

Jenkins

Lab 1

1	LAB SETUP	1
2	CHANGING PORT NUMBERS FOR JENKINS SERVER	2
3	BASIC JENKINS JOB WITH WINDOWS BATCH COMMANDS	3
4	INTRODUCING MAVEN FOR BUILDING JAVA PROJECTS	7
5	INSTALL AND CONFIGURE MAVEN PLUGIN	11
6	BASIC JENKINS JOB TO PERFORM MAVEN BUILD FROM CLI	12
7	CREATING A GITHUB REPO FOR THE PROJECT	15
8	CHECKING / INSTALLING GIT / GITHUB PLUGINS FOR JENKINS	18
9	BASIC JENKINS JOB TO INTEGRATE WITH GITHUB	18
10	CONFIGURING PERIODIC BUILDS AND POLLING.....	24
11	CONFIGURING EMAIL NOTIFICATION	26
12	CONFIGURING TOMCAT SERVER FOR DEPLOYING A JAVA WEB APP.....	30
13	BUILDING AND DEPLOYING A JAVA WEB APP.....	31
14	CREATING A GITHUB REPO FOR THE JAVA WEB APP PROJECT	33
15	SET UP CREDENTIALS AND PLUGIN TO INTEGRATE WITH TOMCAT CONTAINER	35
16	BASIC JENKINS JOB TO BUILD AND DEPLOY A JAVA WEB APP.....	36

1 Lab setup

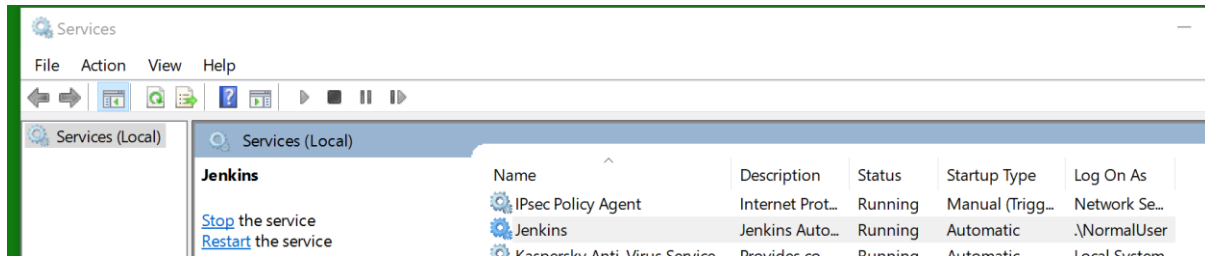
Make sure you have a suitable text editor installed for your OS (e.g. Notepad ++)

Create a new empty directory with the name `jenkinslabs` in any directory on your local machine that you have read/write access to (for e.g. if you are using a company desktop / laptop for this workshop, you probably only have read/write access within your home directory while you are logged into your account). If so, you could create a directory like this (`Desktop\jenkinslabs`)

This `jenkinslabs` directory will serve as the top-level directory to hold the project folders that we will be working with in the upcoming lab sessions.

2 Changing port numbers for Jenkins server

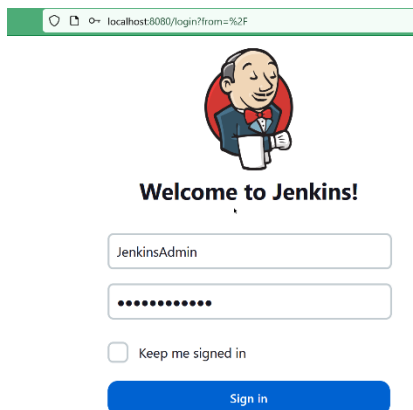
Jenkins is an automation server that runs on the default port of 8080 when installed. Typically, it is installed on Windows as a service, which you can access via the Services app.



If Jenkins service is already running, you can open a browser tab to

<http://localhost:8080>

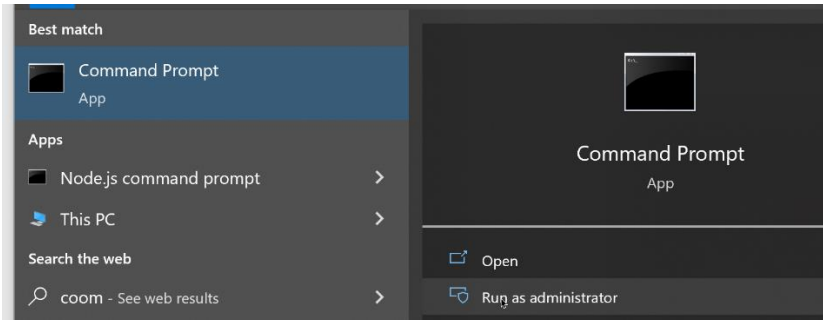
where you will see the main login page for the Jenkins UI, and you can login using the admin username / password combination that you registered during the installation process.



However, there may be other services / apps on your local machine that run on port 8080 by default as well (for e.g. the Tomcat server), so to prevent a port conflict that will cause either one of the conflicting apps to fail to start up, we can choose to change the default port that Jenkins runs on if necessary.

Typically, before changing port numbers for any app / service, we will check which ports are being used by existing running processes and which ones are free. There are 65,535 port numbers, and the restricted port numbers or well-known port numbers are reserved by prominent companies and range from 0 to 1023.

On Windows, to check for processes that are bound to a given port, we can open a command prompt with admin privilege:



To see the list of all ports that are being actively used type:

```
netstat -aon
```

To narrow this down to a specific port number, type:

```
netstat -aon | findstr :port-number
```

This will provide a listing of processes (identified by their PID) which are listening on that specified port

TCP	0.0.0.0:port-number	0.0.0.0:0	LISTENING	PID
TCP	:::port-number	:::0	LISTENING	PID

To find the actual name of the program / process associated with that PID, type:

```
tasklist | findstr PID
```

Finally, if you wish to terminate the program / process running on that port, type:

```
taskkill /PID PID /F
```

The instructions below demonstrate how to change the port number for the Jenkins server

<https://phoenixnap.com/kb/jenkins-change-port>

Practice changing this to different port numbers and starting / restarting your Jenkins server. We may need to repeat this at a later point when we work with the Tomcat server.

3 Basic Jenkins job with Windows batch commands

A Jenkins build job contains the configuration for automating a specific task or step in the application building process. These tasks include gathering dependencies, compiling, archiving, or transforming code, and testing and deploying code in different environments.







Jenkins supports several types of build jobs, such as freestyle projects, pipelines, multi-configuration projects, folders, multibranch pipelines, and organization folders. The freestyle project is the most

common and widely used build job, while the others are specializations of the freestyle project for specific use case scenarios.

From the main Jenkins dashboard, create a new Item and name it: `DemoWindowsBatch`. Make this a Freestyle Project and select Ok.

The configuration for this job is divided into several different sections.




Configure

-  General
-  Source Code Management
-  Build Triggers
-  Build Environment
-  Build Steps
-  Post-build Actions

Type some random info in the Description for this project in the General section.

Configure

General


-  General
-  Source Code Management
-  Build Triggers

Description

Demonstrating how to use DOS commands in a build

In the Build Steps section, choose Add Build step -> Execute Windows Batch command, and add in the following DOS command line commands:

Build Steps

 **Execute Windows batch command** ?

Command

See [the list of available environment variables](#)

```
echo "Hi there from Jenkins"
echo "Showing contents of current job folder"
dir
echo "making a new directory in the job folder and creating a new
file in it"
```

```
mkdir cool
cd cool
echo "I love Jenkins ! Its the most awesome CI tool ever ... " >
hello.txt
echo "Showing the version of Java and Maven installed on this
machine"
java -version
mvn -version
```

The complete list of DOS command line commands:

<https://www.computerhope.com/overview.htm>

The list of most frequently used command line commands:

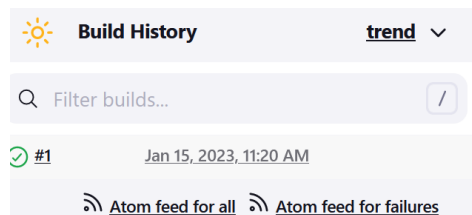
<https://www.computerhope.com/dostop10.htm>

Some basic tutorials to working with the DOS command prompt:

<https://www.computerhope.com/issues/chusedos.htm>

Click Save.

Back at the main Job dashboard, click Build Now to run a build of the job. This should succeed and you should see a green circle next to a number (the particular build attempt) indicating success in the Build History.



Click on the Green Circle to get more detailed info on that particular build attempt.


Status


</> Changes


Console Output

Edit Build Information

Delete build '#1'

 **Build #1 (Jan 15, 2023, 11:20:59 AM)**

 No changes.

 Started by user [Jenkins Admin](#)

The most important option you can explore from the navigation pane on the left is the Console Output, which is the log of all the activities that occurred during the build.


Dashboard > DemoWindowsBatch > #1

Status

Changes

Console Output

View as plain text


Console Output

Started by user [Jenkins Admin](#)

Running as SYSTEM

Building in workspace C:\Users\NormalUser\

[DemoWindowsBatch] \$ cmd /c call C:\Users\

Some points to note:

- The name of the logged in user (`Jenkins Admin`) is shown
- The complete path of the workspace folder is shown. This is typically in the home directory of the currently logged in user in `AppData\Local\Jenkins\.jenkins\workspace`, and the actual workspace folder is the same name as the job (`DemoWindowsBatch`). You can open this location in File Explorer by copying and pasting this into the navigation bar.
- The rest of the listing is the console output corresponding to the execution of the various DOS commands. You can repeat this commands yourself in a separate DOS prompt in any random directory on your machine (for e.g. `jenkinslabs`) to verify that they accomplish the same effect. Delete all the content when you are done.

Click on Edit Build Information to provide more random information on this particular build attempt, and then click Save.

Status

Changes

Console Output

Edit Build Information


DisplayName ?

Description ?

Return back to the main dashboard for the job. Notice that the display name and description for this build attempt is now shown in the Build History.

Build History

trend ▾


My first build attempt

Jan 15, 2023, 11:20 AM

All good to go !

If you click on Workspace, you will be able to get a UI that allows you to navigate the workspace folder structure (which was created as part of the build process) as well as download the entire folder to our local machines (which would be useful if Jenkins was running on a remote server).

The screenshot shows the Jenkins workspace interface. On the left, there's a sidebar with 'Status', 'Changes', 'Workspace', 'Wipe Out Current Workspace', and 'Build Now'. The main area is titled 'Workspace of DemoWindowsBatch on Built-In Node'. It shows the path 'DemoWindowsBatch /' with a search bar and a 'cool' status. Below the path, there's a link '(all files in zip)'. The 'Build Now' button is at the bottom left.

Click Build Now again to perform another build attempt. This should succeed and the build attempt number is shown in the Build History.

The screenshot shows the Jenkins Build History section. It has a 'Build History' header with a 'trend' dropdown. Below the header is a search bar 'Filter builds...'. The build history list shows two builds: '#2' and 'My first build attempt'. The 'My first build attempt' build is highlighted and shows a timestamp 'Jan 15, 2023, 11:20 AM' and a status 'All good to go!'.

Repeat a build a few more times to see the build attempt list grow in the Build History. The Permalinks section in the middle provide links for you to navigate to different build attempts depending on their status (stable, successful, completed). So far for our very simple job, all the builds fall into this category.

Project DemoWindowsBatch

Demonstrating how to use DOS commands in a build

Permalinks

- Last build (My first build attempt), 11 min ago
- Last stable build (My first build attempt), 11 min ago
- Last successful build (My first build attempt), 11 min ago
- Last completed build (My first build attempt), 11 min ago

NOTE: If you are an admin used to writing PowerShell scripts, there is a PowerShell plugin which you can add to allow you to write PowerShell scripts directly into the box similar to what is demonstrated here for the case of DOS commands.

<https://plugins.jenkins.io/powershell/>

4 Introducing Maven for building Java projects

Different programming languages / frameworks utilized different build tools. The most popular build tool for Java projects is Maven, which is an evolution from an earlier Java build tool, Ant.

<https://maven.apache.org/>

Create a folder `githubprojects` in the top-level `jenkinslabs` directory that you created in your initial lab setup. Copy `simple-java-maven-app` from the `labcode` of your downloaded zip into the `githubprojects` folder

Open a DOS prompt in `simple-java-maven-app`, which is the project root folder for a simple Java app that we will build using Maven.

First, check that you have a recent version of Maven installed and accessible from the command prompt by typing:

```
mvn -version
```

You should have version 3.8.6 or later for this lab to work

When working from the command line, you invoke Maven directly using the command line tool `mvn` and passing the required arguments.

Each of the 3 core build lifecycles in Maven (`default`, `clean` and `site`) is defined by a different list of build phases (or stages) in the lifecycle.

<https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#lifecycle-reference>

A particular build phase is performed by executing one or more tasks. Each of these tasks is handled by a plugin goal. A plugin goal may therefore be bound to zero or more build phases. If a goal is bound to one or more build phases, that goal will be called in all those phases.

The bindings for the `clean` and `site` lifecycle phases are fixed. The bindings for the default life cycle phases depending on the `packaging` value.

We can invoke Maven by specifying a phase in any of the build life cycles such as `compile` or `package`

The first thing we will typically do in a standard project is to clean up the artifacts (classes, JARs, etc) from a previous build operation. To do that we can use the `clean` phase from the `clean` life cycle. Type:

```
mvn clean
```

If this is the first time you are running Maven, a whole bunch of related plugins and dependencies will be downloaded from the central Maven repository and stored in the local cache. This may take some time to complete, depending on the capacity of your broadband connection. Once they are available in the local cache, Maven will reference them there for future executions of this phase, thus removing the need for costly network access.

We can also specify two or more phases to be executed: they will be executed in the order they appear.

Select option 4 - Maven Build to bring up the Edit Configuration dialog box and enter for the goal:
`mvn clean compile`

This performs a clean and also a compilation of the source code. The bytecode classes are now placed in a specific directory for this project (by default for most Java projects this will be `\target\classes`). You can check for the bytecode classes there in File Explorer.

Most projects will ship with some test code to test the main application classes. At the most basic, these will take the form of unit tests implemented using some unit testing library (JUnit for the case of Java projects) - notice that this was included in the dependency list of the POM.

To test the compiled source code, we can type:

```
mvn test
```

Console output from the Maven build indicates that the tests were successful, as we are just implementing some very basic dummy tests here.

Finally we will produce a build artifact that we can directly execute. For the case of a Java project, this is typically a JAR file or a WAR file (Web Archive for deploying a web app to a server such as Tomcat).

To accomplish this, we can type:

```
mvn package
```

Notice that the tests are run as part of the `package` phase, since the `test` phase precedes the `package` phase. This makes sense since we want to ensure that the code base passes all tests before it is deployed to a production environment: a key requirement in the CI part of a CI/CD pipeline in DevOps.

We can also see the directory where the JAR artifact is generated in from the Maven console output. We can change into that directory and execute the artifact with:

```
cd target
java -jar my-app-1.0-SNAPSHOT.jar
```

Let's make a minor modification to the source code of the main class for this simple Java project at:

```
src\main\java\com\mycompany\app\App.java
```

using Notepad++ (or any other suitable editor)

Change the parameter of the `System.out.println` to some other random value, for e.g. "Jenkins is awesome"

Save and exit.

In the command prompt, return back out to the root project folder and perform a clean (to remove all files from the previous build) and then generate the build artifact with:

```
cd ..
mvn clean package
```

Again, change into the directory holding the generated build artifact and execute it with:

```
cd target
```

```
java -jar my-app-1.0-SNAPSHOT.jar
```

Verify that your changes have taken effect. Repeat this a few more times (but don't change the value of the String MESSAGE yet).

Finally, let's simulate an error occurring in the unit tests. Back in the same file:

```
src\main\java\com\mycompany\app\App.java
```

Change the value of String MESSAGE to some other value, for e.g. "Dummy String"

Save and exit.

Now if you check the corresponding unit test for the previous class at:

```
src\test\java\com\mycompany\app\AppTest.java
```

You will notice that the change you made will cause the statement

```
assertEquals("Hello World!", app.getMessage());
```

to fail.

In the command prompt, return back out to the root project folder and perform a clean (to remove all files from the previous build) and then generate the build artifact with:

```
cd ..  
mvn clean package
```

This time the Maven output is in red, indicating the tests have failed. Also, notice that there is no JAR file generated in the `target` directory, since the build process did not complete successfully.

If you check the content of `target\surefire-reports\`, you will see two files in different formats (*.txt and *.xml) which contain information pertaining to the tests that failed

Return back to the app source code file:

```
src\main\java\com\mycompany\app\App.java
```

Restore the value of String MESSAGE to its original value: "Hello World!"

Save and exit.

Now repeat the Maven goals again:

```
mvn clean package
```

This time the build process completes without error, and we can change into the directory holding the generated build artifact and execute it with:

```
cd target  
java -jar my-app-1.0-SNAPSHOT.jar
```

The other main alternative to Maven for Java build tools is Gradle

<https://gradle.org/>

There are tools that perform some or part of Maven build functionality (e.g. dependency and package management) for other programming languages / frameworks:

Nuget for C# .NET Core projects

<https://learn.microsoft.com/en-us/nuget/what-is-nuget>

Pip, VirtualEnv and VirtualEnvwrapper for Python

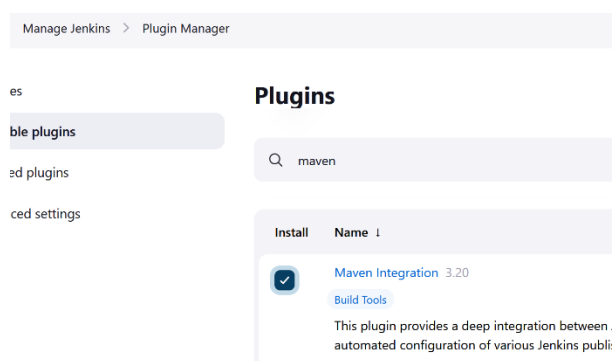
<https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments>

<https://pypi.org/project/pip/>

<https://virtualenvwrapper.readthedocs.io/en/latest/>

5 Install and configure Maven plugin

From the main Jenkins Dashboard, Manage Jenkins -> Manage Plugins -> Available Plugins, look for Maven Integration and select the item.



Click Install without Restart.

After download progress is complete,

Download progress

Preparation	<ul style="list-style-type: none">• Checking internet connectivity• Checking update center connectivity• Success
Javadoc	✓ Success
Maven Integration	✓ Success
Loading plugin extensions	✓ Success

Check the box: Restart Jenkins when installation is complete and no jobs are running.
Wait for Jenkins to restart.



Please wait while Jenkins is restarting .

Your browser will reload automatically when Jenkins is ready.

From the main Jenkins Dashboard, Manage Jenkins -> Manage Plugins -> Installed Plugins, and verify that Maven Integration plugin is installed.

From the main Jenkins Dashboard, Manage Jenkins -> Global Tool configuration, click Add Maven and give it a suitable name and then click Save. This will kick of an auto installation of Maven in the background.

Maven installations

List of Maven installations on this system

Add Maven

≡ Maven

Name

jenkins-maven

☒ Install automatically ?

≡ Install from Apache

Version

3.8.7

Add Installer ▾

6 Basic Jenkins job to perform Maven build from CLI

From the main Jenkins dashboard, create a new Item and name it: `DemoMavenCLI`
Make this a Freestyle Project and select Ok.

In the Build Steps section, choose Add Build step -> Execute Windows Batch command, and add in the single command:

```
xcopy full-file-path-to-simple-java-maven-app full-file-path-to-Jenkins-workspace \ /E /Y /H
```

Fill in the italics with the actual file paths on your system. Make sure to include the `\ /E /Y /H` at the end after the second file path.

A sample of this command on my system would be:

```
xcopy G:\jenkinslab\githubprojects\simple-java-maven-app  
C:\Users\NormalUser\AppData\Local\Jenkins\.jenkins\workspace\DemoMavenCLI\ /E /Y /H
```

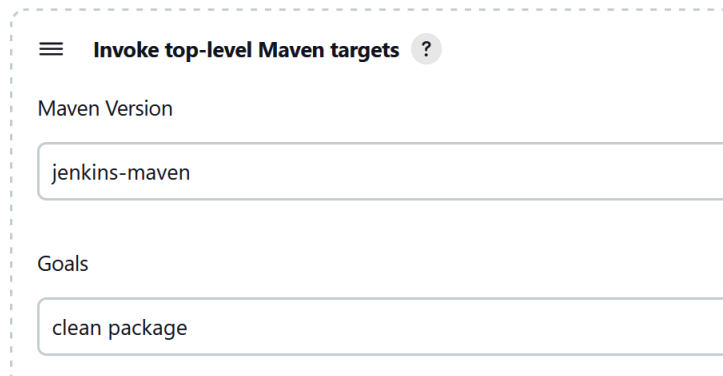
Click Save, then perform a Build.

If the Build is not successful, check the file paths in the command carefully and run it again.

After a successful build, check that the content of the job folder `workspace\DemoMavenCLI` now contains all the files originally in `simple-java-maven-app`

In the main area for the job, select Configure, return to Build Steps and add another Build Step after the first one, this time with Invoke Top Level Maven Targets:

Select the Maven Version that you configured previously and add the goals: `clean package`



≡ **Invoke top-level Maven targets** ?

Maven Version

jenkins-maven

Goals

clean package

Click Save, then perform a Build.

The build should be a success, and the Console output shows the output from Maven, with the JAR artifact being generated successfully at the end.

In the main area for the job, select Configure, go to Post-Build Actions, and add an action -> Publish JUnit test result report

Add in this for the file set: `target/surefire-reports/*.xml`

This is the location for where the test report XMLs are located relative to the workspace root (DemoMavenCLI).

Click Save and perform a Build again.

Click Changes and Status on the main area for the job, or else simply refresh the page. A test result trend should now appear, keeping track of the test results.

Project DemoMavenCLI



Return back to the original project location at:

jenkinslab\githubprojects\simple-java-maven-app

Let's simulate an error occurring in the unit tests. In the file:

src\main\java\com\mycompany\app\App.java

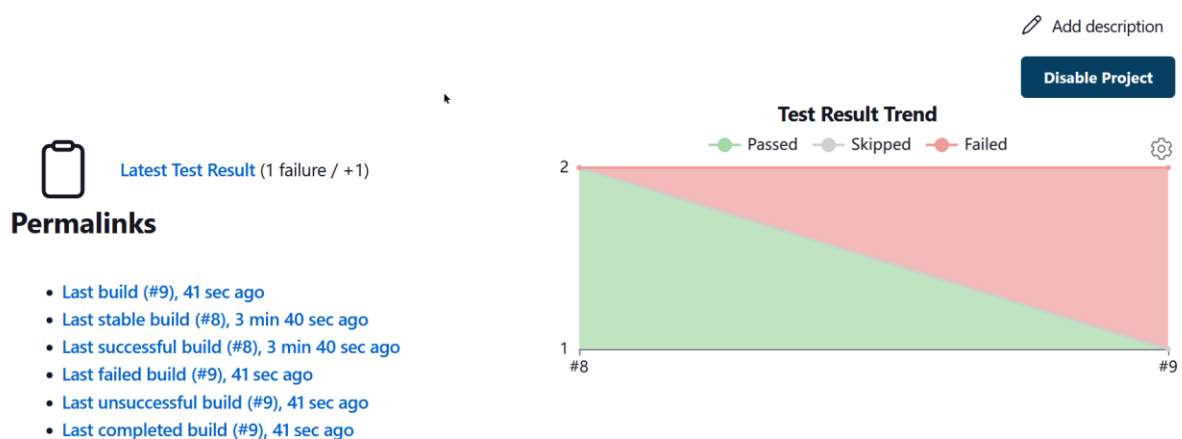
Change the value of String MESSAGE to some other value, for e.g. "Dummy String"

Save and exit.

Run the build again. This time you should see the error reported in the Console Output for that particular build attempt.

The main area for the job should now indicate the build failure in the Test result trend and also provide links to the various builds with various statuses (stable, successful, failed, unsuccessful, etc)

Project DemoMavenCLI



Return back to the app source code file:

src\main\java\com\mycompany\app\App.java

Restore the value of String MESSAGE to its original value: "Hello World!"

Save and exit.

Run the build again. This time the build should be successful, and the various info on the main page for the job should reflect that as well (refresh the page if the Test result trend does not seem to change with the latest build result).

Repeat the creation of errors in `App.java` and the correction of that error across multiple builds. The idea is to simulate what happens in a real life project as code base builds fail and succeed across multiple builds in the life time of the project. You can then click on the various Permalinks to go to the build attempts that failed and were successful, etc....

7 Creating a GitHub repo for the project

Login to your GitHub account.

Create a new repository and give it the same name as the Java project that we have been working with so far: `simple-java-maven-app`

Leave it as a public repository (default setting)


DO NOT initialize the repository with anything. Simply click Create.

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner *

Repository name *

 victor-tan-hk ▾


/

simple-java-maven-app

✓

Great repository names are short, simple, and unique. `simple-java-maven-app` is available. Want to make it more descriptive? How about **verbose-train**?

Description (optional)

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

Open a Git Bash shell in the project folder `simple-java-maven-app` in `github\projects\simple-java-maven-app`

Check to ensure that it is not a proper Git repo yet with

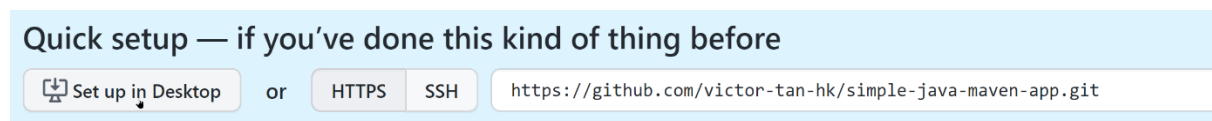
```
git status
```

An error should indicate that this project folder is not yet a Git repo. If it is, you can simply delete the `.git` folder in the project folder.

Next initialize it as a Git repo with all the files in the project folder with the following sequence of command in the Git Bash shell (you can type them in one at a time):

```
git init
git add .
git status
git commit -m "Added all project files in first commit"
git status
```

Return to the main page of your new GitHub repo that you just created. Copy the HTTPS URL shown there:



We will refer to this URL as *remote-url* in the commands to follow.

Back in the Git Bash shell in `simple-java-maven-app`, and type:

```
git remote add origin remote-url
```

Check that the origin handle is set to point to the correct repo URL with:

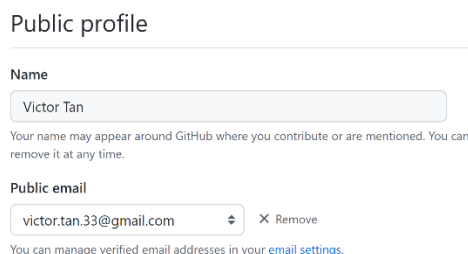
```
git remote --verbose
```

Before pushing this local repo to the empty remote repo, we first need to configure the identity for the local user to be associated with the commits that are going to be pushed to the remote repo.

```
git config --global user.name "GitHub account username"
```

```
git config --global user.email "GitHub account email"
```

The user name and email values for the command above should match the name and email shown in the <https://github.com/settings/profile> of your GitHub account



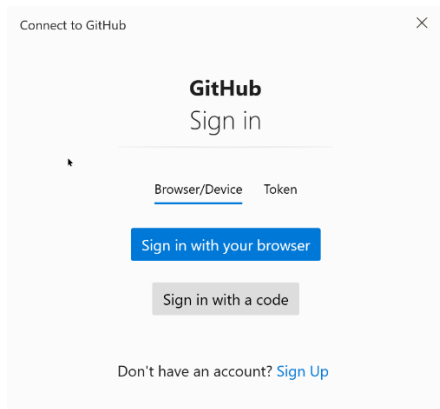
We can also optionally configure auto color scheme for the Git Bash shell

```
git config --global color.ui auto
```

Finally, push the entire contents of the local repo (including all its branches) to the remote repo with:


```
git push -u origin --all
```

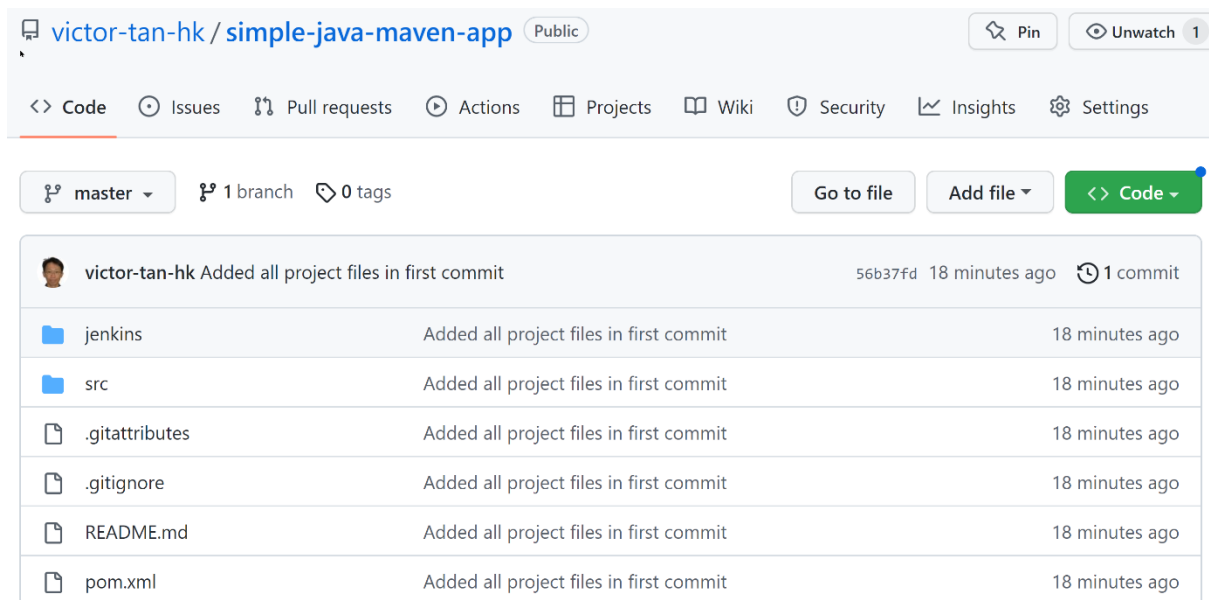
You will be prompted to login:



Select sign in with browser. Assuming you are logged in properly to your GitHub account on the same browser, the command should succeed with appropriate success messages in the Git Bash shell.

```
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
...
branch 'master' set up to track 'origin/master'.
```

Refresh the main repo page, you should see the contents that you have just pushed up.



Back in the Git Bash shell, type the following commands to verify that the remote tracking branches have been set up and that the local and upstream master are both in sync with each other.

```
git remote show origin
```

```
git status
```

Other users (for e.g. team members of a project team) can now clone this remote repo to a local repo on their local machines and work on it via a branching workflow.

8 Checking / installing Git / GitHub plugins for Jenkins

There are a variety of Git related plugins for Jenkins, which should have already been installed during the installation of Jenkins itself if you had selected the Install Suggested Plugins in the installation process itself.

To check, go to the main Jenkins Dashboard, Manage Jenkins -> Manage Plugins -> Installed Plugins and search for the term: git

These are the plugins that should be installed which we will be using in subsequent labs. If they do not show up in your list, then go to Available Plugins and install them in the same way we have done previously.

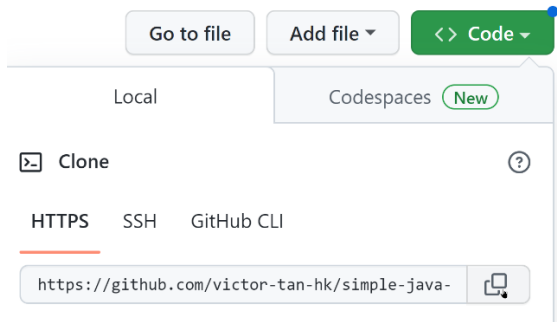
<div> <input type="text" value="git"/> </div>	
Name ↓	Enabled
Git 4.14.3 This plugin integrates Git with Jenkins. Report an issue with this plugin	<input checked="" type="checkbox"/>
Git client 3.13.1 Utility plugin for Git support in Jenkins Report an issue with this plugin	<input checked="" type="checkbox"/>
GitHub 1.36.0 This plugin integrates GitHub to Jenkins. Report an issue with this plugin	<input checked="" type="checkbox"/>
GitHub API Plugin 1.303-400.v35c2d8258028 This plugin provides GitHub API for other plugins. Report an issue with this plugin	<input checked="" type="checkbox"/>
GitHub Branch Source Plugin 1696.v3a_7603564d04 Multibranch projects and organization folders from GitHub. Maintained by CloudBees, Inc. Report an issue with this plugin	<input checked="" type="checkbox"/>
Pipeline: GitHub Groovy Libraries 38.v445716ea_edda_ Allows Pipeline Groovy libraries to be loaded on the fly from GitHub. Report an issue with this plugin	<input checked="" type="checkbox"/>

9 Basic Jenkins job to integrate with GitHub

From the main Jenkins dashboard, create a new Item and name it: `BasicGitHubDemo`
 Make this a Freestyle Project and select Ok.

In the Configure section, go to Source Code Management and select Git

Return to the main page of your GitHub repo, click on the Green Code button and copy the HTTPs URL:



Paste the URL into the Repository URL on the Jenkins configuration page.

Source Code Management

☐ None

☒ Git ?

Repositories ?

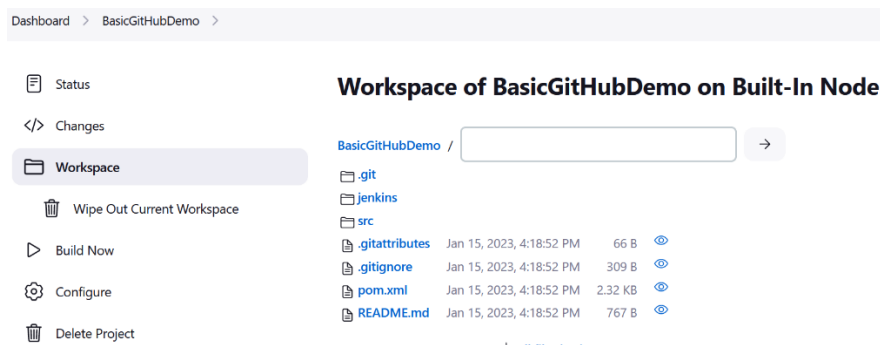
Repository URL ?

https://github.com/victor-tan-hk/simple-java-maven-app.git

Click on Save and perform a build.


The Console Output for this build attempt will show that the remote repo contents have been cloned into the workspace folder for this current job (`workspace\BasicGitHubDemo`).

You can navigate to this folder using File Explorer or use the Workspace item in the navigation pane of the main area for the job to verify this.



In the main area for the job, select Configure, go to Build Steps and add a Build Step -> Invoke Top Level Maven Targets:

Select the Maven Version that you configured previously and add the goals: `clean package`



Click Save, then perform a Build.

The build should be a success, and the Console output shows the output from Maven, with the JAR artifact being generated successfully at the end.

We will now make a change in the local project folder, create a commit from it, push this commit to the remote master and run the build again on our Jenkin server.

Return to the project folder `simple-java-maven-app` in `githubprojects\simple-java-maven-app`

Let's make a minor modification to the source code of the main class for this simple Java project at:

```
src\main\java\com\mycompany\app\App.java
```

using Notepad++ (or any other suitable editor)

Change the parameter of the `System.out.println` to some other random value, for e.g. "Modified from local project !"

Save and exit.

In the Git Bash shell in this project folder, check that the modifications have being made:

```
git status
```

Add the modifications to a new commit with an appropriate message:

```
git commit -am "My first change in the local project folder"
```

And if we check the status again

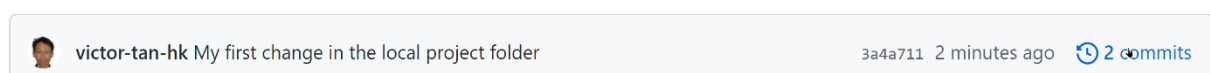
```
git status
```

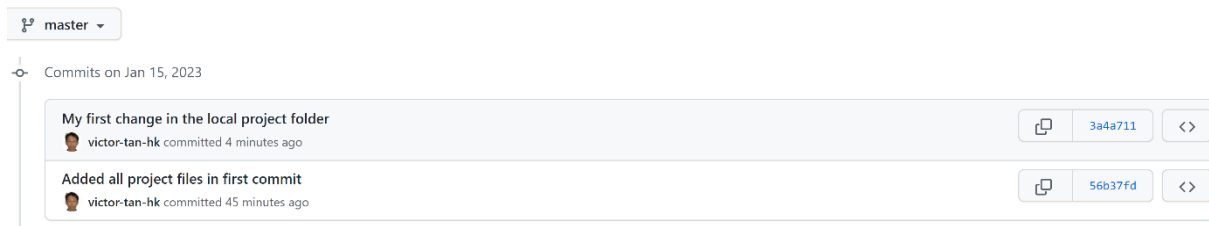
We can see that the latest commit on the master branch is ahead of the remote master by one commit, as we expect.

Push this latest commit to the master on the remote GitHub repo with:

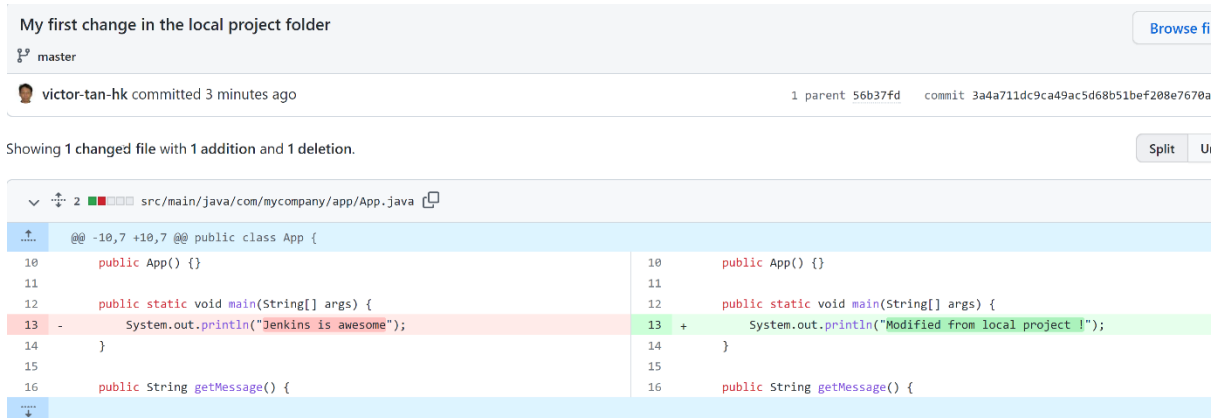
```
git push
```

Back on the GitHub repo main page, click on the Commits link to zoom in to the main commits page





Then click on the hash of the particular commit to see its details.



We can see the latest commit and how it has changed from the previous (initial commit)

Returning back to Jenkins, perform another build of the same job. The latest changes to the remote repo will be pulled down to the local workspace folder, and a build will be run to generate a new build artifact.

To verify that this artifact in fact reflects the latest change, navigate to the appropriate folder (based on the Console Output statements in Jenkin) using File Explorer, open a command prompt in that location and type:

```
java -jar my-app-1.0-SNAPSHOT.jar
```

Returning back to the main area for the job, select Configure, go to Build Steps and add another Build Step -> Execute Windows Batch Command. Type this:

```
echo "*****"
echo "Executing generated JAR"
echo "*****"
```

```
java -jar target/my-app-1.0-SNAPSHOT.jar
```

Click Save and run a build again. Verify from the Console Output that the JAR was executed successfully.

Continue to make some more random changes to the parameter of the System.out.println in the source code of the main class for this simple Java project at:

```
src\main\java\com\mycompany\app\App.java
```

using Notepad++ (or any other suitable editor)

Then commit the changes and push the new commit to the remote repo:

```
git commit -am "My XXXXX change in the local project folder"
```

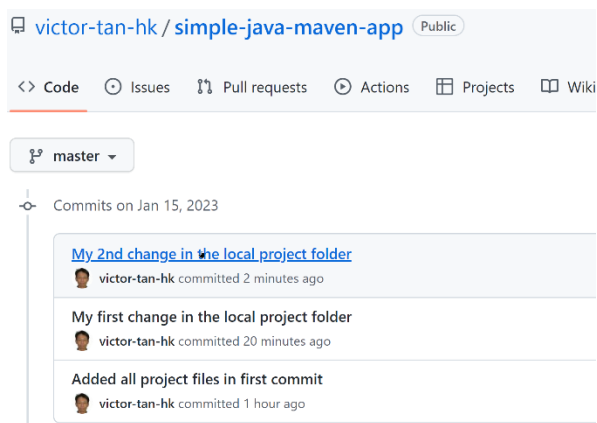
```
git push
```

Then run a build again on the same job in Jenkins and verify from the Console Output that the JAR execution shows that the artifact was built from the latest changes pulled down from the remote GitHub repo.

At any point in time, you can check the commit history of the master branch in the local project folder with:

```
git log --oneline
```

And this history is also visible from the main commits page on the GitHub repo.



In the main area for the job, select Configure, go to Post-Build Actions, and add an action -> Publish JUnit test result report

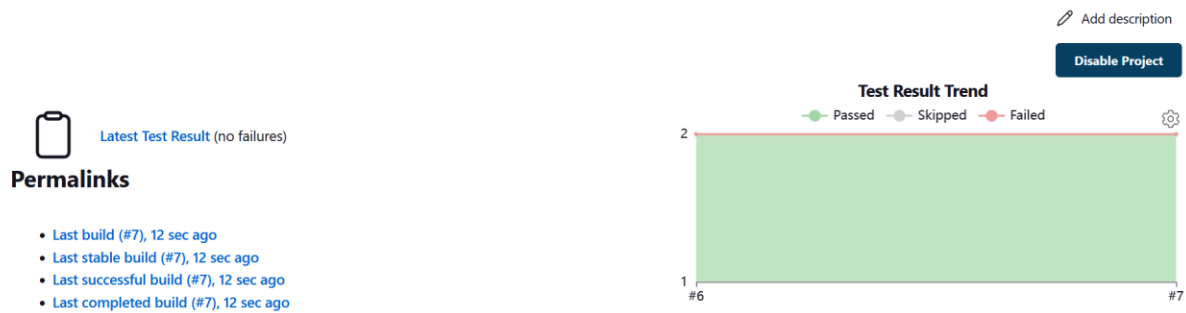
Add in this for the file set: `target/surefire-reports/*.xml`

This is the location for where the test report XMLs are located relative to the workspace root (BasicGitHubDemo).

Click Save and perform a Build again.

Click Changes and Status on the main area for the job, or else simply refresh the page. A test result trend should now appear, keeping track of the test results.

Project BasicGitHubDemo



Return back to the original project location at:

```
jenkinslab\githubprojects\simple-java-maven-app
```

Let's simulate an error occurring in the unit tests. In the file:

```
src\main\java\com\mycompany\app\App.java
```

Change the value of String MESSAGE to some other value, for e.g. "Dummy String"

Save and exit.

As usual commit the changes and push the new commit to the remote repo:

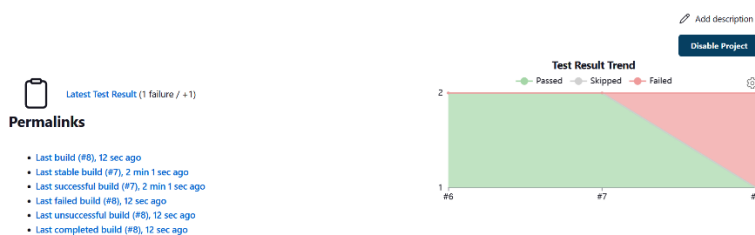
```
git commit -am "Simulating a stupid error !"
```

```
git push
```

Run the build again on Jenkins. This time you should see the error reported in the Console Output for that particular build attempt.

The main area for the job should now indicate the build failure in the Test result trend and also provide links to the various builds with various statuses (stable, successful, failed, unsuccessful, etc)

Project BasicGitHubDemo



Return back to the app source code file:

```
src\main\java\com\mycompany\app\App.java
```

Restore the value of String MESSAGE to its original value: "Hello World!"

Save and exit.

As usual commit the changes and push the new commit to the remote repo:

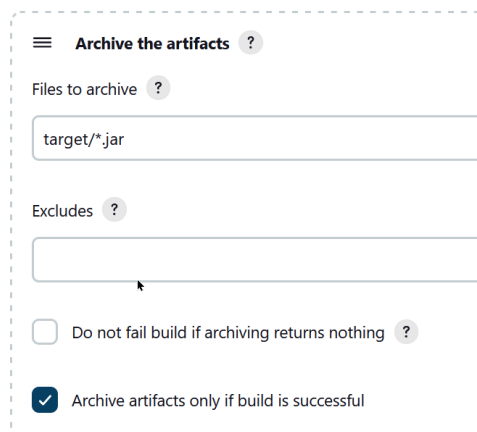
```
git commit -am "Corrected my stupid error !"
```

```
git push
```

Run the build again. This time the build should be successful, and the various info on the main page for the job should reflect that as well (refresh the page if the Test result trend does not seem to change with the latest build result).

In the main area for the job, select Configure, add another Post-build action -> Archive the artifacts. Provide the path of the files to archive relative to the main workspace folder: `target/*.jar`. We can also specify advanced options such as only archiving artifacts if the build is successful.

Post-build Actions



The screenshot shows the 'Archive the artifacts' configuration in Jenkins. It includes a section for 'Files to archive' with a text input field containing 'target/*.jar'. Below this is an 'Excludes' section with an empty text input field. At the bottom, there are two checkboxes: 'Do not fail build if archiving returns nothing' (unchecked) and 'Archive artifacts only if build is successful' (checked).

Click Save and run a build again.

If you now refresh the main page for the job, you should see a link for you to download the artifact.



This is very useful if the Jenkins server is running on a different machine, and it allows you to download the final build artifact to your local machine for further use.

10 Configuring periodic builds and polling

At this point of time, we are manually performing our builds by clicking on the Build Now item in the main page for each job. This is not practical or ideal for real life projects, where team members may be pushing new changes to the central repo at variable times throughout the day.

Jenkins includes what is known as build triggers, which allows a build for a job to be triggered automatically when a specific event occurs. The most common situations would be:

- a) Periodically performing the build, for e.g. once every hour, or every 6 hours, or every day
- b) Checking the remote repo periodically for any updates (this is known as polling SCM) and then performing the build if there are any updates since the last poll. This is nearly the same as periodic builds, except that periodic builds will always occurs, regardless of whether there is any updates to the remote repo.
- c) When a specific event occurs (for e.g. a new commit is pushed to some branch on a remote repo, a pull request is opened to merge a feature branch back into the master branch, etc), the remote repo sends a signal (typically a HTTP POST request) to the Jenkins server to trigger the build. This feature is known as a webhook and is available with most Git cloud hosting services (GitHub, BitBucket, etc)

For the first 2 approaches, the scheduling is based on the CRON format from Linux:

<https://www.hostinger.my/tutorials/cron-job>
<https://phoenixnap.com/kb/set-up-cron-job-linux>

The CRON expression to run a job every minute
<https://cronexpressiontogo.com/every-1-minute>

In the main area for the previous job (BasicGitHubDemo), select Configure, go to Build Triggers, and select Build periodically. For the schedule, enter the CRON expression to run a job every minute (as shown above)

Build Triggers

☐ Trigger builds remotely (e.g., from scripts) ?

☐ Build after other projects are built ?

☒ Build periodically ?

Schedule ?

⚠ Do you really mean "every minute" when you say "*****"? Pe

Of course, in a real-life project, we would not run a build every 1 minute (this would consume too much resources on the build server !!), but for the purposes of demo in this workshop, we will set it to 1 minute. You can choose the CRON expression for the suitable interval of your choice in your real-life projects.

Click Save.

Go back to the main page for the job and wait for several minutes. Notice that a build will automatically be triggered every minute - EVEN IF NO updates have being made to the remote repo.

In the main area for the previous job (`BasicGitHubDemo`), select **Configure**, go to **Build Triggers**, and select **Poll SCM**. For the schedule, enter the CRON expression to run a job every minute, the same as previously.

Click **Save**.

Go back to the main page for the job and wait for 3 - 5 minutes. Notice that unlike before, no builds are triggered.

Return back to the original project location at: `githubprojects\simple-java-maven-app`

Make a minor modification to the source code of the main class for this simple Java project at:
`src\main\java\com\mycompany\app\App.java`

Change the parameter of the `System.out.println` to some other random value
Save and exit.

As usual commit the changes and push the new commit to the remote repo:

```
git commit -am "Make a change to demonstrate Poll SCM"
```

```
git push
```

Go back to the main page for the job and wait for a minute. Now a build will be triggered because the polling of the GitHub repo indicates that there has been change in it since the last poll. Now leave it again for 2-3 minutes and notice that no further builds are triggered.

When you are done verifying both these features, you can disable them in the **Build Trigger** section and **Save** the job again. We will enable them later if we need them in subsequent lab sessions.

We will look at the **webhook** feature when we run the Jenkins server on a publicly accessible machine in a later lab, as this feature requires the Git cloud service to contact the Jenkins server directly, which is only possible if the Jenkins Server is accessible on a public IP address.

11 Configuring email notification

Often, it is useful for the project lead / PIC in a software team to be updated every time a build fails. This is particularly important when the team is rushing to meet a product launch deadline or fixing a critical bug.

The latest approach to integrate Jenkins with Gmail is to first generate an App Password from the Google Account associated with the Gmail account that you wish to configure Jenkins with. Before you can generate an App Password, you will first need to configure 2-Step Verification for your Google Account.

<https://support.google.com/mail/answer/185833?hl=en>

These are the screen shots of the App Password generation process:

← App passwords

App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it. [Learn more](#)

You don't have any app passwords.

Select the app and device you want to generate the app password for.

JenkinsService
X

GENERATE

← App passwords

App passwords let you sign in to your Google Account from apps on devices that don't support 2-Step Verification. You'll only need to enter it once so you don't need to remember it. [Learn more](#)

Your app passwords

Name	Created	Last used
JenkinsService	8:42 PM	–

Select the app and device you want to generate the app password for.

Select app
Select device

Then return to the main Jenkins Dashboard, select Manage Jenkins -> Configure System and go to Jenkins Location and set an email address for the System admin. This email address will be used as the sender for all email notifications sent out from Jenkins.

Jenkins Location

Jenkins URL ?

http://3.1.81.98:8080/

System Admin e-mail address ?

mark.learnfast@gmail.com

Finally, scroll down to the Email Notification at the bottom and configure the details for the Email Notification for using the Gmail SMTP settings. You can use a different email (or the same) as the System Admin that you set up earlier.

<https://kinsta.com/blog/gmail-smtp-server/>

The most important point to note in this configuration process is that the password entered in the field is NOT the actual password associated with the email account specified in the User Name, rather it is the App Password that you generated earlier.

E-mail Notification

SMTP server

smtp.gmail.com

Default user e-mail suffix ?

☒ Use SMTP Authentication ?

User Name

mark.learnfast@gmail.com

Password

.....

☒ Use SSL ?

☐ Use TLS

SMTP Port ?

465

Reply-To Address

mark.learnfast@gmail.com

Charset

UTF-8

☒ Test configuration by sending test e-mail

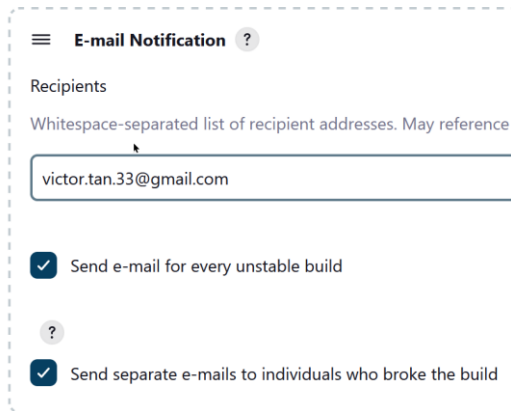
Test e-mail recipient

mark.learnfast@gmail.com

Perform a test configuration by sending test email. The test e-mail recipient can be any valid email address (and not just the username used in SMTP authentication).

If the test is successful, click Save.

In the main area for the previous job (`BasicGitHubDemo`), select **Configure**, and add another Post-build action -> Email notification, enter a suitable email address (this would be the email for the person that is to be notified in the event of build failure) and check appropriate options for when emails will be sent and to whom.



Click Save and manually perform a build.

The build is successful, so no notification email is sent.

Return back to the original project location at:

jenkinslab\githubprojects\simple-java-maven-app

Let's simulate an error occurring in the unit tests. In the file:

src\main\java\com\mycompany\app\App.java

Change the value of String MESSAGE to some other value, for e.g. "Dummy String"

Save and exit.

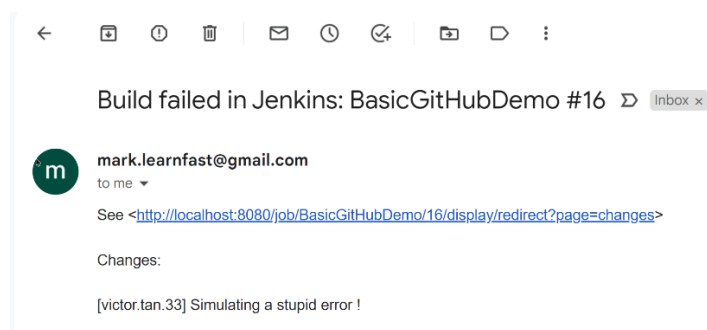
As usual commit the changes and push the new commit to the remote repo:

```
git commit -am "Simulating a stupid error !"
```

```
git push
```

Run the build again on Jenkins. The build will fail. If you check the Console Output, there will be a message sent to the registered user.

Check in that email account to verify that an email has being received from the email registered as the system admin in Jenkins Location, giving the details of the Console Output for that particular build attempt.



In addition to Gmail, there are many other organizations that provide free SMTP servers to cater for this kind of situation. Your company / organization may also have its own internal SMTP server that you can make use of:

<https://moosend.com/blog/free-smtp-server/>
<https://www.emailvondorselection.com/free-smtp-servers/>

In addition to notifications via email, Jenkins provides plugins that support integration with other project management / collaboration tools for the purposes of build notifications (for e.g. Slack and Jira)

<https://plugins.jenkins.io/slack/>
<https://plugins.jenkins.io/jira/>

12 Configuring Tomcat server for deploying a Java web app

Apache Tomcat is a servlet container which is used to run servlet and JavaServer Pages (JSP) web applications. Most modern Java web frameworks are based on servlets, e.g. JavaServer Faces, Struts, Spring. Tomcat can also be used as a web server, although it is typically deployed inside a more conventional and well known web servers such as Apache or Nginx

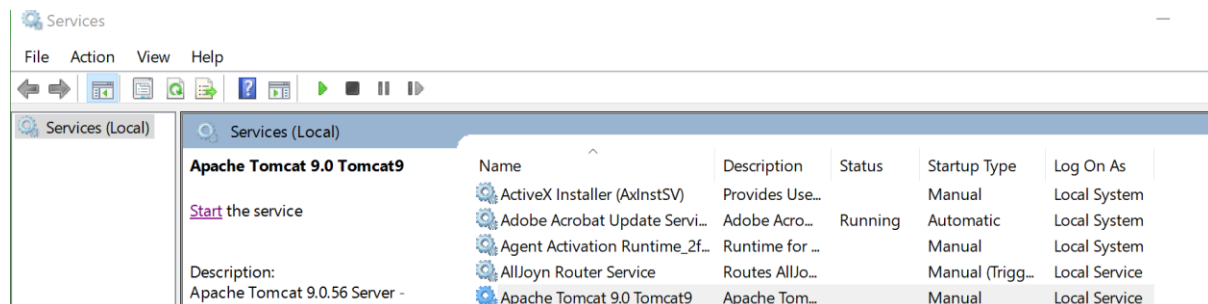
The main issue with Tomcat is that the default port it starts on is also 8080 (same as Jenkins), so we will either need to configure Tomcat or Jenkins with respect to this (either way is fine).

On Windows, the Tomcat installation folder is usually at:

C:\Program Files\Apache Software Foundation\Tomcat xxxx

<https://mkyong.com/tomcat/how-to-change-tomcat-default-port/>
<https://stackoverflow.com/questions/18415578/how-to-change-tomcat-port-number>

If you follow the method above to change the port, you must first stop Tomcat as a service (if it is still running as a service)



and then start Tomcat from the `bin` folder in the root installation folder with:

```
catalina.bat start
```

Otherwise, if you wish to start Tomcat as a service, it will always run on port 8080, and then you should change the default port for the Jenkins server as demonstrated in an earlier lab session.

Access the main Tomcat landing page at the port number that you have chosen:

<http://localhost:xxxx/>

Next, we will configure access to Tomcat Manager in order to allow us to deploy and view web apps on Tomcat properly.

If you have started Tomcat as a service, shut it down as a service.
Otherwise, shutdown Tomcat from the `bin` folder in the root installation folder with:

```
catalina.bat stop
```

Guide on how to setting up access for users to access Tomcat Manager

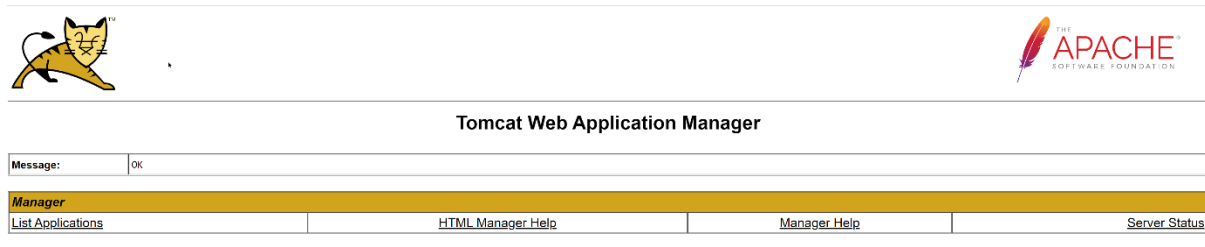
<https://clients.javapipe.com/knowledgebase/129/How-to-access-Tomcat-Manager.html>

Modify `/conf/tomcat-users.xml` and add in the following roles and users with the respective username and passwords:

```
<role rolename="manager-gui"/>
<role rolename="manager-script"/>
<role rolename="manager-jmx"/>
<role rolename="manager-status"/>
<user username="admin" password="admin" roles="manager-gui,
manager-script, manager-jmx, manager-status"/>
<user username="deployer" password="deployer" roles="manager-
script"/>
<user username="user" password="user" roles="manager-gui"/>
```

Start Tomcat again as a service or by typing `catalina.bat start` into the `bin` folder in the root installation folder

Attempt to access the manager app using the username password combination of either `admin/admin` or `user/user` (as defined previously).



13 Building and deploying a Java web app

Copy `basic-java-web-app` from the `labcode` of your downloaded zip into the `githubprojects` folder

Open a command prompt in the root project folder and perform a clean (to remove all files from the previous build) and then generate the build artifact with:

```
mvn clean package
```

This will again download some new dependencies and plugins as specified in the project POM.

The final part of messages from the Maven build console output shows the generation of the WAR artifact (`webapp\target\webapp.war`). This WAR artifact contains the Java web app (in the form of a servlet) which is now ready to be deployed into Tomcat.

There are two widely used ways to deploy this artifact (there are a few other more lesser known approaches as well)

The first way is via the Tomcat Manager UI. Choose the WAR file to deploy, then navigate to the folder containing `webapp.war` in the file dialog box and then select it (you can copy `webapp.war` to your Desktop first to make it a bit faster to access)

WAR file to deploy

Select WAR file to upload Choose File No file chosen
Deploy

Once deployed, you should see its path in the list of applications.

/webapp	None specified	Webapp	true	0	<div> Start Stop Reload Undeploy </div> <div> Expire sessions with idle ≥ <input type="text" value="30"/> minutes </div>
-------------------------	----------------	--------	------	---	---

Click on the path to navigate to the app

Welcome to the coolest Tomcat webapp ever !

Interesting things to do for today

- Learn JavaScript
- Learn Python
- Learn Java

Go back to the Manager UI and Stop and Undeploy this app

Start Stop Reload Undeploy

Expire sessions with idle ≥ minutes

It should now disappear from the list of applications on Tomcat Manager UI.

The second way is to simply copy the WAR file into the `webapps` folder of the main Tomcat installation directory.

After a while, the WAR file will be unpacked and you should again be able to see the name of the app listed in the list of applications and you should be able to navigate to it by clicking on the link as you did previously.

Go back to the Manager UI and Stop and Undeploy this app

Start Stop Reload Undeploy

Expire sessions with idle ≥ minutes

14 Creating a GitHub repo for the Java web app project

Login to your GitHub account.

Create a new repository and give it the same name as the Java project that we have been working with so far: `basic-java-web-app`

Leave it as a public repository (default setting)

DO NOT initialize the repository with anything. Simply click Create.

Open a Git Bash shell in the project folder `basic-java-web-app` in `githubprojects`

Check to ensure that it is not a proper Git repo yet with

```
git status
```

An error should indicate that this project folder is not yet a Git repo. If it is, you can simply delete the `.git` folder in the project folder.

Next initialize it as a Git repo with all the files in the project folder with the following sequence of command in the Git Bash shell (you can type them in one at a time):

```
git init
git add .
git status
git commit -m "Added all project files in first commit"
git status
```

Return to the main page of your new GitHub repo that you just created. Copy the HTTPS URL shown there:

Quick setup — if you've done this kind of thing before



Set up in Desktop

or

HTTPS

SSH

<https://github.com/victor-tan-hk/basic-java-web-app.git>

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#).

We will refer to this URL as `remote-url` in the commands to follow.

Back in the Git Bash shell in `basic-java-web-app`, and type:

```
git remote add origin remote-url
```

Check that the origin handle is set to point to the correct repo URL with:

```
git remote --verbose
```

Before pushing this local repo to the empty remote repo, we first need to configure the identity for the local user to be associated with the commits that are going to be pushed to the remote repo.

```
git config --global user.name "GitHub account username"
```

```
git config --global user.email "GitHub account email"
```

The user name and email values for the command above should match the name and email shown in the <https://github.com/settings/profile> of your GitHub account

Public profile

Name

Victor Tan

Your name may appear around GitHub where you contribute or are mentioned. You can remove it at any time.

Public email

victor.tan.33@gmail.com

✕ Remove

You can manage verified email addresses in your [email settings](#).

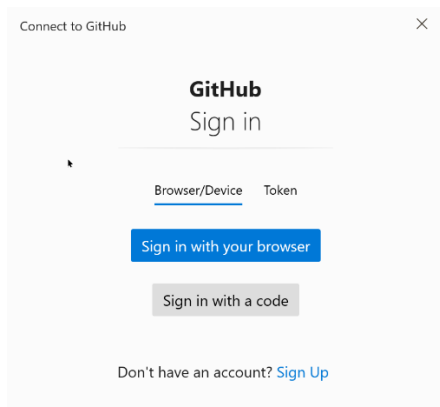
We can also optionally configure auto color scheme for the Git Bash shell

```
git config --global color.ui auto
```

Finally, push the entire contents of the local repo (including all its branches) to the remote repo with:

```
git push -u origin --all
```

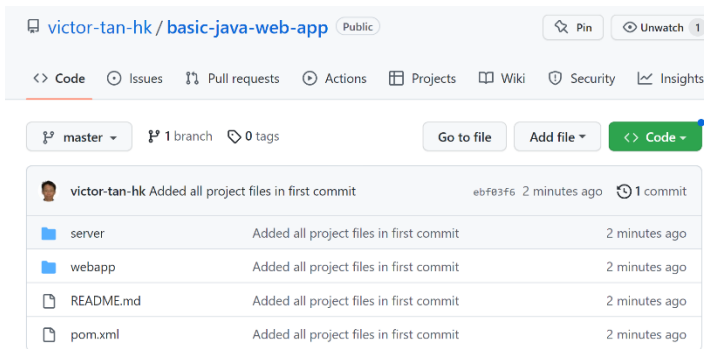
You will be prompted to login:



Select sign in with browser. Assuming you are logged in properly to your GitHub account on the same browser, the command should succeed with appropriate success messages in the Git Bash shell.

```
Enumerating objects: 23, done.
Counting objects: 100% (23/23), done.
...
...
branch 'master' set up to track 'origin/master'.
```

Refresh the main repo page, you should see the contents that you have just pushed up.



Back in the Git Bash shell, type the following commands to verify that the remote tracking branches have been set up and that the local and upstream master are both in sync with each other.

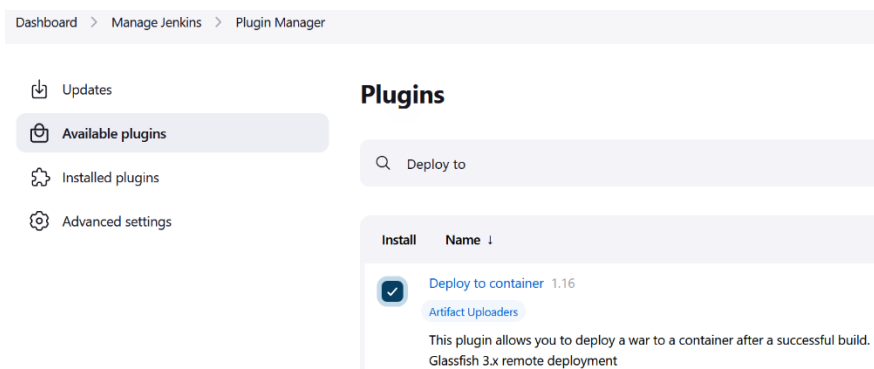
```
git remote show origin
```

```
git status
```

Other users (for e.g. team members of a project team) can now clone this remote repo to a local repo on their local machines and work on it via a branching workflow.

15 Set up credentials and plugin to integrate with Tomcat container

In Jenkins main dashboard, go to Manage Jenkins -> Manage Plugins, and in Available Plugins search for Deploy to Container



Select Install without restart.

After the Download progress is complete, select Restart Jenkins when installation is complete and no jobs are running

In Jenkins main dashboard, go to Manage Jenkins - Manage Credentials -> System (or Jenkins) -> Global Credentials (Unrestricted), and then Add Credentials.

d > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

Global credentials (unrestricted)

+ Add Credentials

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind	Description
This credential domain is empty. How about adding some credentials ?			

Here we intend to use the user `deployer` (password the same as well) with the `manager-script` role that we had previously configured in `tomcat/conf/tomcat-users.xml`, so we will configure that username / password combination here

New credentials

Kind

Username with password

Scope ?

Global (Jenkins, nodes, items, all child items, etc)

Username ?

deployer

☐ Treat username as secret ?

Password ?

••••••••

ID ?

tomcat_deployer

Description ?

Click Create.

Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

Global credentials (unrestricted)

+ Add Credentials

Credentials that should be available irrespective of domain specification to requirements matching.

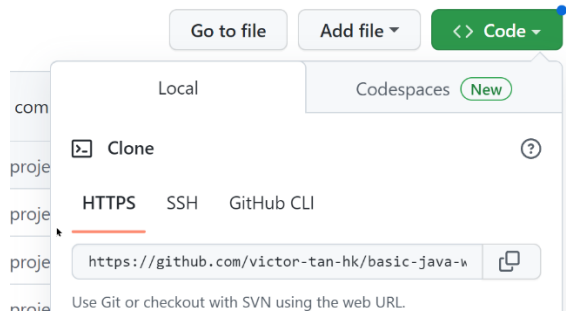
ID	Name	Kind	Description
tomcat_deployer	deployer/***** (tomcat_deployer)	Username with password	tomcat_deployer

16 Basic Jenkins job to build and deploy a Java Web app

From the main Jenkins dashboard, create a new Item and name it: `JavaWebAppDemo`
Make this a Freestyle Project and select Ok.

In the Configure section, go to Source Code Management and select Git

Return to the main page of your GitHub repo, click on the Green Code button and copy the HTTPs URL:



Paste the URL into the Repository URL on the Jenkins configuration page.



Click on Save and perform a build.

The Console Output for this build attempt will show that the remote repo contents have been cloned into the workspace folder for this current job (`workspace\JavaWebAppDemo`). You can navigate to this folder using File Explorer or use the Workspace item in the navigation pane of the main area for the job to verify this.

In the main area for the job, select Configure, go to Build Steps and add a Build Step -> Invoke Top Level Maven Targets:

Select the Maven Version that you configured previously and add the goals: `clean package`



Click Save, then perform a Build. This will take a while to complete due to the downloading of all the Maven dependencies and plugins again.

The build should be a success, and the Console output shows the output from Maven, with the WAR artifact being generated successfully at the end.

In the main area for the job, select Configure, go to Post-build action, and add Choose deploy war/ear to container.

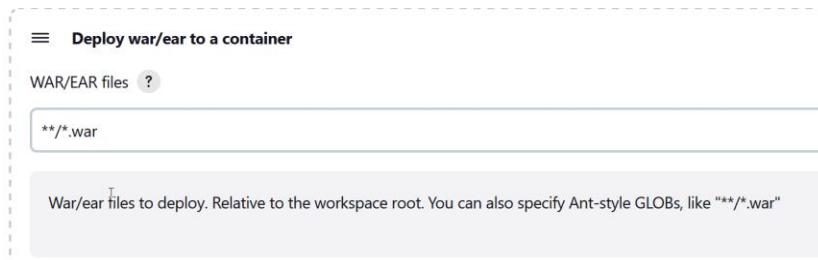
The actual generated WAR file in the workspace folder would be at: `webapp/target/webapp.war`

We only need to specify the path relative from the workspace root folder, i.e.

`webapp/target/webapp.war`

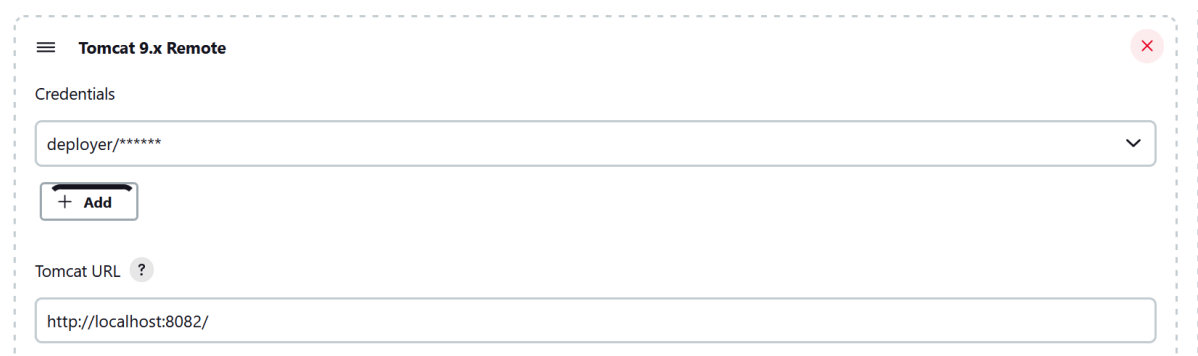
We can also use an Ant-style GLOB to specify the location of the WAR file: `**/*.war`

<https://www.bojankomazec.com/2019/07/apache-ant-patterns.html>



For containers, select Tomcat 9.

Select the credentials you added earlier and provide the URL to the Tomcat server (make sure you have the correct port number).



Click Save and run the build. Make sure your Tomcat server is up and running before this.

If you return to your Tomcat manager, you should see that the web app is now deployed successfully in the list of applications in the UI, and you should be able to click on the link to view it.

Return to the project folder `basic-java-web-app` in `githubprojects`

Let's make a minor modification to the HTML for the main page of the web app at:

`webapp\src\main\webapp\index.jsp`

using Notepad++ (or any other suitable editor)

Change the content of the `<h1>` header to some other random value, for e.g.

`<h1>Welcome to the most boring Tomcat webapp ever ... boo </h1>`

Save and exit.

In the Git Bash shell in this project folder, check that the modifications have being made:

```
git status
```

Add the modifications to a new commit with an appropriate message:

```
git commit -am "My first change in the local project folder"
```

And if we check the status again

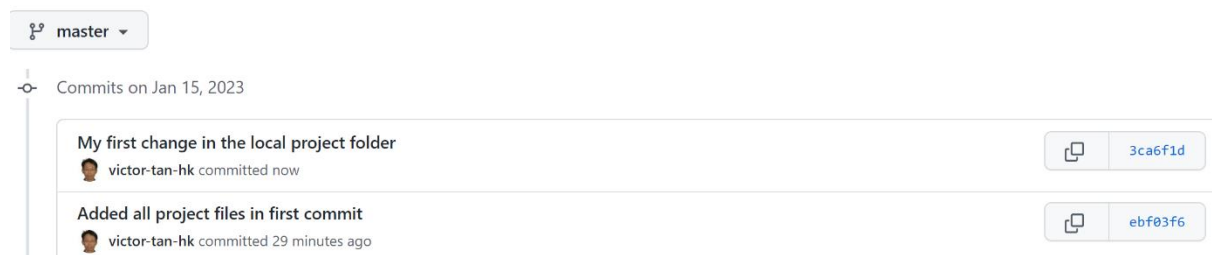
```
git status
```

We can see that the latest commit on the master branch is ahead of the remote master by one commit, as we expect.

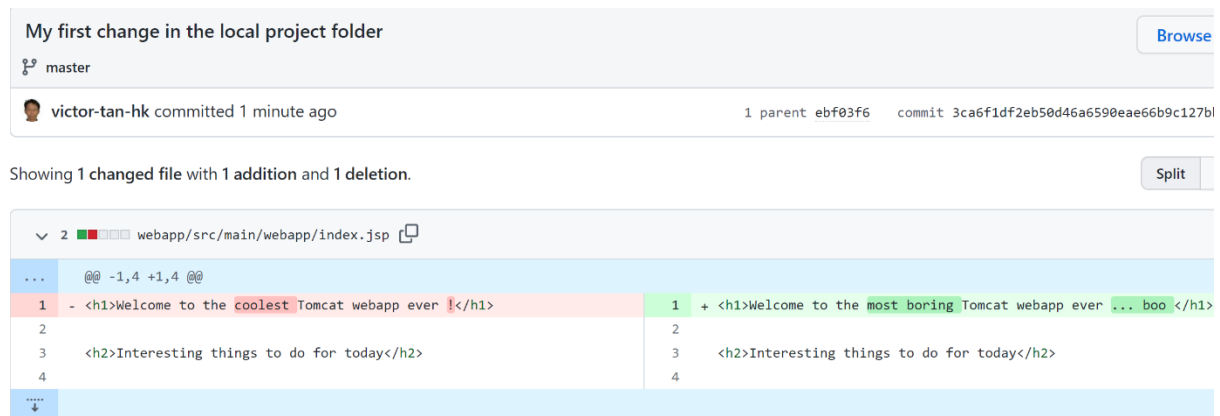
Push this latest commit to the master on the remote GitHub repo with:

```
git push
```

Back on the GitHub repo main page, click on the Commits link to zoom in to the main commits page



Then click on the hash of the particular commit to see its details.



We can see the latest commit and how it has changed from the previous (initial commit)

Returning back to Jenkins, perform another build of the same job. The latest changes to the remote repo will pulled down to the local workspace folder, and a build will be run to generate a new build artifact. This will again be deployed in the Tomcat server, and you should be able to see the latest changes once you navigate to the app again.

Continue to make a few more changes on the local code base and push the commits to update the remote repo and run a build again.

As a matter of practice, you can configure tests, periodic builds and polling and email notification for this job as well, the same way that you have done for the previous basic Java Maven job.