Intro to Jenkins Lab 2

1 LAB SETUP 1
2 PIPELINE BASICS 1
3 PIPELINE BASIC STRUCTURE....................2
4 PIPELINE: NODES AND PROCESSES 3
5 PIPELINE: PLATFORM INDEPENDENT STEPS 4
6 CREATING A BASIC GIT-MAVEN PIPELINE 5
7 CONFIGURING PERIODIC BUILDS, SCM POLLING AND WEBHOOKS IN THE PIPELINE 8
8 SETTING UP THE PIPELINE AS A JENKINSFILE 11

1 Lab setup

We will still be working with the Main Server from Lab 1, so ensure 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.

In addition, we will be using the pipeline scripts from the various text files located in the `labcode` workshop folder. You can open this using Notepad++ before copying and pasting them into the relevant configuration sections of the pipelines that you will be creating.

2 Pipeline basics

We can consider a pipeline as a specialized job that is specifically meant to represent a model of a continuous delivery (CD) pipeline, performing the entire range of tasks that are typical for this pipeline. Just as is in the case of freestyle projects, these tasks are performed by installed plugins.

The functionality for a typical CI / CD pipeline can also be configured and achieved via a freestyle project. The main difference is that pipeline functionality is specified via a special kind of domain specific language (DSL) syntax, in what is known as the pipeline-as-code technique. This is opposed to freestyle projects whose functionality is specified through UI-based configuration options.

The definition of a pipeline is specified in a text file called a Jenkinsfile. There are two main ways to specify a Jenkinsfile (the syntax for both cases are identical)

- written via the UI in the Script text area of the Pipeline section when a pipeline job is created.

- stored as a part of the project files in SCM (GitHub, BitBucket, GitLab, etc), where it is versioned and reviewed just like any other source code artifacts (the more common approach)

There are two main forms of pipeline syntax: Declarative and Scripted. The primary differences between these two approaches are summarized here:

The recommendation for newbies and general users is to use the declarative syntax approach, which is what we will be focusing on for most of this workshop. Scripted syntax offers the most flexibility and power, but requires users be familiar with Groovy, the DSL variant of Java that is used here.

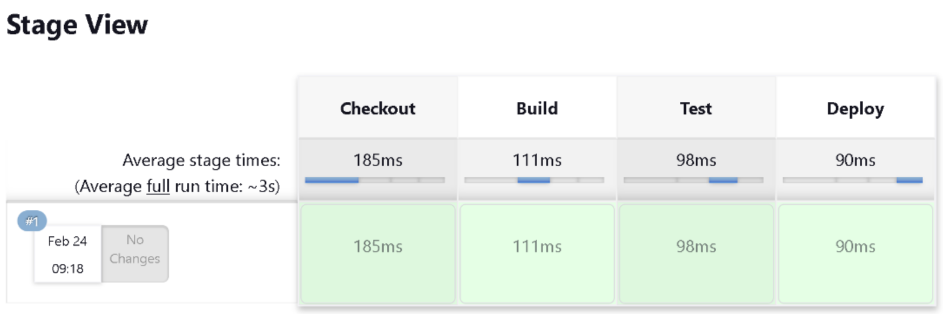
3 Pipeline basic structure

From the main Jenkins dashboard, create a new Item and name it: `userx-Pipeline-Basics`
Make this a Pipeline and select Ok.

Notice now that there are no multiple stages / sections to configure the pipeline as you would in a normal freestyle project. This is because all the functionality of hose stages / sections are going to be implemented as steps in the pipeline script.

Select the Pipeline section and copy and paste the script from: `pipeline-basic-structure`

Run a Build on this job in the main page.

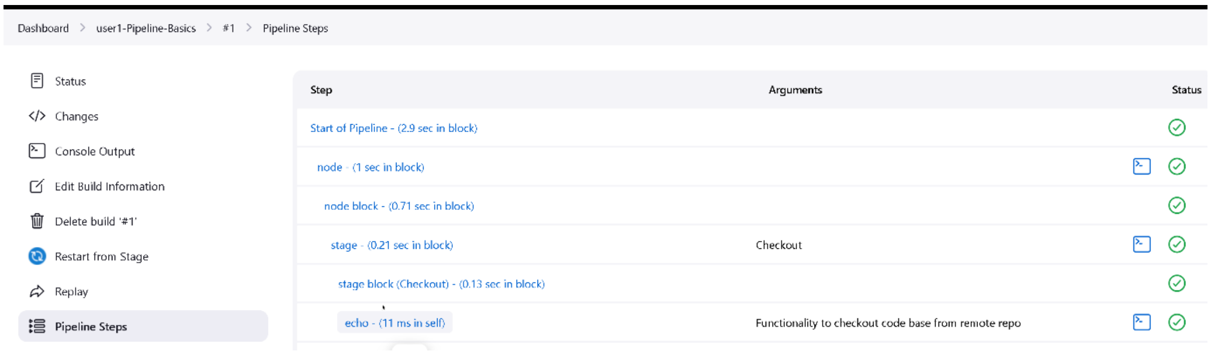


	Checkout	Build	Test	Deploy
Average stage times: (Average full run time: ~3s)	185ms	111ms	98ms	90ms
#1 Feb 24 09:18	185ms	111ms	98ms	90ms

You should see a visualization of the execution of all the main stages in the pipeline after its execution on the main page, which also clearly indicates the average time to execute each stage (across multiple builds).

The stages and steps in a pipeline are conceptually similar to the main stages that are available in the configuration of a freestyle project and the tasks available within each of those stages. All functionality achievable in a freestyle project can also be achieved in a pipeline by incorporating appropriate steps from the relevant plugin in accordance to pipeline syntax, which we will see later.

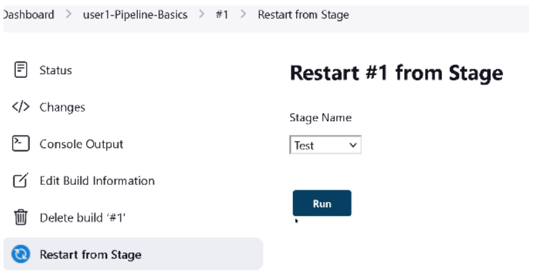
If you check the Console Output for a particular build, you will see the standard output (from the various echo statements in the script). In addition, you have the opportunity to view the Pipeline Steps in detail, where you can see the time taken to execute each stage, the status of execution (success / failure) and also to see the CLI output corresponding to that stage.


Dashboard > user1-Pipeline-Basics > #1 > Pipeline Steps

Step	Arguments	Status
Start of Pipeline - (2.9 sec in block)		✓
node - (1 sec in block)		✓
node block - (0.71 sec in block)		✓
stage - (0.21 sec in block)	Checkout	✓
stage block (Checkout) - (0.13 sec in block)		✓
echo - (11 ms in self)	Functionality to checkout code base from remote repo	✓

You can also choose to Replay (execute all the stages in the pipeline again), or to restart the pipeline from a given stage.

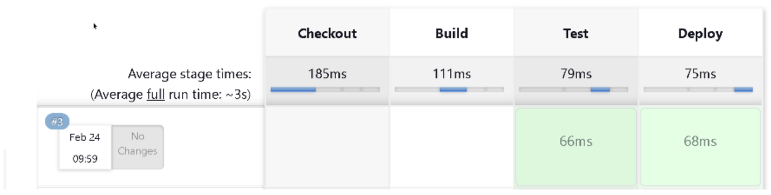
For e.g. you could choose to restart the pipeline again from the Test stage.


Dashboard > user1-Pipeline-Basics > #1 > Restart from Stage

Restart #1 from Stage

Stage Name: Test

A new build will be executed and this time you will only see the stats related to the stages that you chose execution to commence from.



	Checkout	Build	Test	Deploy
Average stage times: (Average full run time: ~3s)	185ms	111ms	79ms	75ms
#3 Feb 24 09:59 No Changes			66ms	68ms

4 Pipeline: Nodes and Processes

In the current pipeline job, select Configure and select the Pipeline section and copy and paste the script from: `pipeline-demo-sh`

Run a build and check the Console Output. This pipeline job essentially replicates some of the functionality that we performed in an earlier freestyle project using Execute Shell Commands.

The `sh` step is one of the steps involved in the [Pipeline: Nodes and process plugin](), which is one of the core plugins that come with Jenkins (installed as part of the recommended plugins during initial installation).

Each step corresponds to functionality provided by the plugin and represents an atomic unit of action to be performed in a pipeline or freestyle project. For a freestyle project, as we have seen in earlier labs, the action is directly configured via UI options / parameters

All the steps that you could use in a pipeline related to most plugins in the Jenkins ecosystem are accessible for reference from here. You can incorporate any of them directly into the `steps` section of the pipeline script, as long as you have their corresponding plugin installed:

The `bat, powershell, pwsh` and `sh` steps allow the execution of commands that are specific to those environments. Therefore, you have to ensure that they are only used on a Jenkins controller that is installed or has access to those environments.

As an optional exercise, if you have Jenkins installed on a Windows machine, create a pipeline job with the same name: `Pipeline-Basics`

Select Configure and select the Pipeline section and copy and paste the script from: `pipeline-demo-bat`
Run a build and check the Console Output. This pipeline job essentially replicates the functionality of the previous job that you created on Jenkins service on a Linux machine based on `pipeline-demo-sh`

Now, try and interchange the scripts, for e.g. copy and paste the script of `pipeline-demo-bat` into the job on Jenkins server in the Linux machine, or copy and paste the script of `pipeline-demo-sh` into the job on Jenkins server in the Windows machine. In both cases, running a build on these jobs will result in an error.

This is because the commands in those scripts are specific to the environment that Jenkins is installed in. For e.g. attempting to use `bat` on Jenkins running on a Linux machine will result in an error, and the same goes for for `sh` on a Windows machine.

5 Pipeline: Platform independent steps

As we have just seen, some basic file / directory operations are platform specific (Windows / Linux). Often it is useful to create a pipeline script which can run independently of the platform that Jenkins is installed on.

From the main Jenkins dashboard, create a new Item and name it: *`userx`*`-Pipeline-Independent`
Make this a Pipeline and select Ok.

Select the Pipeline section and copy and paste the script from: `pipeline-demo-independent`

Run a Build on this job in the main page. The Console Output should indicate success.

The steps in the stage for this pipeline are taken from the Pipeline: Basic Steps plugin

with the exception of the `dir` step which is from Pipeline: Nodes and process

Notice that it replicates some (not all) of the functionality from: `pipeline-demo-sh`

As an optional exercise, if you have Jenkins installed on a Windows machine, create a pipeline job with the same name: `Pipeline-Independent`

Select the Pipeline section and copy and paste the script from: `pipeline-demo-independent`

Run a Build on this job in the main page. The Console Output should indicate success.

In addition to the Pipeline: Basic Steps plugin, there are a few other plugins that allow you to perform basic file / directory related operations in a platform independent manner:

6 Creating a basic Git-Maven pipeline

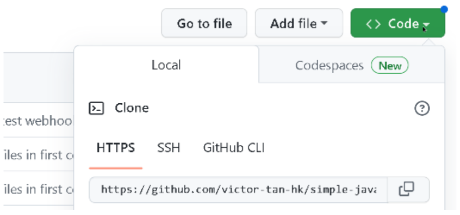
Here, we are going to replicate the functionality of freestyle project that we created in our previous lab where we:

- Downloaded changes to Java Maven project from a remote (GitHub / Bitbucket) repo
- Ran some Maven goals (e.g. clean package) on the project
- Execute the generated artifact (JAR)

From the main Jenkins dashboard, create a new Item and name it: *`userx`*`-Pipeline-GitMaven`
Make this a Pipeline and select Ok.

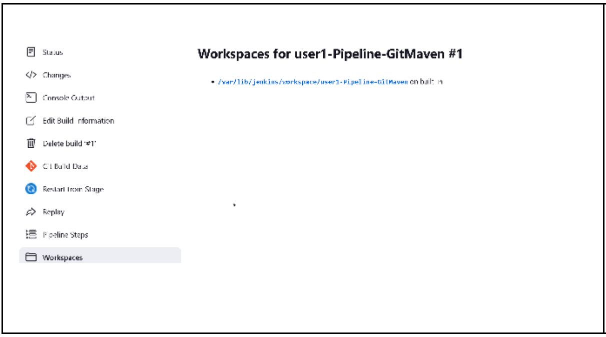
Select the Pipeline section and copy and paste the script from: `pipeline-git-maven-pt1`

Modify the script to specify the HTTPS URL for the GitHub repo for `simple-java-app` that you had created earlier:



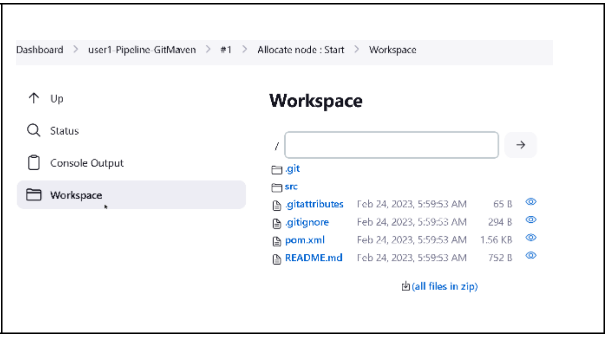
```
https://github.com/victor-tan-hk/simple-java
```

Run a build and check the Workspaces link on the page for that particular Build attempt. You should be able to see that the entire remote repo has now been cloned to the workspace folder for this pipeline job.



Workspaces for user1-Pipeline-GitMaven #1

- /var/lib/jenkins/workspace/user1-Pipeline-GitMaven on built-in


Dashboard > user1-Pipeline-GitMaven > #1 > Allocate node : Start > Workspace
Up
Status
Console Output
Workspace

Workspace

```
/
.git
src
.gitattributes   Feb 24, 2023, 5:59:53 AM     65 B
.gitignore       Feb 24, 2023, 5:59:53 AM    294 B
pom.xml          Feb 24, 2023, 5:59:53 AM  1.56 KB
README.md        Feb 24, 2023, 5:59:53 AM    752 B
(all files in zip)
```


As mentioned earlier, all functionality achievable in a freestyle project is achieved in a pipeline by incorporating appropriate steps from the relevant plugin. Jenkins provides a way to translate from common freestyle project configuration options to a pipeline snippet in order to facilitate the process of writing a pipeline script.

On the main pipeline job page, select Pipeline Syntax.
In the Snippet Generator, the Sample Step drop down gives a list of common options that you would normally configure in a Freestyle project. Select `git:Git`

Notice that you now get the configuration text boxes that you had when you configured the Source Code Management stage in a normal freestyle project. Configure the boxes with the appropriate text and then select Generate Pipeline Script. You will now get the pipeline snippet corresponding to that configuration which you can proceed to copy and paste into the relevant `steps` section in the pipeline. Remember that you can repeat this for most of the common freestyle project configuration options that you might use.

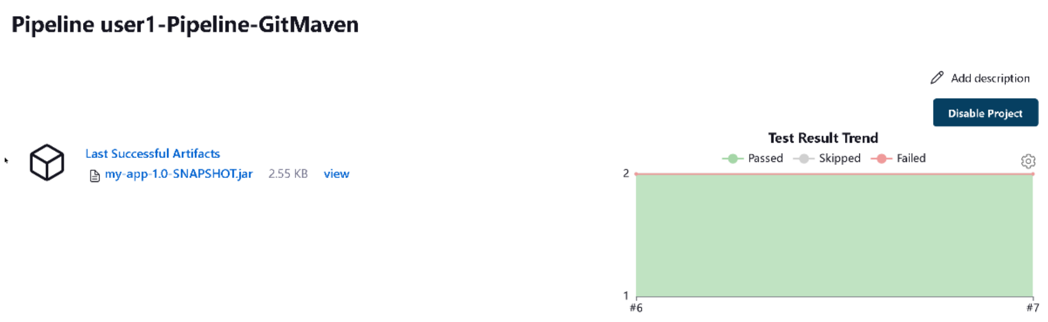
Go back to the main page for the pipeline, select Configure, go to the Pipeline section and copy and paste the script from: `pipeline-git-maven-pt2`
Remember again to modify the script to specify the HTTPS URL for the GitHub repo for `simple-java-app`

Notice now we have a tools directive where we reference the particular Maven version we installed and configured in the Manage Jenkins -> Global Tool configuration section.

Run a build and check the Console Output shows the appropriate Maven goals being executed, with the required JAR artifact generated at the end of the process.

Go back to the main page for the pipeline, select Configure, go to the Pipeline section and copy and paste the script from: `pipeline-git-maven-pt3`
Remember again to modify the script to specify the HTTPS URL for the GitHub repo for `simple-java-app`

Run a build and check the Console Output shows the appropriate Maven goals being executed (this time we have included a test goal) as well as an additional stage to archive the artifact if the build was successful, and then the test result graph is shown on the pipeline job main page (exactly the same as is in the case of the freestyle project that we completed previously).


Pipeline user1-Pipeline-GitMaven

Last Successful Artifacts
my-app-1.0-SNAPSHOT.jar 2.55 KB view

This part includes a `post` section

As we did before previously in the case of the freestyle project, lets test out some builds after we push some new changes from our local repo to the remote GitHub repo.

Return to the project folder `githubprojects/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, for e.g. `"Some changes for pipeline"`
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 "Next change for new pipeline project"
```

If there has been a long lapse since the last time you did a credential cache, you can enable the cache again in case you need to type your password / PAT again:

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

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

```
git push
```

If everything is correct, you should messages indicating that the changes have been pushed successfully to the GitHub repo.

Run a build on this pipeline again. Verify that the execution of the JAR shows the latest changes you had made to the project source code.

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 in the pipeline !"
```

```
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. Notice that the later stages after the stage that failed (Deploy, Archive, etc) were not executed.

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 in pipeline !"
```

```
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).

7 Configuring periodic builds, SCM polling and webhooks in the pipeline

Here, we continue to extend this pipeline to replicate the functionality that we achieved in the freestyle project.

Go back to the main page for the pipeline, select Configure, go to the Pipeline section and copy and paste the script from: `pipeline-git-maven-build-trigger`
Remember again to modify the script to specify the HTTPS URL for the GitHub repo for `simple-java-app`

Here, we set a CRON expression pattern to run a job every minute.

Click Save.

Go back to the main page for the job and run a build first. You will need to run a build when you initially add in a trigger for that trigger to take effect subsequently.

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.

Now go back to the main page for the pipeline, select Configure, go to the Pipeline section. Replace the relevant section with:

```
  triggers {
    pollSCM ('* * * * *')
  }
```

Click Save. Again run an initial build to make sure that this change in trigger takes effect.

Then, 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.

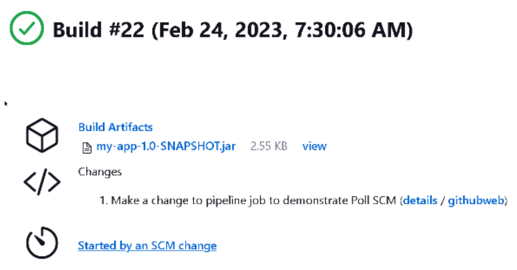
As usual commit the changes and push the new commit to the remote repo:

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

git push
```

Go back to the main page for the job and wait for a minute or so. 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)


Build #22 (Feb 24, 2023, 7:30:06 AM)

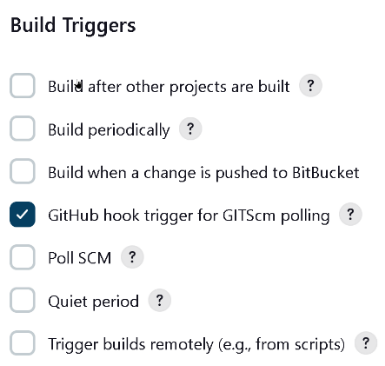
Build Artifacts
my-app-1.0-SNAPSHOT.jar 2.55 KB view

Changes
1. Make a change to pipeline job to demonstrate Poll SCM (details / githubweb)

Started by an SCM change

Finally, we configure this pipeline to work with GitHub Webhook that we configured in an earlier lab. First, go back to the GitHub page for this project and verify that the GitHub Webhook is still in place and is pointing to the correct URL.for e.g.
http://13.229.52.22:8080/github-webhook/

In the configuration for the pipeline, select this option in the Build Triggers



Build Triggers

- ☐ Build after other projects are built ?
- ☐ Build periodically ?
- ☐ Build when a change is pushed to BitBucket
- ☑ GitHub hook trigger for GITScm polling ?
- ☐ Poll SCM ?
- ☐ Quiet period ?
- ☐ Trigger builds remotely (e.g., from scripts) ?

And remove the previous directive

```
triggers {
  pollSCM ('* * * * *')
}
```

and Save.

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.

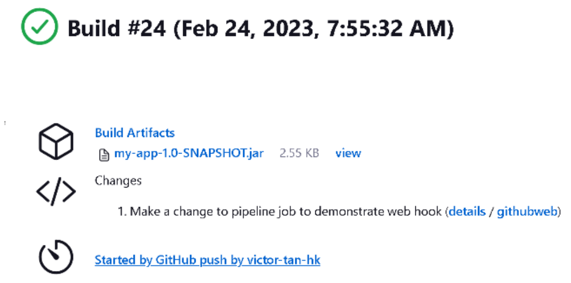
As usual commit the changes and push the new commit to the remote repo:

```
git commit -am "Make a change to pipeline job to demonstrate web hook"

git push
```

Go back to the main page for the job and notice now that a build is automatically triggered as a result of this push.

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 #24 (Feb 24, 2023, 7:55:32 AM)

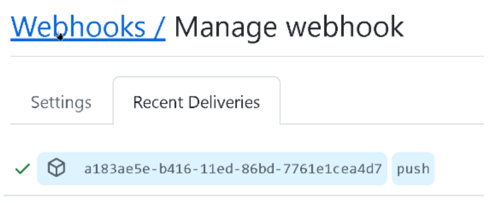
Build Artifacts
my-app-1.0-SNAPSHOT.jar 2.55 KB view

Changes

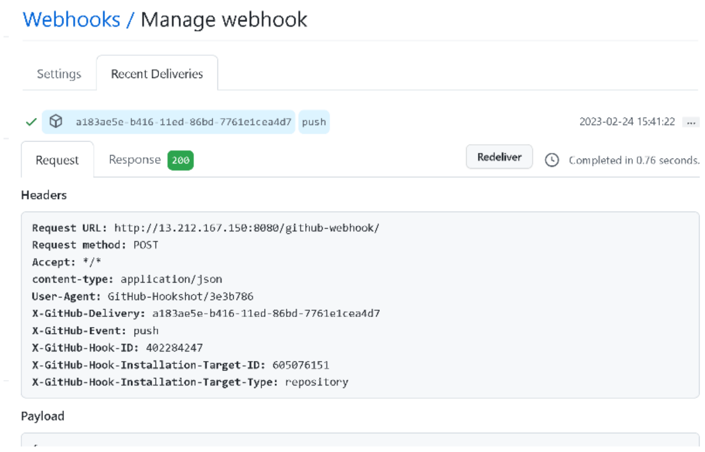
1. Make a change to pipeline job to demonstrate web hook (details / githubweb)

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

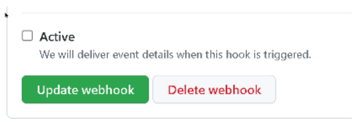
You can drill down into the request and response to perform trouble shooting if necessary.



```
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
```

8 Setting up the pipeline as a Jenkinsfile

Before doing this, go back to the Manage Webhooks section of the WebHooks home page of GitHub repo and deactivate the Webhook we created earlier, then select Update webhook.



Change to the home directory of the Java project:

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

We will create a new file here called `Jenkinsfile`.

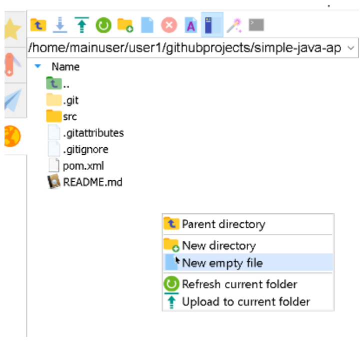
You can either create this using the built in CLI nano editor nano.

```
nano Jenkinsfile
```

and copy and paste the content of `jenkinsfile-git-maven` into here and save and exit. After that, you can verify the content by typing:

```
less Jenkinsfile
```

Alternatively, you can create a new file using the MobaXTerm editor.



Once the file has been created with the correct name and in the correct location, commit this change and push the new commit to the remote repo:

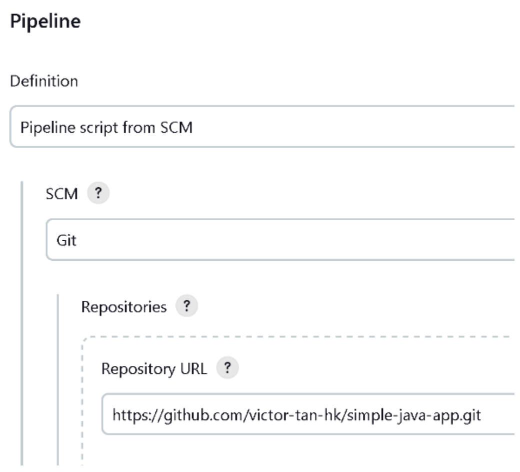
```
git add .

git commit -m "Introduced Jenkinsfile"

git push
```

Go back to the GitHub page for this repo and verify that the Jenkinsfile is now checked in successfully into the root folder of this project.

From the main Jenkins dashboard, create a new Item and name it: `*userx*-Jenkinsfile-GitMaven`
Make this a Pipeline and select Ok.
In the Pipeline section, select a Definition: Pipeline Script from SCM and fill in the Repository URL with the HTTPS URL of your GitHub Repo.


Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories

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

The most important part is the script path: which specifies the path and the name of the Jenkinsfile. Here, we retain the default name (Jenkinsfile) and the default location (the root folder of the Git project). For more complex projects, you might want to provide a different name and different location, in which case, you must explicitly specify it here.

Click Save and Run a Build. The Console Output should show everything running properly and correctly. The build is now being executed based on the pipeline script in the Jenkinsfile.

Let's make a meaningless addition to the Jenkinsfile in the project in `githubprojects/simple-java-app`

Using either the CLI editor `nano` or using the MobaXTerm Editor, put in this snippet between the `build` and `test` stages:

```
    stage('Dummy') {
      steps {
        echo 'Meaningless statement to verify this'
      }
    }
```

Save this.

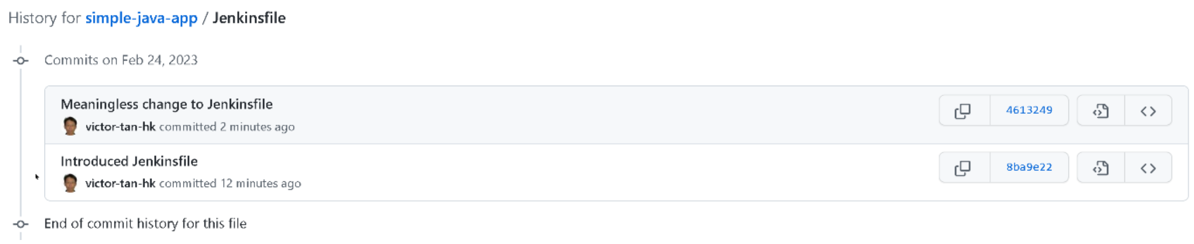
Commit this change and push the new commit to the remote repo:

```
git commit -am "Meaningless change to Jenkinsfile"

git push
```

Back in Jenkins, run another Build on this job and verify that the console output shows the output from the `echo` statement addition that we made in the Jenkinsfile.

The most important result of this change is that since the Jenkinsfile is now part of the Git repo along with the source code and other configuration files, and we can therefore easily track changes to it.


History for simple-java-app / Jenkinsfile

Commits on Feb 24, 2023

Meaningless change to Jenkinsfile
victor-tan-hk committed 2 minutes ago
4613249

Introduced Jenkinsfile
victor-tan-hk committed 12 minutes ago
8ba9e22

End of commit history for this file

This means as the project evolves over its lifetime and changes are made to the Jenkinsfile to control how the build is being performed, we <u>can keep track of how the build was actually performed at a point in time and the author of the associated changes to that Jenkinsfile</u>.

Remember that this is considered best practice for specifying a pipeline script, and you should always endeavour to do this in all your projects.