continuous Integration helps the developers in helping the business. While going through the previous topics, you might have already figured out the benefits of implementing Continuous Integration. However, let's see some of the benefits that Continuous Integration has to offer.

Freedom from long integrations

When every small change in your code is built and integrated, the possibility of catching the integration errors at an early stage increases. Rather than integrating once in 6 months, as seen in the waterfall model, and then spending weeks resolving the merge issues, it is good to integrate frequently and avoid the merge hell. The Continuous Integration tool like Jenkins automatically builds and integrates your code upon check-in.

Production-ready features

Continuous Delivery enables you to release deployable features at any point in time. From a business perspective, this is a huge advantage. The features are developed, deployed, and tested within a timeframe of 2 to 4 weeks and are ready to go live with a click of a button.

Analyzing and reporting

How frequent are the releases? What is the success rate of builds? What is the thing that is mostly causing a build failure? Real-time data is always a must in making critical decisions. Projects are always in the need of recent data to support decisions. Usually, managers collect this information manually, which requires time and efforts. Continuous Integration tools, such as Jenkins provide the ability to see trends and make decisions. A Continuous Integration system provides the following features:

- Real-time information on the recent build status and code quality metrics.
- Since integrations occur frequently with a Continuous Integration system, the ability to notice trends in build, and overall quality becomes possible.

Continuous Integration tools, such as Jenkins provide the team members with metrics about the build health. As all the build, packaging, and deployment work is automated and tracked using a Continuous Integration tool; therefore, it is possible to generate statistics about the health of all the respective tasks. These metrics can be the build failure rate, build success rate, the number of builds, who triggered the build, and so on.

All these trends can help project managers and the team to ensure that the project is heading in the right direction and at the right pace.

Also, Continuous Integration incorporates static code analysis, which again on every build gives a static report of the code quality. Some of the metrics of great interest are code style, complexity, length, and dependency.

Catch issues faster

This is the most important advantage of having a carefully implemented Continuous Integration system. Any integration issue or merge issue gets caught early. The Continuous Integration system has the facility to send notifications as soon as the build fails.

Spend more time adding features

In the past, development teams performed the build, release, and deployments. Then, came the trend of having a separate team to handle build, release, and deployment work. Yet again that was not enough, as this model suffered from communication issues between the development team and the release team.

However, using Continuous Integration, all the build, release, and the deployment work gets automated. Therefore, now the development team need not worry about anything other than developing features. In most of the cases, even the completed testing is automated.

Rapid development

From a technical perspective, Continuous Integration helps teams work more efficiently. This is because Continuous Integration works on the agile principles. Projects that use Continuous Integration follow an automatic and continuous approach while building, testing, and integrating their code. This results in a faster development.

Since everything is automated, developers spend more time developing their code and zero time on building, packaging, integrating, and deploying it. This also helps teams, which are geographically distributed, to work together. With a good software configuration management process in place, people can work on large teams. **Test Driven Development** (**TDD**) can further enhance the agile development by increasing its efficiency.

Summary

> "*Behind every successful agile project, there is a Continuous Integration server.*"

Looking at the evolutionary history of the software engineering process, we now know how Continuous Integration came into existence. Truly, Continuous Integration is a process that helps software projects go agile.

The various concepts, terminologies, and best practices discussed in this chapter form a foundation for the upcoming chapters. Without these, the upcoming chapters are mere technical know-how.

In this chapter, we also learned how various DevOps tools go hand-in-hand to achieve Continuous Integration, and of course, help projects go agile. We can fairly conclude that Continuous Integration is an engineering practice where each chunk of code is immediately built and unit-tested, then integrated and again built and tested on the Integration branch.

We also learned how feedback forms an important part of a Continuous Integration system.

Continuous Integration depends incredibly on automation of various software development processes. This also means that using a Continuous Integration tool alone doesn't help in achieving Continuous Integration, and Continuous Integration does not guarantee zero bugs. But it guarantees early detection.

2
Setting up Jenkins

The current chapter is all about installing Jenkins across various platforms. We will begin with a short introduction to Jenkins and the components that make it. We will also explore why Jenkins is a better choice than the other Continuous Integration tools, and how it fits in as a Continuous Integration server.

Later in the chapter, we will go through an in-detail installation of Jenkins inside a container (the Apache Tomcat server), followed by an analysis of the merits of such an approach, thereby answering why most organizations choose to use Jenkins inside a Web Server.

Last but not least, we will see how to install Jenkins across various types of operating systems as a standalone application.

Introduction to Jenkins

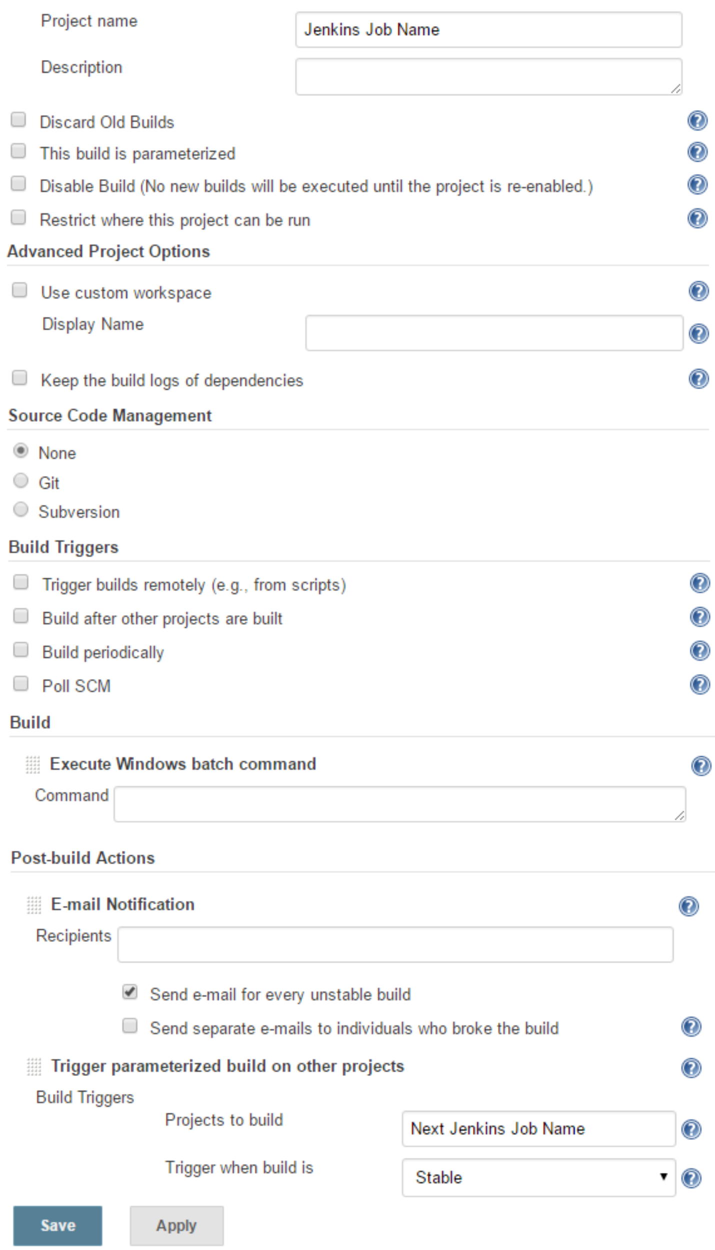
Jenkins is an open source Continuous Integration tool. However, it's not limited to Continuous Integration alone. In the upcoming chapters, we will see how Jenkins can be used to achieve Continuous Delivery, Continuous Testing, and Continuous Deployment. Jenkins is supported by a large number of plugins that enhance its capability. The Jenkins tool is written in Java and so are its plugins. The tool has a minimalistic GUI that can be enhanced using specific plugins if required.

What is Jenkins made of?

Let's have a look at the components that make up Jenkins. The Jenkins framework mainly contains jobs, builds, parameters, pipelines and plugins. Let's look at them in detail.

Jenkins job

At a higher level, a typical Jenkins job contains a unique name, a description, parameters, build steps, and post-build actions. This is shown in the following screenshot:

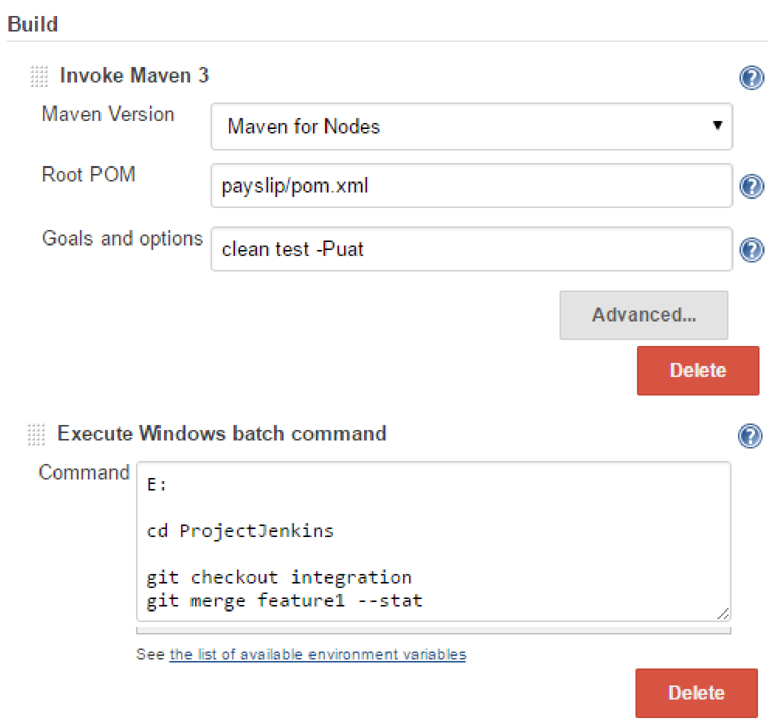


Jenkins parameters

Jenkins parameters can be anything: environment variables, interactive values, pre-defined values, links, triggers, and so on. Their primary purpose is to assist the builds. They are also responsible for triggering pre-build activities and post-build activities.

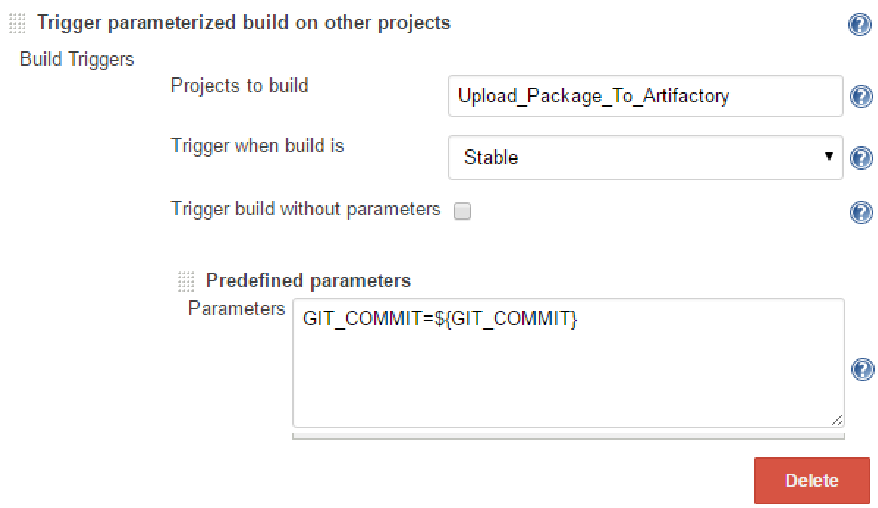
Jenkins build

A Jenkins build (not to be confused with a software build) can be anything from a simple Windows batch command to a complex Perl script. The range is extensive, which include Shell, Perl, Ruby, and Python scripts or even Maven and Ant builds. There can be number of build steps inside a Jenkins job and all of them run in sequence. The following screenshot is an example of a Maven build followed by a Windows batch script to merge code:



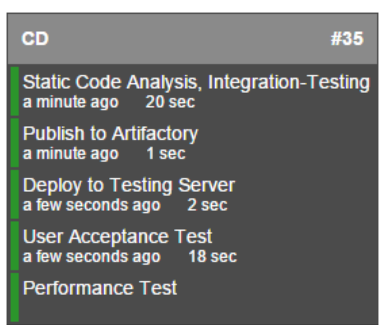
Jenkins post-build actions

Post-build actions are parameters and settings that define the subsequent steps to be performed after a build. Some post-build actions can be configured to perform various activities depending on conditions. For example, we can have a post-build action in our current job, which in the event of a successful build starts another Jenkins job. This is shown in the following screenshot:



Jenkins pipeline

Jenkins pipeline, in simple terms, is a group of multiple Jenkins jobs that run in sequence or in parallel or a combination of both. The following screenshot is an example of a Jenkins Continuous Delivery pipeline. There are five separate Jenkins jobs, all running one after the other.

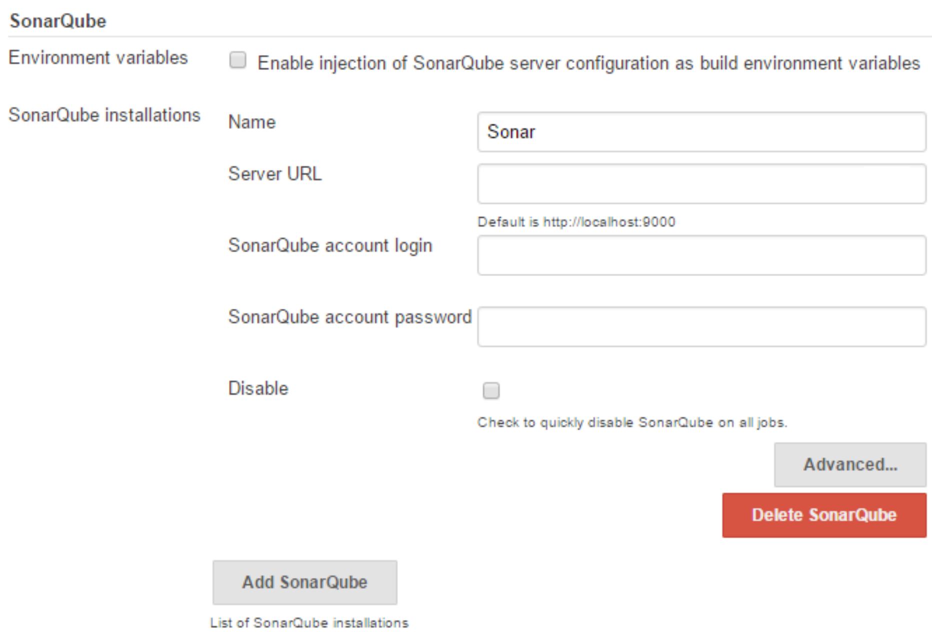


> Jenkins Pipeline is used to achieve a larger goal, like Continuous Integration or Continuous Delivery.

Jenkins plugins

Jenkins plugins are software pieces that enhance the Jenkins' functionality. Plugins after installation, manifest in the form of either system settings or parameters inside a Jenkins job.

There is a special section inside the Jenkins master server to manage plugins. The following screenshot shows the Jenkins system configuration section. It's a setting to configure the SonarQube tool (a static code analysis tool). The configuration is available only after installing the Jenkins plugin for SonarQube named **sonar**.



Why use Jenkins as a Continuous Integration server?

DevOps engineers across the world have their own choice when it comes to Continuous Integration tools. Yet, Jenkins remains an undisputed champion among all. The following are some of the advantages of using Jenkins.

It's open source

There are a number of Continuous Integration tools available in the market, such as Go, Bamboo, TeamCity, and so on. But the best thing about Jenkins is that it's free, simple yet powerful, and popular among the DevOps community.

Community-based support

Jenkins is maintained by an open source community. The people who created the original Hudson are all working for Jenkins after the Jenkins-Hudson split.

Lots of plugins

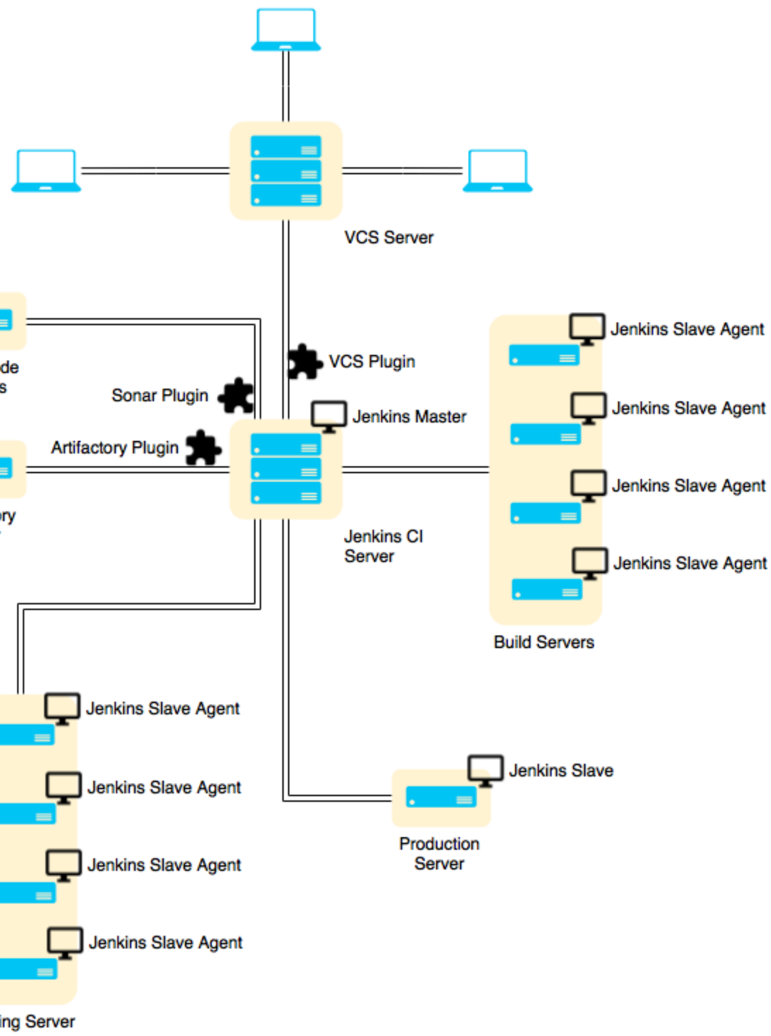
There are more than 300 plugins available for Jenkins and the list keeps increasing. Plugins are simple Maven projects. Therefore, anyone with a creative mind can create and share their plugins on the Jenkins community to serve a purpose.

Jenkins has a cloud support

There are times when the number of builds, packaging, and deployment requests are more, and other times they are less. In such scenarios, it is necessary to have a dynamic environment to perform builds. This can be achieved by integrating Jenkins with a cloud-based service such as AWS. With this set up, build environments can be created and destroyed automatically as per demand.

Jenkins as a centralized Continuous Integration server

Jenkins is clearly an orchestrator. It brings all the other DevOps tools together in order to achieve Continuous Integration. This is clearly depicted in the next screenshot. We can see Jenkins communicating with the version control tool, repository tool, and static code analysis tool using plugins. Similarly, Jenkins communicates with the build servers, testing servers, and the production server using the Jenkins slave agent.



Hardware requirements

Answering the hardware requirements of Jenkins is quite a challenge. Ideally, a system with Java 7 or above and 1-2 GB RAM is enough to run Jenkins master server. However, there are organizations that go way up to 60+ GB RAM for their Jenkins Master Server alone.

Therefore, hardware specifications for a Jenkins master server largely depend on the organization's requirements. Nevertheless, we can make a connection between the Jenkins operations and the hardware as follows:

- The number of users accessing Jenkins master server (number of HTTP requests) will cost mostly the CPU.
- The number of Jenkins slaves connected to Jenkins master server will cost mostly the RAM.
- The number of jobs running on a Jenkins master server will cost the RAM and the disk space.
- The number of builds running on a Jenkins master server will cost the RAM and the disk space (this can be ignored if builds happen on Jenkins slave machines).

> Refer to the sample use cases at the end of the chapter.

Running Jenkins inside a container

Jenkins can be installed as a service inside the following containers:

- Apache Geronimo 3.0
- Glassfish
- IBM WebSphere
- JBoss
- Jetty
- Jonas

- Liberty profile
- Tomcat
- WebLogic

In the current section, we will see how to install Jenkins on the Apache Tomcat server.

Installing Jenkins as a service on the Apache Tomcat server

Installing Jenkins as a service on the Apache Tomcat server is pretty simple. We can either choose to use Jenkins along with other services already present on the Apache Tomcat server, or we may use the Apache server solely for Jenkins.

Prerequisites

I assume that the Apache Tomcat server is installed on the machine where you intend to run Jenkins. In the following section, we will use the Apache Tomcat server 8.0. Nevertheless, Apache Tomcat server 5.0 or greater is sufficient to use Jenkins. A machine with 1 GB RAM is enough to start with. However, as the number of jobs and builds increase, so should the memory.

We also need Java running on the machine. In this section, we are using jre1.8.0_60. While installing the Apache Tomcat server, you will be asked to install Java. Nevertheless, it is suggested that you always use the latest stable version available.

> The current section focuses on running Jenkins inside a container like Apache Tomcat. Therefore, the underlying OS where the Apache Tomcat server is installed can be anything. We are using Windows 10 OS in the current subtopic.

Perform the following steps for installing Jenkins inside a container:

1. Download the latest `jenkins.war` file from `https://jenkins.io/download/`.
2. Click on the **Download Jenkins** link, as shown in the following screenshot:

3. You will be presented with an option to download **LTS Release** and **Weekly Release**.

4. Choose the LTS Release by clicking on the `1.642.4.war` link, as shown in the following screenshot. Do not click on the dropdown menu.

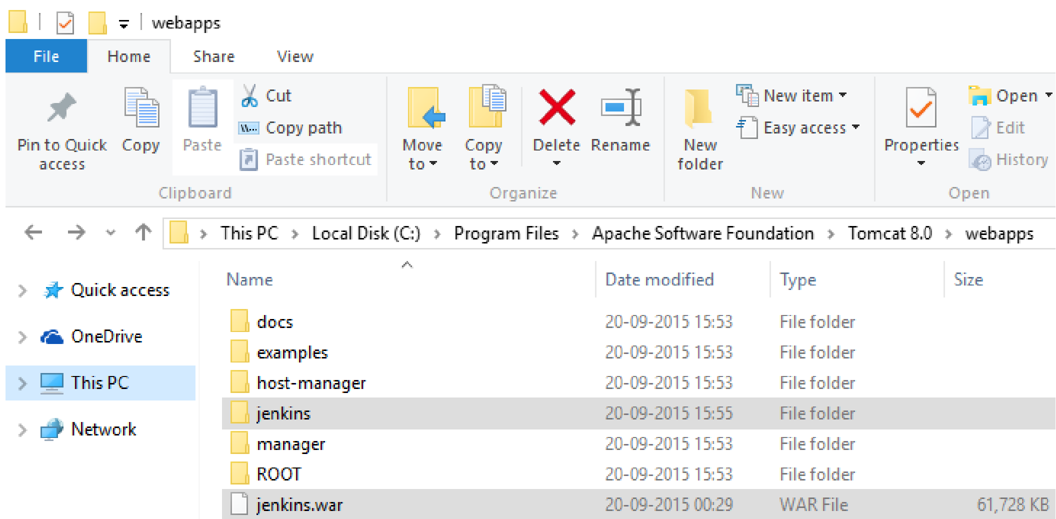
> At the time of writing this book, `1.642.4.war` was the latest Jenkins LTS version available. Readers are free to select whatever Jenkins LTS version appears on their screen.

> Clicking on the dropdown button will provide you with the standalone package for various operating systems.

Installing Jenkins along with other services on the Apache Tomcat server

An organization can follow the current approach if they do not wish to have individual servers for Jenkins master alone, but want to host it along with other services that are already running on their Apache Tomcat server. The steps are as follows:

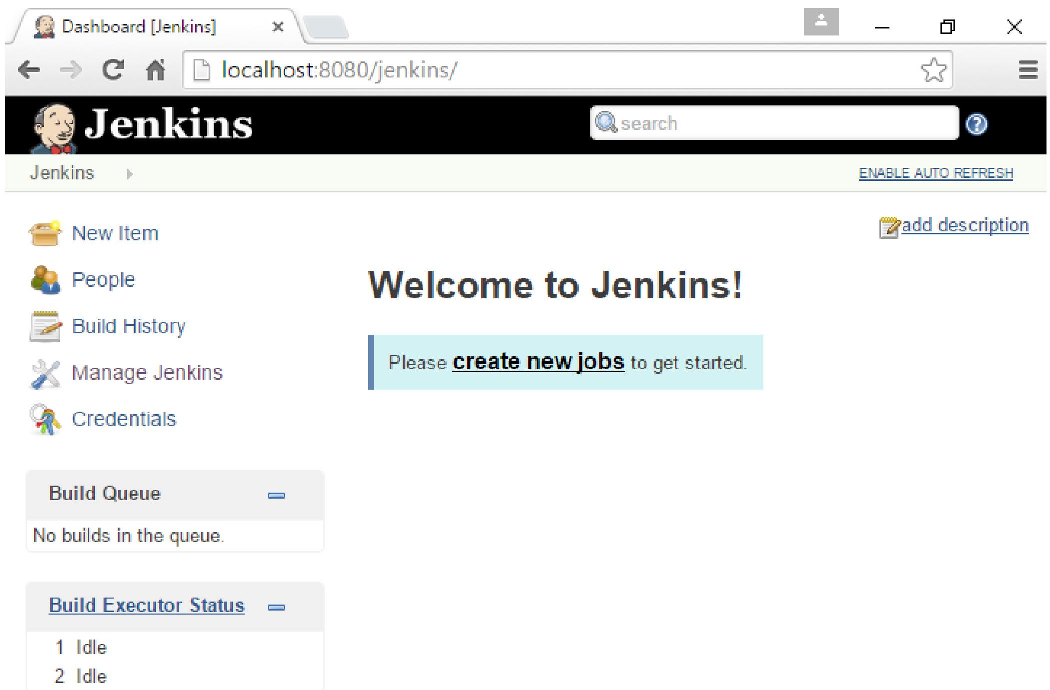
1. Simply move the downloaded `jenkins.war` file to the `webapps` folder, which is present inside the installation directory of your Apache Tomcat server. In our case, it's `C:\Program Files\Apache Software Foundation\Tomcat 8.0\webapps`.



> You will notice that a `jenkins` folder automatically gets created the moment you move the `jenkins.war` package to the `webapps` folder. This is because the `.war` file is a **Web Application Archive** file that automatically gets extracted once deployed to the `webapps` directory. We did a small deployment activity.

2. That's all you need to do. You can access Jenkins using the URL `http://localhost:8080/jenkins`.

3. The Jenkins Dashboard is shown in the following screenshot:



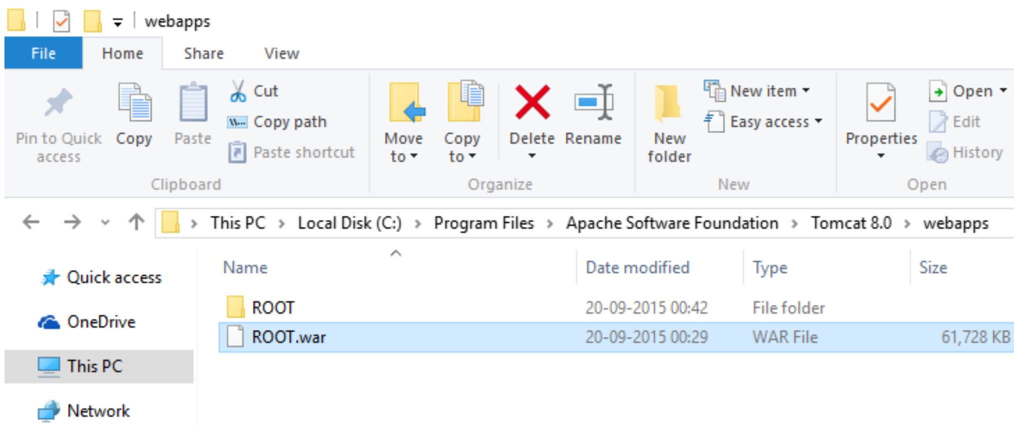
Installing Jenkins alone on the Apache Tomcat server

On the other hand, if you chose to have the Apache Tomcat server solely for using Jenkins then in that case perform the following steps:

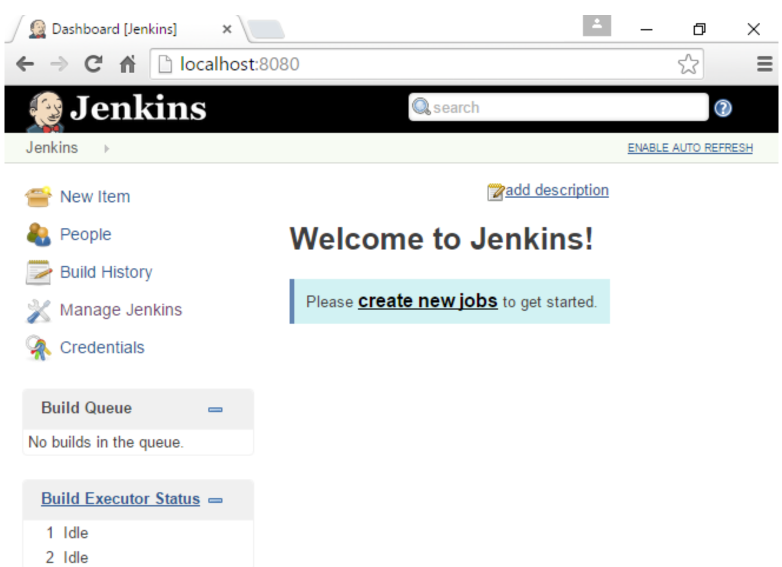
1. Rename the downloaded `jenkins.war` package to `ROOT.war`.
2. Next, delete everything inside the `webapps` folder.

3. Now move the `ROOT.war` (renamed) package to the `webapps` folder. In the end, everything should look like the following screenshot.

> It's always recommended to have a dedicated web server solely for Jenkins.



4. In this way, you can access Jenkins using the URL `http://localhost:8080/` without any additional path. Apparently, the Apache server is now a Jenkins server.



> In the preceding screenshot, we can see a folder named `ROOT` inside the `webapps` folder. This `ROOT` folder gets generated automatically as we move the `ROOT.war` file to the `webapps` folder.
>
> Deleting the content inside the `webapps` folder (leaving behind the original `ROOT` directory and `ROOT.war`) and then moving the `jenkins.war` file to the `webapps` folder is also sufficient to make the Apache Tomcat server solely for Jenkins' use.
>
> The step of renaming `jenkins.war` to `ROOT.war` is only necessary if you want to make `http://localhost:8080/` the standard URL for Jenkins.

Setting up the Jenkins home path

Before we start using Jenkins, there is one important thing to configure: the `JENKINS_HOME` path. This is the location where all of the Jenkins configurations, logs, and builds are stored. Everything that you create and configure on the Jenkins dashboard is stored here.

In our case, by default, the `JENKINS_HOME` variable is set to `C:\Windows\System32\config\systemprofile\.jenkins`. We need to make it something more accessible, for example, `C:\Jenkins`. This can be done in two ways.

Method 1 – configuring the context.xml file

`Context.xml` is a configuration file related to the Apache Tomcat server. We can configure the `JENKINS_HOME` variable inside it using the following steps:

1. Stop the Apache Tomcat server.
2. Go to `C:\Program Files\Apache Software Foundation\Tomcat 8.0\conf`.
3. Modify the `context.xml` file using the following code:

```
<Context>
<Environment name="JENKINS_HOME" value="C:\Jenkins" type="java.
lang.String"/>
</Context>
```

4. After modifying the file, start the Apache Tomcat server.

To stop the Apache Tomcat server on Windows, run the following command in the command prompt as an admin: `net stop Tomcat8`.

To start the Apache Tomcat server on Windows, run the following command in the command prompt as an admin: `net start Tomcat8`.

Method 2 – creating the JENKINS_HOME environment variable

We can create the `JENKINS_HOME` variable using the following steps:

1. Stop the Apache Tomcat server.
2. Now, open the Windows command prompt and run the following command:

   ```
   setx JENKINS_HOME "C:\Jenkins"
   ```

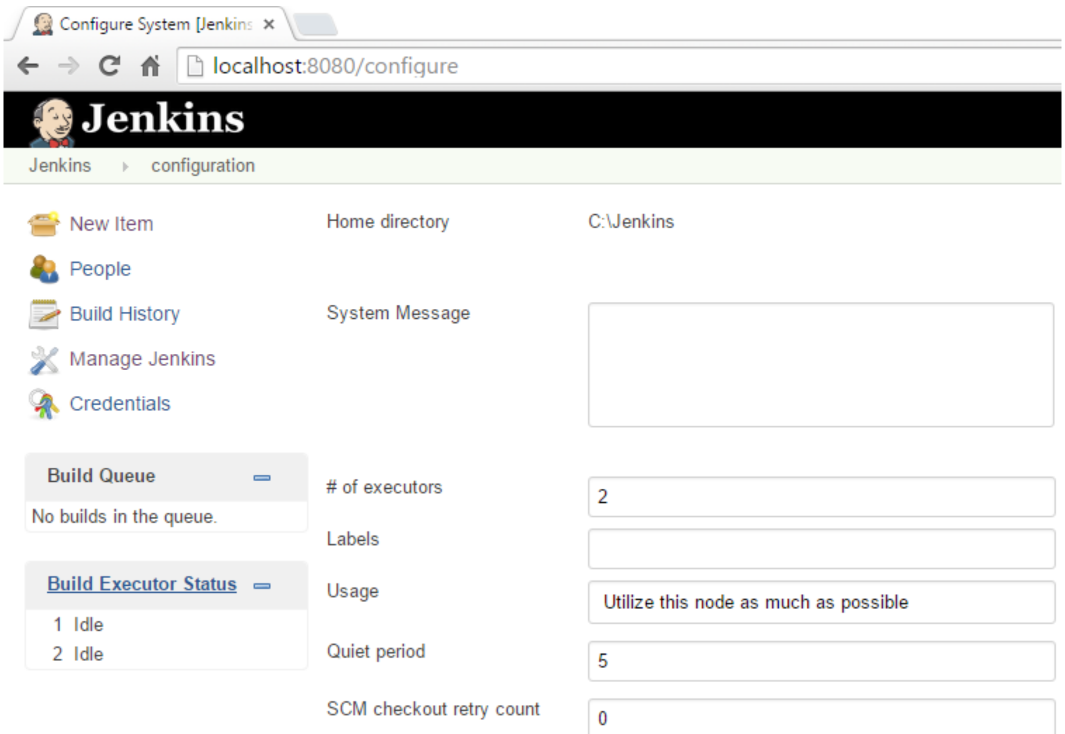
3. After executing the command, check the value of `JENKINS_HOME` with the following command:

   ```
   echo %JENKINS_HOME%
   ```

4. The output should be:

   ```
   C:\Jenkins
   ```

5. Start the Apache Tomcat server.
6. To check if the Jenkins home path is set to `C:\Jenkins`, open the following link: `http://localhost:8080/configure`. You should see the **Home** directory value set to `C:\Jenkins`, as shown in the following screenshot:



Why run Jenkins inside a container?

The reason that most organizations choose to use Jenkins on a web server is the same as the reason most organizations use web servers to host their websites: better traffic management.

The following factors affect Jenkins server performance:

- Number of jobs
- Number of builds
- Number of slaves
- Number of users accessing Jenkins server (number of HTTP requests)

All these factors can push organizations towards any one of the following tactics:

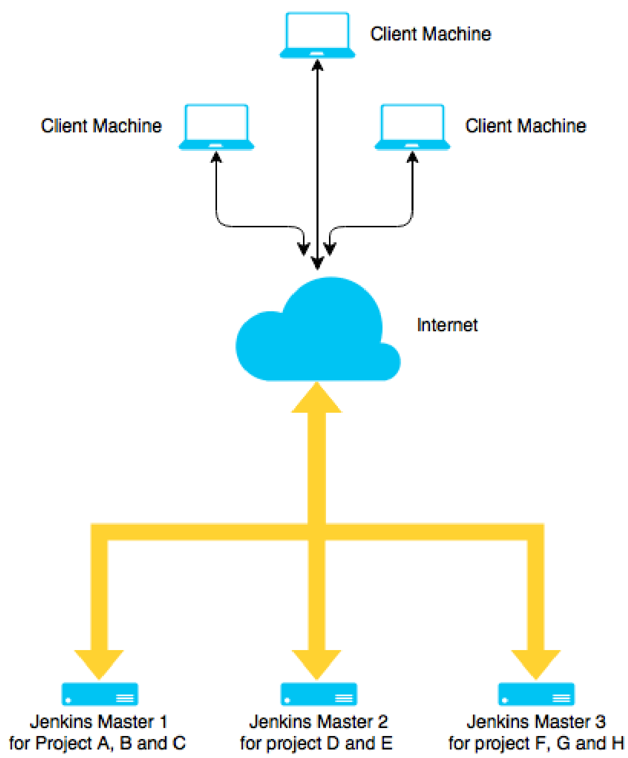
- **Approach 1**: Using multiple Jenkins masters, one each for every project
- **Approach 2**: Maintaining a single Jenkins master on a web server, with an enhanced hardware and behind a reverse proxy Server

The following table measures the merits of both tactics against few performance factors:

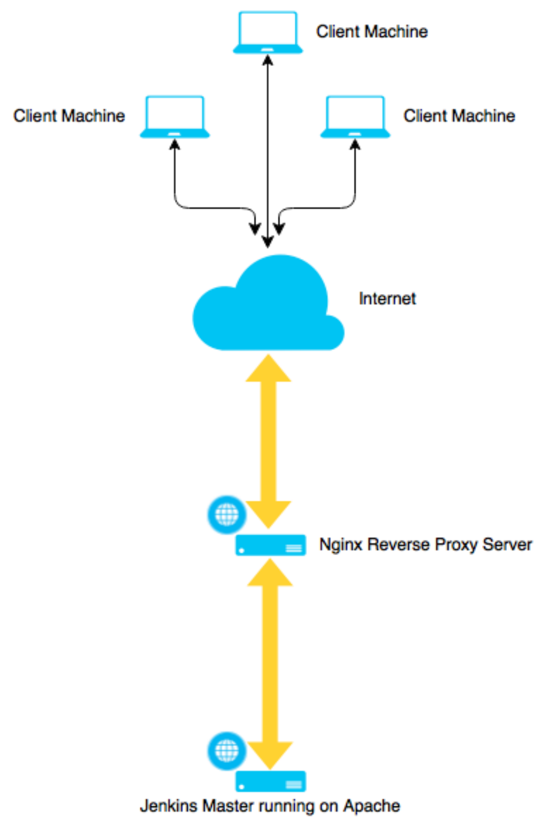
	Approach 1	**Approach 2**
Load balancing	Load balancing is achieved using multiple standalone Jenkins masters, spread across projects. Each Jenkins master has its own set of Jenkins slaves. Individual teams access their respective Jenkins servers.	The hardware is enhanced to manage a large number of builds and jobs.
Web acceleration	Nil	Reverse proxies can also compress inbound and outbound data. It can also cache frequently requested content. Using this feature, the traffic between Jenkins servers and the clients can be improved. Reverse proxy servers can also perform SSL encryption, thus reducing the load off your web servers, thereby boosting their performance. Web servers like Apache or NGINX can help maintain consistency across various clients during file transfer, without lashing up Jenkins threads.
Security	Security is limited to that configured in Jenkins.	A reverse proxy server acts as a defensive firewall against security threats using its Request intercepting feature.

	Approach 1	Approach 2
Administrator	Administration labor is greater, as there are multiple Jenkins masters to manage. Things like plugin updates or Jenkins updates, logs, and configurations need to be taken care separately for each Jenkins master.	Administrator is simple in case of a single Jenkins master.
Disaster Management	If any one of the Jenkins masters goes down, services related to it cease to function. However, the other Jenkins masters function uninterrupted.	Disaster management is limited and recovery is dependent solely on the Jenkins backup.

The following image shows **Approach 1**:



The following image demonstrates **Approach 2**:



Conclusion

The need to have Jenkins inside a container comes only when you think your Jenkins server is going to serve a very large group of projects and users. However, there is no point in using Jenkins within a container if your organization is small with a handful of Jenkins jobs and with no particular demand for scalability in the very near future.

Running Jenkins as a standalone application

Installing Jenkins as a standalone application is simpler than installing Jenkins as a service inside a container. Jenkins is available as a standalone application on the following operating systems:

- Windows
- Ubuntu/Debian
- Red Hat/Fedora/CentOS
- Mac OS X
- openSUSE
- FreeBSD
- openBSD
- Gentoo

Setting up Jenkins on Windows

There are two ways in which you can set up Jenkins on Windows. One is by using the Jenkins native package for Windows, and the other is through the `jenkins.war` file.

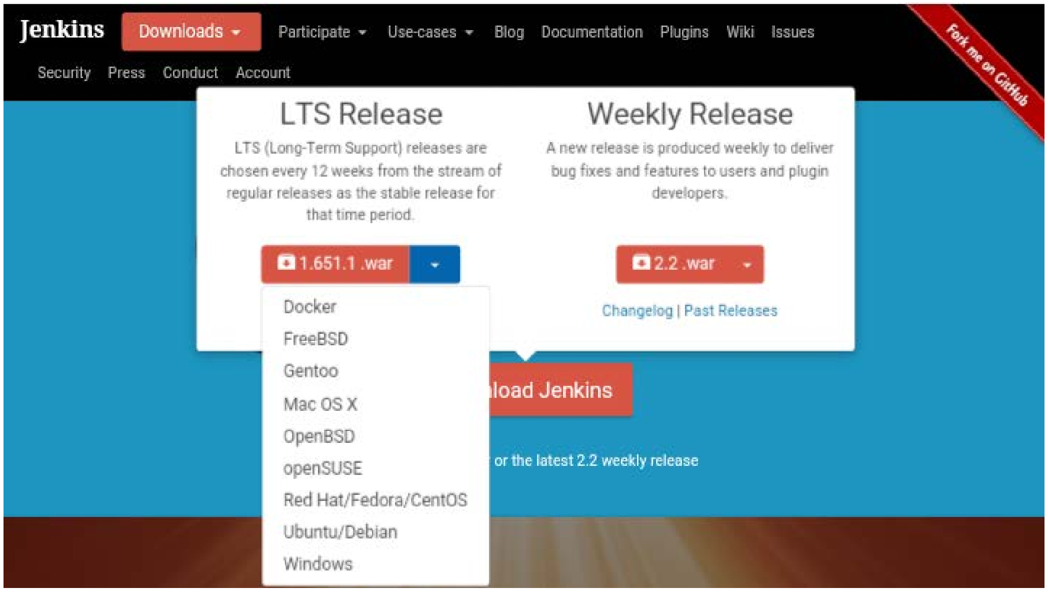
Installing Jenkins using the native Windows package

The following are the steps to install Jenkins using the native Windows package:

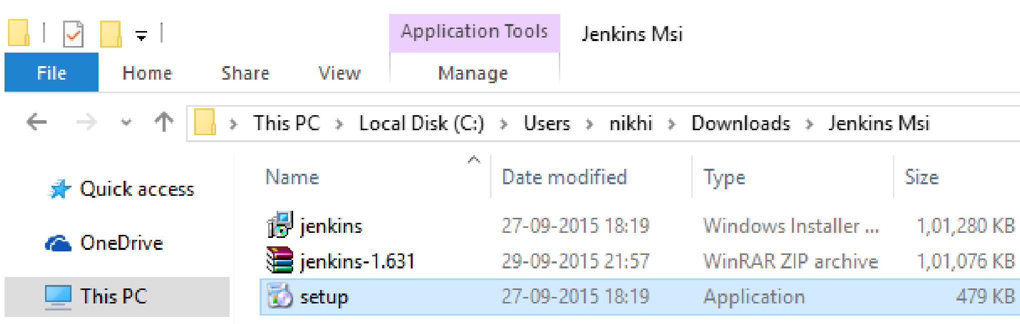
1. To download the latest stable Jenkins package for Windows go to the link `https://jenkins.io/download/`.

2. Once on the page, click on the **Download Jenkins** link, as shown in the following screenshot:

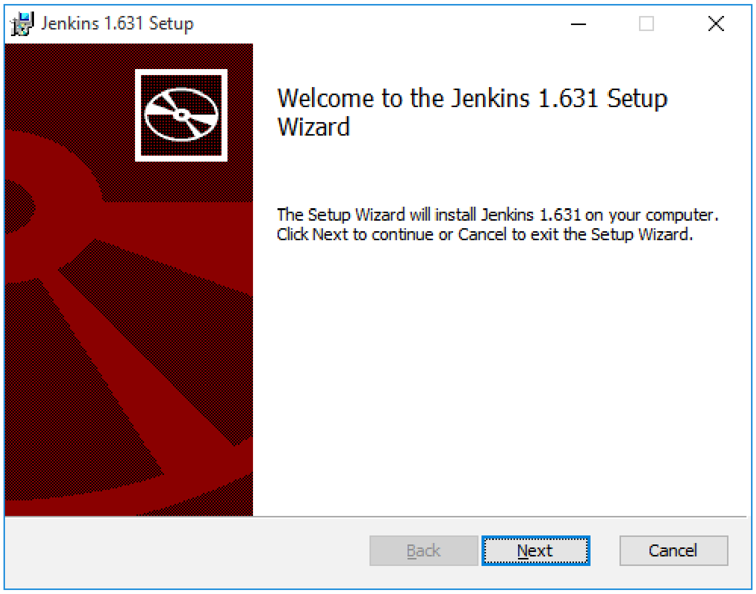
3. Now, click on the drop-down button and select **Windows**.



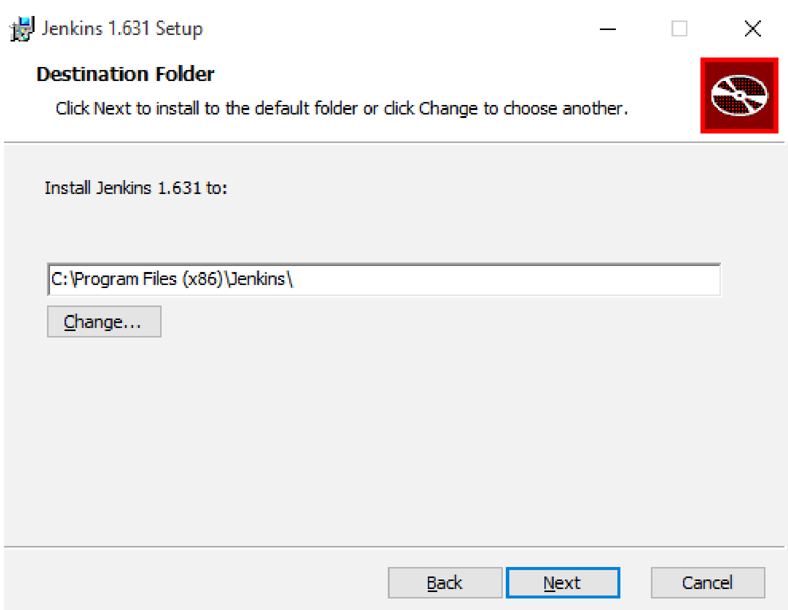
4. Once the download completes, unzip the archive file and you will find a `setup.exe`.



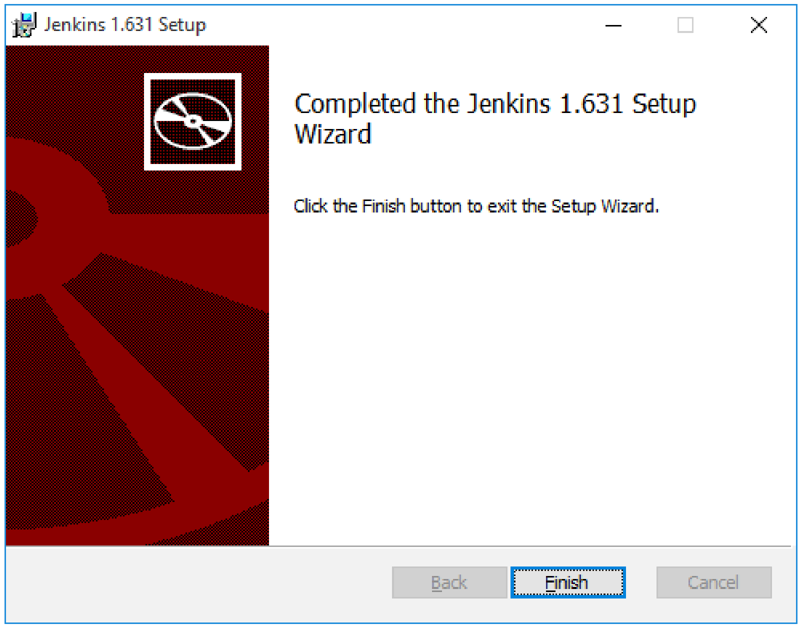
5. Run the `Setup.exe` and follow the installation.



6. During the installation process, you will get an option to choose your Jenkins installation directory (by default, it will be `C:\Program Files\Jenkins`). Leave it as it is and click on the **Next** button.



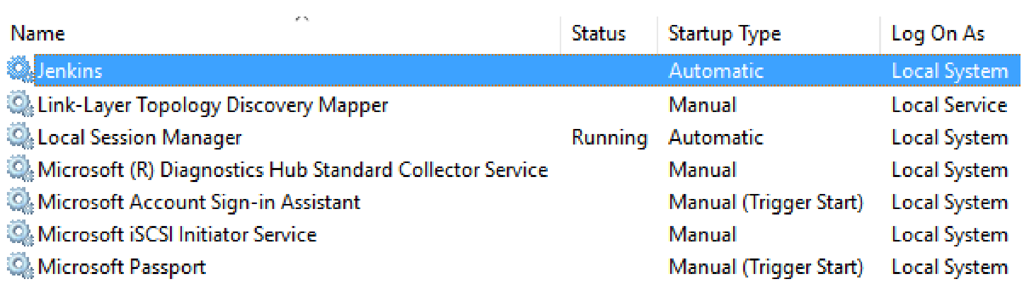
7. Click on the **Finish** button.



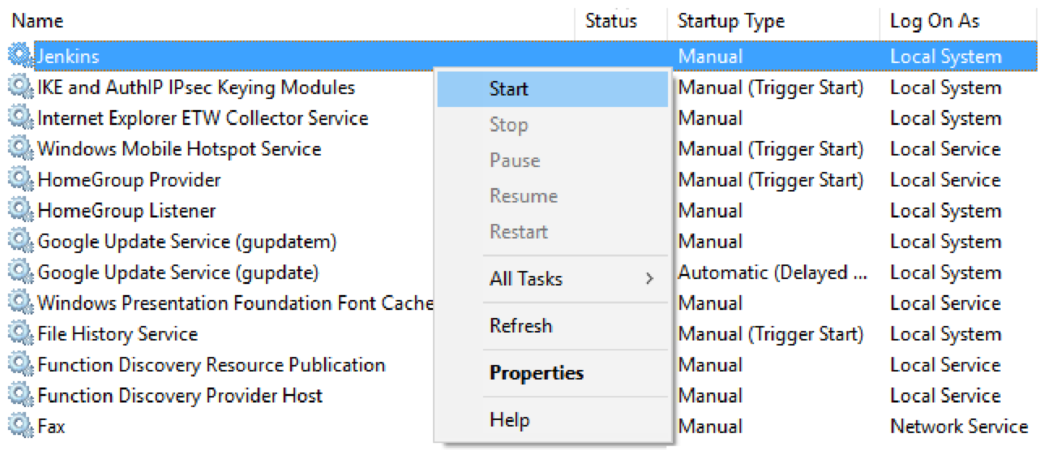
8. Upon successful installation, open the **Services** window from the command prompt using the following command:

   `services.msc`

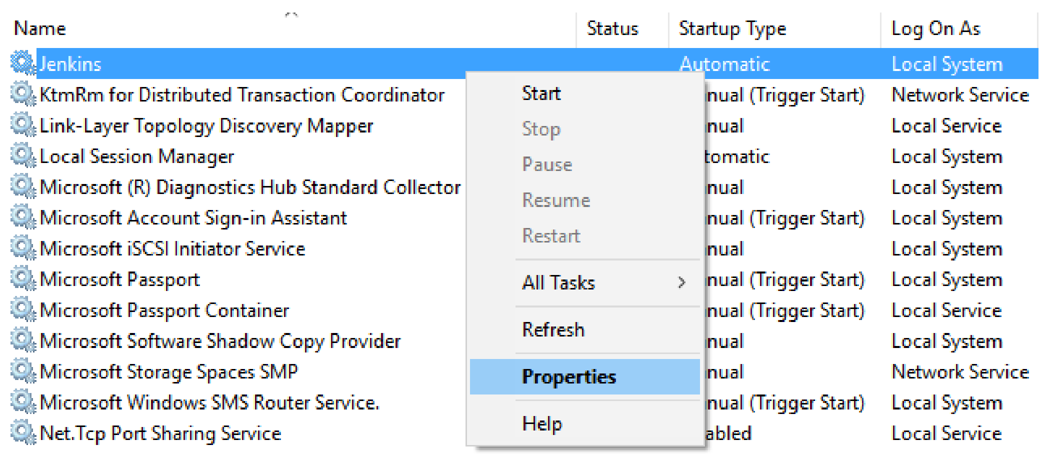
9. This will open the `services` window. Look for a service named `Jenkins`.



10. To start the service, right-click on the Jenkins service and click on **Start**, as shown in the following screenshot:

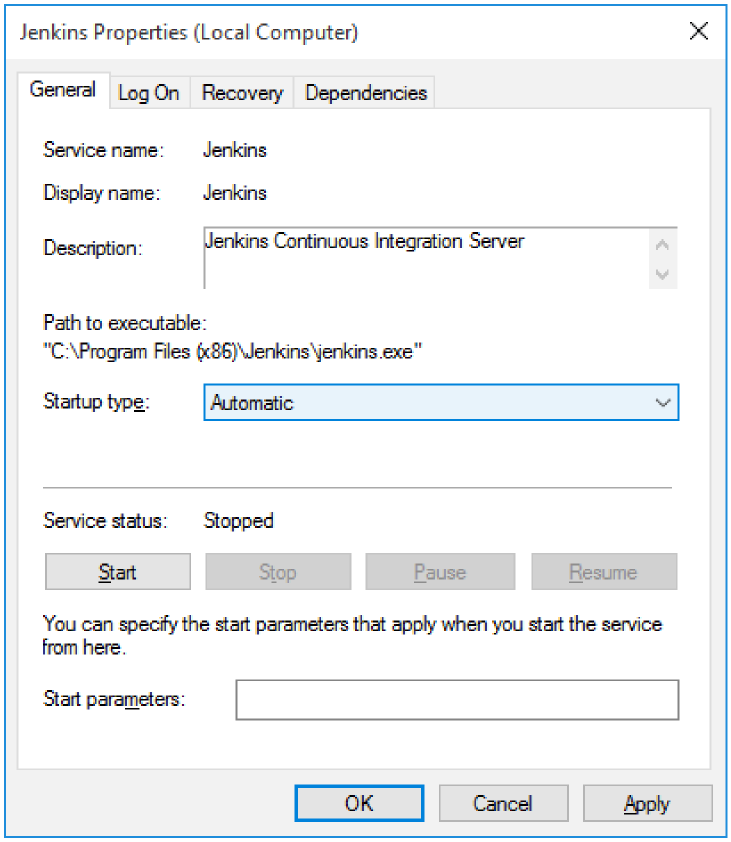


11. Right-click on the Jenkins service again and click on **Properties**, as shown in the following screenshot:



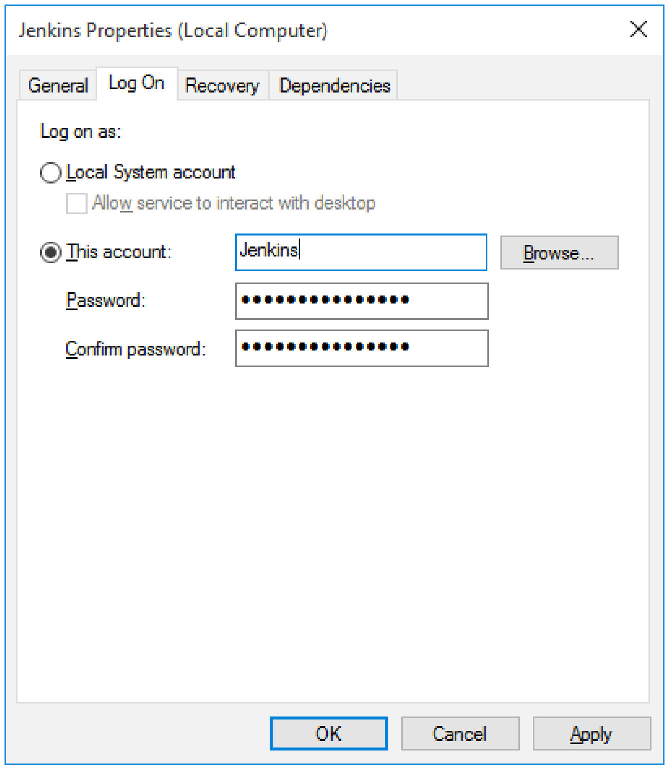
12. Under the **General** tab, you can see the Jenkins **Service name** field, the **Path to executable**, **Service status**, and the **Startup type** parameter.
13. Using the **Startup type** option, we can choose the way Jenkins starts on the Windows machine. We can choose from **Automatic**, **Manual**, and **Automatic (Delayed Start)**.
14. Always choose the **Automatic** option.

15. Below the **Service status** field, there is an option to manually **Start**, **Stop**, **Pause**, and **Resume** the Jenkins service.



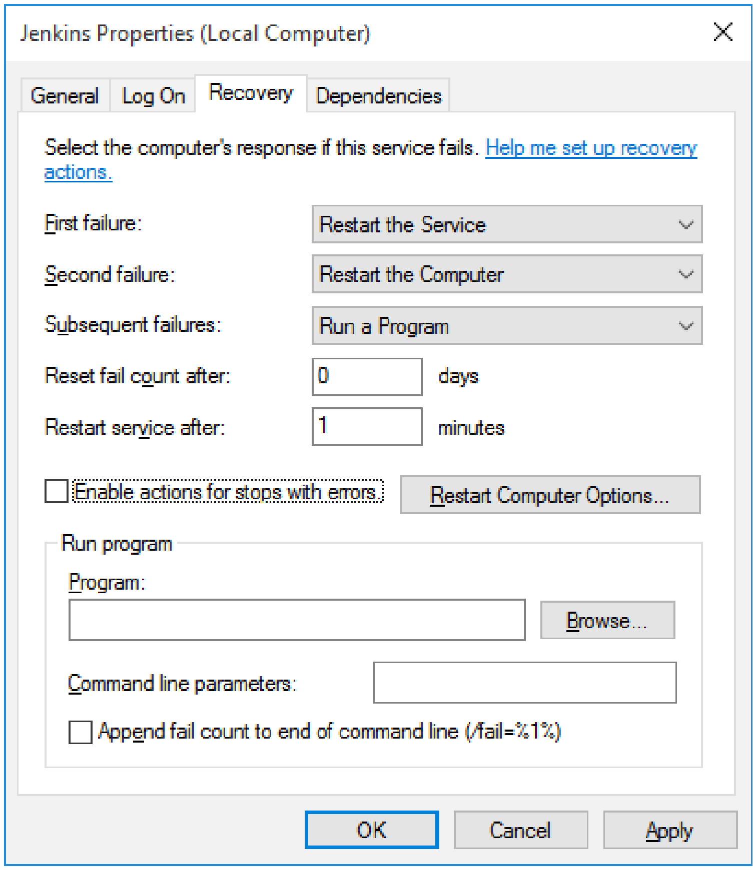
16. Come to the next tab, **Log On**. Here, we define the username with which Jenkins starts.
17. You can either choose to use the **Local System account** (not recommended) option or you can create a special user Jenkins with special permissions (recommended).

> An exclusive account for Jenkins is always preferred. The reason is that the **Local System account** option is not under control. The account might get deleted or the password may expire depending on the organization's policies, whereas the Jenkins user account can be set with preferred policies and privileges.



18. Next is the **Recovery** tab. Here, we can specify the action items in case the Jenkins service fails to start.

    Here's an example: on the **First failure** field, there is an attempt to restart Jenkins; in the **Second failure** field there is an attempt made to restart the computer. Lastly, in the **Subsequent failures** field, a program is run to debug the issue or we can run a script that sends the Jenkins failure log through mail to the Jenkins admin for investigation.



Installing Jenkins using the jenkins.war file

The following are the manual steps to install Jenkins using the `Jenkins.war` file. These steps can also be configured inside a script to automate the Jenkins installation. For example, you may want to install Jenkins remotely on a machine using automated scripts. Let's see the steps in detail.

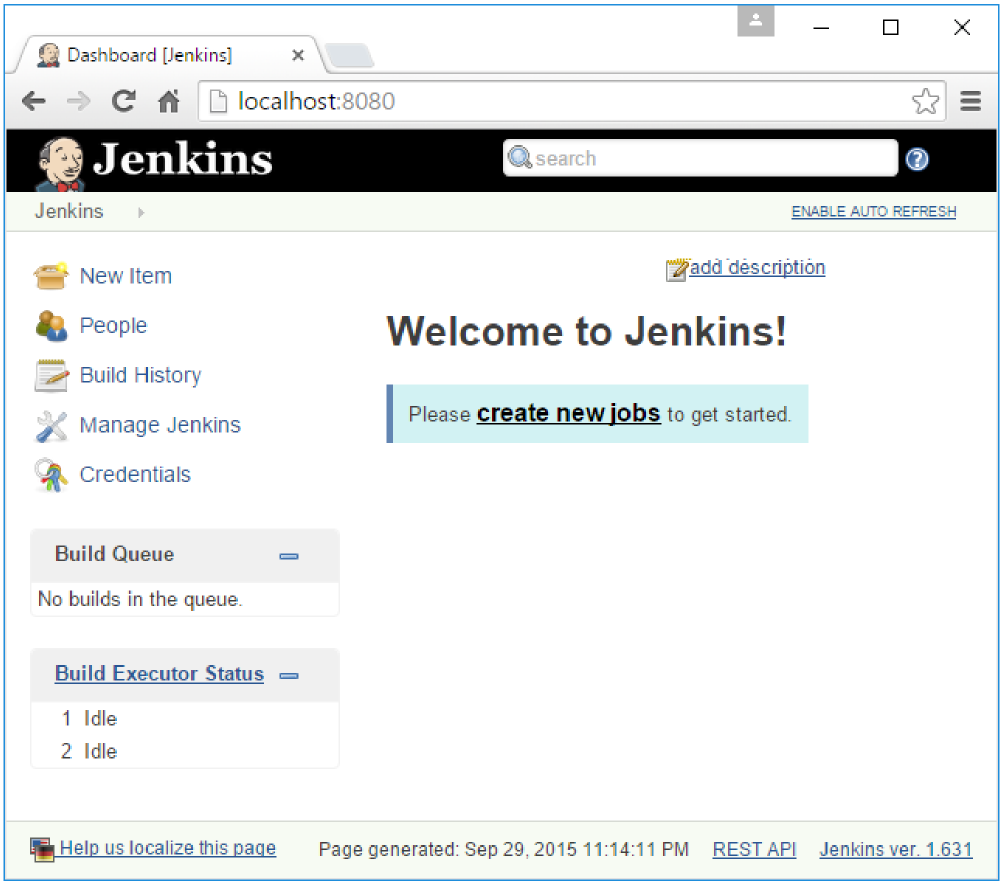
1. Make sure Java is installed on the machine and the `JAVA_HOME` variable is set.
2. Open the command prompt and go to the location where you have downloaded the `jenkins.war` file.

   ```
   cd C:\Users\nikhi\Downloads
   ```

3. Execute the following command to install Jenkins:

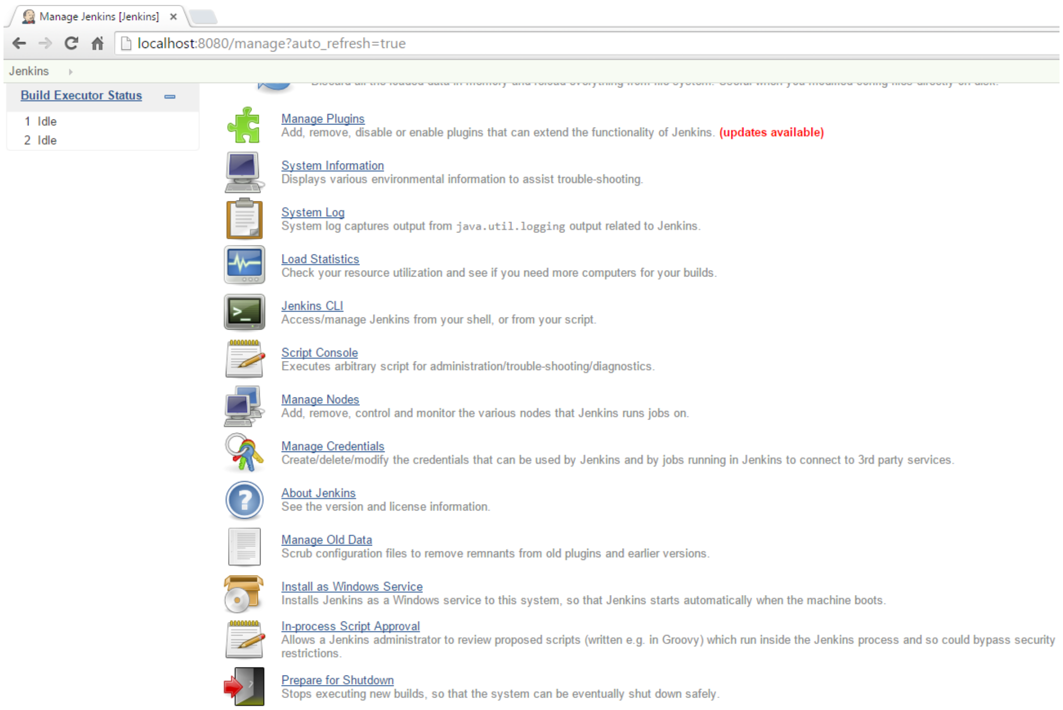
   ```
   java -jar jenkins.war
   ```

4. Once Jenkins is installed successfully, access it using the link `http://localhost:8080`.

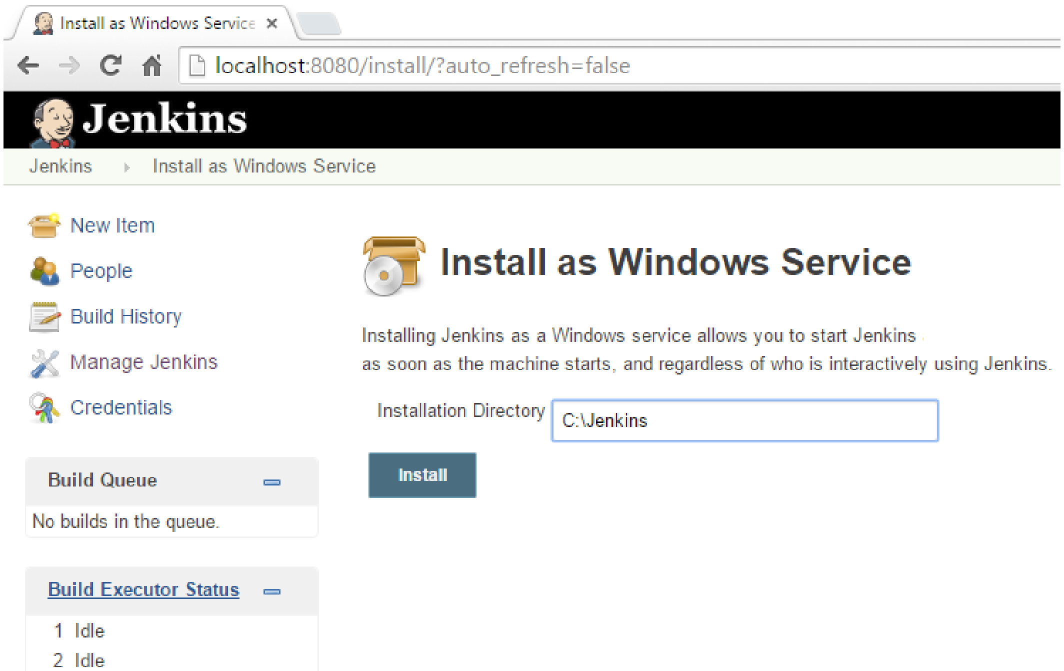


5. From the Jenkins Dashboard, click on the **Manage Jenkins** link on the left-hand side of the dashboard.

6. This will take you to the following page where you can administrate Jenkins. Click on the **Install as Windows Service** link.



7. It will ask for the installation directory. Give a location where you want all your Jenkins metadata to be stored. In our case, it's `C:\Jenkins`.



8. That's all. Jenkins is all ready for use.
9. To confirm whether Jenkins is running as a Windows service, open the services window by running the following command `services.msc` from Windows **Run**. Once the services window opens, check for a service named `Jenkins`.

Installing Jenkins using the native Windows package is much easier than installing Jenkins using the `.war` file. However, it's worth mentioning as this method can be automated. The command: `java -jar jenkins.war`, can be wrapped up in a Windows batch script and can be run remotely through `ssh` or `sftp` on all the Windows machines where Jenkins is anticipated.

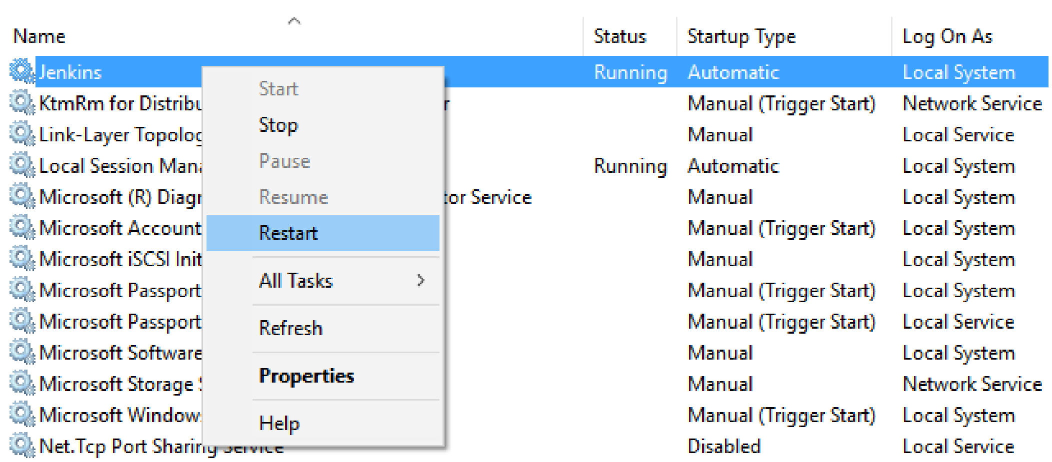
Changing the port where Jenkins runs

By default Jenkins, when installed, runs under port 8080. However, if for some reason you want Jenkins to run on some other port, perform the following steps:

1. Open the `jenkins.xml` file present under the Jenkins installation directory, which is `C:\Program Files (x86)\Jenkins` in our case.
2. Inside the `jenkins.xml` file, go to the following section:

   ```
   <arguments>-Xrs -Xmx256m -Dhudson.lifecycle=hudson.
   lifecycle.WindowsServiceLifecycle -jar "%BASE%\jenkins.war"
   --httpPort=8080</arguments>
   ```

3. The `--httpPort` option is where you can change the port on which Jenkins runs.
4. After making the changes open the `services` window with the command `services.msc` from Windows **Run**.
5. Check for the service named Jenkins. Right-click on it and select **Restart**, as shown in the following screenshot:



Setting up Jenkins on Ubuntu

In order to install Jenkins on Ubuntu, open the terminal. Make sure Java is installed on the machine and the `JAVA_HOME` variable is set.

Installing the latest version of Jenkins

To install the latest version of Jenkins, perform the following steps in sequence:

1. Check for admin privileges; the installation might ask for the admin username and password.
2. Download the latest version of Jenkins using the following command:

```
wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key | 
sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /
etc/apt/sources.list.d/jenkins.list'
```

3. To install Jenkins, issue the following commands:

```
sudo apt-get update
sudo apt-get install jenkins
```

4. Jenkins is now ready for use. By default, the Jenkins service runs on port 8080.
5. To access Jenkins, go to the following link in the web browser, `http://localhost:8080/`.

> The link `https://jenkins-ci.org/debian/jenkins-ci.org.key` mentioned in the first command leads to the Jenkins repository for the latest Jenkins `deb` package.

Installing the latest stable version of Jenkins

If you prefer to install a stable version of Jenkins, then perform the following steps in sequence:

1. Check for admin privileges; the installation might ask for admin username and password.
2. Download the latest version of Jenkins using the following command:

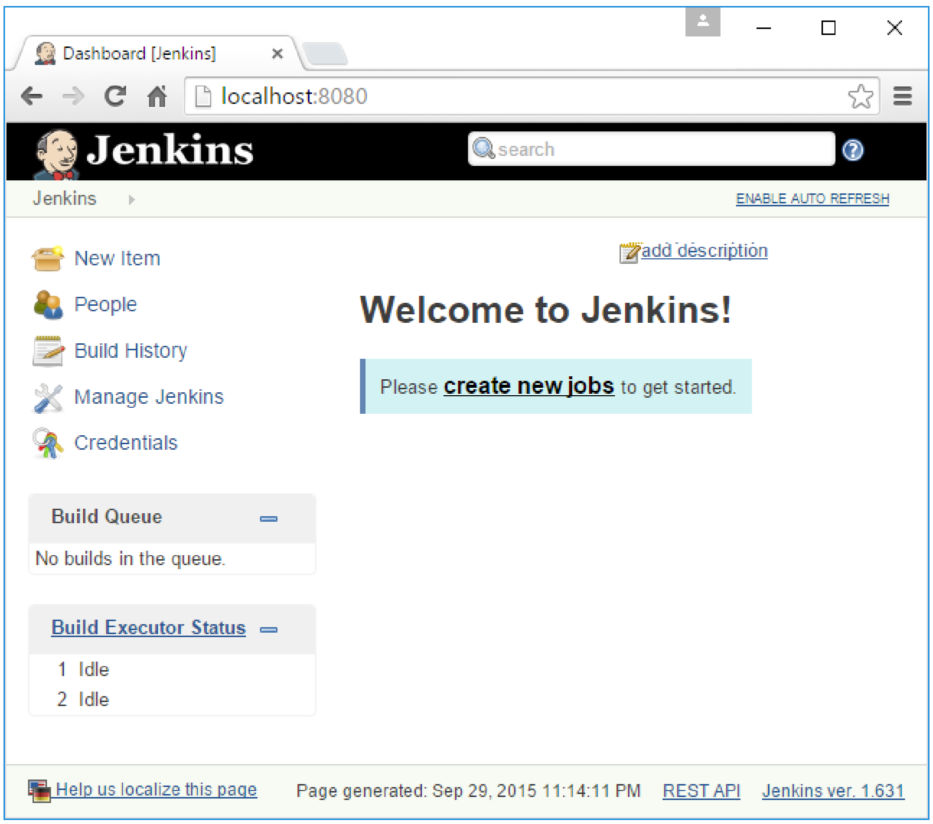
```
wget -q -O - http://jenkins-ci.org/debian-stable/jenkins-ci.org.
key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable
binary/ > /etc/apt/sources.list.d/jenkins.list'
```

3. To install Jenkins, issue the following commands:

```
sudo apt-get update
sudo apt-get install jenkins
```

The link `http://jenkins-ci.org/debian-stable/jenkins-ci.org.key` mentioned in the first command leads to the Jenkins repository for the latest stable Jenkins deb package.

4. Jenkins is now ready for use. By default, the Jenkins service runs on port 8080.
5. To access Jenkins, go to the following link in the web browser: `http://localhost:8080/`.



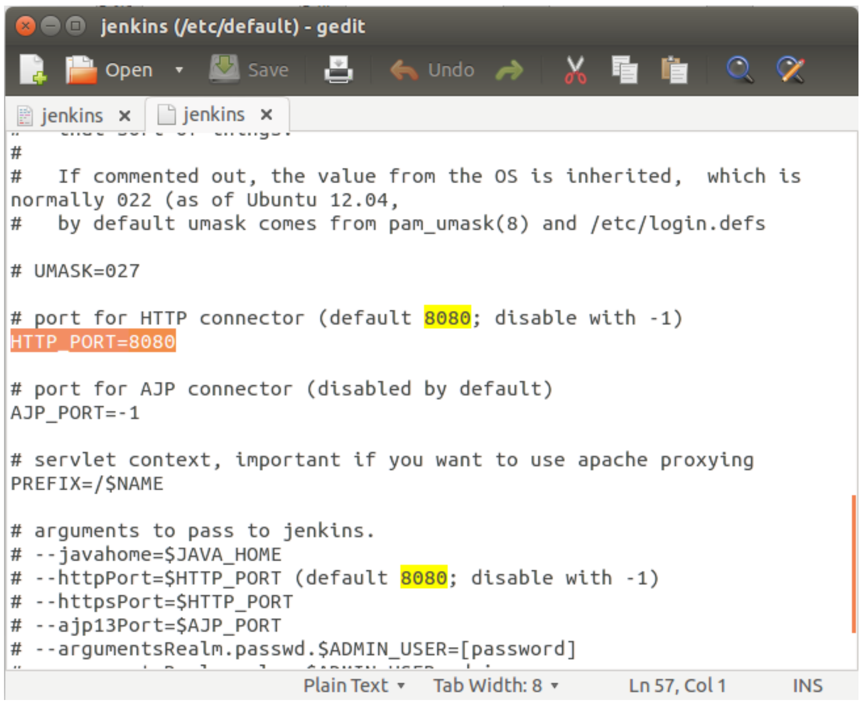
In order to troubleshoot Jenkins, access the logs present at `/var/log/jenkins/jenkins.log`.

The Jenkins service runs with the user `Jenkins`, which automatically gets created upon installation.

Changing the Jenkins port on Ubuntu

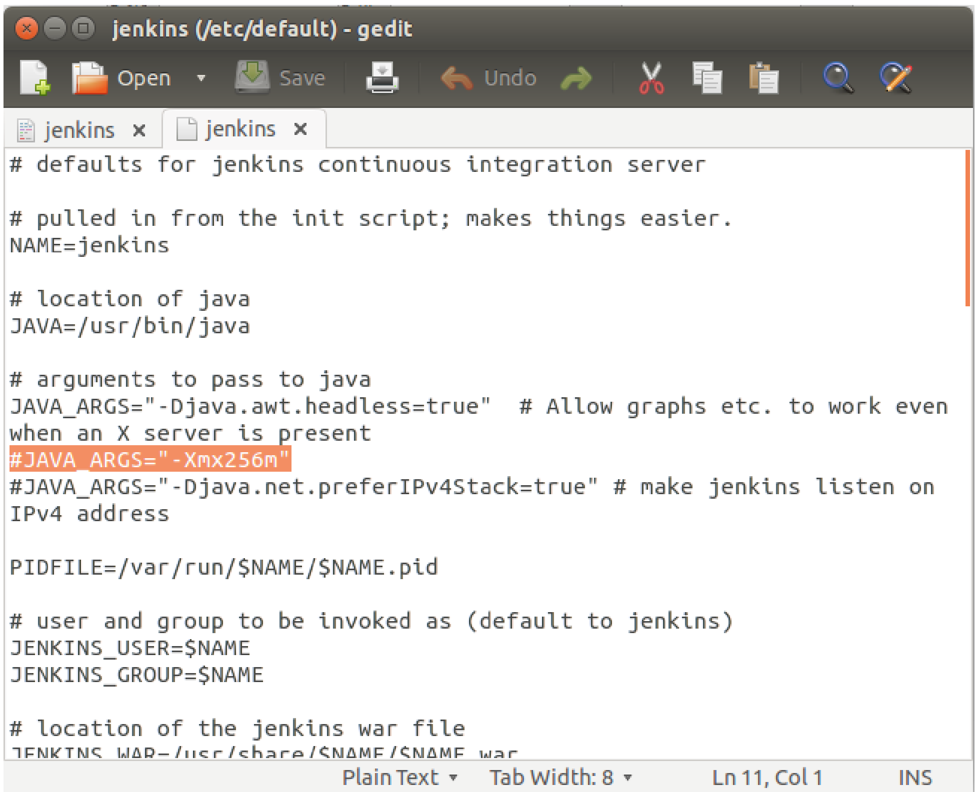
To change the Jenkins port on Ubuntu, perform the following steps:

1. In order to change the Jenkins port, open the `jenkins` file present inside `/etc/default/`.
2. As highlighted in the following screenshot, the `HTTP_PORT` variable stored the port number:



3. Inside the same file, there is another important thing to note, the memory heap size. Heap size is the amount of memory allocated for the Java Virtual Machine to run properly.
4. You can change the heap size by modifying the `JAVA_ARGS` variable as shown in the following example.

5. We can also change the user with which the Jenkins service runs on Ubuntu. In the following screenshot, we can see a variable `NAME` with a value `jenkins`. We can change this to any user we want.



Setting up Jenkins on Fedora

In order to install Jenkins on Fedora, open the Terminal. Make sure Java is installed on the machine and `JAVA_HOME` variable is set.

> Installing Jenkins on Red Hat Linux is similar to installing Jenkins on Fedora.

Installing the latest version of Jenkins

To install the latest version of Jenkins, perform the following steps in sequence:

1. Check for admin privileges; the installation might ask for admin username and password.
2. Download the latest version of Jenkins using the following command
   ```
   sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.
   org/redhat/jenkins.repo
   sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
   ```
3. To install Jenkins, issue the following commands:
   ```
   sudo yum install Jenkins
   ```

> The link `https://pkg.jenkins-ci.org/redhat/jenkins.repo` mentioned in the first command leads to the Jenkins repository for the latest Jenkins rpm package.

Installing the latest stable version of Jenkins

If you prefer to install a stable version of Jenkins, then perform the following step in sequence:

1. Check for admin privileges; the installation might ask for admin username and password.
2. Download the latest version of Jenkins using the following command:
   ```
   sudo wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.
   org/redhat-stable/jenkins.repo
   sudo rpm --import https://jenkins-ci.org/redhat/jenkins-ci.org.key
   ```
3. To install Jenkins issue the following commands:
   ```
   sudo yum install Jenkins
   ```

> The link `http://pkg.jenkins-ci.org/redhat-stable/jenkins.repo` mentioned in the first command leads to the Jenkins repository for the latest stable Jenkins rpm package.

4. Once the Jenkins installation is successful, it will automatically run as a daemon service. By default Jenkins runs on the port 8080.

5. To access Jenkins, go to the following link in the web browser `http://localhost:8080/`.

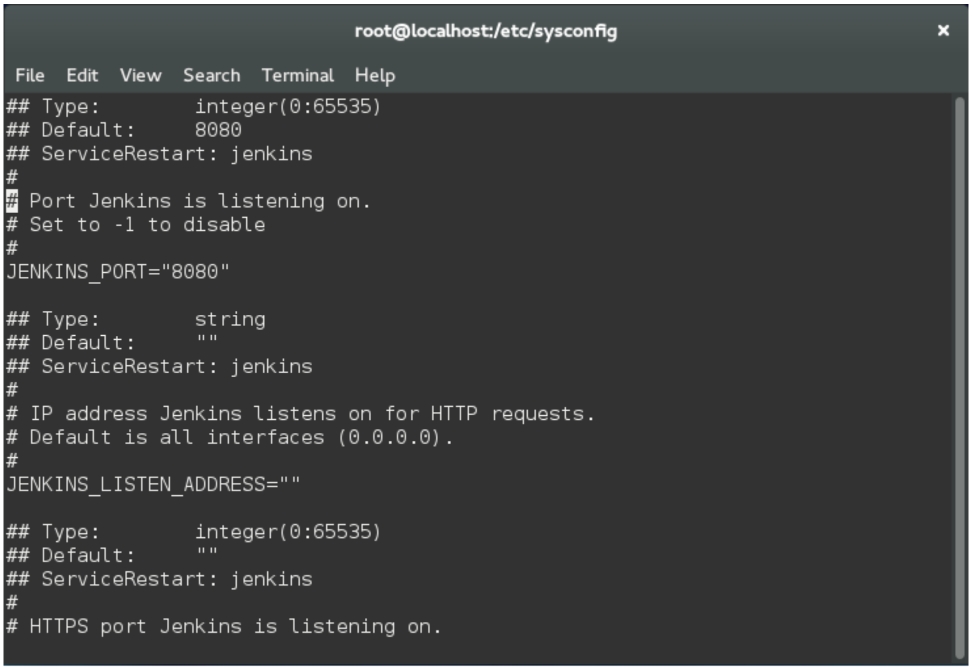
> If for some reason you are unable to access Jenkins, then check the firewall setting. This is because, by default, the firewall will block the ports. To enable them, give the following commands (you might need admin privileges):
>
> ```
> firewall-cmd --zone=public --add-port=8080/tcp –
> permanent
> firewall-cmd --zone=public --add-service=http –permanent
> firewall-cmd --reload
> ```
>
> In order to troubleshoot Jenkins, access the logs present at `var/log/jenkins/jenkins.log`.
>
> The Jenkins service runs with the user `Jenkins` which automatically gets created upon installation.

Changing the Jenkins port on Fedora

To change the Jenkins port on Fedora, perform the following steps:

1. Open the terminal in Fedora.
2. Switch to the admin account using the following command:
   ```
   sudo su –
   ```
3. Enter the password when prompted.
4. Execute the following commands to edit the file named `jenkins` present at `/etc/sysconfig/`:
   ```
   cd /etc/sysconfig/
   vi jenkins
   ```

5. Once the file is open in the terminal, move to the line where you see `JENKINS_PORT="8080"`, as shown in the following screenshot:



Sample use cases

It is always good to learn from others' experiences. The following are the use cases published by some famous organizations that can give us some idea of the hardware specification.

Netflix

In 2012, Netflix had the following configuration:

Hardware configuration:

- 2x quad core x86_64 for the Jenkins master with 26 GB RAM
- 1 Jenkins master with 700 engineers using it
- Elastic slaves with Amazon EC2 + 40 ad-hoc slaves in Netflix's data center

Work load:

- 1,600 Jenkins jobs
- 2,000 Builds per day
- 2 TB of build data

Yahoo!

In 2013, Yahoo! had the following configuration:

Hardware configuration:

- 2 x Xeon E5645 2.40GHz, 4.80GT QPI (HT enabled, 12 cores, 24 threads) with 96 GB RAM, and 1.2 TB of disk space
- 1 Jenkins master with 1,000 engineers using it
- 48 GB max heap to JVM
- `$JENKINS_HOME*` lives on NetApp
- 20 TB filer volume to store Jenkins job and build data
- 50 Jenkins slaves in three data centers

Workload:

- 13,000 Jenkins jobs
- 8,000 builds per day

> `$JENKINS_HOME` is the environment variable that stores the Jenkins home path. This is where all the Jenkins metadata, logs, and build data gets stored.

Summary

In this chapter, we saw the various constituents that make up Jenkins and its hardware specifications. We also saw how Jenkins can be installed as a service inside a container such as the Apache Tomcat server, along with its advantages. We discussed this example because most of the real world Jenkins servers run solely on the Apache Tomcat server. We also saw Jenkins installation on Windows, Ubuntu, and Fedora as a standalone application.

The details about configuring Jenkins were kept to a minimum, as the main objective of the current chapter was to show how diverse Jenkins is when it comes to the installation process and the variety of operating systems that it supports.

3
Configuring Jenkins

The previous chapter was all about installing Jenkins on various platforms. In this chapter, we will see how to perform some basic Jenkins administration. We will also familiarize ourselves with some of the most common Jenkins tasks, like creating jobs, installing plugins, and performing Jenkins system configurations. We will discuss the following:

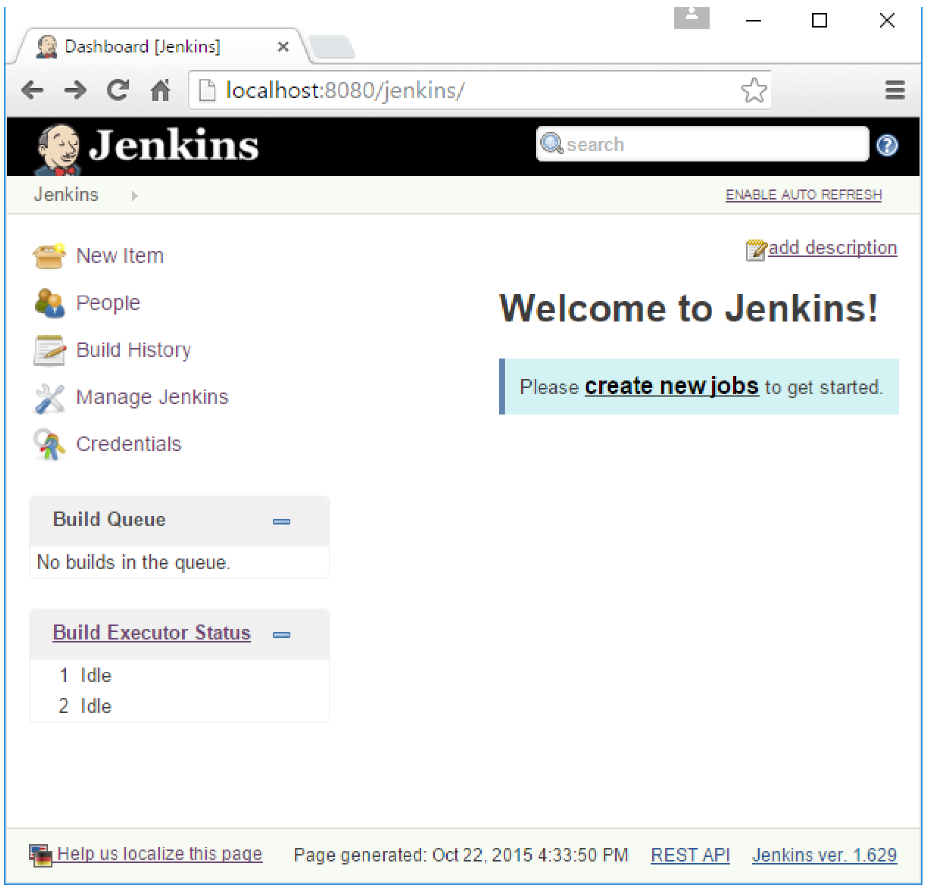
- Creating a simple Jenkins job with an overview of its components
- An overview of the Jenkins home directory
- Jenkins backup and restore
- Upgrading Jenkins
- Managing and configuring plugins
- Managing users and permissions

Every small thing that we discuss in the current chapter will form the foundation for the upcoming chapters, where Jenkins will be configured in many ways to achieve Continuous Integration and Continuous Delivery.

Creating your first Jenkins job

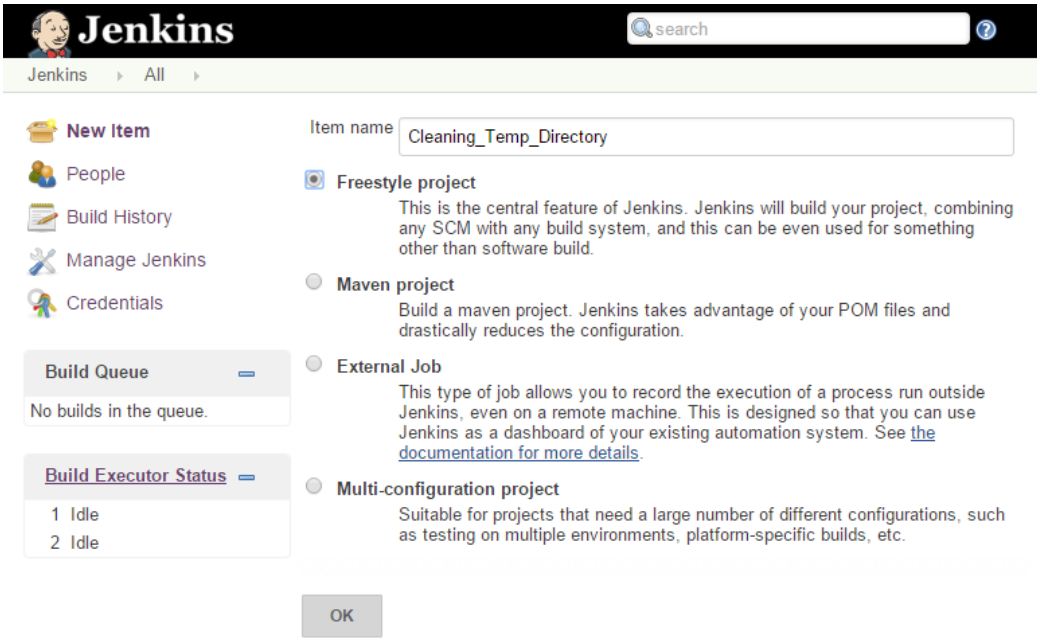
In the current section, we will see how to create a Jenkins Job to clean up the `%temp%` directory on our Windows machine where the Jenkins master server is running. We will also configure it to send an e-mail notification. We will also see how Jenkins incorporates variables (Jenkins system variable and Windows system variable) while performing various tasks. The steps are as follows:

1. From the Jenkins Dashboard, click on the **New Item** link present on the left side. This is the link to create a new Jenkins job.



2. Name your Jenkins job `Cleaning_Temp_Directory` in the **Item name** field.
3. Select the **Freestyle project** option that is present right below the **Item name** field.

4. Click on the **OK** button to create the Jenkins job.

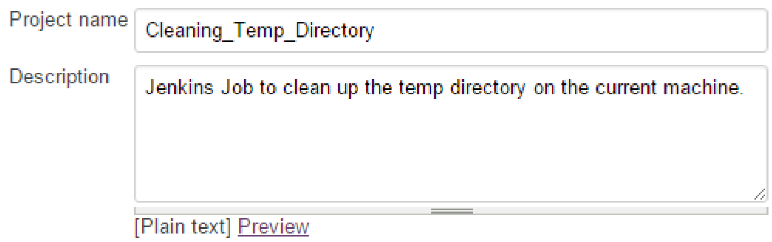


5. You will be automatically redirected to the page where you can configure your Jenkins job.

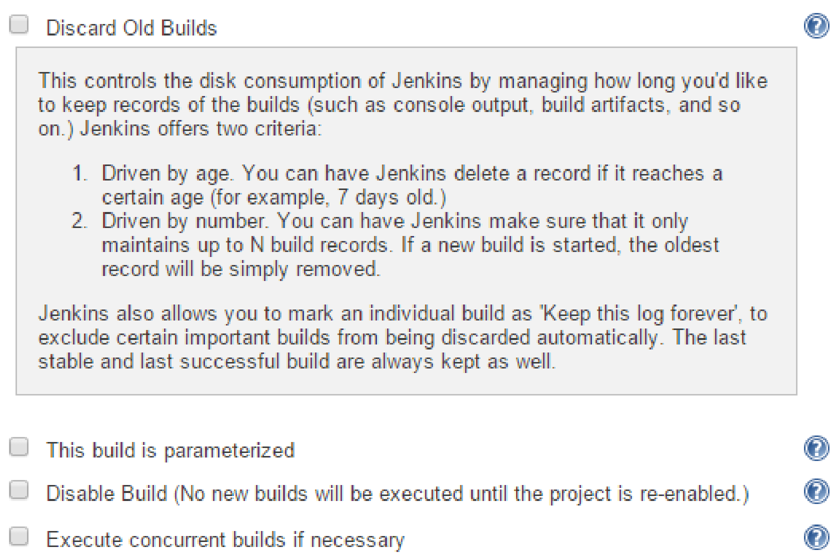
> The Jenkins job name contains underscores between the words. But this is not strictly necessary, as Jenkins has its own way of dealing with blank spaces. However, maintaining a particular naming standard helps in managing and comprehending Jenkins jobs better.
>
> Below the **Item name** field, there are four options to choose from: **Freestyle project**, **Maven project**, **External Job**, and **Multi-configuration project**. These are predefined templates, each having various options that define the functionality and scope of the Jenkins job. All of them are self-explanatory.

6. The **Project name** field contains the name of our newly created Jenkins job.

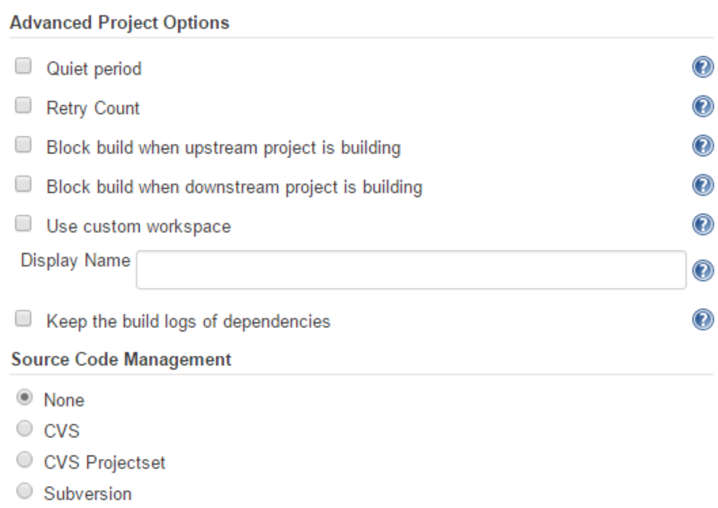
7. Below that, we have the option to add some description about our Jenkins job. I added one for our Jenkins job.



8. Below the **Description** section, there are other options that can be ignored for now. Nevertheless, you can click on the question mark icon, present after each option to know its functionality, as shown in the following screenshot:



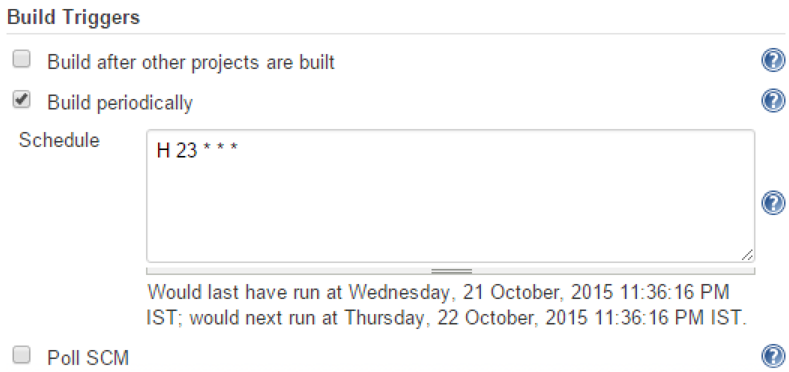
9. Scrolling down further, you will see the **Advanced Project Options** section and the **Source Code Management** section. Skip them for now as we don't need them.



> We will discuss more about the **Advanced Project Options** and the **Source Code Management** section in the upcoming chapters.
>
> Installing plugins will show the number of parameters available under these sections.
>
> For example, installing the Git plugin will bring a new parameter under the **Source Code Management** section that connects Jenkins with Git.

10. On scrolling down further, you will see the **Build Triggers** option.

11. Under the **Build Triggers** section, select the **Build periodically** option and add `H 23 * * *` inside the **Schedule** field. We would like our Jenkins job to run daily around 11:59 PM throughout the year.



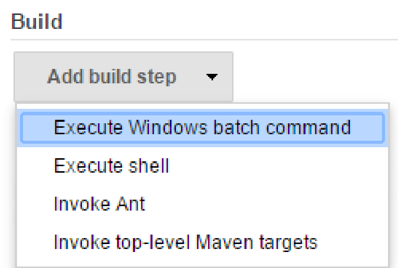
> The schedule format is Minute (0-59) Hour (0-23) Day (1-31) Month (1-12) Weekday (0-7). In the weekday section, 0 & 7 are Sunday.
>
> You might ask the significance of the symbol `H` in place of the minute. Imagine a situation where you have more than 10 Jenkins jobs scheduled for the same time, say `59 23 * * *`. There is a chance Jenkins will get overloaded when all the Jenkins jobs start at once. To avoid this, we use an option `H` in the minute place. By doing so, Jenkins starts each job with a gap of 1 minute.
>
> For example, use the `H` option (`H H * * *`) to avoid a situation where you have multiple projects inside Jenkins, with each project containing jobs that perform nightly-builds scheduled at 00:00 hours (`0 0 * * *`).

12. Moving further down brings you to the most important part of the job's configuration: the **Build** section.

Adding a build step

Build steps are sections inside the Jenkins jobs that contain scripts, which perform the actual task. You can run a Windows batch script or a shell script or any script for that matter. The steps are as follows:

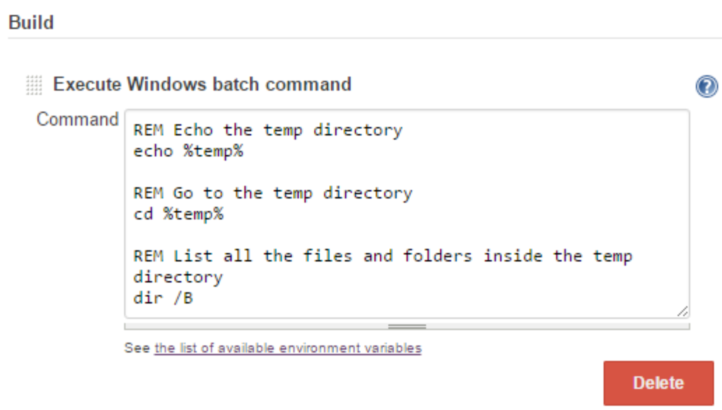
1. Click on the **Add build step** button and select the **Execute Windows batch command** option.



2. In the **Command** field, add the following command. This build step will take us to the `%temp%` directory and will list its contents. The code is as follows:

```
REM Echo the temp directory
echo %temp%

REM Go to the temp directory
cd %temp%

REM List all the files and folders inside the temp directory
dir /B
```

The following screenshot displays the **Command** field in the **Execute Windows batch command** option:

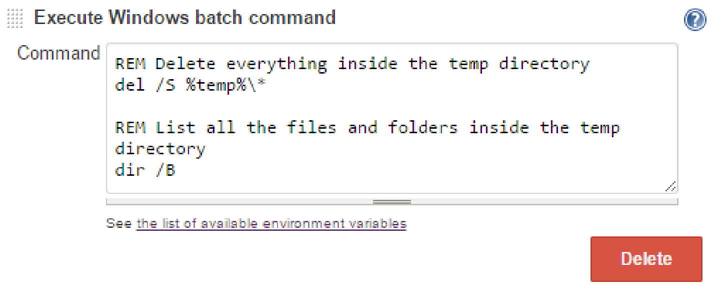


> Instead of giving a complete path to the `temp` directory, I used `%temp%`, which is a system environment variable that stores the path to the `temp` directory. This is one beautiful feature of Jenkins where we can boldly use the system environment variables.

3. You can create as many builds as you want, using the **Add build step** button. Let's create one more build step that deletes everything inside the `%temp%` directory and then lists its content after deletion:

```
REM Delete everything inside the temp directory
del /S %temp%\*

REM List all the files and folders inside the temp directory
dir /B
```

The following screenshot displays the **Command** field in the **Execute Windows batch command** option:



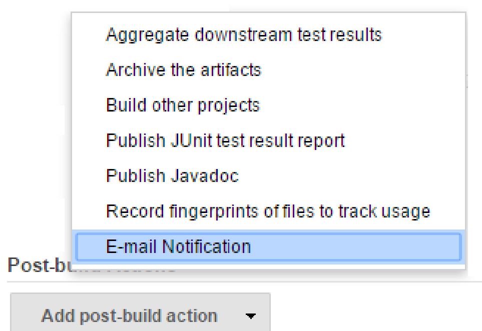
4. That's it. To summarize, the first build takes us to the `%temp%` directory and the second build deletes everything inside it. Both the builds list the content of the temp directory.

Adding post-build actions

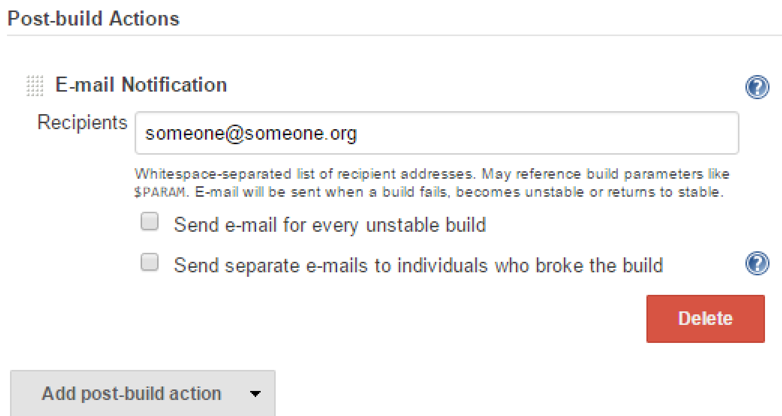
Perform the following steps to add post-build actions:

1. Scroll down further and you'll come across the **Post-build Actions** option.

2. Click on the **Add post-build action** button and select the **E-mail Notification** option from the menu.



3. In the **Recipients** field, add the list of e-mail addresses (team members), separated by a space.



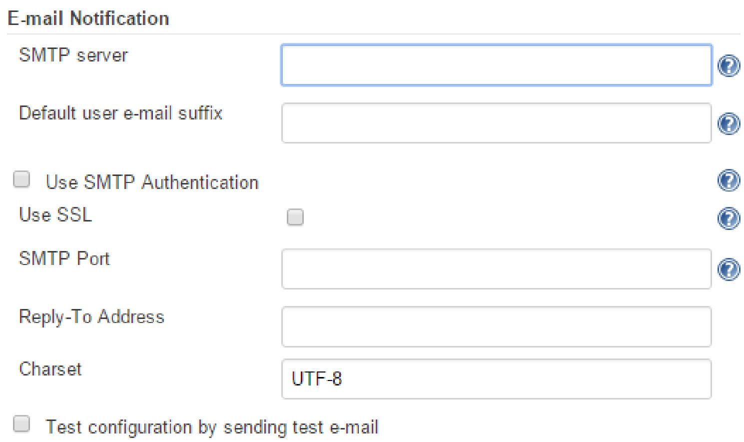
4. There are a few options under the **E-mail Notification** section that can be ignored for now. Nevertheless, you can explore them.
5. Click on the **Save** button, present at the end of the page, to save the preceding configuration. Failing to do so will scrap the whole configuration.

Configuring the Jenkins SMTP server

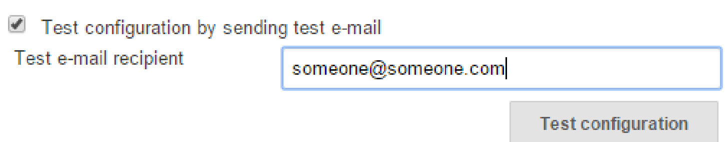
Now that we have created a Jenkins job, let's move on to configure the SMTP server without which the **E-mail Notification** wouldn't work:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. On the **Manage Jenkins** page, click on the **Configure System** link.

3. On the configuration page, scroll down until you see the **E-mail Notification** section.



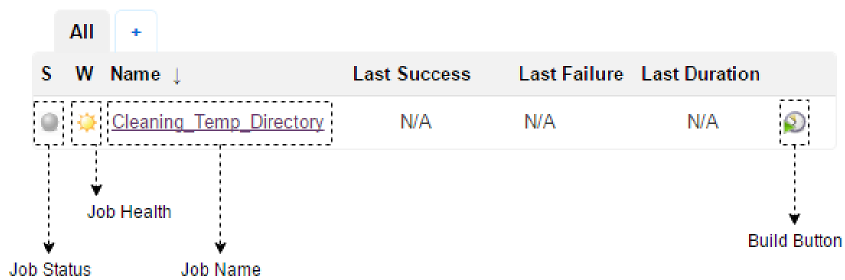
4. Add the **SMTP server** and **SMTP Port** details. Use authentication if applicable. Add an e-mail address in the **Reply-To-Address** field in case you want the recipient to reply to the auto-generated emails.
5. You can test the **E-mail Notification** feature using the **Test configuration by sending test e-mail** option. Add the e-mail address to receive the test e-mail and click on the **Test Configuration** button. If the configuration is correct, the recipient will receive a test e-mail.



Running a Jenkins job

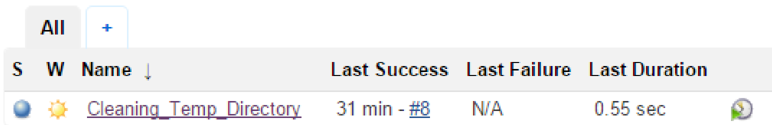
We have successfully created a Jenkins job, now let's run it. The steps are as follows:

1. Go to the Jenkins Dashboard, either by clicking on the Jenkins logo on the top-left corner or by going to the link `http://localhost:8080/jenkins/`.
2. We should see our newly created Jenkins job **Cleaning_Temp_Directory**, listed on the page.



> Although our Jenkins job is scheduled to run at a specific time (anywhere between 23:00 and 23:59), clicking on the **Build** button will run it right away.
>
> The **Job Status** icon represents the status of the most recent build. It can have the following colors that represent various states: *blue for Success*, *red for Failure*, and *gray for Disabled/ Never Executed*.
>
> The **Job Health** icon represents the success rate of a Jenkins job. *Sunny* represents 100 percent success rate, *Cloudy* represents 60 percent success rate, and *Raining* represents 40 percent success rate.

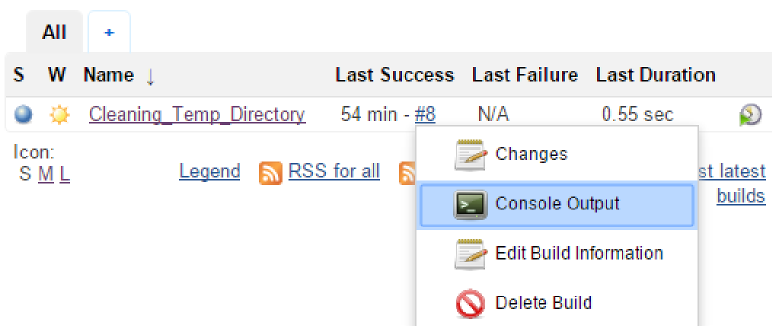
3. Click on the **Build** button to run the job. If everything is right, the job should run successfully.
4. Here's a screenshot of a successful Jenkins job. On my system, the Jenkins job took 0.55 seconds to execute. **#8** represents the build number. It's 8 because I ran the Jenkins job eight times.



Jenkins build log

Now, let's see the build logs:

1. Hover the mouse over the build number (**#8** in our case) and select **Console Output**.



2. The following screenshot is what you will see. It's the complete log of the Windows batch script.

Console Output

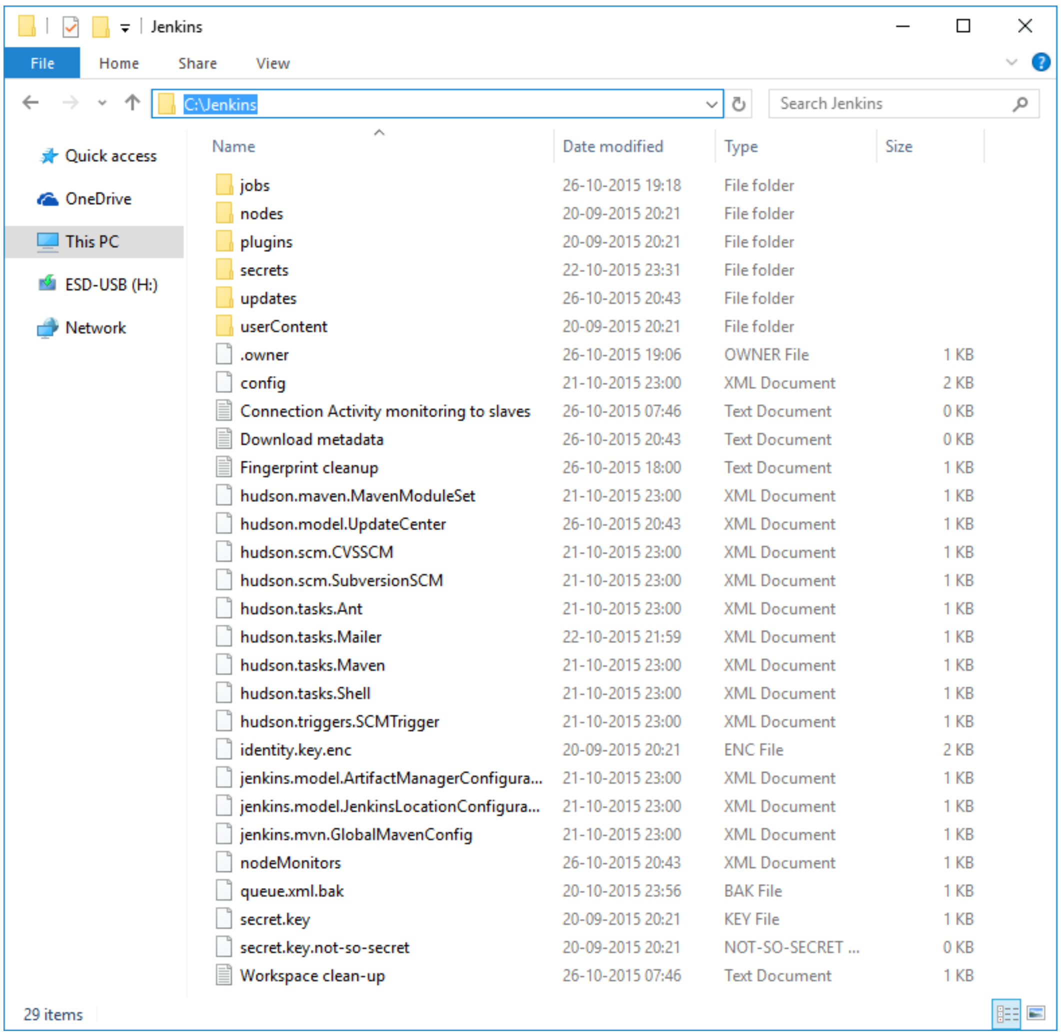
```
Started by user anonymous
Building in workspace C:\Jenkins\jobs\Cleaning_Temp_Directory\workspace
[workspace] $ cmd /c call "C:\Program Files\Apache Software
Foundation\Tomcat 8.0\temp\hudson8071334469743261573.bat"

C:\Jenkins\jobs\Cleaning_Temp_Directory\workspace>REM Echo the temp
directory

C:\Jenkins\jobs\Cleaning_Temp_Directory\workspace>echo C:\WINDOWS\TEMP
C:\WINDOWS\TEMP

C:\Jenkins\jobs\Cleaning_Temp_Directory\workspace>REM Go to the temp
directory

C:\Jenkins\jobs\Cleaning_Temp_Directory\workspace>cd C:\WINDOWS\TEMP

C:\Windows\Temp>REM List all the files and folders inside the temp
directory

C:\Windows\Temp>dir /B
CProgram Files (x86)Opera32.0.1948.69opera_autoupdate.download.lock
CR_4CBB8.tmp
FAB367FF-8277-4D07-9B22-B4996BF16D49-Sigs
hsperfdata_DESKTOP-6NVBTVC$
jetty-0.0.0.0-8080-war--any-
jna--1137314184
Low
Microsoft Visual C++ 2010  x64 Redistributable Setup_10.0.30319
Microsoft Visual Studio Tools for Office Runtime 2010 Setup_10.0.50903
MpCmdRun.log
MPInstrumentation
MpSigStub.log
MPTelemetrySubmit
MRT
opera autoupdate
ScheduledHeartbeat.log
SDIAG_d7e969f8-8db9-47f8-b669-59c628fe4224
```

> The build has run under an `anonymous` group; this is because we have not configured any users yet.

Jenkins home directory

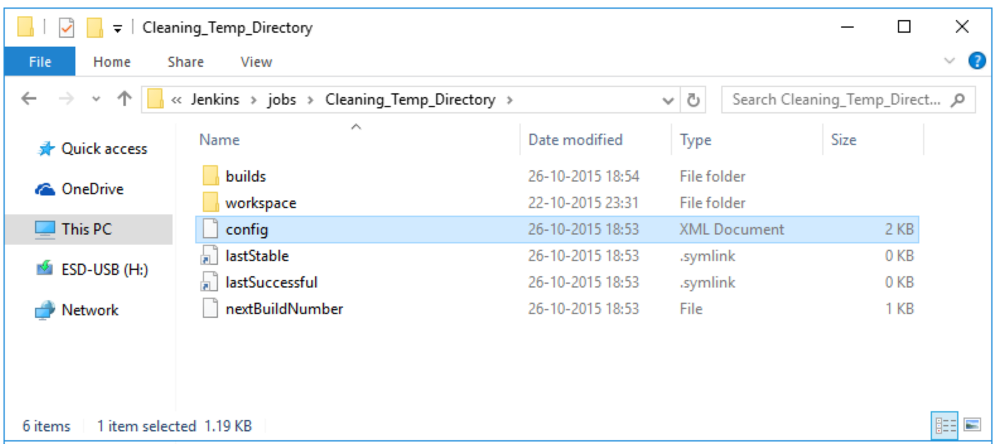
We saw how to create a simple Jenkins job. We also configured the SMTP server details for e-mail notifications. Now, let's see the location where all the data related to the Jenkins jobs gets stored. The steps are as follows:

1. Go to `C:\Jenkins\`, our Jenkins home path. This is the place where all of the Jenkins configurations and metadata is stored, as shown in the following screenshot:

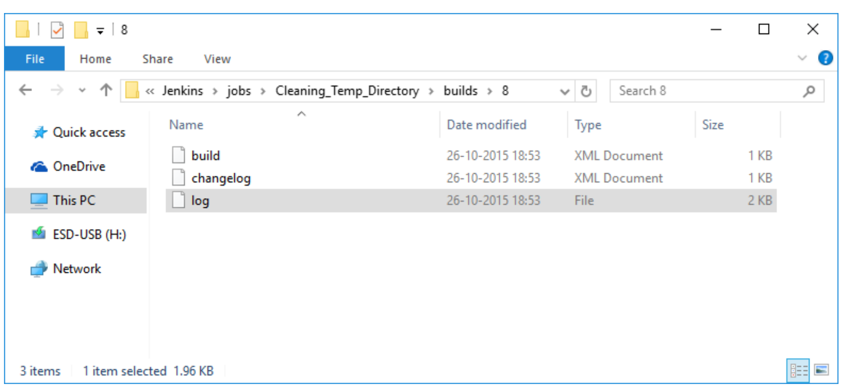


2. Now go to the folder named `jobs\Cleaning_Temp_Directory`. This is the place where all the information related to our Jenkins job is stored.
    - The `config.xml` file is an XML document that contains the Jenkins job configuration. This is something that should be backed up in case you want to restore a Jenkins job.
    - The `workspace` folder contains the output of a build. In our case, it's empty because the Jenkins job does not produce any output file or content.
    - The `builds` folder contains the log information of all the builds that have ran with respect to the respective Jenkins job.

3. This screenshot displays the `config.xml` file, the `workspace` folder, and the `builds` folder:



4. Now, go to the `builds\8` directory, as shown in the next screenshot. The log file shown contains the same logs that we saw on the Jenkins Dashboard.



Jenkins backup and restore

What happens if someone accidentally deletes important Jenkins configurations? Although this can be avoided using stringent user permissions, which we will see in the *User administration* section, nevertheless imagine a situation where the Jenkins server crashes or someone working on the Jenkins configuration wants to restore to a previous stable state of Jenkins. This leaves us with a few questions like, what to back up? When to back up? And how to backup?

From what we have learned so far, the entire Jenkins configuration is stored under the Jenkins home directory, which is `C:\jenkins\` in our case. Everything related to Jenkins jobs like build logs, job configurations, and a workspace gets stored in the `C:\jenkins\jobs` folder.

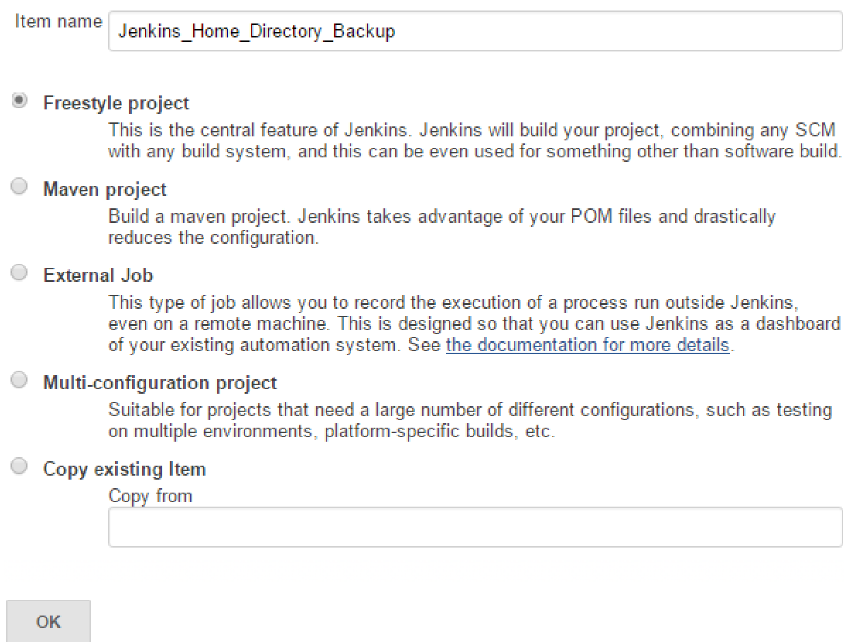
Depending on the requirement, you can choose to backup only the configurations or choose to back up everything. The frequency of Jenkins backup can be anything depending on the project requirement. However, it's always good to back up Jenkins before we perform any configuration changes. Let's understand the Jenkins backup process by creating a Jenkins job.

Creating a Jenkins job to take periodic backup

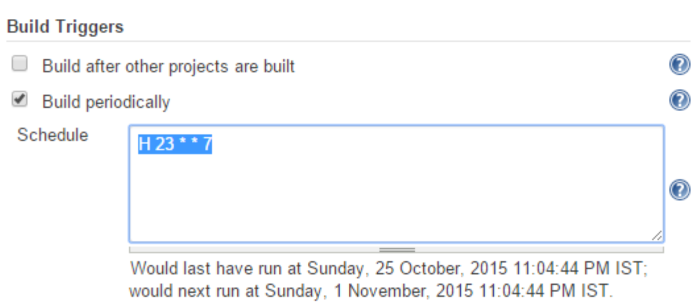
We will create a Jenkins job to take a complete backup of the whole Jenkins home directory. The steps are as follows:

1. You need the 7-Zip package installed on your machine. Download 7-Zip.exe from `http://www.7-zip.org/`.
2. From the Jenkins Dashboard, click on the **New Item** link.

3. Name your new Jenkins job `Jenkins_Home_Directory_Backup`. Select the **Freestyle project** option and click on **OK**.

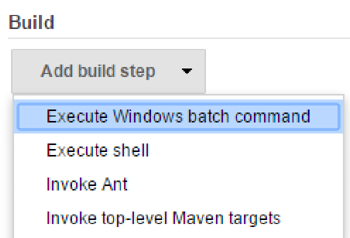


4. On the configuration page, add some description say, `Periodic Jenkins Home directory backup`.
5. Scroll down to the **Build Triggers** section and select the **Build periodically** option.
6. Add `H 23 * * 7` in the **Schedule** section.



> We want our Jenkins backup to take place every Sunday somewhere between 23:00 to 23:49 hours. You can opt for a daily backup, or you can simply run the Jenkins job whenever you want to take a backup.

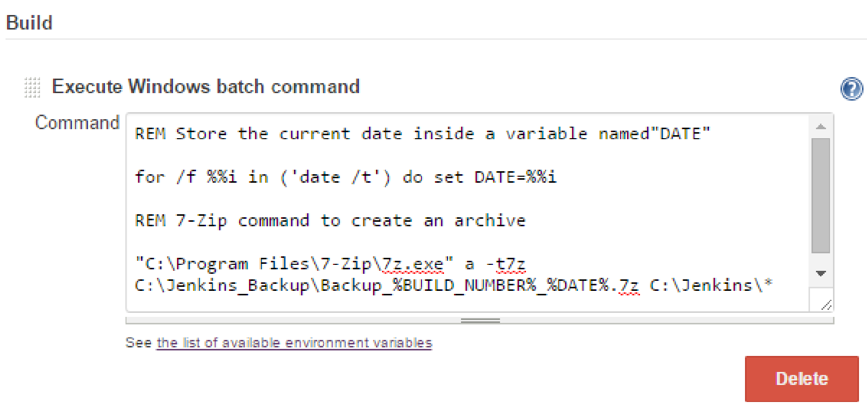
7. Scroll down to the **Build** section. Create a new build by selecting **Execute Windows batch command** from **Add build step**.



8. Add the following content inside the **Command** section:

```
REM Store the current date inside a variable named "DATE"

for /f %%i in ('date /t') do set DATE=%%i

REM 7-Zip command to create an archive

"C:\Program Files\7-Zip\7z.exe" a -t7z C:\Jenkins_Backup\
Backup_%DATE%.7z C:\Jenkins\*
```

9. The following screenshot displays the **Command** field in the **Execute Windows batch command** option:



The following line of code stores the output of date `/t` in a variable `DATE`.

```
for /f %%i in ('date /t') do set DATE=%%i
```

The following command is responsible for creating an archive of the complete Jenkins home directory. It's a single-line command:

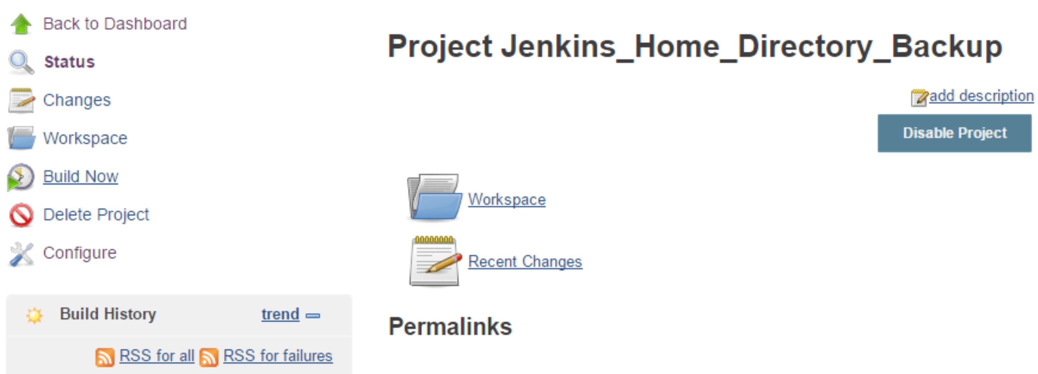
```
"C:\Program Files\7-Zip\7z.exe" a -t7z C:\Jenkins_Backup\
Backup_%BUILD_NUMBER%_%DATE%.7z C:\Jenkins\*
```

Here:

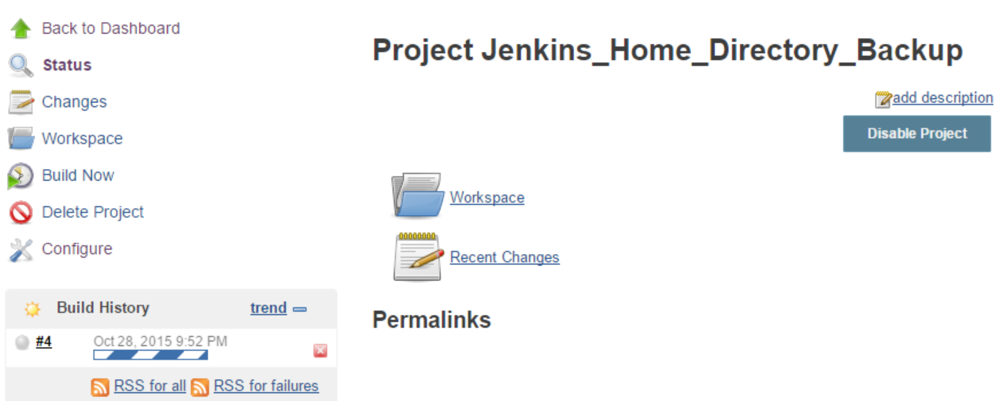
- `C:\Program Files\7-Zip\7z.exe` is the path to the 7-Zip executable
- `a` is a parameter that we pass to the `7z.exe` API, asking it to create an archive
- `-t7z` is the archive format that we have chosen
- `C:\Jenkins_Backup\Backup_%BUILD_NUMBER%_%DATE%.7z` is the backup location
- We have opted to create an archive named `Backup_<build number>_<date>.7z` inside the `C:\Jenkins_Backup\` directory
- Finally, `C:\Jenkins\*` represents the content that we want to archive

> The `date /t` command is a Windows DOS command to get the current date.

10. After adding the code inside the **Command** section, scroll to the end of the page and click on the **Save** button.
11. You will be taken to the jobs homepage, as shown in the following screenshot:

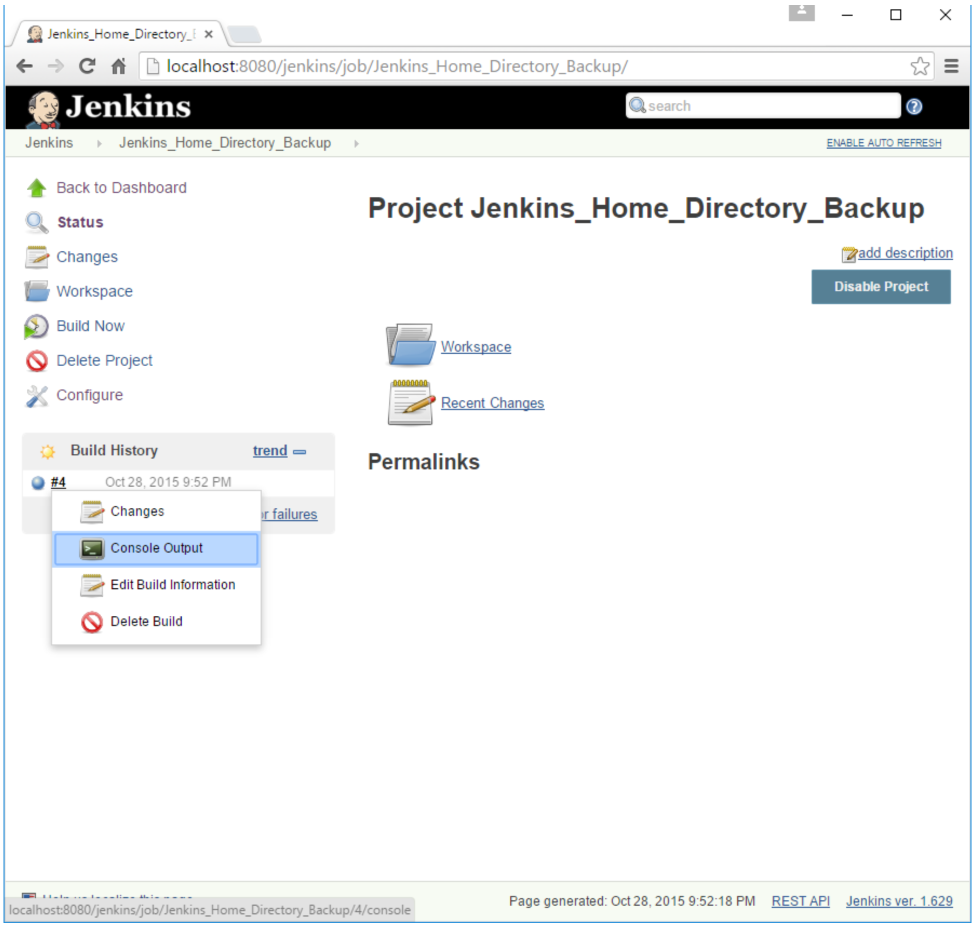


12. Click on the **Build Now** link to run the Jenkins job. Although it's scheduled to run every day around 23:00 hours, there is no harm in running a backup now.
13. Once you run the build, we can see its progress in the **Build History** section as shown in the following screenshot. Here, we can find all the builds that ran for the respective Jenkins job.

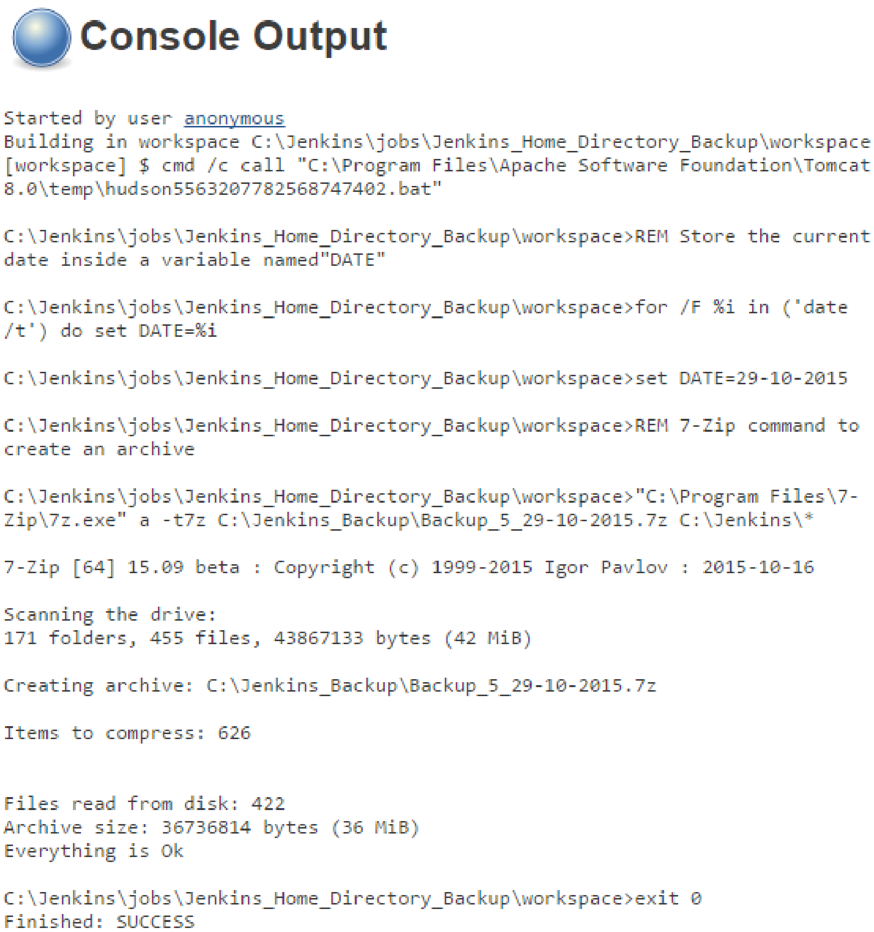


14. The build is successful once the buffering stops and the dot turns blue.
15. Once the build is complete, hover your mouse over the build number to get the menu items, as shown in the following screenshot.

16. Select the **Console Output** option. This will take you to the log page.



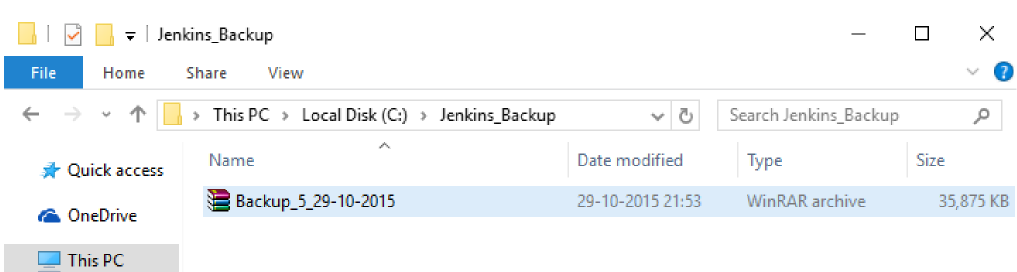
17. Here's the complete log of the Jenkins job:



Console Output

```
Started by user anonymous
Building in workspace C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace
[workspace] $ cmd /c call "C:\Program Files\Apache Software Foundation\Tomcat
8.0\temp\hudson5563207782568747402.bat"

C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace>REM Store the current
date inside a variable named"DATE"

C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace>for /F %i in ('date
/t') do set DATE=%i

C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace>set DATE=29-10-2015

C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace>REM 7-Zip command to
create an archive

C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace>"C:\Program Files\7-
Zip\7z.exe" a -t7z C:\Jenkins_Backup\Backup_5_29-10-2015.7z C:\Jenkins\*

7-Zip [64] 15.09 beta : Copyright (c) 1999-2015 Igor Pavlov : 2015-10-16

Scanning the drive:
171 folders, 455 files, 43867133 bytes (42 MiB)

Creating archive: C:\Jenkins_Backup\Backup_5_29-10-2015.7z

Items to compress: 626


Files read from disk: 422
Archive size: 36736814 bytes (36 MiB)
Everything is Ok

C:\Jenkins\jobs\Jenkins_Home_Directory_Backup\workspace>exit 0
Finished: SUCCESS
```

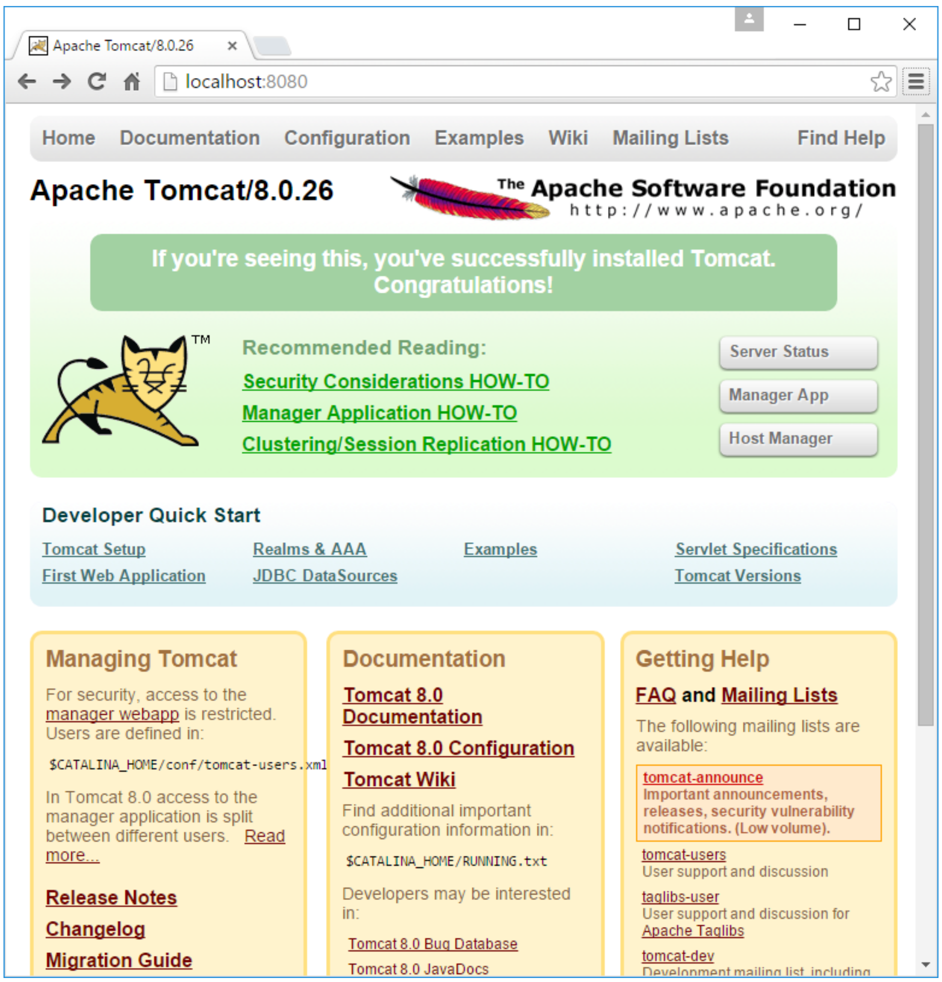
18. From Windows Explorer, go to the `C:\Jenkins_Backup` directory. We can see that the backup archive has been created.



Restoring a Jenkins backup

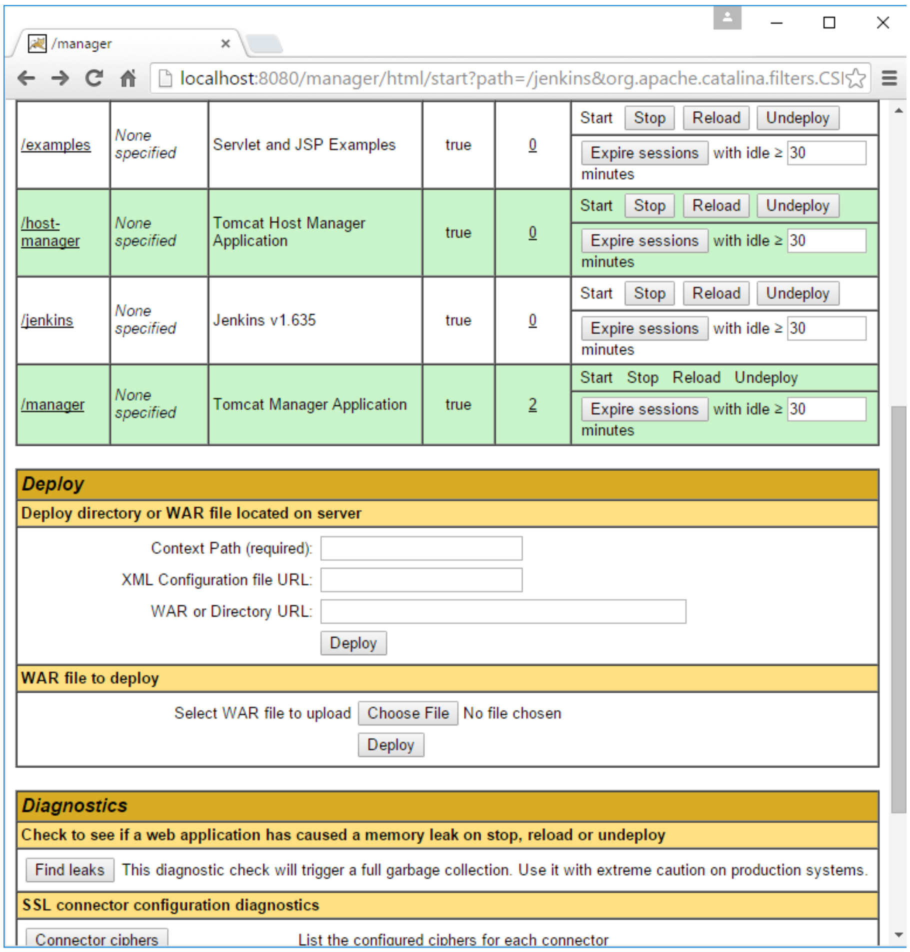
Restoring a Jenkins backup is simple:

1. First, stop the Jenkins service running on the Apache Tomcat server.
2. To do this, go to the admin console at `http://localhost:8080/`.
3. Here's the Apache Tomcat server admin console:



4. From the admin console, click on the **Manager App** button.
5. You will be taken to the **Tomcat Web Application Manager** page.

6. Scroll down and under the **Applications** table, you should see the Jenkins service running along with the version number, as shown in the following screenshot:



7. Click on the **Stop** button to stop the running Jenkins instance. Once it has stopped, the Jenkins Dashboard will be inaccessible.
8. Then, simply unzip the desired backup archive into the Jenkins home directory, which is `C:\Jenkins\` in our case.
9. Once done, start the Jenkins service from the Apache Tomcat server's **Tomcat Web Application Manager** page by clicking on the **Start** button.

Upgrading Jenkins

Jenkins has weekly releases that contain new features and bug fixes. There are also stable Jenkins releases called **Long Term Support** (**LTS**) releases. However, it's recommended that you always choose an LTS release for your Jenkins master server.

In this section, we will see how to upgrade Jenkins master server that is installed inside a container like Apache Tomcat and also a Jenkins standalone master server.

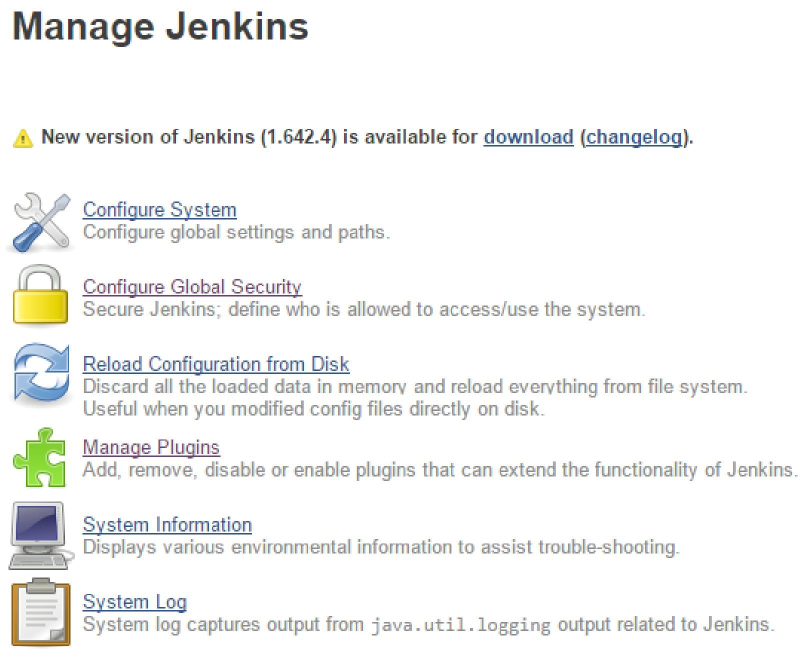
> It is recommended not to update Jenkins until and unless you need to. For example, upgrade Jenkins to an LTS release that contains a bug fix that you need desperately.

Upgrading Jenkins running on the Tomcat server

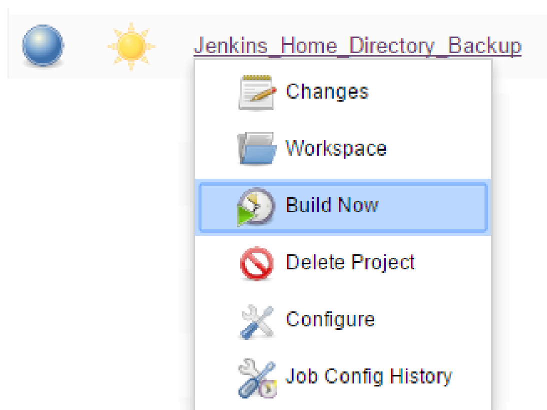
The following are the steps to upgrade Jenkins running on the Tomcat server:

1. Download the latest `jenkins.war` file from `https://jenkins.io/download/`.

2. You can also download Jenkins from the **Manage Jenkins** page, which automatically list the most recent Jenkins release. However, this is not recommended.

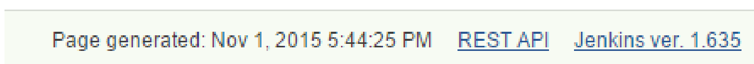


3. From the Jenkins Dashboard, right-click on the Jenkins job **Jenkins_Home_Directory_Backup** and select **Build Now**.



> Always run a backup of Jenkins before upgrading Jenkins.
>
> This is important because, there should be some mechanism to rollback the Jenkins master setup, just in case if Jenkins fails to upgrade or the newer version proves to be unstable.

4. Our Jenkins server is running on Apache Tomcat server. Therefore, go to the location where the current `jenkins.war` file is running. In our case, it's `C:\Program Files\Apache Software Foundation\Tomcat 8.0\webapps`.
5. Stop the Jenkins service from the Apache Tomcat server admin console.
6. Now, replace the current `jenkins.war` file inside the `webapps` directory with the new `jenkins.war` file that you have downloaded.
7. Start the Jenkins service from the Apache Tomcat server's **Tomcat Web Application Manager** page.
8. Go to the Jenkins Dashboard using the link `http://localhost:8080/jenkins`.
9. Check the Jenkins version on the Jenkins Dashboard.

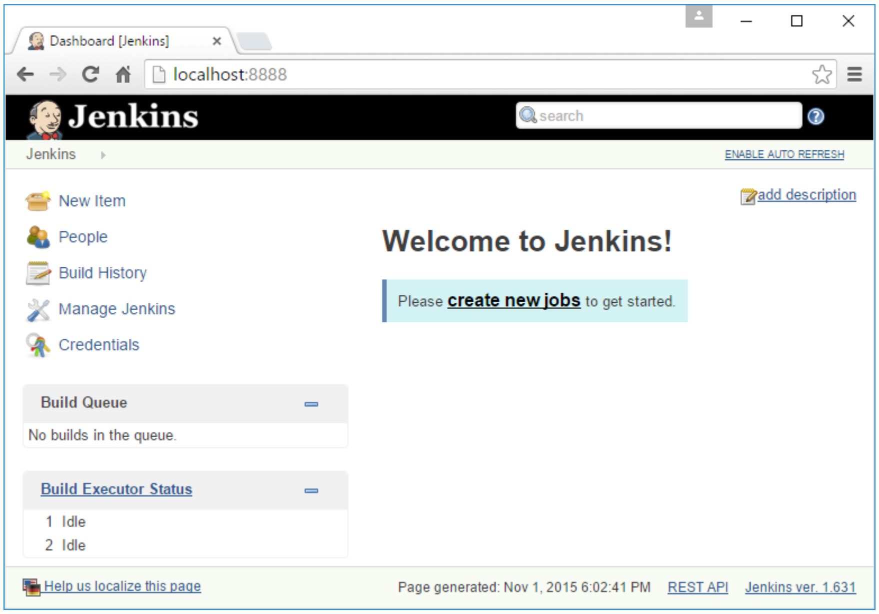


> All the Jenkins settings, configurations, and jobs are intact.

Upgrading standalone Jenkins master on Windows

The upgrade steps mentioned in this section are for a Jenkins master running as a Windows service on the Windows operating system.

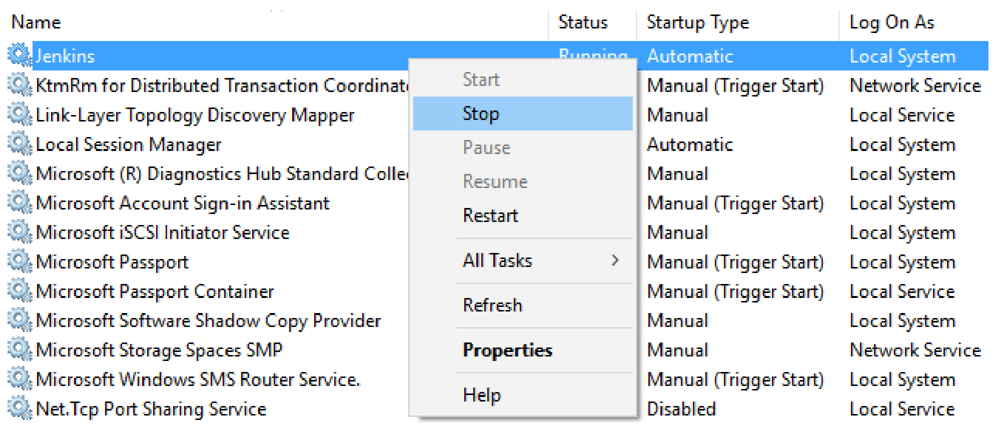
I have a Jenkins instance running as a Windows service on port 8888. The Jenkins version is 1.631, as shown in the following screenshot:



Follow these steps to upgrade Jenkins:

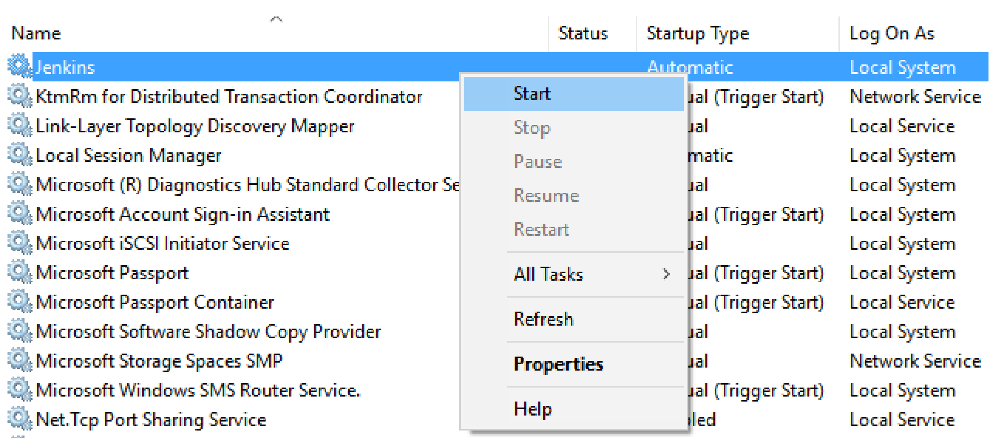
1. Download the latest `jenkins.war` file from the Jenkins website.
2. As mentioned earlier, run a backup of Jenkins before we upgrade it to a newer version.
3. Go to the location where Jenkins is installed on your machine. It should be located at `C:\Program Files (x86)\Jenkins`.
4. Inside the location `C:\Program Files (x86)\Jenkins`, you will see the `jenkins.war` file. We simply need to replace it with the newly downloaded `jenkins.war` file.
5. Before doing so, we need to stop the Jenkins service. To do this, type `services.msc` from Windows Run. This will open the Windows **Services** page.

6. Stop the Jenkins service as shown in the following screenshot. Keep the **Services** window open as you may have to come back here to start the Jenkins service.



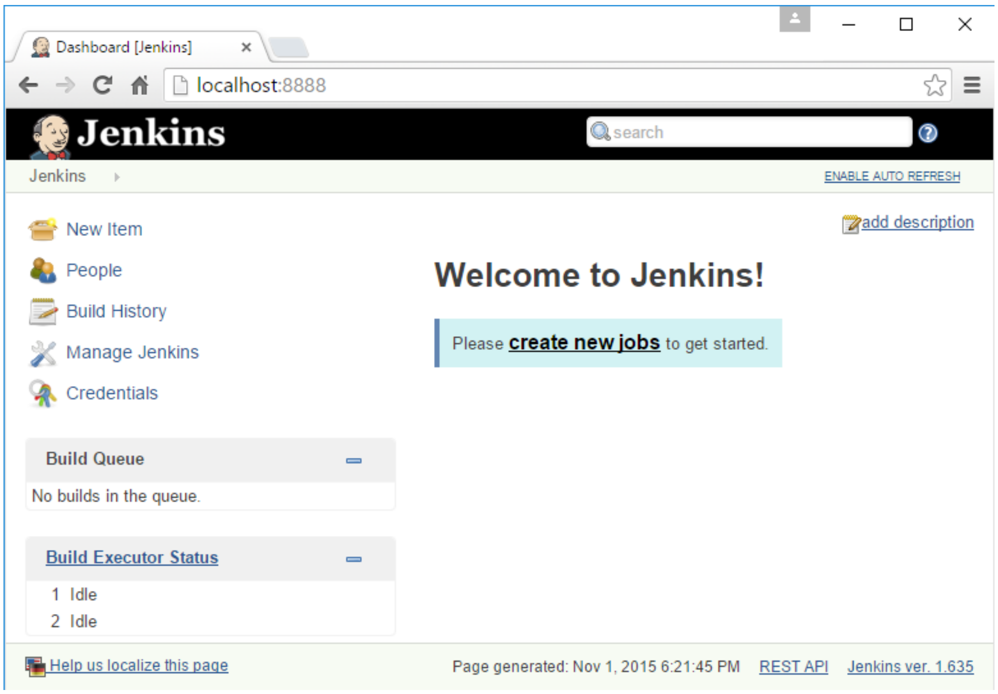
7. Once the Jenkins service is stopped, replace the `jenkins.war` file present under `C:\Program Files (x86)\Jenkins\` with the new version of the `jenkins.war` file.

8. After replacing the file start the Jenkins service from the `services` window, you will get the following screen:



9. Access the Jenkins console using the link `http://localhost:8080/jenkins` to see the changes.

10. As you can see now, our new Jenkins has been upgraded to Version 1.635:



Upgrading standalone Jenkins master running on Ubuntu

Upgrading Jenkins on Ubuntu is simple. Make sure Java is installed on the machine and the `JAVA_HOME` variable is set.

Upgrading to the latest version of Jenkins

To install the latest version of Jenkins, perform the following steps in sequence. However, this is not recommended.

1. Check for admin privileges; the installation might ask for the admin username and password.
2. Backup Jenkins before the upgrade.

3. Execute the following commands to update Jenkins to the latest version available:

```
wget -q -O - https://jenkins-ci.org/debian/jenkins-ci.org.key |
sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian binary/ > /
etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

Upgrading to the latest stable version of Jenkins

If you prefer to upgrade to a new stable version of Jenkins, then perform the following steps in sequence:

1. Check for admin privileges; the installation might ask for admin username and password.
2. Backup Jenkins before the upgrade.
3. Execute the following commands to update Jenkins to the latest stable version available:

```
wget -q -O - https://jenkins-ci.org/debian-stable/jenkins-ci.org.
key | sudo apt-key add -
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable
binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins
```

Upgrading Jenkins to a specific stable version

If you prefer to upgrade to a specific stable version of Jenkins, then perform the following steps in sequence. In the following steps, let's assume I want to update Jenkins to v1.580.3:

1. Check for admin privileges; the installation might ask for the admin username and password.
2. Backup Jenkins before the upgrade.
3. Execute the following commands to update Jenkins to the latest stable version available:

```
wget -q -O - https://jenkins-ci.org/debian-stable/jenkins-ci.org.
key | sudo apt-key add -
```

```
sudo sh -c 'echo deb http://pkg.jenkins-ci.org/debian-stable
binary/ > /etc/apt/sources.list.d/jenkins.list'
sudo apt-get update
sudo apt-get install jenkins=1.580.3
```

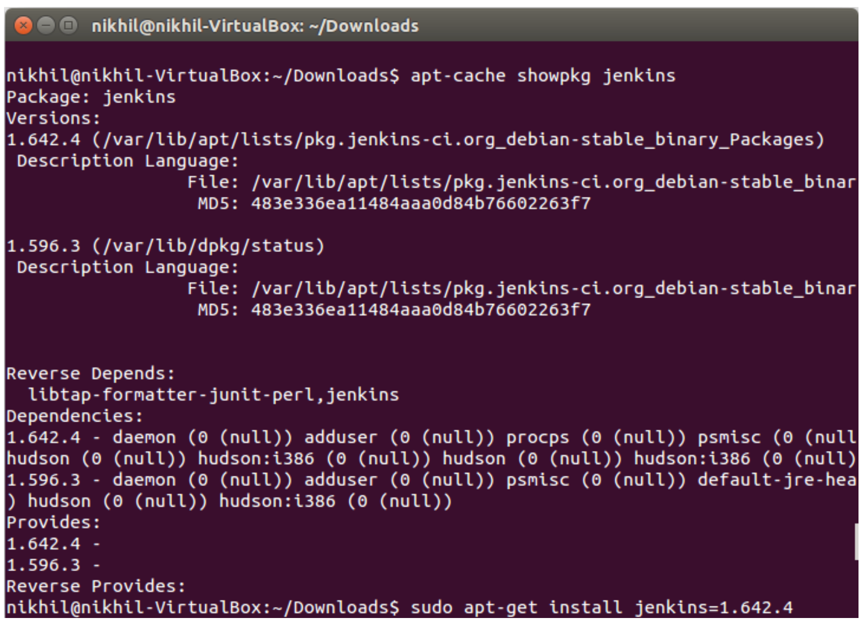
4. You might end up with the following error:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Version '1.580.3' for 'jenkins' was not found
```

5. In that case, run the following command to check the list of available versions:

```
apt-cache showpkg jenkins
```

6. This will give the following output:



7. Notice the Jenkins version suggested; it's 1.642.4 and 1.596.3.
8. If you are ok with any of the available versions, select them and re-run the following command:

```
sudo apt-get install jenkins=1.596.3
```

9. You might get the following error:

```
nikhil@nikhil-VirtualBox: ~/Downloads
nikhil@nikhil-VirtualBox:~/Downloads$ sudo apt-get install jenkins=1.596.3
Reading package lists... Done
Building dependency tree
Reading state information... Done
jenkins is already the newest version.
You might want to run 'apt-get -f install' to correct these:
The following packages have unmet dependencies:
 jenkins : Depends: daemon but it is not going to be installed
E: Unmet dependencies. Try 'apt-get -f install' with no packages (or specify a s
nikhil@nikhil-VirtualBox:~/Downloads$ apt-get -f install
```

10. Run the following command:

    `sudo apt-get -f install`

11. This will give the following output:

```
nikhil@nikhil-VirtualBox:~/Downloads$ sudo apt-get -f install
Reading package lists... Done
Building dependency tree
Reading state information... Done
Correcting dependencies... Done
The following extra packages will be installed:
  daemon
The following NEW packages will be installed:
  daemon
0 upgraded, 1 newly installed, 0 to remove and 333 not upgraded.
1 not fully installed or removed.
Need to get 98.2 kB of archives.
After this operation, 287 kB of additional disk space will be used.
Do you want to continue? [Y/n] y
Get:1 http://in.archive.ubuntu.com/ubuntu/ trusty/universe daemon amd64 0.6.4-1 [98.2 kB]
Fetched 98.2 kB in 1s (69.9 kB/s)
Selecting previously unselected package daemon.
(Reading database ... 168557 files and directories currently installed.)
Preparing to unpack .../daemon_0.6.4-1_amd64.deb ...
Unpacking daemon (0.6.4-1) ...
Processing triggers for man-db (2.6.7.1-1ubuntu1) ...
Setting up daemon (0.6.4-1) ...
Setting up jenkins (1.596.3) ...
 * Starting Jenkins Continuous Integration Server jenkins
Processing triggers for ureadahead (0.100.0-16) ...
```

12. Now run the command to install Jenkins again:

    `sudo apt-get install jenkins=1.596.3`

13. This should install Jenkins on your Ubuntu server.

> If the `apt-cache showpkg` Jenkins command does not list the required Jenkins version you desire, you have the following options:
>
> Download the `jenkins.war` (required version) from the following link: `http://mirrors.jenkins-ci.org/war-stable/`.
>
> Stop the Jenkins service using the command `sudo service jenkins stop`.
>
> Replace the `jenkins.war` file present inside `/usr/share/jenkins` with your newly downloaded `Jenkins.war` file.
>
> Start the Jenkins service using the command `sudo service jenkins start`.

Script to upgrade Jenkins on Windows

We can create a Windows batch script or a Perl script or any other script outside Jenkins to upgrade it. The Windows batch script discussed below is capable of updating a standalone Jenkins master running on Windows to the latest version of Jenkins available. The steps are as follows:

1. Download the `curl.exe` application for Windows from `https://curl.haxx.se/download.html`.
2. Open Notepad and paste the following code inside it. Save the file as `Jenkins_Upgrade.bat`.
3. Set the variables `Backup_Dir`, `Jenkins_Home`, `jenkinsURL`, and `curl` accordingly.
4. Also, set the Jenkins web address accordingly:

```
@echo off
REM === pre-declared variables ===
set Backup_Dir="C:\Jenkins_Backup"
set Jenkins_Home="C:\Jenkins"
set jenkinsURL="http://mirrors.jenkins-ci.org/war/latest/jenkins.
war"
set curl="C:\Users\nikhi\Downloads\curl.exe"

Echo === Stopping Current Jenkins Service ===
sc stop Jenkins

Echo === Sleeping to wait for file cleanup ===
```

```
ping -n 4 http://localhost:8080 > NUL

Echo === clean files ===
copy /Y %Jenkins_Home%\jenkins.war %Backup_Dir%\jenkins.war.bak
del /Y %Jenkins_Home%\jenkins.war

Echo === download new files ===
cd %Jenkins_Home%
%curl% -LOk %jenkinsURL%

Echo *** Starting new upgraded Jenkins
sc start Jenkins

Echo *** Sleeping to wait for service startup
ping -n 4 http://localhost:8080 > NUL
```

5. Try running the Windows batch script as an administrator.

> In the preceding Windows batch script, set the `jenkinsURL` variable to point to a stable version of `jenkins.war` using the following link `http://mirrors.jenkins-ci.org/war-stable/latest/`.
>
> To update your Jenkins master server to a particular stable release, set the `jenkinsURL` variable to a stable release version link. For example, to install Jenkins 1.642.4, use the link `http://mirrors.jenkins-ci.org/war-stable/1.642.4/jenkins.war`.
>
> To get the list of stable Jenkins releases, use the link `http://mirrors.jenkins-ci.org/war-stable/`.

Script to upgrade Jenkins on Ubuntu

The shell script discussed in the following steps is capable of updating a standalone Jenkins master running on Ubuntu to the latest version of Jenkins available.

1. Open `gedit` and paste the following code inside it. Save the file as `Jenkins_Upgrade.sh`.
2. Set the variables `Backup_Dir`, `Jenkins_Home`, and `jenkinsURL` accordingly.

3. Also, set the Jenkins web address accordingly:

```
#!/bin/bash

# pre-declared variables

Backup_Dir="/tmp/Jenkins_Backup"
Jenkins_Home="/usr/share/jenkins"
jenkinsURL="http://mirrors.jenkins-ci.org/war/latest/jenkins.war"

# Stopping Current Jenkins Service
sudo service jenkins stop

# Sleeping to wait for file cleanup
ping -q -c5 http://localhost:8080 > /dev/null

# clean files
sudo cp -f $Jenkins_Home/jenkins.war $Backup_Dir/jenkins.war.bak
sudo rm -rf $Jenkins_Home/jenkins.war

# Download new files
cd $Jenkins_Home
sudo wget "$jenkinsURL"

# Starting new upgraded Jenkins
sudo service jenkins start

# Sleeping to wait for service startup
ping -q -c5 http://localhost:8080 > /dev/null
```

4. Try running the shell script with a user having `sudo` access.

> In the preceding shell script, set the `jenkinsURL` variable to point to a stable version of `jenkins.war` using the following link `http://mirrors.jenkins-ci.org/war-stable/latest/`.
>
> To update your Jenkins master server to a particular stable release, just set the `jenkinsURL` variable to a stable release version link. For example, to install Jenkins 1.642.4, use the link `http://mirrors.jenkins-ci.org/war-stable/1.642.4/jenkins.war`.
>
> To get the list of stable Jenkins releases, use the link `http://mirrors.jenkins-ci.org/war-stable/`.

Managing Jenkins plugins

Jenkins derives most of its power from plugins. As discussed in the previous chapter, every plugin that gets installed inside Jenkins manifests itself as a parameter, either inside Jenkins system configurations or inside a Jenkins job. Let's see where and how to install plugins.

In the current section, we will see how to manage plugins using the Jenkins plugins manager. We will also see how to install and configure plugins.

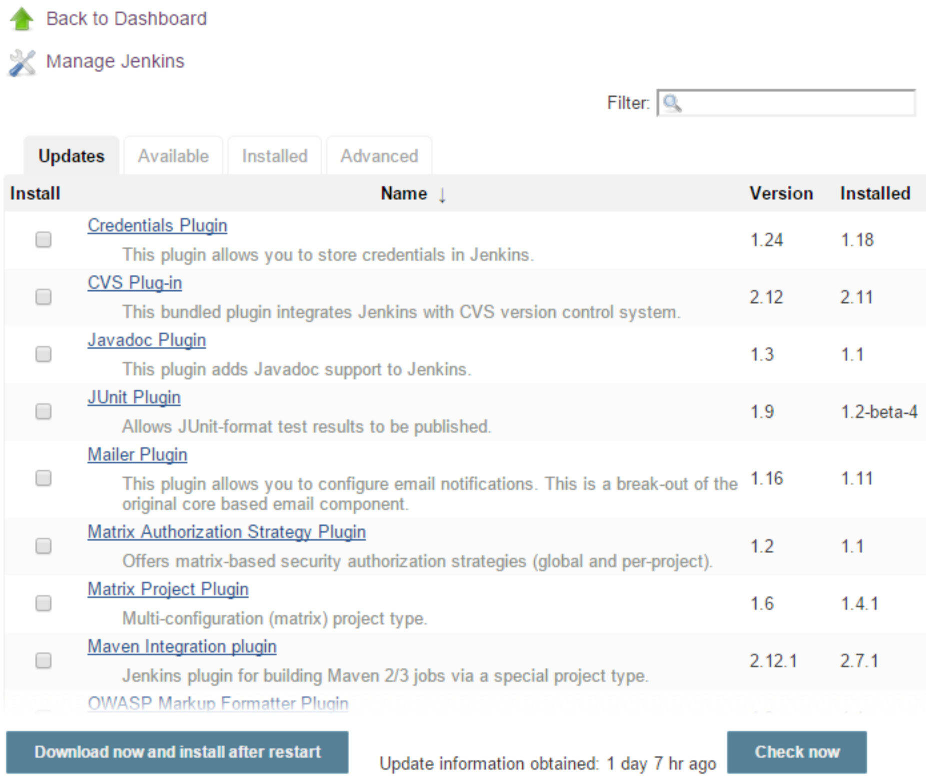
The Jenkins Plugins Manager

The Jenkins **Plugin Manager** section is a place to install, uninstall, and upgrade Jenkins plugins. Let us understand it in detail:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. From the **Manage Jenkins** page, click on the **Manage Plugins** link.

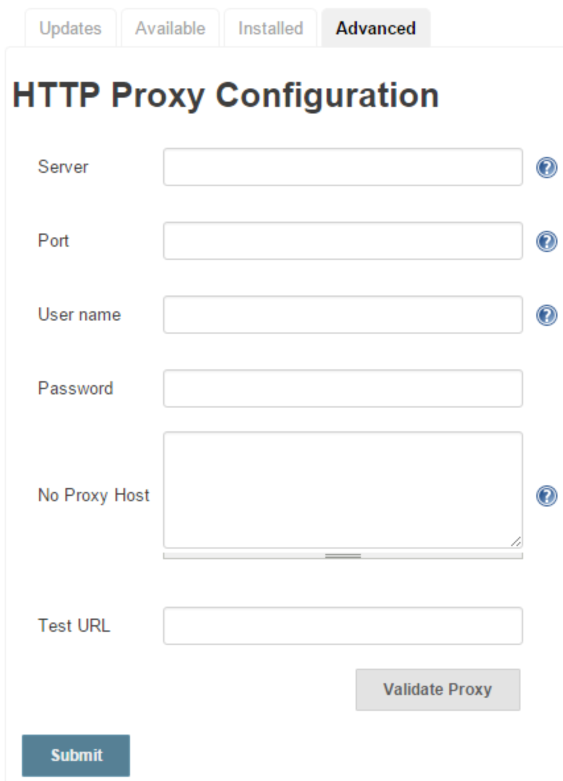
> You can also access the same Jenkins **Plugin Manager** page using the link `http://localhost:8080/jenkins/pluginManager/`.

3. The following screenshot is what you see when you land on the Jenkins **Plugin Manager** page.



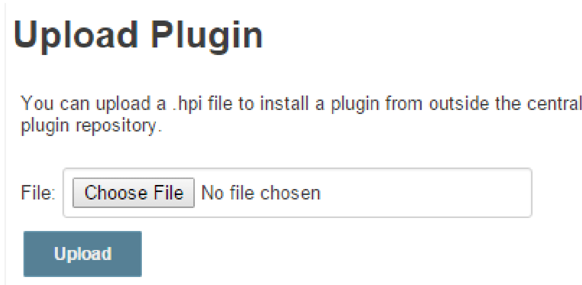
4. The following four tabs are displayed in the screenshot:
    - The **Updates** tab lists updates available for the plugins installed on the current Jenkins instance.
    - The **Available** tab contains the list of all the plugins available for Jenkins across the Jenkins community.
    - The **Installed** tab lists all the plugins currently installed on the current Jenkins instance.
    - The **Advanced** tab is used to configure Internet settings and also to update Jenkins plugins manually.
5. Let's see the **Advanced** tab in detail by clicking on it.

6. Right at the beginning, you will see a section named **HTTP Proxy Configuration**. Here, you can specify the HTTP proxy server details.
7. Provide the proxy details pertaining to your organization, or leave these fields empty if your Jenkins server is not behind a firewall.



> Jenkins uses the **HTTP Proxy Configuration** details when you try to update a Jenkins plugin from the **Update** tab, or when you install new plugins from the **Available** tab.

8. Just below the **HTTP Proxy Configuration** section, you will see the **Upload Plugin** section. It provides the facility to upload and install your own Jenkins plugin.



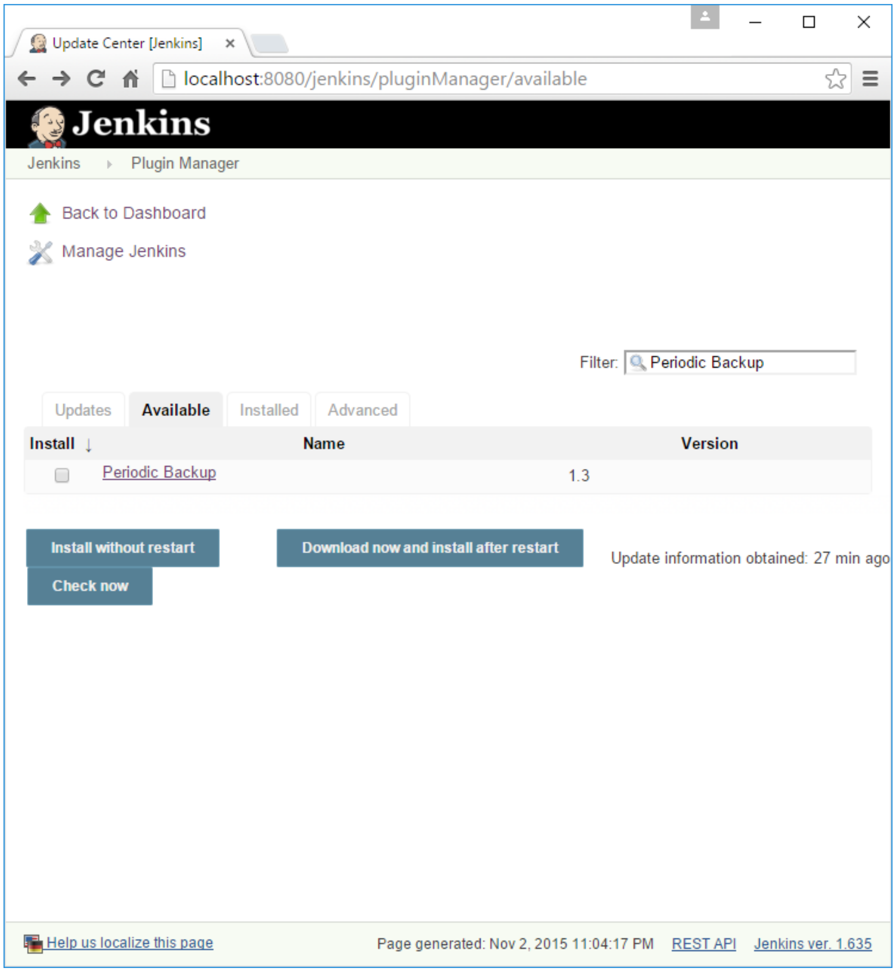
> The **Upload Plugin** section can also be used to install an existing Jenkins plugin that has been downloaded from `https://updates.jenkins-ci.org/download/plugins/`.
>
> You may ask why? Imagine a situation where you have a Jenkins instance running inside a local area network, but with no access to the Internet.
>
> In such scenarios, you will first download the required Jenkins plugin from the online Jenkins repository, and then you will transport it to the Jenkins master server using a removable media. Finally, you will use the **Upload Plugin** section to upload the required Jenkins plugin.

Installing a Jenkins plugin to take periodic backup

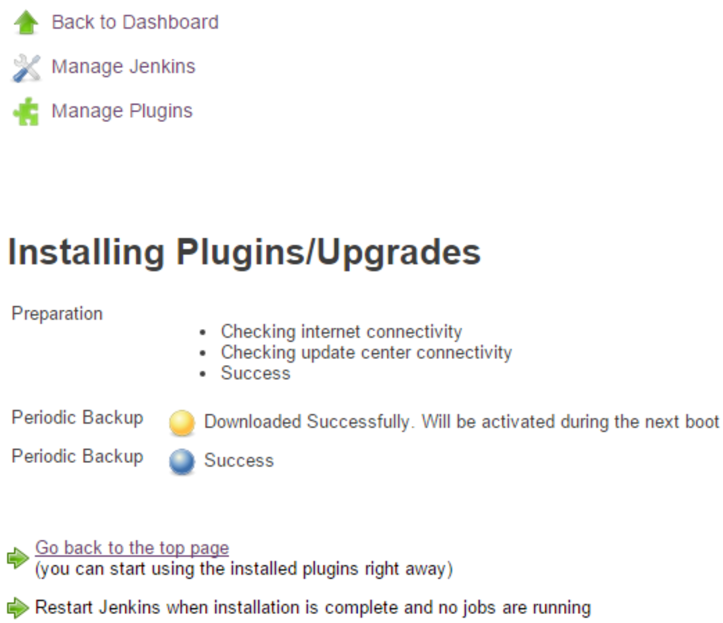
Let's try installing a plugin. In the previous sections, we saw a Jenkins job that creates a backup. Let's now install a plugin to do the same:

1. On the Jenkins **Plugin Manager** home page, go to the **Available** tab.
2. In the **Filter** field, type **Periodic Backup**.

3. Tick the checkbox beside the **Periodic Backup** plugin and click on **Install without restart**. This will download the plugin and then install it.

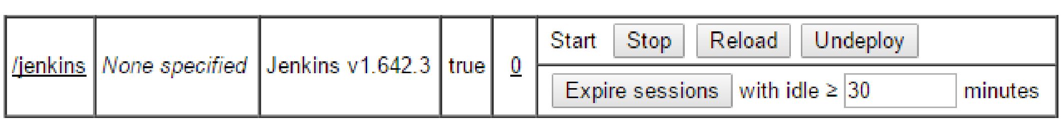


4. Jenkins immediately connects to the online plugin repository and starts downloading and installing the plugin, as shown in the following screenshot:



> Jenkins first tried to check its connectivity to the Jenkins online plugin repository. After a successful connection, it tried to download the desired plugin and at last the plugin was installed.
>
> This was a simple example. But there are cases where a Jenkins plugin has dependencies on other Jenkins plugins. In those cases, installing the plugin also means installing the dependencies. Nevertheless, it's automatically taken care of.

5. For the plugin to work, we need to restart the Jenkins server.
6. To restart Jenkins, go to the Apache Tomcat server home page and click on the **Manage App** button.
7. From the **Tomcat Web Application Manager** page, restart Jenkins by first clicking on the **Stop** button. Once Jenkins stops successfully, click on the **Start** button.

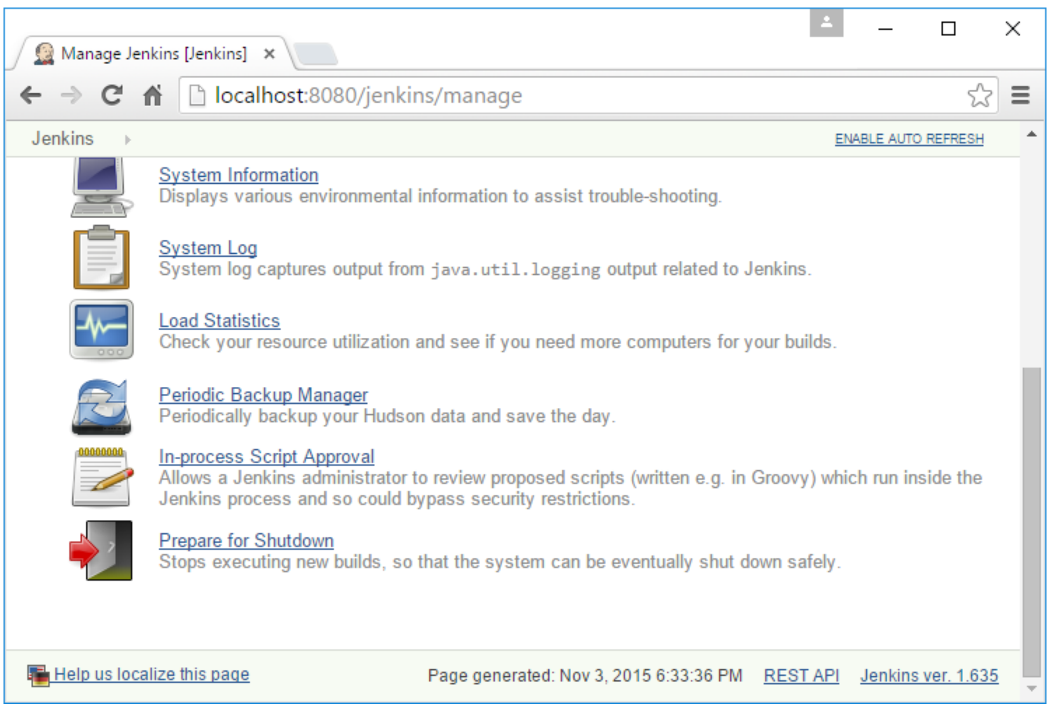


> You can also click on **Reload** button to restart Jenkins. After a restart, the Jenkins dashboard becomes inactive for some time and then resumes.

Configuring the periodic backup plugin

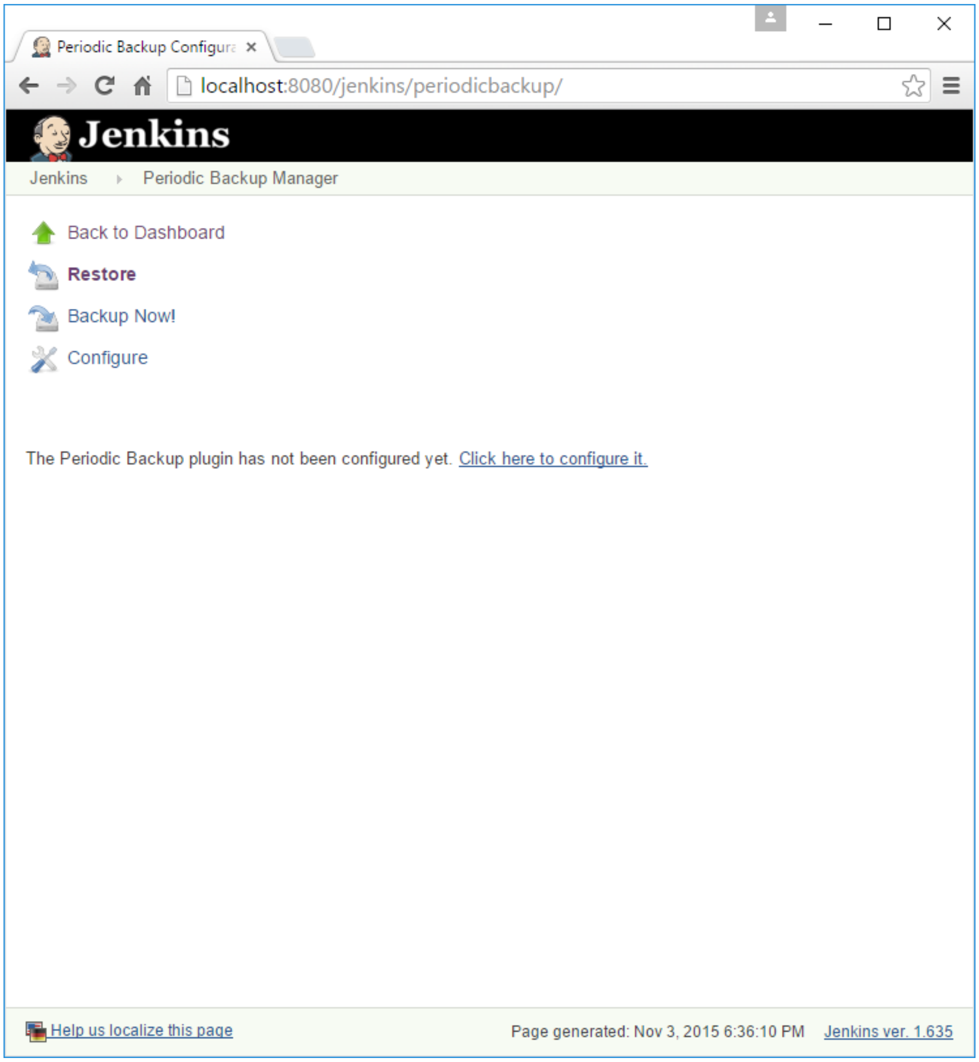
We have successfully installed the periodic backup plugin. Now, let's configure it:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. On the **Manage Jenkins** page, you will see the **Periodic Backup Manager** link.
3. Clicking on the **Periodic Backup Manager** link will take you to the **Periodic Backup Manager** page as shown in the following screenshot:



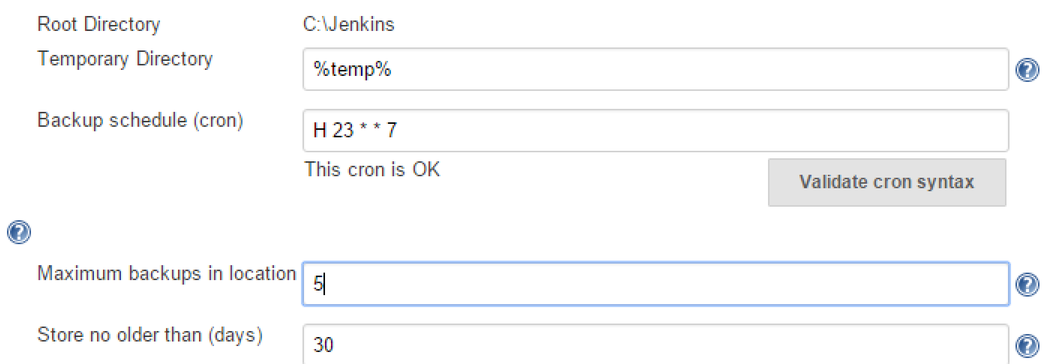
4. Clicking on **Backup Now!** creates a backup. However, it won't work presently as we have not configured the backup plugin.

5. The **Configure** link will allow you to configure the plugin.

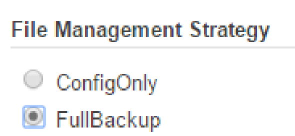


6. Click on the **Configure** link and you will see many options to configure your backup plugin:
    - **Temporary Directory**: This is where Jenkins will temporarily expand the archive files while restoring any backup. As you can see, I used an environment variable `%temp%`, but you can give any path on the machine.

- **Backup schedule (cron)**: This is the schedule that you want your backup to follow. I used `H 23 * * 7`, which is every Sunday anywhere between 23:00 to 23:59 hours throughout the year.
- **Maximum backups in location**: This is the total number of backups you want to store in a particular backup location. Does that mean we can have more than one backup location? Yes. We will see more on this soon.
- **Store no older than (days)**: This ensures any backup in any location which is older than the number of days specified is deleted automatically.

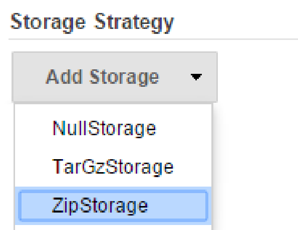


7. Scroll down to the **File Management Strategy** section. You will see the options to choose from **FullBackup** and **ConfigOnly**. Choose **FullBackup**.

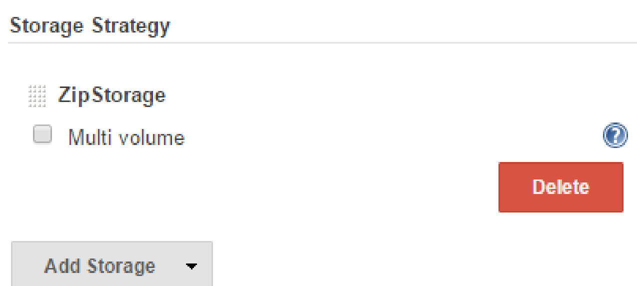


> **FullBackup** takes a backup of the whole Jenkins home directory. **ConfigOnly** takes only the backup of configurations and excludes the builds and logs.

8. In the following screenshot, you will see **Storage Strategy** section. Click on it and you will have options to choose from `.zip`, `.targz`, and **NullStorage**. I chose the `.zip` archive.



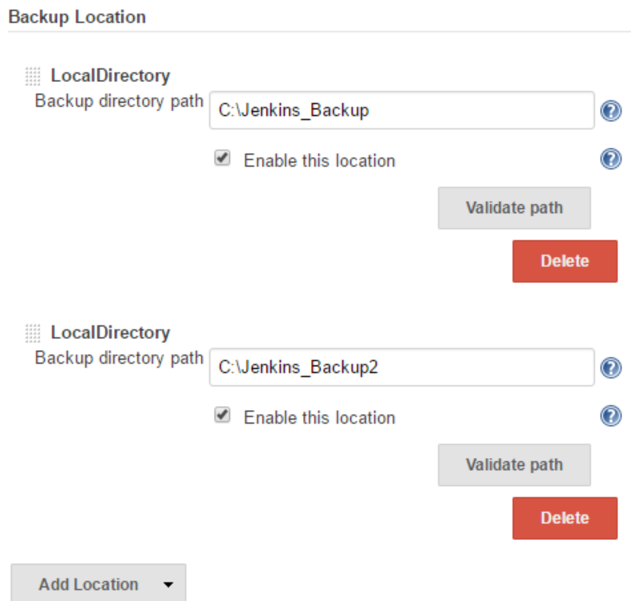
9. Clicking on the **ZipStorage** strategy provides us an option to select the **Multi volume** zip file, that is, one huge, single zip file split into many.



10. Just below **Storage Strategy**, you can see the **Backup Location** section where you can add as many backup locations as you want.

11. In my example, I added two backup locations, `C:\Jenkins_Backup` and `C:\Jenkins_Backup2` respectively.

12. As you can see from the following screenshot, I enabled both the locations.



13. Once done, click on the **Save** button.

> Click on the **Validate path** to validate the paths. Jenkins will not store more than five backups in any of the preceding backup locations; recall our option **Maximum backups in location** = **5**. By the time its gets overloaded, the backups will be deleted monthly; recall our option **Store no older than (days)** = **30**.

User administration

So far, all our Jenkins Jobs were running anonymously under an unidentified user. All the configurations that we did inside Jenkins were also done anonymously. But as we know, this is not how things should be. There needs to be a mechanism to manage users and define their privileges. Let's see what Jenkins has to offer in the area of user administration.

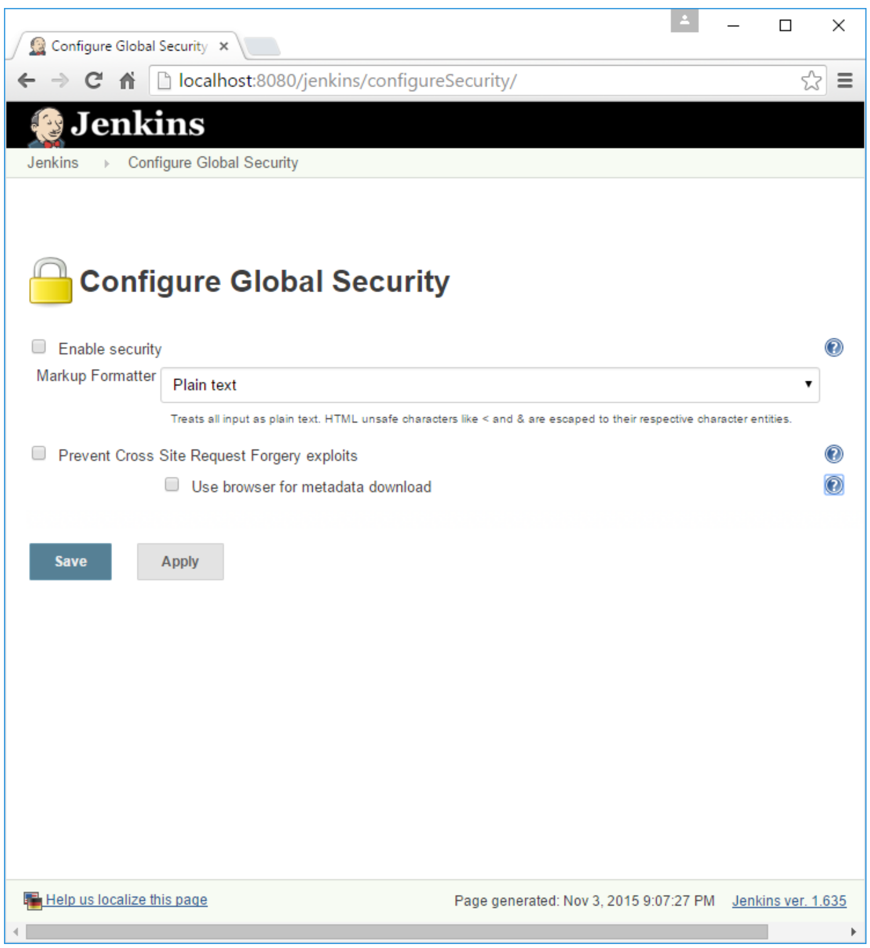
Enabling global security on Jenkins

The **Configure Global Security** section is the place where you get various options to secure Jenkins. Let see it in detail.

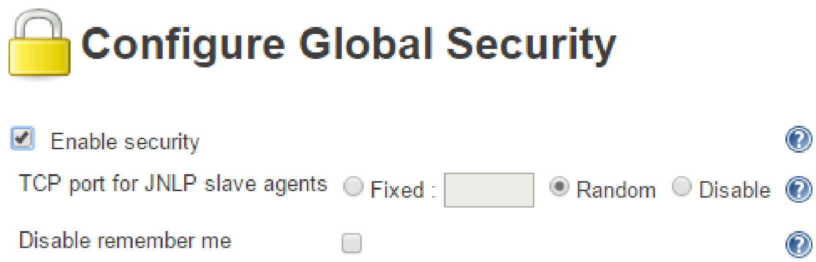
1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. From the Manage Jenkins page, click on the **Configure Global Security** link.

> You can also access the **Configure Global Security** page by using the link `http://localhost:8080/jenkins/configureSecurity/`.

3. The following screenshot shows what the **Configure Global Security** page looks like:



4. Click on the **Enable security** checkbox and a new set of options will be available to configure.
5. Leave the **TCP port for JNLP slave agents** option as it is (**Random**).
6. Leave the **Disable remember me** option unchecked.

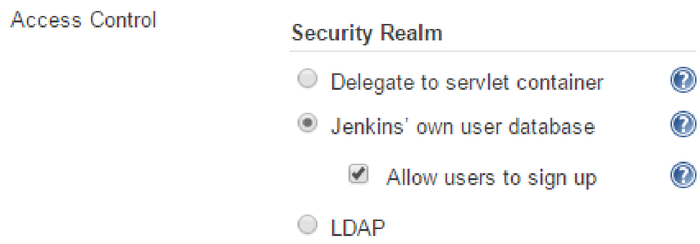


> Enable the **Disable remember me** option if you don't want the browser to remember the usernames.

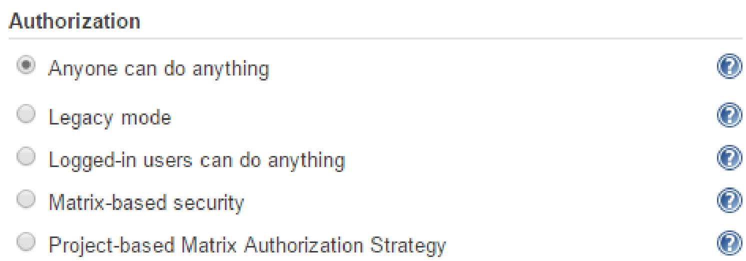
7. Go to the **Security Realm** subsection which is under the **Access Control** section. In our example, we will use the **Jenkins' own user database** option to manage users and permissions.

> The **Delegate to servlet container** option allows you to inherit all the users from the servlet containing your Jenkins master server (Apache Tomcat server).
>
> The **Jenkins' own user database** option allows you to create and define your own set of user accounts and permissions.
>
> If you have an active directory server configured for your organization, with all the users in it, use the **LDAP** option.

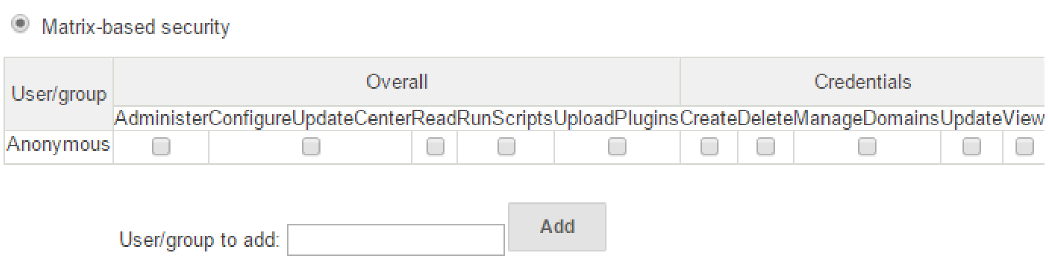
8. Select the **Jenkins' own user database** and you will get another option, which allows users to sign up. This is shown in the following screenshot:



9. Come down to the **Authorization** section, and you will see the following options:



10. Choose the **Matrix-based security** option.
11. The following illustration is partial, that means there is more towards the right side.
12. To add users, enter the user names in the **User/group** to add a field, and click the **Add** button. For now, do not add any users.



In a **Matrix-based security** setting, all the **Users/Groups** are listed across rows and all the Jenkins tasks are listed across columns. It's a matrix of users and tasks. This matrix makes it possible to configure permissions at the task level for each user.

13. Select all the checkboxes for the **Anonymous** user. By doing this, we are giving the **Anonymous** user admin privileges.

User/group	Overall				
	Administer	ConfigureUpdateCenter	Read	RunScripts	UploadPlugins
Anonymous	✔	✔	✔	✔	✔

Credentials					Slave					
Create	Delete	ManageDomains	Update	View	Build	Configure	Connect	Create	Delete	Disconnect
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

Job								Run		View				SCM
Build	Cancel	Configure	Create	Delete	Discover	Read	Workspace	Delete	Update	Configure	Create	Delete	Read	Tag
✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

14. Click on the **Save** button at the bottom of the page once done.

Creating users in Jenkins

Currently, we do not have any users configured on our Jenkins master. At this point, everyone is free to access Jenkins and perform any given action. This is because the **Anonymous** group has all the privileges.

Creating an admin user

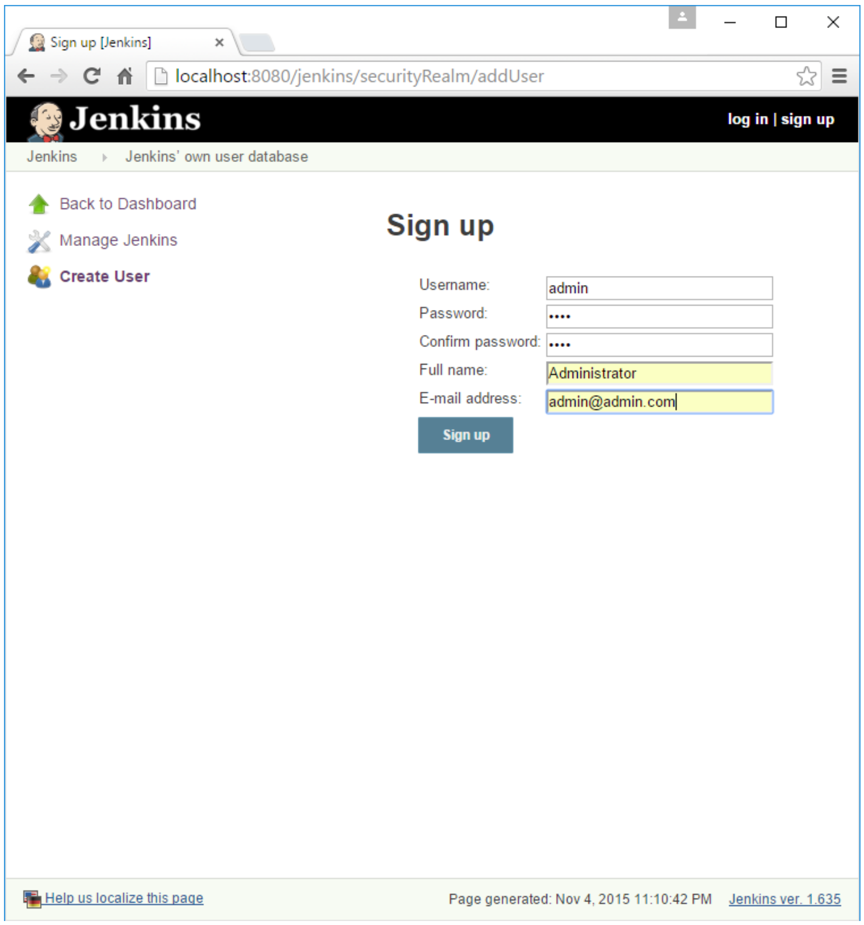
So, let us first create a user named `admin`. Then we will move all the privileges from the **Anonymous** group to our new `admin` user.

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. On the **Manage Jenkins** page, scroll down and click on **Manage Users**.
3. On the **Users** page, using the menu on the left side, click on the **Create User** link to create users, as shown in the following screenshot:

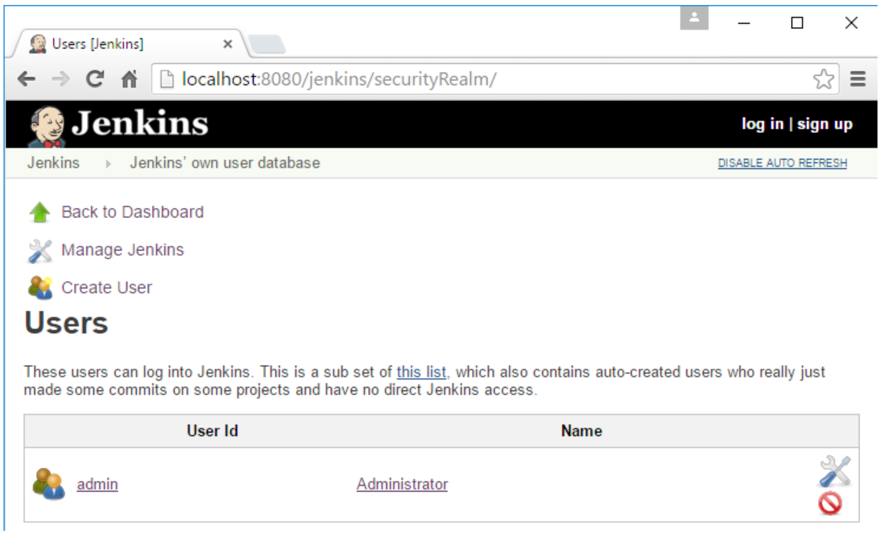
All the unsigned users who access the Jenkins master, fall under the **Anonymous** group and inherit all its privileges.

4. You will see a **Sign up** form to fill. Give the username as `admin` (or you can choose any name for that matter), give a strong password of your choice. Remember the password.

5. Fill the other details like **Full name** and **E-mail address** accordingly and click on the **Sign up** button.

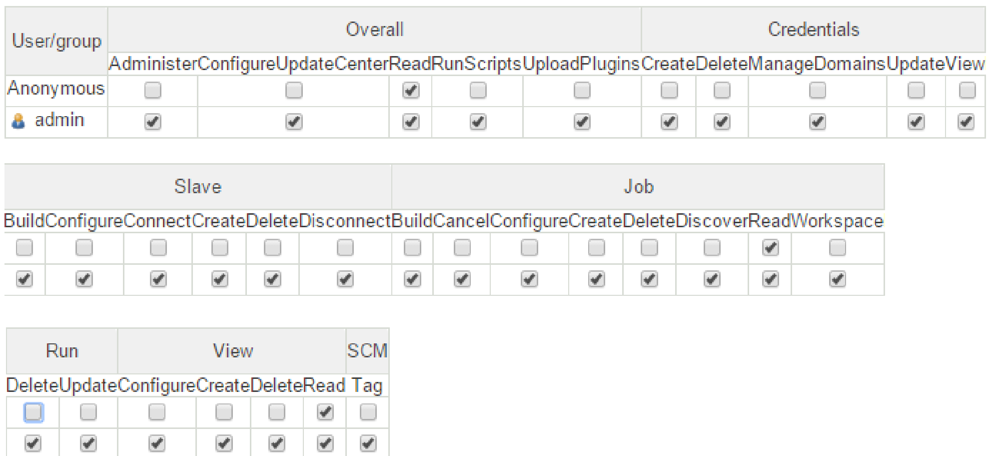


6. A user named `admin` gets created, as shown in the following screenshot:

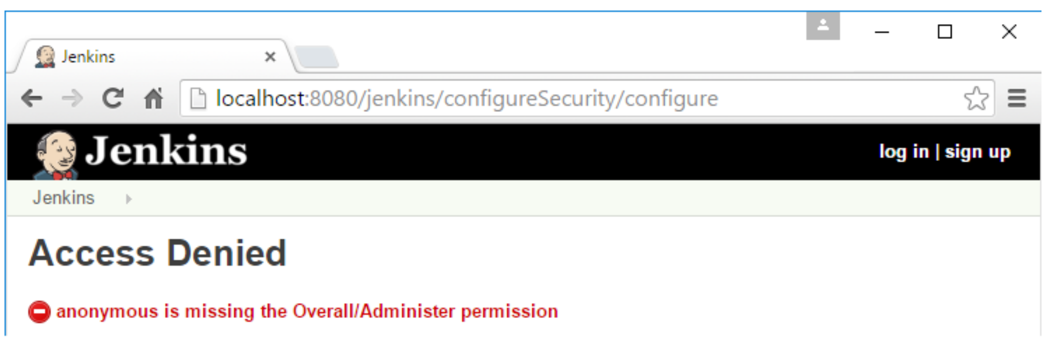


7. Do not log in with this user for now. At this point of time, the user `admin` is just a regular user with no privileges. The real admin at this moment is the **Anonymous** group.
8. From the **Manage Jenkins** page, go to the **Configure Global Security** page. Here, we will make the newly created `admin` user an administrator in real terms.
9. On the **Configure Global Security** page, scroll down to the **Authorization** section.
10. Type `admin` inside the **User/group to add** field and click on the **Add** button.
11. Once added, check all the boxes for the user `admin`.

12. On the other hand, uncheck everything and keep only the read-only type privileges for the **Anonymous** group, as shown in the following screenshot:

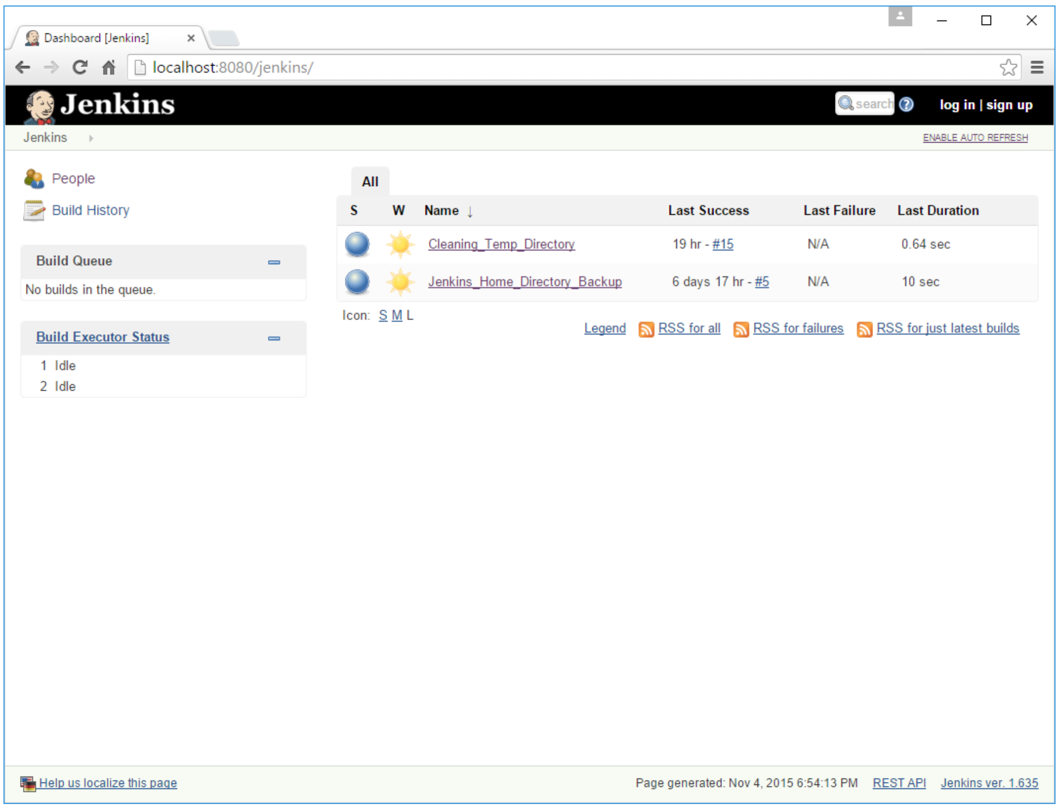


13. Click on the **Save** button and you will be redirected to a new page:

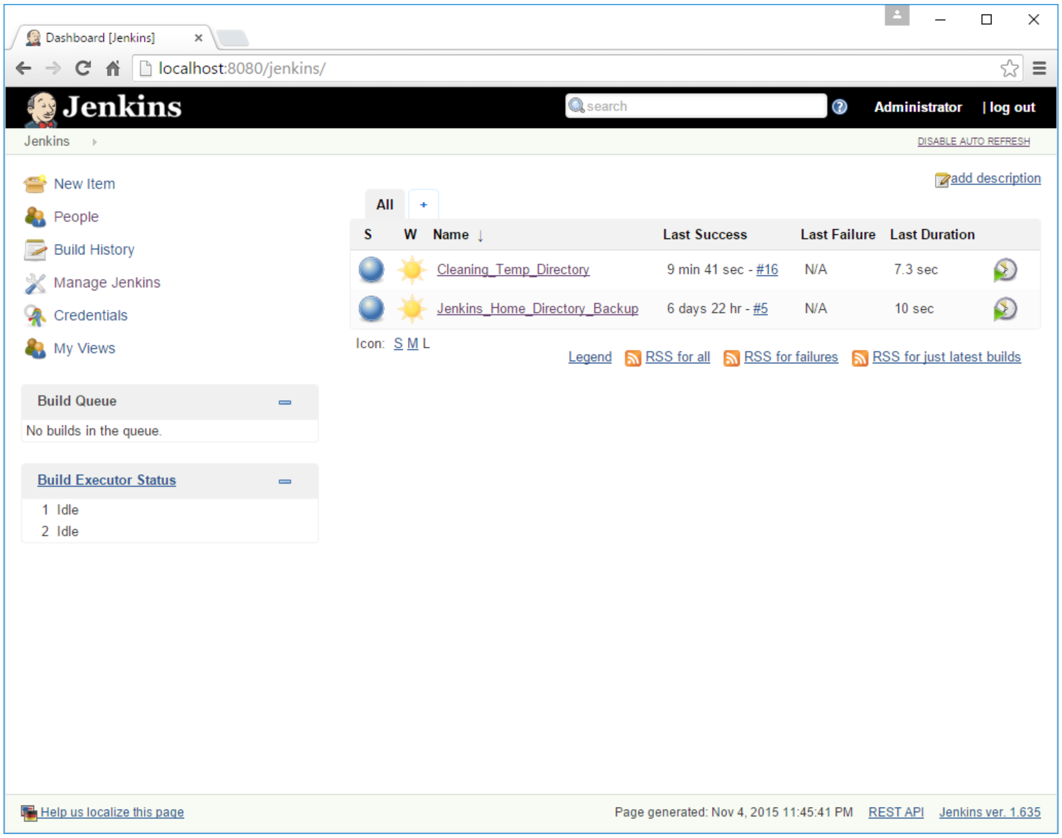


14. This was inevitable, as we have stripped the admin privileges from **Anonymous** group.
15. Nevertheless, we have also transferred the admin privileges to the `admin` user that we created recently.
16. Just to be on the safer side, restart Jenkins.

17. Upon restart, you will see the Jenkins Dashboard as shown in the following screenshot. We are currently using Jenkins as an anonymous user. You can see the build buttons have been disabled. The **Manage Jenkins** link is also disabled.
18. Click on the **log in** link present at the top-right corner and log in as the `admin` user.



19. The following is how you should see the dashboard after logged-in as `admin`. All the admin privileges have been granted.

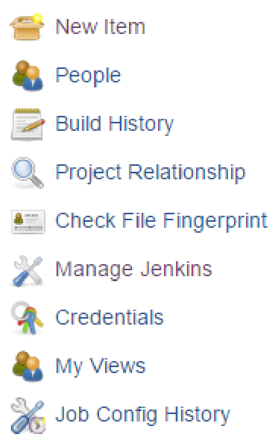


Creating other users

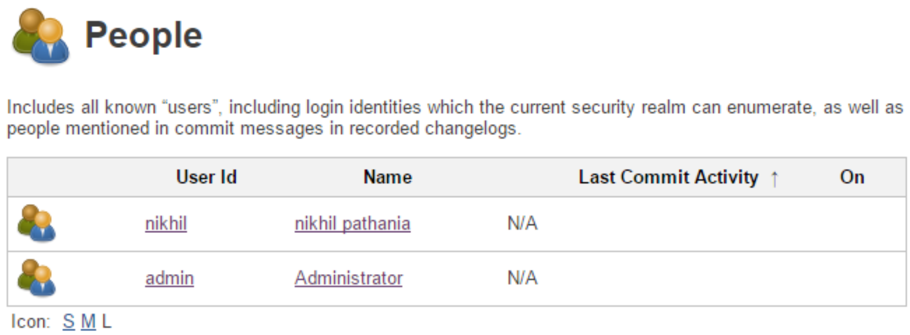
Users can always sign up and create their account in Jenkins using the **sign up** link at the top-right corner. All such users by default get all the privileges of an anonymous group.

1. The following screenshot shows an example where I created my own account.

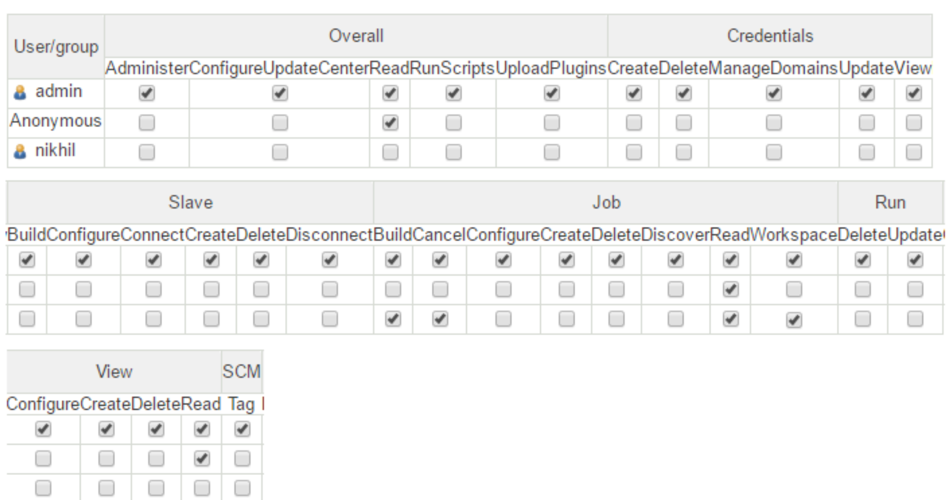
2. You can try creating as many accounts as you want and see all those come under the anonymous category by default.
3. To see the list of Jenkins users, from the Jenkins Dashboard, click on the **People** link present at the top-left section.



4. All the users are listed on the **People** page, as shown in the following screenshot:



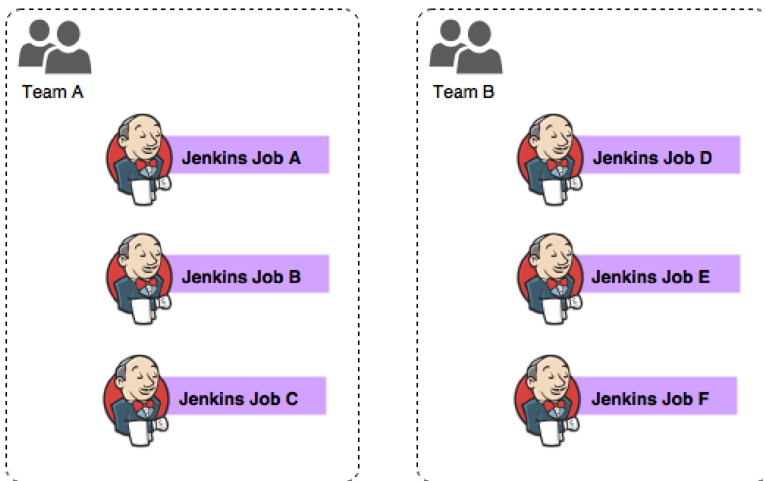
5. To give permissions to our newly created user, log into Jenkins as the `admin` user.
6. From the **Manage Jenkins** page, go to the **Configure Global Security** page.
7. On the **Configure Global Security** page, scroll down to the **Authorization** section.
8. Inside the **User/group to add** field, add the username that has signed up on the Jenkins master and click on the **Add** button. In my example, I added a user `nikhil` that I recently created.
9. Once added, give the new user permissions to **Build**, **Cancel**, **Workspace**, and **Read** a Jenkins jobs, as shown in the following screenshot:



10. Click on the **Save** button at the end of the page to save the settings.
11. Log in as the new user and you will notice that you can only execute builds, but you cannot change the job configuration or the Jenkins system settings.

Using the Project-based Matrix Authorization Strategy

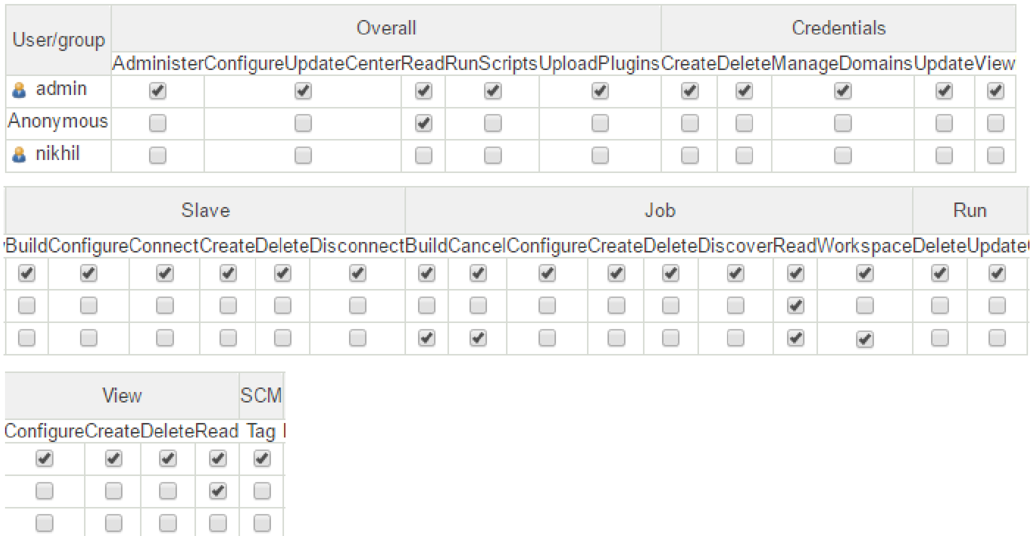
In the previous section, we saw the **Matrix-based security** authorization feature which gave us a good amount of control over the users and permissions. However, imagine a situation where your Jenkins master server has grown to a point, where it contains multiple projects (software projects), hundreds of Jenkins jobs and many users. You want the users to have permissions only on the jobs they use. In such a case, we need the **Project-based Matrix Authorization Strategy** feature.



Let's learn to configure the **Project-based Matrix Authorization Strategy** feature:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. On the **Manage Jenkins** page, click on the **Configure Global Security** link.

3. Here's what our current configuration looks like:

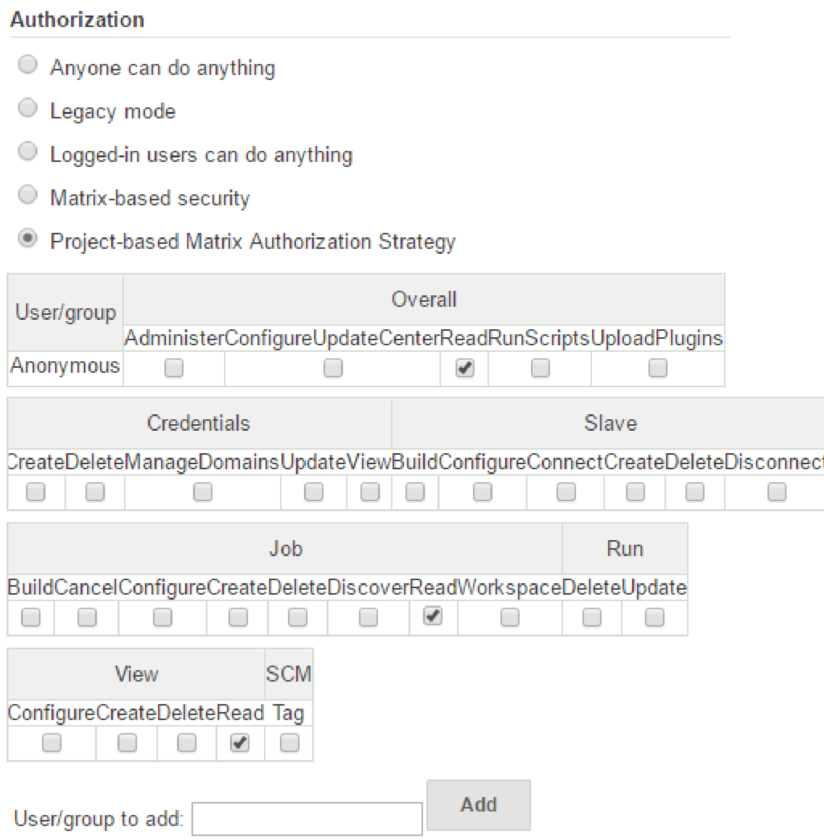


User/group	Overall					Credentials				
	Administer	ConfigureUpdateCenter	Read	RunScripts	UploadPlugins	Create	Delete	ManageDomains	Update	View
admin	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Anonymous			✔							
nikhil										

User/group	Slave						Job								Run	
	Build	Configure	Connect	Create	Delete	Disconnect	Build	Cancel	Configure	Create	Delete	Discover	Read	Workspace	Delete	Update
admin	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
Anonymous													✔			
nikhil							✔	✔					✔	✔		

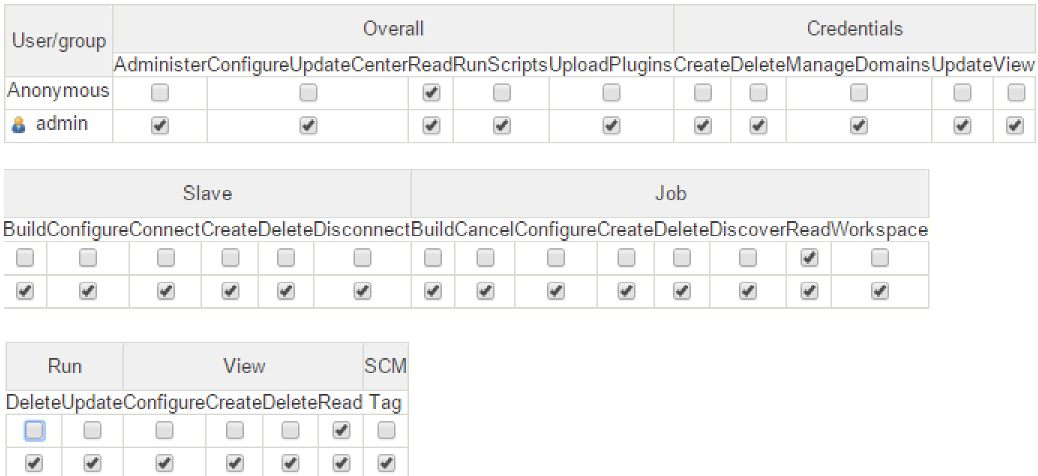
User/group	View				SCM
	Configure	Create	Delete	Read	Tag
admin	✔	✔	✔	✔	✔
Anonymous				✔	
nikhil					

4. Select the **Project-based Matrix Authorization Strategy** option.



5. Inside the **User/group to add** field, add the username that has signed up on the Jenkins master and click on the **Add** button. Do not forget to add the `admin` user.

6. The output should look like the following screenshot:

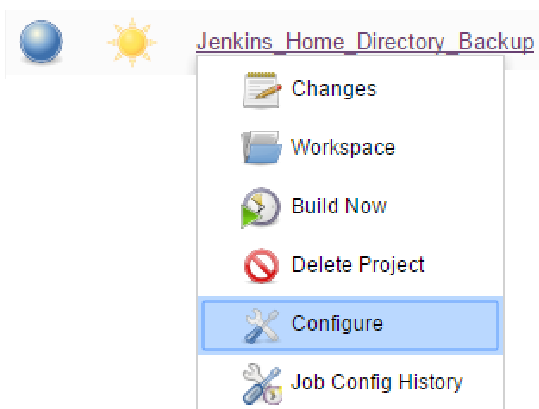


User/group	Overall					Credentials				
	Administer	ConfigureUpdateCenter	Read	RunScripts	UploadPlugins	Create	Delete	ManageDomains	Update	View
Anonymous	☐	☐	✓	☐	☐	☐	☐	☐	☐	☐
admin	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

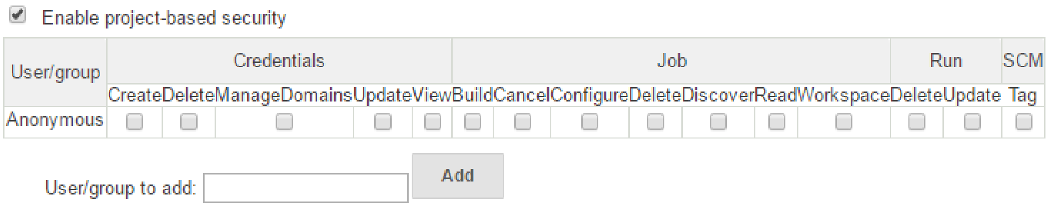
Slave						Job							
Build	Configure	Connect	Create	Delete	Disconnect	Build	Cancel	Configure	Create	Delete	Discover	Read	Workspace
☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓	☐
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Run		View				SCM
Delete	Update	Configure	Create	Delete	Read	Tag
☐	☐	☐	☐	☐	✓	☐
✓	✓	✓	✓	✓	✓	✓

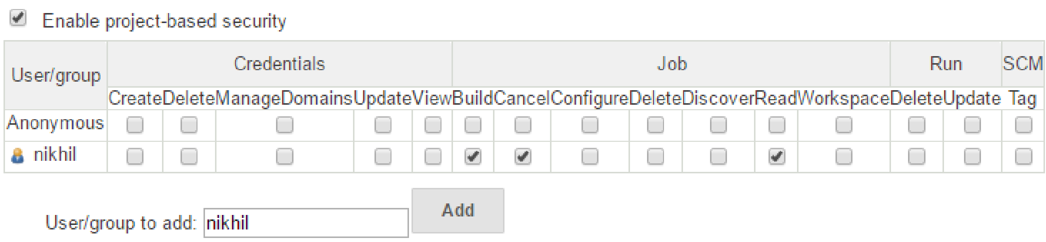
7. Click on the **Save** button at the end of the page to save the configuration.
8. From the Jenkins Dashboard, right-click on any of the Jenkins jobs and select **Configure**.



9. On the job's configuration page, select the newly available option **Enable project-based security**, which is right at the beginning.



10. Now, inside the **User/group to add** field, add the username that you want to give access to the current job.
11. As shown in the following screenshot, I added a user `nikhil` who has the permission to build the current job.



12. Once done, click on the **Save** button at the end of the page.

Summary

In this chapter, we saw how to configure some of the basic but important stuff in Jenkins, all with the help of some practical examples. We created a few Jenkins jobs and also wrote some simple scripts inside it.

Jenkins upgrade, Jenkins backup, and Jenkins user management are some of the important things we discussed in this chapter. However, if you think that you can perform these respective tasks in a better way, incorporating some more options that you encountered in Jenkins, then the objective of this chapter has been achieved.

The idea is that any particular task simple or complex can be performed in many ways using Jenkins. All that matters is how creative you are with the features provided by the tool.

4

Continuous Integration Using Jenkins – Part I

We will begin the current chapter with a Continuous Integration Design that covers the following areas:

- A branching strategy
- List of tools for Continuous Integration
- The Jenkins pipeline structure

The **Continuous Integration** (**CI**) Design will serve as a blueprint that will guide the readers in answering the how, why, and where of the Continuous Integration being implemented. The design will cover all the necessary steps involved in implementing an end-to-end CI pipeline. Therefore, due to the huge amount of information, the implementation of CI has been spread across this chapter and *Chapter 5*, *Continuous Integration Using Jenkins – Part II*.

The CI design discussed in this chapter should be considered as a template for implementing Continuous Integration and not a full and final model. The branching strategy and the tools used can be modified and replaced to suit the purpose.

We will also discuss installing and configuring Git, a popular version control system. The current chapter and the next chapter will also give the readers an idea of how well Jenkins gels with many other tools to achieve Continuous Integration.

Jenkins Continuous Integration Design

I have used a new term here: *Continuous Integration Design*. Almost every organization creates one before they even begin to explore the CI and DevOps tools. In the current section, we will go through a very general Continuous Integration Design.

Continuous Integration includes not only Jenkins or any other similar CI tool for that matter, but it also deals with how you version control your code, the branching strategy you follow, and so on. If you are feeling that we are overlapping with **software configuration management**, then you are right.

Various organizations may follow different kinds of strategies to achieve Continuous Integration. Since, it all depends on the project requirements and type.

The branching strategy

It's always good to have a branching strategy. Branching helps you organize your code. It is a way to isolate your working code from the code that is under development. In our Continuous Integration Design, we will start with three types of branches:

- Master branch
- Integration branch
- Feature branch

Master branch

You can also call it the **production branch**. It holds the working copy of the code that has been delivered. The code on this branch has passed all the testing stages. No development happens on this branch.

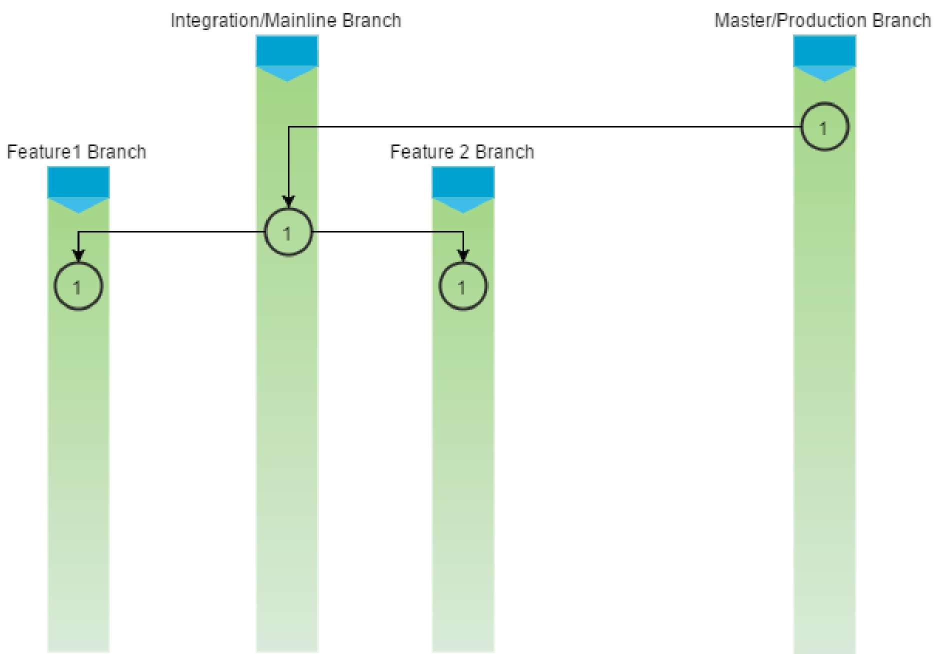
Integration branch

The integration branch is also known as the **mainline branch**. This is where all the features are integrated, built, and tested for integration issues. Again, no development happens here. However, developers can create feature branches out of the integration branch and work on them.

Feature branch

Lastly we have the feature branch. This is where the actual development takes place. We can have multiple feature branches spanning out of the integration branch.

The following image shows a typical branching strategy that we will use as part of our Continuous Integration Design. We will create two feature branches spanning out from the integration/mainline branch, which itself spans out from the master branch.



- A successful commit (code check-in) on the feature branch will go through a build and unit test phase. If the code passes these phases successfully, it is merged to the integration branch.
- A commit on the integration branch (a merge will create a commit) will go through a build, static code analysis, and integration testing phase. If the code passes these phases successfully, the resultant package is uploaded to Artifactory (binary repository).

The Continuous Integration pipeline

We are now at the heart of the Continuous Integration Design. We will create two pipelines in Jenkins, which are as follows:

- Pipeline to poll the feature branch
- Pipeline to poll the integration branch

These two pipelines work in sequence and, as a whole, form the Continuous Integration pipeline. Their purpose is to automate the process of continuously building, testing (unit test and integration test), and integrating the code changes. Reporting failure/success happens at every step.

Let's discuss these pipelines and their constituents in detail.

Jenkins pipeline to poll the feature branch

The Jenkins pipeline to poll the feature branch is coupled with the feature branch. Whenever a developer commits something on the feature branch, the pipeline gets activated. It contains two Jenkins jobs that are as follows:

Jenkins job 1

The first Jenkins Job in the pipeline performs the following tasks:

- It polls the feature branch for changes on a regular interval
- It performs a build on the modified code
- It executes the unit tests

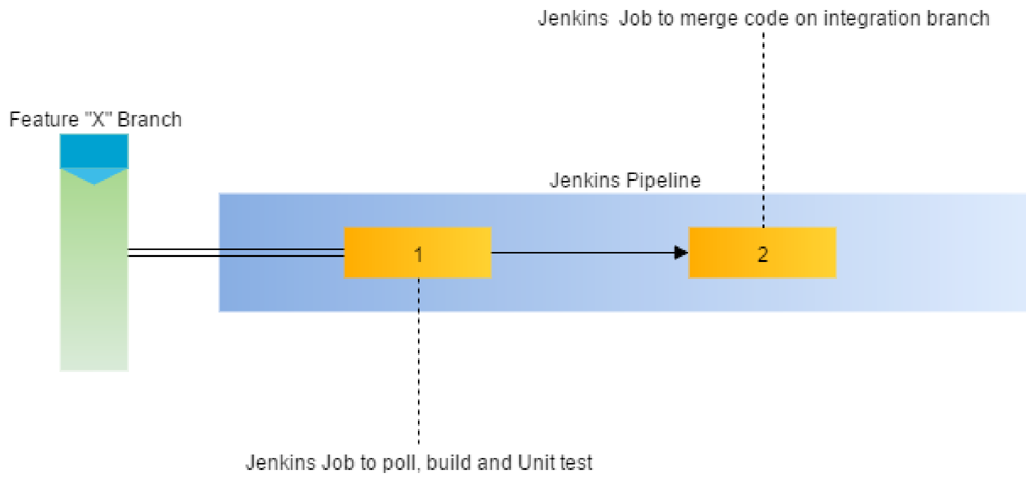
Jenkins job 2

The second Jenkins Job in the pipeline performs the following task:

- It merges the successfully built and tested code onto the integration branch

If this is the first time you are seeing a Jenkins job performing automated merges, then you are not alone. The reason is such automation is mostly done across projects that are very mature in using Continuous Integration and where almost everything is automated and configured well.

The following figure depicts the pipeline to poll the feature branch:



Jenkins pipeline to poll the integration branch

This Jenkins pipeline is coupled with the integration branch. Whenever there is a new commit on the integration branch, the pipeline gets activated. It contains two Jenkins jobs that perform the following tasks.

Jenkins job 1

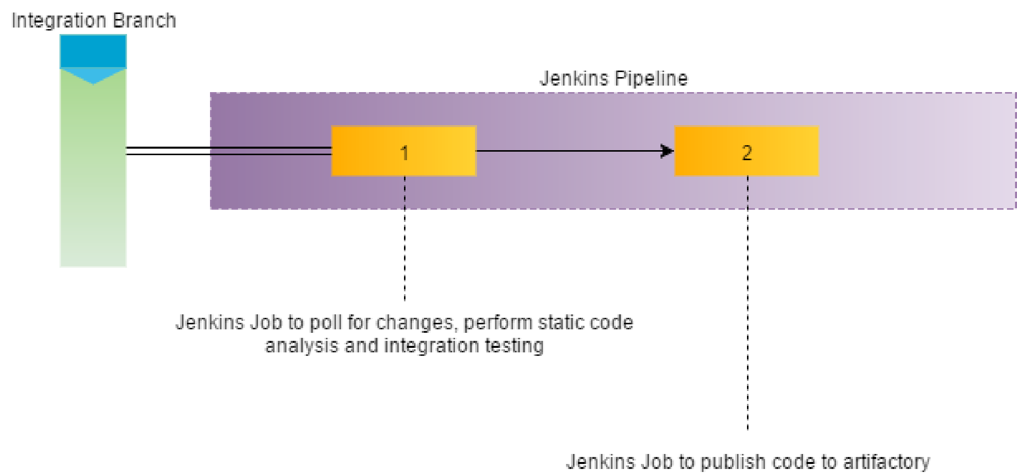
The first Jenkins job in the pipeline performs the following tasks:

- It polls the integration branch for changes at regular intervals
- Performs static code analysis on the code
- It builds and executes the integration tests

Jenkins job 2

The second Jenkins job in the pipeline performs the following tasks:

- It uploads the built package to Artifactory (binary code repository)



- Merge operations on the integration branch creates a new commit on it
- Each consecutive Jenkins job runs only when its preceding Jenkins job is successful
- Any success/failure event is quickly circulated among the respective teams using notifications

Toolset for Continuous Integration

The example project for which we are implementing Continuous Integration is a Java-based web application. Therefore, we will see Jenkins working closely with many other tools.

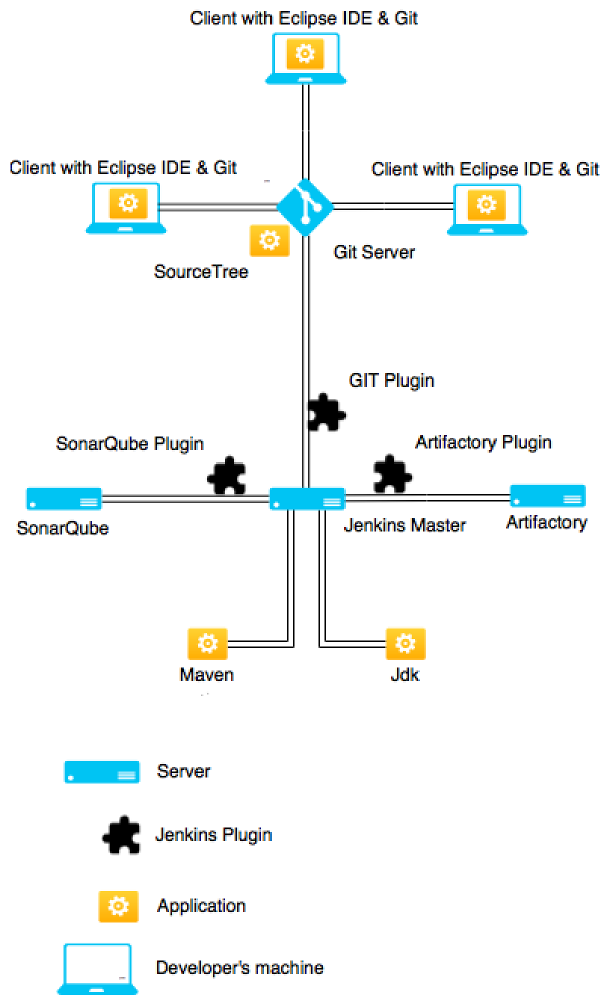
The following table contains a list of the tools and technologies involved in everything that we will see later in this chapter and in *Chapter 5, Continuous Integration Using Jenkins – Part II*.

Technologies	**Description**
Java	Primary programming language used for coding
Maven	Build tool
JUnit	Unit test and integration test tools
Apache Tomcat server	Servlet to host the end product
Eclipse	IDE for Java development
Jenkins	Continuous Integration tool
Git	Version control system
SourceTree	Git client
SonarQube	Static code analysis tool

The next figure shows how Jenkins fits in as a CI server in our Continuous Integration Design, along with the other DevOps tools.

- The developers have got Eclipse IDE and Git installed on their machines. This Eclipse IDE is internally configured with the Git server. This enables the developers to clone the feature branch from the Git server onto their machines.
- The Git server is connected to the Jenkins master server using the Git plugin. This enables Jenkins to poll the Git server for changes.
- The Apache Tomcat server, which hosts the Jenkins master, has also got Maven and JDK installed on it. This enables Jenkins to build the code that has been checked in on the Git server.
- Jenkins is also connected to SonarQube server and the Artifactory server using the SonarQube plugin and the Artifactory plugin respectively.

- This enables Jenkins to perform a static code analysis on the modified code. And once the build, testing, and integration steps are successful, the resultant package is uploaded to the Artifactory for further use.



Setting up a version control system

Now that we have our Continuous Integration Design ready, let's begin with the **version control system** (**VCS**) installation. In this section, we will see how to install and configure Git. This includes:

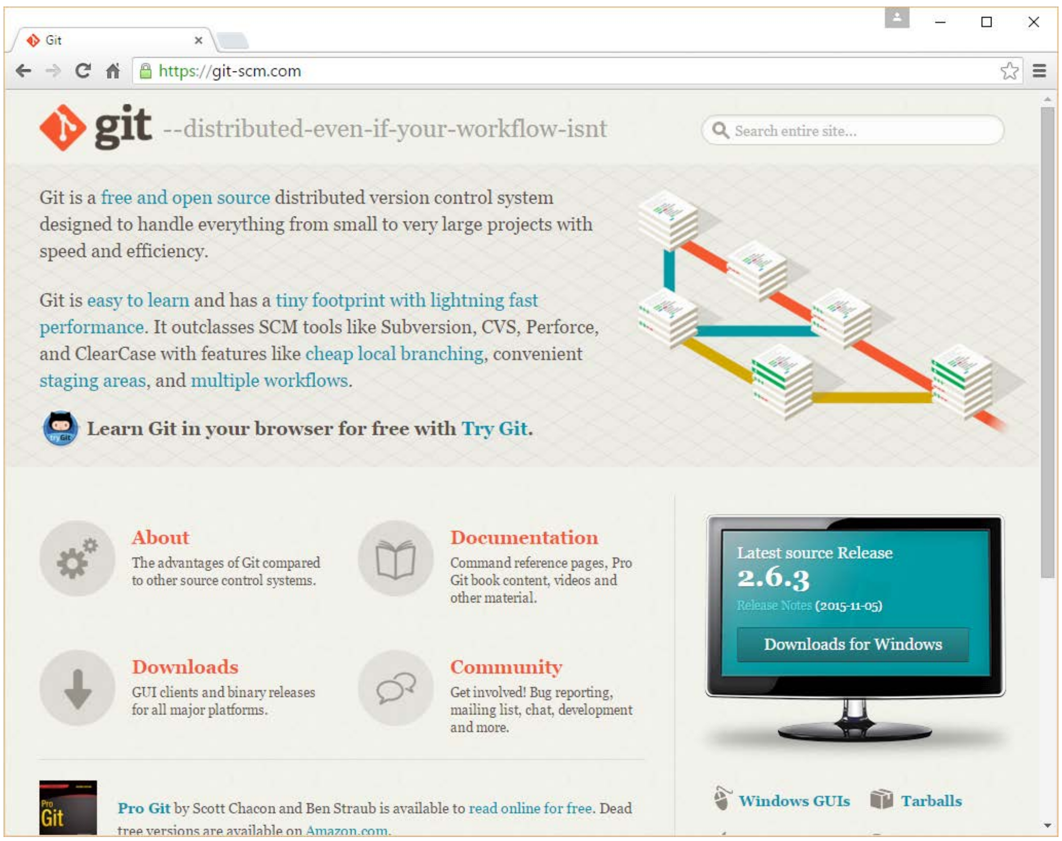
- Downloading and installing Git

- Downloading and installing the Git client
- Creating a Git repository and uploading code onto it
- Creating branches

Installing Git

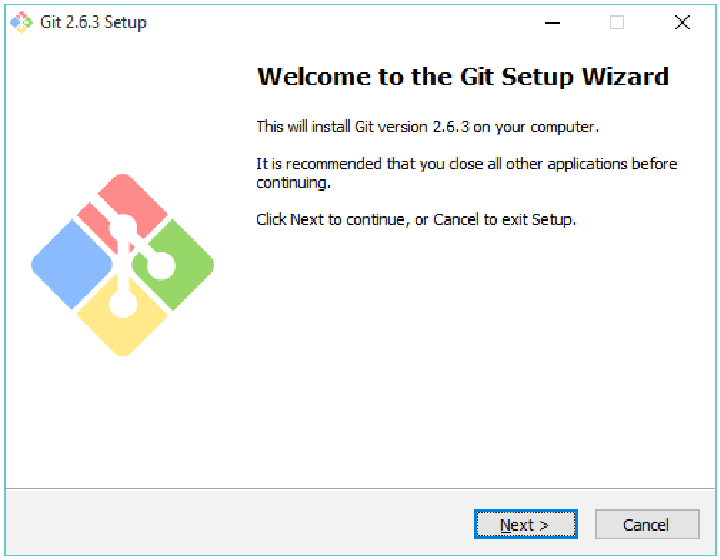
Perform the following steps to install Git:

1. We will install Git on a Windows machine. You can download the latest Git executable from `https://git-scm.com/`, as shown in the following screenshot:

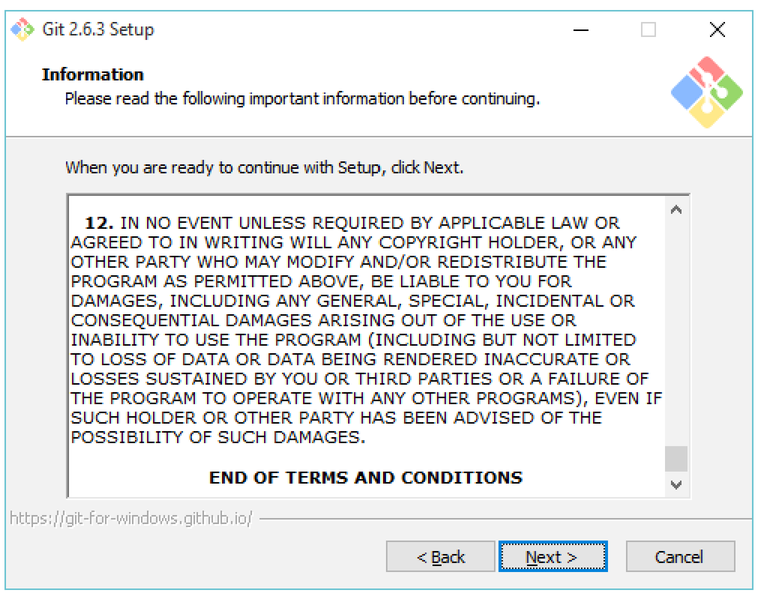


2. Begin the installation by double-clicking on the downloaded executable file.

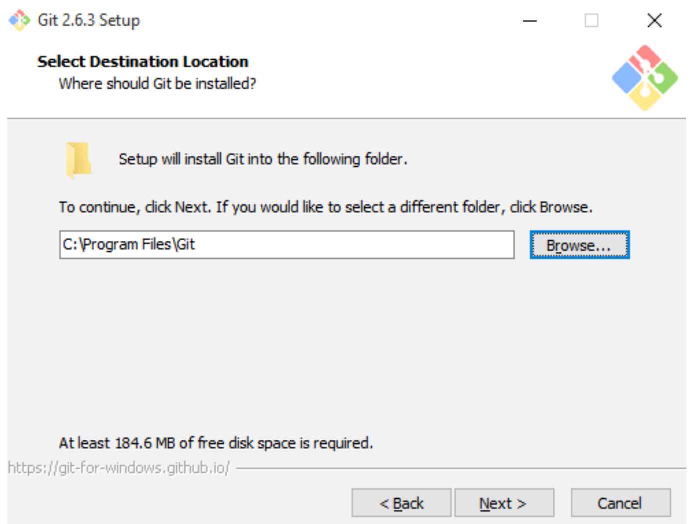
3. Click on the **Next** button.



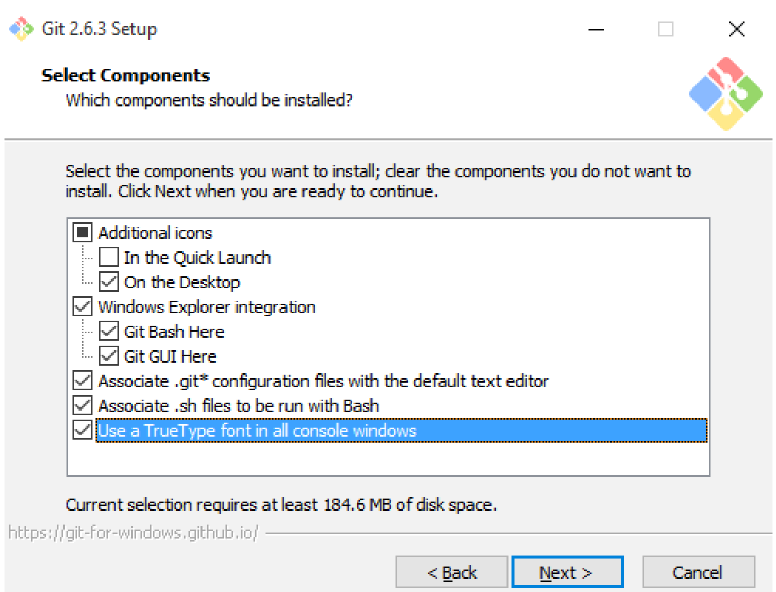
4. Accept the terms and conditions and click on the **Next** button.



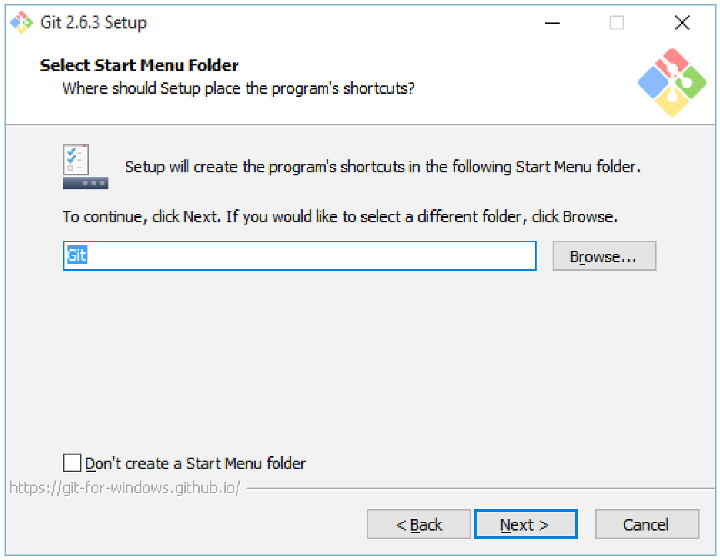
5. Choose the installation directory `C:\Program Files\Git`. Click on the **Next** button to proceed.



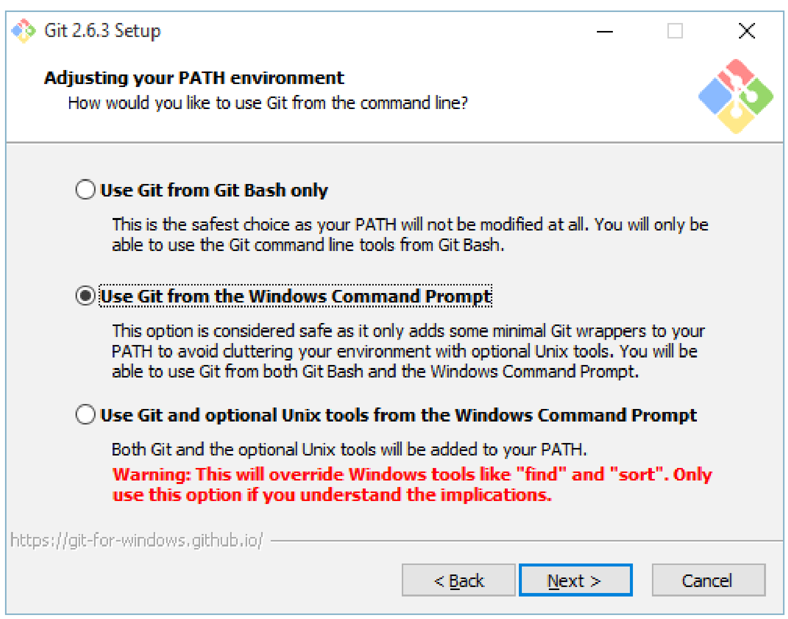
6. Select all the necessary options, as shown in the following screenshot. Click on the **Next** button to proceed.



7. Give the start menu folder a name. You can choose to skip this by selecting the **Don't create a Start Menu Folder** option.

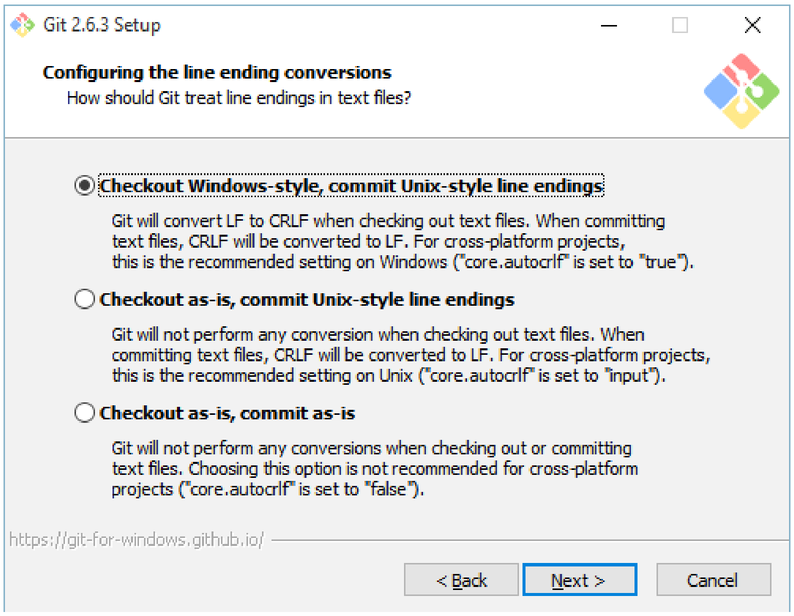


8. The following options are self-explanatory. Since we are installing Git on a Windows machine, the second option is preferable, as shown in the following screenshot. Click on the **Next** button to proceed.

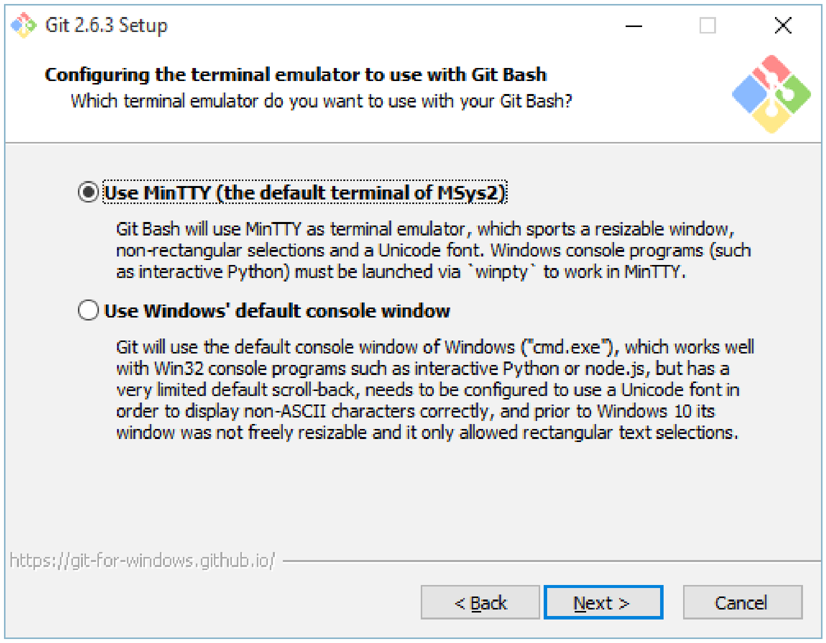


9. Again, the options in the following screenshot are self-explanatory. Choose the first option as we are installing Git on a Windows machine.

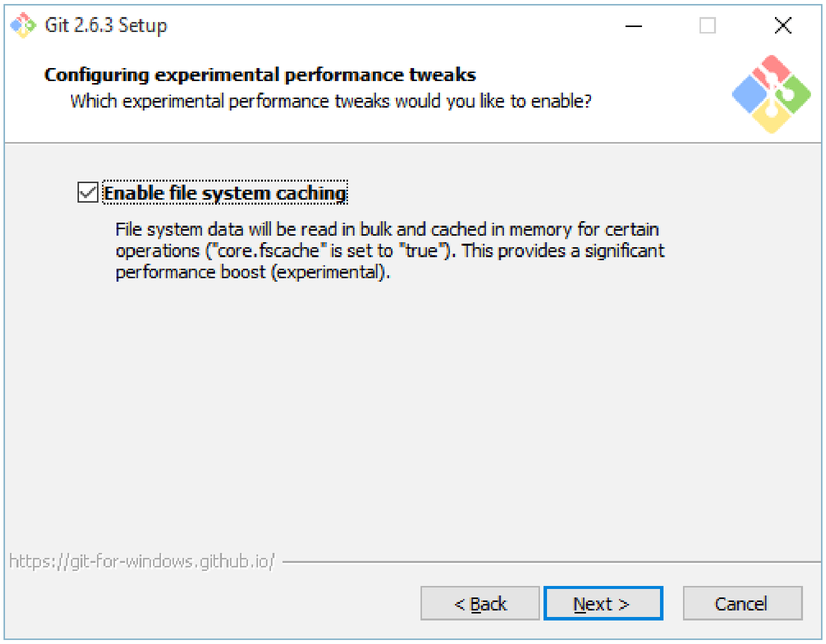
> We would have chosen the second option if we were installing Git on a Linux or a Unix machine.



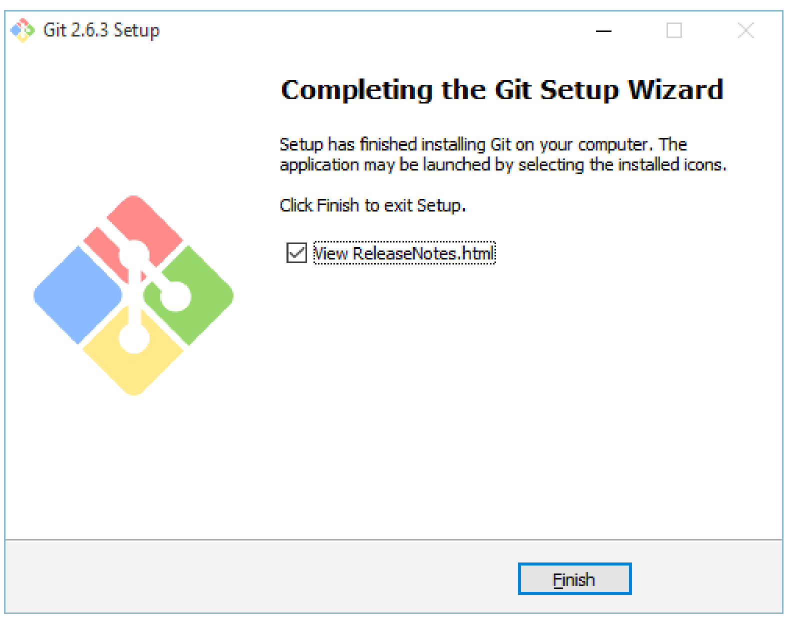
10. Choose as per your preference and click on the **Next** button to proceed.



11. This option is experimental. Selecting the option **Enable file system caching** may improve performance. Click on the **Next** button to proceed.



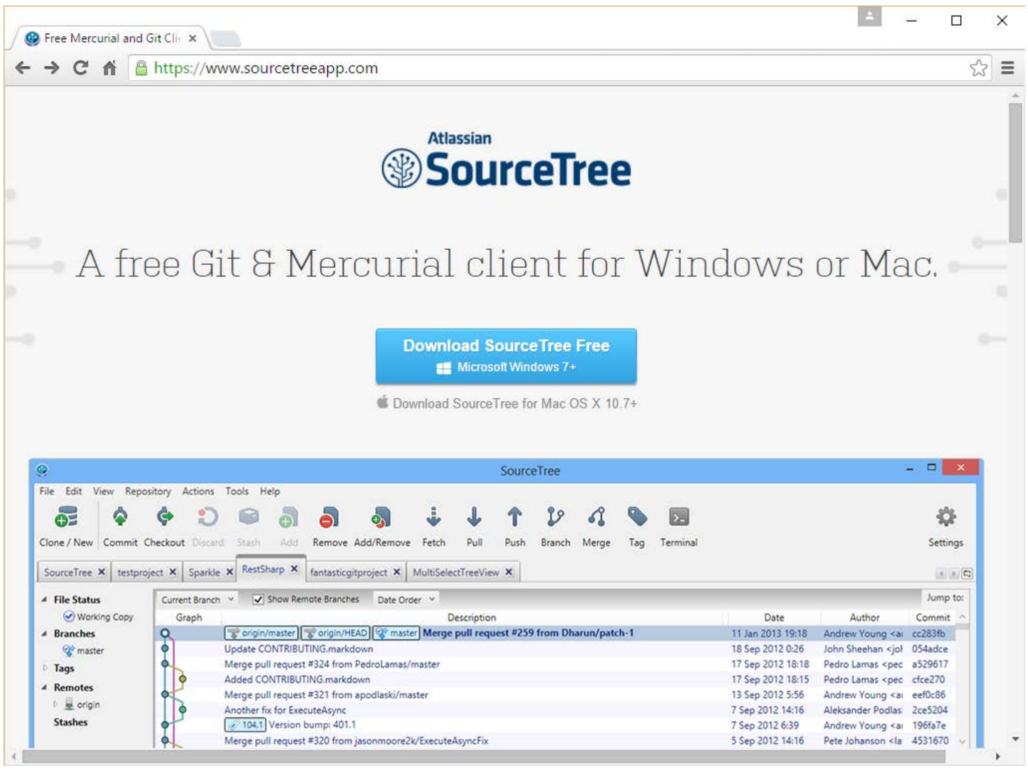
12. Click on the **Finish** button to complete the installation.



Installing SourceTree (a Git client)

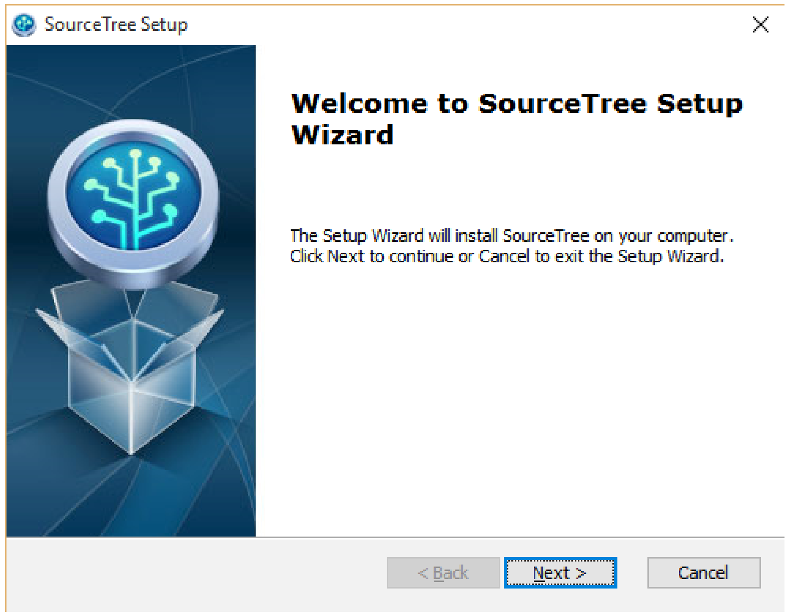
In this chapter, we will use Atlassian SourceTree, which is a free and open source client for Git:

1. Download SourceTree from `www.sourcetreeapp.com`, as shown in the following screenshot:

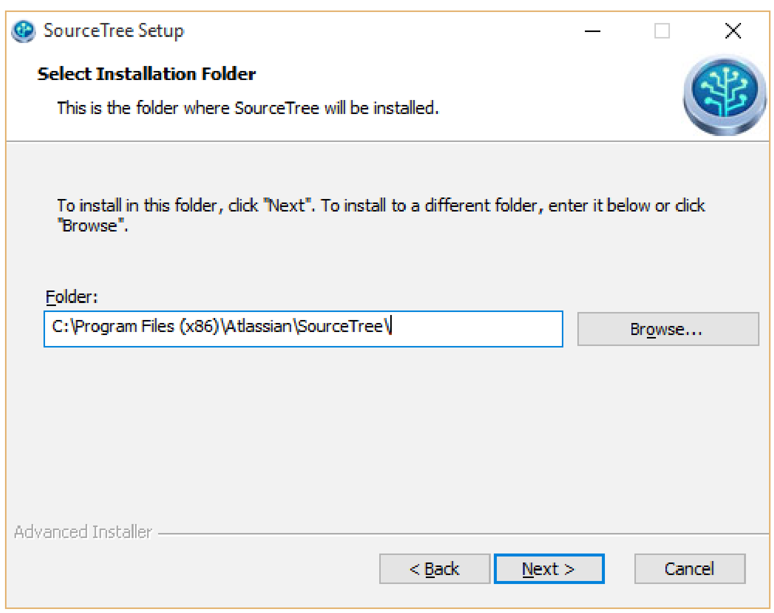


> There are a lot of open source clients available for Git. You are free to choose any one of them. Nevertheless, the basic Git operations are the same in all the tools. Git itself comes with a GUI that is minimalist in every sense.
>
> At the end of the installation, the software will prompt to install Git and Mercurial. Say no, as we have already installed Git.

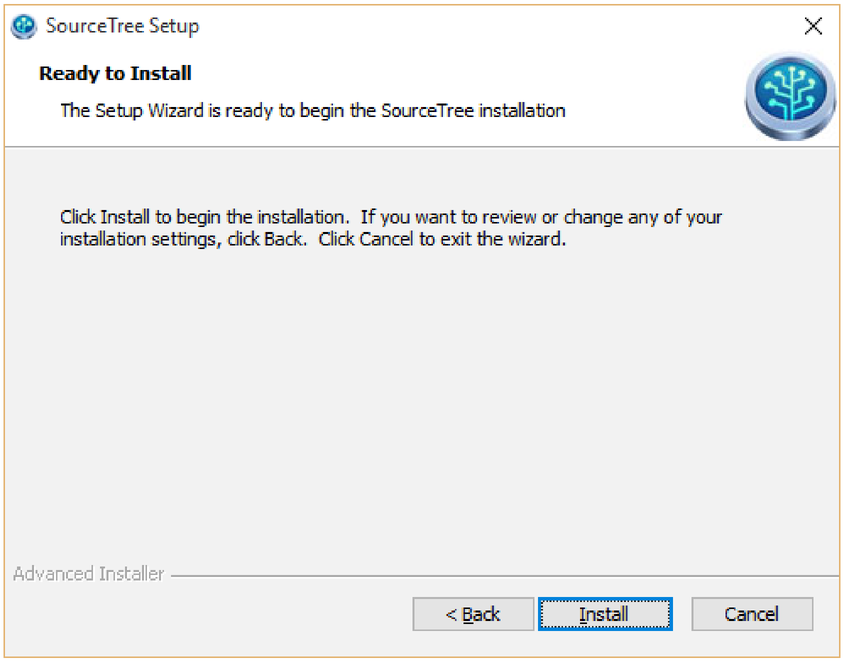
2. Begin the installation by double-clicking on the downloaded executable file.



3. Choose the installation directory `C:\Program Files(x86)\Atlassian\SourceTree\`. Click on the **Next** button to proceed.



4. Click on **Install** to begin the installation.



Creating a repository inside Git

We have now successfully installed Git and a Git client. Let's now create a repository in Git.

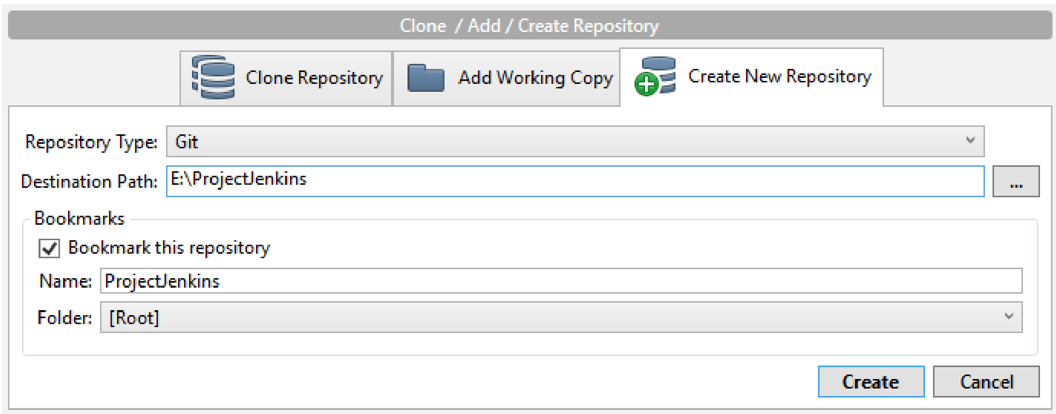
Using SourceTree

Git clients like SourceTree are gaining popularity among newcomers as they are intuitive and simple to understand. Let's create a Git repository using SourceTree:

1. Open SourceTree and click on the **Clone/New** button present at the top-left corner.
2. A window will pop up displaying three tabs **Clone Repository**, **Add Working Copy**, and **Create New Repository**.
3. Select the **Crete New Repository** tab and fill in the blanks as follows:
   - Specify **Repository Type** as `Git`.

  - Specify **Destination Path** as any local directory path on your Git server where you wish to store your version controlled files. For example, I have created a folder called `ProjectJenkins` inside the `E:\` drive on my Git server.

4. As you do that, by default, the Git repository will take the folder's name, which in my case is `ProjectJenkins`.
5. Once done, click on the **Create** button.



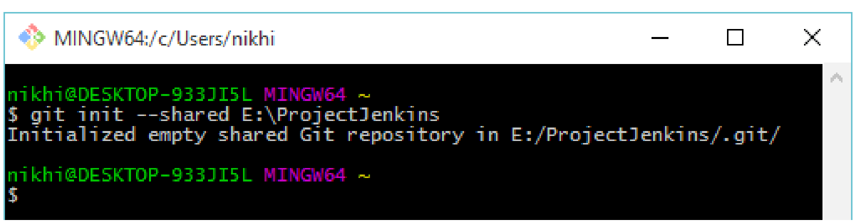
Using the Git commands

You can perform the same action from the command line:

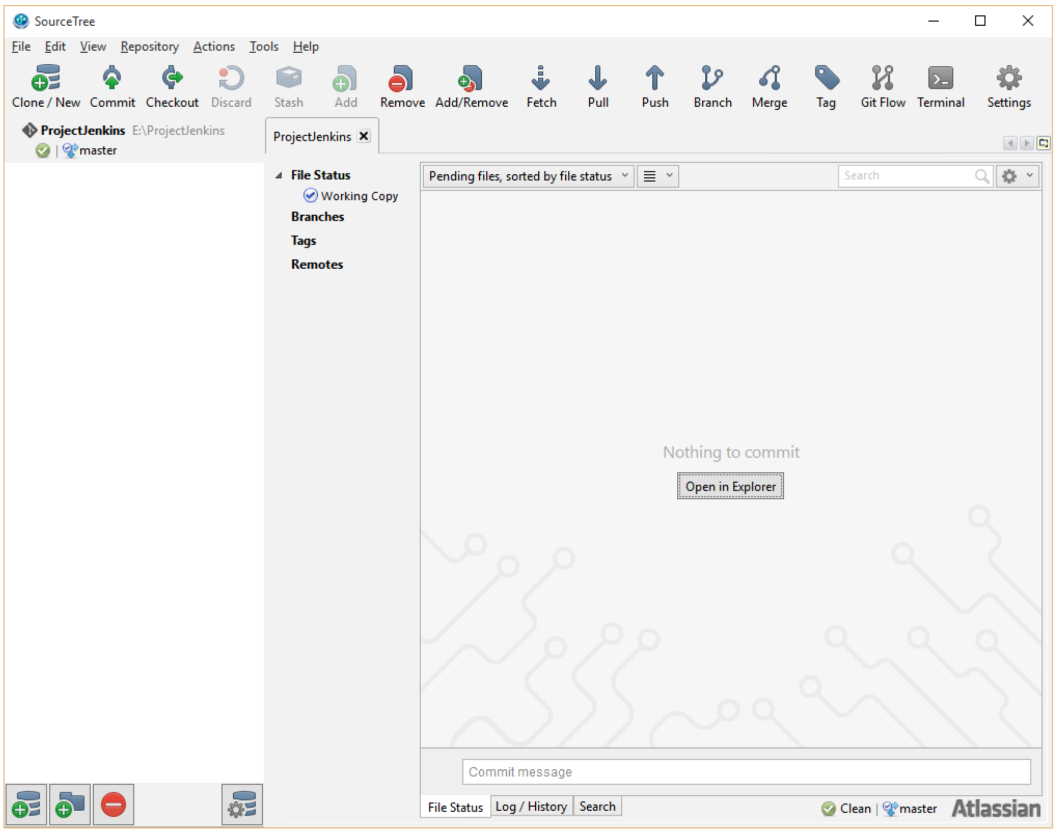
1. Open the Git bash console using the `Git-bash.exe`. It is present inside the directory `C:\Program Files\Git\`. A desktop shortcut also gets created while installing Git though.
2. Once you successfully open the Git bash console give the following command:

   ```
   git init --shared E:\ProjectJenkins
   ```

3. In my example, I have given the following command:



4. Coming back to the SourceTree dashboard, we can see `ProjectJenkins` created with one master branch. But right now, the repository is empty.



Uploading code to Git repository

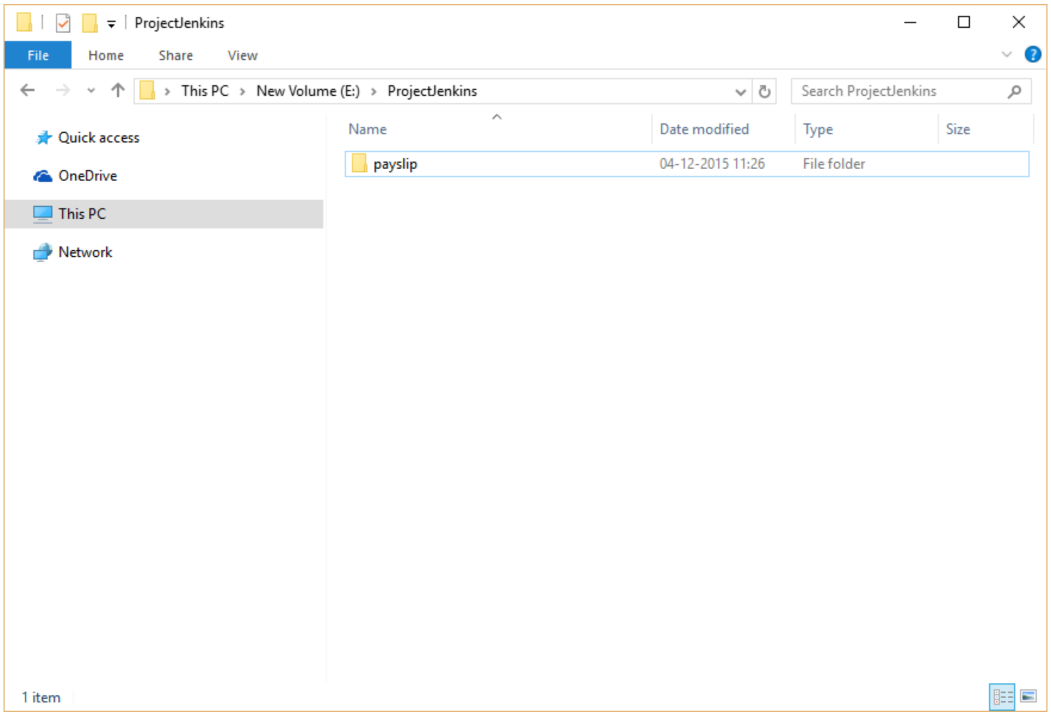
In this chapter, we will use an example code that is a simple Maven web app project. I have chosen a very unpretentious code as our main focus is to learn Continuous Integration, testing, delivery, and lots more.

Using SourceTree

Let's upload code to the Git repository using SourceTree:

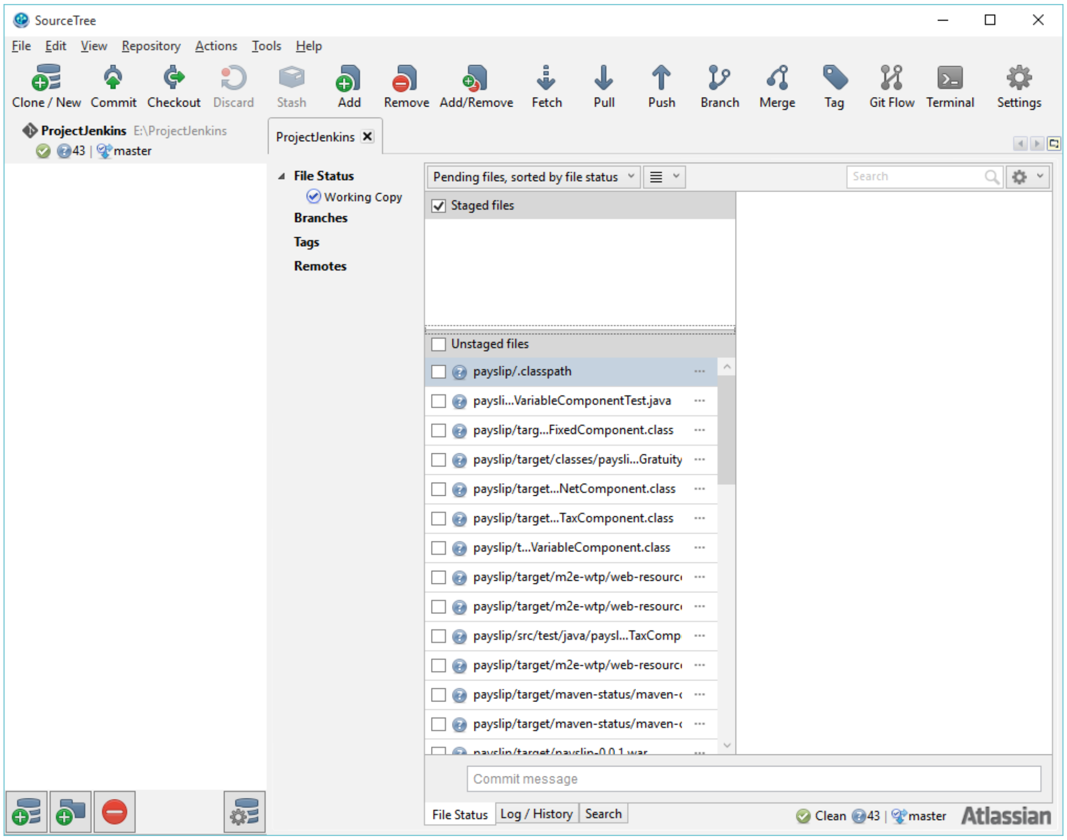
1. The code can be downloaded from the following GitHub repository: `https://github.com/nikhilpathania/ProjectJenkins`.

2. Download the `payslip` folder from the online repository and place it inside the Git repository's folder, as shown in the following screenshot:

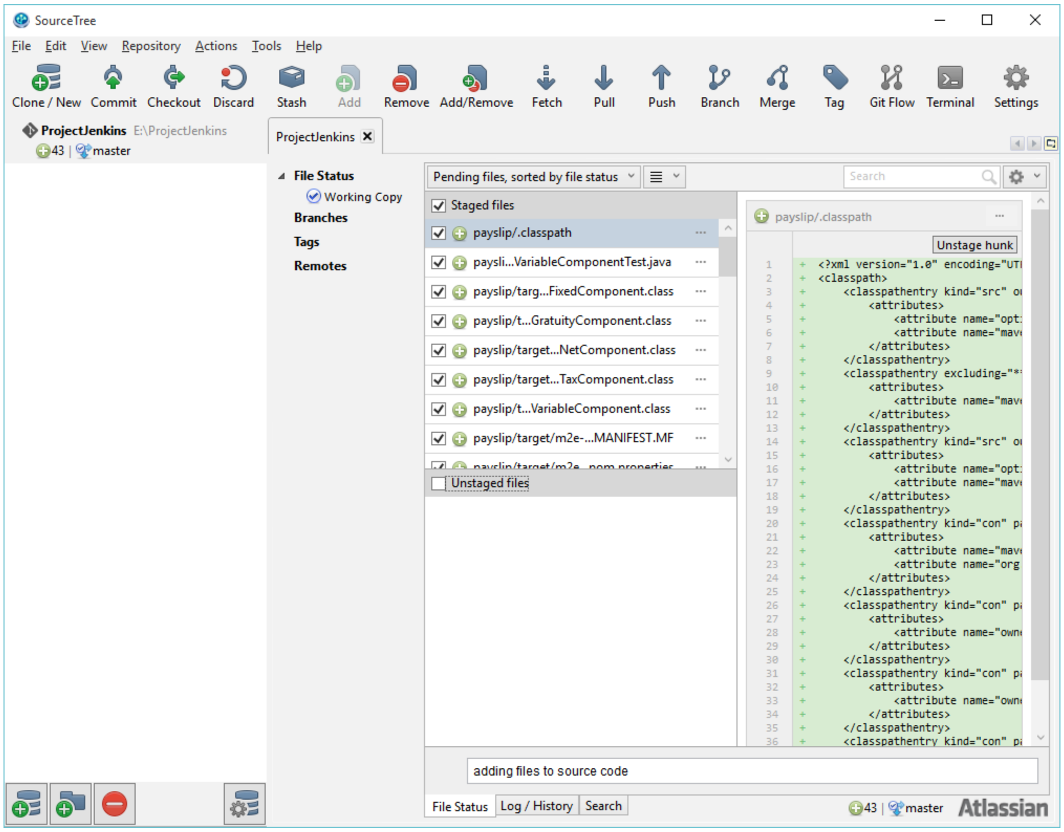


3. Open the SourceTree application and you will see the code reflected under the **Unstaged files** section.

4. If you don't see the code listed under **Unstaged files**, press *F5* or click on the **Add/Remove** button from the menu to refresh the view.

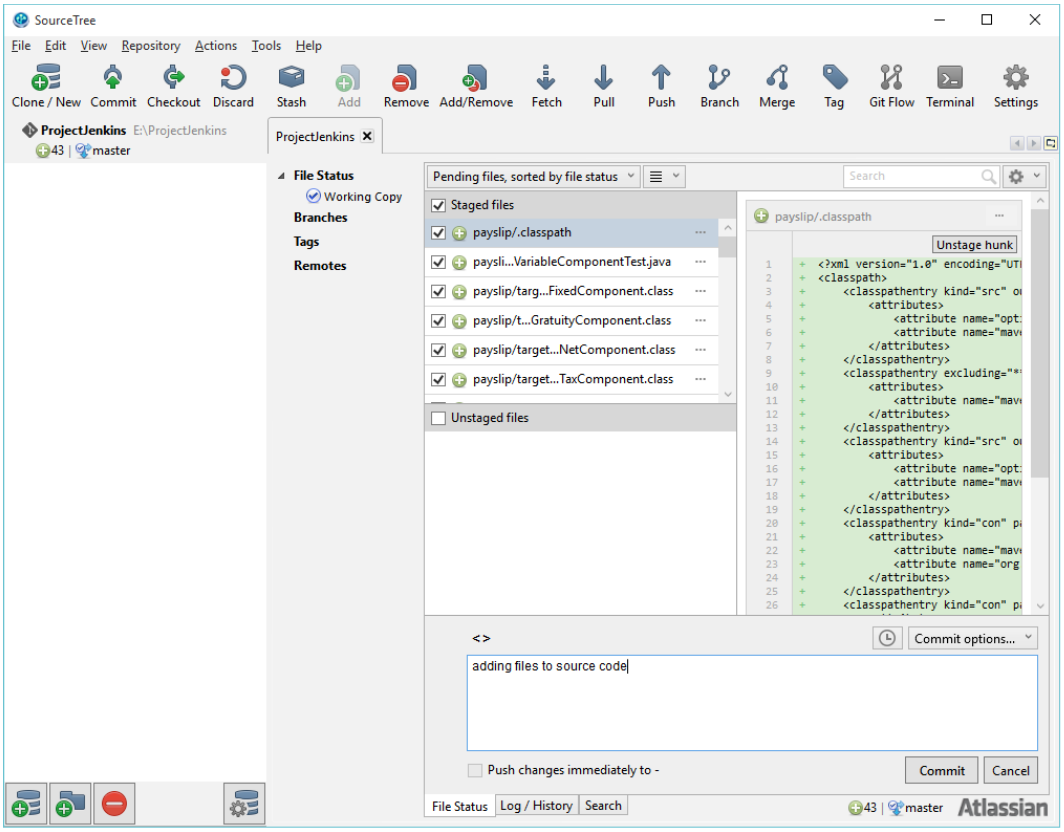


5. Click on the **Staged files** checkbox to stage **Unstaged files**. The entire code will be staged, as shown in the following screenshot:



6. Commit the code by clicking on the **Commit** button from the menu bar. A small section opens up at the bottom-right corner.

7. Add some comments and click on the **Commit** button.



Using the Git commands

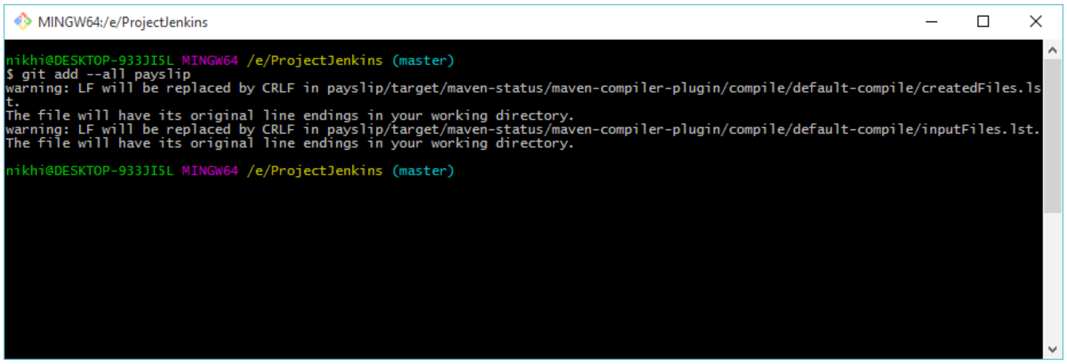
To perform the same action using the command line, give the following command in the Git bash console. Make sure the master branch is in the checked out state.

1. Use the following command to go to the Git repository:

```
cd e:/ProjectJenkins
```

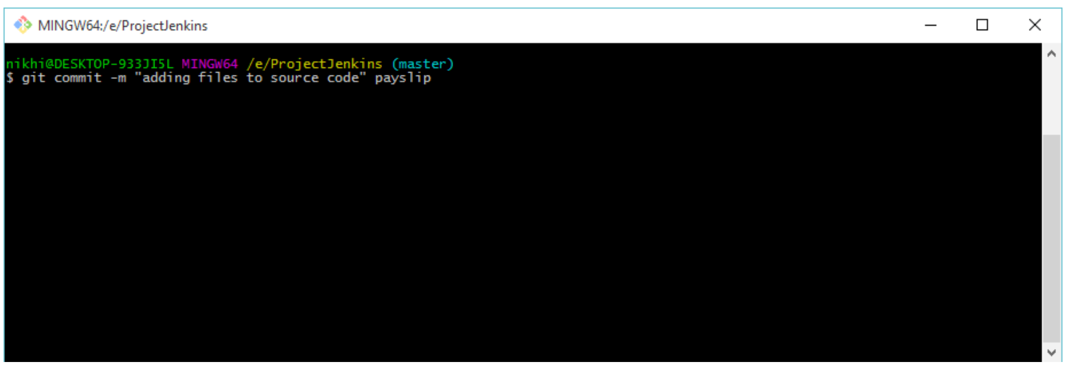
2. Use the following command to add the code:

```
Git add --all payslip
```

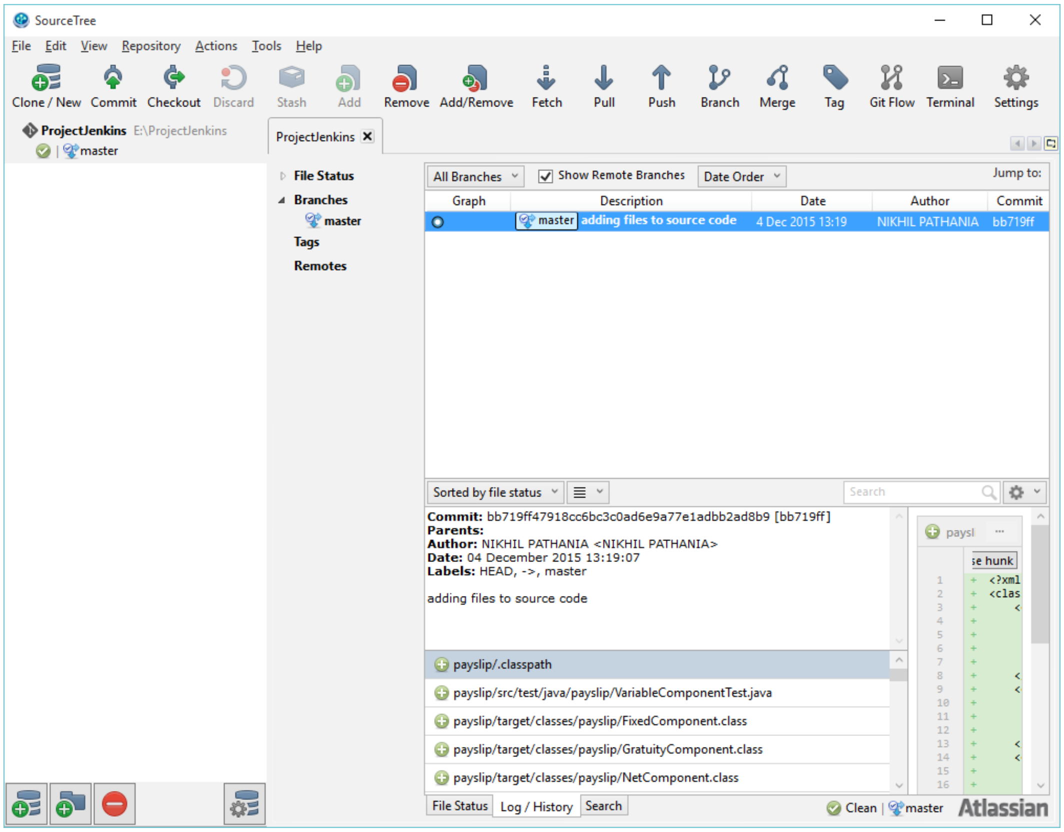


3. Now, use the following command to commit the changes to the source control:

```
git commit -m "adding files to source code" payslip
```



4. In the SourceTree dashboard, we can see the code has been added to our master branch inside the Git repository `ProjectJenkins`, as shown in the following screenshot:



Configuring branches in Git

Now that we have added the code to our Git repository, let's create some branches as discussed in our CI design.

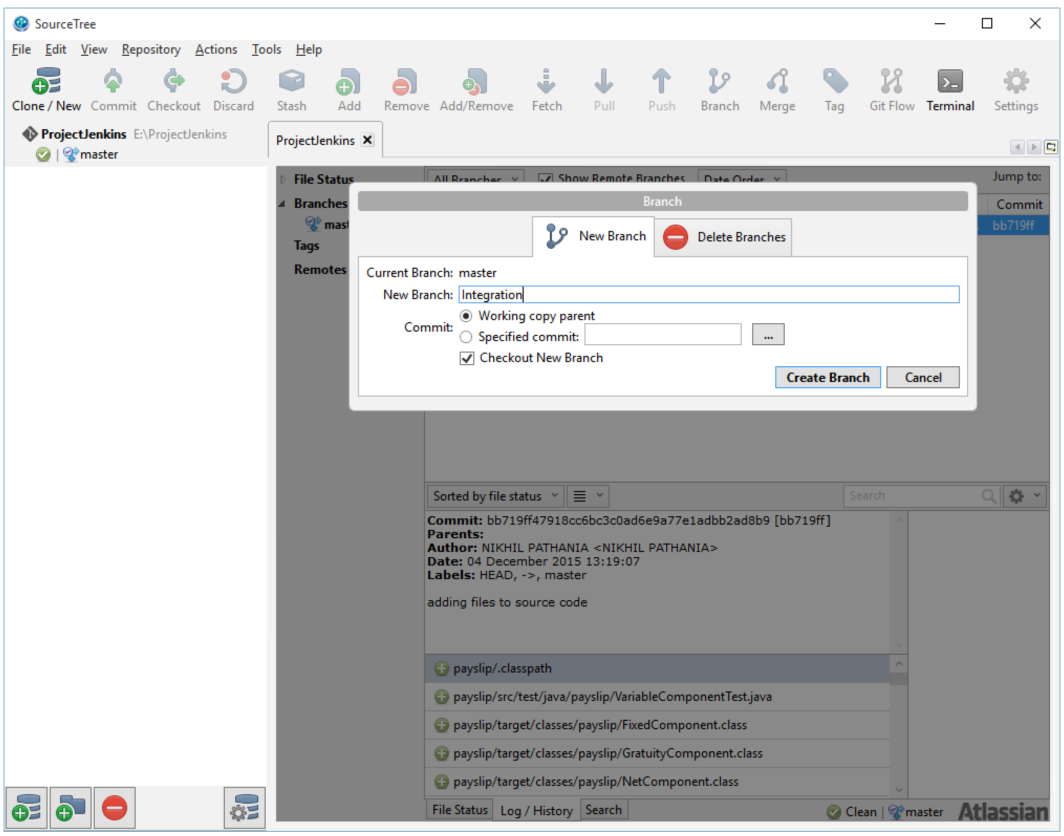
We will create an integration branch out of the master branch and two feature branches out of the integration branch. All the development will happen on the respective feature branches, and all the integration will happen on the integration branch.

The master branch will remain neat and clean and only code that has been built and tested thoroughly will reside on it.

Using SourceTree

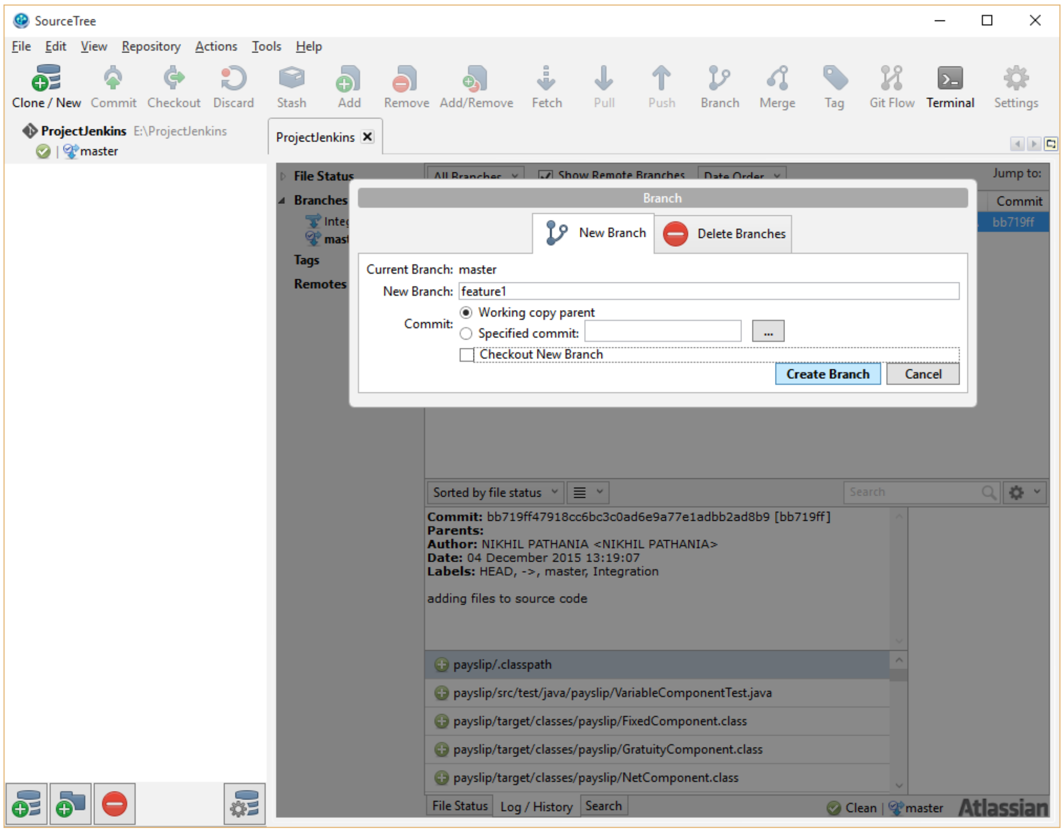
Let's create branches in the Git repository using SourceTree:

1. Select the master branch and click on the **Branch** button.
2. A small window with the two tabs **New Branch** and **Delete Branches** will pop up.
3. Select the **New Branch** tab and fill the value `Integration` in the **New Branch** field.
4. Select the **Checkout New Branch** option. Once done, click on the **Create Branch** button.



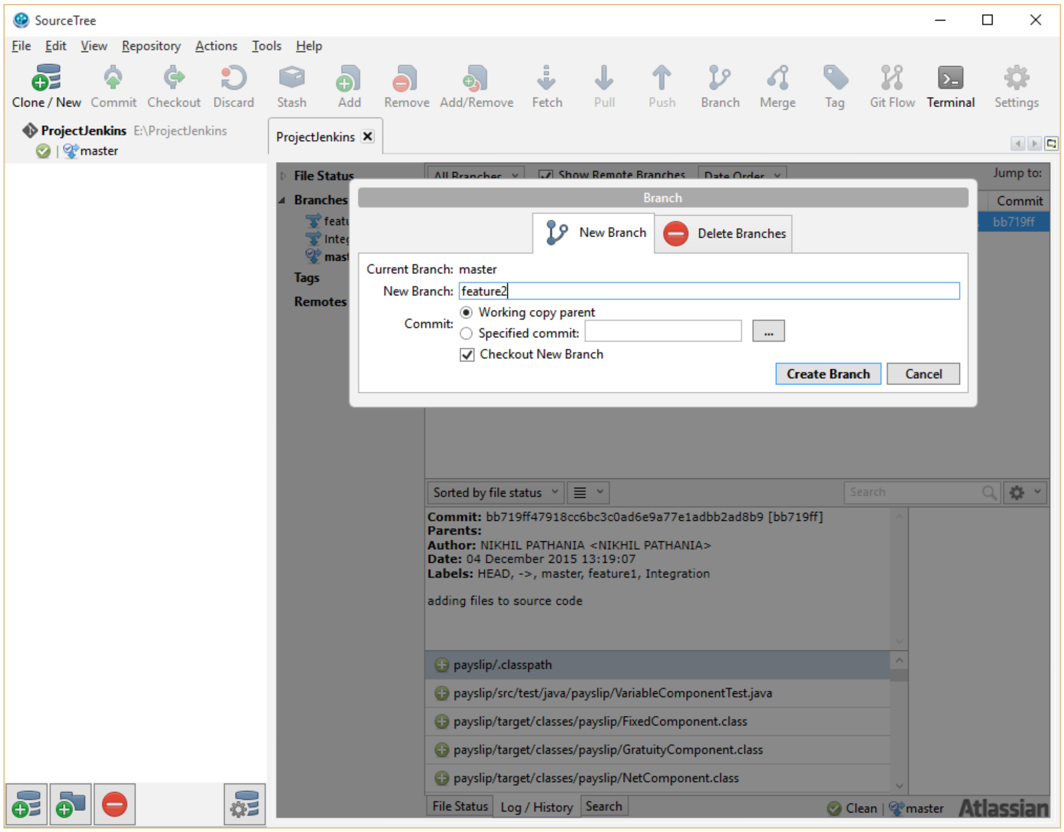
5. The integration branch will be created and will be in the checkout state.
6. Now, we will create two branches out of the integration branch: `feature1` and `feature2`.

7. The integration branch is already selected and checked out. So, simply click on the **Branch** button again.
8. Create a new branch named `feature1` following the same process as we did while creating the integration branch.
9. Uncheck the **Checkout New Branch** option. We do not want to switch to our new branch by checking it out after its creation.



10. Create the `feature2` branch in the same fashion by clicking on the **Branch** button again.

11. This time, check the **Checkout New Branch** option.



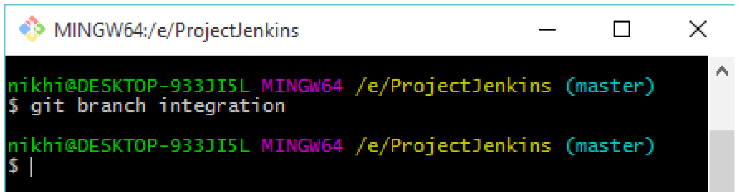
Using the Git commands

Follow these steps to perform the same action using the command line:

1. Open Git bash console and type to following command to create the integration branch:

```
cd e:/ProjectJenkins
git branch integration
```

2. You will get the following output:



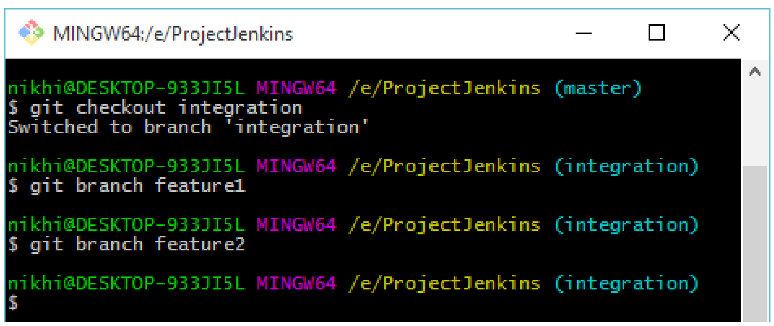
3. In order to create the feature branches, first check out the integration branch with the following command:

   ```
   git checkout integration
   ```

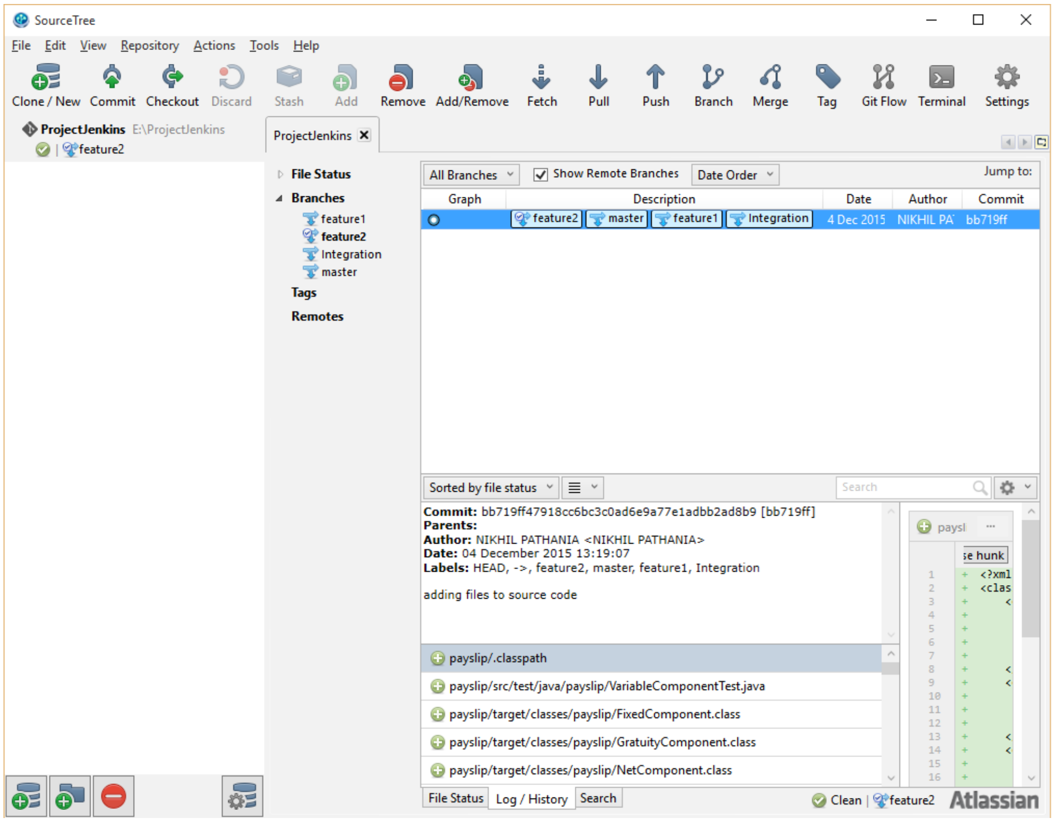
4. Then, use the following command to create the feature branches one by one:

   ```
   git branch feature1
   ```

   ```
   git branch feature2
   ```



5. In the SourceTree dashboard, we can see all the branches we want with the `feature2` branch checked out, as shown in the following screenshot:



You can also see that all the branches are at the same level, which means all the branches currently have the same version of the code without any difference.

Git cheat sheet

The following table contains the list of Git commands used in the current chapter:

Branches	
git branch	List all of the branches in your repository.
git branch <branch>	Create a new branch.
git checkout <branch>	Create and check out a new branch named <branch>.
git merge <branch>	Merge <branch> into the current branch.

Repository	
git init <directory>	Create empty Git repository in the specified directory.
git add <directory>	Stage all changes in <directory> for the next commit. Replace <directory> with <file> to change a specific file.
git status	List which files are staged, unstaged, and untracked.
git commit -m "<message>"	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
Rebase	
git rebase -i <branch>	Interactively rebase the current branch onto another branch named <branch>.

> You can take a look at all the Git commands at the following link https://git-scm.com/docs.

Configuring Jenkins

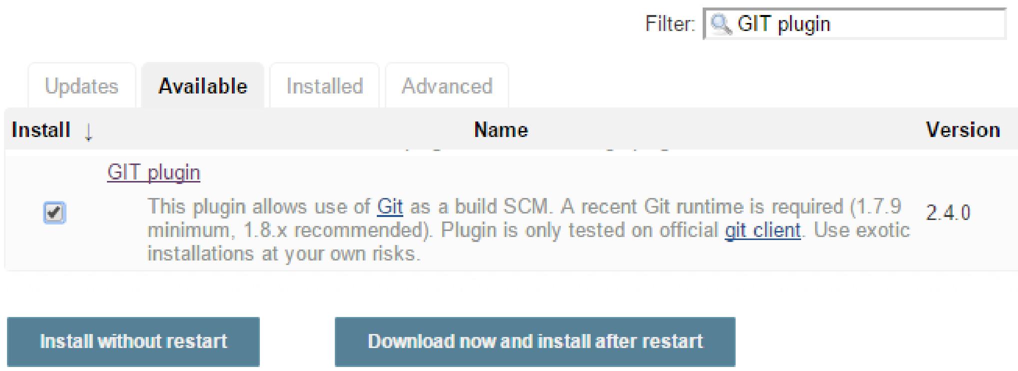
Notification and reporting are an important part of Continuous Integration. Therefore, we need an advanced e-mail notification plugin. We will also need a plugin to make Jenkins interact with Git.

Along with these plugins, we will also need to install and configure Java and Maven inside Jenkins. This will enable Jenkins to perform builds.

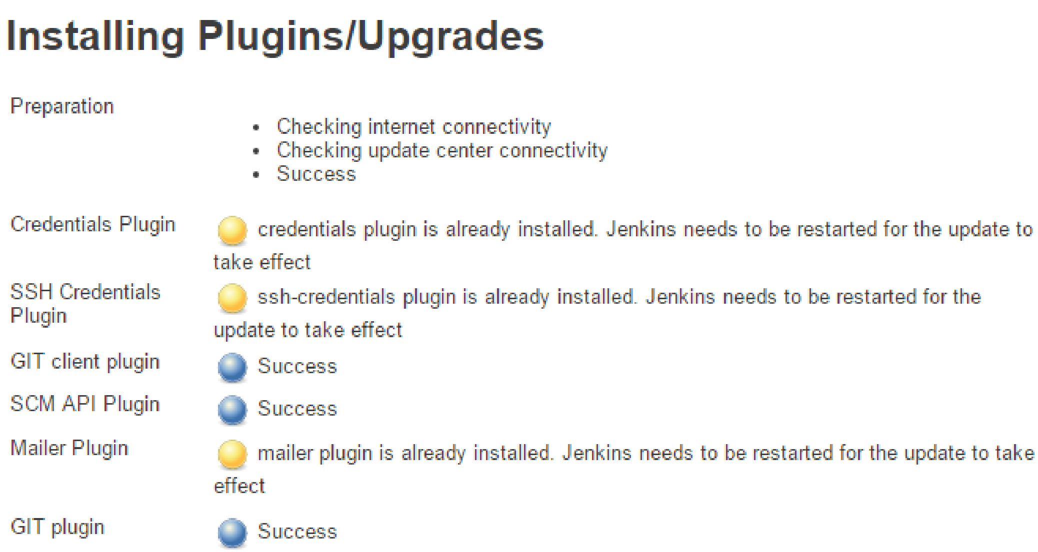
Installing the Git plugin

In order to integrate Git with Jenkins, we need to install the GIT plugin. The steps are as follows:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. This will take you to the **Manage Jenkins** page. From here, click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `GIT plugin` in the search box. Select **GIT plugin** from the list and click on the **Install without restart** button.

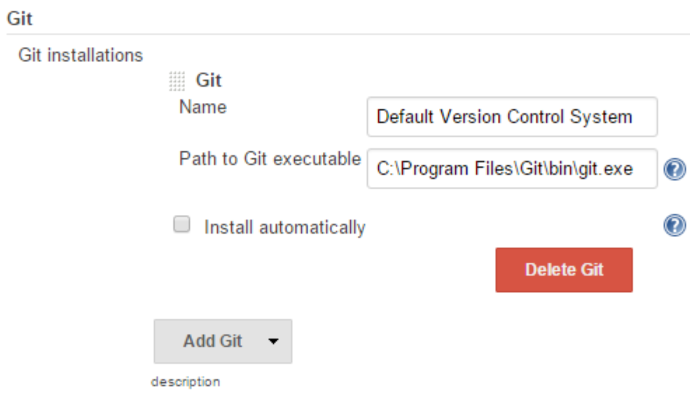


4. The download and installation starts automatically. You can see the GIT plugin has a lot of dependencies that get downloaded and installed.



5. Upon successful installation of the GIT plugin, go to the **Configure System** link from the **Manage Jenkins** page.
6. Scroll down until you see the **Git** section and fill the blanks as shown in the following screenshot.
7. You can name your Git installation whatever you want. Point the **Path to Git executable** to the location where you have installed Git. In our example, it's `C:\Program Files\Git\bin\git.exe`.

8. You can add as many Git servers as you want by clicking on the **Add Git** button.



[If you have more than one Git server to choose from, provide a different name for each Git instance.]

Installing and configuring JDK

First, download and install Java on your Jenkins server, which I guess you might have already done as part of the Apache Tomcat server installation in the previous chapter. If not, then simply download the latest Java JDK from the internet and install it.

Setting the Java environment variables

Let's configure the Java environment variable **JAVA_HOME:**

1. After installing Java JDK, make sure to configure **JAVA_HOME** using the following command:

```
setx JAVA_HOME " C:\Program Files\Java\jdk1.8.0_60" /M
```

2. To check the home directory of Java, use the following command:

```
echo %JAVA_HOME%
```

3. You should see the following output:

```
C:\Program Files\Java\jdk1.8.0_60
```

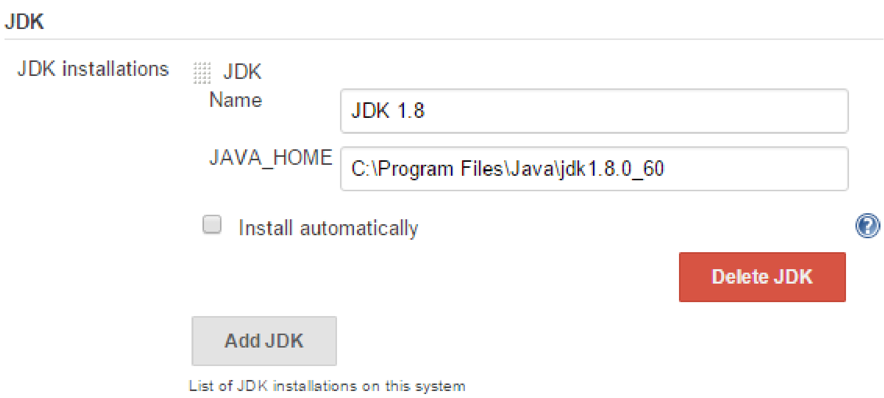
4. Also, add the Java executable path to the system `PATH` variable using the following command:

```
setx PATH "%PATH%\;C:\Program Files\Java\jdk1.8.0_60\bin" /M
```

Configuring JDK inside Jenkins

You have installed Java and configured the system variables. Now, let Jenkins know about the JDK installation:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. On the **Manage Jenkins** page, click on the **Configure System** link.
3. Scroll down until you see the JDK section. Give your JDK installation a name. Also, assign the **JAVA_HOME** value to the JDK installation path, as shown in the following screenshot:



> You can configure as many JDK instances as you want. Provide a unique name to each JDK installation.

Installing and configuring Maven

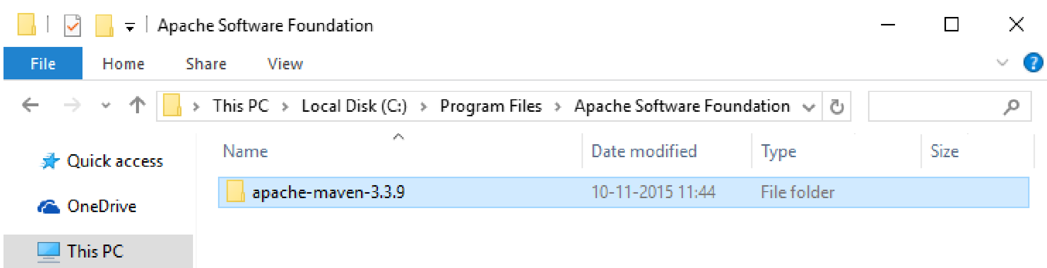
The example code used in this chapter is written in Java. Hence, we need Maven to build it. In this section, we will see how to install and configure Maven on the Jenkins master server.

Installing Maven

Let's see how to install Maven on the Jenkins Master server first:

1. Download Maven from the following link: `https://maven.apache.org/download.cgi`.

2. Extract the downloaded zip file to `C:\Program Files\Apache Software Foundation\`.



Setting the Maven environment variables

To set the Maven environment variables, perform the following steps:

1. Set the Maven `M2_HOME`, `M2`, and `MAVEN_OPTS` variables using the following commands:

```
setx M2_HOME "C:\Program Files\Apache Software Foundation\apache-maven-3.3.9" /M
setx M2 "%M2_HOME%\bin" /M
setx MAVEN_OPTS "-Xms256m -Xmx512m" /M
```

2. To check the variables, use the following commands:

```
echo %M2_HOME%
echo %M2%
echo %MAVEN_OPTS%
```

3. Also, add the Maven `bin` directory location to the system path using the following command:

```
setx PATH "%PATH%;%M2%" /M
```

4. To check if Maven has been installed properly, use the following command:

```
mvn -version
```

5. You should see the following output:

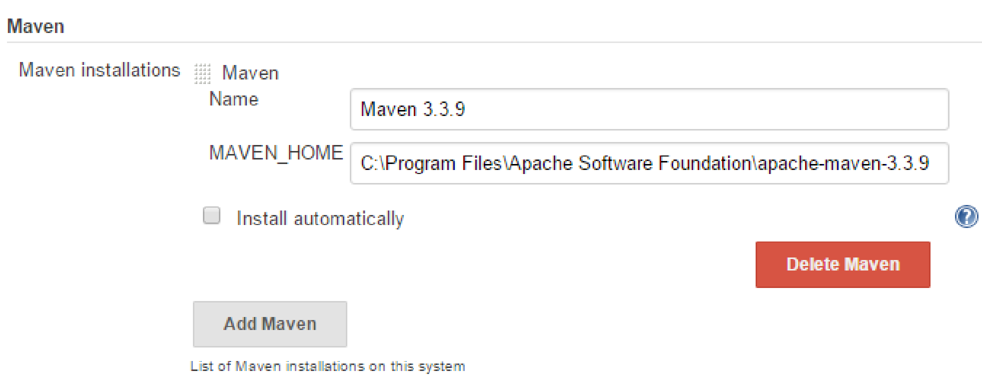
```
Apache Maven 3.3.9 (bb52d8502b132ec0a5a3f4c09453c07478323dc5; 2015-11-10T22:11:47+05:30)
Maven home: C:\Program Files\Apache Software Foundation\apache-maven-3.3.9
Java version: 1.8.0_60, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_60\jre
```

```
Default locale: en_IN, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family:
"dos"
```

Configuring Maven inside Jenkins

We have successfully installed Maven. Now, let us see how to connect it with Jenkins.

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. On the **Manage Jenkins** page, click on the **Configure System** link.
3. Scroll down until you see the **Maven** section.
4. Assign the **MAVEN_HOME** field to the Maven installation directory. Name your Maven installation by giving it a unique name.



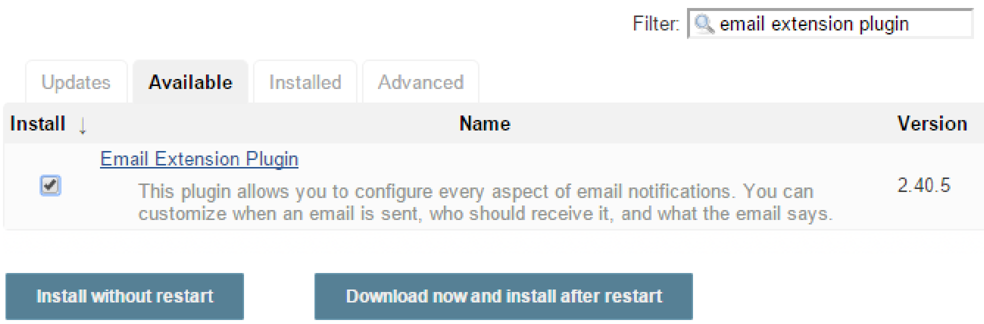
> We can configure as many Maven instances as we want. Provide a unique name for each Maven instance.

Installing the e-mail extension plugin

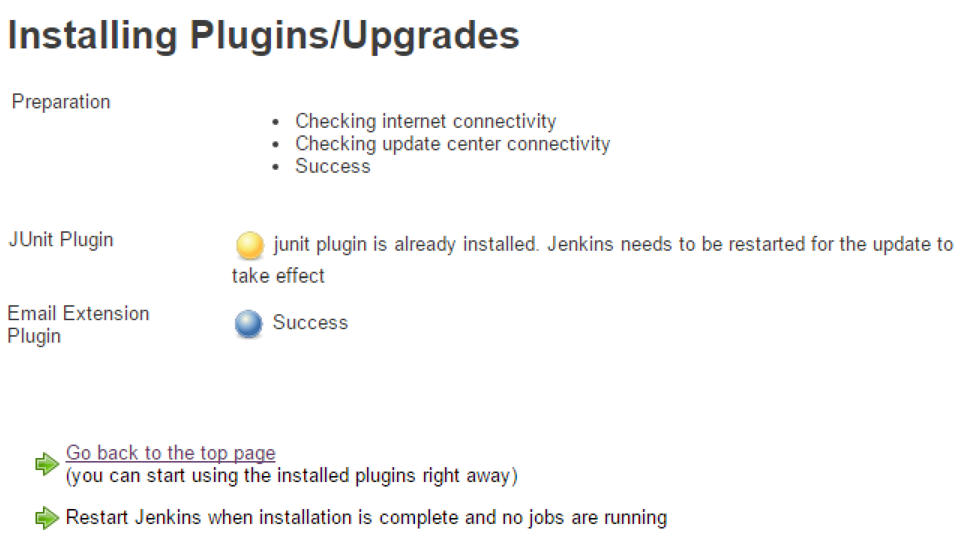
The e-mail notification facility that comes with the Jenkins is not enough. We need a more advanced version of e-mail notification such as the one provided by **Email Extension** plugin. To do this, perform the following steps:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `email extension plugin` in the search box.

4. Select **Email Extension Plugin** from the list and click on the **Install without restart** button.



5. The plugin will install as shown in the following screenshot:



The Jenkins pipeline to poll the feature branch

In the following section, we will see how to create both the Jenkins jobs that are part of the pipeline to poll the feature branch. This pipeline contains two Jenkins jobs.

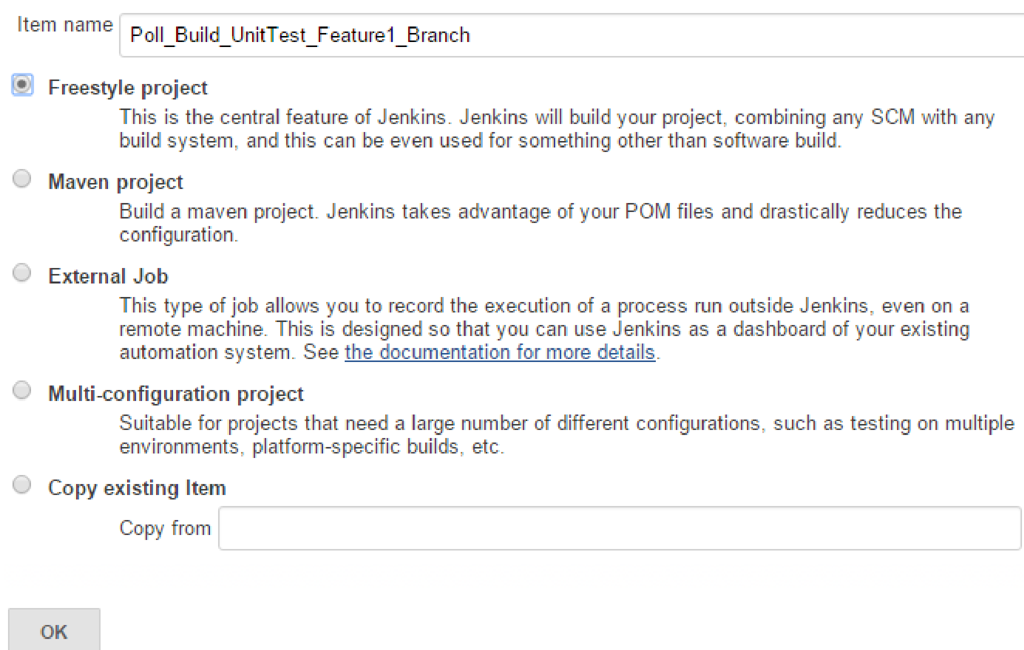
Creating a Jenkins job to poll, build, and unit test code on the feature1 branch

The first Jenkins job from the pipeline to poll the feature branch does the following tasks:

- It polls the feature branch for changes at regular intervals
- It performs a build on the modified code
- It executes unit tests

Let's start creating the first Jenkins job. I assume you are logged in to Jenkins as an admin and have privileges to create and modify jobs. The steps are as follows:

1. From the Jenkins Dashboard, click on the **New Item** link.
2. Name your new Jenkins job `Poll_Build_UnitTest_Feature1_Branch`.
3. Select the type of job as **Freestyle project** and click on **OK** to proceed.

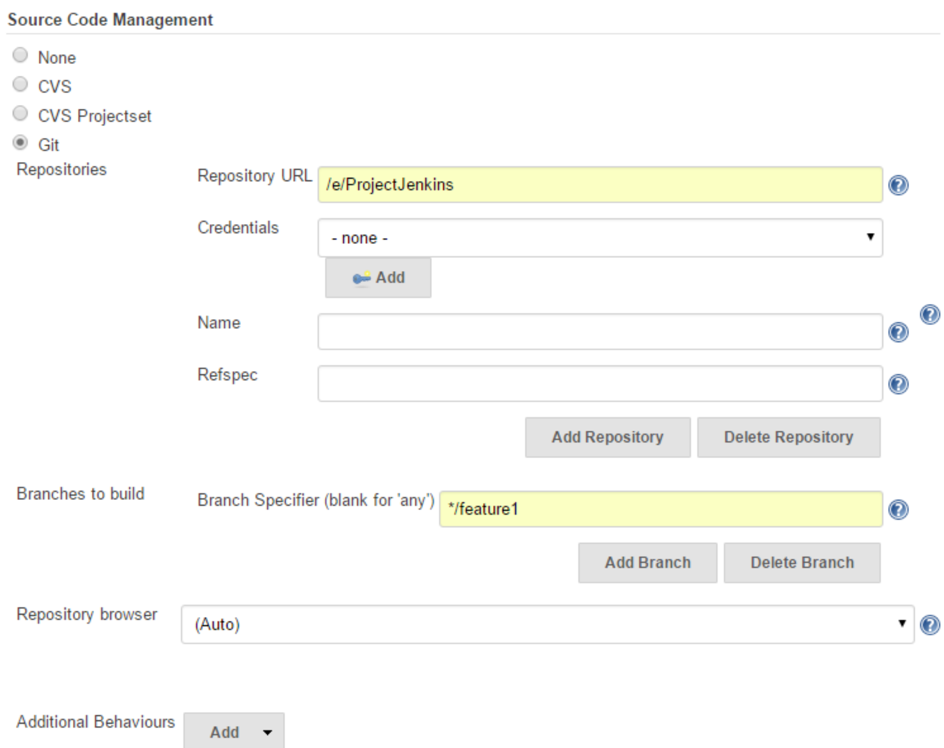


4. Add a meaningful description about the job in the **Description** section.

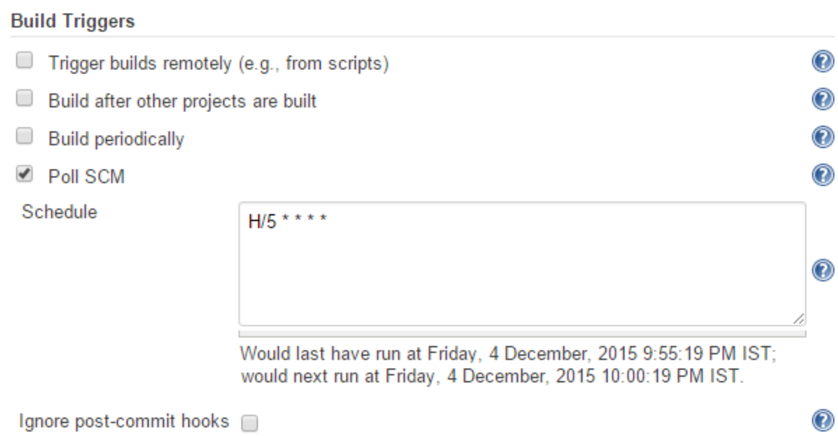
Polling version control system using Jenkins

This is a critical step where we connect Jenkins with the Version Control System. This configuration enables Jenkins to poll the correct branch inside Git and download the modified code:

1. Scroll down to the **Source Code Management** section and select the **Git** option.
2. Fill the blanks as follows:
    - **Repository URL**: Specify the location of the Git repository. It can be a GitHub repository or a repository on a Git server. In our case it's `/e/ProjectJenkins`, as the Jenkins server and the Git server is on the same machine.
    - Add `*/feature1` in the **Branch to build** section, since we want our Jenkins job to poll the `feature1` branch.
3. Leave rest of the fields at their default values.



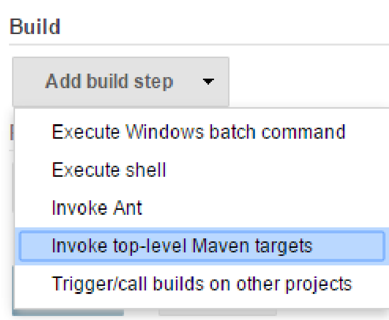
4. Scroll down to the **Build Triggers** section.
5. Select **Poll SCM** and type `H/5 * * * *` in the **Schedule** field. We want our Jenkins job to poll the feature branch every 5 minutes. However, feel free to choose the polling duration as you wish depending on your requirements.



Compiling and unit testing the code on the feature branch

This is an important step in which we tell Jenkins to build the modified code that was downloaded from Git. We will use Maven commands to build our Java code.

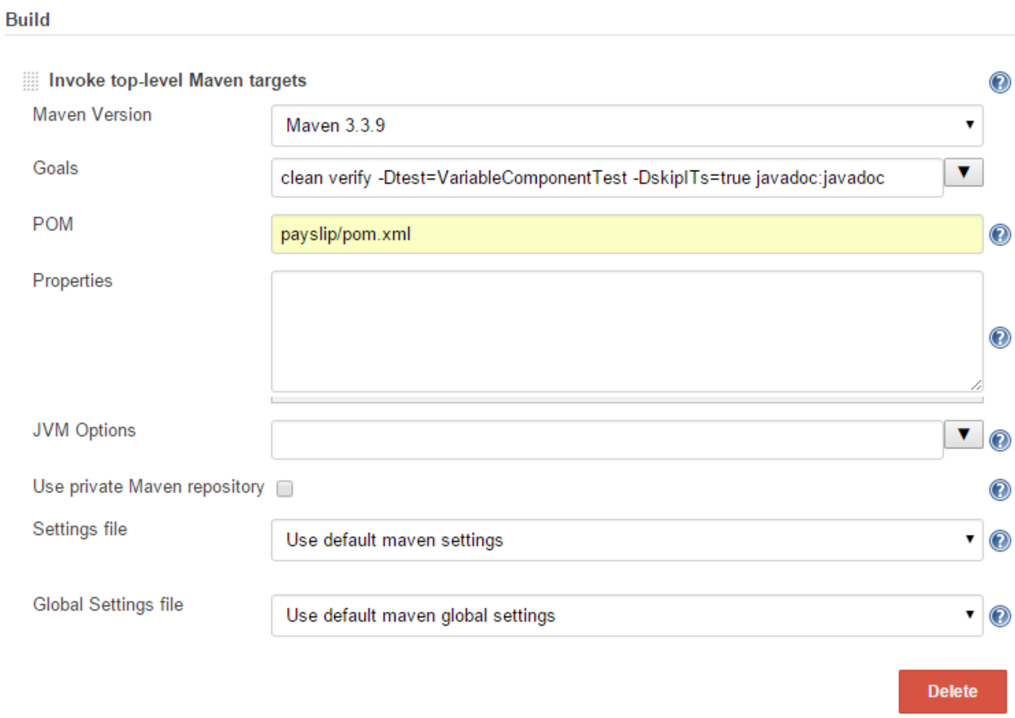
1. Scroll down to the **Build** section.
2. Click on the **Add build step** button and select **Invoke top-level Maven targets** from the available options.



3. Configure the fields as shown in the following screenshot:
    - Set **Maven Version** as `Maven 3.3.9`. Remember this is what we configured on the **Configure System** page in the **Maven** section. If we had configured more than one Maven, we would have a choice here.
    - Type the following line in the **Goals** section:

      ```
      clean verify -Dtest=VariableComponentTest -DskipITs=true
      javadoc:Javadoc
      ```

    - Type `payslip/pom.xml` in the **POM** field. This tells Jenkins the location of `pom.xml` in the downloaded code.

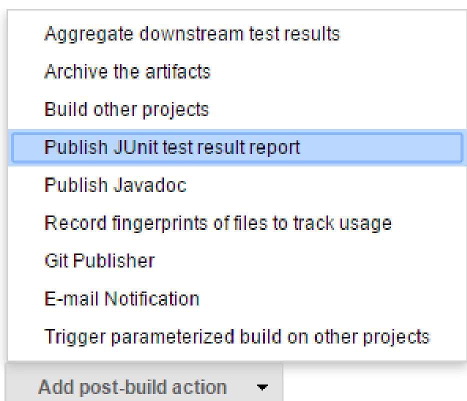


4. Let's see the Maven command inside the **Goals** field in detail:
    - The `clean` command will clean any old built files
    - The `-Dtest=VariableComponentTest` command will execute a unit test named `VariableComponentTest.class`
    - The `-DskipITs=true` command will skip the integration test, if any, as we do not need them to execute at this point
    - The `javadoc:javadoc` command will tell Maven to generate Java documentations

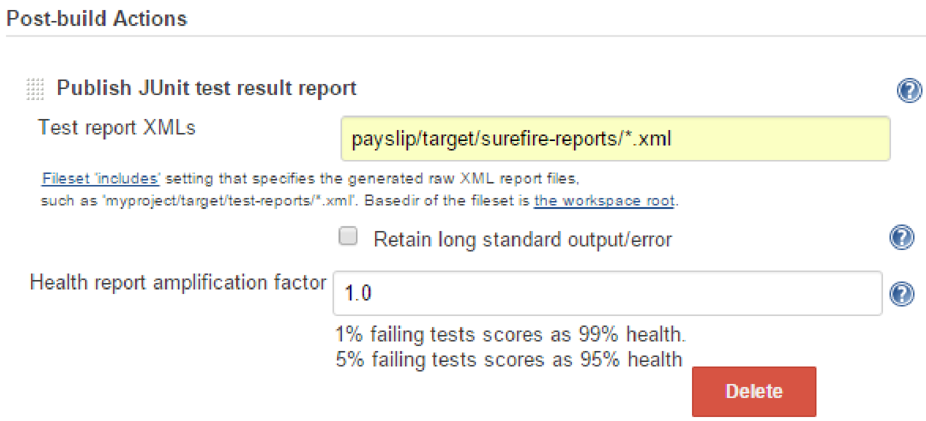
Publishing unit test results

Publishing unit test results falls under post build actions. In this section we configure the Jenkins job to publish JUnit test results:

1. Scroll down to the **Post build Actions** section.
2. Click on the **Add post-build action** button and select **Publish JUnit test result report**, as shown in the following screenshot:



3. Under the **Test report XMLs** field, add `payslip/target/surefire-reports/*.xml`. This is the location where the unit test reports will be generated once the code has been built and unit tested.

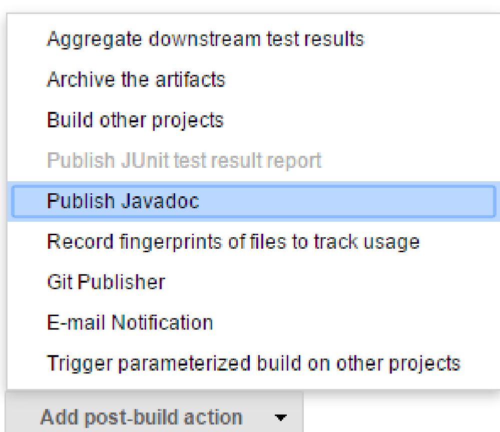


> Jenkins will access all the `*.xml` files present in the `payslip/target/surefire-reports` directory and publish the report. We will shortly see this when we run this Jenkins job.

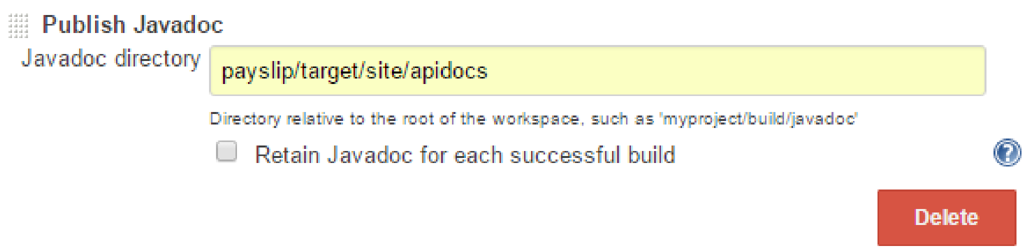
Publishing Javadoc

The steps to publish Javadoc are:

1. Once again, click on the **Add post-build action** button. This time, select **Publish Javadoc**.



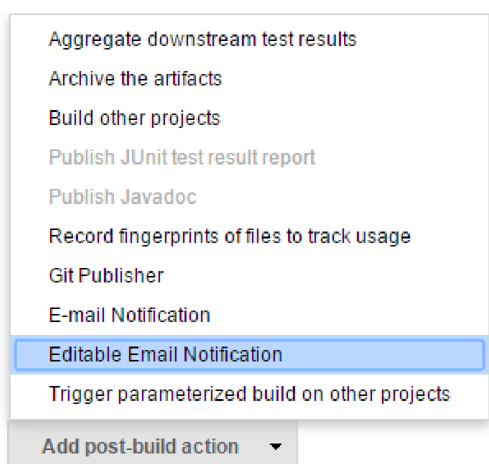
2. Add the path `payslip/target/site/apidocs` in the **Javadoc directory** field, as shown in the following screenshot:



Configuring advanced e-mail notification

Notification forms are an important part of CI. In this section, we will configure the Jenkins job to send e-mail notifications based on few conditions. Let's see the steps in detail:

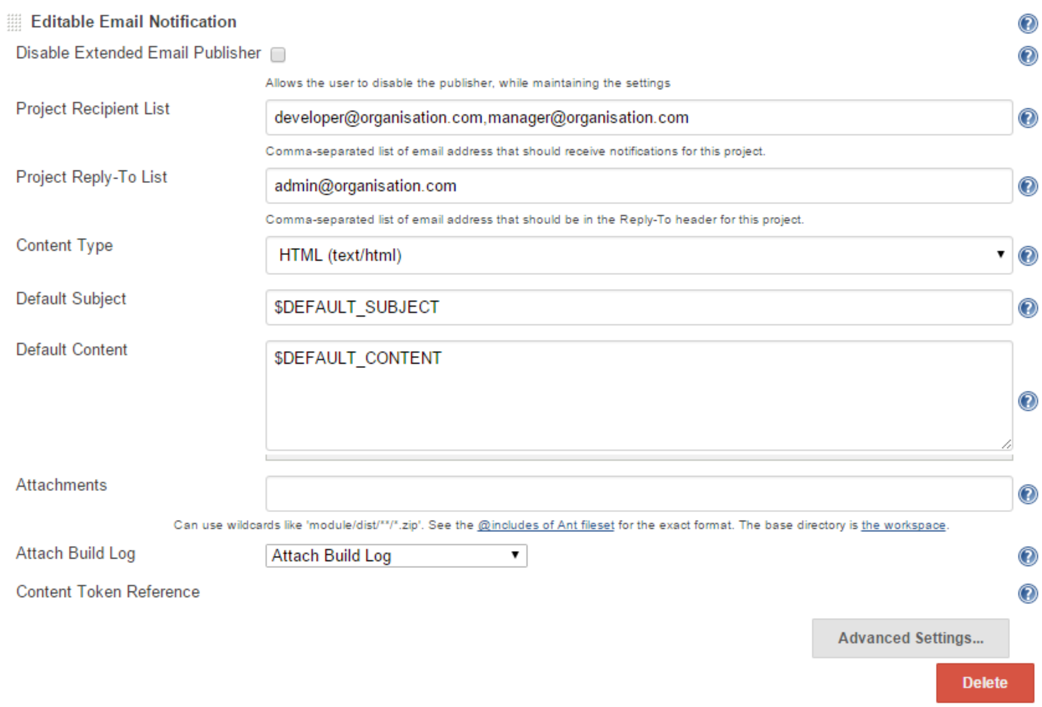
1. Click on the **Add post-build action** button and select **Editable Email Notification**, as shown in the following screenshot:



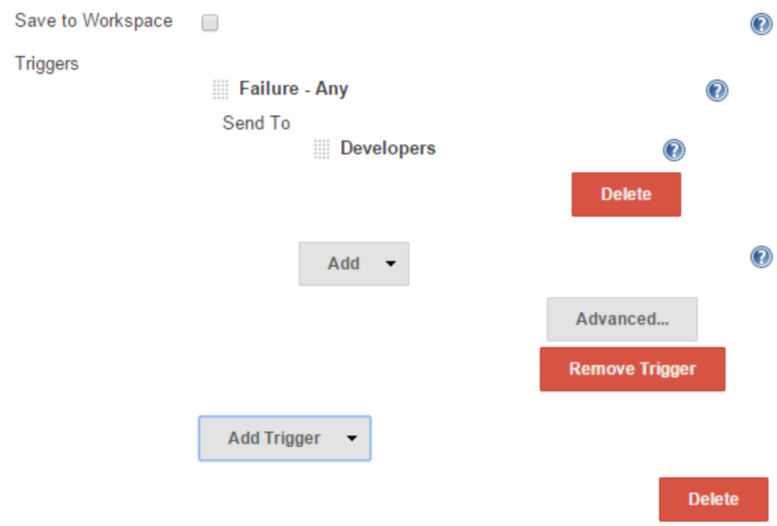
2. Configure **Editable Email Notification** as follows:
    - Under **Project Recipient List**, add the e-mail IDs separated by a comma. You can add anyone who you think should be notified for build and unit test success/failure.

  - You can add the e-mail ID of the Jenkins administrator under **Project Reply-To List**.
  - Select **Content Type** as **HTML (text/html)**.

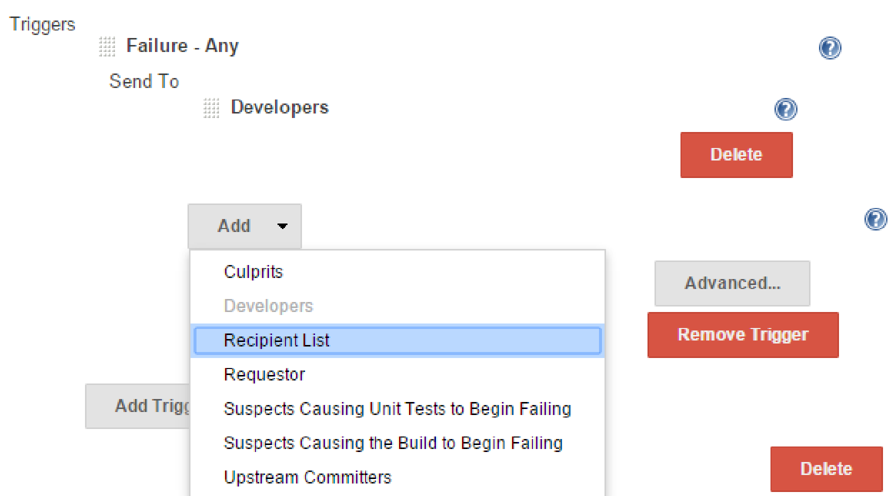
3. Leave all the rest of the options at their default values.



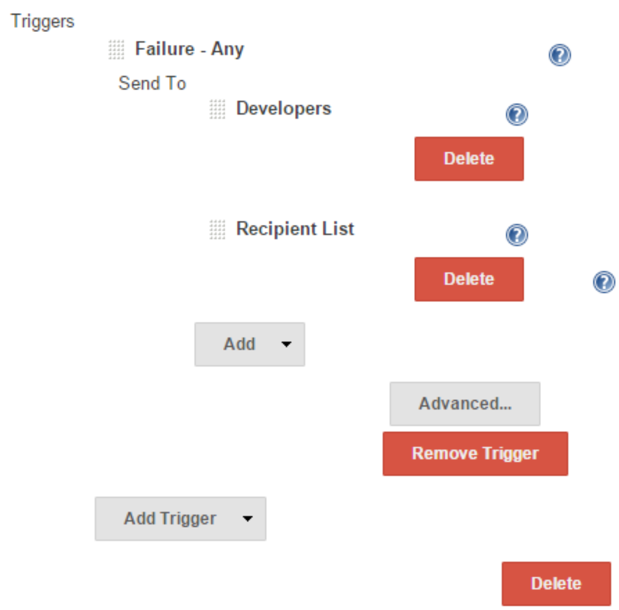
4. Now, click on the **Advanced Settings…** button.
5. By default, there is a trigger named **Failure – Any** that sends an e-mail notification in the event of a failure (any kind of failure).
6. By default, the **Send To** option is set to **Developers**.



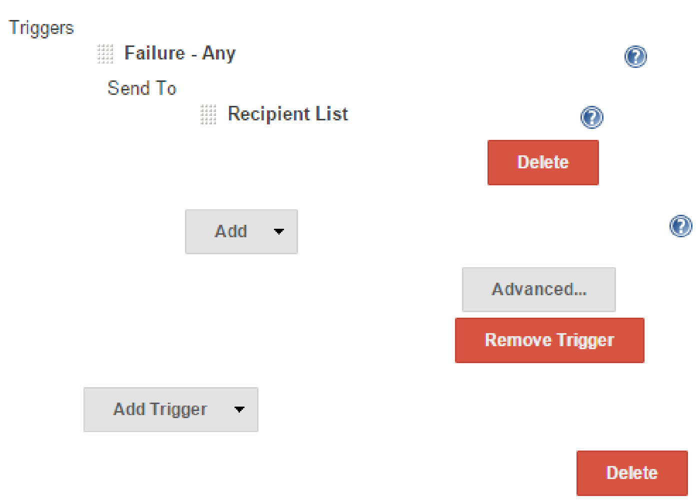
7. But we don't want that; we have already defined whom to send e-mails to. Therefore, click on the **Add** button and select the **Recipient List** option, as shown in the following screenshot:



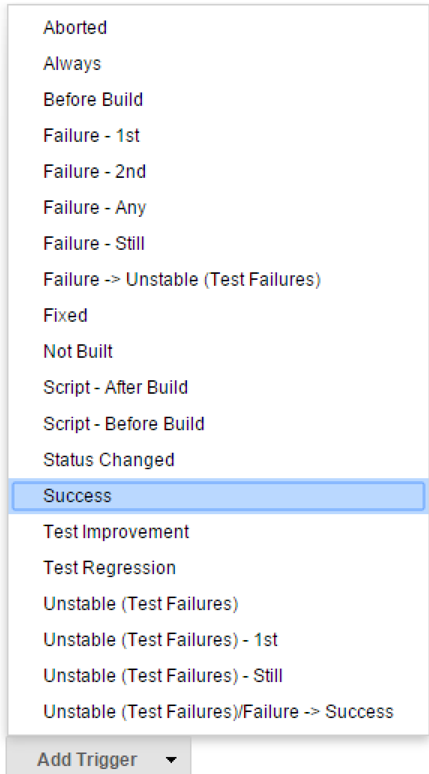
8. The result will look something like the following screenshot:



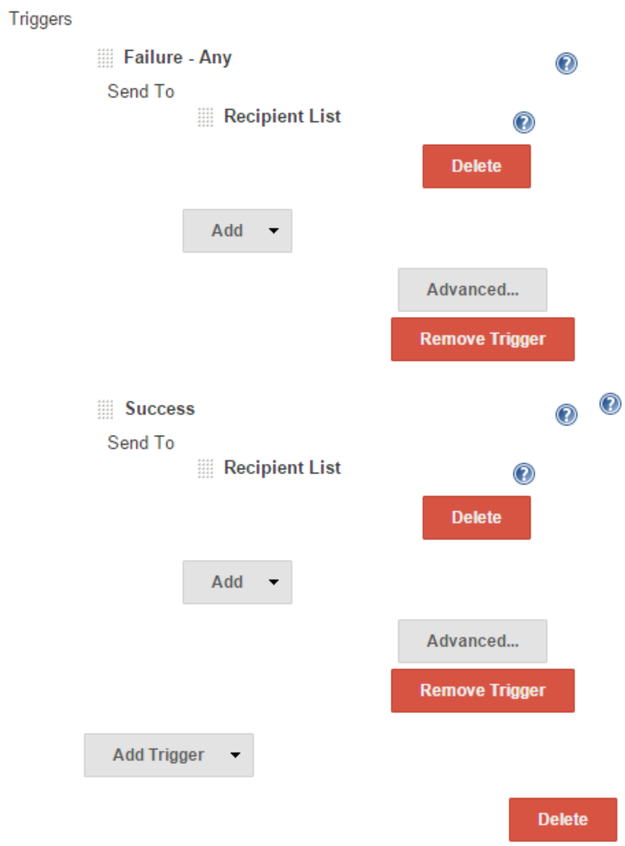
9. Delete **Developers** from the **Send To** section by clicking on the **Delete** button adjacent to it. The result should look something like the following screenshot:



10. Let's add another trigger to send an e-mail when the job is successful.
11. Click on the **Add Trigger** button and select the **Success** option, as shown in the following screenshot:



12. Configure this new success trigger in a similar fashion, by removing **Developers** and adding **Recipient List** under the **Send To** section. Finally, everything should look like the following screenshot:

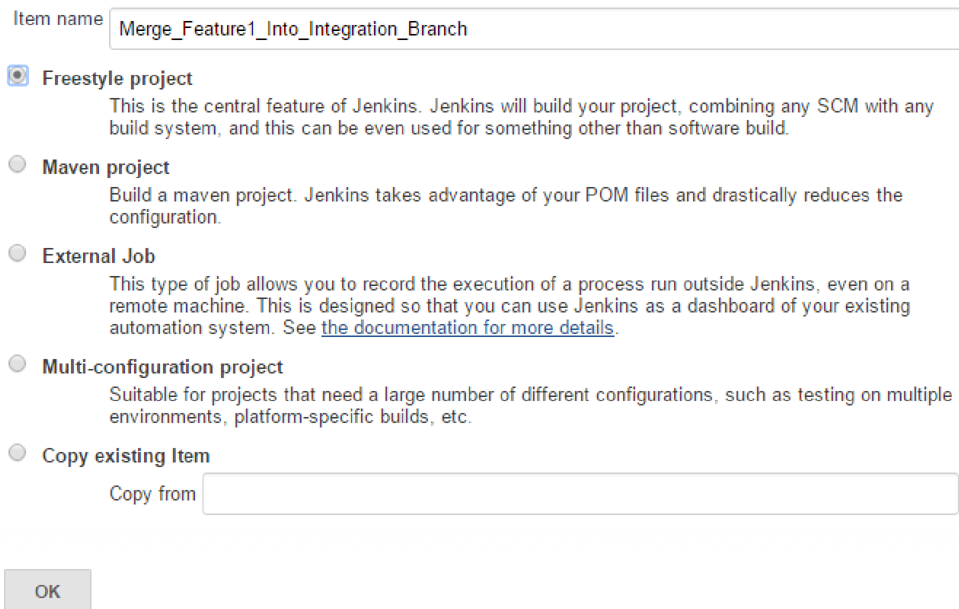


Creating a Jenkins job to merge code to the integration branch

The second Jenkins job in the pipeline performs the task of merging the successfully built and tested code into the integration branch:

1. I assume you are logged in to Jenkins as an admin and have privileges to create and modify jobs.

2. From the Jenkins Dashboard, click on the **New Item**.
3. Name your new Jenkins job `Merge_Feature1_Into_Integration_Branch`.
4. Select the type of job as **Freestyle project** and click on **OK** to proceed.



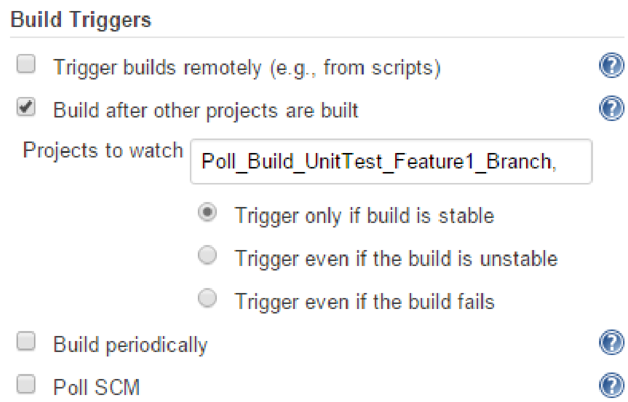
5. Add a meaningful description of the job in the **Description** section.

Using the build trigger option to connect two or more Jenkins jobs

This is an important section wherein we will connect two or more Jenkins jobs to form a Jenkins pipeline to achieve a particular goal:

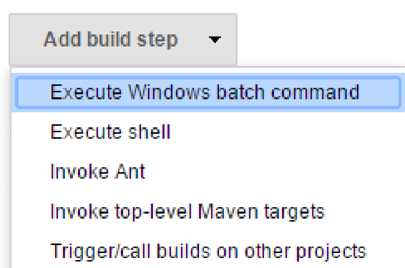
1. Scroll down to the **Build Triggers** section and select the **Build after other projects are built** option.
2. Under the **Projects to watch** field, add `Poll_Build_UnitTest_Feature1_Branch`.

3. Select the **Trigger only if the build is stable** option.



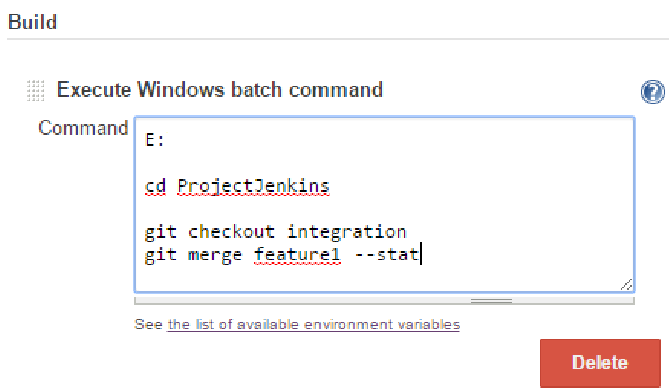
> In this way, we are telling Jenkins to initiate the current Jenkins job `Merge_Feature1_Into_Integration_Branch` only after the `Poll_Build_UnitTest_Feature1_Branch` job has completed successfully.

4. Scroll down to the **Build** section. From the **Add build step** dropdown, select **Execute Windows batch command**.



5. Add the following code into the **Command** section:

```
E:
cd ProjectJenkins
git checkout integration
git merge feature1 -stat
```



6. Let's see the code in detail:
    - The following line of code sets the current directory to `E:\ProjectJenkins`:

      ```
      E:
      cd ProjectJenkins
      ```
    - The following code checks out the integration branch:

      ```
      git checkout integration
      ```
    - The following line of code merges changes on the `feature1` branch into the integration branch.

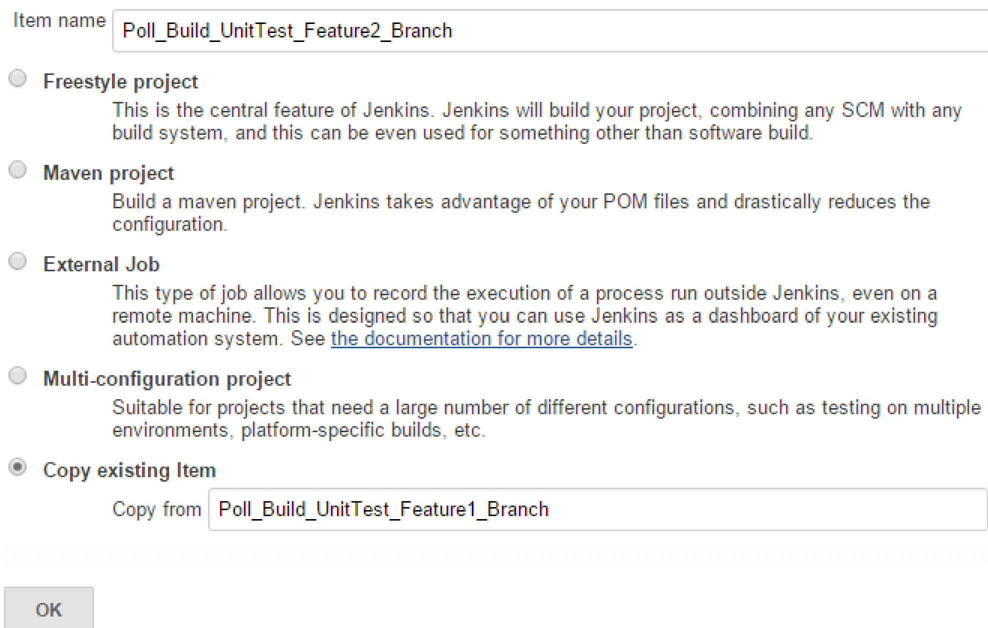
      ```
      git merge feature1 --stat
      ```
    - `--stat` gives a side-by-side comparison of the code merged.
7. Configure advanced e-mail notifications exactly the same way as mentioned earlier.
8. Save the Jenkins job by clicking on the **Save** button.

Creating a Jenkins job to poll, build, and unit test code on the feature2 branch

Since we have the two feature branches in place, we need to create a Jenkins Job to poll, build, and unit test code on the `feature2` branch. We will do this by cloning the already existing Jenkins job that polls the `feature1` branch. The steps are as follows:

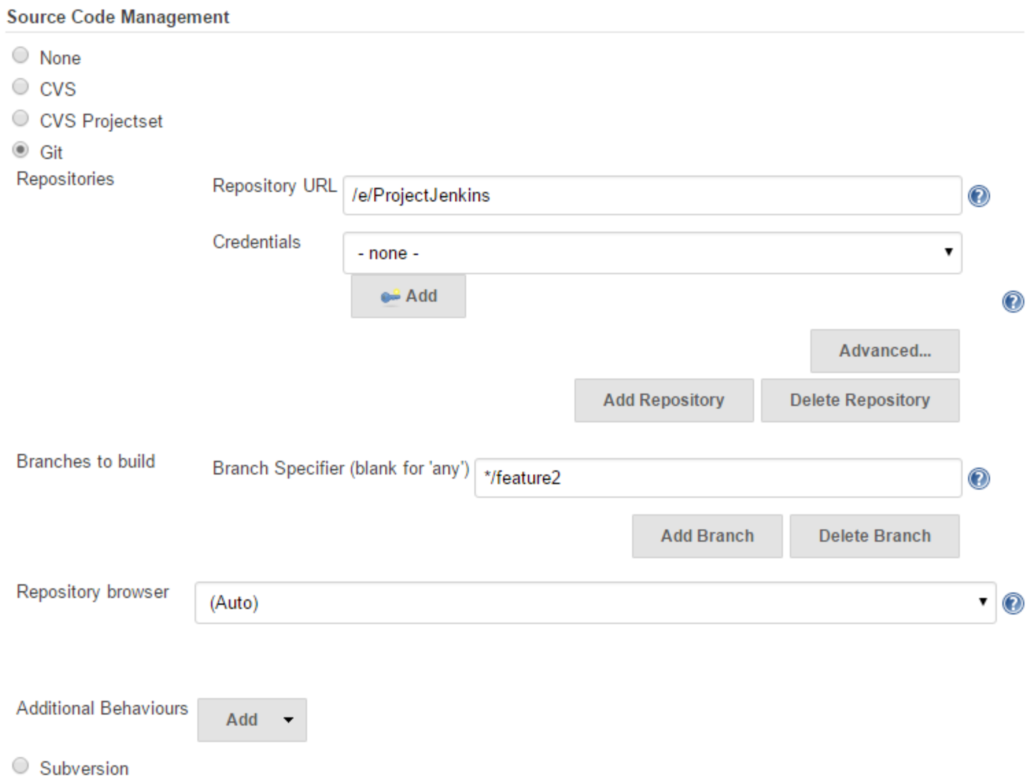
1. From the Jenkins Dashboard, click on **New Item**.
2. Name your new Jenkins job `Poll_Build_UnitTest_Feature2_Branch`.

3. Select the type of job as **Copy existing Item** and type `Poll_Build_UnitTest_Feature1_Branch` in the **Copy from** field.
4. Click on **OK** to proceed.

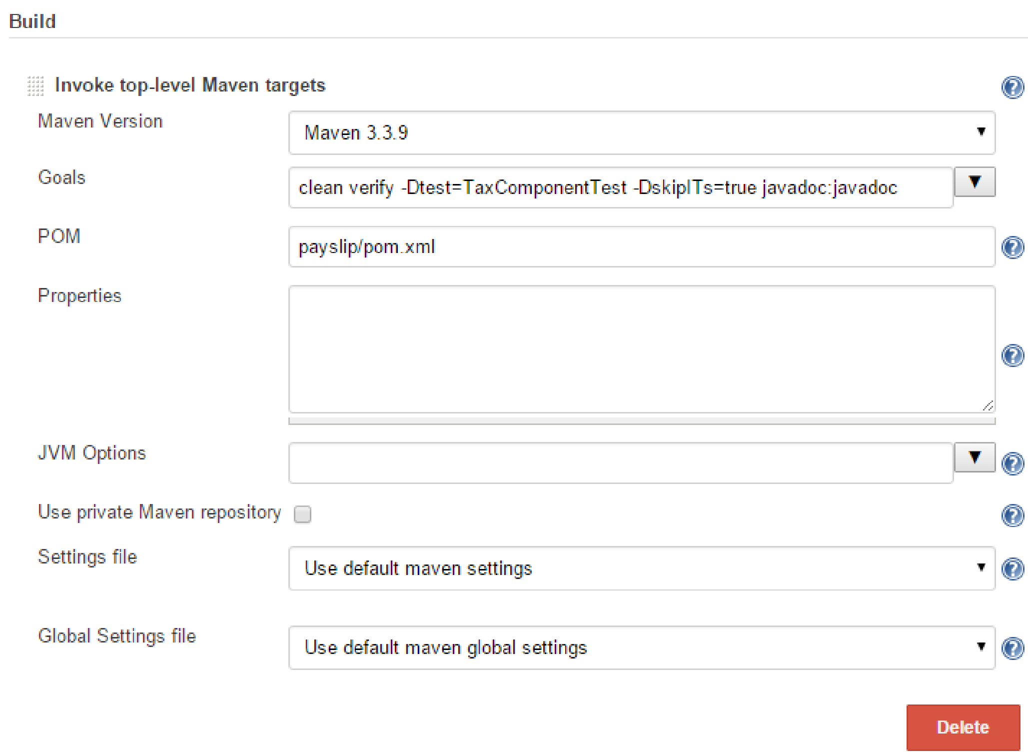


5. Scroll down to the **Source Code Management** section. You will find everything prefilled, as this is a copy of the Jenkins job `Poll_Build_UnitTest_Feature1_Branch`.

6. Change the **Branch to build** section from `*/feature1` to `*/feature2`, since we want our Jenkins job to poll the `feature2` branch.



7. Scroll down to the **Build** section. Modify the **Goals** field. Replace the existing one with `clean verify -Dtest=TaxComponentTest -DskipITs=true javadoc:javadoc`.



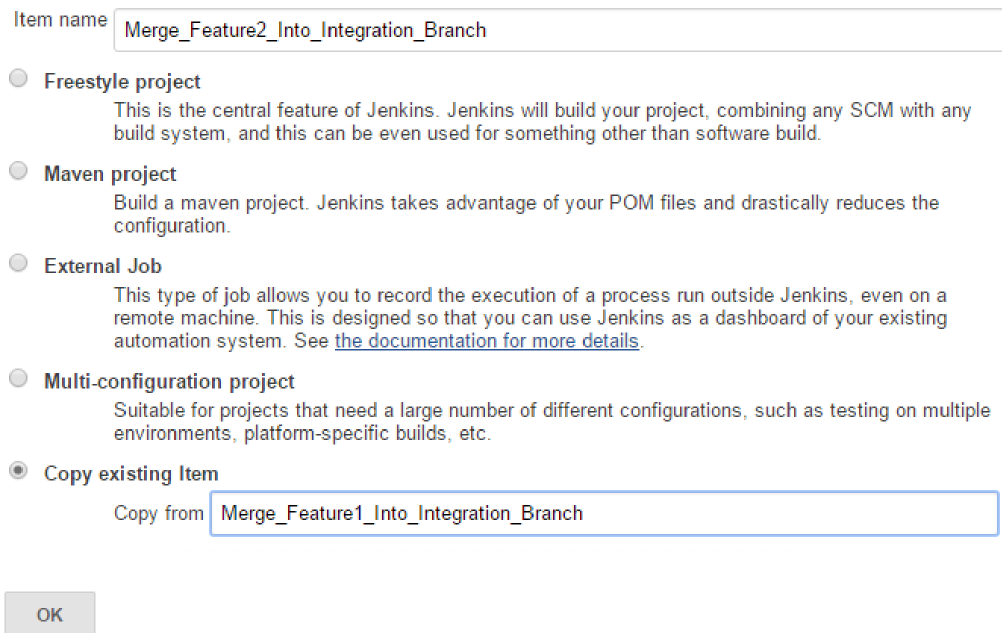
8. Leave everything as it is.
9. Scroll down to the **Editable Email Notification** section and you can change the **Project Recipient List** values if you want to.

Creating a Jenkins job to merge code to the integration branch

Similarly, we need to create another Jenkins job that will merge the successfully built and unit tested code on the `feature1` branch into the integration branch. And, we will do this by cloning the already existing Jenkins job that merges the successfully build and unit tested code from `feature1` branch into the Integration branch. The steps are as follows:

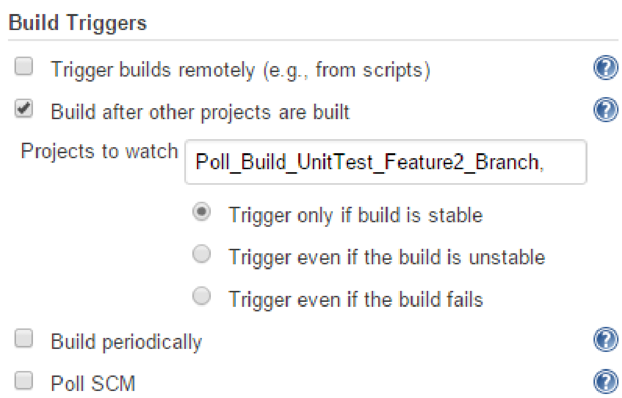
1. From the Jenkins Dashboard, click on **New Item**.

2. Name your new Jenkins job `Merge_Feature2_Into_Integration_Branch`. Alternatively, use any name that makes sense.
3. Select the type of job as **Copy existing Item** and type `Merge_Feature1_Into_Integration_Branch` in the **Copy from** field.
4. Click on **OK** to proceed.



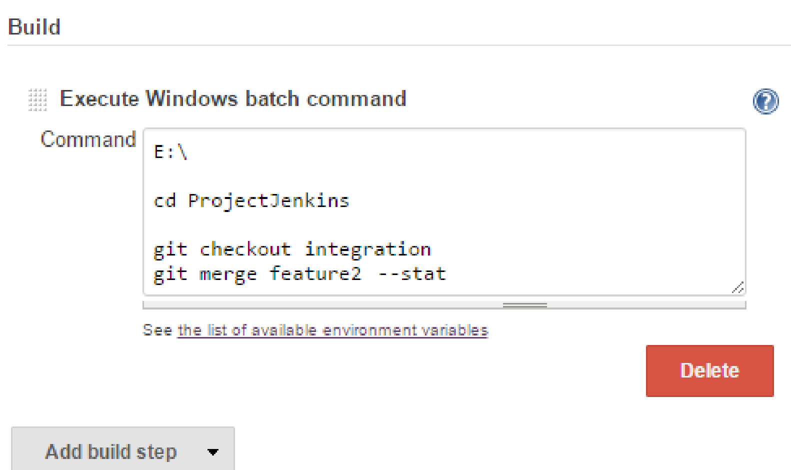
5. Scroll down to the **Build Triggers** section and select the **Build after other projects are built** option.

6. Under the **Projects to watch** field, replace `Poll_Build_UnitTest_Feature1_Branch` with `Poll_Build_UnitTest_Feature2_Branch`.



7. Scroll down to the **Build** section. Replace the existing code with the following:

```
E:

cd ProjectJenkins

Git checkout integration
Git merge feature2 --stat
```



8. Leave everything as it is.
9. Scroll down to the **Editable Email Notification** section. You can change the **Project Recipient List** values if you want to.

Summary

We began the chapter by discussing a Continuous Integration Design that contained a branching strategy, some tools for CI, and a CI Pipeline structure. We also saw how to install and configure Git along with the plugin that connects it with Jenkins.

The CI pipeline structure discussed as part of the CI design contained two pipelines: one for polling the feature branch and another one for polling the integration branch. Creating the pipeline to poll the feature branch was what we did in the second half of the chapter. This involved polling the feature branch for changes, performing a Maven build, unit testing, and publishing Javadoc. Later, we saw how to merge the successfully built and tested code into the integration branch.

However, this was half the story. The next chapter is all about creating the Jenkins pipeline to poll the Integration branch, wherein we will see how the successfully merged code on the integration branch is built and tested for integration issues and lots more.

5

Continuous Integration Using Jenkins – Part II

In this chapter, we will continue with the remaining portion of our Continuous Integration Design. Here, we will cover the following topics:

- Installing SonarQube
- Installing SonarQube Scanner
- Installing Artifactory (binary repository)
- Installing and configuring Jenkins plugin for SonarQube and Artifactory
- Creating the Jenkins pipeline to poll the integration branch
- Configuring Eclipse IDE with Git

Later in this chapter, after implementing the Continuous Integration Design, we will walk through our newly created Continuous Integration pipeline. We will do this by assuming the role of a developer and making some changes to the feature branch. We will then see how these changes propagate through the Jenkins CI pipeline and how Continuous Integration happens in real time.

Installing SonarQube to check code quality

Apart from integrating code in a continuous way, CI pipelines also include tasks that perform Continuous Inspection—inspecting code for its quality in a continuous approach.

Continuous Inspection deals with inspecting and avoiding code that is of poor quality. Tools such as SonarQube help us to achieve this. Every time a code gets checked in (committed), it is analyzed. This analysis is based on some rules defined by the code analysis tool. If the code passes the error threshold, it's allowed to move to the next step in its life cycle. If it crosses the error threshold, it's dropped.

Some organizations prefer checking the code for its quality right when the developer tries to check in the code. If the analysis is good, the code is allowed to be checked in, or else the check in is canceled and the developer needs to work on the code again.

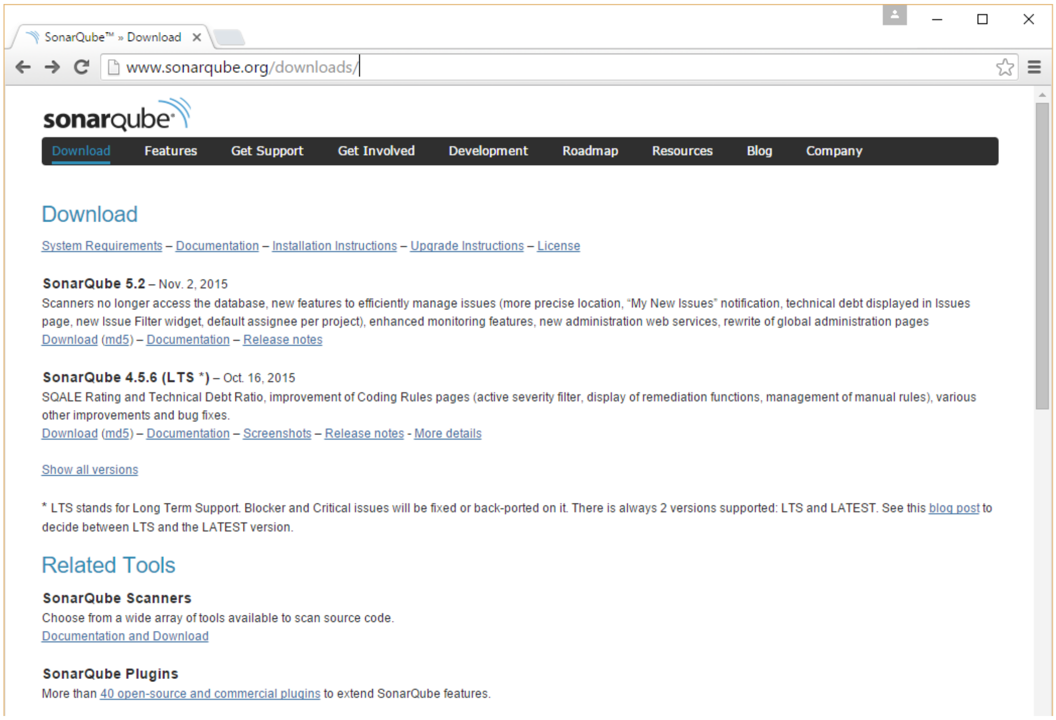
SonarQube is a code quality management tool that allows teams to manage, track, and improve the quality of their source code. It is a web-based application that contains rules, alerts, and thresholds, all of which can be configured. It covers the seven types of code quality parameters: architecture and design, duplications, unit tests, complexity, potential bugs, coding rules, and comments.

SonarQube is an open source tool that supports almost all popular programming languages with the help of plugins. SonarQube can also be integrated with a CI tool such as Jenkins to perform Continuous Inspection, which we will see shortly.

> SonarQube 5.1.2 is not the latest version of SonarQube. Nevertheless, we are using it in our example, as it's the only recent version of SonarQube that supports the build breaker plugin. We will see more about the build breaker plugin in the coming sections.

First, let's see how to install SonarQube. We will install SonarQube 5.1.2 on Windows 10 with the following steps:

1. To do this, download SonarQube 5.1.2 from `http://www.sonarqube.org/downloads/`, as shown in the following screenshot:



2. Once you have successfully downloaded the SonarQube 5.1.2 archive, extract it to `C:\Program Files\`. I have extracted it to `C:\Program Files\sonarqube-5.1.2`.

Setting the Sonar environment variables

Perform the following steps to set the `%SONAR_HOME%` environment variable:

1. Set the `%SONAR_HOME%` environment variable to the installation directory which, in our example, is `C:\Program Files\sonarqube-5.1.2`. Use the following command:

   ```
   setx SONAR_HOME "C:\Program Files\sonarqube-5.1.2" /M
   ```

2. To check the environment variable, use the following command:

   ```
   echo %SONAR_HOME%
   ```

3. The output should be as follows:

   ```
   C:\Program Files\sonarqube-5.1.2
   ```

Running the SonarQube application

To install SonarQube, open command prompt using admin privileges. Otherwise, this doesn't work. The steps are as follows:

1. Use the following commands to go to the directory where the scripts to install and start SonarQube are present:

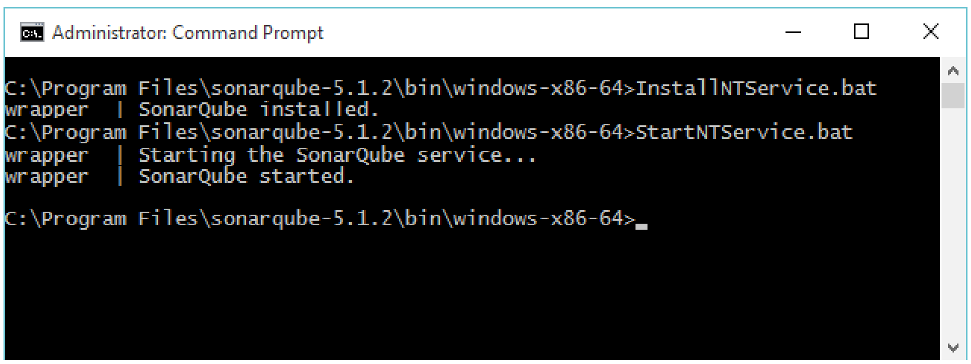
   ```
   cd %SONAR_HOME%\bin\windows-x86-64
   ```

2. To install SonarQube, run the `InstallNTService.bat` script:

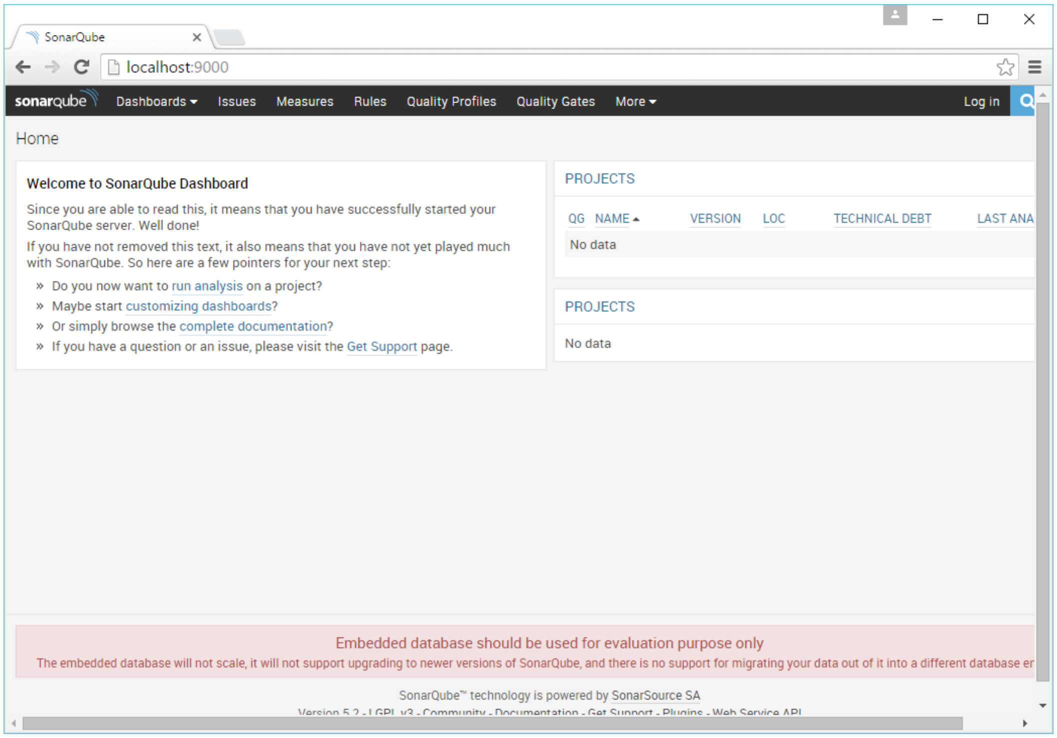
   ```
   InstallNTService.bat
   ```

3. To start SonarQube, run the `StartNTService.bat` script:

   ```
   StartNTService.bat
   ```



4. To access SonarQube, type the following link in your favorite web browser `http://localhost:9000/`.

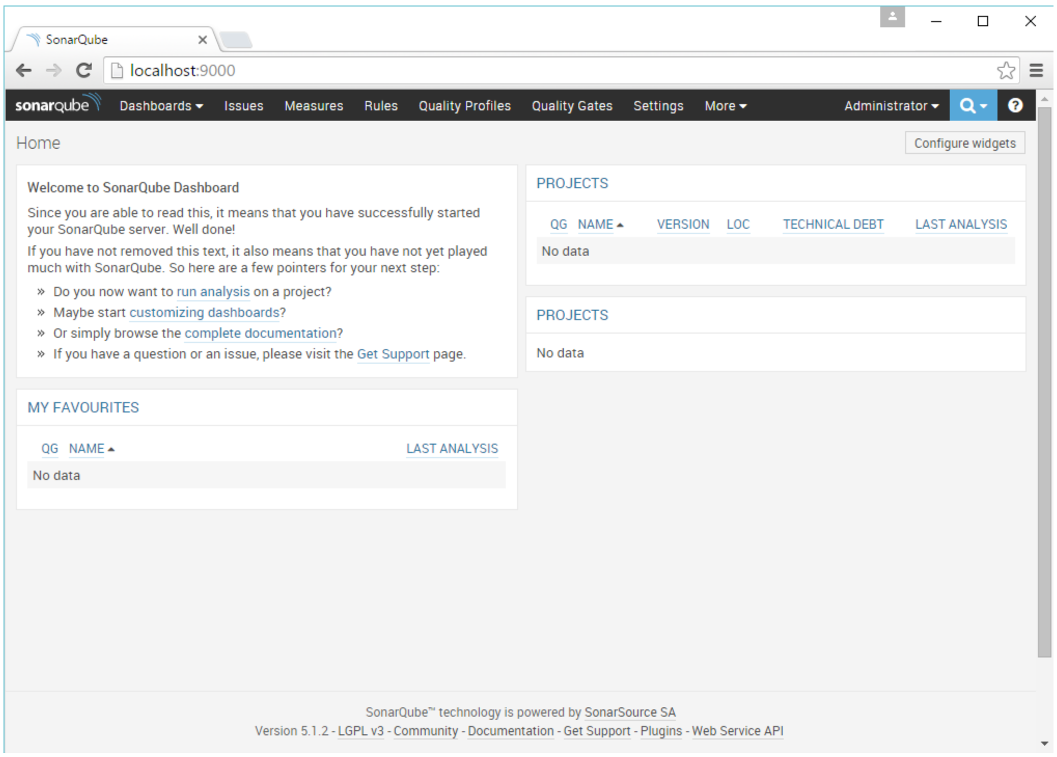


> Right now, there are no user accounts configured in SonarQube. However, by default, there is an admin account with the username `admin` and the password `admin`.

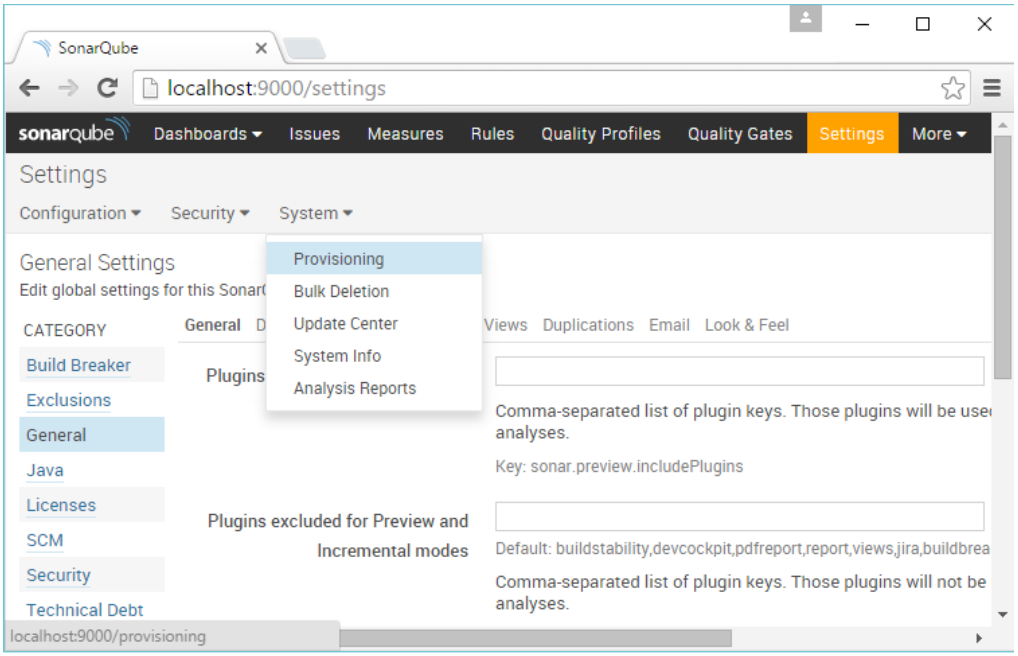
Creating a project inside SonarQube

To create the project in SonarQube, use the following steps:

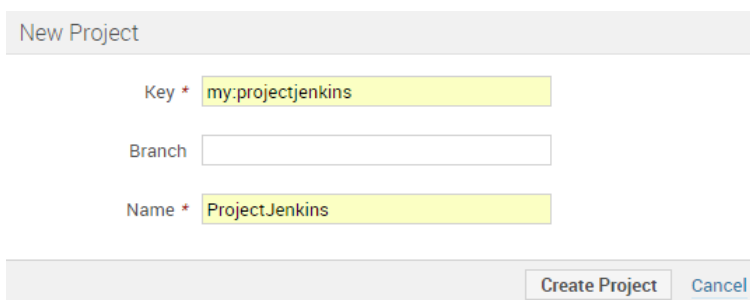
1. Log in as an admin by clicking on the **Log in** link at the top-right corner on the Sonar Dashboard. You will see some more items in the menu bar, as shown in the following screenshot:



2. Click on the **Settings** link on the menu bar.
3. On the **Settings** page, click on **System** and select the **Provisioning** option, as shown in the following screenshot:



4. On the **Provisioning** page, click on the **Create** link present at the right corner to create a new project.
5. A pop-up window will appear asking for **Key**, **Branch**, and **Name** values. Fill the blanks as shown in the following screenshot and click on the **Create Project** button.



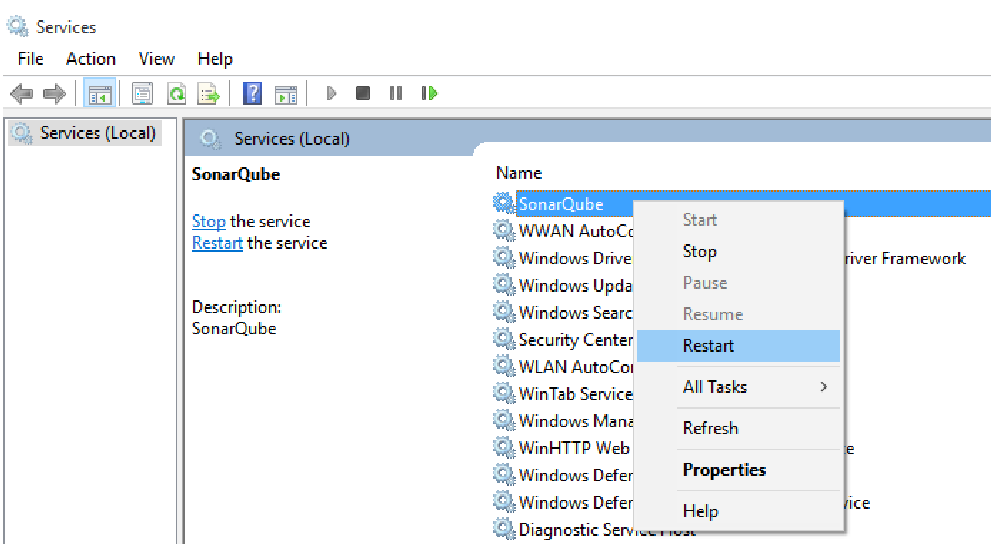
6. That's it. We have successfully created a project inside SonarQube.

Installing the build breaker plugin for Sonar

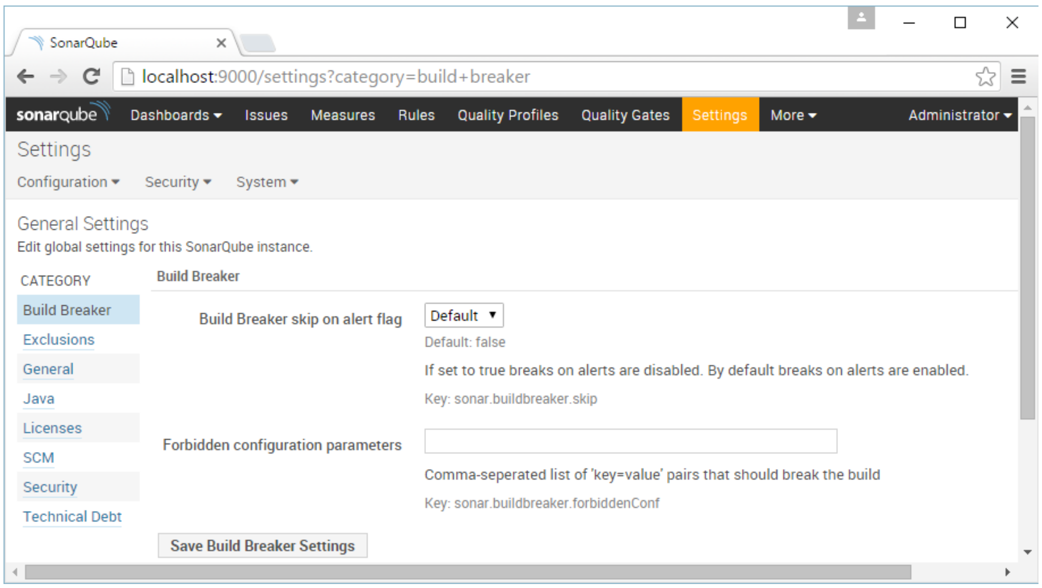
The build breaker plugin is available for SonarQube. It's exclusively a SonarQube plugin and not a Jenkins plugin. This plugin allows the Continuous Integration system (Jenkins) to forcefully fail a Jenkins build if a quality gate condition is not satisfied. To install the build breaker plugin, follow these steps:

1. Download the build breaker plugin from the following link: `http://update.sonarsource.org/plugins/buildbreaker-confluence.html`.
2. Place the downloaded `sonar-build-breaker-plugin-1.1.rar` file in the following location: `C:\Program Files\sonarqube-5.1.2\extensions\plugins`.
3. We need to restart SonarQube service. To do so, type `services.msc` in Windows **Run**.
4. From the **Services** window, look for a service named **SonarQube**. Right-click on it and select **Restart**.

> The support for the build breaker plugin has been discontinued since SonarQube 5.2. If you intend to use the latest version of SonarQube, then you won't be able to use the build breaker plugin.



5. After a successful restart, go to the SonarQube dashboard and log in as `admin`. Click on the **Settings** link from the menu options.
6. On the **Settings** page, you will find the **Build Breaker** option under the **CATEGORY** sidebar as shown in the following screenshot. Do not configure anything.

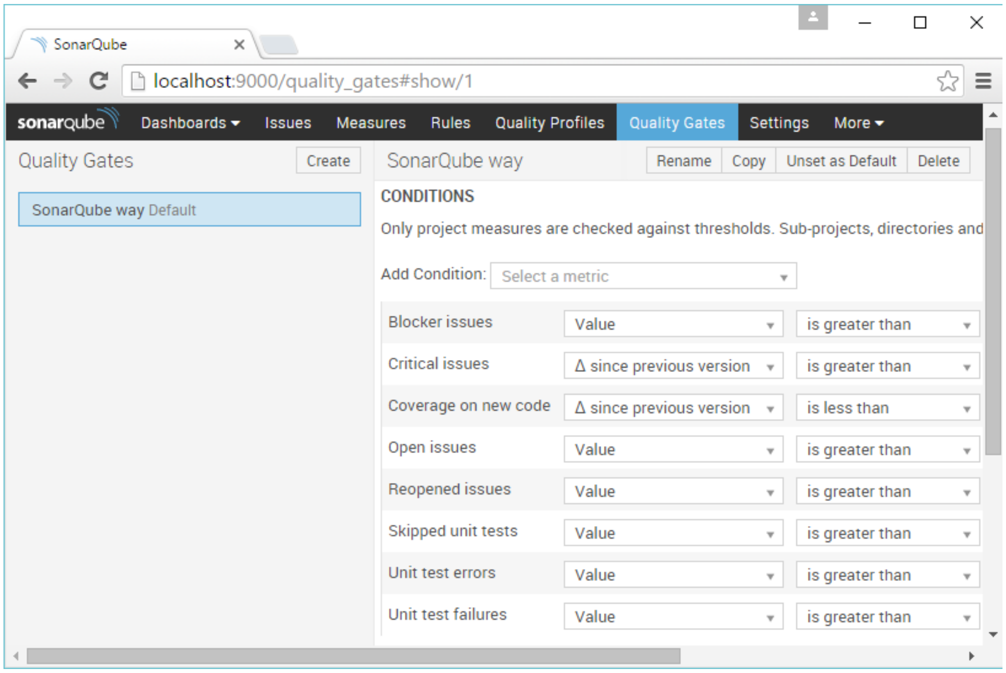


Creating quality gates

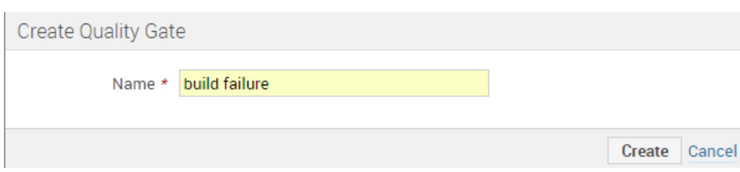
For the build breaker plugin to work, we need to create a **quality gate**. It's a rule with some conditions. When a Jenkins job that performs a static code analysis is running, it will execute the **quality profiles** and the **quality gate**. If the quality gate check passes successfully, then the Jenkins job continues. If it fails, then the Jenkins job is aborted. Nevertheless, the static code analysis still happens. To create a quality gate, perform the following steps:

1. Click on the **Quality Gates** link on the menu. By default, we have a quality gate named **SonarQube way**.

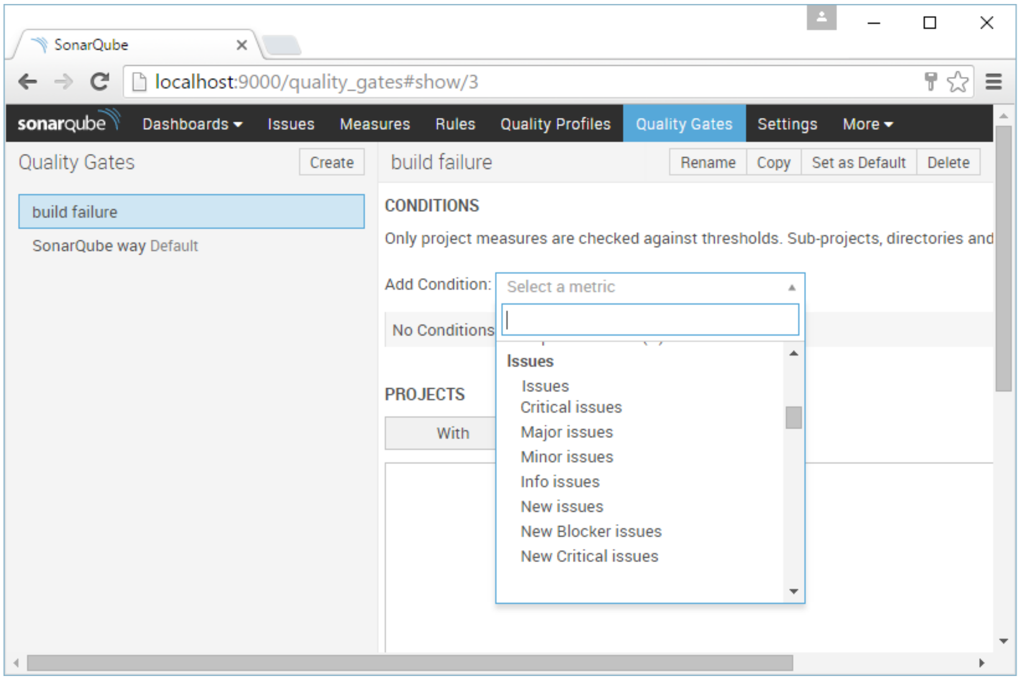
2. Click on the **Create** button to create a new quality gate.



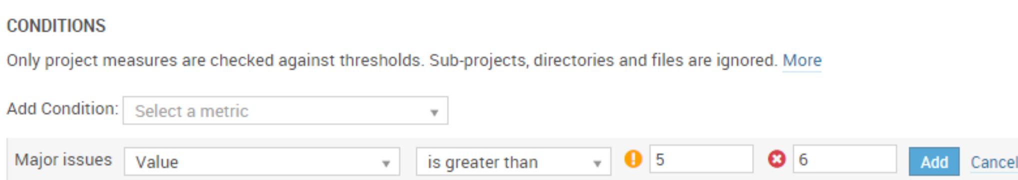
3. In the pop-up window, add the name that you wish to give to your new quality gate in the **Name** field. In our example, I have used `build failure`.
4. Once done, click on the **Create** button.



5. You will see a new quality gate named `build failure` on the left-hand side of the page.
6. Now, click on the `build failure` quality gate and you will see a few settings on the right-hand side of the page.
7. Set the `build failure` quality gate as the default quality gate by clicking on the **Set as Default** button.
8. Now, in the **Add Condition** field, choose a condition named **Major issues** from the drop-down menu, as shown in the following screenshot:



9. Now let's configure our condition. If the number of **Major issues** is greater than six, the build should fail; and if it is between five and six, it should be a warning. To achieve this, set the condition parameters as shown here:



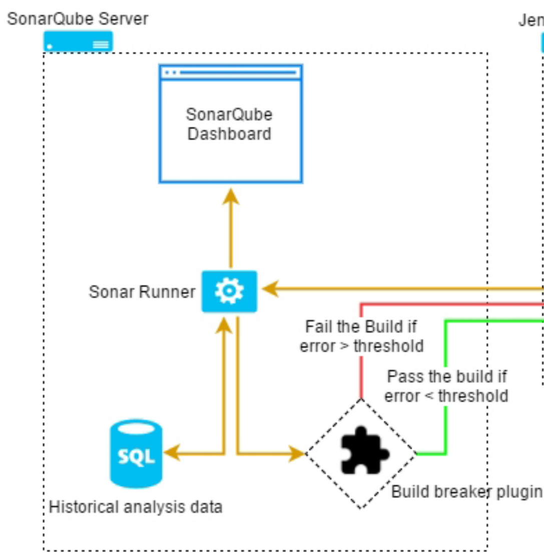
Installing SonarQube Scanner

SonarQube Scanner, also called **SonarQube Runner**, is an important application that actually performs the code analysis. SonarQube Scanner scans the code for its quality, based on some predefined rules. It then helps the SonarQube web application to display this analysis along with other metrics.

The following image clearly depicts how SonarQube Server, SonarQube Scanner, and the build breaker plugin work together with Jenkins.

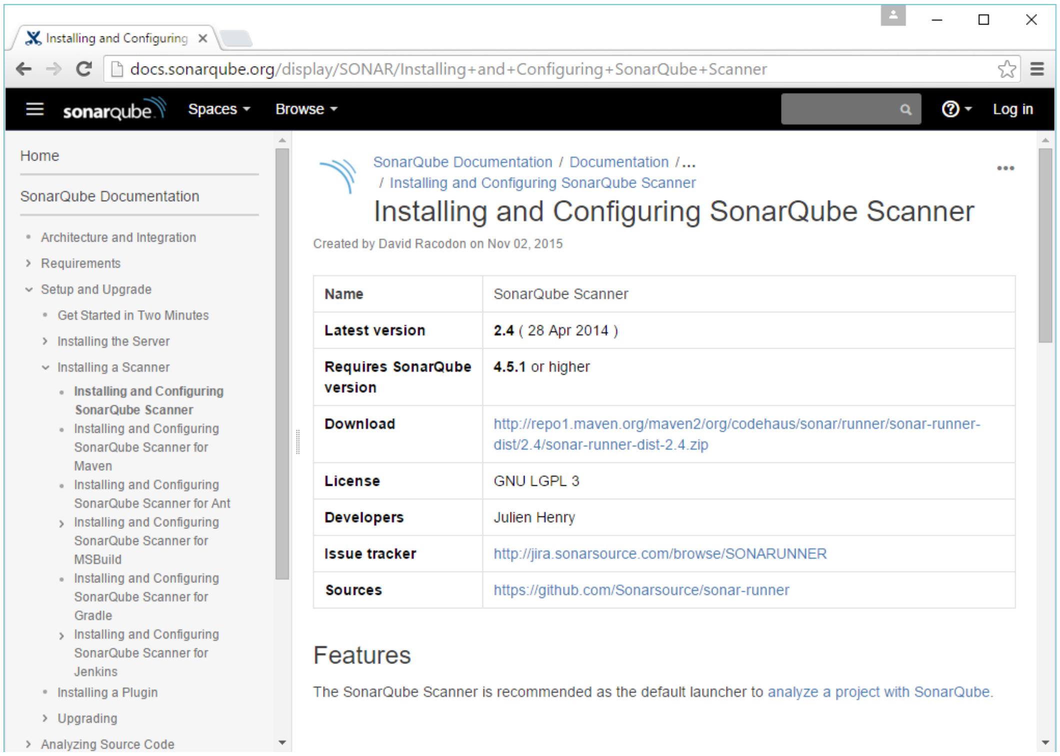
SonarQube Scanner is invoked through Jenkins to perform the code analysis. The code analysis is presented on the SonarQube dashboard and also passed to the build breaker plugin.

There are conditions defined inside the quality gates. If the analysis passes these conditions, then the Jenkins job is signed to proceed. However, if the analysis fails the condition, then the build breaker plugin terminates the Jenkins job.



Follow these steps to install SonarQube Scanner:

1. Download the SonarQube Scanner (that is, SonarQube Runner) from the link `http://docs.sonarqube.org/display/SONAR/Analyzing+with+SonarQube+Scanner`.
2. The link keeps updating, so just look for SonarQube Scanner on Google if you don't find it.



3. Extract the downloaded file to `C:\Program Files\`. I have extracted it to `C:\Program Files\sonar-runner-2.4`.
4. That's it, SonarQube Runner is installed.

Setting the Sonar Runner environment variables

Perform the following steps to set the `%SONAR_RUNNER_HOME%` environment variable:

1. Set the `%SONAR_RUNNER_HOME%` environment variable to the installation directory of SonarQube Runner by giving the following command:

   ```
   setx SONAR_RUNNER_HOME "C:\Program Files\sonar-runner-2.4" /M
   ```

2. To check the environment variable, use the following command:

   ```
   echo %SONAR_RUNNER_HOME%
   ```

3. You should get the following output:

```
C:\Program Files\sonar-runner-2.4
```

4. Add the `%SONAR_RUNNER_HOME%\bin` directory to your path using the following command:

```
setx PATH "%PATH%\;C:\Program Files\sonar-runner-2.4\bin" /M
```

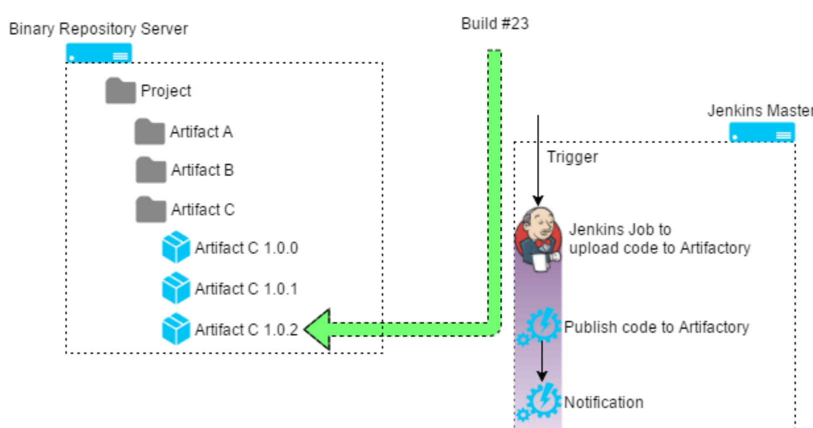
Installing Artifactory

Continuous Integration results in frequent builds and packages. Hence, there is a need for a mechanism to store all this binary code (builds, packages, third-party plugins, and so on) in a system akin to a version control system.

Since, version control systems such as Git, TFS, and SVN store code and not binary files, we need a binary repository tool. A **binary repository** tool such as Artifactory or Nexus that is tightly integrated with Jenkins provides the following advantages:

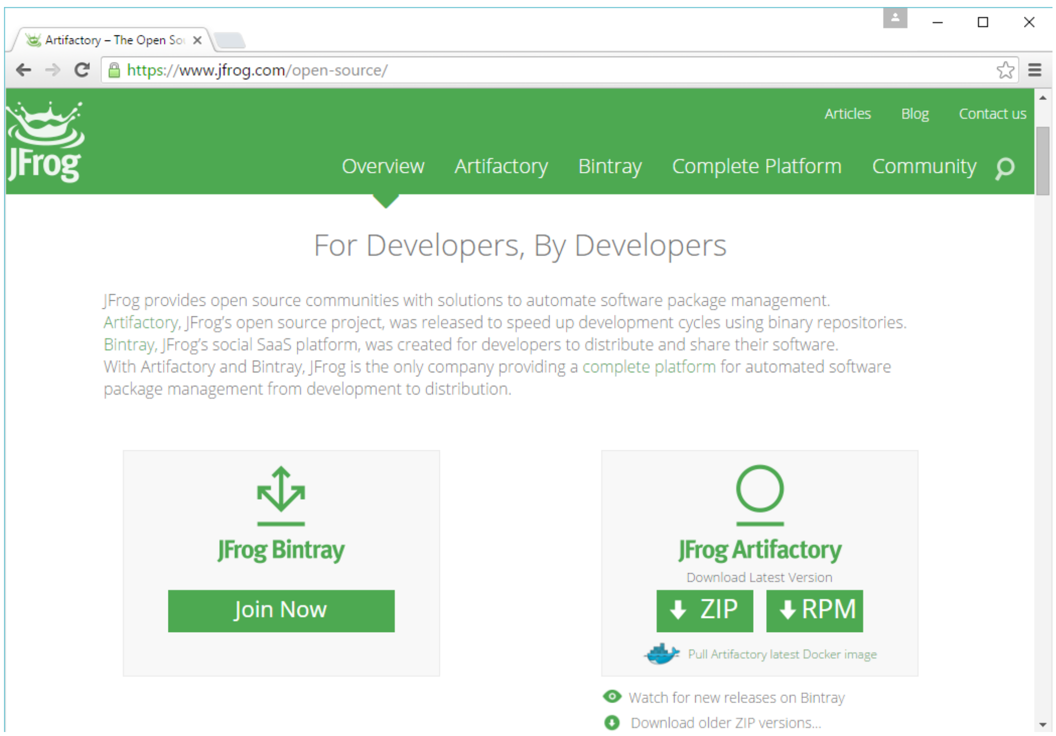
- Tracking builds (Who triggers a build? What version of code in the VCS was build?)
- Dependencies
- Deployment history

The following image depicts how a binary repository tool such as Artifactory works with Jenkins to store build artifacts. In the coming sections, we will see how to achieve this by creating a Jenkins job to upload code to Artifactory.



In this book, we will use Artifactory to store our builds. Artifactory is a tool used to version control binaries. The binaries can be anything from built code, packages, executables, Maven plugins, and so on. We will install Artifactory on Windows 10. The steps are as follows:

1. Download the latest stable version of Artifactory from `https://www.jfrog.com/open-source/`. Download the **ZIP** archive.



2. Extract the downloaded file to `C:\Program Files\`. I have extracted it to `C:\Program Files\artifactory-oss-4.3.2`.

Setting the Artifactory environment variables

Perform the following steps to set the `%ARTIFACTORY_HOME%` environment variable:

1. Set the `%ARTIFACTORY_HOME%` environment variable to the installation directory of Artifactory with the following command:

```
setx ARTIFACTORY_HOME "C:\Program Files\artifactory-oss-4.3.2" /M
```

2. To check the environment variable, use the following command:

```
C:\WINDOWS\system32>echo %ARTIFACTORY_HOME%
```

3. You should get the following output:

```
C:\Program Files\artifactory-oss-4.3.2
```

Running the Artifactory application

To run Artifactory, open **Command Prompt** using admin privileges. Otherwise, this doesn't work.

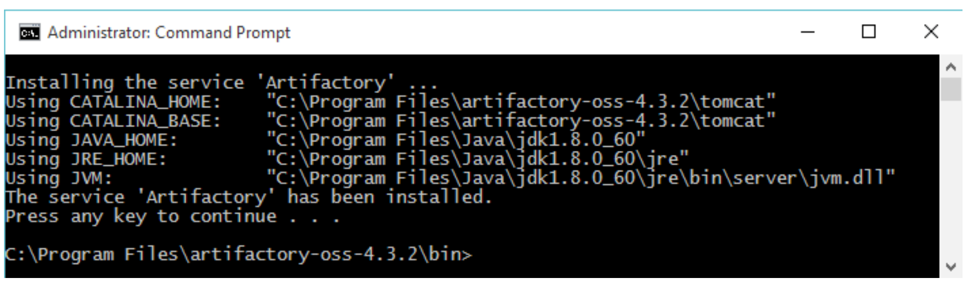
1. Go to the location where the script to run Artifactory is present:

```
cd %ARTIFACTORY_HOME%\bin
```

2. Execute the `installService.bat` script:

```
installService.bat
```

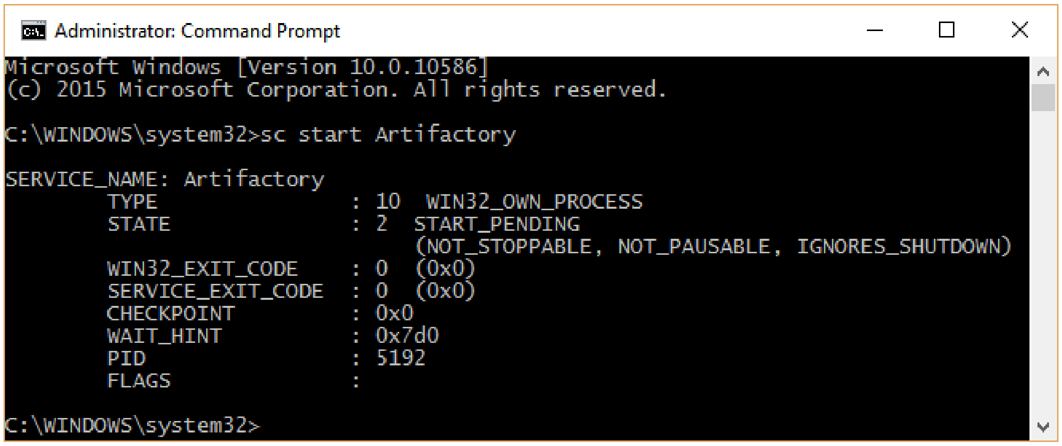
3. This will open up a new **Command Prompt** window that will install Artifactory as a windows service.



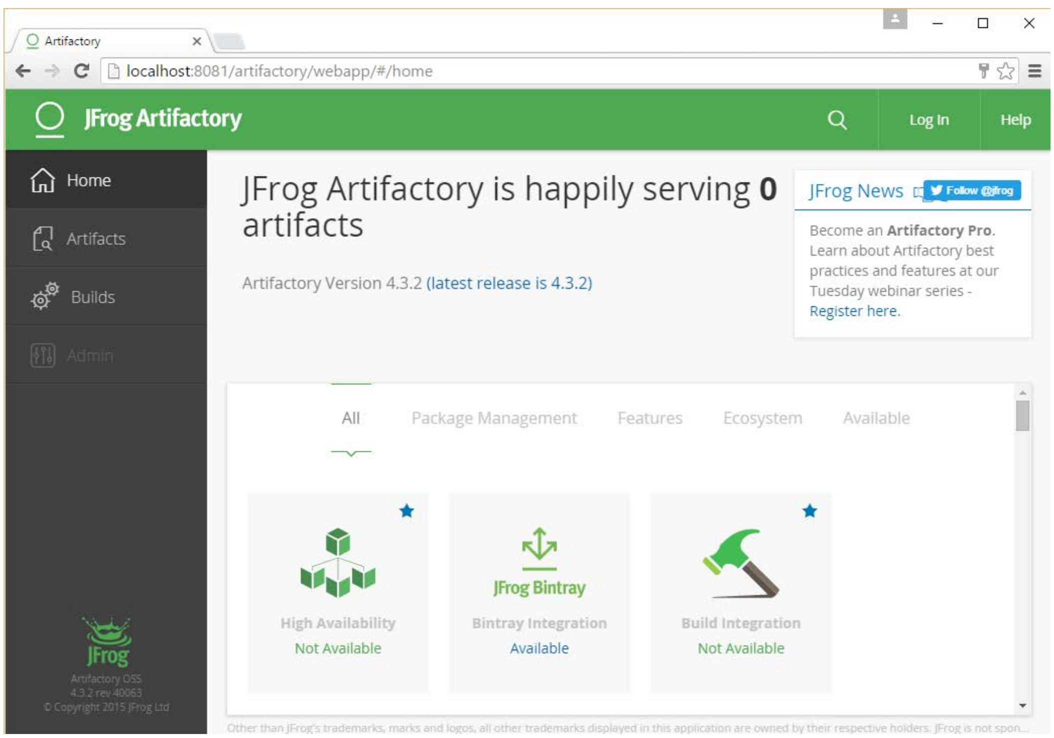
4. To start Artifactory, open **Command Prompt** using admin privileges and use the following command:

```
sc start Artifactory
```

5. This will provide the following output:



6. Access Artifactory using the following link: `http://localhost:8081/artifactory/`.

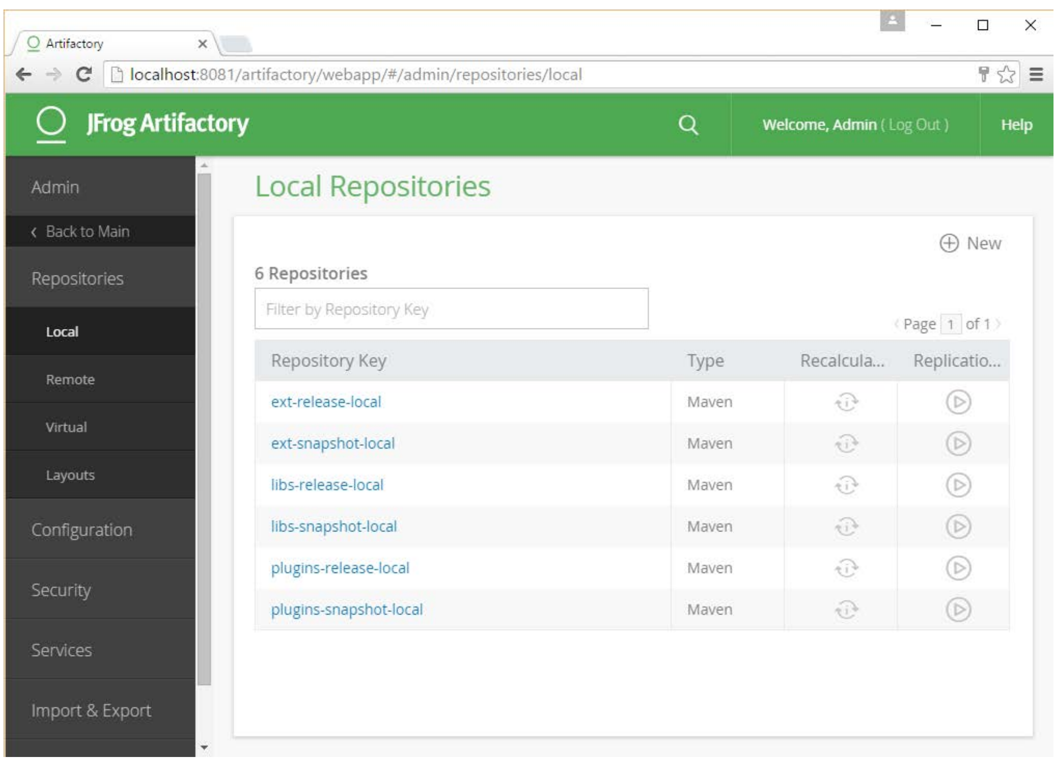


> Right now, there is no user account configured in Artifactory. However, by default, there is an admin account with the username `admin` and the password `password`.

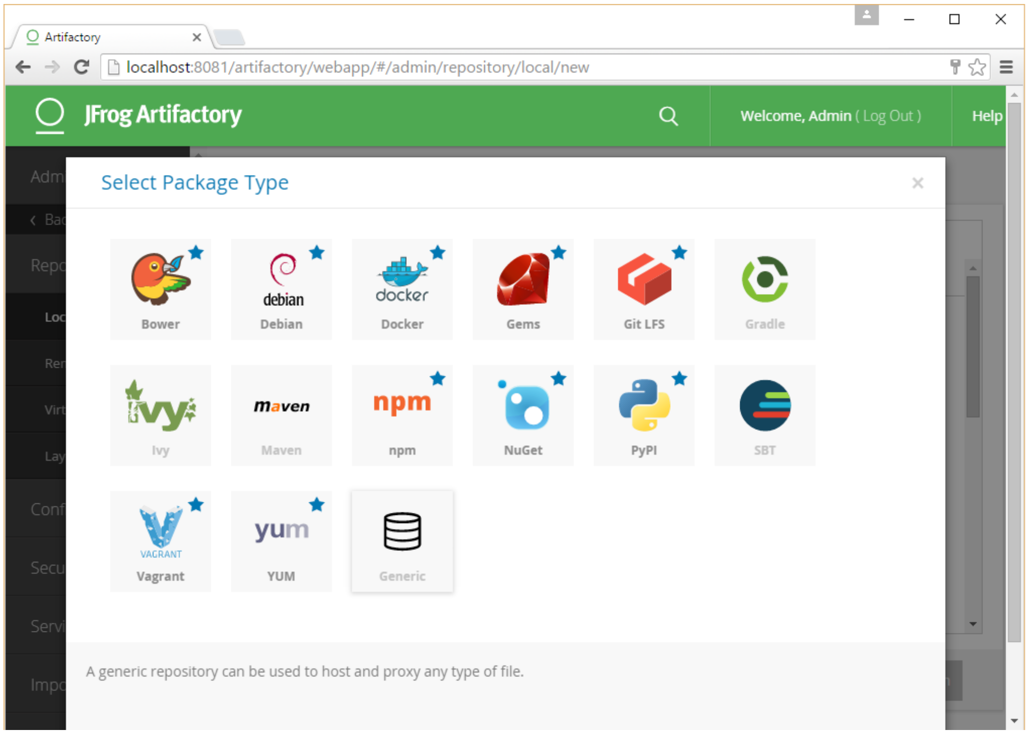
Creating a repository inside Artifactory

We will now create a repository inside Artifactory to store our package. The steps are as follows:

1. Log in to Artifactory using the `admin` account.
2. On the menu on the left-hand side, click on **Repositories** and then select **Local**. You will see a list of repositories that are present by default.
3. Click on the **New** button with a plus symbol, which is present on the right-hand side of the page.

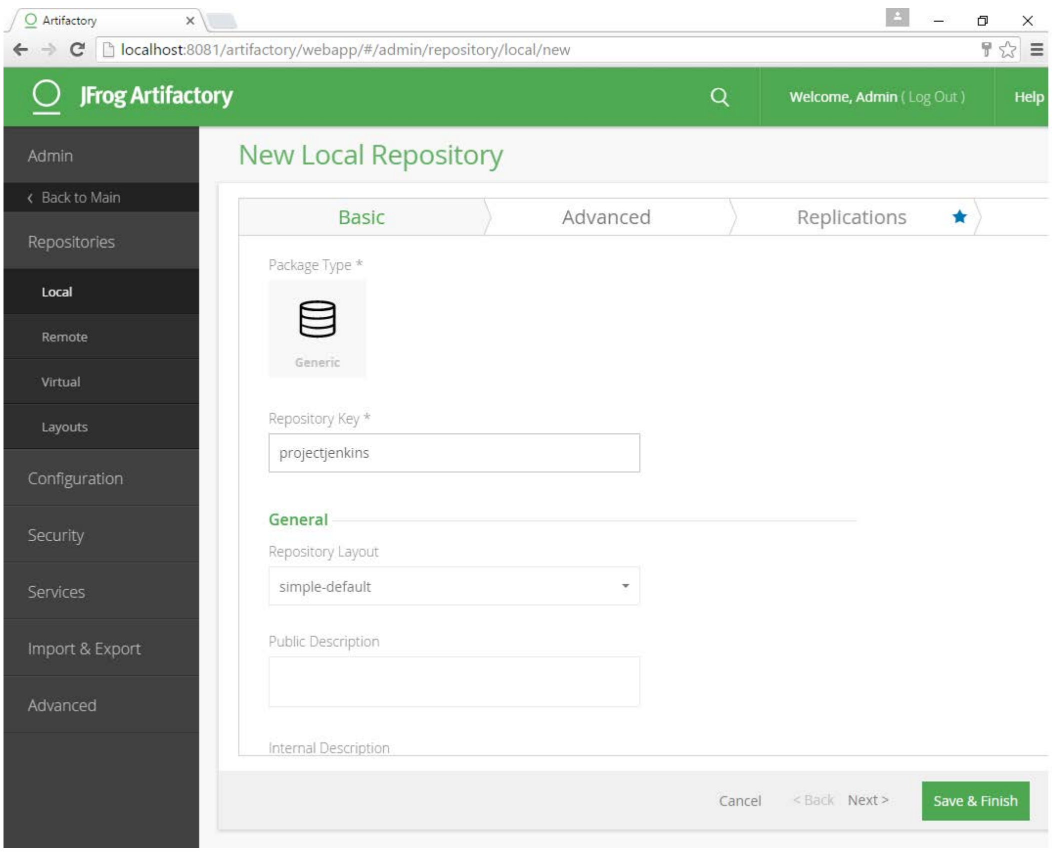


4. In the window that pops-up, select the package type as **Generic**.

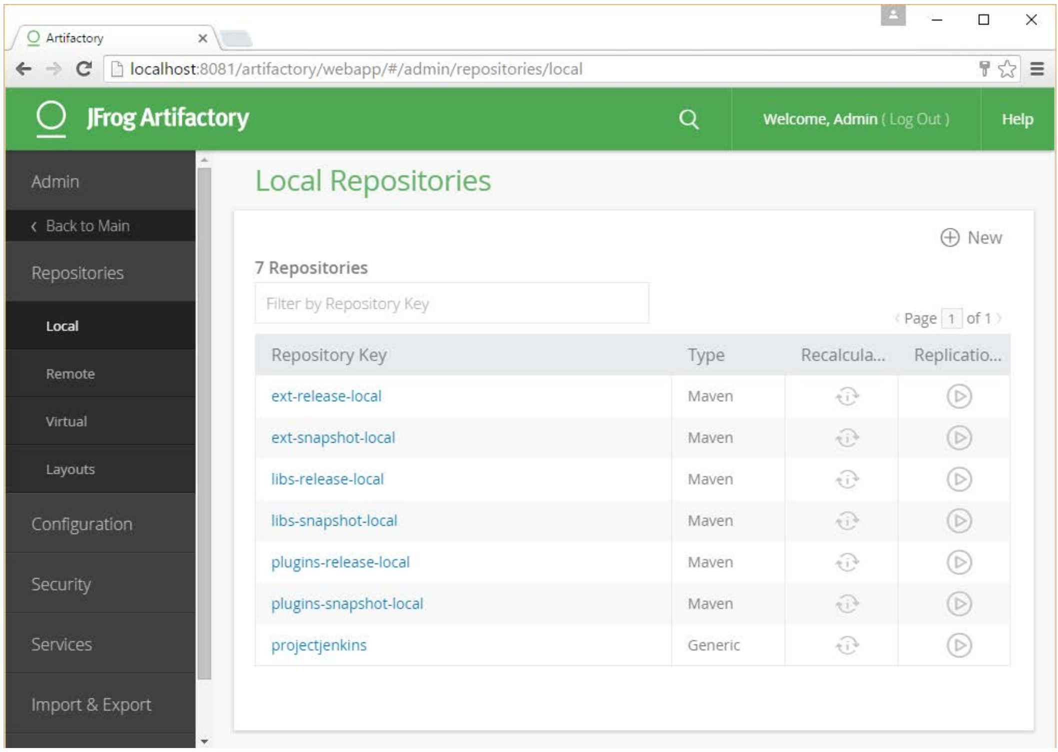


5. Give a name in the **Repository Key** * field. In our example I have used `projectjenkins`.

6. Leave the rest of the fields at their default values and click on the **Save & Finish** button.



7. As you can see in the following screenshot, there is a new repository named `projectjenkins`.



Jenkins configuration

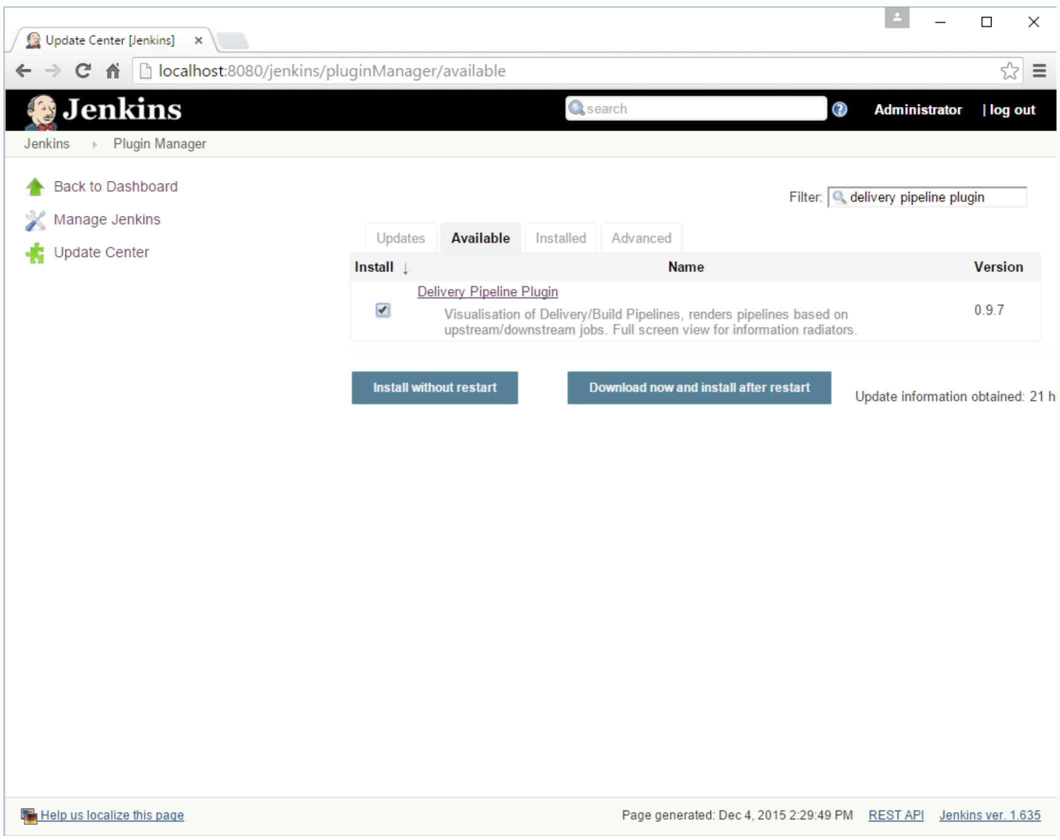
In the previous sections, we saw how to install and configure Artifactory and SonarQube along with SonarQube Runner. For these tools to work in collaboration with Jenkins, we need to install their respective Jenkins plugins.

Also, we will see the installation of a special plugin named *delivery pipeline plugin*, which is used to give a visual touch to our Continuous Integration pipeline.

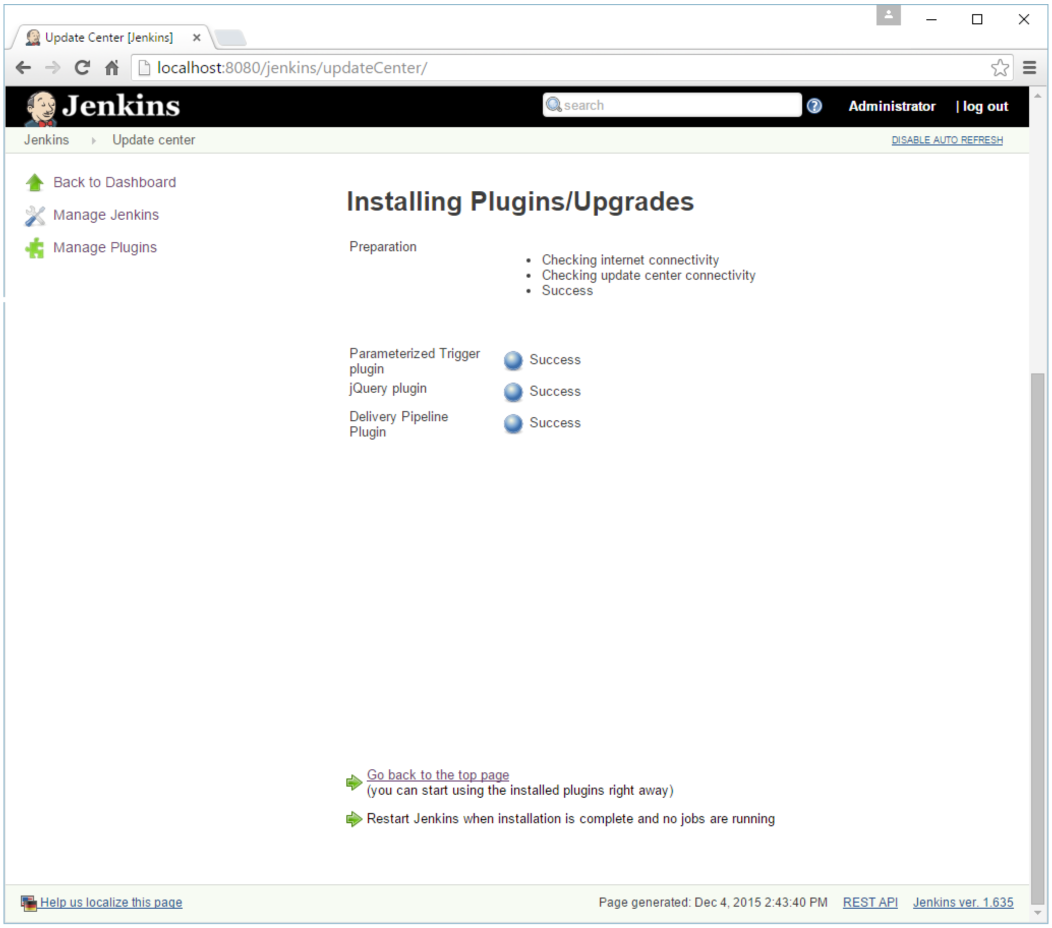
Installing the delivery pipeline plugin

To install the delivery pipeline plugin, perform the following steps:

1. On the Jenkins Dashboard, click on the **Manage Jenkins** link. This will take you to the Manage Jenkins page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `delivery pipeline plugin` in the search box.
4. Select **Delivery Pipeline Plugin** from the list and click on the **Install without restart** button.



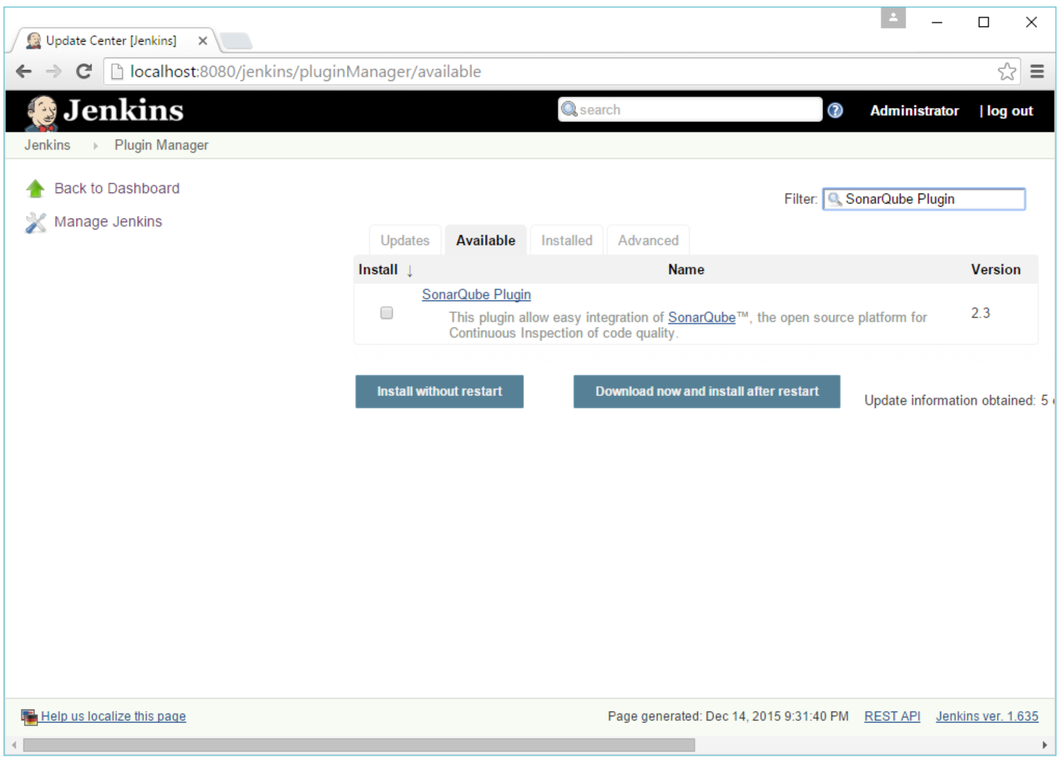
5. The download and installation of the plugin starts automatically. You can see **Delivery Pipeline Plugin** has some dependencies that get downloaded and installed.



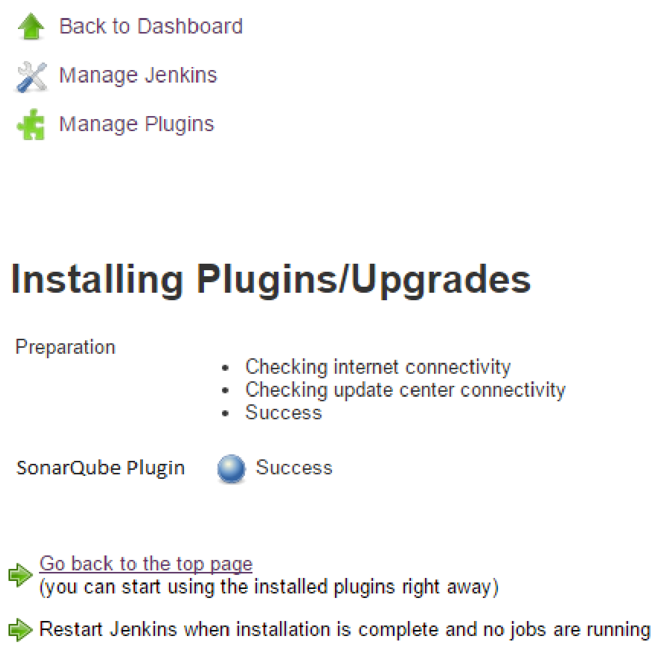
Installing the SonarQube plugin

To install the SonarQube plugin, perform the following steps:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `SonarQube plugin` in the search box. Select **SonarQube Plugin** from the list and click on the **Install without restart** button.

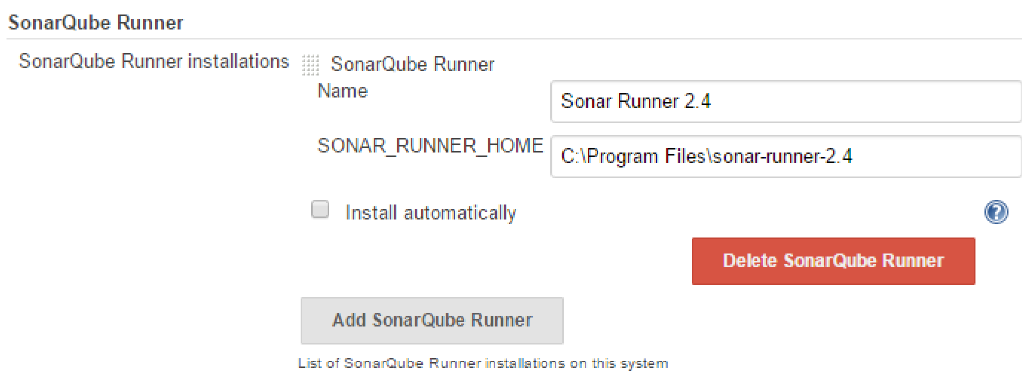


4. As it can be seen in the next screenshot, the plugin is installed immediately:



5. Upon successful installation of the SonarQube Plugin, go to the **Configure System** link on the **Manage Jenkins** page.

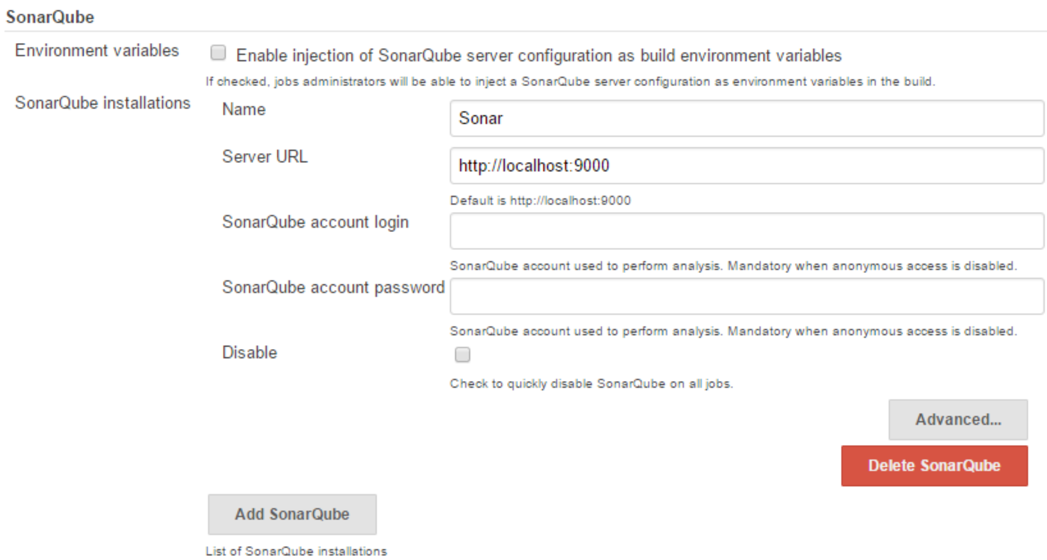
6. Scroll down until you see the **SonarQube Runner** section and fill in the blanks as shown here:
   - You can name your SonarQube Runner installation using the **Name** field.
   - Set the **SONAR_RUNNER_HOME** value to the location where you have installed SonarQube Runner. In our example, it's `C:\Program Files\sonar-runner-2.4`.



> You can add as many Sonar Runner instances as you want by clicking on the **Add SonarQube Runner** button. Although not necessary, if you do, provide each SonarQube Runner installation a different name.

7. Now, scroll down until you see the **SonarQube** section and fill in the blanks as shown here:
   - Name your SonarQube installation using the **Name** field.

◦ Provide the **Server URL** field for the SonarQube. In our example, it's `http://localhost:9000`.



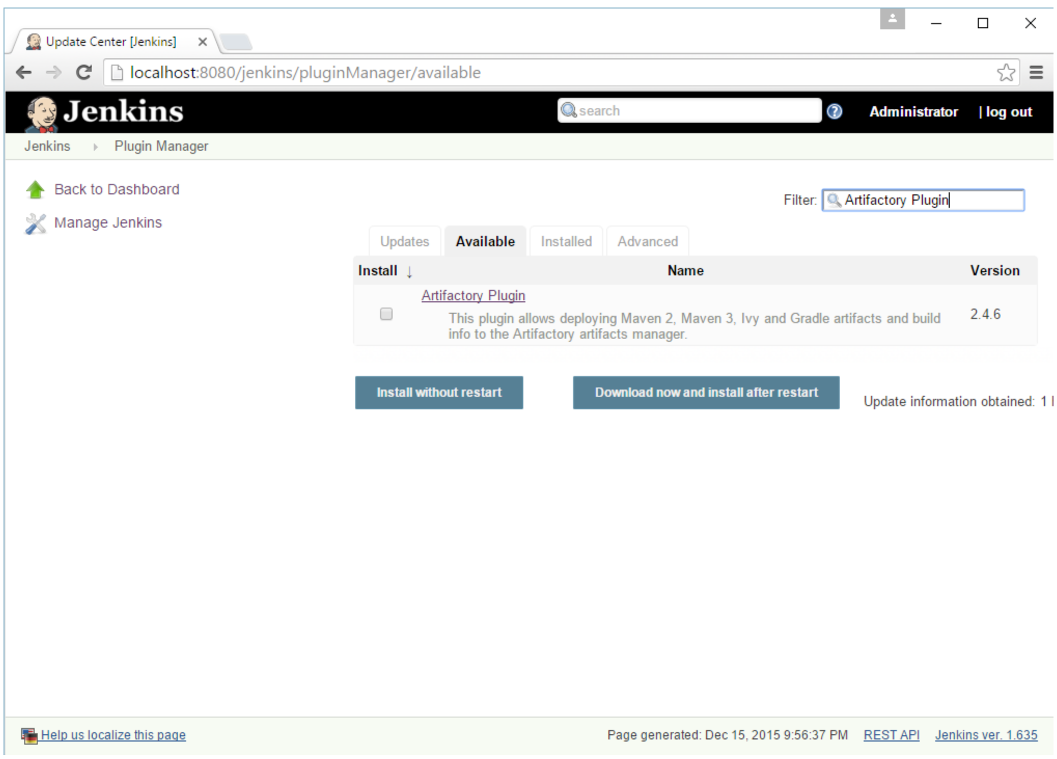
8. Save the configuration by clicking on the **Save** button at the bottom of the screen.

> You can add as many SonarQube instances as you want by clicking on the **Add SonarQube** button. Although not necessary, if you do, provide each SonarQube installation a different name.

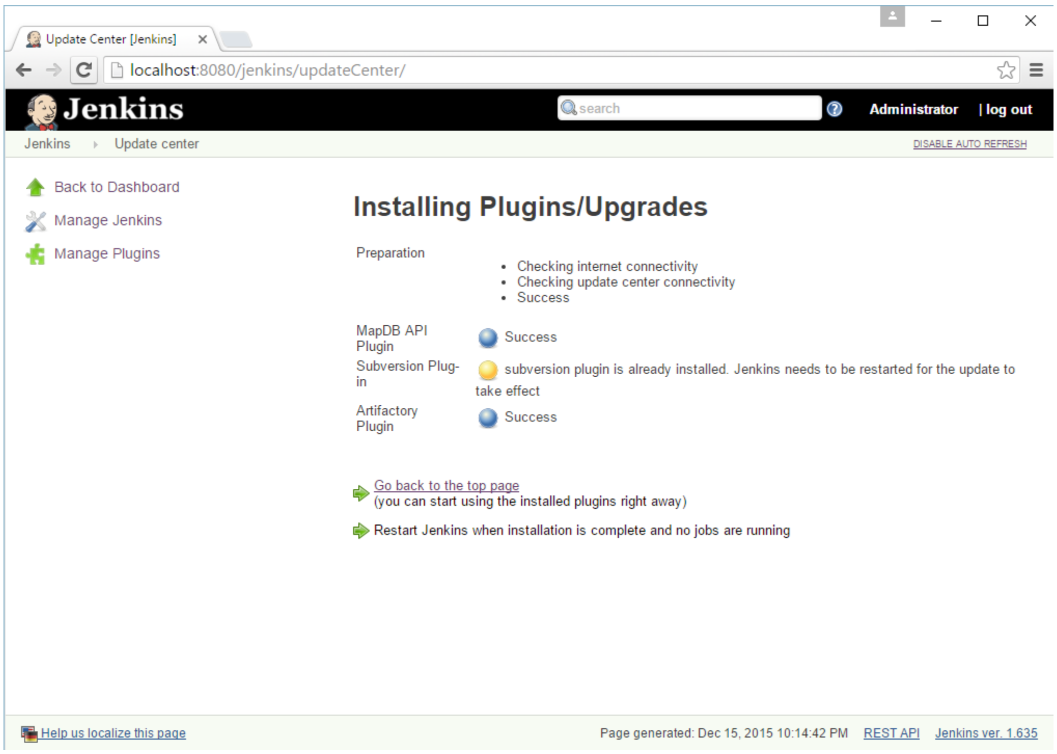
Installing the Artifactory plugin

To install the Artifactory plugin, perform the following steps:

1. From the Jenkins Dashboard, click on the **Manage Jenkins** link. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `Artifactory Plugin` in the search box. Select **Artifactory Plugin** from the list and click on the **Install without restart** button.

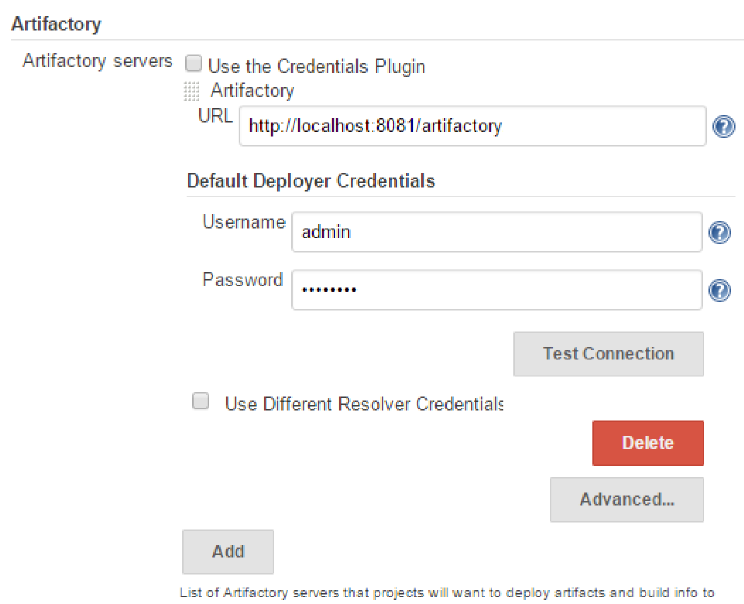


4. The download and installation of the plugin starts automatically. You can see the Artifactory Plugin has some dependencies that get downloaded and installed.

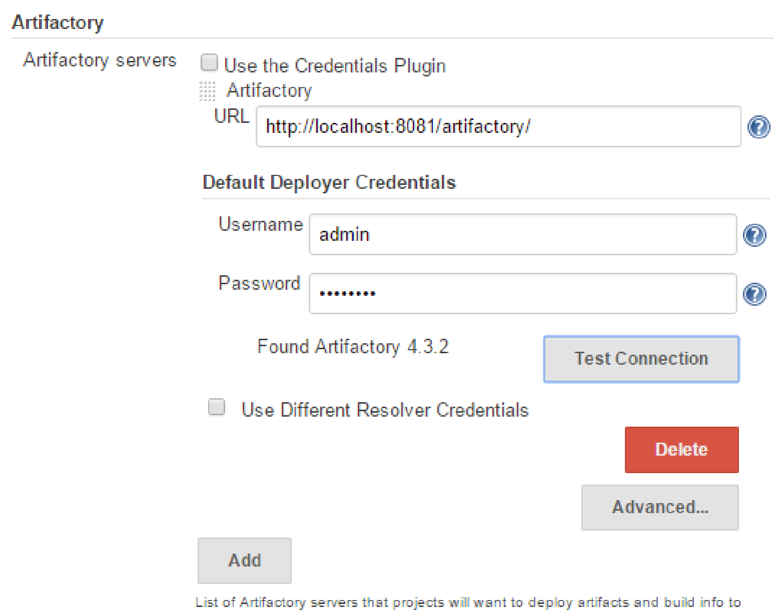


5. Upon successful installation of the Artifactory Plugin, go to the **Configure System link** on the **Manage Jenkins** page.

6. Scroll down until you see the **Artifactory** section and fill in the blanks as shown here:
    - Provide the **URL** field as the default Artifactory URL configured at the time of installation. In our example, it is `http://localhost:8081/artifactory`.
    - In the **Default Deployer Credentials** field, provide the values for **Username** and **Password**.



7. That's it. You can test the connection by clicking on the **Test Connection** button. You should see your Artifactory version displayed, as shown in the following screenshot:



8. Save the configuration by clicking on the **Save** button at the bottom of the screen.

The Jenkins pipeline to poll the integration branch

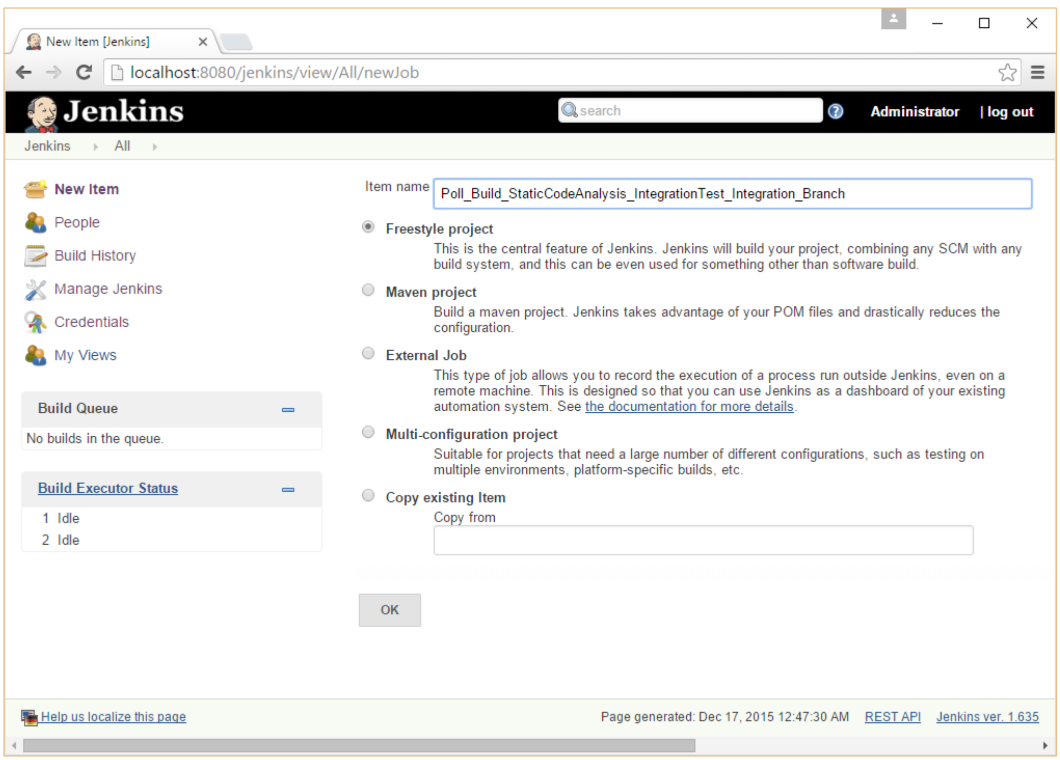
This is the second pipeline of the two, both of which are part of the CI pipeline structure discussed in the previous chapter. This pipeline contains two Jenkins jobs. The first Jenkins job does the following tasks:

- It polls the integration branch for changes at regular intervals
- It executes the static code analysis
- It performs a build on the modified code
- It executes the integration tests

Creating a Jenkins job to poll, build, perform static code analysis, and integration tests

I assume you are logged in to Jenkins as an admin and have privileges to create and modify jobs. From the Jenkins Dashboard, follow these steps:

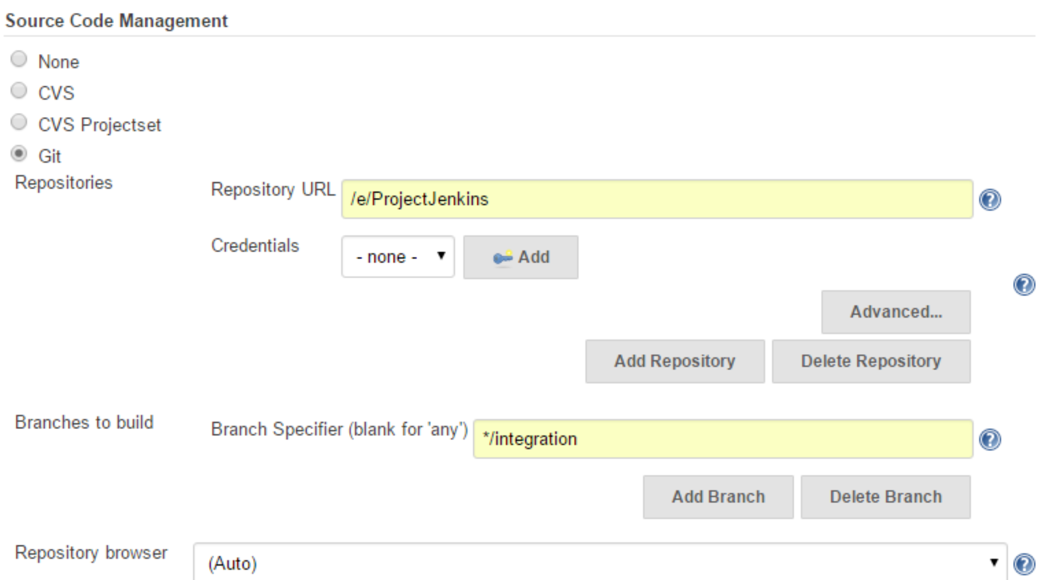
1. Click on **New Item**.
2. Name your new Jenkins job `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`.
3. Set the type of job as **Freestyle project** and click on **OK** to proceed.



Polling the version control system for changes using Jenkins

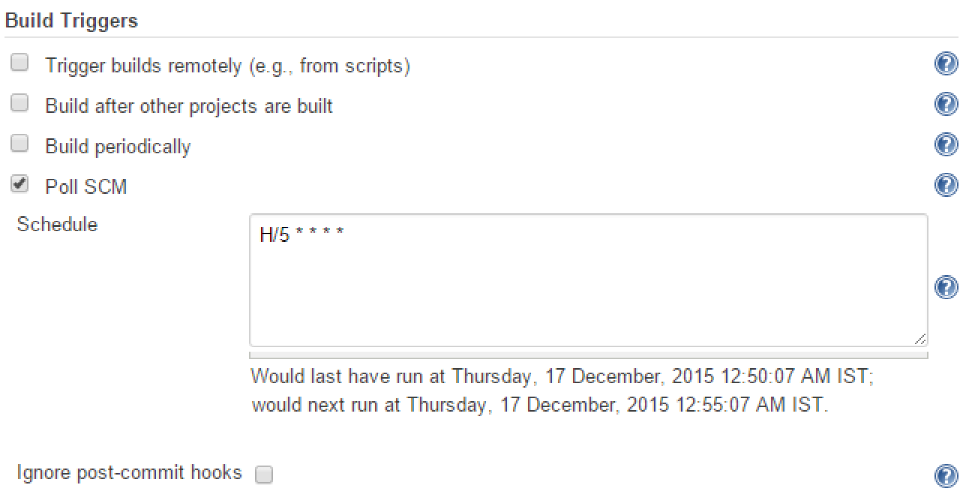
This is a critical step in which we connect Jenkins with the version control system. This configuration enables Jenkins to poll the correct branch inside Git and download the modified code.

1. Scroll down to the **Source Code Management** section.
2. Select **Git** and fill in the blanks as follows:
    - Specify **Repository URL** as the location of the Git repository. It can be a GitHub repository or a repository on a Git server. In our case, it's `/e/ProjectJenkins` because the Jenkins server and the Git server is on the same machine.
    - Add `*/integration` in the **Branch to build** section, since we want our Jenkins job to poll integration branch. Leave rest of the fields as they are.



3. Scroll down to the **Build Triggers** section.

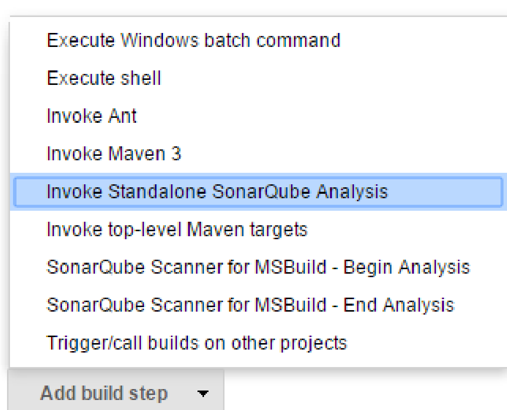
4. We want our Jenkins job to poll the feature branch every 5 minutes. Nevertheless, you are free to choose the polling duration that you wish depending on your requirements. Therefore, select the **Poll SCM** checkbox and add `H/5 * * * *` in the **Schedule** field.



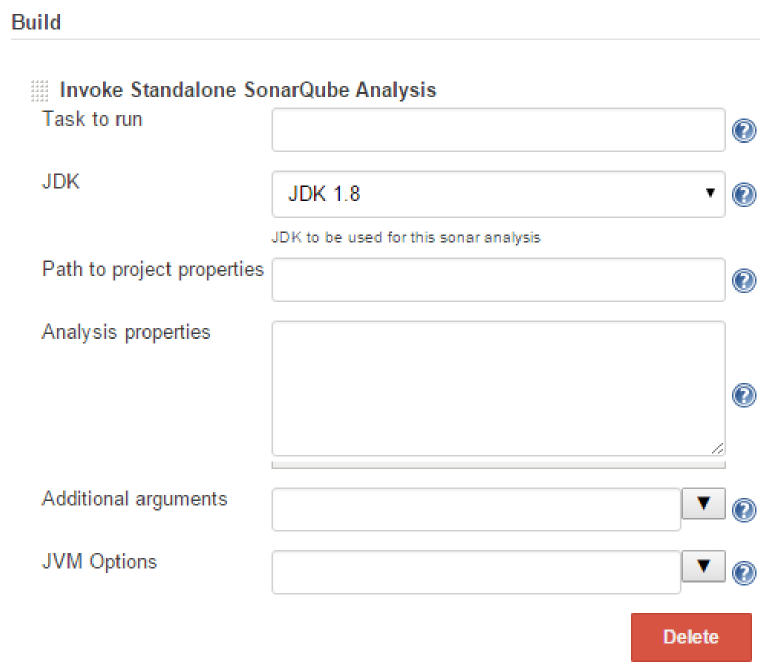
Creating a build step to perform static analysis

The following configuration tell Jenkins to perform a static code analysis on the downloaded code:

1. Scroll down to the **Build** section and click on the **Add build step** button. Select **Invoke Standalone SonarQube Analysis**.



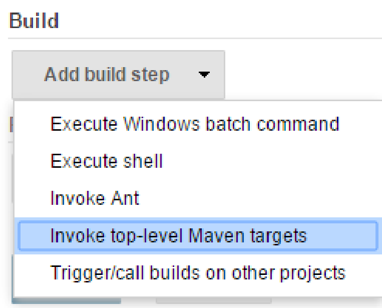
2. Leave all the fields empty except the **JDK** field. Choose the appropriate version from the menu. In our example, it's **JDK 1.8**.



Creating a build step to build and integration test code

After successfully completing the static code analysis using SonarQube, the next step is to build the code and perform integration testing:

1. Click on the **Add build step** button again. Select **Invoke top-level Maven targets**.

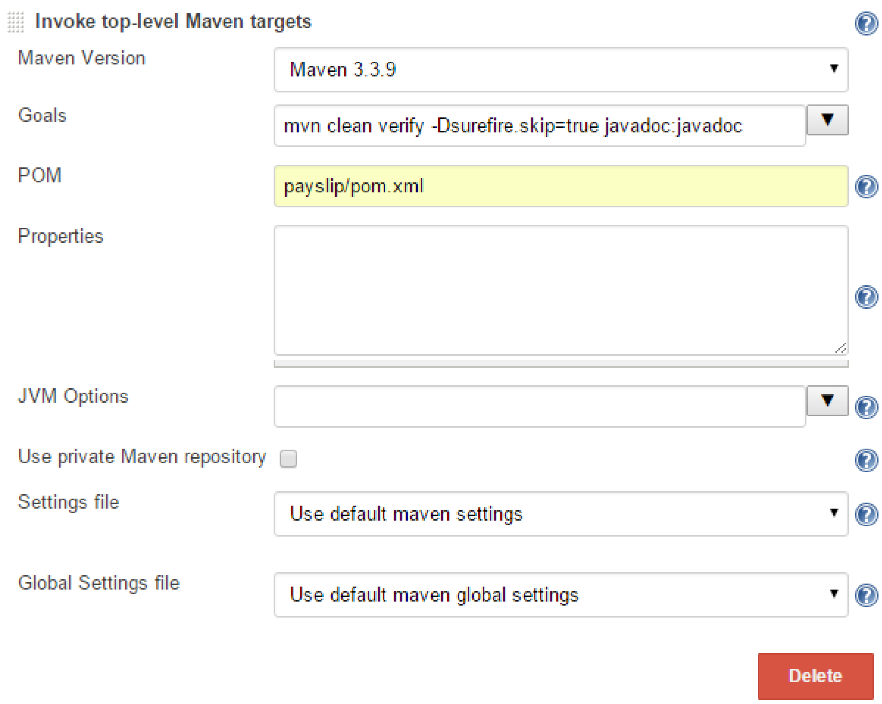


2. We will be present with the following options:
    - Set the **Maven Version** field as **Maven 3.3.9**. Remember, this is what we configured on the **Configure System** page in the **Maven** section. If we had configured more than one Maven, we would have a choice here.
    - Add the following line to the **Goals** section:

      ```
      clean verify -Dsurefire.skip=true javadoc:javadoc
      ```

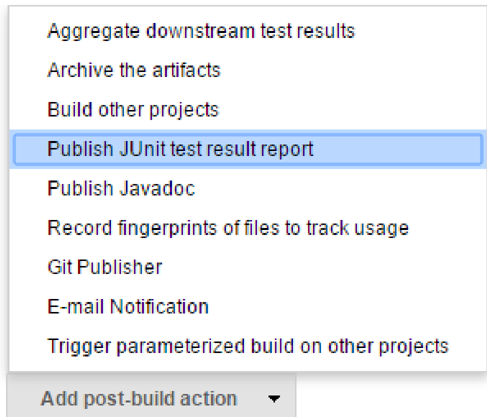
    - Type `payslip/pom.xml` in the **POM** field. This tells Jenkins the location of the `pom.xml` file in the downloaded code.

3. The following screenshot displays the **Invoke top-level Maven targets** window and the mentioned fields:

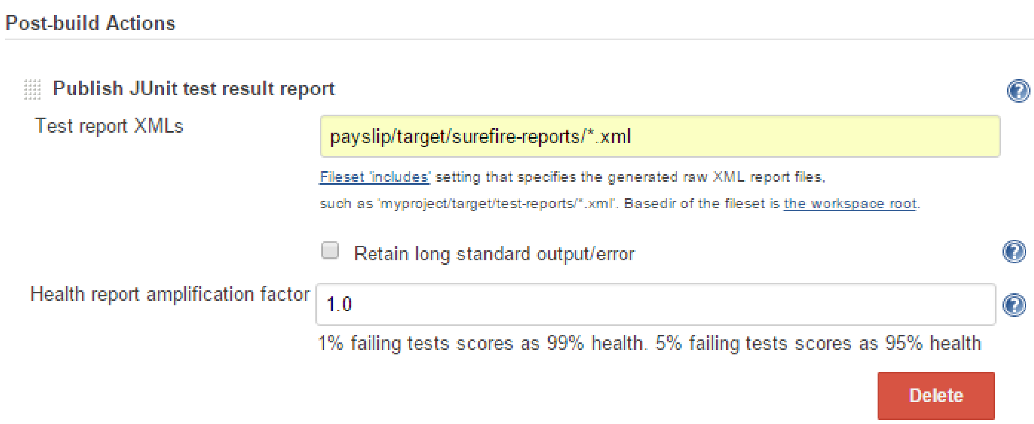


4. Let's see the Maven command inside the **Goals** field in detail:
   - `clean` will clean any old built files
   - `-Dsurefire.skip=true` will execute the integration test
   - `javadoc:javadoc` will tell Maven to generate Java documentation
5. Scroll down to the **Post build Actions** section.

6. Click on the **Add post-build action** button and select **Publish JUnit test result report**, as shown in the following screenshot:

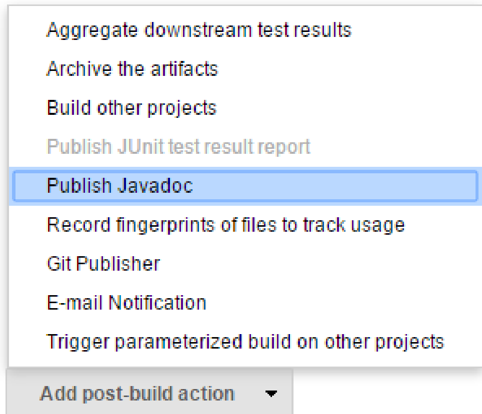


7. Under the **Test report XMLs** field, type `payslip/target/surefire-reports/*.xml`.

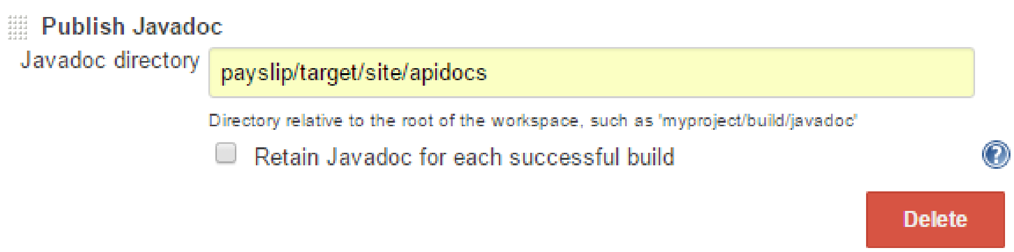


> This is the location where the unit test reports will get generated once the code is built and unit tested.
>
> Jenkins will access all the `*.xml` files present under the `payslip/target/surefire-reports` directory and publish the report. We will see this when we run this Jenkins job.

8. Next, click on the **Add post-build action** button. This time, select **Publish Javadoc**.



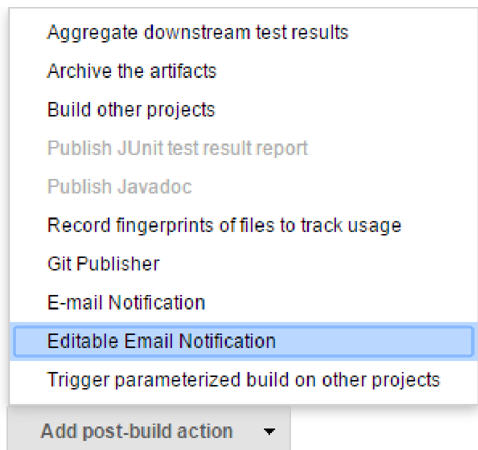
9. Type the path `payslip/target/site/apidocs` under the **Javadoc directory** field.



Configuring advanced e-mail notifications

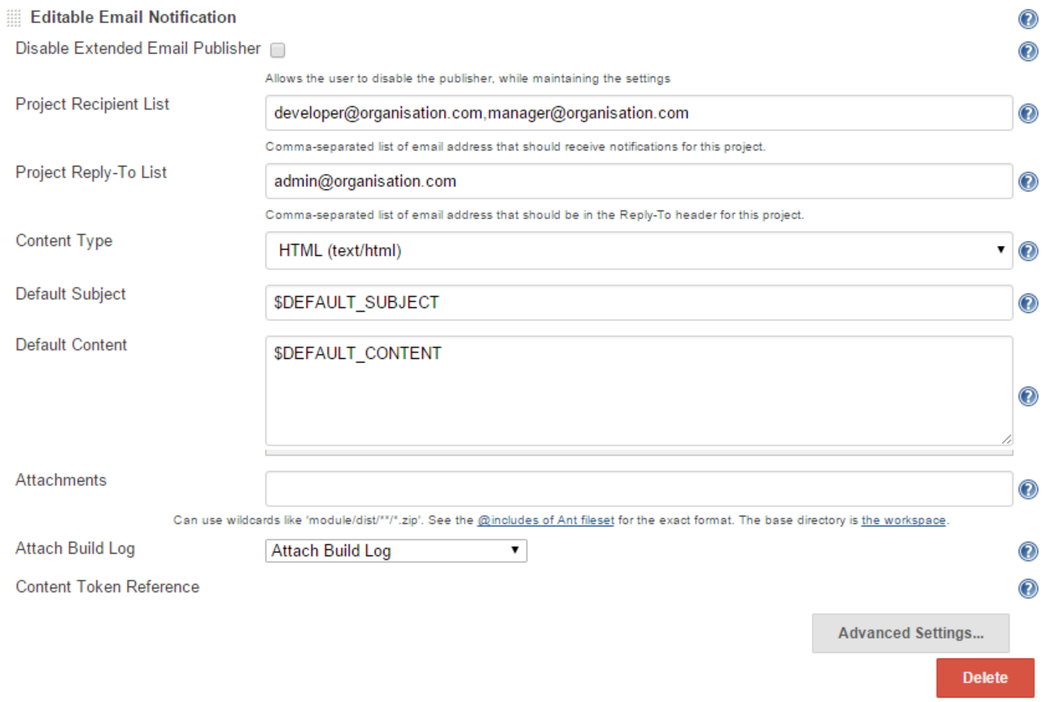
Notification forms are an important part of CI. In this section, we will configure the Jenkins job to send e-mail notifications based on few conditions. Let's see the steps in detail:

1. Click on the **Add post-build action** button and select **Editable Email Notification**.



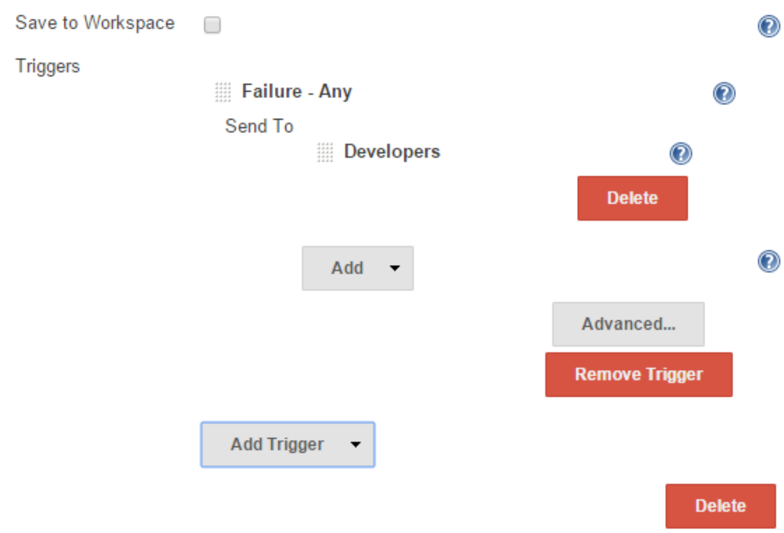
2. Configure **Editable Email Notification** as shown here:
    - Under **Project Recipient List**, add the e-mail IDs separated by commas. You can add anyone whom you think should be notified for build and unit test success/failure.
    - You can add the e-mail ID of the Jenkins administrator under **Project Reply-To List**.
    - Set **Content Type** as **HTML (text/html)**.

3. Leave all the rest of the options at their default values.

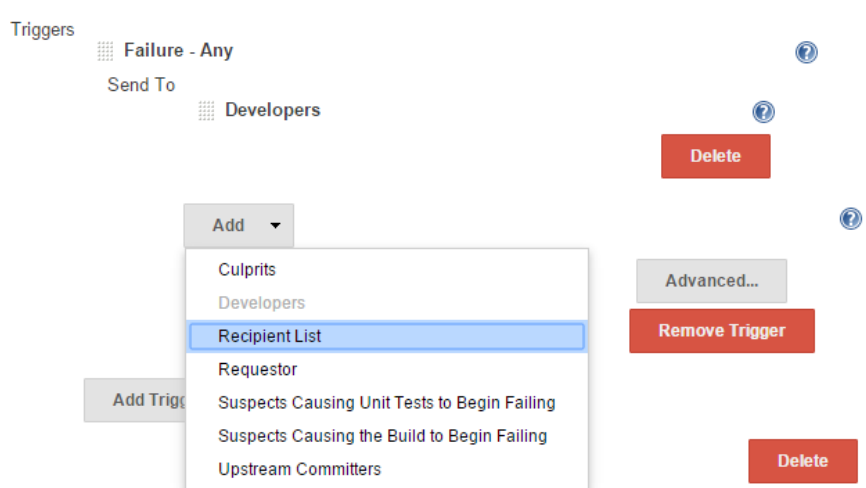


4. Now, click on the **Advanced Settings...** button.
5. By default, there is a trigger named **Failure – Any** that sends e-mail notifications in the event of failure (any kind of failure).

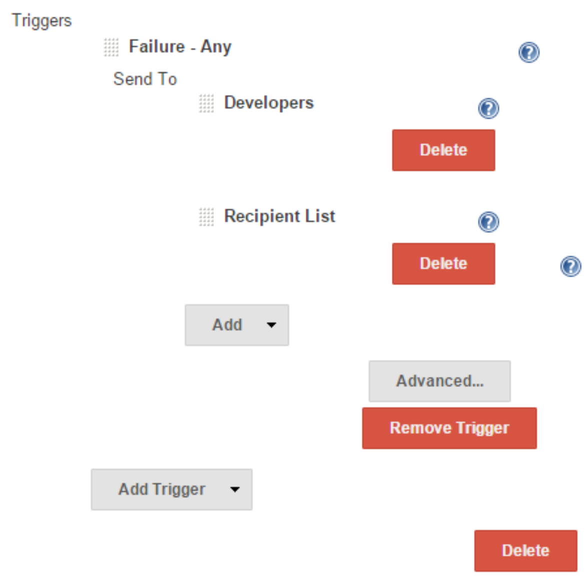
6. By default, the **Send To** option is set to **Developers**.



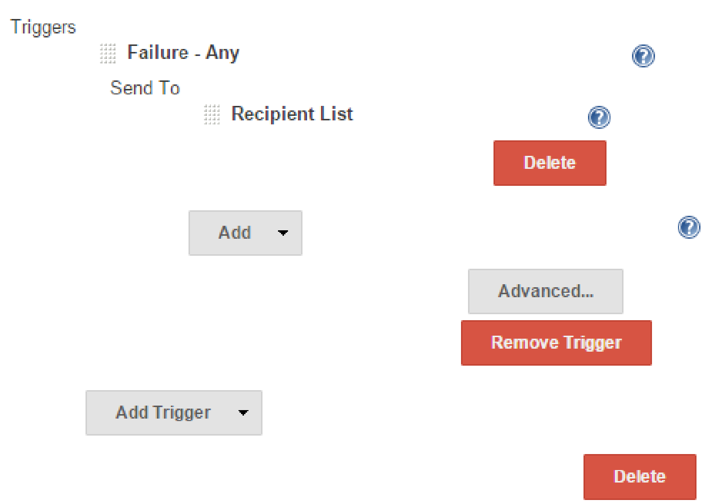
7. But we don't want that, we have already defined whom to send e-mails to. Therefore, click on the **Add** button and select the **Recipient List** option, as shown in the following screenshot:



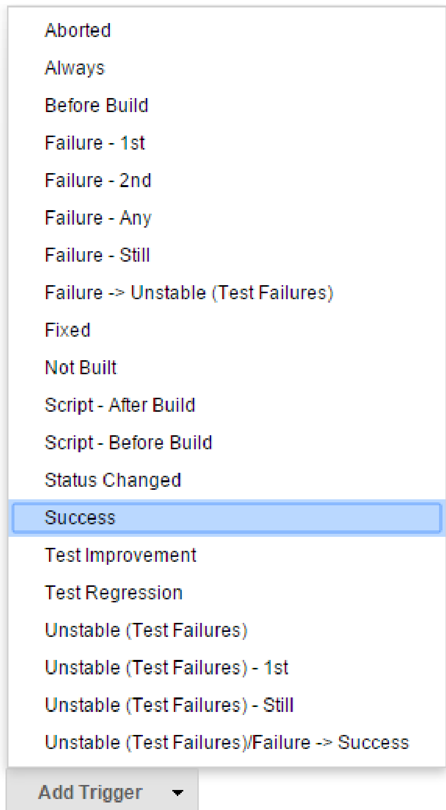
8. The result will look something like this:



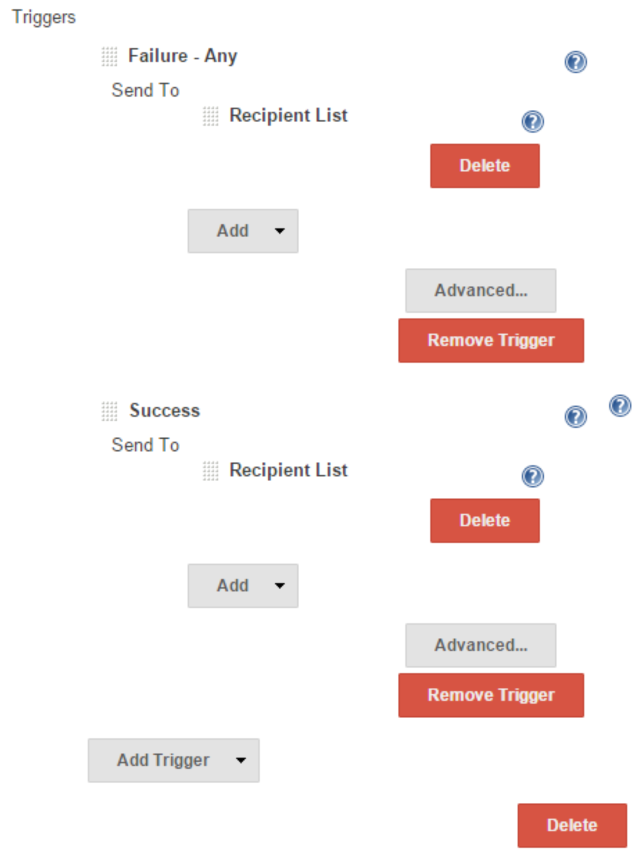
9. Delete **Developers** from the **Send To** section by clicking on the **Delete** button adjacent to it. The result should look something like this:



10. Let's add another trigger to send an e-mail when the job is successful.
11. Click on the **Add Trigger** button and select the **Success** option.



12. Configure this new success trigger in the similar fashion by removing **Developers** and adding **Recipient List** under the **Send To** section. Finally, everything should look like this:



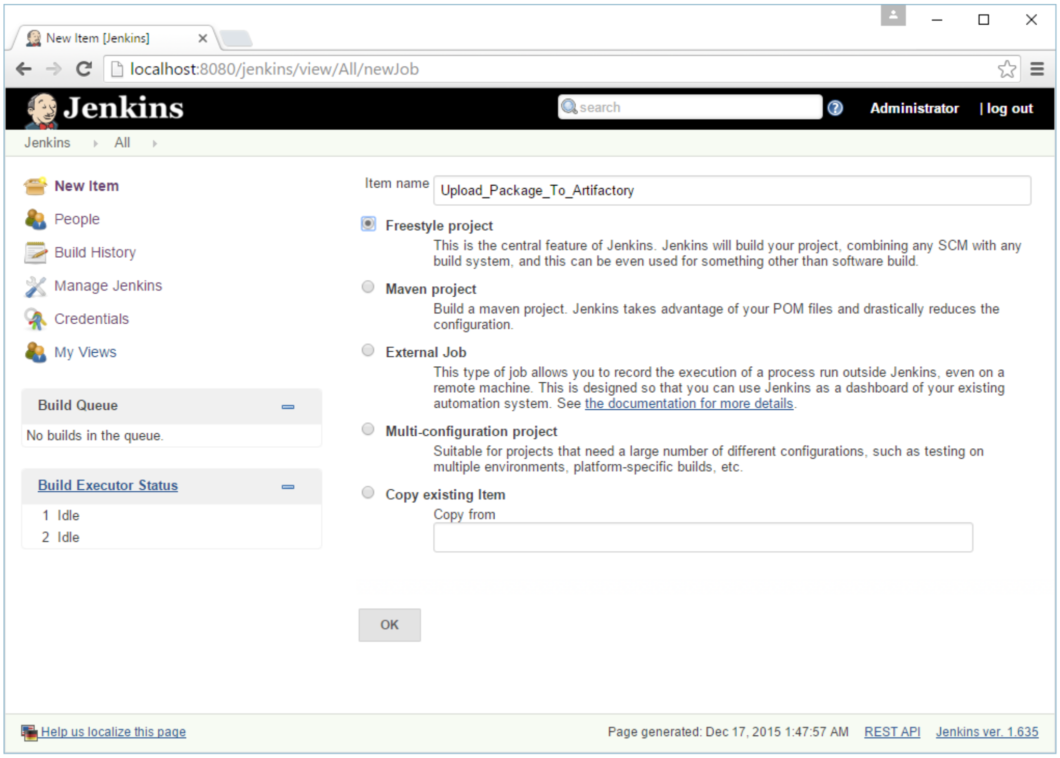
13. Save the Jenkins job by clicking on the **Save** button.

Creating a Jenkins job to upload code to Artifactory

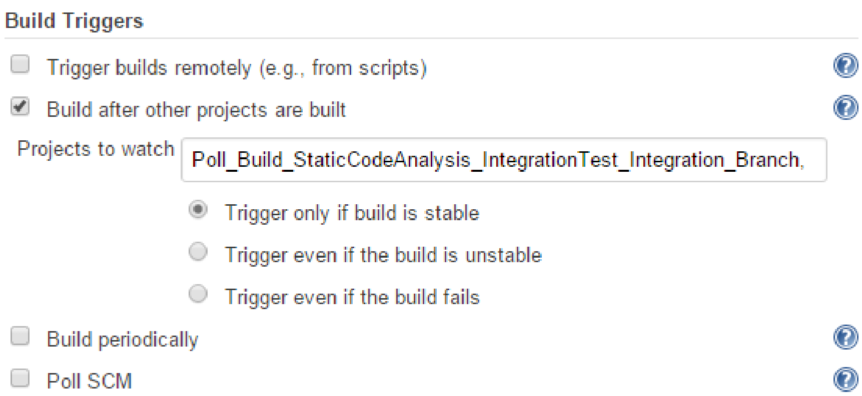
The second Jenkins job in the pipeline uploads the build package to Artifactory (binary code repository). From the Jenkins Dashboard:

1. Click on **New Item**.
2. Name your new Jenkins job `Upload_Package_To_Artifactory`.

3. Select the type of job as **Freestyle project** and click on **OK** to proceed.



4. Scroll down to the **Build Triggers** section and select the **Build after other projects are built** option.
5. Under **Projects to watch** field, type `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`.
6. Select the **Trigger only if build is stable** option.



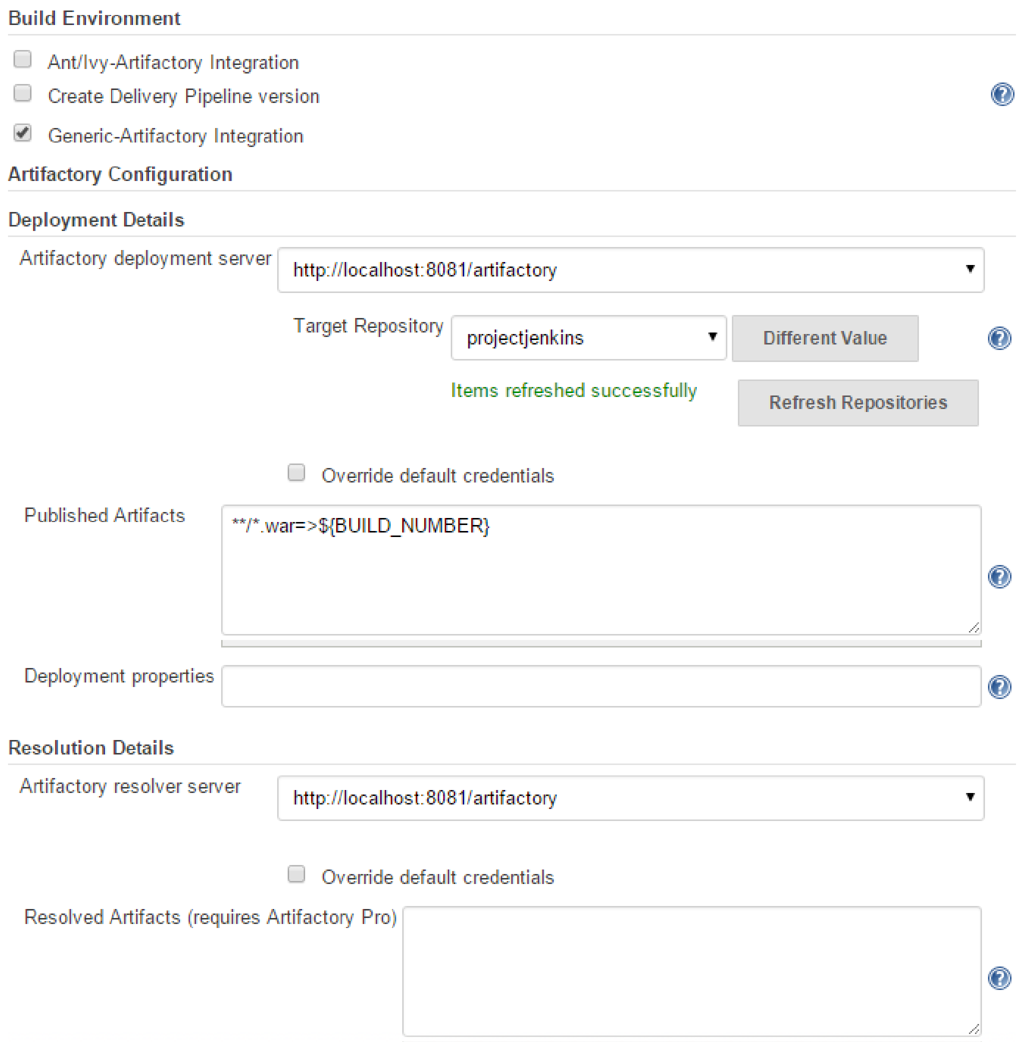
> In this way, we are telling Jenkins to initiate the current Jenkins job `Upload_Package_To_Artifactory` only after the `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch` job has completed successfully.

Configuring the Jenkins job to upload code to Artifactory

The following configuration will tell Jenkins to look for a potential `.war` file under the Jenkins job's workspace to upload it to Artifactory:

1. Scroll down further until you see the **Build Environment** section. Check the **Generic-Artifactory Integration** option. Doing so will display a lot of options for Artifactory. Fill them in as follows:
    - **Artifactory deployment server** is your Artifactory web link. In our case, it is `http://localhost:8081/artifactory`.
    - Next is the **Target Repository** field. Select **projectjenkins** from the drop-down menu. You will notice that all the repositories present inside Artifactory will be listed here.
    - To refresh the list, click on the **Refresh Repositories** button.
    - Add `**/*.war=>${BUILD_NUMBER}` to the **Published Artifacts** field.

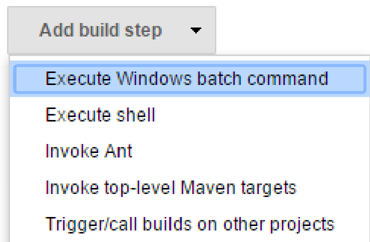
- Leave rest of the fields at their default values.



2. Let's see what the **Published Artifacts** field means.
    - `**/*.war` tells Jenkins to search for and pick a WAR file anywhere inside the current workspace
    - `${BUILD_NUMBER}` is a Jenkins variable that stores the current build number

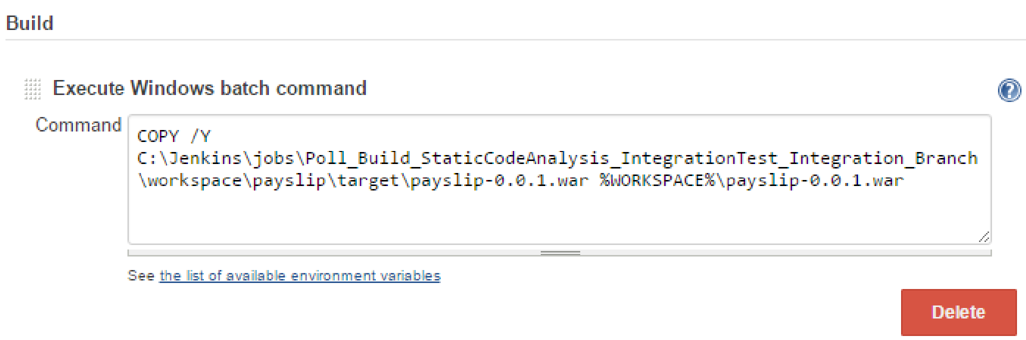
    ◦ Finally, `**/*.war=>${BUILD_NUMBER}` means *search and pick any .war file present inside the workspace, and upload it to Artifactory with the current build number as its label*

3. Scroll down to the **Build** section and add a build step to **Execute Windows batch command**.



4. Add the following code into the **Command** section:

```
COPY /Y
C:\Jenkins\jobs\Poll_Build_StaticCodeAnalysis_
IntegrationTest_Integration_Branch\workspace\payslip\target
\payslip-0.0.1.war %WORKSPACE%\payslip-0.0.1.war
```



> This simply copies the `payslip-0.0.1.war` package file generated in the previous Jenkins job from its respective workspace to the current job's workspace. This build step happens first and then the upload to Artifactory takes place.

5. Configure advanced e-mail notifications exactly the same way as mentioned earlier.
6. Save the Jenkins job by clicking on the **Save** button.

Creating a nice visual flow for the Continuous Integration pipeline

So far, we have created around six Jenkins jobs in total, segregated across three Jenkins pipelines:

- Pipeline to poll the `Feature1` branch:
    - `Poll_Build_UnitTest_Feature1_Branch`
    - `Merge_Feature1_Into_Integration_Branch`
- Pipeline to poll the `Feature2` branch:
    - `Poll_Build_UnitTest_Feature2_Branch`
    - `Merge_Feature2_Into_Integration_Branch`
- Pipeline to poll the integration branch:
    - `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`
    - `Upload_Package_To_Artifactory`

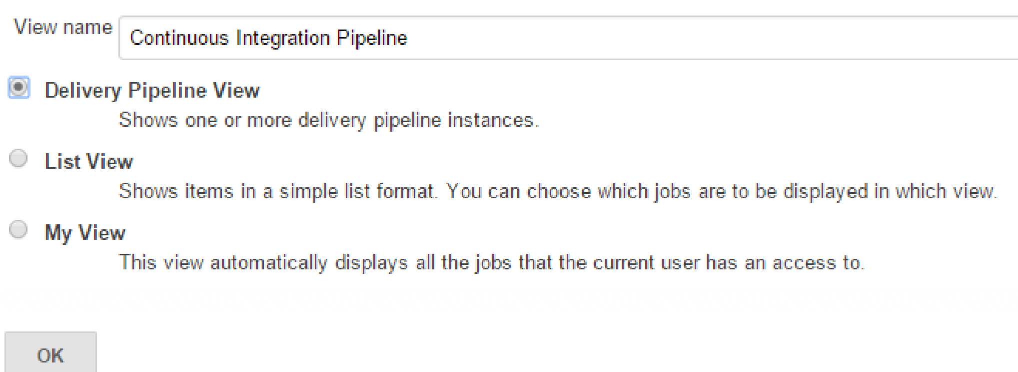
All the three pipelines combined complete our CI Design.

> There were actually two Jenkins pipelines discussed as part of our CI Design. However, we have three now. This is just because we have two feature branches; we still have two types of Jenkins pipeline.

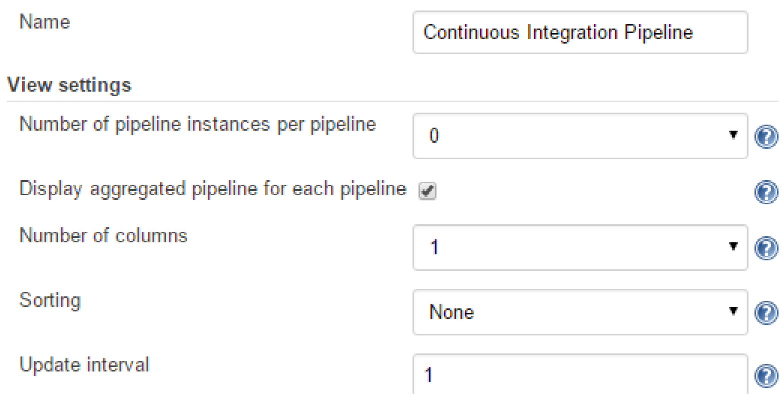
In this section, we will create a view inside the Jenkins Dashboard using the *delivery pipeline plugin*. This view is a nice way of presenting the CI flow. The same plugin will also be used to create a **Continuous Delivery Flow** (**CD**). The steps are as follows:

1. Go to Jenkins Dashboard and click on the plus button as highlighted in the following screenshot:

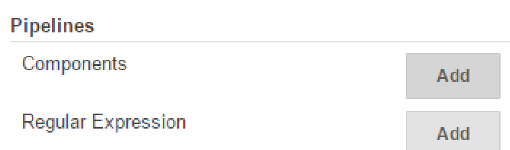
2. Type `Continuous Integration Pipeline` as the **View name** and select **Delivery Pipeline View** from the options, as shown in the following screenshot:
3. Click on **OK** to finish.



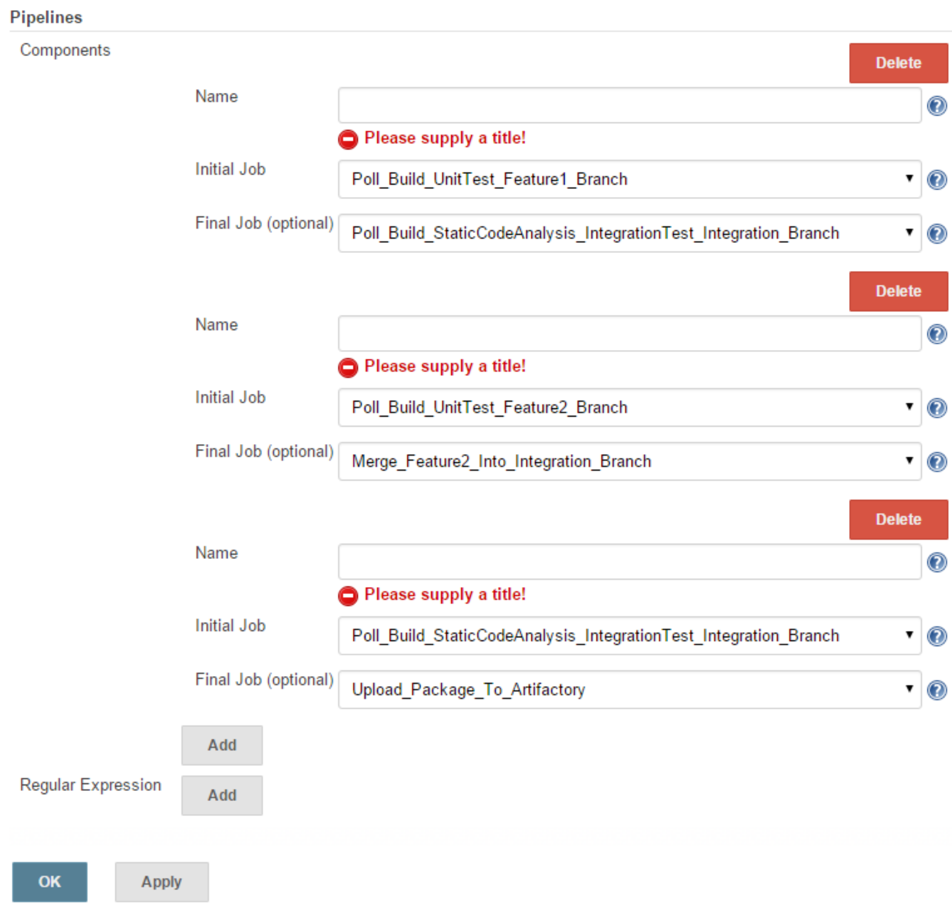
4. Now, you will see a lot of options (mentioned in the following list) and blanks to fill in. Scroll down until you see the **View settings** section:
   - Set the **Number of pipeline instances per pipeline** field as **0**.
   - Set the **Number of columns** field as **1**.
   - Set the **Update interval** field as **1**.
   - Check the **Display aggregated pipeline for each pipeline** option.
   - Leave rest of the options at their default values.



5. Scroll down until you see the **Pipelines** section.
6. Click thrice on the **Add** button besides the **Components** option.

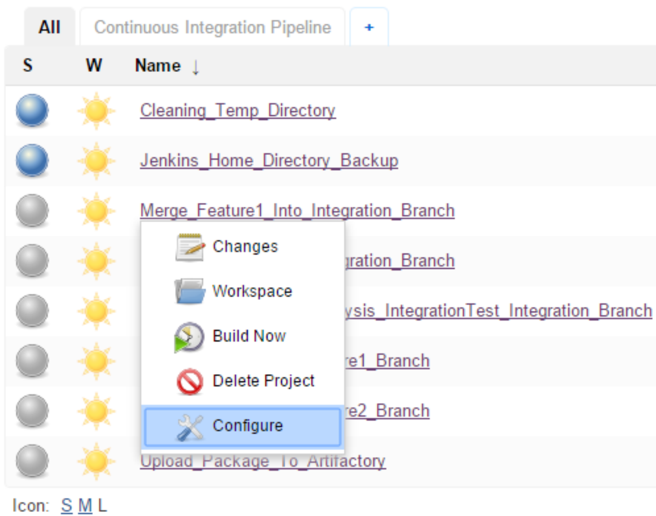


7. Fill in the options exactly as shown in the following screenshot:

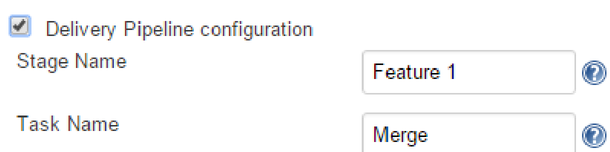


8. Click on **OK** to save the configuration.
9. Now, come back to the Jenkins Dashboard.

10. Right-click on the **Merge_Feature1_Into_Integration_Branch** Jenkins job and select **Configure**.

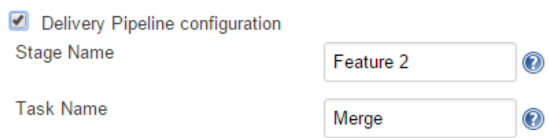


11. Look for the **Delivery Pipeline configuration** option and select it.
12. Set **Stage Name** as `Feature 1` and **Task Name** as `Merge`.

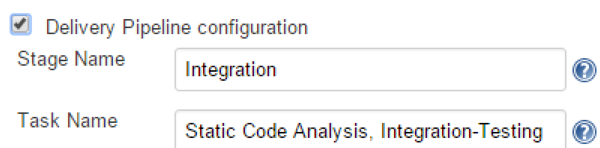


13. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
14. Now, come back to the Jenkins Dashboard.
15. Right-click on the **Merge_Feature2_Into_Integration_Branch** Jenkins job and select **Configure**.
16. Look for the **Delivery Pipeline configuration** option and select it.

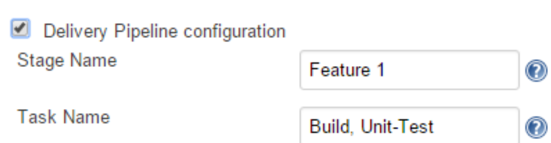
17. Set **Stage Name** as `Feature 2` and **Task Name** as `Merge`.



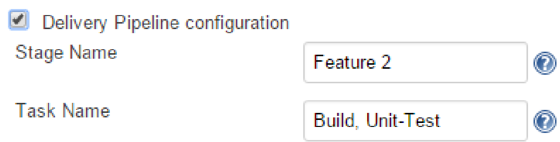
18. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
19. Come back to the Jenkins Dashboard.
20. Right-click on the **Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch** Jenkins job and select **Configure**.
21. Look for the **Delivery Pipeline configuration** option and select it.
22. Set **Stage Name** as `Integration` and **Task Name** as `Static Code Analysis, Integration-Testing`.



23. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
24. Come back to the Jenkins Dashboard.
25. Right-click on the **Poll_Build_UnitTest_Feature1_Branch** Jenkins job and select **Configure**.
26. Look for the **Delivery Pipeline configuration** option and select it.
27. Set **Stage Name** as `Feature 1` and **Task Name** as `Build, Unit-Test`.

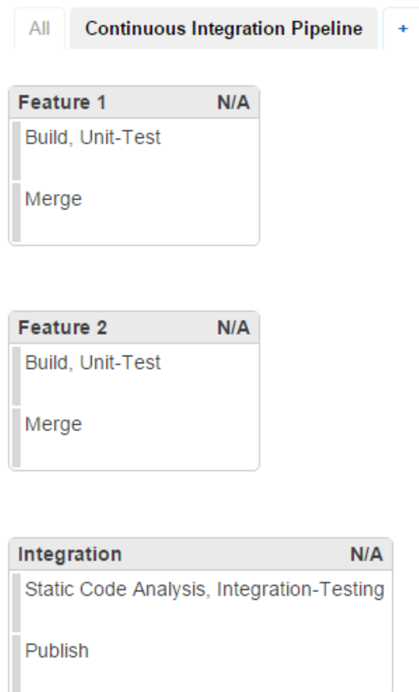


28. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
29. Come back to the Jenkins Dashboard.
30. Right-click on the **Poll_Build_UnitTest_Feature2_Branch** Jenkins job and select **Configure**.
31. Look for the **Delivery Pipeline configuration** option and select it.
32. Set **Stage Name** as `Feature 2` and **Task Name** as `Build, Unit-Test`.



33. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
34. Come back to the Jenkins Dashboard.
35. Right-click on the **Upload_Package_To_Artifactory** Jenkins job and select **Configure**.
36. Look for the **Delivery Pipeline configuration** option and select it.
37. Set **Stage Name** as `Integration` and **Task Name** as `Publish`.
38. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.

39. Come back to the Jenkins Dashboard and click on the **Continuous Integration Pipeline** view. Tada!! Here's what you will see:



> The pipeline may not appear to be continuous and connected visually because the so-called **Delivery Pipeline Plugin** only groups the Jenkins jobs that are connected through triggers.

Continuous Integration in action

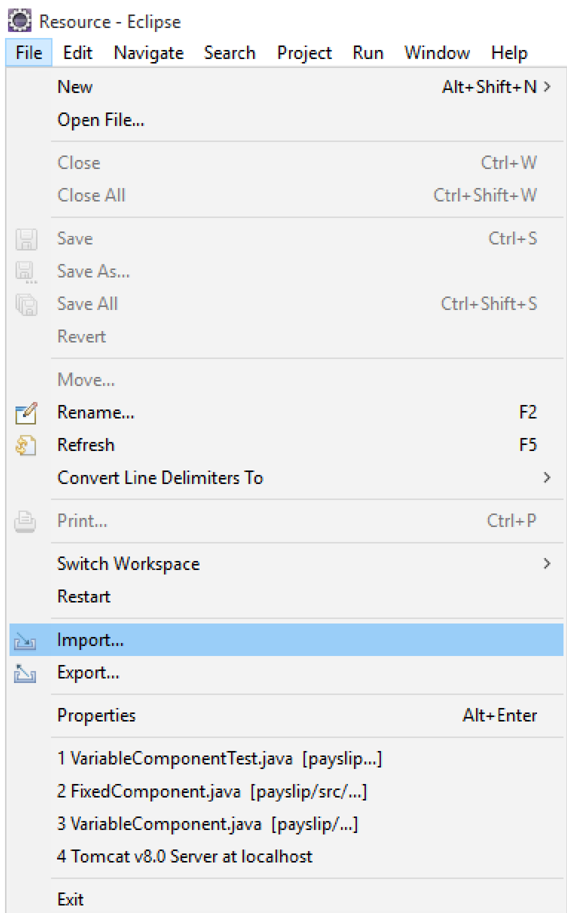
Let's assume the role of a developer who intends to work on the `Feature1` branch. Our developer is working on a Windows 10 machine with the following software installed on it:

- Latest version of Eclipse (Eclipse Mars)
- Apache Tomcat server 8
- Git 2.6.3
- SourceTree
- Java JDK 1.8.0_60
- Maven 3.3.9

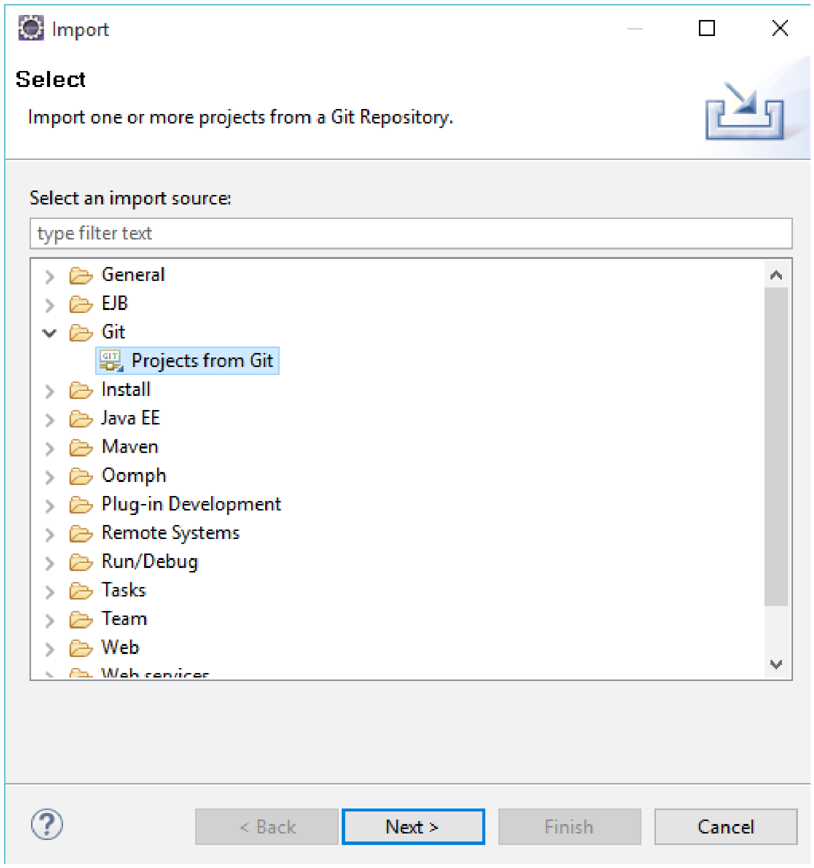
Configuring Eclipse to connect with Git

We will first see how to connect Git with Eclipse so that the developer can work seamlessly without jumping between Eclipse and Git.

1. Open Eclipse and select **File** | **Import...** from the menu option.

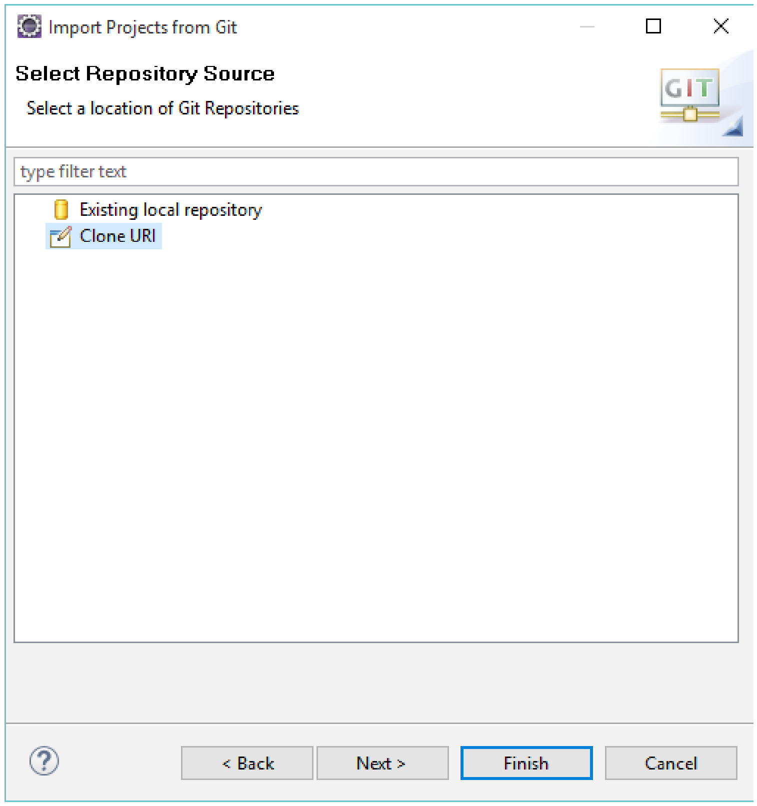


2. The **Import** window appears. Select the **Projects from Git** option under **Git**, as shown in the following screenshot:



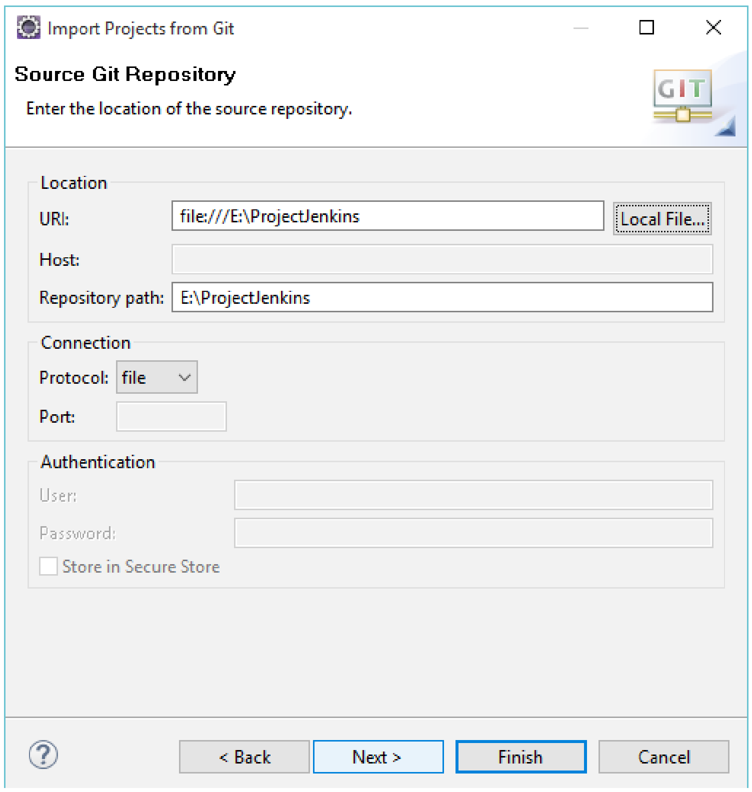
3. Click on **Next**.

4. Select the **Clone URI** option.



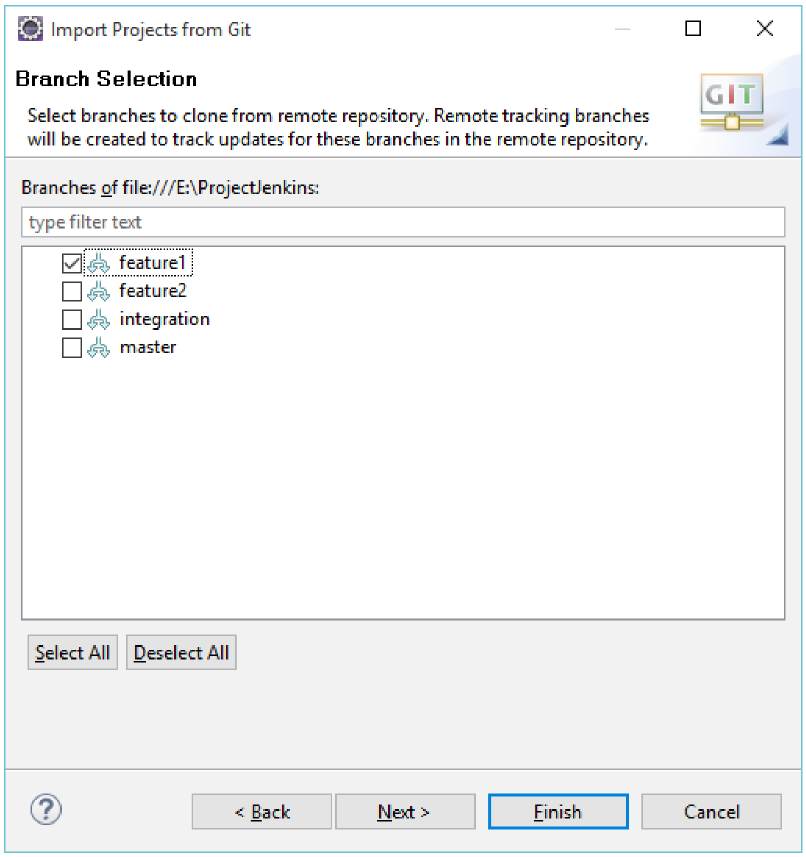
5. Click on **Next**.

6. Now, we need to provide the link to the Git source repository. In an ideal situation, the Git server resides on a separate machine. Therefore, we should provide the link of the Git source repository in the **URI** field. However, if the Git repository is on the same machine, we click on the **Local File...** button and select the local folder containing the Git source repository.



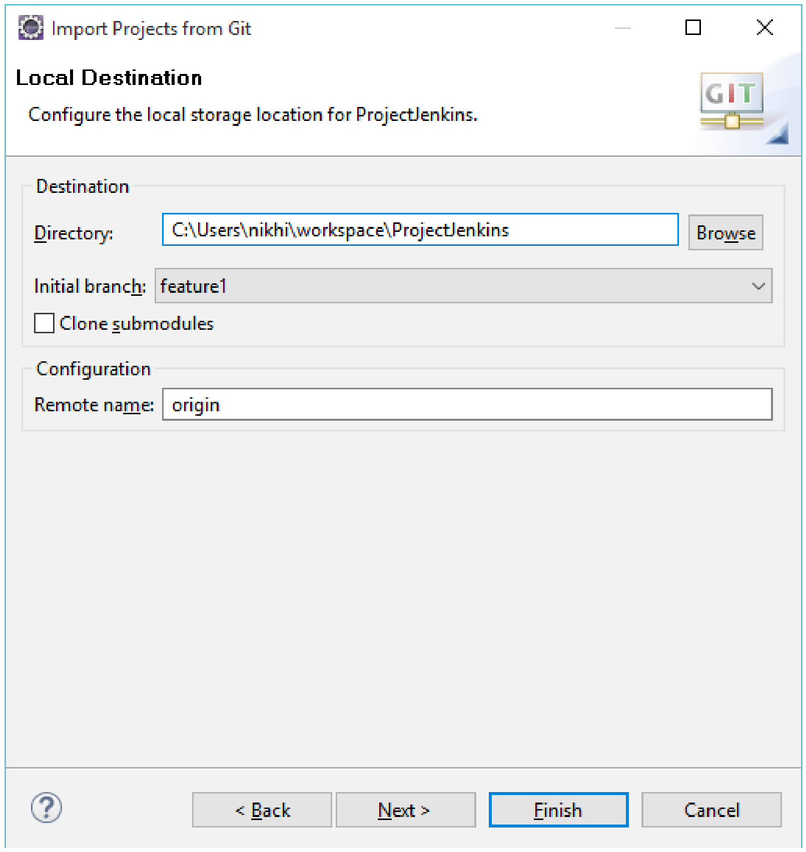
7. Click on **Next**.

8. Select the branch that you want to clone. Since we are performing this example from the perspective of a developer who works on the `feature1` branch, we select the `feature1` branch from the branches as shown in the next screenshot:



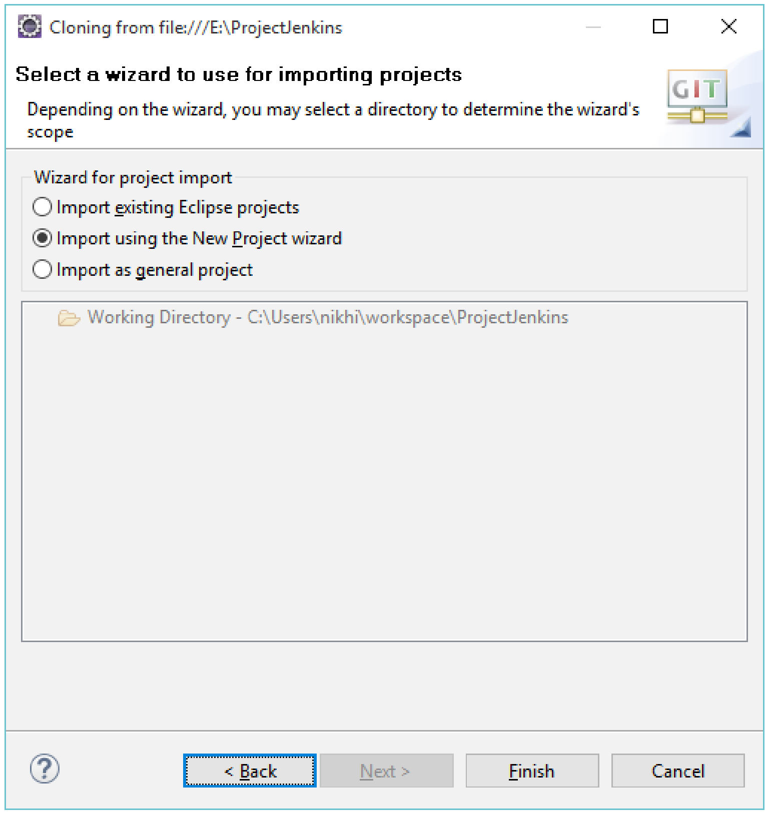
9. Click on **Next**.

10. Here, we get to choose the local directory path where we would wish to keep the cloned `feature1` branch. I have chosen the folder `ProjectJenkins`, which is inside the Eclipse workspace.



11. Click on **Next**.

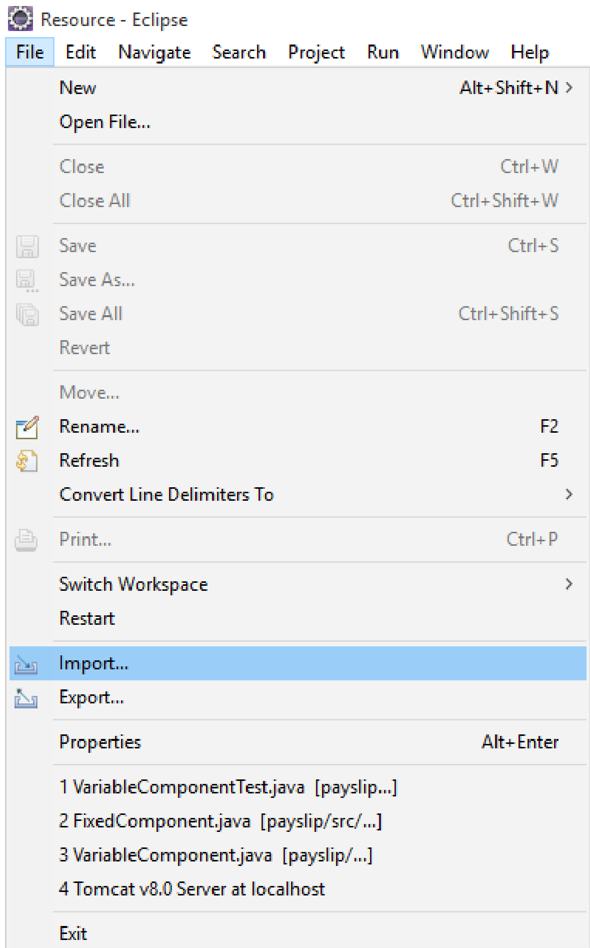
12. Select the **Import using the New Project wizard** option.



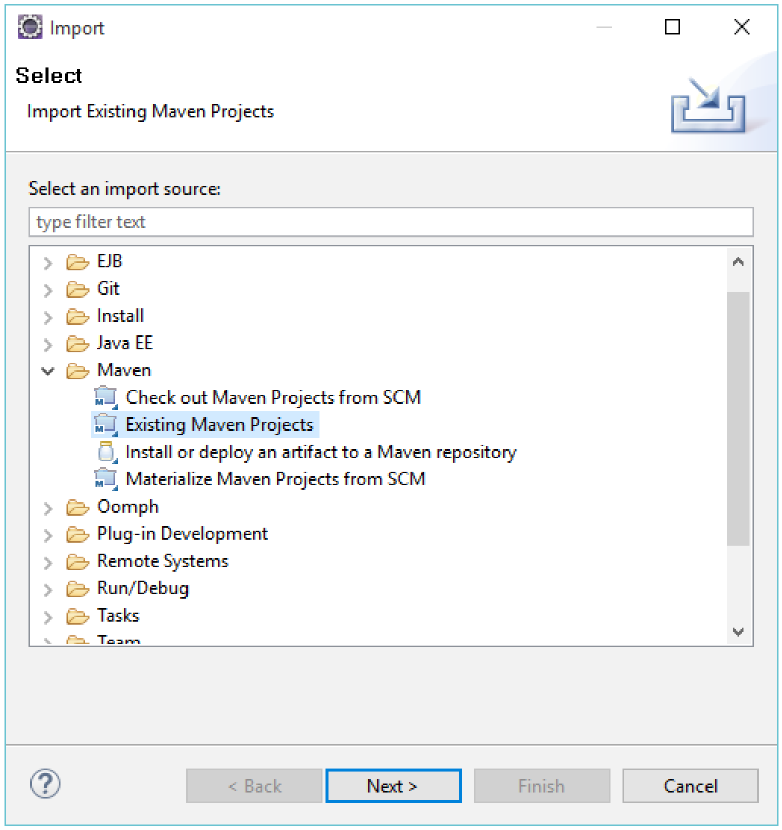
13. Click on the **Finish** button.

> A window might appear once you click on the **Finish** button. Ignore and close it.

14. From the menu options, select **File** | **Import...**.

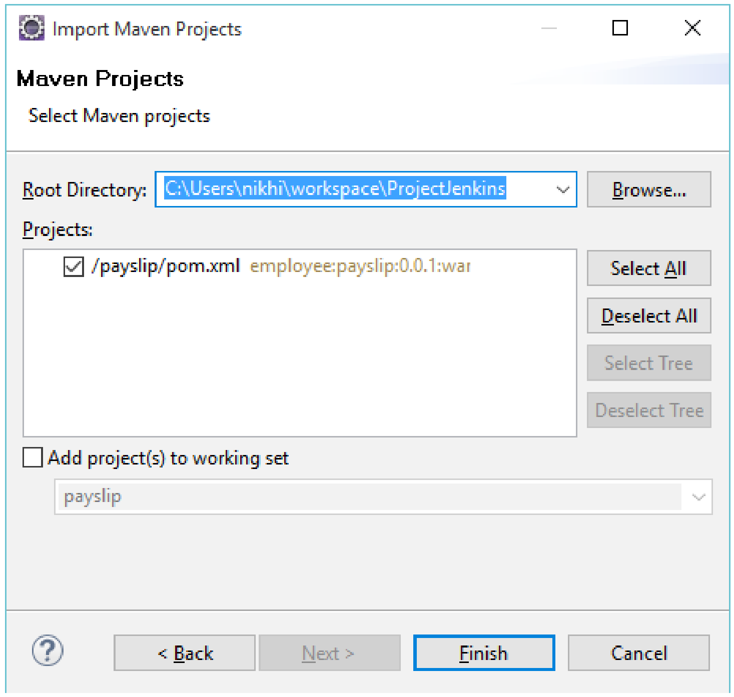


15. This time, select **Existing Maven Projects** under **Maven**.



16. Click on **Next**.
17. Click on the **Browse...** button to navigate to the folder inside the Eclipse workspace where we kept our cloned copy of the Git repository.

18. Select the `/payslip/pom.xml` option and click on the **Finish** button.

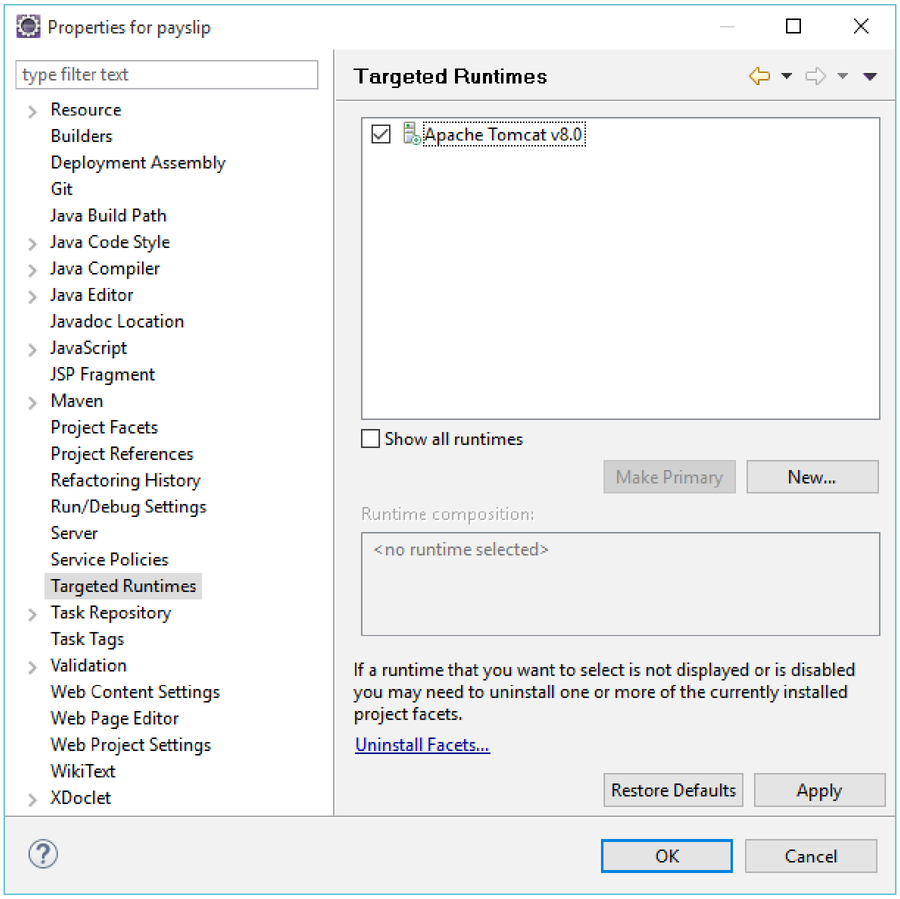


Adding a runtime server to Eclipse

Our application is hosted on an Apache Tomcat server. Therefore, let's configure the Apache Tomcat server with Eclipse so that we can quickly test the changes:

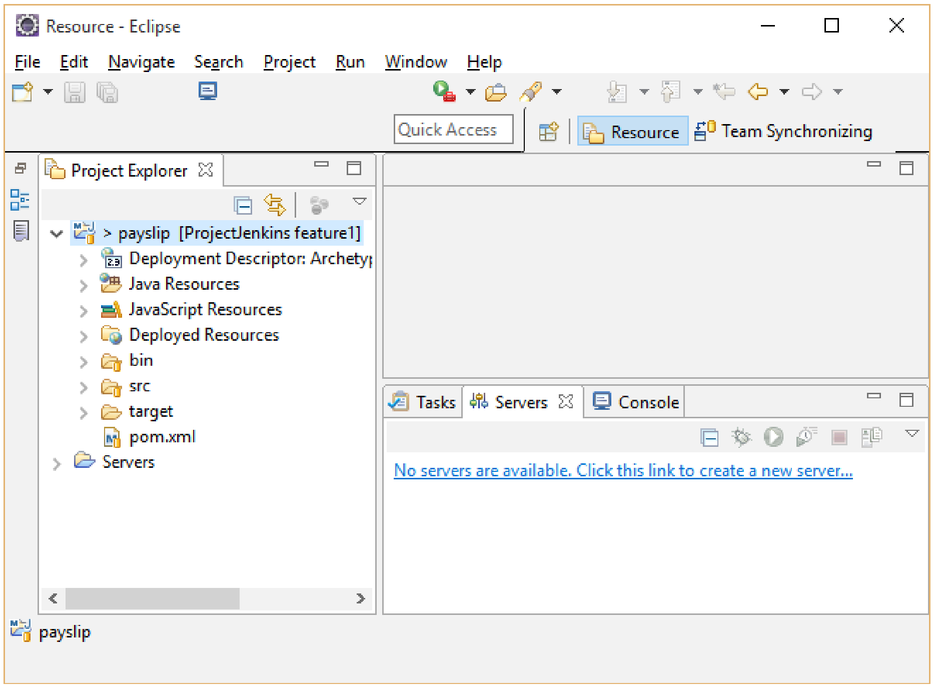
1. To do so, right-click on the project `payslip` and click on **Properties**.
2. Inside the **Properties** window, click on **Targeted Runtimes**. If the Apache Tomcat server is already installed, then it will appear in the **Targeted Runtimes** list on the right-hand side of the window, as shown in the next screenshot.

3. Select the Apache Tomcat server instance, **Apache Tomcat v8.0** in our case.



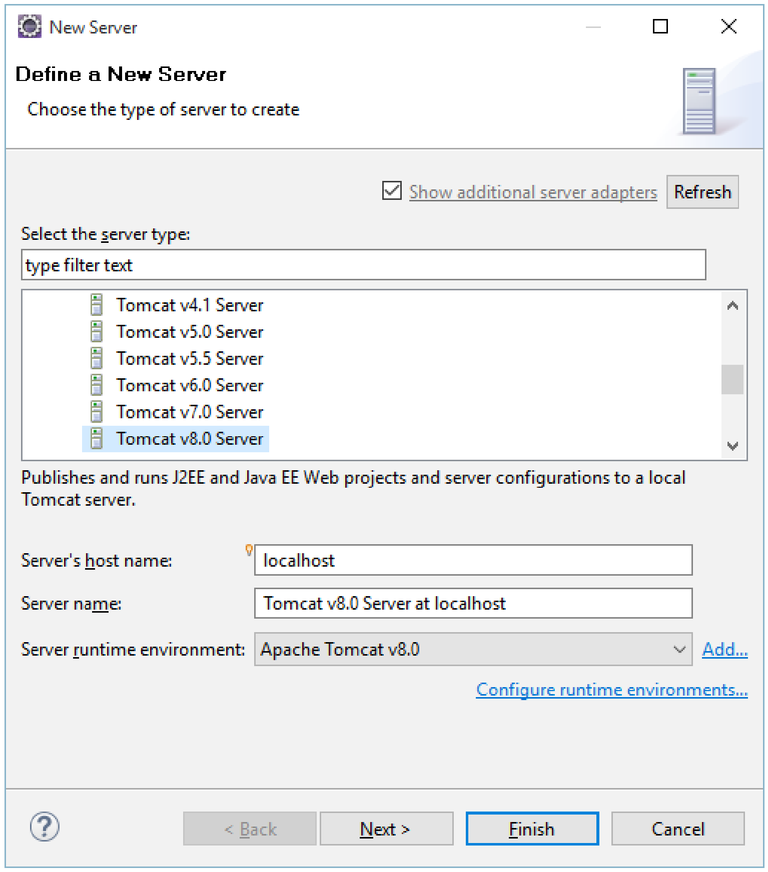
4. Click on the **OK** button to save.

5. Click on the link **No servers are available. Click this link to create a new server...**, as shown in the following screenshot:

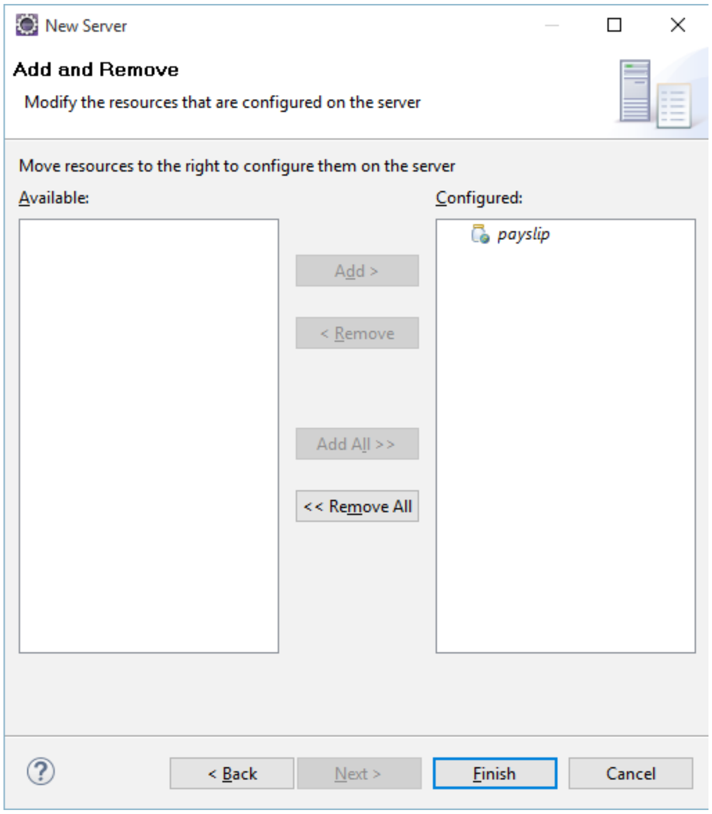


6. Select **Tomcat v8.0 Server** as the server type. Leave rest of the fields at their default values.

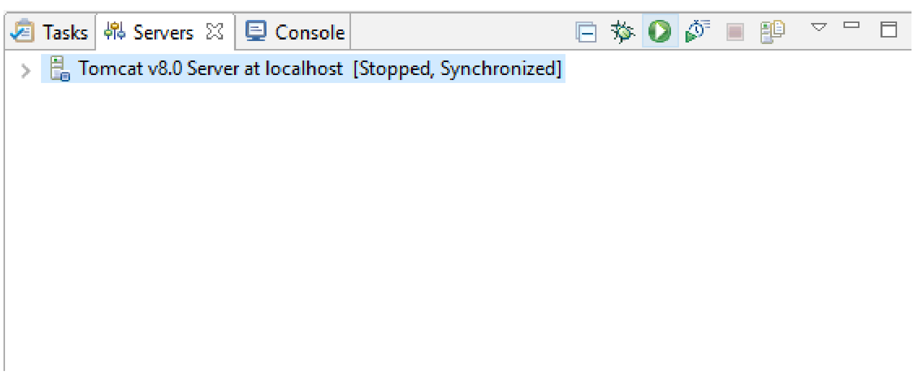
7. Click on the **Next** button.



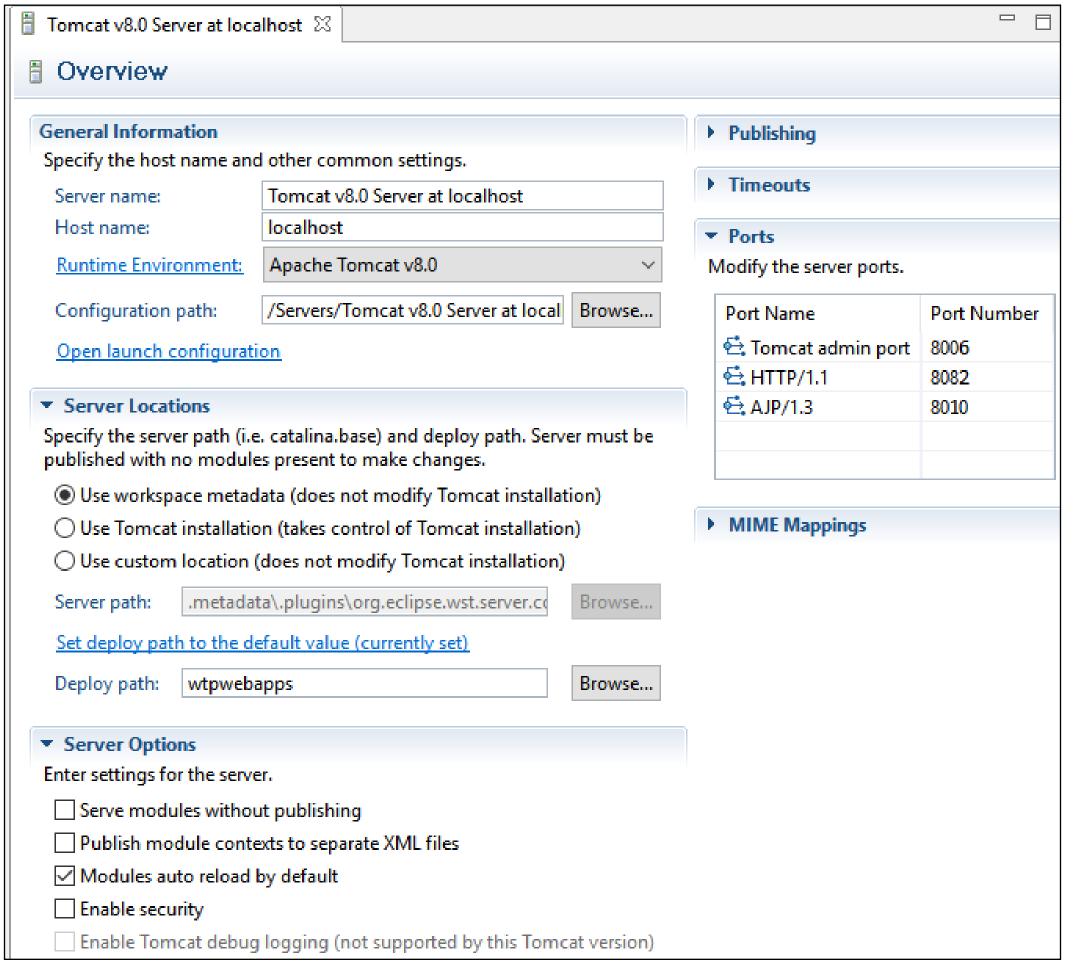
8. You will see `payslip` under the list of **Available** resources. Move it from **Available** to **Configured** by clicking the **Add** button, as shown in the following screenshot:



9. Double-click on the available servers under the **Servers** tab, as shown in the following screenshot:

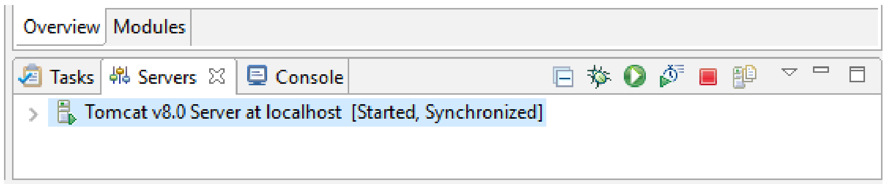


10. A long configurations page appears. Change the following values of **Ports**:
    - **Tomcat admin port = 8006**
    - **HTTP/1.1 = 8082**
    - **AJP/1.3 = 8010**

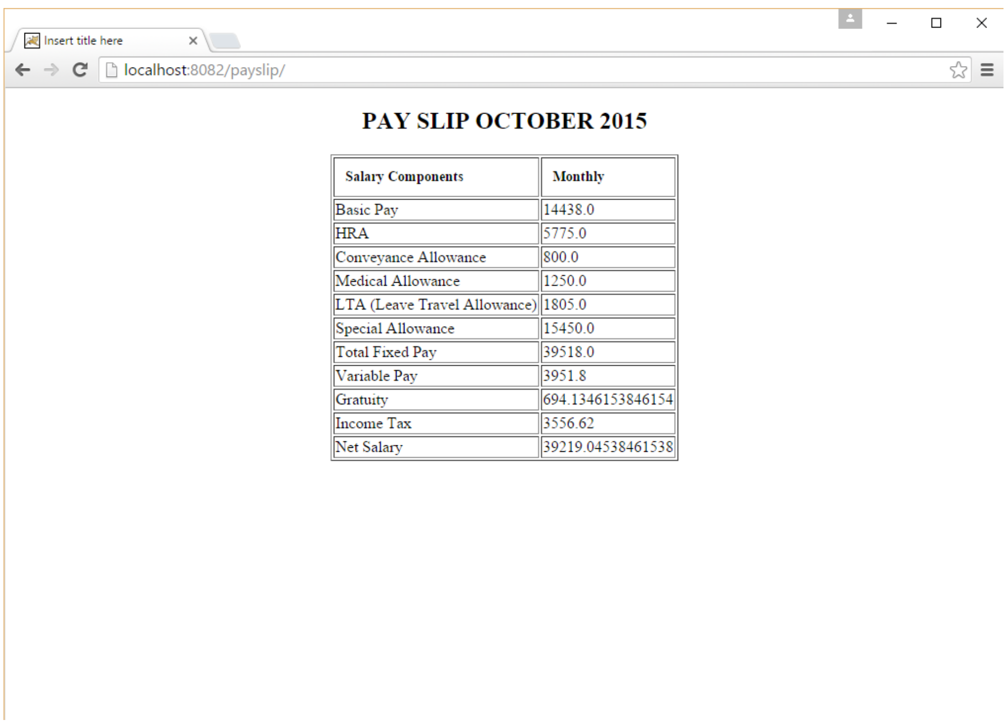


> Change the ports only if there is more than one Apache Tomcat server installed on your machine.

11. Start the server by clicking on the green play button. That's it. Your development workspace is ready. The outcome of your code changes can be quickly seen on the web browser.

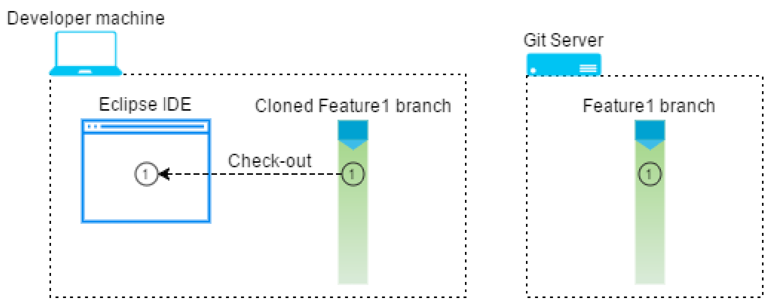


12. Here's what the `payslip` page looks like. The following preview is from the application servlet configured on the developer's machine. It is a monthly payslip describing most of the salary components.



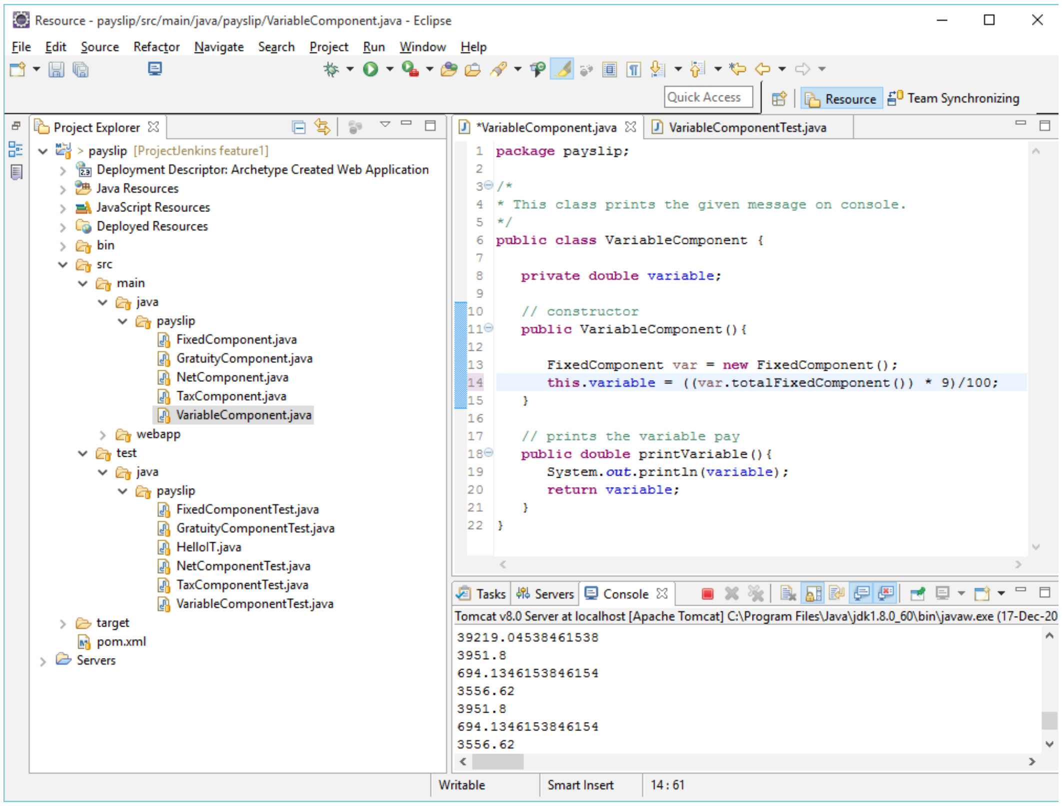
Making changes to the Feature1 branch

The following figure will help you understand the overall task of the current section:



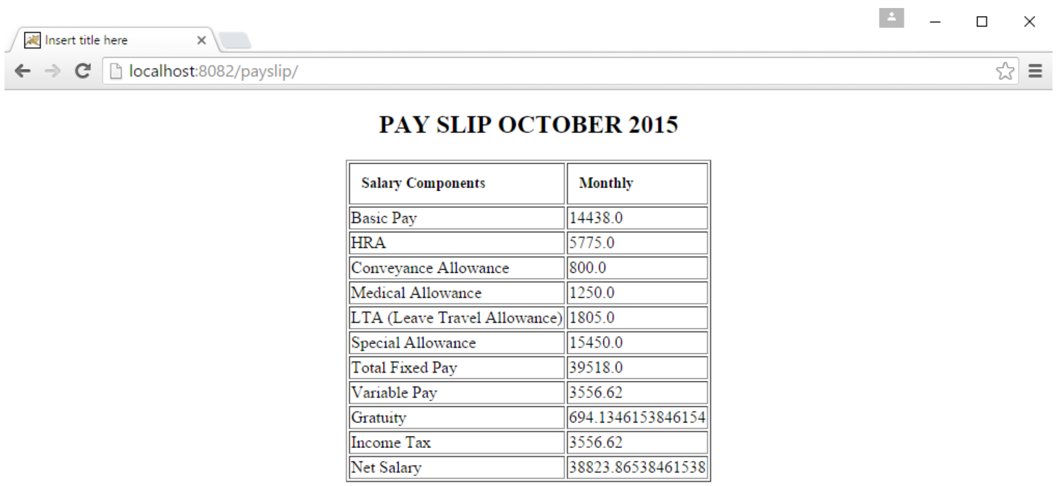
Let's test the CI pipeline by making some changes:

1. Open Eclipse and expand the `payslip` project. Go to `src` | `main` | `java` | `payslip`. You will see the Java files that compute the various components of the `payslip`.
2. Let's make some modifications. Open the `VariableComponent.java` file by double-clicking on it.
3. Go to line number 14 and change the percentage value from `10` to `9`, as shown in the following screenshot:



4. Your changes directly reflect on the webpage, as we have tightly integrated Eclipse with the Apache Tomcat server.

5. You can see the **Variable Pay** value changing from **3951.8** to **3556.62**. That's a 1 % decrease.



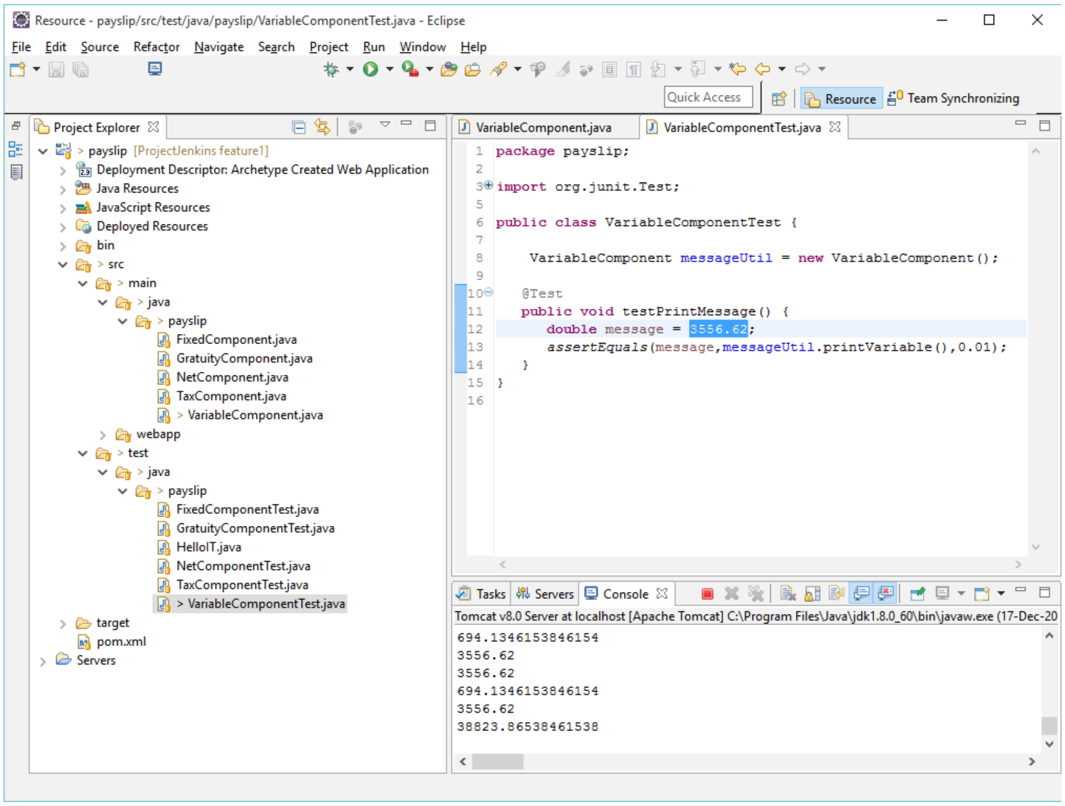
PAY SLIP OCTOBER 2015

Salary Components	Monthly
Basic Pay	14438.0
HRA	5775.0
Conveyance Allowance	800.0
Medical Allowance	1250.0
LTA (Leave Travel Allowance)	1805.0
Special Allowance	15450.0
Total Fixed Pay	39518.0
Variable Pay	3556.62
Gratuity	694.1346153846154
Income Tax	3556.62
Net Salary	38823.86538461538

Similarly, we also have unit test cases written for each Java file. Let's make some changes to the unit test case file; otherwise, the unit test will fail. The steps are as follows:

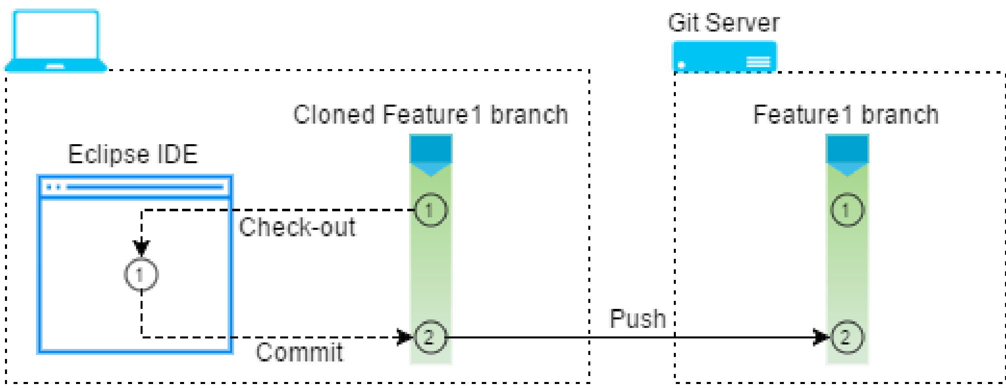
1. Go to `src` | `test` | `java` | `payslip`. You can see a number of unit test cases files.
2. Double-click on the `VariableComponetTest.java` file to open it.

3. Go to line number 12 and change the value from `3951.8` to `3556.62`.



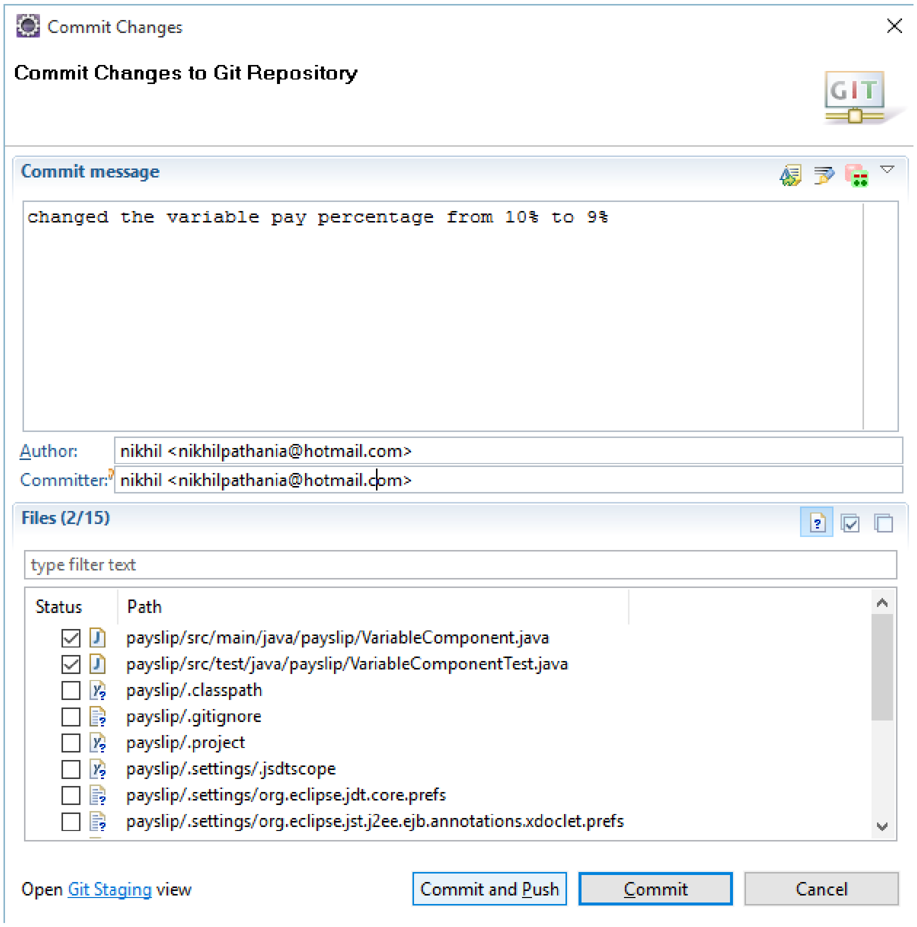
Committing and pushing changes to the Feature1 branch

The following figure will help us understand the overall task that we will be performing in this this section:



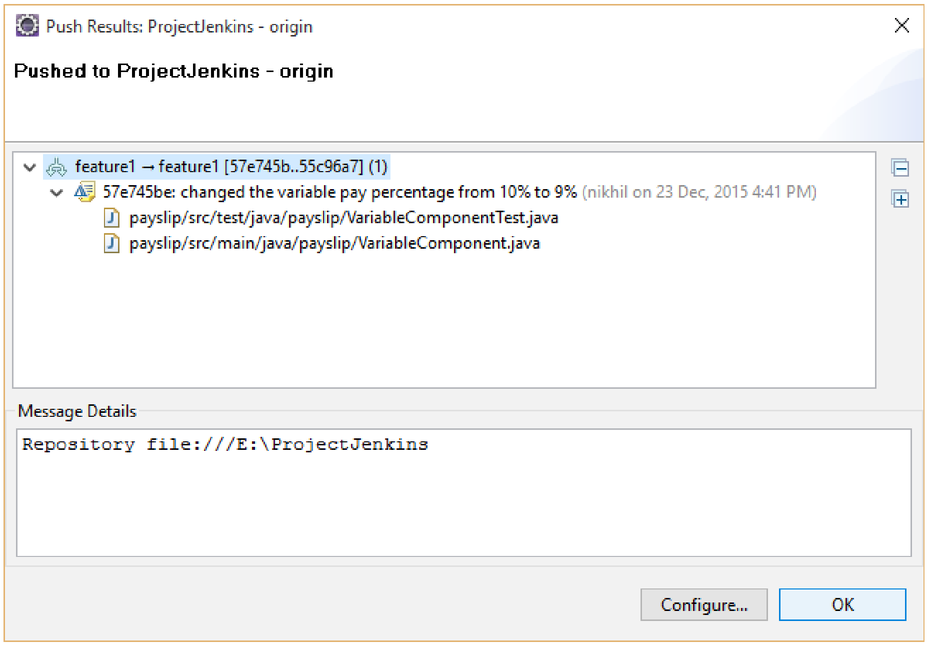
Perform the following steps to commit and push the changes made in previous section:

1. Right-click on the project `payslip` and go to **Team** | **Commit...**.
2. In the window that opens, add some comments as shown in the next screenshot, and check the modified code files.

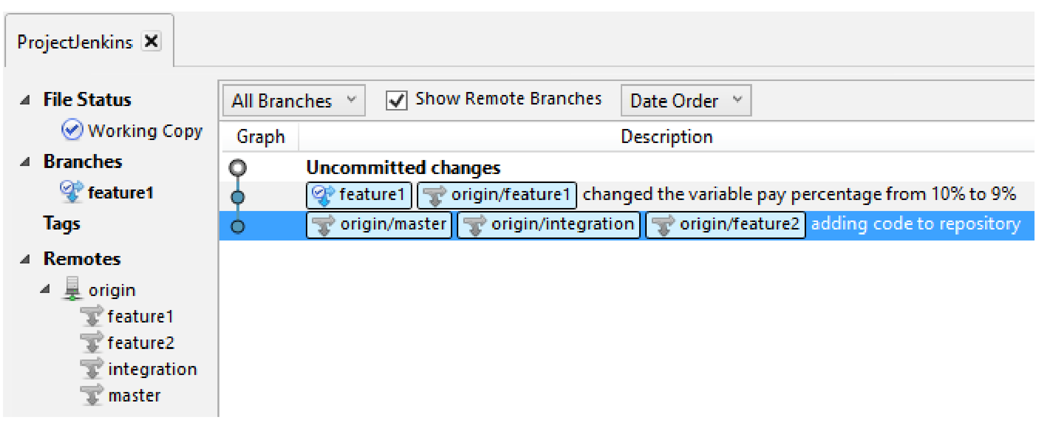


3. Click on the **Commit and Push** button.
4. You can see the code is committed on the cloned `feature1` branch and we are pushed to the remote `feature1` branch.

5. Click on the **OK** button to proceed.



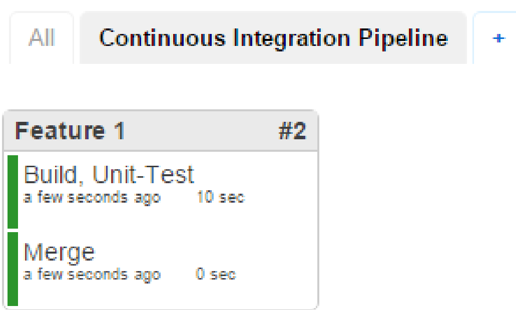
6. Similarly, if you open source tree client for Git, you can see that local `feature1` (cloned) and the original `feature1` are at the same level after the push operation.



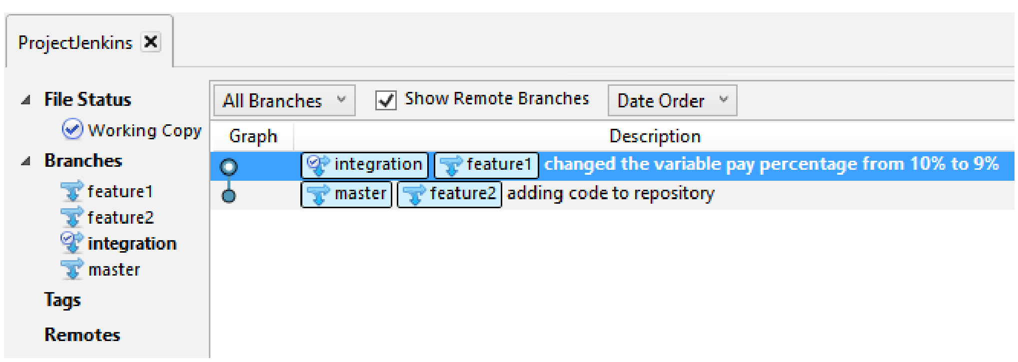
Real-time Jenkins pipeline to poll the Feature1 branch

Some changes were made on the `Feature1` branch. Let's see if Jenkins has detected it:

1. Go to the Jenkins Dashboard and click on the **Continuous Integration Pipeline** view.
2. If you are fast enough, you can see it in action. If you are late, here's what you will see. The Jenkins job to build and unit test code on the feature branch is successful, and the Jenkins job to merge the committed code to the integration branch is also successful.

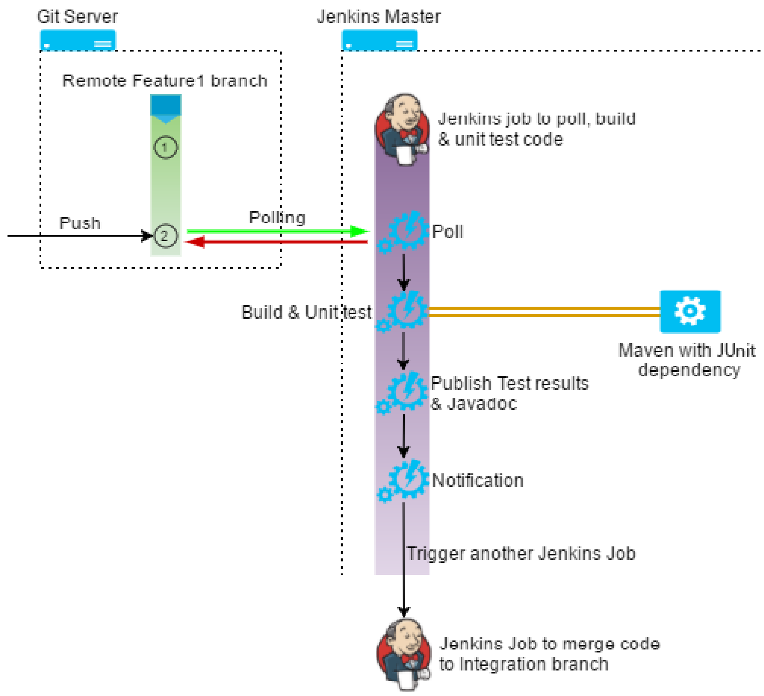


3. Come back to the source tree client for Git. You can see that the `feature1` branch and the `integration` branch are at the same level now after the merge.



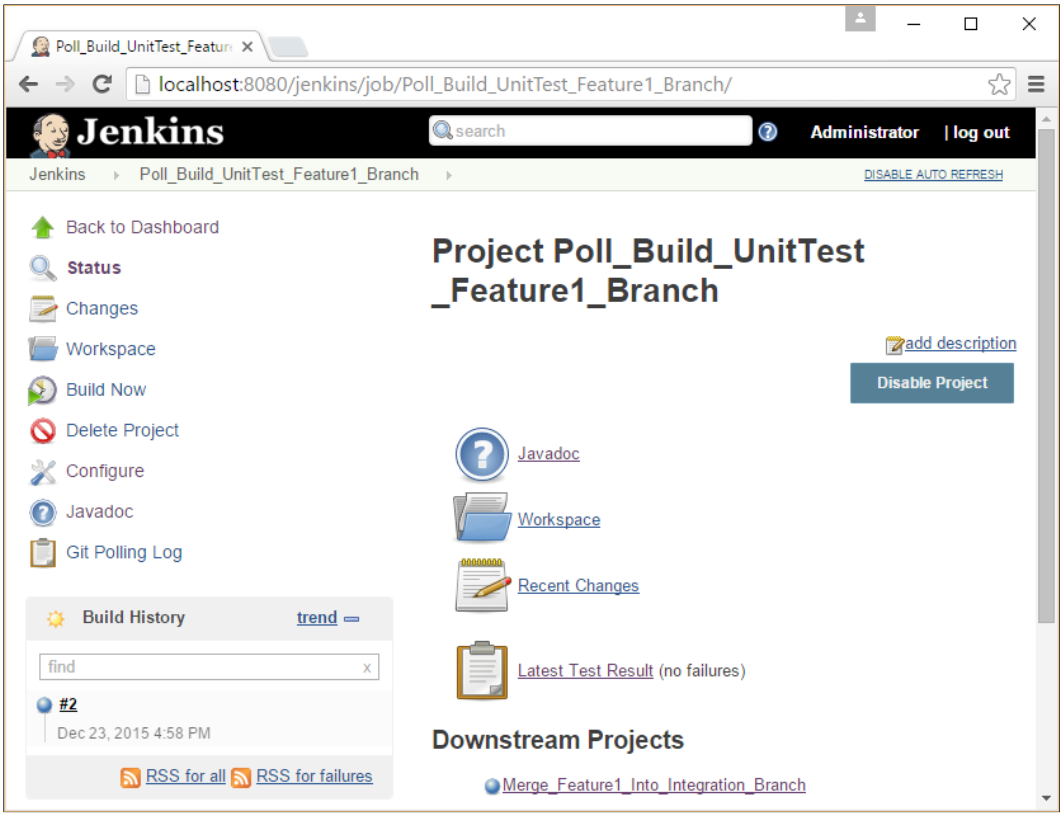
The Jenkins job to poll, build, and unit test code on the Feature1 branch

The following figure will help us understand the overall task performed by the current Jenkins job:

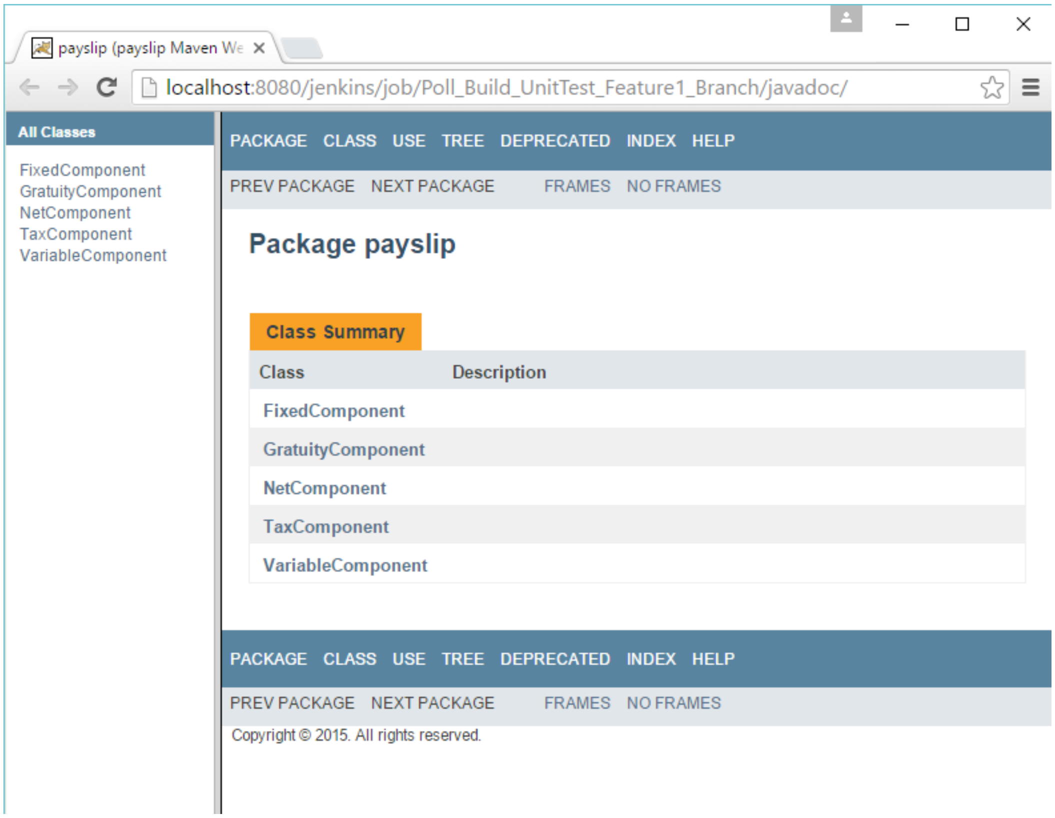


Now, let's see the Jenkins job to build and unit test code on the `Feature1` branch in detail:

1. On the Jenkins Dashboard, click on the **Poll_Build_UnitTest_Feature1_Branch** Jenkins job.
2. You can see a link to access **Javadoc** and the test results, as shown in the following screenshot:



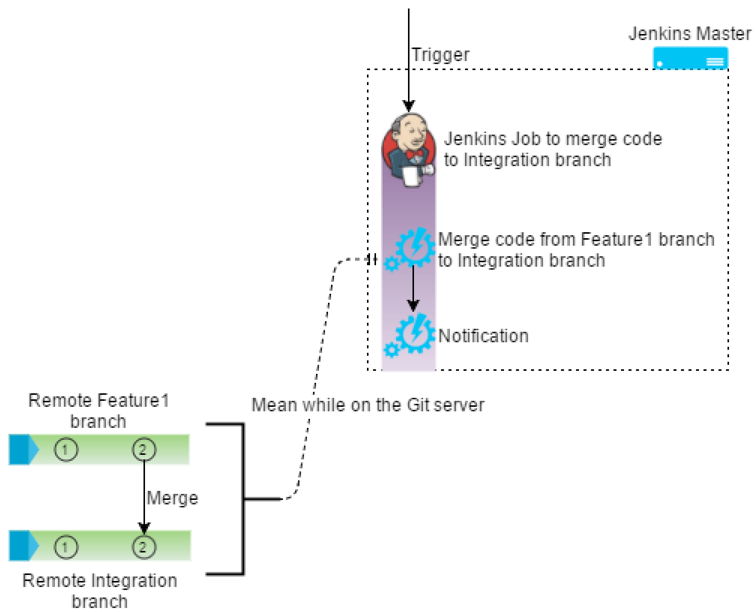
3. This is what the Javadoc looks like:



4. Here are the test results:

The Jenkins job to merge code to integration branch

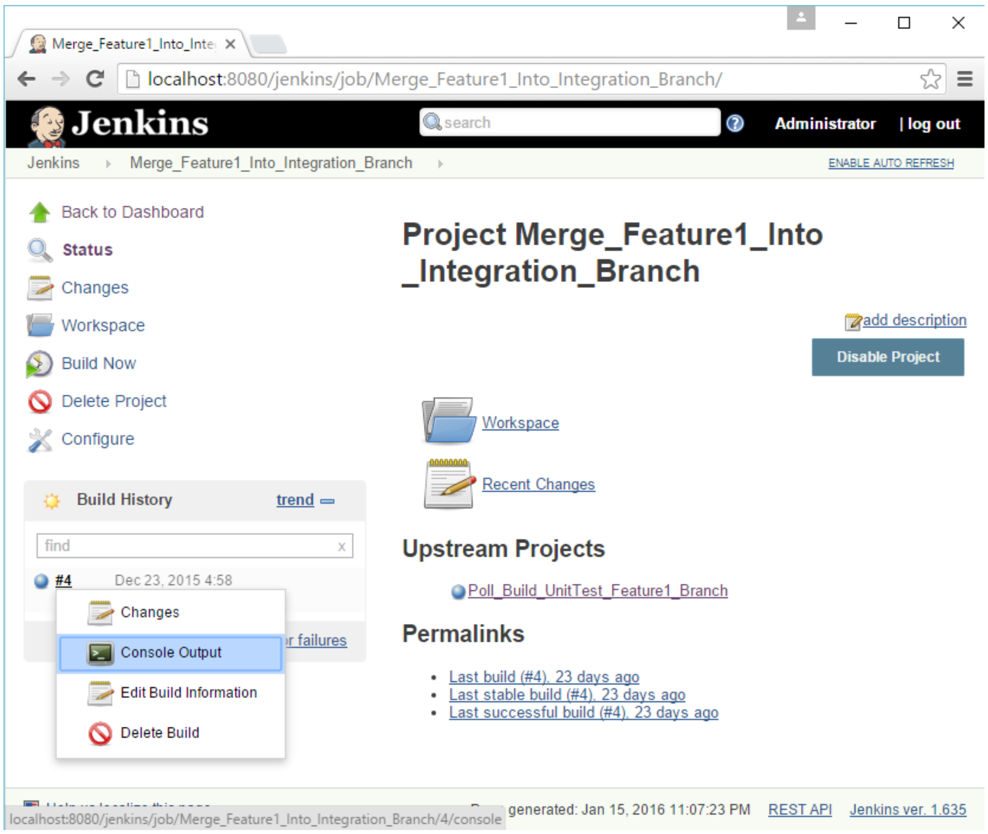
The following figure will help us understand the overall task performed by the current Jenkins job:



Let's take a look at the Jenkins job to merge code on the integration branch:

1. On the Jenkins Dashboard, click on the **Merge_Feature1_Into_Integration_Branch** Jenkins job.
2. Under **Build History**, you will see some builds.

3. Right-click on any one of them and select **Console Output** to see the actual logs.



4. From the logs, you can see how code was merged from the `Feature1` branch to the integration branch.

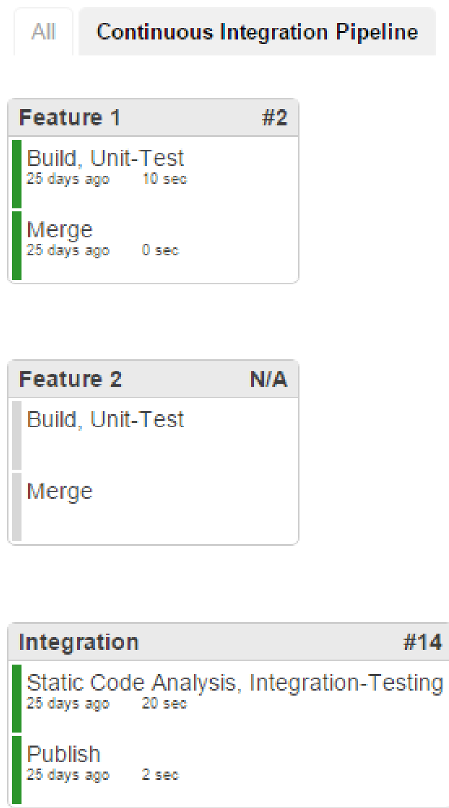
```
Console Output

Started by upstream project "Poll_Build_UnitTest_Feature1_Branch" build number 2
originally caused by:
 Started by an SCM change
Building in workspace C:\Jenkins\jobs\Merge_Feature1_Into_Integration_Branch\workspace
[workspace] $ cmd /c call "C:\Program Files\Apache Software Foundation\Tomcat 8.0\temp\hudson3974299262958976381.bat"

C:\Jenkins\jobs\Merge_Feature1_Into_Integration_Branch\workspace>E:

E:\>cd ProjectJenkins

E:\ProjectJenkins>git checkout integration
Switched to branch 'integration'

E:\ProjectJenkins>git merge feature1 --stat
Updating 55c96a7..57e745b
Fast-forward
 .../src/main/java/payslip/VariableComponent.java    | 42 +++++++++++-----------
 .../test/java/payslip/VariableComponentTest.java    | 30 ++++++++--------
 2 files changed, 36 insertions(+), 36 deletions(-)

E:\ProjectJenkins>exit 0
Finished: SUCCESS
```

Real-time Jenkins pipeline to poll the integration branch

By this time, the rest of the Jenkins job in the CI pipeline should be complete.

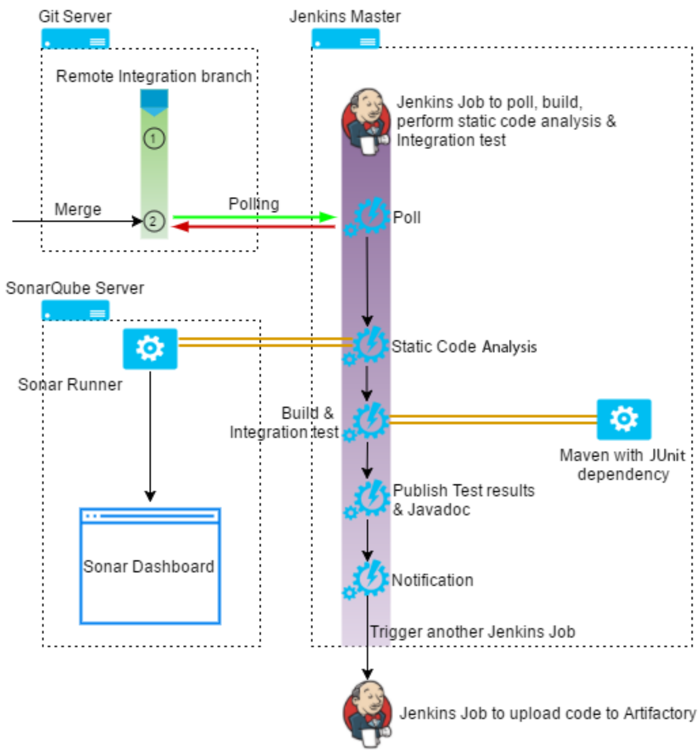
1. Go to the Jenkins Dashboard and click on the **Continuous Integration Pipeline** view.

2. We can see the Jenkins job to poll the integration branch for changes is successfully completed, and the Jenkins job to publish the changes to Artifactory is also completed too.



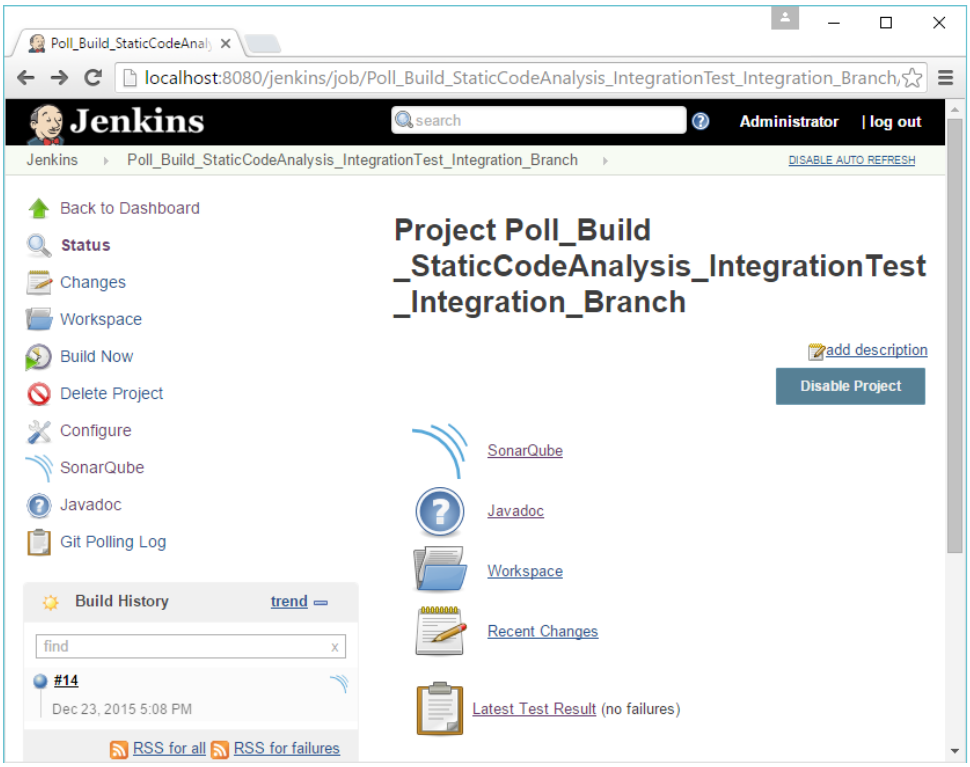
The Jenkins job to poll, build, perform static code analysis, and perform integration tests

The following figure will help us understand the overall task performed by the current Jenkins job:

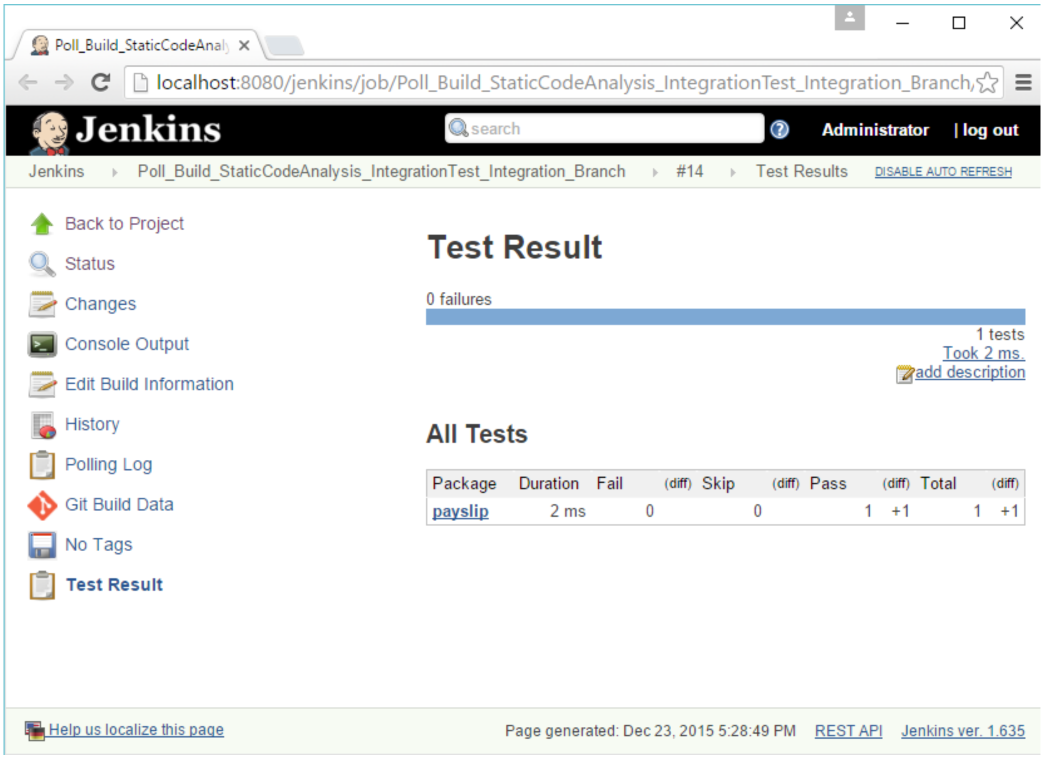


Let's take a look at the Jenkins job to poll, build, and perform static code analysis and Integration test:

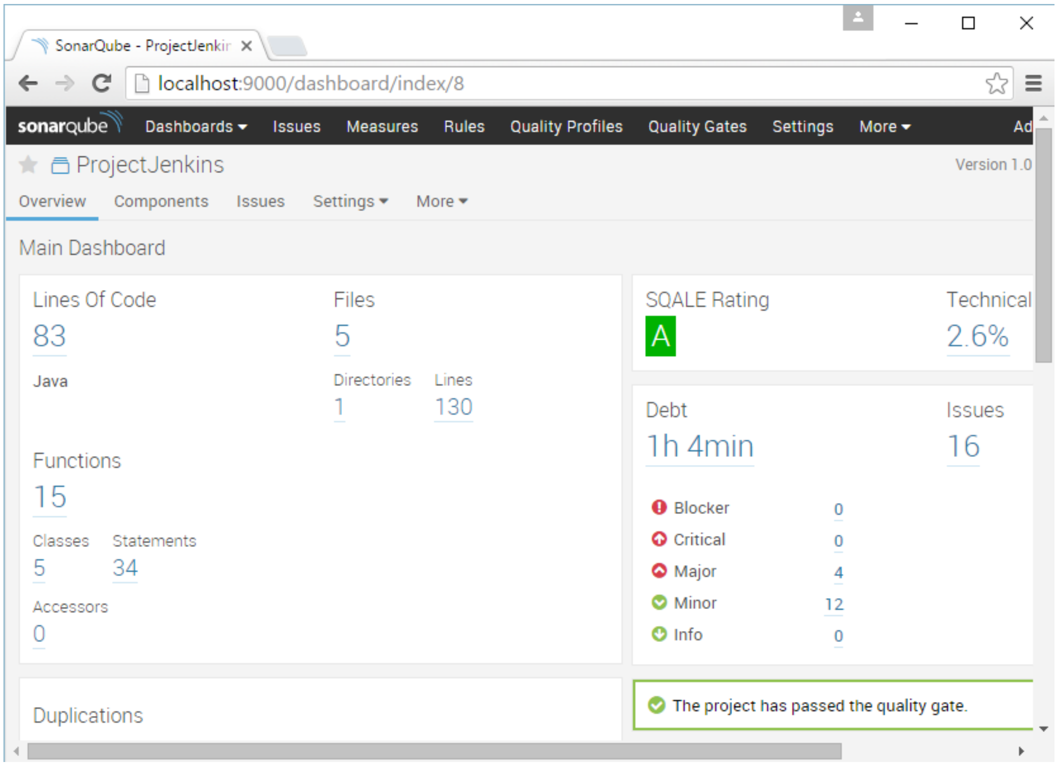
1. From the Jenkins Dashboard, click on the **Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch** Jenkins job.
2. You can see a link to access **Javadoc** and the test results in the following screenshot:



3. Here are the test results:

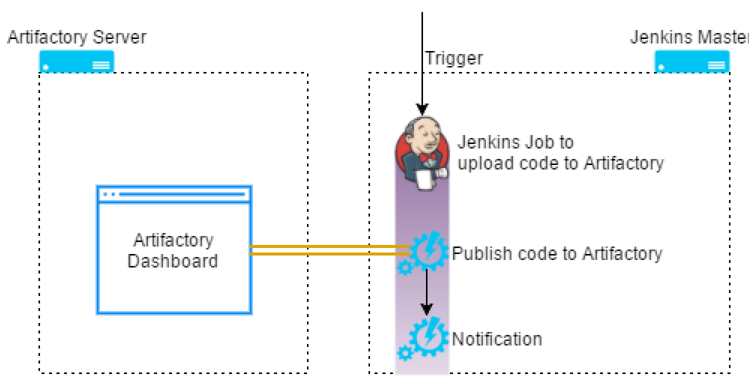


4. To see the Sonar analysis, go to the SonarQube dashboard or click on the Sonar analysis link inside the console output logs: `http://localhost:9000/dashboard/index/my:projectjenkins`.



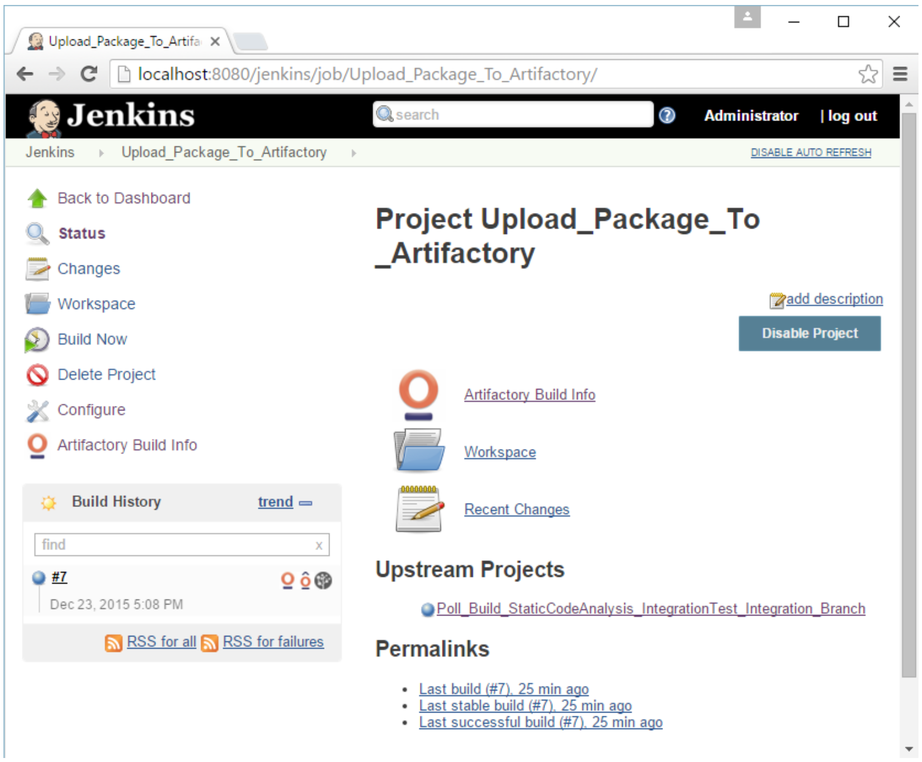
The Jenkins job to upload code to Artifactory

The following figure will help us understand the overall task performed by the current Jenkins job:

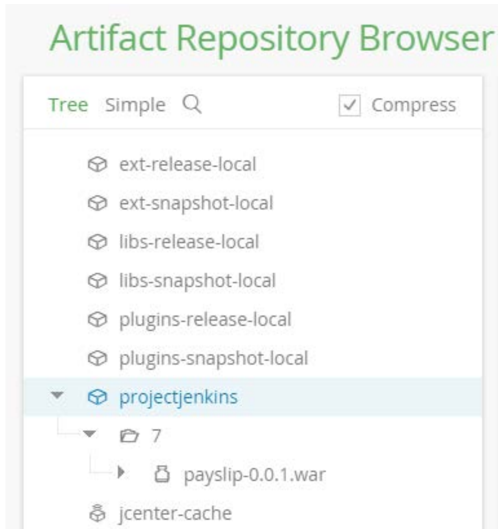


Let's take a look at the Jenkins job to upload code to Artifactory:

1. From the Jenkins Dashboard, click on the **Upload_Package_To_Artifactory** Jenkins job.



2. Click on the **Artifactory build info** link and it will take you to the Artifactory Dashboard, where you can see the `payslip-0.0.1.war` file under the `prjectjenkins` project.



Summary

In this chapter, we first saw how to install and configure SonarQube. We saw how to create a project inside SonarQube and how to integrate it with Jenkins using plugins. We discussed how to install and configure Artifactory.

We then created the remaining jobs in the Continuous Integration pipeline that poll the integration branch for changes, perform static code analysis, perform integration testing, and upload the successfully tested code to Artifactory.

We also saw how to install and configure the delivery pipeline plugin. Although not necessary, but it gave a good look to our Continuous Integration pipeline.

We saw how to configure the Eclipse tool with Git. In this way, a developer can seamlessly work on Eclipse and perform all the code check in, check out, and push operations from the Eclipse IDE alone.

Lastly, using an example, we saw the whole Continuous Integration pipeline in action from the perspective of a developer working on a feature branch.

The Continuous Integration Design discussed in the book can be modified to suit the needs of any type of project. The users just need to identify the right tools and configurations that can be used with Jenkins.

6
Continuous Delivery Using Jenkins

This chapter begins with the definition of **Continuous Delivery** (**CD**) and its relation to Continuous Integration, followed by a Continuous Delivery Design. While working on the Continuous Delivery Design, we will create various new Jenkins jobs, but in a slightly different manner. For the very first time in this book, we will use the parameterized triggers in Jenkins.

These parameterized triggers have proved to be the most useful and versatile features in Jenkins. Using such triggers, we can pass the parameters among connected Jenkins jobs, which makes communication among Jenkins jobs more powerful.

We will also see how to configure slaves in Jenkins. The Jenkins master-slave configuration can be used in various scenarios. However, in this chapter, we will use it to let a Jenkins master perform various tests on a Jenkins slave agent (testing server).

In the process, we will see how Jenkins can be configured with various test automation tools, such as Selenium, JMeter, and TestNG, to achieve **Continuous Testing**. Automated testing in a continuous manner is an integral part of Continuous Delivery. While implementing Continuous Delivery, we will modify some of the Jenkins jobs that were part of Continuous Integration Design.

We will also create a nice visual flow for the Continuous Delivery pipeline using the Jenkins delivery pipeline plugin, similar to the one we saw in the previous chapter. Lastly, we will see our Continuous Delivery pipeline in action using a simple example.

These are the important topics that we will cover in this chapter:

- Jenkins master-slave architecture
- Passing parameters across Jenkins jobs

- User acceptance testing using Selenium and TestNG
- Performance testing using JMeter
- Configuring applications such as Maven, JDK, and Git on a Jenkins master that can be used across all Jenkins slave machines

What is Continuous Delivery?

Continuous Delivery is the software engineering practice wherein production-ready features are produced in a continuous manner.

When we say production-ready features, we mean only those features that have passed the following check points:

- Unit testing
- Integration
- Static code analysis (code quality)
- Integration testing
- System integration testing
- User acceptance testing
- Performance testing
- End-to-end testing

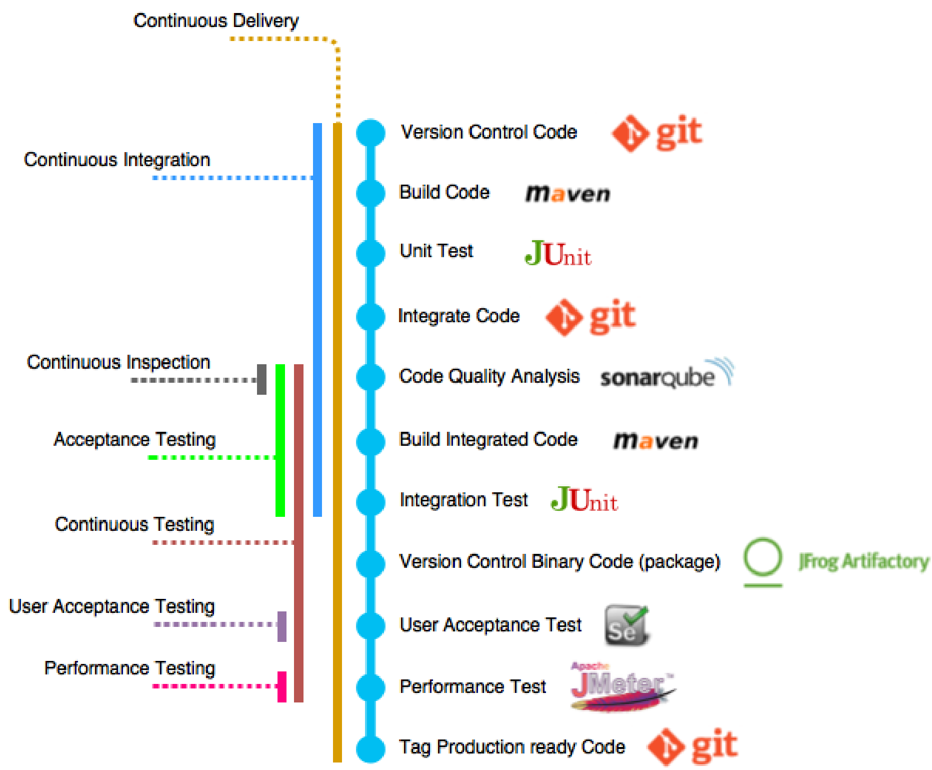
However, the list is not complete. You can incorporate as many types of testing as you want to certify that the code is production ready.

From the preceding list, the first four check points are covered as part of the Continuous Integration Design discussed in the previous chapter. This Continuous Integration Design, when combined with deployments (not listed here) and with all sorts of automated testing can be safely called Continuous Delivery.

In other words, Continuous Delivery is an extension of the Continuous Integration methodology to the deployment and testing phases of a **Software Development Life Cycle** (**SDLC**). Testing in itself is a vast area.

In any organization, big or small, the previously mentioned testing is either performed on a single environment or on multiple environments. If there are multiple testing environments, then there is a need to deploy the package in all those testing environments. Therefore, deployment activities are also part of Continuous Delivery.

The next figure will help us understand the various terminologies that were discussed just now. The various steps a software code goes through, from its inception to its utilization (development to production) are listed in the following figure. Each step has a tool associated with it, and each one is part of a methodology:



Continuous Delivery Design

The Continuous Delivery Design that we are going to discuss now is a simple extension of the Continuous Integration Design that we discussed in *Chapter 4, Continuous Integration Using Jenkins – Part I*. This includes creating new Jenkins jobs as well as modifying the already existing Jenkins jobs that are part of the Continuous Integration Design.

Continuous Delivery pipeline

Continuous Integration is an integral part of Continuous Delivery. Hence, all Jenkins jobs that were created as part of the Continuous Integration Design will fall into the Continuous Delivery Design by default. From the previous chapters, we are familiar with the following Continuous Integration pipelines:

- The pipeline to poll the feature branch
- The pipeline to poll the integration branch

However, as part of our CD Design, the pipeline to poll the integration branch will be modified by reconfiguring the existing Jenkins jobs and adding new Jenkins jobs. Together, these new Jenkins pipelines will form our Continuous Delivery pipeline.

Pipeline to poll the feature branch

The pipeline to poll the feature branch will be kept as it is, and there will be no modifications to it. This particular Jenkins pipeline is coupled with the feature branch. Whenever a developer commits something on the feature branch, the pipeline is activated. It contains two Jenkins jobs that are as follows.

Jenkins job 1

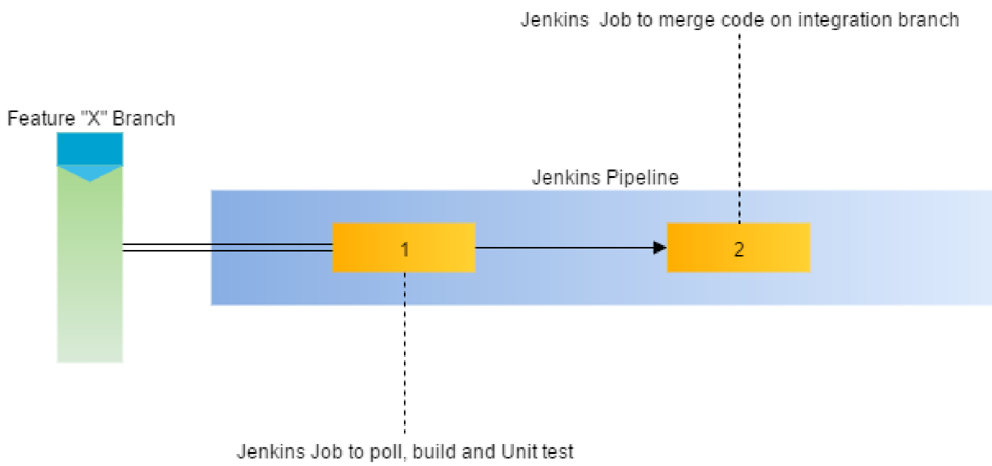
The first Jenkins job in the pipeline performs the following tasks:

- It polls the feature branch for changes at regular intervals
- It performs a build on the modified code
- It executes the unit tests

Jenkins job 2

The second Jenkins job in the pipeline performs the following task:

- It merges the successfully built and tested code into the integration branch



Pipeline to poll the integration branch

This Jenkins pipeline is coupled with the integration branch. Whenever there is a new commit on the integration branch, the pipeline gets activated. However, it will now contain five Jenkins jobs (two older and three new ones) that perform the following tasks:

Jenkins job 1

The first Jenkins job in the pipeline will now perform the following tasks:

- It polls the integration branch for changes at regular intervals
- It performs a static code analysis of the downloaded code
- It executes the integration tests
- It passes the `GIT_COMMIT` variable to the Jenkins job that uploads the package to Artifactory (new functionality)

> The `GIT_COMMIT` variable is a Jenkins system variable that contains the SHA-1 checksum value. Each Git commit has a unique SHA-1 checksum. In this way, we can track which code to build.

Jenkins job 2

The second Jenkins job in the pipeline will now perform the following tasks:

- It uploads the built package to the binary repository
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variable to the Jenkins job that deploys the package to the testing server (new functionality)

> The variable `BUILD_NUMBER` is a Jenkins system variable that contains the build number. Each Jenkins job has a build number for every run.
>
> We are particularly interested in the build number corresponding to Jenkins job 2. This is because this job uploads the built package to Artifactory. We might need this successfully uploaded artifact later during Jenkins job 3 to deploy the package to testing server.

We will create three new Jenkins jobs 3, 4, and 5 with the following functionalities.

Jenkins job 3

The third Jenkins job in the pipeline performs the following tasks:

- It deploys a package to the testing server using the `BUILD_NUMBER` variable
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variable to the Jenkins job that performs the user acceptance test

Jenkins job 4

The fourth Jenkins job in the pipeline performs the following tasks:

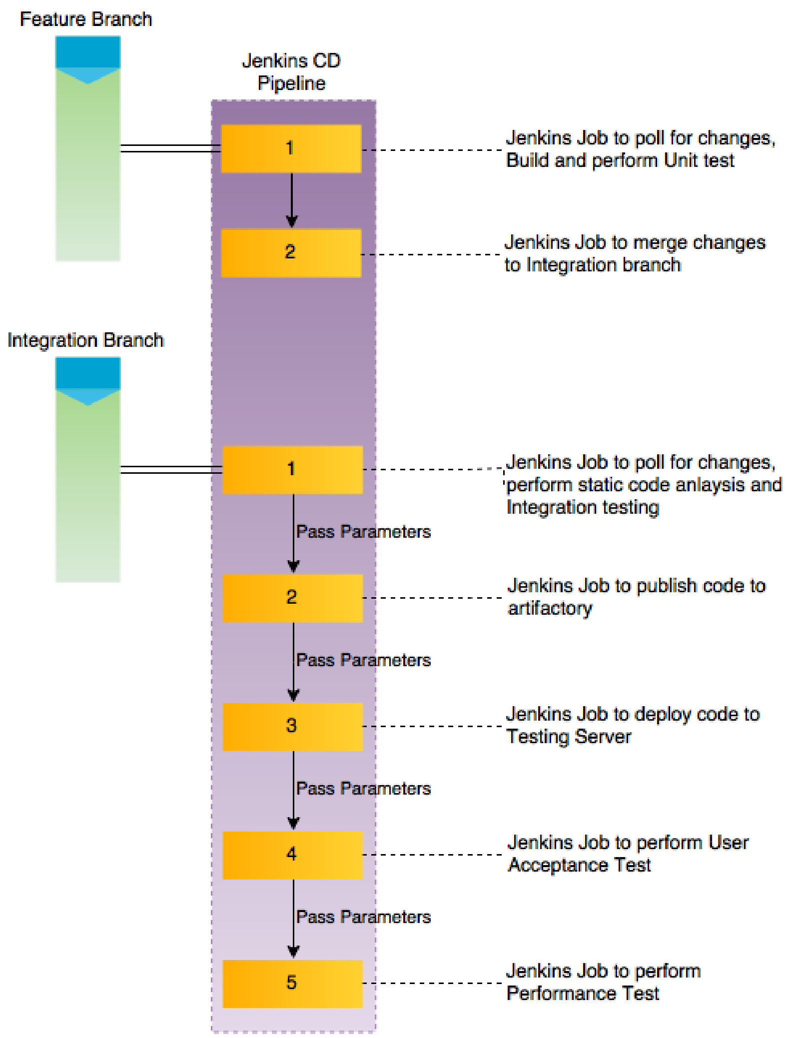
- It downloads the code from Git using the `GIT_COMMIT` variable
- It performs the user acceptance test
- It generates the test results report
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variable to the Jenkins job that performs the performance test

Jenkins job 5

The last Jenkins job in the pipeline performs the following tasks:

- It performs the performance test

- It generates the test results report



All the Jenkins jobs should have a notification step that can be configured using advanced e-mail notifications.

Toolset for Continuous Delivery

The example project for which we are implementing Continuous Delivery is a Java-based web application. It's the same example project that was used in *Chapter 4*, *Continuous Integration Using Jenkins – Part I*, and *Chapter 5*, *Continuous Integration Using Jenkins – Part II*

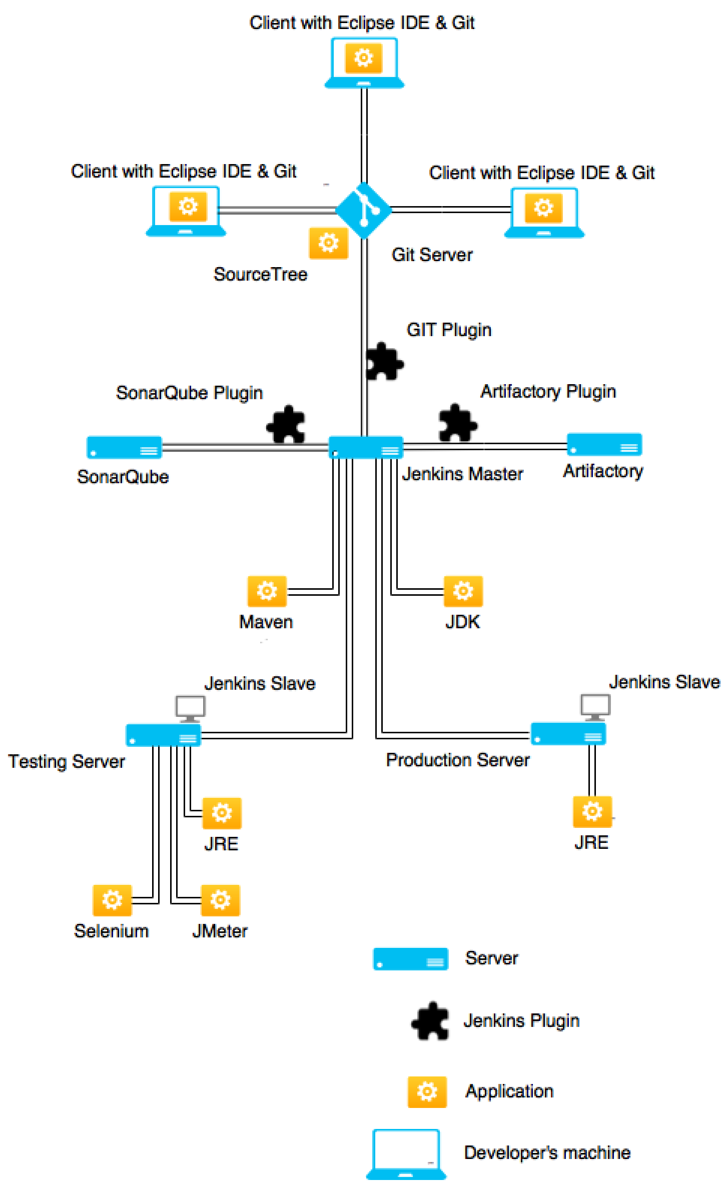
The following table contains the list of tools and technologies involved in everything that we will see in this chapter:

Tools and technologies	**Description**
Java	The primary programming language used for coding
Maven	Build tool
JUnit	Unit test and Integration test tools
Apache Tomcat server	Servlet to host the end product
Eclipse	IDE for Java development
Jenkins	Continuous Integration tool
Git	Version control system
Artifactory	Binary repository
SourceTree	Git client
SonarQube	Static code analysis tool
JMeter	Performance testing tool
TestNG	Unit test and integration test tool
Selenium	User acceptance testing tool

The next figure demonstrates how Jenkins fits in as a CD server in our Continuous Delivery Design, along with the other DevOps tools:

- The developers have the Eclipse IDE and Git installed on their machines. This Eclipse IDE is internally configured with the Git server. This enables the developers to clone the feature branch from the Git server on their machines.
- The Git server is connected to the Jenkins master server using the Git plugin. This enables Jenkins to poll the Git server for changes.
- The Apache Tomcat server, which hosts the Jenkins master, also has Maven and JDK installed on it. This enables Jenkins to build the code that has been checked-in on the Git Server.
- Jenkins is also connected to the SonarQube server and the Artifactory server using the SonarQube plugin and the Artifactory plugin, respectively.
- This enables Jenkins to perform a static code analysis of the modified code. Once all the build, quality analysis, and integration testing is successful, the resultant package is uploaded to the Artifactory for further use.

- The package also gets deployed on a testing server that contains testing tools such as JMeter, TestNG, and Selenium. Jenkins, in collaboration with the testing tools, will perform user acceptance tests and performance tests on the code.



Configuring our testing server

There are many types of testing that are performed by organizations to ensure they deliver an operational code. However, in this book, we will see only user acceptance testing and performance testing. We will do all this on a single testing server.

I chose an Ubuntu machine as our testing server. We need to set up some software on our testing server that will assist us while we implement Continuous Delivery and Continuous Testing.

Installing Java on the testing server

The testing server will contain an Apache Tomcat server to host applications such as JMeter that run performance testing. Following are the steps to install JRE on the testing server.

1. To install JRE on the machine, open a terminal and use the following command. This will update all the current application installed on the testing server:

   ```
   sudo apt-get update
   ```

2. Generally, Linux ships with the Java package. Therefore, check if Java is already installed with the following command:

   ```
   java -version
   ```

3. If the preceding command returns a Java version, make a note of it. However, if you see **the program Java cannot be found in the following packages**, then Java hasn't been installed. Execute the following command to install it:

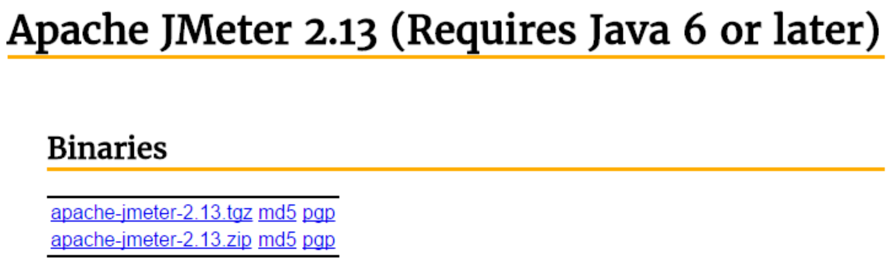
   ```
   sudo apt-get install default-jre
   ```

Installing Apache JMeter for performance testing

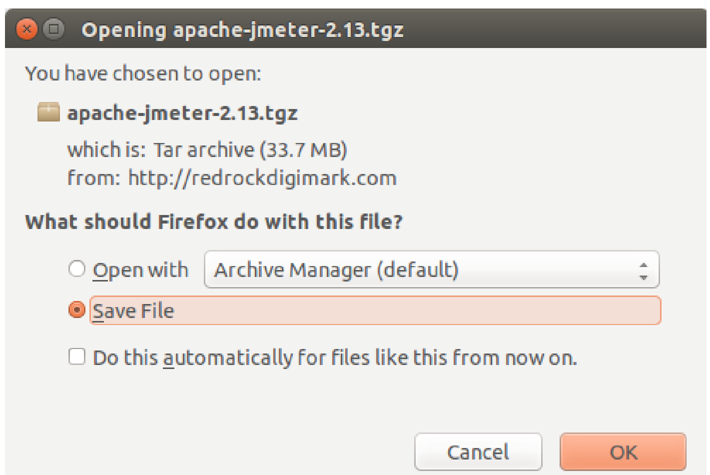
Apache JMeter is a good tool for performance testing. It's free and open source. It can run in both GUI and command line mode, which makes it a suitable candidate for automating performance testing.

Follow these steps to install Apache JMeter for performance testing:

1. Download `apache-jmeter-2.13.tgz` or whichever is the latest stable version from `http://jmeter.apache.org/download_jmeter.cgi`:

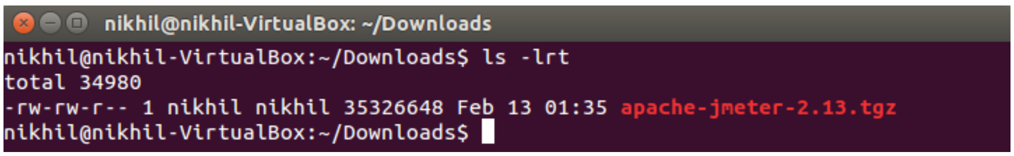


2. Download the respective archive package, as shown in the following screenshot:



3. The archive file gets downloaded to the directory /home/<user>/Downloads. To check, go to the download location and list the files by executing the following commands. Substitute <user> with the user account on your testing server machine by using the following command:

```
cd /home/<user>/Downloads
ls -lrt
```

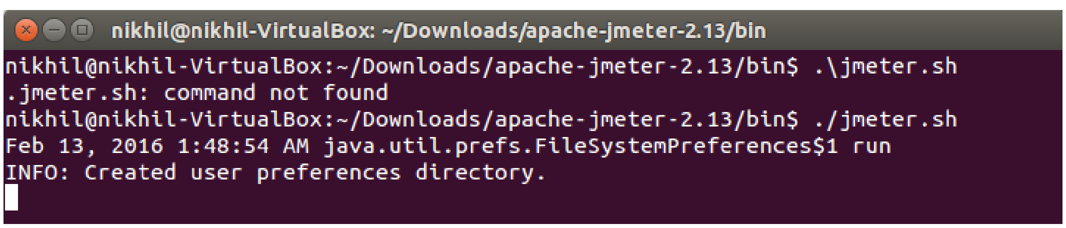


4. The installation is simple and only requires you to extract the archive file. To do so, use the following command. The archive will be extracted inside the same location:

```
tar zxvf apache-jmeter-2.13.tgz
```

5. To create a performance test case, we will use the GUI mode. To open the JMeter console, navigate to the location where the `jmeter.sh` file is present and run the `jmeter.sh` script as follows:

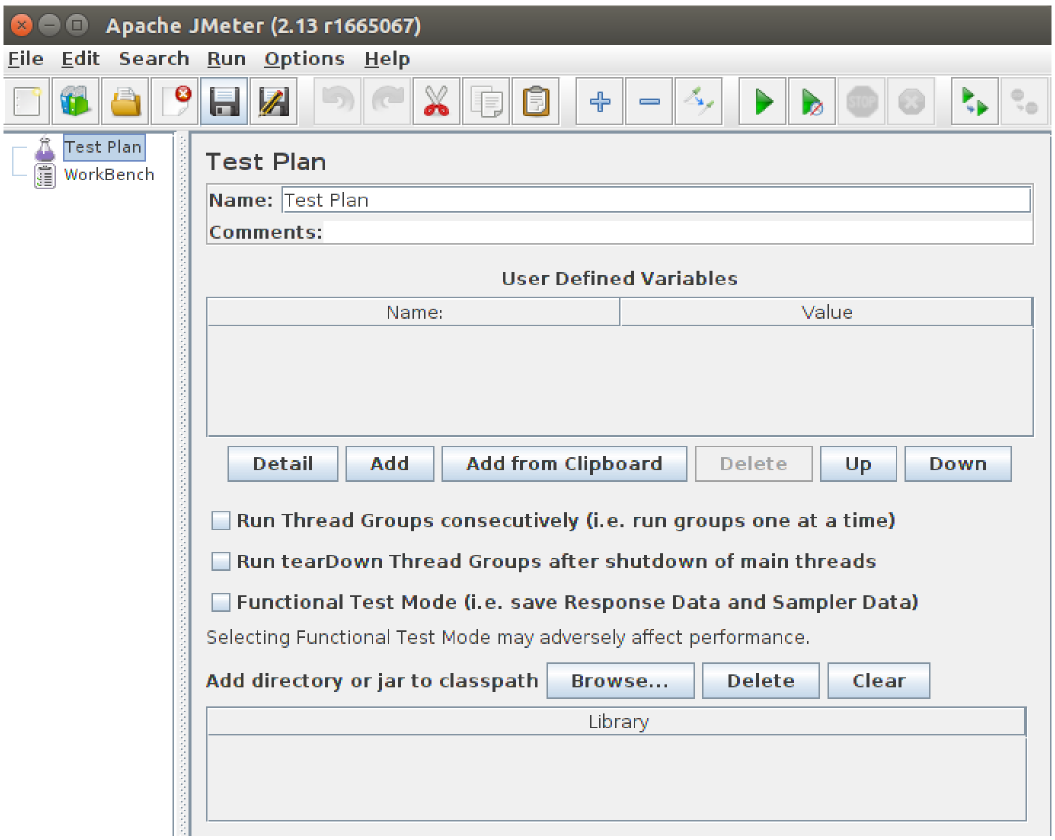
```
cd apache-jmeter-2.13/bin
/jmeter.sh
```



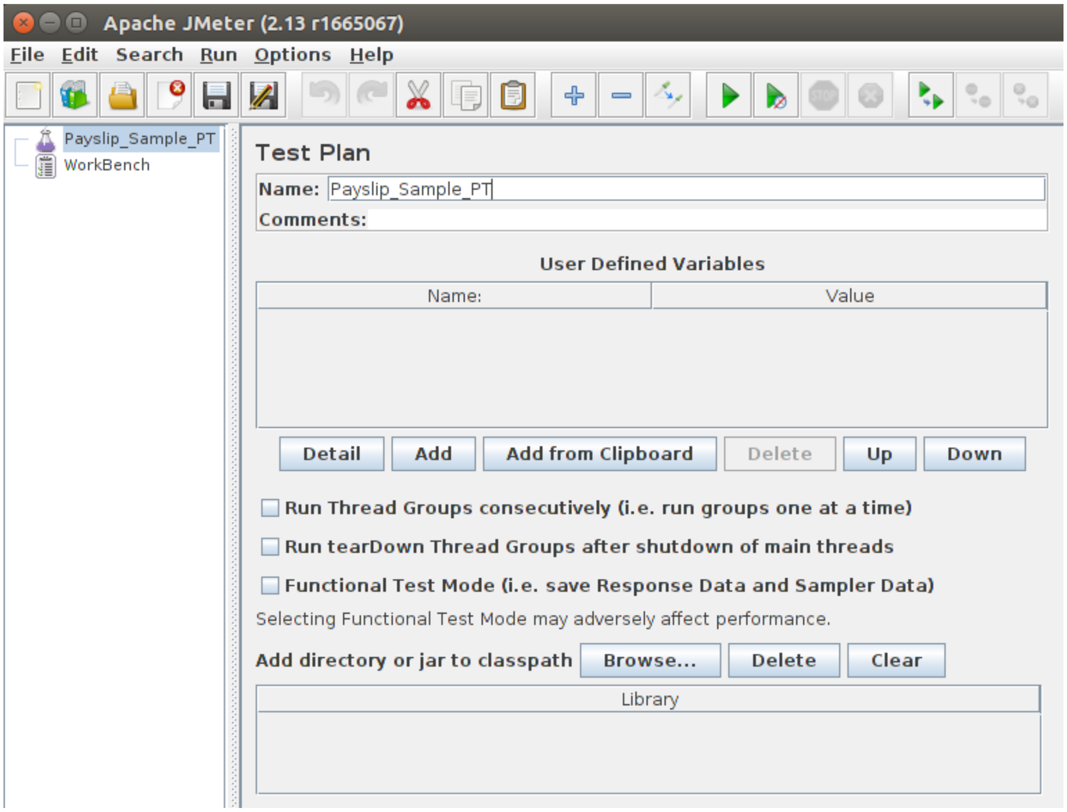
Creating a performance test case

The following are the steps to create a performance test case:

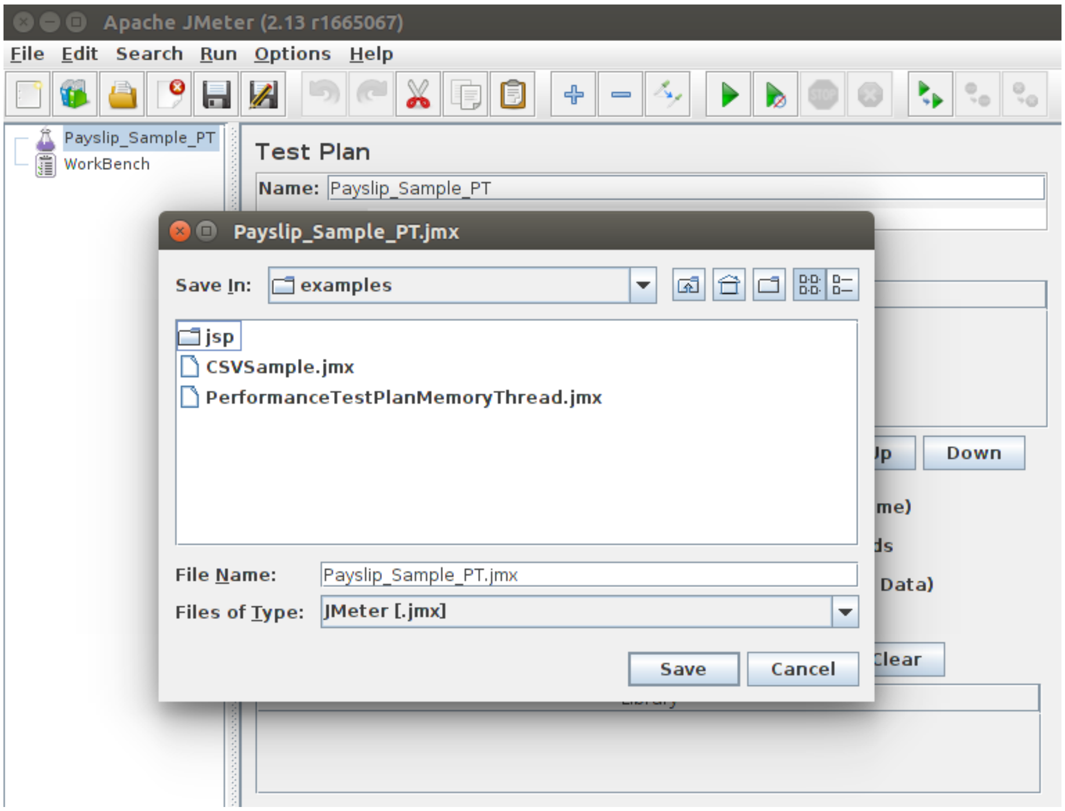
1. We will create a new test plan by modifying the existing template:



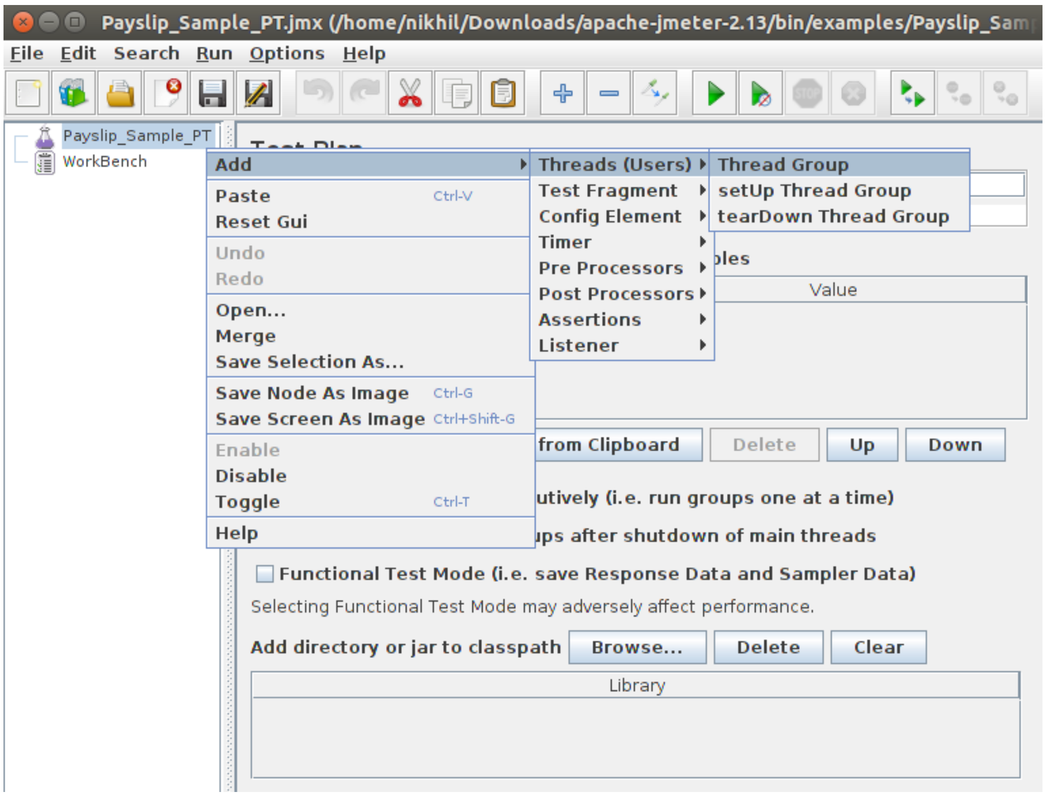
2. Rename the test plan to `Payslip_Sample_PT`, as shown in the following screenshot:



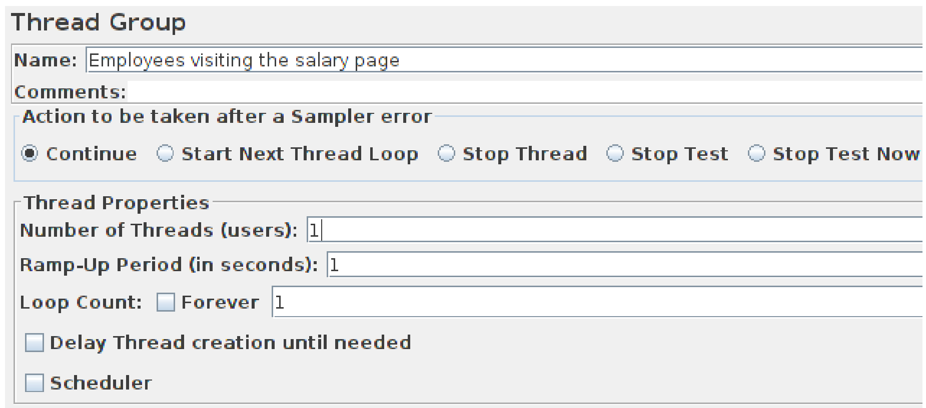
3. Save it inside the `examples` folder by clicking on the **Save** button from the menu items or by pressing *Ctrl* + *S*:



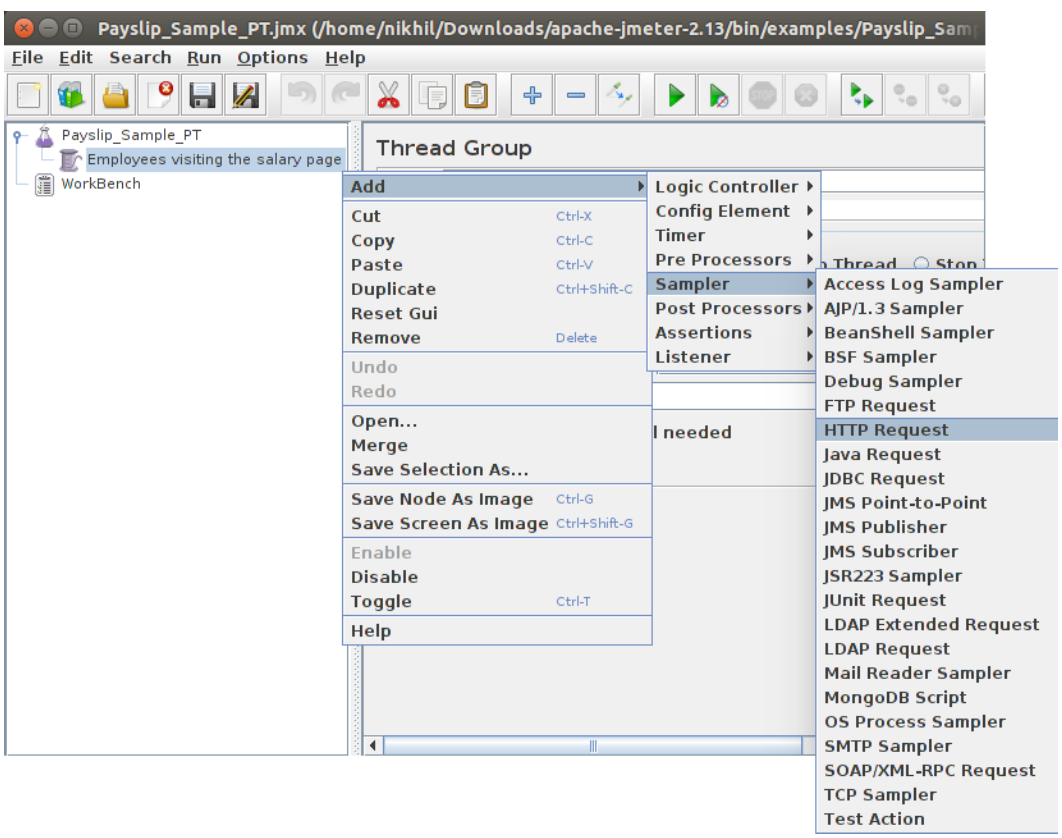
4. Add a thread group. To do so, right-click on the `Payslip_Sample_PT` and go to **Add** | **Threads (Users)** | **Thread Group**, as shown in the following screenshot:



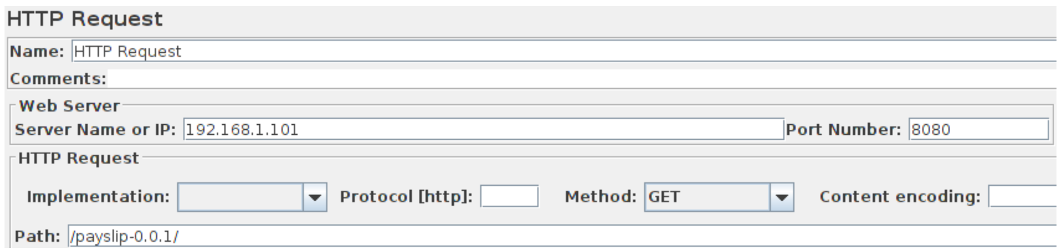
5. Name it appropriately and fill in the options as follows:
   - Select **Continue** for the option **Action to be taken after a Sampler error**
   - Add **Number of Threads (users)** = `1`
   - Add **Ramp-Up Period (in seconds)** = `1`
   - Add **Loop Count** = `1`



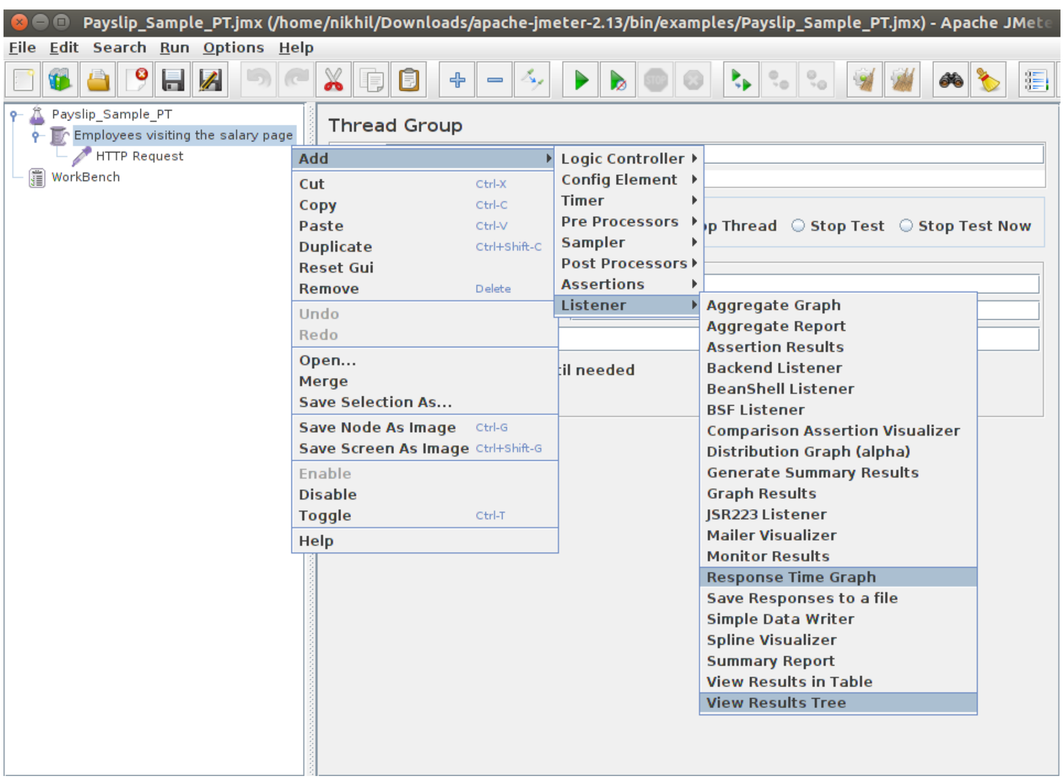
6. Add a sampler by right-clicking on `Payslip_Sample_PT` and navigating to **Add** | **Sampler** | **HTTP Request**, as shown in the following screenshot:



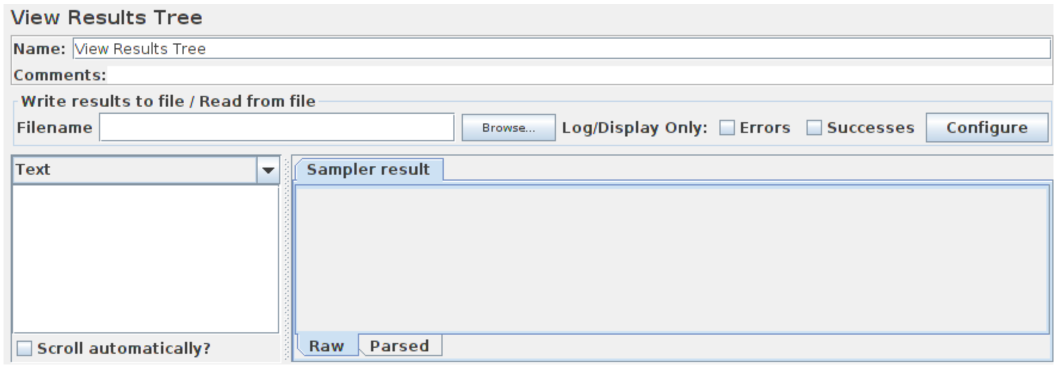
7. Name the **HTTP Request** appropriately and fill in the options as follows:
    - **Server Name or IP** = `<ip address of your testing server machine>`
    - **Port Number** = `8080`
    - **Path** = `/payslip-0.0.1/`



8. Add a listener by right-clicking on `Payslip_Sample_PT` and navigating to **Add** | **Listener** | **View Results Tree**, as shown in the following screenshot:



9. Leave all the fields with their default values:



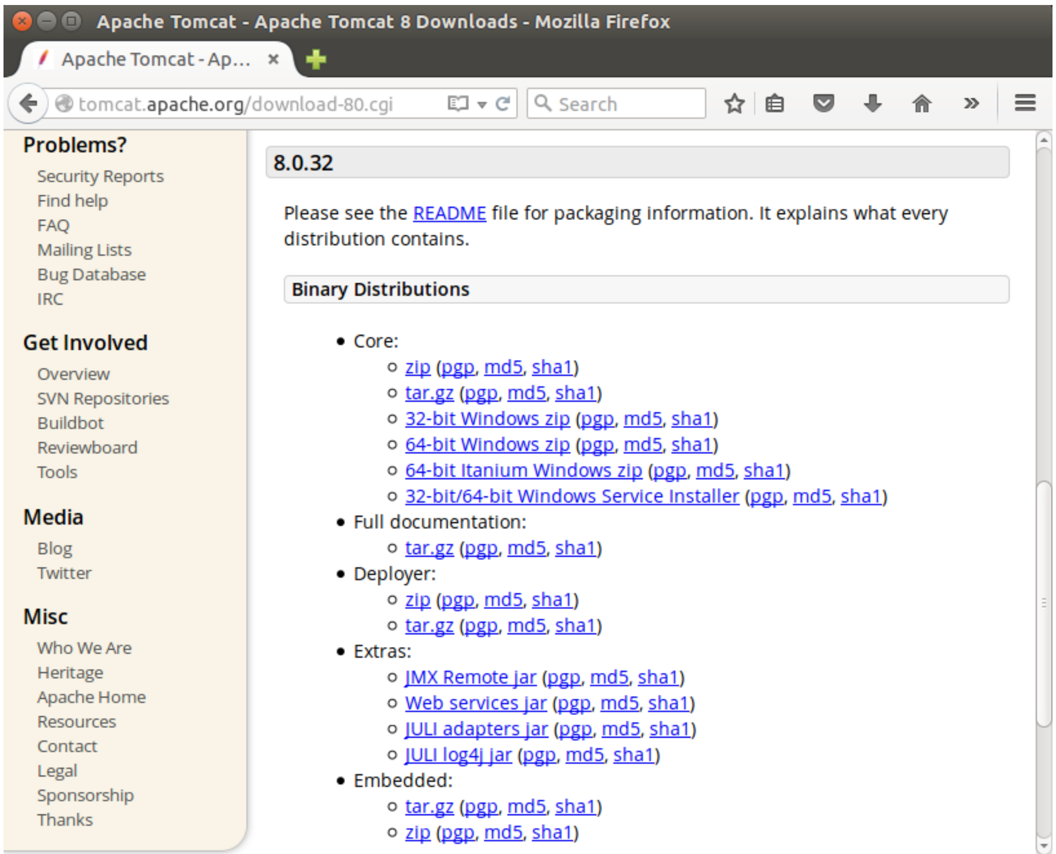
10. Save the whole configuration by clicking on the **Save** button in the menu items or by pressing *Ctrl* + *S*.

Installing the Apache Tomcat server on the testing server

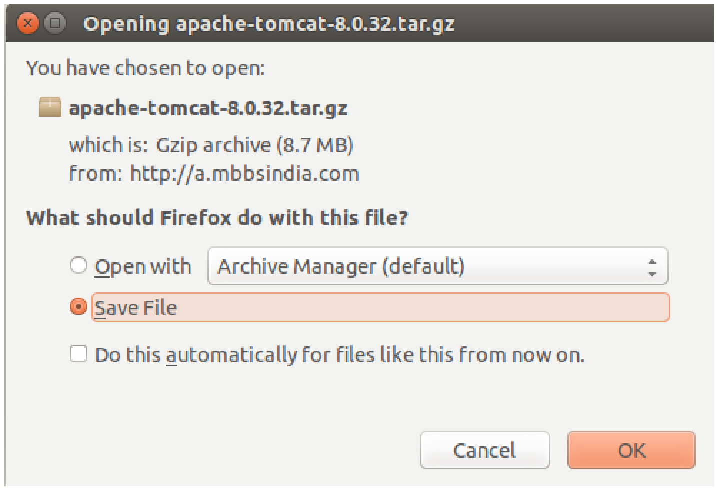
Installing the Apache Tomcat server on Ubuntu is simple. We are doing this to host our application so that it can be tested separately in an isolated environment.

The steps are as follows:

1. Download the latest Apache Tomcat sever distribution from `http://tomcat.apache.org/download-80.cgi`. Download the `tar.gz` file, as shown in the following screenshot:



2. Download the `tar.gz` file to the `Downloads` folder:



3. We're going to install Tomcat in the `/opt/tomcat` directory. To do so, open a terminal in Ubuntu.
4. Create the directory `tomcat` inside `opt` using the following command:

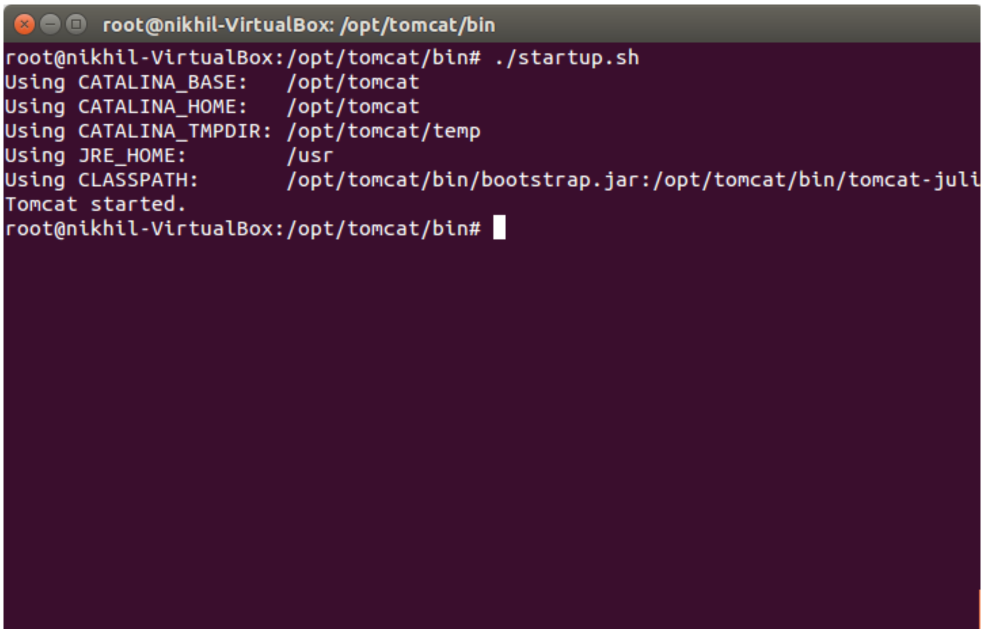
```
sudo mkdir /opt/tomcat
```

5. Then, extract the archive using the following command:

```
sudo tar xvf apache-tomcat-8*tar.gz -C /opt/tomcat --strip-
components=1
```

6. Start the Apache Tomcat server by executing the following command:

```
sudo su -
cd /opt/tomcat/bin
./startup.sh
```

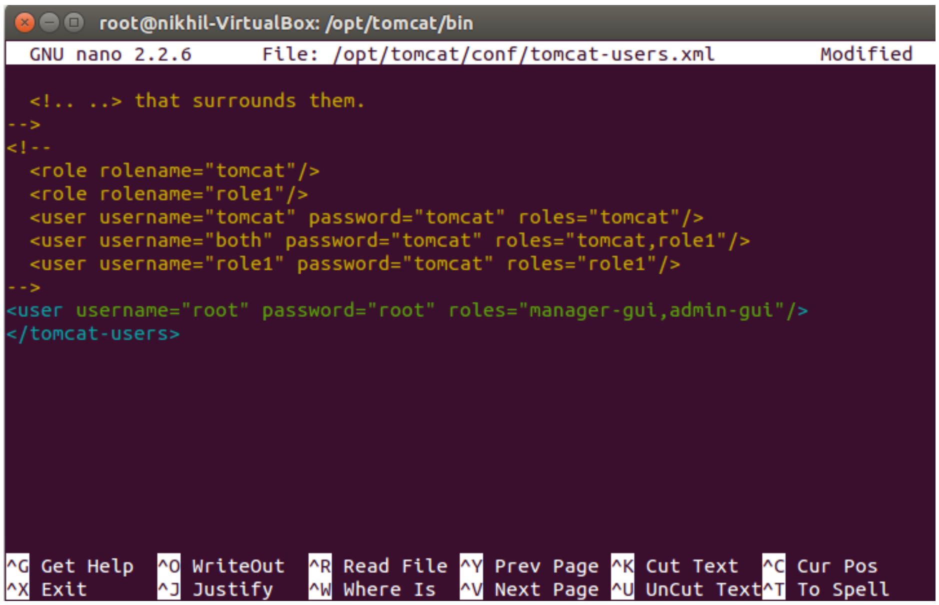


7. That's it! The Apache Tomcat server is up and running. To see it running, open the following link in your favorite web browser: `http://localhost:8080/`.
8. We must now create a user account in order to manage the services using the **manager app** feature that is available on the Apache Tomcat server's dashboard. We will do this by editing the `tomcat-users.xml` file.
9. To do so, use the following command in the terminal:

```
sudo nano /opt/tomcat/conf/tomcat-users.xml
```

10. Add the following line of code between `<tomcat-users>` and `</tomcat-users>`:

```
<user username=admin password=password roles=manager-
gui,admin-gui/>
```



11. Save and quit the `tomcat-users.xml` file by pressing *Ctrl* + *X* and then *Ctrl* + *Y*.
12. To put our changes into effect, restart the Tomcat service by executing the following commands:

```
cd /opt/tomcat/bin
sudo su -
./shutdown.sh
./startup.sh
```

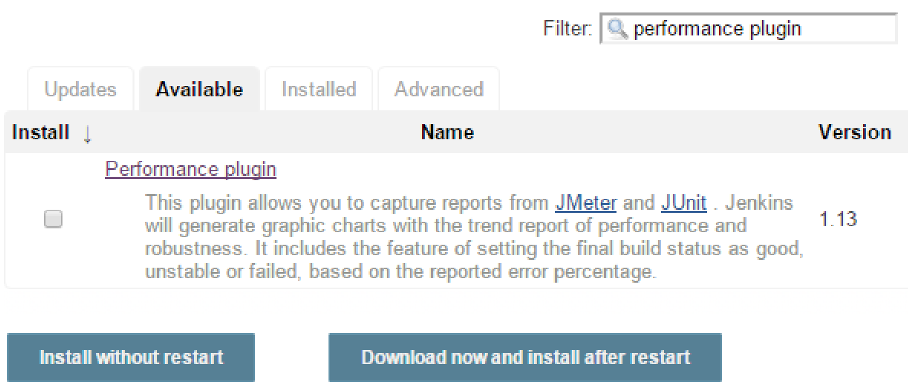
Jenkins configuration

In order to assist the Jenkins jobs that perform various functions to achieve Continuous Delivery, we need to make some changes in the Jenkins configuration. We will see some newly introduced features in Jenkins in this section.

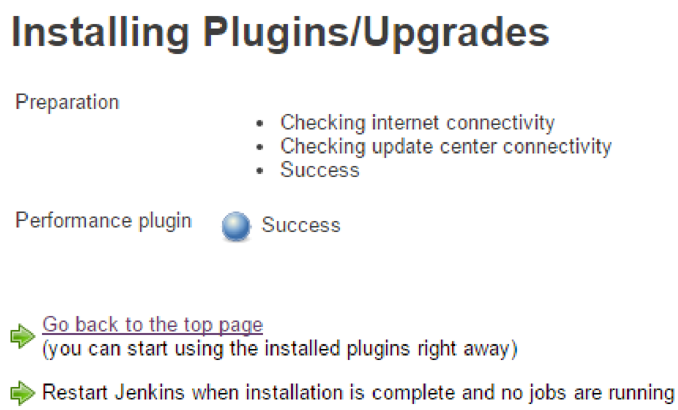
Configuring the performance plugin

The performance plugin will be used to publish performance test report. Follow the these steps to install it:

1. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `performance plugin` in the search box.
4. Select **Performance plugin** from the list and click on the **Install without restart** button:



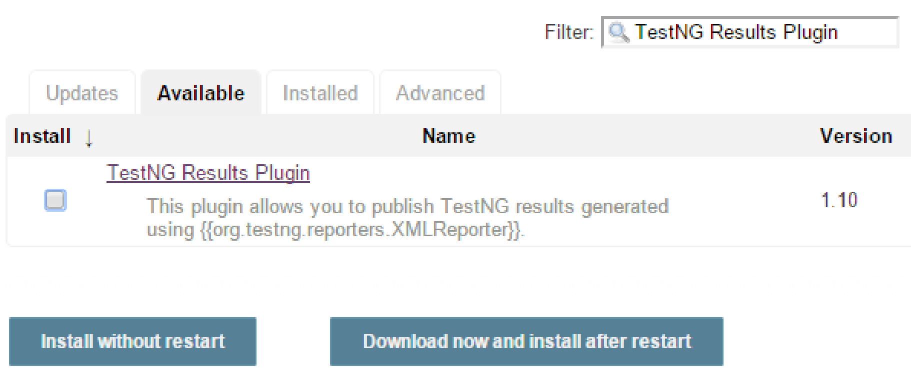
5. The download and installation of the plugin will start automatically:



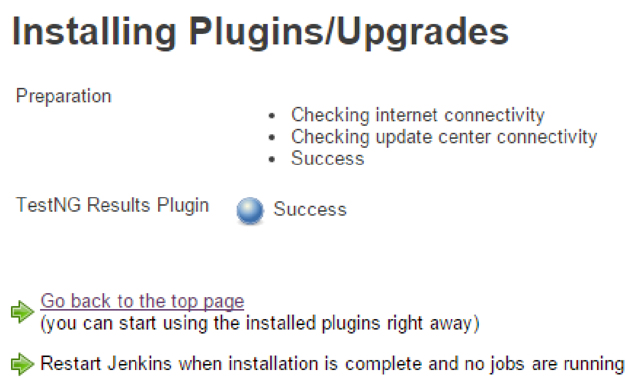
Configuring the TestNG plugin

The TestNG plugin will be used to publish the user acceptance test report. Follow these steps to install it:

1. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `TestNG Results Plugin` in the search box.
4. Select **TestNG Results Plugin** from the list and click on the **Install without restart** button, as shown in the following screenshot:



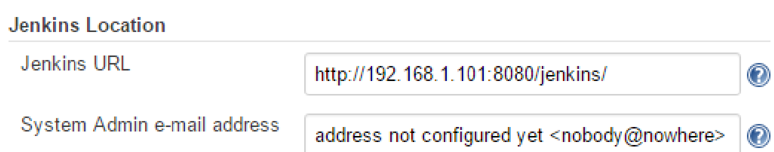
5. The download and installation of the plugin will start automatically:



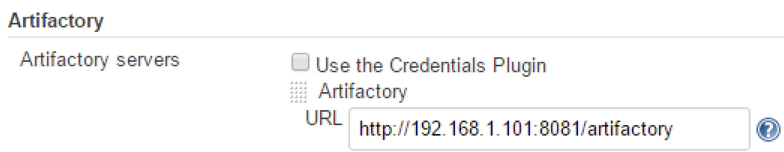
Changing the Jenkins/Artifactory/Sonar web URLs

You can ignore the following steps if your Jenkins/Artifactory and SonarQube URLs are configured to anything apart from localhost:

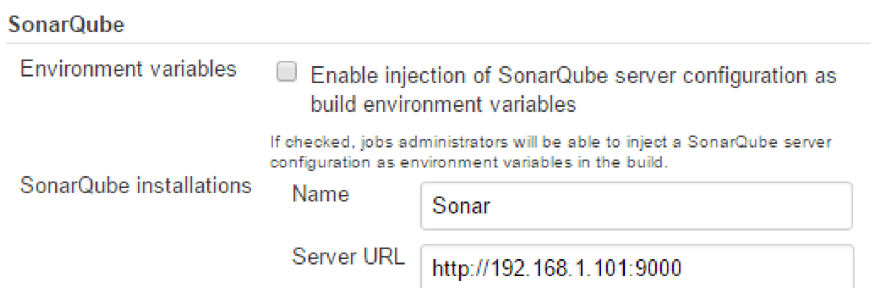
1. Go to the **Configure System** link from the **Manage Jenkins** page.
2. Scroll down until you see the **Jenkins Location** section. Modify **Jenkins URL** to `http://<ip address>:8080/jenkins` as shown in the following screenshot. `<ip address>` is the IP of your Jenkins server:



3. Scroll down until you see the **Artifactory** section. Modify the **URL** field to `http://<ip address>:8080/artifactory` as shown in the screenshot. `<ip address>` is the IP of your Artifactory server:



4. Scroll down until you see the **SonarQube** section. Modify the **Server URL** field to `http://<ip address>:9000` as shown in the following screenshot. `<ip address>` is the IP of your SonarQube server:

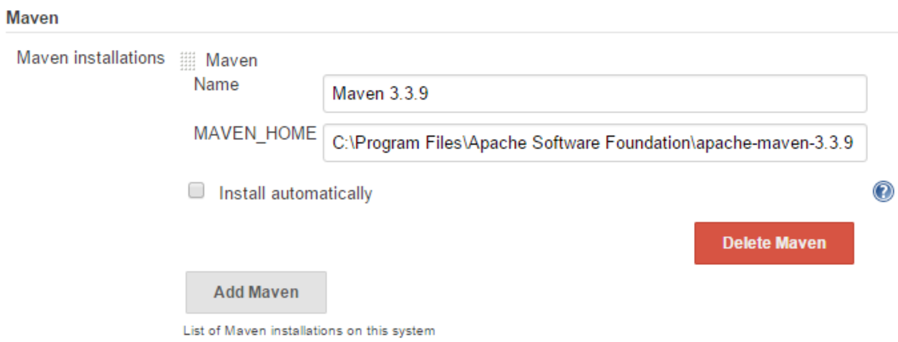


Modifying the Maven configuration

We already discussed Maven installation and configuration in *Chapter 4*, *Continuous Integration Using Jenkins – Part I*. Here, we need to install another instance of Maven in Jenkins. This is because in the up coming topics, we will configure Jenkins slave on the testing server. The Jenkins jobs to perform user acceptance testing and performance testing will run on the slave and will require a separate copy of Maven. The current Maven installation will only work for Jenkins jobs that run on the master node, that is, the Jenkins server itself. Follow the next few steps to configure another instance of Maven:

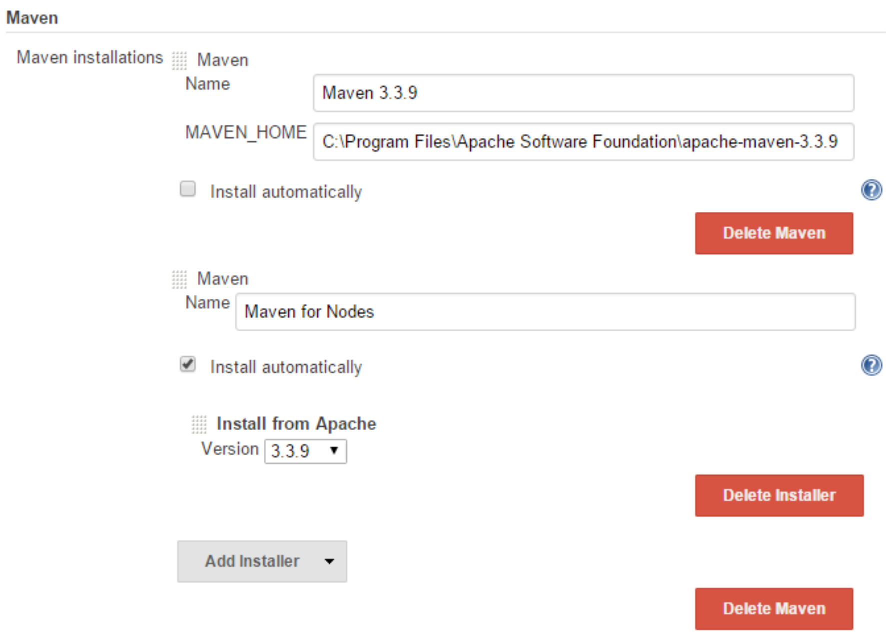
1. On the **Manage Jenkins** page, scroll down until you see the **Maven** section.

2. You will see **Maven installations** already present. Click on the **Add Maven** button to add a new one:



> We could have also installed Maven on the machine running the Jenkins slave. However, imagine a situation where there are many slaves. Installing Maven on all of them would be tiring and time consuming. Hence, the preceding configuration comes in handy—single installation, but multiple uses.

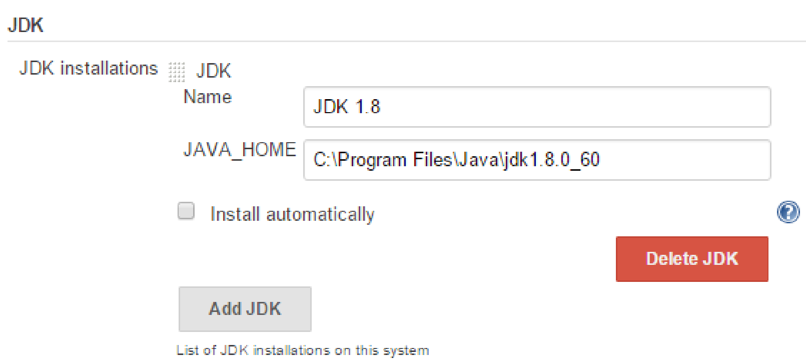
3. Name it `Maven for Nodes`, select the check box **Install automatically**, and choose the appropriate Maven version from the drop-down list. This is shown in the following screenshot:



Modifying the Java configuration

Like Maven, we also need to install another instance of Java to be used by the Jenkins jobs running on slave agents. Follow the next few steps to configure another instance of Java:

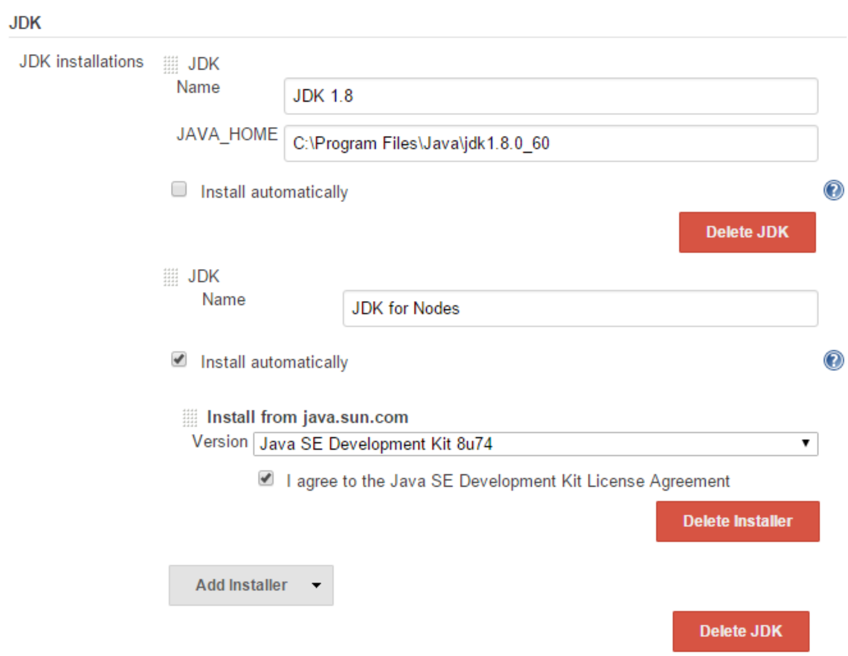
1. On the same page, scroll down until you see the **JDK** section.
2. You will see existing **JDK installations**. Click on the **Add JDK** button to add a new one:



3. Name it `JDK for Nodes`, select the check box **Install automatically**, and choose the appropriate JDK version from the drop-down list.

> The JDK version depends on your Maven project. In our example, we are using JDK 1.8.0.

4. Agree to the terms and conditions by checking the option **I agree to the Java SE Development License Agreement**.
5. The moment you do so, a new tab will open that will take you to the Oracle website asking you to sign in or log in using an Oracle account.
6. Log in using your existing Oracle account or create a new one. This is required to download the JDK:

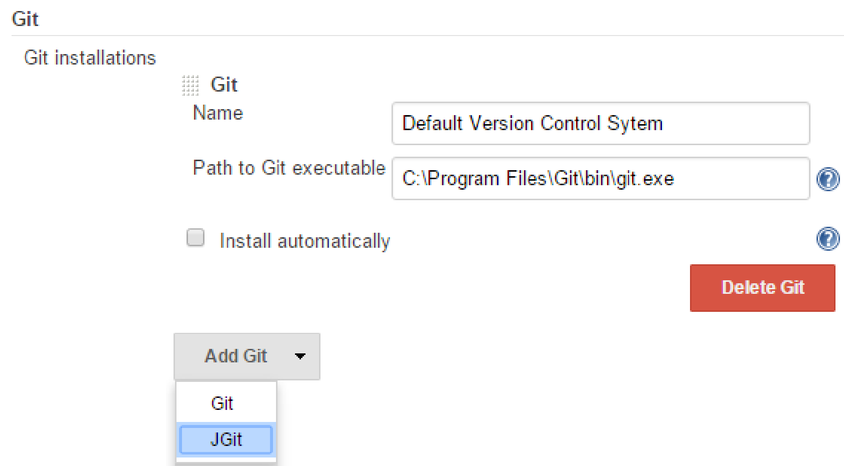


Modifying the Git configuration

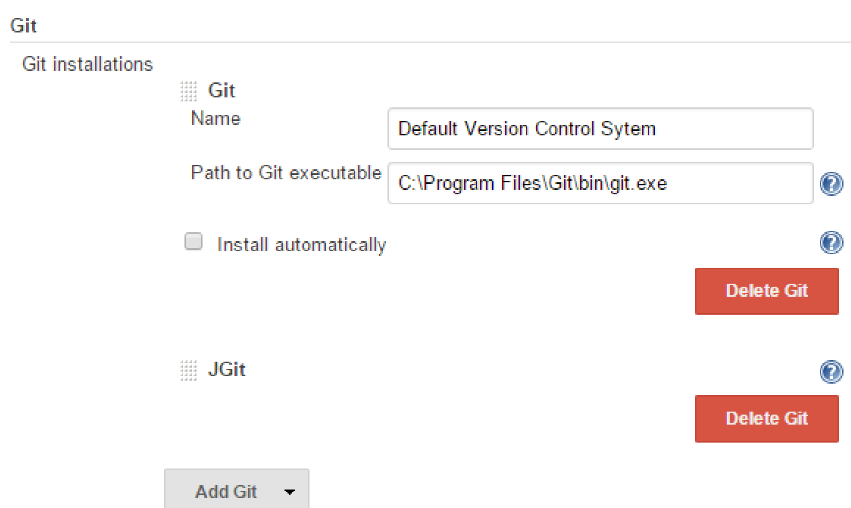
Just like Maven and Java, we also need to create another instance of Git to be used by Jenkins job running on slave machines. Follow these steps to configure another Git instance:

1. On the same page, scroll down until you see the **Git** section.

2. You can see existing **Git installations**. Click on the **Add Git** button to add a new one.
3. From the options under **Add Git** menu, select **JGit**. This is an experimental feature:

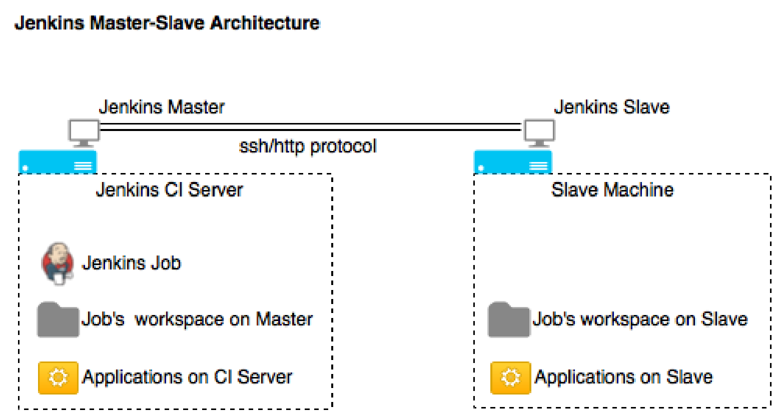


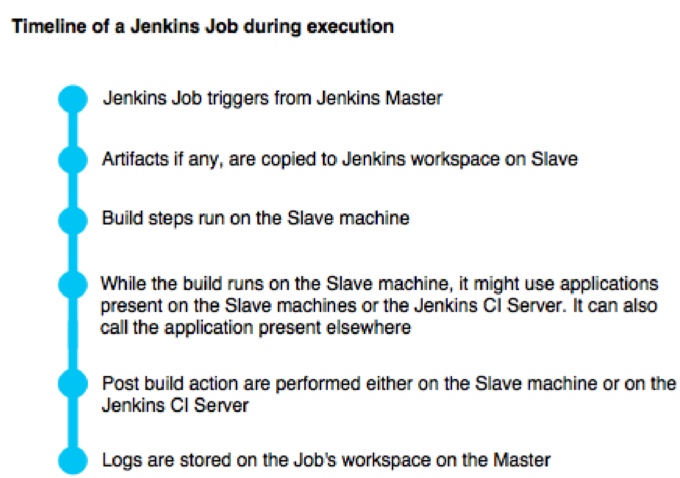
4. That's it! There are no other configurations to it:



Configuring Jenkins slaves on the testing server

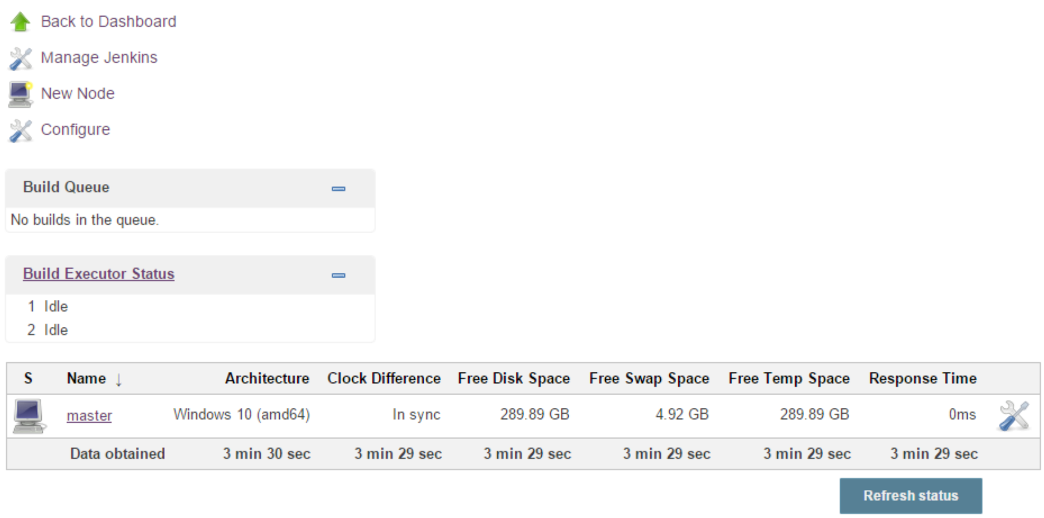
In the previous section, we saw how to configure the testing server. Now, we will see how to configure the Jenkins slave to run on the testing server. In this way, the Jenkins master will be able to communicate and run Jenkins jobs on the slave. Follow the next few steps to set up the Jenkins slave:



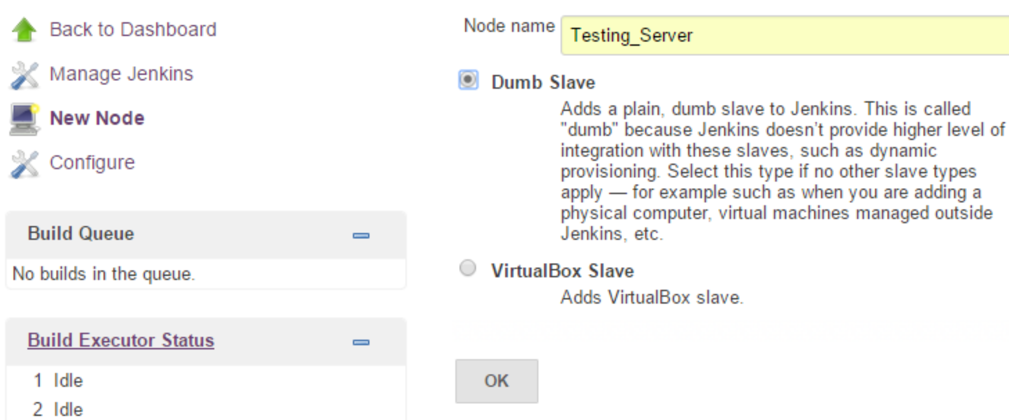


1. Log in to the testing server and open the Jenkins dashboard from the browser using the following link: `http://<ip address>:8080/jenkins/`. Remember, you are accessing the Jenkins master from the testing server. `<ip address>` is the IP of your Jenkins server.

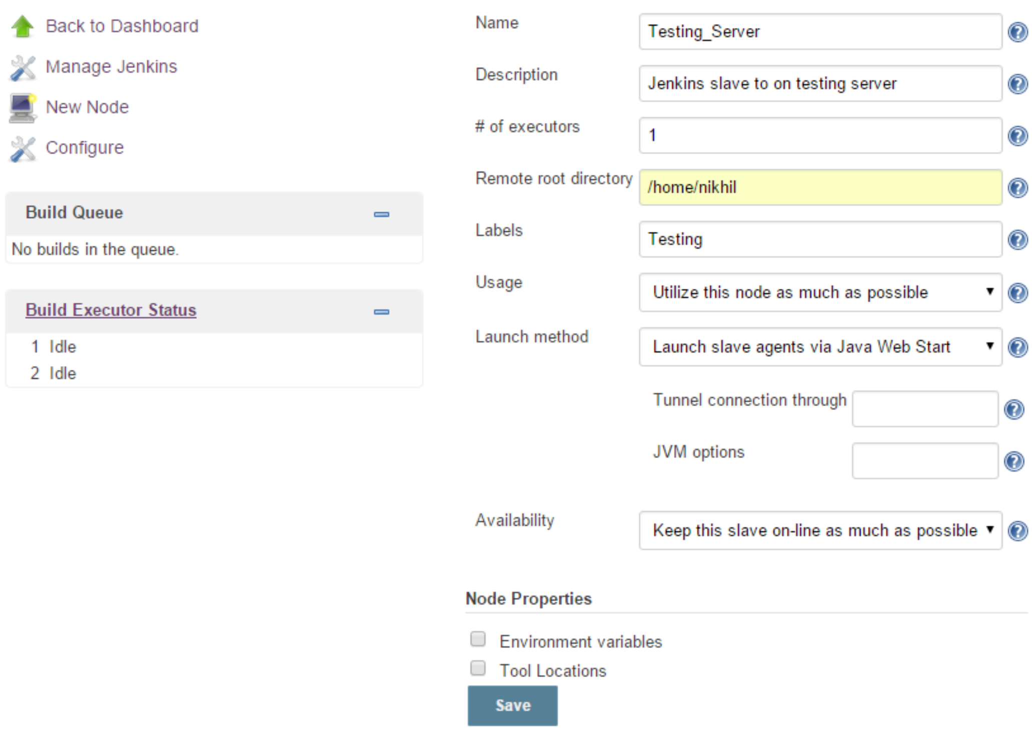
2. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page. Make sure you have logged in as an `Admin` in Jenkins.
3. Click on the **Manage Nodes** link. In the following screenshot, we can see that the master node (which is the Jenkins server) is listed:



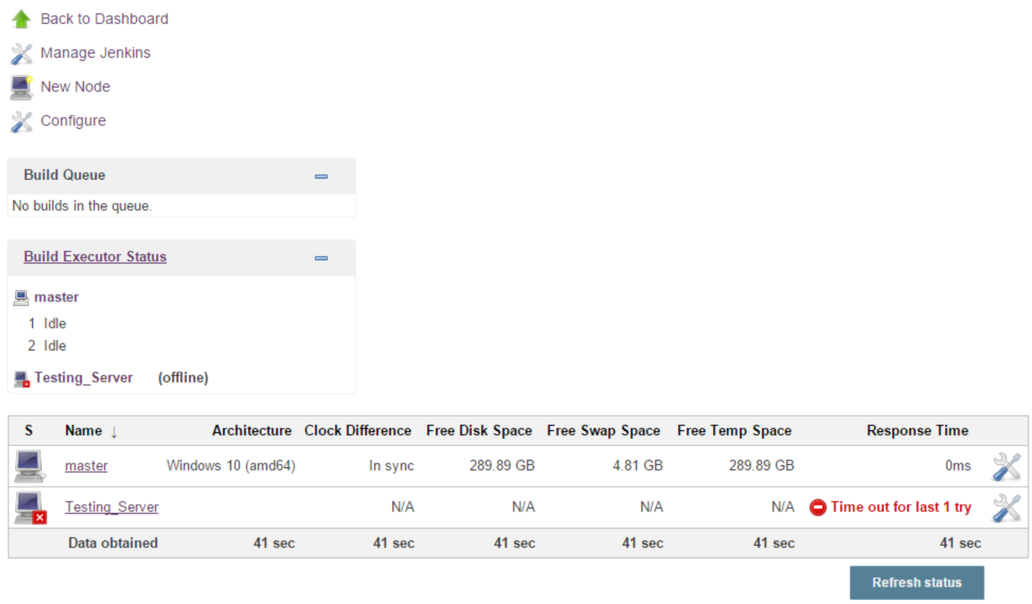
4. Click on the **New Node** button on the left-hand side panel. Name the new node `Testing_Server` and select the option **Dumb Slave**. Click on the **OK** button to proceed:



5. Add some description, as shown in the next screenshot. The **Remote root directory** value should be the local user account on the testing server. It should be `/home/<user>`. The **Labels** filed is extremely important, so add `Testing` as the value.
6. The **Launch Method** should be **launch slave agents via Java Web Start**:



7. Click on the **Save** button. As you can see from the following screenshot, the Jenkins node on the testing server is configured but it's not running yet:

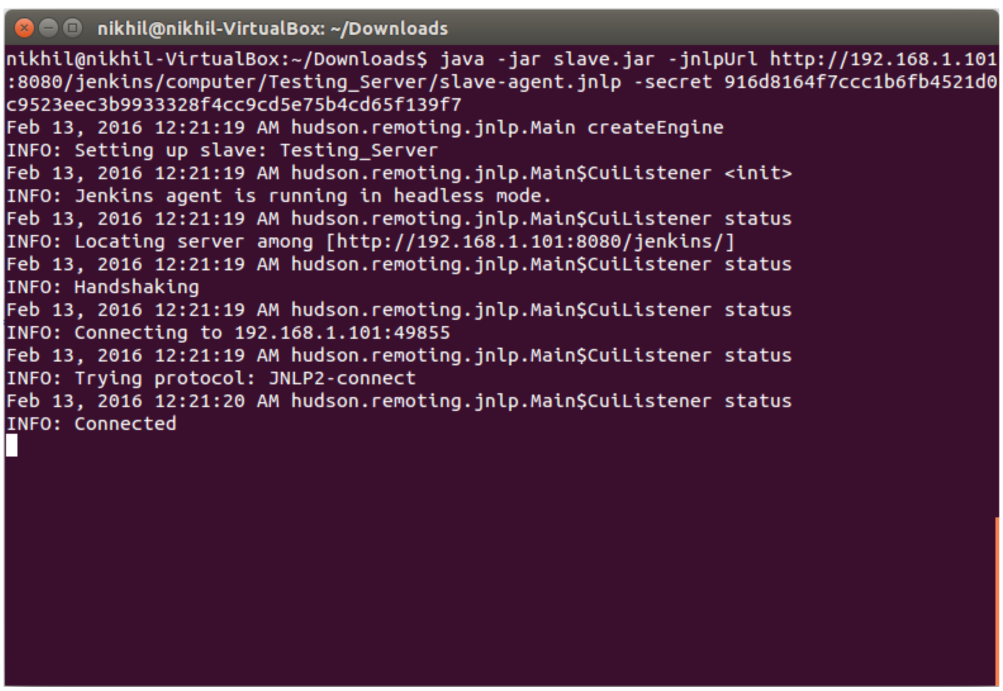


8. Click on the **Testing_Server** link from the list of nodes. You will see something like this:

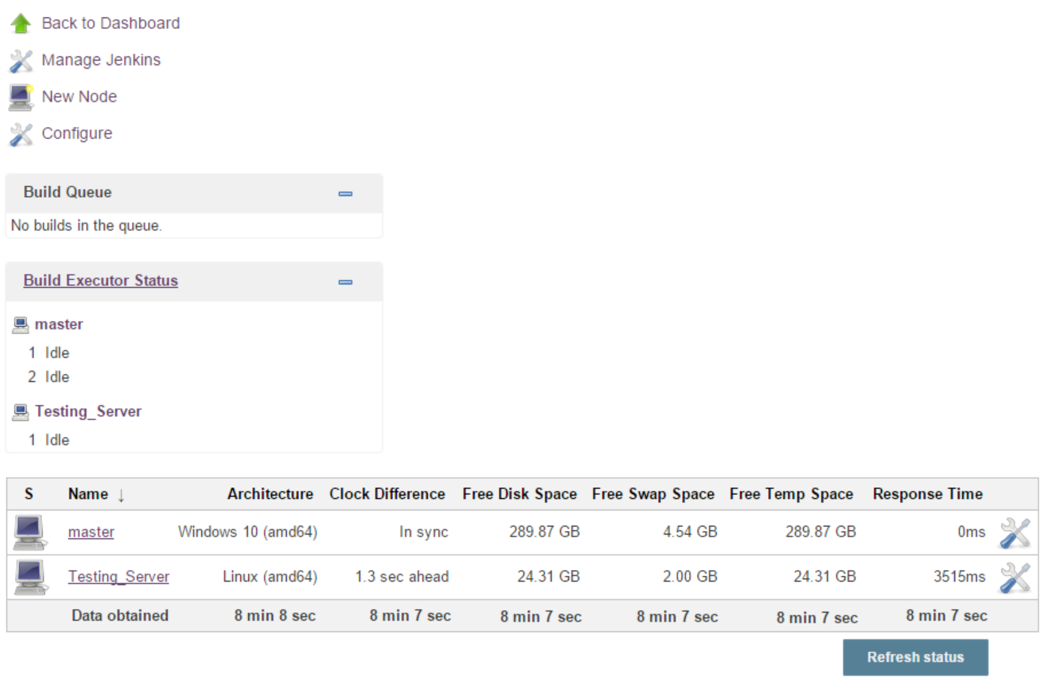
9. You can either click on the orange **Launch** button, or you can execute the long command mentioned below it from the terminal.
10. If you choose the latter option, then download the slave.jar file mentioned in the command by clicking on it. It will be downloaded to /home/<user>/Downloads/.
11. Execute the following commands in sequence:

```
cd Downloads


java -jar slave.jar -jnlpUrl

http://192.168.1.101:8080/jenkins/computer/
Testing_Server/slav e-agent.jnlp -secret
916d8164f7ccc1b6fb4521d0c9523eec3b9933328f4cc9cd5e75b4cd65f139f7
```

> The preceding command is machine specific. Do not copy and paste and execute the same. Execute the command that appears on your screen.



12. The node on testing server is up and running, as shown in the following screenshot:



Creating Jenkins Continuous Delivery pipeline

This Continuous Delivery pipeline contains five Jenkins jobs: two old and three new ones. In the current section, we will first modify the two existing Jenkins Jobs, and later we will create three new Jenkins Jobs.

Modifying the existing Jenkins job

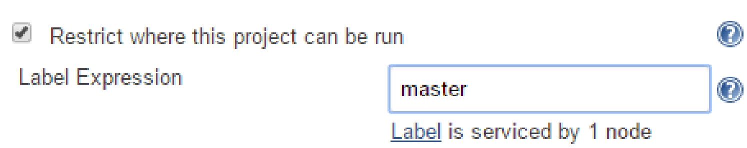
Before we begin creating new jobs in Jenkins to achieve Continuous Delivery, we need to modify all the existing ones. The modifications that we intend to do are of two types:

- **Map all the existing Jenkins jobs to a particular Jenkins node**: We will do this by modifying advanced project options in all the existing Jenkins jobs. This is because the existing Jenkins jobs are currently running on the Jenkins master node; this is a default behavior. However, since we have introduced a new Jenkins slave node, it's important to tell all the Jenkins jobs where to run. Not doing so will make Jenkins jobs choose nodes by themselves, leading to failures.
- **Modifying the method through which a Jenkins job triggers another Jenkins job**: In our current Jenkins pipeline, which is the Continuous Integration pipeline, the triggering phenomena for connected Jenkins jobs are very simple. A Jenkins job simply triggers another Jenkins job without passing any parameters. That was fine as long as we didn't feel the need to do so. However, now we need to pass some important parameters across the pipeline for use.

Modifying the advanced project

Follow these steps for all the Jenkins jobs:

1. From the Jenkins dashboard, begin by clicking on any existing Jenkins job.
2. Click on the **Configure** link present on the left-hand side panel.
3. Scroll down until you see the **Advanced Project Options** section.
4. From the options, choose **Restrict where this project can be run** and add `master` as the value for the **Label Expression** field, as shown in the following screenshot:

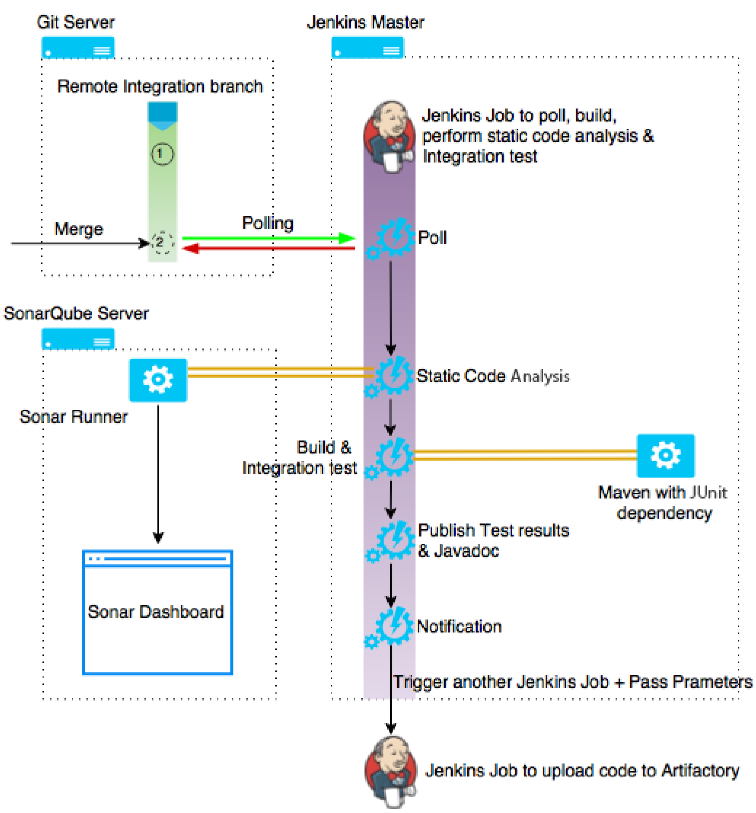


Modifying the Jenkins job that performs the Integration test and static code analysis

The first Jenkins job in the pipeline performs the following tasks:

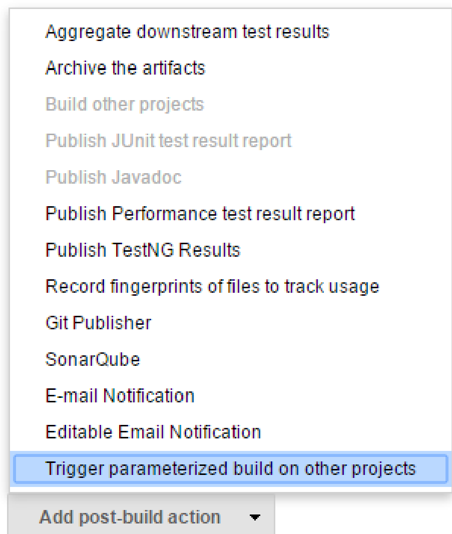
- It polls the integration branch for changes at regular intervals
- It performs a static code analysis of the downloaded code
- It executes the integration tests
- It passes `GIT_COMMIT` variable to the Jenkins job that uploads the package to Artifactory (new functionality)

The following figure will help us understand what the following Jenkins job does. It's a slightly modified version of what we saw in the previous chapter:

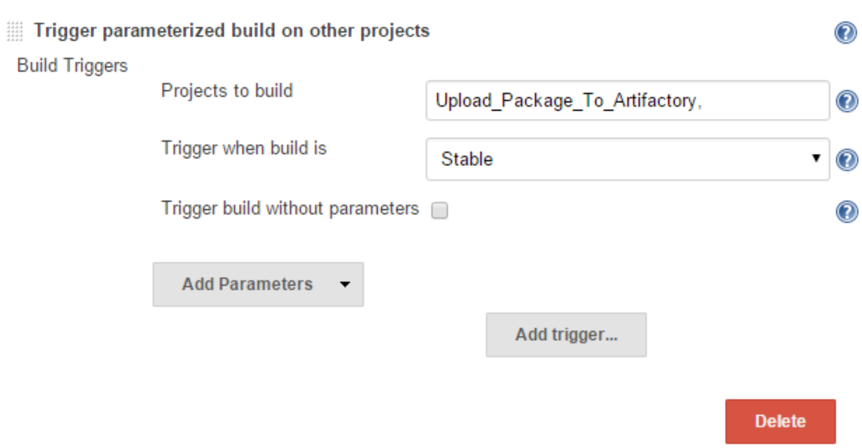


Follow the next few steps to create it:

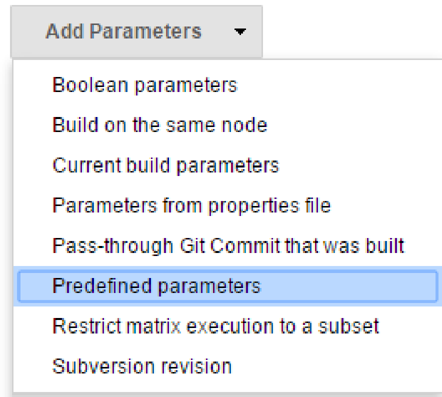
1. From the Jenkins dashboard, click on the `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch` job.
2. Click on the **Configure** link present on the left-hand side panel.
3. Scroll down until you see **Post-build Actions** section.
4. Click on the **Add post-build action** button and from the drop-down list, choose the option **Trigger parameterized build on the other projects**:



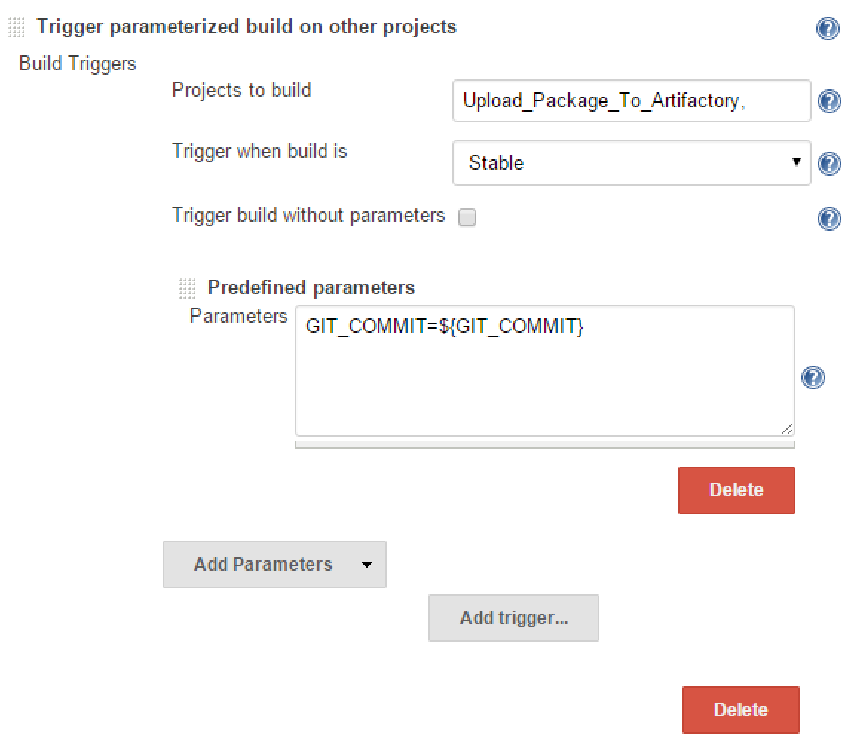
5. Add the values as shown in the next screenshot:



6. Click on the **Add Parameters** button and choose **Predefined parameters**, as shown in the following screenshot:



7. Add the values as shown in the following screenshot:



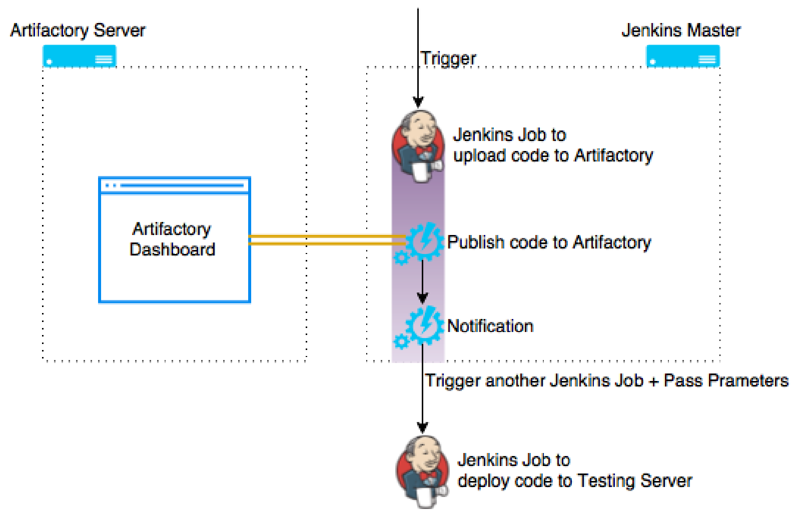
8. Save the Jenkins job by clicking on the **Save** button.

Modifying the Jenkins job that uploads the package to Artifactory

The second Jenkins job in the pipeline performs the following tasks:

- It uploads the built package to the binary repository
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variables to the Jenkins job that deploys the package in the testing server (new functionality)

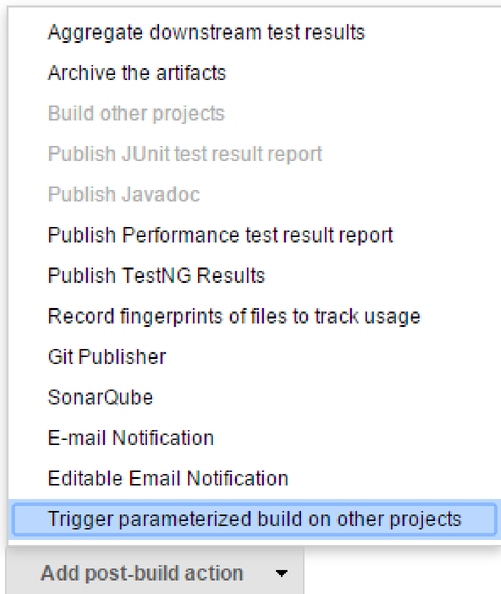
The following figure will help us understand what the following Jenkins job does. It's a slightly modified version of what we saw in the previous chapter:



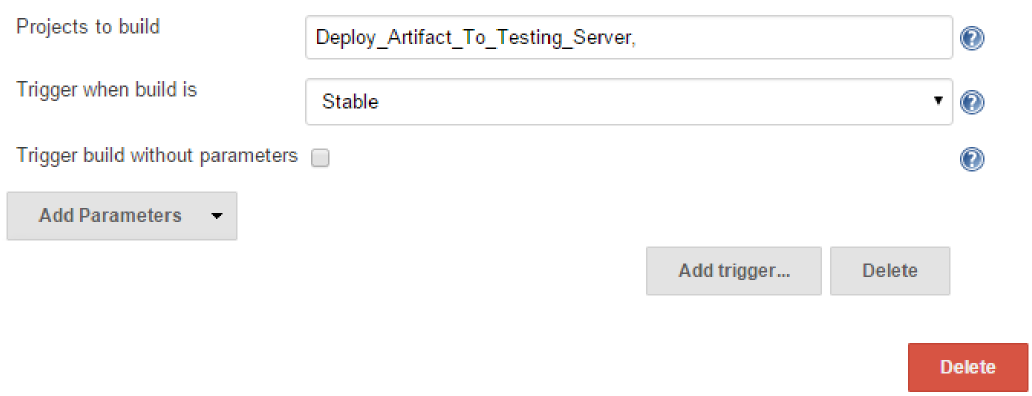
Follow the next few steps to create the Jenkins job:

1. From the Jenkins dashboard, click the on `Upload_Package_To_Artifactory` job.
2. Click on the **Configure** link present on the left-hand side panel.
3. Scroll down to the **Build Triggers** section and deselect the **Build after other projects are built** option.
4. Scroll down until you see **Post-build Actions** section.

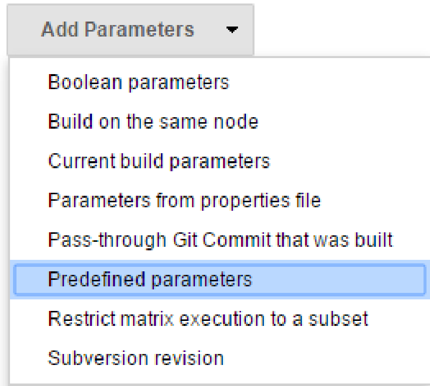
5. Click on the **Add post-build action** button and choose the option **Trigger parameterized build on the other projects** from the drop-down list:



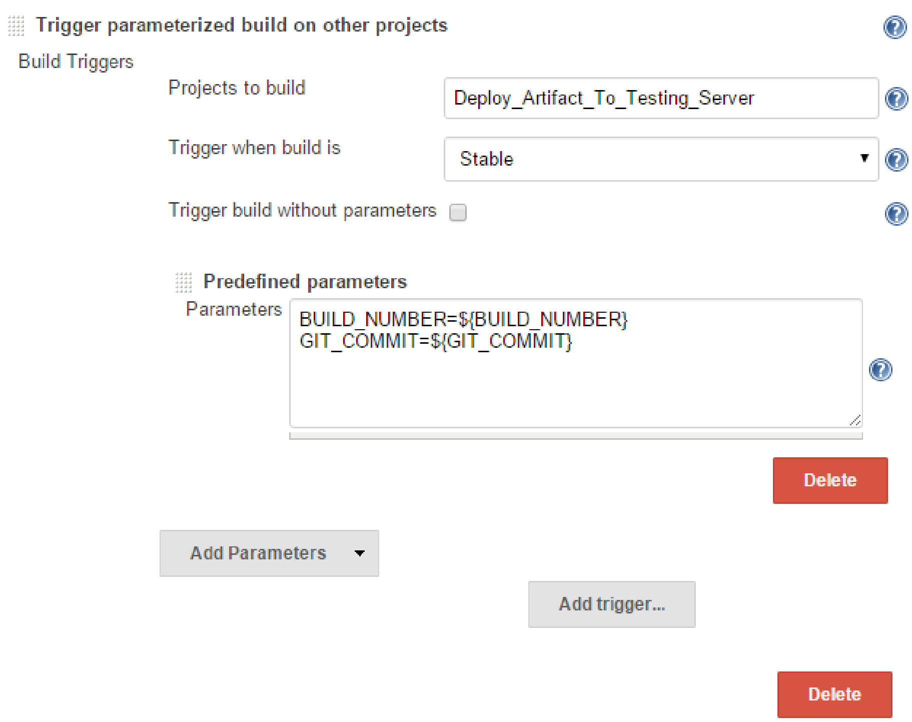
6. Add the values as shown in the screenshot:



7. Click on the **Add Parameters** button and choose **Predefined parameters**:



8. Add the values as shown in the screenshot:



9. Save the Jenkins job by clicking on the **Save** button.

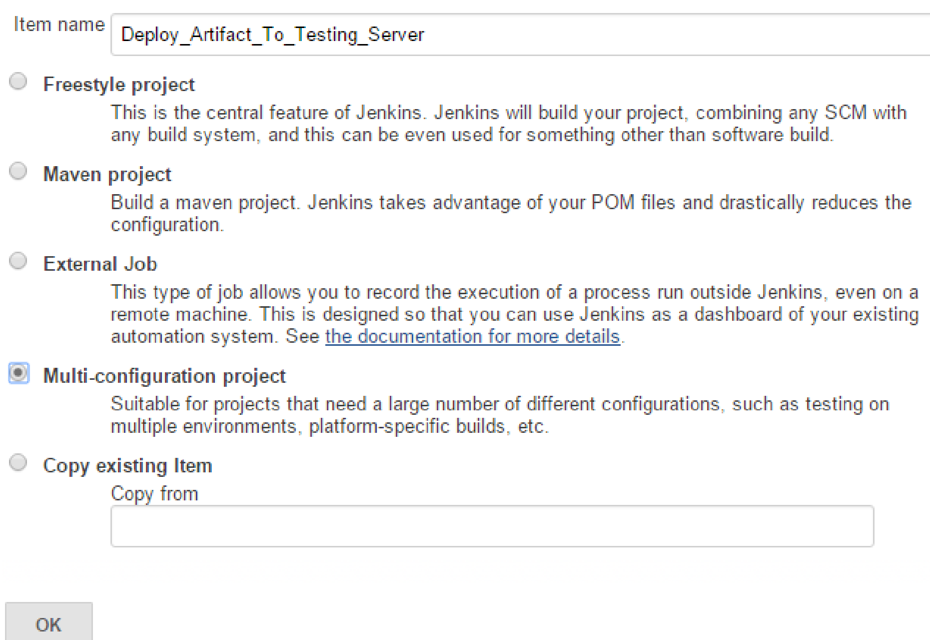
Creating a Jenkins job to deploy code on the testing server

The third job in the Continuous Delivery pipeline performs the following tasks:

- It deploys packages to the testing server using the `BUILD_NUMBER` variable
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variable to the Jenkins job that performs the user acceptance test

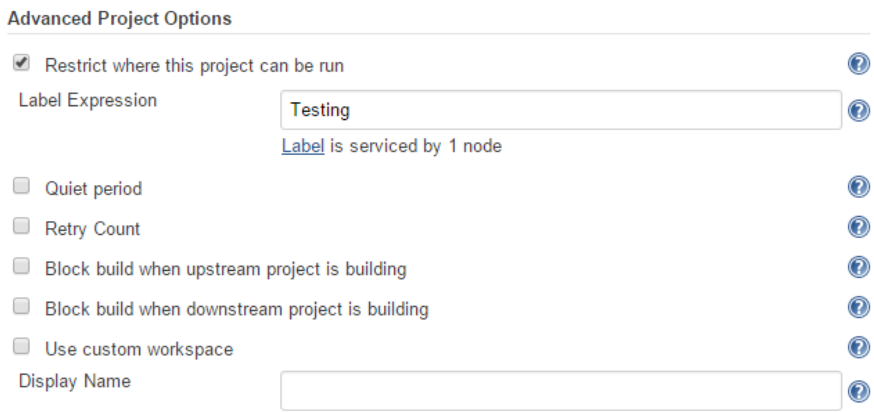
Follow the next few steps to create it:

1. From the Jenkins dashboard, click on **New Item**.
2. Name your new Jenkins job `Deploy_Artifact_To_Testing_Server`.
3. Select the type of job as **Multi-configuration project** and click on **OK** to proceed:

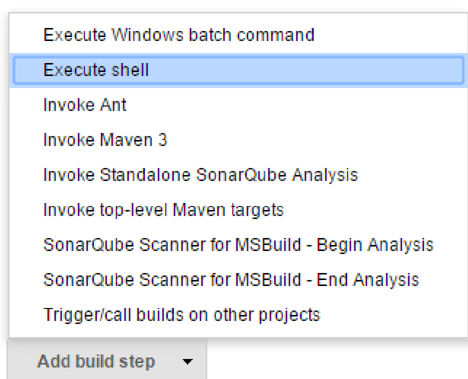


4. Scroll down until you see **Advanced Project Options**. Select **Restrict where this project can be run**.

5. Add `Testing` as the value for **Label Expression**:



6. Scroll down to the **Build** section.
7. Click on the **Add build step** button and choose the option **Execute shell**:



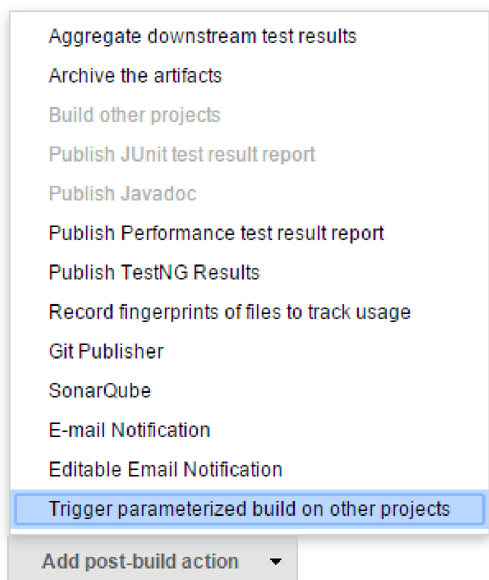
8. Add the following code in the **Command** field:
    - The first line of the command downloads the respective package from Artifactory to the Jenkins workspace:

```
wget http://192.168.1.101:8081/artifactory/
projectjenkins/$BUILD_NUMBER/payslip-0.0.1.war
```

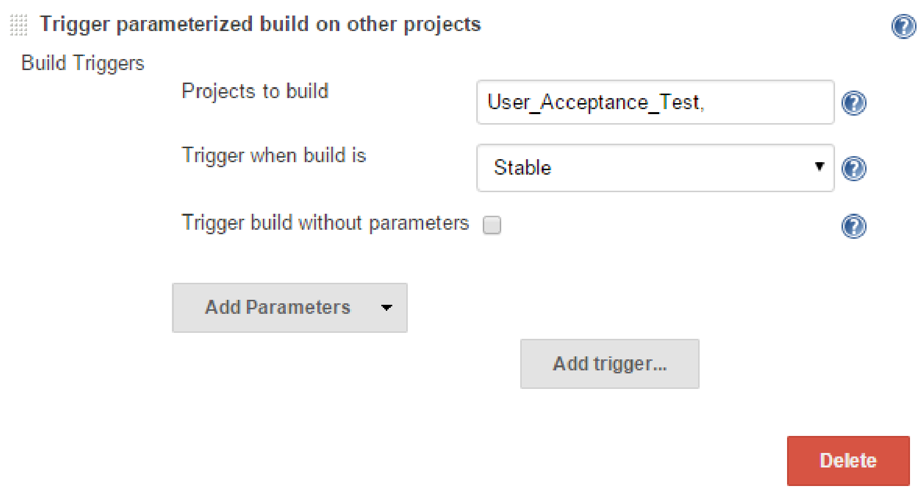
- The second line of command deploys the downloaded package to the Apache Tomcat server's `webapps` directory:

```
mv payslip-0.0.1.war /opt/tomcat/webapps/payslip-0.0.1.war -f
```

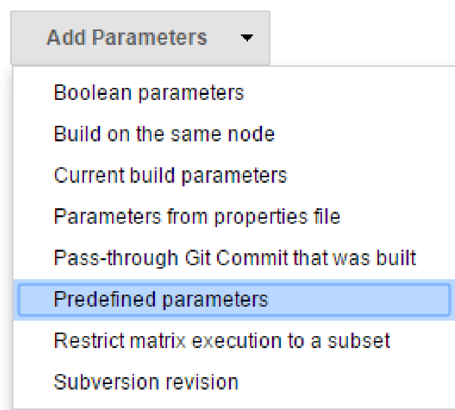
9. Scroll down to the **Post-build Actions** section. Click on the **Add post-build action** button. From the drop-down list, choose the option **Trigger parameterized build on the other projects**:



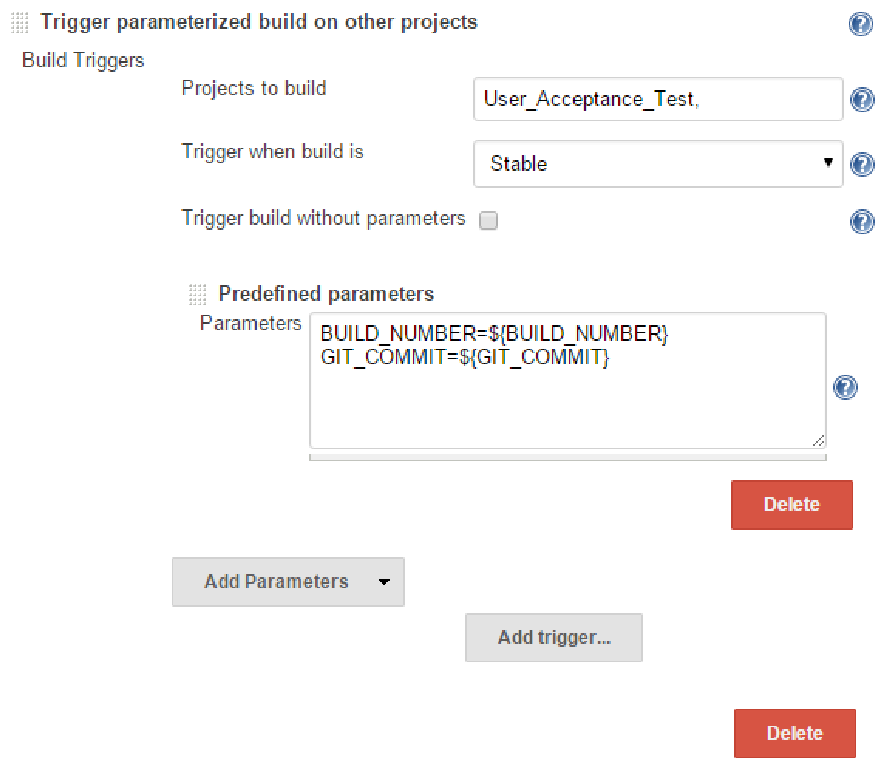
10. Add the values as shown in the screenshot:



11. Along with triggering the build, we would also like to pass some predefined parameters to it. Click on the **Add Parameters** button and select **Predefined parameters**:



12. Add the following values:



13. Save the Jenkins job by clicking on the **Save** button.

Creating a Jenkins job to run UAT

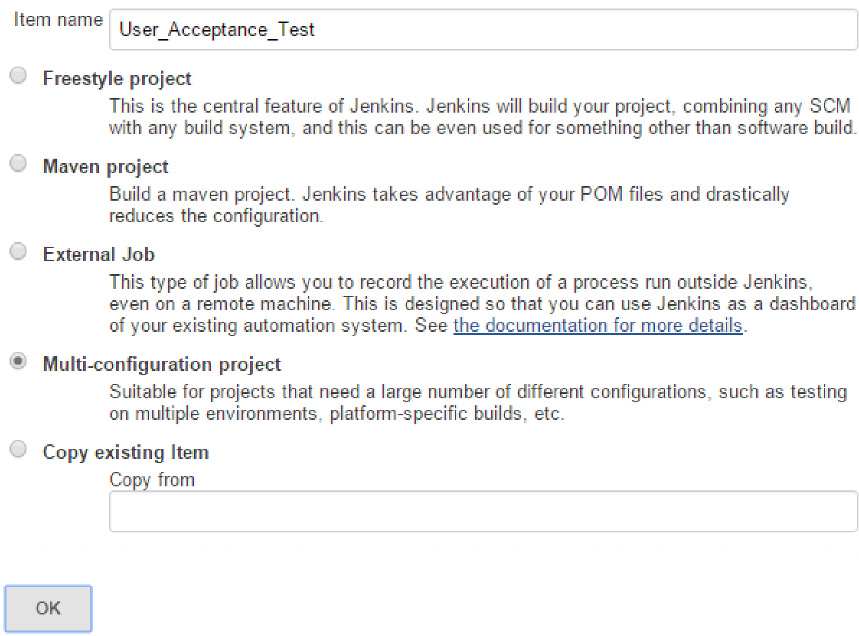
The fourth job in the Continuous Delivery pipeline performs the following tasks:

- It downloads the code from Git using the `GIT_COMMIT` variable
- It performs the user acceptance test
- It generates the test results report
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variables to the Jenkins job that performs performance test

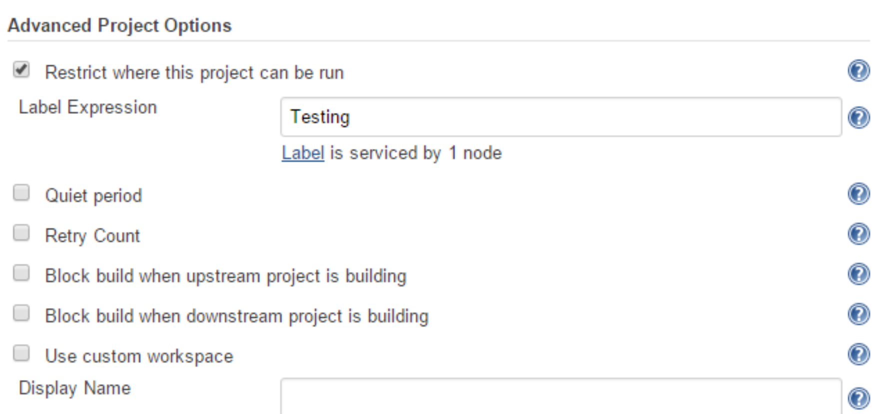
Follow the next few steps to create it:

1. From the Jenkins dashboard, click on **New Item**.
2. Name your new Jenkins job `User_Acceptance_Test`.

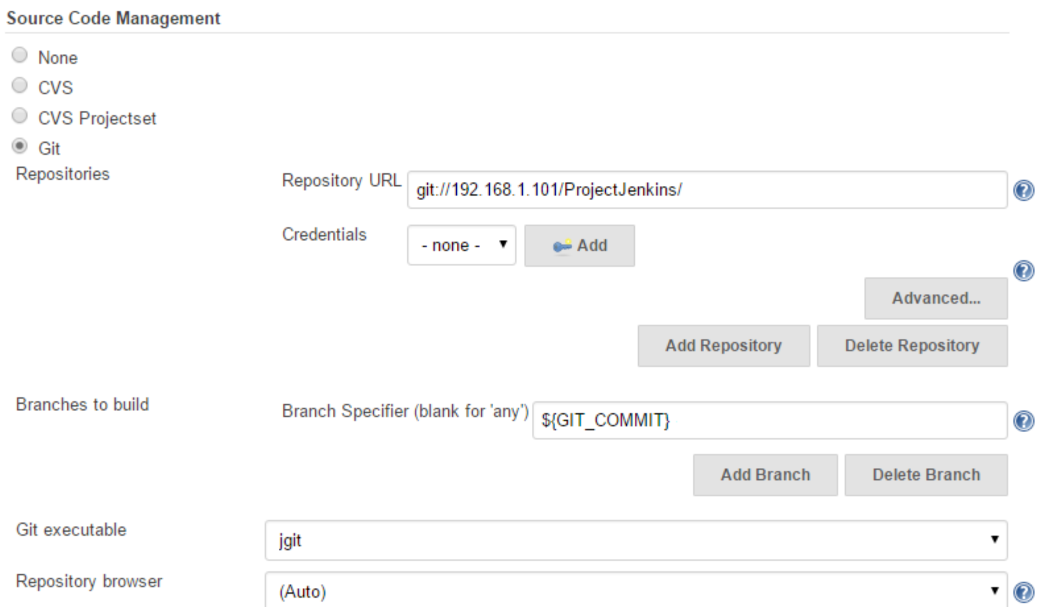
3. Select the type of job as **Multi-configuration project** and click on **OK** to proceed:



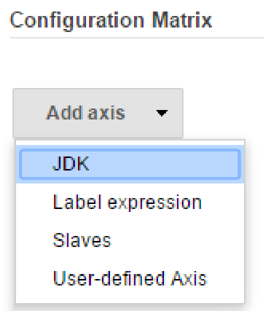
4. Scroll down until you see **Advanced Project Options**. Select **Restrict where this project can be run**.
5. Add `Testing` as the value for **Label Expression**:



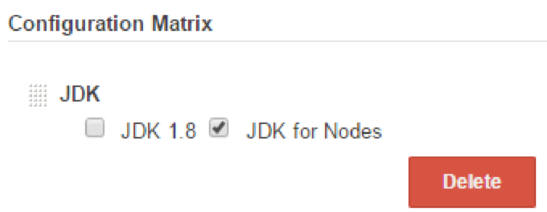
6. Scroll down to the **Source Code Management** section.
7. Select the **Git** option and fill in the blanks as follows:
    - **Repository URL** is the location of the Git repository. It can be a GitHub repository or a repository on a Git server. In our case it's `git://<ip address>/ProjectJenkins/`, where `<ip address>` is the Jenkins server IP.
    - Add `${GIT_COMMIT}` in the **Branches to build** section. `${GIT_COMMIT}` is the variable that contains the SHA-1 checksum value. Each Git commit has a unique SHA-1 checksum. In this way, we can track which code to build:



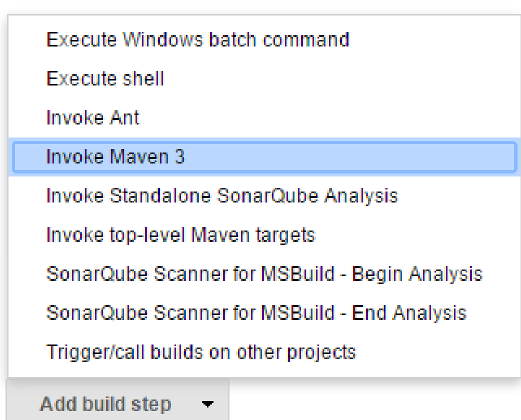
8. Scroll down to the **Configuration Matrix** section and click on the **Add axis** button. Select **JDK**:



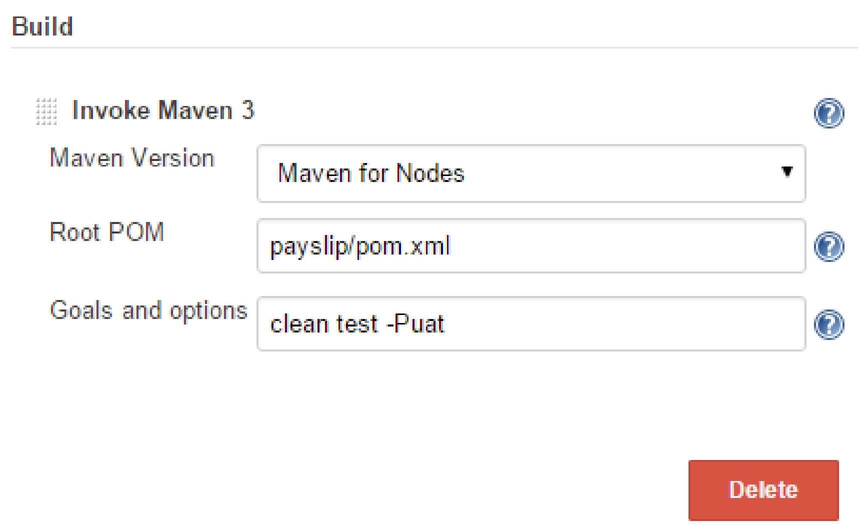
9. If you remember, we have two JDK installations. Both will get listed, as shown in the screenshot. However, select **JDK for Nodes**:



10. Scroll down to the **Build** section and click on the **Add build step** button. Select the **Invoke Maven 3** option:

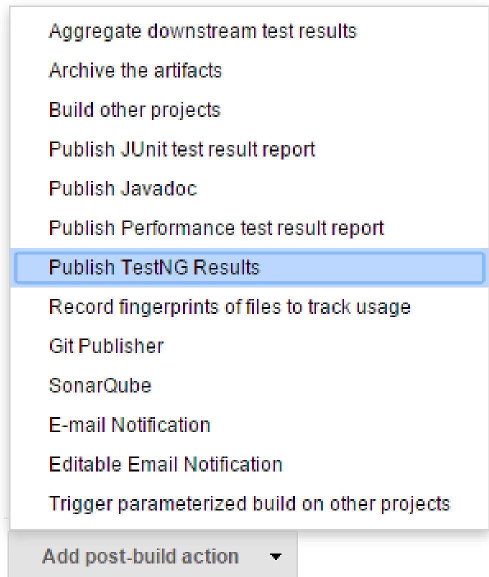


11. Add the values as shown in the following screenshot:

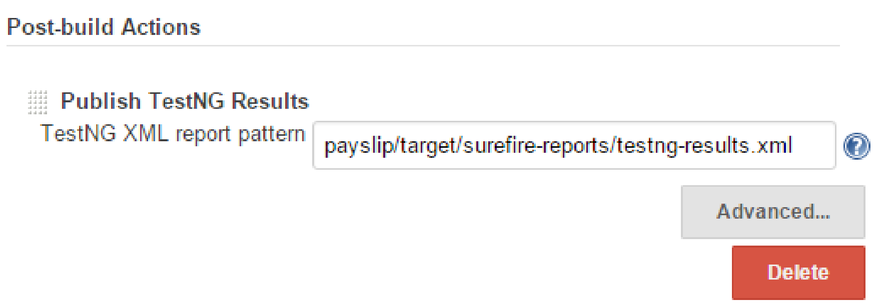


12. Let's see the Maven command inside the **Goals and options** field in detail:
    - `clean` will clean any old built files
    - The `-Puat` option in the Maven command will invoke the project named `uat` inside the `pom.xml` file

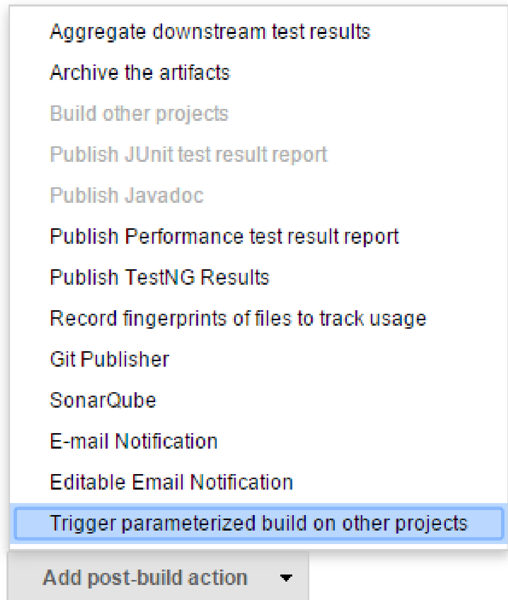
13. Scroll down to the **Post-build Actions** section and click on the **Add post-build action** button. Select **Publish TestNG Results** from the options:



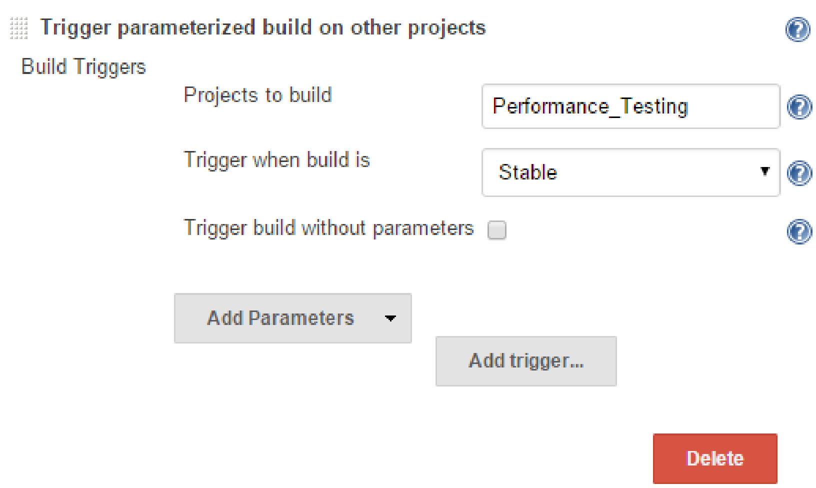
14. Add the location of the `testng-results.xml` file, as shown in the next screenshot:



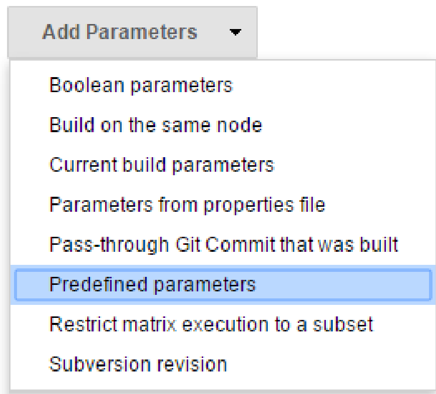
15. Click on the **Add post-build action** button again. From the drop-down list, choose the option **Trigger parameterized build on the other projects**:



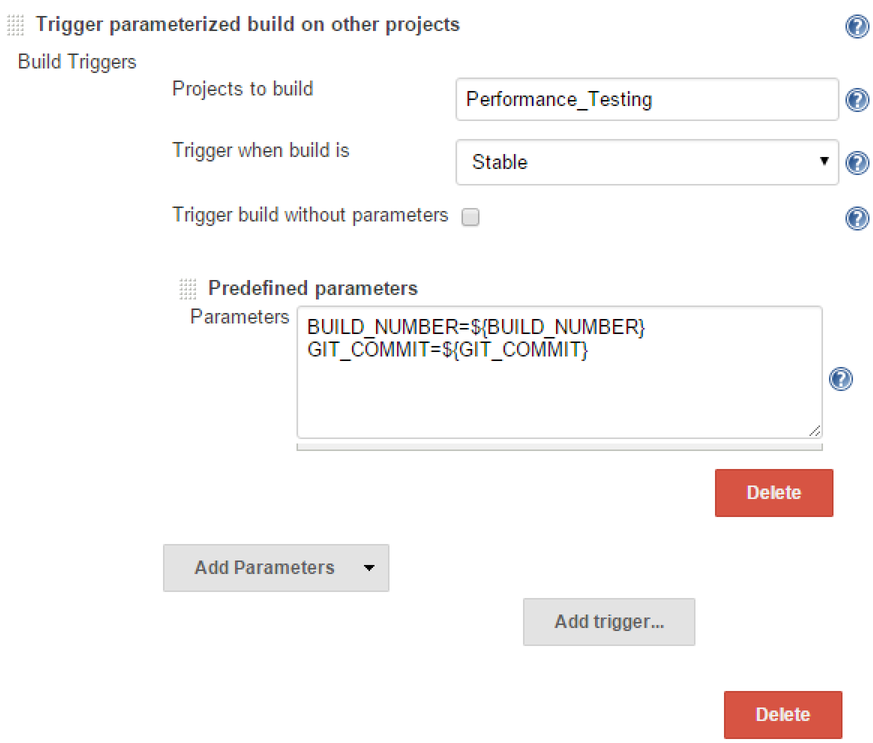
16. Add the values as shown here:



17. Along with triggering the build, we would also like to pass some predefined parameters to it. Click on the **Add Parameters** button and select **Predefined parameters**:



18. Add the following values:



19. Configure advanced e-mail notifications exactly the same way as mentioned in the previous chapters.
20. Save the Jenkins job by clicking on the **Save** button.

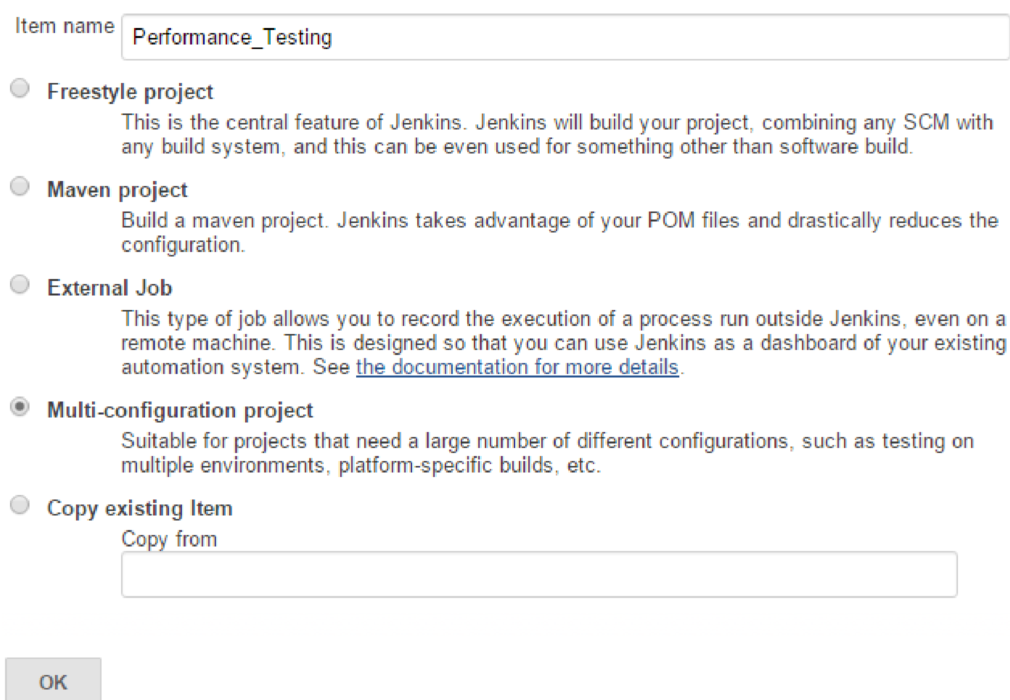
Creating a Jenkins job to run the performance test

The fifth Jenkins job in the Continuous Delivery pipeline performs the following tasks:

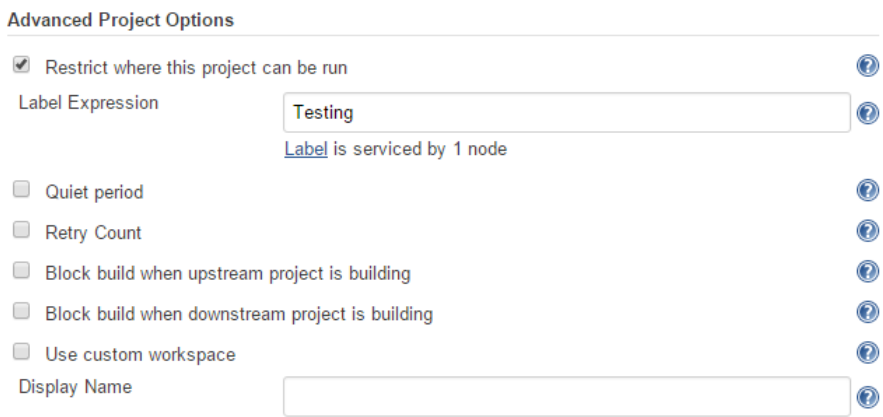
- It performs the performance test
- It generates the test results report

Follow the next few steps to create it:

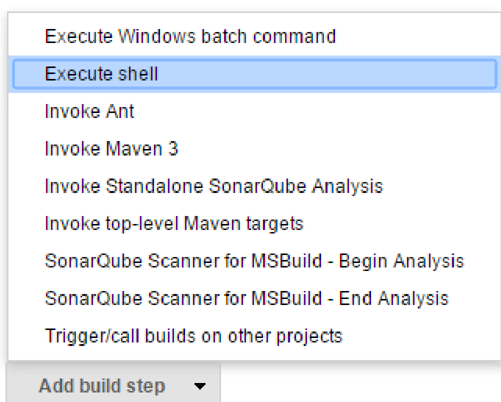
1. From the Jenkins dashboard, click on **New Item**.
2. Name your new Jenkins job `Performance_Testing`.
3. Select the type of job as **Multi-configuration project** and click on **OK** to proceed:



4. Scroll down until you see **Advanced Project Options**. Select **Restrict where this project can be run**.
5. Add `Testing` as the value for **Label Expression**:



6. Scroll down to the **Build** section.
7. Click on the **Add build step** button and choose the option **Execute shell**:

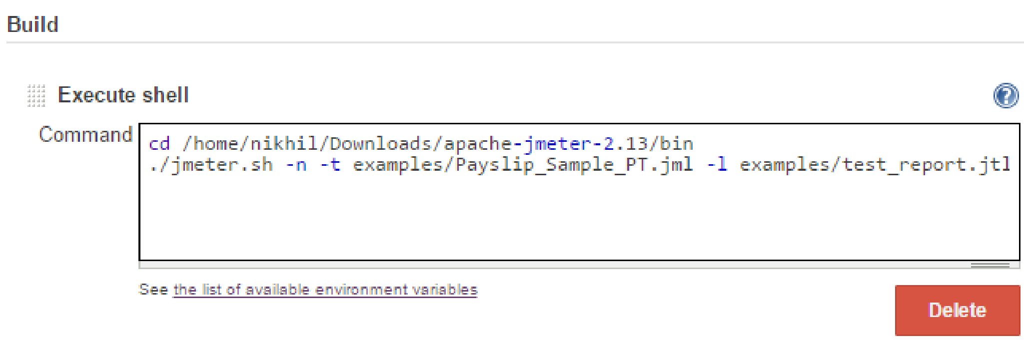


8. Add the following commands in the **Command** field:
    - The first line of command goes to the directory where `jmeter.sh` is located:

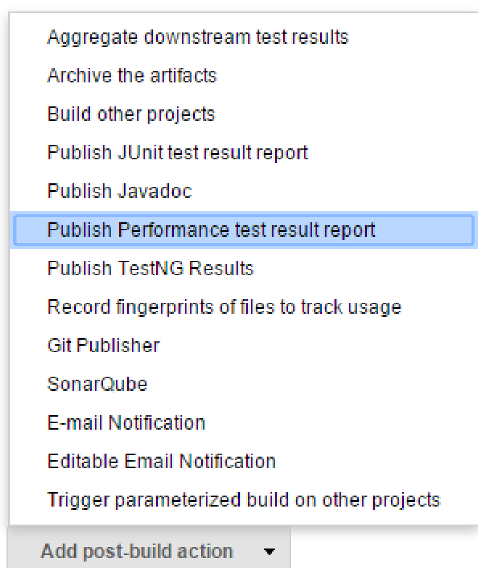
    ```
    cd /home/<user>/Downloads/apache-jmeter-2.13/bin
    ```

- The second line of code executes the performance test:

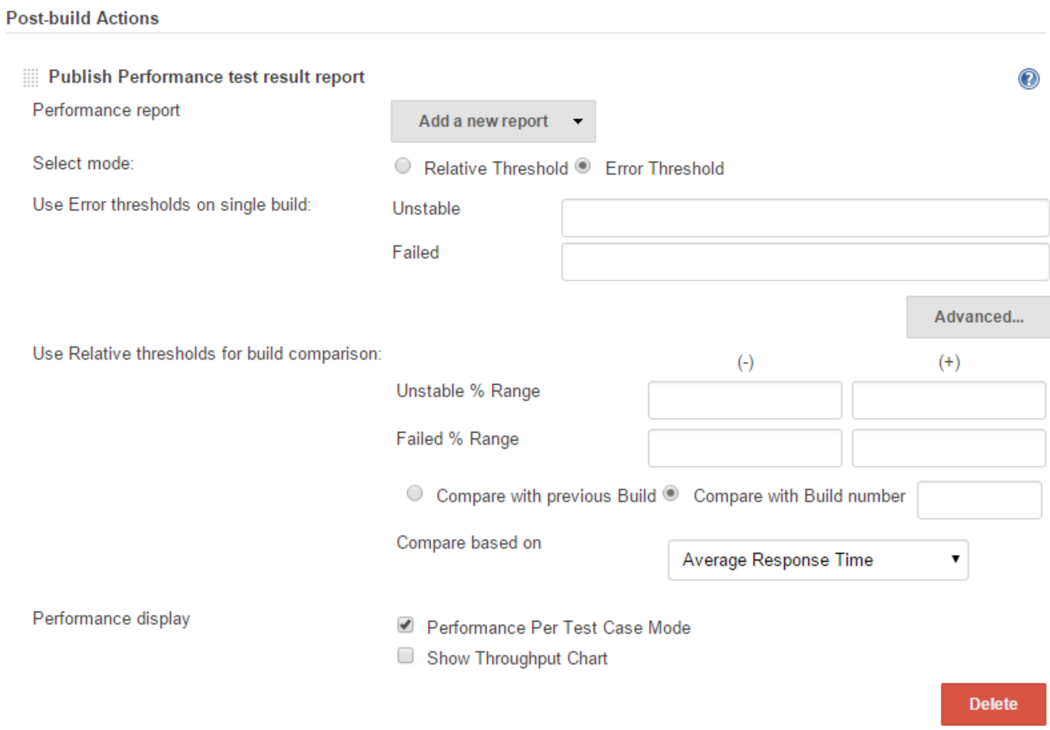
  ```
  ./jmeter.sh -n -t examples/Payslip_Sample_PT.jmx -l
  examples/test_report.jtl
  ```



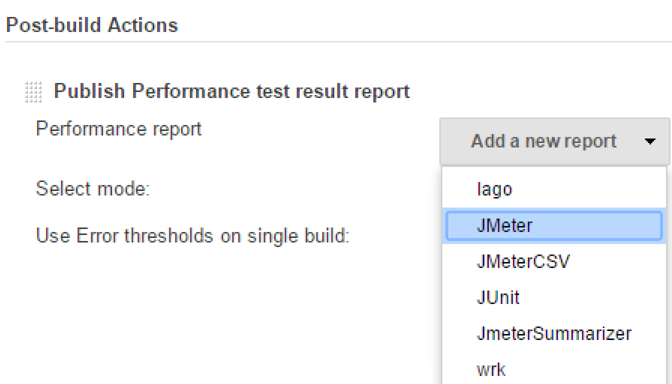
9. Scroll down to the **Post-build Actions** section. Click on the **Add post-build action** button. From the drop-down list, choose the option **Publish Performance test result report**:



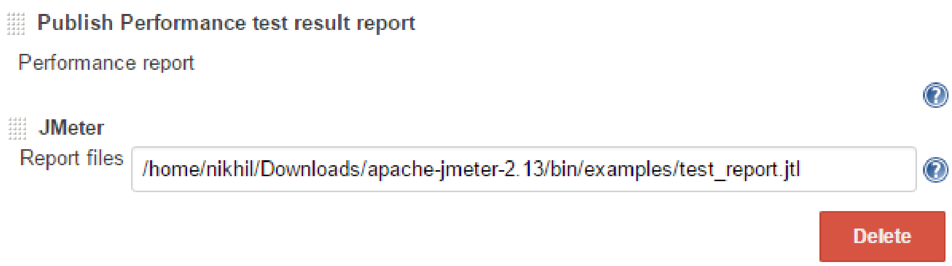
10. Choose the values shown here:



11. Click on the **Add a new report** button and select **JMeter** from the options:



12. Add the location of the test_reportr.jtl file as /home/<local user account>/Downloads/apache-jmeter-2.13/bin/examples/test_report.jtl:



> Add your respective user account on the testing server in place of <local user account>.

13. Configure advanced e-mail notifications exactly the same way as mentioned in the previous chapters.
14. Save the Jenkins job by clicking on the **Save** button.

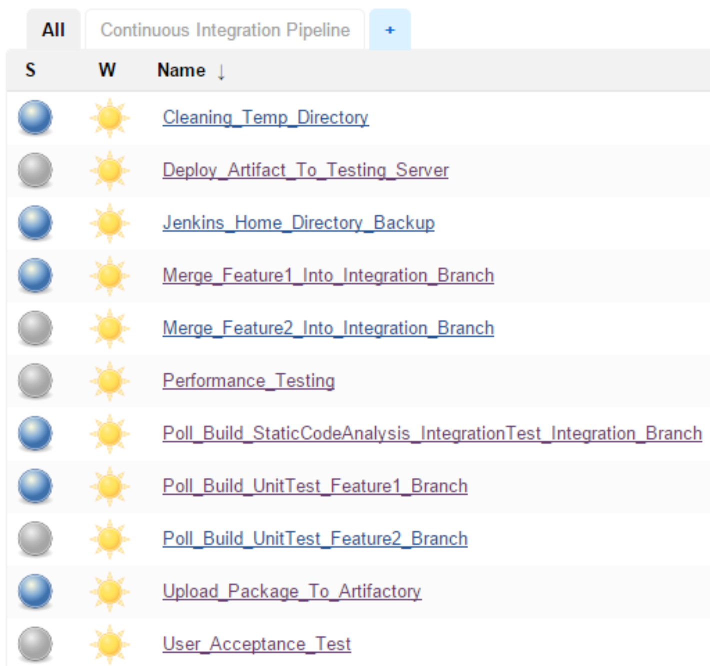
Creating a nice visual flow for the Continuous Delivery pipeline

Our pipeline to perform the Continuous Delivery now contains the following Jenkins jobs:

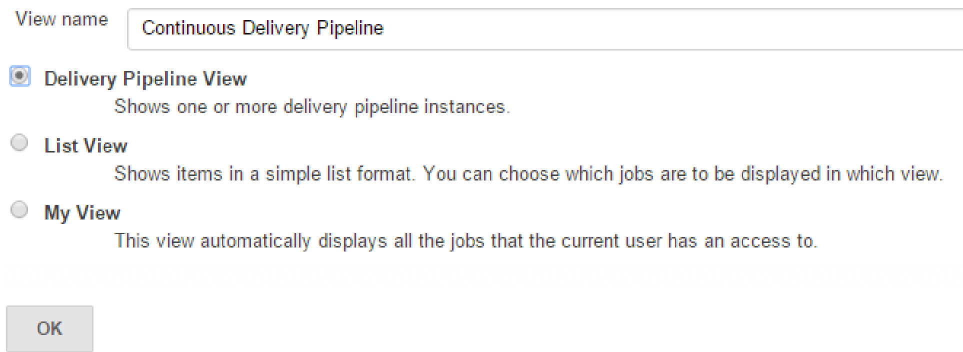
- Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch
- Upload_Package_To_Artifactory
- Deploy_Artifact_To_Testing_Server
- User_Acceptance_Test
- Performance_Testing

In this section, we will create a view inside the Jenkins dashboard using the delivery pipeline plugin. This view is nothing but a nice way of presenting the Continuous Delivery flow:

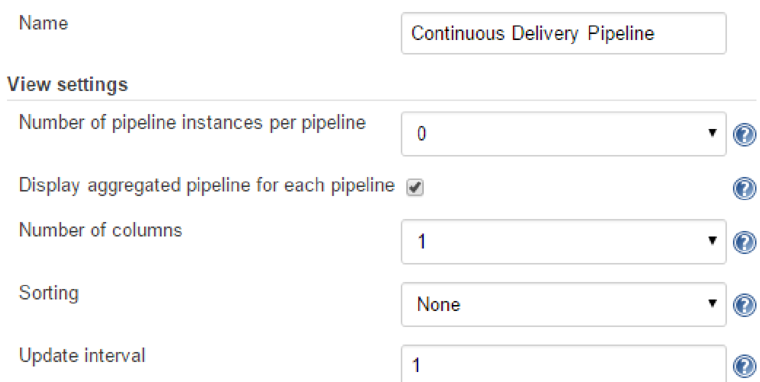
1. Go to the Jenkins dashboard and click on the **+** tab highlighted in the screenshot:



2. Provide `Continuous Delivery Pipeline` as the **View name** and select **Delivery Pipeline View** from the options, as shown in the next screenshot.
3. Click on **OK** to finish:

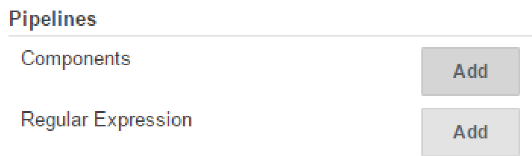


4. Now, you will see a lot of options and blanks to fill in. Scroll down until you see the **View settings** section and fill it in as follows:
    - Select the **Number of pipeline instances per pipeline** = `0`
    - **Number of columns** = `1`
    - **Update interval** = `1`
    - Also, check the **Display aggregated pipeline for each pipeline** option

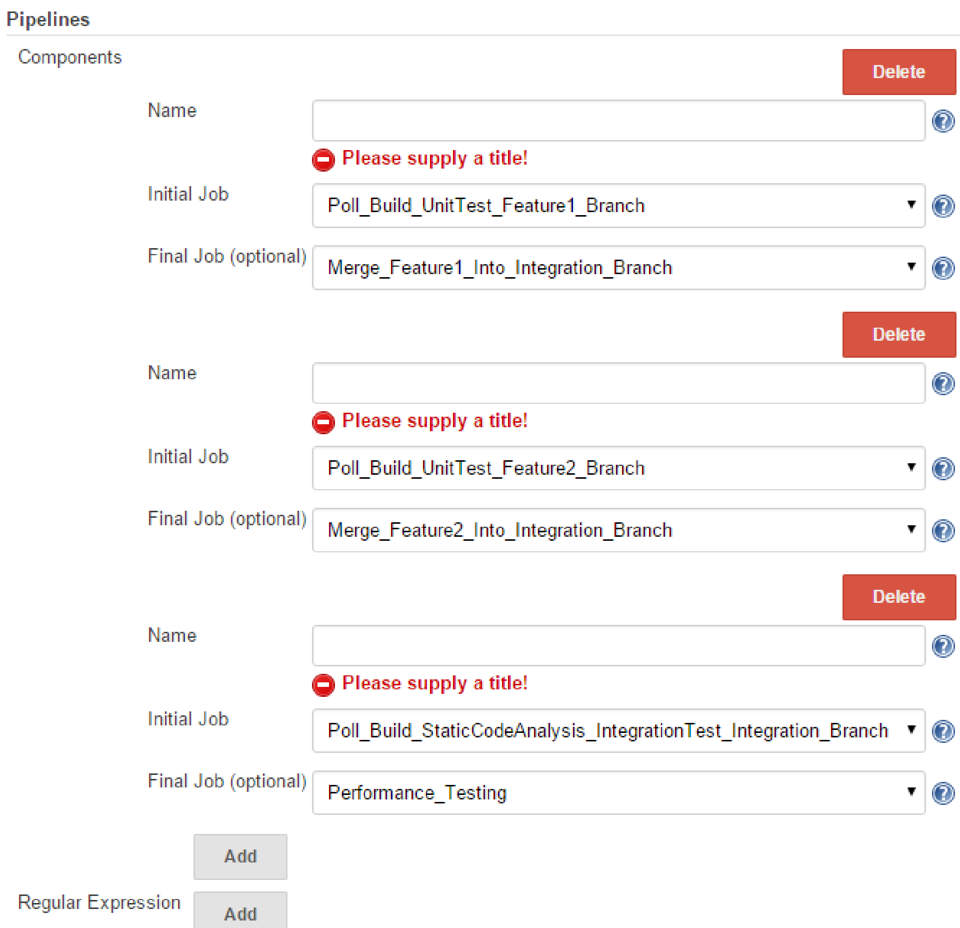


5. Leave the rest of the options at their default values and scroll down until you see the **Pipelines** section shown in the next screenshot.

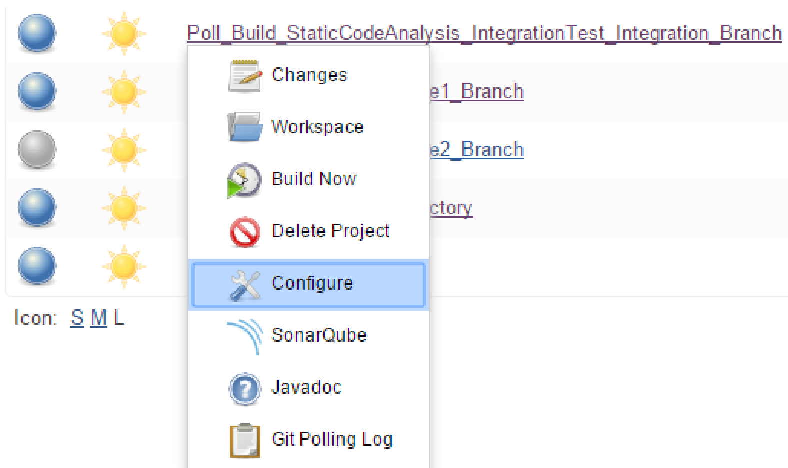
6. Click on the **Add** button beside **Components** three times:



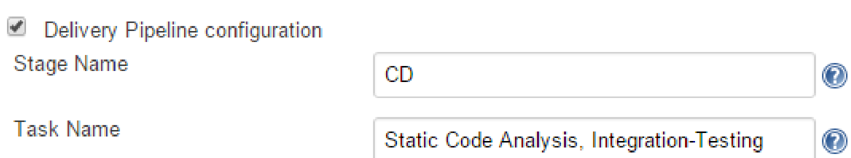
7. Fill in the values exactly as shown in this screenshot:



8. Click on **OK** to save the configuration.
9. Now, come back to the Jenkins dashboard.
10. Right-click on the **Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch** Jenkins job and select **Configure**, as shown in the following screenshot:

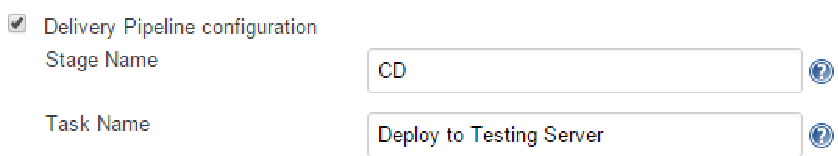


11. Look for the **Delivery Pipeline configuration** option and select it.
12. Here, set **Stage Name** as `CD` and **Task Name** as `Static Code Analysis, Integration-Testing`.
13. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.



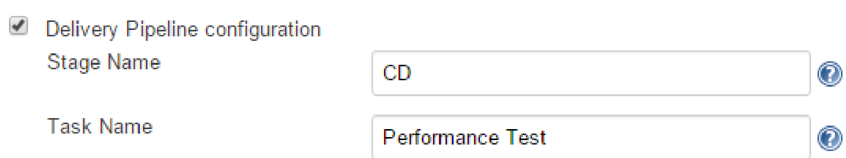
14. Now, come back to the Jenkins dashboard.
15. Right-click on the **Upload_Package_To_Artifactory** Jenkins job and select **Configure**.
16. Look for the **Delivery Pipeline configuration** option and select it.
17. Then, set **Stage Name** as `CD` and **Task Name** as `Publish to Artifactory`.
18. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.

19. Now, come back to the Jenkins dashboard.
20. Right-click on the **Deploy_Artifact_To_Testing_Server** Jenkins job and select **Configure**.
21. Look for the **Delivery Pipeline configuration** option and select it.
22. Here, set **Stage Name** as `CD` and **Task Name** as `Deploy to Testing Server`.
23. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.

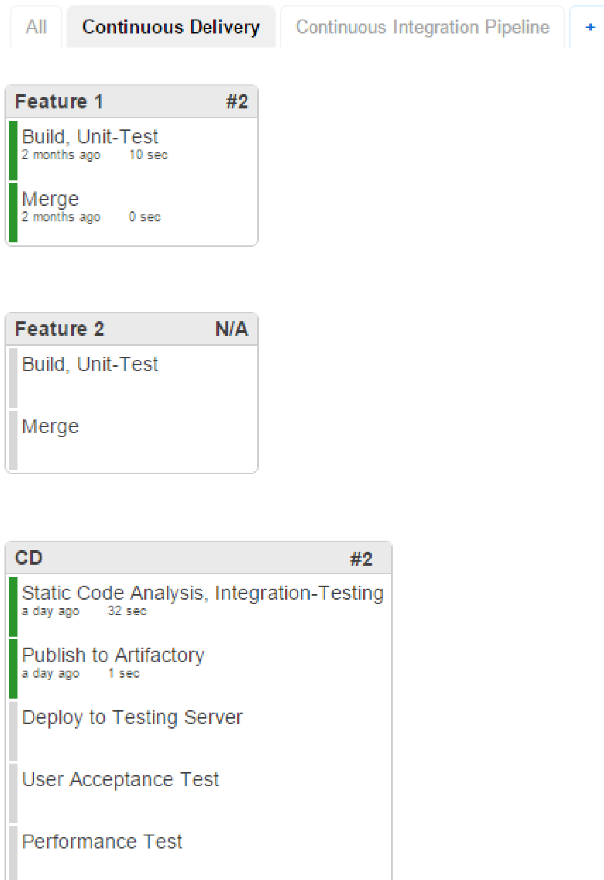


24. Now, come back to the Jenkins dashboard.
25. Right-click on the **User_Acceptance_Test** Jenkins job and select **Configure**.
26. Look for the **Delivery Pipeline configuration** option and select it.
27. Here, set **Stage Name** as `CD` and **Task Name** as `User Acceptance Test`.
28. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on:

29. Now, come back to the Jenkins dashboard.
30. Right-click on the **Performance_Testing Jenkins** job and select **Configure**.
31. Look for the **Delivery Pipeline configuration** option and select it.
32. Here, set **Stage Name** as `CD` and **Task Name** as `Performance Test`.
33. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on:



34. Come back to the Jenkins dashboard and click on the **Continuous Integration Pipeline** view. Tada!! This is what you will see:



Creating a simple user acceptance test using Selenium and TestNG

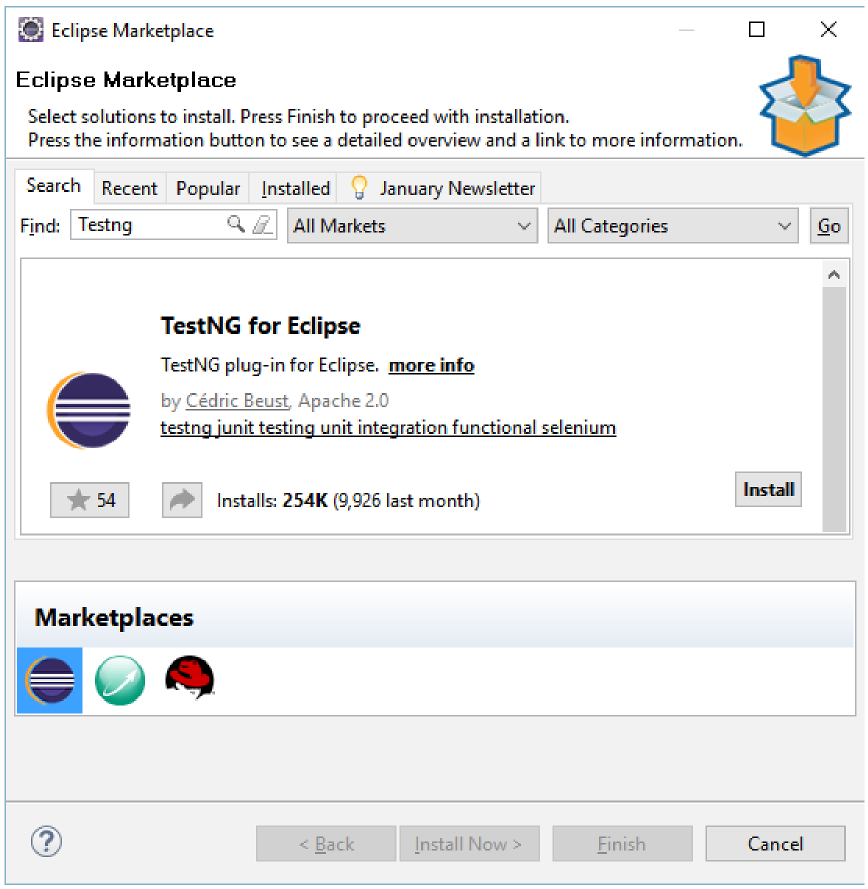
In order to perform a user acceptance test, we won't be installing any tool or software on the testing server nor anything on the Jenkins server or the developer's machine.

Tools such as Selenium and TestNG will be defined as part of the `pom.xml` file and everything will be done using Eclipse. The user acceptance test will be a part of the code, just like the unit test and the integration test.

Installing TestNG for Eclipse

To install TestNG, follow these steps:

1. From the Eclipse IDE menu, go to **Help** | **Eclipse Marketplace**.
2. In the window that opens, select the **Search** tab and look for `Testng`.
3. Once you see **TestNG for Eclipse**, install it:



Modifying the index.jsp file

Our user acceptance test is going to be a simple example in which we will try to check the payslip page title. It should be `PAY SLIP`.

Following are the steps to modify the `index.jsp` file:

1. Open `index.jsp` from the following path: `/payslip/src/main/webapp/`.
2. Add the title `PAY SLIP` to the page by modifying the title, as shown in the screenshot:

Modifying the POM file

This is where we will configure Selenium and the TestNG plugin. Along with that, we will also create two profiles inside the `pom.xml` file—one named `sit` and another named `uat`.

The `sit` profile will be executed as part of the Continuous Integration. The `uat` profile will be executed while performing user acceptance testing. The steps are as follows:

1. Open the `pom.xml` file from the following path: `/payslip/`.
2. Replace the content of the file with the following code:

```
<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/maven-v4_0_0.xsd>
  <modelVersion>4.0.0</modelVersion>
  <groupId>employee</groupId>
  <artifactId>payslip</artifactId>
  <packaging>war</packaging>
  <version>0.0.1</version>
  <name>payslip Maven Webapp</name>
  <url>http://maven.apache.org</url>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
    <dependency>
            <groupId>org.seleniumhq.selenium</groupId>
            <artifactId>selenium-java</artifactId>
            <version>2.51.0</version>
    </dependency>
    <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <version>6.8</version>
            <scope>test</scope>
       </dependency>
  </dependencies>
  <profiles>
      <profile>
      <id>sit</id>
```

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.19</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-failsafe-plugin</artifactId>
        <version>2.19</version>
        <executions>
          <execution>
            <goals>
              <goal>integration-test</goal>
              <goal>verify</goal>
            </goals>
            <configuration>
            <includes>
            <include>**/IT*.java</include>
            <include>**/*IT.java</include>
          </includes>
          </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </pluginManagement>
 <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.7</source>
        <target>1.7</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
    </plugin>
```

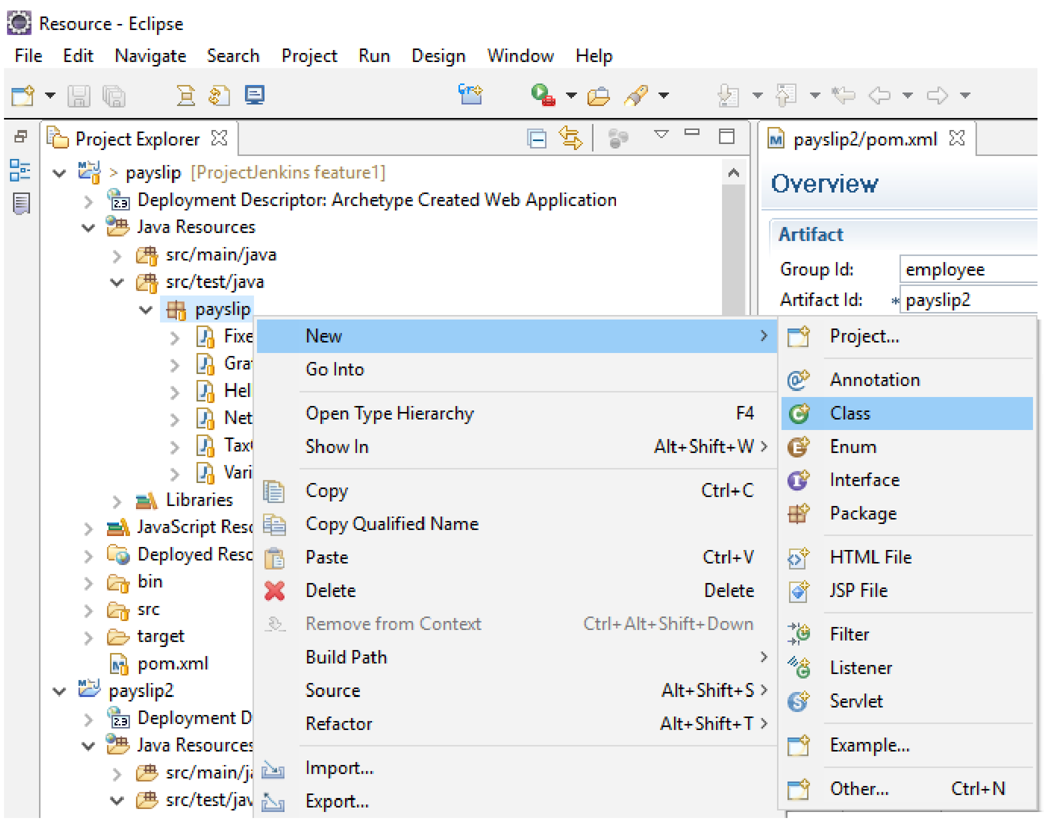
```
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skip>${surefire.skip}</skip>
      </configuration>
    </plugin>
  </plugins>
</build>
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-report-
      plugin</artifactId>
      <version>2.19</version>
    </plugin>
  </plugins>
</reporting>
</profile>
    <profile>
<id>uat</id>
<build>
<plugins>
     <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-compiler-plugin</artifactId>
           <version>3.3</version>
           <configuration>
           <source>1.7</source>
           <target>1.7</target>
           </configuration>
     </plugin>
     <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-surefire-plugin</artifactId>
           <version>2.19</version>
           <inherited>true</inherited>
           <configuration>
           <suiteXmlFiles>
           <suiteXmlFile>testng.xml</suiteXmlFile>
           </suiteXmlFiles>
```

```
            </configuration>
        </plugin>
        </plugins>
        </build>
        </profile>
    </profiles>
</project>
```

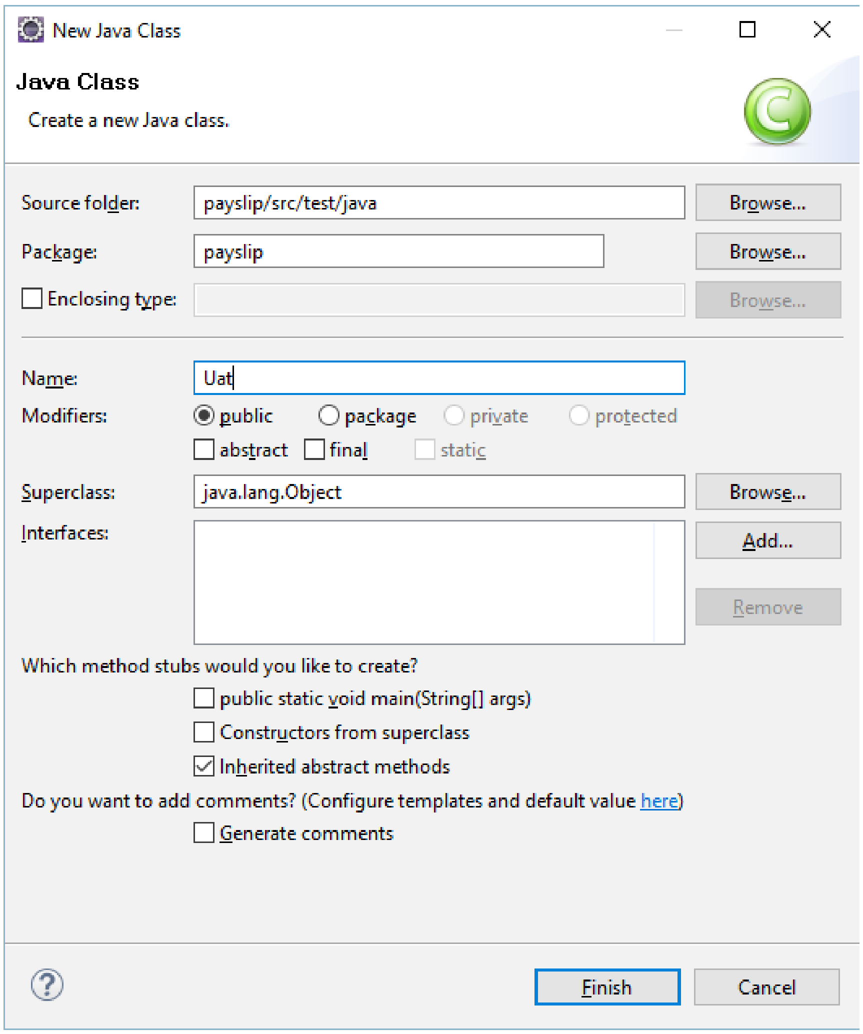
Creating a user acceptance test case

Perform the following steps to create a user acceptance test case:

1. Right-click on the `payslip` package and go to **New** | **Class**, as shown in the following screenshot:



2. In the window that opens, name the class file `Uat` and leave the rest of the options at their default values.
3. Click on the **Finish** button:



4. The file will open for editing. Replace the content of the file with the following code:

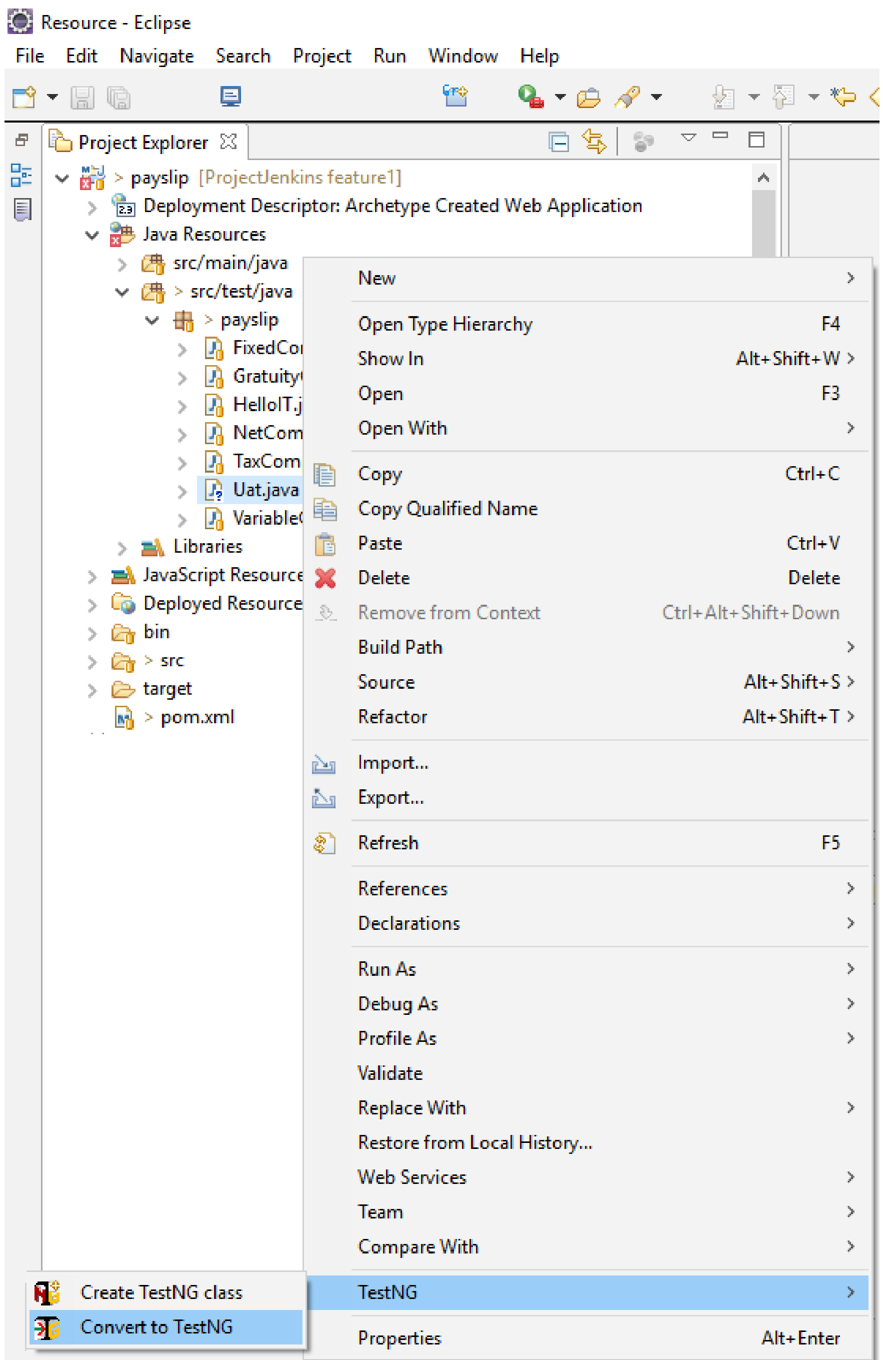
```
package payslip;

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 Uat {
        private WebDriver driver;
        @Test
        public void testEasy() {
            driver.get(http://<ip address>:8080/payslip-
            0.0.1/);
            String title = driver.getTitle();
            Assert.assertTrue(title.contains(PAY SLIP));
        }
        @BeforeTest
        public void beforeTest() {
            driver = new FirefoxDriver();
        }
        @AfterTest
        public void afterTest() {
            driver.quit();
        }
}
```

5. Replace `<ip address>` in the preceding code with the IP address of the testing server.
6. Save the file.

Generating the testng.xml file

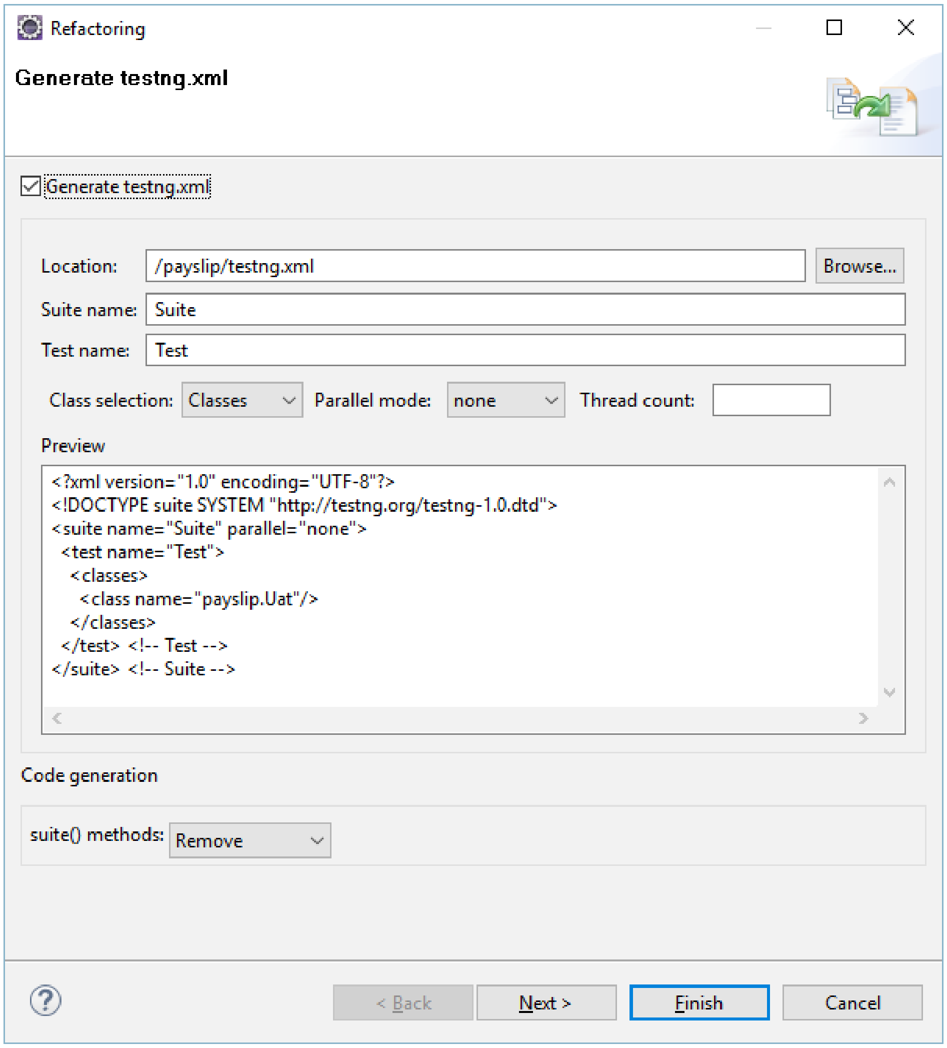
Perform the following steps to generate the `testng.xml` file:

1. Right-click on the newly created `Uat.java` file and navigate to **TestNG | Convert to TestNG**:



Resource - Eclipse
File Edit Navigate Search Project Run Window Help
Project Explorer
payslip [ProjectJenkins feature1]
Deployment Descriptor: Archetype Created Web Application
Java Resources
src/main/java
src/test/java
payslip
Uat.java
Libraries
bin
src
target
pom.xml
New
Open Type Hierarchy F4
Show In Alt+Shift+W
Open F3
Open With
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
References
Declarations
Run As
Debug As
Profile As
Validate
Replace With
Restore from Local History...
Web Services
Team
Compare With
TestNG
Properties Alt+Enter
Create TestNG class
Convert to TestNG

2. In the window that opens, you will see some code listed in the preview field:



3. Replace it with the following code:

```
<?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=payslip.Uat/>
    </classes>
```

```
    </test> <!-- Test -->
  </suite> <!-- Suite -->
```

4. Save the changes to the file.

Continuous Delivery in action

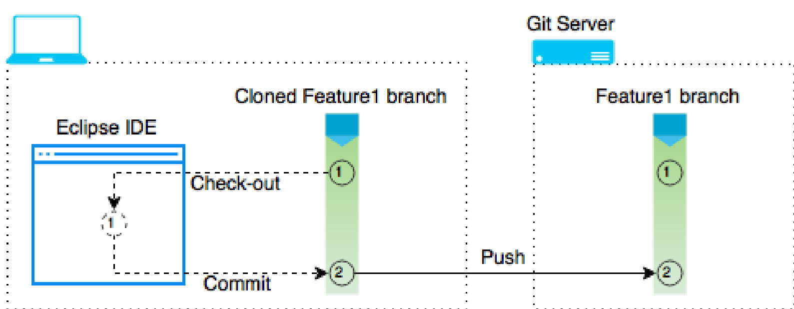
Now, we are ready to test our Continuous Delivery pipeline. Let's assume the role of a developer who intends to work on the `feature1` branch. Our developer is working on a Windows 10 machine with the following software installed on it:

- The latest version of Eclipse (Eclipse Mars)
- Apache Tomcat server 8
- Git 2.6.3
- SourceTree
- Java JDK 1.8.0_60
- Maven 3.3.9

We won't modify any code in order to trigger our Continuous Delivery pipeline, as in the previous section we made considerable changes to our code base. I guess checking in those changes would be more than sufficient. Nevertheless, you are free to make changes to your code.

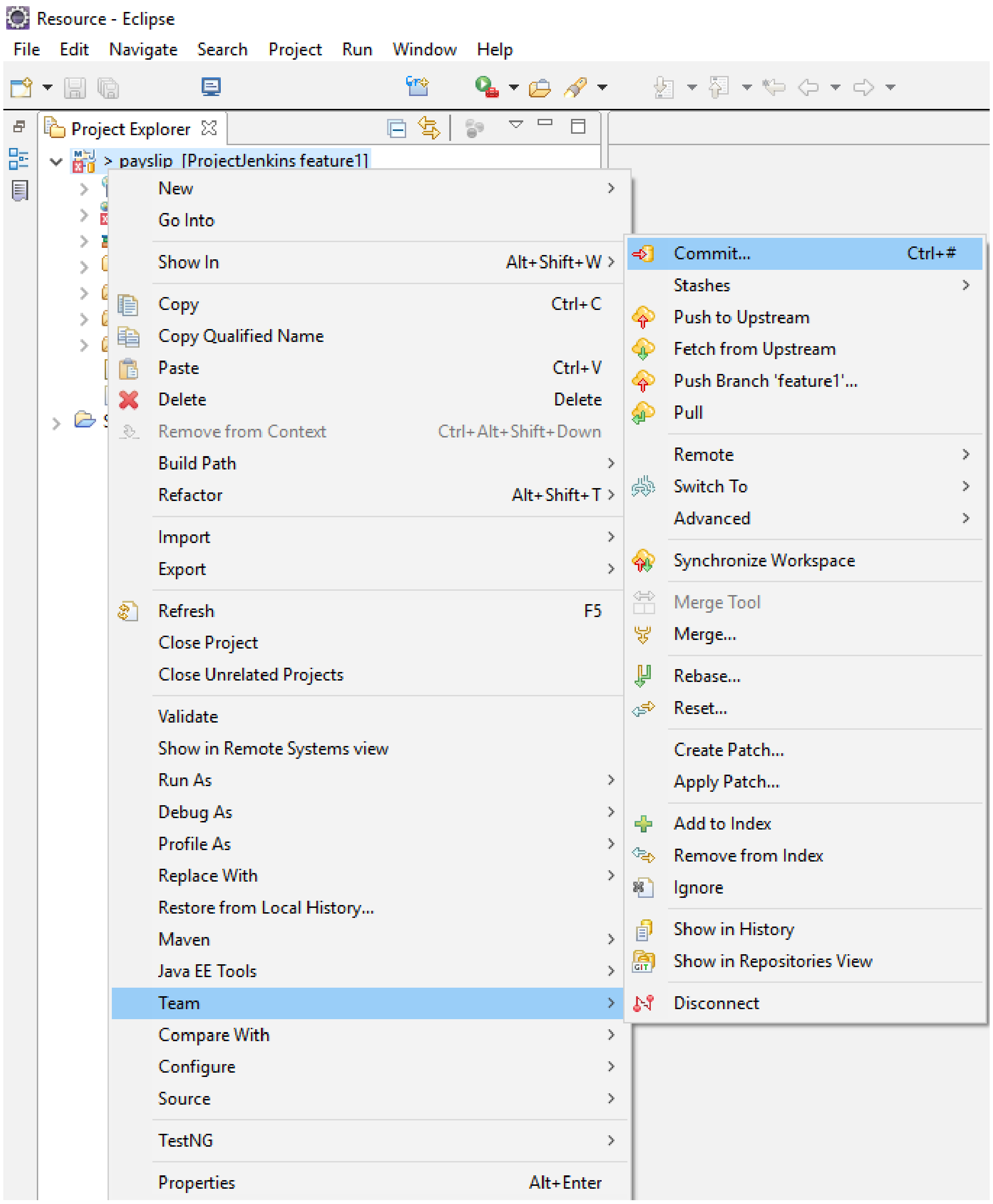
Committing and pushing changes on the feature1 branch

The following figure gives an overview of the operation that we will perform:

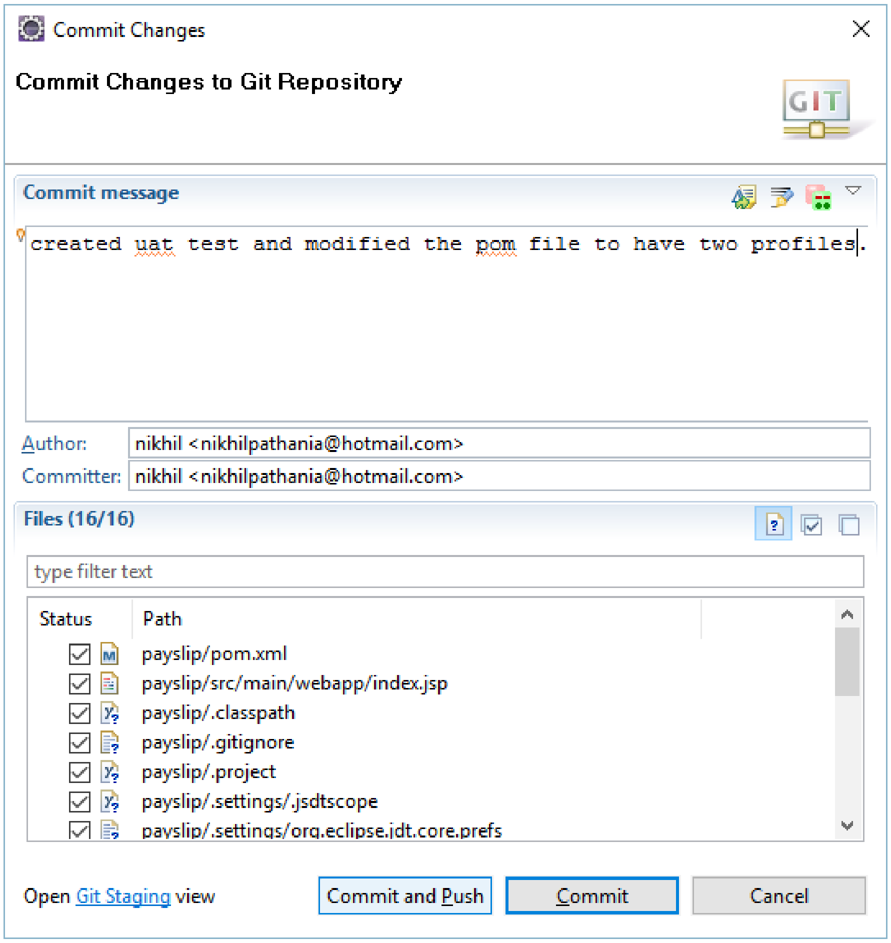


Perform the following instructions to commit and push changes:

1. Open Eclipse IDE.
2. Right-click on the project **payslip** and go to **Team** | **Commit...**:

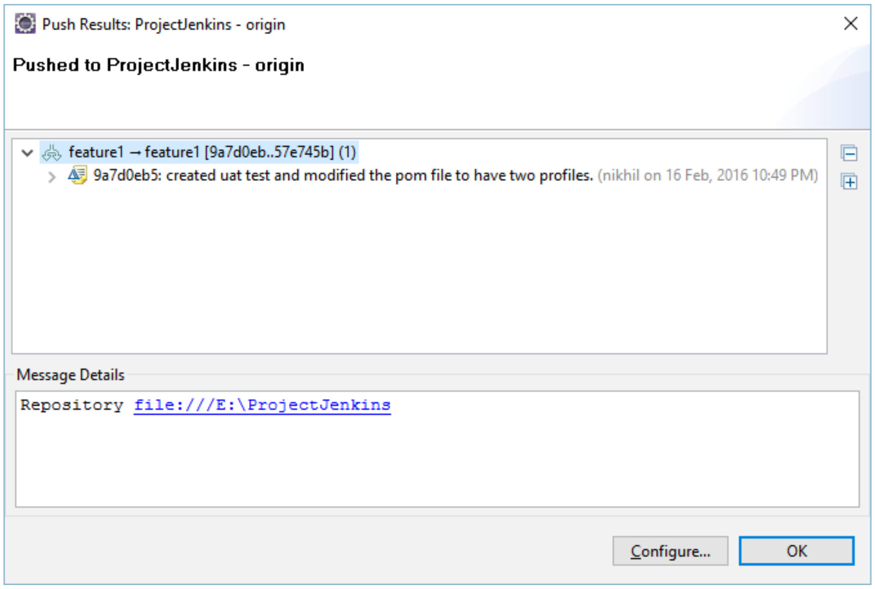


3. In the window that opens, add some comments (as shown in the screenshot) and select the modified code files that you wish to commit:



4. Once done, click on the **Commit and Push** button.
5. You can see the code has been committed on the cloned feature1 branch, and it's also pushed to the remote feature1 branch.

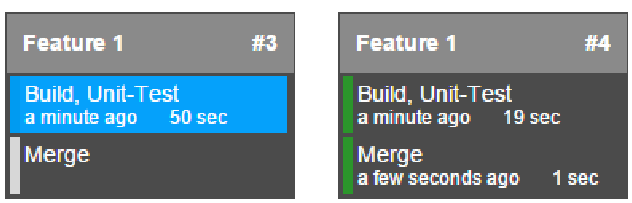
6. Click on the **OK** button to confirm the commit and push operations:



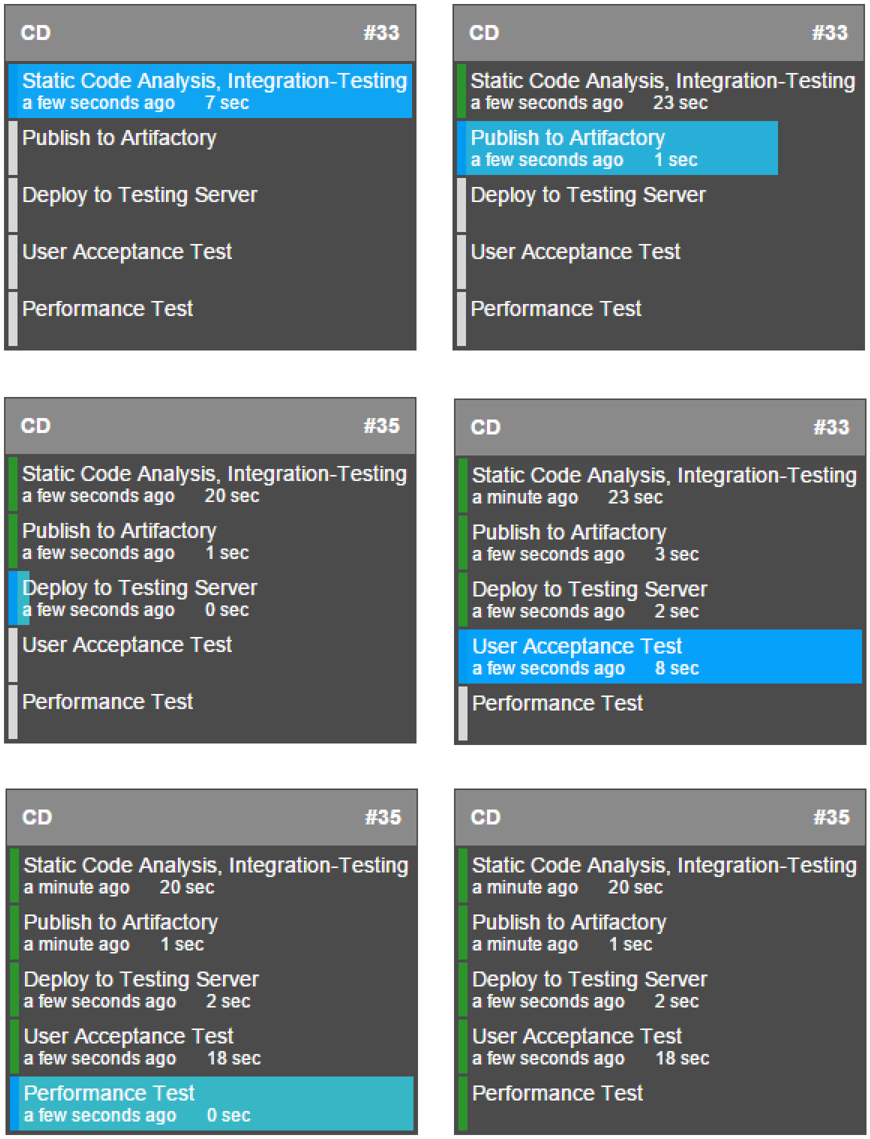
Jenkins Continuous Delivery pipeline in action

Now, there have been some changes to the feature1 branch. Let's see if Jenkins has detected it. Follow these steps:

1. Go to the Jenkins dashboard and click on the **Continuous Delivery Pipeline** view.
2. In the menu on the left-hand side, click on the **View Fullscreen** link.
3. You will see the following Jenkins jobs in the **CD Pipeline**:



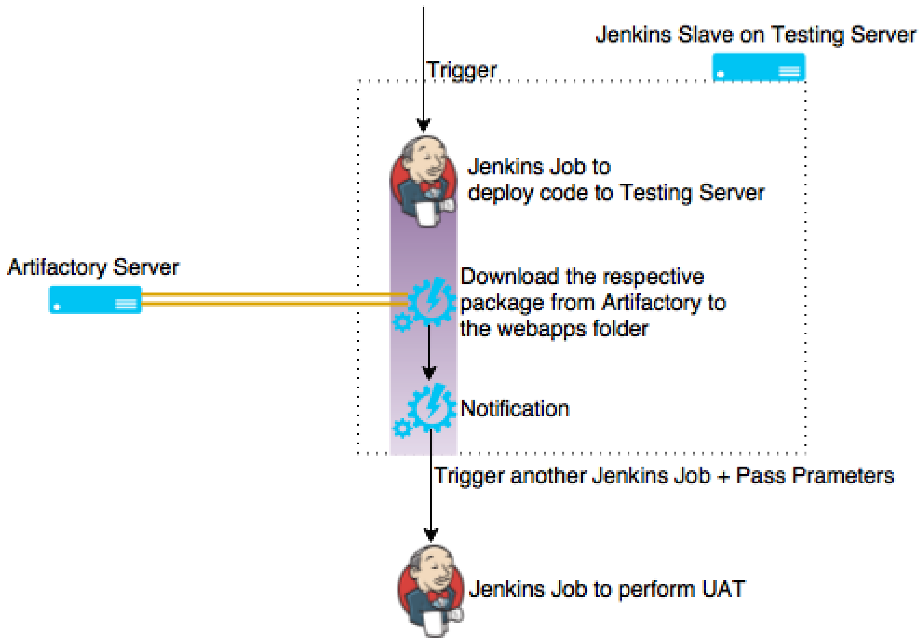
The proceeding image shows the CD pipeline in progress.



[The proceeding image shows the CD pipeline in progress.]

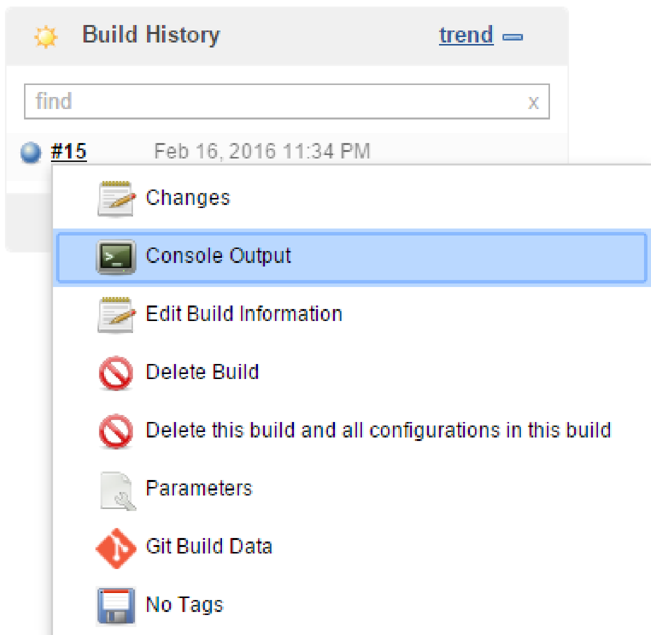
Exploring the job to perform deployment in the testing server

The following figure gives an overview of the tasks that happen while this particular Jenkins job runs:



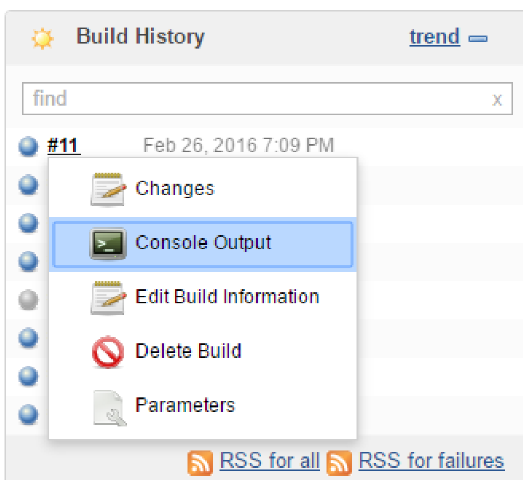
The Continuous Delivery pipeline has worked well. Let's go through the Jenkins job that performs deployment on the testing server, using the following steps:

1. From the Jenkins dashboard, click on the **Deploy_Artifact_To_Testing_Server** job.
2. On the **Build History** panel, right-click on any of the builds:



3. You will see the following build log. This is the log from the Jenkins master server's perspective:

4. Click on the **Deploy_Artifact_To_Testing_Server** » **default** link.
5. From the **Build History** panel, right-click on any of the builds:



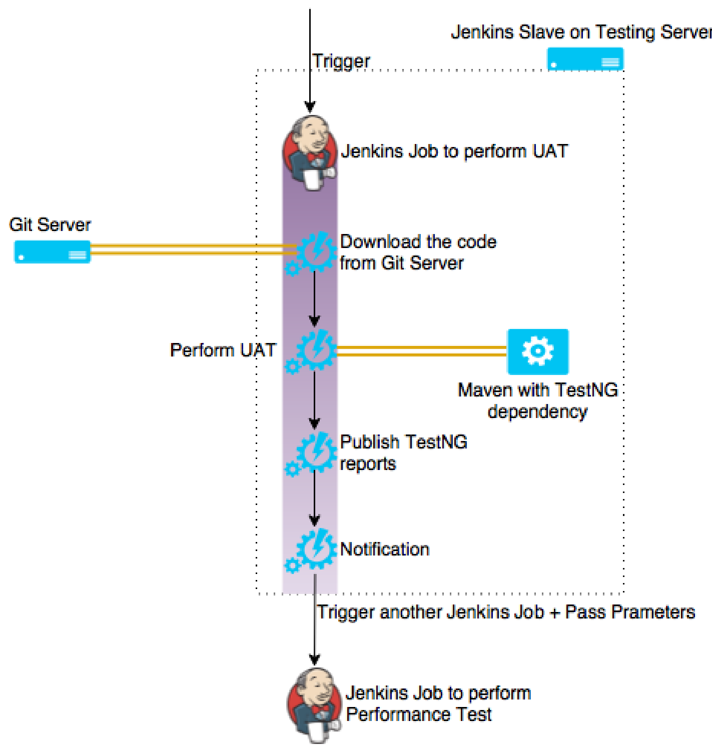
6. You will see the following build log. This is the log from the Jenkins slave's perspective:

Console Output

```
Started by upstream project "Deploy_Artifact_To_Testing_Server" build number 11
originally caused by:
 Started by upstream project "Upload_Package_To_Artifactory" build number 37
 originally caused by:
  Started by upstream project
"Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch" build number
40
  originally caused by:
   Started by user Administrator
Building remotely on Production_Server (production)No JDK named 'null' found
 in workspace /home/nikhil/workspace/Deploy_Artifact_To_Testing_Server/default
No JDK named 'null' found
[default] $ /bin/sh -xe /tmp/hudson6298033921962754780.sh
+ wget http://192.168.1.104:8081/artifactory/projectjenkins/37/payslip-0.0.1.war
--2016-02-26 19:09:34--
http://192.168.1.104:8081/artifactory/projectjenkins/37/payslip-0.0.1.war
Connecting to 192.168.1.104:8081... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17545016 (17M) [application/java-archive]
Saving to: 'payslip-0.0.1.war'

2016-02-26 19:09:34 (41.3 MB/s) - 'payslip-0.0.1.war' saved [17545016/17545016]

+ mv payslip-0.0.1.war /opt/tomcat/webapps/payslip-0.0.1.war -f
No JDK named 'null' found
No JDK named 'null' found
Finished: SUCCESS
```

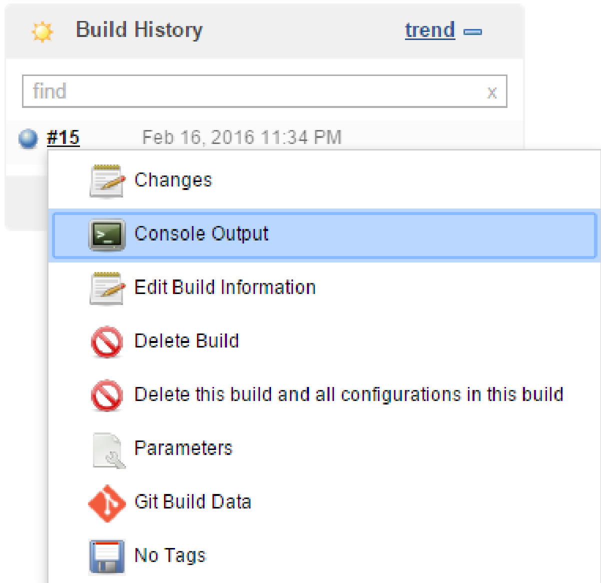
Exploring the job to perform a user acceptance test

The following figure gives an overview of the tasks that happen while this particular Jenkins job runs:

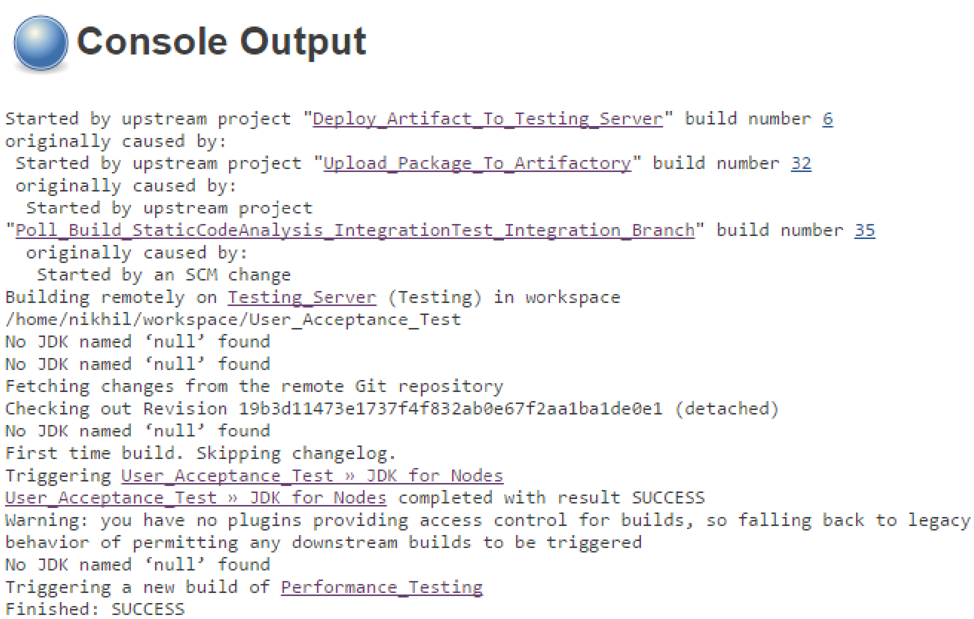


Let's go through the user acceptance test results. To do so, follow these steps:

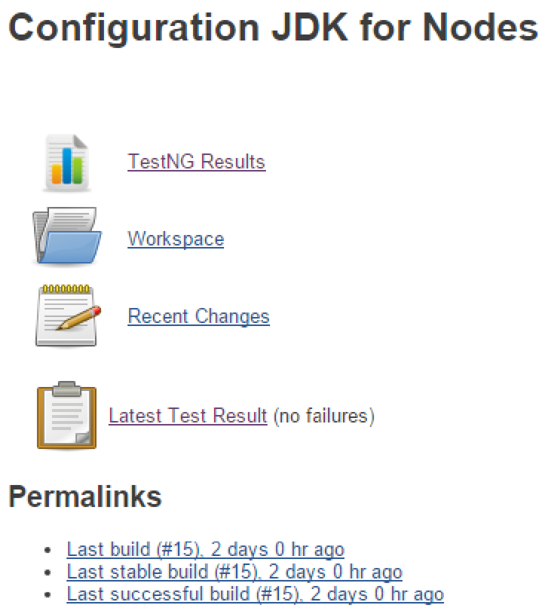
1. From the Jenkins dashboard, click on the `User_Acceptance_Test` job.
2. From the **Build History** panel, right-click on any of the builds:



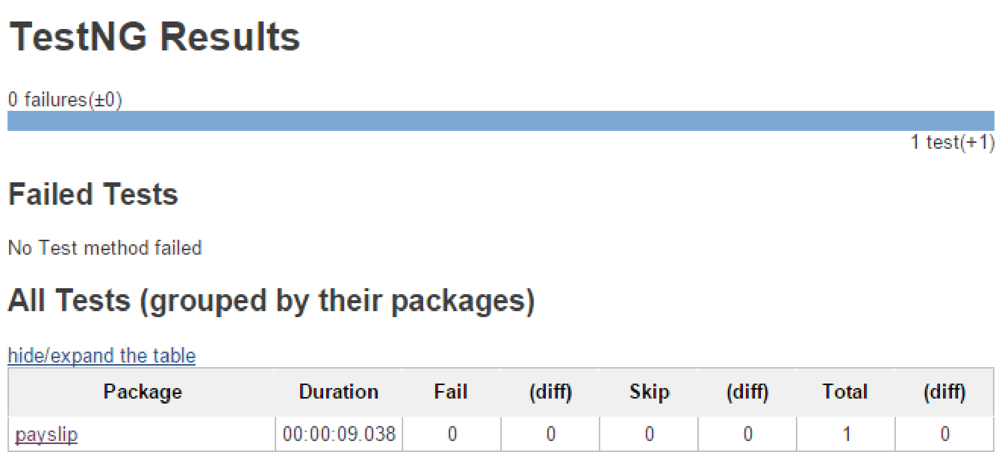
3. You will see the following build log. This is the log from the Jenkins master server's perspective:



4. Click on the **User_Acceptance_Test** » **JDK for Nodes** link.
5. You will see the following on the resulting page:

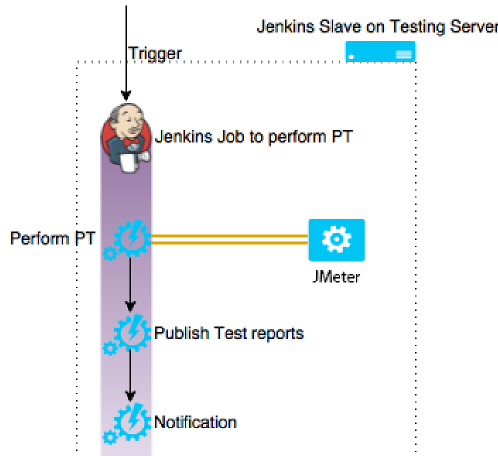


6. Click on the **Latest Test Result** link:



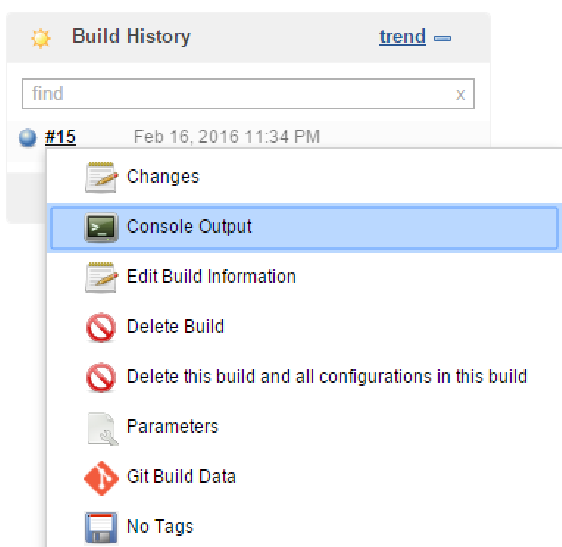
Exploring the job for performance testing

The following figure gives an overview of the tasks that will happen while this particular Jenkins job runs:

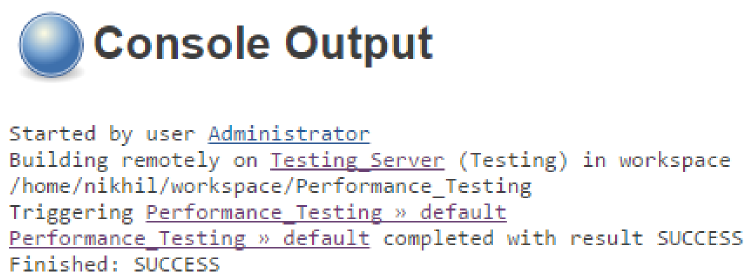


Let's go through the performance test results. To do so, follow the next few steps:

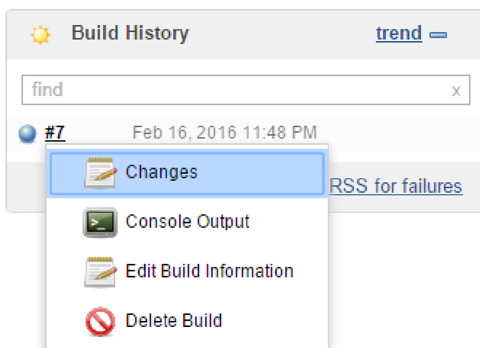
1. From the Jenkins dashboard, click on the **Performance_Test** job.
2. From the **Build History** panel, right-click on any of the builds:



3. You will see the build log as shown here. This is the log from the Jenkins master server's perspective:



4. Click on the **Performance_Testing** » **default** link.
5. On the landing page, from the **Build History** panel, right-click on any of the builds. This is shown in the following screenshot:

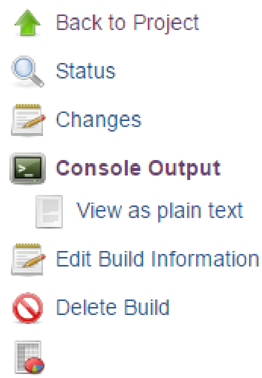


6. You will see the following build log. This is the log from the Jenkins slave's perspective:

Console Output

```
Started by upstream project "Performance_Testing" build number 7
originally caused by:
 Started by user Administrator
Building remotely on Testing_Server (Testing)No JDK named ‘null’ found
 in workspace /home/nikhil/workspace/Performance_Testing/default
No JDK named ‘null’ found
[default] $ /bin/sh -xe /tmp/hudson6461187722122104071.sh
+ cd /home/nikhil/Downloads/apache-jmeter-2.13/bin
+ ./jmeter.sh -n -t examples/Payslip_Sample_PT.jmx -l examples/test_report.jtl
Creating summariser <summary>
Created the tree successfully using examples/Payslip_Sample_PT.jmx
Starting the test @ Tue Feb 16 23:48:49 IST 2016 (1455646729464)
Waiting for possible shutdown message on port 4445
summary =      1 in     1s =    1.6/s Avg:    88 Min:    88 Max:    88 Err:     0 (0.00%)
Tidying up ...    @ Tue Feb 16 23:48:50 IST 2016 (1455646730190)
... end of run
No JDK named ‘null’ found
Performance: Percentage of errors greater or equal than 100% sets the build as unstable
Performance: Percentage of errors greater or equal than 100% sets the build as failure



Performance: Recording JMeter reports '/home/nikhil/Downloads/apache-jmeter-
2.13/bin/examples/test_report.jtl'
Performance: Parsing JMeter report file
'C:\Jenkins\jobs\Performance_Testing\configurations\builds\7\performance-
reports\JMeter\test_report.jtl'.
test_report.jtl has an average of: 100
Performance: File test_report.jtl reported 100.0% of errors [SUCCESS]. Build status is:
SUCCESS



Finished: SUCCESS
```

7. On the same page, click on the **Performance Trend** link, the one with the pie chart and spreadsheet in its logo:



8. You will see the following data which is from the `test_report.jtl` file:

URI	Samples	Samples diff	Average (ms)	Average diff (ms)	Median (ms)	Median diff (ms)	Line90 (ms)
	4	0	100	0	96	0	130
All URIs	4	0	100	0	96	0	130

Minimum (ms)	Maximum (ms)	Http Code	Previous Http Code	Errors (%)	Errors diff (%)	Average (KB)	Total (KB)
88	130	200		100.0 %	0.0 %		
88	130			100.0 %	0.0 %	0.0	0.0

Summary

In this chapter, we saw how to implement Continuous Delivery using Jenkins along with testing tools such as JMeter, TestNG, and Selenium. We also saw how to create parameterized Jenkins jobs and configure Jenkins slave agents.

The parameter plugin that comes by default in Jenkins helped our Jenkins jobs pass important information among themselves, such as the version of code to build and version of artifact to deploy.

To keep things simple, we chose to perform all the testing on a single testing server, where we also configured our Jenkins slave agent. However, this is not something that you will see in most organizations. There can be many Jenkins nodes running on many testing servers, with each testing server dedicated to performing a specific test.

Feel free to experiment yourself by configuring a separate machine for user acceptance testing and performance testing. Install the Jenkins node agent on both the machines and modify your Jenkins jobs that perform user acceptance tests and performance tests to run on their respective testing servers.

7

Continuous Deployment Using Jenkins

This chapter will cover Continuous Deployment and explain the difference between Continuous Deployment and Continuous Delivery. We will discuss a simple Continuous Deployment Design and the means to achieve it.

These are the important topics that we will discuss in this chapter:

- The difference between Continuous Deployment and Continuous Delivery
- Who needs Continuous Deployment?
- Continuous Deployment Design

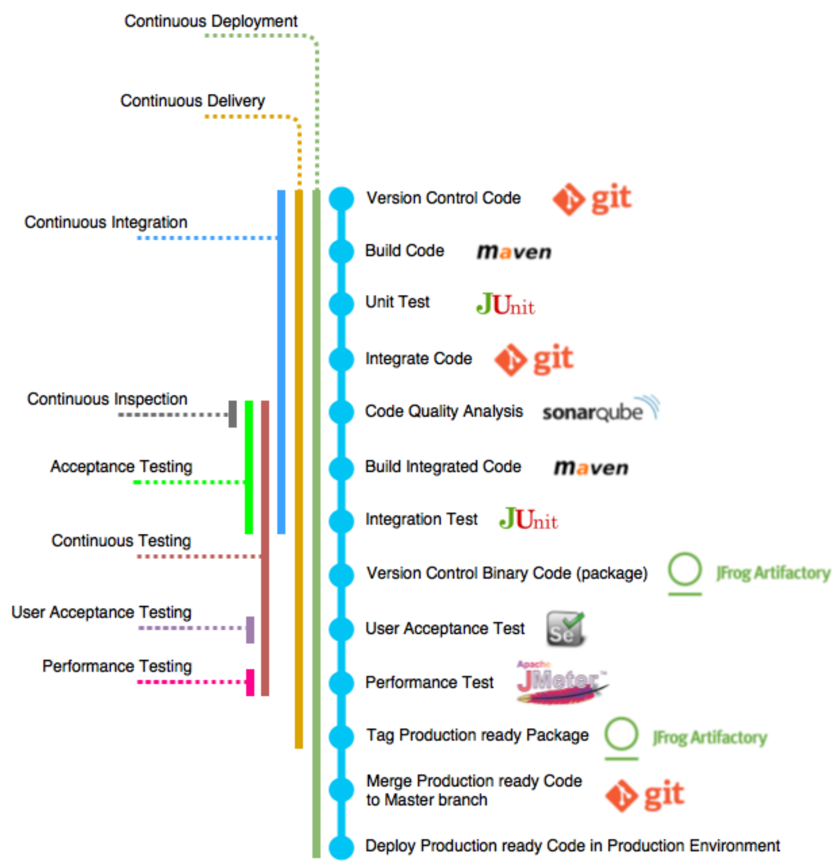
Continuous Deployment is a simple tweaked version of the Continuous Delivery pipeline. Hence, we won't be seeing any major Jenkins configuration changes or any new tools.

What is Continuous Deployment?

The process of continuously deploying production-ready features into the production environment, or to the end-user, is termed **Continuous Deployment**. Continuous Deployment in the holistic sense refers to the process of making production-ready features go live instantly without any intervention. This includes building features in an agile manner, integrating and testing them continuously, and deploying them into the production environment without any break. This is what we are trying to achieve in this chapter.

On the other hand, Continuous Deployment in a literal sense means deploying any given package continuously in any given environment. Therefore, the task of deploying packages into testing server and production server conveys the literal meaning of Continues Deployment.

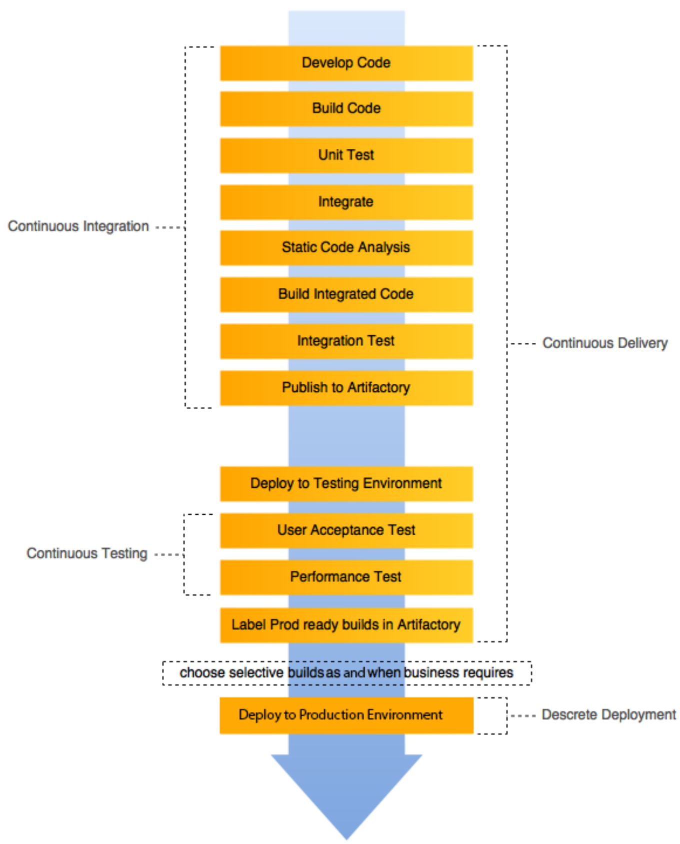
The following figure will help us understand the various terminologies that we discussed just now. We also saw this in the previous chapter. The various steps a software code goes through, from its inception to its utilization (development to production) are listed here. Each step has a tool associated with it, and each one is part of one or another methodology.



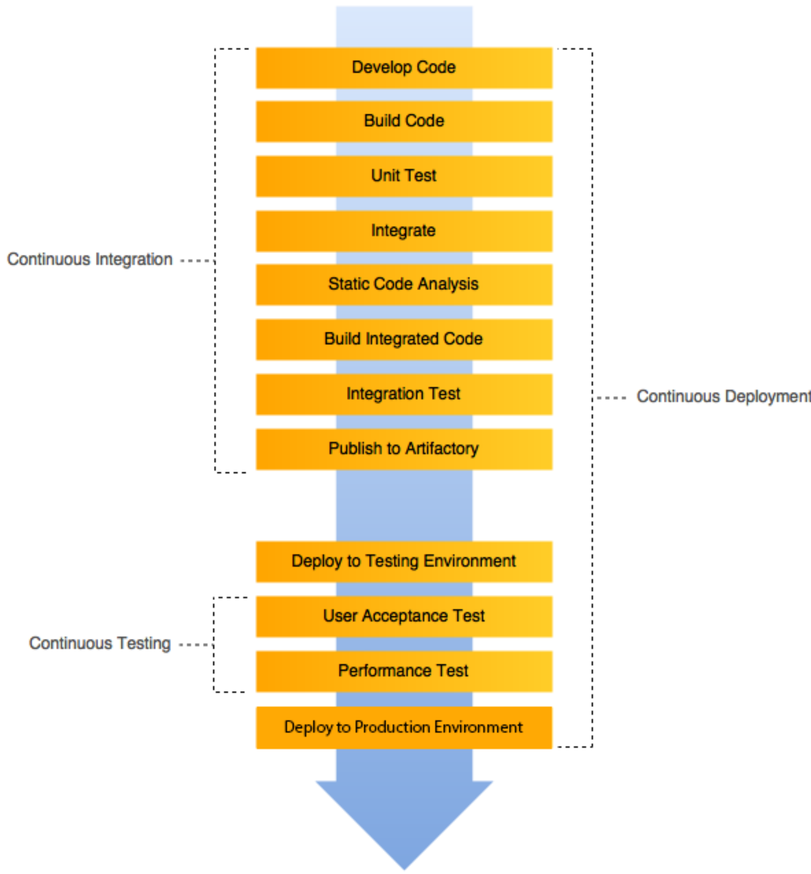
How Continuous Deployment is different from Continuous Delivery

First, the features are developed, then they go through a cycle of Continuous Integration and later through all kinds of testing. Anything that passes the various tests are considered production-ready features. These production-ready features are then labeled in Artifactory (not shown in this book) or are kept separately to segregate them from non-production-ready features.

This is very similar to the manufacturing production line. The raw product goes through phases of modifications and testing. Finally, the finished product is packaged and stored in the warehouses. From the warehouses, depending on the orders, it gets shipped to various places. The product doesn't get shipped immediately after it's packaged. We can safely call this practice Continuous Delivery. The following figure depicts the Continuous Delivery life cycle:



On the other hand, a Continuous Deployment life cycle looks somewhat as shown in the next figure. The deployment phase is immediate without any break. The production-ready features are immediately deployed into the production.



Who needs Continuous Deployment?

You might be wondering about the following things:

- How to achieve Continuous Deployment in your organization?
- What could be the challenges?
- How much testing do I need to incorporate and automate?

And the list goes on. Technical challenges are one thing. What's more important is to realize the fact that do we really need it? Do we really need Continuous Deployment?

The answer is, not always and not in every case. From our definition of Continuous Deployment and our understanding from the previous topic, production-ready features are deployed instantly into the production environments.

In many organizations, it's the business that decides whether or not to make a feature live, or when to make a feature live. Therefore, think of Continuous Deployment as an option and not a compulsion.

On the other hand, Continuous Delivery, which means creating production-ready features in a continuous way, should be the motto for any organization.

Continuous Deployment is easy to achieve in an organization that has just started; in other words, organizations that do not own a large amount of code, have small infrastructures, and have less number of releases per day. On the other hand, it's difficult for organizations with massive projects to move to Continuous Deployment. Nevertheless, organizations with large projects should first target Continuous Integration, then Continuous Delivery, and finally Continuous Deployment.

Frequent downtime of the production environment with Continuous Deployment

Continuous Deployment, though a necessity in some organizations, may not be a piece of cake. There are a few practical challenges that may surface while performing frequent releases to the production server. Let's see some of the hurdles.

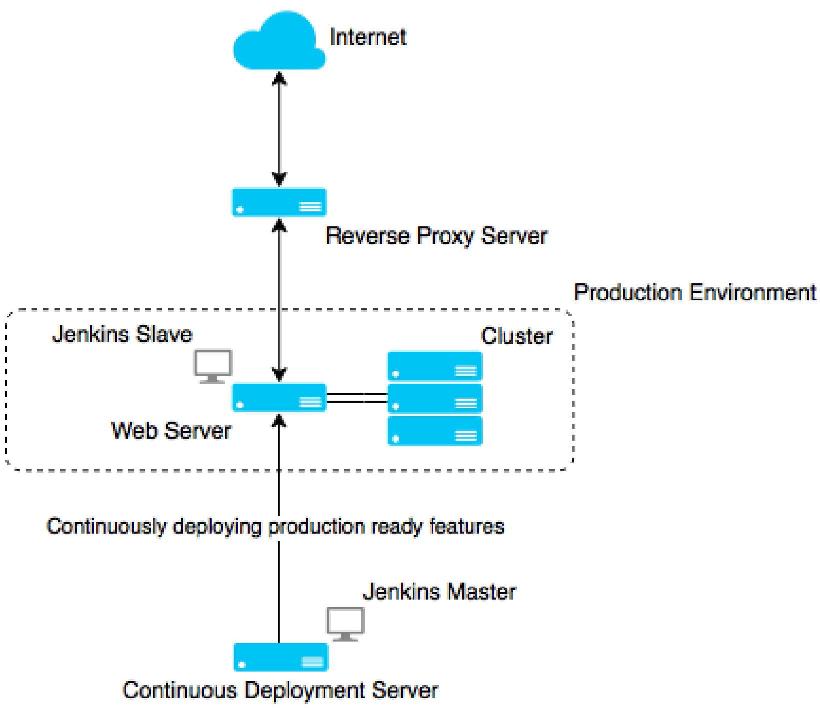
Deployment to any application server or a web server requires downtime. The following activities take place during downtime. I have listed some of the general tasks; they might vary from project to project:

- Bringing down the services
- Deploying the package
- Bringing up the services
- Performing sanity checks

The Jenkins job that performs the deployment in production server may include all the aforementioned steps. Nevertheless, running deployments every now and then on the production server may result in frequent unavailability of the services. The solution to this problem is using a clustered production environment, as shown in the next figure.

This is a very generic example in which the web server is behind a reverse proxy server such as NGINX, which also performs load balancing. The web server is a cluster environment (the web server has many nodes running the same services) that makes it highly available.

Usually, a clustered web application server will have a master-slave architecture, where a single node manager controls a few node agents. When a deployment takes place on the web application server, the changes, the restart activities, and the sanity checks take place on each node—one at a time. This resolves the downtime issues.



Continuous Deployment Design

From the previous sections, we know what Continuous Deployment is and how different it is from Continuous Delivery. It is safe for us to conclude that Continuous Deployment is not an integral part or an extension of Continuous Delivery, but it is a slightly twisted version of it.

Our Continuous Deployment Design will include all the jobs that were the part of the Continuous Delivery Design, with the addition of two more Jenkins jobs and a modification to one of the existing Jenkins jobs. Let's see this in detail.

The Continuous Deployment pipeline

The Continuous Deployment pipeline will include new Jenkins jobs as well as the existing Jenkins jobs that are part of the Continuous Delivery Design. Our new design will grow to around seven Jenkins jobs.

From the previous chapters, we are familiar with the following the Continuous Delivery pipeline:

- Pipeline to poll the feature branch
- Pipeline to poll the integration branch

However, as part of our Continuous Deployment Design, the pipeline to poll the integration branch will be again modified by reconfiguring one of the existing Jenkins jobs and adding additional Jenkins jobs. Together, these new Jenkins pipelines will form our Continuous Delivery pipeline.

Pipeline to poll the feature branch

The Pipeline to poll the feature branch will be kept as it is, and there will be no modifications to it. This particular Jenkins pipeline is coupled with the feature branch. Whenever a developer commits something on the feature branch, the pipeline gets activated. It contains two Jenkins jobs that are as follows.

Jenkins job 1

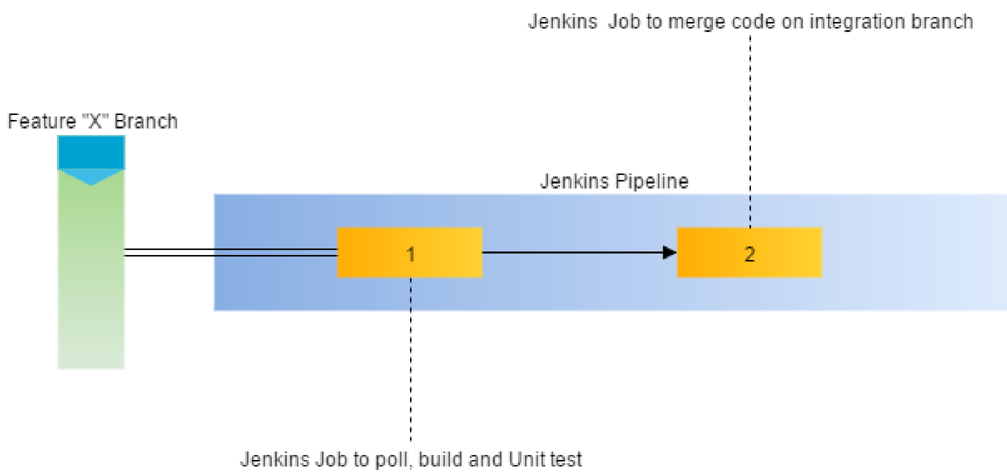
The first Jenkins job in the pipeline performs the following tasks:

- It polls the feature branch for changes at regular intervals
- It performs a build on the modified code
- It executes the unit tests

Jenkins job 2

The second Jenkins job in the pipeline performs the following task:

- It merges the successfully built and tested code into integration branch



Pipeline to poll the integration branch

This Jenkins pipeline is coupled with the integration branch. Whenever there is a new commit on the integration branch, the pipeline gets activated. However, it will now contain seven Jenkins jobs (five older and two new) that perform the following tasks:

Jenkins job 1

The first Jenkins job in the pipeline performs the following tasks:

- It polls the integration branch for changes at regular intervals
- It performs a static code analysis of the downloaded code
- It executes the integration tests
- It passes the `GIT_COMMIT` variable to the Jenkins job that uploads the package to Artifactory

> The GIT_COMMIT variable is a Jenkins system variable that contains the SHA-1 checksum value. Each Git commit has a unique SHA-1 checksum. In this way, we can track which code to build.

Jenkins job 2

The second Jenkins job in the pipeline performs the following tasks:

- It uploads the built package to the binary repository
- It passes the GIT_COMMIT and BUILD_NUMBER variables to the Jenkins job that deploys the package in the production server

> The variable BUILD_NUMBER is a Jenkins system variable that contains the build number. Each Jenkins job has a build number for every run.
>
> We are particularly interested in the build number corresponding to Jenkins job 2. This is because this job uploads the built package to Artifactory. We might need this successfully uploaded artifact later during Jenkins job 3 and Jenkins job 7 to deploy the package to the testing server and production server, respectively.

Jenkins job 3

The third Jenkins job in the pipeline performs the following tasks:

- It deploys the package to the production server using the BUILD_NUMBER variable
- It passes the GIT_COMMIT and BUILD_NUMBER variables to the Jenkins job that performs user acceptance tests

Jenkins job 4

The fourth Jenkins job in the pipeline performs the following tasks:

- It downloads the code from Git using the `GIT_COMMIT` variable
- It performs the user acceptance test
- It generates the test results report
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variables to the Jenkins job that performs the performance test

Jenkins job 5

The fifth Jenkins job in the pipeline performs the following tasks:

- It performs the performance test
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variable to the Jenkins job that performs the performance test (new functionality)

We will create two new Jenkins jobs 6 and 7 with the following functionalities.

Jenkins job 6

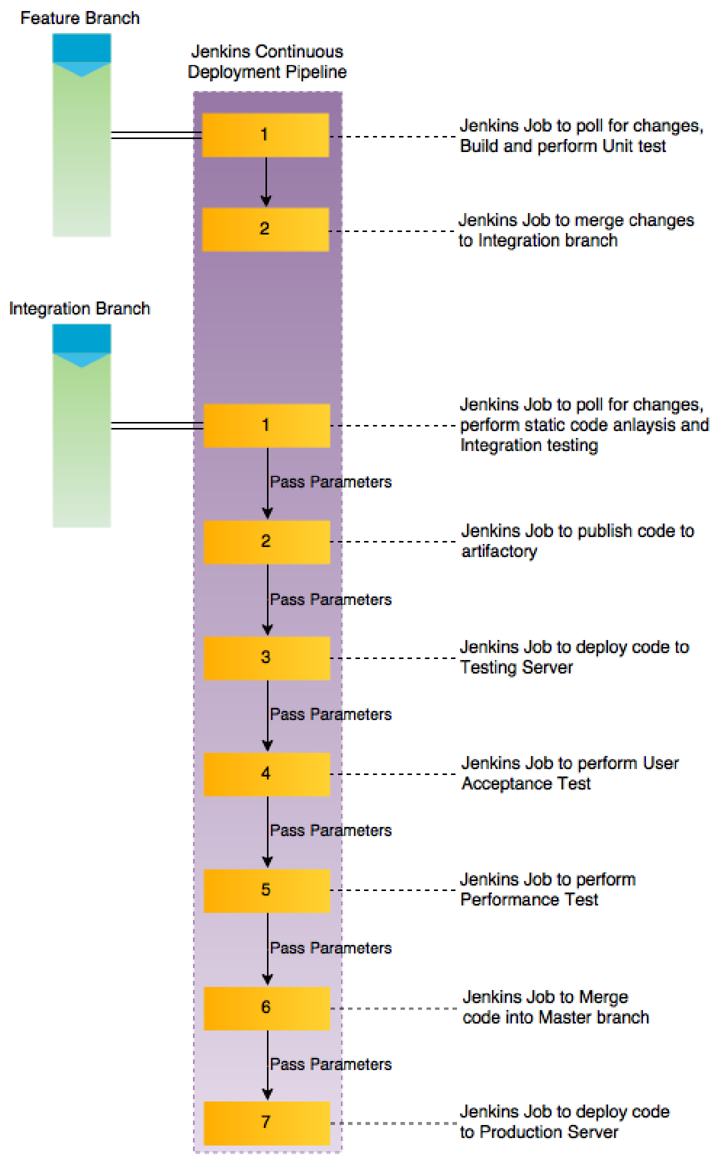
The sixth Jenkins job in the pipeline performs the following tasks:

- It merges successfully tested code into the production branch
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variables to the Jenkins job that performs the performance test

Jenkins job 7

The seventh Jenkins job in the pipeline performs the following task:

- It deploys package to the production server using the `BUILD_NUMBER` variable:



All the Jenkins jobs should have a notification step that can be configured using advanced e-mail notifications.

Toolset for Continuous Deployment

The example project for which we are implementing Continuous Delivery is a Java-based web application; the same one that we used in *Chapter 4*, *Continuous Integration Using Jenkins – Part I*, *Chapter 5*, *Continuous Integration Using Jenkins – Part II*, and *Chapter 6*, *Continuous Delivery Using Jenkins*.

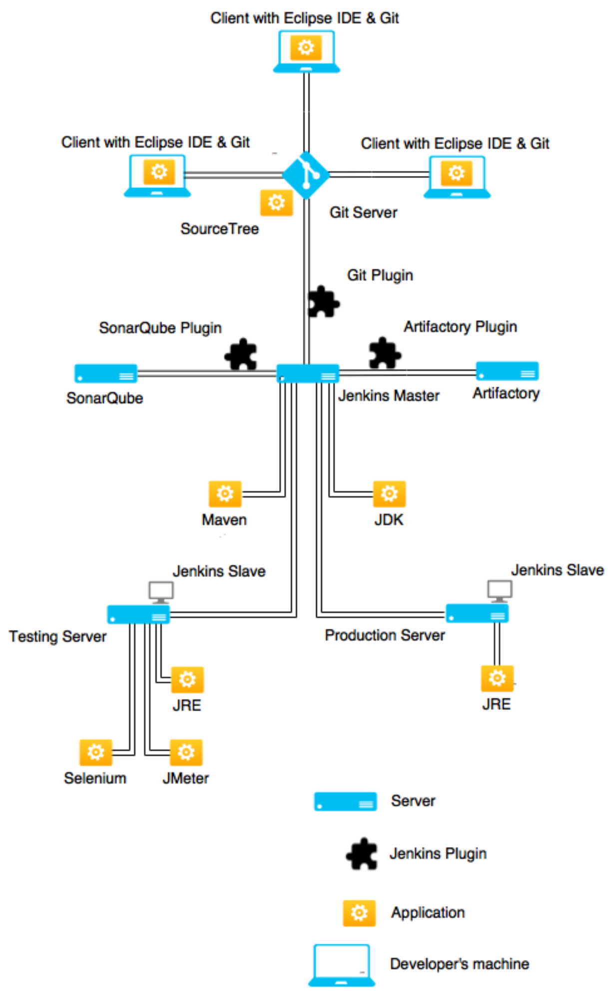
The following table contains the list of tools and technologies involved in everything that we will see in this chapter:

Tools and technologies	**Description**
Java	The primary programming language used for coding
Maven	Build tool
JUnit	Unit test and integration test tools
Apache Tomcat server	Servlet to host the end product
Eclipse	IDE for Java development
Jenkins	Continuous integration tool
Git	Version control system
Artifactory	Binary repository
Source Tree	Git client
SonarQube	Static code analysis tool
JMeter	Performance testing tool
TestNG	Unit test and Integration test tool
Selenium	User acceptance testing tool

The next figure shows how Jenkins fits in as a Continuous Deployment server in our Continuous Deployment Design, along with the other DevOps tools. We can understand the following points from the figure:

- The developers have got the Eclipse IDE and Git installed on their machines. This Eclipse IDE is internally configured with the Git Server. This enables the developers to clone the feature branch from the Git server on their machines.
- The Git server is connected to the Jenkins master server using the Git plugin. This enables Jenkins to poll the Git server for changes.
- The Apache Tomcat server, which hosts the Jenkins master, has also got Maven and JDK installed on it. This enables Jenkins to build the code that has been checked in on the Git server.

- Jenkins is also connected to SonarQube server and the Artifactory server using the SonarQube plugin and the Artifactory plugin, respectively.
- This enables Jenkins to perform a static code analysis of the modified code. Once all the build, quality analysis, and integration testing are successful, the resultant package is uploaded to the Artifactory for further use.
- Consecutively, the package also gets deployed on a testing server that contains testing tools such as JMeter, TestNG, and Selenium. Jenkins, in collaboration with the testing tools, will perform a user acceptance test and a performance test on the code.
- Any code that passes the user acceptance test and the performance test gets deployed in the production server.



Configuring the production server

I chose an Ubuntu machine as our production server. We need to set up some software on it that will assist us while we implement Continuous Deployment. The following steps are almost same as discussed in *Chapter 6*, *Continuous Delivery Using Jenkins*, where we configured the testing server. However, we won't need the testing tools here.

Installing Java on the production server

The production server will have the Apache Tomcat server to host the application. The tool needs Java Runtime Environment running on the machine. Follow the next few steps to install Java JRE on the production server:

1. To install Java JRE on the machine, open a Terminal and give the following commands. This will update all the current application installed on the production server:

   ```
   sudo apt-get update
   ```

2. Generally, Linux OS comes shipped with Java packages. Therefore, check whether Java is already installed using the following command:

   ```
   java -version
   ```

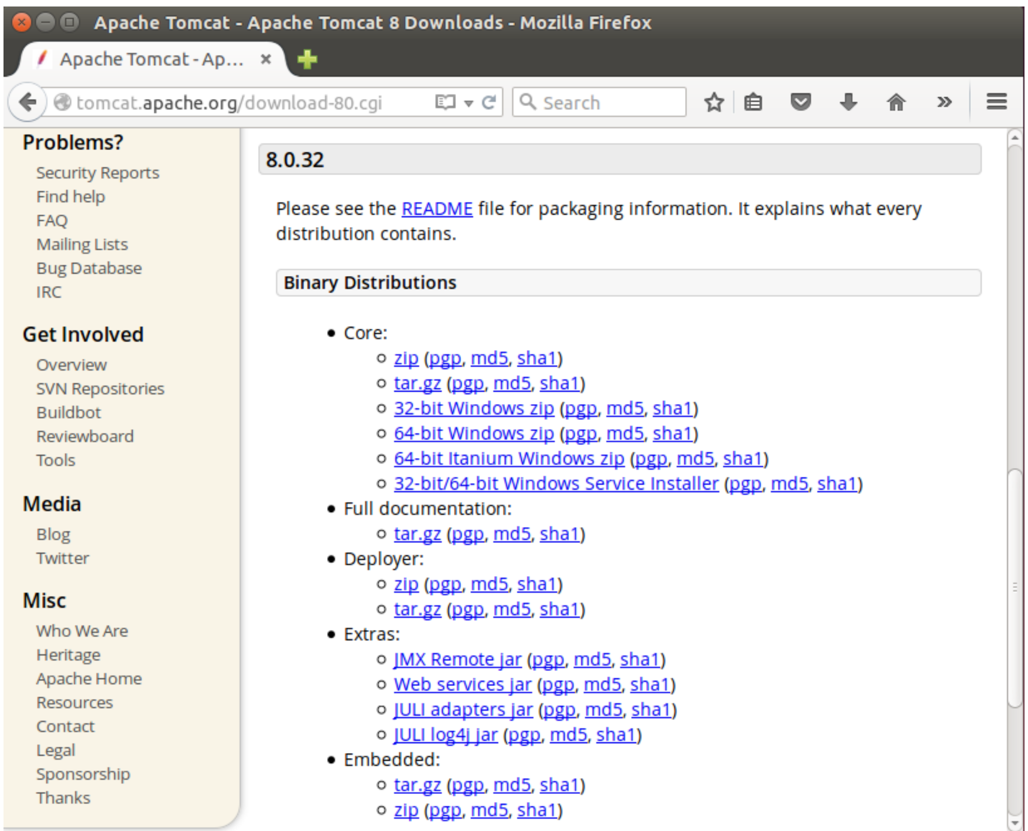
3. If the preceding command returns a Java version, make a note of it. However, if you see the program Java can be found in the following packages, Java hasn't been installed. Execute the following command to install it:

   ```
   sudo apt-get install default-jre
   ```

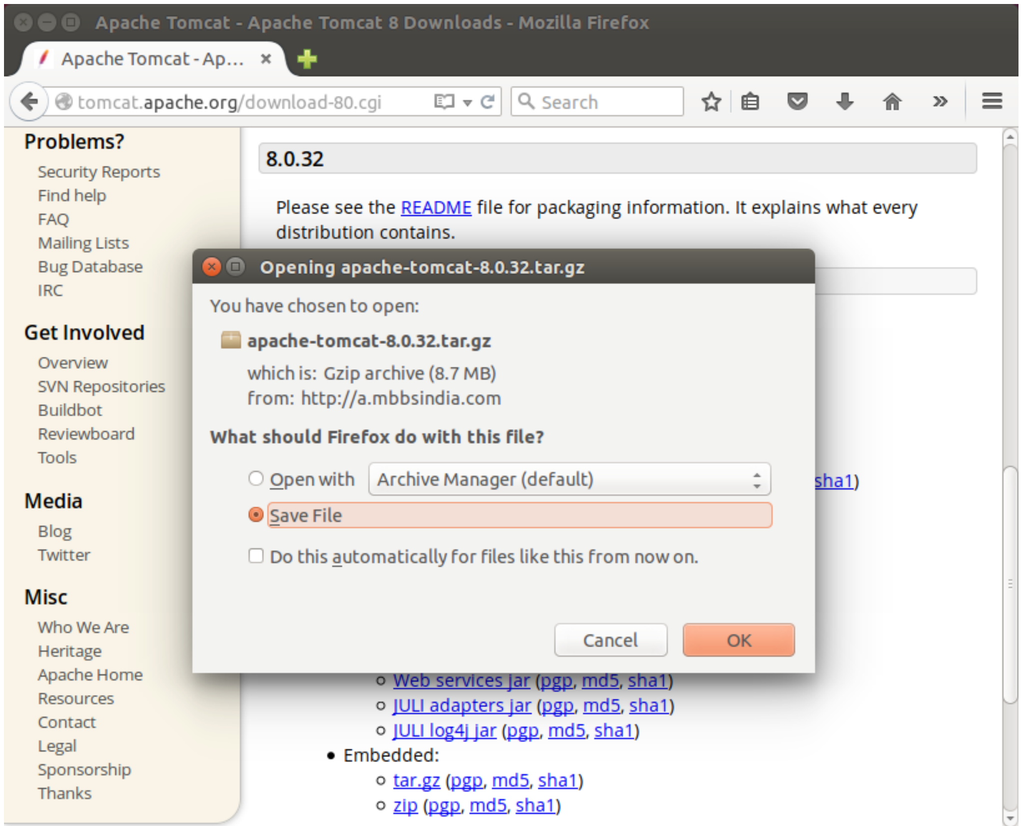
Installing the Apache Tomcat server on the production server

Installing the Apache Tomcat server on Ubuntu is simple. We are doing this to host our application so that it can be tested separately in an isolated environment. The steps are as follows:

1. Download the latest Apache Tomcat server distribution from `http://tomcat.apache.org/download-80.cgi`. Download the `tar.gz` file.



2. Download it to the folder `Downloads`.



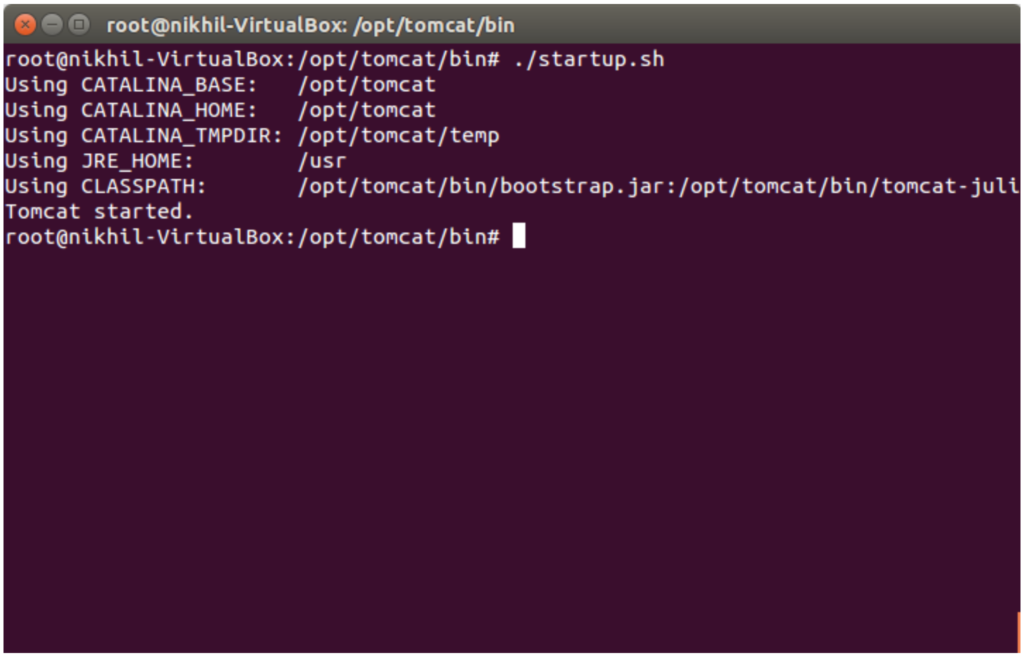
3. We're going to install Tomcat to the `/opt/tomcat` directory. To do so, open a Terminal in Ubuntu.
4. Create the directory, then extract the archive using the following commands:

```
sudo mkdir /opt/tomcat
sudo tar xvf apache-tomcat-8*tar.gz -C /opt/tomcat --strip-components=1
```

5. Start the Apache Tomcat server by executing the following commands:

```
sudo su -
cd /opt/tomcat/bin
./startup.sh
```

6. This will provide the following output:



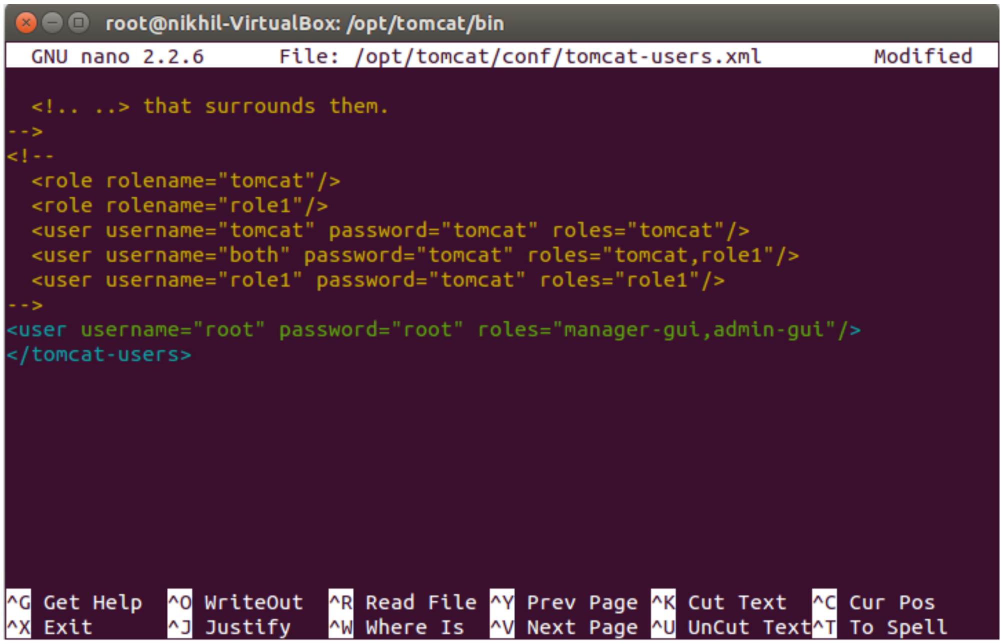
7. That's it! The Apache Tomcat server is up and running. To see it running, open the following link in your favorite web browser: `http://localhost:8080/`.
8. We must now create a user account in order to manage the services using the **manager app** feature that is available on the Apache Tomcat server's dashboard. We will do this by editing the `tomcat-users.xml` file.
9. To do so, type the following command in the Terminal:

   ```
   sudo nano /opt/tomcat/conf/tomcat-users.xml
   ```

10. Add the following line of code between <tomcat-users> and </tomcat-users>:

    ```
    <user username=admin password=password
    roles=manager-gui,admin-gui/>
    ```

11. This is shown in the following screenshot:



```
root@nikhil-VirtualBox: /opt/tomcat/bin
  GNU nano 2.2.6       File: /opt/tomcat/conf/tomcat-users.xml          Modified

  <!.. ..> that surrounds them.
-->
<!--
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
-->
<user username="root" password="root" roles="manager-gui,admin-gui"/>
</tomcat-users>

^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

12. Save and quit the `tomcat-users.xml` file by pressing *Ctrl* + *X* and then *Ctrl* + *Y*.
13. To put our changes into effect, restart the Tomcat server by executing the following commands:

```
cd /opt/tomcat/bin
sudo su -
./shutdown.sh
./startup.sh
```

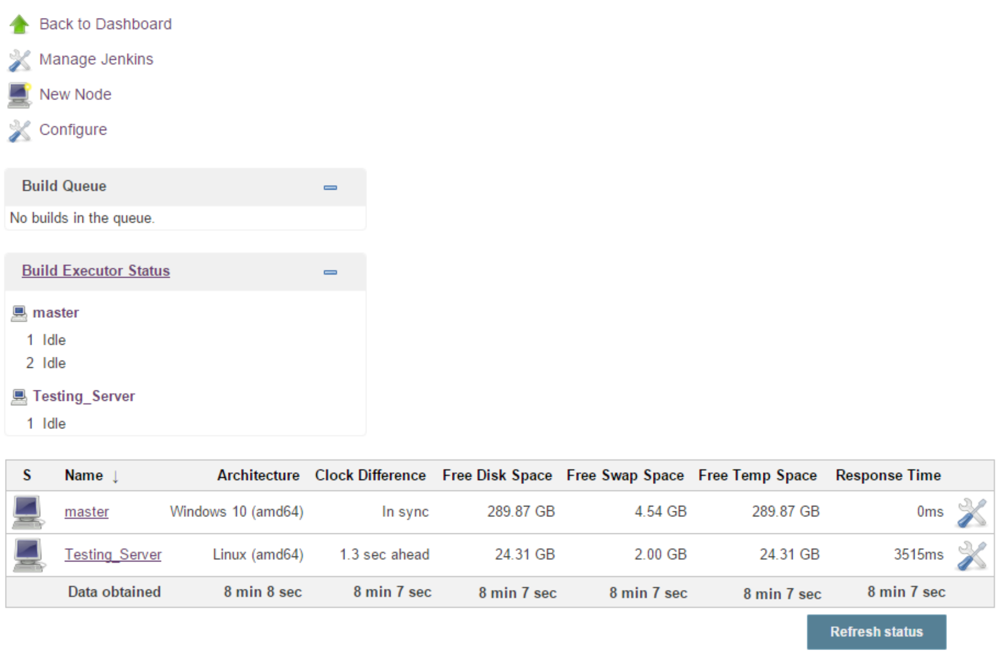
Jenkins configuration

In order to assist the Jenkins jobs that perform various functions to achieve Continuous Deployment, we need to make some changes in the Jenkins configuration. This includes configuring the Jenkins slave agent on the production server and nothing else.

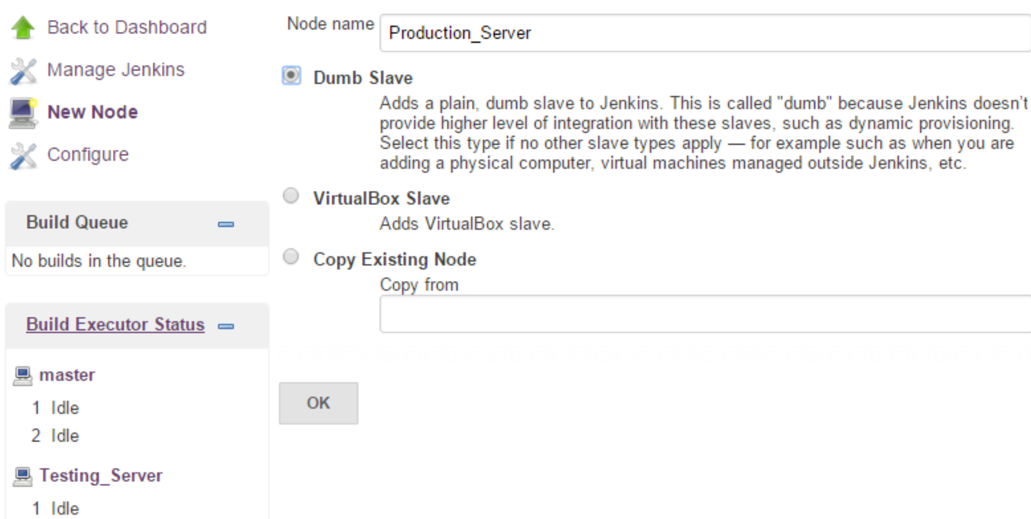
Configuring Jenkins slaves on the production server

In the previous chapter, we saw how to configure a Jenkins slave on the testing server. Here, we will see how to configure a Jenkins slave to run on the production server. In this way, the Jenkins master will be able to communicate and run Jenkins jobs on the slave. Follow the next few steps to set up Jenkins slaves:

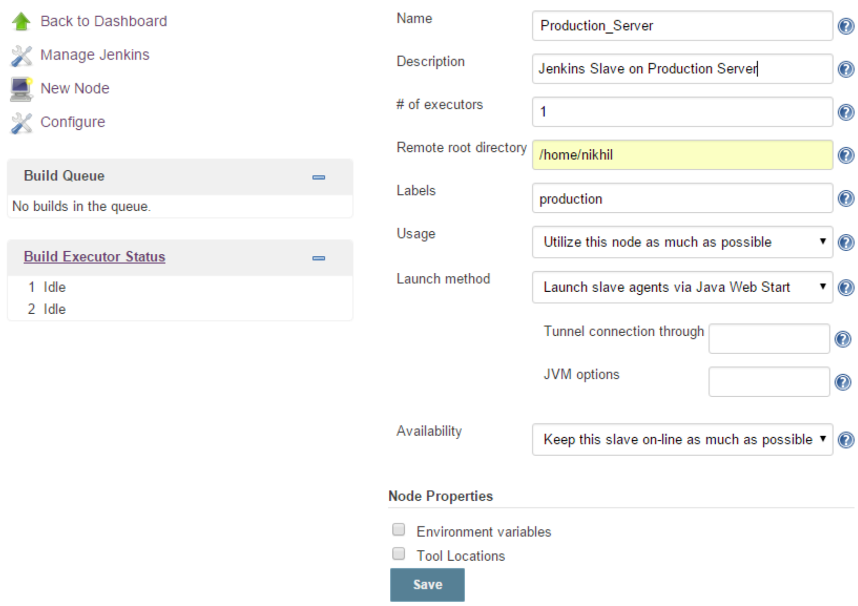
1. Log in to the production server. Open the Jenkins Dashboard from the web browser using the following link: `http://<ip address>:8080/jenkins/`. Remember, you are accessing the Jenkins master from the production server. Here, `<ip address>` is the IP of your Jenkins server.
2. From the Jenkins Dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page. Make sure you have logged in as an admin in Jenkins.
3. Click on the **Manage Nodes** link. In the following screenshot, we can see that the master node (which is the Jenkins server) along with one slave node running on the testing server is listed:



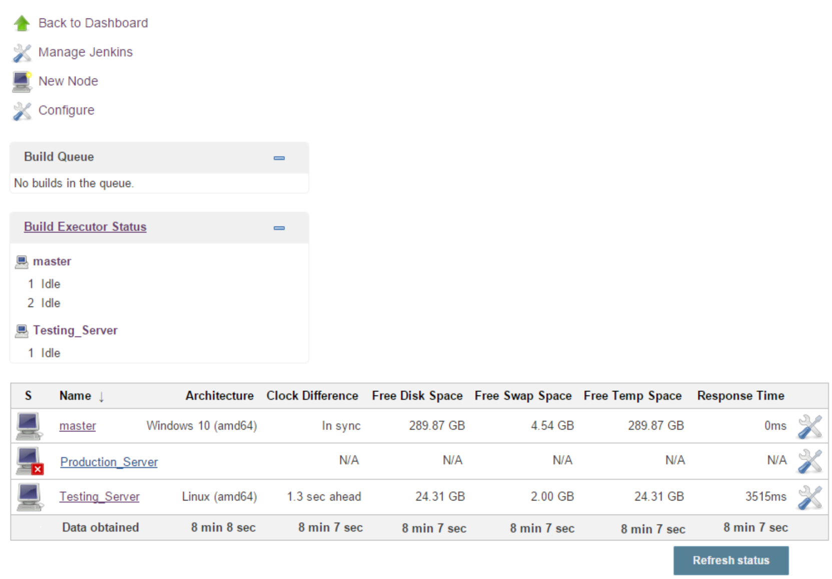
4. Click on the **New Node** button on the left-hand panel. Name the new node Production_Server and select the option **Dumb Slave**. Click on the **OK** button to proceed.



5. Add a description as shown in the screenshot. The **Remote root directory** value should be the local user account on the production server. It should be /home/<user>. The **Labels** field is extremely important; add production as the value.
6. The **Launch method** field should be **Launch slave agents via Java Web Start**:



7. Click on the **Save** button. As you can see in the following screenshot, the Jenkins node on the production server has been configured but it's not running:

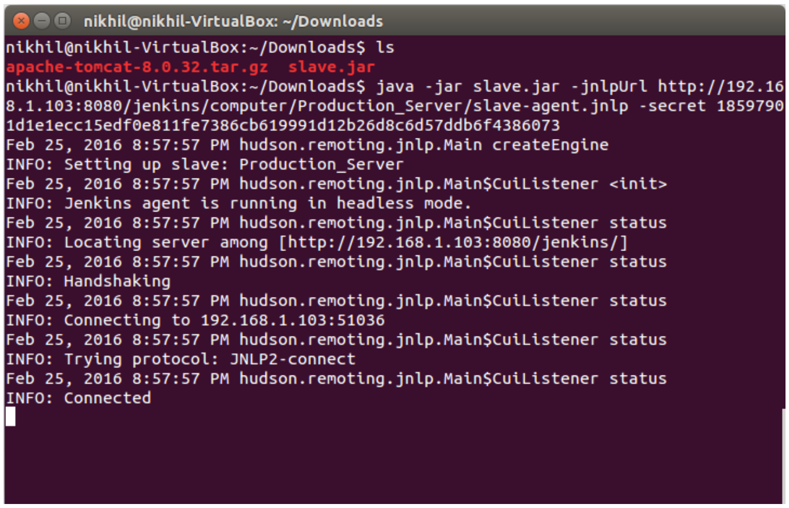


8. Click on the **Production_Server** link in the list of nodes. You will see something like this:

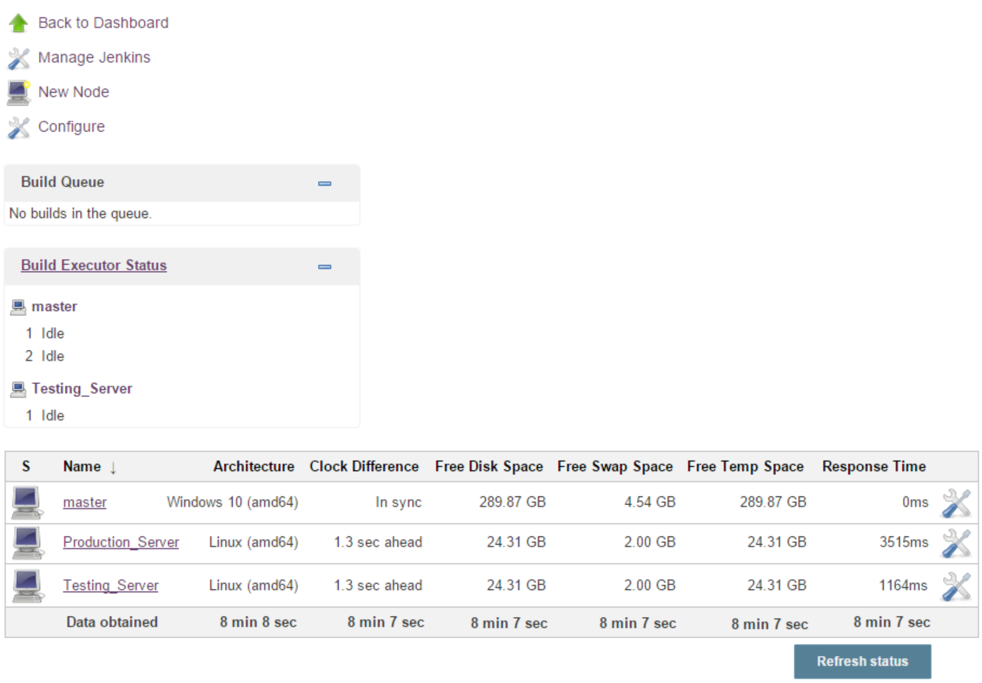
9. You can either click on the **Launch** button in orange, or execute the long command mentioned below it in the Terminal.
10. If you choose the latter option, then download the `slave.jar` file mentioned in the command by clicking on it. It will be downloaded to `/home/<user>/Downloads/`.
11. Execute the following commands in sequence:

```
cd Downloads


java -jar slave.jar -jnlpUrl
http://192.168.1.101:8080/jenkins/computer/
Production_Server/slave-agent.jnlp -secret
916d8164f7ccc1b6fb4521d0c9523eec3b9933328f4cc9cd5e75b4cd65f139f7
```

> The preceding command is machine specific. Do not copy and paste and execute it. Execute the command that appears on your screen.



12. The node on the production server is up and running, as shown in the following screenshot:



Creating the Jenkins Continuous Deployment pipeline

This Continuous Deployment pipeline contains seven Jenkins jobs (five old and two new ones). In this section, we will modify Jenkins job 5 and create two new ones.

Modifying the existing Jenkins job

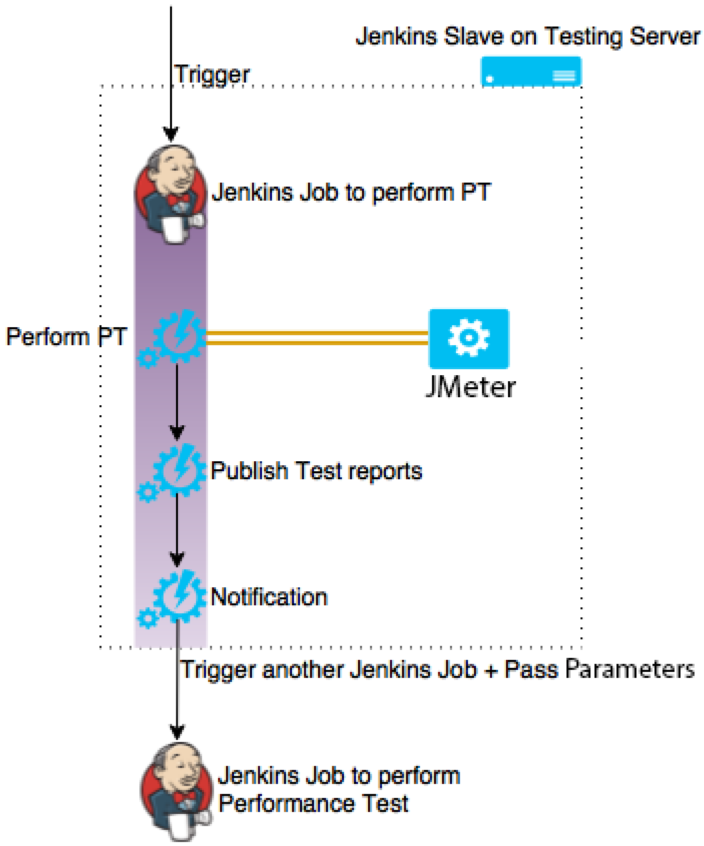
The modification is pretty simple. We need to add a post build step to an existing Jenkins job, `Performance_Testing`. In this way, the `Performance_Testing` job will be able to trigger the new Jenkins job, `Merge_Production_Ready_Code_Into_Master_Branch`, that we will be creating in the coming sections.

Modifying the Jenkins job that performs the performance test

The fifth Jenkins job in the Continuous Deployment pipeline performs the following tasks:

- It performs the performance test
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variables to the Jenkins job that performs the performance test (new functionality)

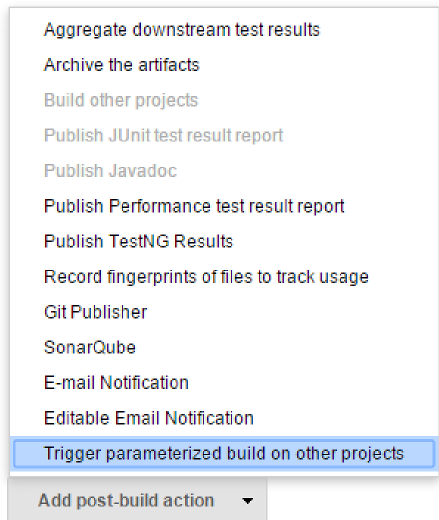
The following figure will help us understand what the Jenkins job does. It's a slightly modified version of what we saw in the previous chapter.



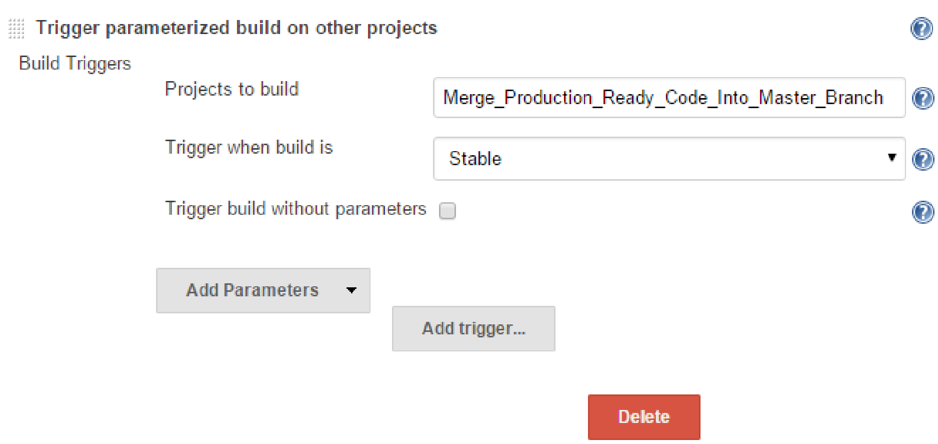
Follow the next few steps to create it:

1. On the Jenkins Dashboard, click on the `Performance_Testing` job.
2. Click on the **Configure** link present on the left-hand panel.
3. Scroll down until you see the **Post-build Actions** section.

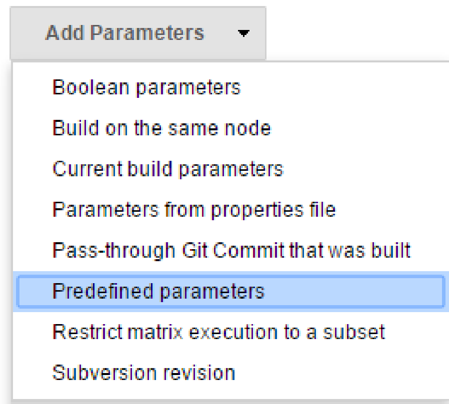
4. Click on the **Add post-build action** button. From the drop-down list, choose the option **Trigger parameterized build on the other projects**.



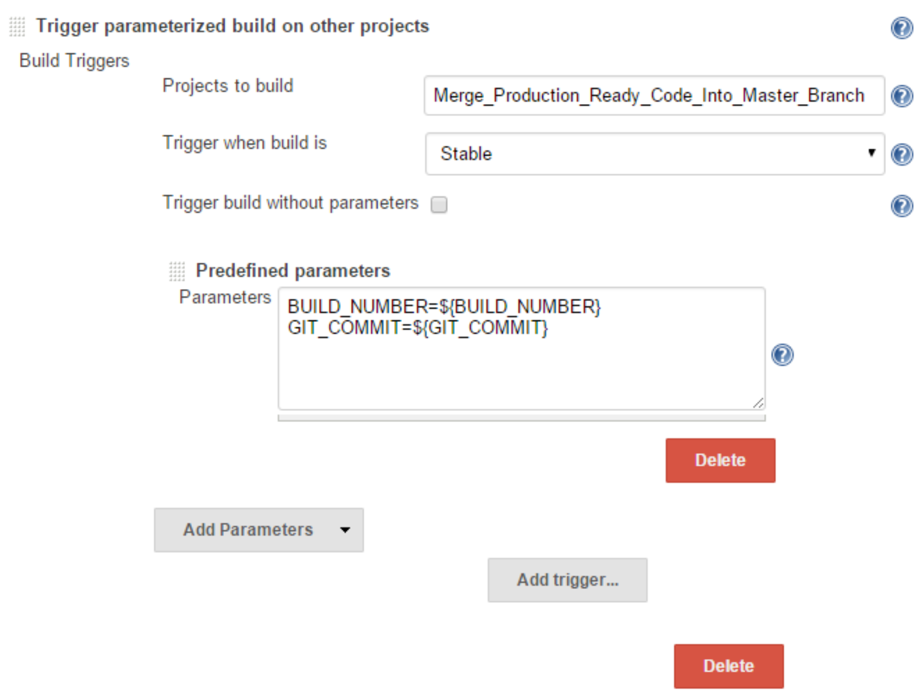
5. Add the values as shown here:



6. Click on the **Add Parameters** button and choose **Predefined parameters**:



7. Add the values as shown here:



8. Save the Jenkins job by clicking on the **Save** button.

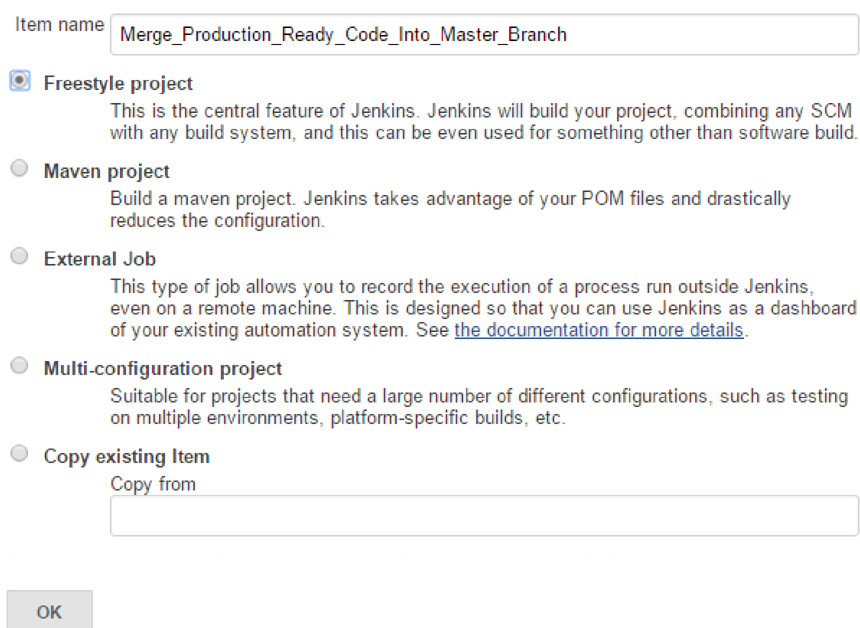
Creating a Jenkins job to merge code from the integration branch to the production branch

The sixth job in the Continuous Deployment pipeline performs the following tasks:

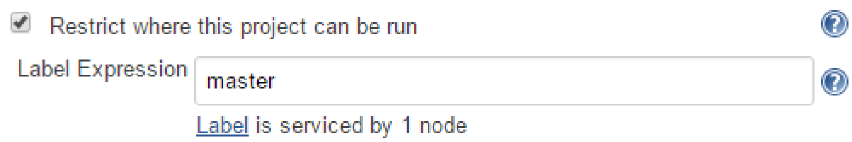
- It merges successfully tested code into the production branch
- It passes the `GIT_COMMIT` and `BUILD_NUMBER` variables to the Jenkins job that performs the performance test

Follow the next few steps to create it:

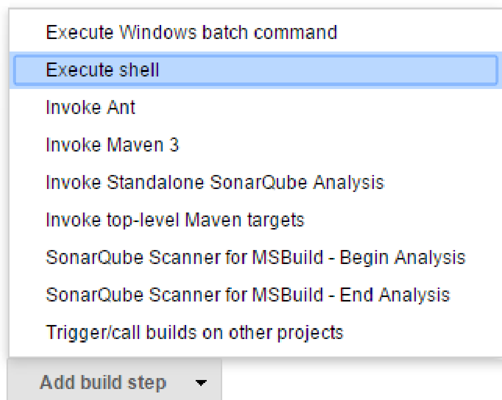
1. On the Jenkins Dashboard, click on **New Item**.
2. Name your new Jenkins job `Merge_Production_Ready_Code_Into_Master_Branch`.
3. Select the type of job as **Freestyle project** and click on **OK** to proceed.



4. Scroll down until you see **Advanced Project Options**. Select **Restrict where this project can be run**.
5. Add `master` as the value for **Label Expression**:



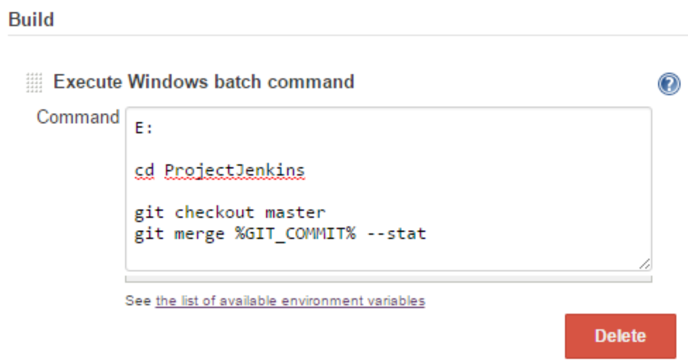
6. Scroll down to the **Build** section.
7. Click on the **Add build step** button and choose the option **Execute shell**.



8. Add the following code to the **Command** field:

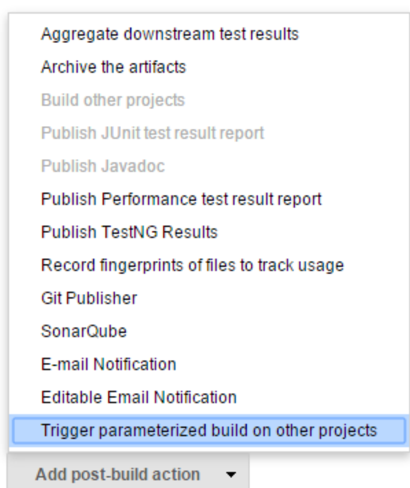
```
E:
cd ProjectJenkins
git checkout master
git merge %GIT_COMMIT% --stat
```

The **Execute Windows batch command** window is shown in the following screenshot:

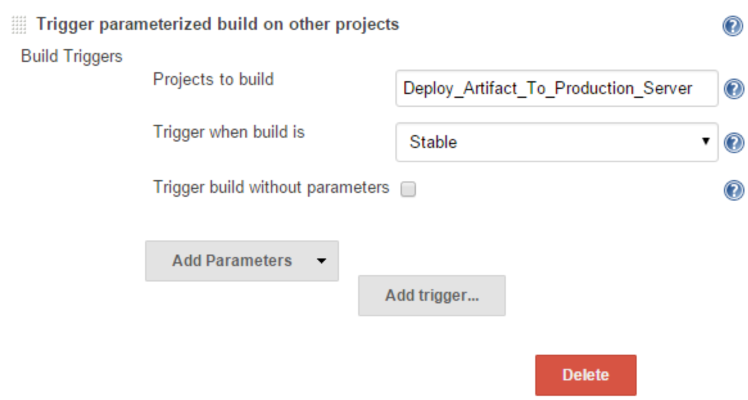


The first line of code switches the current directory to the `E:` drive. The second line of code moves to the Git repository named `ProjectJenkins`. The third line of code checks out the master branch. The fourth line of code merges the particular Git version on the integration branch that is production-ready to the master branch. The `%GIT_COMMIT%` variable represents the successfully tested, production-ready code on the integration branch.

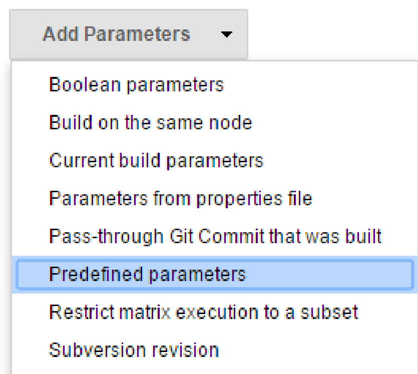
9. Click on the **Add post-build** action button again. From the drop-down list, choose the option **Trigger parameterized build on the other projects**.



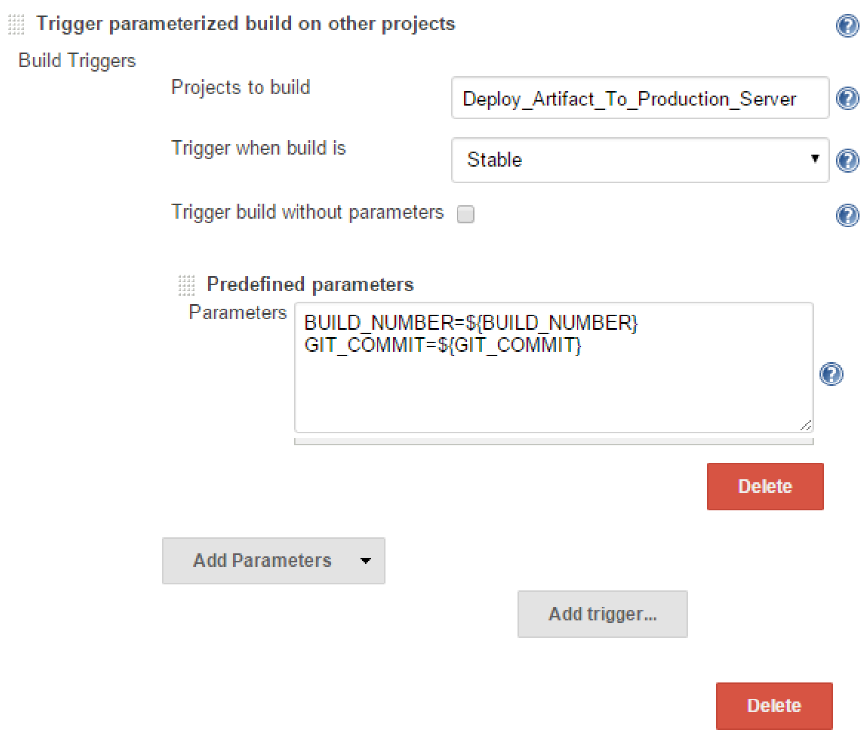
10. Add the values as shown in the next screenshot:



11. Along with triggering the build, we would also like to pass some predefined parameters to it. Click on the **Add Parameters** button and select **Predefined parameters**.



12. Add the values as shown in the next screenshot:



13. Save the Jenkins job by clicking on the **Save** button.

Creating the Jenkins job to deploy code to the production server

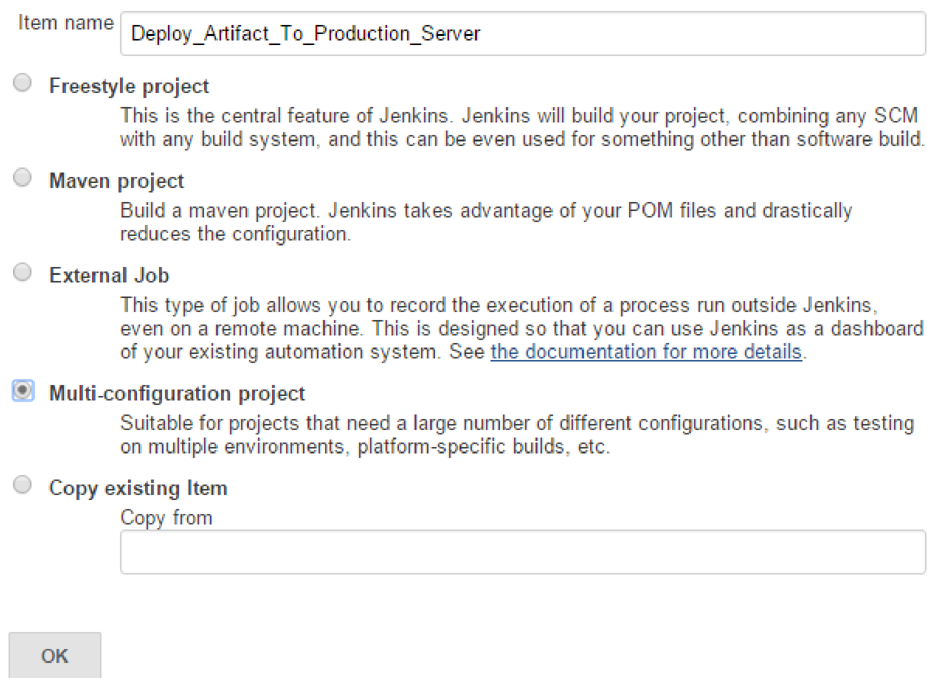
The seventh job in the Continuous Deployment pipeline performs the following task:

- It deploys package to the production server using the `BUILD_NUMBER` variable

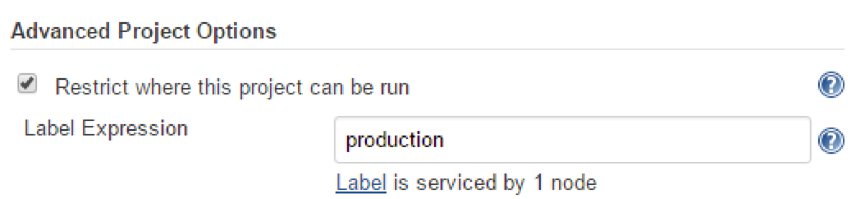
Follow the next few steps to create it:

1. On the Jenkins Dashboard, click on **New Item**.
2. Name your new Jenkins job `Deploy_Artifact_To_Production_Server`.

3. Select the type of job as **Multi-configuration project** and click on **OK** to proceed.

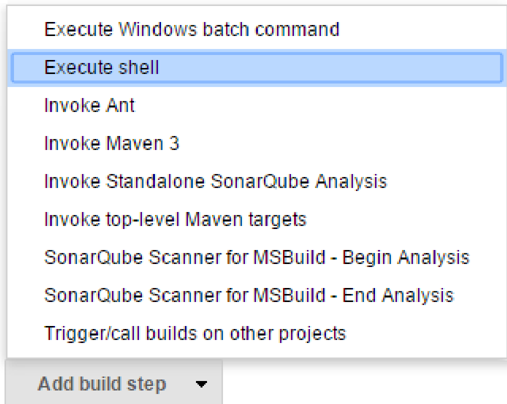


4. Scroll down until you see **Advanced Project Options**. Select **Restrict where this project can be run**.
5. Add `production` as the value for **Label Expression**.



6. Scroll down to the **Build** section.

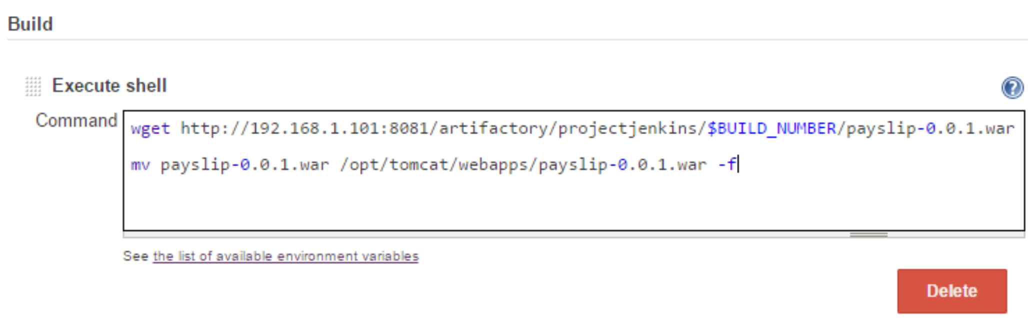
7. Click on the **Add build step** button and choose the option **Execute shell**.



8. Add the following code to the **Command** field:

```
wget http://192.168.1.101:8081/artifactory/projectjenkins/$BUILD_
NUMBER/payslip-0.0.1.war
mv payslip-0.0.1.war /opt/tomcat/webapps/payslip-0.0.1.war -f
```

9. The **Execute shell** window is shown in the following screenshot:



> The first line of the command downloads the respective package from Artifactory to the Jenkins workspace, and the second line of command deploys the downloaded package to Apache Tomcat server's `webapps` directory.

10. Save the Jenkins job by clicking on the **Save** button.

Creating a nice visual flow for the Continuous Delivery pipeline

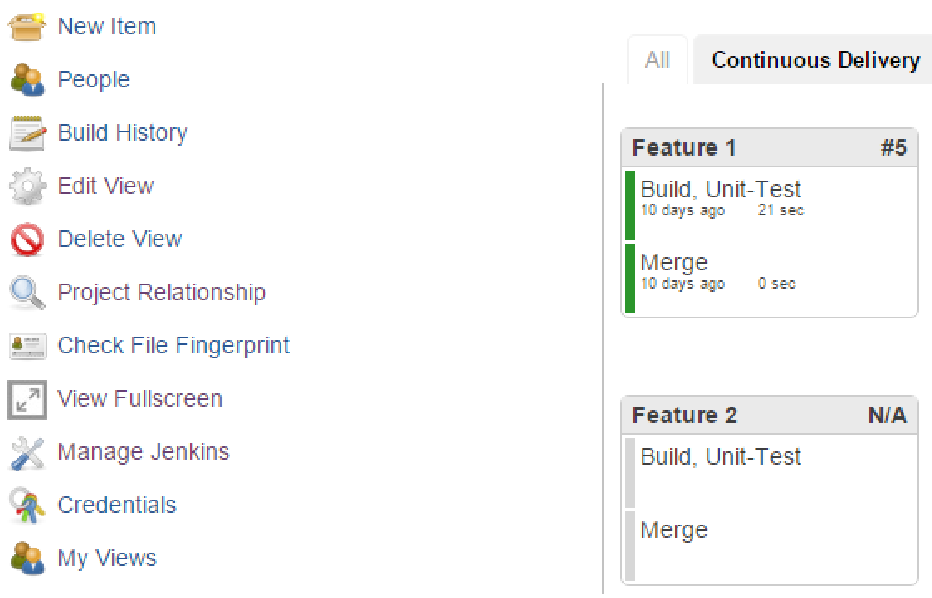
The pipeline to perform Continuous Deployment now contains the following Jenkins jobs:

- `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`
- `Upload_Package_To_Artifactory`
- `Deploy_Artifact_To_Testing_Server`
- `User_Acceptance_Test`
- `Performance_Testing`
- `Merge_Production_Ready_Code_Into_Master_Branch`
- `Deploy_Artifact_To_Production_Server`

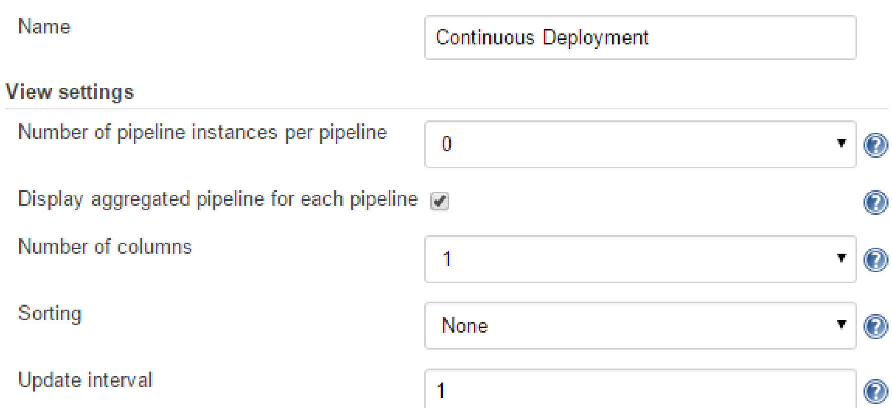
In this section, we will modify the Continuous Delivery view that we created in the previous chapter using the delivery pipeline plugin. The steps are as follows:

1. Go to the Jenkins Dashboard and click on the **Continuous Delivery** tab, as shown in the following screenshot:

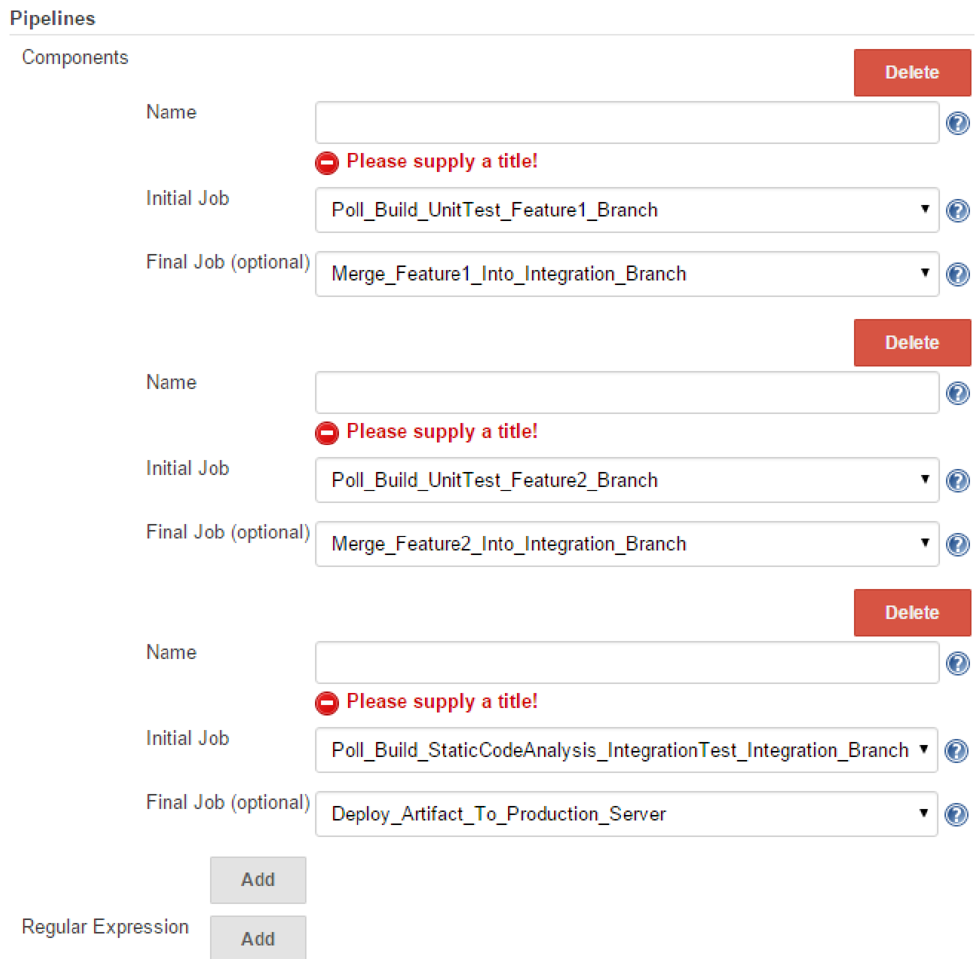
2. You will see the following page. Click on the **Edit View** link present on the left-hand side menu.



3. Now, you will see a lot of options that are already filled. Scroll down until you see the **View settings** section.
4. Change the value of **Name** from `Continuous Delivery` to `Continuous Deployment`.

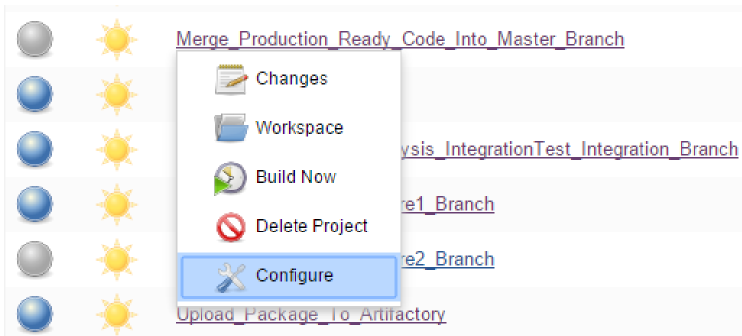


5. Leave the rest of the options at their default values and scroll down until you see the **Pipelines** section.
6. You can see in the following screenshot that three components are listed:

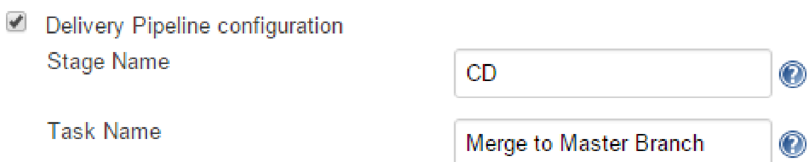


7. In the last component, change the **Final Job (optional)** value from `Performance_Testing` to `Deploy_Artifact_To_Production_Server`.
8. Click on **OK** to save the configuration.
9. Now, come back to the Jenkins Dashboard.

10. Right-click on the **Merge_Production_Ready_Code_Into_Master_Branch** Jenkins job and select **Configure**, as shown in the following screenshot:

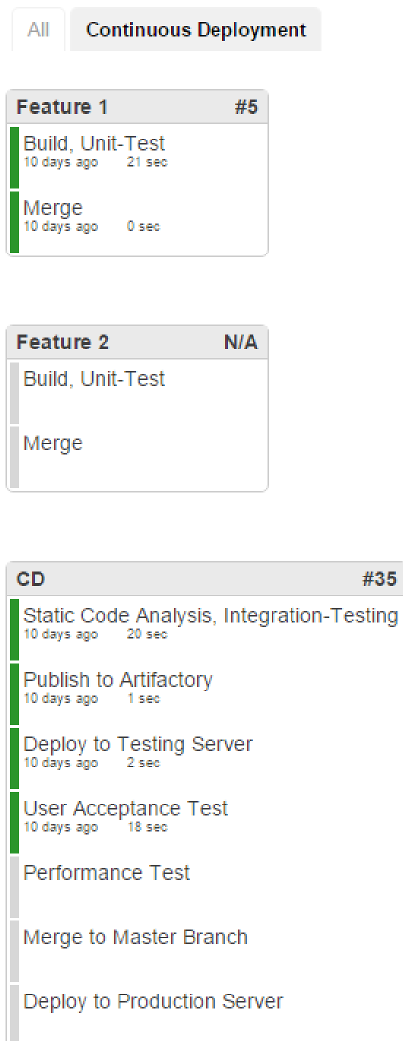


11. Look for the **Delivery Pipeline configuration** option and select it.
12. Under the same, add **Stage Name** as `CD` and **Task Name** as `Merge to Master Branch`:



13. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
14. Now, come back to the Jenkins Dashboard.
15. Right-click on the **Deploy_Artifact_To_Production_Server** Jenkins job and select **Configure**.
16. Look for the **Delivery Pipeline configuration** option and select it.
17. Here, set **Stage Name** as `CD` and **Task Name** as `Deploy to Production Server`.

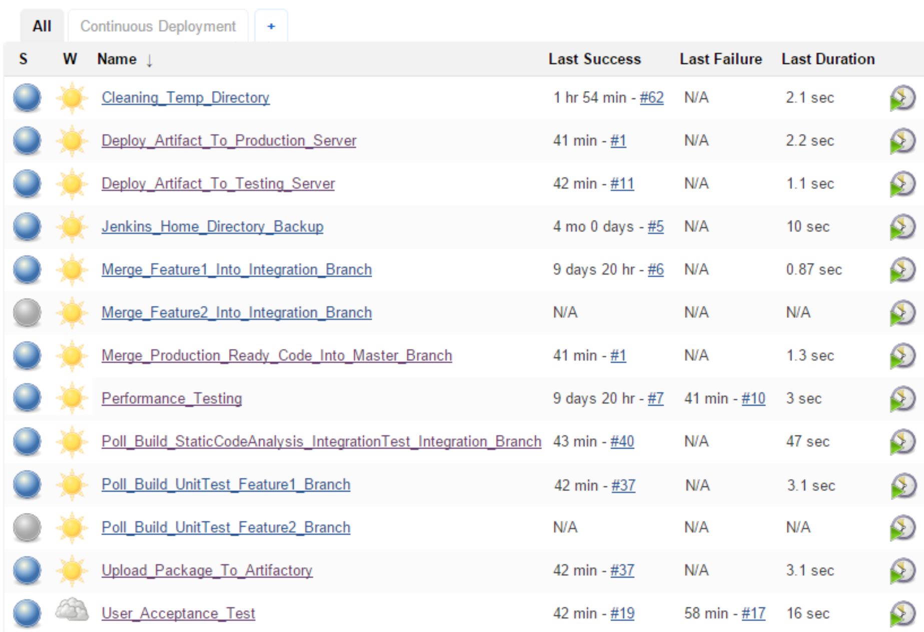
18. Save the configuration by clicking on the **Save** button at the bottom of the page before moving on.
19. Come back to the Jenkins Dashboard and click on the **Continuous Deployment** view. Tada!! This is what you will see:



Continuous Deployment in action

To keep things simple, we won't be making any code changes. Instead, we will simply retrigger our Jenkins job to poll the integration branch, that is, `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`, to begin the Continuous Deployment pipeline. The steps are as follows:

1. From the Jenkins Dashboard, click on the build button for the **Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch Jenkins** job.
2. That's it! The pipeline begins.

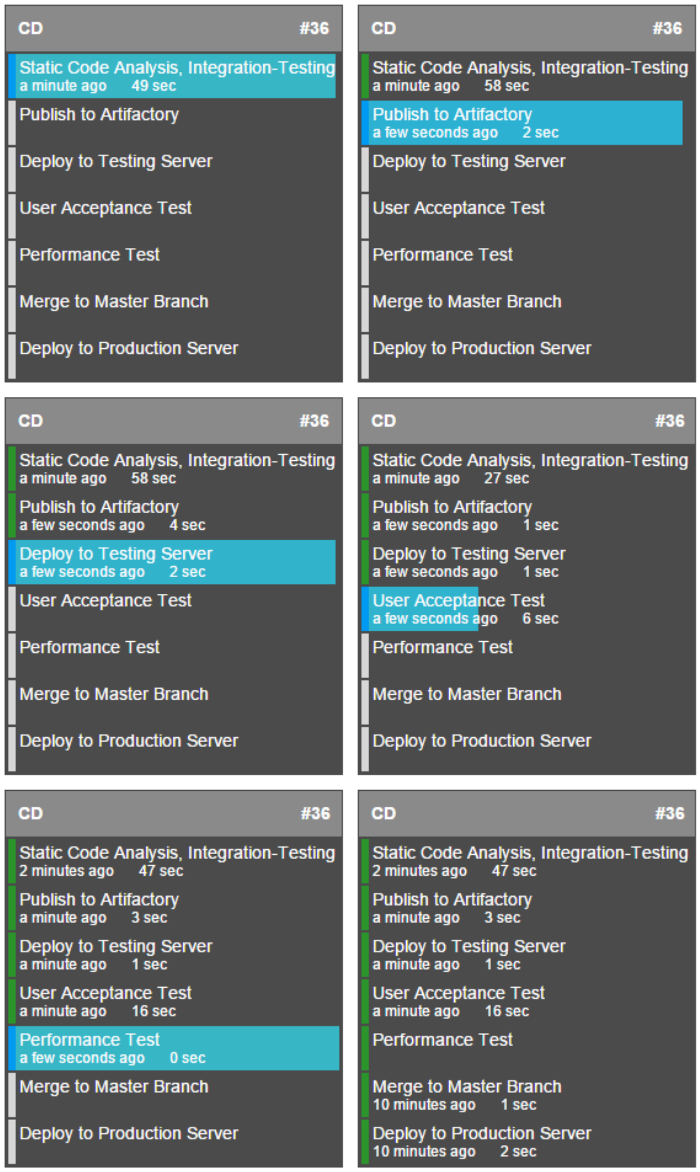


Jenkins Continuous Deployment pipeline flow in action

We have successfully triggers the Jenkins Continuous Deployment pipeline. Now let's see it in action:

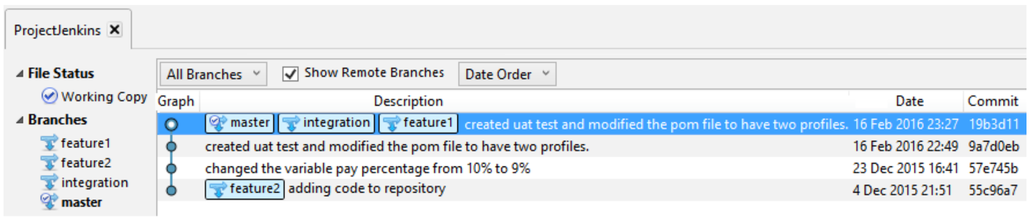
1. Go to the Jenkins Dashboard and click on the **Continuous Deployment** view.
2. From the menu present on the left-hand side, click on the **View Fullscreen** link.

3. You will see the Jenkins jobs in the Continuous Deployment pipeline in action, as shown here:



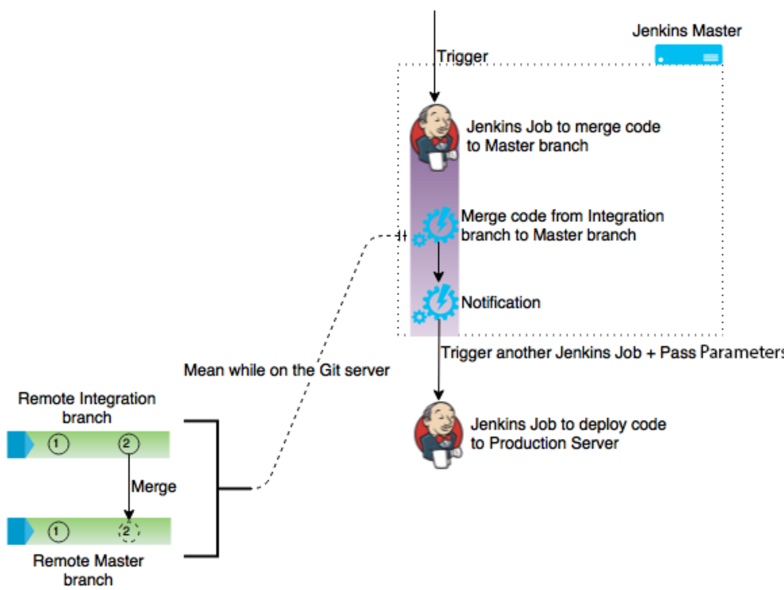
The proceeding image shows the **Continuous Delivery** (**CD**) pipeline in progress.

4. Open the source tree and you can see the `master`, `integration`, and `feature1` branches are all at the same level.



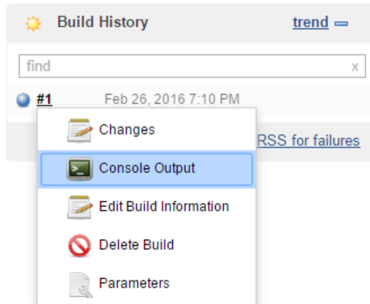
Exploring the Jenkins job to merge code to the master branch

The Continuous Deployment pipeline has worked well. The following figure shows an overview of the tasks that happen while this particular Jenkins job runs:

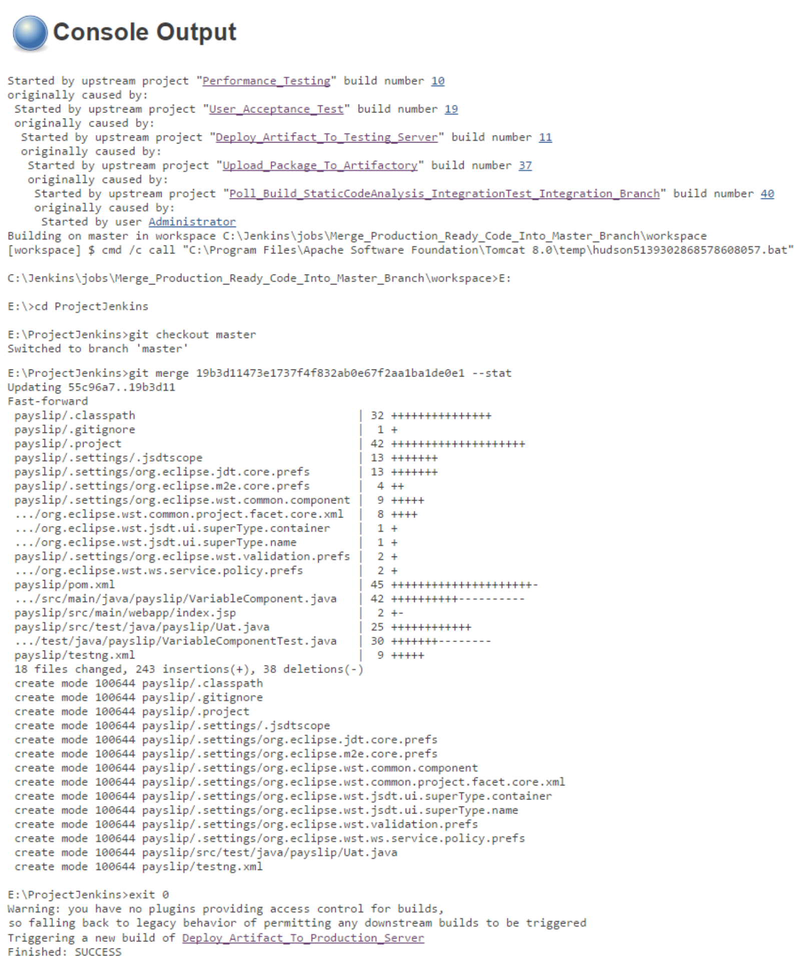


Let's go through the Jenkins job:

1. From the Jenkins Dashboard, click on the **Project Merge_Production_Ready_Code_Into_Master_Branch** job.
2. From the **Build History** panel, right-click on any of the builds.



3. You will see the following build log. This is the log from the Jenkins master server's perspective:

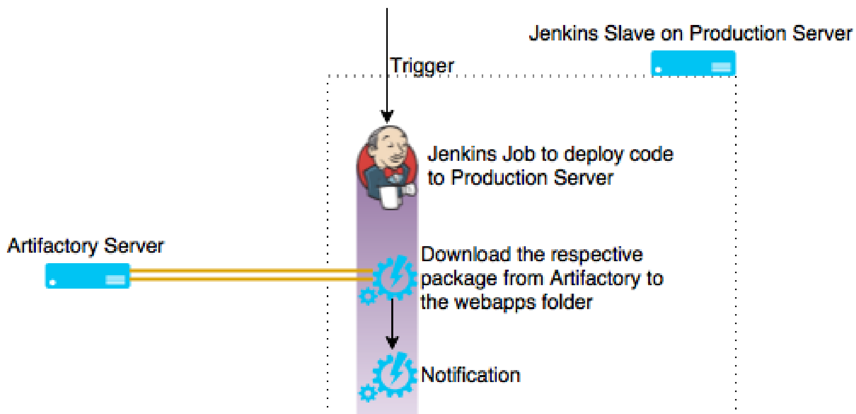


Console Output

```
Started by upstream project "Performance_Testing" build number 10
originally caused by:
 Started by upstream project "User_Acceptance_Test" build number 19
 originally caused by:
  Started by upstream project "Deploy_Artifact_To_Testing_Server" build number 11
  originally caused by:
   Started by upstream project "Upload_Package_To_Artifactory" build number 37
   originally caused by:
    Started by upstream project "Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch" build number 40
    originally caused by:
     Started by user Administrator
Building on master in workspace C:\Jenkins\jobs\Merge_Production_Ready_Code_Into_Master_Branch\workspace
[workspace] $ cmd /c call "C:\Program Files\Apache Software Foundation\Tomcat 8.0\temp\hudson5139302868578608057.bat"

C:\Jenkins\jobs\Merge_Production_Ready_Code_Into_Master_Branch\workspace>E:

E:\>cd ProjectJenkins

E:\ProjectJenkins>git checkout master
Switched to branch 'master'

E:\ProjectJenkins>git merge 19b3d11473e1737f4f832ab0e67f2aa1ba1de0e1 --stat
Updating 55c96a7..19b3d11
Fast-forward
 payslip/.classpath                                 | 32 +++++++++++++++
 payslip/.gitignore                                 |  1 +
 payslip/.project                                   | 42 ++++++++++++++++++++
 payslip/.settings/.jsdtscope                       | 13 +++++++
 payslip/.settings/org.eclipse.jdt.core.prefs       | 13 +++++++
 payslip/.settings/org.eclipse.m2e.core.prefs       |  4 ++
 payslip/.settings/org.eclipse.wst.common.component |  9 +++++
 .../org.eclipse.wst.common.project.facet.core.xml  |  8 ++++
 .../org.eclipse.wst.jsdt.ui.superType.container    |  1 +
 .../org.eclipse.wst.jsdt.ui.superType.name         |  1 +
 payslip/.settings/org.eclipse.wst.validation.prefs |  2 +
 .../org.eclipse.wst.ws.service.policy.prefs        |  2 +
 payslip/pom.xml                                    | 45 +++++++++++++++++++++-
 .../src/main/java/payslip/VariableComponent.java   | 42 ++++++++++----------
 payslip/src/main/webapp/index.jsp                  |  2 +-
 payslip/src/test/java/payslip/Uat.java             | 25 ++++++++++++
 .../test/java/payslip/VariableComponentTest.java   | 30 +++++++--------
 payslip/testng.xml                                 |  9 +++++
 18 files changed, 243 insertions(+), 38 deletions(-)
 create mode 100644 payslip/.classpath
 create mode 100644 payslip/.gitignore
 create mode 100644 payslip/.project
 create mode 100644 payslip/.settings/.jsdtscope
 create mode 100644 payslip/.settings/org.eclipse.jdt.core.prefs
 create mode 100644 payslip/.settings/org.eclipse.m2e.core.prefs
 create mode 100644 payslip/.settings/org.eclipse.wst.common.component
 create mode 100644 payslip/.settings/org.eclipse.wst.common.project.facet.core.xml
 create mode 100644 payslip/.settings/org.eclipse.wst.jsdt.ui.superType.container
 create mode 100644 payslip/.settings/org.eclipse.wst.jsdt.ui.superType.name
 create mode 100644 payslip/.settings/org.eclipse.wst.validation.prefs
 create mode 100644 payslip/.settings/org.eclipse.wst.ws.service.policy.prefs
 create mode 100644 payslip/src/test/java/payslip/Uat.java
 create mode 100644 payslip/testng.xml

E:\ProjectJenkins>exit 0
Warning: you have no plugins providing access control for builds,
so falling back to legacy behavior of permitting any downstream builds to be triggered
Triggering a new build of Deploy_Artifact_To_Production_Server
Finished: SUCCESS
```

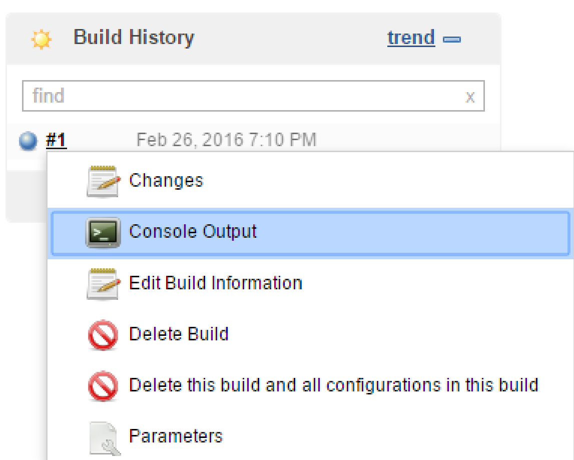
Exploring the Jenkins job that deploys code to production

The following figure gives an overview of the tasks that happen while this particular Jenkins job runs:

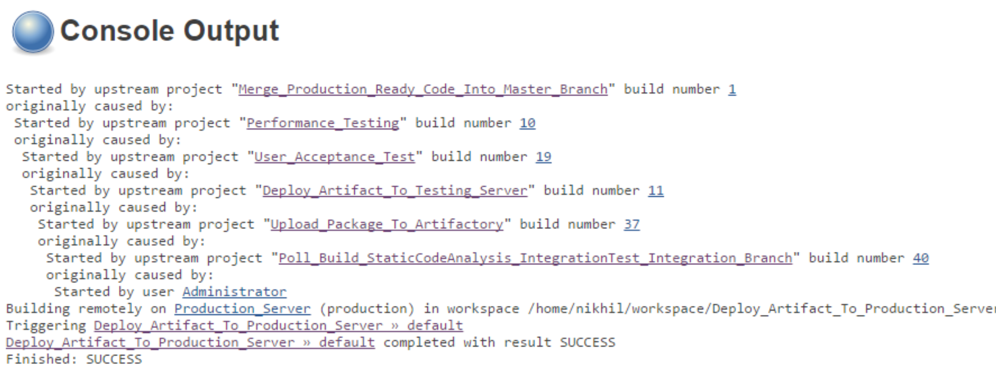


Let's go through the deployment steps:

1. On the Jenkins Dashboard, click on the **Project Merge_Production_Ready_Code_Into_Master_Branch** job.
2. On the **Build History** panel, right-click on any of the builds.

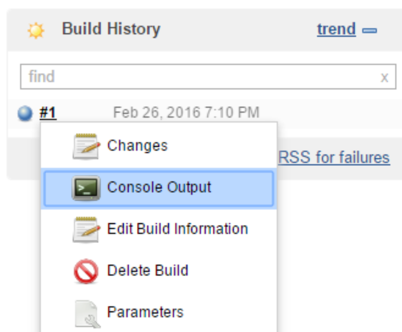


3. You will see the following build log. This is the log from the Jenkins master server's perspective.



```
Console Output

Started by upstream project "Merge_Production_Ready_Code_Into_Master_Branch" build number 1
originally caused by:
 Started by upstream project "Performance_Testing" build number 10
 originally caused by:
  Started by upstream project "User_Acceptance_Test" build number 19
  originally caused by:
   Started by upstream project "Deploy_Artifact_To_Testing_Server" build number 11
   originally caused by:
    Started by upstream project "Upload_Package_To_Artifactory" build number 37
    originally caused by:
     Started by upstream project "Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch" build number 40
     originally caused by:
      Started by user Administrator
Building remotely on Production_Server (production) in workspace /home/nikhil/workspace/Deploy_Artifact_To_Production_Server
Triggering Deploy_Artifact_To_Production_Server » default
Deploy_Artifact_To_Production_Server » default completed with result SUCCESS
Finished: SUCCESS
```

4. Click on the **Deploy_Artifact_To_Production_Server " default** link. On the landing page, go to the **Build History** panel and right-click on any of the builds.



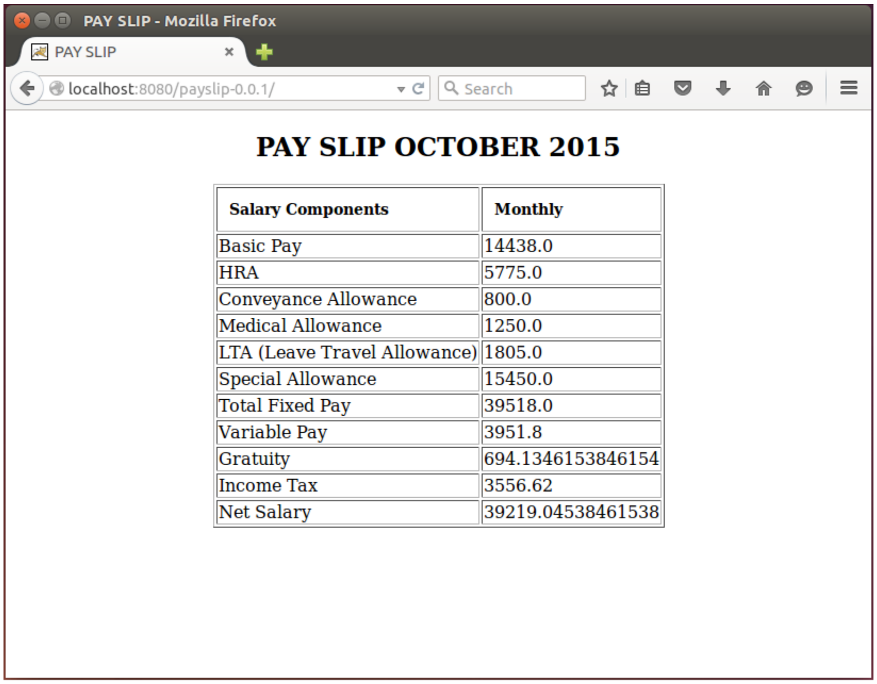
5. You will see the following build log. This is the log from the Jenkins slave's perspective.

Console Output

```
Started by upstream project "Deploy_Artifact_To_Production_Server" build number 1
originally caused by:
 Started by upstream project "Merge_Production_Ready_Code_Into_Master_Branch" build number 1
 originally caused by:
  Started by upstream project "Performance_Testing" build number 10
  originally caused by:
   Started by upstream project "User_Acceptance_Test" build number 19
   originally caused by:
    Started by upstream project "Deploy_Artifact_To_Testing_Server" build number 11
    originally caused by:
     Started by upstream project "Upload_Package_To_Artifactory" build number 37
     originally caused by:
      Started by upstream project "Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch" build number 40
      originally caused by:
       Started by user Administrator
Building remotely on Testing_Server (Testing)No JDK named 'null' found
 in workspace /home/nikhil/workspace/Deploy_Artifact_To_Production_Server/default
No JDK named 'null' found
[default] $ /bin/sh -xe /tmp/hudson9205098568008895173.sh
+ wget http://192.168.1.104:8081/artifactory/projectjenkins/37/payslip-0.0.1.war
--2016-02-26 19:10:29--  http://192.168.1.104:8081/artifactory/projectjenkins/37/payslip-0.0.1.war
Connecting to 192.168.1.104:8081... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17545016 (17M) [application/java-archive]
Saving to: 'payslip-0.0.1.war'

2016-02-26 19:10:30 (31.7 MB/s) - 'payslip-0.0.1.war' saved [17545016/17545016]

+ mv payslip-0.0.1.war /opt/tomcat/webapps/payslip-0.0.1.war -f
Finished: SUCCESS
```

6. Log in to the production server and open `http://localhost:8080/payslip-0.0.1/` from your favorite web browser.

   Alternatively, open the link `http://<ip address>:8080/payslip-0.0.1/` from any machine. Here, `<ip address>` is the IP address of the production server.



Summary

This marks the end of Continuous Deployment. In this chapter, we saw how to achieve Continuous Deployment using Jenkins. We also discussed the difference between Continuous Delivery and Continuous Deployment. There were no major setups and configurations in this chapter, as all the necessary things were achieved in the previous chapters while implementing Continuous Integration and Continuous Delivery.

In the next chapter, we will see some of the best practices of Jenkins. We will also see the distributed build architecture that is used to balance the load on the Jenkins master server.

8
Jenkins Best Practices

This chapter is all about Jenkins best practices. We will begin the chapter with the distributed builds, where we will see how to harness the Jenkins master-slave architecture to achieve load balancing while performing builds.

We will also see how to version control Jenkins system configuration and job configuration, along with auditing Jenkins. This will give us more control over Jenkins in the event of system failures.

Next, we will see how to connect Jenkins with communication tools to send notifications. This will give us an edge over the older e-mail-based notification system.

Lastly, we will discuss some other best practices related to Jenkins jobs and Jenkins updates. If some of you are not happy with the Jenkins GUI, there is section at the end to install Jenkins themes.

These are the important topics that we will cover in this chapter:

- Creating a build farm using Jenkins slaves
- Installing and configuring a jobConfigHistory plugin to version control Jenkins configurations
- Installing and configuring the Audit Trail plugin to audit Jenkins
- Installing and configuring HipChat
- Configuring HipChat with Jenkins to send notifications
- Configuring Jenkins to automatically clean up the job workspace
- Installing Jenkins themes using the Simple Theme Plugin

Distributed builds using Jenkins

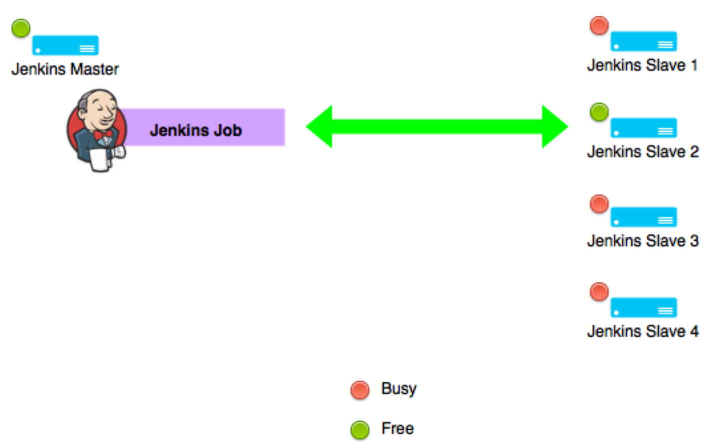
In the previous chapters, we saw how to configure Jenkins on node machines. These node machines act as Jenkins slaves. So far, we have configured Jenkins on two nodes, one for testing the code (the testing server) and the other to host the live application (the production server). However, we used the Jenkins master server to perform all our Jenkins builds.

Executing builds on the Jenkins master server may seem to be fine as long as you have sufficient hardware configuration for the Jenkins master server. Nevertheless, imagine a situation where the number of builds per day increases from single digit to multiple digits. What would happen to the Jenkins master server?

- The builds may execute slowly, one after the other, since everything is happening on a single machine, which is the Jenkins master server
- Total build time may increase due to CPU load, assuming we do not upgrade the Jenkins master server
- We may face disk space issues. As the number of builds per day increase, the size occupied by build logs and artifacts also increase exponentially

The preceding case becomes a reality if we use a single Jenkins master machine to perform all the builds. This is where distributed build architecture comes to the rescue.

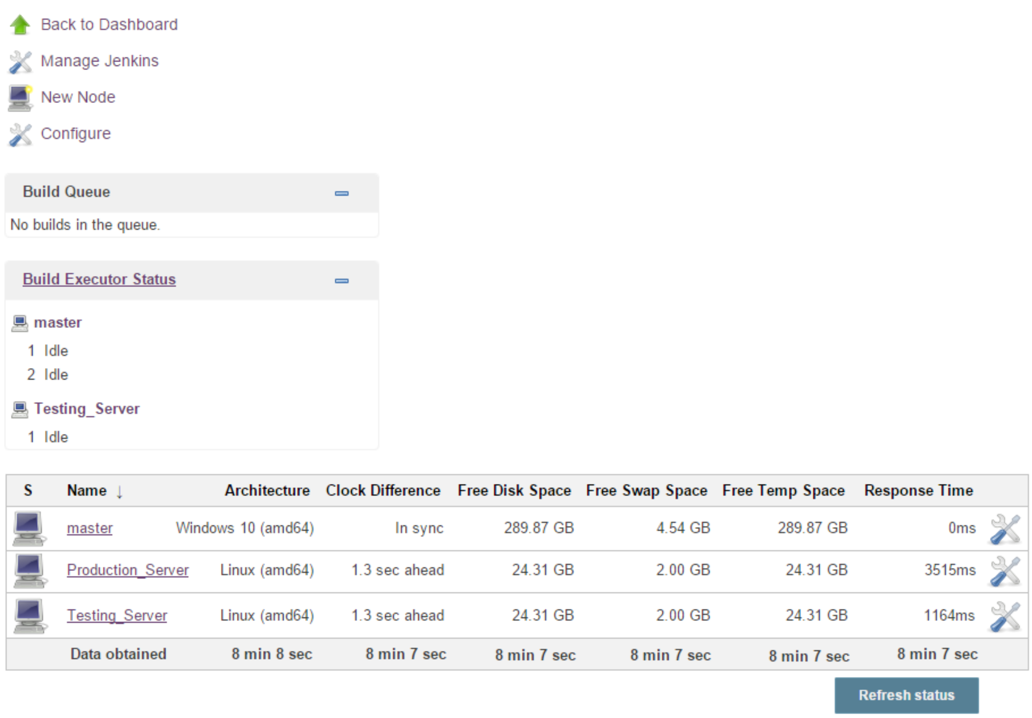
In the distributed build architecture, we configure Jenkins slaves on multiple node machines. The Jenkins jobs remain in the Jenkins master machine, but the build execution takes place on any one of the ideal Jenkins slaves:



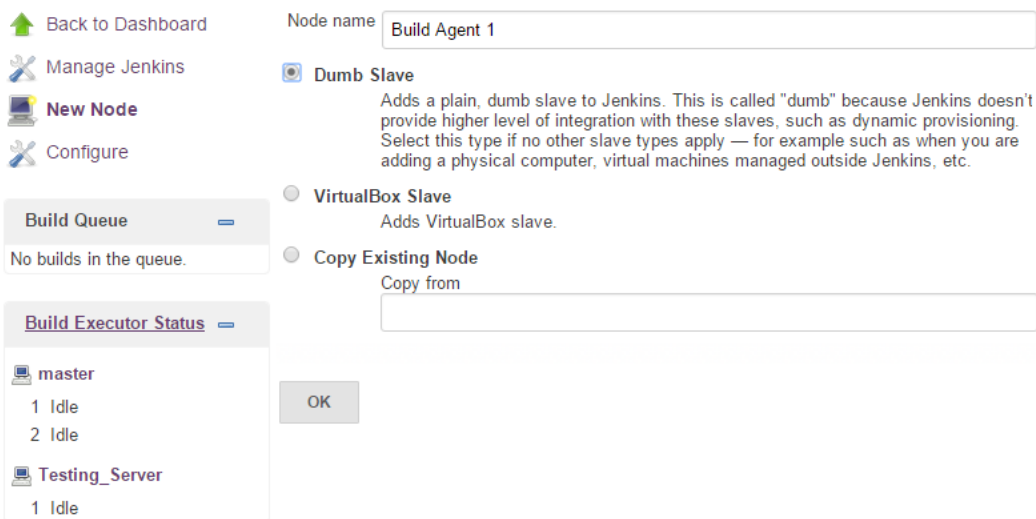
Configuring multiple build machines using Jenkins nodes

We will first configure Jenkins slaves on the node machines by following these steps. Later, we will modify the existing Jenkins job to harness the power of these slaves by performing builds on them:

1. Log in to the identified node machine. Open the Jenkins dashboard from the web browser using the link `http://<ip address>:8080/jenkins/`. Remember, you are accessing the Jenkins master from the node machine. The `<ip address>` is the IP of your Jenkins server.
2. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page. Make sure you have logged in as an admin in Jenkins.
3. Click on the **Manage Nodes** link. From the following screenshot, we can see that the master node (which is the Jenkins server along with one slave node) running on the testing server and one running on the production server are listed:

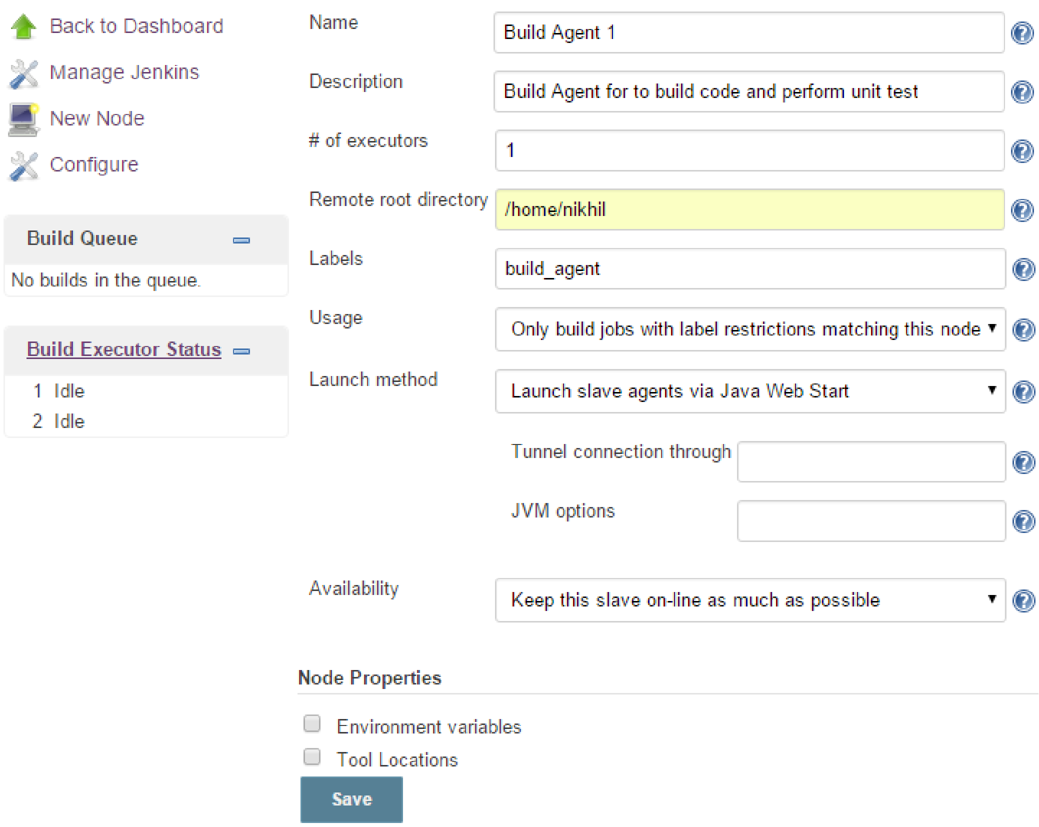


4. Click on the **New Node** button from the left-hand side panel. Name the new node Build Agent 1 and select the option **Dumb Slave**. Click on the **OK** button to proceed:

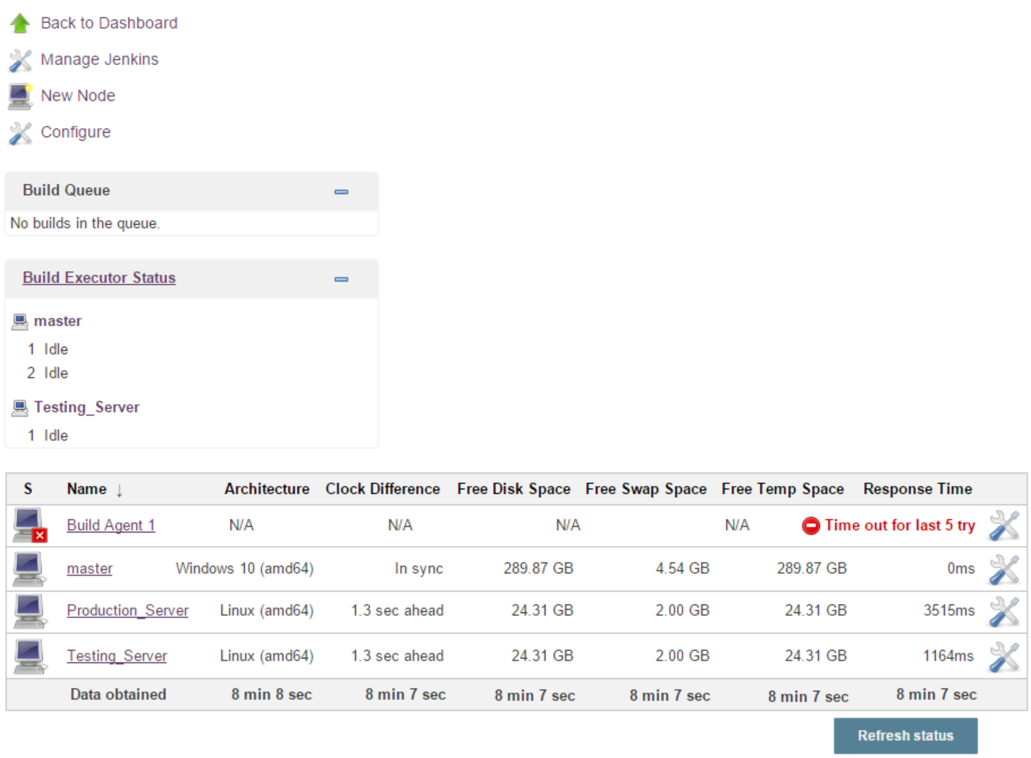


5. Add some description as shown in the next screenshot. The **Remote root directory** value should be the local user account on the production server. It should be /home/<user>. The **Labels** field is extremely important; add build_agent as the value.

6. The **Launch method** should be **Launch slave agents via Java Web Start**.



7. Click on the **Save** button. As you can see from the following screenshot, the Jenkins slave on the node agent has been configured but it's not yet running:

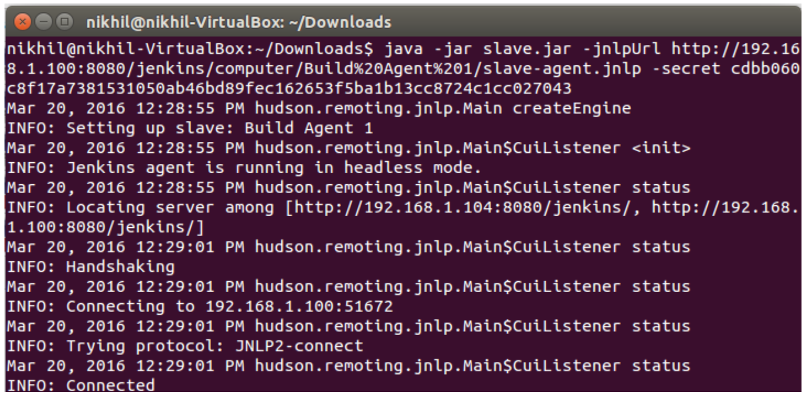


8. Click on the **Build Agent 1** link from the list of nodes. You will see something like this:

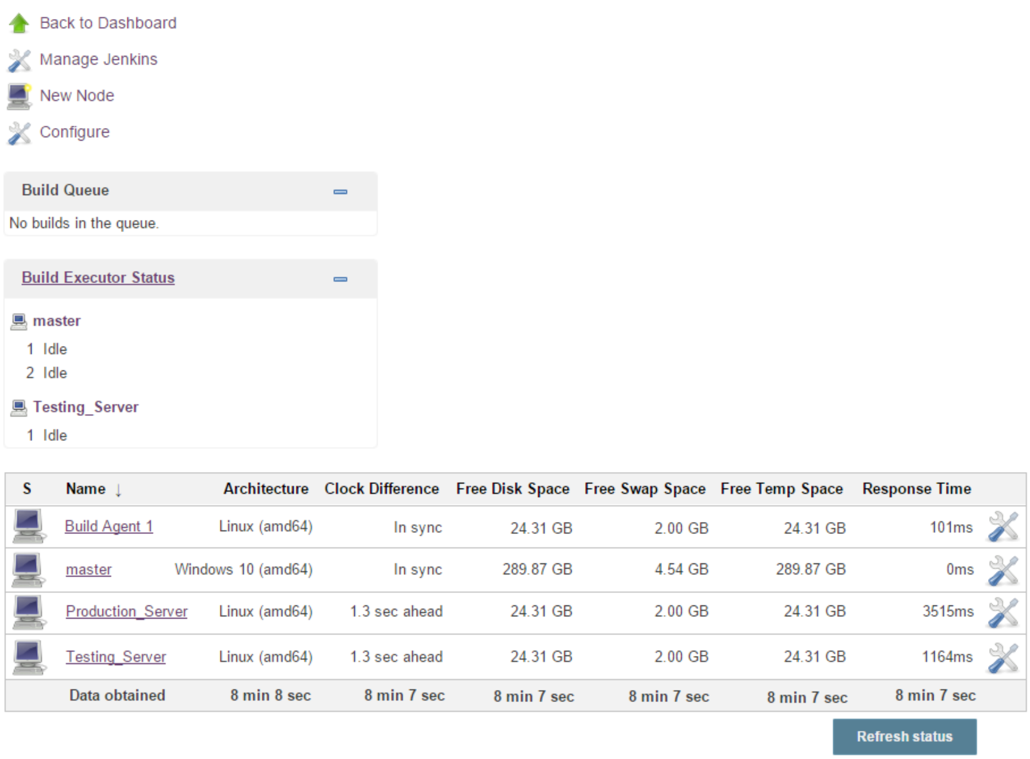
9. Either you can click on the **Launch** button in orange or you can execute the following long command from the terminal.
10. If you choose the latter option, then download the `slave.jar` file mentioned in the command by clicking on it. It will download the file to `/home/<user>/Downloads/`.
11. Execute the following commands in sequence:

```
cd Downloads

java -jar slave.jar -jnlpUrl http://192.168.1.104:8080/
jenkins/computer/Build%20Agent%201/slave-agent.jnlp -secret
59ea94be5288322fe7bc1e07c4ef6b9d8329764a4730a240fb6fb0dd47e1784a
```

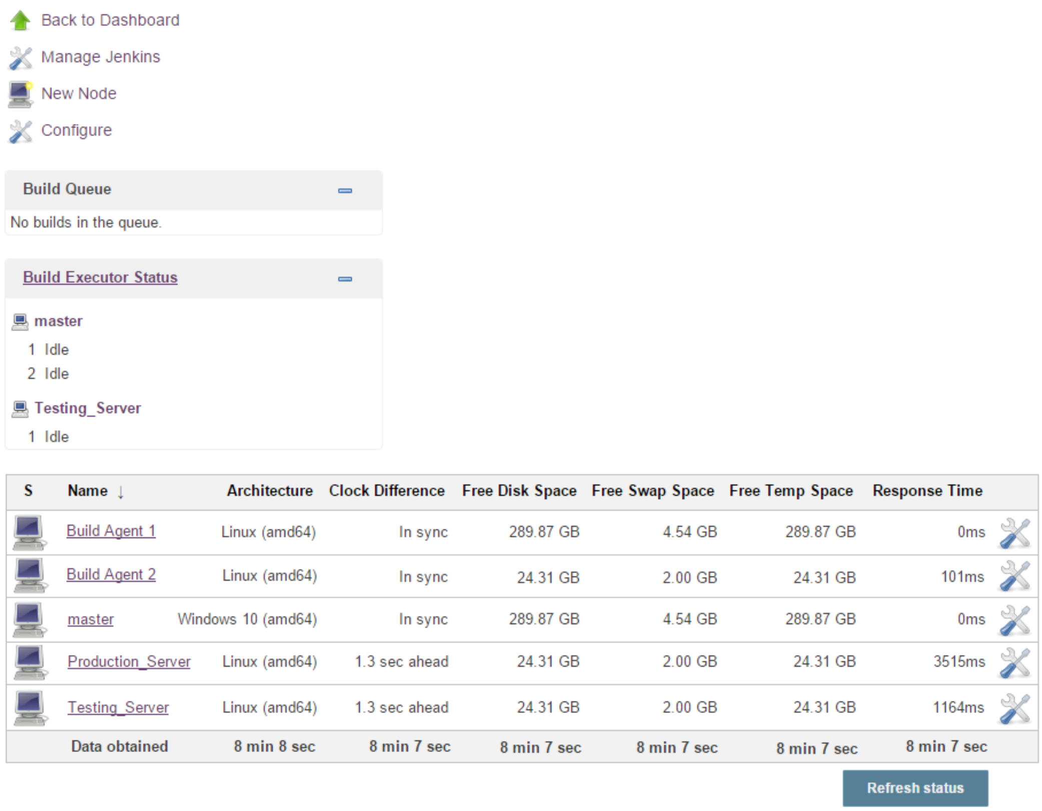
> The preceding command is machine specific. Do not copy-paste and execute the same. Execute the command that appears on your screen.



12. The node on **Build Agent 1** is up and running:



13. Identify another spare machine and configure a Jenkins slave on it in a similar fashion and name it `Build Agent 2`. However, while configuring this new build agent, label it as `build_agent`.
14. Finally, everything should look like this:



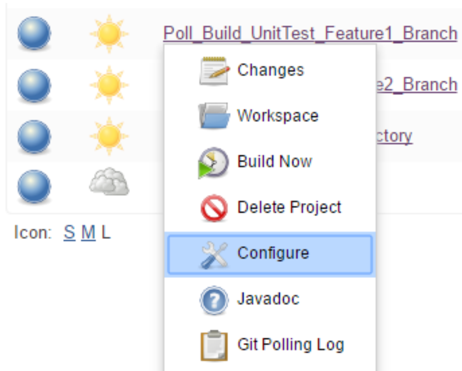
You can configure as many build agents as required. Nevertheless, keep the label same across all build agents.

Modifying the Jenkins job

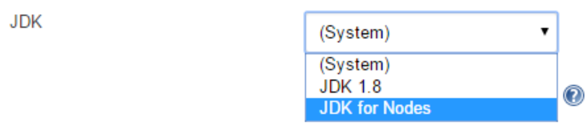
Let's experiment with the Jenkins job to `Poll_Build_UnitTest_Feature1_Branch`:

1. From the Jenkins dashboard right click on `Poll_Build_UnitTest_Feature1_Branch` Job.

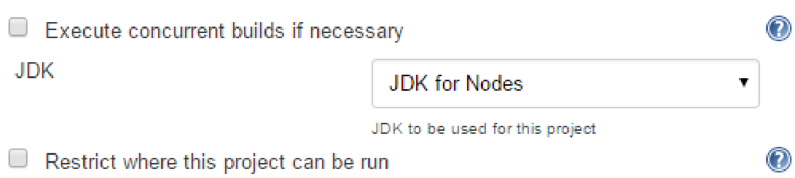
2. Click on the **Configure** link from the menu:



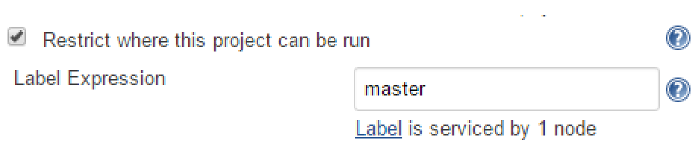
3. Scroll down until you see the **JDK** section.
4. Click on the drop-down menu and choose the option **JDK for Nodes**:



5. Things should look like this:

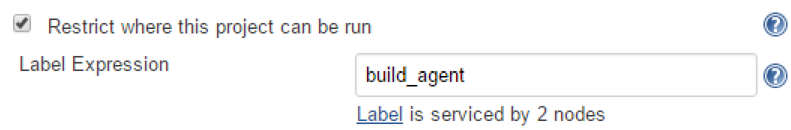


6. Under the JDK option, you will see the setting **Restrict where this project can be run**. Right now, it's configured to run on the master:



7. Change the value of the **Label Expression** field from **master** to `build_agent`.

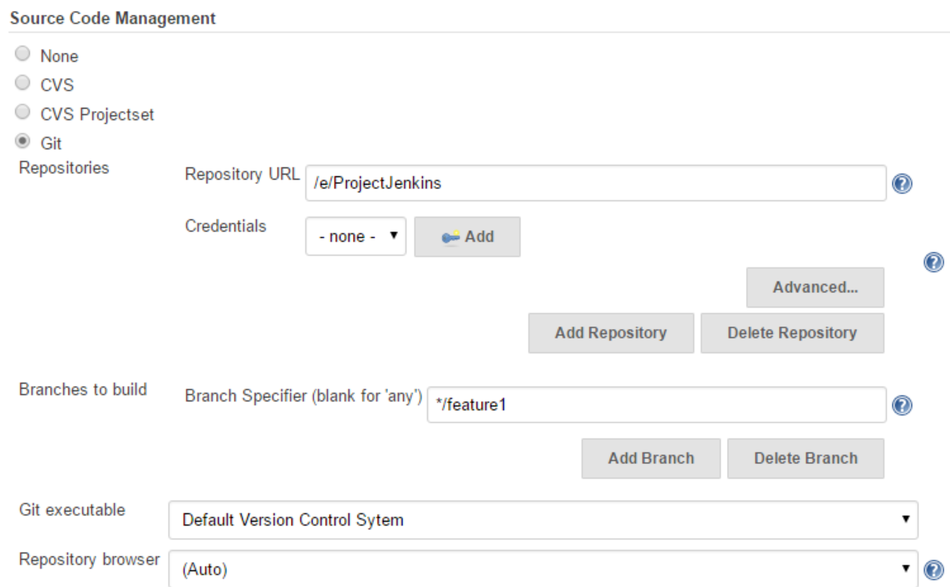
8. As you can see, the moment you add the label, a notification appears saying **Label is serviced by 2 nodes**. This is because we have configured two node machines with the label **build_agent**:



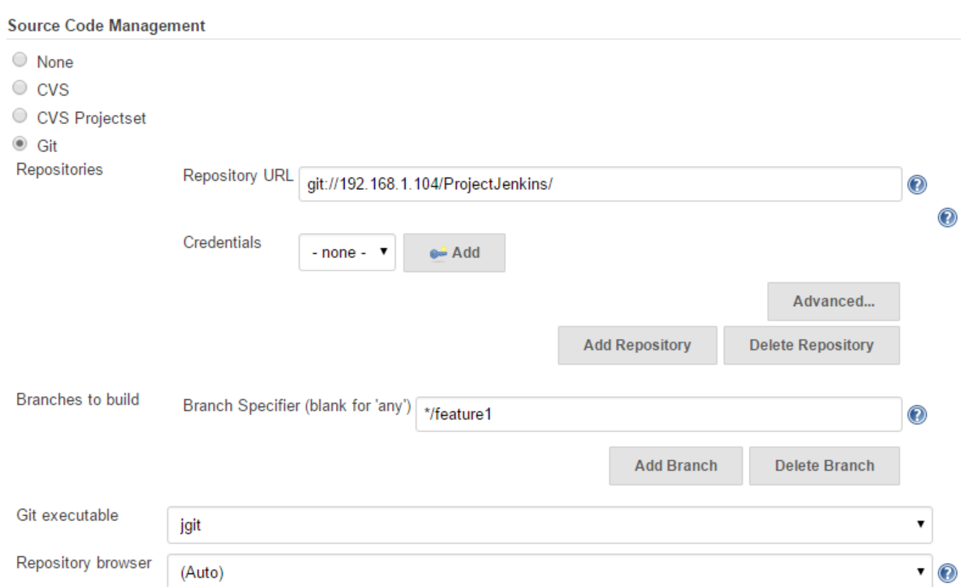
9. Click on the link **Label**.
10. You will see the following page. The label **build_agent** is mapped to the nodes **Build Agent 1** and **Build Agent 2**:

11. Scroll down to the **Source Code Management** section.

12. Here's the current configuration:

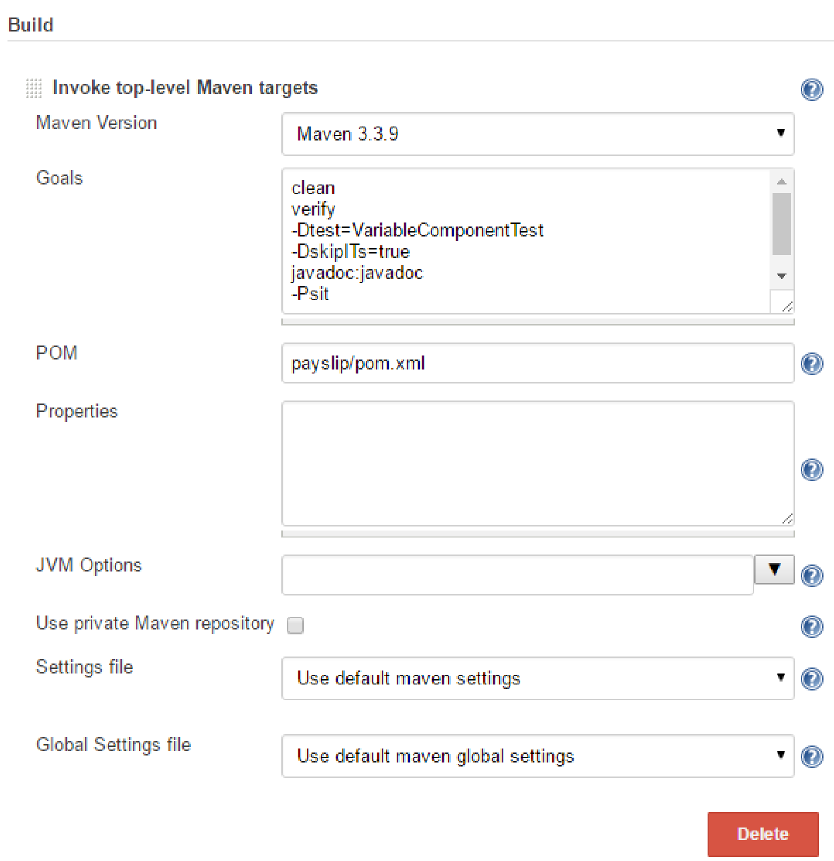


13. Modify the **Repository URL**. It can be a GitHub repository or a repository on a Git server. In our case, it's git://<ip address>/ProjectJenkins/, where <ip address> is the Jenkins server IP:

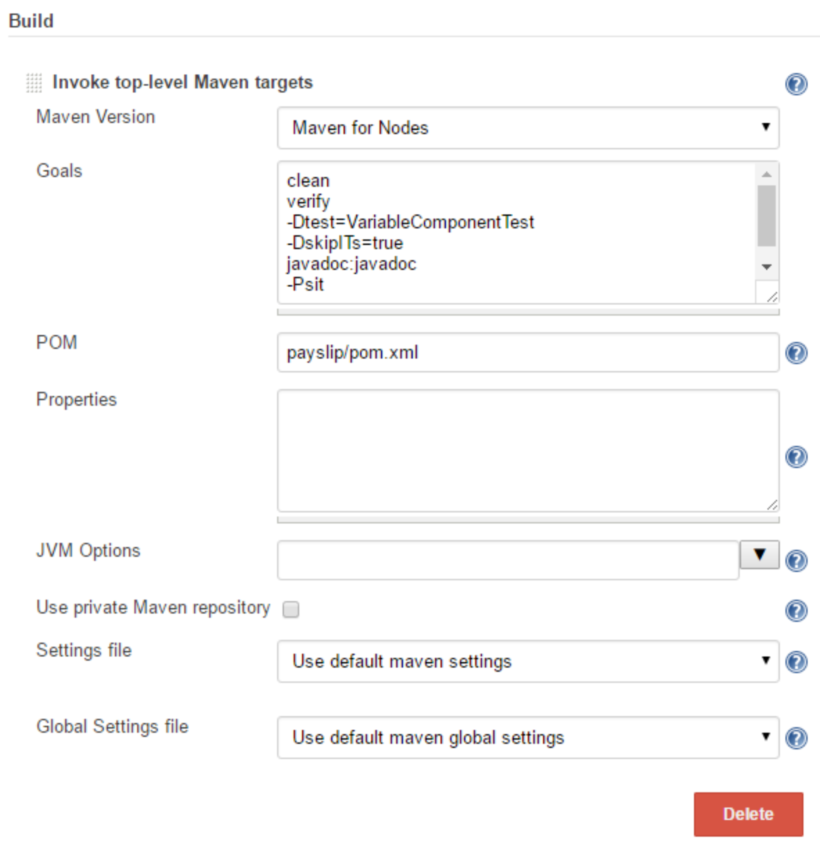


14. Leave the **Poll SCM** option at its default value:

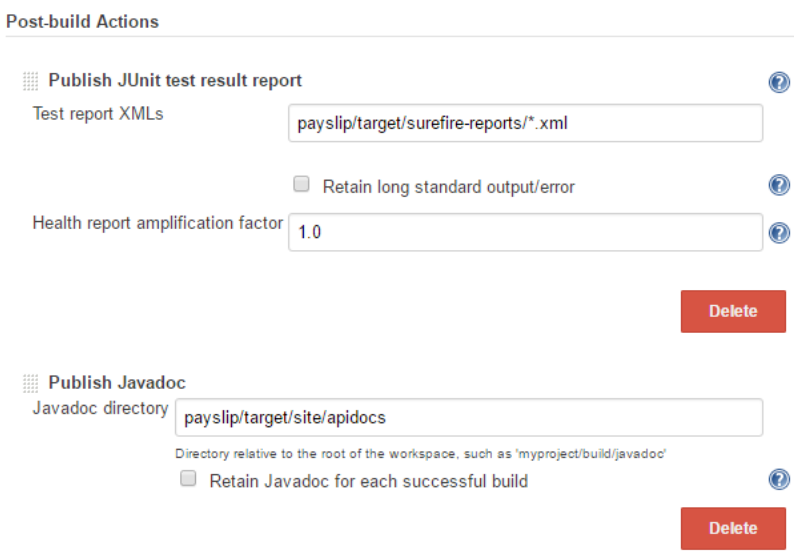
15. Scroll down to the **Build** section. The current configuration looks like this:



16. Modify the **Maven Version** from **Maven 3.3.9** to **Maven for Nodes** from the drop-down menu, as shown in the following screenshot:



17. Leave the **Post-build Action** and the rest of the configuration at their default values:

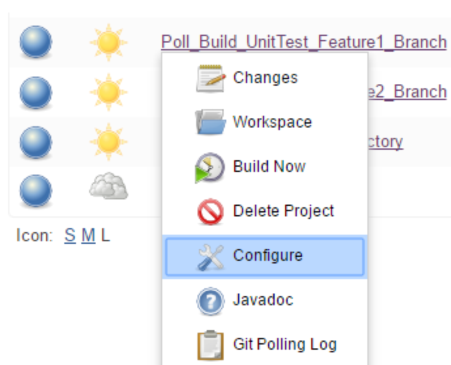


18. Save the Jenkins job by clicking on the **Save** button.

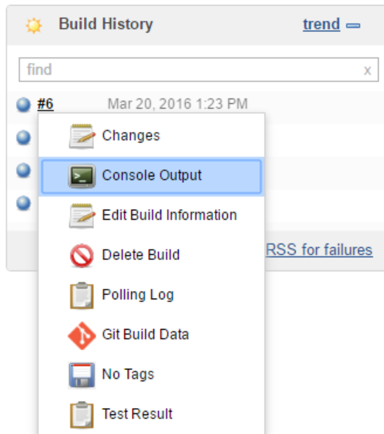
Running a build

To run a build, perform the following steps:

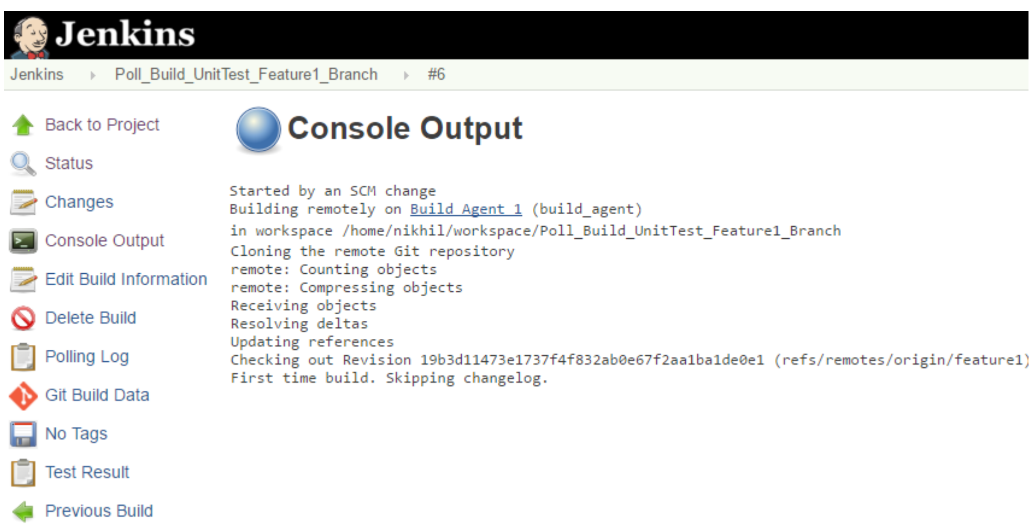
1. From the Jenkins dashboard, right click on the `Poll_Build_UnitTest_Feature1_Branch` job.
2. Click on the **Build Now** link from the menu:



3. Once the build is running, click on the job link from the dashboard.
4. The build must have been completed by now. On the job page, under the **Build History** section, right-click on the build and select **Console Output** from the menu. This is shown in the following screenshot:



5. Once you get to the logs, you will notice that the build is running on **Build Agent 1**:



6. Ideally, the Jenkins job randomly chooses from the list of available node agents with the label **build_agent**.

Version control Jenkins configuration

In the first few chapters, we saw how to take Jenkins backup. We did this in two ways—either by creating a Jenkins job that regularly takes Jenkins backup of the whole Jenkins folder, or by using the Jenkins backup and restore plugin.

This in itself is a version control, as we are saving the whole Jenkins configurations at a desired point of time and at regular intervals, or whenever we do a major Jenkins configuration. However, this is not the best way to record every miniscule change in the Jenkins configuration separately.

Nevertheless, Jenkins backup is the best way to restore Jenkins during a catastrophic event where the whole server goes haywire.

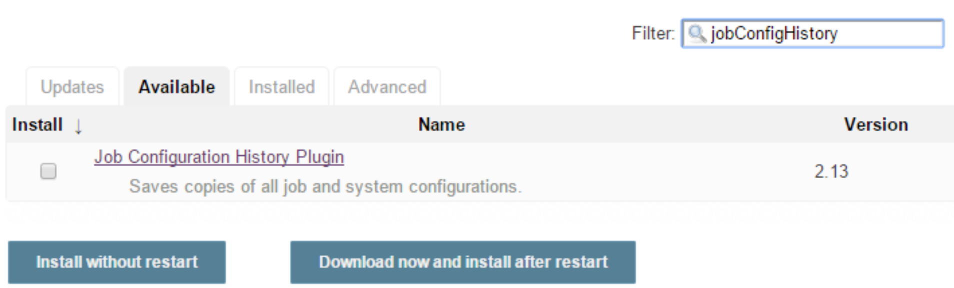
Let's see how to version control Jenkins configuration using a plugin.

Using the jobConfigHistory plugin

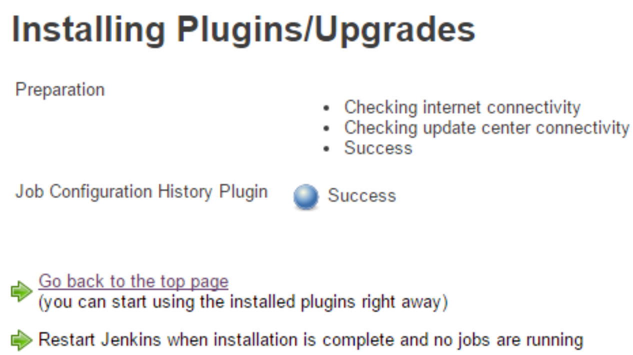
This plugin saves a copy of the configuration file of a job (`config.xml`) for every change made and of the system configuration.

It is also possible to get a side-by-side view of the differences between two configurations and to restore an old version of a job's configuration. However, this option is available only for jobs and not for Jenkins system changes.

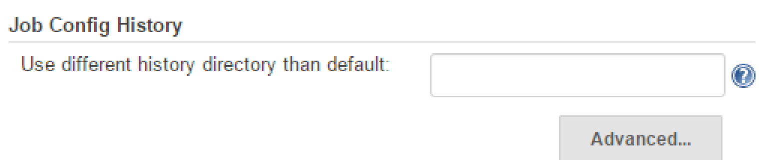
1. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `jobConfigHistory` in the search box.
4. Select **Job Configuration History Plugin** from the list and click on the **Install without restart** button:



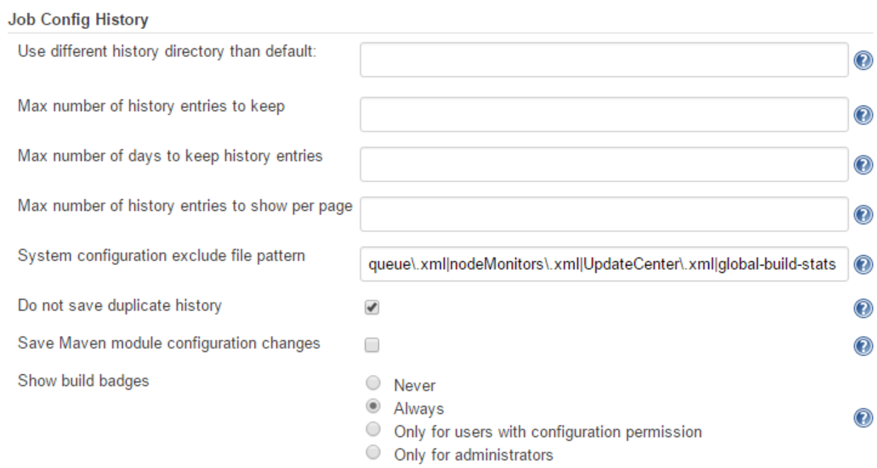
5. The download and installation of the plugin starts automatically:



6. Go to the **Configure System** link from the **Manage Jenkins** page.
7. Scroll down until you see the **Job Config History** section:



8. Click on the **Advanced...** button.
9. The default directory for storing history information is `JENKINS_HOME/config-history`. If you want to use a different location, you can enter its path in the **Use different history directory than default** field.
10. Either an absolute or a relative path may be specified. If a relative path is entered, it will be created below `JENKINS_HOME`. If an absolute path is entered, the value will be used directly.
11. Enter the maximum number of history entries to keep in the **Max number of history entries to keep** field. Leave it blank to keep all entries.
12. Enter the maximum number of days that history entries should be kept in the **Max number of days to keep history entries** field. Leave it blank to keep all entries:



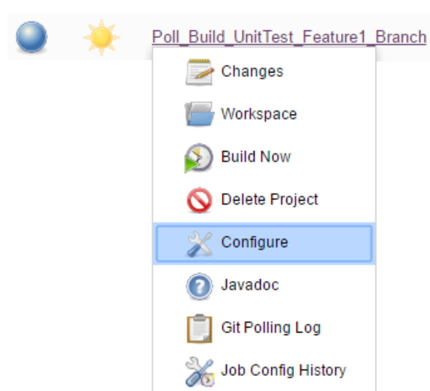
13. Save the configuration by clicking on the **Save** button.

> **Warning!**
> If this path gets changed, existing history files will not be found by the plugin any longer. If you still want to have them listed, you must move them manually to the new root history folder.

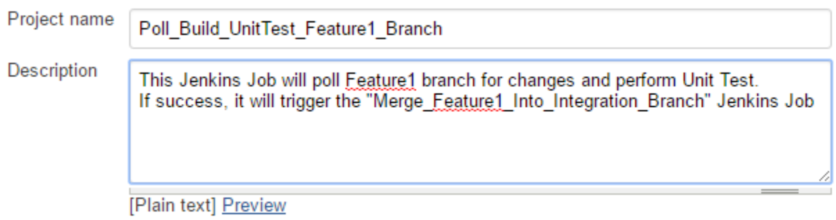
Let's make some changes

Now that we have configured the jobConfigHistory plugin. Let's make some changes to see how it works:

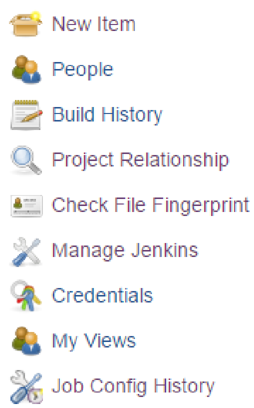
1. From the Jenkins dashboard, right-click on any of the Jenkins jobs and click on **Configure**:



2. Scroll to the **Description** section and add some text, or make any modification you want:

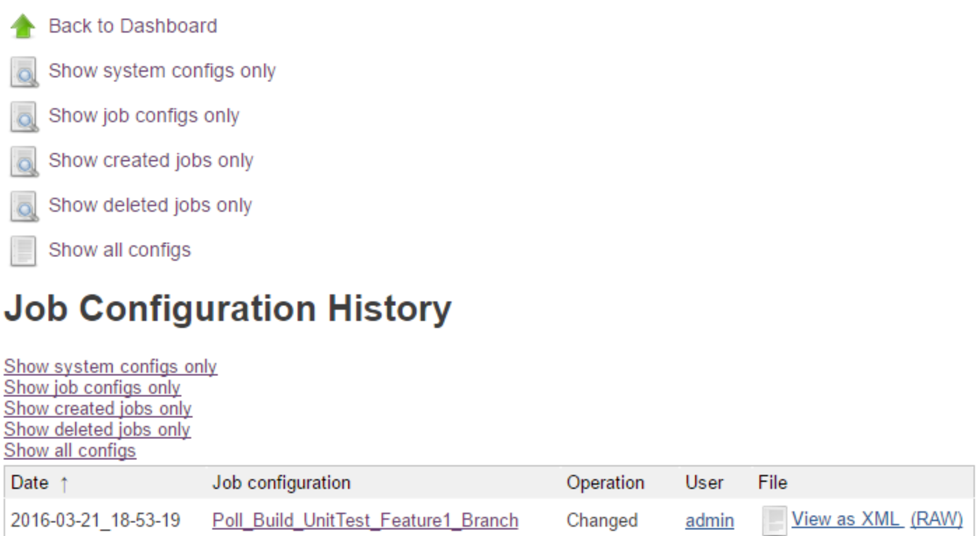


3. Save the Jenkins job by clicking on the **Save** button.
4. From the Jenkins job page, click on the **Job Config History** link:

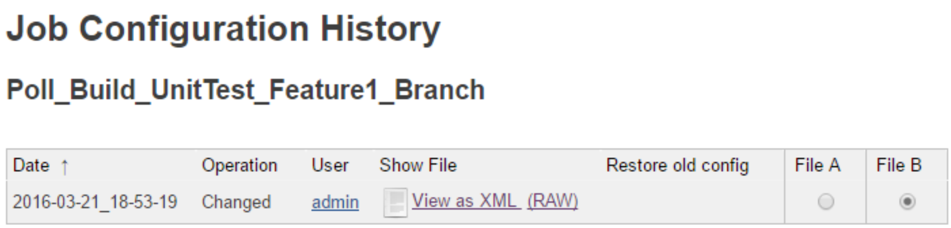


5. We can already see some changes listed. However, this is the Jenkins global configuration change:

6. Click on the **Show job configs only** link to see changes made to the Jenkins jobs.
7. You will be taken to the following page, where you will be able to see the Jenkins jobs that have been modified:



8. Click on the Jenkins job link and you will see the changes made to it. So far, we have only one change:



Later in this chapter, we will add more configurations to Jenkins. The preceding list will build up and some new features will come up.

Auditing in Jenkins

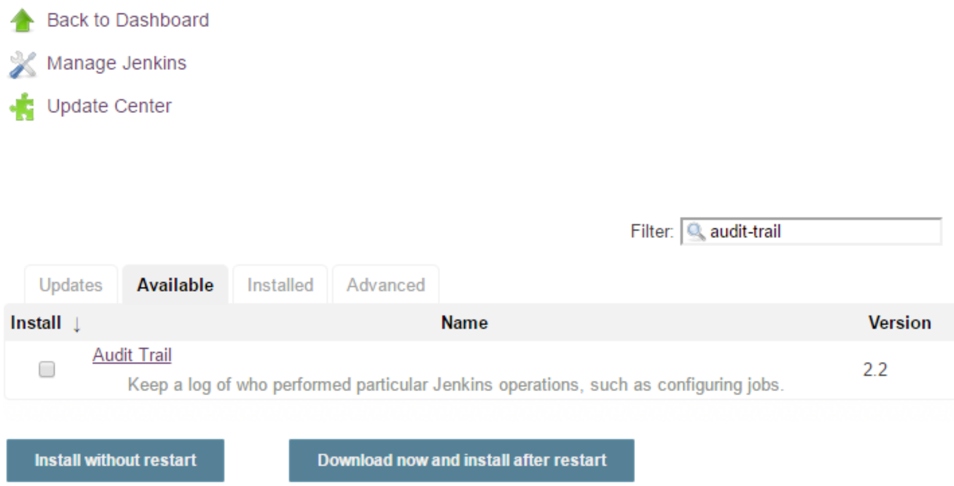
In the previous section, we saw how to use the jobConfigHistory plugin to record and version control changes made to Jenkins.

In this section, we cover how to audit Jenkins using the Audit Trail plugin.

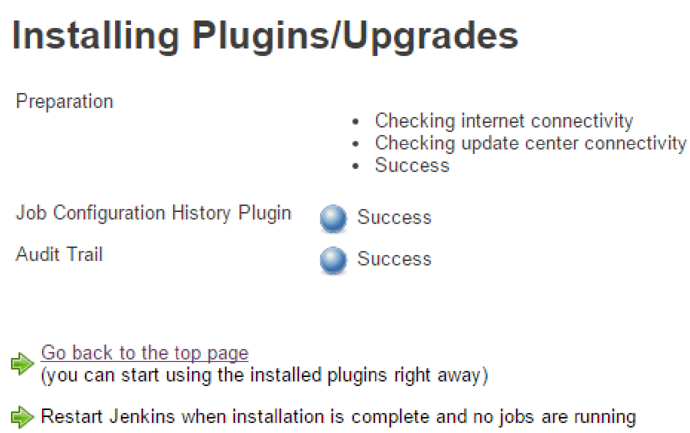
Using the Audit Trail plugin

Perform the following steps to audit Jenkins using the Audit Trail plugin:

1. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `audit-trail` in the search box.
4. Select the **Audit trail** from the list and click on the **Install without restart** button:

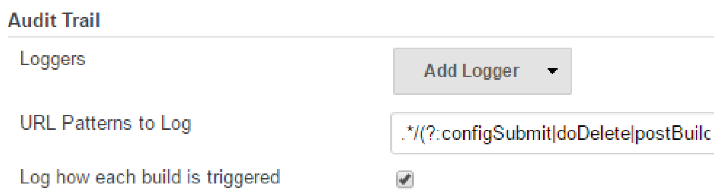


5. The download and installation of the plugin starts automatically:

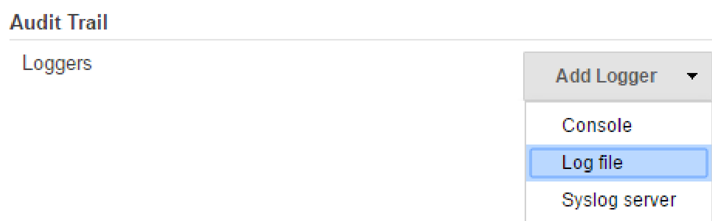


6. Go to the **Configure System** link from the **Manage Jenkins** page.

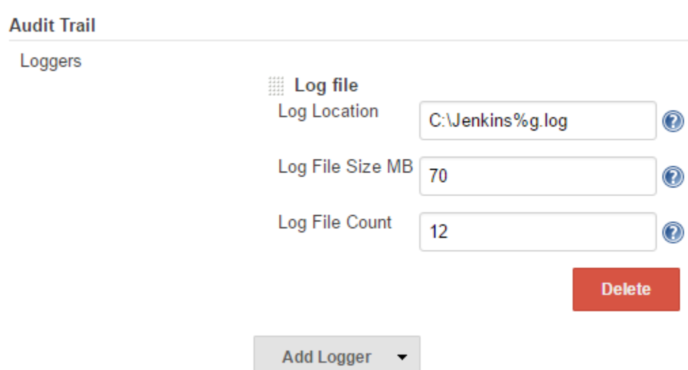
7. Scroll down until you see the **Audit Trail** section, as shown in the following screenshot:



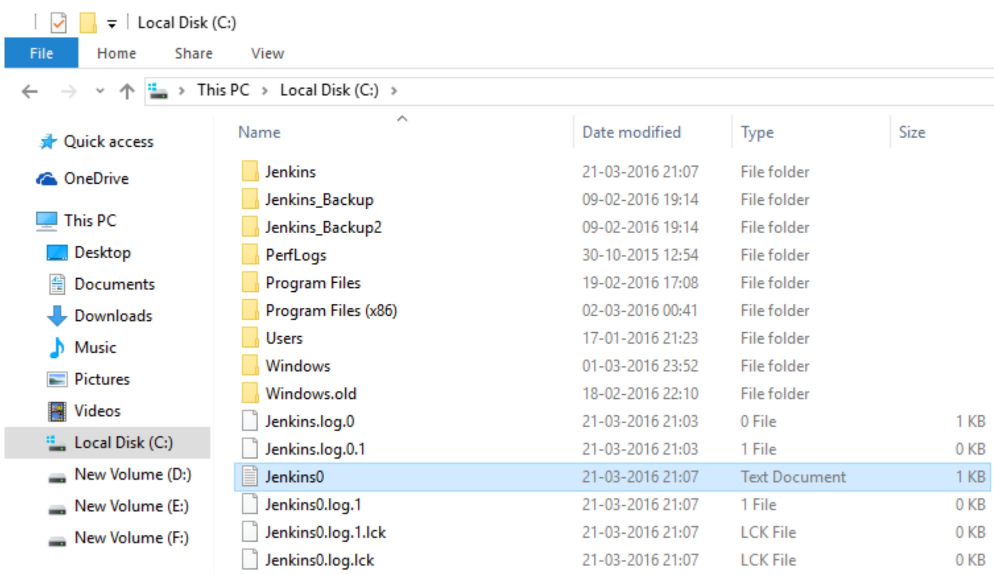
8. Click on the **Add logger** button and select **Log file** from the options. In this way, we will save all the audit logs to a log file:



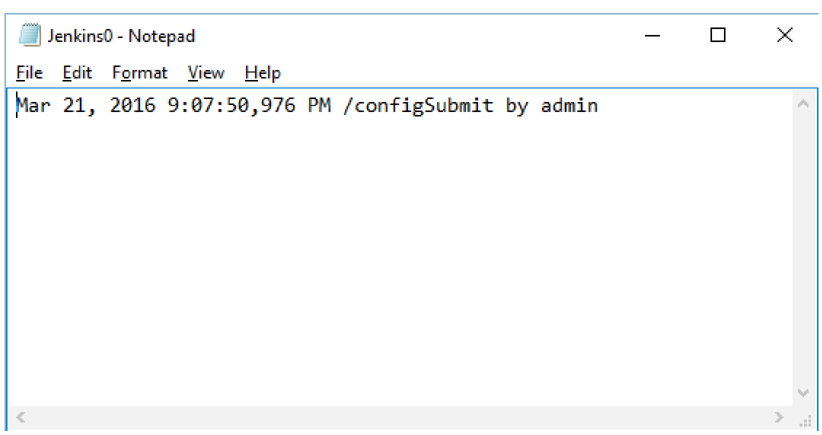
9. The moment you select the **Log file** option, a few settings appear.
10. Add the **Log Location** as shown in the next screenshot. `%g` is the date time stamp.
11. If a log file grows beyond the specified limit configured in the **Log File Size MB** field, then a new log file is created. The number of log files to keep can be configured using the **Log File Count** field:



12. Save the configuration by clicking on the **Save** Button.
13. Navigate to the location where the log files get created. In our case, it's `C:\Jenkins`.
14. You can see a file named `Jenkins0`:



15. Open it to view the content.
16. We can see some data inside it. The user admin has performed a `/configSubmit` operation:



Notifications

Notification forms an important part of the Continuous Integration and Continuous Delivery process. Breaking tasks into multiple Jenkins jobs and having e-mail notifications for each is the best way to act quickly.

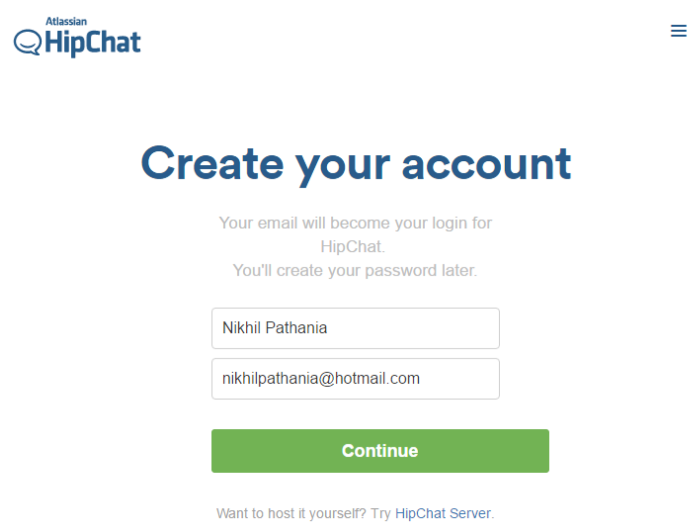
We have already seen e-mail notifications in the previous chapters. However, from the past few years, defect tracking tools and team collaboration tools are gaining momentum, for example, Asana, Slack, Trello, and HipChat.

In this section, let's see how to configure Jenkins with HipChat to get continuous notifications.

Installing HipChat

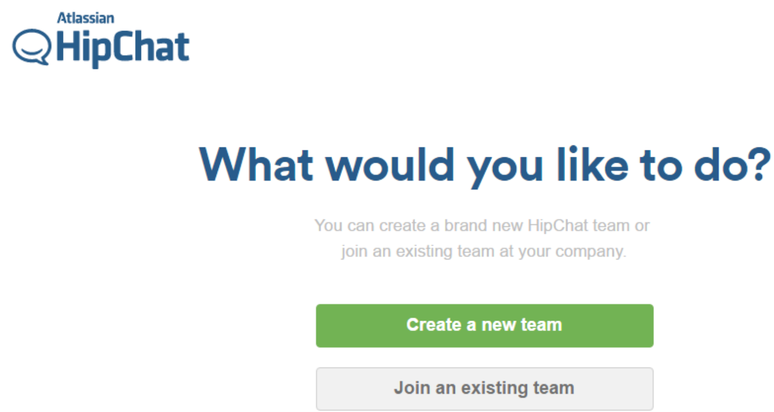
Perform the following steps to install HipChat:

1. From the Atlassian HipChat website, create a new account:

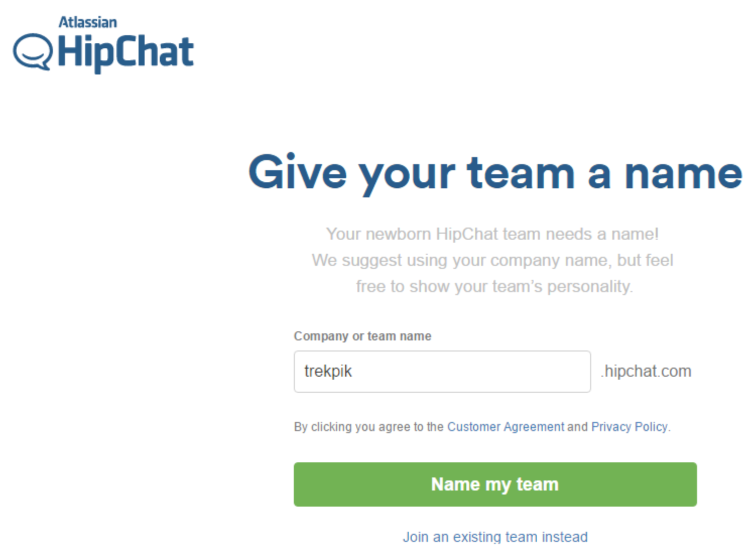


2. If your organization already has a team, you may click on the **Join an existing team** button.

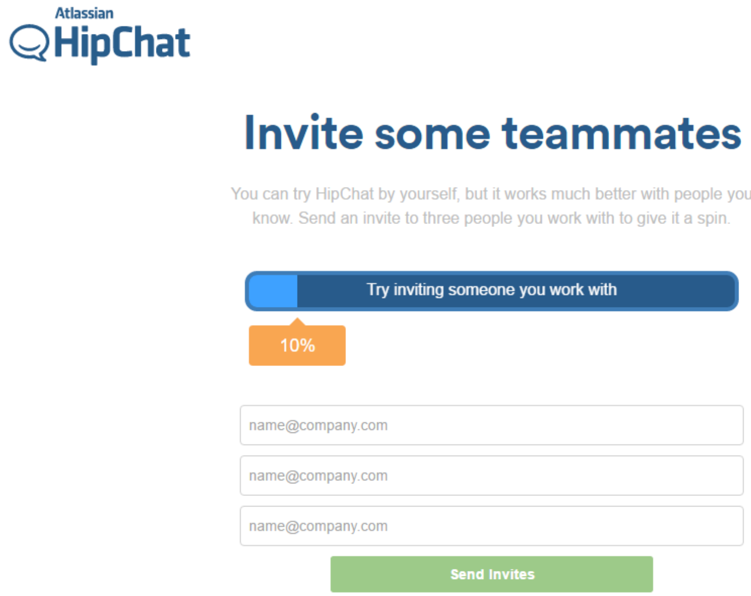
3. If that is not the case, then click on the **Create a new team** button:



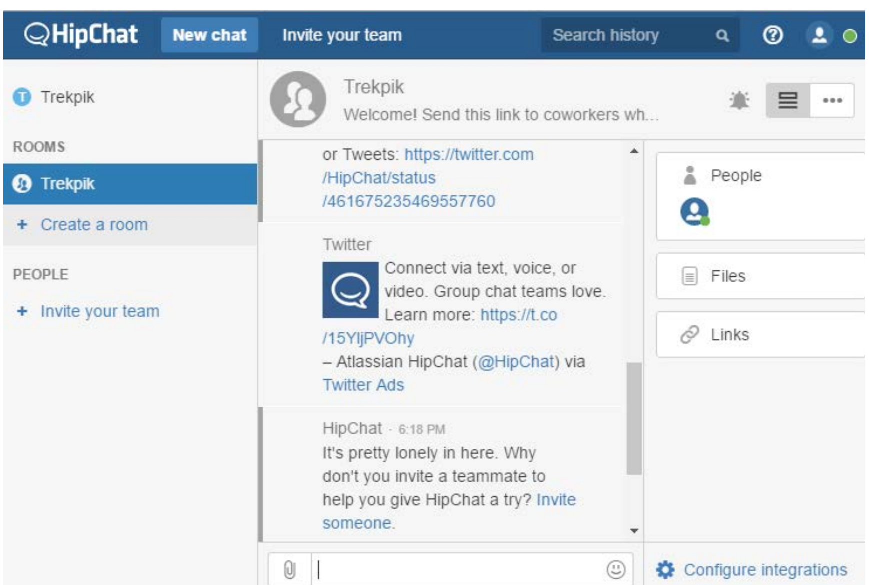
4. Provide a name for your team:



5. Skip this step if you don't want to invite people:



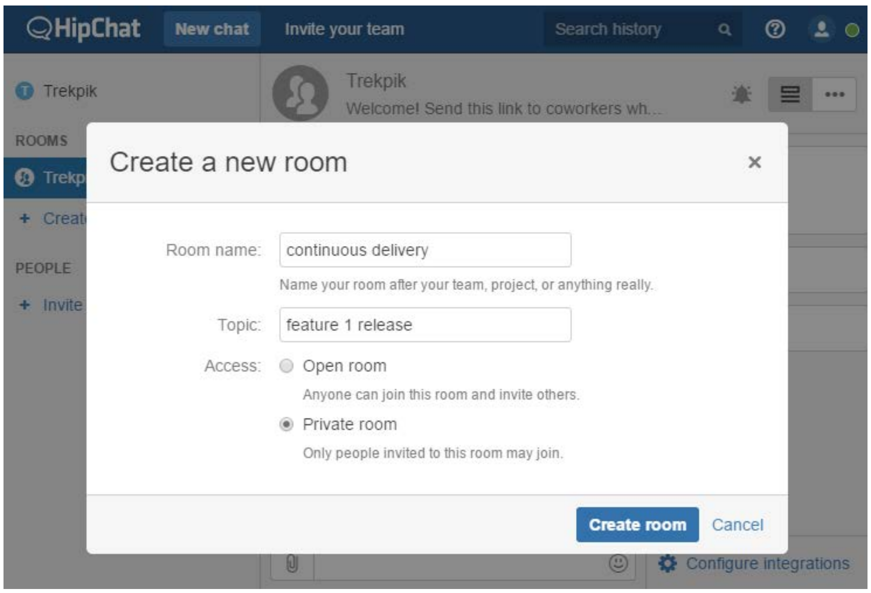
6. This is what your HipChat dashboard page looks like:



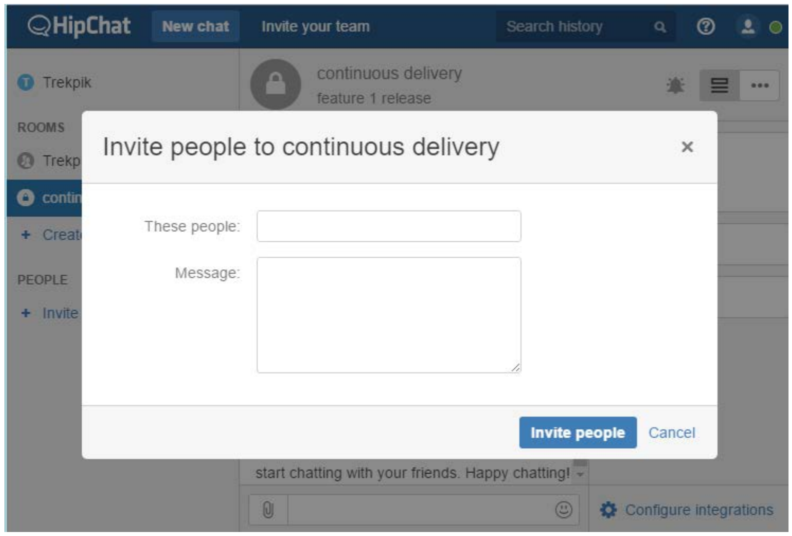
Creating a room or discussion forum

Perform the following steps to create a room or discussion forum:

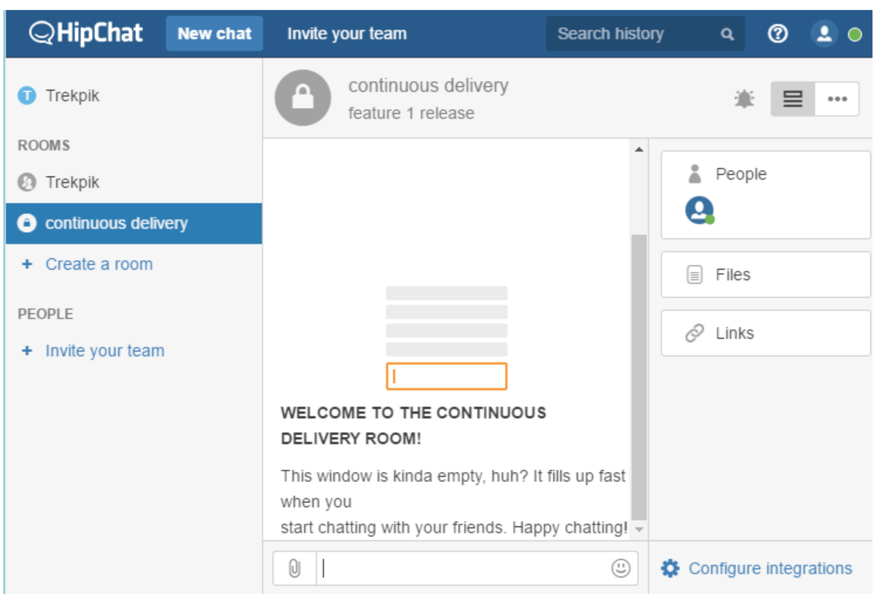
1. Click on the **Create a room** link.
2. Give your room a meaningful name and select a topic name.
3. You also have the option to make the room public or private. Choose appropriately.
4. Click on the **Create room** button when done:



5. Next, you will be asked to invite people. You can skip this as we can do this later. If you want to invite people, click on the **Invite people** button:



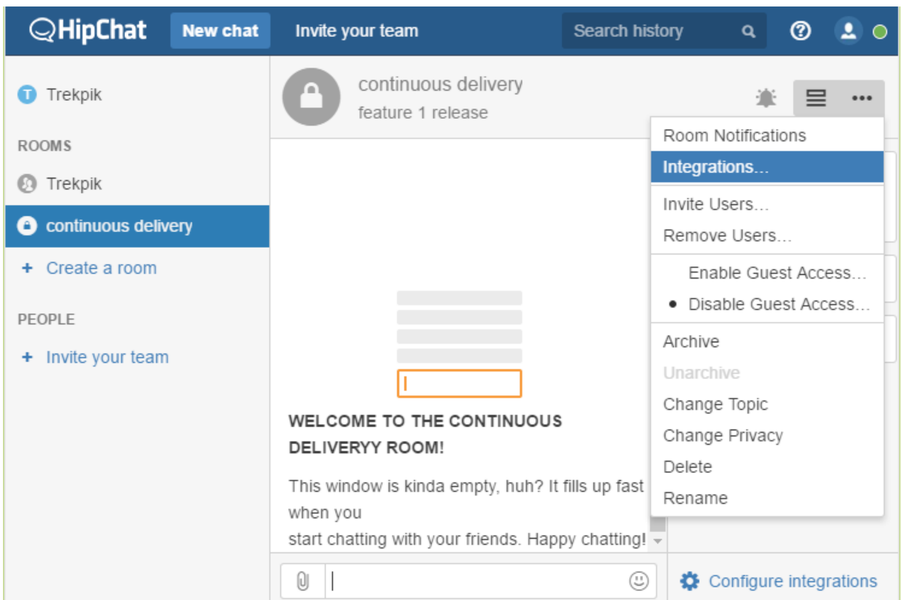
6. Your newly created room will be listed on the left-hand side bar:



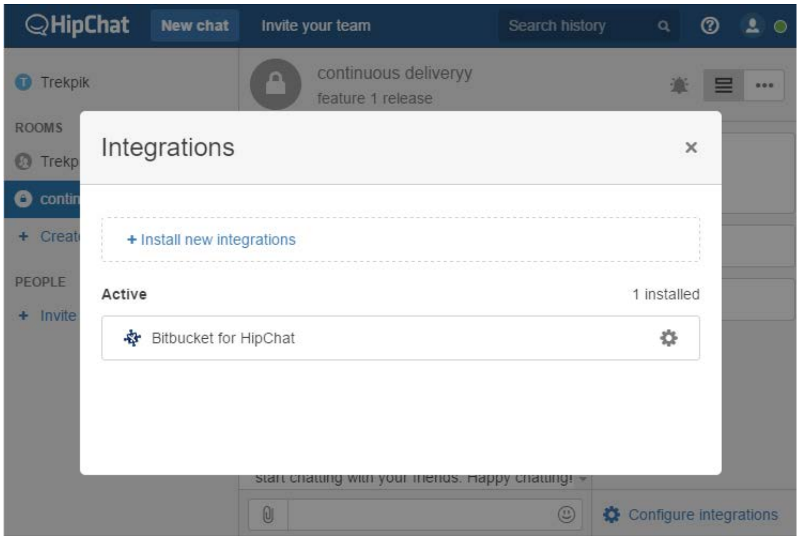
Integrating HipChat with Jenkins

To integrate HipChat with Jenkins, perform the following steps:

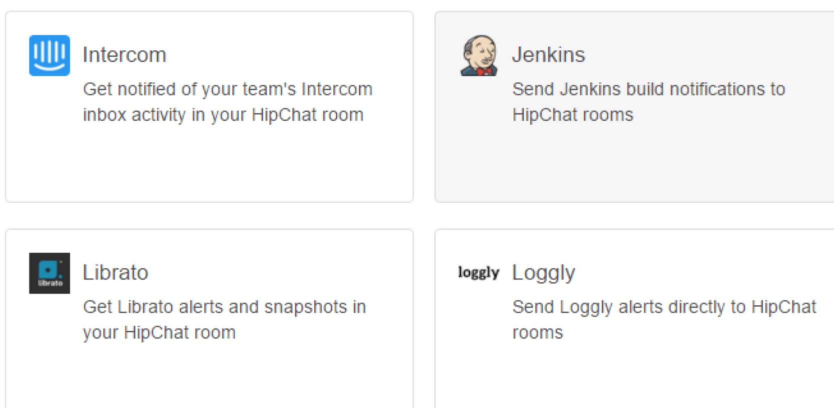
1. From the HipChat dashboard, click on the settings button.
2. Click on the **Integrations...** link, as shown in the following screenshot:



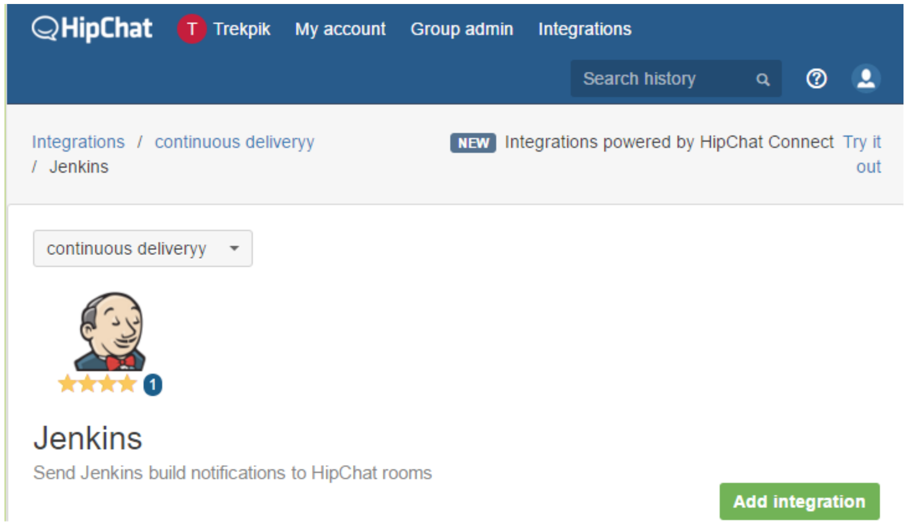
3. Click on the **Install new integrations** link:



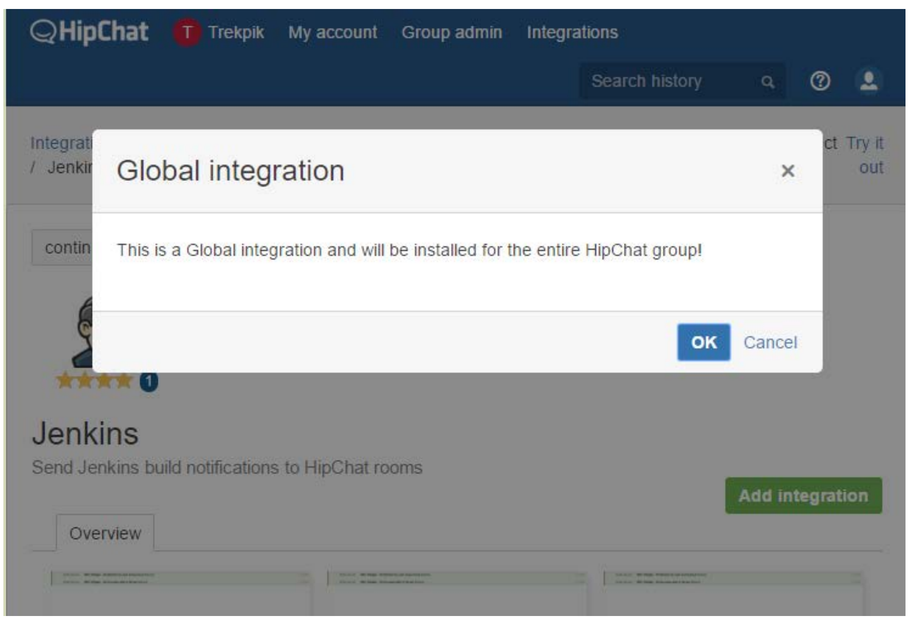
4. You will end up on a page with a long list of tools with which you can integrate HipChat. Search for Jenkins by scrolling down the list.
5. Once you find **Jenkins**, click on the link:



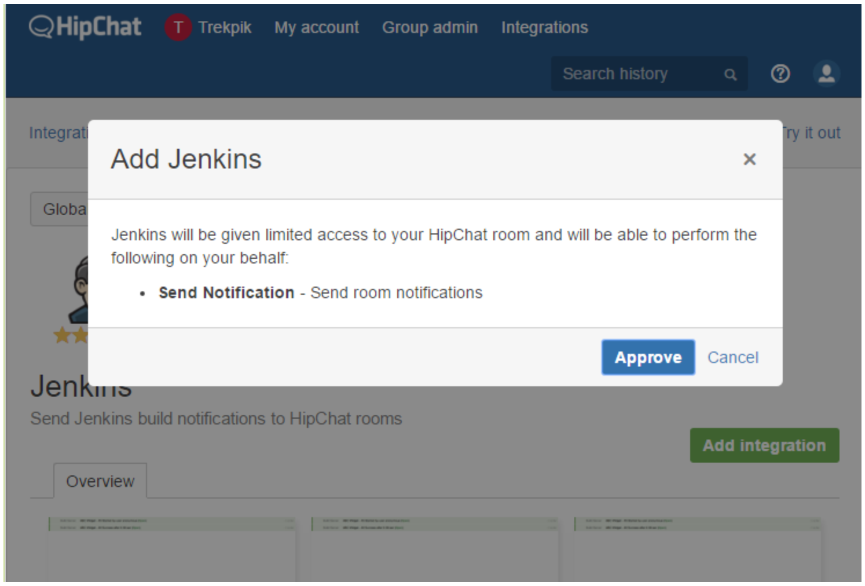
6. This is what you will see. Click on the **Add integration** button:



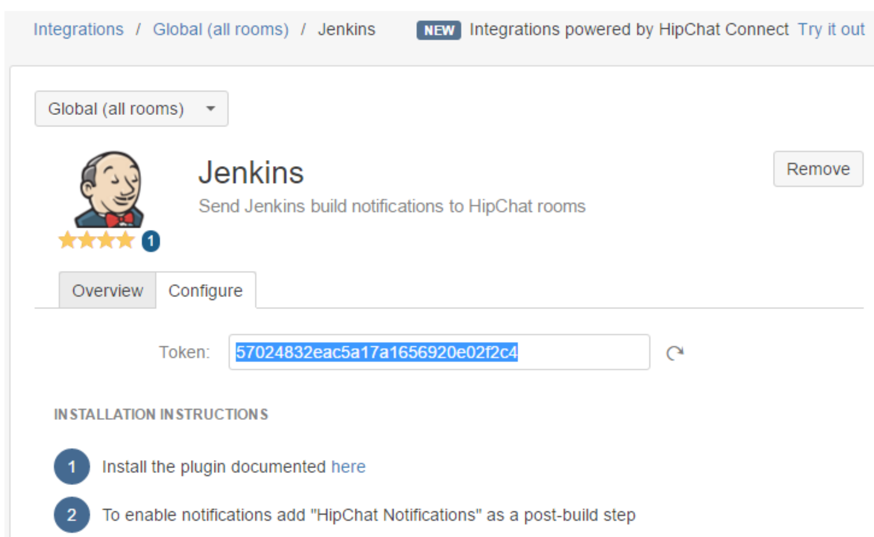
7. You will see the following notification. Click on **OK**:



8. Click on the **Approve** button to proceed:



9. You will land up on the next page. This is an important step.
10. Under the **Configure** tab, you will see a token key. It can be regenerated by clicking on the refresh button right beside it.
11. Copy it and make a note of it. We will need it later while configuring Jenkins:

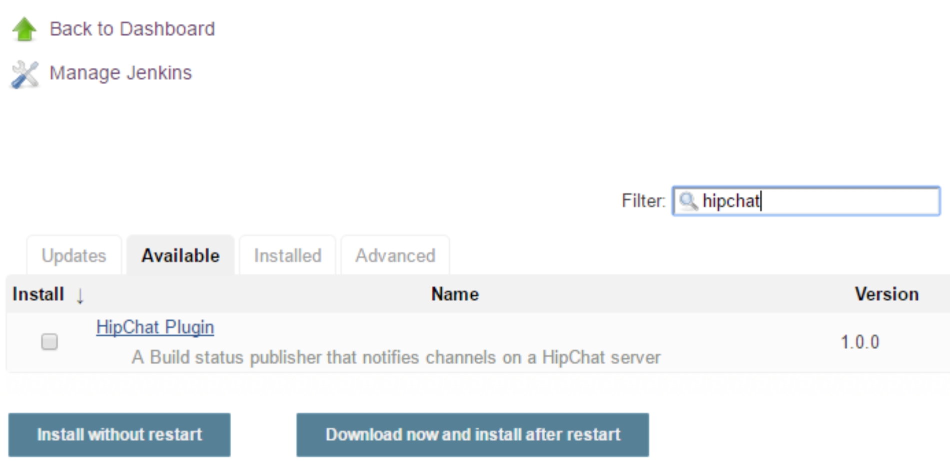


Installing the HipChat plugin

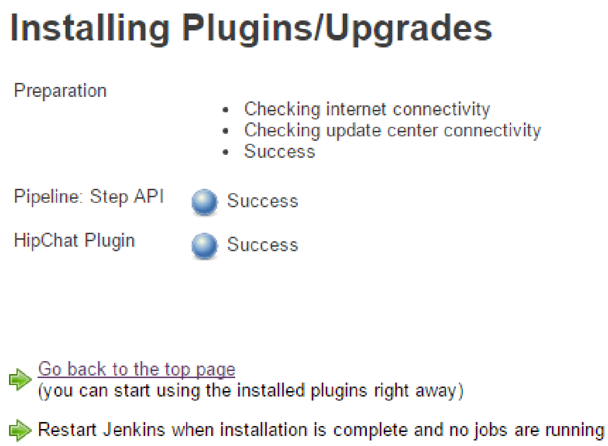
Now come to your Jenkins server and follow these steps:

1. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page:
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `hipchat` in the search box.
4. Select the **HipChat Plugin** from the list and click on the **Install without restart** button:

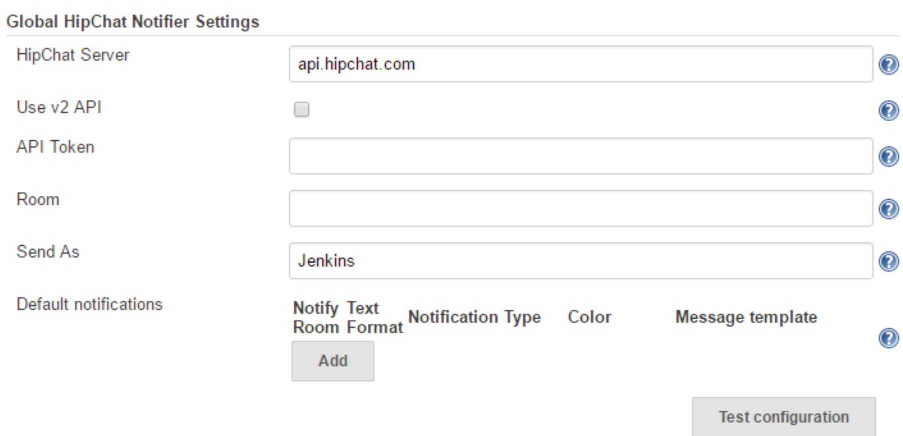
> **Warning!**
> This plugin requires dependent plugins that are built for Jenkins 1.642.1 or above. The dependent plugins may or may not work in your Jenkins, and consequently this plugin may or may not work in your Jenkins.



5. The download and installation of the plugin starts automatically. You can see the **HipChat Plugin** has some dependencies that get downloaded and installed:

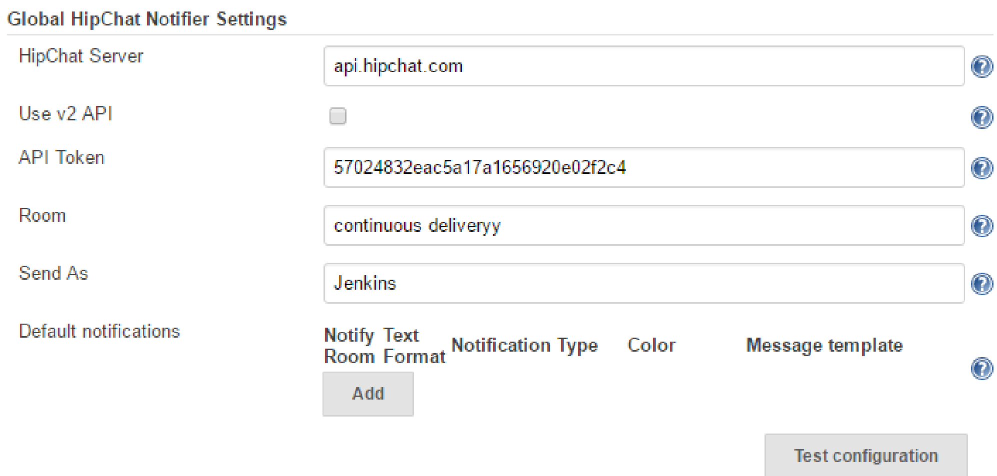


6. Go to the **Configure System** link from the **Manage Jenkins** page.
7. Scroll down until you see the **Global HipChat Notifier Settings** section:

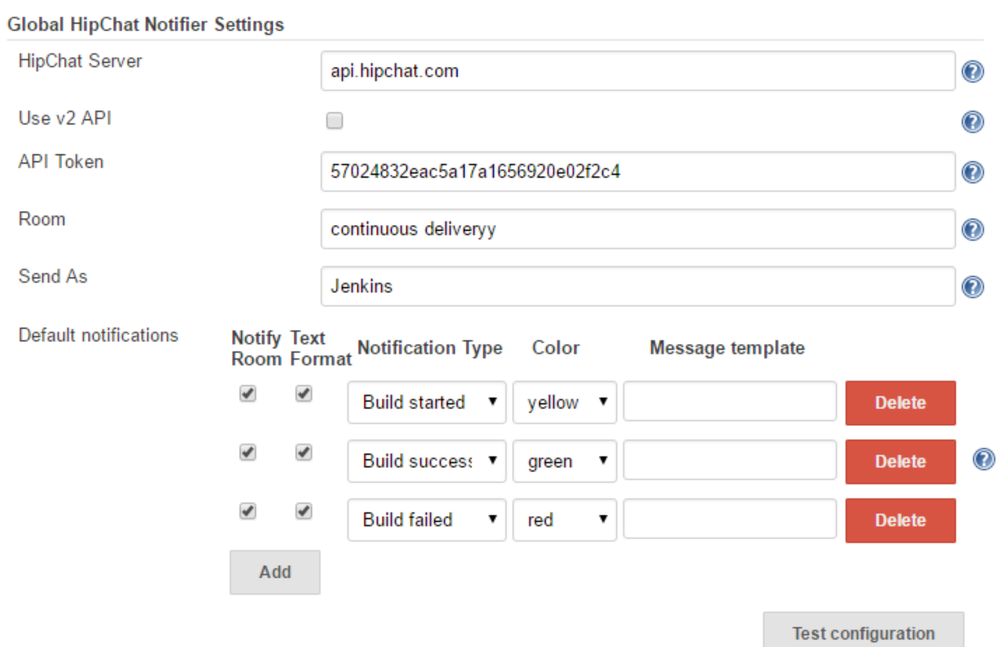


8. Add the key that we copied earlier into the **API Token** field.

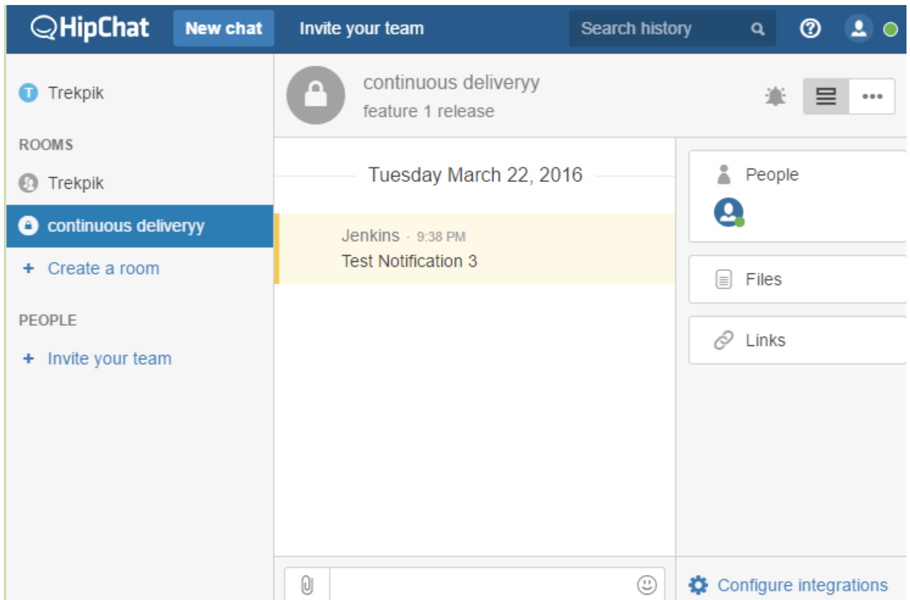
9. Add the room name that we created earlier from the HipChat dashboard:



10. Click on the **Add** button to add a few notifications:



11. Click on the **Test configuration** button to test the connection.
12. This is what you will see on your HipChat dashboard:



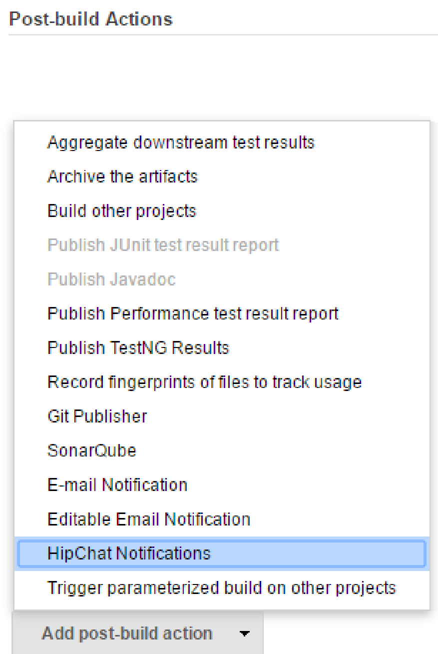
13. Now HipChat and Jenkins are connected.

Configuring a Jenkins job to send notifications using HipChat

For configuring a Jenkins job to send notifications using HipChat, perform the following steps:

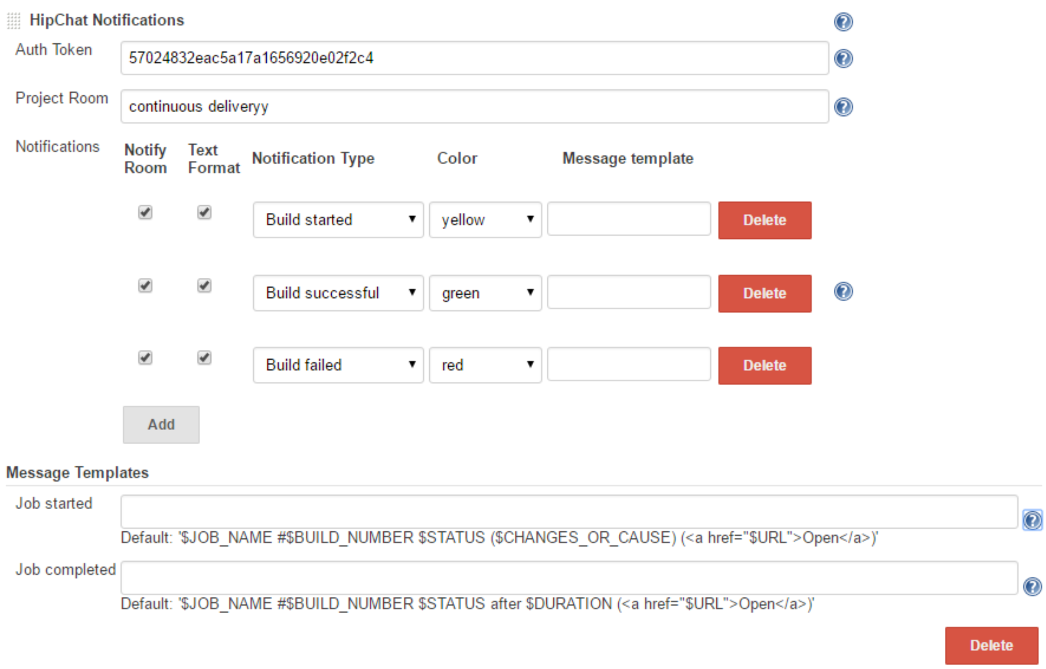
1. From the Jenkins dashboard, right-click on any of the Jenkins jobs that you want to configure to send notifications using HipChat and select **Configure**.
2. On the Jenkins job's configuration page, scroll down to the **Post-build Actions** section.

3. Click on the **Add post-build action** button and select **HipChat Notifications**:

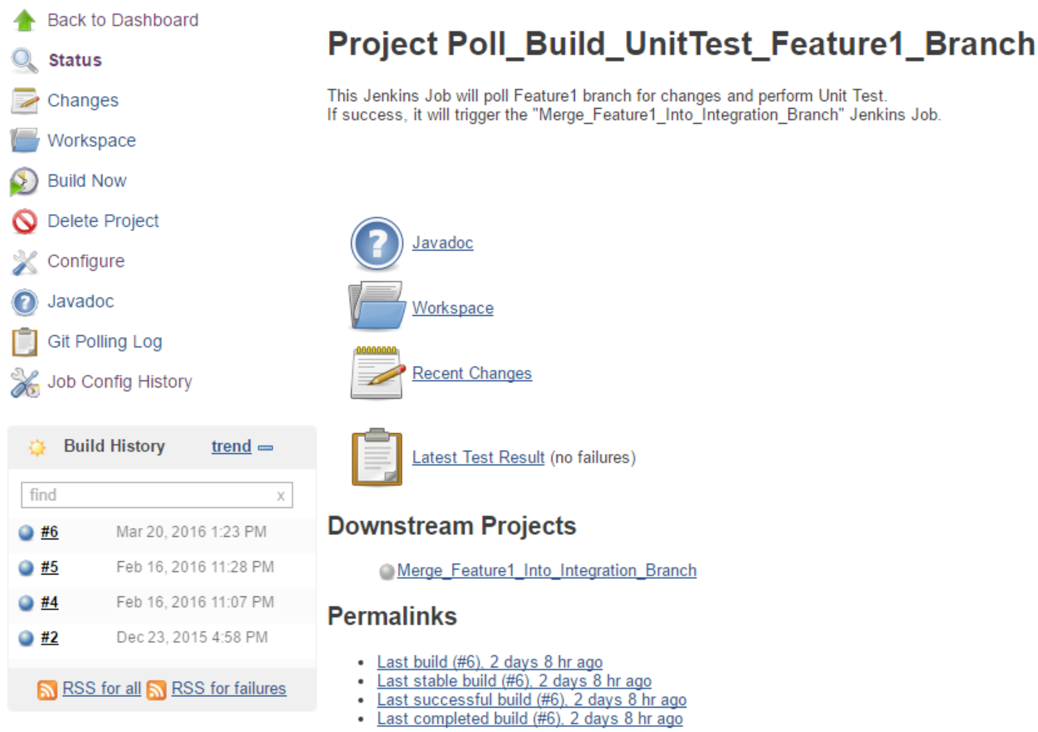


4. You will see the same configuration as you saw earlier. The only difference is that those were global configurations and these are specific to the current Jenkins job.
5. Add the key that we configured earlier under the **Auth Token** field.
6. Add the room name that we created earlier under the **Project Room** field.
7. Click on the **Add** button to add few notifications.

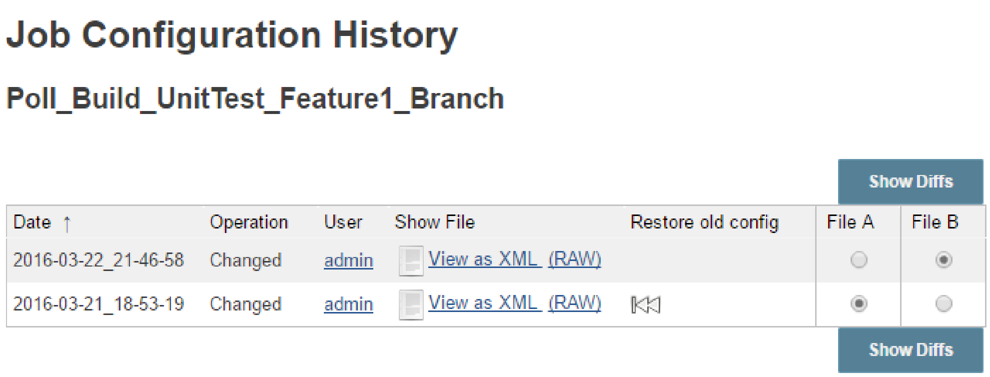
8. Leave the **Message Templates** as it is:



9. Save the Jenkins job by clicking on the **Save** button.
10. After saving the configurations, you will land on the next page.
11. Click on the **Job Config History** link:

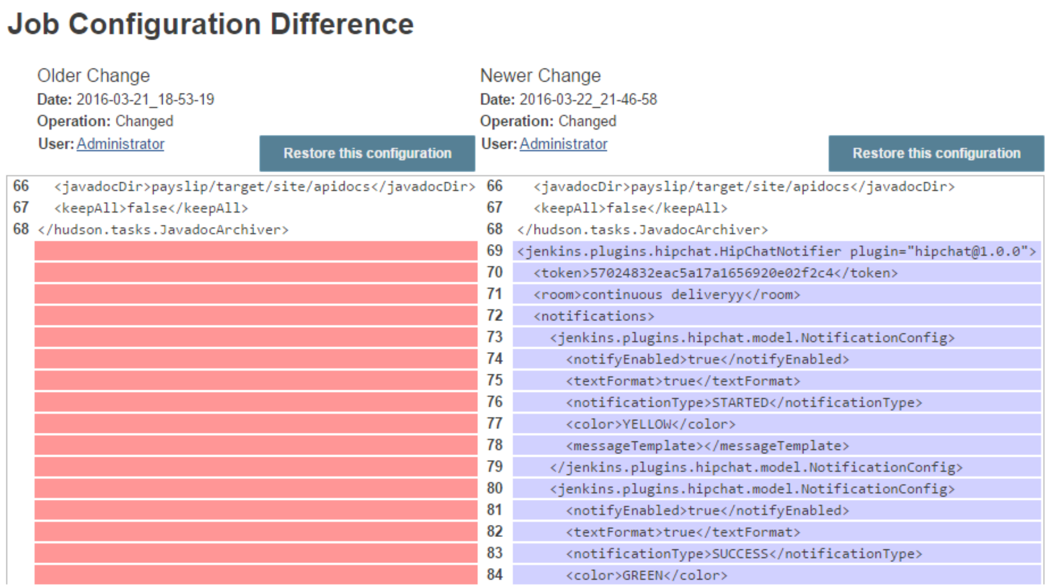


12. The changes made to the Jenkins job are listed, as shown in the following screenshot:



13. You can also compare the changes by clicking on the **Show Diffs** button.

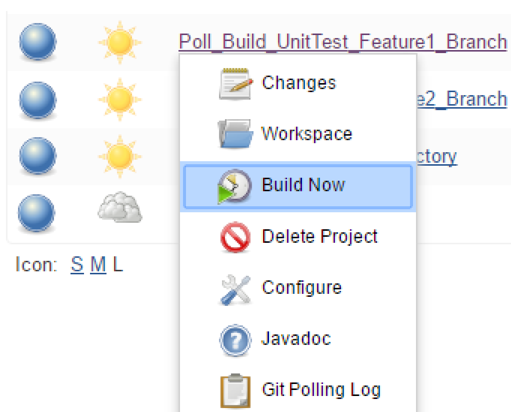
14. Here's what you will see:



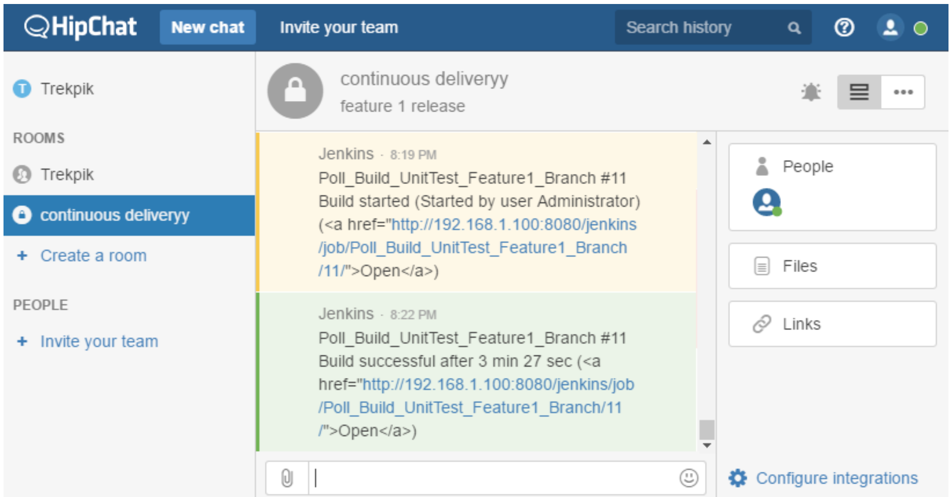
Running a build

Let's run a build to check whether the notification is working:

1. From the Jenkins dashboard, right-click on the Jenkins job that was configured to send HipChat notifications and click on **Build Now**:



2. Once the build gets completed successfully, the notifications are immediately received on the HipChat dashboard, as shown in the following screenshot:



Best practices for Jenkins jobs

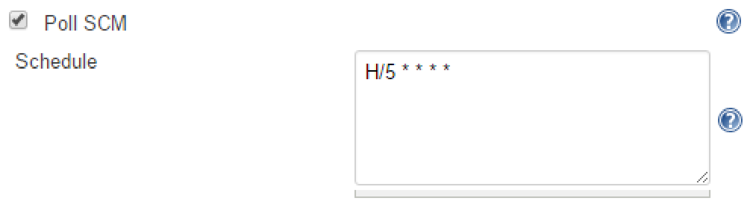
Using Distributed builds, version controlling the Jenkins configuration, implementing auditing of Jenkins and all the Jenkins configurations, features and plugins that we have seen in the current book were implemented in the best possible way. However, there are few critical things that were not discussed so far and need our attention. Let's see them one by one.

Avoiding scheduling all jobs to start at the same time

Multiple Jenkins jobs triggered at the same time may choke your Jenkins. To avoid this, avoid scheduling all jobs to start at the same time.

To produce even load on the system, use the symbol H. For example, using `0 0 * * *` for a dozen daily jobs will cause a large bottleneck at midnight. Instead, using `H H * * *` would still execute each job once a day, but not all at the same time.

The `H` symbol can be used with a range. For example, `H H(0-7) * * *` means sometime between 12:00 A.M. (midnight) to 7:59 A.M. You can also use step intervals with `H`, with or without ranges:



Here's the syntax for the **Schedule** field: <*Minute*><*Hour*><*Date of month*><*Month*><*Day of week*>

- `Minute`: Minutes within the hour (0–59)
- `Hour`: The hour of the day (0–23)
- `Date of month`: The day of the month (1–31)
- `Month`: The month (1–12)
- `Day of week`: The day of the week (0–7) where 0 and 7 are Sunday

Use `#` to add comments.

Examples

The examples to avoid scheduling all jobs at the same time are:

- Every 15 minutes (perhaps at :07, :22, :37, :52):

  ```
  H/15 * * * *
  ```

- Every 10 minutes in the first half of every hour (three times, perhaps at :04, :14, and :24):

  ```
  H(0-29)/10 * * * *
  ```

- Once every 2 hours every weekday (perhaps at 10:38 AM, 12:38 PM, 2:38 PM, and 4:38 PM):

  ```
  H 9-16/2 * * 1-5
  ```

- Once a day on the 1st and 15th of every month, except December:

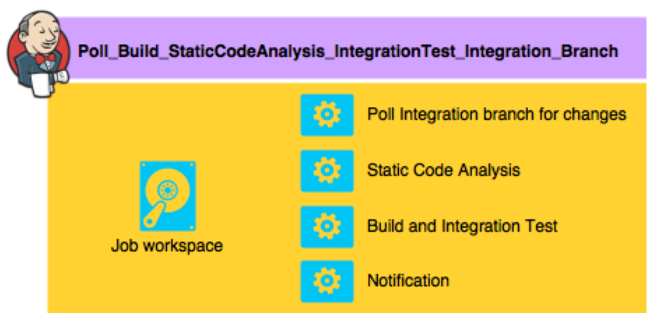
  ```
  H H 1,15 1-11 *
  ```

Dividing a task across multiple Jenkins jobs

The Jenkins Continuous Integration pipeline and the Jenkins Continuous Delivery pipeline contain multiple tasks. We are familiar with them from the previous chapters. However, you might have noticed that throughout the examples discussed in the book, most of the Jenkins jobs were a collection of individual tasks that could have been separate.

For example, consider the Jenkins job `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`.

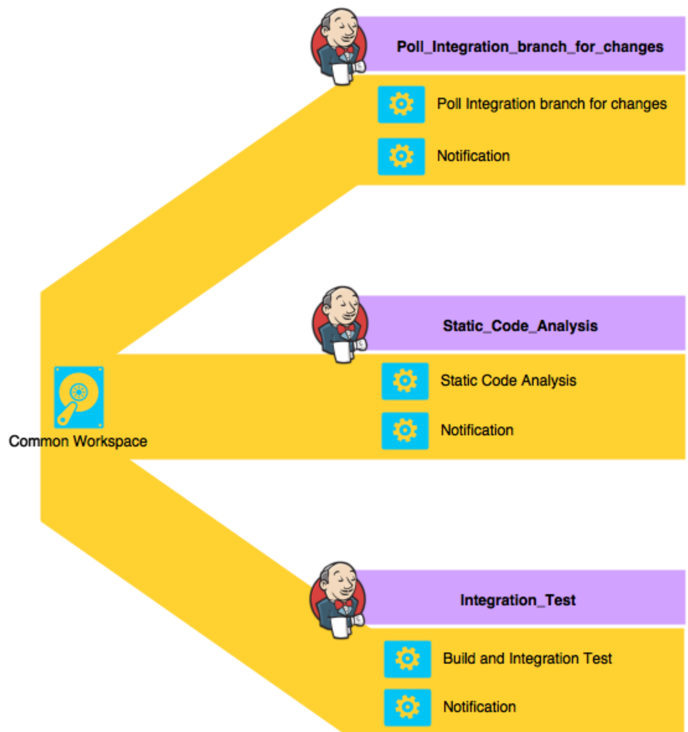
The aforementioned Jenkins job polls the Integration branch for changes and downloads them. It performs static code analysis on the downloaded code, builds it, and performs an integration test, followed by a notification. All this happens in the job's workspace, as shown in the following screenshot:



It is good to have the preceding configuration to keep things simple. However, dividing the tasks across multiple Jenkins jobs is a better option in many ways:

- First, it is easy for any new team member to grasp the Continuous Integration or Continuous Delivery design.
- Second, in case of failure, it is easy to narrow down to the error zone and debug the failure.
- Third, the notifications become more specific. For example, in the case of a single Jenkins job performing multiple operations, the notifications are at the end. In case of failure, you have to look for the logs and find the step at which the job has failed—whether it's the static code analysis or the integration test and so on.

The scenario is completely different if we divide the task across multiple Jenkins jobs. Every step has a notification, as shown in the following figure:

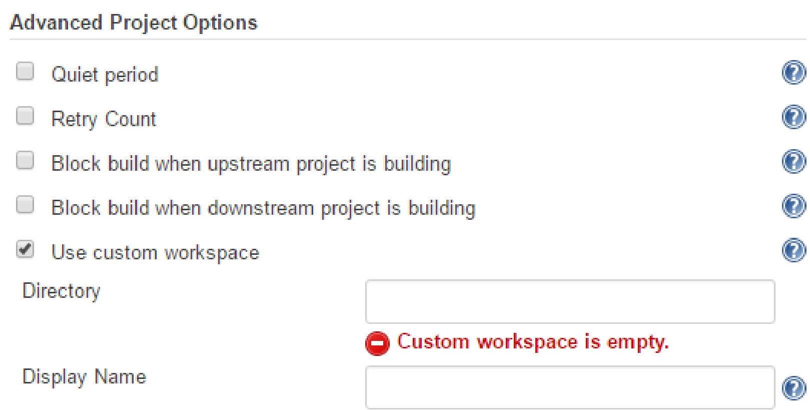


There is a problem: multiple Jenkins jobs will have their individual workspaces. However, the code on which all the preceding tasks need to be performed is a single change of code. So, having a common workspace for all the different Jenkins jobs is a must:

1. To achieve this, right-click on a Jenkins job from the Jenkins dashboard and select **Configure**.
2. Scroll down until you see **Advanced Project Options**:

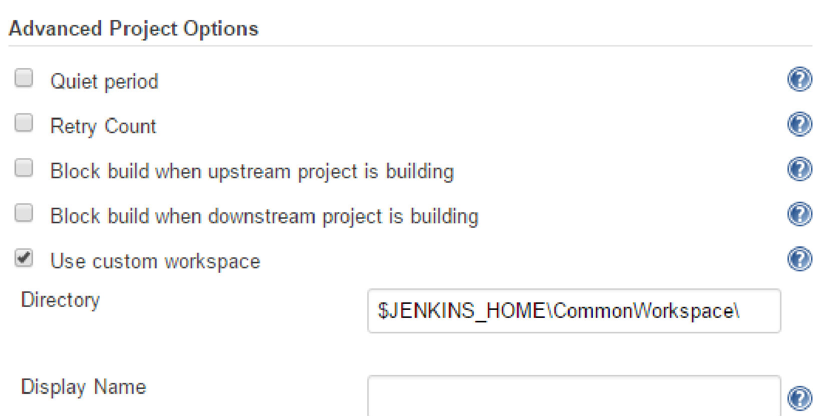
3. Click on the **Advanced...** button.

4. Select the **Use custom workspace** option, as shown in the following screenshot:



5. Add the value `$JENKINS_HOME\CommonWorkspace\`:

   `$JENKINS_HOME` is the environment variable that holds the Jenkins home path. `CommonWorkspace` is a workspace folder that will serve as a common place.



> You need to perform this configuration in all the Jenkins jobs that will share this common workspace.

Choosing stable Jenkins releases

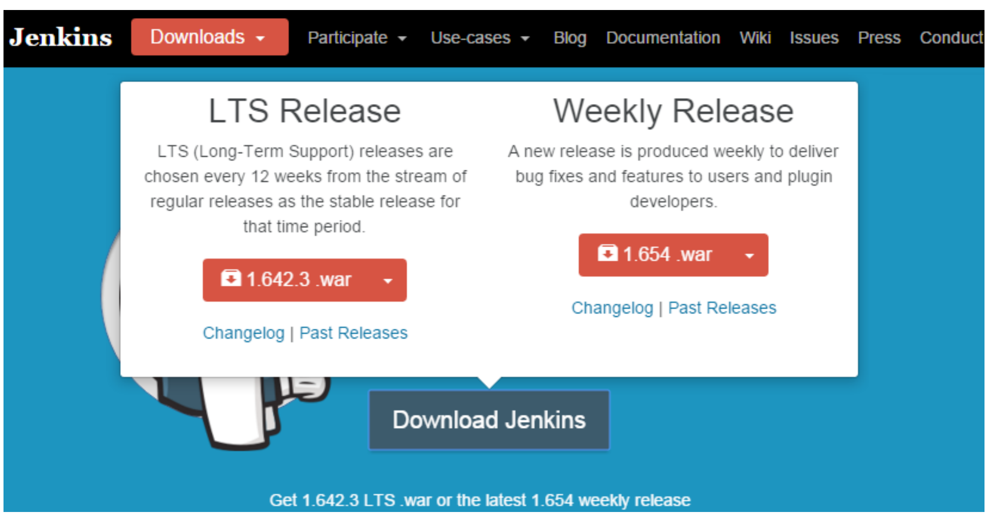
The Jenkins software frequently gets updated with new features and fixes. Usually, this happens weekly. There are a huge number of contributors, and a Jenkins community that constantly works to fix the issues faced by the millions of users worldwide.

However, it's not recommended that you update your Jenkins Continuous Integration server every week. They are not like Windows updates. The weekly Jenkins releases are not stable. Hence, they should not be treated like Windows updates.

Update Jenkins when you think you need to. For example, a recent Jenkins release contains a fix for the issue that has been lingering in your Jenkins setup. In such a scenario, you should update your Jenkins to the respective release with the fix.

Alternatively, you can always update Jenkins to a new stable release. These releases are called **Long Term Support** (**LTS**) releases. They happen every 3 months. A particularly stable release is chosen, a branch is created from it, and the new branch is rigorously tested.

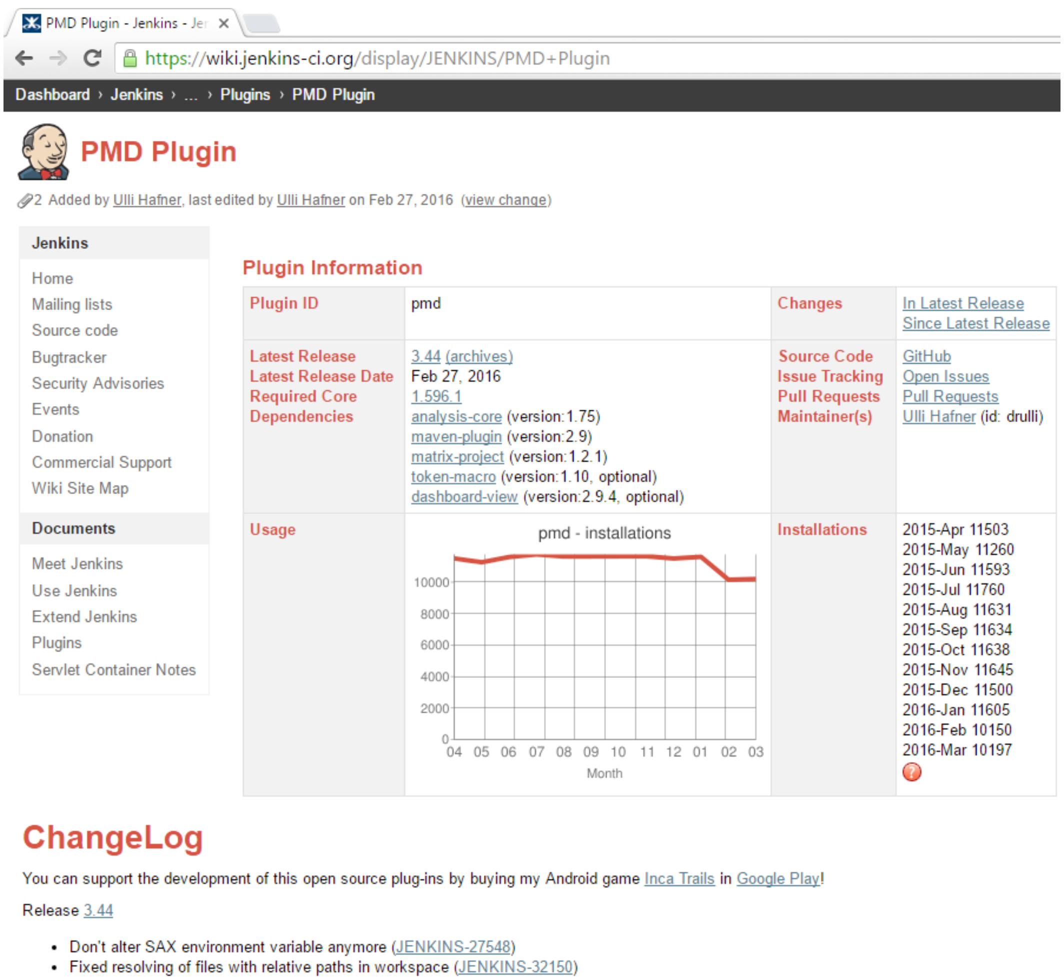
Here's the preview from the Jenkins website. By clicking on the **Download Jenkins** button on the Jenkins website, we get the option to choose between **LTS release** and **Weekly release**.



This is the page where you end up after clicking on the **Past Releases** link under the **LTS Release** section:

This is where you land on clicking on the **Changelog** link under the **LTS release section**:

The same is the case with plugins. Usually, plugins get updated whenever its creator does so. However, it's ideal to refrain from updating plugins as long as things are running smoothly in your team. Always check plugin wiki pages for a changelog. In the update center, click the link for the plugin name. This will take you to the wiki page, as shown in the following screenshot:



Cleaning up the job workspace

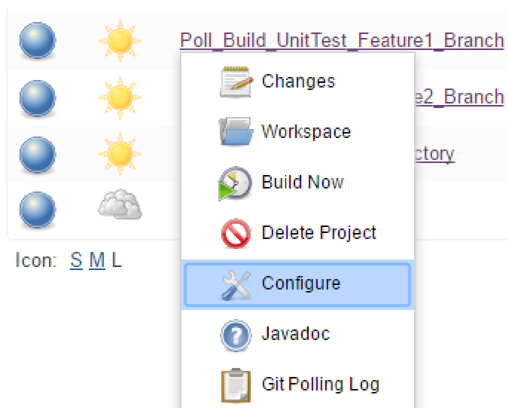
Jenkins jobs generate mammoth logs and artefacts inside the workspace. This is usually the case with a job that polls and builds the code. A Continuous Delivery solution that uses the distributed build architecture is also prone to space issues after a certain period of time.

However, there is one proactive step that can prove helpful to permanently eradicate the fear of bumping into the disk space issue. It is a very simple configuration that is available inside every Jenkins job. It's called **Discard Old Builds**:

> Ideally, Jenkins stores all the build logs, unless you manage them using the **Discard Old Builds** option or some other measures.

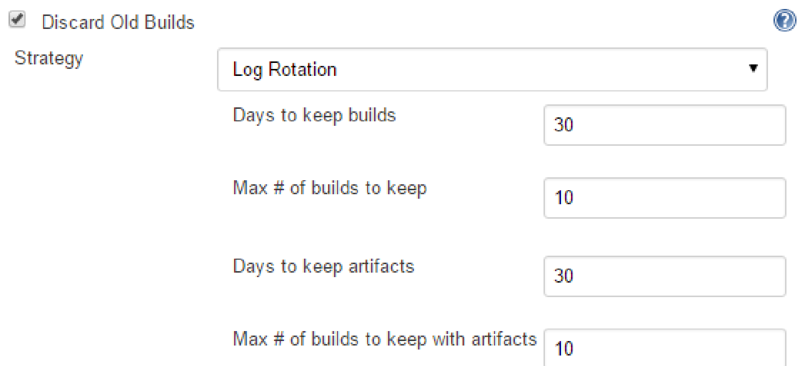
Let's see how to use the **Discard Old Builds** option. This configuration can be done right at the beginning of creating a new Jenkins job:

1. From the Jenkins dashboard, right-click on any of the Jenkins jobs you want and select **Configure**:



2. Once inside the Jenkins job configuration page, scroll down until you see **Discard Old Builds**:

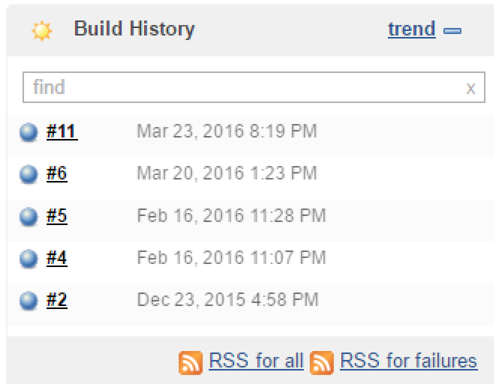
3. Once you select the option, a whole new set of fields appear, as shown in the next screenshot:
    - **Days to keep builds**: Using this option, you can tell Jenkins as to how long a build record is stored
    - **Max # of builds to keep**: Using this option, you can tell Jenkins how many builds to keep
    - The same applies to artefacts inside the workspace:



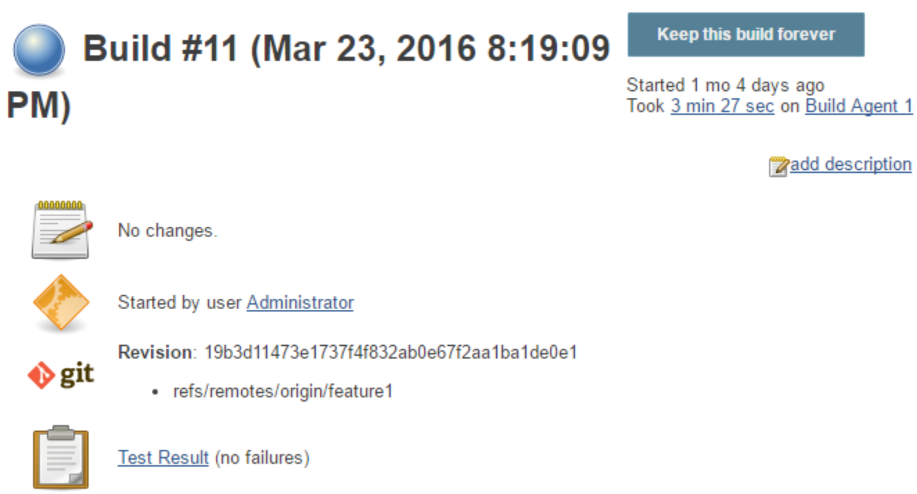
Using the Keep this build forever option

If you want to keep a particular build for future references, then you can use the **Keep this build forever** option. To do so, follow these steps:

1. From the Jenkins dashboard, click on the required Jenkins job.
2. From the **Build History** section on the Jenkins job page, click on the build that you would like to preserve. This will take you to the respective build page:

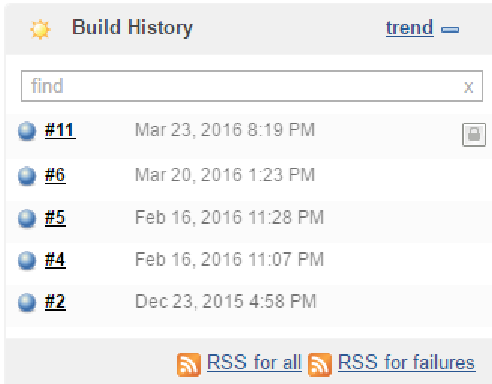


3. On the build page, you will find a button named **Keep this build forever** at the top-right corner of the screen, as shown in the following screenshot:



4. Click on **Keep this build forever** to save the build.

5. Come back to the job page, and you will see that the particular build has been locked. Therefore, it will not be deleted during the cleanup activity:



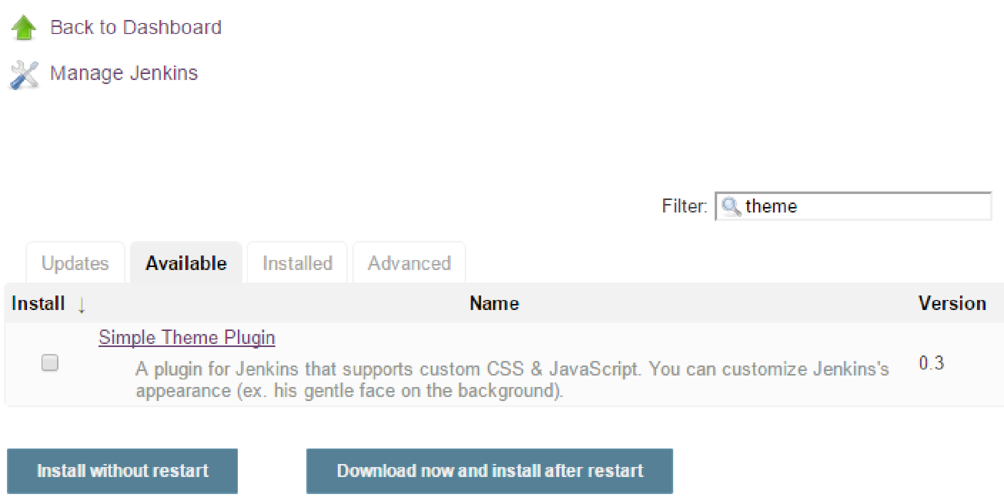
The last stable and last successful build are always stored as well.
The **Keep this build forever** option is available only if the **Discard Old Builds** option is selected inside the Jenkins job.

Jenkins themes

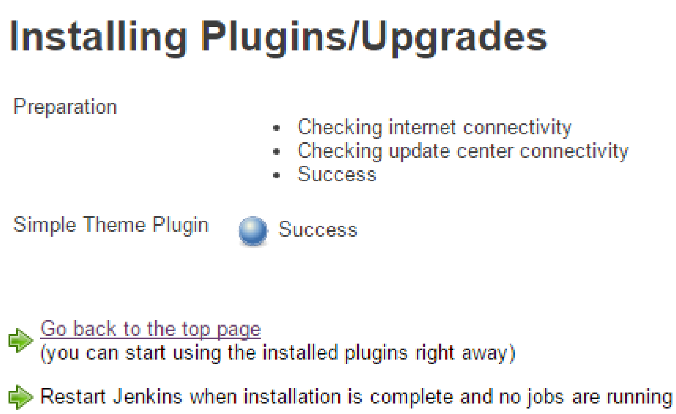
The steps to manage Jenkins themes are:

1. From the Jenkins dashboard, click on **Manage Jenkins**. This will take you to the **Manage Jenkins** page.
2. Click on the **Manage Plugins** link and go to the **Available** tab.
3. Type `theme` in the search box.

4. Select **Simple Theme Plugin** from the list and click on the **Install without restart** button:

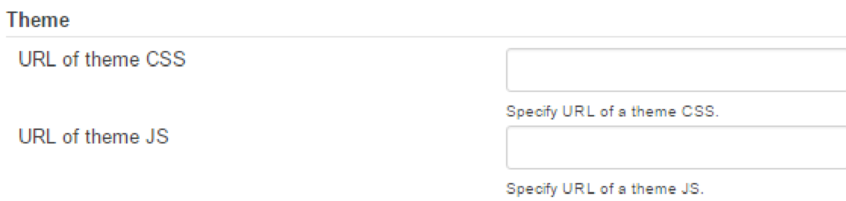


5. The download and installation of the plugin starts automatically:

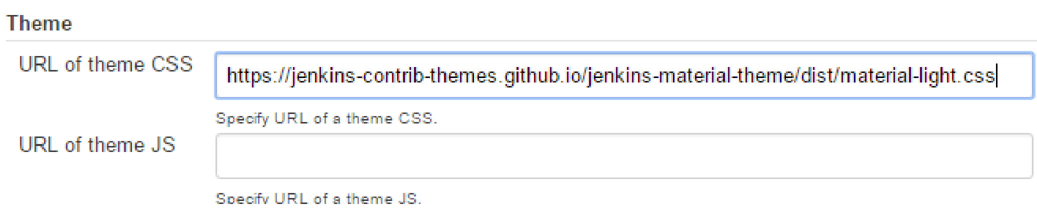


6. Go to the **Configure System** link from the **Manage Jenkins** page.

7. Scroll down until you see the **Theme** section, as shown in the following screenshot:



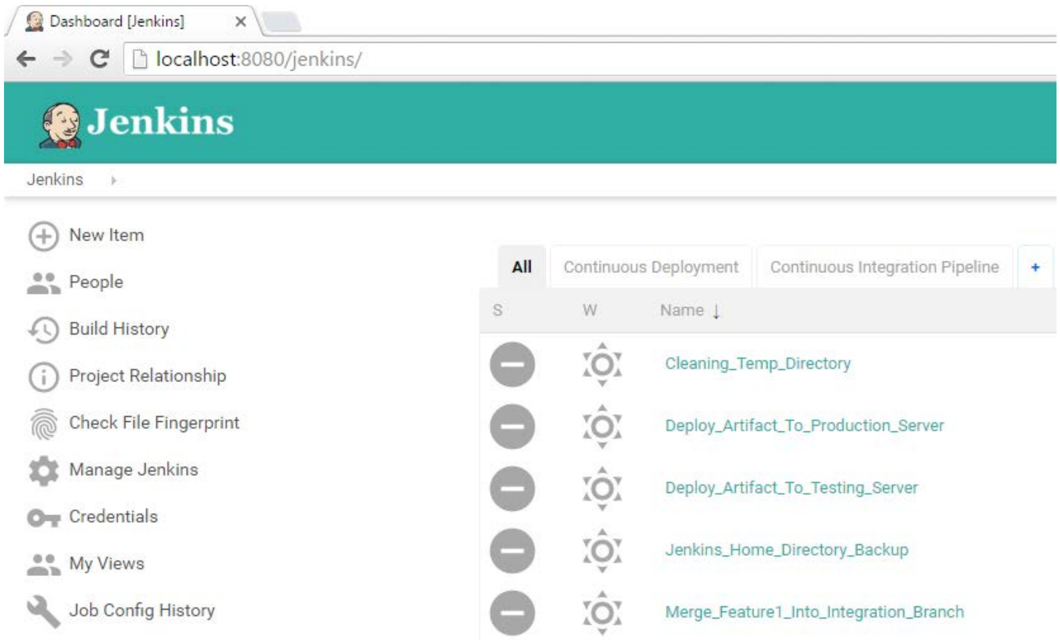
8. Add the link `https://jenkins-contrib-themes.github.io/jenkins-material-theme/dist/material-light.css` in the **URL of theme CSS** field.
9. In addition, you can search for Jenkins themes on the Internet and add the URL in the **Theme** section:



> The preceding theme is licensed under `http://afonsof.mit-license.org/`.

10. Save the configuration by clicking on the **Save** button.
11. If required, restart the Jenkins server.

12. Once done, move to the Jenkins dashboard and this is what you will see:



Summary

In this chapter, we saw how to use the distributed build architecture using Jenkins slaves to achieve load balancing. We can scale the Jenkins build cluster to any number of slaves by adding new Jenkins nodes and grouping them under an appropriate label. In any organization, audit plays an important role in identifying what caused an issue. This is where the Jenkins plugins named jobConfigHistory and Audit Trail come in handy. You might want to use both together. We also saw how to send notifications using the HipChat tool. We discussed how to configure Jenkins jobs to optimize disk usage by regular workspace cleanup.

Lastly, we saw how to use the Jenkins Simple Theme Plugin to make Jenkins visually appealing.

Index

Symbol

A

B

D

E

F

G

H

I

J

U

V

W

Y