Jenkins
Lab 1

1   Lab setup

You should have been given a lab guide detailing the various configuration details for AWS EC2 instances that you will be using for the duration of this workshop.

Verify that you can access the main Jenkins UI at your designated server:

http://server-ip-address:port-number

Login with the provided username / passwd combination at Jenkins UI main page. After successful login, the main UI should look something like this:
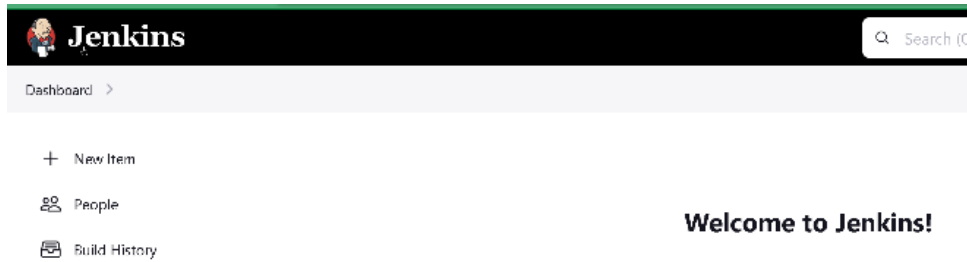
In all the labs that follow, `userx` refers to your assigned username in the lab guide.

## 2 Basic Jenkins freestyle project with Shell commands

A Jenkins job (sometimes called project) is a user-configured description of work which Jenkins should perform, which is usually building a piece of software. Each job consists of various stages, which in turn contain a group of steps, where a step is a single fundamental task that tells Jenkins what to do. Typical examples of these tasks include gathering dependencies, compiling, archiving, or transforming code, and testing and deploying code in different environments.

Jenkins supports several types of jobs: such as freestyle projects, pipelines, multi-configuration projects, folders, multibranch pipelines, and organization folders. The two most commonly used jobs which we will cover extensively in this workshop are freestyle projects and pipelines. The other kinds of jobs are specializations of pipelines used for very specific use case scenarios.

From the main Jenkins dashboard, create a new Item and name it: `userx-Basic-Shell-Job` Make this a Freestyle Project and select Ok.

The configuration for this job is divided into several different stages / sections, which are also found in other projects. Each stage incorporates specific actions / steps to be performed. This step functionality is provided from the plugins that already installed on the main Jenkins controller, and any additional specialized functionality required will be obtained by adding more appropriate plugins.
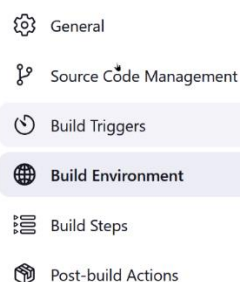
Type some random info in the Description for this project in the General section.

In the Build Steps section, choose Add Build step -> Execute Shell commands, and add in the following Linux shell commands:


```
echo "Current active account is $(whoami)"
echo 'Creating a new directory if it does not exist and a new file'
mkdir -p awesome
cd awesome
NAME=Richard
echo "Hello, $NAME. The date is $(date)" > newfile
ls -l
pwd
echo "The content of newfile is "
cat newfile
```

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.
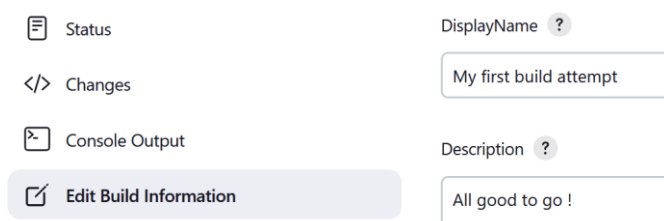
Status

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.

Some points to note:

- The name of the logged in user (`Jenkins Main Administrator`) is shown
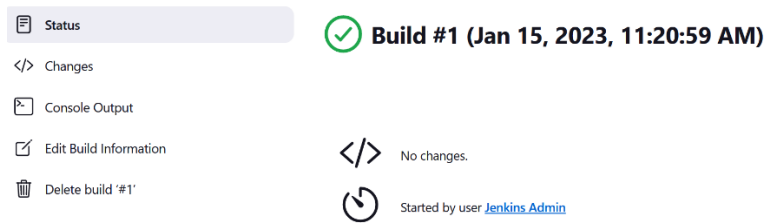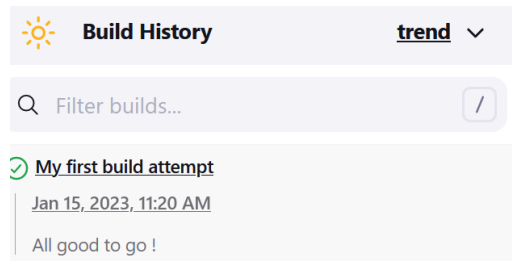- The rest of the listing is the console output corresponding to the execution of the various shell commands. You can see that the active user account that is running the Jenkins main controller service is called `jenkins`, which is system user account (this is different from normal user accounts).
- The complete path of the workspace folder is shown. The default installation directory for Jenkins on a Linux machine is `/var/lib/jenkins`. Within this installation directory is a folder called `workspace`, and within `workspace` are all the workspace folders corresponding to all the jobs / items that you create in Jenkins. In this case, the complete path to the current workspace would be will be: `/var/lib/jenkins/workspace/user1-Basic-Shell-Job`. The final folder name will be the same as the job name, and all items / files related to that job will be placed within this workspace folder.
- All build-related activity for that job (creation of new directories, files, etc) will take place relative to the workspace folder for that job by default, unless the full path for another location is explicitly specified in one of the steps / tasks of the job.
- If you have any commands that manipulate directories or files, these directories or files will be relative to the workspace folder of that job. So, for e.g. if the `newfile` file that was created by one of the shell commands is within the `awesome` directory, which itself is within the top-level workspace folder.
- If you wish to access locations or files outside of the workspace folder, then remember to specify an absolute path: e.g. `/home/user1`
- However, remember that the Jenkins service is running as the `jenkins` user, so if you execute a shell command that attempt to access a directory / file / process that this user does not have permissions to access, an access error will occurs just as in the case for a standard Linux command.

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 ?

My first build attempt
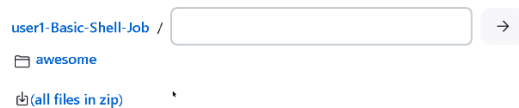
Description ?

All good to go !

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 ∨

Filter builds… /

⊘ **My first build attempt**
Jan 15, 2023, 11:20 AM
All good to go !
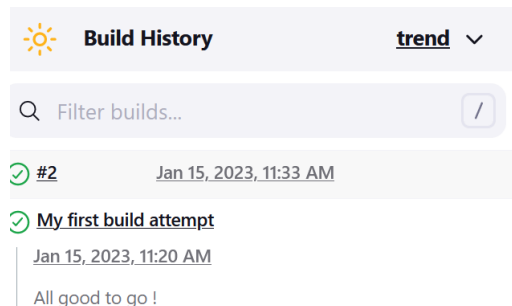
If you click on Workspace, you will be able to get a UI that allows you to navigate the workspace folder structure (which was created as part of the build process) as well as download the entire folder to our local machines in a zip format. This would be useful here since the Jenkins service will typically be running on a remote server in a real life scenario. Go ahead and try that here.

Workspace of user1-Basic-Shell-Job on Built-In Node

user1-Basic-Shell-Job / →

📁 awesome

⊕ (all files in zip)

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

Build History trend ∨

Filter builds… /

⊘ **#2**   Jan 15, 2023, 11:33 AM
⊘ **My first build attempt**
Jan 15, 2023, 11:20 AM
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). For build status concepts:
https://www.jenkins.io/doc/book/glossary/#build-status

So far for our very simple job, all the builds fall into this category.

Project user1-Basic-Shell-Job

Using basic Linux shell commands in a simple job

Permalinks

- Last build (#4), 4.9 sec ago
- Last stable build (#4), 4.9 sec ago
- Last successful build (#4), 4.9 sec ago
- Last completed build (#4), 4.9 sec ago

If you now return to the main Dashboard, you will be able to see all the jobs that have been created on the Jenkins server by all logged in users up to this point of time (this means all the other participants in your assigned group, since they are all also working on the same Jenkins server).

S	W	Name ↓	Last Success	
⊘	☀	user1-Basic-Shell-Job	13 min	#4
⊘	☀	user2-Basic-Shell-Job	9.1 sec	#1

The various icons next to the job listing demonstrate the status of that job, and you can get more detailed info by just hovering your mouse over the particular icon.

As the lab continues to progress the list of jobs available for inspection will become very long and it may be cumbersome to identify the jobs that you created. We can impose a filter on the list of jobs in the dashboard to simplify this.

3   Adding a filter to view the jobs in the dashboard

Select New View.

All   +⟊
      New View
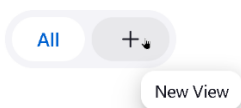
Give it the name: `view-for-userx`
And select the Type as List View.

New view

Name

view-for-user1

Type

● List View

Shows items in a simple list format. You can choose which jobs are to be displayed in which view.

Select Create.

In the Job Filters section, select Use a regular expression to include jobs into the view
And specify the regular expression: `userx-.*`

☑ Use a regular expression to include jobs into the view ?

Regular expression

user1-.*

Add Job Filter ▾

Click Ok.

You should now only be able to see jobs that start with `userx-` in the main Dashboard.

| All | view-for-user1 | + |

S	W	Name ↓
⊘	☀	user1-Basic-Shell-Job

You can always click on All to return back to view all available jobs in the main Dashboard.

4   Basic Jenkins freestyle project with Windows batch commands

This is an optional exercise if you have managed to install Jenkins on your Windows machine successfully. Typically, it is installed on Windows as a service and will start on the default port of 8080. You can access it via the Services app.
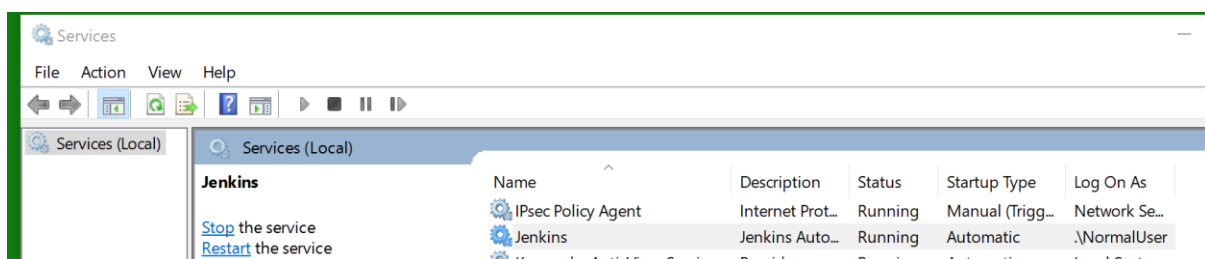
If Jenkins service is already running, you can open a browser tab to
http://localhost:8080
where you will see the main login page for the Jenkins UI, and you can login using the admin username / password combination that you registered during the installation process.

Repeat the process of creating a job that we did earlier, but this time in the Build Steps section, choose Add Build step -> Execute Windows Batch command

Build Steps

≡ **Execute Windows batch command** ?

Command

See the list of available environment variables
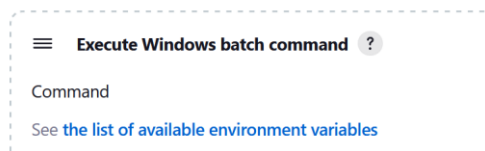
Now, see what happens if you attempt to reuse the list of shell commands that we entered previously in the lab on Linux shell command in the area for Windows batch command. Save and run a build on this job.

This time you will get an error for the build

☀ **Build History**   trend ∨

🔍 Filter builds...   /

⊗ #1   Feb 22, 2023, 5:25 PM
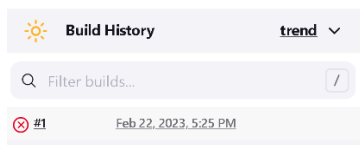
And if you check the Console Output for this particular build, you will notice that errors are due to certain Linux shell commands not being recognized in Windows (`ls, pwd, cat`)

Now Configure this job again with proper DOS commands as shown below:

```
echo "Hi there from Jenkins"
echo "I love Jenkins ! Its the most awesome CI tool ever ... " >
hello.txt
echo "Showing contents of current job folder"
dir
type hello.txt
```

Save and build again. This time the build succeeds, because we are using DOS commands which can be interpreted and executed correctly by the Windows machine that the Jenkins server is running on. You can the Console Output to verify the actions that occur.

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

Now if you repeat the process of taking these DOS commands and attempting to use them in the Execute Shell command section of the job that we created earlier on the Jenkins server running on a Linux machine, we will get similar errors for the same reason (DOS commands are not recognized and cannot be interpreted by the Linux Bash shell).

This demonstrates that there are certain aspects of a job configuration that are platform specific, although most aspects are platform independent. Later on, we will see when we are working with pipelines on how to execute generic file / directory operations in a general way through a Jenkins job regardless of the platform that Jenkins is running on.

5   Connecting to the main server via SSH

We will establish a Bash shell to an existing user account (`mainuser`) on the Main Server so that we can work with some sample Java projects that have already been placed there.

First open Windows Powershell (`powershell`) or Powershell 7 (`pwsh`)

To connect to Bash Shell in Main Server / Secondary Server, execute in either shell:

```
ssh mainuser@server-IP
```

Type Yes to any prompts related to `continue connecting (yes/no/[fingerprint])?`

Enter password: `mainuser`

When you are logged in you should see a Bash prompt similar to:
`mainuser@main-server`

You will type all your Linux shell commands into this prompt. Type:

```
ls -l
```

You should be able to see 15 subdirectories (all under the ownership of the current user mainuser) with names corresponding to your assigned usernames.

You will establish a connection to the Main Server using the MobaXTerm client so that you can easily navigate and view the contents of the folders corresponding to your assigned usernames, and also transfer files between your local machine and the Main Server.

Configure a new SSH connection with the IP address of the Main Server and port 22 (default port for SSH).
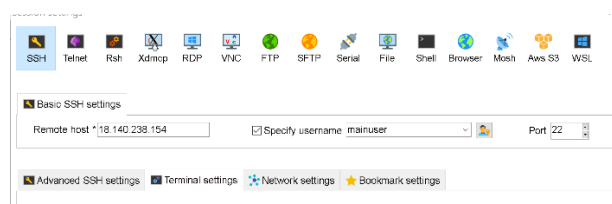
You can set terminal font settings to get a larger font when you login,
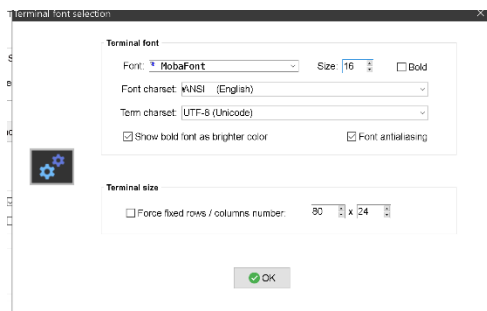
Click Ok to connect and type the password when prompted. Select Yes to the prompt to save the password.
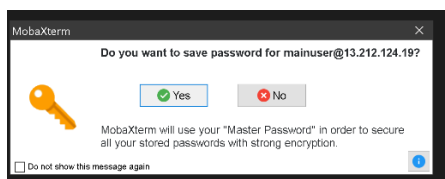

With the connection established, you will have a navigation pane on your left. You can navigate through the contents of the folder of your assigned username - similar to the case of Windows Explorer.

Navigate with care and be careful not to accidentally delete any of the files and directories of other user's folders.

6 Introducing Maven for building Java projects

Different programming languages / frameworks utilize different build tools. The most popular build tool for most modern day Java projects is Maven, which is an evolution from an earlier build tool, Ant. https://maven.apache.org/

In the folders corresponding to your usernames, there are two folders: `bitbucketprojects` and `githubprojects`. Inside `githubprojects` there are two more folders, which are the root folders of a Maven Java project.

We will work with Maven using the first project: `simple-java-app`.

In the Bash shell, navigate into this folder by typing:

```
cd ~/userx/githubprojects/simple-java-app
```

First, you can check the version of Maven installed on this machine with:

```
mvn -version
```

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.

```
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 through MobaXTerm client.

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 execute the artifact directly with:

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

This single line output is from the source code of the main class for this simple Java project at:

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

You can use MobaXTerm's default editor to open it for editing:
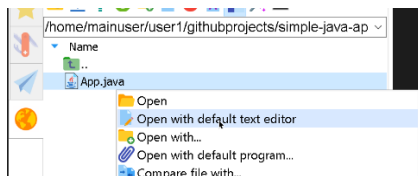
Let's make a minor modification to the source code. Change the parameter of the System.out.println to some other random value, for e.g. `"Jenkins is awesome"`
Save the changes (Ctrl+S) and select Yes to the prompt that appears.
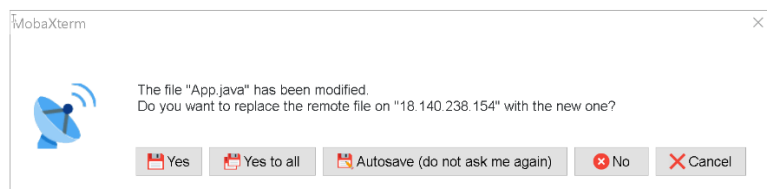
You can verify that the changes have actually been made by directly viewing the content of this file from the CLI:

```
cat src/main/java/com/mycompany/app/App.java
```

In the shell, we can again repeat the clean (to remove all files from the previous build) and then generate the build artifact again with:

```
mvn clean package
```

Now we can execute the newly generated build artifact again with:

```
java -jar target/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, lets 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("I love Jenkins", app.getMessage());
```
to fail.

In the command prompt, perform a clean (to remove all files from the previous build) and then generate the build artifact with:

```
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: `" I love Jenkins"`
Save and exit.

Now repeat the Maven goals again:

```
mvn clean package
```

This time the build process completes without error, and we obtain the newly generated build artifact and can execute it again with :

```
java -jar target/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 is the package manager for C# .NET Core projects
https://learn.microsoft.com/en-us/nuget/what-is-nuget
https://www.syncfusion.com/blogs/post/how-to-use-nuget-packages.aspx

MSBuild is the build tool for .NET C# applications
https://markheath.net/post/getting-started-with-msbuild
https://learn.microsoft.com/en-us/visualstudio/msbuild/walkthrough-using-msbuild?view=vs-2022

MSTest is one of the testing frameworks for .NET C# applications
https://learn.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-mstest
https://www.lambdatest.com/blog/most-complete-mstest-framework-tutorial-using-net-core-2/

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/
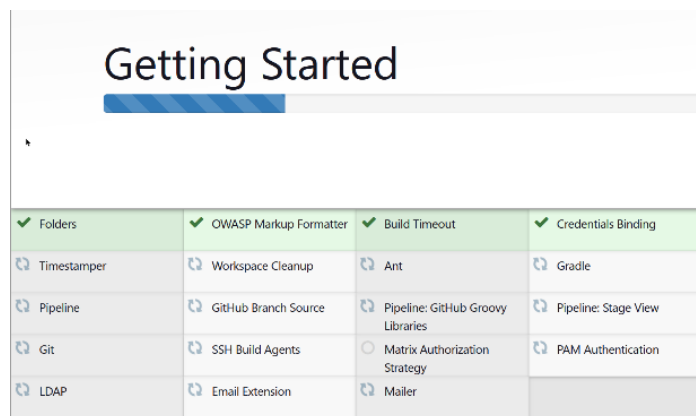
7   Install and configure Maven plugin

Plugins are key to the functionality of Jenkins. When a plugin is installed, it provides functionality to Jenkins by adding available steps that can be used in the actions / options to be selected in the key stages of a standard free style project (General, SCM, Build Triggers, Build Environment, Build Steps, Post Build Actions). These steps / options represent a fundamental unit of action that is performed by Jenkins.

There are a set of suggested plugins which are typically installed during initial installation of Jenkins,

In addition there is a large ecosystem of plugins accessible from the Update Center which can then be further installed to add additional functionality

https://plugins.jenkins.io/

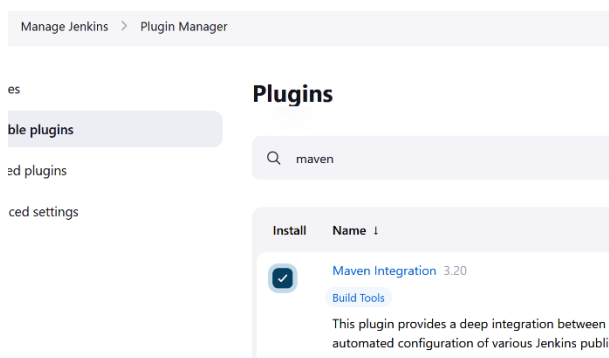You can perform a filter on the plugins based on various categories from the main search page:
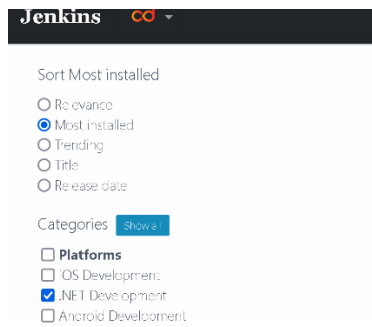
Notice that you have plugins to implement the functionality for the various build tools related to .NET C# application development: MSBuild, MSTest, Nuget, etc

The same goes for python: do a search for this term.

The action below of installing a plugin only needs to be performed once by a user with Admin Privilege.

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,

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.

The restart may take some time to complete or may occasionally fail. In Linux, you can start / stop the Jenkins service from the CLI via:

```
sudo systemctl start / stop jenkins
```

You can check the status of the Jenkins service using the command:

```
sudo systemctl status jenkins
```

You may have to log back in again once the restart is complete.

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.

Maven installations
List of Maven installations on this system

Add Maven

≡ Maven

Name

jenkins-maven

☑ Install automatically ?

≡ Install from Apache

Version

3.9.0

Add Installer ▼

8 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-app`

Leave it as a public repository (default setting)

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

Back in the Bash shell, make sure you are in the folder `userx/githubprojects/simple-java-app`

If not, you can change to it with:

```
cd ~/userx/githubprojects/simple-java-app
```

First, we remove any previously existing generated artifacts

```
mvn clean
```

Git is already installed on the server. First, we check that this folder 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
```

Before creating the first commit in this newly initialized repository, you need provide an identify for the individual creating the commit (the commit author). Since we are going to be pushing our local repo to populate the GitHub repo, we will use our GitHub identity (name and email). Go to

https://github.com/settings/profile

and note down the name and email address there. Use them in the configuration commands below:

```
git config --local user.name "your name"

git config --local user.email "your email address"
```

We are configuring these details at the local level so that they apply only to this repository, rather than to the user account since all users are now using this user account `mainuser`.

Then we can create our first commit.

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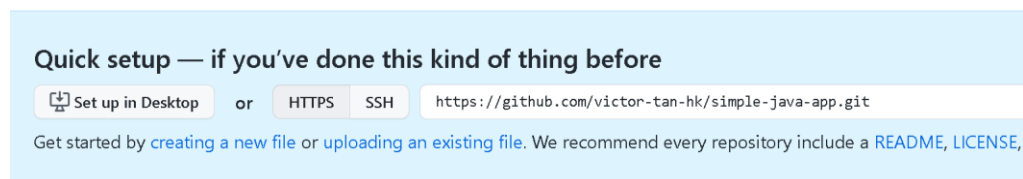
We will refer to this URL as `remote-url` in the commands to follow.

Back in the Bash shell in `simple-java-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
```

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

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

Before you can push the contents of the newly initialize repo to your remote GitHub repo, you must first obtain a classic personal access token (PAT):

https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token#creating-a-personal-access-token-classic

For scopes, select the following:

Once the token generation page appears, make sure you copy down the PAT to NotePad++ or somewhere where you can access it later.

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 for your username and password:

```
Username for 'https://github.com': yourusername
Password for 'https://githubaccountname@github.com': PAT
```

Type in the PAT at the password section. You can copy and paste the PAT, but be very careful to do this slowly and correctly.

Assuming you have entered the correct GitHub user account name and PAT, 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
```
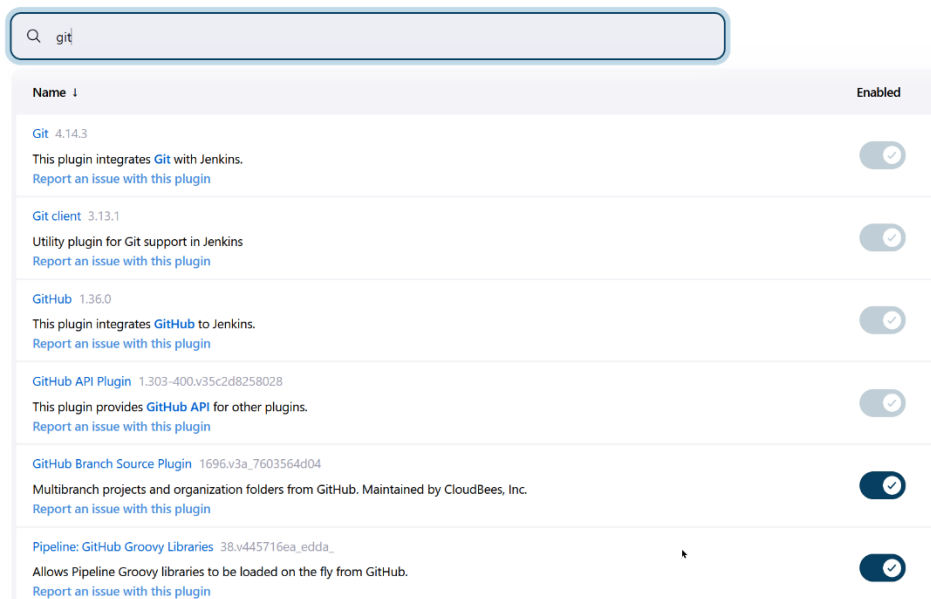
```
git status
```

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.

9   Checking / installing Git / GitHub plugins for Jenkins

There are a variety of Git related plugins for Jenkins, which should have already being 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
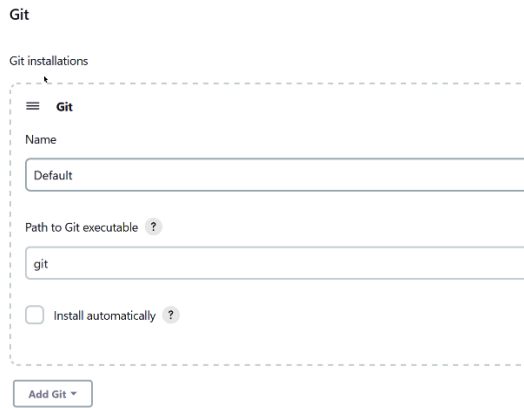
These are the plugins that should be installed which we will be using in subsequent labs. If they do not show up in your list, then go to Available Plugins and install them in the same way we have done previously.

You can initially configure Git in Manage Jenkins -> Global Tool Configuration, to ensure that the correct version of Git is used by Jenkins.

Typically, you can just use the default installation of Git (whether Linux / Windows), which should already be immediately accessible for standard installations of GIT (via the PATH environment variable).

10 Basic Jenkins job to integrate with GitHub

From the main Jenkins dashboard, create a new Item and name it: `userx-Basic-GitHubJob`
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:
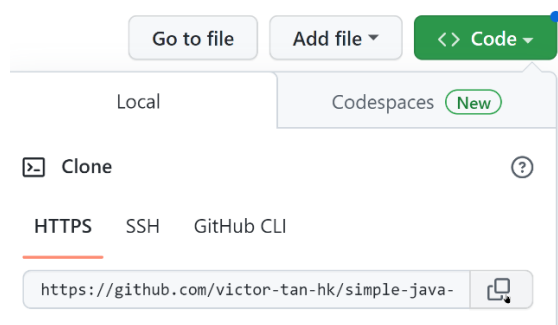
Paste the URL into the Repository URL on the Jenkins configuration page.

Click on Save and perform a build.
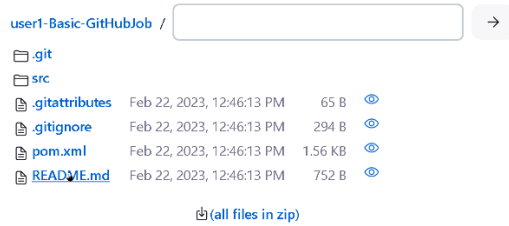
The Console Output for this build attempt will show that the remote repo contents have been cloned into the workspace folder for this current job (`workspace\userx-Basic-GitHubJob`).

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

Workspace of user1-Basic-GitHubJob on Built-In Node

user1-Basic-GitHubJob /   →

📁 .git
📁 src
📄 .gitattributes   Feb 22, 2023, 12:46:13 PM   65 B   👁
📄 .gitignore   Feb 22, 2023, 12:46:13 PM   294 B   👁
📄 pom.xml   Feb 22, 2023, 12:46:13 PM   1.56 KB   👁
📄 README.md   Feb 22, 2023, 12:46:13 PM   752 B   👁

⬇ (all files in zip)

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`

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

Maven Version

jenkins-maven
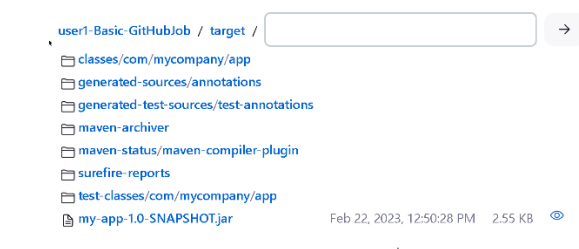
Goals

clean package
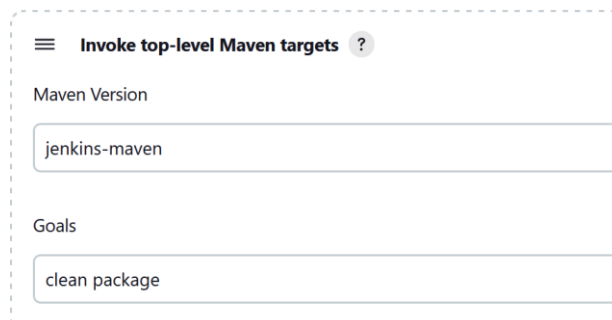
Click Save, then perform a Build.

The build may take some time to complete because the new installation of Maven that you completed earlier will be downloaded the dependencies it needs. When the build completes successfully, the Console output should show the log related to all these dependencies being downloaded, with the JAR artifact being generated successfully at the end.

You can check in the Workspace item from the navigation pane to verify the final JAR artifact was generated successfully.

Workspace of user1-Basic-GitHubJob on Built-In Nod

user1-Basic-GitHubJob / target /   →

📁 classes/com/mycompany/app
📁 generated-sources/annotations
📁 generated-test-sources/test-annotations
📁 maven-archiver
📁 maven-status/maven-compiler-plugin
📁 surefire-reports
📁 test-classes/com/mycompany/app
📄 my-app-1.0-SNAPSHOT.jar   Feb 22, 2023, 12:50:28 PM   2.55 KB   👁

Returning back to the main area for the job, select Configure, go to Build Steps and add another Build Step -> Execute Shell. Type this:
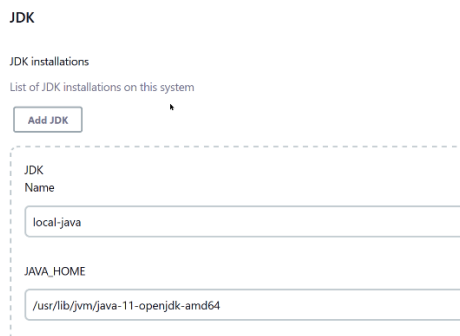
```
echo "*********************"
echo "Executing generated JAR"
echo "*********************"

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 with its corresponding output.

The locally installed JDK is directly accessible in the Shell build step (`java -jar ….`) without any further configuration for this purpose since the path to the Java binary is already on the $PATH environment variable.

Nevertheless, you can also explicitly configure its location in the same way as you did for Maven: Manage Jenkins -> Global Tool Configuration

JDK

JDK installations

List of JDK installations on this system

[Add JDK]

JDK
Name

| local-java |

JAVA_HOME

| /usr/lib/jvm/java-11-openjdk-amd64 |

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-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`

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 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
```

Enter your GitHub account username and PAT again to authenticate. If everything is correct, you should messages indicating that the changes have been pushed successfully to the GitHub repo.

To avoid constantly having to reenter the authentication credentials, we can use the credential helper to cache the password.

```
git config --local credential.helper "cache --timeout=360000"
```

We have set a long time period for the username / PAT to be retained in the cache, however if at any time in a future lab session, you find yourself being prompted again for a username / PAT combination, remember to type in this command.

IMPORTANT NOTE: Sometimes, during the caching the password, one of the other users on the shared server will not enter the command above correctly and as a result their cached username / password combination will override yours. If you find you are not able to push your commits to your GitHub repo after entering this command, then you will need to disable the caching with:

```
git config --local --unset credential.helper
```

```
git config --local credential.helper ""
```

Back on the GitHub repo main page, click on the Commits link to zoom in to the main commits page
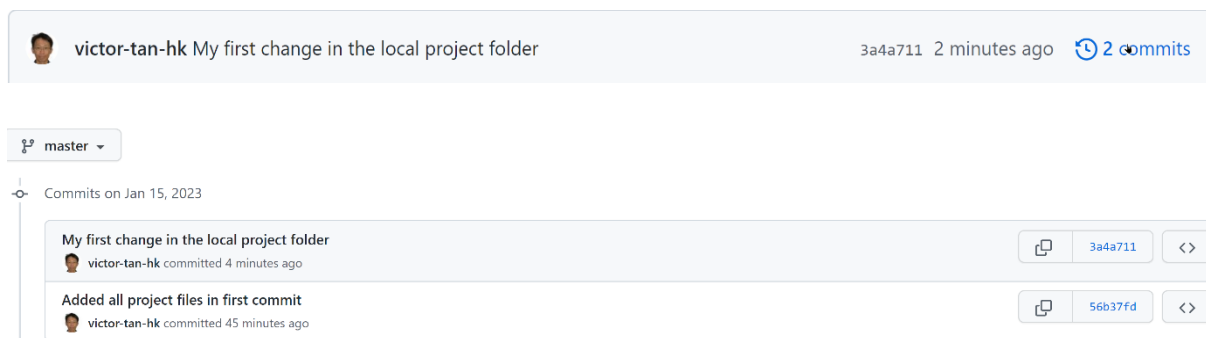
Then click on the hash of the particular commit to see its details.

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 and it will executed, with the output reflecting the latest change correctly.

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
```

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
```
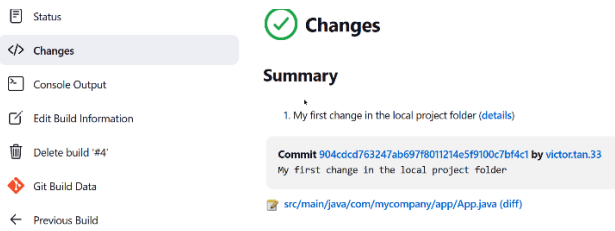
Then run a build again on the same job in Jenkins and verify from the Console Output that the JAR execution shows that the artifact was built from the latest changes pulled down from the remote GitHub repo.
You should not have to repeat the typing of your password after the next push since it should have been cached with the previous command you created.

You should be able to see the changes to the remote repo this via the Changes item in the left hand menu and also in the main Status item.
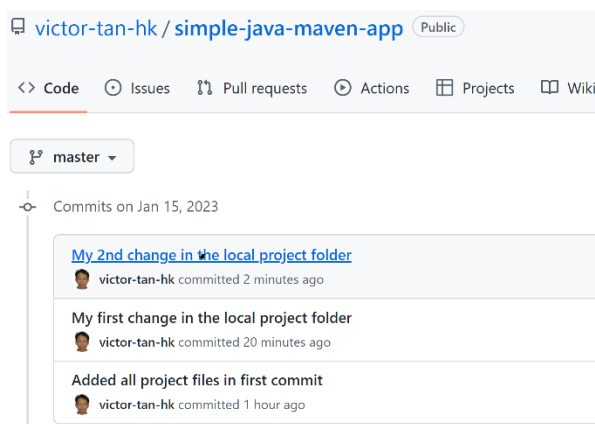
At any point in time, you can check the commit history of the master branch in the local project folder with:

```
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
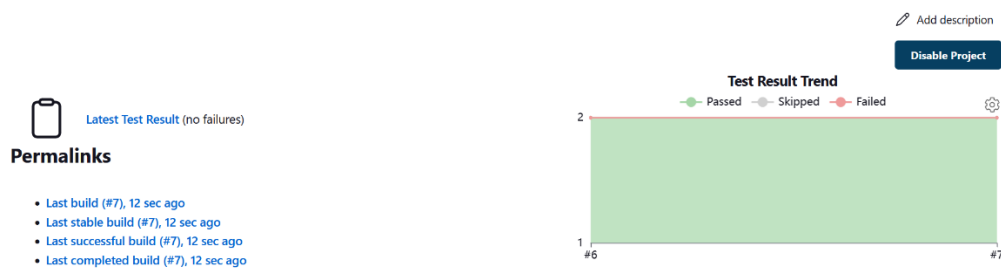
Add in this for the file set: `target/surefire-reports/*.xml`
This is the location for where the test report XMLs are located relative to the workspace folder

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.

In the main page for the build, you should see some info pertaining to the tests as well as link to get details on the Test Results up to that point in time (e.g. how many pass, how many fail, the difference, etc)

Return back to the project at: `simple-java-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)
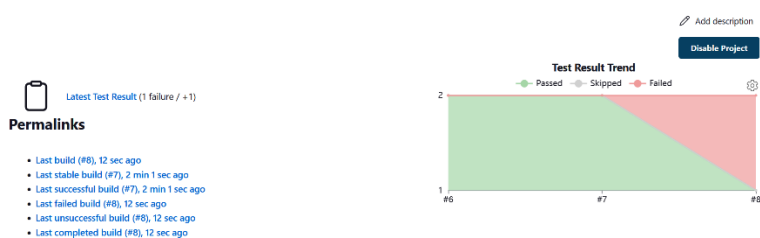
Return back to the app source code file:

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

Restore the value of String MESSAGE to its original value: `"I love Jenkins"`
Save and exit.

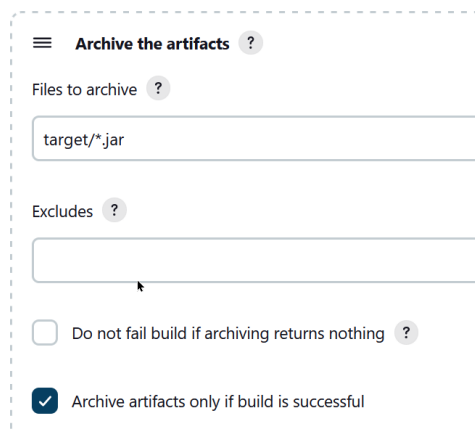As usual commit the changes and push the new commit to the remote repo:

```
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

≡ **Archive the artifacts** ?

Files to archive ?

```
target/*.jar
```

Excludes ?

☐ Do not fail build if archiving returns nothing ?

☑ Archive artifacts only if build is successful

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.

Last Successful Artifacts
📄 **my-app-1.0-SNAPSHOT.jar**   2.67 KB   view

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

11 Configuring periodic builds and SCM 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 are 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

The Jenkins CRON expression is nearly similar to the Linux CRON syntax with some minor differences:
https://www.jenkins.io/doc/book/pipeline/syntax/#cron-syntax

In the main area for the previous job (`Basic-GitHubJob`), 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)

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.
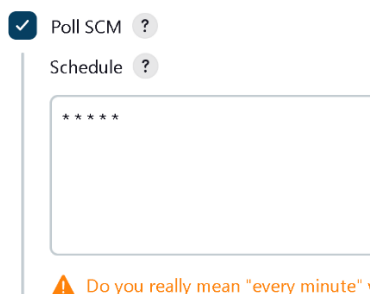
Click Save.

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.

The build main page also explicitly indicates that the changes were started by time (rather than manually by a user).
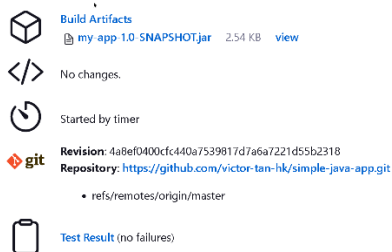
In the main area for the previous job (`Basic-GitHubJob`), select Configure, go to Build Triggers, and select Poll SCM. For the schedule, enter the CRON expression to run a job every minute, the same as previously.

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: `simple-java-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.

Notice as well that the build page clearly shows the build was triggered by a SCM change, and also provides details of the change (via the Changes link)

The main job page also shows the Changes detected in that Poll SCM.

Clicking on the github web link links you directly to the commit details in the GitHub repo.
All this additional info is provided back from a service in GitHub.

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.

12 Chaining jobs / projects

You can configure a build of Job B to execute immediately after the build of Job A completes. There are 2 common use cases for this:
- Job B has a dependency that is produced from the build for Job A (for e.g. a build artifact)
- Splitting a long build process into two (or more) smaller builds for easier management

Job B is considered a downstream project in this example, and we can perform the configuration in the Build Trigger section for Job B.

From the main Jenkins dashboard, create a new Item and name it: `userx-GitHubJob-Followup`
Make this a Freestyle Project and select Ok.
In the Build Triggers Section, select Build after other projects are built, and enter `userx-Basic-GitHubJob`
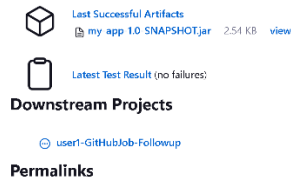
In the Build Steps section, choose Add Build step -> Execute Shell commands, and add in the following Linux shell commands:

```
echo "Executing follow up job"
java      -jar            /var/lib/jenkins/workspace/user1-Basic-
GitHubJob/target/my-app-1.0-SNAPSHOT.jar
```

Return back to `userx-Basic-GitHubJob` and run a build on it. You should see a link to the chained job after the build completes on the main page.

Project user1-Basic-GitHubJob

If you check the Console Output for the latest build on `userx-Basic-GitHubJob`, you should see the triggering of `userx-GitHubJob-Followup` at the end.

However, you will not see any output from that chained / linked job. Instead, you must click on the link to transition to the main page of that job. On the main page of that job, you will see `userx-Basic-GitHubJob` marked as an upstream project.

Project user1-GitHubJob-Followup

Upstream Projects

⊘ user1-Basic-GitHubJob

Click on the particular build in the Build history for this chained job and you can see its Console Output shows that it is able to access the JAR in the workspace folder of the upstream job in order to execute it. This shows that jobs of a given user are able to access files / directories of other jobs created by the same user. You can create access controls to prevent such access if necessary (where multiple jobs are run by different users on a single Jenkins Server), which is more advanced topic that we will look at later.

This chaining of jobs (`Basic-GitHubJob -> GitHubJob-Followup`) can also be accomplished alternatively by configuring `Basic-GitHubJob` instead of `GitHubJob-Followup`

In `userx-GitHubJob-Followup`, remove the Build Trigger and save.

Return to `userx-Basic-GitHubJob` and run another build. This time, `userx-GitHubJob-Followup` is not triggered.

In the configuration of this current job, in the Post Build Actions section, select Build Other Projects and add `userx-GitHubJob-Followup`

Now run another build again on `userx-Basic-GitHubJob`
Again you should see the triggering of `userx-GitHubJob-Followup` in Console Output and you can link to it again in the usual way.

Terminology: upstream, downstream
https://www.jenkins.io/doc/book/glossary/#downstream-1
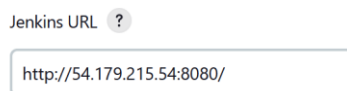
13 Configuring Webhooks with GitHub

In the previous labs, we looked at 3 different build triggers: Periodical builds, Poll SCM and Build after other projects are complete. Another popular and useful trigger is webhooks.

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)

https://docs.github.com/en/developers/webhooks-and-events/webhooks/about-webhooks
https://docs.github.com/en/get-started/customizing-your-github-workflow/exploring-integrations/about-webhooks

At the Jenkins UI, go to Manage Jenkins -> Configure System -> Jenkins Location
Make sure the Jenkins URL reflects the correct public IP address of the Main Server AWS EC2 instance (check the address bar to verify this or check the EC2 Instance details main dashboard).

Click Save when done.

To setup a WebHook for the GitHub repo `simple-java-app,` follow the guide below:
https://www.blazemeter.com/blog/how-to-integrate-your-github-repository-to-your-jenkins-project

Make sure you type the webhook URL with an ending / , for e.g.
http://13.229.52.22:8080/github-webhook/
or else error will occur when processing the HTTP POST request sent from GitHub on the Jenkins server end.

After the setup is complete on the GitHub end, it will send out a test ping to the Jenkins server at the specified URL and you should receive a success message if everything has been configured correctly.
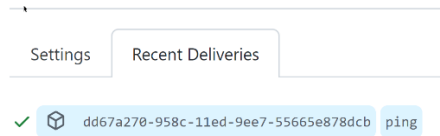
In the main area for the `userx-Basic-GitHubJob` select Configure, Build Triggers -> GitHub hook trigger for GitScm polling.

Click Save.

Wait for a few minutes. Notice that nothing happens.

Return to the project folder in `simple-java-app`
Let's make a minor modification to the source code of the main class at:

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

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 "Some changes to test webhooks"`

Check the status again

`git status`

Return to the main job page for `userx-Basic-GitHubJob` in the Jenkins UI, and keep your eye on the Build History

Back in the Git Bash shell, push this latest commit to the master on the remote GitHub repo with:

`git push`

Notice now that a build is automatically triggered as a result of the webhook that you have setup.

The main page clearly indicates the reason for the build being triggered (due to a GitHub push from a local repo by a user with the specified identity)

Build #16 (Feb 23, 2023, 8:14:25 AM)

Build Artifacts
my-app-1.0-SNAPSHOT.jar 2.55 KB view

Changes
1. Some changes to test webhooks (details / githubweb)

Started by GitHub push by victor-tan-hk

The GitHub Webhooks page should also correspondingly show the HTTP POST request that was sent out to the Jenkins server in order to trigger this build.

Webhooks / Manage webhook

Settings   Recent Deliveries

✓ ⊕ a183ae5e-b416-11ed-86bd-7761e1cea4d7 push

You can drill down into the request and response to perform trouble shooting if necessary.

Webhooks / Manage webhook

Settings   Recent Deliveries

✓ ⊕ a183ae5e-b416-11ed-86bd-7761e1cea4d7 push   2023-02-24 15:41:22 ...

Request   Response 200   Redeliver ⏱ Completed in 0.76 seconds.

Headers

```
Request URL: http://13.212.167.150:8080/github-webhook/
Request method: POST
Accept: */*
content-type: application/json
User-Agent: GitHub-Hookshot/3e3b786
X-GitHub-Delivery: a183ae5e-b416-11ed-86bd-7761e1cea4d7
X-GitHub-Event: push
X-GitHub-Hook-ID: 402284247
X-GitHub-Hook-Installation-Target-ID: 605076151
X-GitHub-Hook-Installation-Target-Type: repository
```

Payload

14 Setting up a BitBucket repo

We will now create a new empty remote repo on BitBucket and push the contents of a local repo to populate it, the same way that we did for our GitHub repo.

Login to your Bitbucket account. Go through the process of creating a new repository, but create a bare repository with no content to facilitate the process of pushing the contents of our local repository to it:

Enter the values for the following fields.
Project Name: MyJavaProject

Repository Name: `simple-java-app`

Make sure this is **NOT** a private repository, and **DON'T** include either a readme file or .gitignore. This is to ensure that the repository is a bare repository that we can push new content into from our local repo.

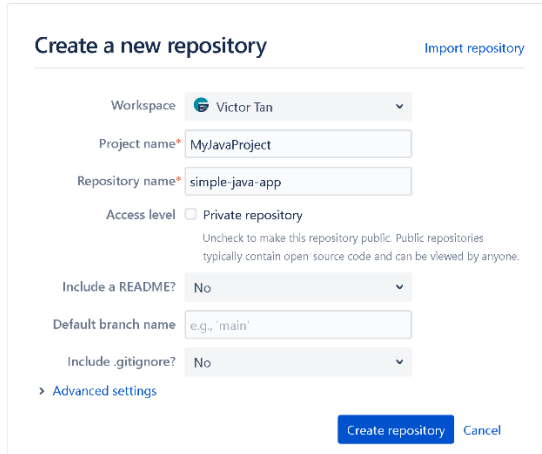When you are done specifying the values for the fields as shown above, click Create Repository. You will be transitioned to the Source view for the newly created repo, where some instructions will be provided on how to get started.
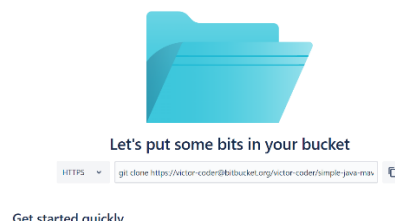
Click on the Clone button to copy the URL (e.g. https://xxx@bitbucket.org/yyy/simple-java-maven-app.git) for this new remote repo to an empty document. We will refer to this URL as *remote-url* in the commands to follow.

In your main userx folder, there is a a `bitbucketprojects` folder, which in turn contains a `simple-java-app` folder. This is an exact copy of Java Maven project that we were working with earlier in the `githubprojects` folder

In the Bash shell, change to this folder with:

```
cd ~/userx/bitbucketprojects/simple-java-app
```

As we did earlier, initialize this as a new Git repo with:

```
git init
git add .
git status
```

Before creating the first commit in this newly initialized repository, you need provide an identify for the individual creating the commit (the commit author). Since we are going to be pushing our local repo to populate the BitBucket repo, we will use our BitBucket identity (name and email). This is accessible from the upper right hand corner of the page.
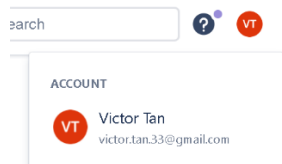
and note down the name and email address there. Use them in the configuration commands below:

```
git config --local user.name "your name"

git config --local user.email "your email address"
```

We are configuring these details at the local level so that they apply only to this repository, rather than to the user account since all users are now using this user account `mainuser.`

Then we can create our first commit.

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

Next, we associate the remote repo URL with our newly initialized local repo with:

```
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 you can push the contents of the newly initialize repo to your remote BitBucket repo, you must first obtain an app password. This is conceptually similar to classic personal access token (PAT) that we used in GitHub repo.

https://support.atlassian.com/bitbucket-cloud/docs/create-an-app-password/

Give the app password a suitable label and the permissions shown below:
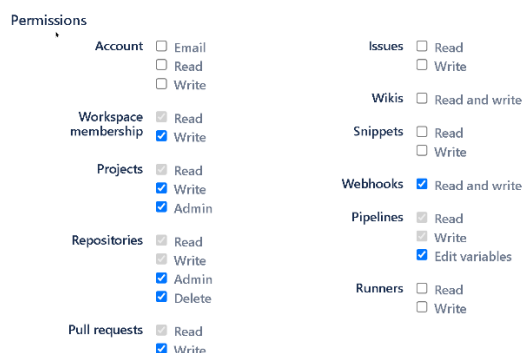
After creating the app password, make sure you copy it down somewhere where you can access it.

Next, set up password caching so that you don't have to type your username / app password repeatedly after you use it for the first time:

```
git config --local credential.helper "cache --timeout=36000"
```

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 for your app password:

```
Password for 'https://victor-coder@bitbucket.org':
```

You can copy and paste the app password, but be very careful to do this slowly and correctly.

Assuming you have entered the correct app password, the command should succeed with appropriate success messages in the Bash shell.

```
Enumerating objects: 39, done.
….
…..
* [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Return back to the browser and check through the Source, Commits and Branches main view to verify that these mirrors the status of the local repo that you just pushed up into it. Note that you may need to navigate out to the main Repositories tab in the main menu, and then click on the `simple-java-app` repo entry in order for it to refresh properly and show the uploaded content. This may take some time to complete correctly.
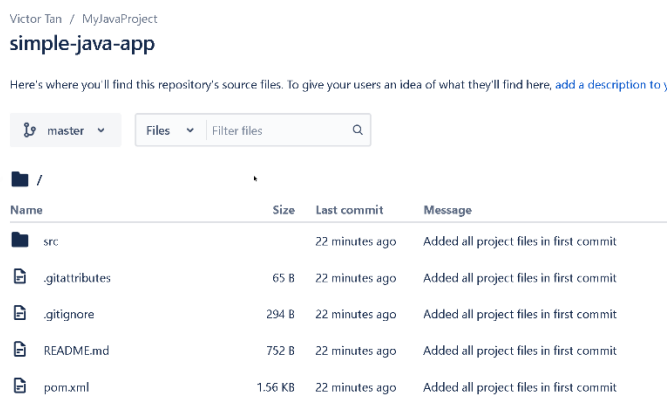
Back in the Bash shell, type the following commands to verify that the remote tracking branches have been set up and that the local and upstream master are both in sync with each other.

```
git remote show origin
```

```
git status
```

15 EXERCISE: Configuring Webhooks with BitBucket

We are now going to configure a WebHook for our BitBucket repo, which is conceptually the same as what we did for GitHub earlier.

Only one person with admin access needs to perform the plugin installation detailed below.
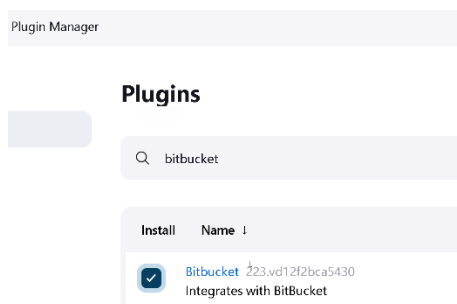
From the main Jenkins Dashboard, Manage Jenkins -> Manage Plugins -> Available Plugins, look for BitBucket and select the item.

Plugin Manager

Plugins

🔍 bitbucket

Install   Name ↓

☑ Bitbucket 223.vd12f2bca5430
   Integrates with BitBucket

Select Install without Restart, and click Restart Jenkins when installation is complete and no jobs are running once the download is complete.

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

As an exercise on your own, set up the relevant steps / tasks in this new job to pull changes from your new BitBucket repo, run a Maven build to create a JAR artifact and then execute it. Finally setup a configuration for a BitBucket Webhook and test it by pushing new commits from your local repo.

A sample guide to do this is shown below.

https://medium.com/ampersand-academy/integrate-bitbucket-jenkins-c6e51103d0fe
IMPORTANT NOTE:
- This guide shows also the configuration of a username/password credential for the BitBucket repo. This is not necessary to perform for our case since we have made our BitBucket repository a public repository earlier. Otherwise, we would need to configure credentials (similar to the case of GitHub repo) to clone or pull changes from a remote repo.
- In Step 5, to test the Bitbucket and Jenkins Integration, a change is made to the project code base directly in the BitBucket UI itself. You can do this, but a more realistic way is to perform what we did earlier to test out the webhook for GitHub, i.e. by changing the source code of the main class at:

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

Then add the modifications to a new commit with an appropriate message:

`git commit -am "Testing out BitBucket webhook"`

and then push this latest commit to update the master on the remote Bitbucket repo with:

```
git push
```

Make sure to use paste the app password you created earlier.

16 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.

NOTE: This only needs to be done by one person from the group that is working on the Main Server.

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

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

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
```
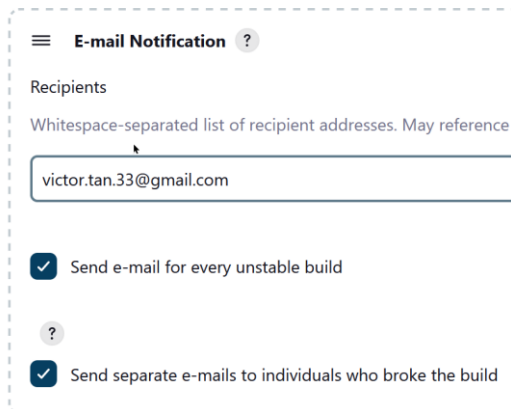
✅ Test configuration by sending test e-mail

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 (`userx-Basic-GitHubJob`), 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: `githubprojects/simple-java-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 Ouput, 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.
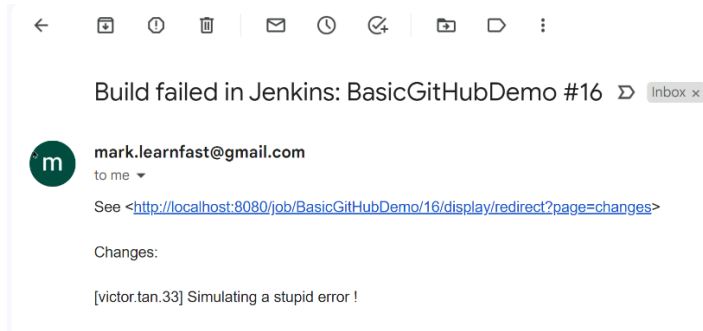
In addition to Gmail, there are many other organizations that provide free SMTP servers to cater for this kind of situation. Your company / organization may also have its own internal SMTP server that you can make use of:

https://moosend.com/blog/free-smtp-server/
https://www.emailvendorselection.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/