Continuous Integration with Jenkins

Contents

1. Concepts of Continuous Integration

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.

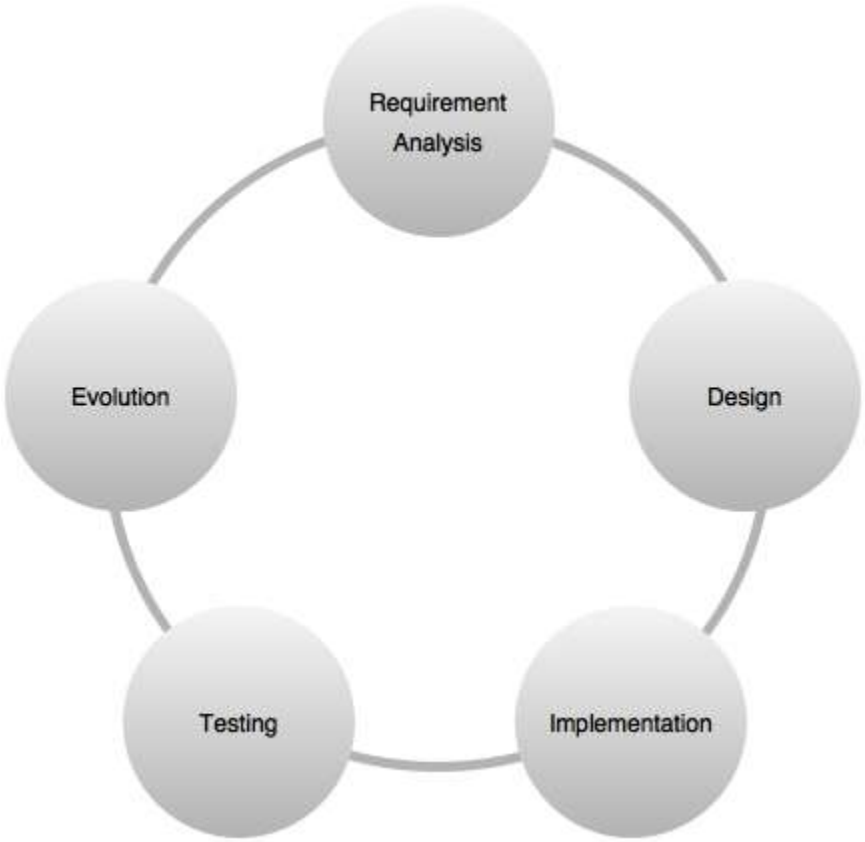
The agile software development process

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 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:



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.

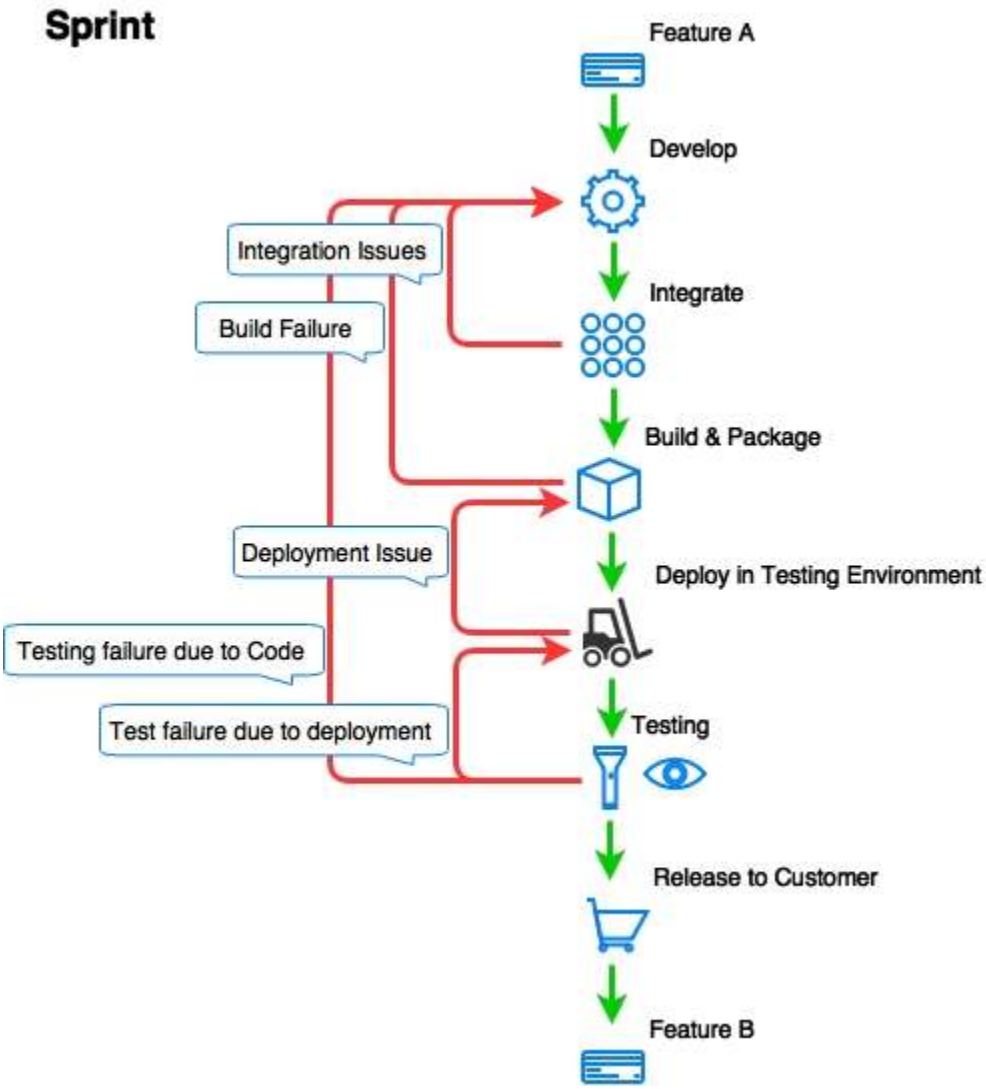
Integration is the act of submitting your personal work (modified code) to the common work area (the potential software solution). This is technically done by merging your personal work (personal branch) with the common work area (Integration branch). Continuous Integration is necessary to bring out 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.



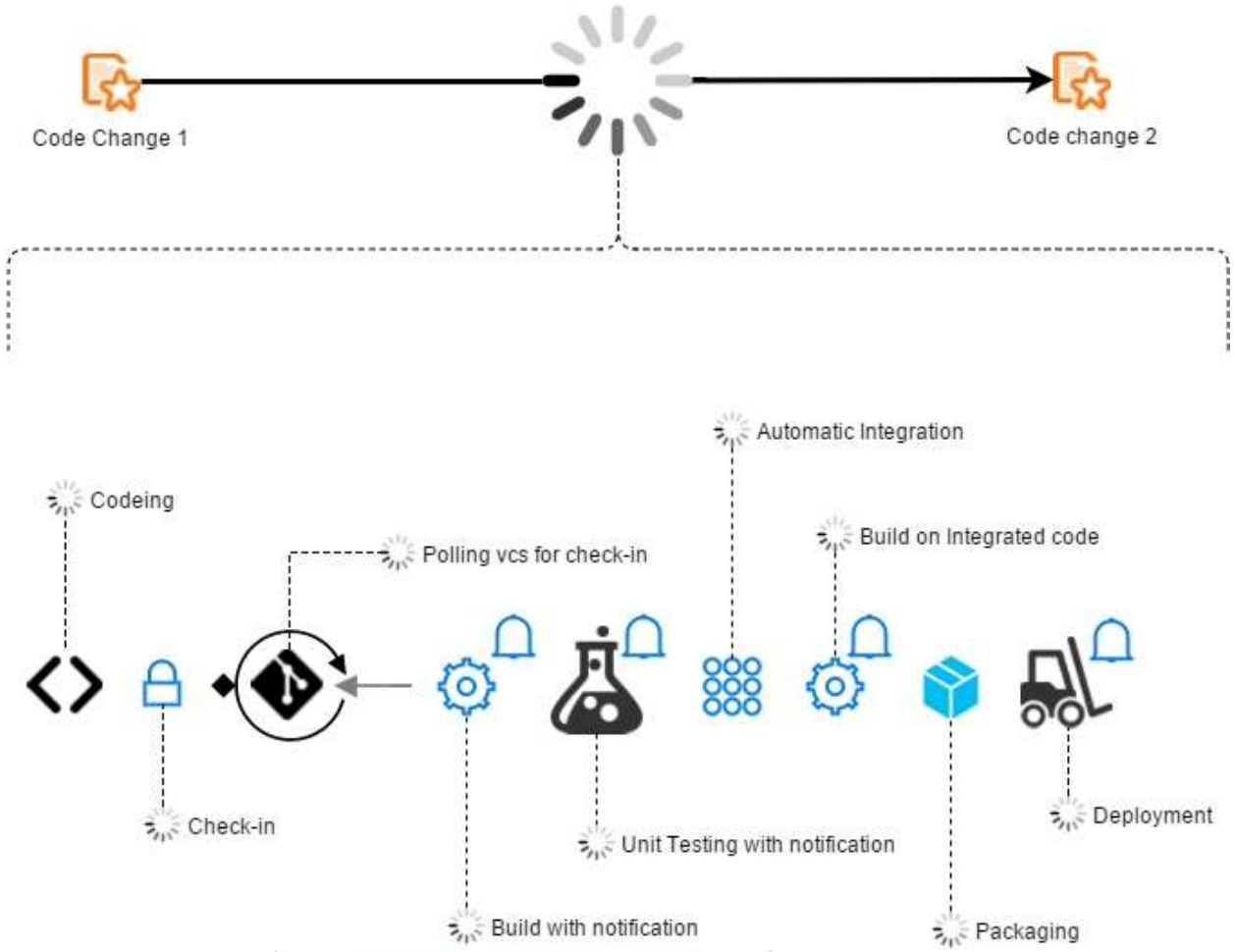
Note

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.

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:



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

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

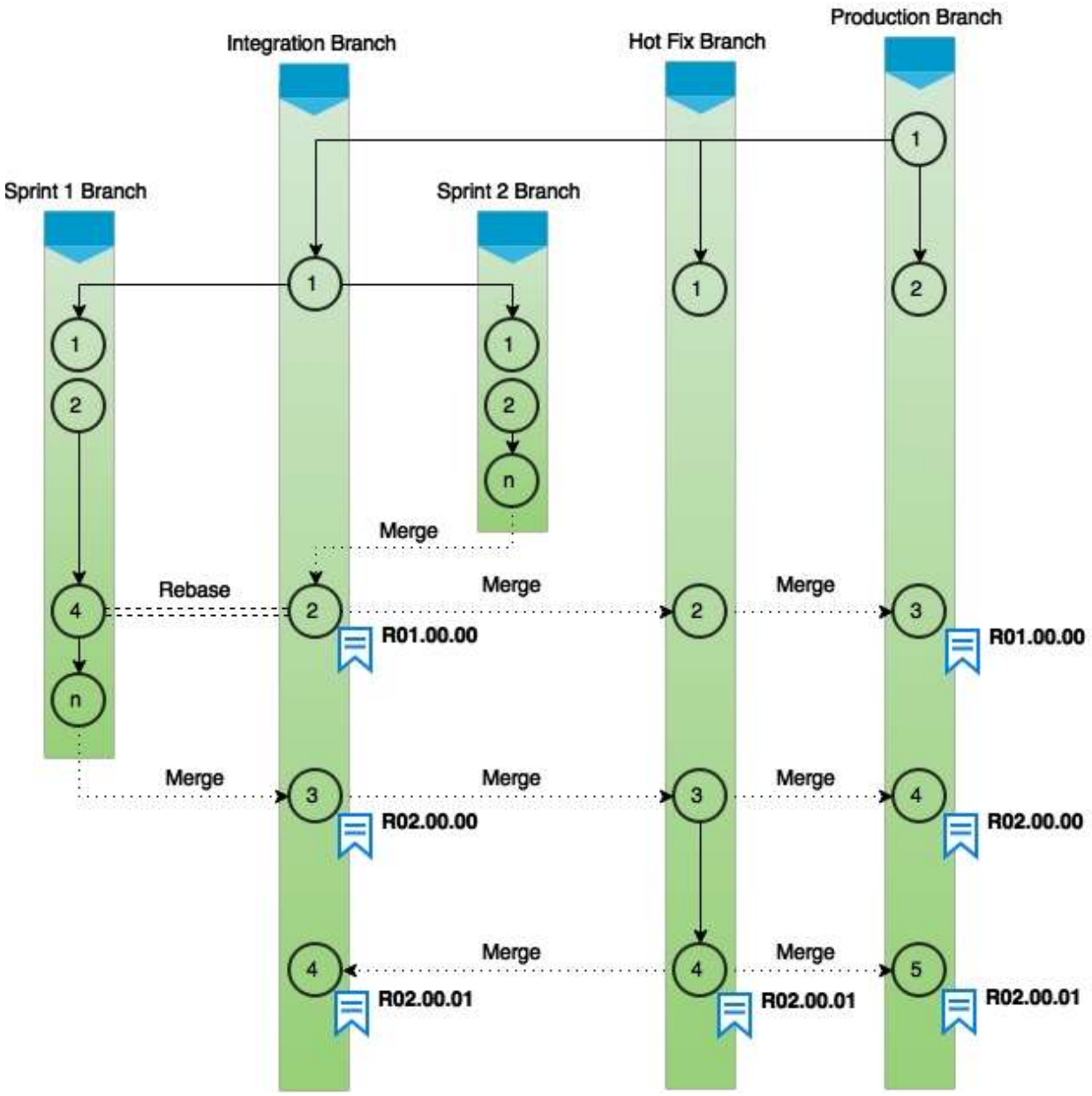
This is the most basic and the most important requirement to implement Continuous Integration. A version control system, or sometimes it's also called a **revision control system**, is a tool used to manage your code history. It can be centralized or distributed. Two of the famously centralized version 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.

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:

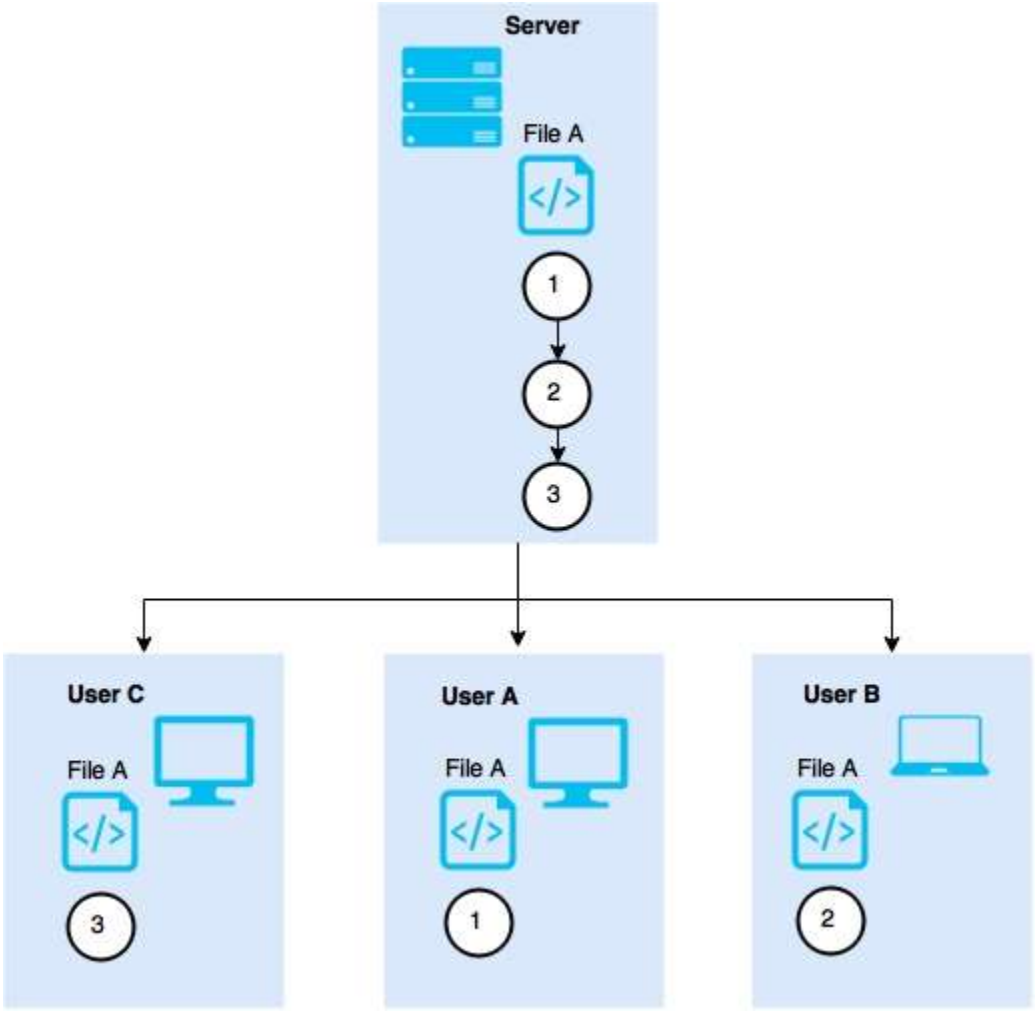
- Local version control systems
- Centralized version control systems
- Distributed version control systems

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.

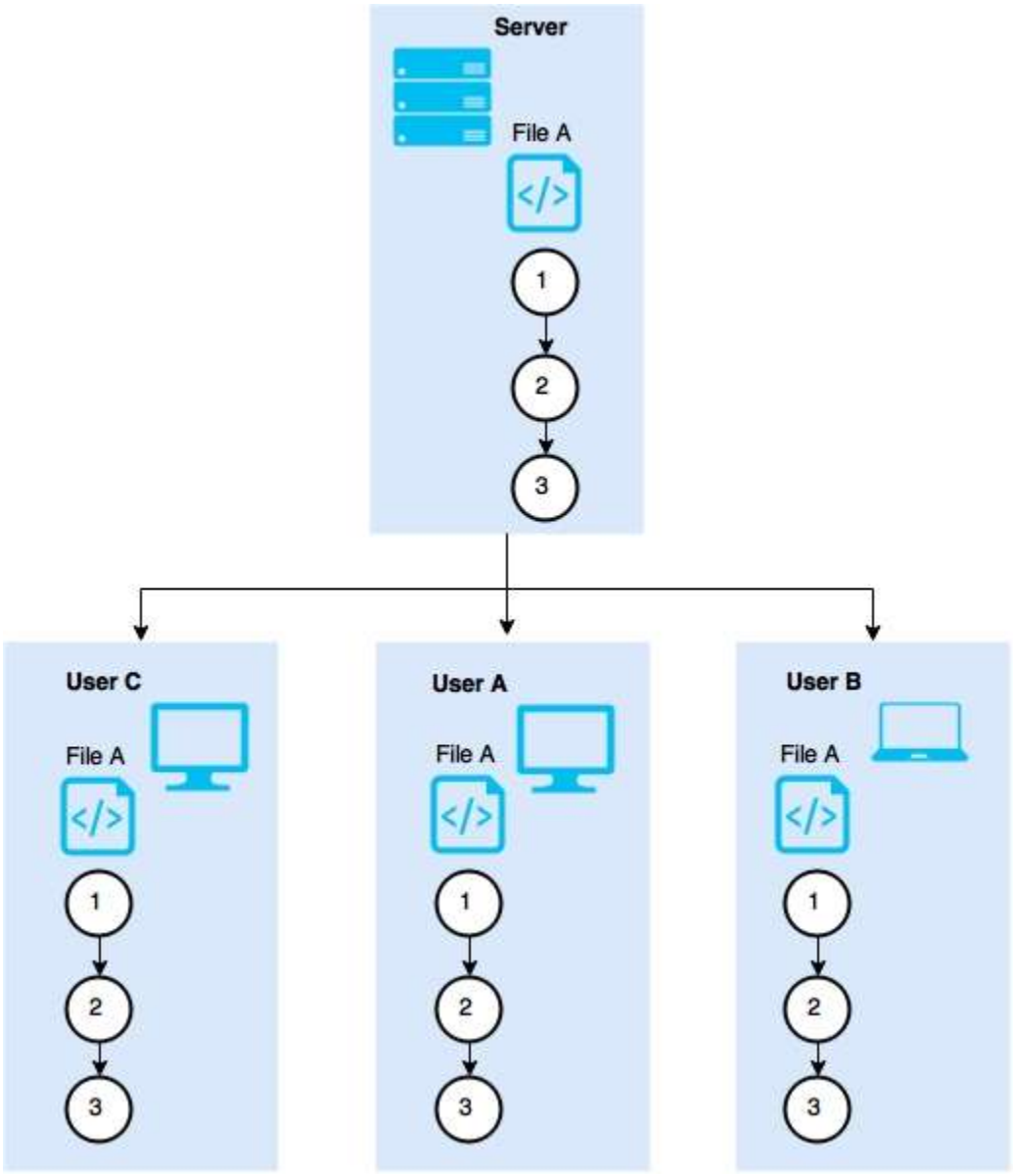
The following diagram illustrates a centralized VCS:



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.



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?

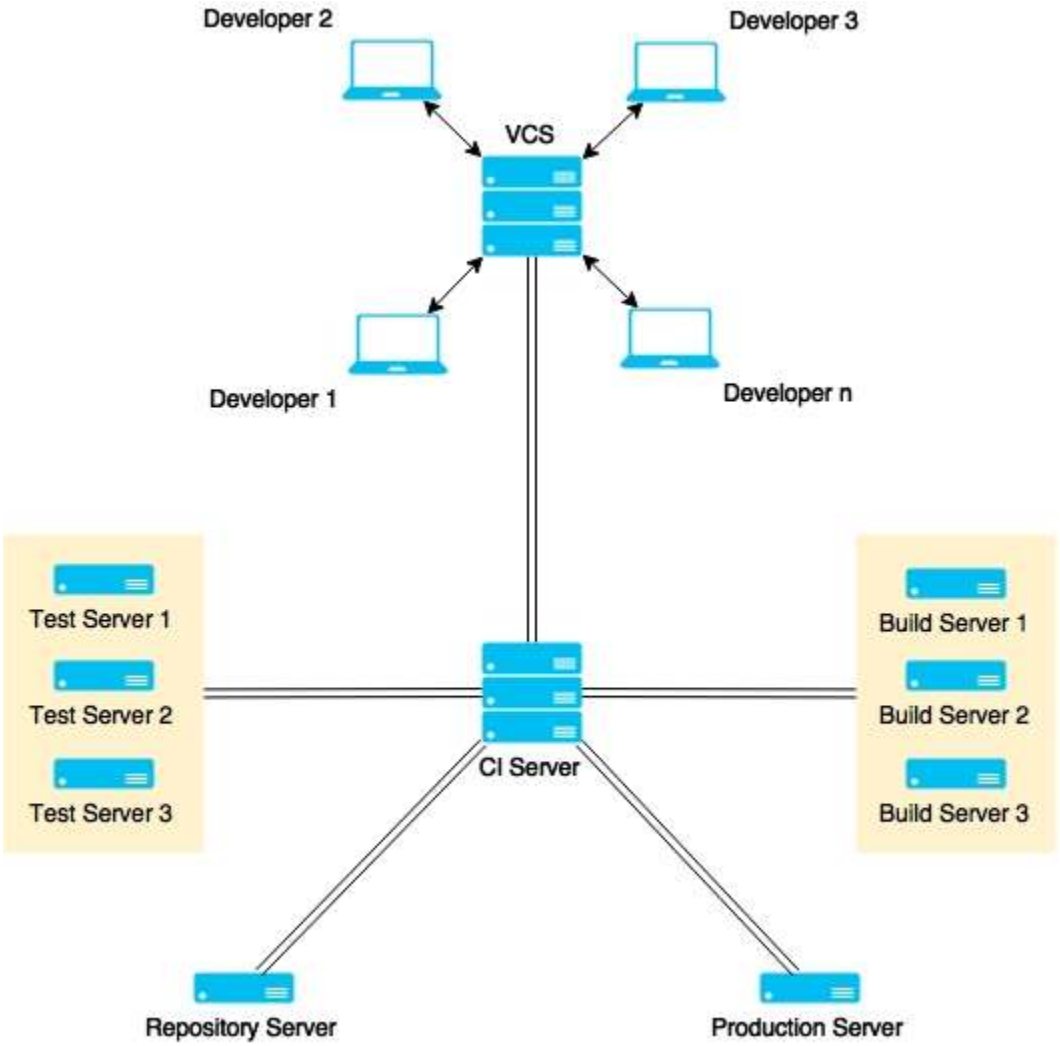
A repository tool does the following:

- 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.

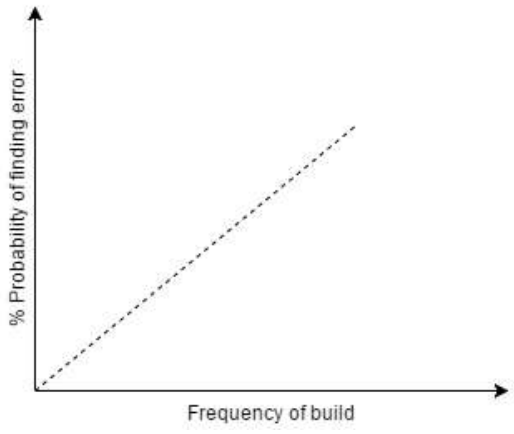
Creating a self-triggered build

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.

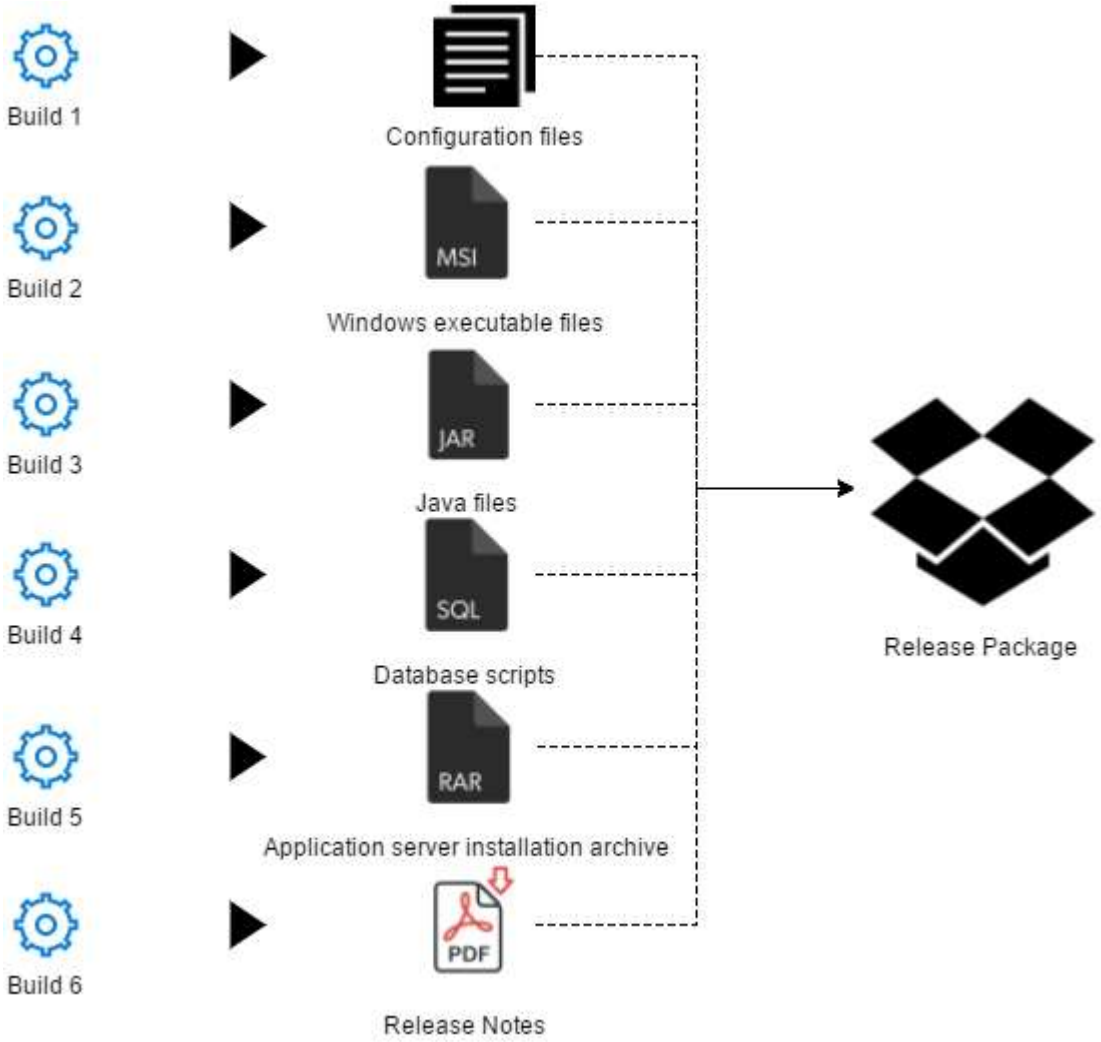
When builds are frequent and fast, the probability of finding errors (a build error, compilation error, and integration error) is also greater and faster.



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.

Note

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.

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.

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.

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

2. Setting up Jenkins

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

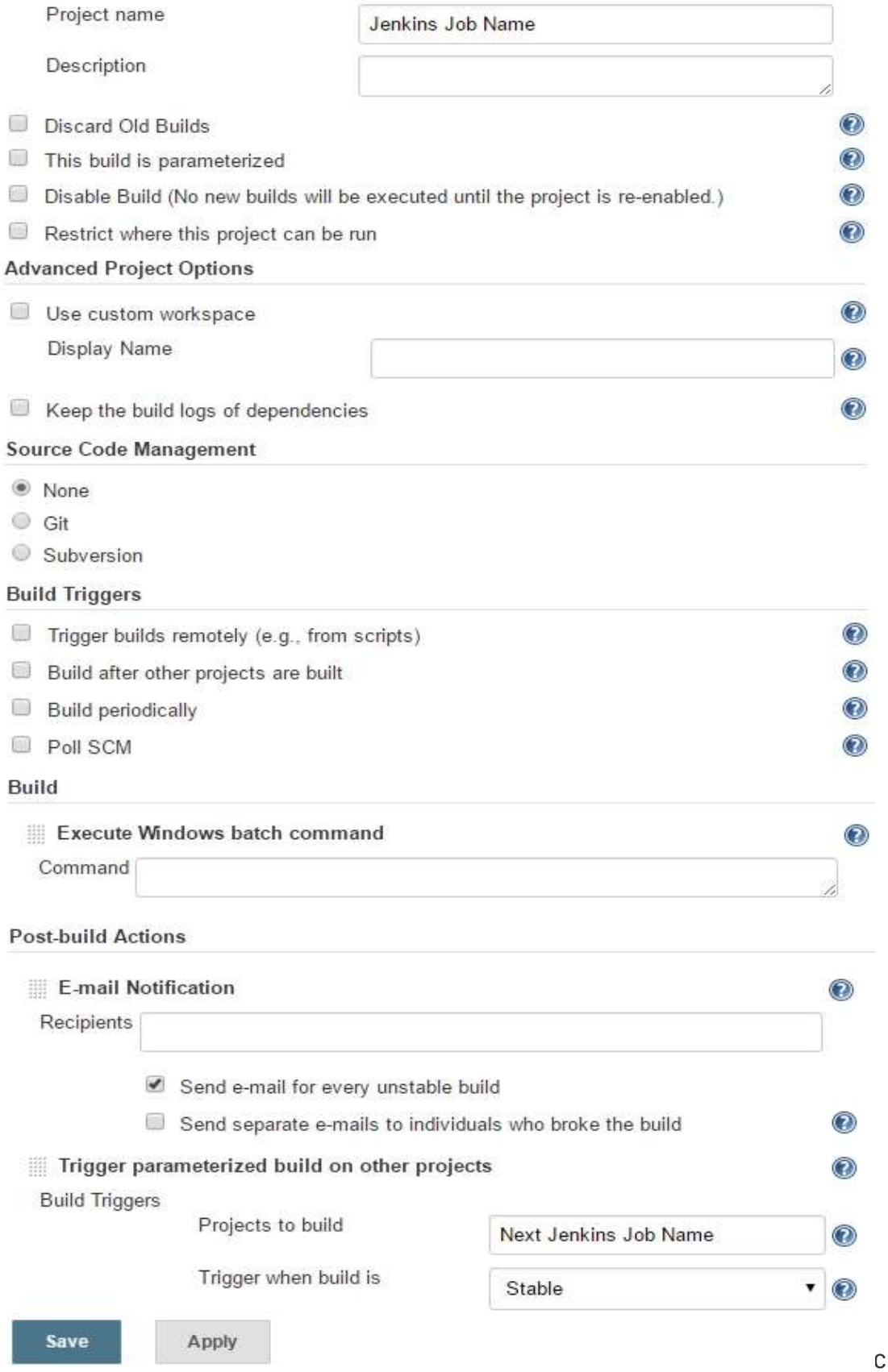
Deployment. Jenkins is supported by a large number of plugins that enhance its capability. The Jenkins tool is written in Java and so are its plugins. The tool has a minimalistic GUI that can be enhanced using specific plugins if required.

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:


Project name
Jenkins Job Name
Description
Discard Old Builds
This build is parameterized
Disable Build (No new builds will be executed until the project is re-enabled.)
Restrict where this project can be run
Advanced Project Options
Use custom workspace
Display Name
Keep the build logs of dependencies
Source Code Management
None
Git
Subversion
Build Triggers
Trigger builds remotely (e.g., from scripts)
Build after other projects are built
Build periodically
Poll SCM
Build
Execute Windows batch command
Command
Post-build Actions
E-mail Notification
Recipients
Send e-mail for every unstable build
Send separate e-mails to individuals who broke the build
Trigger parameterized build on other projects
Build Triggers
Projects to build
Next Jenkins Job Name
Trigger when build is
Stable
Save
Apply

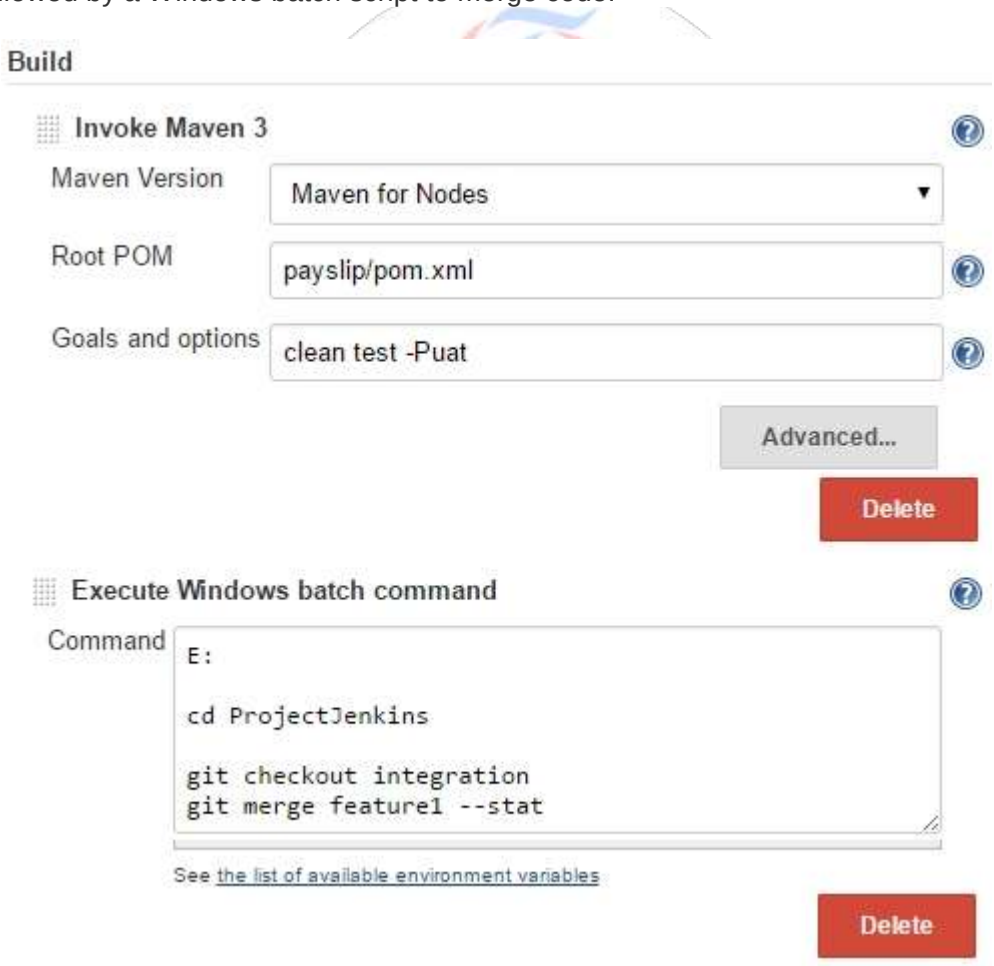
C

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.

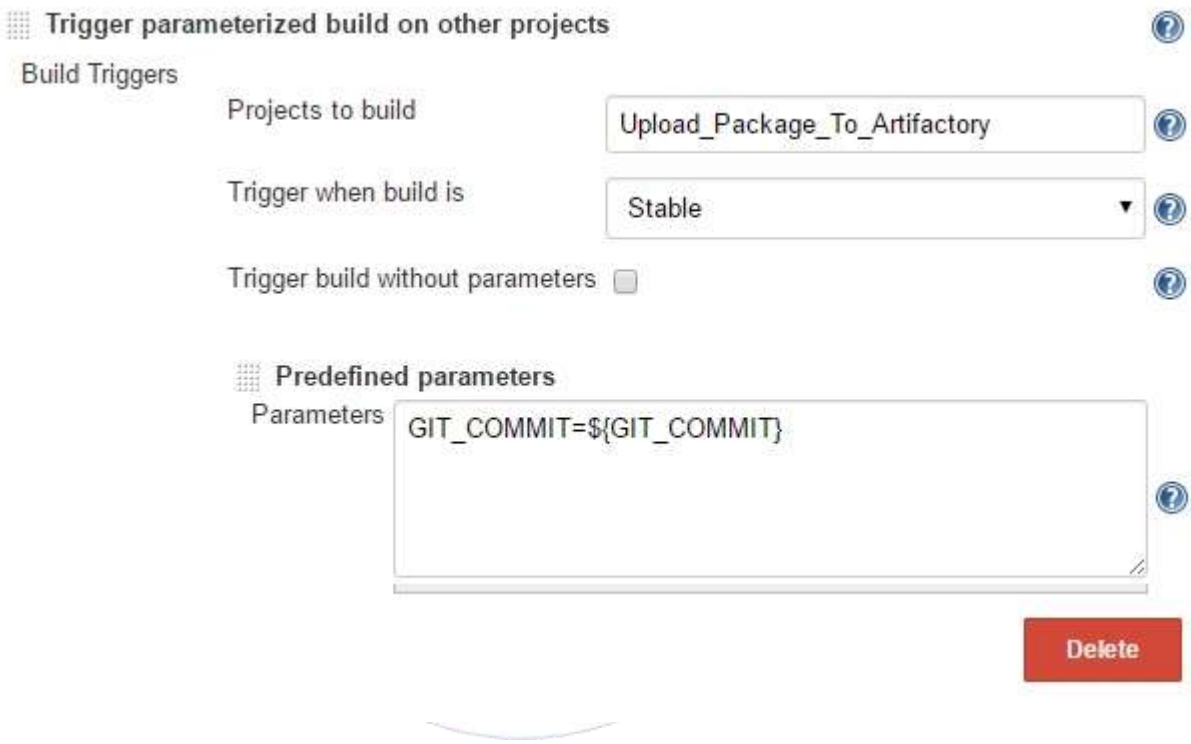
Jenkins build

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:



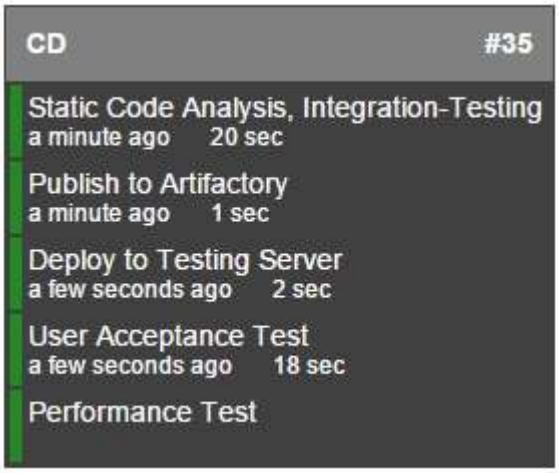
Jenkins post-build actions

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.



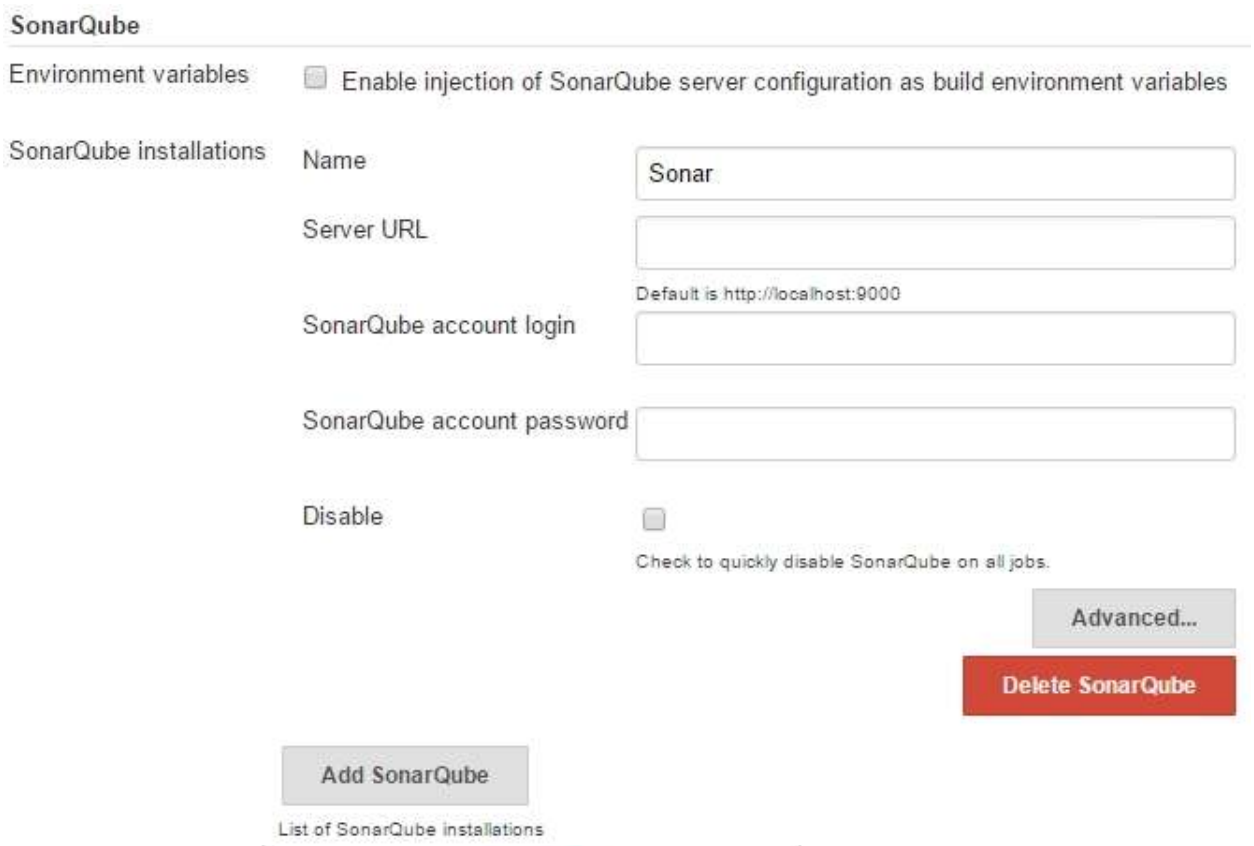
Note

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

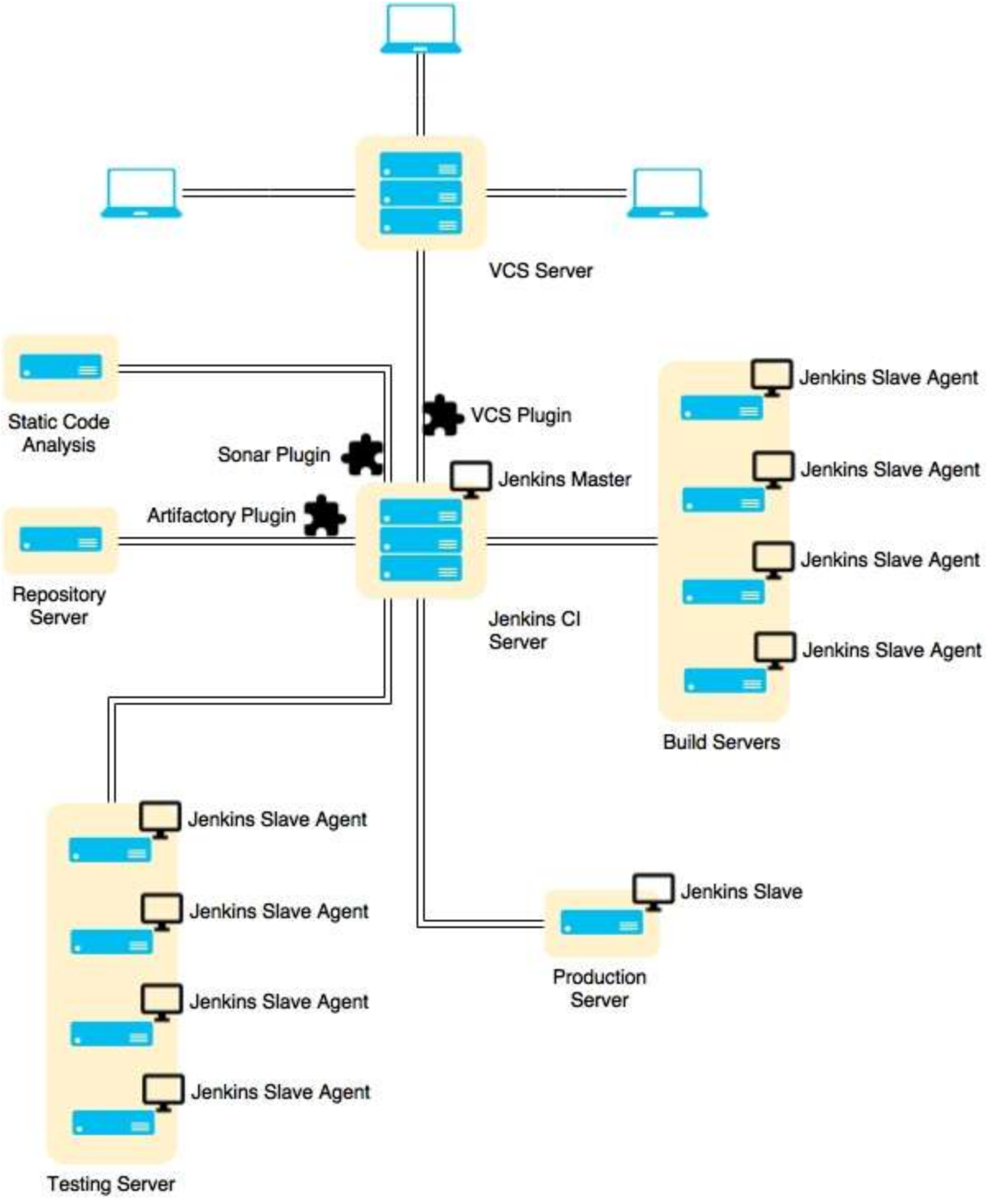
There are more than 300 plugins available for Jenkins and the list keeps increasing. Plugins are simple Maven projects. Therefore, anyone with a creative mind can create and share their plugins on the Jenkins community to serve a purpose.

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.


VCS Server
Static Code Analysis
Sonar Plugin
VCS Plugin
Jenkins Master
Artifactory Plugin
Repository Server
Jenkins CI Server
Jenkins Slave Agent
Jenkins Slave Agent
Jenkins Slave Agent
Jenkins Slave Agent
Build Servers
Jenkins Slave Agent
Jenkins Slave Agent
Jenkins Slave Agent
Jenkins Slave Agent
Testing Server
Jenkins Slave
Production Server

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).

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

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.

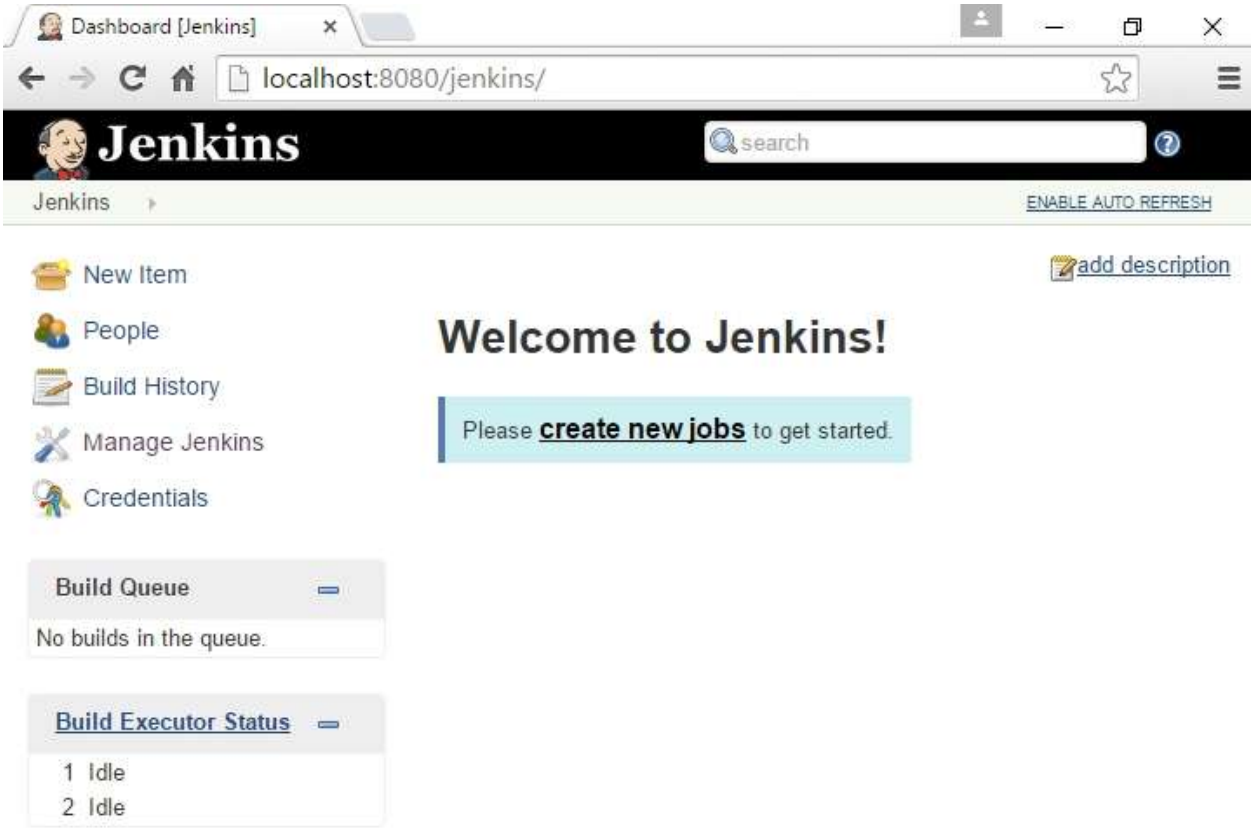
We also need Java running on the machine. In this section, we are using jre1.8.0_60. While installing the Apache Tomcat server, you will be asked to install Java. Nevertheless, it is suggested that you always use the latest stable version available.

Note

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. Simply move the downloaded `jenkins.war` file to the `webapps` folder, which is present inside the installation directory of your Apache Tomcat server.
3. That's all you need to do. You can access Jenkins using the URL `http://localhost:8080/jenkins`.
4. The Jenkins Dashboard is shown in the following screenshot:



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>
```


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

Method 2 – creating the JENKINS_HOME environment variable

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

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

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

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

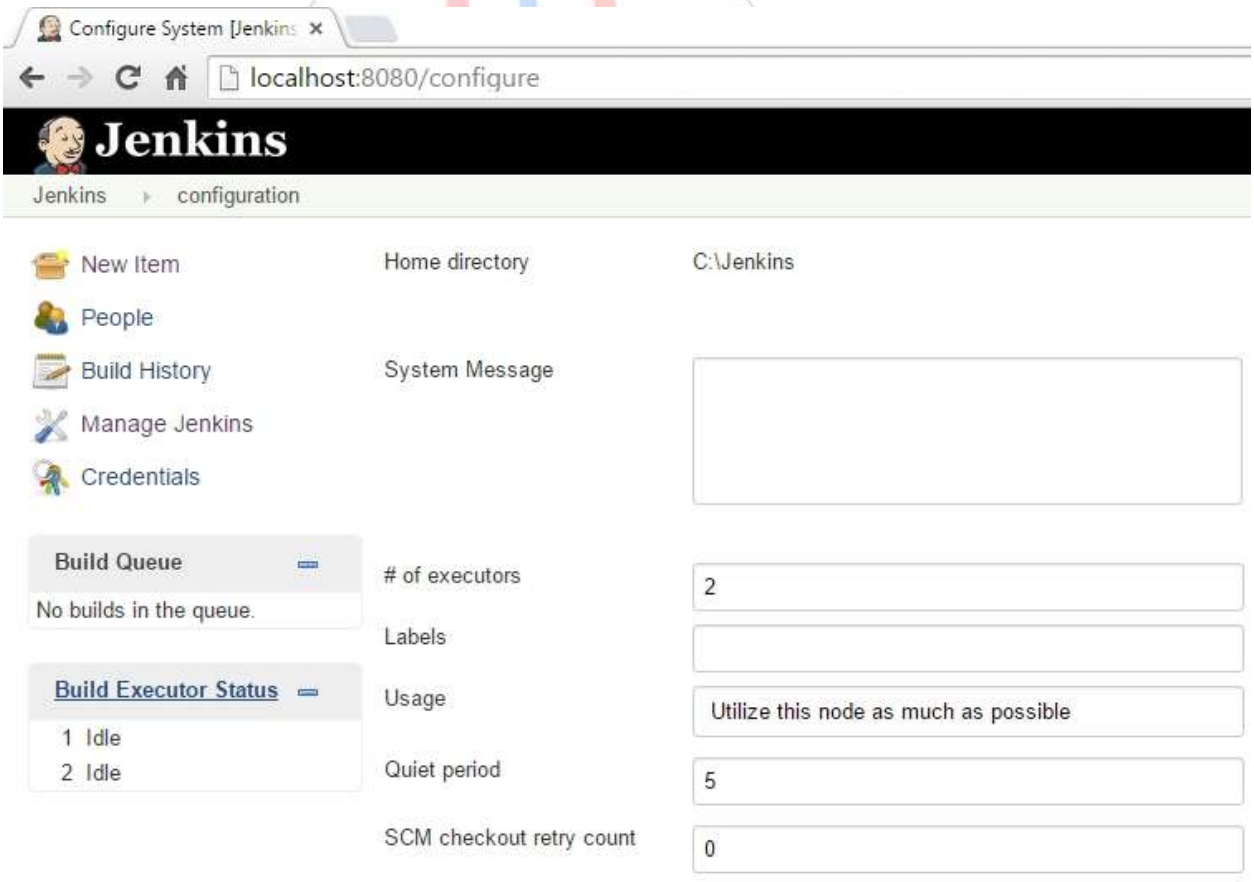
```
echo %JENKINS_HOME%
```

5. The output should be:

```
C:\Jenkins
```

6. Start the Apache Tomcat server.

7. 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:

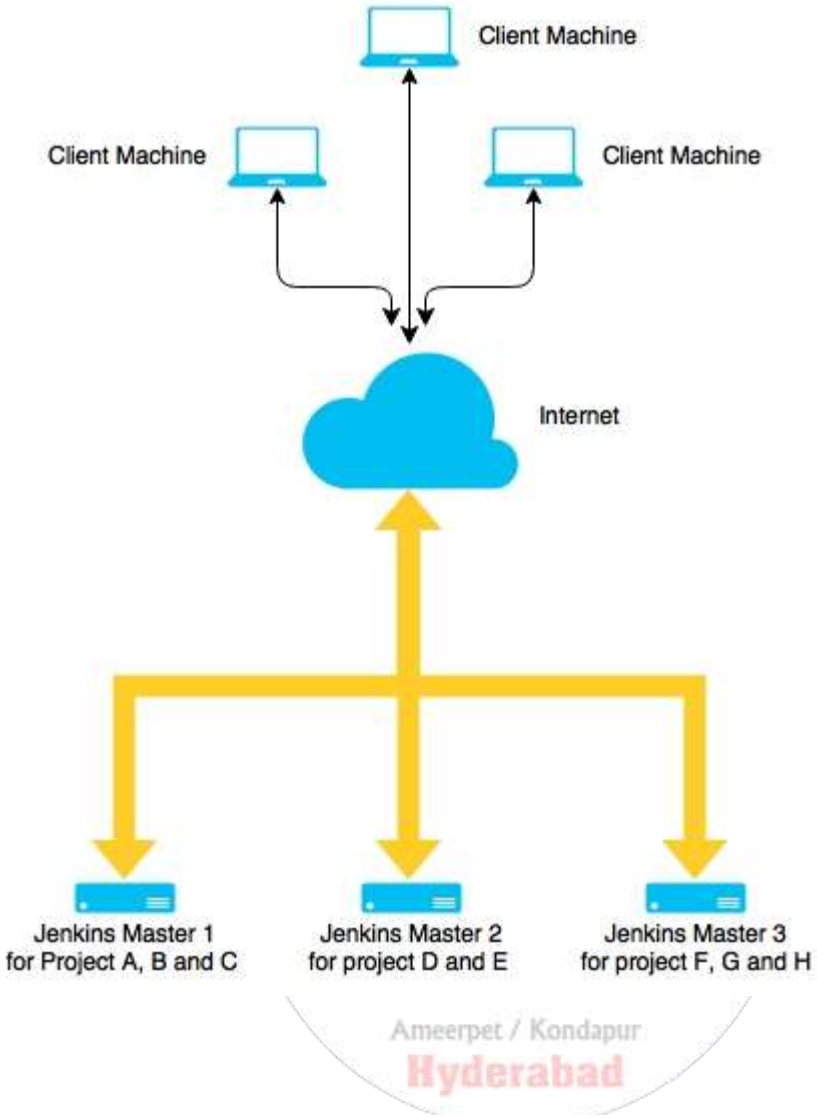
- Number of jobs
- Number of builds
- Number of slaves
- Number of users accessing Jenkins server (number of HTTP requests)

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

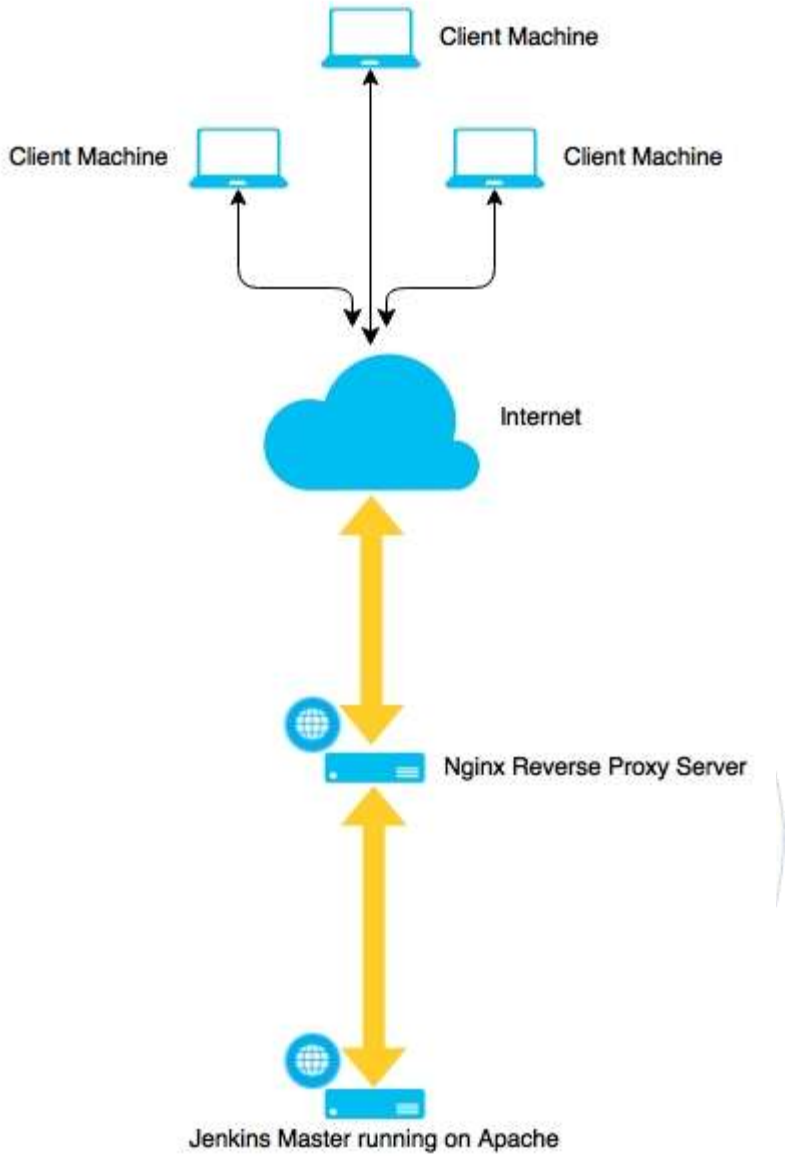
- **Approach 1**: Using multiple Jenkins masters, one each for every project
- **Approach 2**: Maintaining a single Jenkins master on a web server, with an enhanced hardware and behind a reverse proxy Server

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

  The following image shows **Approach 1**:



The following image demonstrates **Approach 2**:


Client Machine
Client Machine
Client Machine
Internet
Nginx Reverse Proxy Server
Jenkins Master running on Apache

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 Ubuntu

To install the latest stable 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 - 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
```

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/

Note

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:

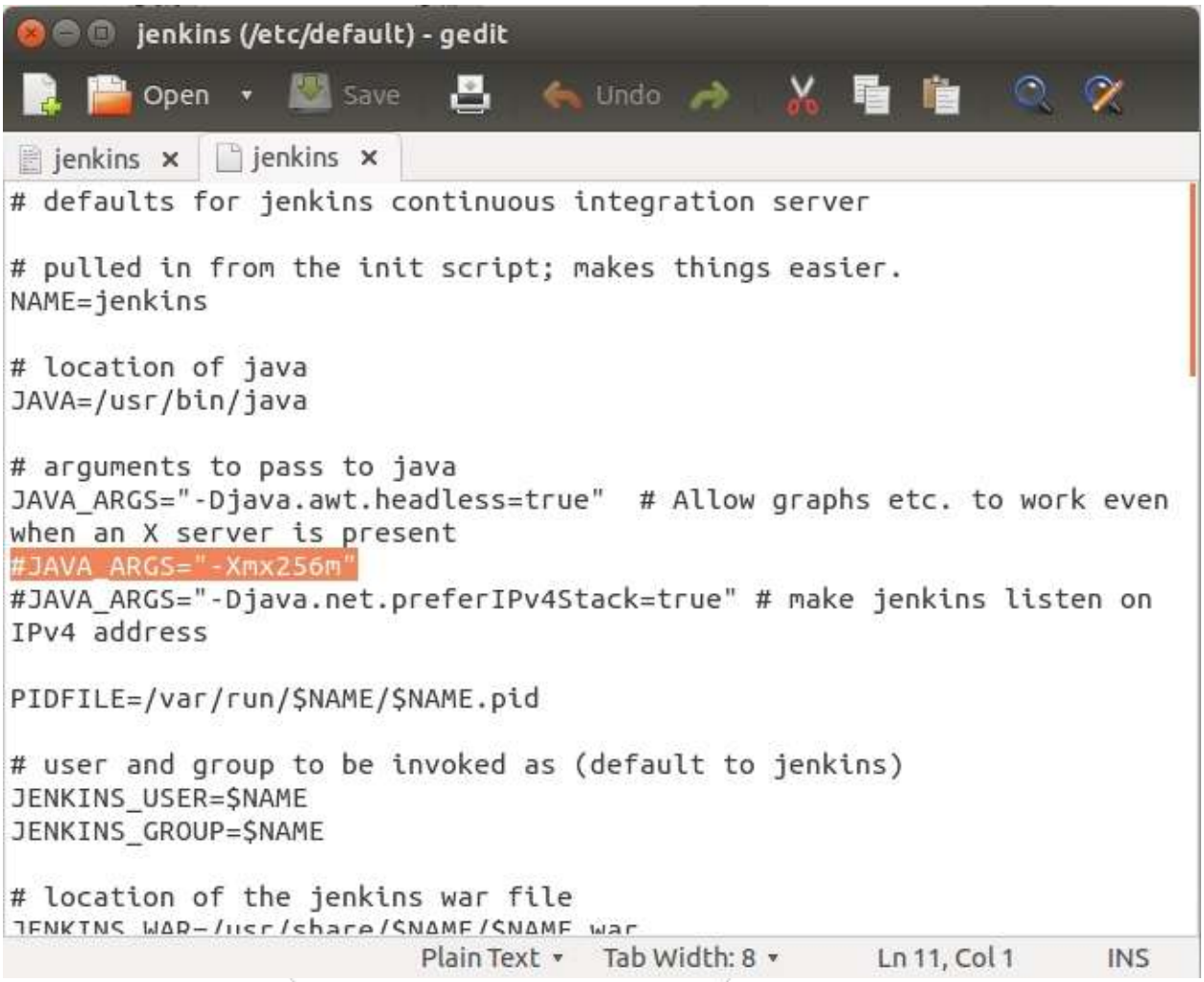
1. In order to change the Jenkins port, open the `jenkins` file present inside `/etc/default/`.

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/Centos

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

Note

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

Installing the latest stable version of Jenkins

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
```

Note

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/`.

Tip

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:

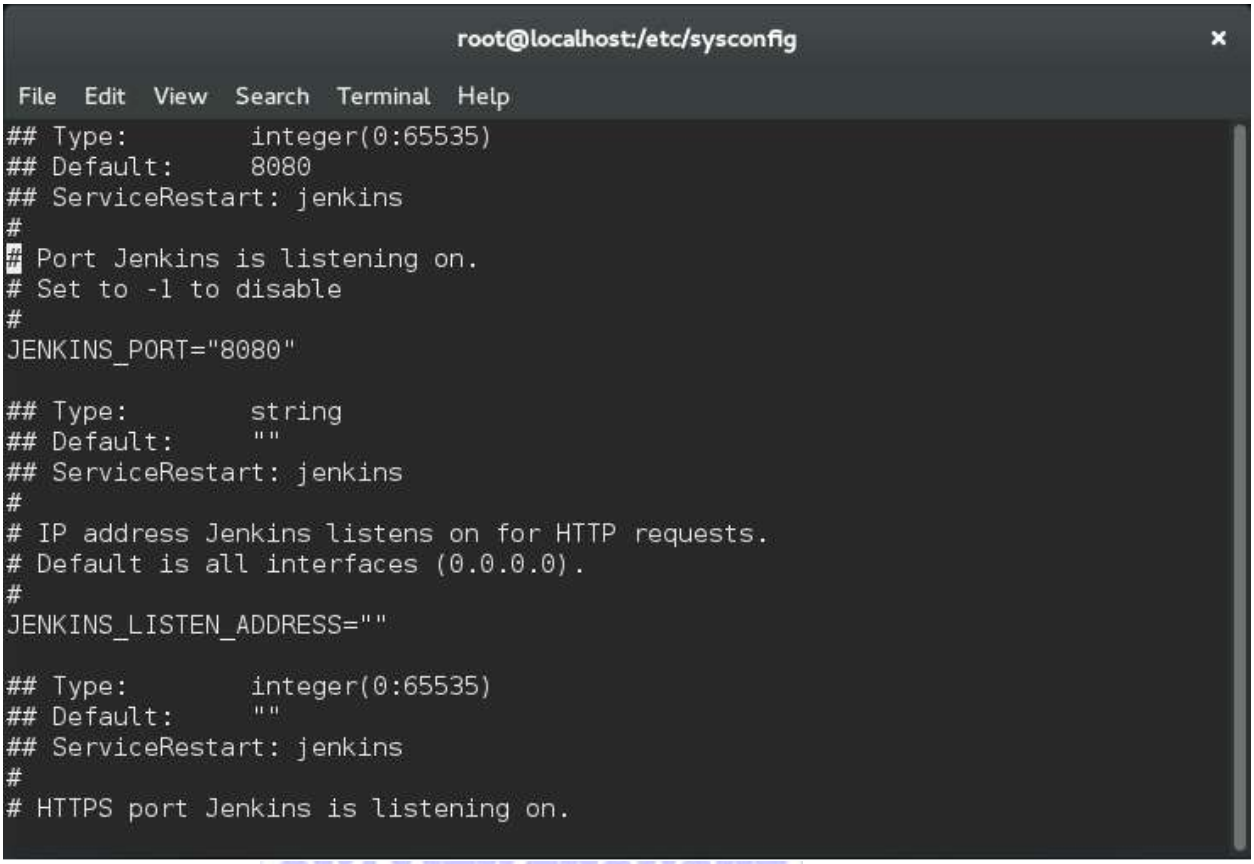
1. Open the terminal in Fedora.
2. Switch to the admin account using the following command:

```
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

Note

`$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.

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

Creating your first Jenkins job

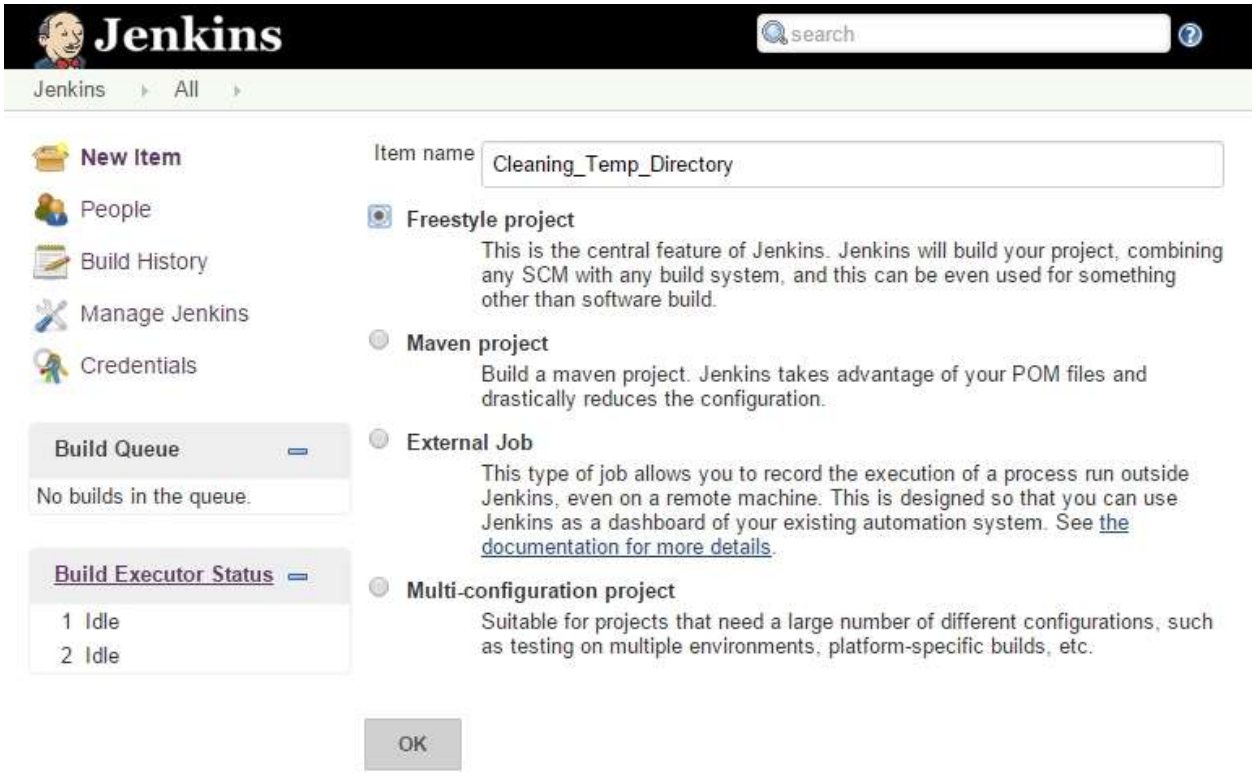
In the current section, we will see how to create a Jenkins Job to clean up the `%temp%` directory on our Windows machine where the Jenkins master server is running. We will also configure it to send an e-mail notification. We will also see how Jenkins incorporates variables (Jenkins system variable 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.

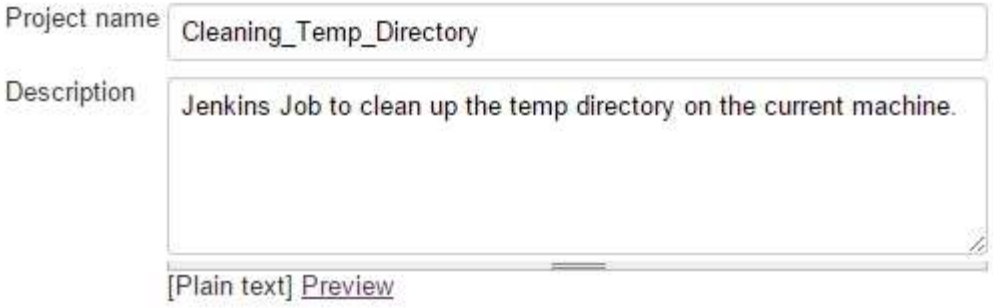
Note

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.

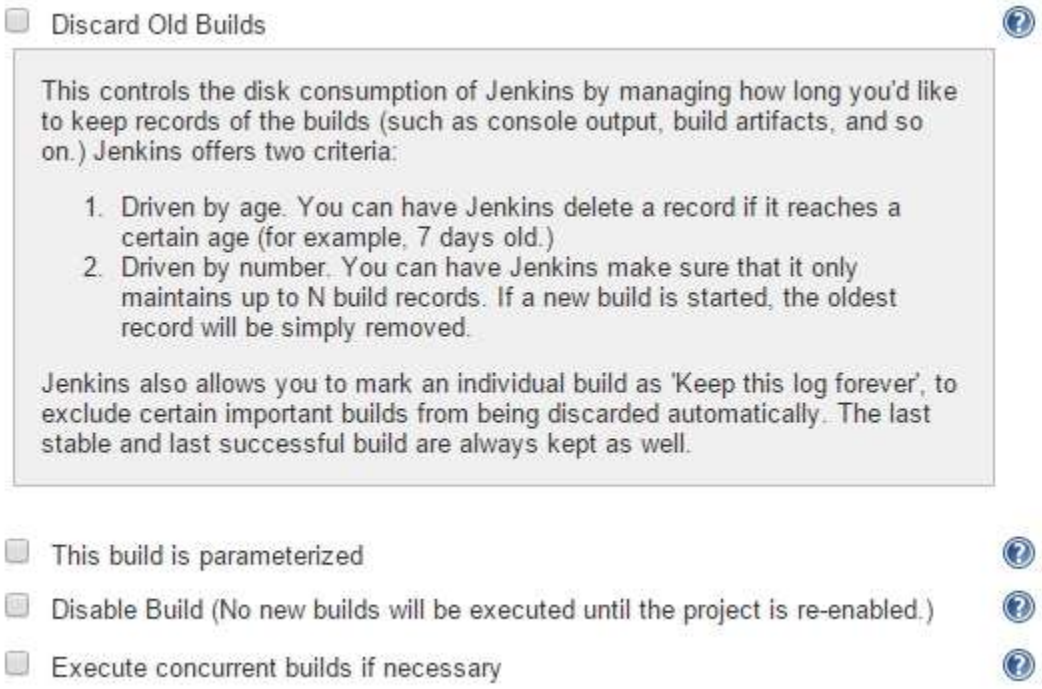
Below the **Item name** field, there are four options to choose from: **Freestyle project**, **Maven project**, **External Job**, and **Multi-configuration project**. These are predefined templates, each having various options that define the functionality and scope of the Jenkins job. All of them are self-explanatory.

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

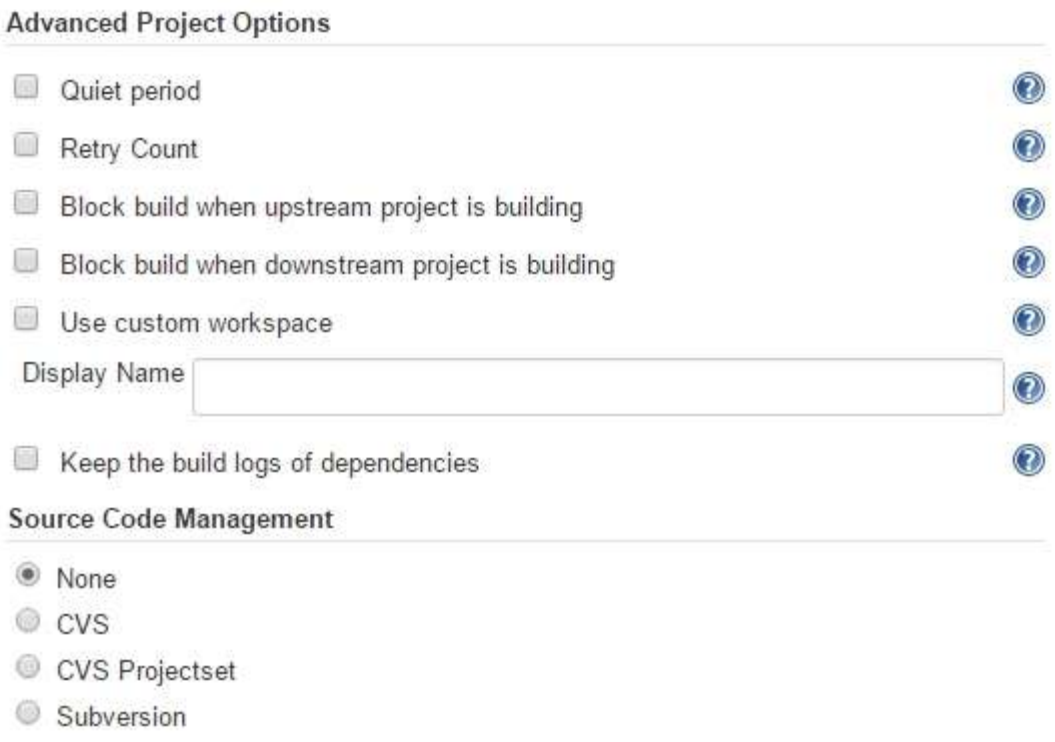
7. Below that, we have the option to add some description about our Jenkins job. I added one for our Jenkins job.



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:

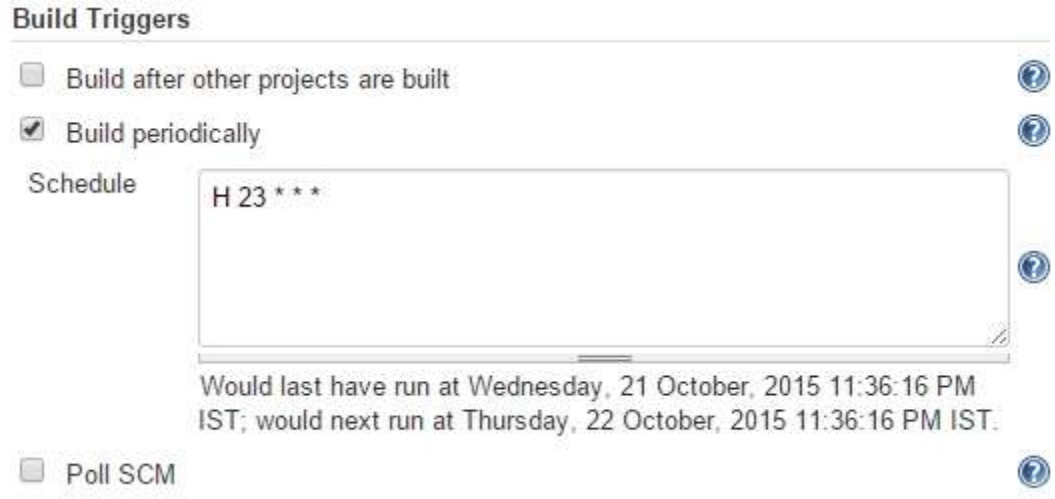


9. Scrolling down further, you will see the **Advanced Project Options** section and the **Source Code Management** section. Skip them for now as we don't need them.



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.



Note

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.

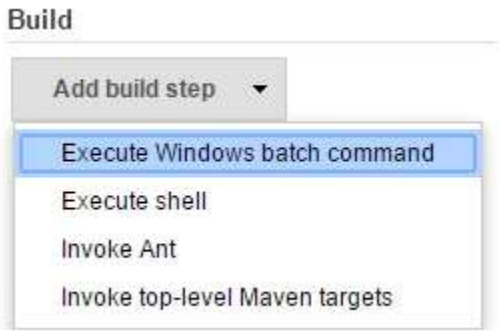
You might ask the significance of the symbol `H` in place of the minute. Imagine a situation where you have more than 10 Jenkins jobs scheduled for the same time, say `59 23 * * *`. There is a chance Jenkins will get overloaded when all the Jenkins jobs start at once. To avoid this, we use an option `H` in the minute place. By doing so, Jenkins starts each job with a gap of 1 minute.

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:

Note

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:

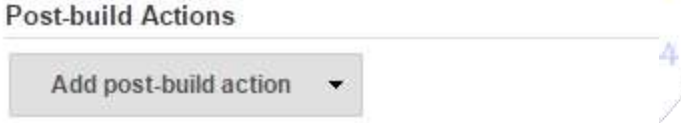
   The following screenshot displays the **Command** field in the **Execute Windows batch command** option:

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.

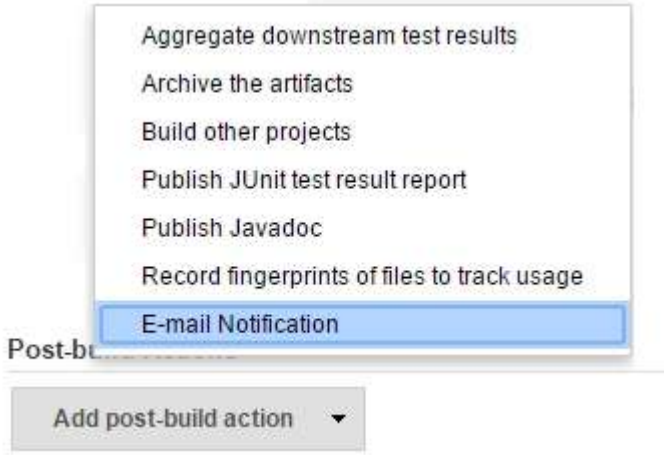
Adding post-build actions

Perform the following steps to add post-build actions:

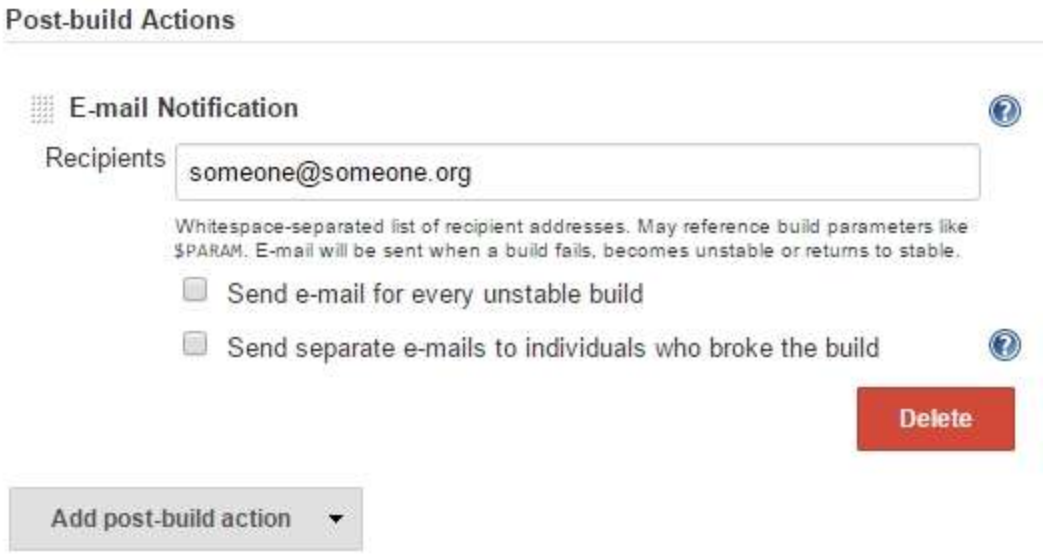
1. Scroll down further and you'll come across the **Post-build Actions** option.



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.

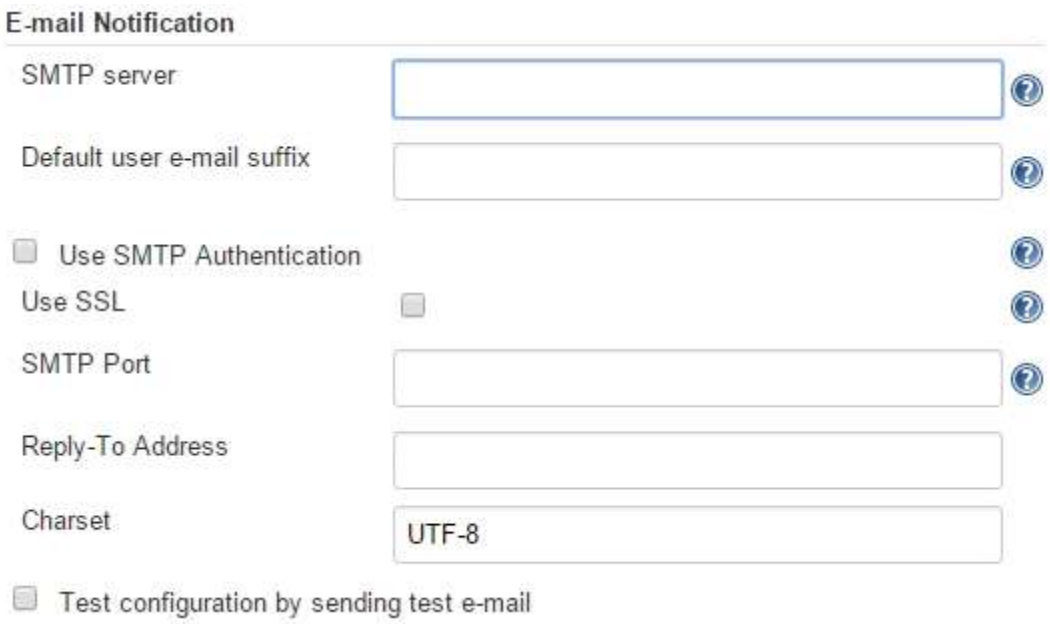


4. There are a few options under the **E-mail Notification** section that can be ignored for now. Nevertheless, you can explore them.
5. Click on the **Save** button, present at the end of the page, to save the preceding configuration. Failing to do so will scrap the whole configuration.

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.

Note

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.

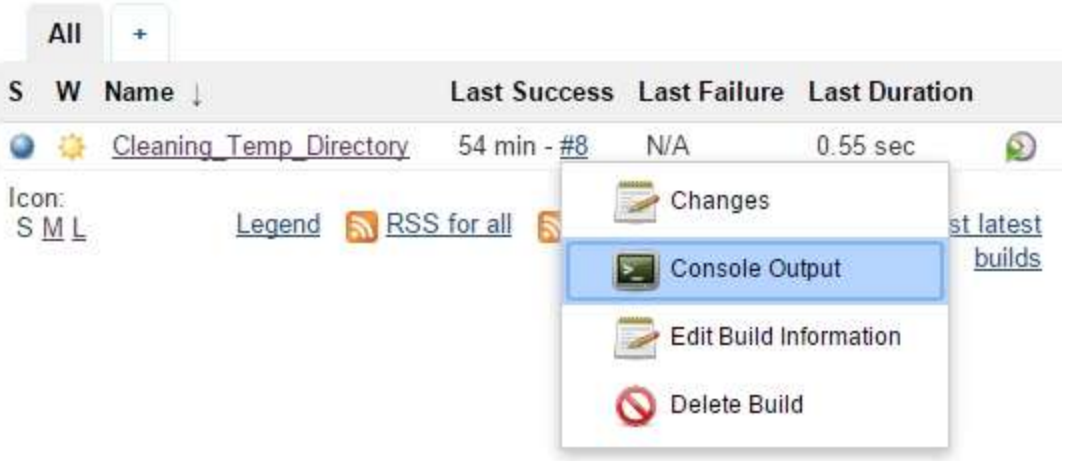
3. Click on the **Build** button to run the job. If everything is right, the job should run successfully.

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
```

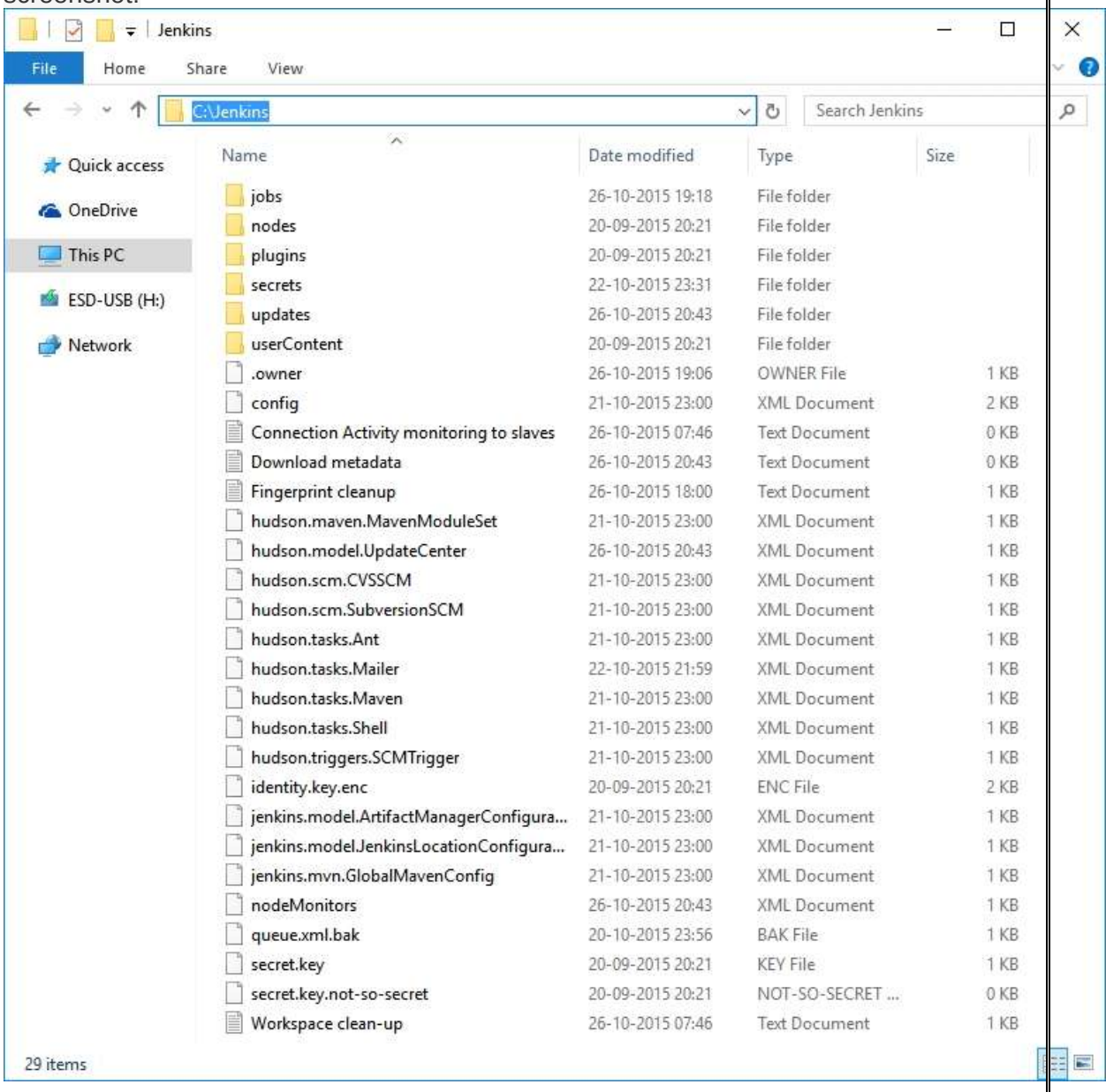
Note

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

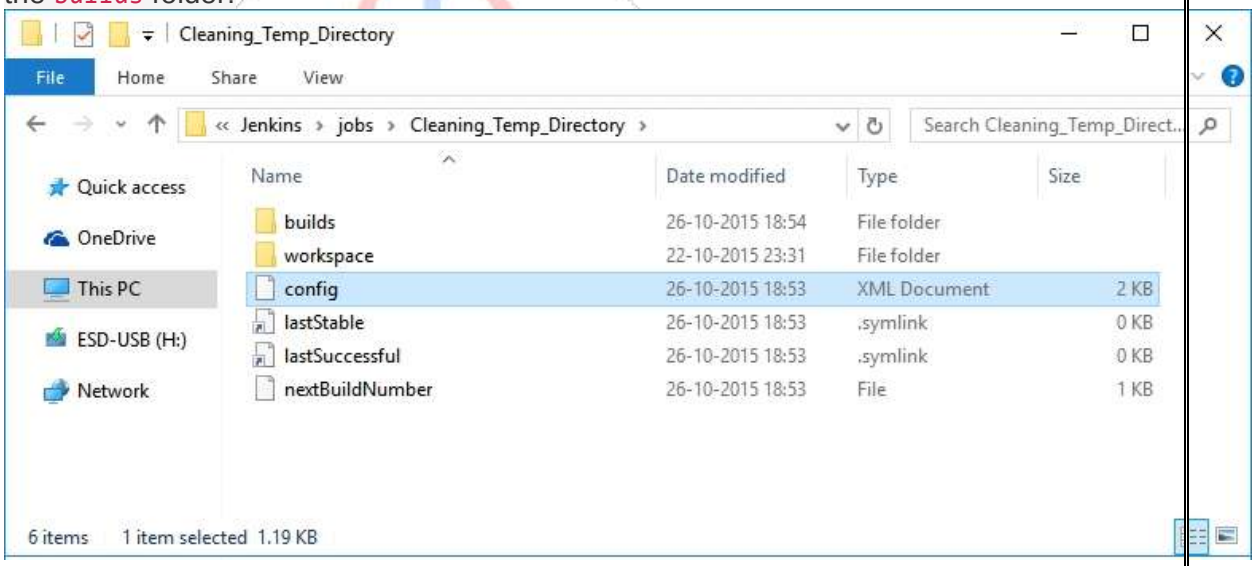
Jenkins home directory

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:

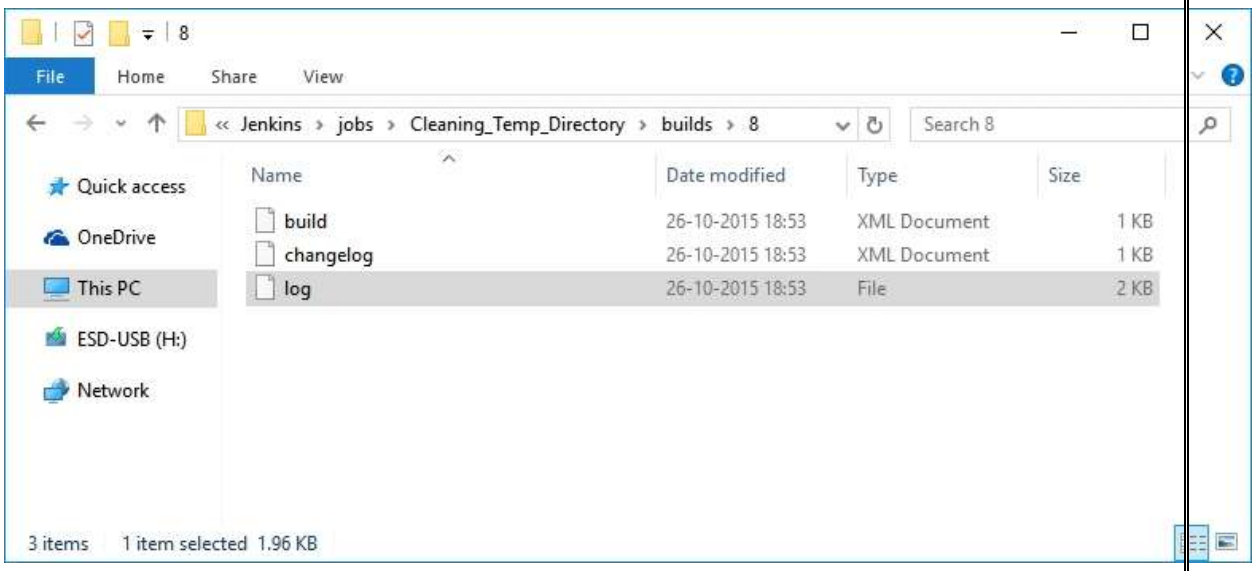
1. Go to `C:\Jenkins\`, our Jenkins home path. This is the place where all of the Jenkins configurations and metadata is stored, as shown in the following screenshot:



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.
    a. 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.
    b. 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.
    c. 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.
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:

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
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. 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
16. Select the **Console Output** option. This will take you to the log page.
17. From Windows Explorer, go to the `C:\Jenkins_Backup` directory. We can see that the backup archive has been created.

Restoring a Jenkins backup

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.

Note

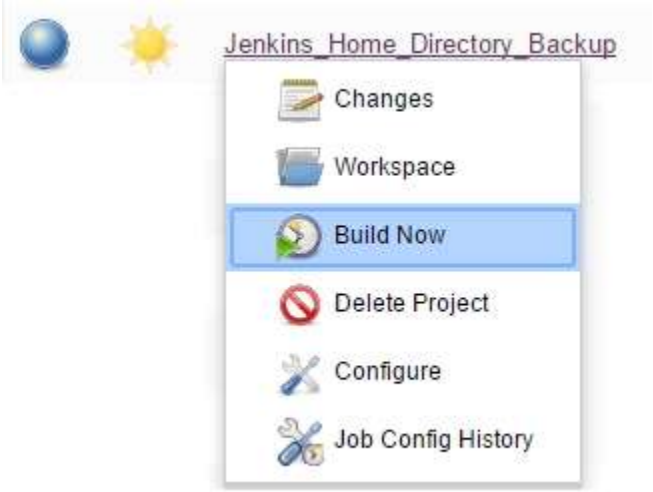
It is recommended not to update Jenkins until and unless you need to. For example, upgrade Jenkins to an LTS release that contains a bug fix that you need desperately.

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**.



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.

Upgrading standalone Jenkins master running on Ubuntu

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:

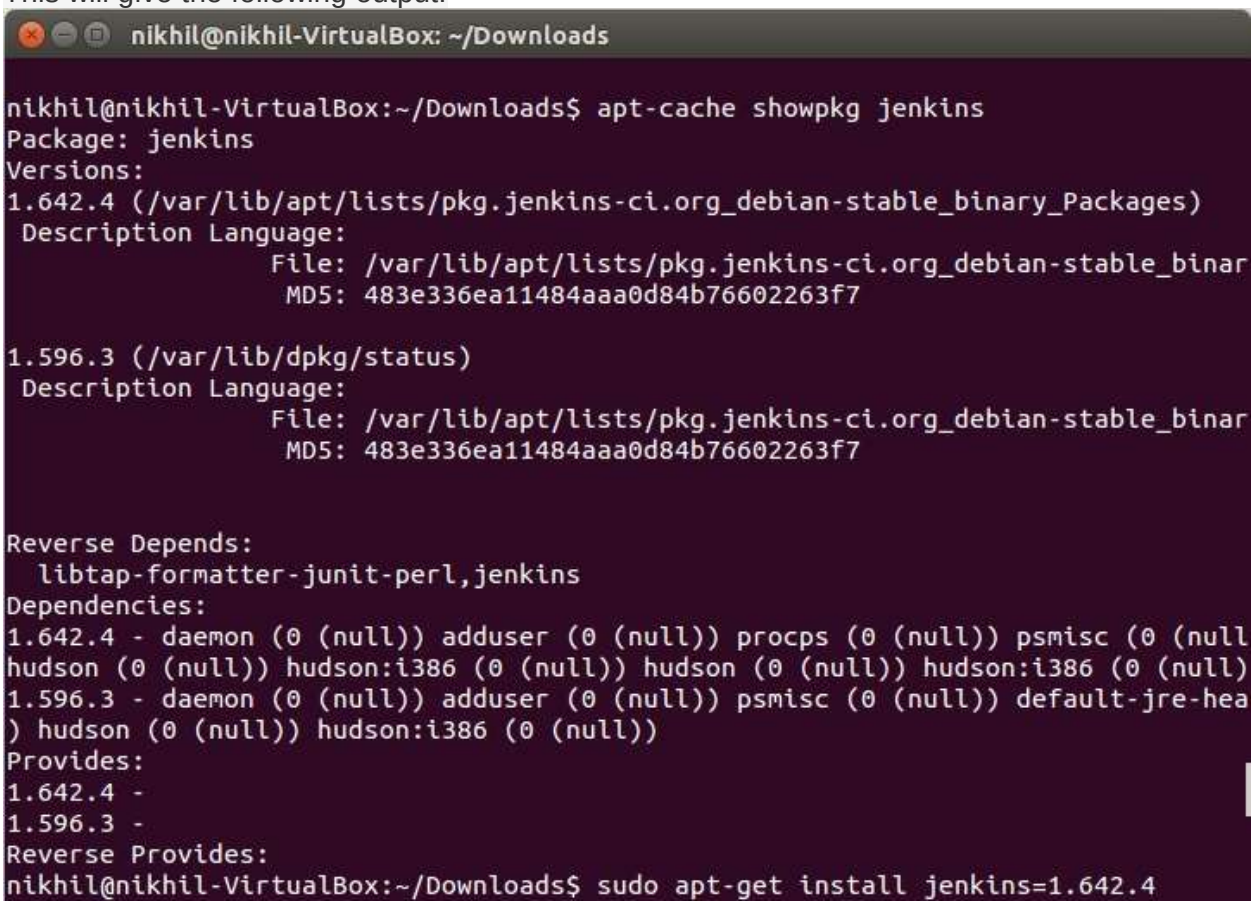
```
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

```
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 showpk
g 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:

10. Run the following command:

```
sudo apt-get –f install
```

11. This will give the following output:

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.

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.

Managing Jenkins plugins

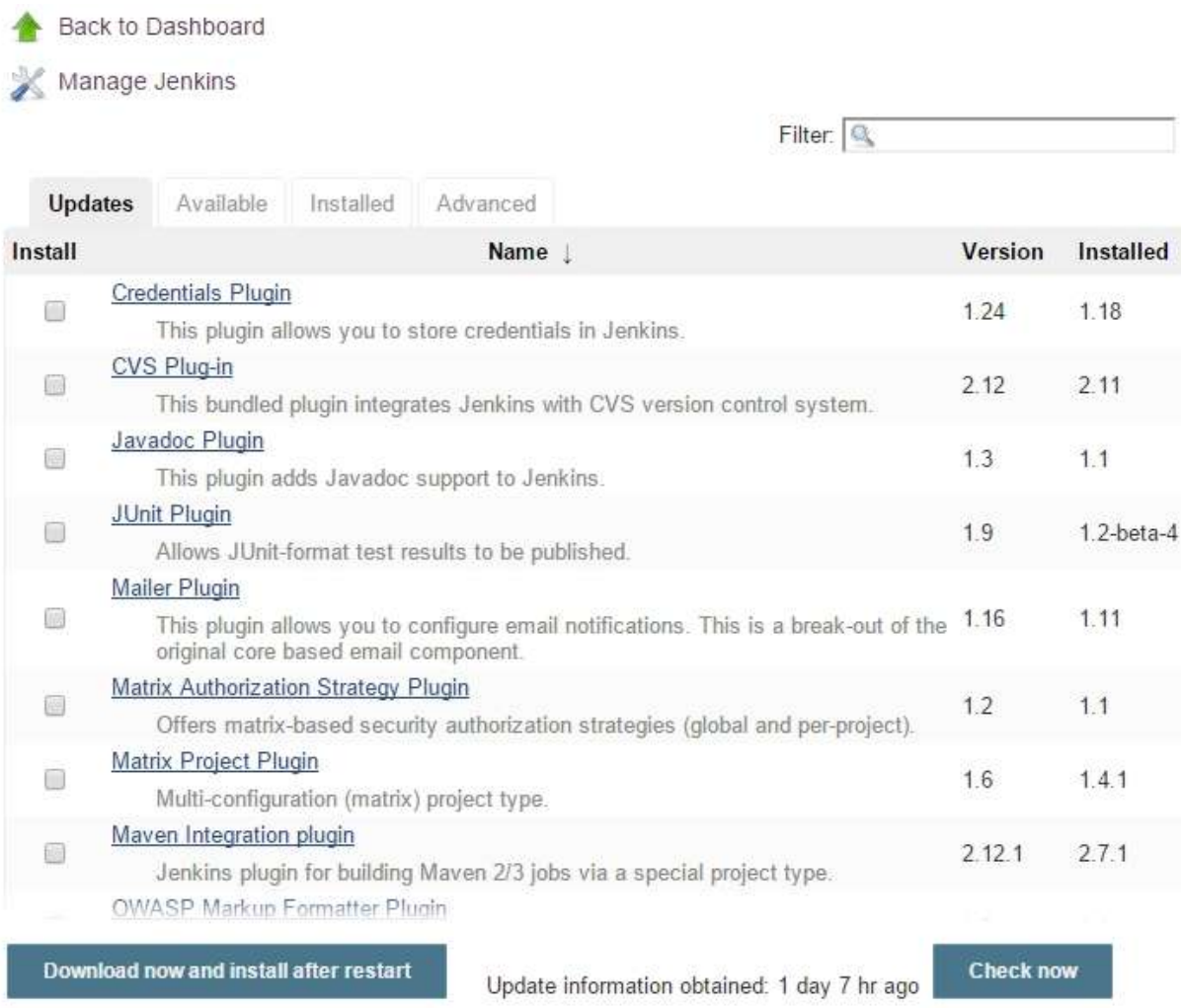
Jenkins derives most of its power from plugins. As discussed in the previous chapter, every plugin that gets installed inside Jenkins manifests itself as a parameter, either inside Jenkins system configurations or inside a Jenkins job. Let's see where and how to install plugins.

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:

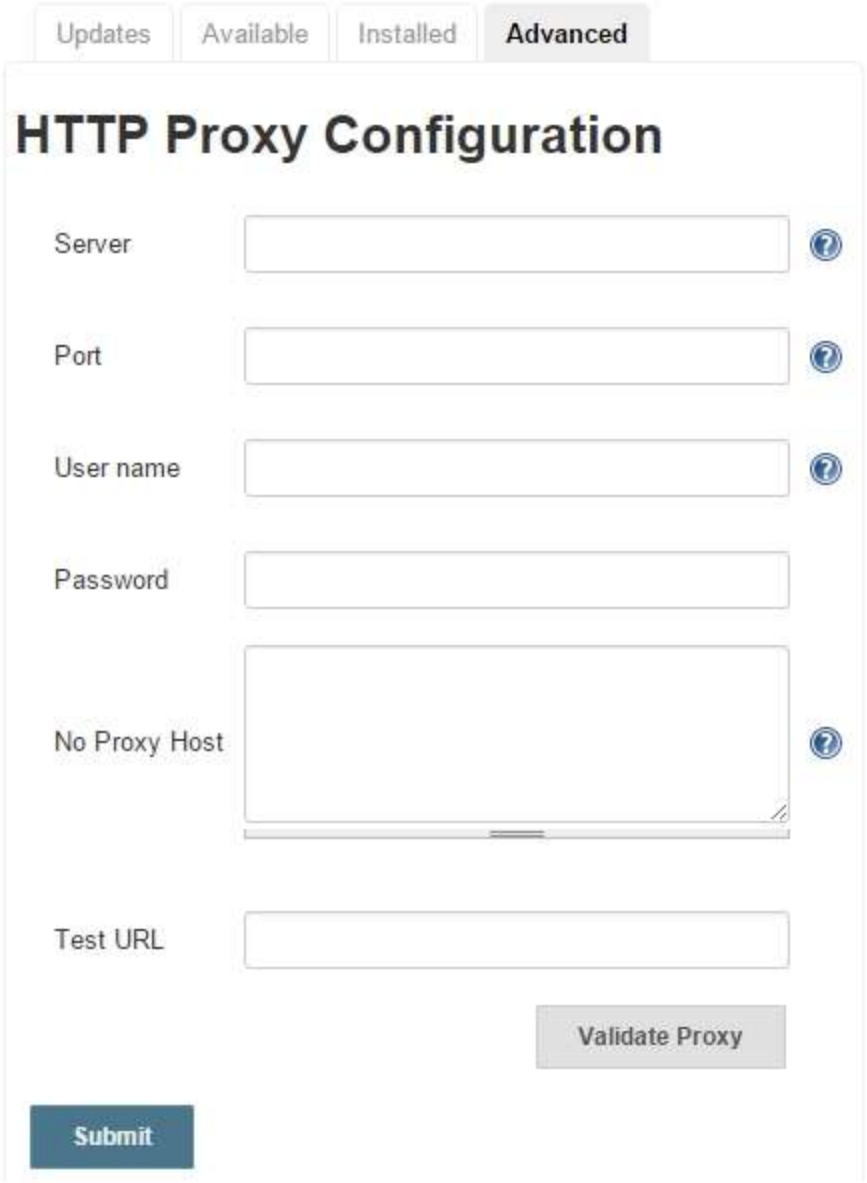
1. From the Jenkins Dashboard, click on the **Manage Jenkins** link.
2. From the **Manage Jenkins** page, click on the **Manage Plugins** link.
3. The following screenshot is what you see when you land on the Jenkins **Plugin Manager** page.



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.

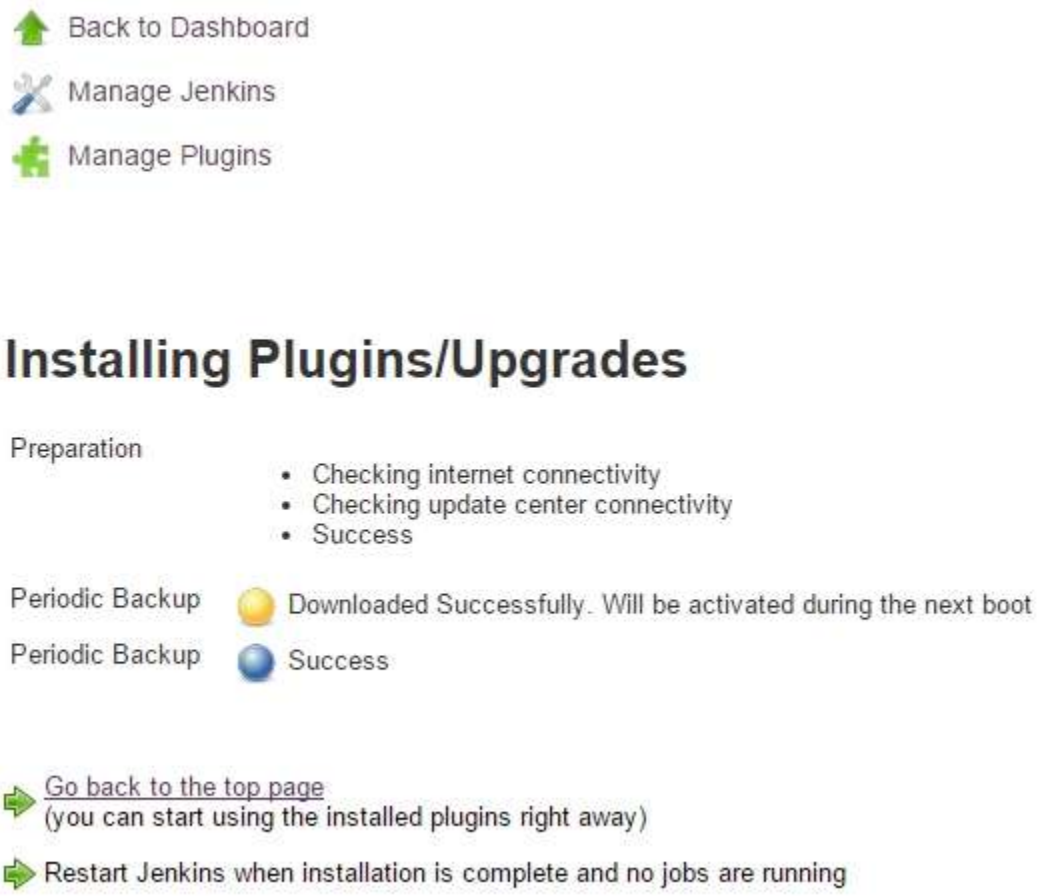


8. Just below the **HTTP Proxy Configuration** section, you will see the **Upload Plugin** section. It provides the facility to upload and install your own Jenkins plugin.

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:



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.

/jenkins	*None specified*	Jenkins v1.642.3	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes

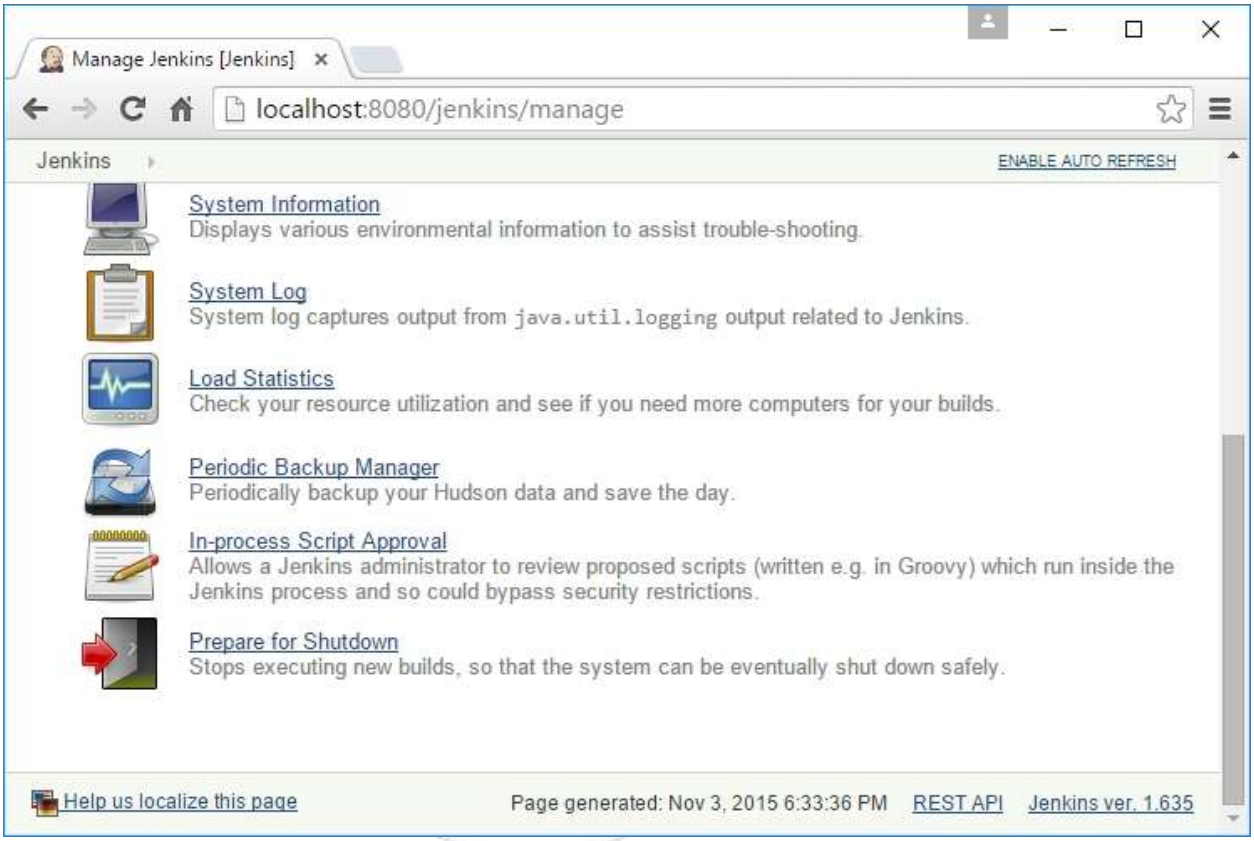
Configuring the periodic backup plugin

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.

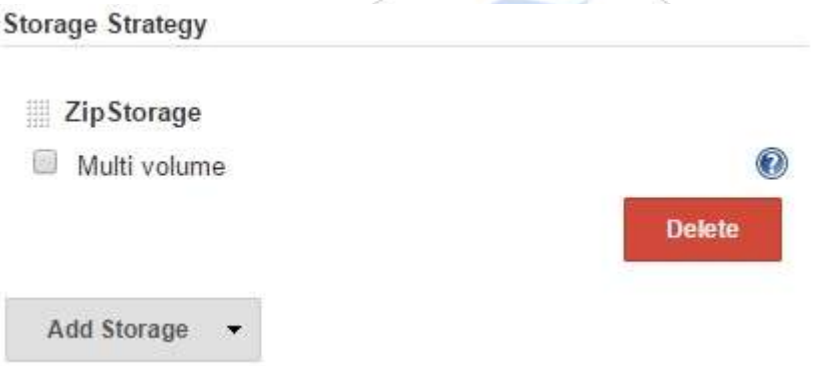
7. Scroll down to the **File Management Strategy** section. You will see the options to choose from **FullBackup** and **ConfigOnly**. Choose **FullBackup**.

Note

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.

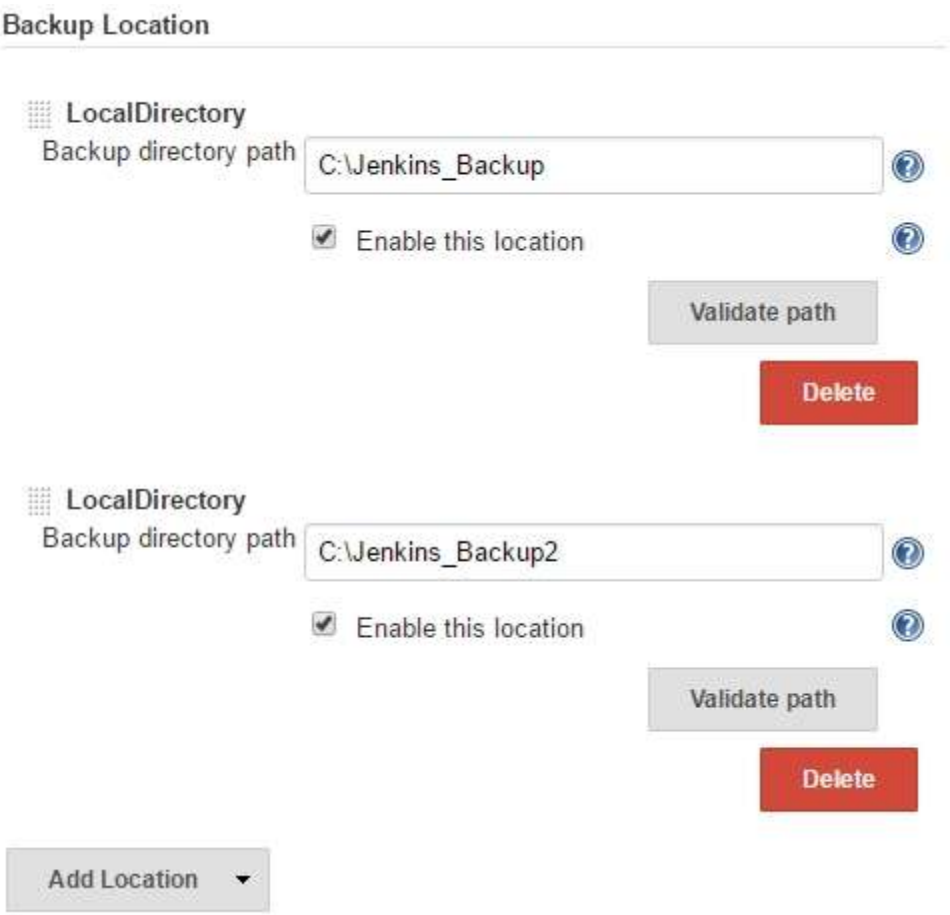
9. Clicking on the **ZipStorage** strategy provides us an option to select the **Multi volume** zip file, that is, one huge, single zip file split into many.



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.

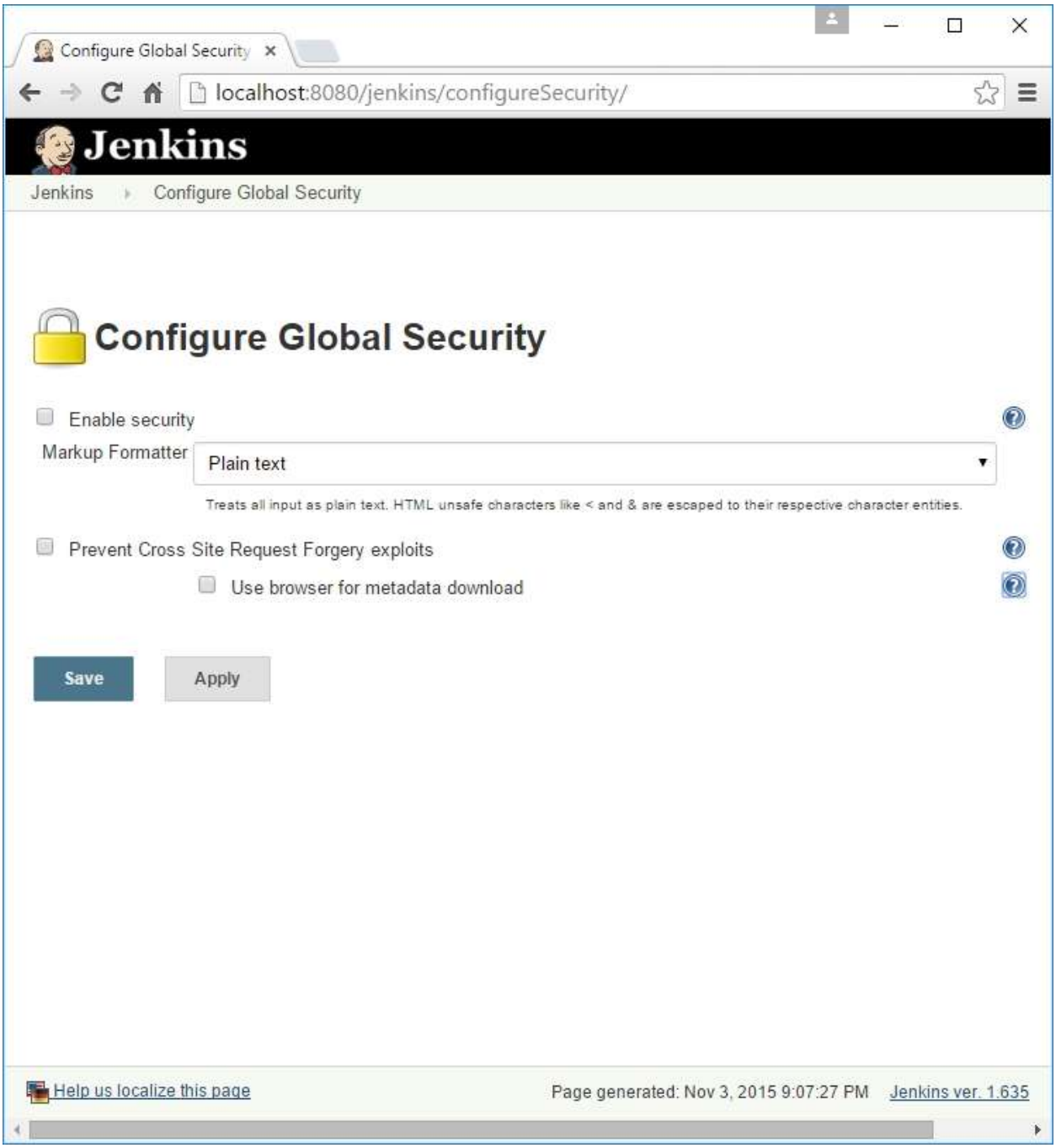
User administration

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.
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.

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.

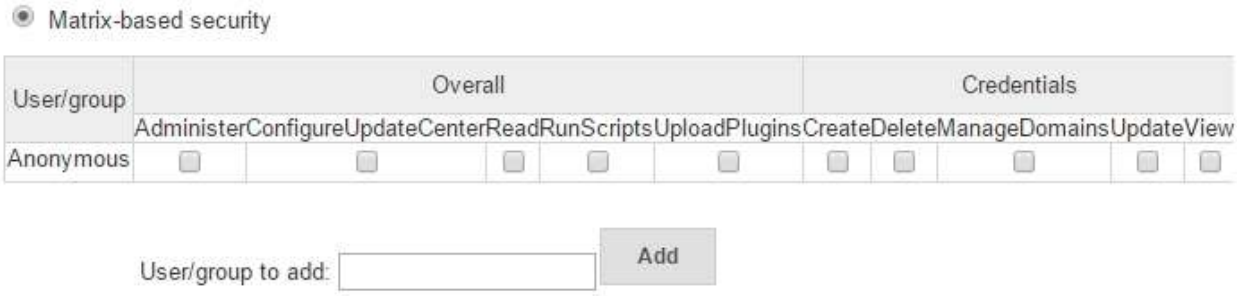
8. Select the **Jenkins' own user database** and you will get another option, which allows users to sign up. This is shown in the following screenshot:

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.

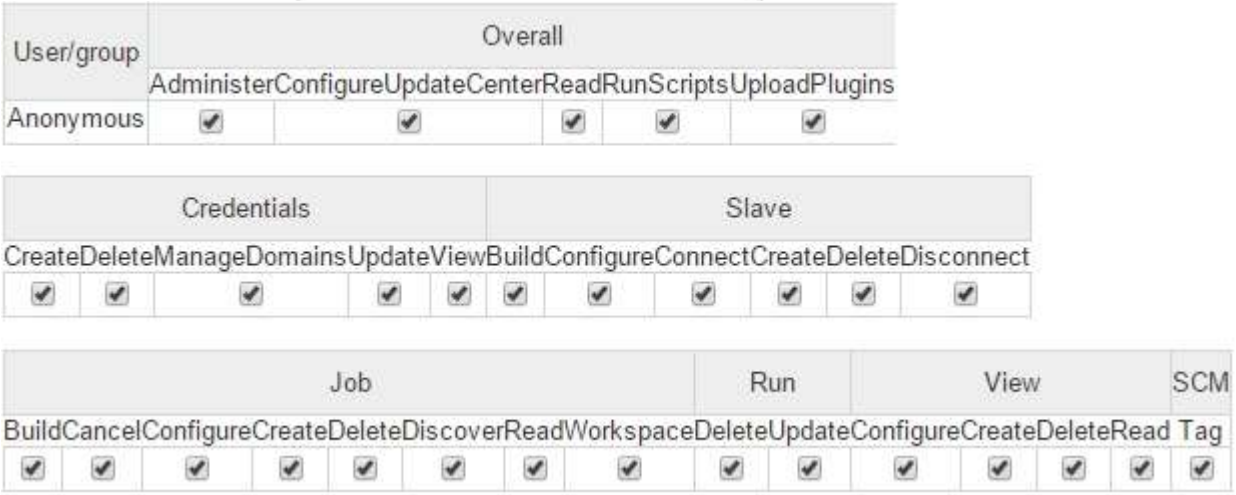
12. To add users, enter the user names in the **User/group** to add a field, and click the **Add** button. For now, do not add any users.



Note

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.



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

Creating other users

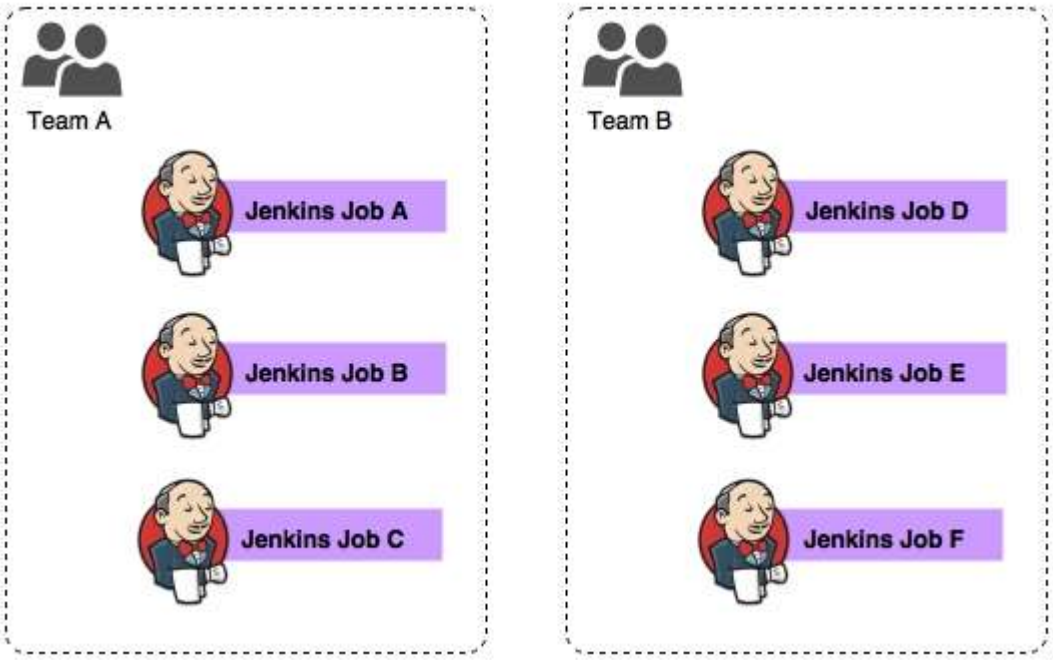
Users can always sign up and create their account in Jenkins using the **sign up** link at the top-right corner. All such users by default get all the privileges of an anonymous group.

1. You can try creating as many accounts as you want and see all those come under the anonymous category by default.
2. To see the list of Jenkins users, from the Jenkins Dashboard, click on the **People** link present at the top-left section.
3. All the users are listed on the **People** page, as shown in the following screenshot:
4. To give permissions to our newly created user, log into Jenkins as the `admin` user.
5. From the **Manage Jenkins** page, go to the **Configure Global Security** page.
6. On the **Configure Global Security** page, scroll down to the **Authorization** section.
7. 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 <`username`> that I recently created.
8. Once added, give the new user permissions to **Build**, **Cancel**, **Workspace**, and **Read** a Jenkins jobs, as shown in the following screenshot:
9. Click on the **Save** button at the end of the page to save the settings.
10. 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

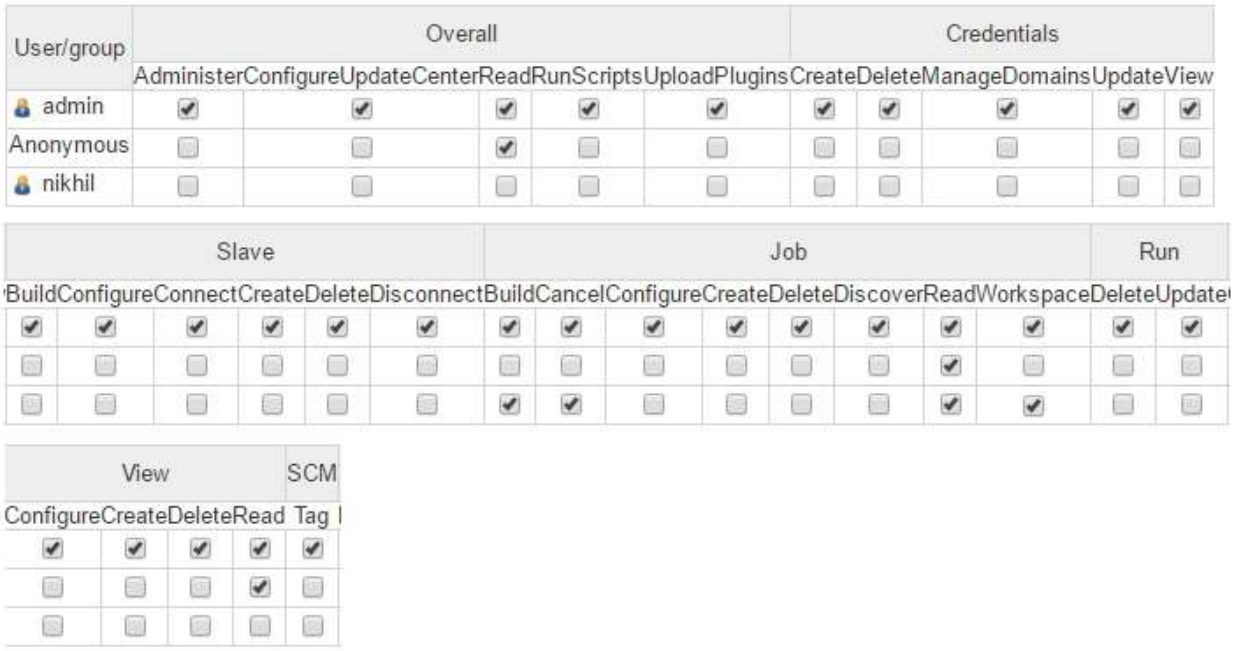
In the previous section, we saw the **Matrix-based security** authorization feature which gave us a good amount of control over the users and permissions. However, imagine a

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.

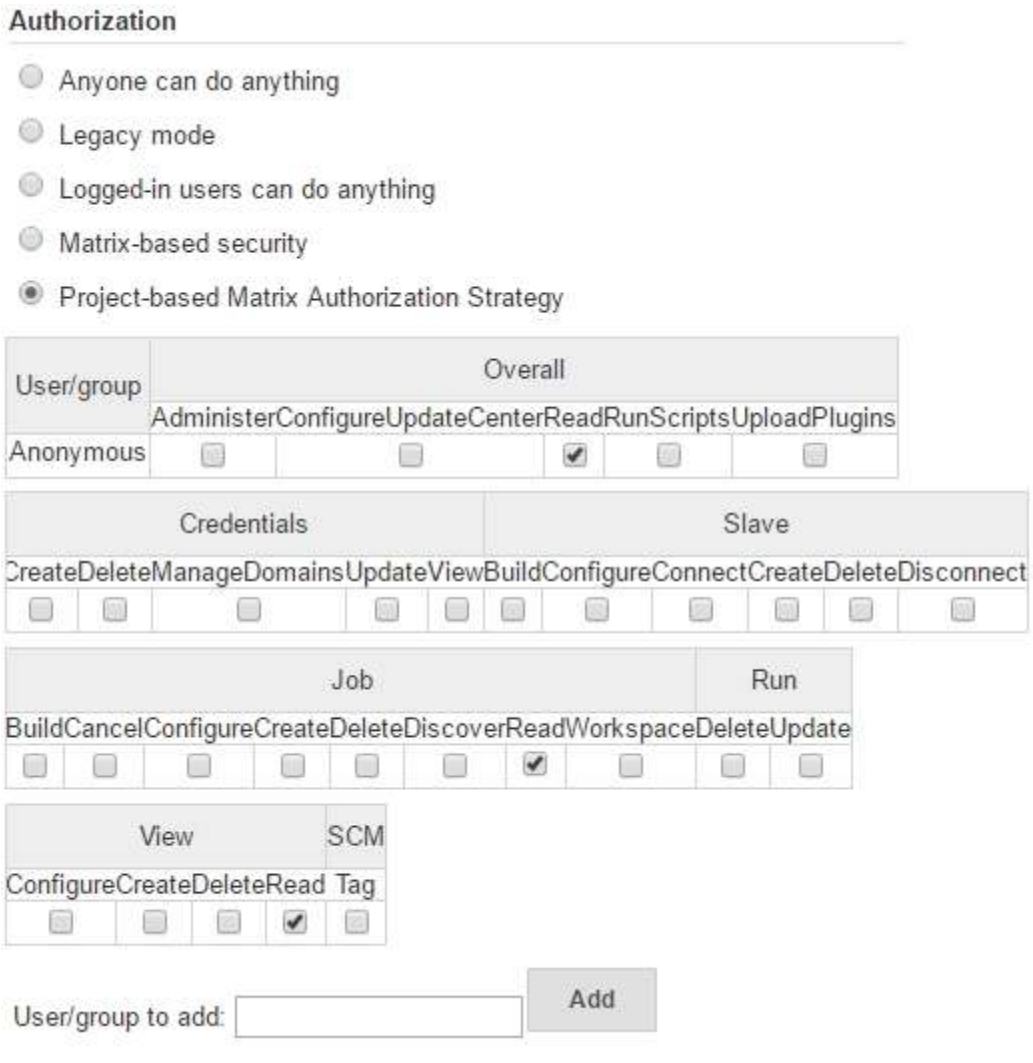


Let's learn to configure the **Project-based Matrix Authorization Strategy** feature:

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:



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:

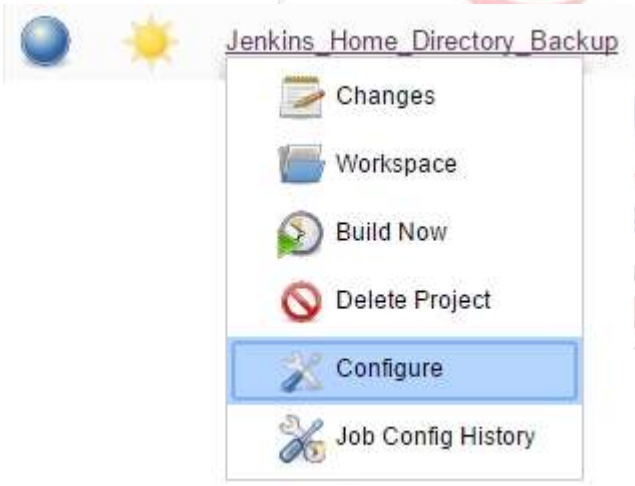
User/group	Overall					Credentials				
	Administer	Configure	Update Center	Read	Run Scripts	Upload Plugins	Create	Delete	Manage Domains	Update	View
Anonymous	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐
admin	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑

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

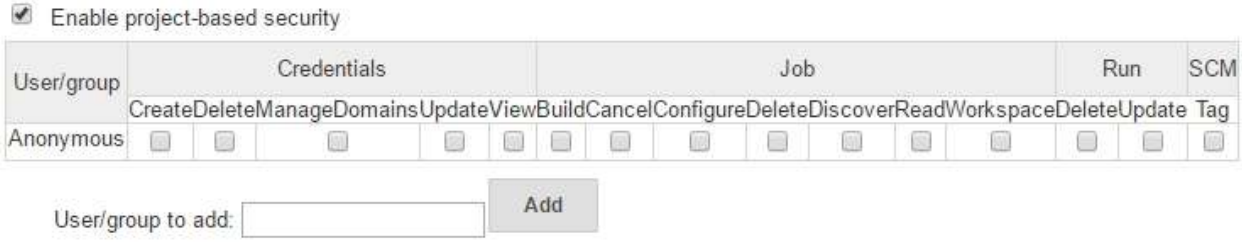
Run		View				SCM
Delete	Update	Configure	Create	Delete	Read	Tag
☐	☐	☐	☐	☐	☑	☐
☑	☑	☑	☑	☑	☑	☑

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.

4. Continuous Integration Using Jenkins – Part I

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.

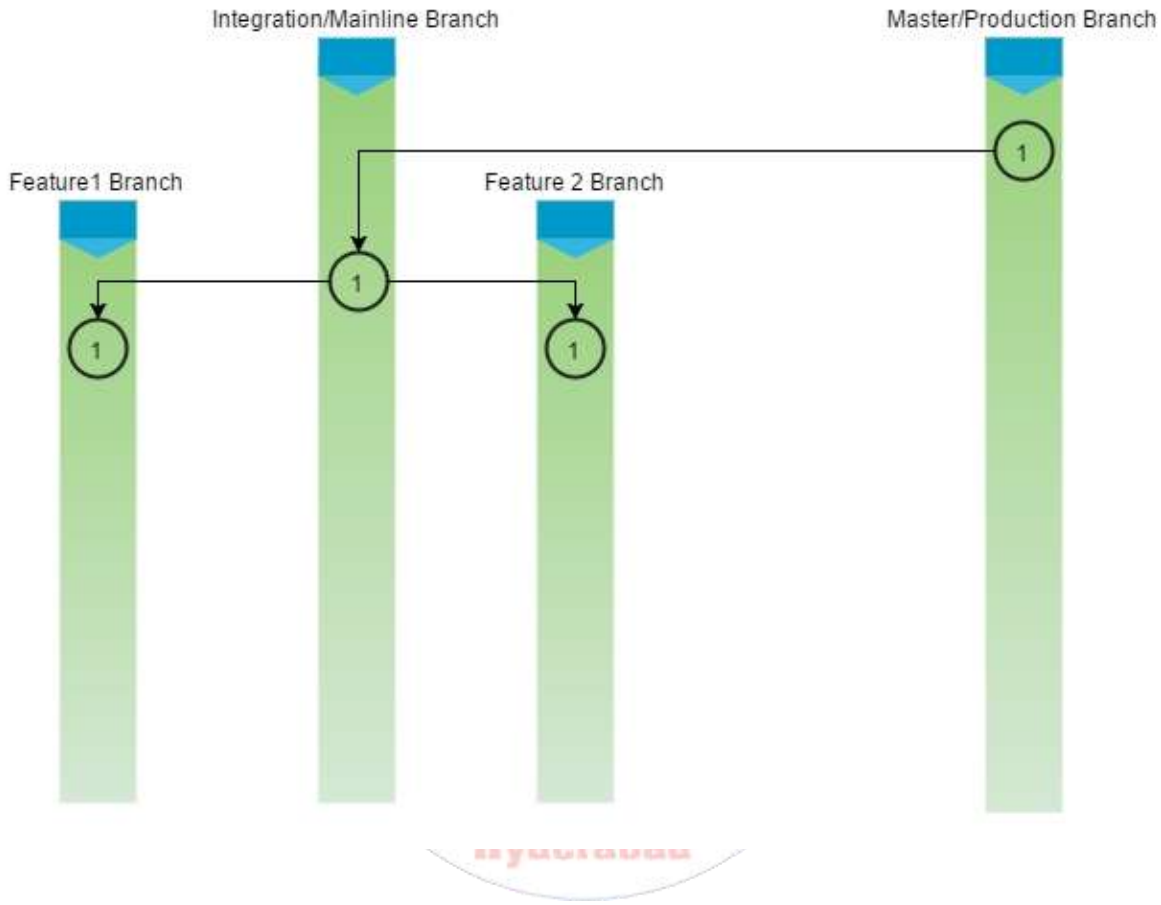
Integration branch

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

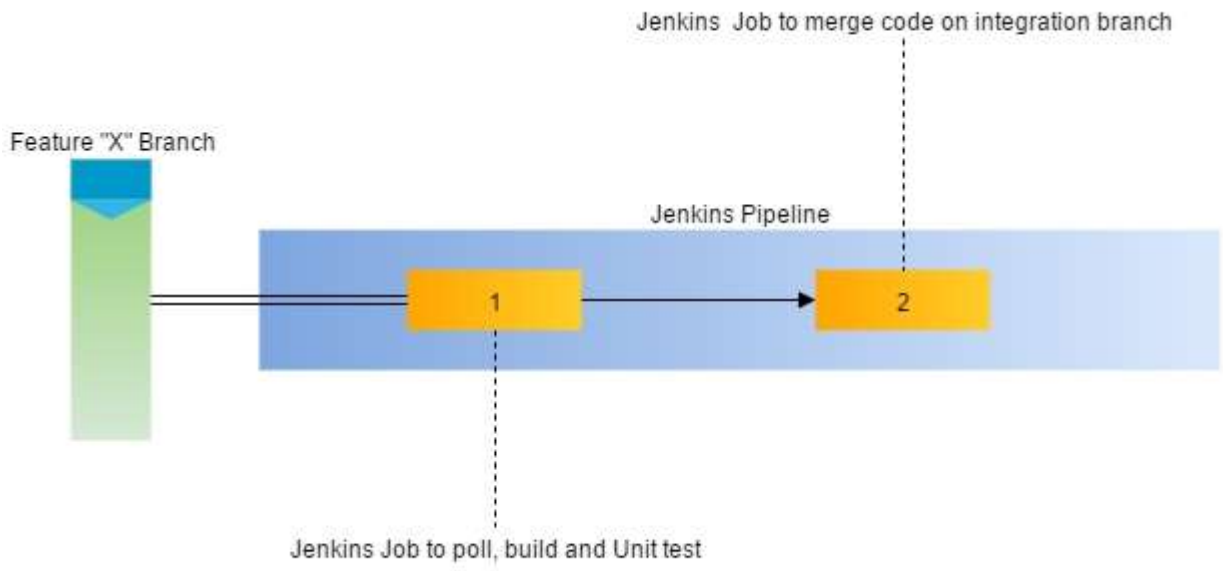
Jenkins job 2

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:

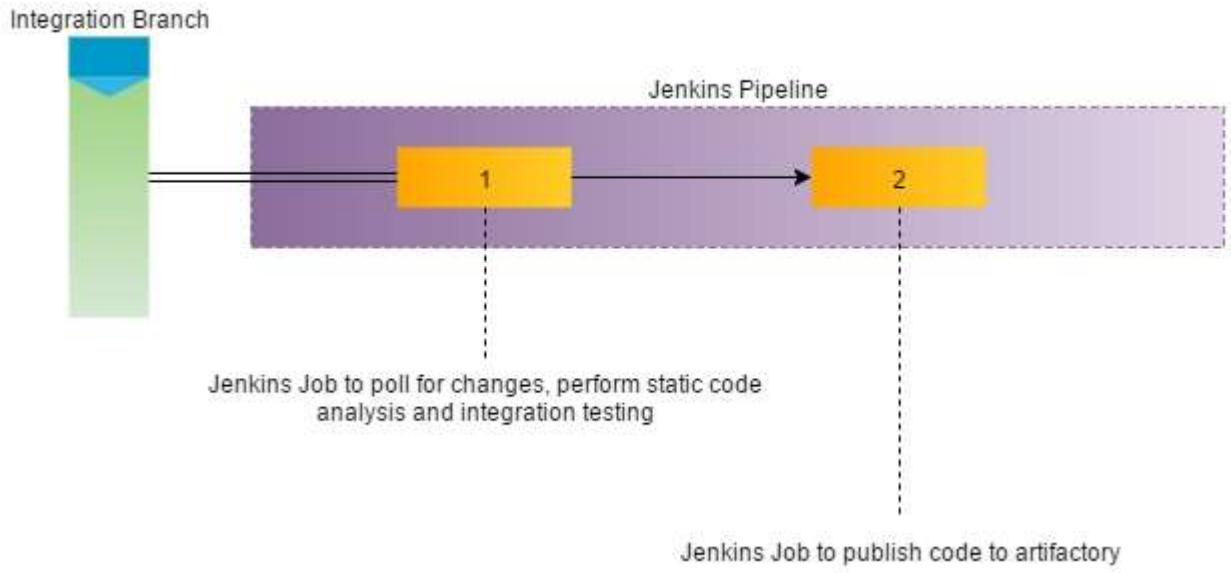
- It polls the integration branch for changes at regular intervals
- Performs static code analysis on the code

- 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/Nexus (binary code repository)



Note

- 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

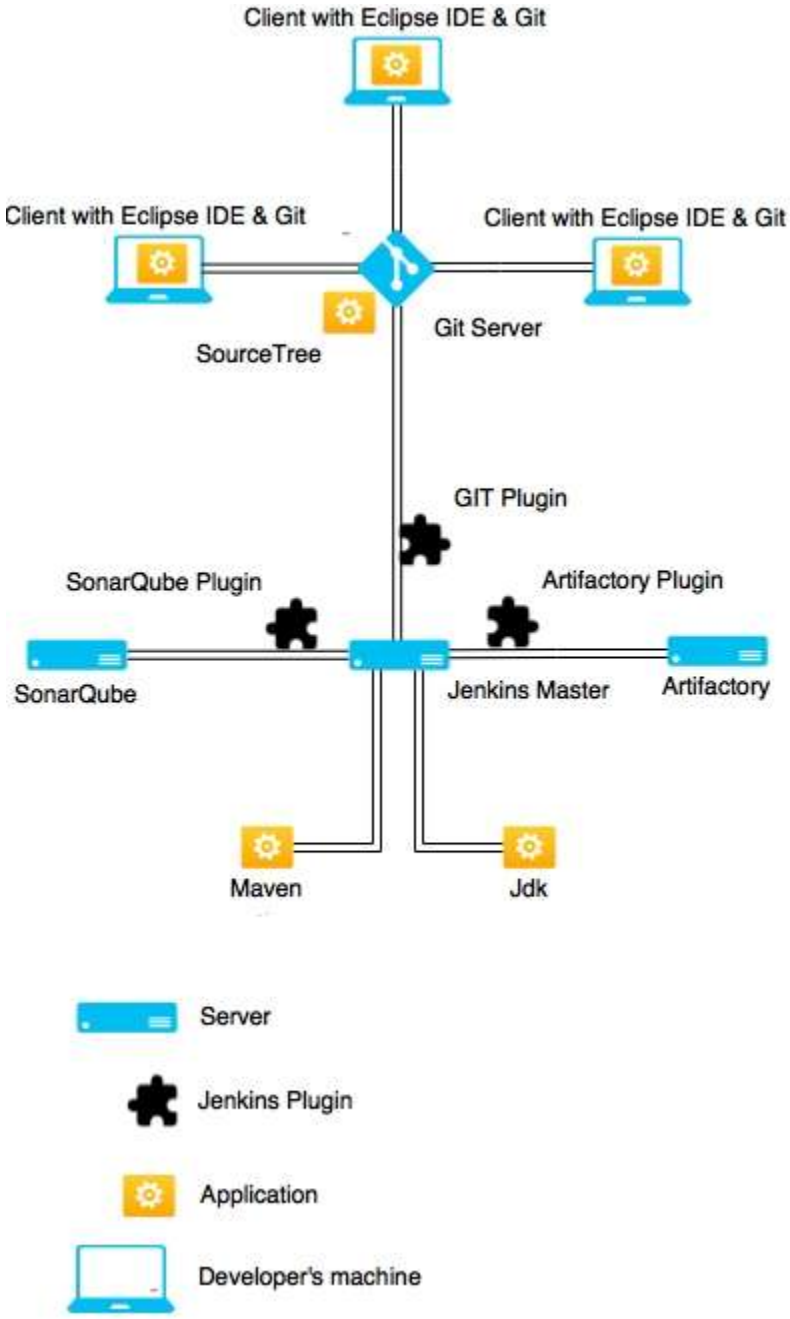
Toolset for Continuous Integration

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.

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.


Client with Eclipse IDE & Git
Client with Eclipse IDE & Git
Client with Eclipse IDE & Git
SourceTree
Git Server
GIT Plugin
SonarQube Plugin
Artifactory Plugin
SonarQube
Jenkins Master
Artifactory
Maven
Jdk
Server
Jenkins Plugin
Application
Developer's machine

Setting up a version control system

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/
2. Begin the installation by double-clicking on the downloaded executable file.
3. Click on the **Next** button. (Until it says finish)

Installing SourceTree (a Git client)

1. Download SourceTree from www.sourcetreeapp.com
2. Begin the installation by double-clicking on the downloaded executable file.
3. And finish the installation

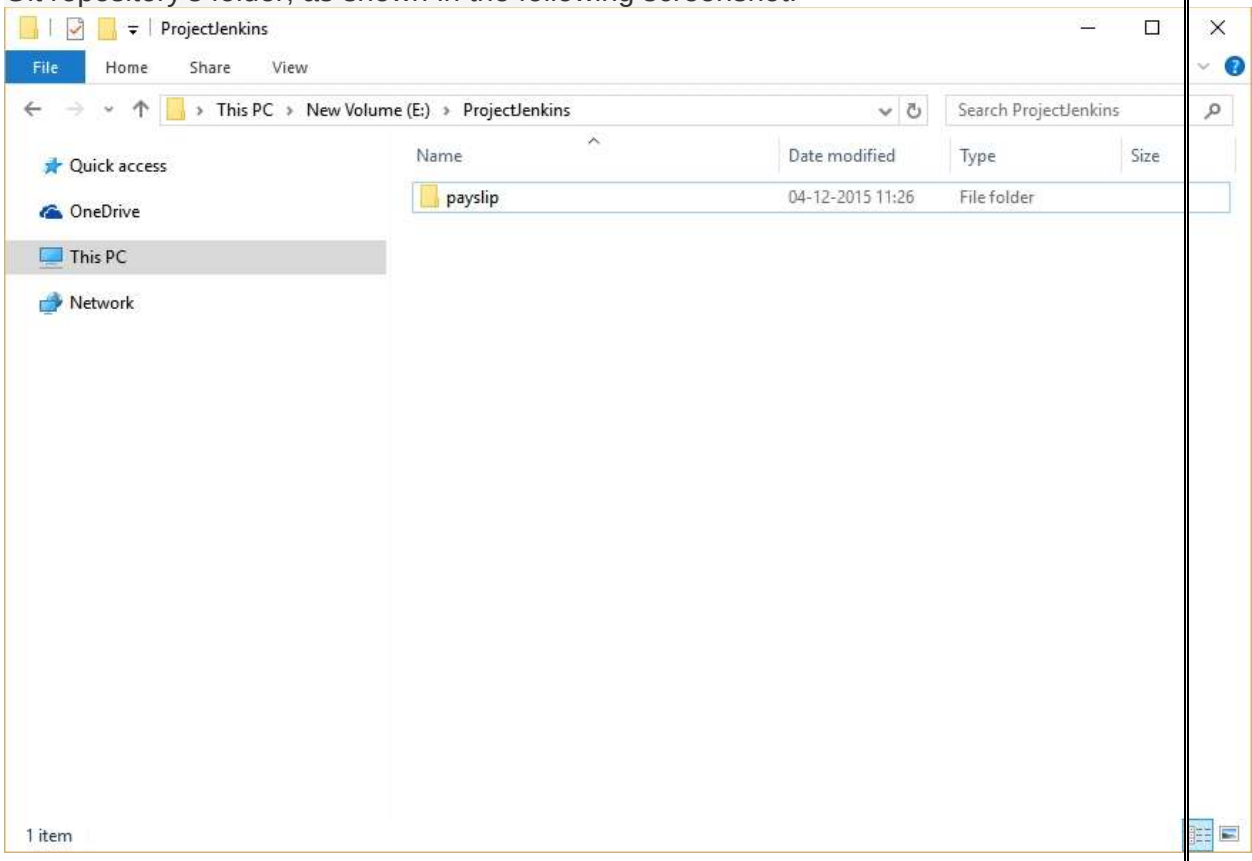
Creating a repository inside Git

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
```

Uploading code to Git repository

1. The code can be downloaded from the following GitHub repository: https://github.com/QualityThoughtTechnologies/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. Use the following command to add the code:

```
git add --all payslip
```

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

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

5. 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.

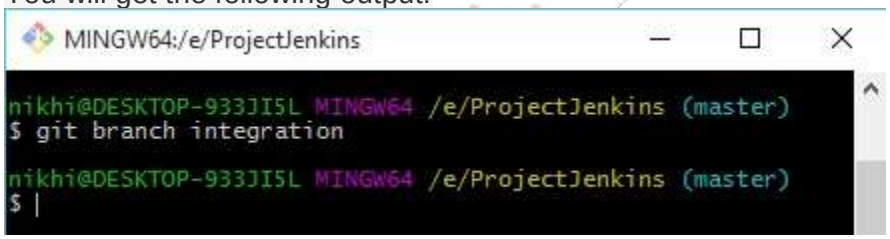
The master branch will remain neat and clean and only code that has been built and tested thoroughly will reside on it.

Using the Git commands

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
```

Note

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>` wi 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` message.
Rebase	
`git rebase -i`	Interactively rebase the current branch onto another branch named `<branch>`

Configuring Jenkins

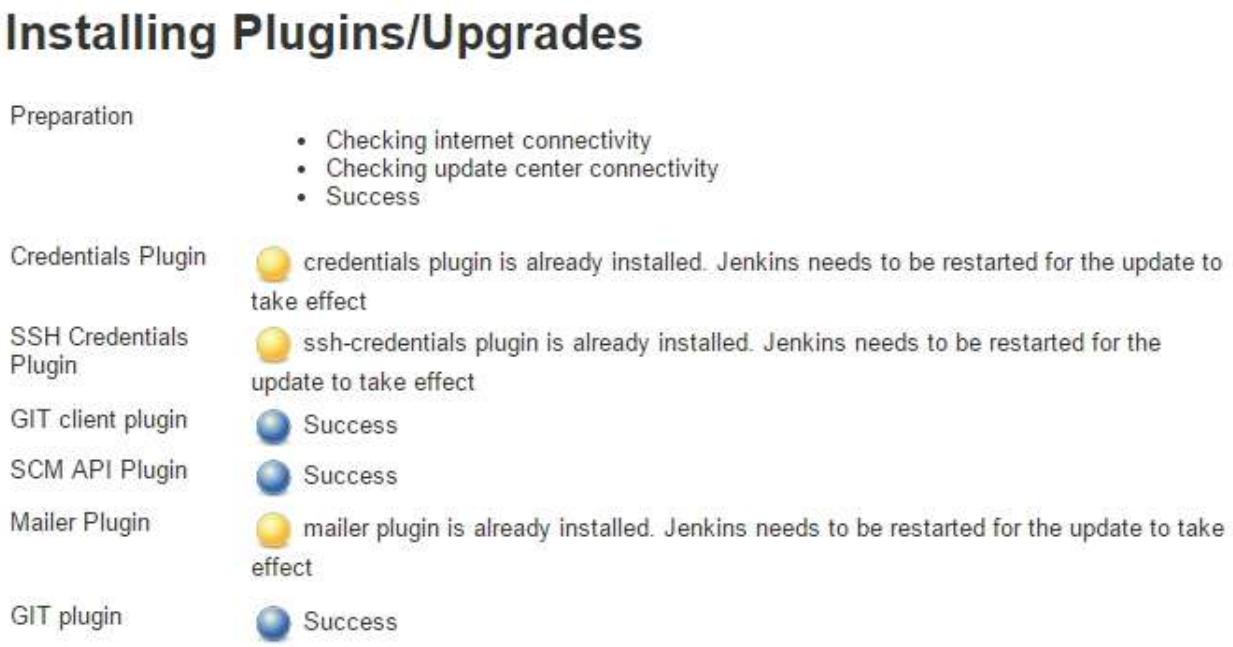
Notification and reporting are an important part of Continuous Integration. Therefore, we need an advanced e-mail notification plugin. We will also need a plugin to make Jenkins interact with Git.

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

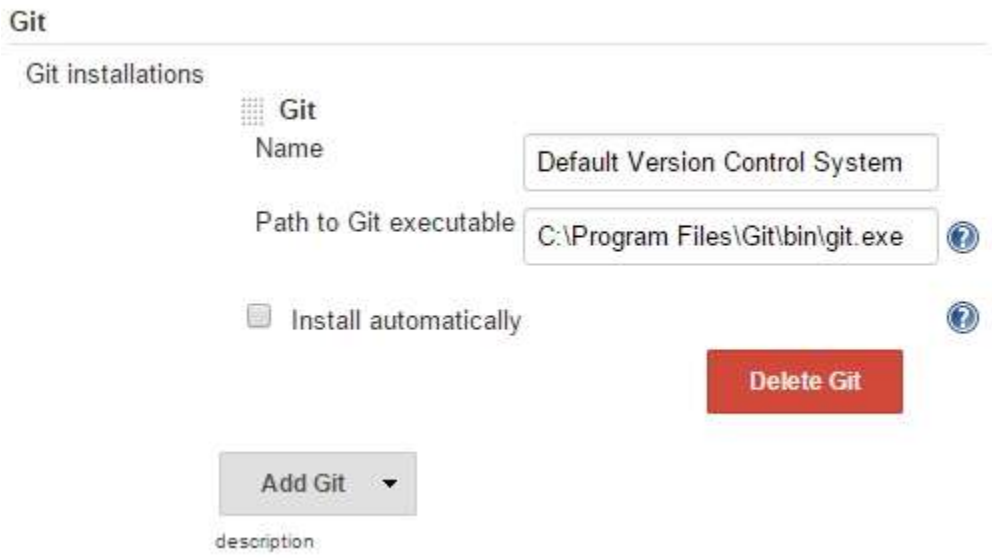
Installing the Git plugin

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.



Note

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.

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

   Copy

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

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

   Copy

```
echo %JAVA_HOME%
```

3. You should see the following output:

   Copy

```
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:

Note

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

Installing and configuring Maven

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

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" /Msetx MAVEN_OPTS "-Xms256m -Xmx512m" /M
```

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

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

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

```
mvn –version
```

and output should be

```
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.

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.

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.

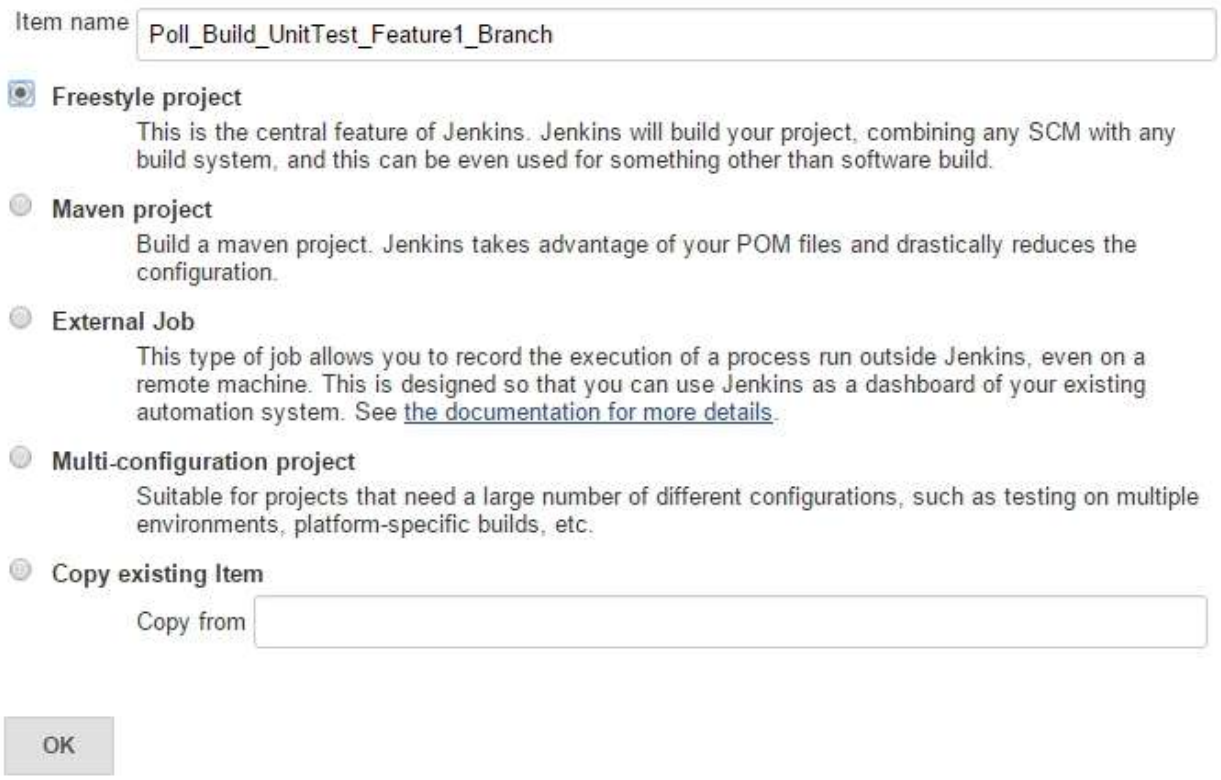
Creating a Jenkins job to poll, build, and unit test code on the feature1 branch

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.

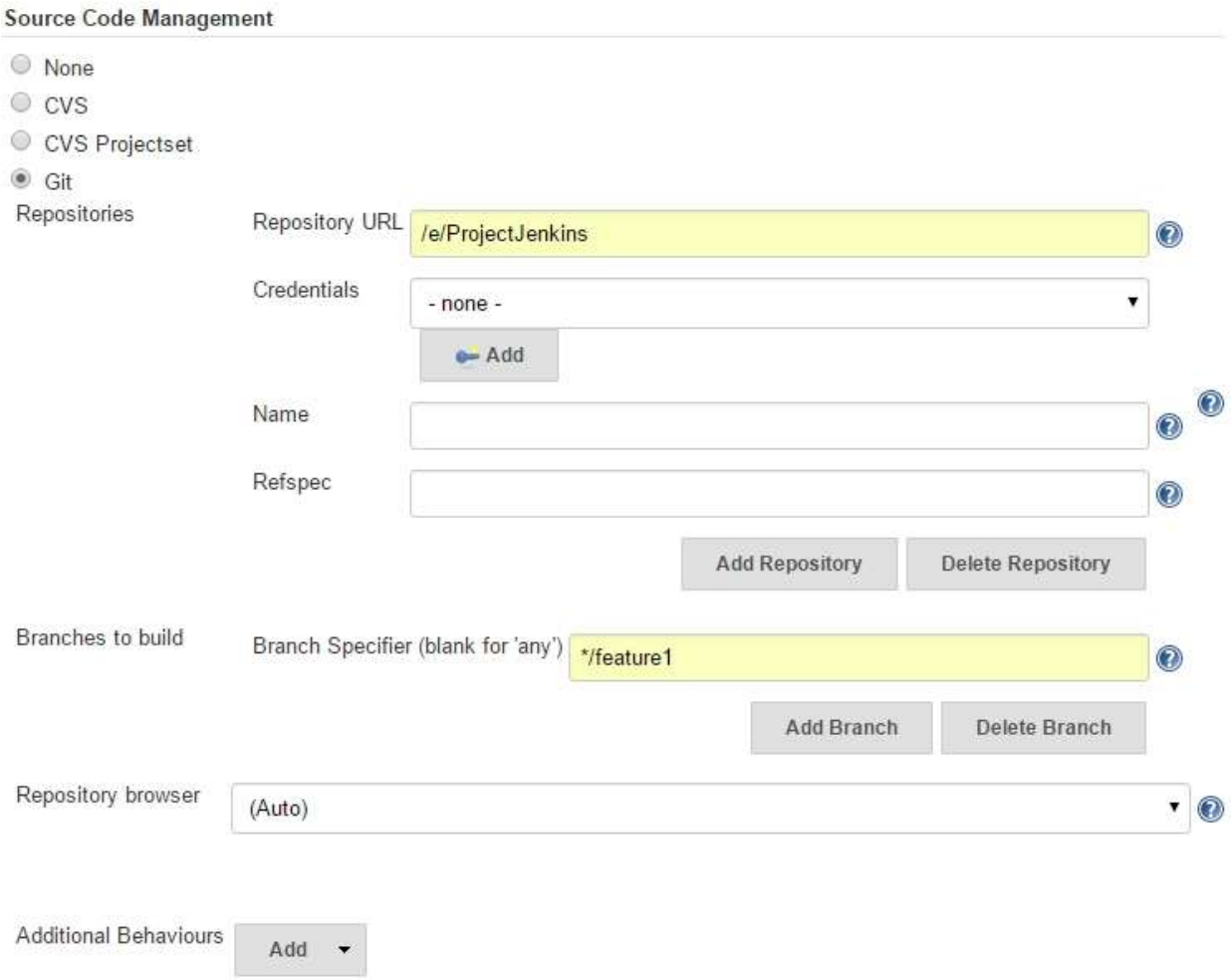


4. Add a meaningful description about the job in the **Description** section.

Polling version control system using Jenkins

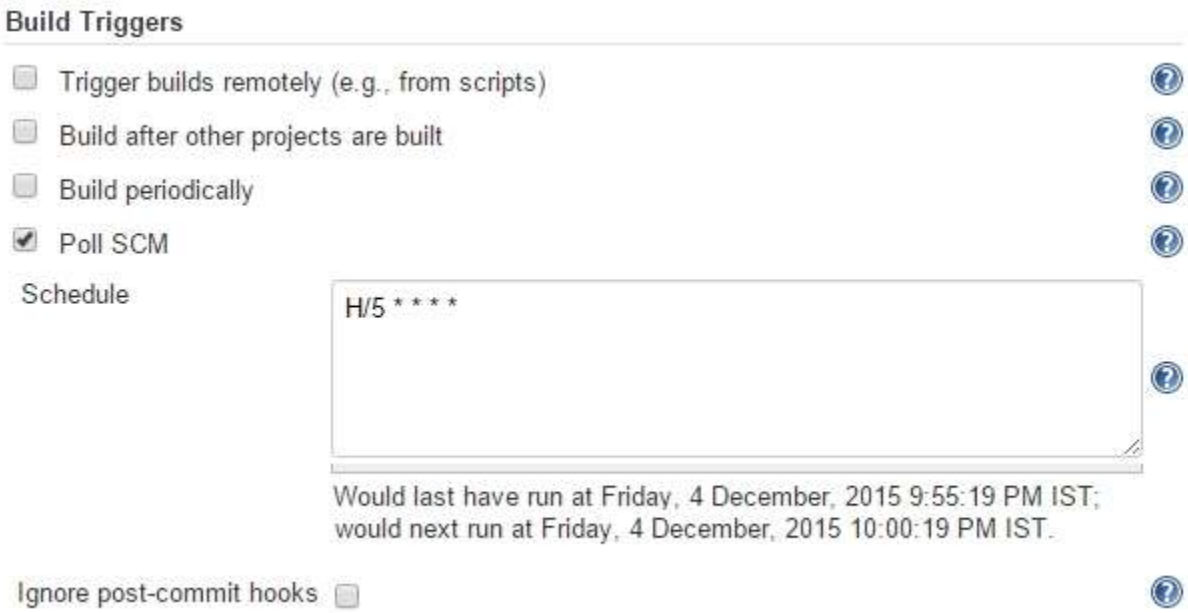
This is a critical step where we connect Jenkins with the Version Control System. This configuration enables Jenkins to poll the correct branch inside Git and download the modified code:

1. Scroll down to the **Source Code Management** section and select the **Git** option.
2. Fill the blanks as follows:
    1. **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.
    2. 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.



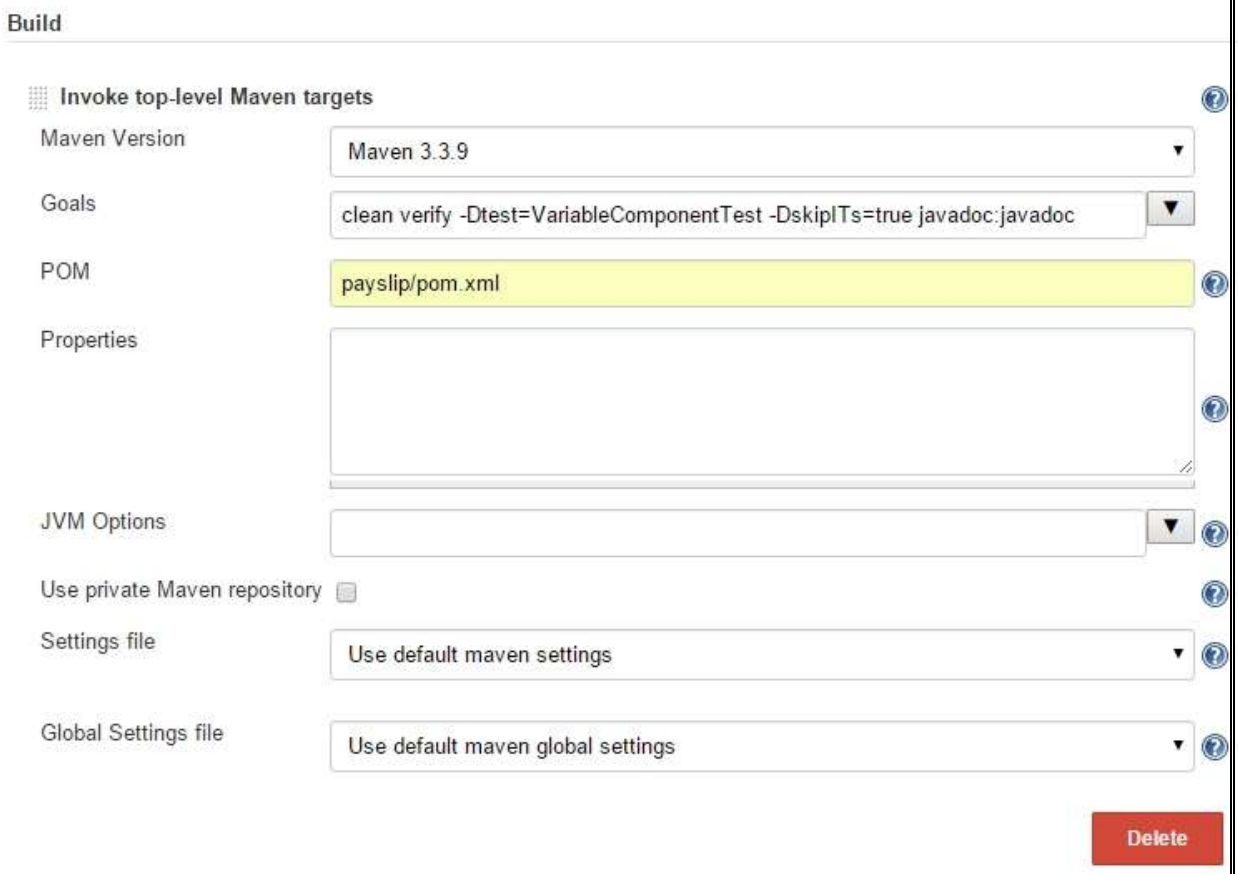
Compiling and unit testing the code on the feature branch

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:
    1. 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.
    2. Type the following line in the **Goals** section:

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

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

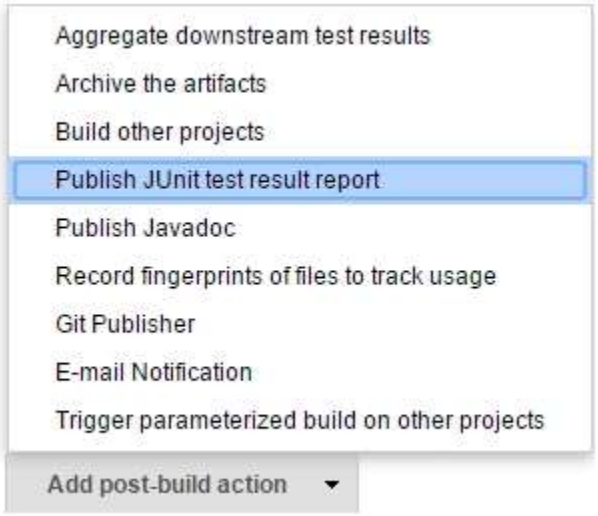


4. Let's see the Maven command inside the **Goals** field in detail:
    1. The `clean` command will clean any old built files
    2. The `-Dtest=VariableComponentTest` command will execute a unit test named `VariableComponentTest.class`
    3. The `-DskipITs=true` command will skip the integration test,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

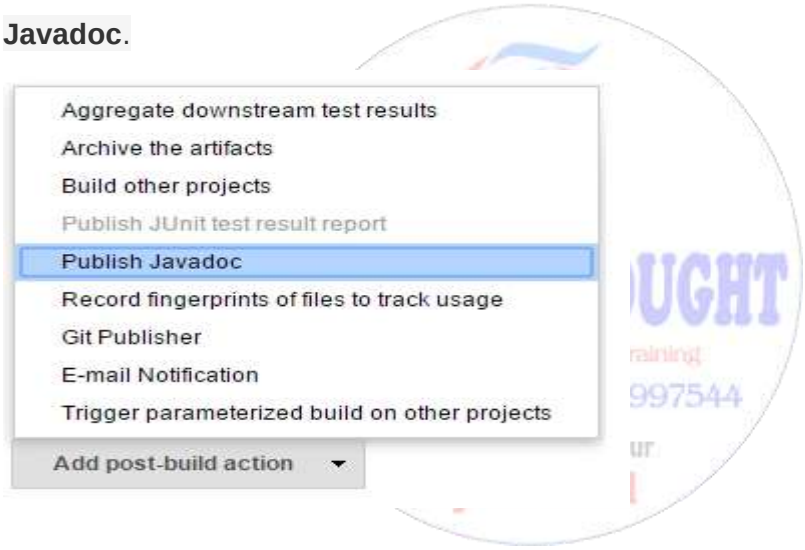
Note

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.

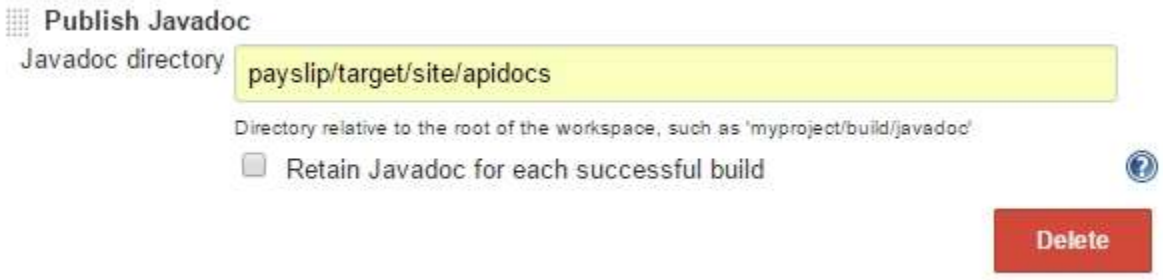
Publishing Javadoc

The steps to publish Javadoc are:

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



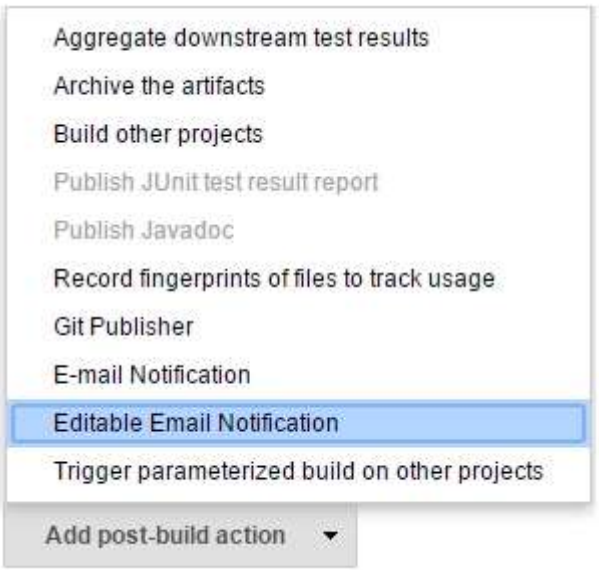
2. Add the path `payslip/target/site/apidocs` in the **Javadoc directory** field, as shown in the following screenshot:



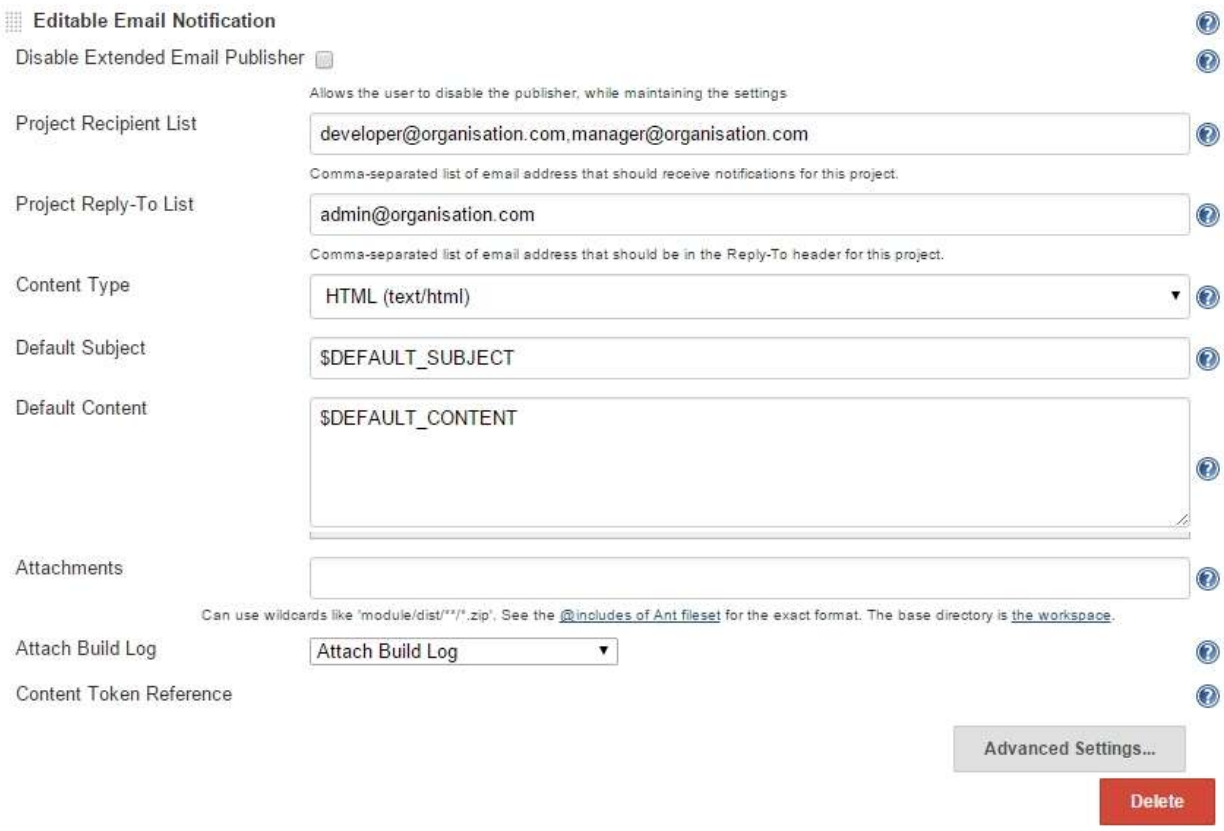
Configuring advanced e-mail notification

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:

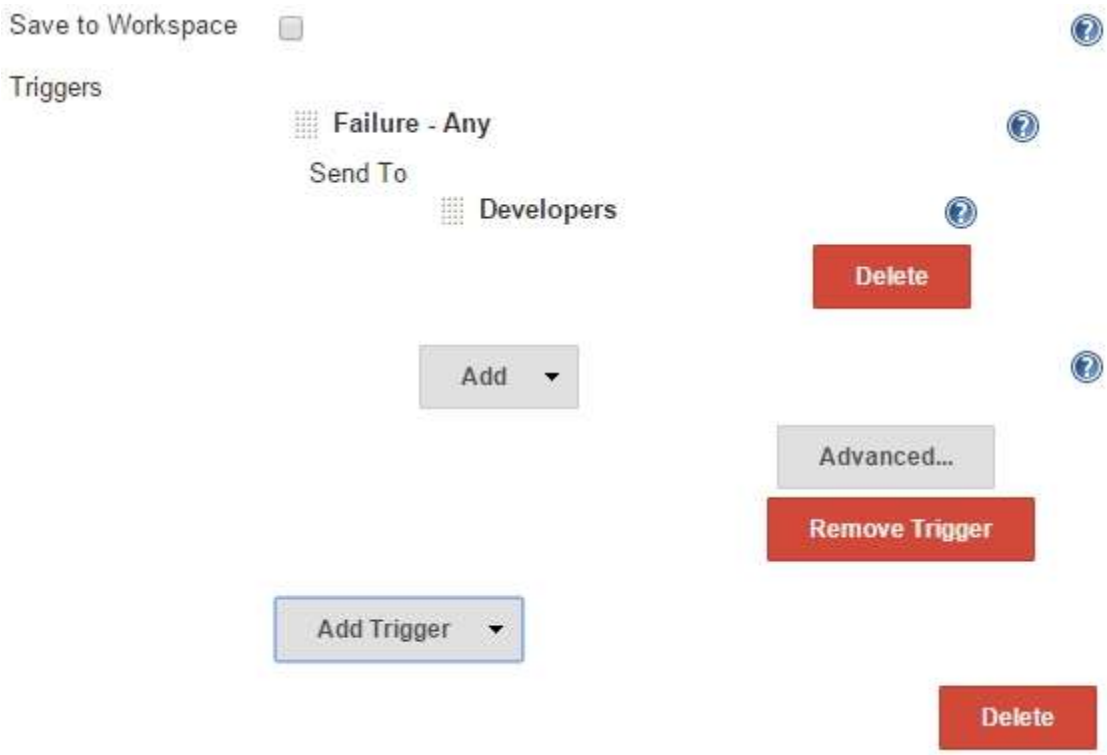
1. Click on the **Add post-build action** button and select **Editable Email Notification**, as shown in the following screenshot:



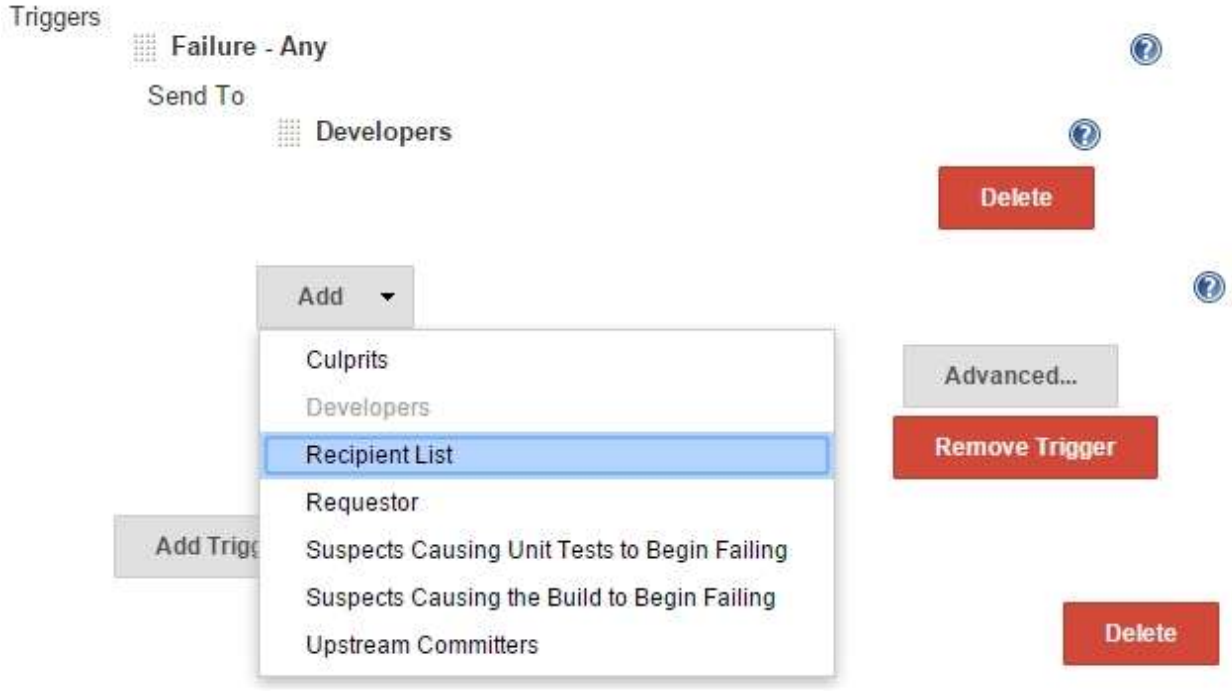
2. Configure **Editable Email Notification** as follows:
    - Under **Project Recipient List**, add the e-mail IDs separated by a comma. You can add anyone who you think should be notified for build and unit test success/failure.
    - You can add the e-mail ID of the Jenkins administrator under **Project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

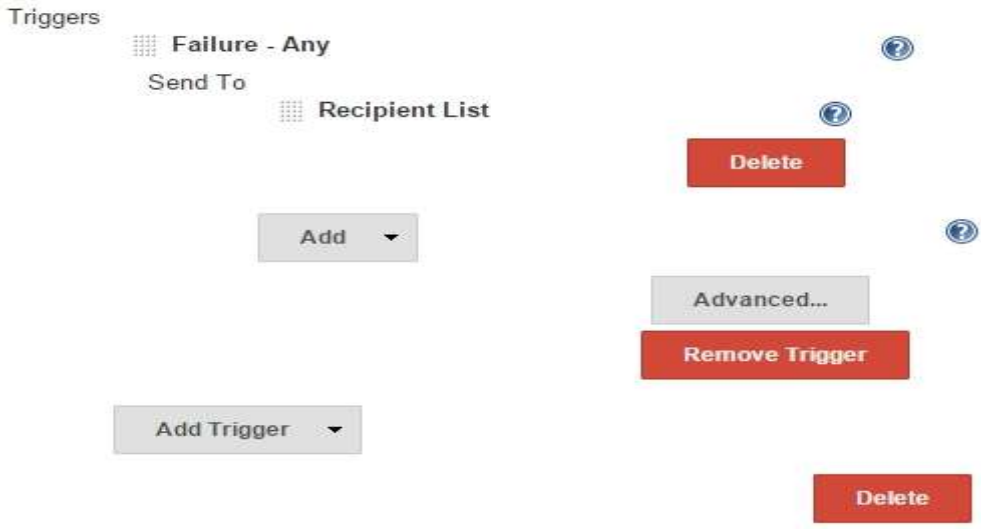


7. But we don't want that; we have already defined whom to send e-mails to. Therefore, click on the **Add** button and select the **Recipient List** option, as shown in the following screenshot:



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

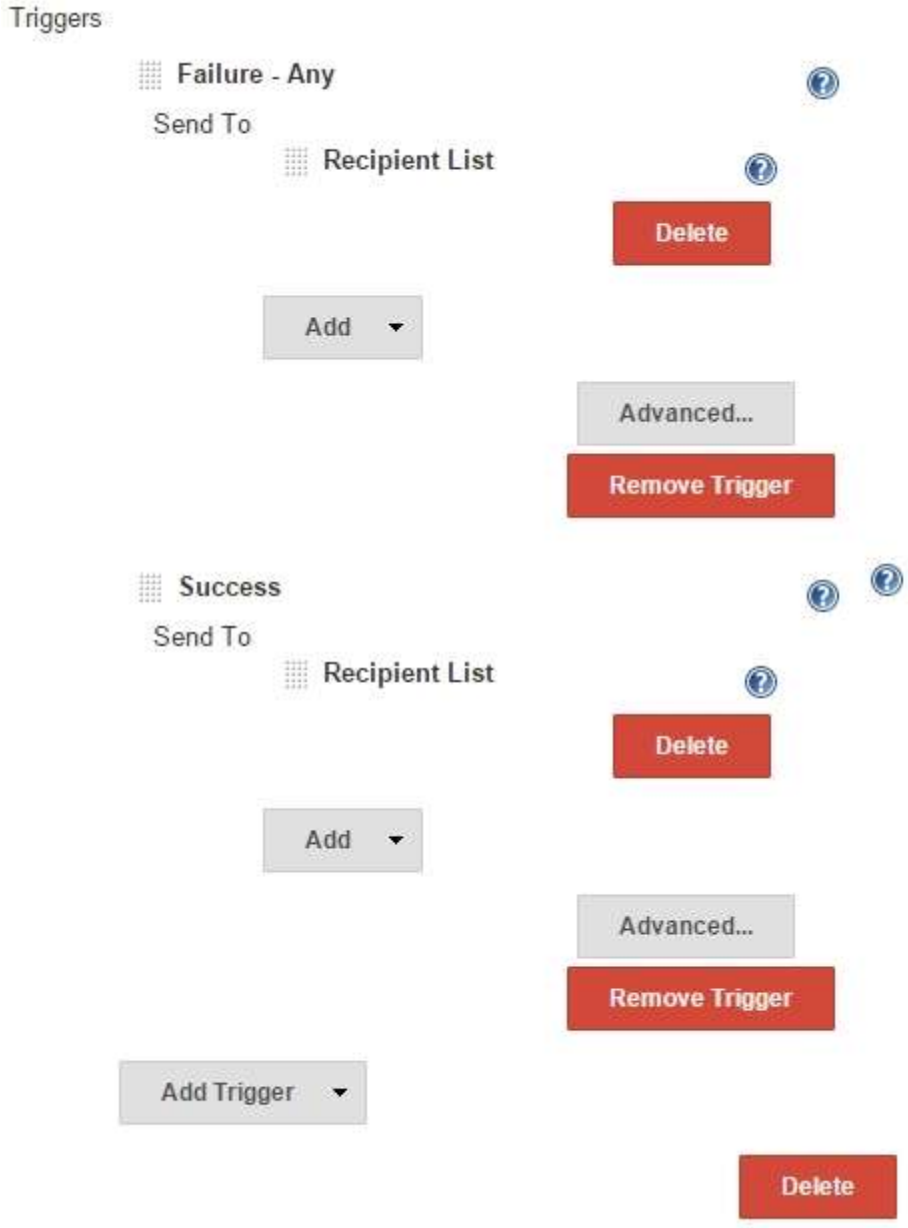
9. Delete **Developers** from the **Send To** section by clicking on the **Delete** button adjacent to it. The result should look something like the following screenshot:



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

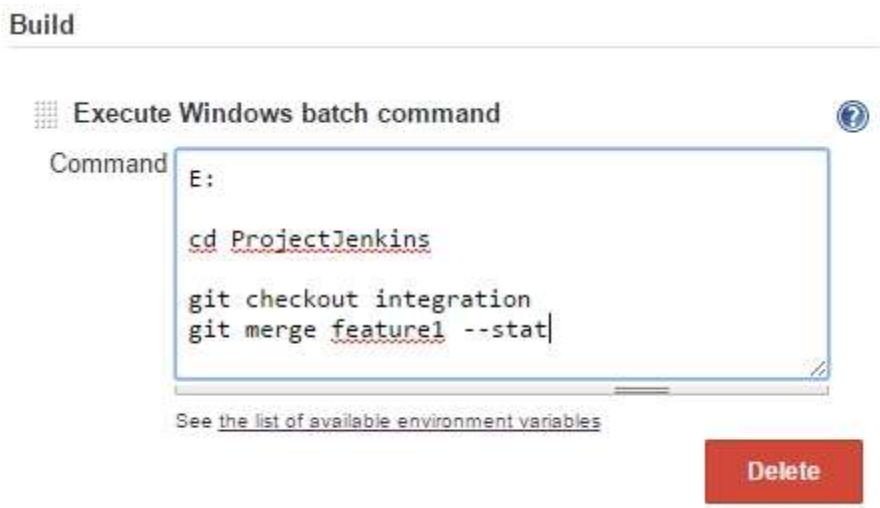
The second Jenkins job in the pipeline performs the task of merging the successfully built and tested code into the integration branch:

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.
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:



6. Configure advanced e-mail notifications exactly the same way as mentioned earlier.
7. 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

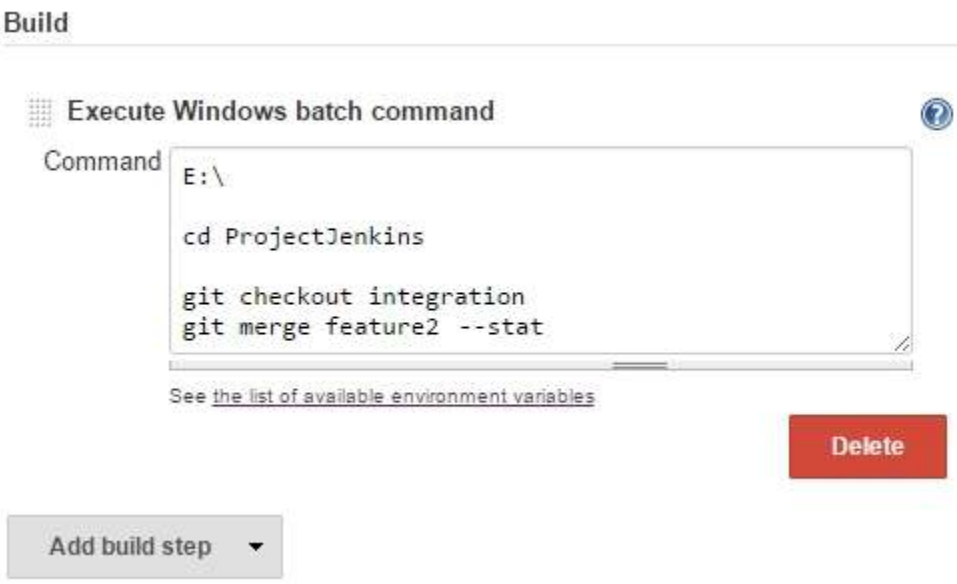
Similarly, we need to create another Jenkins job that will merge the successfully built and unit tested code on the `feature1` branch into the integration branch. And, we will do this by cloning the already existing Jenkins job that merges the successfully build and unit tested code from `feature1` branch 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:



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.

5. Continuous Integration Using Jenkins – Part II

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

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

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/
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:

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:

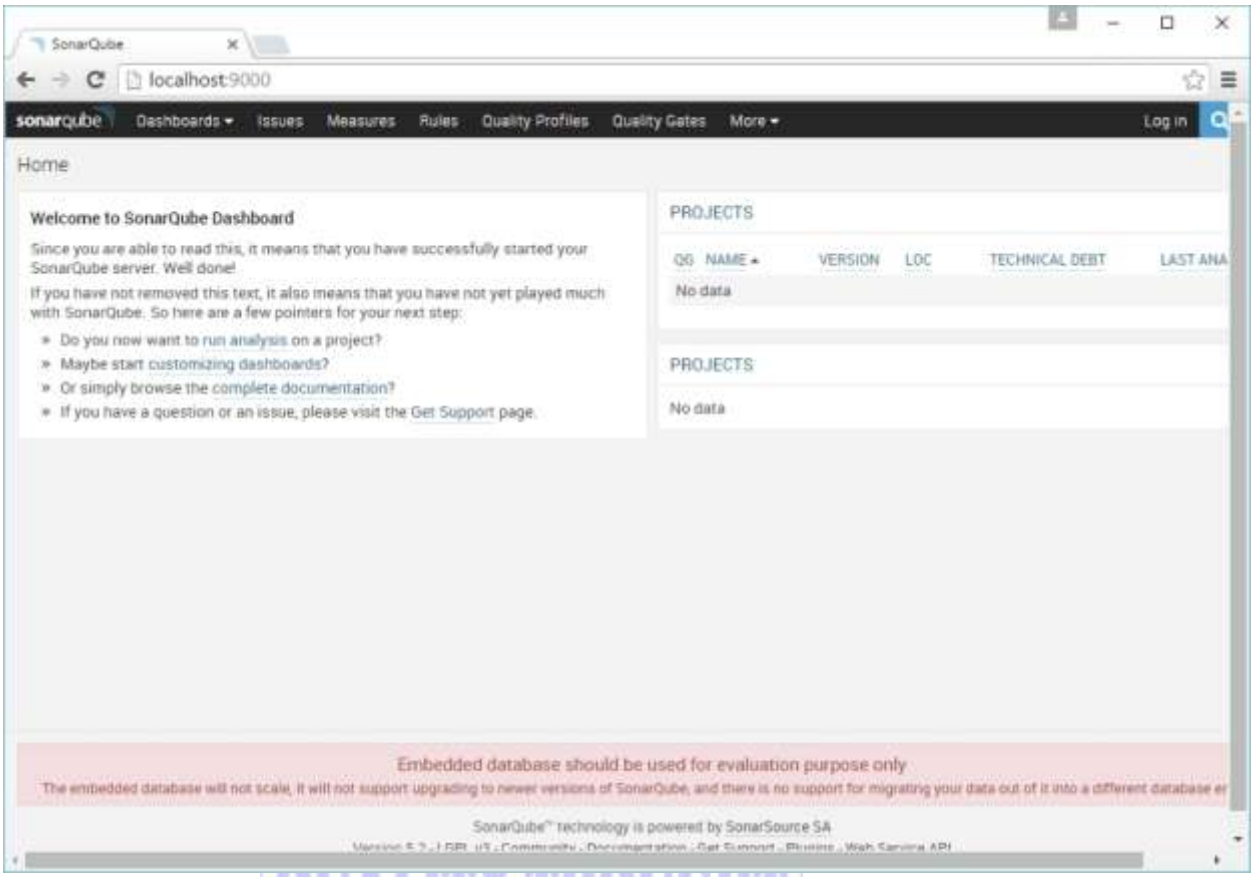
```
setx SONAR_HOME "C:\Program Files\sonarqube-5.1.2" /M
```


Running the SonarQube application

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:
3. To start SonarQube, run the `StartNTService.bat` script:
4. To access SonarQube, type the following link in your favorite web browser http://localhost:9000/.



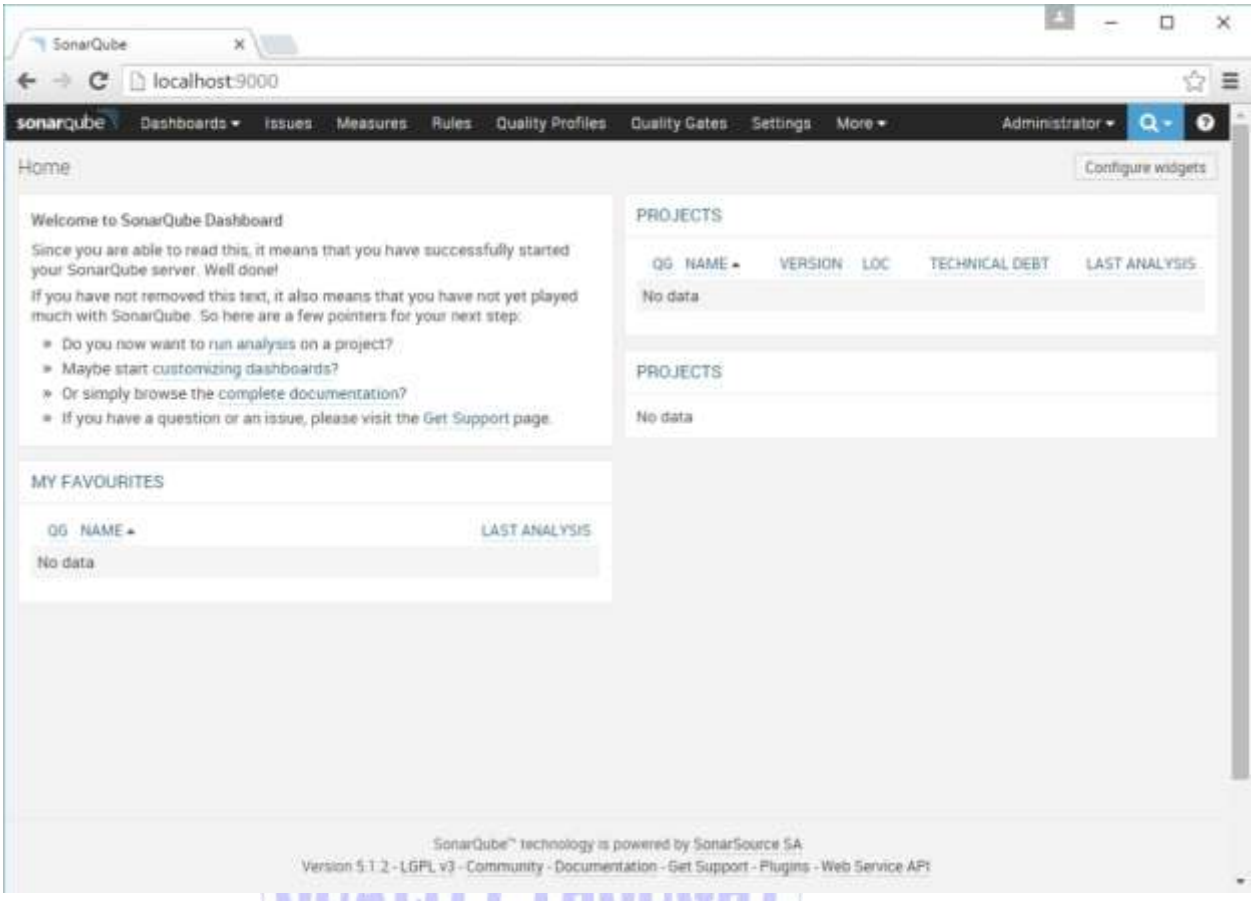
Note

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

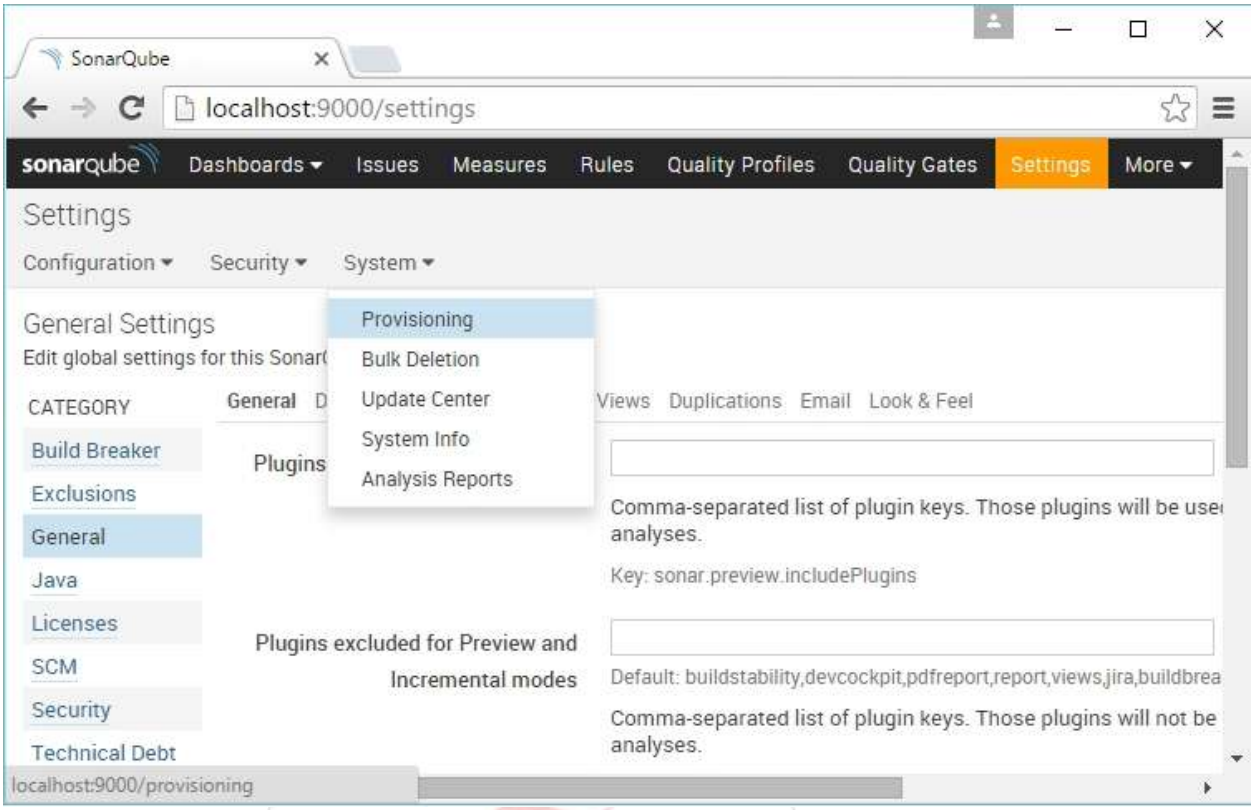
To create the project in SonarQube, use the following steps:

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:



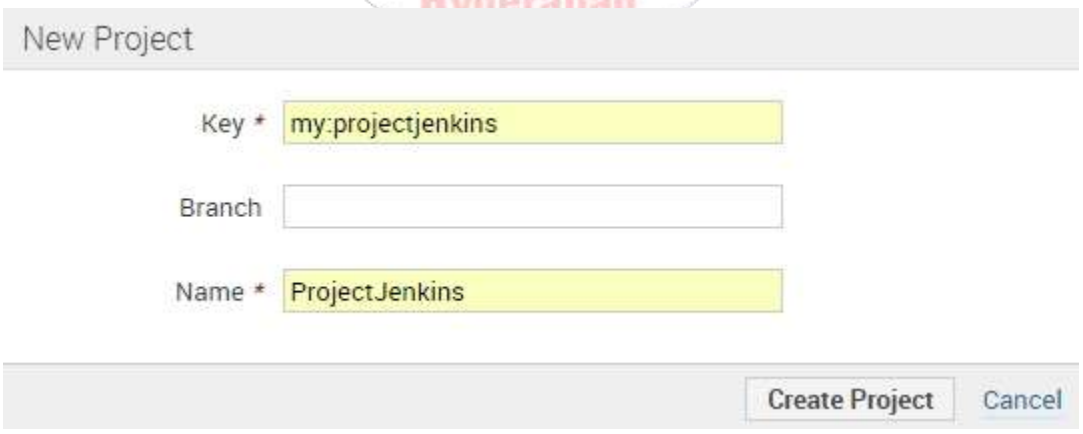
2. Click on the **Settings** link on the menu bar.

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:

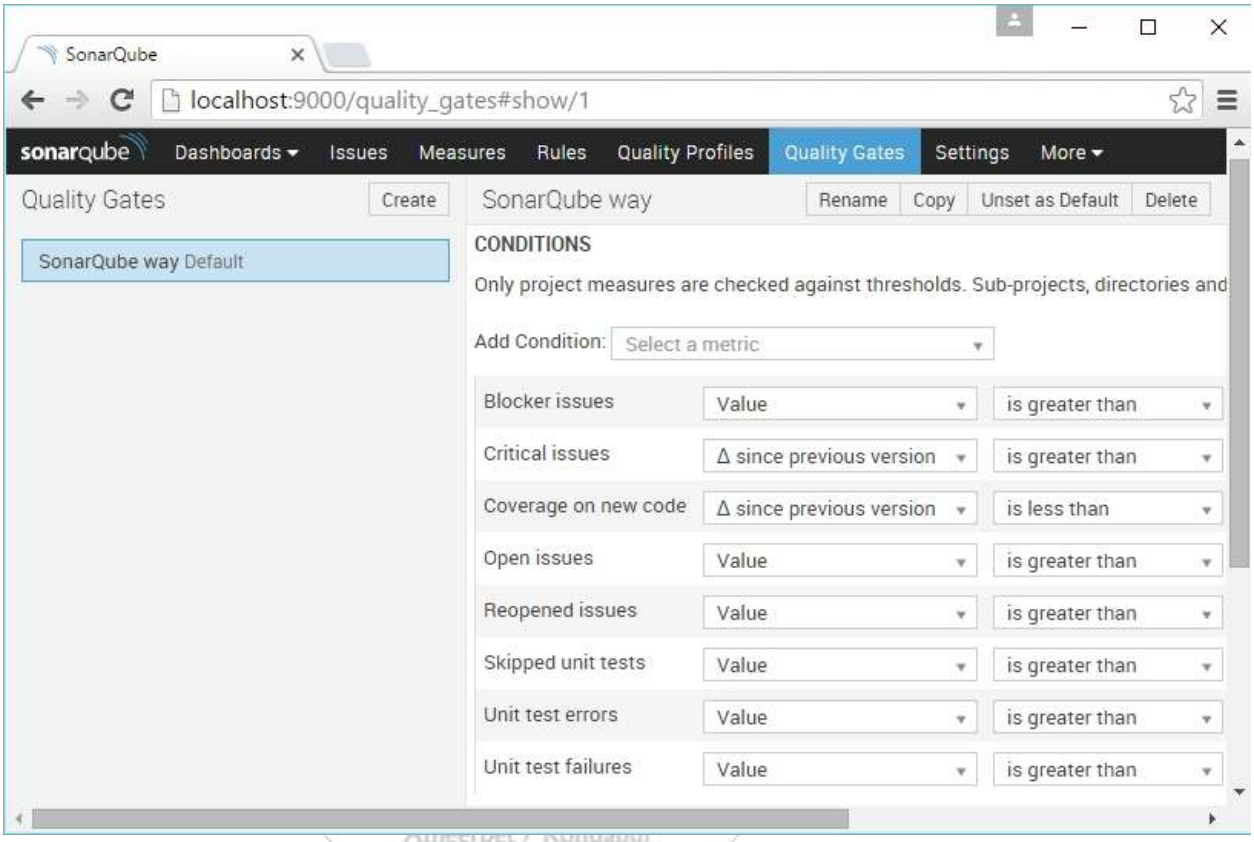
1. Download the build breaker plugin from the following link: http://update.sonarsource.org/plugins/buildbreaker-confluence.html.
2. Place the downloaded `sonar-build-breaker-plugin-1.1.rar` file in the following location: `C:\Program Files\sonarqube-5.1.2\extensions\plugins`.
3. We need to restart SonarQube 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**.
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`.

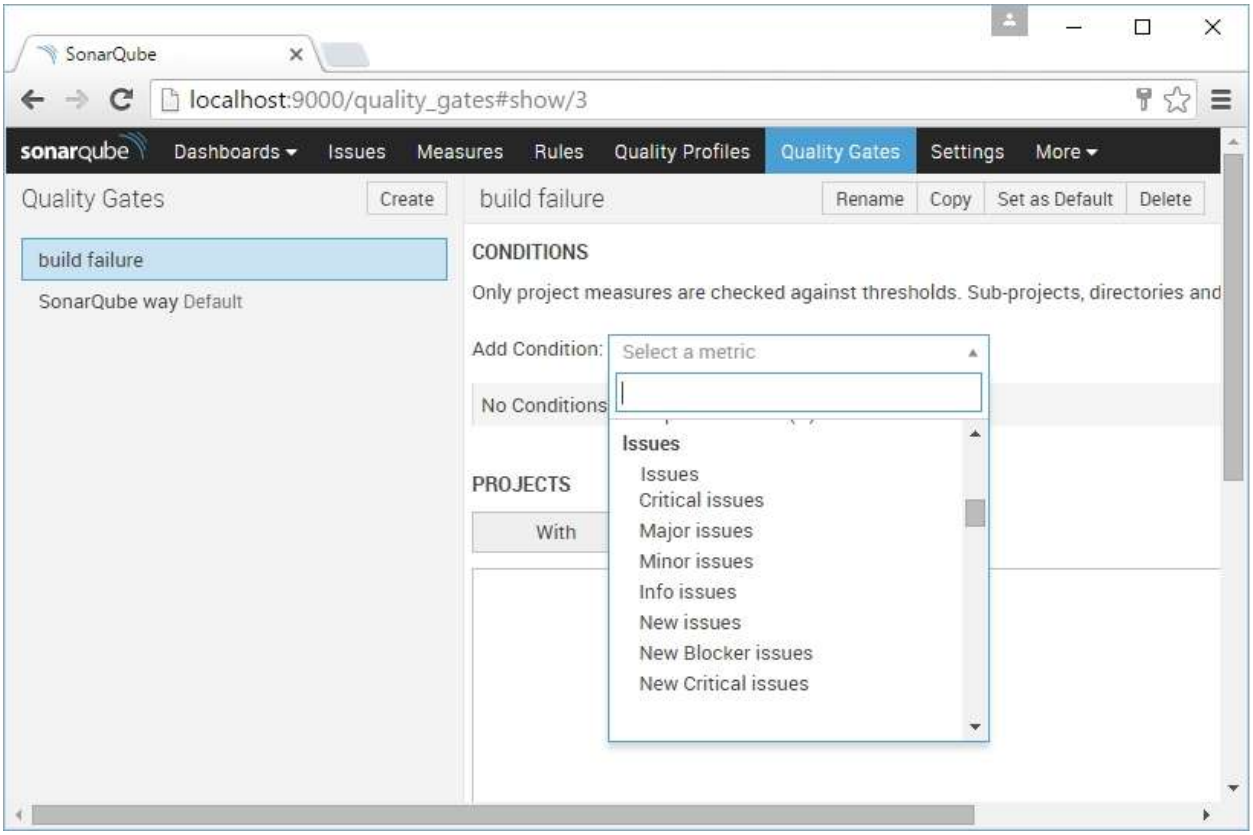
4. Once done, click on the **Create** button.

5. You will see a new quality gate named `build failure` on the left-hand side of the page.

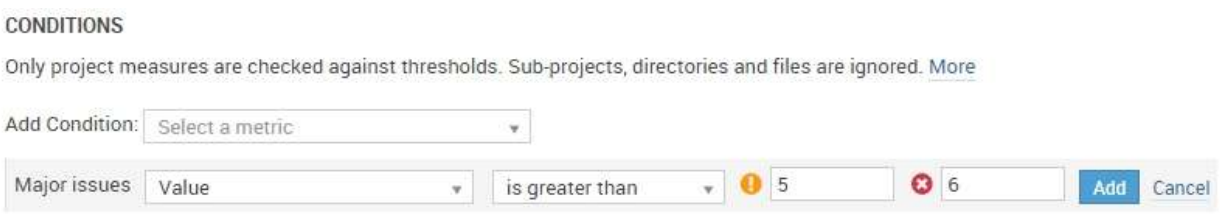
6. Now, click on the `build failure` quality gate and you will see a few settings on the right-hand side of the page.

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:



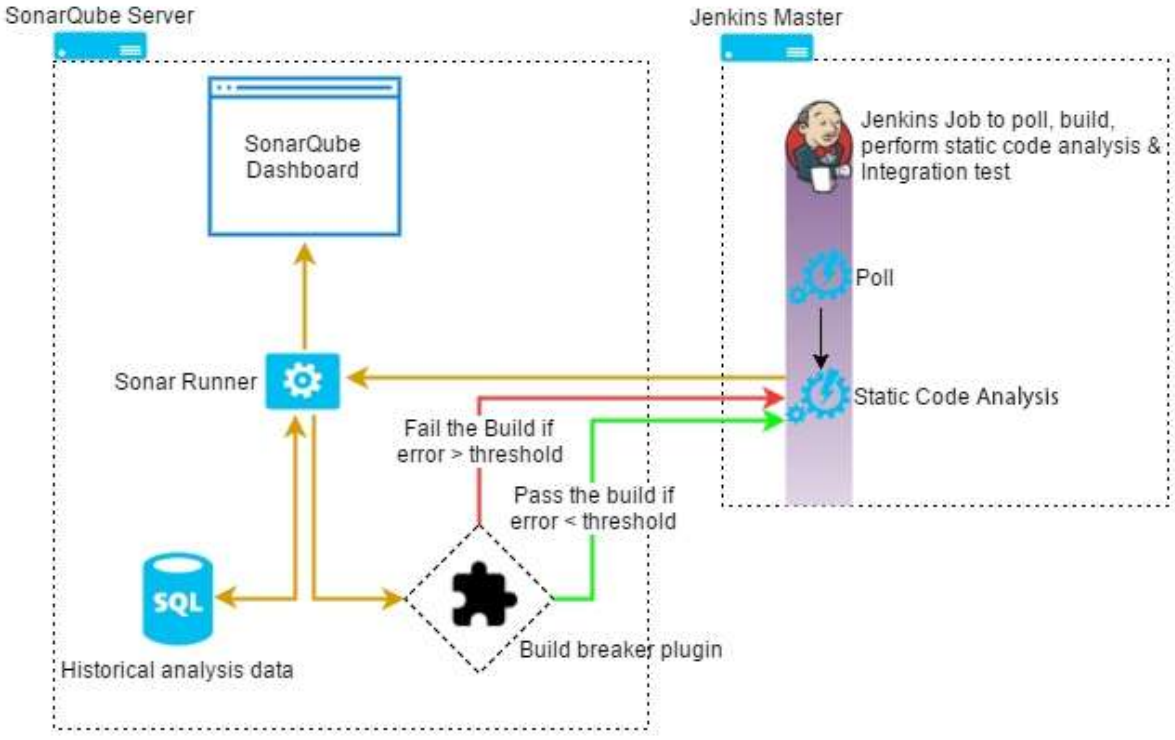
Installing SonarQube Scanner

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.

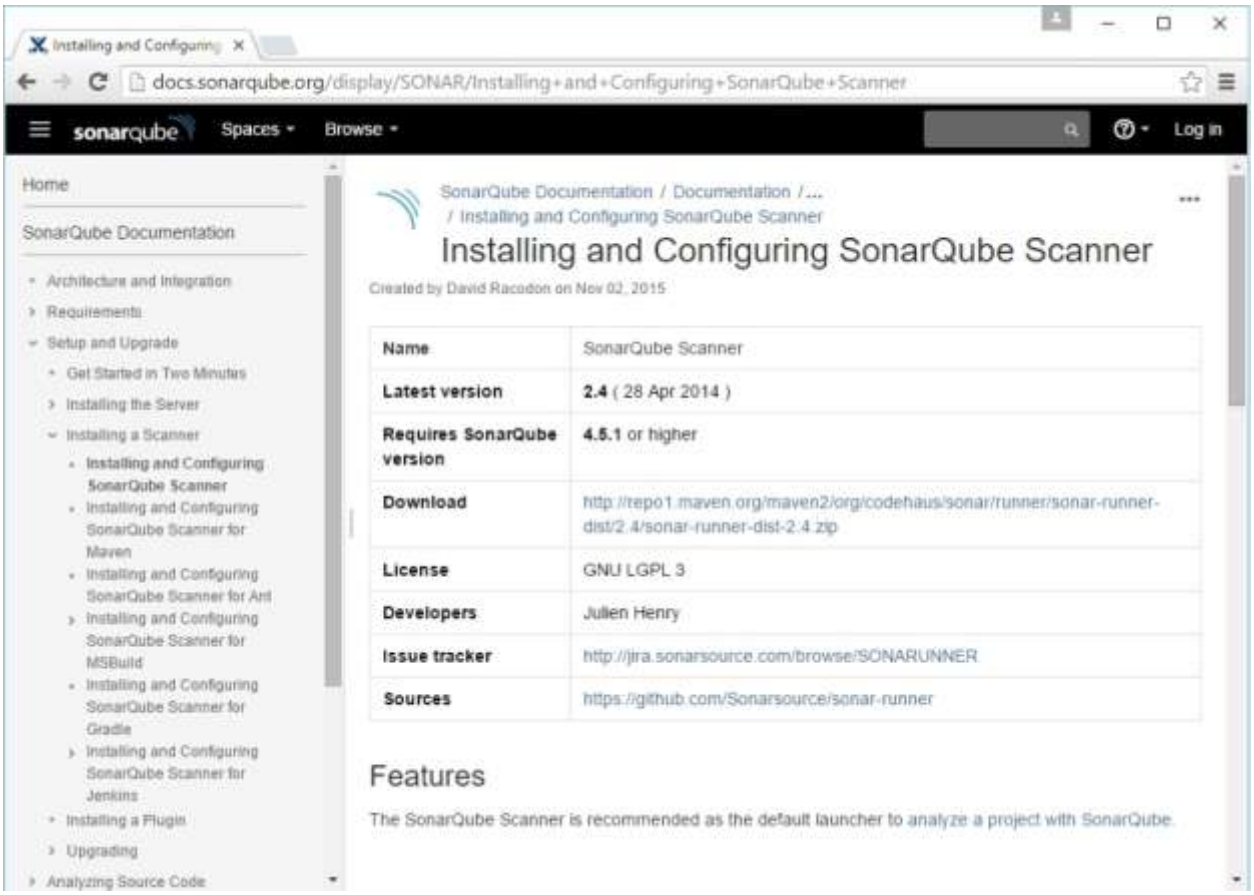
There are conditions defined inside the quality gates. If the analysis passes these conditions, then the Jenkins job is signed to proceed. However, if the analysis fails the condition, then the build breaker plugin terminates the Jenkins job.



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:

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
```

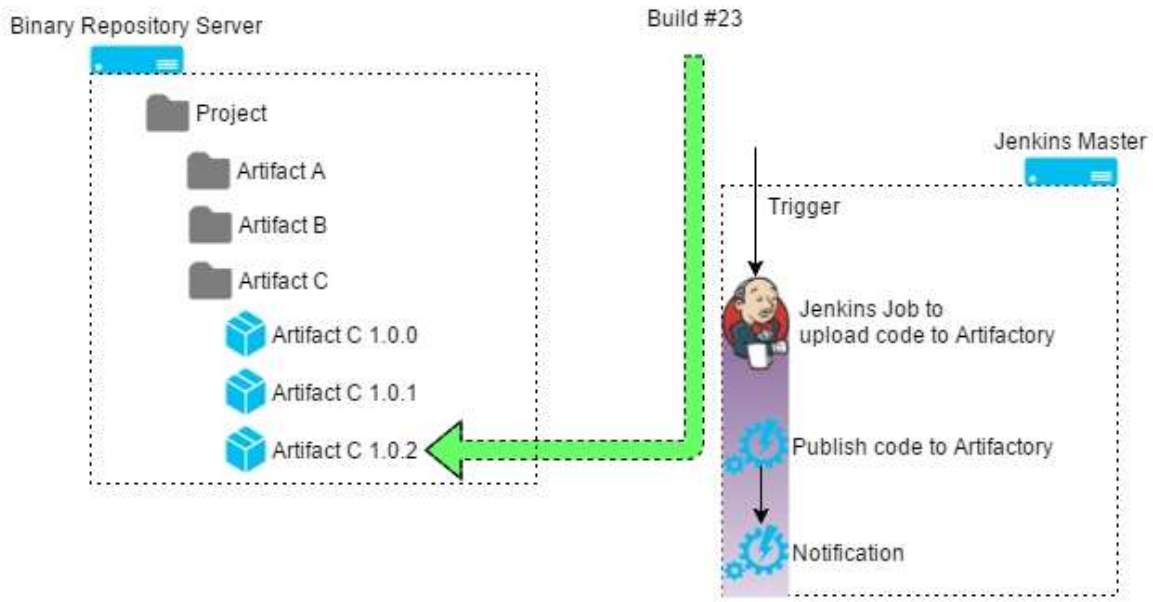
Installing Artifactory

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.



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:

```
setx ARTIFACTORY_HOME "C:\Program Files\artifactory-oss-4.3.2" /M
```

Running the Artifactory application

To run Artifactory, open **Command Prompt** using admin privileges. Otherwise, this doesn't work.

Go to the location where the script to run Artifactory is present:

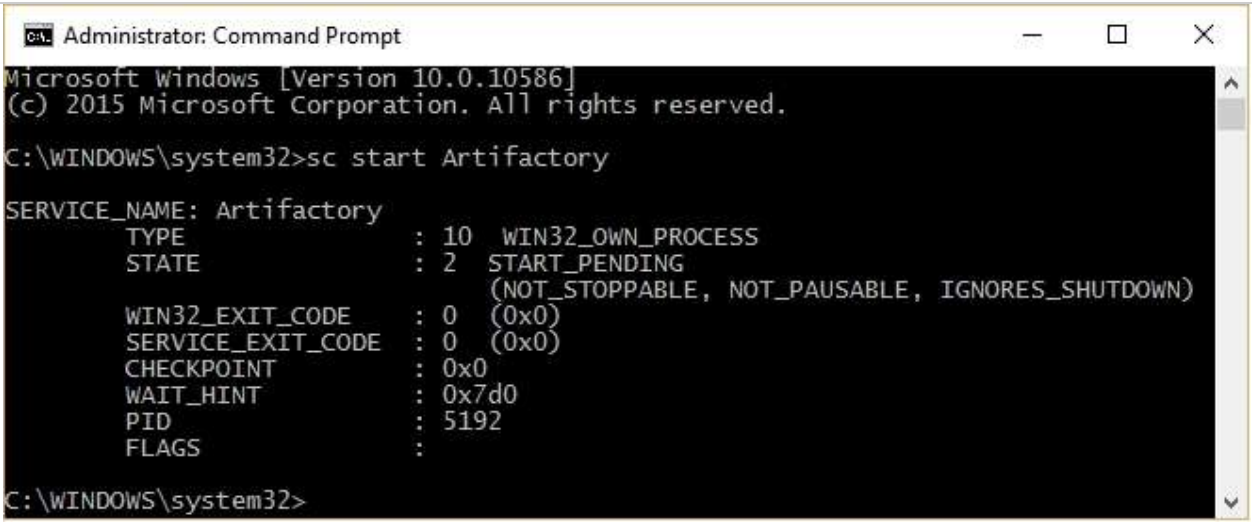
```
cd %ARTIFACTORY_HOME%\bin
```

Execute the `installService.bat` script:

This will open up a new **Command Prompt** window that will install Artifactory as a windows service.

To start Artifactory, open **Command Prompt** using admin privileges and use the following command:

```
sc start Artifactory
```

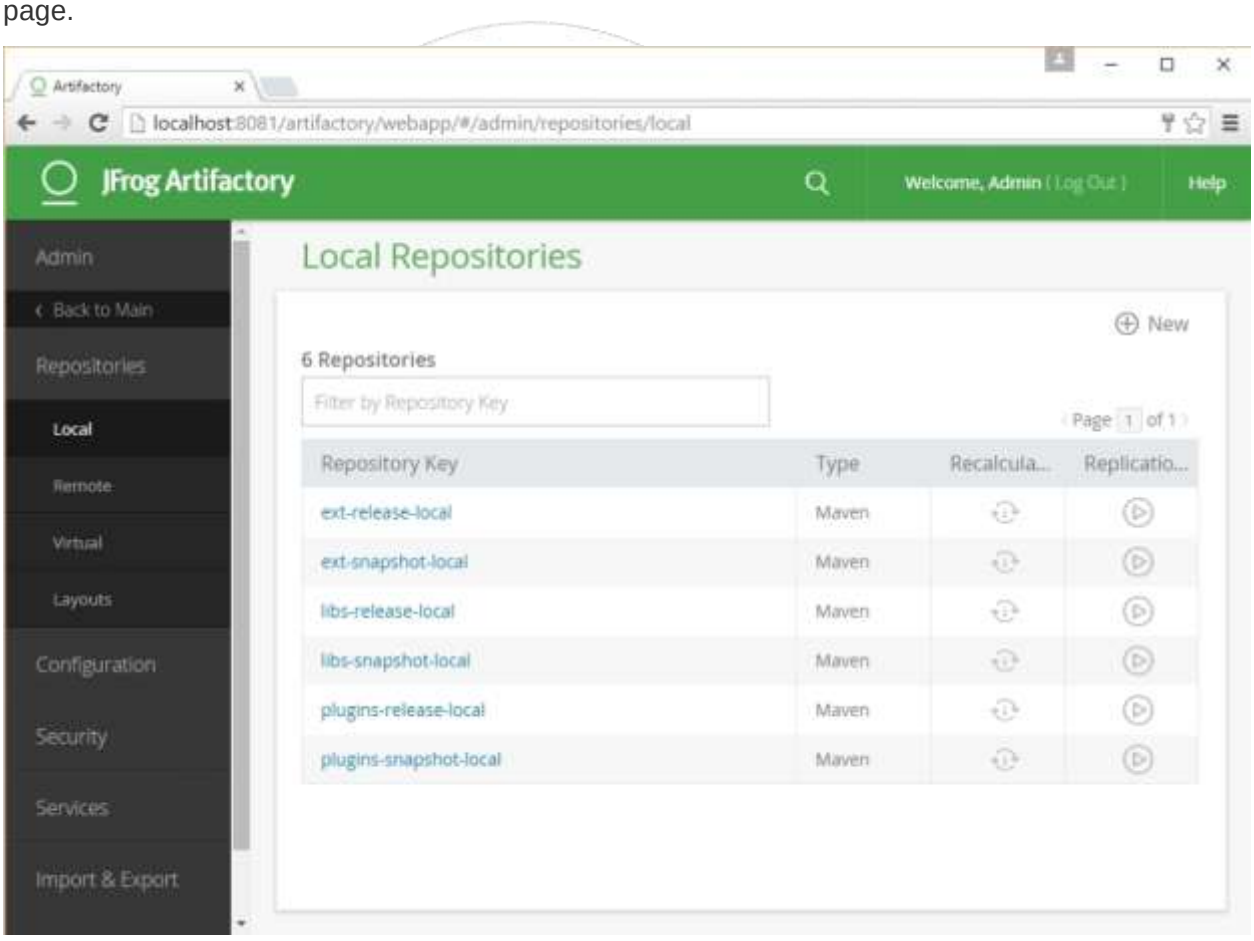


Access Artifactory using the following link: http://localhost:8081/artifactory/.

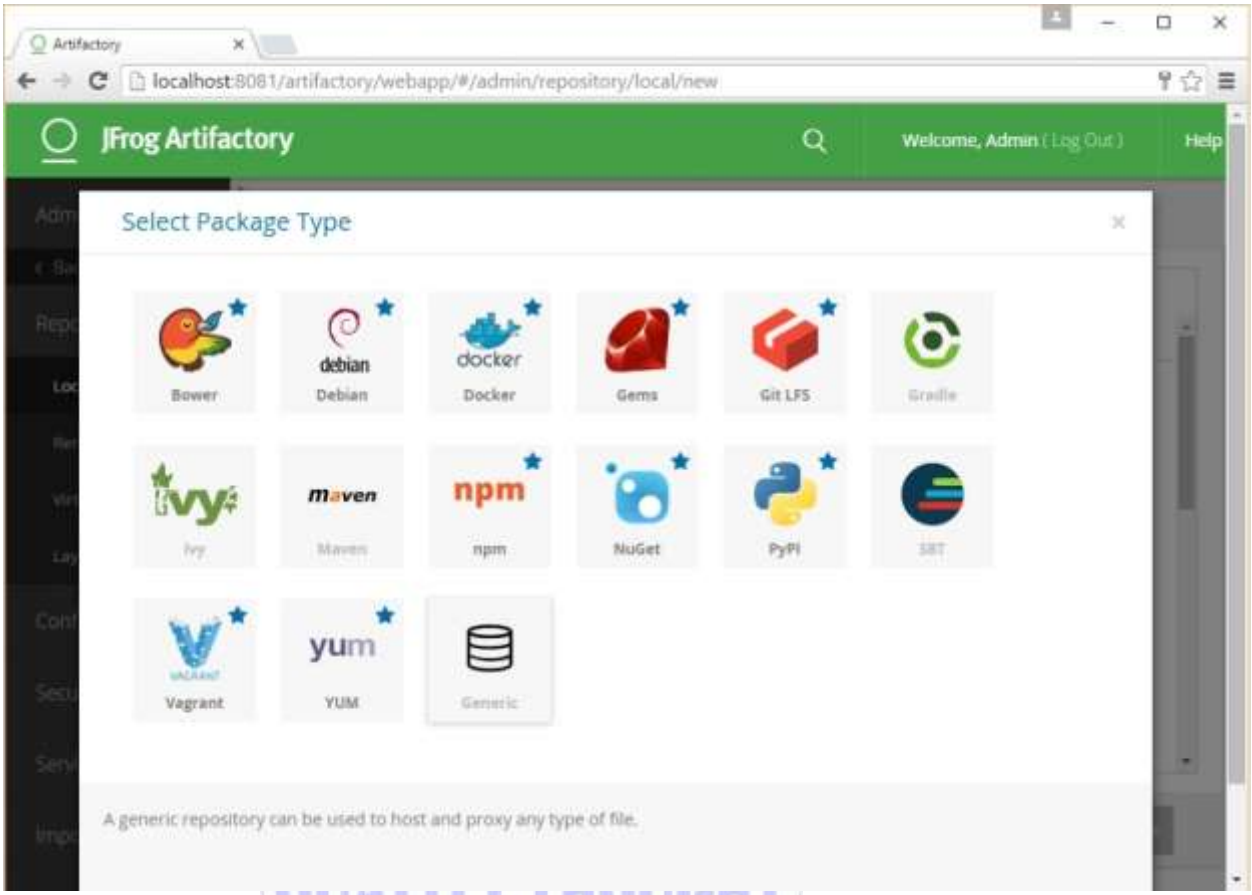
Creating a repository inside Artifactory

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.

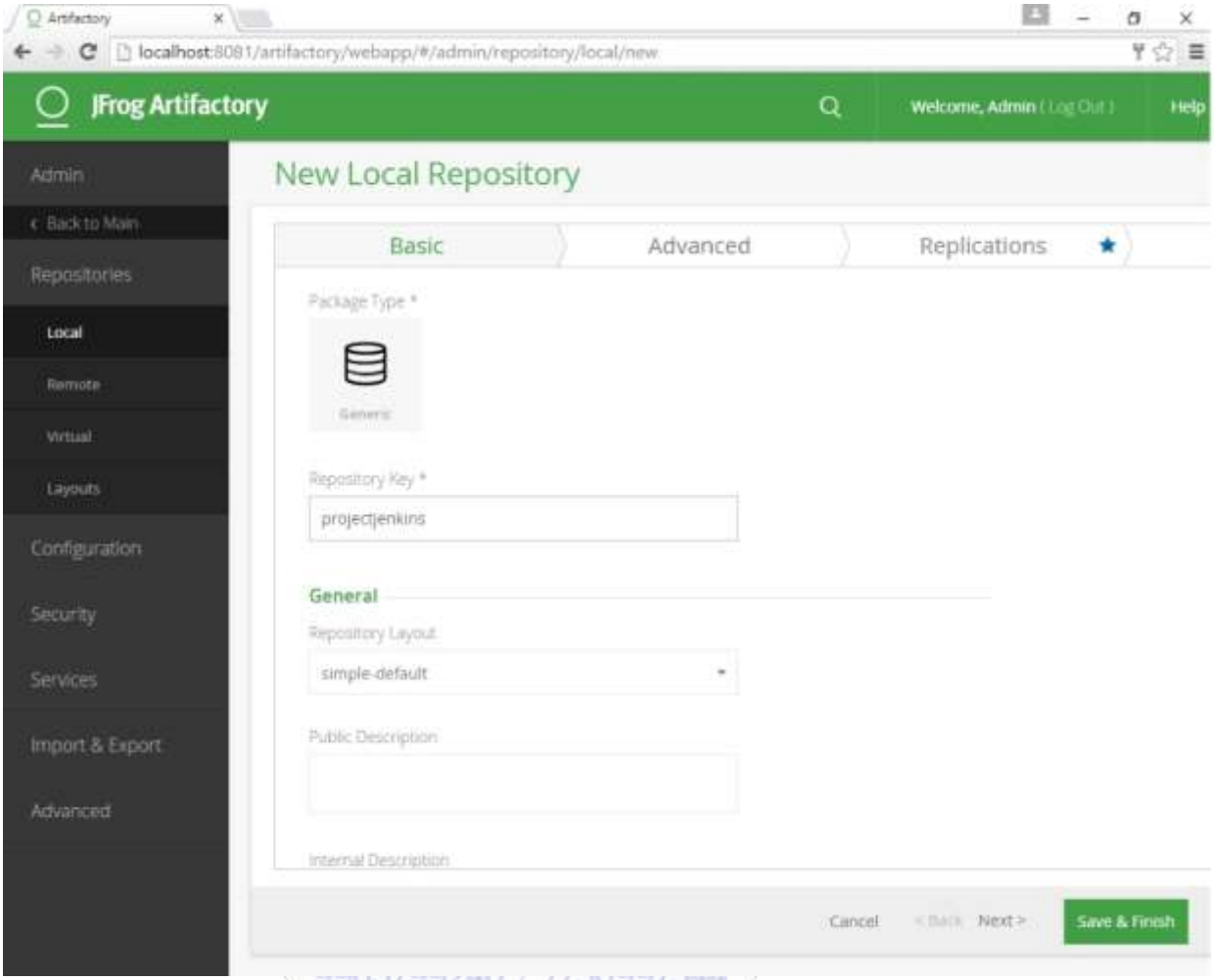


4. In the window that pops-up, select the package type as **Generic**.

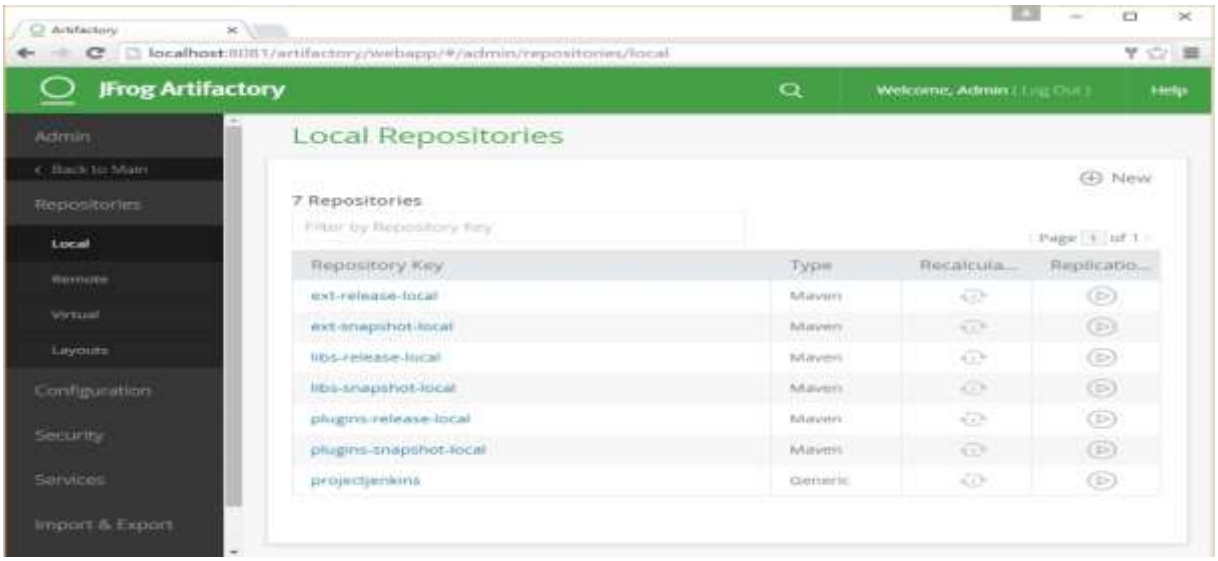


5. Give a name in the **Repository Key *** field. In our example I have used `projectjenkins`.

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

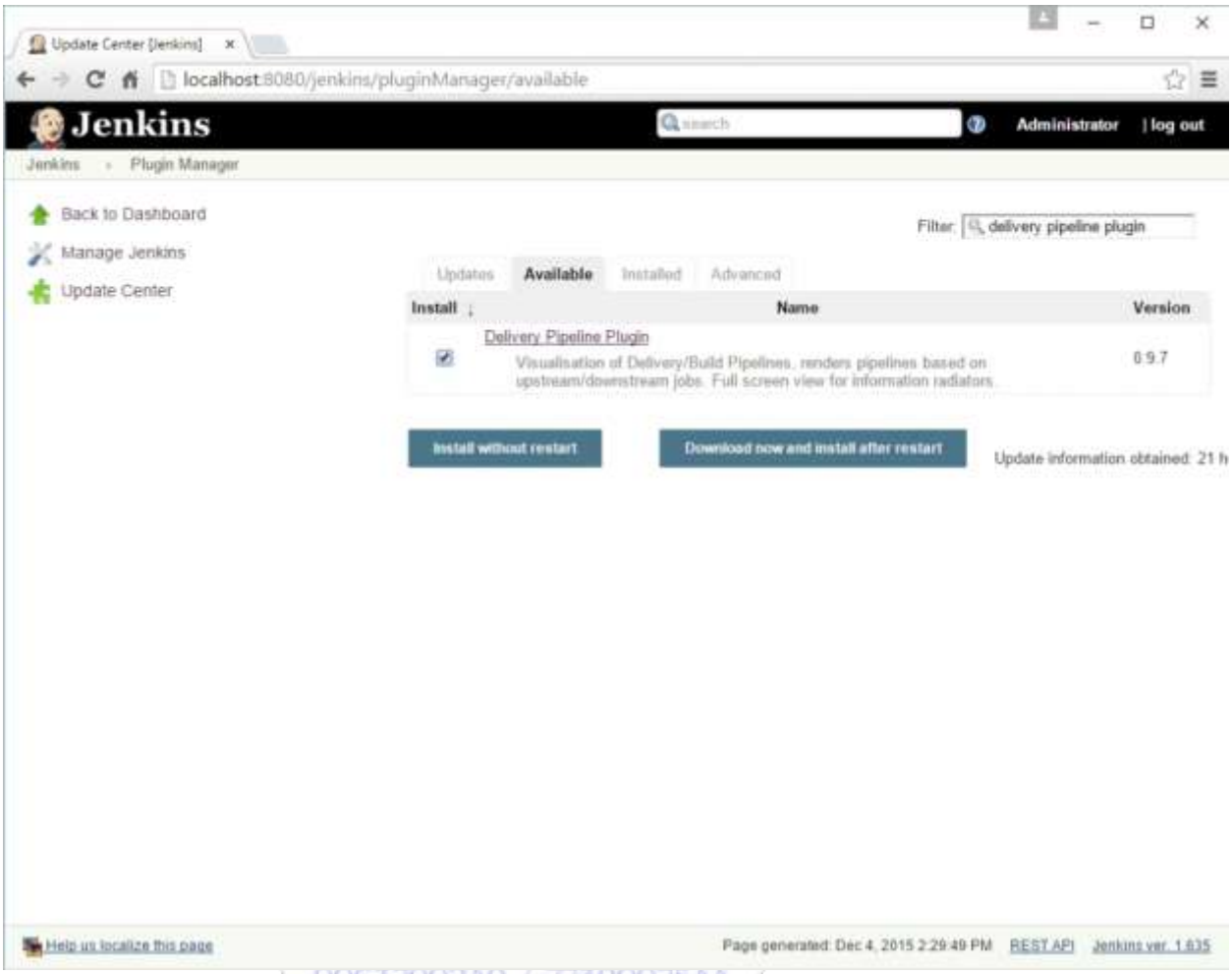
In the previous sections, we saw how to install and configure Artifactory and SonarQube along with SonarQube Runner. For these tools to work in collaboration with Jenkins, we need to install their respective Jenkins plugins.

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.

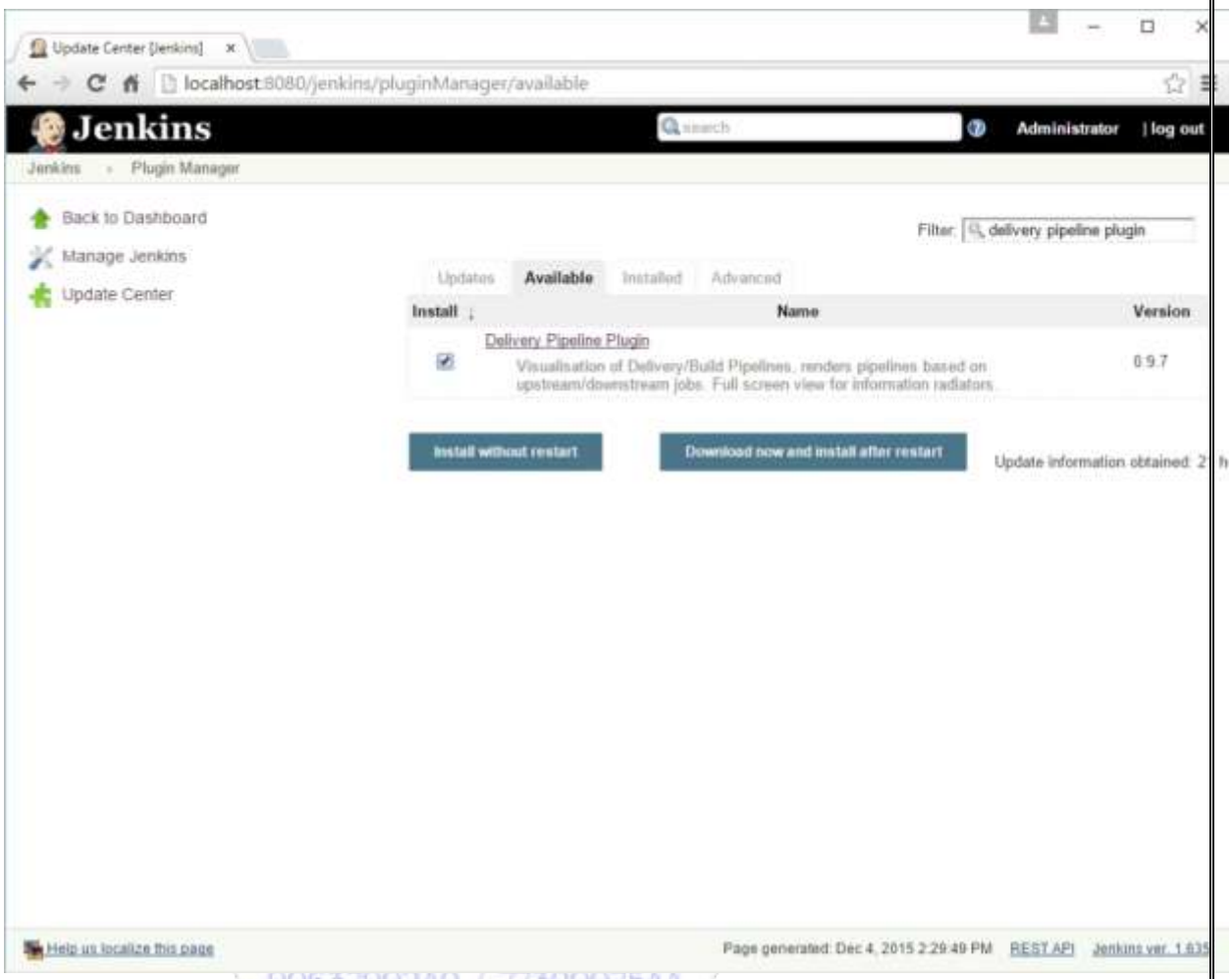
Installing the delivery pipeline plugin

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.



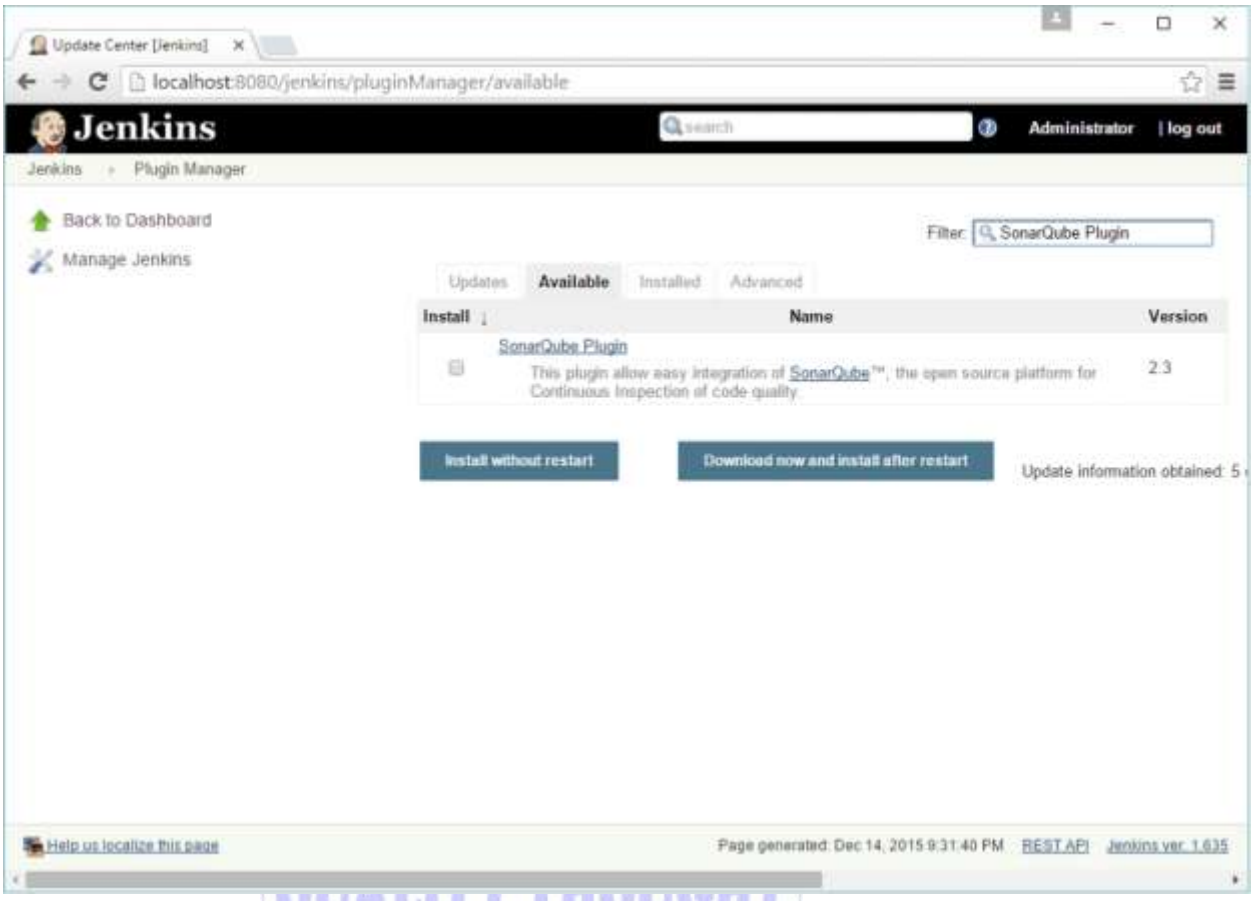
5. The download and installation of the plugin starts automatically. You can see **Delivery Pipeline Plugin** has some dependencies that get downloaded and installed.



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.

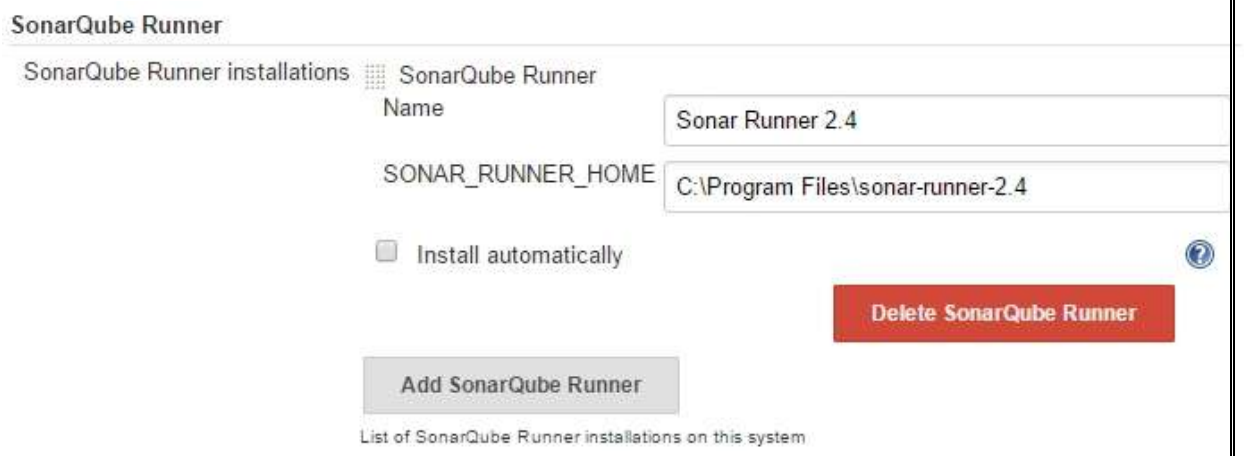


4. As it can be seen in the next screenshot, the plugin is installed immediately:

5. Upon successful installation of the SonarQube Plugin, go to the **Configure System** link on the **Manage Jenkins** page.

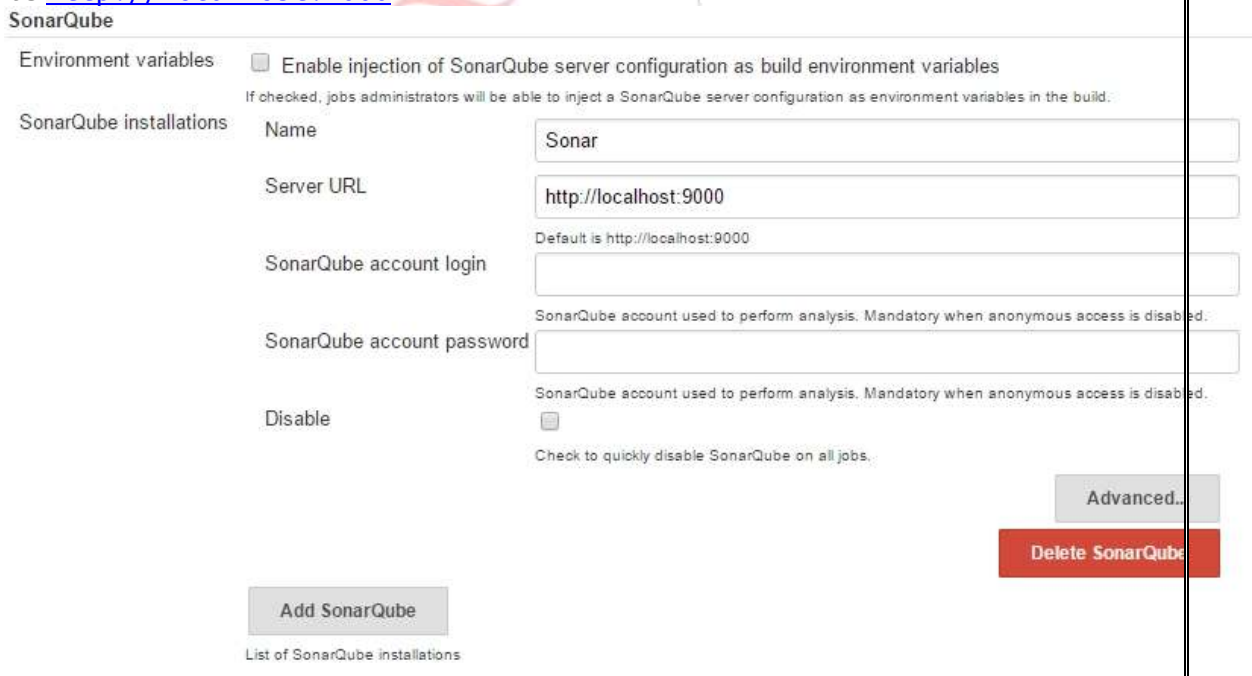
6. Scroll down until you see the **SonarQube Runner** section and fill in the blanks as shown here:

   - 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`.



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.

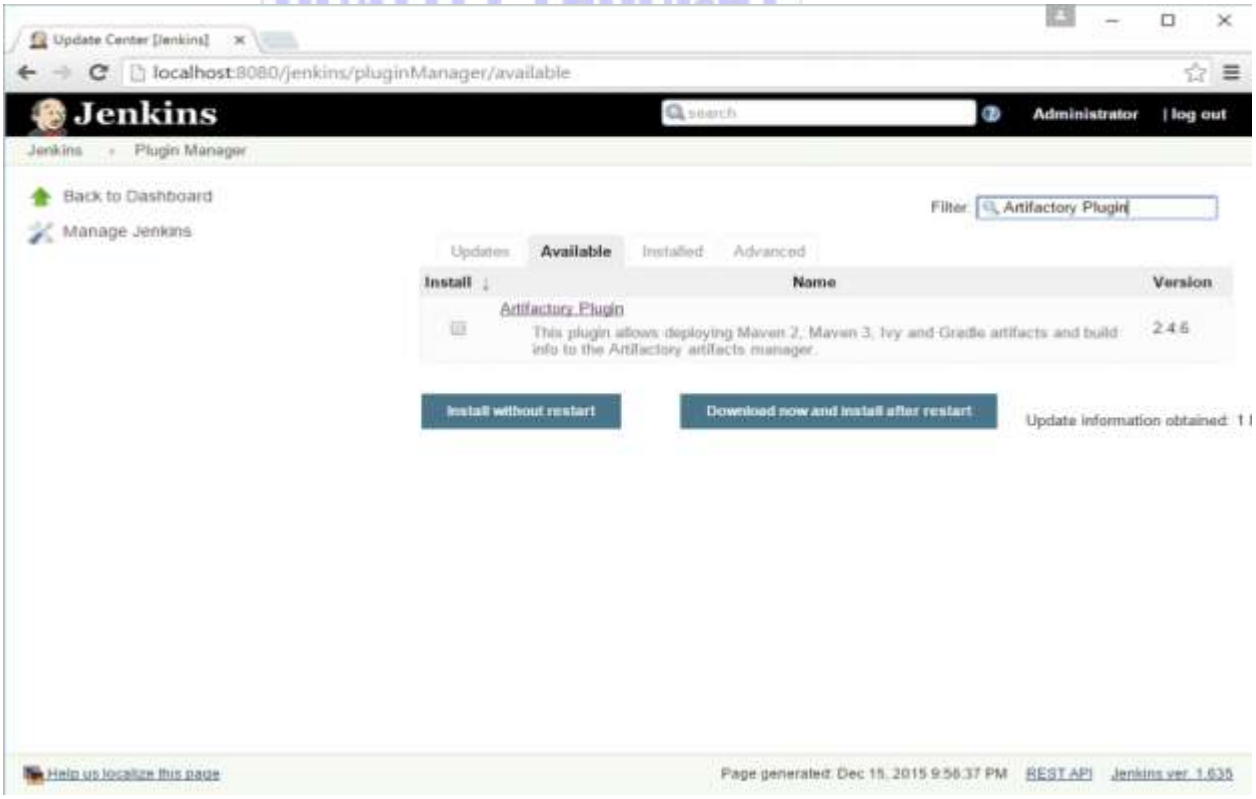
Note

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.

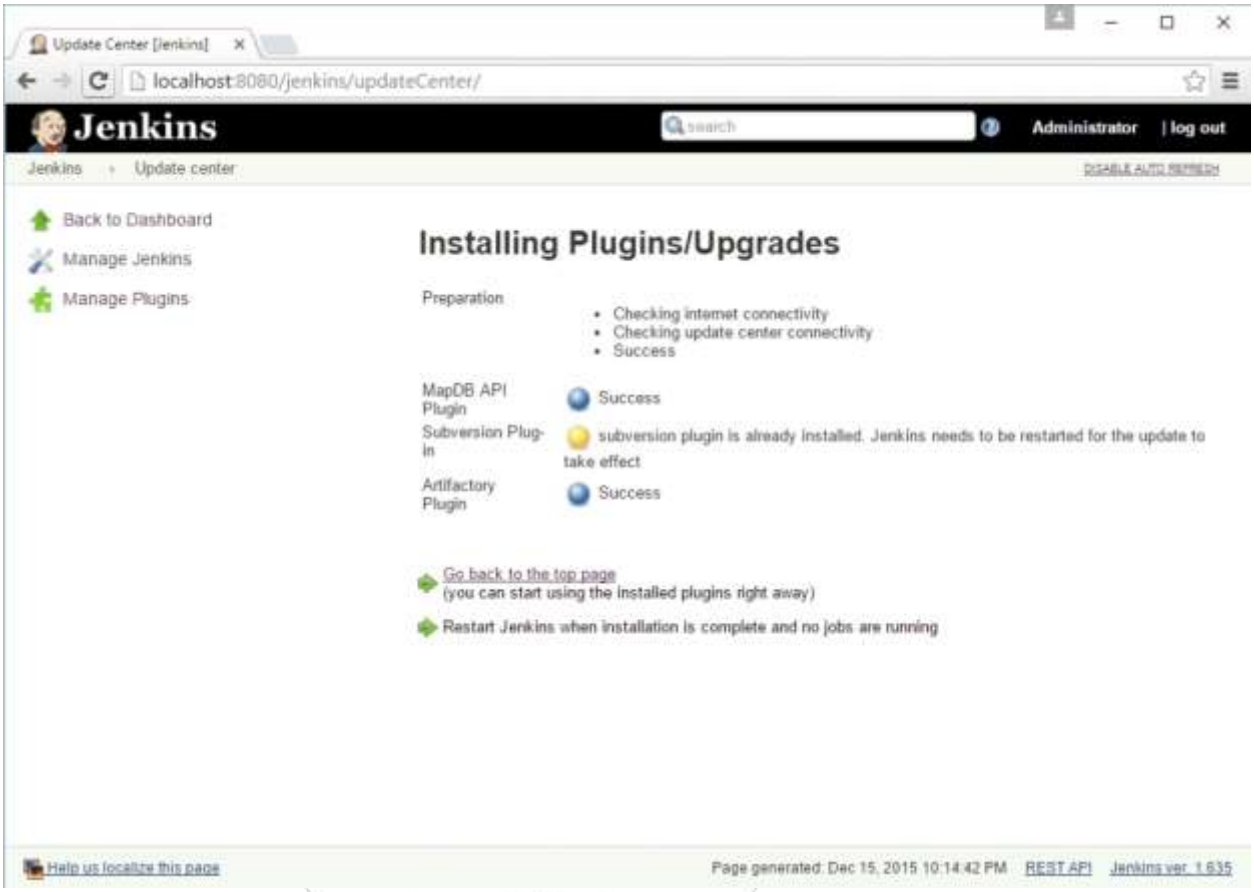
Installing the Artifactory plugin

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.

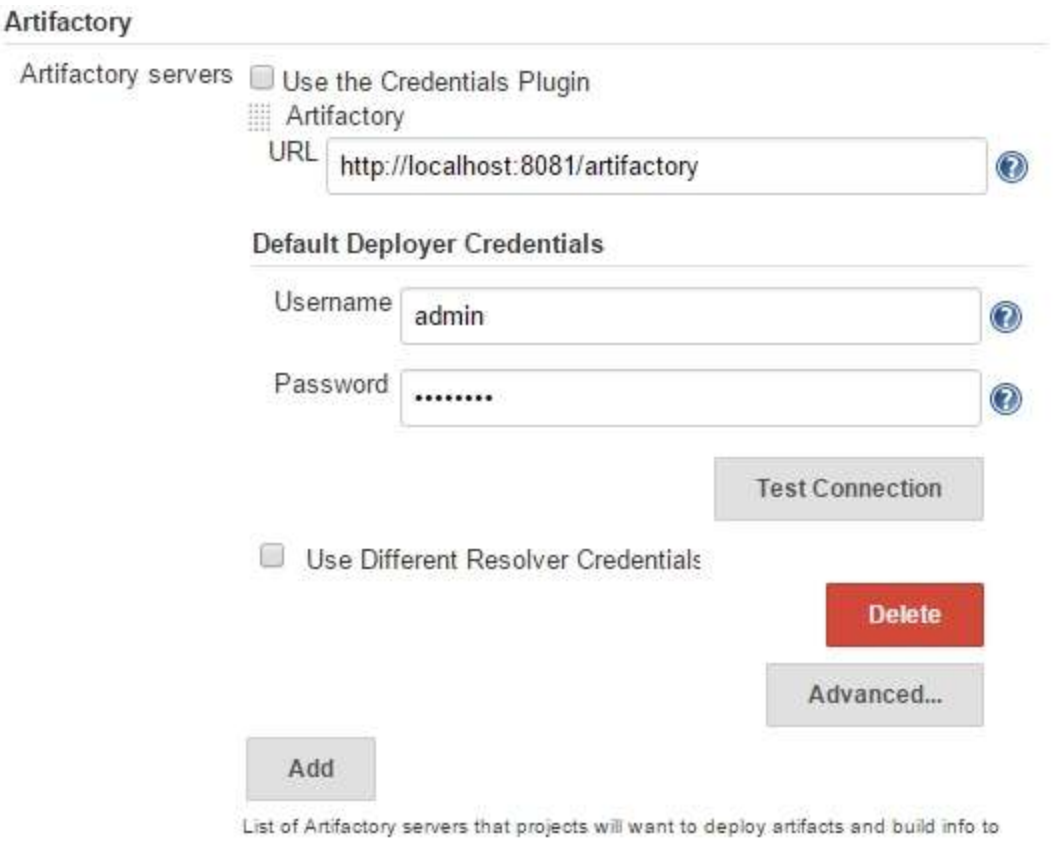


5. Upon successful installation of the Artifactory Plugin, go to the **Configure System link** on the **Manage Jenkins** page.

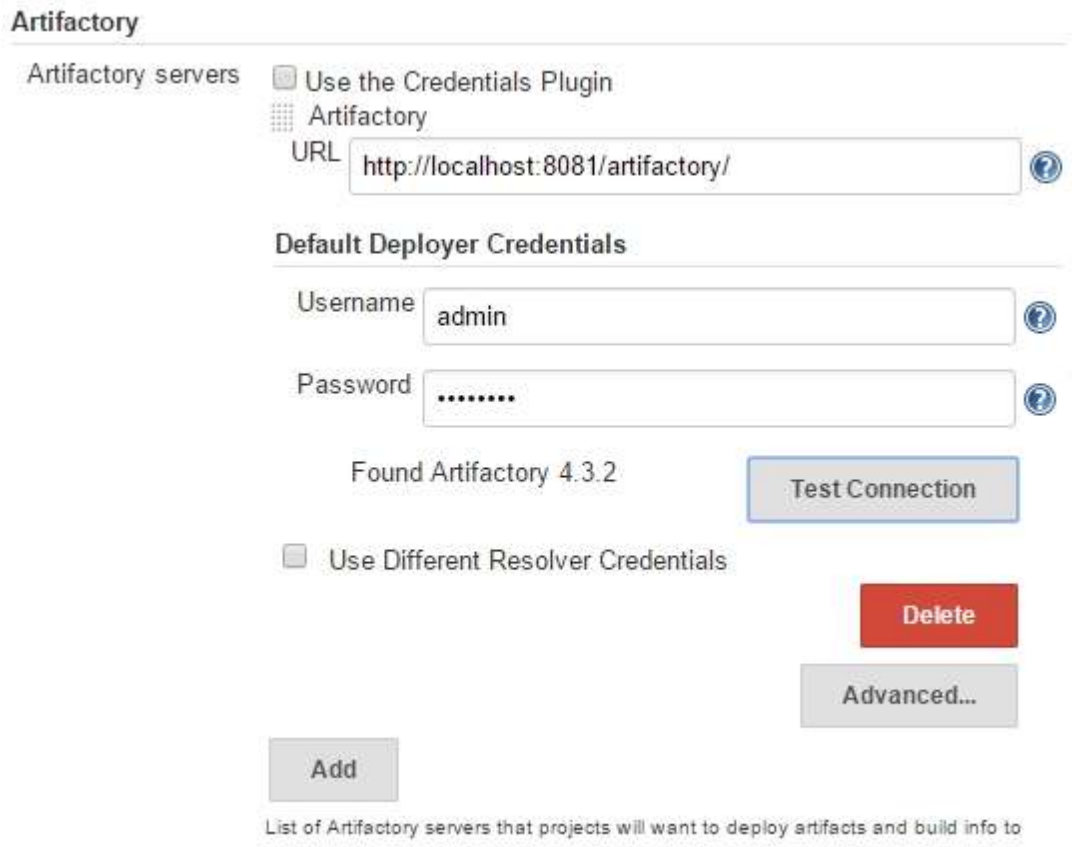
6. Scroll down until you see the **Artifactory** section and fill in the blanks as shown here:

    1. Provide the **URL** field as the default Artifactory URL configured at the time of installation. In our example, it is `http://localhost:8081/artifactory`.

    2. 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:

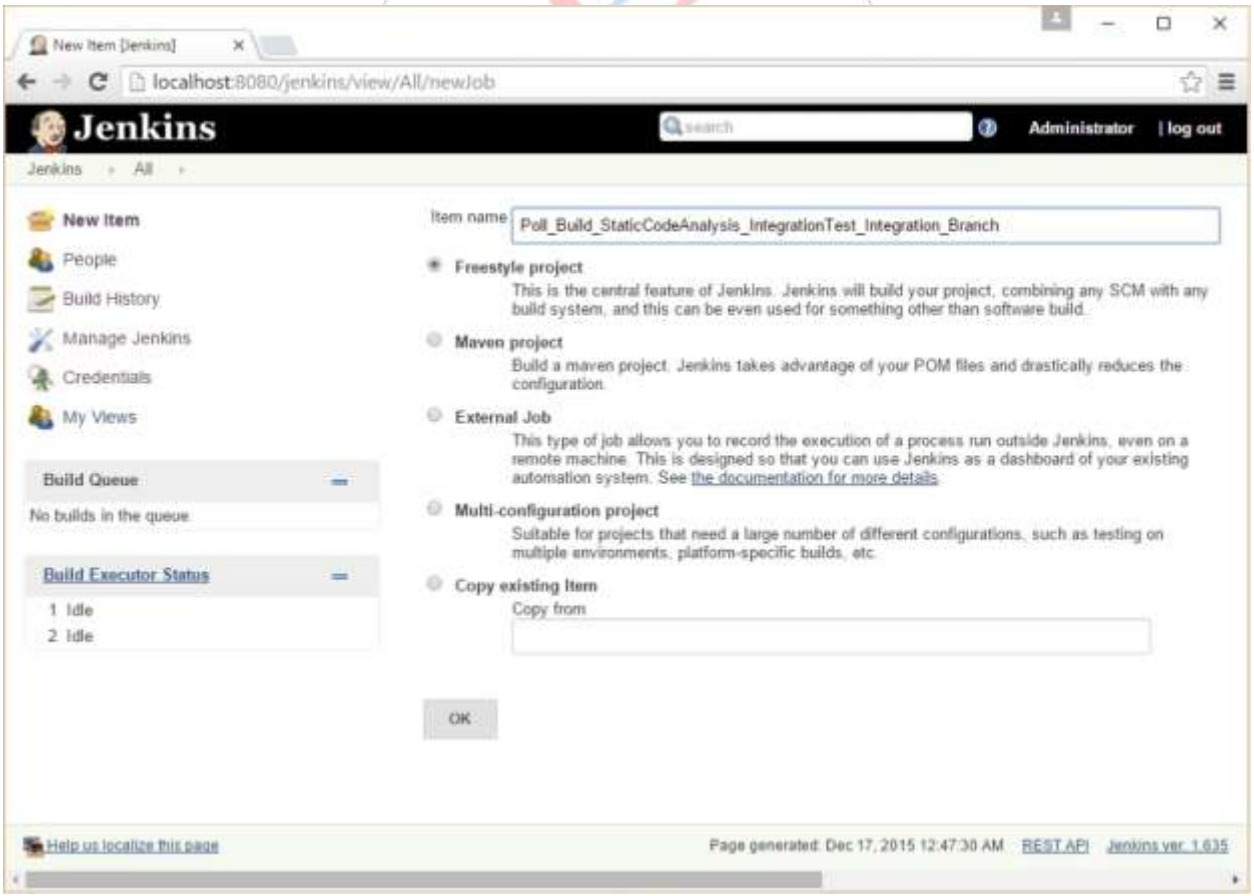
- It polls the integration branch for changes at regular intervals
- It executes the static code analysis
- It performs a build on the modified code

- 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:

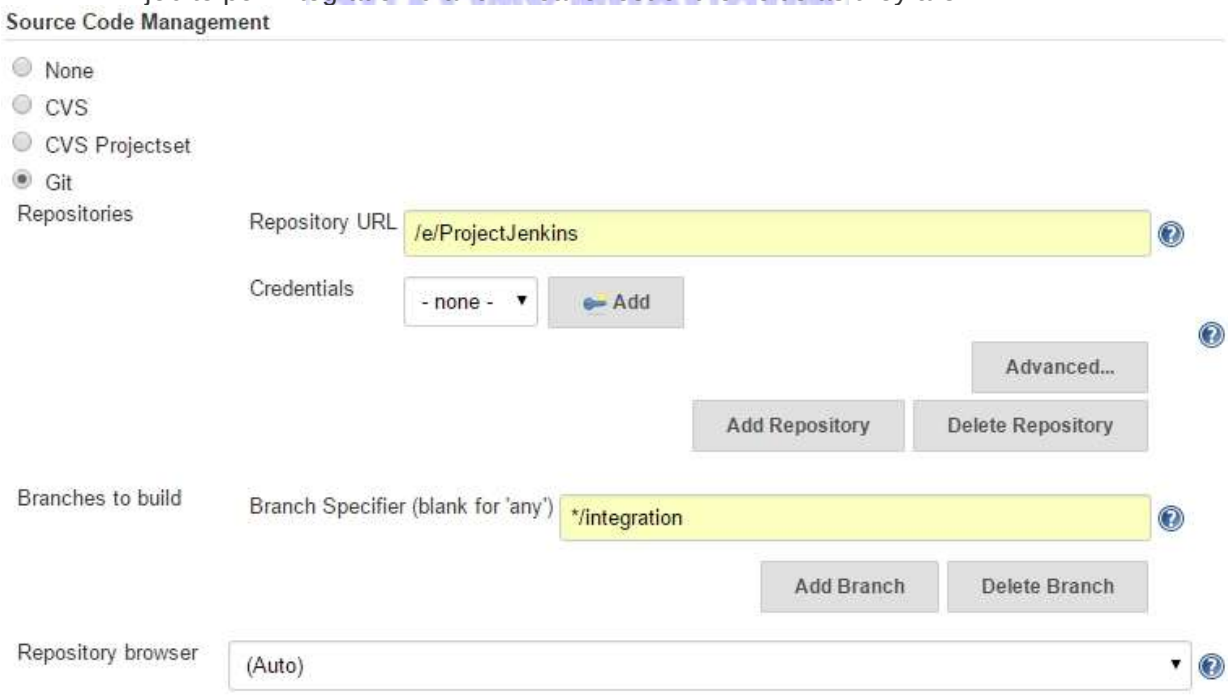
1. Click on **New Item**.
2. Name your new Jenkins job `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch`.
3. Set the type of job as **Freestyle project** and click on **OK** to proceed.



Polling the version control system for changes using Jenkins

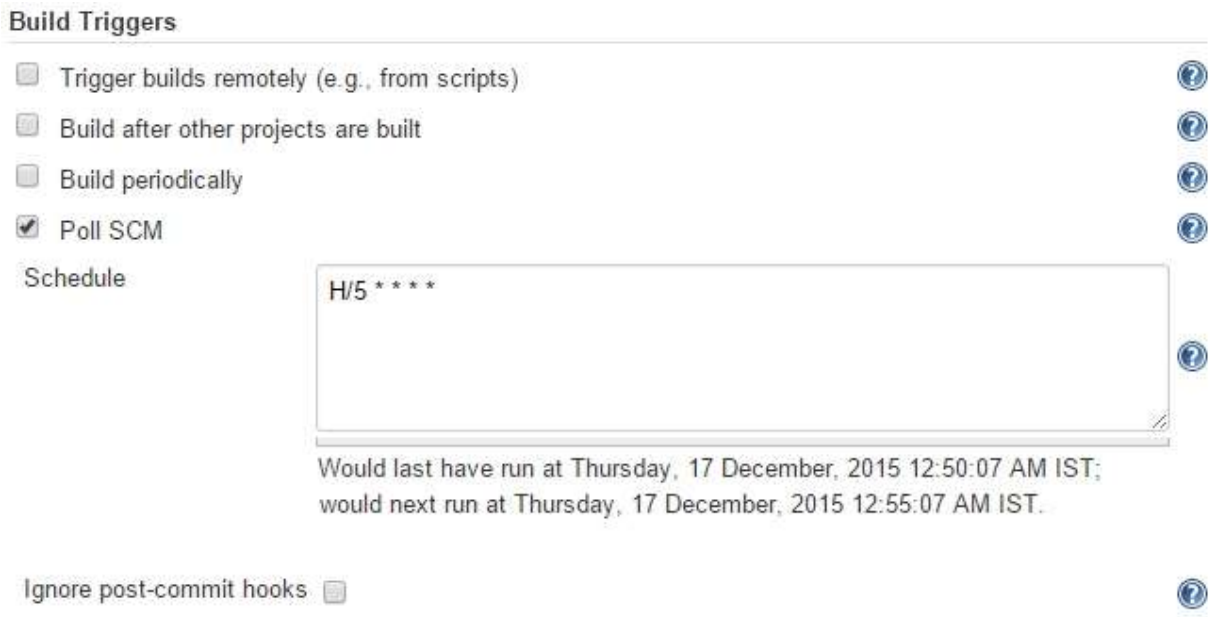
This is a critical step in which we connect Jenkins with the version control system. This configuration enables Jenkins to poll the correct branch inside Git and download the modified code.

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.

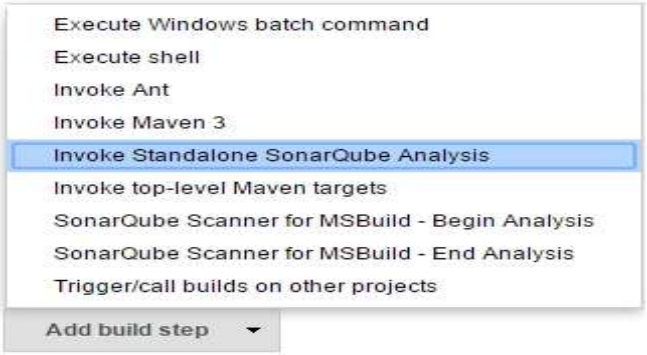
4. We want our Jenkins job to poll the feature branch every 5 minutes. Nevertheless, you are free to choose the polling duration that you wish depending on your requirements. Therefore, select the **Poll SCM** checkbox and add `H/5 * * * *` in the **Schedule** field.



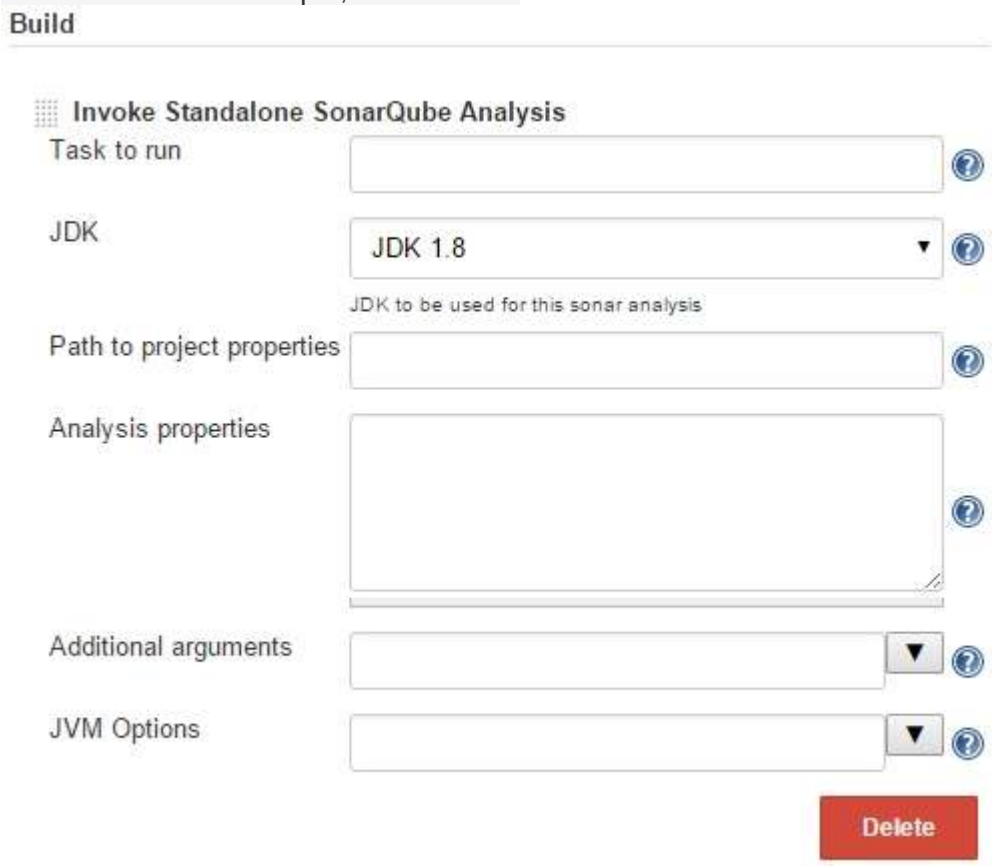
Creating a build step to perform static analysis

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**.



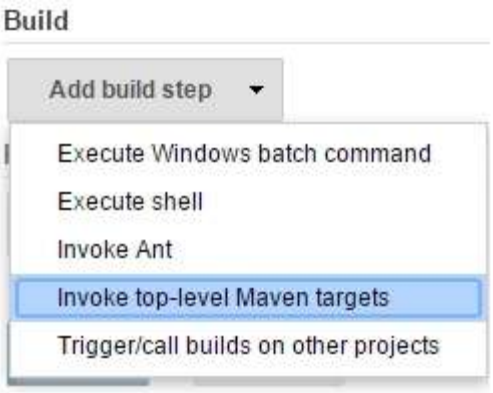
2. Leave all the fields empty except the **JDK** field. Choose the appropriate version from the menu. In our example, it's **JDK 1.8**.



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**.



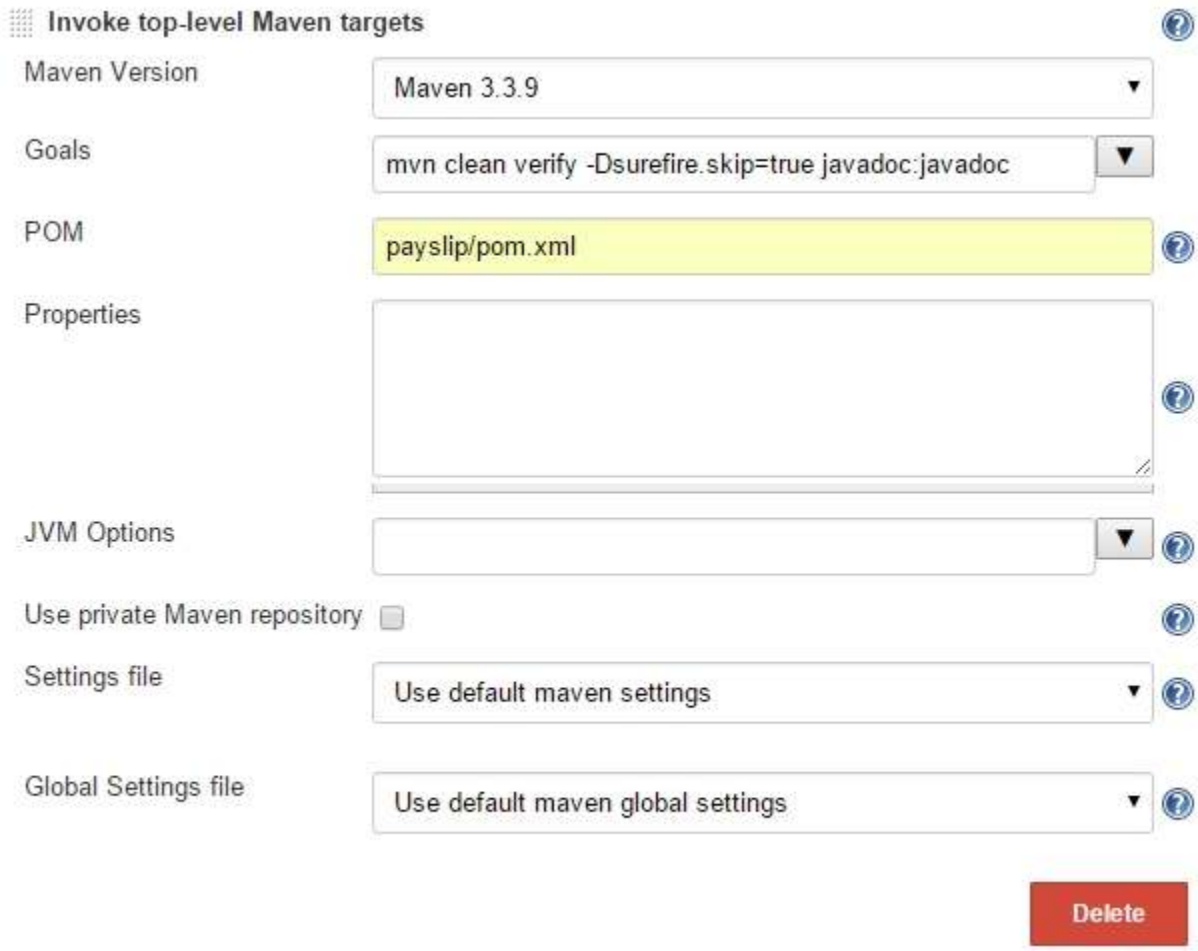
2. We will be present with the following options:

   - 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:

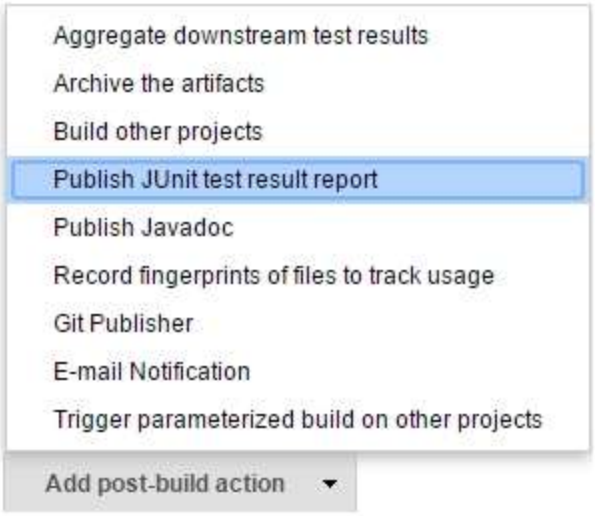
```
clean verify -Dsurefire.skip=true javadoc:javadoc
```

   - 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:

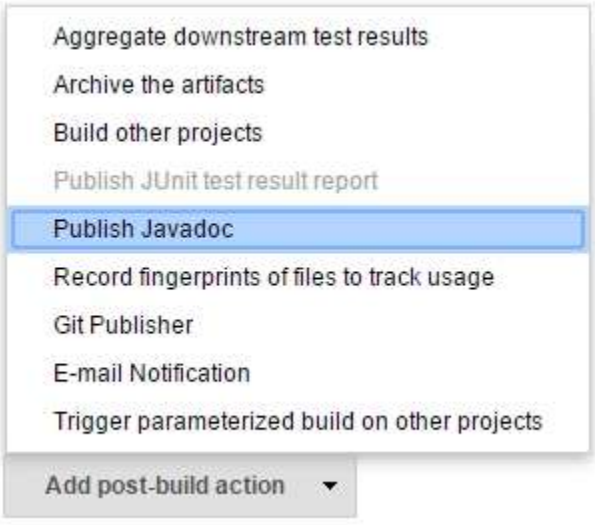


4. Let's see the Maven command inside the **Goals** field in detail:
   - `clean` will clean any old built files
   - `-Dsurefire.skip=true` will execute the integration test
   - `javadoc:javadoc` will tell Maven to generate Java documentation
5. Scroll down to the **Post build Actions** section.
6. Click on the **Add post-build action** button and select **Publish JUnit test result report**, as shown in the following screenshot:

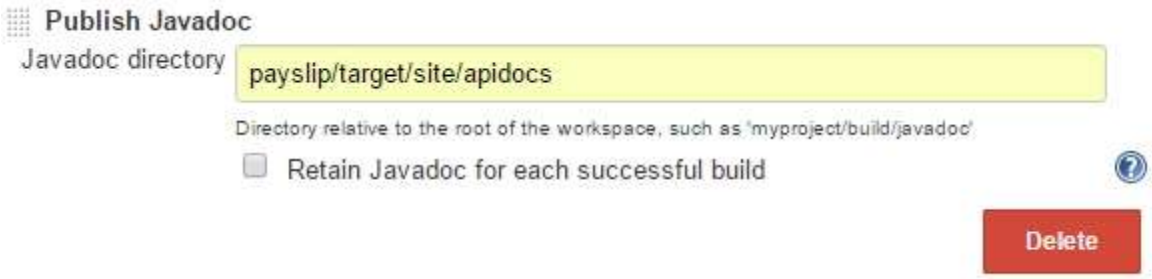


7. Under the **Test report XMLs** field, type `payslip/target/surefire-reports/*.xml`.

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.

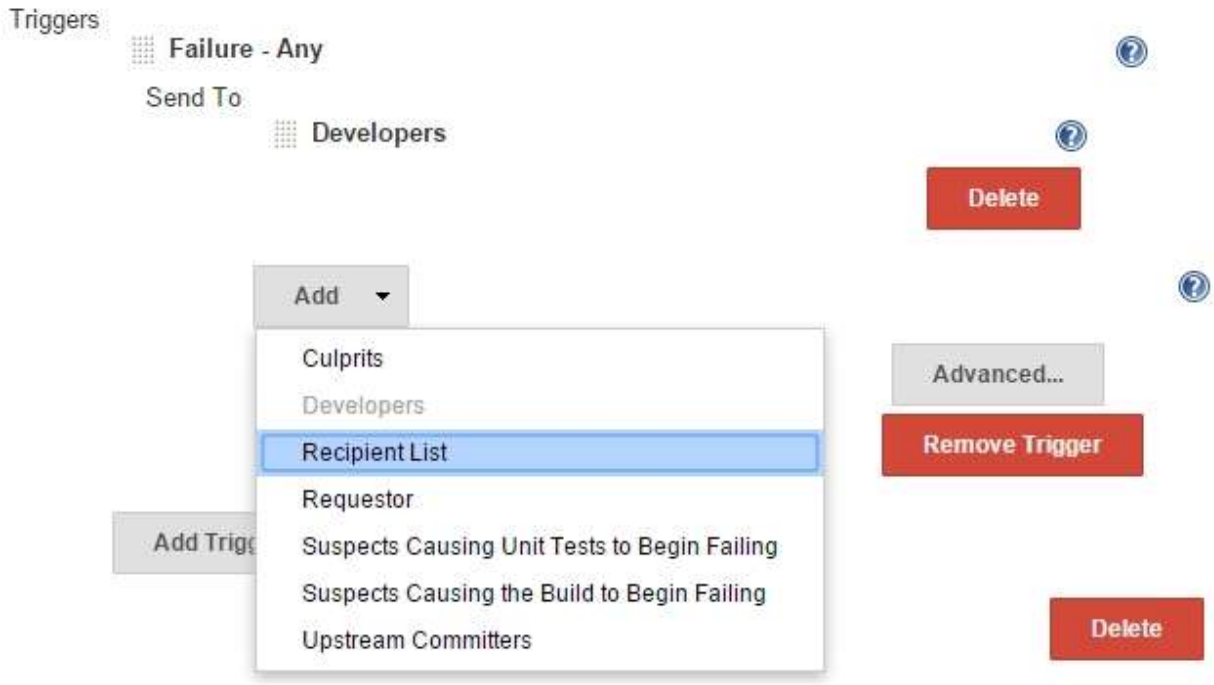


Configuring advanced e-mail notifications

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:

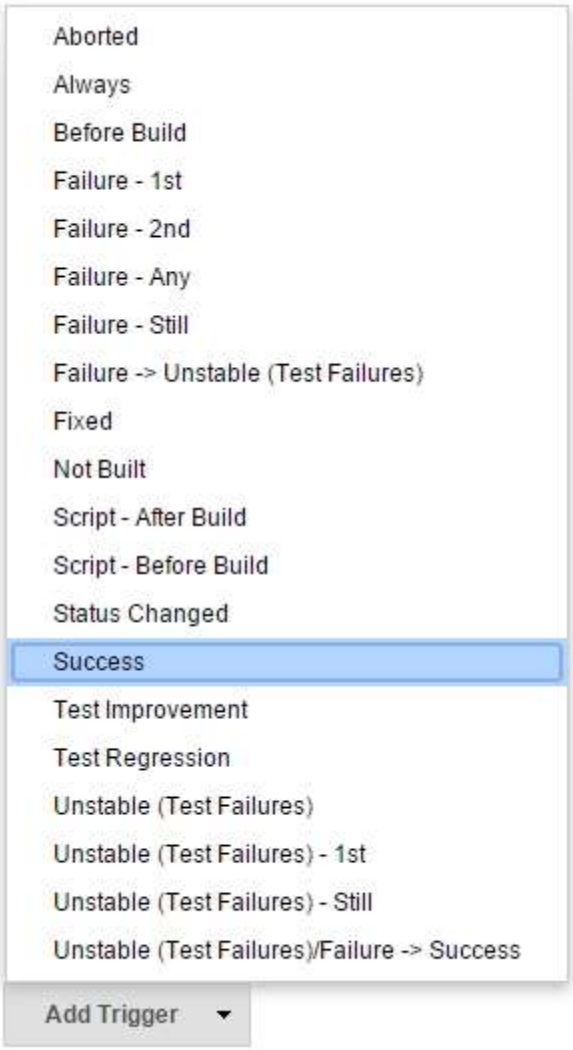
1. Click on the **Add post-build action** button and select **Editable Email Notification**.
2. Configure **Editable Email Notification** as shown here:
    - Under **Project Recipient List**, add the e-mail IDs separated by commas. You can add anyone whom you think should be notified for build and unit 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:

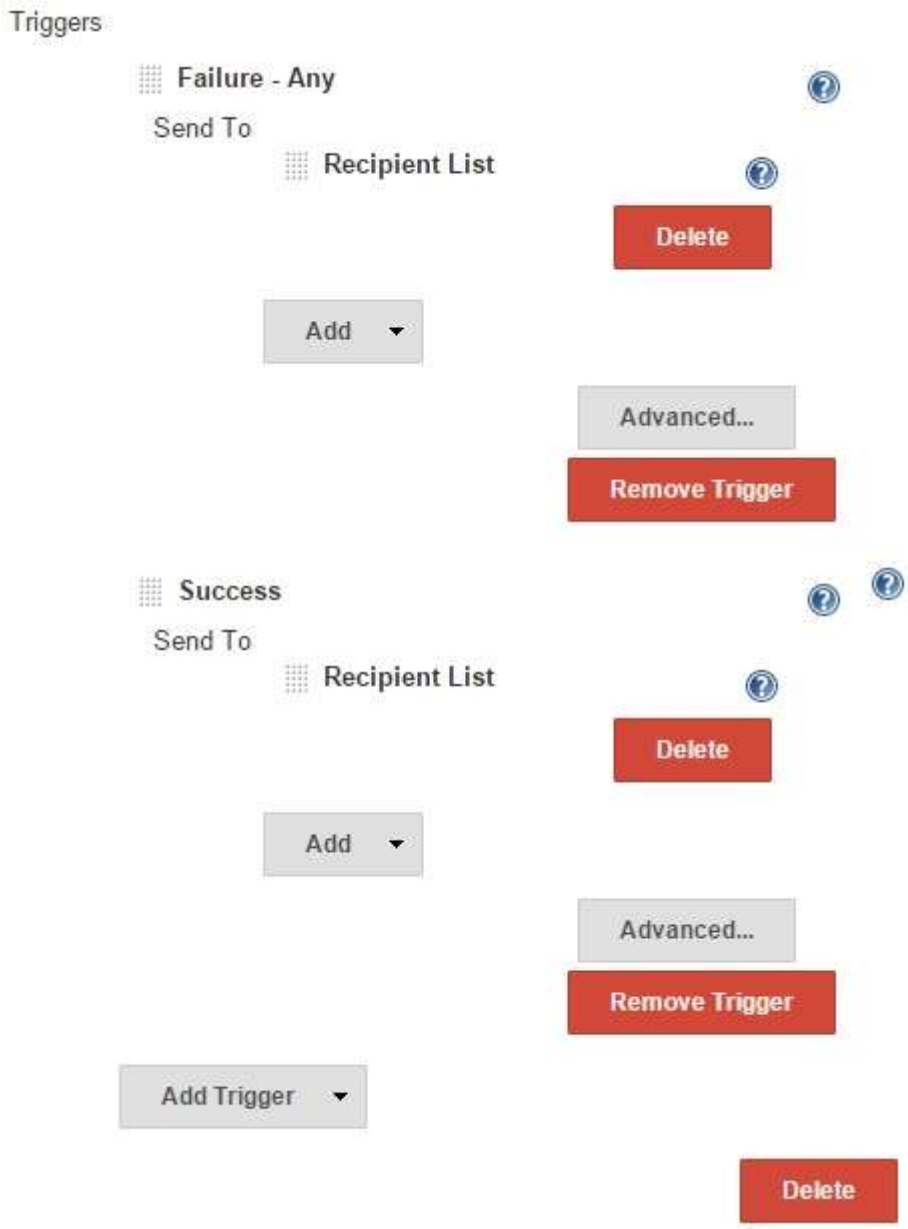


8. Let's add another trigger to send an e-mail when the job is successful.

9. Click on the **Add Trigger** button and select the **Success** option.



10. 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:



11. 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.

Note

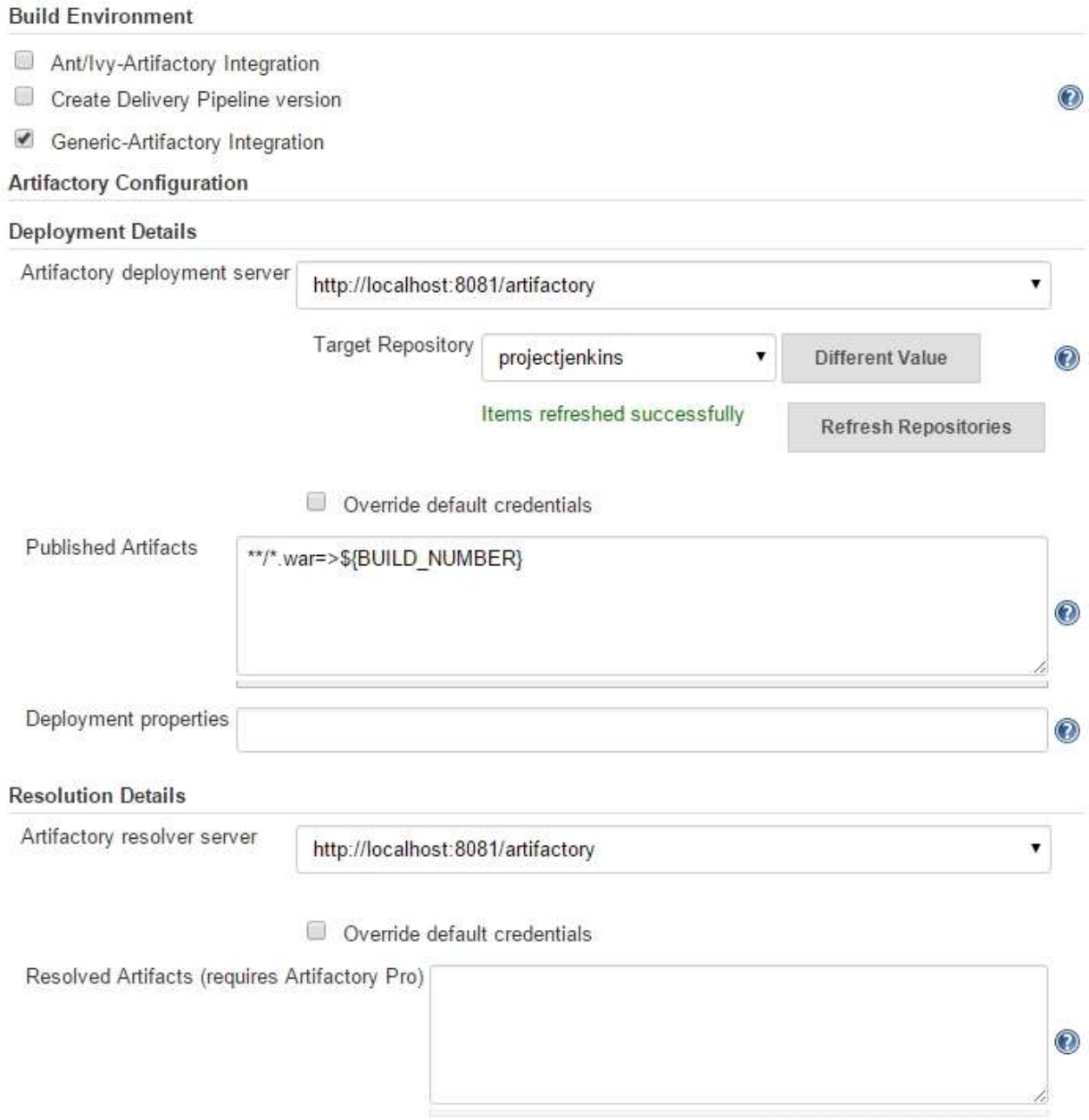
In this way, we are telling Jenkins to initiate the current Jenkins job `Upload_Package_To_Artifactory` only after the `Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_Branch` job has completed successfully.

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
/YC:\Jenkins\jobs\Poll_Build_StaticCodeAnalysis_IntegrationTest_Integration_B
ranch\workspace\payslip\target\payslip-0.0.1.war %WORKSPACE%\payslip-
0.0.1.war
```

Note

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.

6. Continuous Delivery Using Jenkins

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

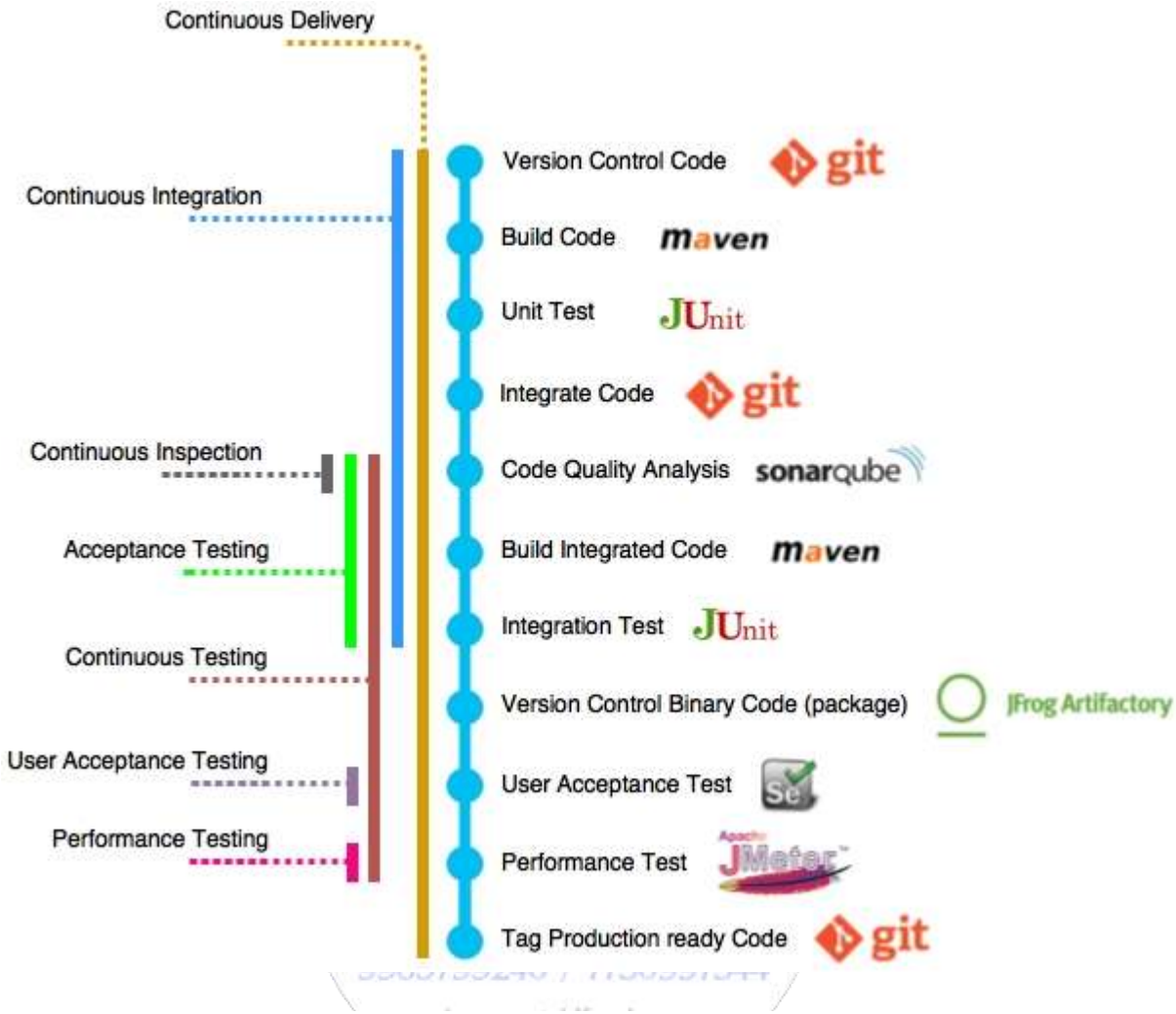
- However, the list is not complete. You can incorporate as many types of testing as you want to certify that the code is production ready.

- 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:

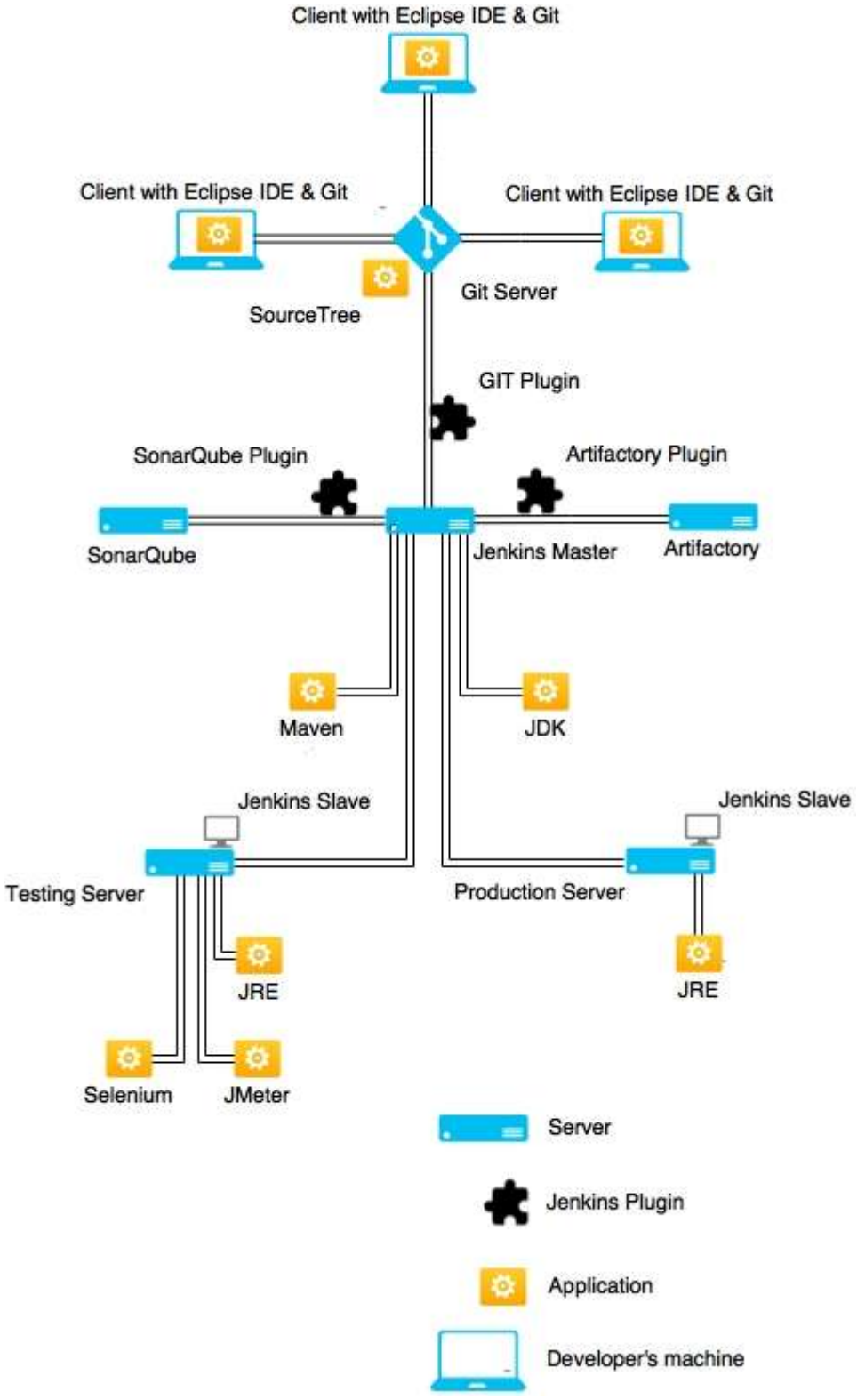


Continuous Delivery Design

The 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.

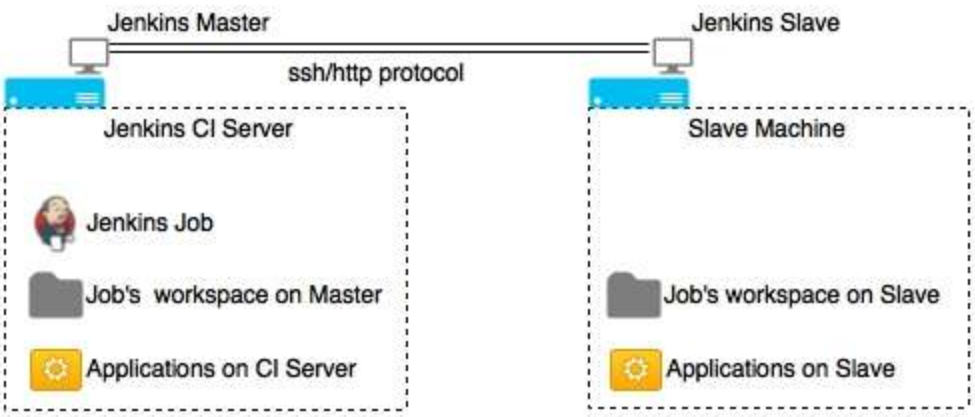

Client with Eclipse IDE & Git
Client with Eclipse IDE & Git
Client with Eclipse IDE & Git
SourceTree
Git Server
GIT Plugin
SonarQube Plugin
Artifactory Plugin
SonarQube
Jenkins Master
Artifactory
Maven
JDK
Jenkins Slave
Jenkins Slave
Testing Server
Production Server
JRE
JRE
Selenium
JMeter
Server
Jenkins Plugin
Application
Developer's machine

Jenkins configuration

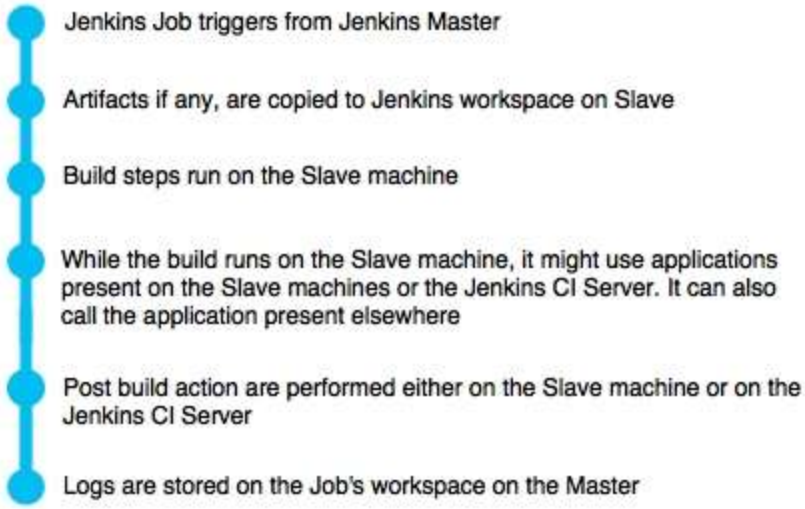
Configuring Jenkins slaves on the testing server

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:

Jenkins Master-Slave Architecture



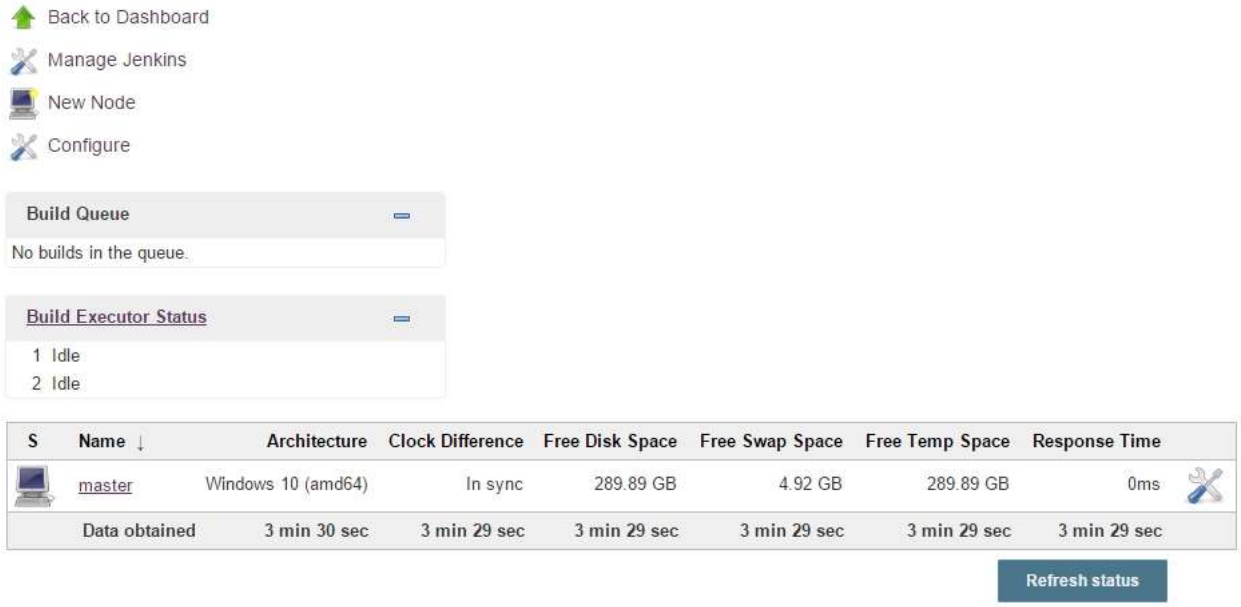
Timeline of a Jenkins Job during execution



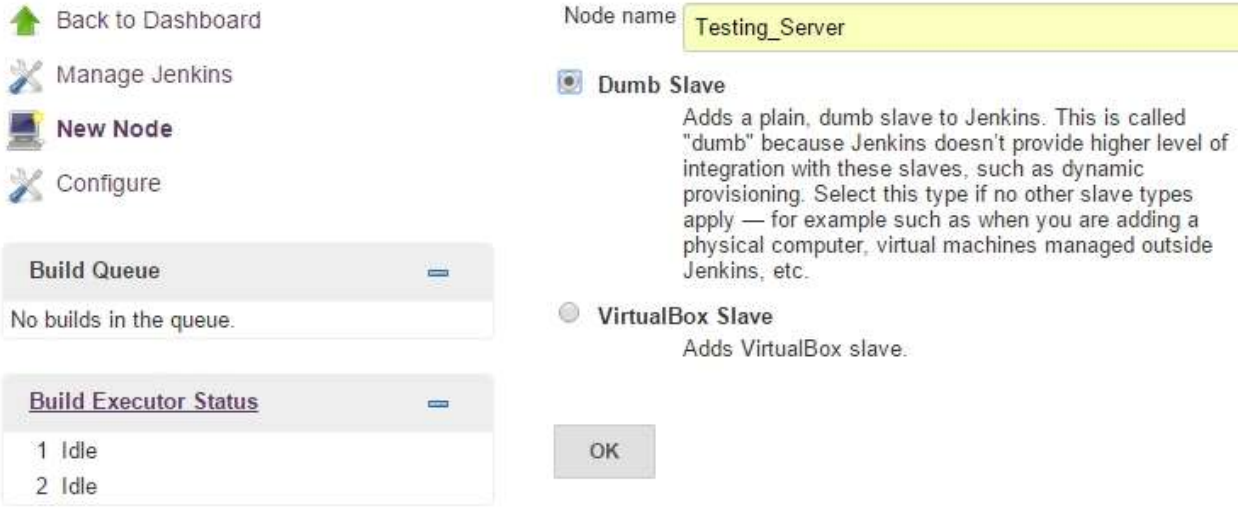
1. Log in to the testing server and open the Jenkins dashboard from the browser using the following link: `http://<ip address>:8080/jenkins/`. Remember, you are accessing the Jenkins master from the testing server. `<ip address>` is the IP of your Jenkins server.

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:



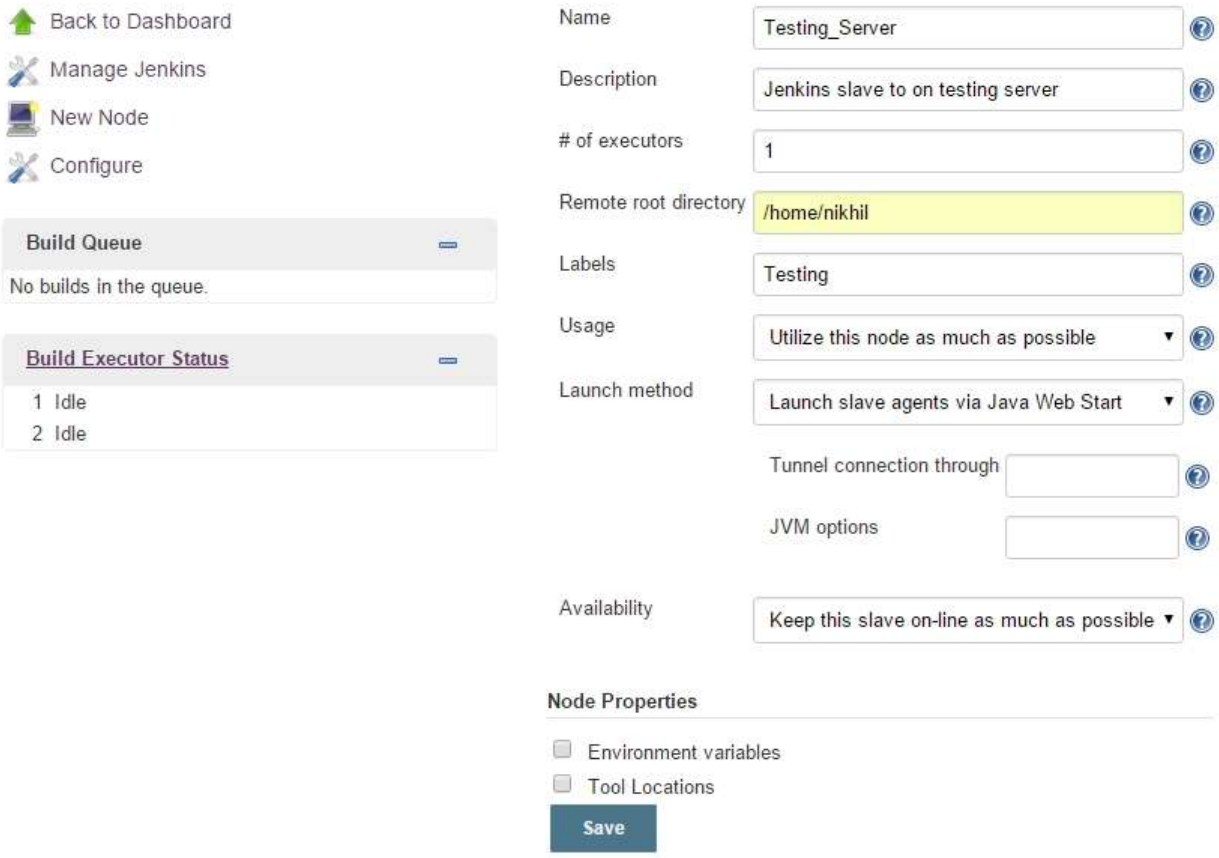
4. Click on the **New Node** button on the left-hand side panel. Name the new node `Testing_Server` and select the option **Dumb Slave**. Click on the **OK** button to proceed:



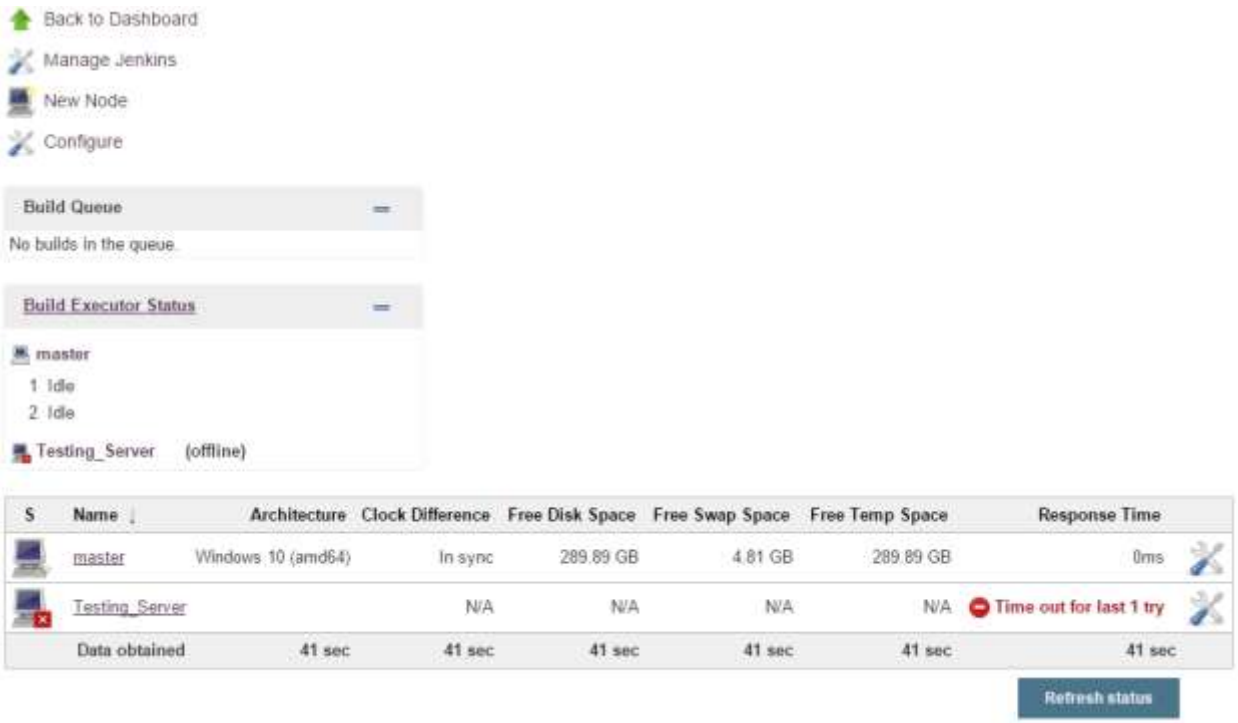
5. Add some description, as shown in the next screenshot. The **Remote root directory** value should be the local user account on the testing server. It should be `/home/<user>`.

   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:

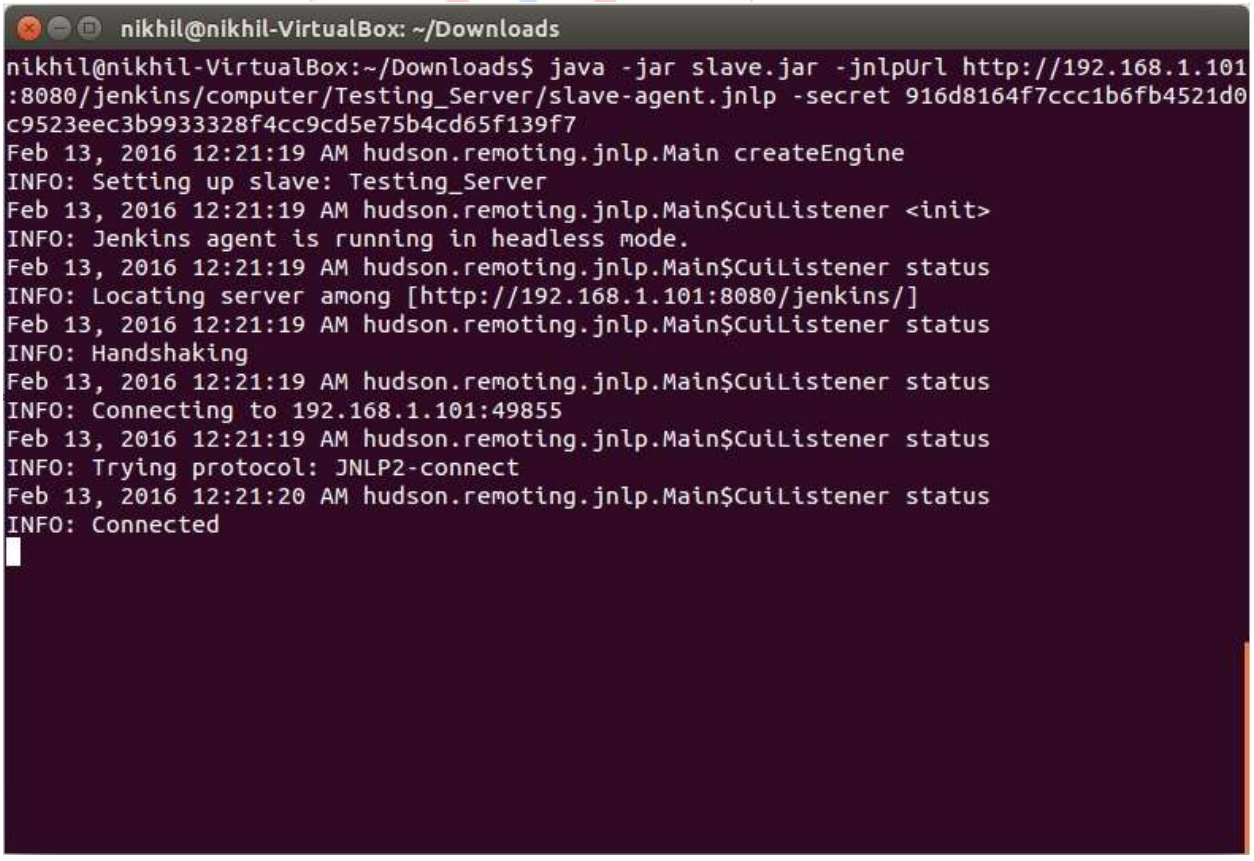
```
cd Downloads

java -jar slave.jar -jnlpUrl
http://192.168.1.101:8080/jenkins/computer/Testing_Server/slav e-agent.jnlp -
secret 916d8164f7ccc1b6fb4521d0c9523eec3b9933328f4cc9cd5e75b4cd65f139f7
```

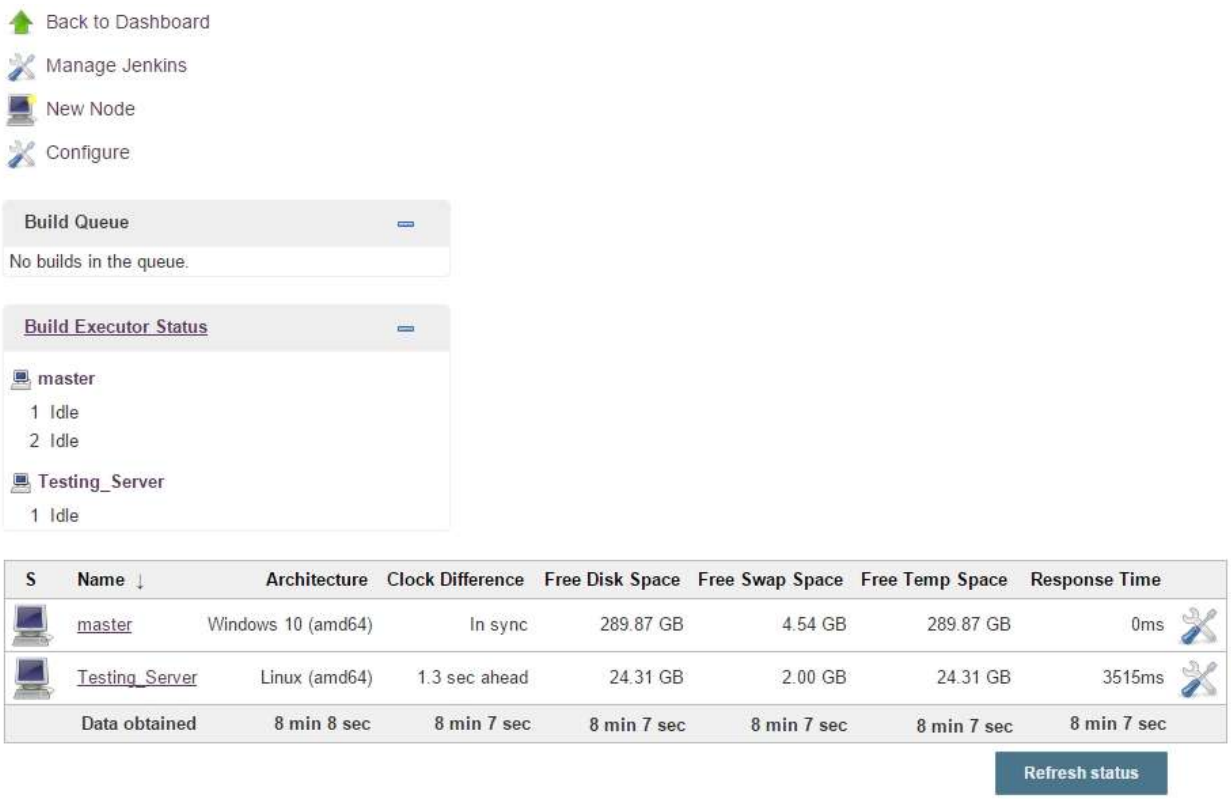
Note

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:



Slave agents via SSH tunneling

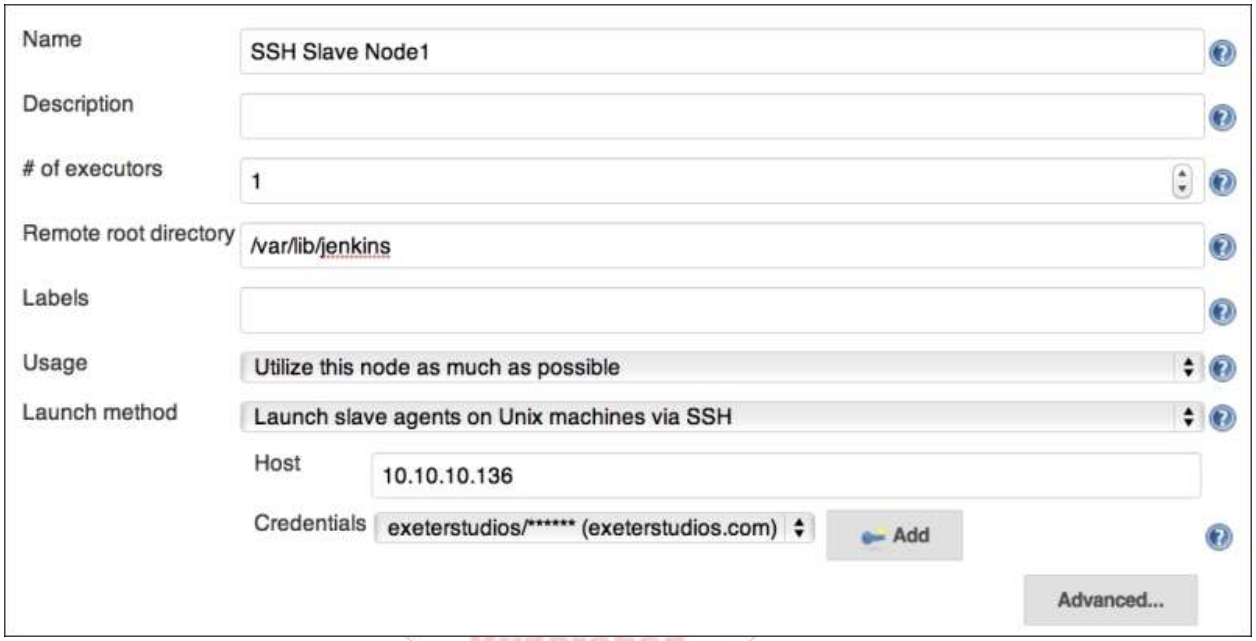
The widely preferred approach for Jenkins slave nodes on Linux, Unix, and OS X hosts is to leverage SSH tunneling. This launch method starts by sending commands over an SSH connection, which downloads the `slave.jar` and launches the slave agent on the host. For the installation process to work, Java 1.7 or later must be installed; the slave host needs to be reachable from the master, and the account specified in Jenkins will need to have SSH logon rights for the target machine.

The SSH launch method provides a number of valuable features that make this an attractive option when connecting Jenkins slave agents to the master. These benefits include:

- More reliable connectivity and stability
- Encrypted communications

- Auto restart and reconnect functionality
- No need for slave services or `init.d` scripts

To use the SSH launch method, select **Launch slave agents on Unix machines via SSH**, SSH authorized user credentials, Host IP address, and click the SAVE button to create the new slave node. Once the slave has been saved, Jenkins will automatically attempt to connect to the slave and install the slave agent using SSH and the credentials provided.



When configuring a new SSH slave node, the best approach for configuring authentication is to use the Jenkins credential management system. This will store the login and password information for the SSH slaves in Jenkins directly. The Jenkins credential management system allows the Jenkins administrator to manage credentials and later reuse them when executing jobs, connecting SSH slave agents, and connecting Jenkins to third-party services. To add usernames and passwords to the credentials manager, navigate to the credential management system and select add credentials in the UI:

Manage Jenkins | Manage Credentials | Add credentials

Once any credentials have been added, they will appear as available credentials in the SSH Host **Credentials** dropdown

Upon saving the configuration for an SSH slave node, Jenkins will immediately attempt to connect to and install the slave agent service on the target host.

Detailed logs related to the connection can be viewed by clicking the Log button on the left-hand side of the slave node status screen. If everything was successful, the logs will contain text similar to the following:

```
JNLP agent connected from /127.0.0.1
<===[JENKINS REMOTING CAPACITY]===>Slave.jar version: 2.49
This is a Unix slave
Slave successfully connected and online.
```

Modifying the project to run on selected node

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:

Creating a Jenkins job to deploy code on the testing server

- 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:

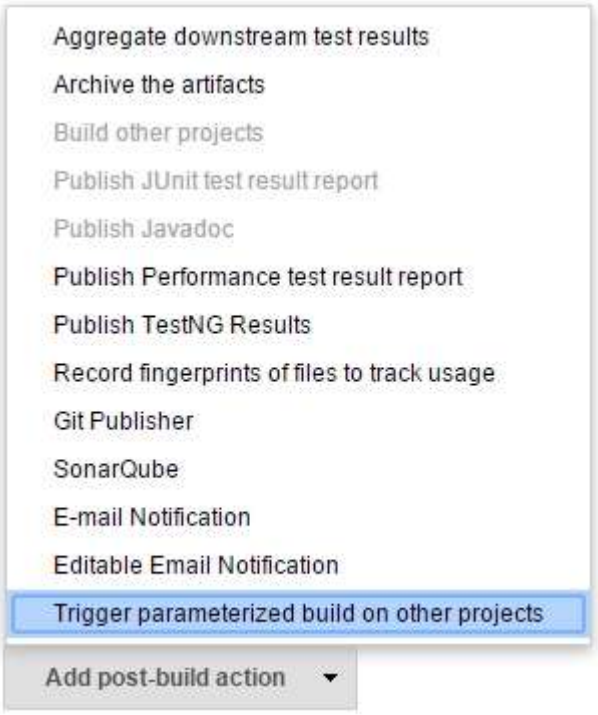
      wget

      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:

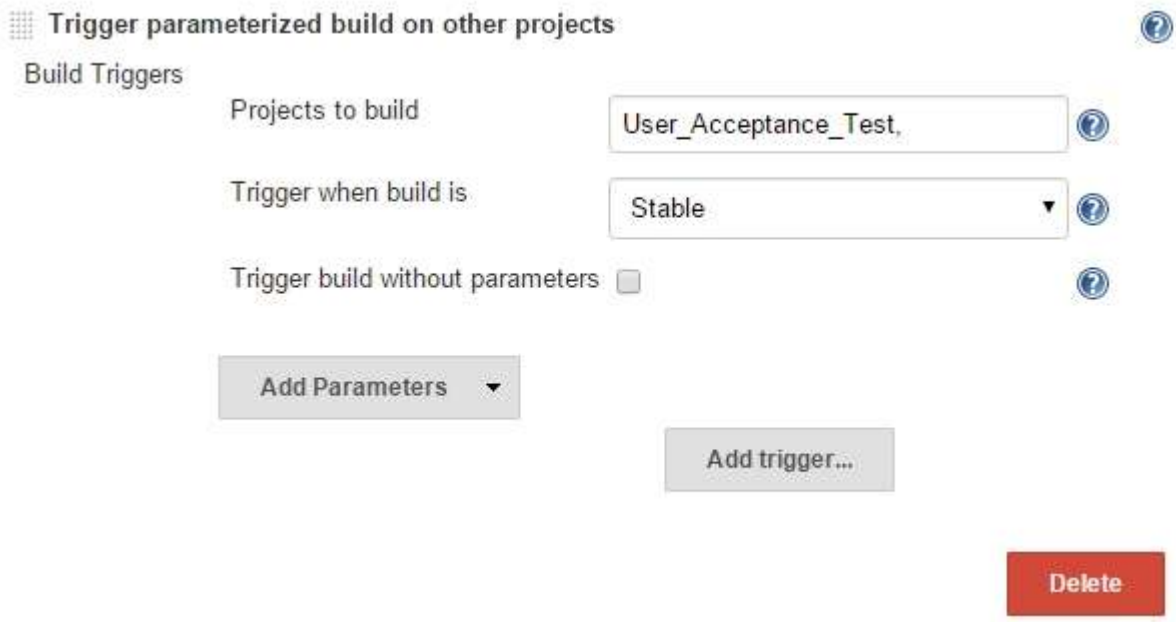
      mv payslip-0.0.1

      .war /opt/tomcat/webapps/payslip-0.0.1.war –f

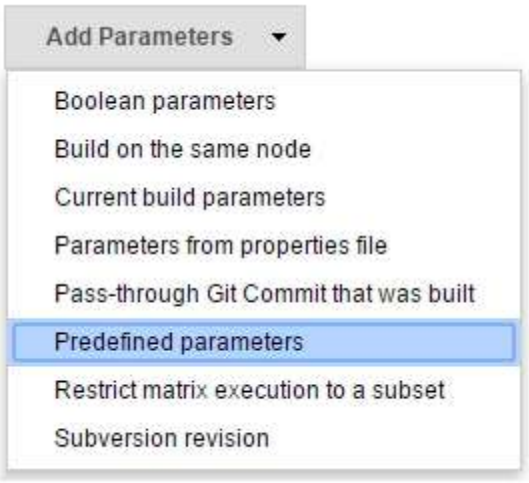
9. Scroll down to the **Post-build Actions** section. Click on the **Add post-build action** button. From the drop-down list, choose the option **Trigger parameterized build on the other projects**:



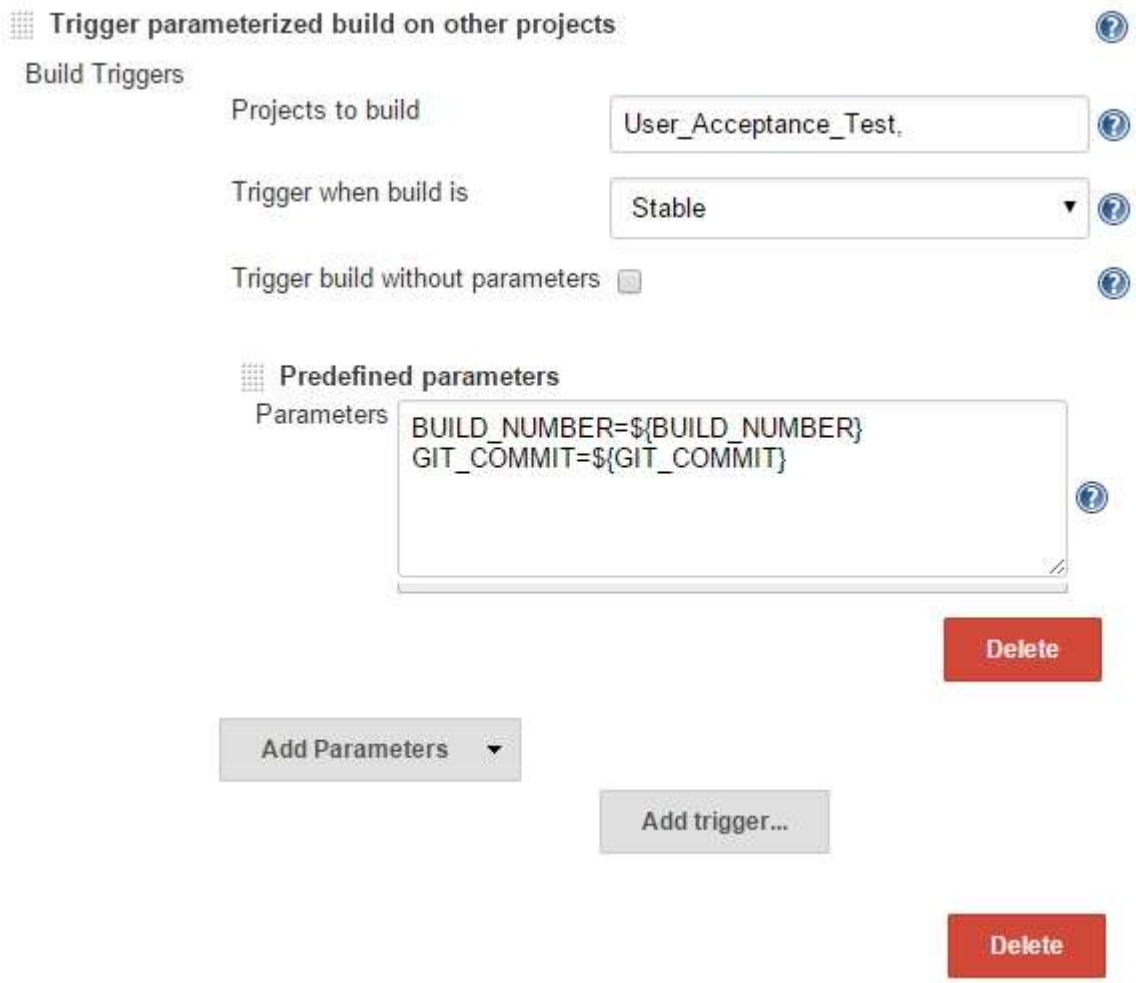
10. Add the values as shown in the screenshot:



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.