JAVA BOOTCAMP TRAINING DAY 17

Presented by

G2Academy

GITLAB TUTORIAL

G2 | Academy

What is Gitlab?

- Before we dive into definition for Gitlab, first we need to understand few terminologies. We often come across these terms like Git, Gitlab, GitHub, and Bitbucket. Let's see definiton of all these as below −
- **Git -** It is a source code versioning system that lets you locally track changes and push or pull changes from remote resources.
- **GitLab, GitHub, and Bitbucket -** Are services that provides remote access to Git repositories. In addition to hosting your code, the services provide additional features designed to help manage the software development lifecycle. These additional features include managing the sharing of code between different people, bug tracking, wiki space and other tools for 'social coding'.
  - **GitHub** is a publicly available, free service which requires all code (unless you have a paid account) be made open. Anyone can see code you push to GitHub and offer suggestions for improvement. GitHub currently hosts the source code for tens of thousands of open source projects.
  - **GitLab** is a github like service that organizations can use to provide internal management of git repositories. It is a self hosted Git-repository management system that keeps the user code private and can easily deploy the changes of the code.

Why to use GitLab?

- GitLab is great way to manage git repositories on centralized server. GitLab gives you complete control over your repositories or projects and allows you to decide whether they are public or private for free.
- Features
  - GitLab hosts your (private) software projects for free.
  - GitLab is a platform for managing Git repositories.
  - GitLab offers free public and private repositories, issue-tracking and wikis.
  - GitLab is a user friendly web interface layer on top of Git, which increases the speed of working with Git.
  - GitLab provides its own *Continuous Integration* (CI) system for managing the projects and provides user interface along with other features of GitLab.

Why to use GitLab?

- Advantages
  - GitLab provides *GitLab Community Edition* version for users to locate, on which servers their code is present.
  - GitLab provides unlimited number of private and public repositories for free.
  - The *Snippet* section can share small amount of code from a project, instead of sharing whole project.
- Disadvantages
  - While pushing and pulling repositories, it is not as fast as GitHub.
  - GitLab interface will take time while switching from one to another page.

GitLab - Installation

- You can install the GitLab runner on different operating systems, by installing *Git* versioning system and creating user account in the GitLab site.
- *Git* is a version control system used for −
  - Handling the source code history of projects
  - Tracking changes made to files
  - Handling small and large projects with speed and efficiency
  - To collaborate with other developers on different projects

GitLab - Installation

- *GitLab* is a Git-based platform provides remote access to Git repositories and helpful for software development cycle by creating private and public repositories for managing the code.
- GitLab supports different types of operating systems such as Windows, Ubuntu, Debian, CentOS, open SUSE and Raspberry Pi 2.

Installation of GitLab on Windows:

- **Step 1 –** First create a folder called 'GitLab-Runner' in your system. For instance, you can create in C drive as C:\GitLab-Runner.
- **Step 2 –** Now download the binary for x86 or amd64 and copy it in the folder created by you. Rename the downloaded binary to *gitlab-runner.exe*.
- **Step 3 –** Open the command prompt and navigate to your created folder. Now type the below command and press enter.

```
C:\GitLab-Runner>gitlab-runner.exe register
```

Installation of GitLab on Windows:

- **Step 4 –** After running the above command, it will ask to enter the gitlab-ci coordinator URL

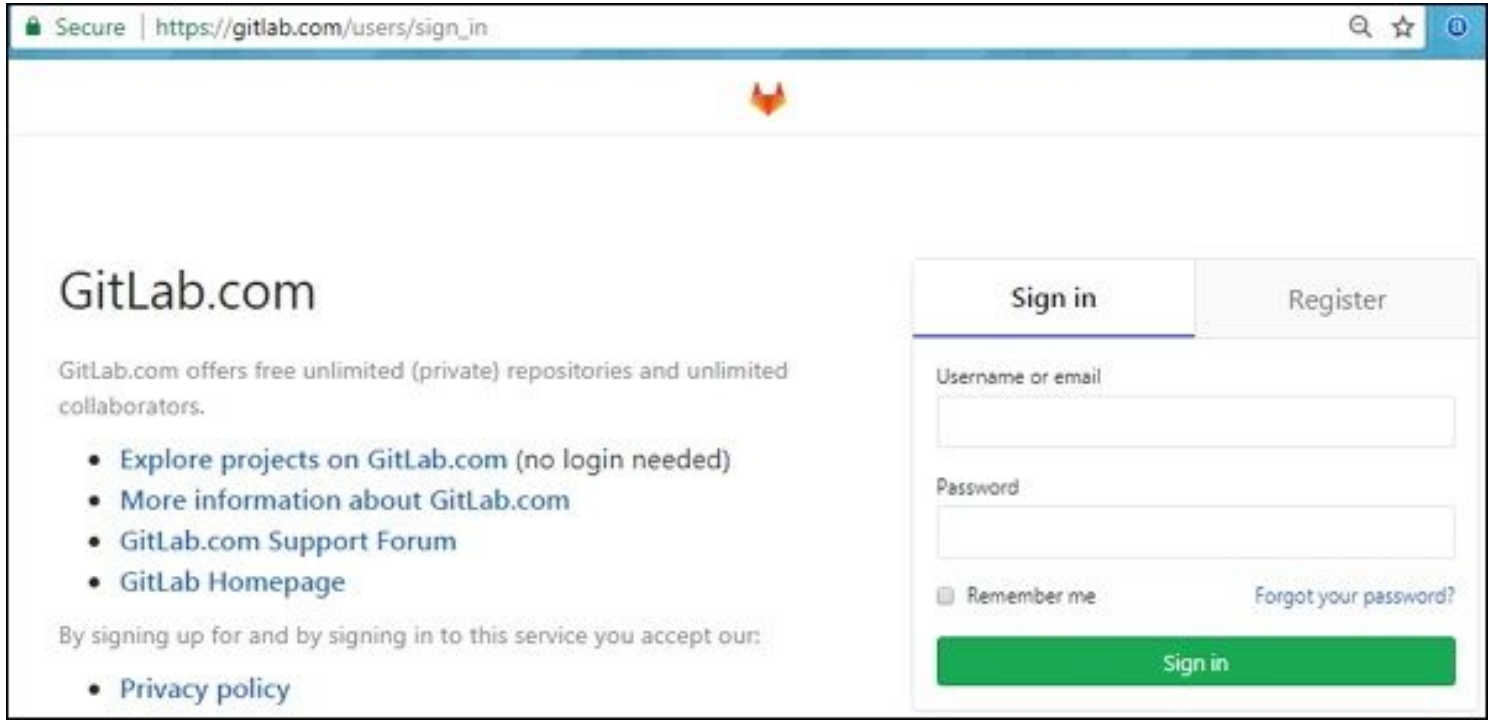
```
Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com/):
https://gitlab.com
```

- **Step 5 –** Enter the gitlab-ci token for the runner.

```
Please enter the gitlab-ci token for this runner:
xxxxx
```

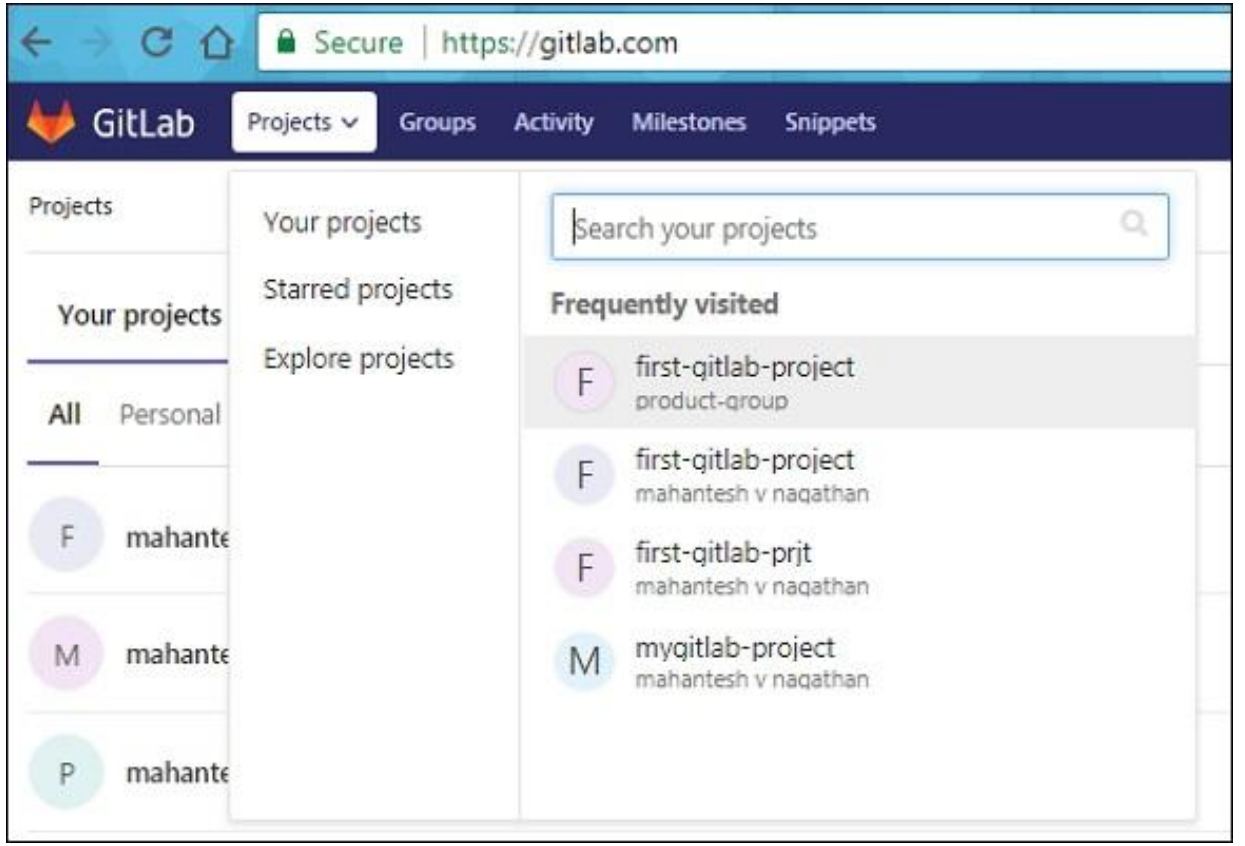
Installation of GitLab on Windows:

- To get the token, login to your GitLab account –



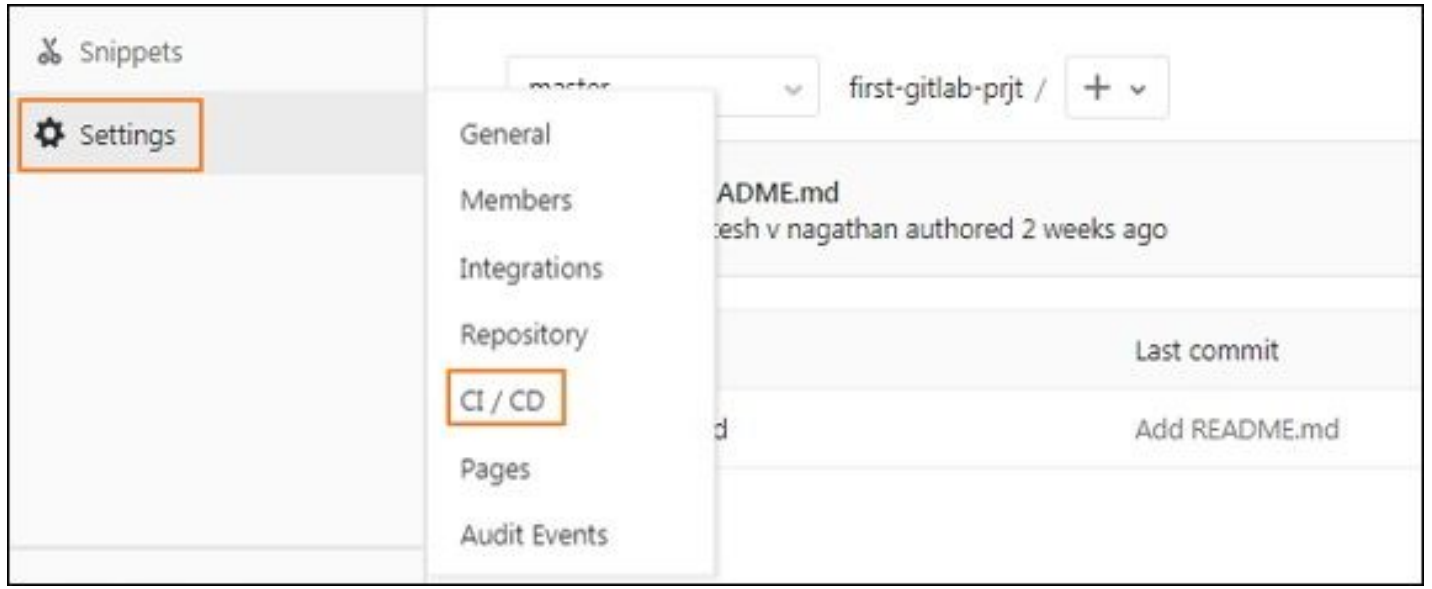
Installation of GitLab on Windows:

- Now go to your project –



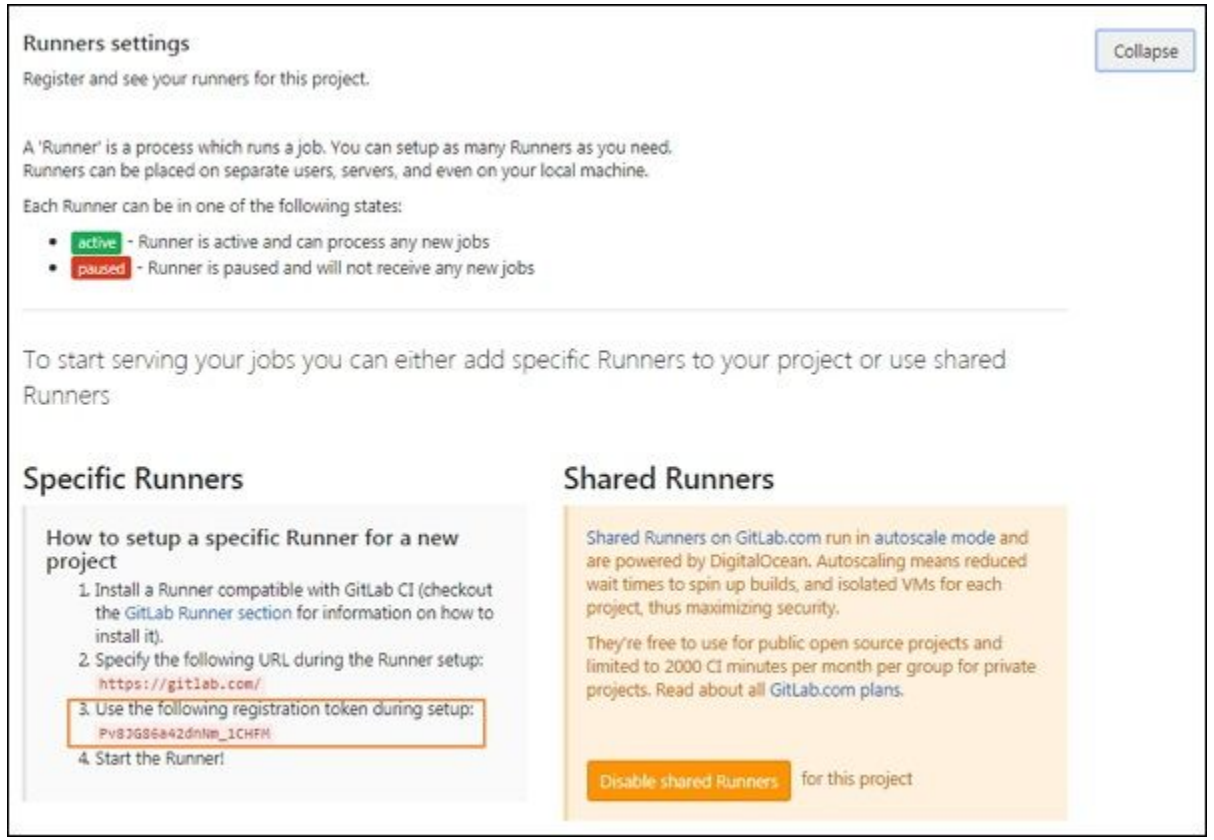
Installation of GitLab on Windows:

- Click on the *CI/CD* option under *Settings* tab and expand the *Runners Settings* option.



Installation of GitLab on Windows:

- Under *Runners Settings* section, you will get the token as shown in the image below –



Installation of GitLab on Windows:

- **Step 6 –** Enter the gitlab-ci description for the runner.

```
Please enter the gitlab-ci description for this runner:
[Admin-PC]: Hello GitLab Runner
```

- **Step 7 –** It will ask to enter the gitlab-ci tags for the runner.

```
Please enter the gitlab-ci tags for this runner (comma separated):
tag1, tag2
```

- **Step 8 –** You can lock the Runner to current project by setting it to true value.

```
Whether to lock the Runner to current project [true/false]:
[true]: true
```

- After completing above steps, you will get the successful message as 'Registering runner... succeeded'.

Installation of GitLab on Windows:

- **Step 9 –** Now enter the Runner executor for building the project.

```
Please enter the executor: parallels, shell, docker+machine, kubernetes, docker-
ssh+machine, docker, docker-ssh, ssh, virtualbox:
docker
```

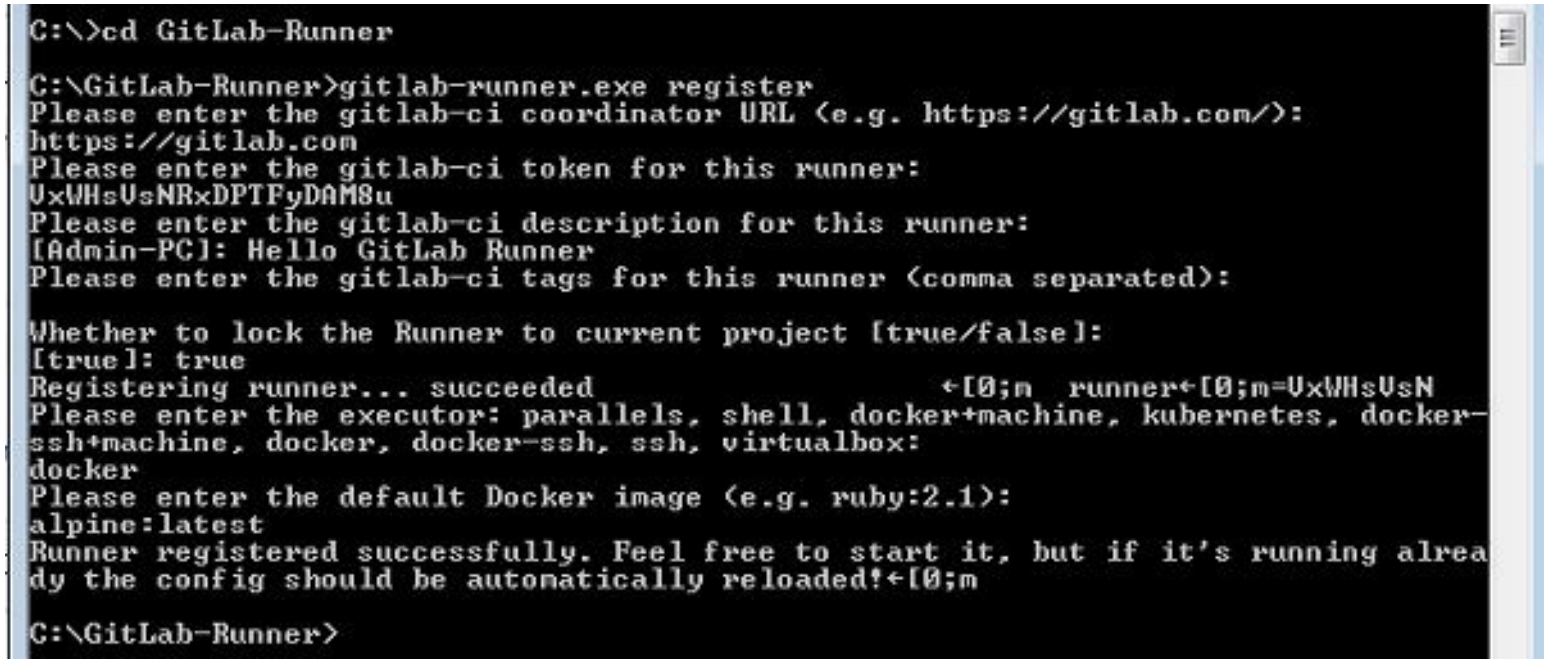
- We have used the selector as 'docker' which creates build environment and manages the dependencies easily for developing the project.

- **Step 10 –** Next it will ask for default image to be set for docker selector.

```
Please enter the default Docker image (e.g. ruby:2.1):
alpine:latest
```

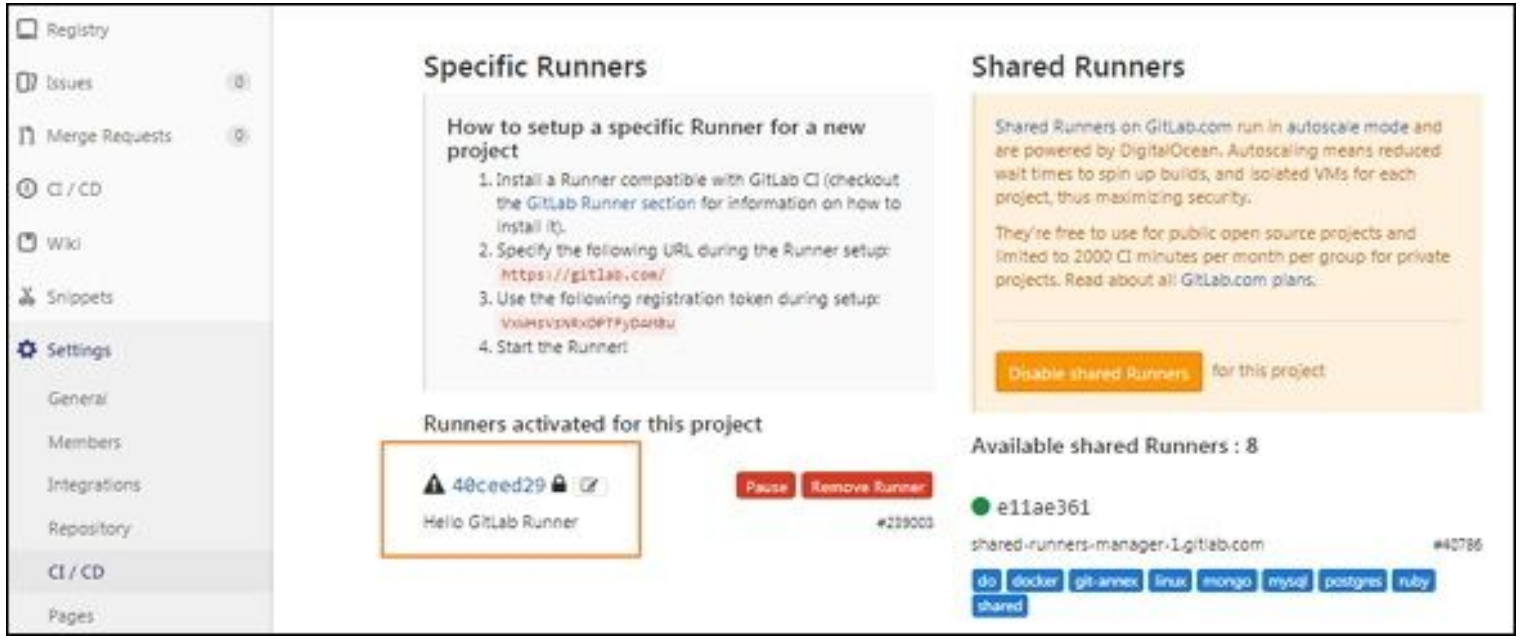
Installation of GitLab on Windows:

- **Step 11 –** After completing the above steps, it will display the message as 'Runner registered successfully'. The below image will describe the working flow of above commands –



Installation of GitLab on Windows:

- **Step 12 –** Now go to your project, click on the *CI/CD* option under *Settings* section and you will see the activated Runners for the project.



Installation of GitLab on Windows:

- You can see the GitLab Runner configuration in the *config.toml* file under the *GitLab-Runner* folder as shown below –

```
concurrent = 1
check_interval = 0
[[runners]]
  name = "Hello GitLab Runner"
  url = "https://gitlab.com"
  token = "40ceed29eec231fa9e306629cae4d7"
  executor = "docker"
  [runners.docker]
      tls_verify = false
      image = "alpine:latest"
      privileged = false
      disable_cache = false
      volumes = ["/cache"]
      shm_size = 0
  [runners.cache]
```

Git Commands

Following are the some basic Git commands can be used to work with Git –

- The version of the Git can be checked by using the below command –

```
$ git --version
```

- Add Git username and email address to identify the author while committing the information. Set the username by using the command as –

```
$ git config --global user.name "USERNAME"
```

- After entering user name, verify the entered user name with the below command –

```
$ git config --global user.name
```

- Next, set the email address with the below command –

```
$ git config --global user.email "email_address@example.com"
```

- You can verify the entered email address as –

```
$ git config --global user.email
```

Git Commands

- Use the below command to check the entered information –

```
$ git config --global --list
```

- You can pull the latest changes made to the master branch by using the below command –

```
$ git checkout master
```

- You can fetch the latest changes to the working directory with the below command –

```
$ git pull origin NAME-OF-BRANCH -u
Here, NAME-OF-BRANCH could be 'master' or any other existing branch.
```

- Create a new branch with the below command –

```
$ git checkout -b branch-name
```

- You can switch from one branch to other branch by using the command as –

```
$ git checkout branch-name
```

Git Commands

- Check the changes made to your files with the below command –

```
$ git status
```

- You will see the changes in red color and add the files to staging as –

```
$ git add file-name
```

- Or you can add all the files to staging as –

```
$ git add *
```

- Now send your changes to master branch with the below command –

```
$ git push origin branch-name
```

- Delete the all changes, except unstaged things by using the below command –

```
$ git checkout .
```

Git Commands

- You can delete the all changes along with untracked files by using the command as –

```
$ git clean -f
```

- To merge the different branch with the master branch, use the below command –

```
$git checkout branch-name
$ git merge master
```

- You can also merge the master branch with the created branch, by using the below command –

```
$git checkout master
$ git merge branch-name
```

GitLab - SSH Key Setup

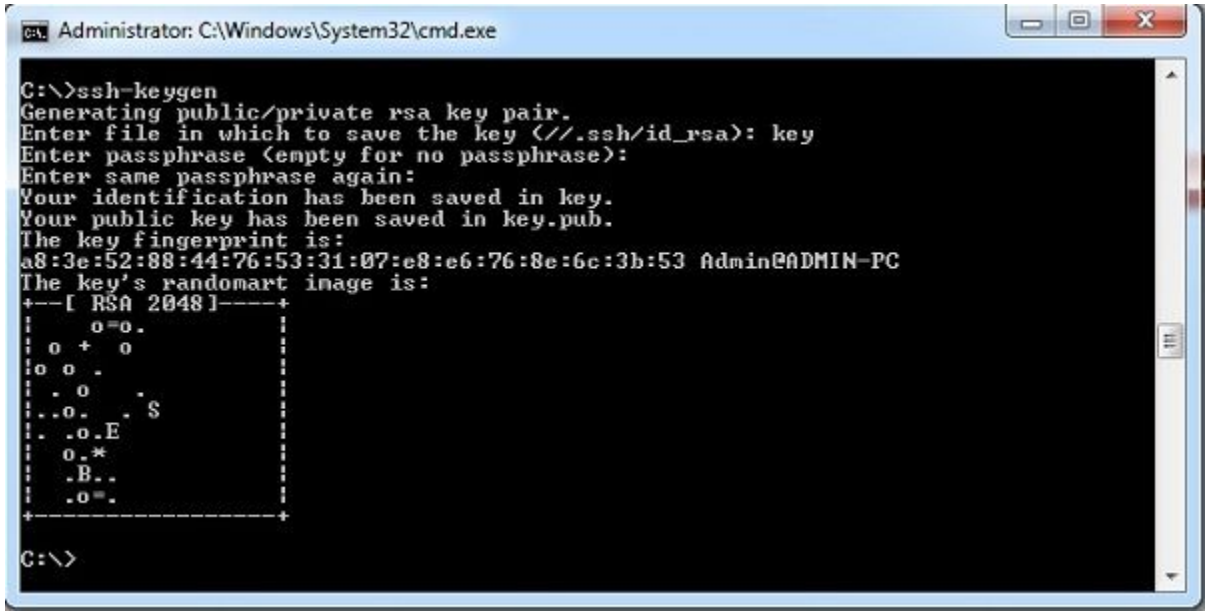
- The SSH stands for *Secure Shell* or *Secure Socket Shell* used for managing the networks, operating systems and configurations and also authenticates to the GitLab server without using username and password each time.
- You can set the SSH keys to provide a reliable connection between the computer and GitLab.
- Before generating ssh keygen, you need to have Git installed in your system.

Creating SSH Key

- **Step 1 –** To create SSH key, open the command prompt and enter the command as shown below –

  ```
  C:\-ssh-keygen
  ```

- It will prompt for 'Enter file in which to save the key (//.ssh/id_rsa):', just type file name and press enter. Next a prompt to enter password shows 'Enter passphrase (empty for no passphrase):'. Enter some password and press enter. You will see the generated SSH key as shown in the below image –

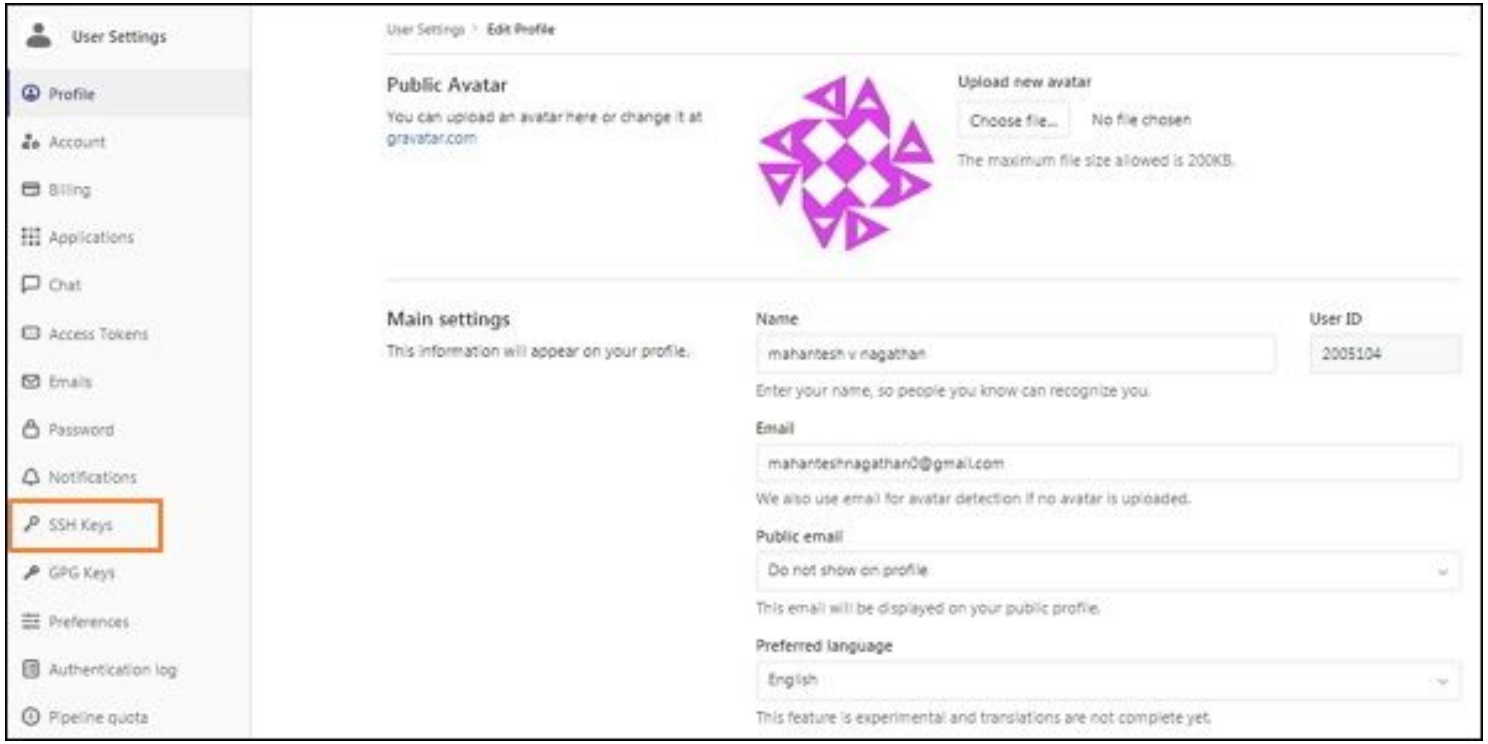


Creating SSH Key

- **Step 2 –** Now login to your GitLab account and click on the *Settings* option.

Creating SSH Key

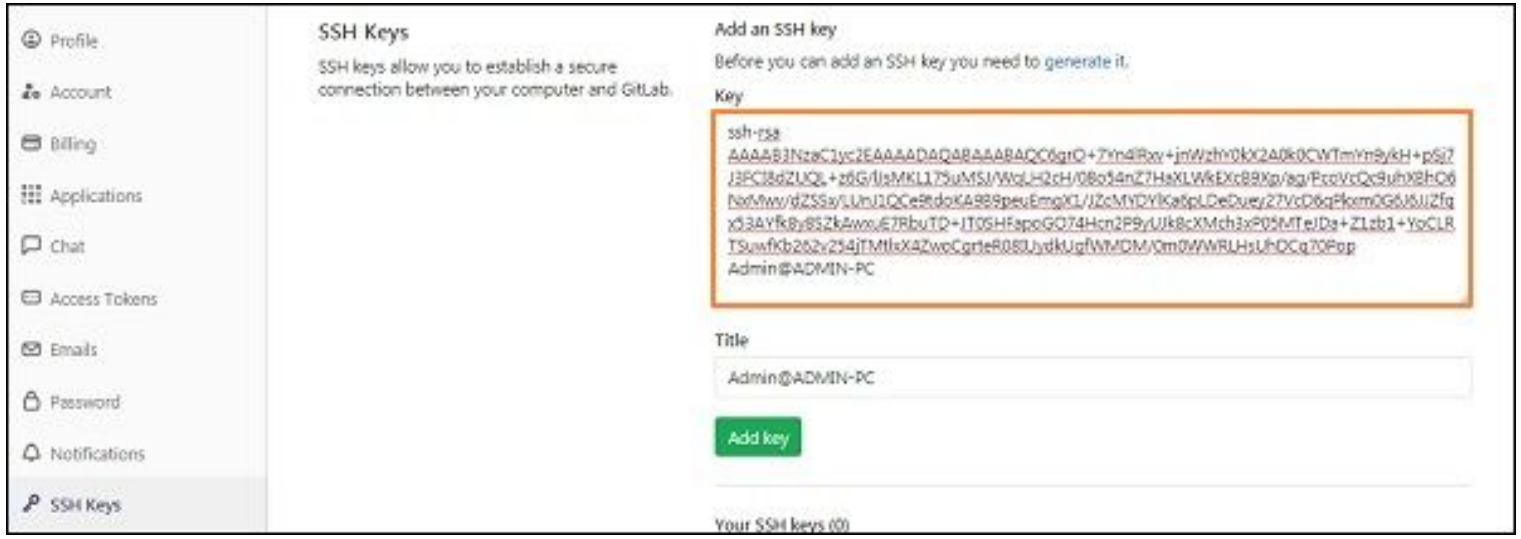
- **Step 3 −** To create SSH key, click on the SSH keys tab at left side of the menu.



Creating SSH Key

- **Step 4 −** Now go to C drive, you will see the file with *.pub* extension which was generated in the first step.

- **Step 5 −** Next open the *key.pub* file, copy the SSH key and paste it in the highlighted *Key* box as shown in the below image −



Creating SSH Key

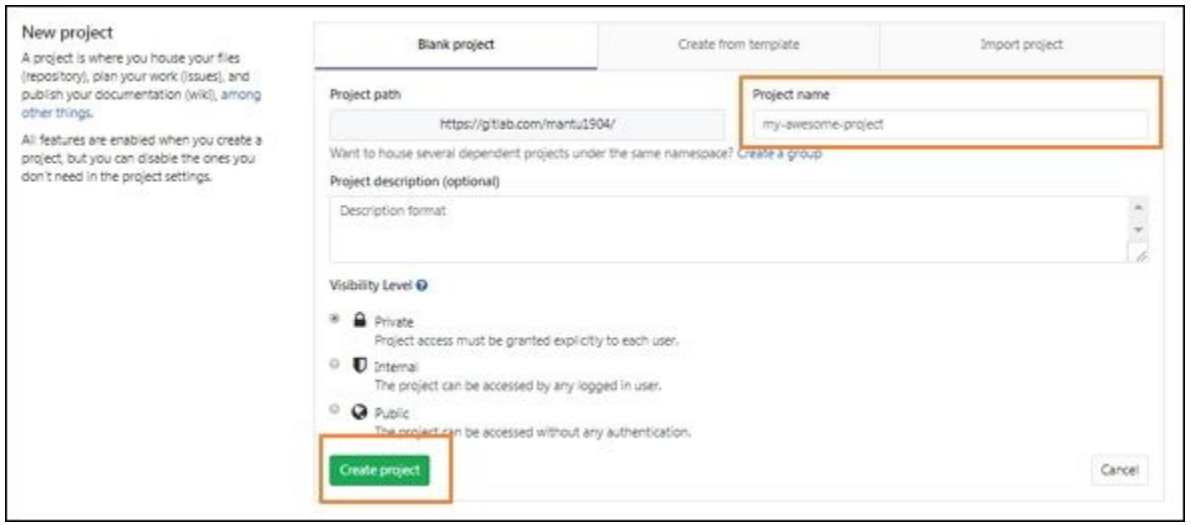
- **Step 6 −** Click on the *Add Key* button, to add SSH key to your GitLab. You will see the fingerprint (it is a short version of SSH key), title and created date as shown in the image below −

GitLab - Create Project

- **Step 1 −** To create new project, login to your GitLab account and click on the *New project* button in the dashboard −

GitLab - Create Project

- **Step 2** – It will open the New project screen as shown below in the image –

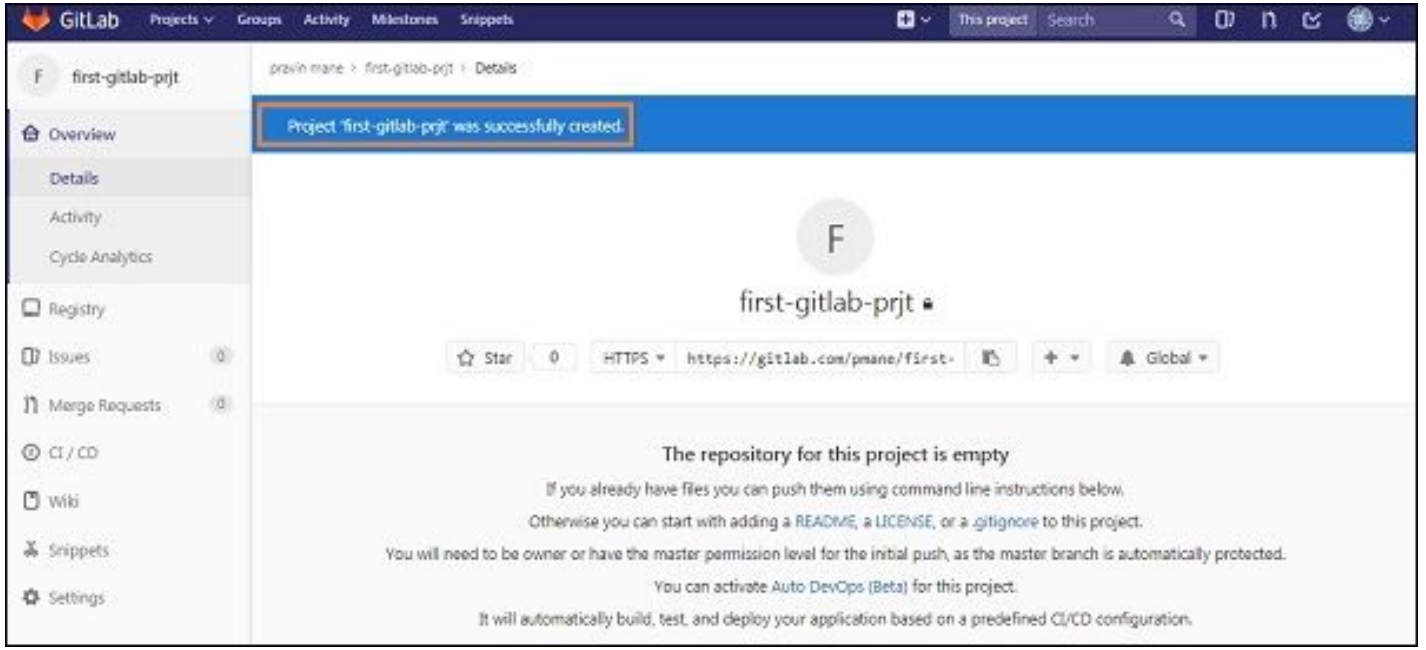


- Enter the project name, description for the project, visibility level (accessing the project's visibility in publicly or internally) and click on the *Create project* button.

G2 | Academy

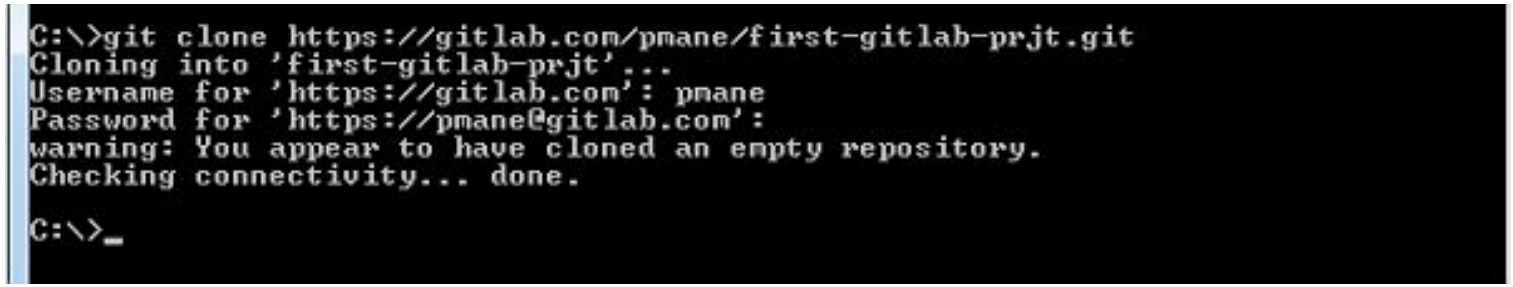
GitLab - Create Project

- **Step 3 –** Next it will create a new project (here given the project name as first-gitlab-prjt) with successful message as shown below –



GitLab - Create Project

- Push the Repository to Project
- **Step 4 –** You can clone the repository to your local system by using the *git-clone* command –



- The clone command makes a copy of repository into a new directory called *first-gitlab-prjt*.

GitLab - Create Project

- **Step 5 –** Now go to your newly created directory and type the below command –

  ```
  C:\>cd first-gitlab-prjt
  C:\first-gitlab-prjt>touch README.md
  ```

  The above command creates a *README.md* file in which you can put the information about your folder.

- **Step 6 –** Add the *README.md* file to your created directory by using the below command –

  ```
  C:\first-gitlab-prjt>git add README.md
  ```

- **Step 7 –** Now store the changes to the repository along with the log message as shown below –

  ```
  C:\first-gitlab-prjt>git commit -m "add README"
  ```

  The flag *-m* is used for adding a message on the commit.

- **Step 8 –** Push the commits to remote repository which are made on the local branch –

  ```
  C:\first-gitlab-prjt>git push -u origin master
  ```

GitLab - Create Project

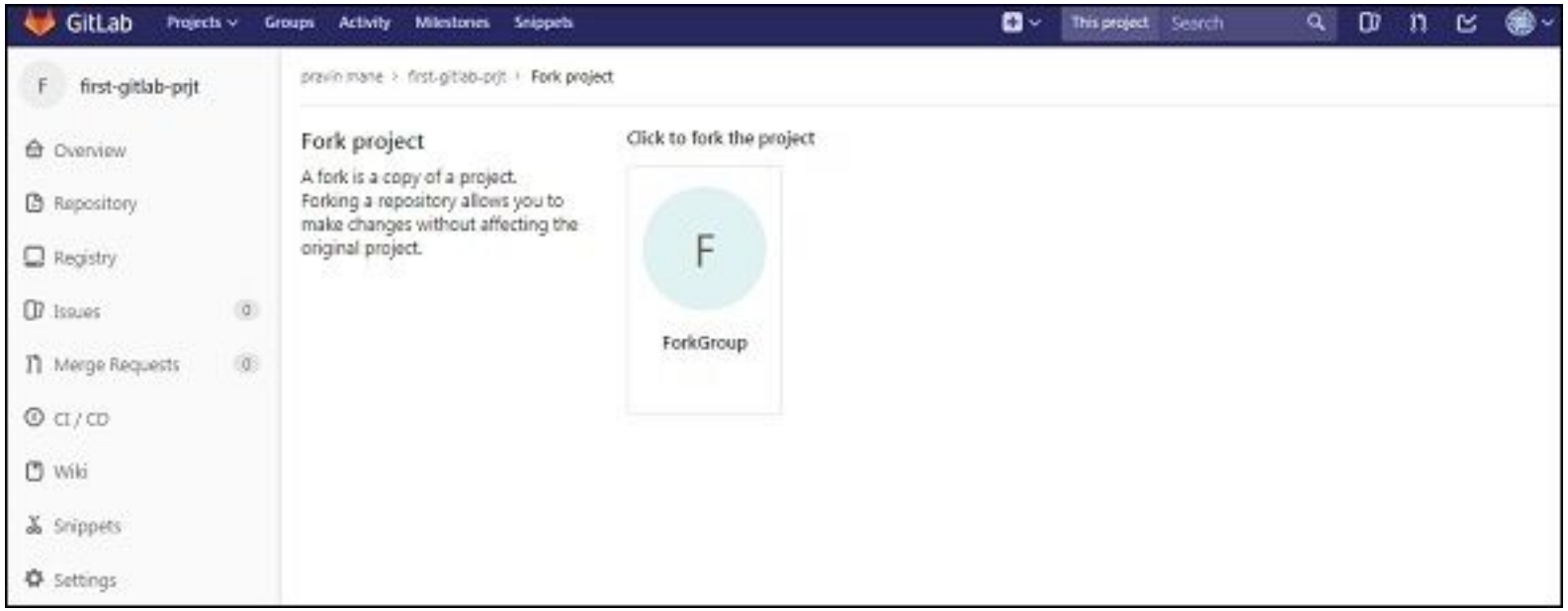
- The below image depicts the usage of above commands in pushing the commits to remote repository –

GitLab - Fork a Project

- Fork is a duplicate of your original repository in which you can make the changes without affecting the original project.
- **Step 1 –** To fork a project, click on the *Fork* button as shown below –

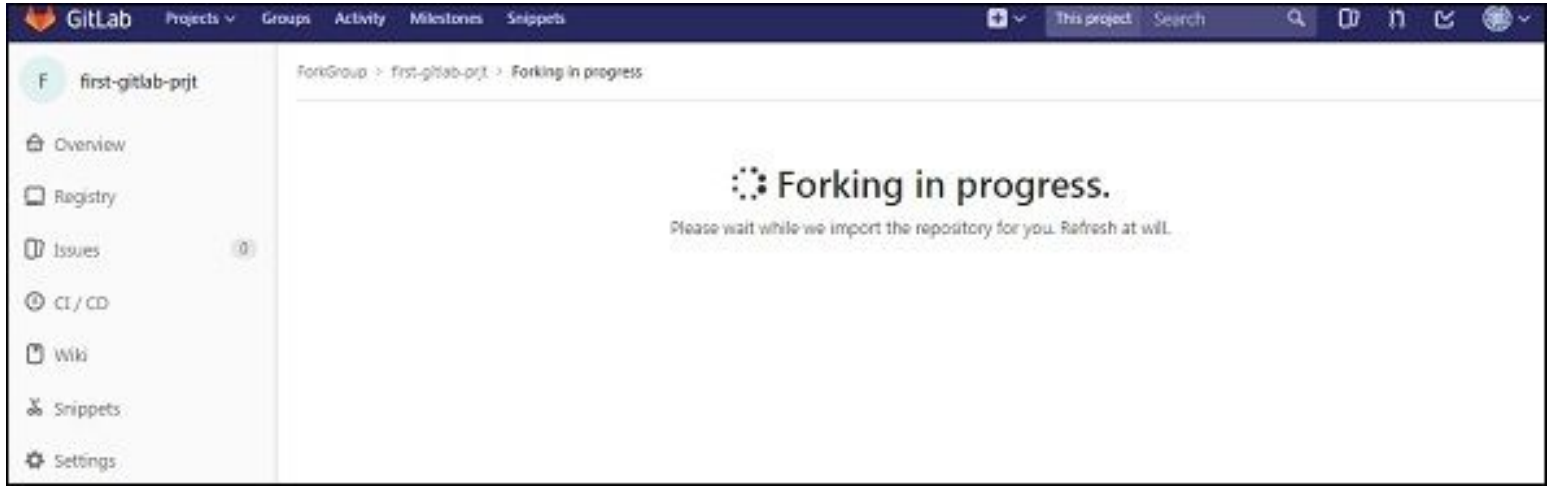
GitLab - Fork a Project

- **Step 2 –** After forking the project, you need to add the forked project to a fork group by clicking on it –



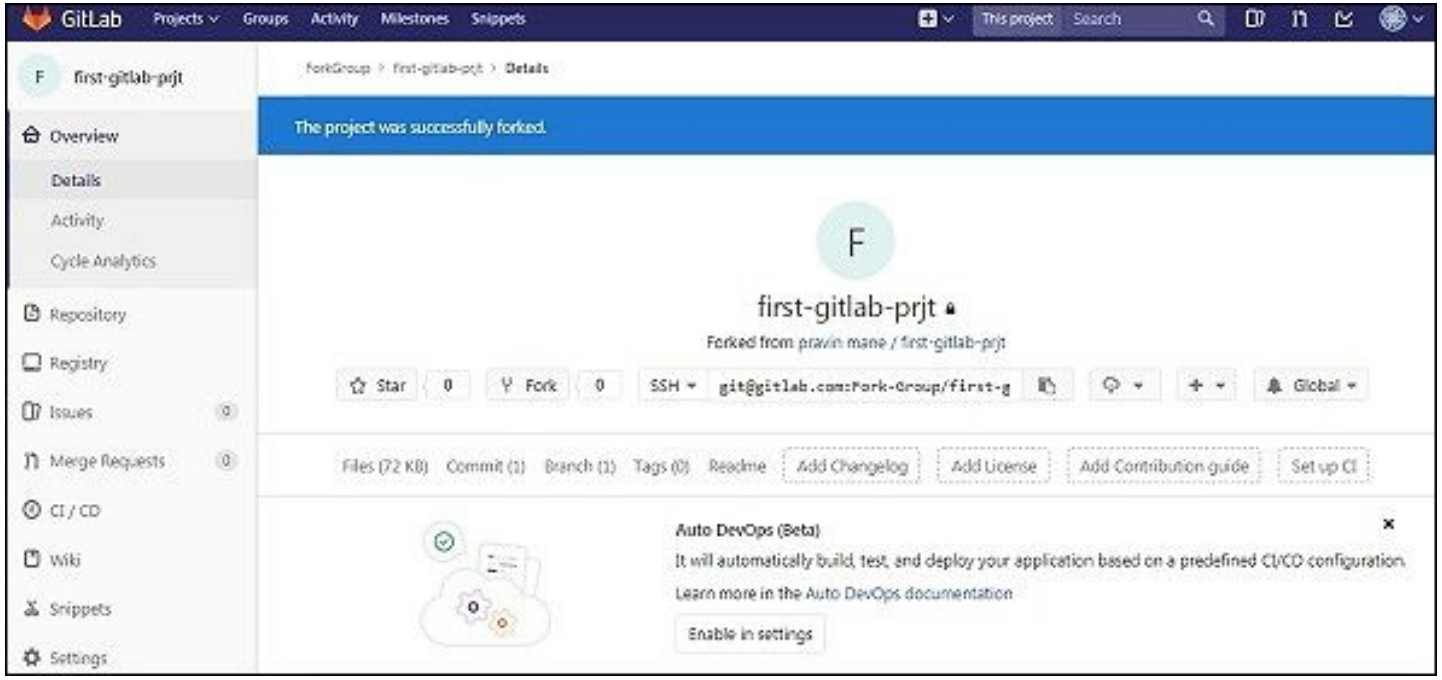
GitLab - Fork a Project

- **Step 3 −** Next it will start processing of forking a project for sometime as shown below −



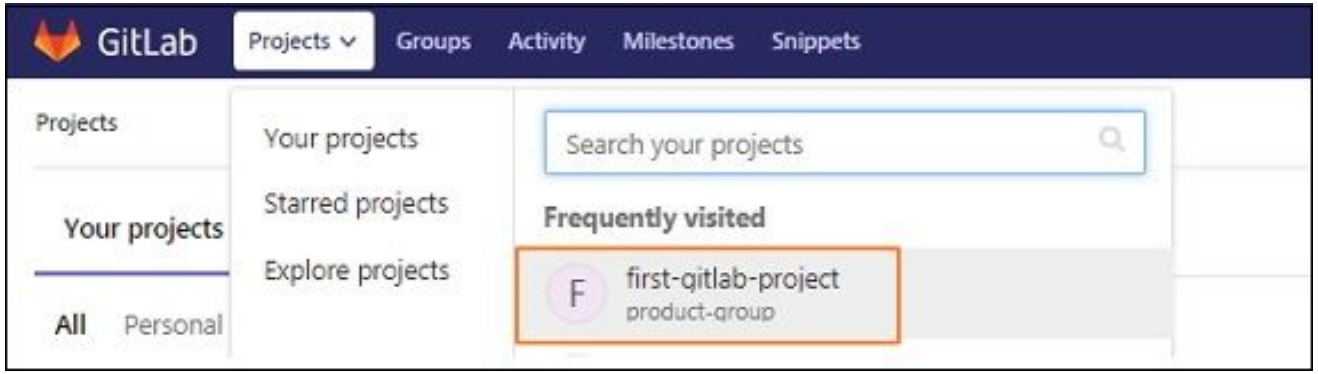
GitLab - Fork a Project

- **Step 4 –** It will display the success message after completion of forking the project process –



GitLab - Create a Branch

- Branch is independent line and part of the development process. The creation of branch involves following steps.
- **Step 1 –** Login to your GitLab account and go to your project under *Projects* section.



GitLab - Create a Branch

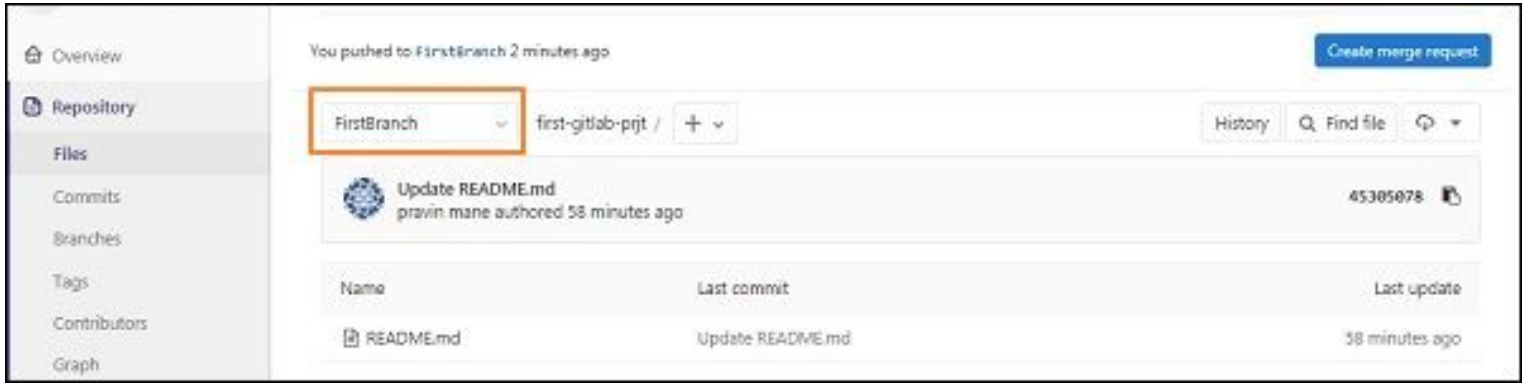
- **Step 2 –** To create a branch, click on the *Branches* option under the *Repository* section and click on the *New branch* button.

GitLab - Create a Branch

- **Step 3 –** In the *New branch* screen, enter the name for branch and click on the *Create branch* button.

GitLab - Create a Branch

- **Step 4 –** After creating branch, you will get a below screen along with the created branch.



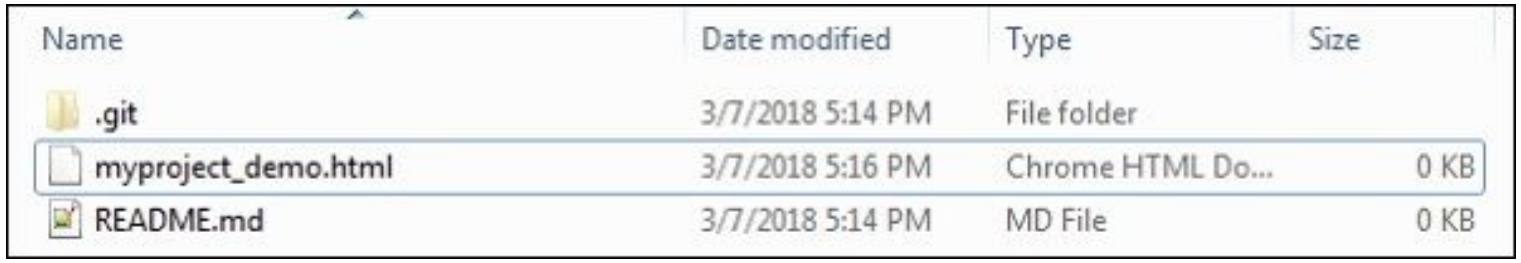
GitLab - Add a File

- Creating a file using Command Line Interface
- **Step 1 −** To create a file by using command line interface, type the below command in your project directory −

```
C:\first-gitlab-prjt>touch myproject_demo.html

C:\first-gitlab-prjt>
```

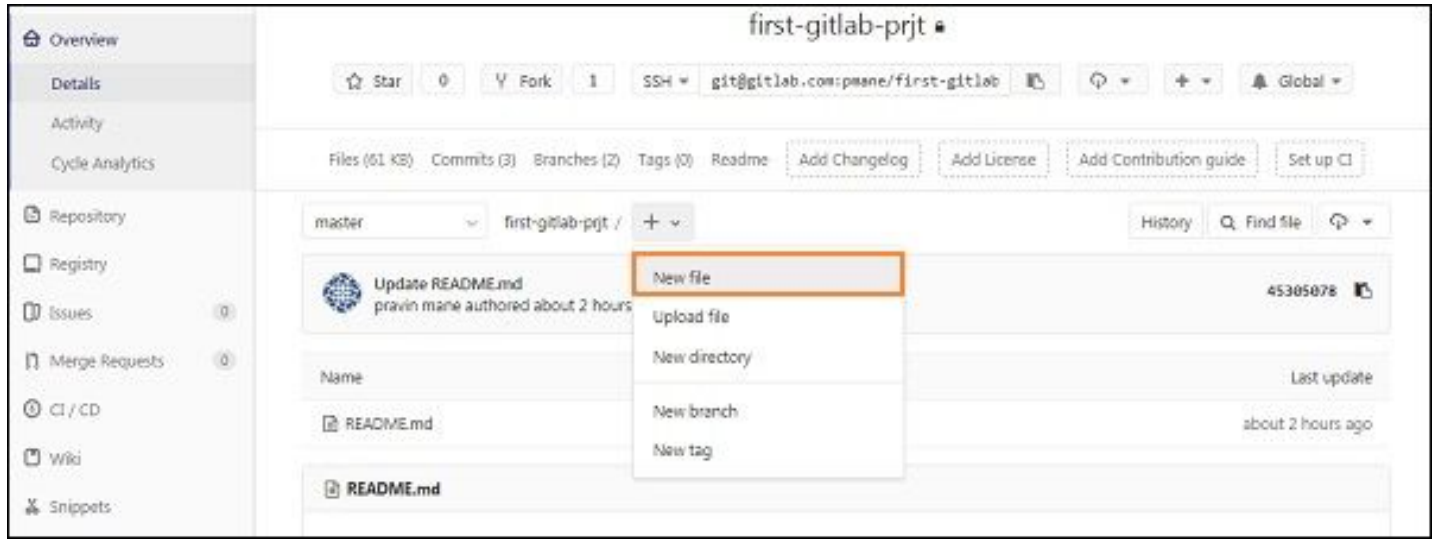
- **Step 2 −** Now go to your project directory and you will see the created file −



GitLab - Add a File

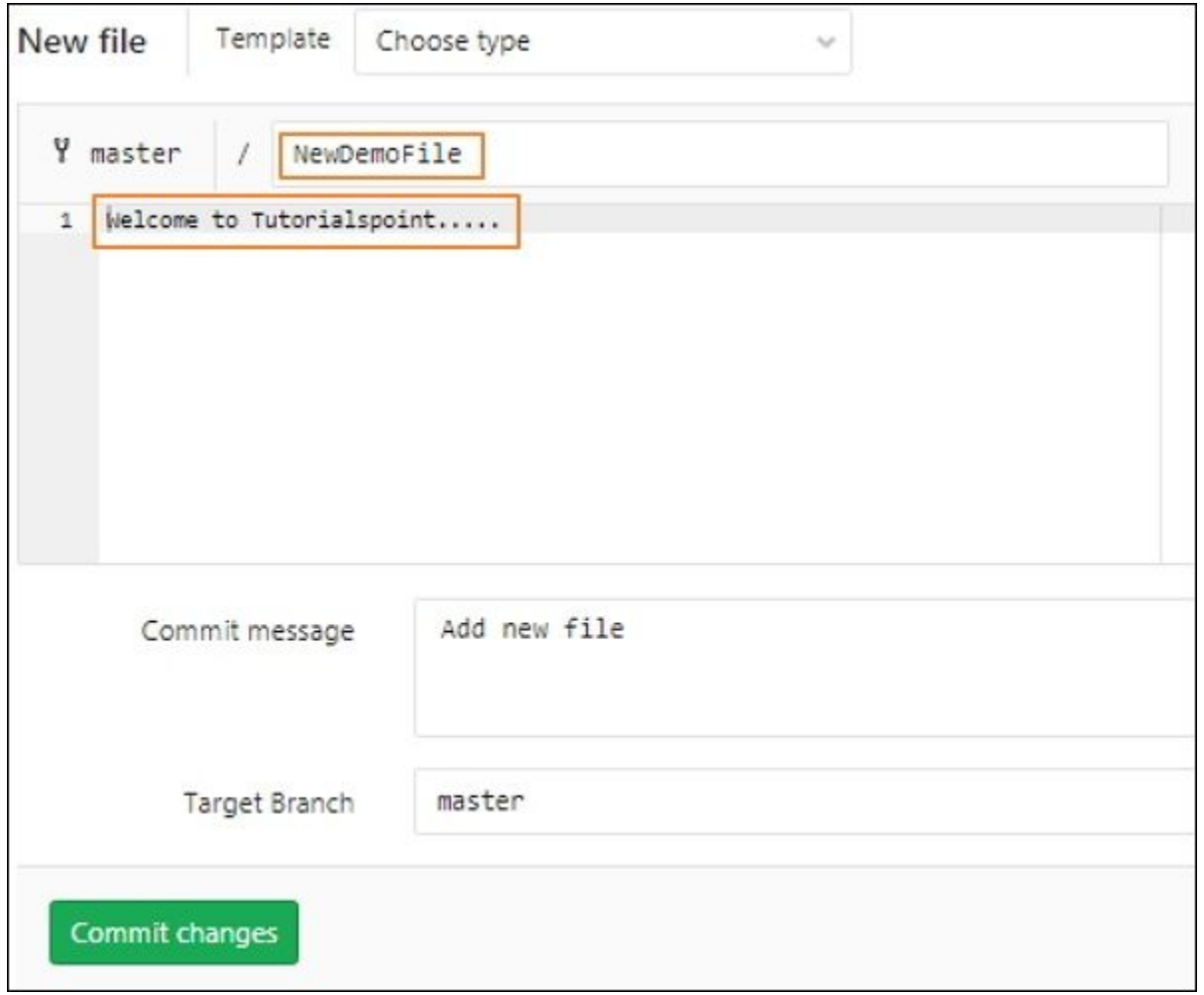
Creating a file using Web Interface

- **Step 1 −** You can create a new file, by clicking on the '+' button which is at the right side of the branch selector in the dashboard −



GitLab - Add a File

- **Step 2** – Enter the file name, add some content in the editor section and click on the *Commit changes* button to create the file.



GitLab - Add a File

- **Step 3** − Now you will get a successful message after creating the file as shown below −

GitLab - Rebase Operation

Rebase is a way of merging *master* to your branch when you are working with long running branch.

- **Step 1 –** Go to your project directory and create a new branch with the name *rebase-example* by using the *git checkout* command –

The flag *-b* indicates new branch name.

GitLab - Rebase Operation

- **Step 2 −** Now, create a new file and add some content to that file as shown below −

```
C:\first-gitlab-prjt>echo "Welcome to Tutorialspoint" >> rebase_file.md

C:\first-gitlab-prjt>_
```

The content 'Welcome to Tutorialspoint' will be added to the *rebase_file.md* file.

- **Step 3 −** Add the new file to working directory and store the changes to the repository along with the message (by using the *git commit* command) as shown below −

```
C:\first-gitlab-prjt> git add .

C:\first-gitlab-prjt>git commit -m "Rebase file added"
[rebase-example f4c63d0] Rebase file added
 create mode 100644 rebase_file.md
```

The flag *-m* is used for adding a message on the commit.

GitLab - Rebase Operation

- **Step 4 −** Now, switch to the 'master' branch. You can fetch the remote branch(*master* is a branch name) by using the *git checkout* command −

```
C:\first-gitlab-prjt>git checkout master
Switched to branch 'master'
Your branch and 'origin/master' have diverged,
and have 1 and 3 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

C:\first-gitlab-prjt>
```

- **Step 5 −** Next, create an another new file, add some content to that file and commit it in the *master* branch.

```
C:\first-gitlab-prjt>echo "text in main branch" >> README.md

C:\first-gitlab-prjt>git add .

C:\first-gitlab-prjt>git commit -m "Commit in master"
[master 7dd6a44] Commit in master
 1 file changed, 1 insertion(+)

C:\first-gitlab-prjt>_
```

GitLab - Rebase Operation

- **Step 6 –** Switch to the *rebase-branch* to have the commit of *master* branch.

```
C:\first-gitlab-prjt>git checkout rebase-branch
Switched to branch 'rebase-branch'

C:\first-gitlab-prjt>
```

- **Step 7 –** Now, you can combine the commit of *master* branch to *rebase-branch* by using the *git rebase* command –

```
C:\first-gitlab-prjt>git rebase master
First, rewinding head to replay your work on top of it...
Applying: Another commit

C:\first-gitlab-prjt>
```

GitLab - Squashing Commits

Squashing is a way of combining all commits into one when you are obtaining a merge request.

- **Step 1 −** Go to your project directory and check out a new branch with the name *squash-chapter* by using the *git checkout* command **−**

```
C:\first-gitlab-prjt>git checkout -b squash-chapter
Switched to a new branch 'squash-chapter'
```

The flag -*b* indicates new branch name.

GitLab - Squashing Commits

- **Step 2 –** Now, create a new file with two commits, add that file to working directory and store the changes to the repository along with the commit messages as shown below –

```
C:\first-gitlab-prjt>echo "message1" >> README.md

C:\first-gitlab-prjt>git add .

C:\first-gitlab-prjt>git commit -a -m "message1 commited"
[squash-chapter 771bb9a] message1 commited
 1 file changed, 1 insertion(+)

C:\first-gitlab-prjt>
```

```
C:\first-gitlab-prjt>echo "message2" >> README.md

C:\first-gitlab-prjt>git add .

C:\first-gitlab-prjt>git commit -a -m "message2 commited"
[squash-chapter 6b67004] message2 commited
 1 file changed, 1 insertion(+)

C:\first-gitlab-prjt>_
```

GitLab - Squashing Commits

- **Step 3** – Now, squash the above two commits into one commit by using the below command –

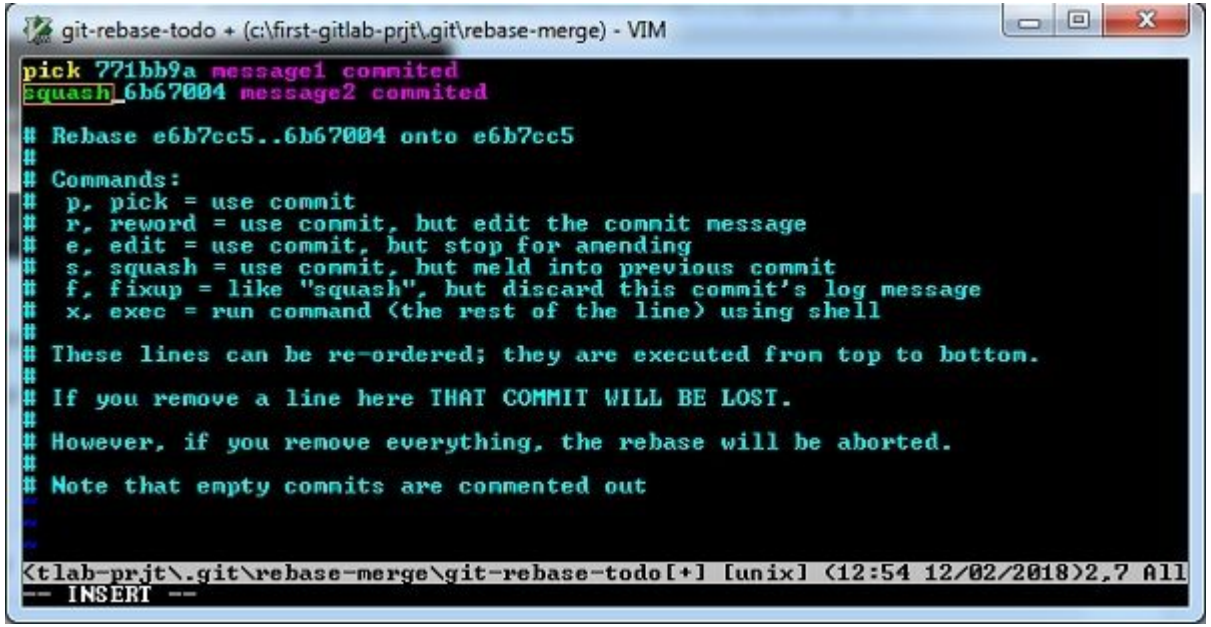
```
$ git rebase -i HEAD~2
```

Here, *git rebase* command is used to integrate changes from one branch to another and *HEAD~2* specifies last two squashed commits and if you want to squash four commits, then you need to write as *HEAD~4*.

One more important point is, you need atleast two commits to complete the squash operation.

GitLab - Squashing Commits

- **Step 4 –** After entering the above command, it will open the below editor in which you have to change the *pick* word to *squash* word in the second line (you need to squash this commit).



Now press the *Esc* key, then colon(:) and type *wq* to save and exit from the screen.

GitLab - Squashing Commits

- **Step 5** – Now push the branch to remote repository as shown below –

```
C:\first-gitlab-prjt>git push origin squash-chapter
Username for 'https://gitlab.com': pmane
Password for 'https://pmane@gitlab.com':
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 631 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote:
remote: To create a merge request for squash-chapter, visit:
remote:   https://gitlab.com/pmane/first-gitlab-prjt/merge_requests/new?nerge_re
quest%5Bsource_branch%5D=squash-chapter
remote:
To https://gitlab.com/pmane/first-gitlab-prjt.git
 * [new branch]      squash-chapter -> squash-chapter

C:\first-gitlab-prjt>_
```

MAVEN

What is Maven?

- Maven is a project management and comprehension tool that provides developers a complete build lifecycle framework. Development team can automate the project's build infrastructure in almost no time as Maven uses a standard directory layout and a default build lifecycle.
- In case of multiple development teams environment, Maven can set-up the way to work as per standards in a very short time. As most of the project setups are simple and reusable, Maven makes life of developer easy while creating reports, checks, build and testing automation setups.

What is Maven?

- Maven provides developers ways to manage the following −
  - Builds
  - Documentation
  - Reporting
  - Dependencies
  - SCMs
  - Releases
  - Distribution
  - Mailing list
- To summarize, Maven simplifies and standardizes the project build process. It handles compilation, distribution, documentation, team collaboration and other tasks seamlessly. Maven increases reusability and takes care of most of the build related tasks.

Objective

- The primary goal of Maven is to provide developer with the following −
  - A comprehensive model for projects, which is reusable, maintainable, and easier to comprehend.
  - Plugins or tools that interact with this declarative model.
- Maven project structure and contents are declared in an xml file, pom.xml, referred as Project Object Model (POM), which is the fundamental unit of the entire Maven system. In later chapters, we will explain POM in detail.

Convention over Configuration

- Maven uses **Convention** over **Configuration**, which means developers are not required to create build process themselves.
- Developers do not have to mention each and every configuration detail. Maven provides sensible default behavior for projects. When a Maven project is created, Maven creates default project structure. Developer is only required to place files accordingly and he/she need not to define any configuration in pom.xml.
- As an example, following table shows the default values for project source code files, resource files and other configurations. Assuming, **${basedir}** denotes the project location –

Convention over Configuration

Item	Default
source code	${basedir}/src/main/java
Resources	${basedir}/src/main/resources
Tests	${basedir}/src/test
Complied byte code	${basedir}/target
distributable JAR	${basedir}/target/classes

Convention over Configuration

- In order to build the project, Maven provides developers with options to mention life-cycle goals and project dependencies (that rely on Maven plugin capabilities and on its default conventions). Much of the project management and build related tasks are maintained by Maven plugins.
- Developers can build any given Maven project without the need to understand how the individual plugins work. We will discuss Maven Plugins in detail in the later chapters.

Features of Maven

- Simple project setup that follows best practices.
- Consistent usage across all projects.
- Dependency management including automatic updating.
- A large and growing repository of libraries.
- Extensible, with the ability to easily write plugins in Java or scripting languages.
- Instant access to new features with little or no extra configuration.
- **Model-based builds** − Maven is able to build any number of projects into predefined output types such as jar, war, metadata.
- **Coherent site of project information** − Using the same metadata as per the build process, maven is able to generate a website and a PDF including complete documentation.
- **Release management and distribution publication** − Without additional configuration, maven will integrate with your source control system such as CVS and manages the release of a project.
- **Backward Compatibility** − You can easily port the multiple modules of a project into Maven 3 from older versions of Maven. It can support the older versions also.
- **Automatic parent versioning** − No need to specify the parent in the sub module for maintenance.
- **Parallel builds** − It analyzes the project dependency graph and enables you to build schedule modules in parallel. Using this, you can achieve the performance improvements of 20-50%.
- **Better Error and Integrity Reporting** − Maven improved error reporting, and it provides you with a link to the Maven wiki page where you will get full description of the error.

Environment Setup

- Maven is a Java based tool, so the very first requirement is to have JDK installed on your machine.
- System Requirement

JDK	1.7 or above.
Memory	No minimum requirement.
Disk Space	No minimum requirement.
Operating System	No minimum requirement.

Step 1 - Verify Java Installation on your Machine

- Open console and execute the following **java** command.

OS	Task	Command
Windows	Open Command Console	c:\> java -version
Linux	Open Command Terminal	$ java -version
Mac	Open Terminal	machine:~ joseph$ java -version

Step 1 - Verify Java Installation on your Machine

- Let's verify the output for all the operating systems –

OS	Output
Windows	java version "1.7.0_60" Java(TM) SE Runtime Environment (build 1.7.0_60-b19) Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
Linux	java version "1.7.0_60" Java(TM) SE Runtime Environment (build 1.7.0_60-b19) Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)
Mac	java version "1.7.0_60" Java(TM) SE Runtime Environment (build 1.7.0_60-b19) Java HotSpot(TM) 64-Bit Server VM (build 24.60-b09, mixed mode)

Step 2 - Set JAVA Environment

- Set the **JAVA_HOME** environment variable to point to the base directory location where Java is installed on your machine. For example −

OS	Output
Windows	Set the environment variable JAVA_HOME to C:\Program Files\Java\jdk1.7.0_60
Linux	export JAVA_HOME=/usr/local/java-current
Mac	export JAVA_HOME=/Library/Java/Home

Step 2 - Set JAVA Environment

- Append Java compiler location to System Path.

OS	Output
Windows	Append the string ";C:\Program Files\Java\jdk1.7.0.60\bin" to the end of the system variable, Path.
Linux	export PATH=$PATH:$JAVA_HOME/bin/
Mac	not required

- Verify Java Installation using **java -version** command as explained above.

Step 3 - Download Maven Archive

- Download Maven X.X.X from https://maven.apache.org/download.cgi

OS	Archive name
Windows	apache-maven-3.3.1-bin.zip
Linux	apache-maven-3.3.1-bin.tar.gz
Mac	apache-maven-3.3.1-bin.tar.gz

Step 4 - Extract the Maven Archive

- Extract the archive, to the directory you wish to install Maven 3.3.1. The subdirectory apache-maven-3.3.1 will be created from the archive.

OS	Location (can be different based on your installation)
Windows	C:\Program Files\Apache Software Foundation\apache-maven-3.3.1
Linux	/usr/local/apache-maven
Mac	/usr/local/apache-maven

Step 5 - Set Maven Environment Variables

- Add M2_HOME, M2, MAVEN_OPTS to environment variables.

OS	Output
Windows	Set the environment variables using system properties. M2_HOME=C:\Program Files\Apache Software Foundation\apache-maven-3.3.1 M2=%M2_HOME%\bin MAVEN_OPTS=-Xms256m -Xmx512m
Linux	Open command terminal and set environment variables. export M2_HOME=/usr/local/apache-maven/apache-maven-3.3.1 export M2=$M2_HOME/bin export MAVEN_OPTS=-Xms256m -Xmx512m
Mac	Open command terminal and set environment variables. export M2_HOME=/usr/local/apache-maven/apache-maven-3.3.1 export M2=$M2_HOME/bin export MAVEN_OPTS=-Xms256m -Xmx512m

Step 6 - Add Maven bin Directory Location to System Path

- Now append M2 variable to System Path.

OS	Output
Windows	Append the string ;%M2% to the end of the system variable, Path.
Linux	export PATH=$M2:$PATH
Mac	export PATH=$M2:$PATH

Step 7 - Verify Maven Installation

- Now open console and execute the following **mvn** command.

OS	Task	Command
Windows	Open Command Console	c:\> mvn --version
Linux	Open Command Terminal	$ mvn --version
Mac	Open Terminal	machine:~ joseph$ mvn --version

Step 7 - Verify Maven Installation

- Finally, verify the output of the above commands, which should be as follows –

OS	Output
Windows	Apache Maven 3.3.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.7.0_60 Java home: C:\Program Files\Java\jdk1.7.0_60 \jre
Linux	Apache Maven 3.3.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.7.0_60 Java home: C:\Program Files\Java\jdk1.7.0_60 \jre
Mac	Apache Maven 3.3.1 (r801777; 2009-08-07 00:46:01+0530) Java version: 1.7.0_60 Java home: C:\Program Files\Java\jdk1.7.0_60 \jre

POM

- POM stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml.
- The POM contains information about the project and various configuration detail used by Maven to build the project(s).
- POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal.

POM

- Some of the configuration that can be specified in the POM are following −
  - project dependencies
  - plugins
  - goals
  - build profiles
  - project version
  - developers
  - mailing list
- Before creating a POM, we should first decide the project **group** (groupId), its **name** (artifactId) and its version as these attributes help in uniquely identifying the project in repository.

POM Example

```
<project xmlns = "http://maven.apache.org/POM/4.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
   <modelVersion>4.0.0</modelVersion>

   <groupId>com.companyname.project-group</groupId>
   <artifactId>project</artifactId>
   <version>1.0</version>
</project>
```

POM Example

- It should be noted that there should be a single POM file for each project.
  - All POM files require the **project** element and three mandatory fields: **groupId, artifactId, version**.
  - Projects notation in repository is **groupId:artifactId:version**.
  - Minimal requirements for a POM –

POM Example

Sr.No.	Node & Description
1	**Project root** This is project root tag. You need to specify the basic schema settings such as apache schema and w3.org specification.
2	**Model version** Model version should be 4.0.0.
3	**groupId** This is an Id of project's group. This is generally unique amongst an organization or a project. For example, a banking group com.company.bank has all bank related projects.
4	**artifactId** This is an Id of the project. This is generally name of the project. For example, consumer-banking. Along with the groupId, the artifactId defines the artifact's location within the repository.
5	**version** This is the version of the project. Along with the groupId, It is used within an artifact's repository to separate versions from each other. For example – **com.company.bank:consumer-banking:1.0** **com.company.bank:consumer-banking:1.1.**

Super POM

- The Super POM is Maven's default POM. All POMs inherit from a parent or default (despite explicitly defined or not). This base POM is known as the **Super POM**, and contains values inherited by default.
- Maven use the effective POM (configuration from super pom plus project configuration) to execute relevant goal. It helps developers to specify minimum configuration detail in his/her pom.xml. Although configurations can be overridden easily.
- An easy way to look at the default configurations of the super POM is by running the following command: **mvn help:effective-pom**

Super POM

- Create a pom.xml in any directory on your computer.Use the content of above mentioned example pom.
- In example below, We've created a pom.xml in C:\MVN\project folder.
- Now open command console, go the folder containing pom.xml and execute the following **mvn**command.

```
C:\MVN\project>mvn help:effective-pom
```

Super POM

- Maven will start processing and display the effective-pom.

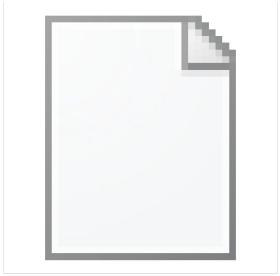
```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'help'.
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.project-group:project-name:jar:1.0
[INFO]    task-segment: [help:effective-pom] (aggregator-style)
[INFO] ------------------------------------------------------------------------
[INFO] [help:effective-pom {execution: default-cli}]
[INFO]

.....

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Thu Jul 05 11:41:51 IST 2012
[INFO] Final Memory: 6M/15M
[INFO] ------------------------------------------------------------------------
```

Super POM

- Effective POM displayed as result in console, after inheritance, interpolation, and profiles are applied.



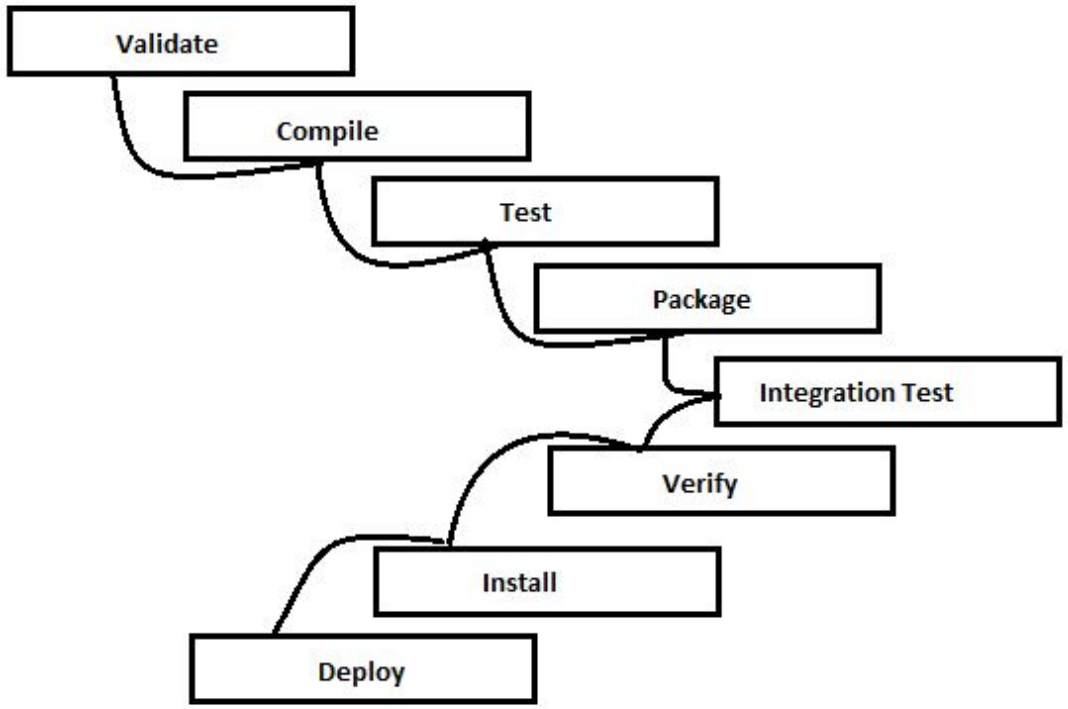
pom.xml

Super POM

- In above pom.xml, you can see the default project source folders structure, output directory, plug-ins required, repositories, reporting directory, which Maven will be using while executing the desired goals.
- Maven pom.xml is also not required to be written manually. Maven provides numerous archetype plugins to create projects, which in order, create the project structure and pom.xml

What is Build Lifecycle?

- A Build Lifecycle is a well-defined sequence of phases, which define the order in which the goals are to be executed. Here phase represents a stage in life cycle. As an example, a typical **Maven Build Lifecycle** consists of the following sequence of phases.



What is Build Lifecycle?

Phase	Handles	Description
prepare-resources	resource copying	Resource copying can be customized in this phase.
validate	Validating the information	Validates if the project is correct and if all necessary information is available.
compile	compilation	Source code compilation is done in this phase.
Test	Testing	Tests the compiled source code suitable for testing framework.
package	packaging	This phase creates the JAR/WAR package as mentioned in the packaging in POM.xml.
install	installation	This phase installs the package in local/remote maven repository.
Deploy	Deploying	Copies the final package to the remote repository.

What is Build Lifecycle?

- There are always **pre** and **post** phases to register **goals**, which must run prior to, or after a particular phase.
- When Maven starts building a project, it steps through a defined sequence of phases and executes goals, which are registered with each phase.
- Maven has the following three standard lifecycles –
  - clean
  - default(or build)
  - site

What is Build Lifecycle?

- A **goal** represents a specific task which contributes to the building and managing of a project. It may be bound to zero or more build phases. A goal not bound to any build phase could be executed outside of the build lifecycle by direct invocation.
- The order of execution depends on the order in which the goal(s) and the build phase(s) are invoked. For example, consider the command below.
  The **clean** and **package** arguments are build phases while
  the **dependency:copy-dependencies** is a goal.

```
mvn clean dependency:copy-dependencies package
```

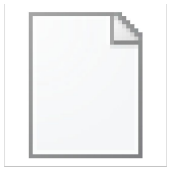
- Here the *clean* phase will be executed first, followed by
  the **dependency:copy-dependencies goal**, and finally *package* phase will be executed.

Clean Lifecycle

- When we execute *mvn post-clean* command, Maven invokes the clean lifecycle consisting of the following phases.
  - pre-clean
  - clean
  - post-clean
- Maven clean goal (clean:clean) is bound to the *clean* phase in the clean lifecycle. Its **clean:cleangoal** deletes the output of a build by deleting the build directory. Thus, when *mvn clean* command executes, Maven deletes the build directory.

Clean Lifecycle

- We can customize this behavior by mentioning goals in any of the above phases of clean life cycle.
- In the following example, We'll attach maven-antrun-plugin:run goal to the pre-clean, clean, and post-clean phases. This will allow us to echo text messages displaying the phases of the clean lifecycle.
- We've created a pom.xml in C:\MVN\project folder.



pom2.xml

Clean Lifecycle

- Now open command console, go to the folder containing pom.xml and execute the following mvn command.

```
C:\MVN\project>mvn post-clean
```

- Maven will start processing and displaying all the phases of clean life cycle.

- You can try tuning **mvn clean** command, which will display **pre-clean** and clean. Nothing will be executed for **post-clean** phase.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
-
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO] task-segment: [post-clean]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-clean}]
[INFO] Executing tasks
[echo] pre-clean phase
[INFO] Executed tasks
[INFO] [clean:clean {execution: default-clean}]
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
[echo] clean phase
[INFO] Executed tasks
[INFO] [antrun:run {execution: id.post-clean}]
[INFO] Executing tasks
[echo] post-clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
```

Default (or Build) Lifecycle

- This is the primary life cycle of Maven and is used to build the application. It has the following 21 phases.

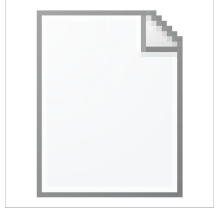
Default (or Build) Lifecycle.png

Default (or Build) Lifecycle

- There are few important concepts related to Maven Lifecycles, which are worth to mention −
  - When a phase is called via Maven command, for example **mvn compile**, only phases up to and including that phase will execute.
  - Different maven goals will be bound to different phases of Maven lifecycle depending upon the type of packaging (JAR / WAR / EAR).
- In the following example, we will attach maven-antrun-plugin:run goal to few of the phases of Build lifecycle. This will allow us to echo text messages displaying the phases of the lifecycle.

Default (or Build) Lifecycle

- We've updated pom.xml in C:\MVN\project folder.



pom3.xml

Default (or Build) Lifecycle

- Now open command console, go the folder containing pom.xml and execute the following mvn command.

```
C:\MVN\project>mvn compile
```

- Maven will start processing and display phases of build life cycle up to the compile phase.

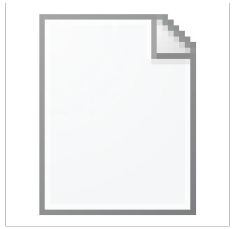
```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
_
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO] task-segment: [compile]
[INFO] ------------------------------------------------------------------------
_
[INFO] [antrun:run {execution: id.validate}]
[INFO] Executing tasks
[echo] validate phase
[INFO] Executed tasks
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered
resources,
i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory
C:\MVN\project\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [antrun:run {execution: id.compile}]
[INFO] Executing tasks
[echo] compile phase
```

Site Lifecycle

- Maven Site plugin is generally used to create fresh documentation to create reports, deploy site, etc. It has the following phases –
  - pre-site
  - site
  - post-site
  - site-deploy
- In the following example, we will attach **maven-antrun-plugin:run** goal to all the phases of Site lifecycle. This will allow us to echo text messages displaying the phases of the lifecycle.

Site Lifecycle

- We've updated pom.xml in C:\MVN\project folder.



pom4.xml

Site Lifecycle

- Now open the command console, go the folder containing pom.xml and execute the following mvn command.

```
C:\MVN\project>mvn site
```

- Maven will start processing and displaying the phases of site life cycle up to site phase.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO] task-segment: [site]
[INFO] ------------------------------------------------------------------------
[INFO] [antrun:run {execution: id.pre-site}]
[INFO] Executing tasks
[echo] pre-site phase
[INFO] Executed tasks
[INFO] [site:site {execution: default-site}]

[INFO] Generating "About" report.
[INFO] Generating "Issue Tracking" report.
[INFO] Generating "Project Team" report.
[INFO] Generating "Dependencies" report.
[INFO] Generating "Project Plugins" report.
[INFO] Generating "Continuous Integration" report.
[INFO] Generating "Source Repository" report.
[INFO] Generating "Project License" report.
[INFO] Generating "Mailing Lists" report.
[INFO] Generating "Plugin Management" report.
[INFO] Generating "Project Summary" report.
```

What is Build Profile?

- A Build profile is a set of configuration values, which can be used to set or override default values of Maven build. Using a build profile, you can customize build for different environments such as Production v/s Development environments.
- Profiles are specified in pom.xml file using its activeProfiles/profiles elements and are triggered in variety of ways. Profiles modify the POM at build time, and are used to give parameters different target environments (for example, the path of the database server in the development, testing, and production environments).

Types of Build Profile

- Build profiles are majorly of three types.

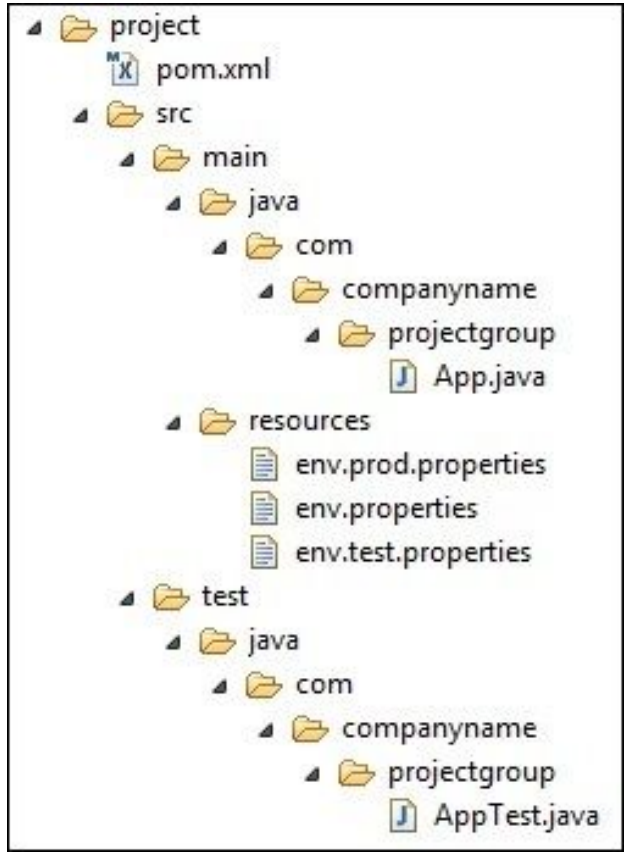
Type	Where it is defined
Per Project	Defined in the project POM file, pom.xml
Per User	Defined in Maven settings xml file (%USER_HOME%/.m2/settings.xml)
Global	Defined in Maven global settings xml file (%M2_HOME%/conf/settings.xml)

Profile Activation

- A Maven Build Profile can be activated in various ways.
  - Explicitly using command console input.
  - Through maven settings.
  - Based on environment variables (User/System variables).
  - OS Settings (for example, Windows family).
  - Present/missing files.

Profile Activation Examples

- Let us assume the following directory structure of your project –



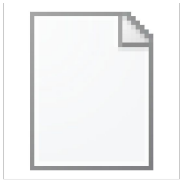
Profile Activation Examples

- Now, under **src/main/resources**, there are three environment specific files –

Sr.No.	File Name & Description
1	**env.properties** default configuration used if no profile is mentioned.
2	**env.test.properties** test configuration when test profile is used.
3	**env.prod.properties** production configuration when prod profile is used.

Explicit Profile Activation

- In the following example, we will attach maven-antrun-plugin:run goal to test the phase. This will allow us to echo text messages for different profiles. We will be using pom.xml to define different profiles and will activate profile at command console using maven command.
- Assume, we've created the following pom.xml in C:\MVN\project folder.



pom5.xml

Explicit Profile Activation

- Now open the command console, go to the folder containing pom.xml and execute the following mvn command. Pass the profile name as argument using -P option.

```
C:\MVN\project>mvn test -Ptest
```

- Maven will start processing and displaying the result of test build profile.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO] task-segment: [test]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources {execution: default-resources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] Copying 3 resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources {execution: default-testResources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] skip non existing resourceDirectory C:\MVN\project\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test {execution: default-test}]
```

Explicit Profile Activation

- Now as an exercise, you can perform the following steps –
  - Add another profile element to profiles element of pom.xml (copy existing profile element and paste it where profile elements ends).
  - Update id of this profile element from test to normal.
  - Update task section to echo env.properties and copy env.properties to target directory.
  - Again repeat the above three steps, update id to prod and task section for env.prod.properties.
  - That's all. Now you've three build profiles ready (normal/test/prod).

Explicit Profile Activation

- Now open the command console, go to the folder containing pom.xml and execute the following mvn commands. Pass the profile names as argument using -P option.

```
C:\MVN\project>mvn test -Pnormal
C:\MVN\project>mvn test -Pprod
```

- Check the output of the build to see the difference.

Profile Activation via Maven Settings

- Open Maven **settings.xml** file available in %USER_HOME%/.m2 directory where **%USER_HOME%** represents the user home directory. If settings.xml file is not there, then create a new one.
- Add test profile as an active profile using active Profiles node as shown below in example.

Profile Activation via Maven Settings

```
<settings xmlns = "http://maven.apache.org/POM/4.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/settings-1.0.0.xsd">
   <mirrors>
      <mirror>
         <id>maven.dev.snaponglobal.com</id>
         <name>Internal Artifactory Maven repository</name>
         <url>http://repo1.maven.org/maven2/</url>
         <mirrorOf>*</mirrorOf>
      </mirror>
   </mirrors>
   <activeProfiles>
      <activeProfile>test</activeProfile>
   </activeProfiles>
</settings>
```

Profile Activation via Maven Settings

- Now open command console, go to the folder containing pom.xml and execute the following mvn command. Do not pass the profile name using -P option. Maven will display result of test profile being an active profile.

```
C:\MVN\project>mvn test
```

Profile Activation via Environment Variables

- Now remove active profile from maven settings.xml and update the test profile mentioned in pom.xml. Add activation element to profile element as shown below.

- The test profile will trigger when the system property "env" is specified with the value "test". Create an environment variable "env" and set its value as "test".

```
<profile>
   <id>test</id>
   <activation>
      <property>
         <name>env</name>
         <value>test</value>
      </property>
   </activation>
</profile>
```

Profile Activation via Environment Variables

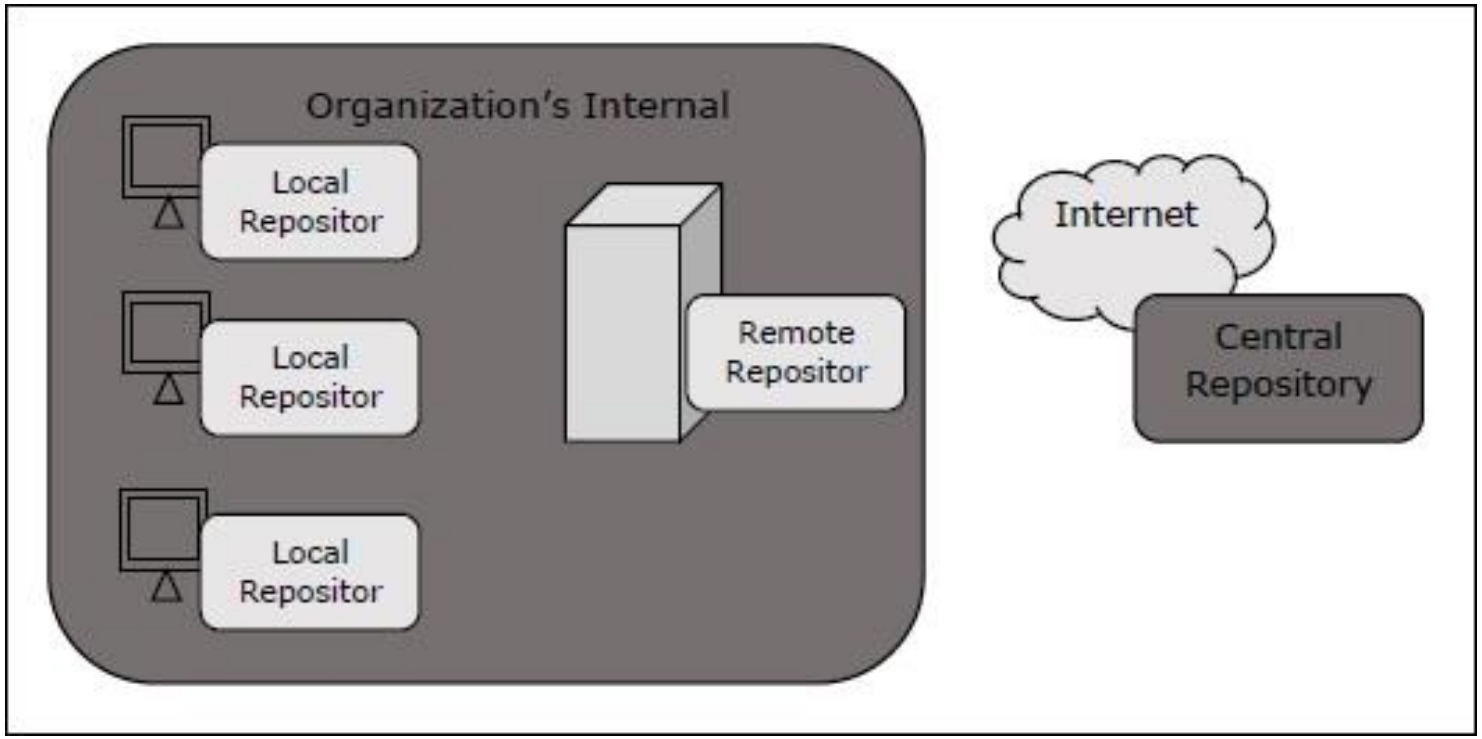
- Let's open command console, go to the folder containing pom.xml and execute the following mvn command.

```
C:\MVN\project>mvn test
```

What is a Maven Repository?

- In Maven terminology, a repository is a directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily.
- Maven repository are of three types. The following illustration will give an idea regarding these three types.
  - local
  - central
  - remote

What is a Maven Repository?



Local Repository

- Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time.
- Maven local repository keeps your project's all dependencies (library jars, plugin jars etc.). When you run a Maven build, then Maven automatically downloads all the dependency jars into the local repository. It helps to avoid references to dependencies stored on remote machine every time a project is build.

Local Repository

- Maven local repository by default get created by Maven in %USER_HOME% directory. To override the default location, mention another path in Maven settings.xml file available at %M2_HOME%\conf directory.

```
<settings xmlns = "http://maven.apache.org/SETTINGS/1.0.0"
   xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation = "http://maven.apache.org/SETTINGS/1.0.0
   http://maven.apache.org/xsd/settings-1.0.0.xsd">
   <localRepository>C:/MyLocalRepository</localRepository>
</settings>
```

- When you run Maven command, Maven will download dependencies to your custom path.

Central Repository

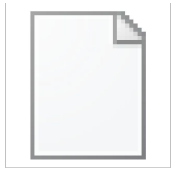
- Maven central repository is repository provided by Maven community. It contains a large number of commonly used libraries.
- When Maven does not find any dependency in local repository, it starts searching in central repository using following URL
  - https://repo1.maven.org/maven2/

Central Repository

- Key concepts of Central repository are as follows −
  - This repository is managed by Maven community.
  - It is not required to be configured.
  - It requires internet access to be searched.
- To browse the content of central maven repository, maven community has provided a URL − https://search.maven.org/#browse. Using this library, a developer can search all the available libraries in central repository.

Remote Repository

- Sometimes, Maven does not find a mentioned dependency in central repository as well. It then stops the build process and output error message to console. To prevent such situation, Maven provides concept of **Remote Repository**, which is developer's own custom repository containing required libraries or other project jars.
- For example, using below mentioned POM.xml, Maven will download dependency (not available in central repository) from Remote Repositories mentioned in the same pom.xml.



pom6.xml

Maven Dependency Search Sequence

- When we execute Maven build commands, Maven starts looking for dependency libraries in the following sequence –
  - **Step 1** – Search dependency in local repository, if not found, move to step 2 else perform the further processing.
  - **Step 2** – Search dependency in central repository, if not found and remote repository/repositories is/are mentioned then move to step 4. Else it is downloaded to local repository for future reference.
  - **Step 3** – If a remote repository has not been mentioned, Maven simply stops the processing and throws error (Unable to find dependency).
  - **Step 4** – Search dependency in remote repository or repositories, if found then it is downloaded to local repository for future reference. Otherwise, Maven stops processing and throws error (Unable to find dependency).

What are Maven Plugins?

- Maven is actually a plugin execution framework where every task is actually done by plugins. Maven Plugins are generally used to −
  - create jar file
  - create war file
  - compile code files
  - unit testing of code
  - create project documentation
  - create project reports

What are Maven Plugins?

- A plugin generally provides a set of goals, which can be executed using the following syntax –

```
mvn [plugin-name]:[goal-name]
```

- For example, a Java project can be compiled with the maven-compiler-plugin's compile-goal by running the following command.

```
mvn compiler:compile
```

Plugin Types

- Maven provided the following two types of Plugins –

Sr.No.	Type & Description
1	**Build plugins** They execute during the build process and should be configured in the <build/> element of pom.xml.
2	**Reporting plugins** They execute during the site generation process and they should be configured in the <reporting/> element of the pom.xml.

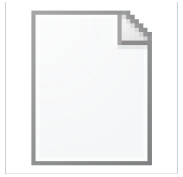
Plugin Types

- Following is the list of few common plugins −

Sr.No.	Plugin & Description
1	**clean** Cleans up target after the build. Deletes the target directory.
2	**compiler** Compiles Java source files.
3	**surefire** Runs the JUnit unit tests. Creates test reports.
4	**jar** Builds a JAR file from the current project.
5	**war** Builds a WAR file from the current project.
6	**javadoc** Generates Javadoc for the project.
7	**antrun** Runs a set of ant tasks from any phase mentioned of the build.

Example

- We've used **maven-antrun-plugin** extensively in our examples to print data on console. Refer Build Profiles chapter. Let us understand it in a better way and create a pom.xml in C:\MVN\project folder.



pom7.xml

Example

- Next, open the command console and go to the folder containing pom.xml and execute the following mvn command.

```
C:\MVN\project>mvn clean
```

- Maven will start processing and displaying the clean phase of clean life cycle.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - com.companyname.projectgroup:project:jar:1.0
[INFO]    task-segment: [post-clean]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] [antrun:run {execution: id.clean}]
[INFO] Executing tasks
     [echo] clean phase
[INFO] Executed tasks
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: < 1 second
[INFO] Finished at: Sat Jul 07 13:38:59 IST 2012
[INFO] Final Memory: 4M/44M
[INFO] ------------------------------------------------------------------------
```

Example

- The above example illustrates the following key concepts −
  - Plugins are specified in pom.xml using plugins element.
  - Each plugin can have multiple goals.
  - You can define phase from where plugin should starts its processing using its phase element. We've used **clean** phase.
  - You can configure tasks to be executed by binding them to goals of plugin. We've bound **echo** task with **run** goal of *maven-antrun-plugin*.
  - Maven will then download the plugin if not available in local repository and start its processing.

Creating Project

- Maven uses archetype plugins to create projects. To create a simple java application, we'll use maven-archetype-quickstart plugin. In example below, we'll create a maven based java application project in C:\MVN folder.
- Let's open the command console, go to the C:\MVN directory and execute the following mvn command.

```
C:\MVN>mvn archetype:generate
-DgroupId = com.companyname.bank
-DartifactId = consumerBanking
-DarchetypeArtifactId = maven-archetype-quickstart
-DinteractiveMode = false
```

Creating Project

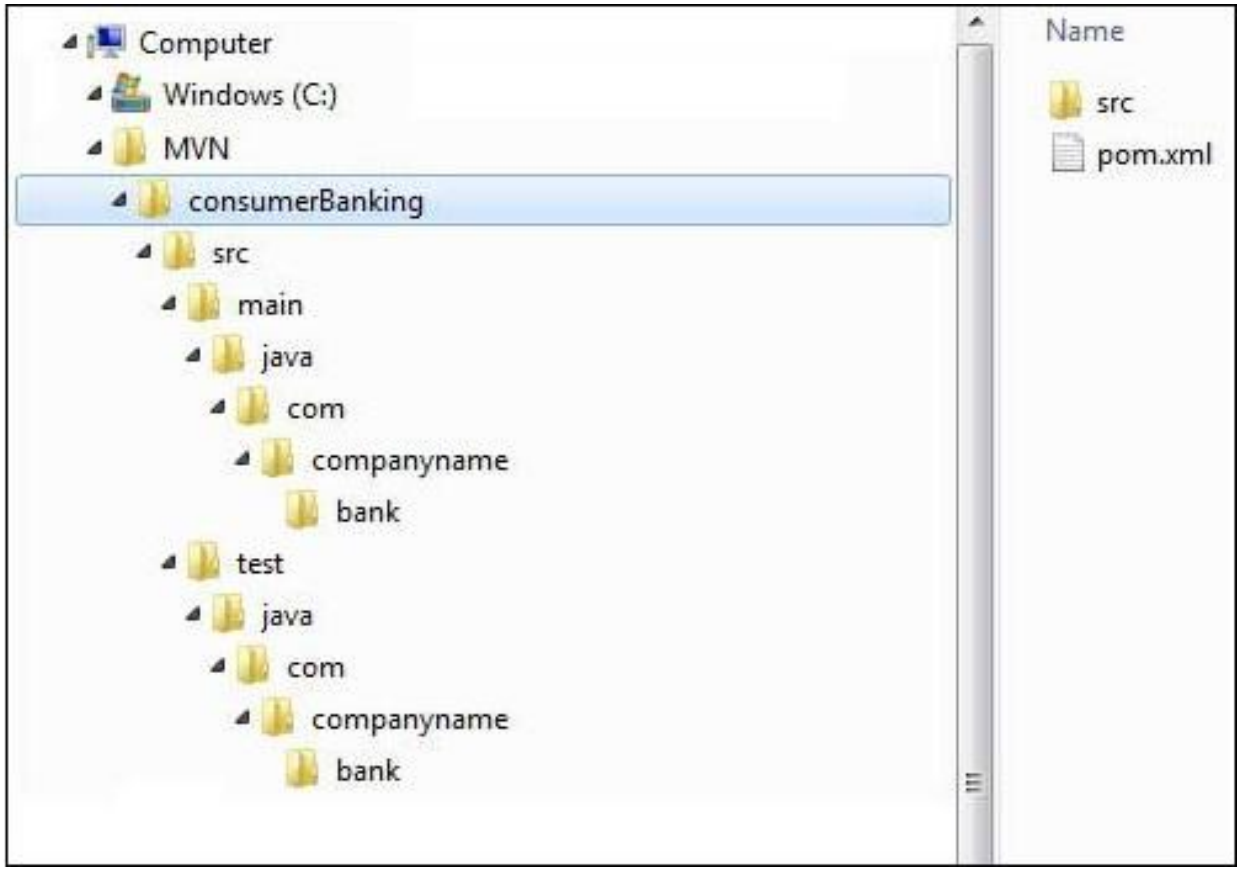
- Maven will start processing and will create the complete java application project structure.

```
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO]
------------------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:generate] (aggregator-style)
[INFO]
------------------------------------------------------------------------
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Batch mode
[INFO]
------------------------------------------------------------------------
[INFO] Using following parameters for creating project
from Old (1.x) Archetype: maven-archetype-quickstart:1.0
[INFO]
------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.companyname.bank
[INFO] Parameter: packageName, Value: com.companyname.bank
[INFO] Parameter: package, Value: com.companyname.bank
[INFO] Parameter: artifactId, Value: consumerBanking
[INFO] Parameter: basedir, Value: C:\MVN
[INFO] Parameter: version, Value: 1.0-SNAPSHOT

[INFO] project created from Old (1.x) Archetype in dir: C:\MVN\consumerBanking
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 14 seconds
[INFO] Finished at: Tue Jul 10 15:38:58 IST 2012
[INFO] Final Memory: 21M/124M
[INFO] ------------------------------------------------------------------------
```

Creating Project

- Now go to C:/MVN directory. You'll see a java application project created, named consumer Banking (as specified in artifactId). Maven uses a standard directory layout as shown below –



Creating Project

- Using the above example, we can understand the following key concepts −

Sr.No.	Folder Structure & Description
1	**consumerBanking** contains src folder and pom.xml
2	**src/main/java** contains java code files under the package structure (com/companyName/bank).
3	**src/main/test** contains test java code files under the package structure (com/companyName/bank).
4	**src/main/resources** it contains images/properties files (In above example, we need to create this structure manually).

Creating Project

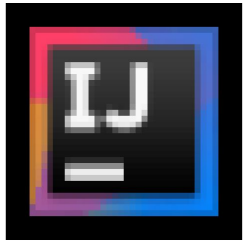
- If you observe, you will find that Maven also created a sample Java Source file and Java Test file. Open C:\MVN\consumerBanking\src\main\java\com\companyname\bank folder, you will see App.java.

```
package com.companyname.bank;

/**
 * Hello world!
 *
 */
public class App {
    public static void main( String[] args ){
        System.out.println( "Hello World!" );
    }
}
```

Creating Project

- Open C:\MVN\consumerBanking\src\test\java\com\companyname\bank folder to see AppTest.java

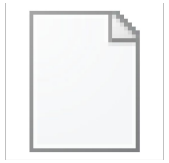


AppTest.java

- Developers are required to place their files as mentioned in table above and Maven handles all the build related complexities.

Build & Test Project

- What we learnt in Project Creation chapter is how to create a Java application using Maven. Now we'll see how to build and test the application.
- Go to C:/MVN directory where you've created your java application.
  Open **consumerBanking**folder. You will see the **POM.xml** file with the following contents.



pom8.xml

Build & Test Project

- Here you can see, Maven already added Junit as test framework. By default, Maven adds a source file App.java and a test file AppTest.java in its default directory structure, as discussed in the previous chapter.
- Let's open the command console, go the C:\MVN\consumerBanking directory and execute the following mvn command.

```
C:\MVN\consumerBanking>mvn clean package
```

Build & Test Project

- Maven will start building the project.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building consumerBanking
[INFO] task-segment: [clean, package]
[INFO] ------------------------------------------------------------------------
[INFO] [clean:clean {execution: default-clean}]
[INFO] Deleting directory C:\MVN\consumerBanking\target
[INFO] [resources:resources {execution: default-resources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] skip non existing resourceDirectory C:\MVN\consumerBanking\src\main\resources
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Compiling 1 source file to C:\MVN\consumerBanking\target\classes
[INFO] [resources:testResources {execution: default-testResources}]

[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
i.e. build is platform dependent!

[INFO] skip non existing resourceDirectory C:\MVN\consumerBanking\src\test\resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
```

Build & Test Project

- You've built your project and created final jar file, following are the key learning concepts −
  - We give maven two goals, first to clean the target directory (clean) and then package the project build output as jar (package).
  - Packaged jar is available in consumerBanking\target folder as consumerBanking-1.0-SNAPSHOT.jar.
  - Test reports are available in consumerBanking\target\surefire-reports folder.
  - Maven compiles the source code file(s) and then tests the source code file(s).
  - Then Maven runs the test cases.
  - Finally, Maven creates the package.

Build & Test Project

- Now open the command console, go the C:\MVN\consumerBanking\target\classes directory and execute the following java command.

```
>java com.companyname.bank.App
```

- You will see the result as follows –

```
Hello World!
```

Adding Java Source Files

- Let's see how we can add additional Java files in our project. Open C:\MVN\consumerBanking\src\main\java\com\companyname\bank folder, create Util class in it as Util.java.

```
package com.companyname.bank;

public class Util {
   public static void printMessage(String message){
      System.out.println(message);
   }
}
```

Adding Java Source Files

- Now open the command console, go the C:\MVN\consumerBanking directory and execute the following mvn command.

```
>mvn clean compile
```

- After Maven build is successful, go to the C:\MVN\consumerBanking\target\classes directory and execute the following java command.

```
>java -cp com.companyname.bank.App
```

- You will see the result as follows –

```
Hello World!
```

External Dependencies

- As you know, Maven does the dependency management using the concept of Repositories. But what happens if dependency is not available in any of remote repositories and central repository? Maven provides answer for such scenario using concept of **External Dependency**.
- For example, let us do the following changes to the project created in ‘Creating Java Project’ chapter.
  - Add **lib** folder to the src folder.
  - Copy any jar into the lib folder. We've used **ldapjdk.jar**, which is a helper library for LDAP operations.

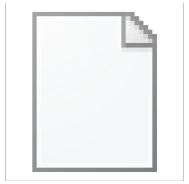
External Dependencies

- Now our project structure should look like the following –

- Here you are having your own library, specific to the project, which is an usual case and it contains jars, which may not be available in any repository for maven to download from. If your code is using this library with Maven, then Maven build will fail as it cannot download or refer to this library during compilation phase.

External Dependencies

- To handle the situation, let's add this external dependency to maven **pom.xml** using the following way.



pom9.xml

External Dependencies

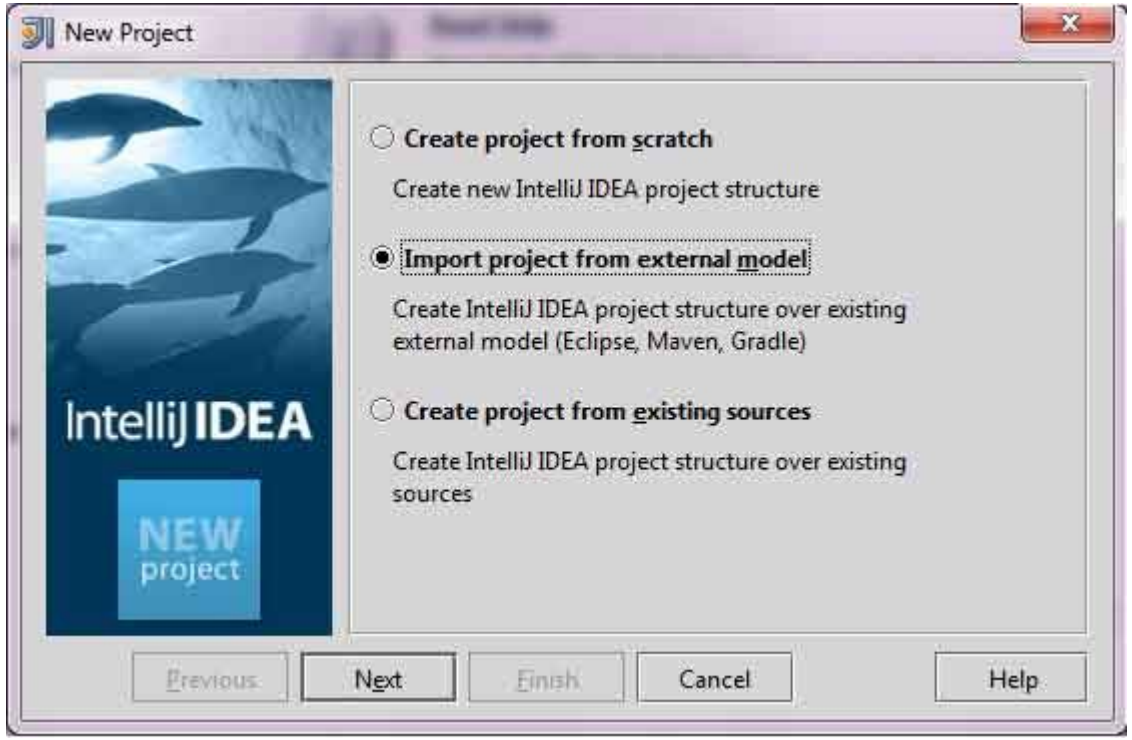
- Look at the second dependency element under dependencies in the above example, which clears the following key concepts about **External Dependency**.
  - External dependencies (library jar location) can be configured in pom.xml in same way as other dependencies.
  - Specify groupId same as the name of the library.
  - Specify artifactId same as the name of the library.
  - Specify scope as system.
  - Specify system path relative to the project location.
- Hope now you are clear about external dependencies and you will be able to specify external dependencies in your Maven project.

IntelliJ IDEA IDE Integration

- IntelliJ IDEA has in-built support for Maven. We are using IntelliJ IDEA Community Edition 11.1 in this example.
- Some of the features of IntelliJ IDEA are listed below −
  - You can run Maven goals from IntelliJ IDEA.
  - You can view the output of Maven commands inside the IntelliJ IDEA using its own console.
  - You can update maven dependencies within IDE.
  - You can Launch Maven builds from within IntelliJ IDEA.
  - IntelliJ IDEA does the dependency management automatically based on Maven's pom.xml.
  - IntelliJ IDEA resolves Maven dependencies from its workspace without installing to local Maven repository (requires dependency project be in same workspace).
  - IntelliJ IDEA automatically downloads the required dependencies and sources from the remote Maven repositories.
  - IntelliJ IDEA provides wizards for creating new Maven projects, pom.xml.

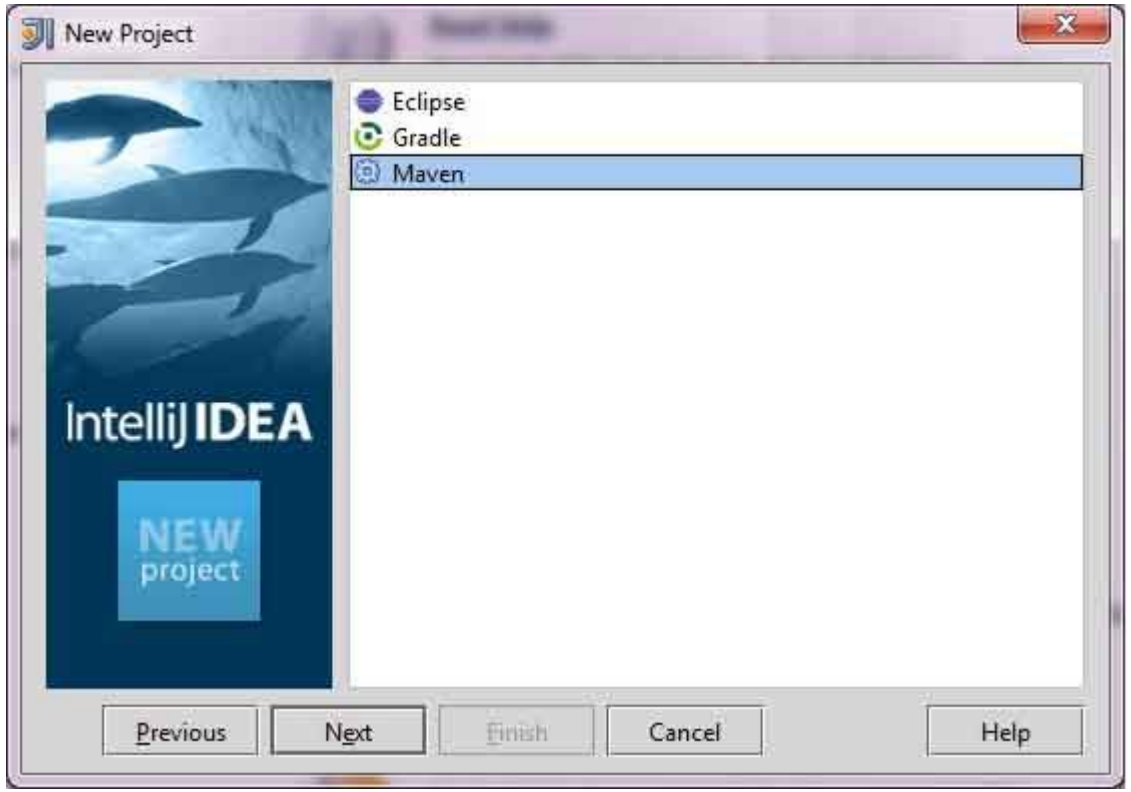
Create a new project in IntelliJ IDEA

- We will import Maven project using New Project Wizard.
  - Open IntelliJ IDEA.
  - Select **File Menu > New Project** Option.
  - Select import project from existing model.



Create a new project in IntelliJ IDEA

- Select Maven option



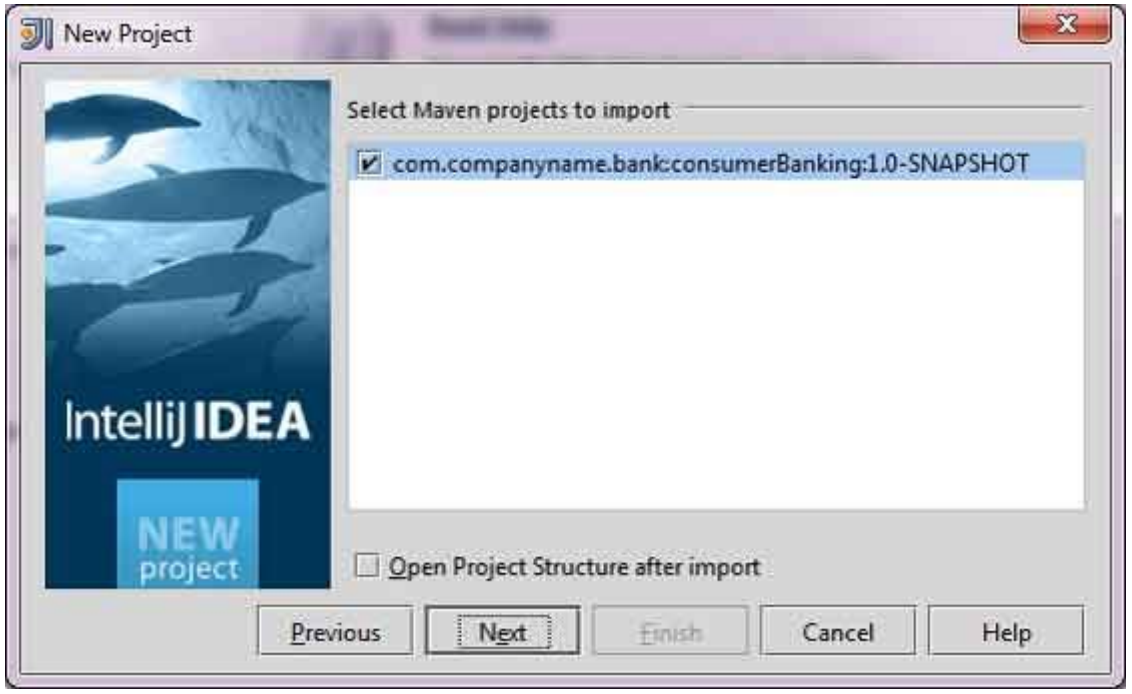
Create a new project in IntelliJ IDEA

- Select Project location, where a project was created using Maven. We have created a Java Project consumerBanking. Go to ‘Creating Java Project' chapter, to see how to create a project using Maven.

G2 | Academy

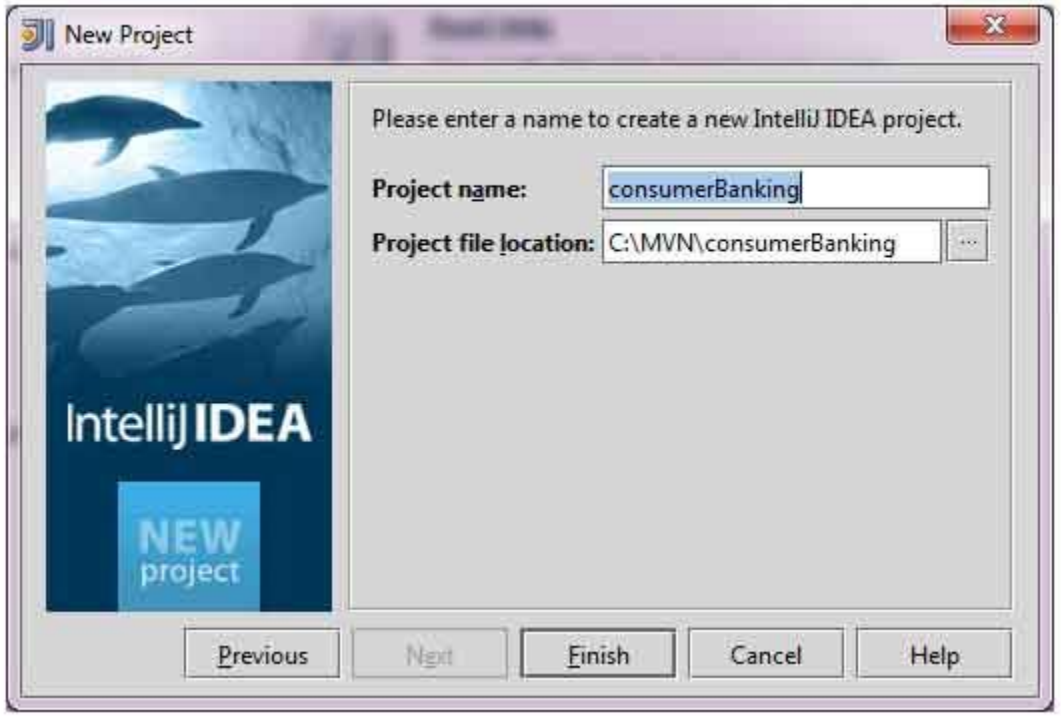
Create a new project in IntelliJ IDEA

- Select Maven project to import.



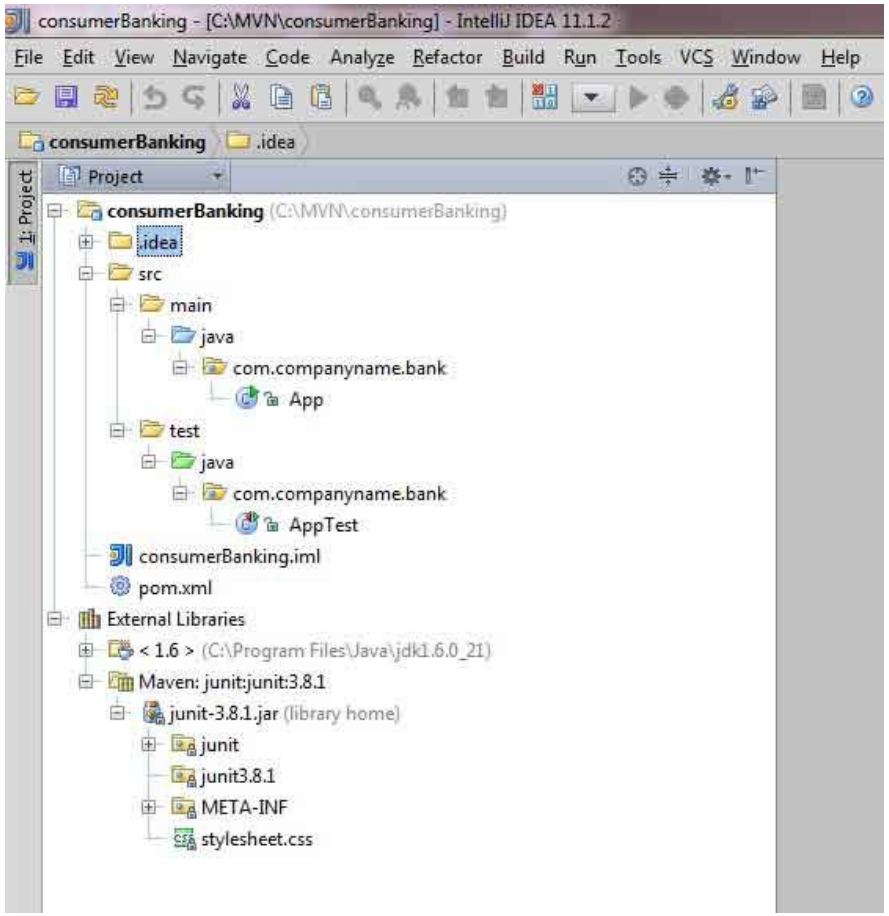
Create a new project in IntelliJ IDEA

- Enter name of the project and click finish.



Create a new project in IntelliJ IDEA

- Now, you can see the maven project in IntelliJ IDEA. Have a look at consumerBanking project external libraries. You can see that IntelliJ IDEA has added Maven dependencies to its build path under Maven section.



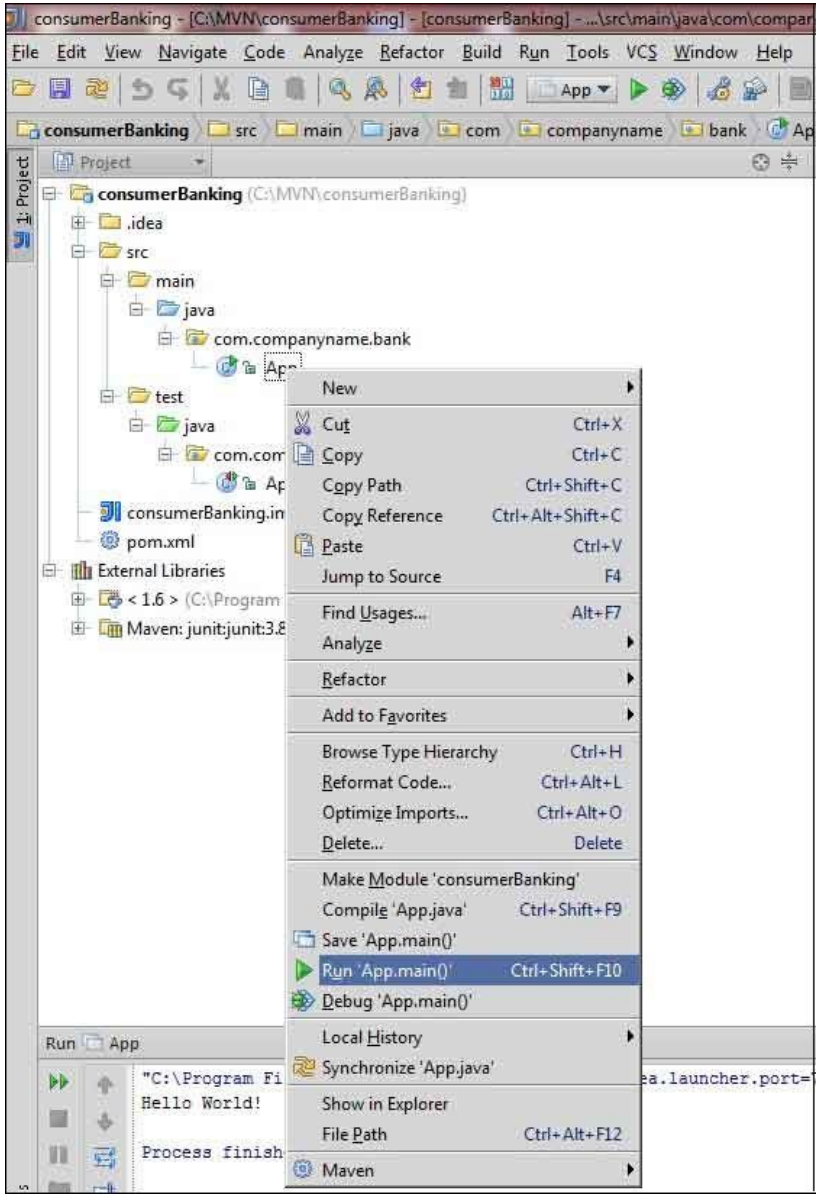
Build a maven project in IntelliJ IDEA

- Now, it is time to build this project using capability of IntelliJ IDEA.
  - Select consumerBanking project.
  - Select Buid menu > Rebuild Project Option
- You can see the output in IntelliJ IDEA Console

```
4:01:56 PM Compilation completed successfully
```

Run Application in IntelliJ IDEA

- Select consumerBanking project.
- Right click on App.java to open context menu.
- select **Run App.main()**



Run Application in IntelliJ IDEA

- You will see the result in IntelliJ IDEA Console.

```
"C:\Program Files\Java\jdk1.6.0_21\bin\java"
-Didea.launcher.port=7533
"-Didea.launcher.bin.path=
C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 11.1.2\bin"
-Dfile.encoding=UTF-8
-classpath "C:\Program Files\Java\jdk1.6.0_21\jre\lib\charsets.jar;

C:\Program Files\Java\jdk1.6.0_21\jre\lib\deploy.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\javaws.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\jce.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\jsse.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\management-agent.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\plugin.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\resources.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\rt.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\dnsns.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\localedata.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunjce_provider.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunmscapi.jar;
C:\Program Files\Java\jdk1.6.0_21\jre\lib\ext\sunpkcs11.jar

C:\MVN\consumerBanking\target\classes;
```

G2 | Academy