Chapter 3 Simple Calculator Program

Introduction

Now that we have gone through the installation procedure, the best way to learn the process is by taking on a simple project. We will aim to develop a calculator program which will utilise all the tools that we have installed and produce a DevOps workflow. By the end of this project, you will have a good grasp of the tools used and will be in a much better position to take on the real-world case studies detailed later in the book.

Objectives

The goal of the chapter is to not focus on the code or the program but the DevOps tools and workflows that we incorporate. We will begin by creating a local project in an IDE. Then creating a local repository using version control. Once the local repository is created, we will push it to a remote version control repository.

Once we have functioning version control, we will focus on creating a pipeline that will automatically build the project and run test cases every time changes are pushed to the repository. This will be followed by automatic containerisation of the project and pushing to a remote container repository. Finally, the Jenkins pipeline will pull from the container and run the project completing the deployment.

This entire process highlights the typical DevOps flow. The development phase deals with version control and automated builds. The operations part is the containerisation and deployment of the containers on target machines.

DevOps Solution

We will utilise **Git** to handle the repository creation locally. Then we will connect to **GitHub** as our remote repository solution. For the creation of pipeline and automation tool, we will use **Jenkins**. Finally for the containerisation we will use **Docker** and **Docker Hub** to remotely store our containers. On the analysis side, we have **ELK Stack** to help us visualise the logs generated by our program.

While we have the prerequisites installed it is also important to note that tools are opensource solutions. Consequently, there are amazing communities that will help with problems and questions that one can encounter. Not to mention, years of painstaking documentation and bug fixes.

Implementation

Getting Started with Project Creation and Maven

Our first step will be to create the project in the IDE. We launch IntelliJ IDEA and create a new project:

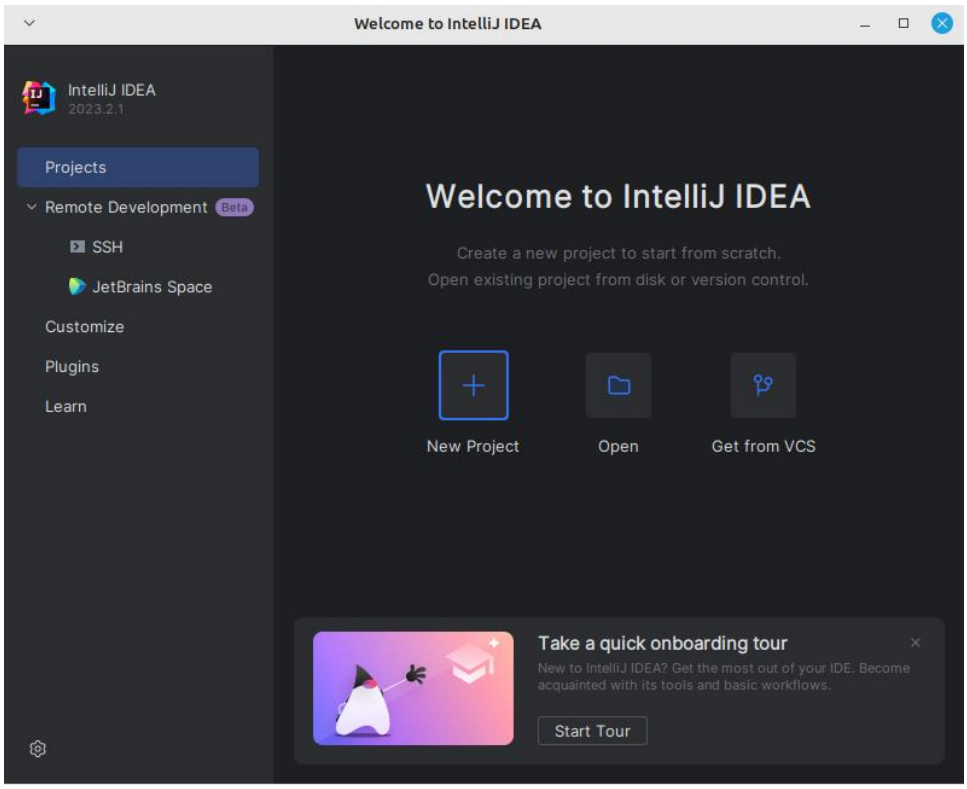

Fig 3.1 IntelliJ IDEA New Project

We will create a new project. In the pop up for the new project, we name our project **Calculator**, select **Maven** for our build and choose the **default java** in the JDK dropdown:

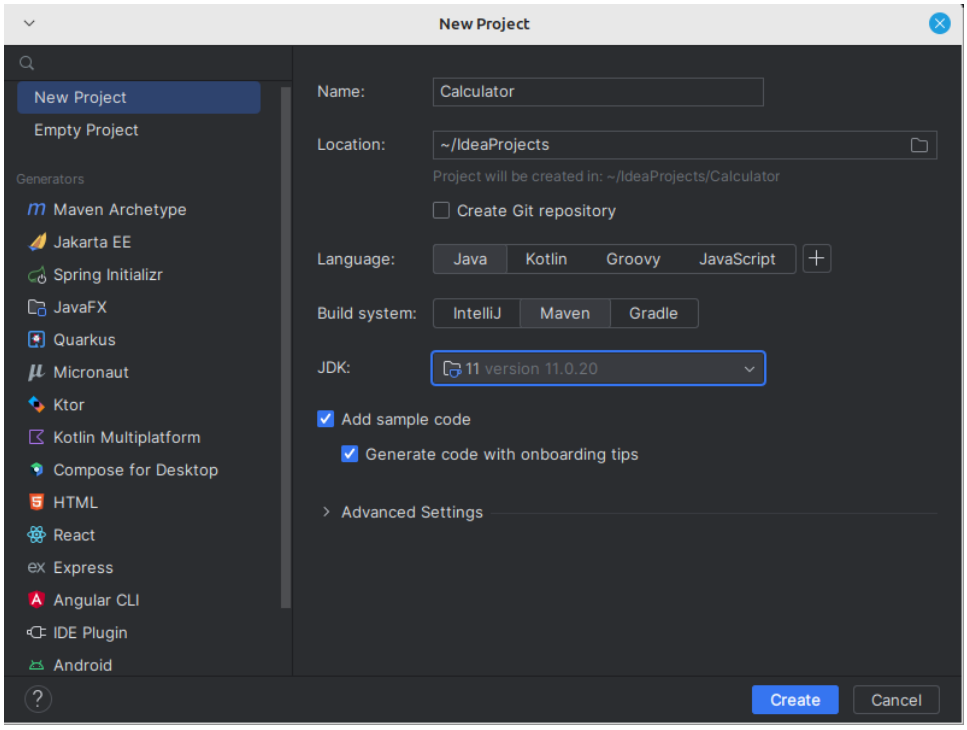

Fig 3.2 New Project Details

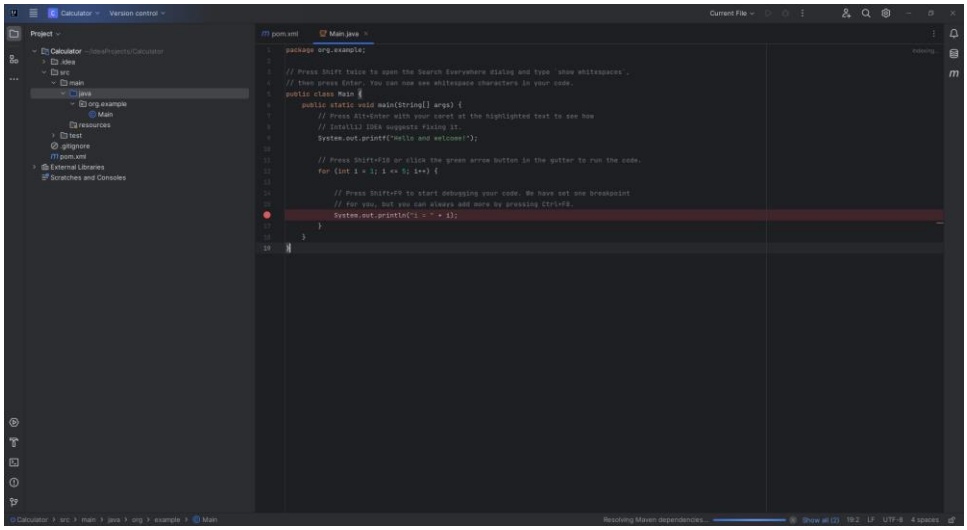

Fig 3.3 Project opened in IDE

For now, we are going to leave the boiler plate code as it is and focus on getting the DevOps part of the project up and rolling. The first thing to configure is **Maven** which will handle building the project, running test cases, and generating JAR files of the project. We first verify if Maven is installed or not using the following command:

```
mvn --version
```

If Maven is installed, then the appropriate version will be displayed. Else you will get an output like the following:

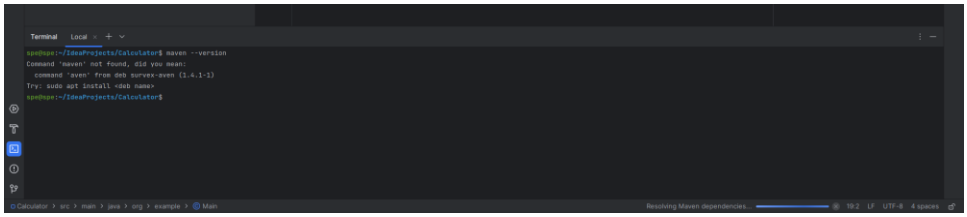

Fig 3.4 Maven not installed

To install Maven, we will use the following command:

```
sudo apt install maven
```

Fig 3.5 Installing Maven

Once we install Maven and check for the version we will see:

```
spe@spe:~/IdeaProjects/Calculator$ mvn --version
Apache Maven 3.6.3
Maven home: /usr/share/maven
Java version: 11.0.20.1, vendor: Ubuntu, runtime: /usr/lib/jvm/java-11-openjdk-amd64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "5.15.0-76-generic", arch: "amd64", family: "unix"
```

Fig 3.6 Maven installed and version

Then we are going to run the following commands:

```
mvn clean

mvn compile

mvn install
```

Once we have Maven, we are going to clean, compile and install the project. **Maven clean** will remove the target folder and make sure that your compile will be the latest file. **Maven compile** will compile the project and the test cases to make sure there are no errors and then **maven install** will create the JAR file.

When you run any maven command for the first time, you might see that it will download a lot of necessary dependency like the following figure. Maven will download the necessary dependencies for each command so this can repeat for one or more of the commands that you try to run for the first time:

```
spe@spe:~/IdeaProjects/Calculator$ mvn clean
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.example:Calculator >----------------------
[INFO] Building Calculator 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.pom (3.9 kB at 691 B/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-plugins/22/maven-plugins-22.pom (13 kB at 74 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/21/maven-parent-21.pom (26 kB at 116 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/apache/10/apache-10.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/apache/10/apache-10.pom (15 kB at 70 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.jar
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/plugins/maven-clean-plugin/2.5/maven-clean-plugin-2.5.jar (25 kB at 138 kB/s)
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ Calculator ---
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-plugin-api/2.0.6/maven-plugin-api-2.0.6.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-plugin-api/2.0.6/maven-plugin-api-2.0.6.pom (1.5 kB at 12 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven/2.0.6/maven-2.0.6.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven/2.0.6/maven-2.0.6.pom (9.0 kB at 66 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/5/maven-parent-5.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/maven/maven-parent/5/maven-parent-5.pom (15 kB at 91 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/apache/apache/3/apache-3.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/apache/apache/3/apache-3.pom (3.4 kB at 20 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.0/plexus-utils-3.0.pom
Downloaded from central: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.0/plexus-utils-3.0.pom (4.1 kB at 34 kB/s)
Downloading from central: https://repo.maven.apache.org/maven2/org/sonatype/spice/spice-parent/16/spice-parent-16.pom
```

Fig 3.7 Maven dependencies being downloaded first time

```
spe@spe:~/IdeaProjects/Calculator$ mvn clean
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.example:Calculator >----------------------
[INFO] Building Calculator 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-clean-plugin:2.5:clean (default-clean) @ Calculator ---
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  0.805 s
[INFO] Finished at: 2023-08-31T19:43:49+05:30
[INFO] ------------------------------------------------------------------------
```

Fig 3.8 Running Maven clean command

```
spe@spe:~/IdeaProjects/Calculator$ mvn compile
[INFO] Scanning for projects...
[INFO]
[INFO] ----------------------< org.example:Calculator >----------------------
[INFO] Building Calculator 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ Calculator ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ Calculator ---
[INFO] Nothing to compile - all classes are up to date
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  2.262 s
[INFO] Finished at: 2023-08-31T19:45:58+05:30
[INFO] ------------------------------------------------------------------------
spe@spe:~/IdeaProjects/Calculator$
```

Fig3.9 Running Maven compile command

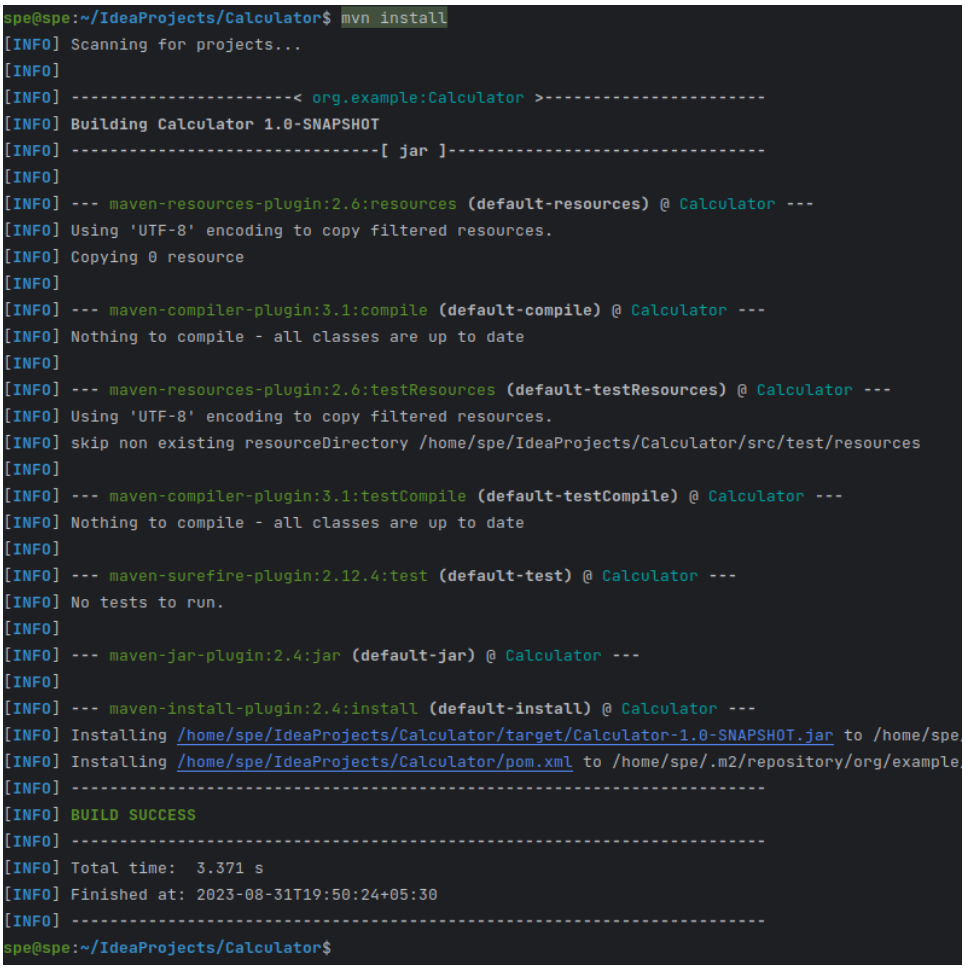

Fig 3.10 Running Maven install command

Now that we have run the compile and install Maven commands, we will be able to see the output JAR file in the **Target** subdirectory of the project:

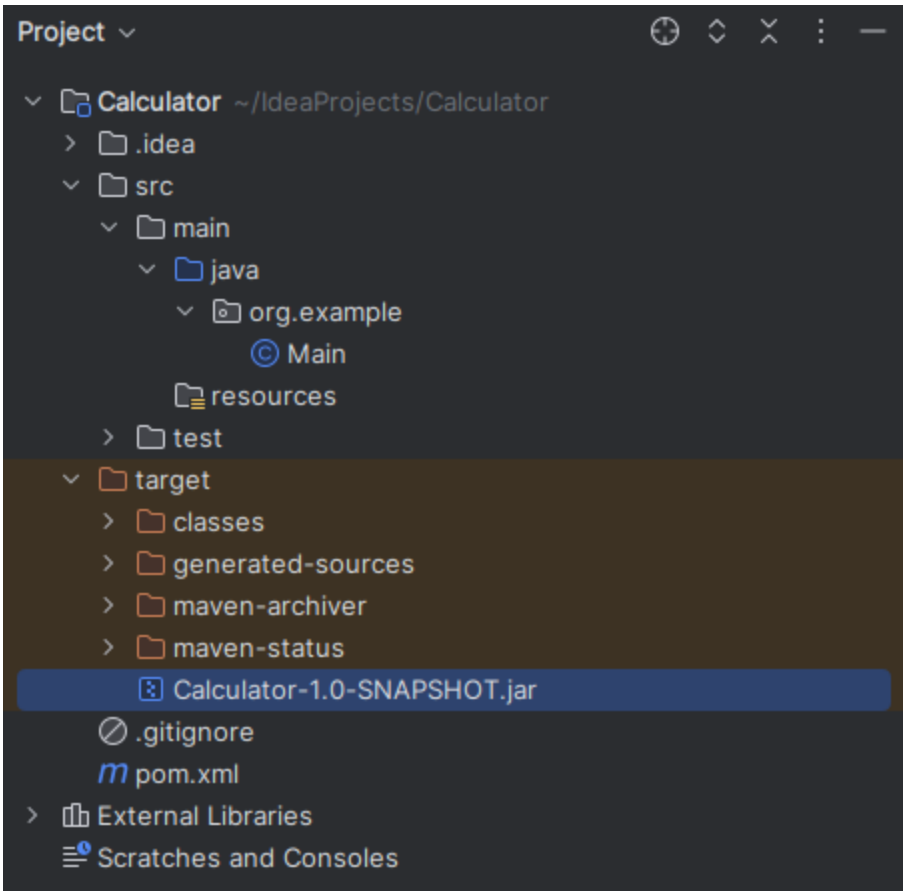

Fig 3.11 The output JAR file in the Target subdirectory

The generated JAR file defaults to a name that is predefined based on the project name that we have given along with the keyword SNAPSHOT. We are going to make some changes to the pom.xml file to create a JAR file with a different name. This is done to ensure that the target JAR file has a constant name which we set. This allows us to specify which JAR file we want to use for our deployment. To achieve this, we open the pom.xml and add the following XML within the <project>...</project> tags:

```
<build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>3.3.0</version>
                <executions>
                    <execution>
                        <phase>package</phase>
```

```
                        <goals>
                            <goal>single</goal>
                        </goals>
                        <configuration>
                            <archive>
                                <manifest>

<mainClass>org.example.Main</mainClass>
                                </manifest>
                            </archive>
                            <descriptorRefs>
                                <descriptorRef>jar-with-
dependencies</descriptorRef>
                            </descriptorRefs>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
```

Your pom.xml should something like this before:

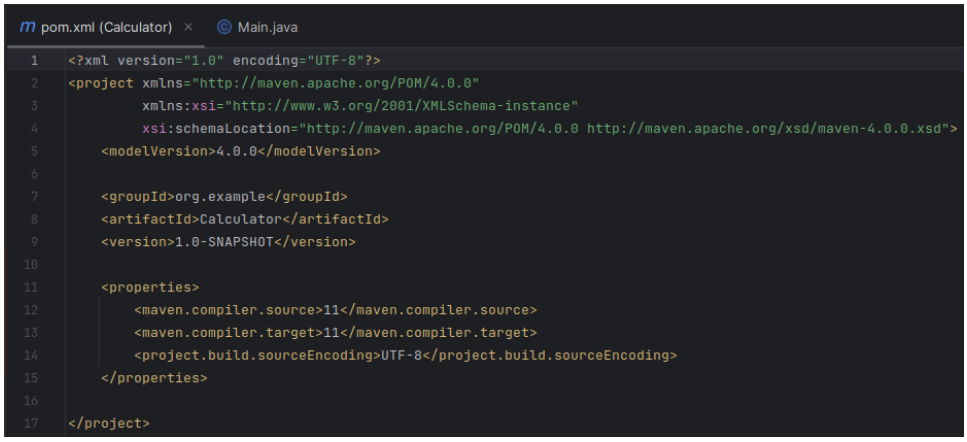

Fig 3.12 pom.xml before adding the build XML

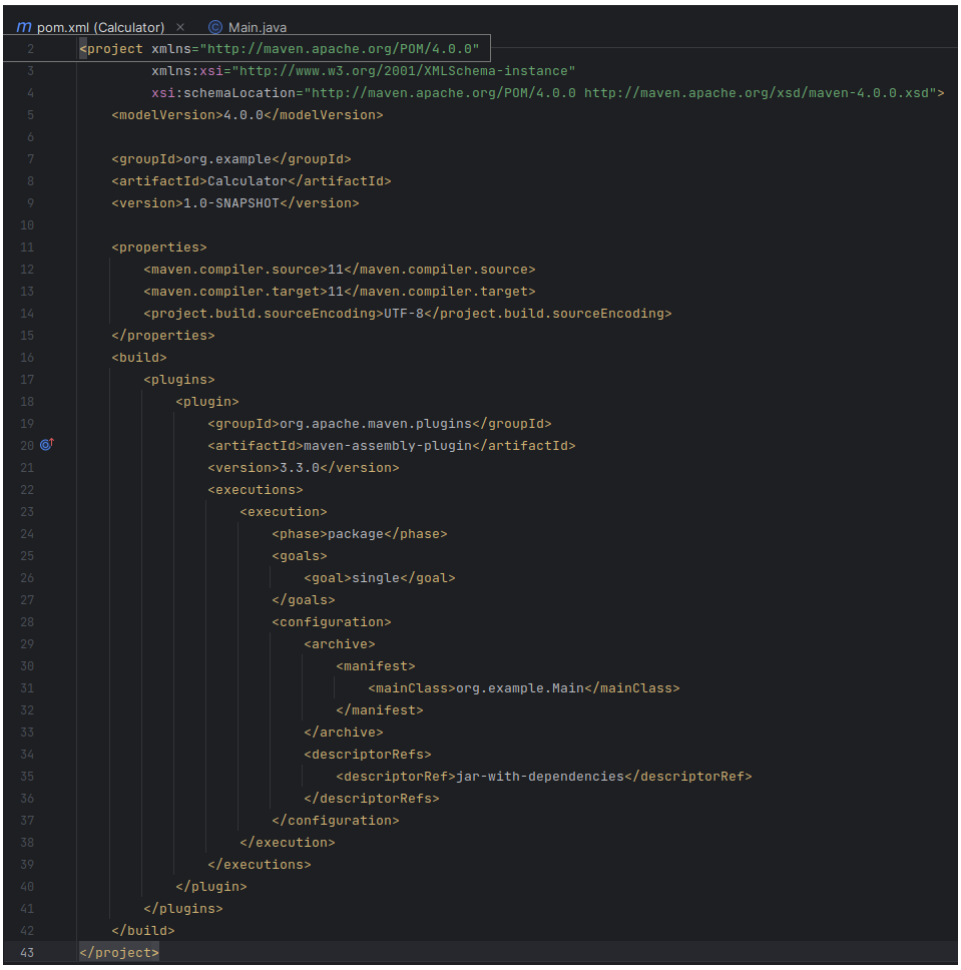

Fig 3.13 pom.xml after adding the build XML

Note that when you add the XML into pom.xml you might see some errors highlighted by the IDE. You must click on the *Load Maven Changes* icon to resolve it. Remember this every time you make any changes to pop.xml:

Fig 3.14 Load Maven Changes Icon

The reason we added this was the following snippet inside the XML:

```
<descriptorRef>jar-with-dependencies</descriptorRef>
```

When we now run the 3 maven commands again, we see the JAR file name is:

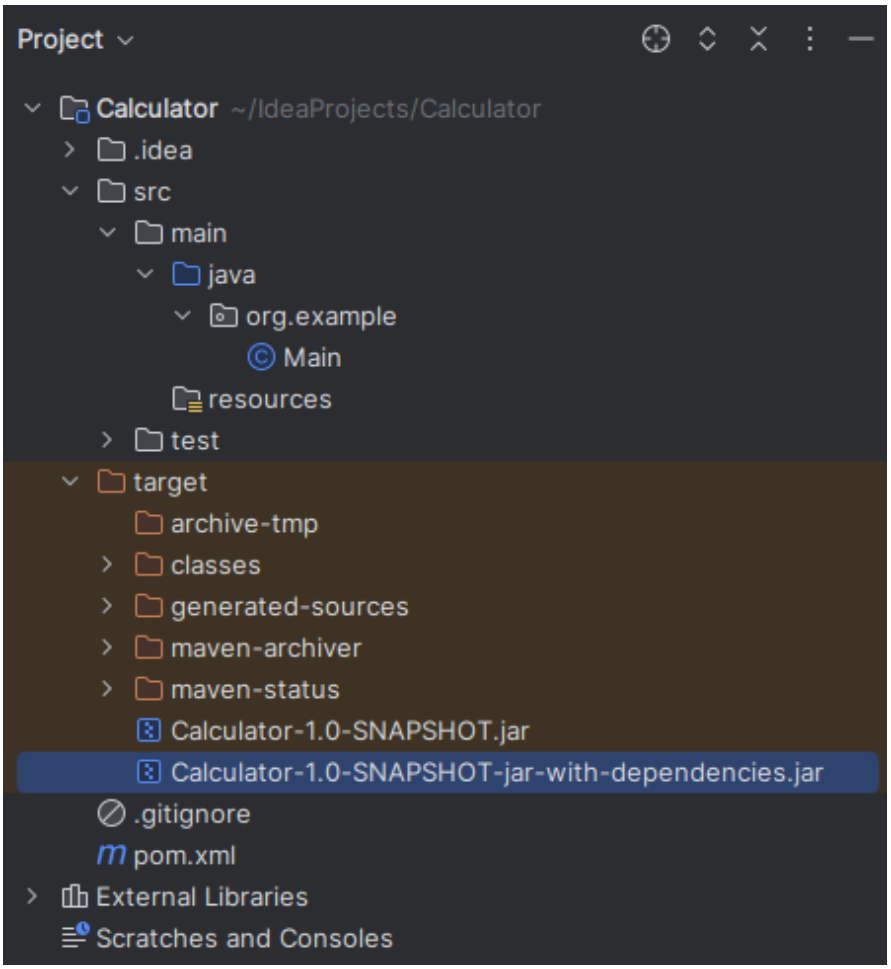

Fig 3.15 JAR file with the description name

We see that the string that we put inside the <descriptorRef>... </descriptorRef> tag now gets appended to the file name and generates a seperate JAR file. This file will be the one that we will be using for deployment, not the autogenerated JAR file.

Also, the following XML snippet:

```
<archive>
    <manifest>
        <mainClass>org.example.Main</mainClass>
    </manifest>
</archive>
```

This XML is used to correctly target the class that contains Main. This should match the project package that you set up when you created the project. Our project is **org.example.Main** and you can see it in the directory structure as well:

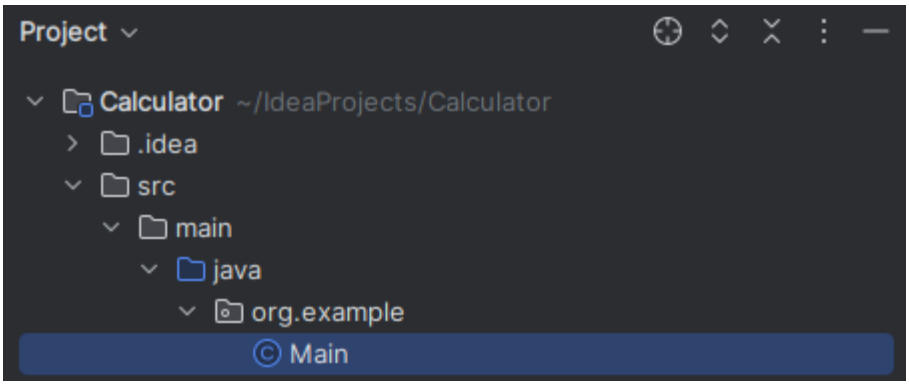

Fig 3.16 Directory structure matching the XML

The explicit mentioning of the class will ensure that we will not face the class not found exception when Maven is compiling the code.

Finally, to ensure that the JAR file works, we will run the file and expect to the output of the placeholder code in the main methods.

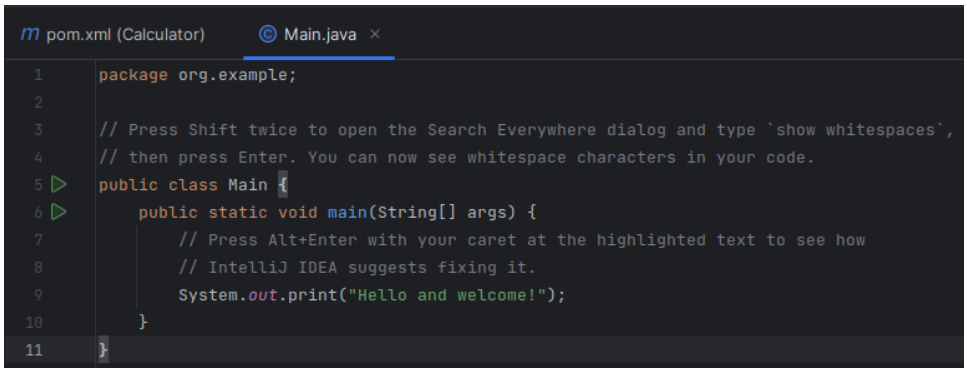

Fig 3.17 Placeholder code in main method

To run the JAR file, we use the following command:

```
cd target

java -jar <projectname-modified>.jar
```

In our case, the JAR file name will be **Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar**

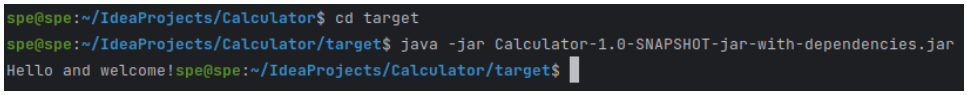

Fig 3.18 Output when running the JAR file

As you can see, the output prints the string that was there in the code. So, we can rest assured that Maven is successfully working.

Adding Version Control using Git and GitHub

Now that we have a base project that is working, we are going to add version controlling. This will be handled by Git and will allow us to work on branches, commit changes and rollback if something goes wrong. We are also going to use GitHub to host our repository remotely so that Jenkins later will be able to fetch changes remotely decoupling our development from deployment.

For starters, we will first ensure that we are in the root directory of the project. We used the command **cd target** to enter the subdirectory that contained the JAR file. Hence, we will use **cd ..** to return to the root directory of the project.

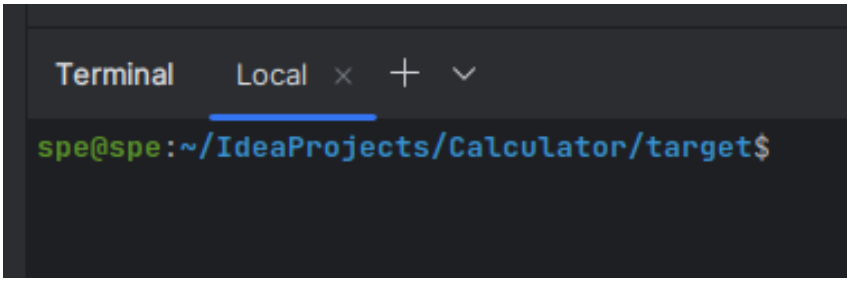

Fig 3.19 Wrong Directory

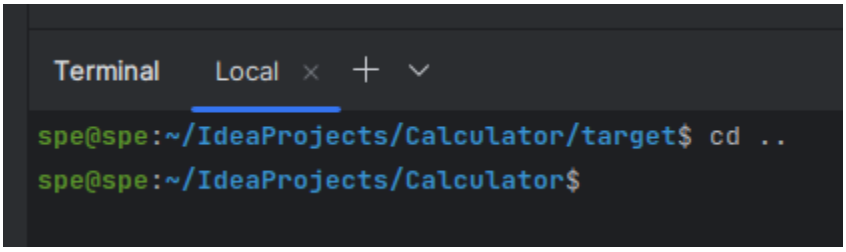

Fig 3.20 Right Directory

In the root directory of the project, we are going to initialise git. This will make the entire project into a local repository that has version control handled by git. To do this, we run the following command:

```
git init
```

You will see the following output once the initialisation is complete:

```
spe@spe:~/IdeaProjects/Calculator$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/spe/IdeaProjects/Calculator/.git/
spe@spe:~/IdeaProjects/Calculator$
```

Fig 3.21 Initialising git for the project

By default, we are not set to be working on the master branch of the project. Keep in mind the name of the branch, **master**. The next step is to create a remote repository where we can push the changes to. We start by logging into our GitHub account or creating one if you don't have one:

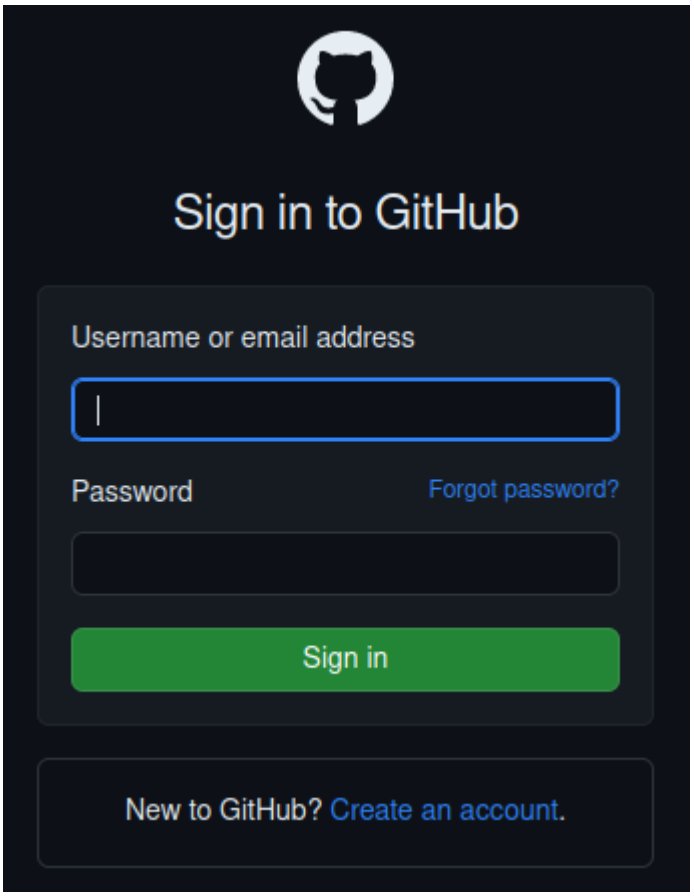

Fig 3.22 Logging into GitHub

Once you log into GitHub, we are going to create a new empty repository. Start by clicking on the New button on the top left:

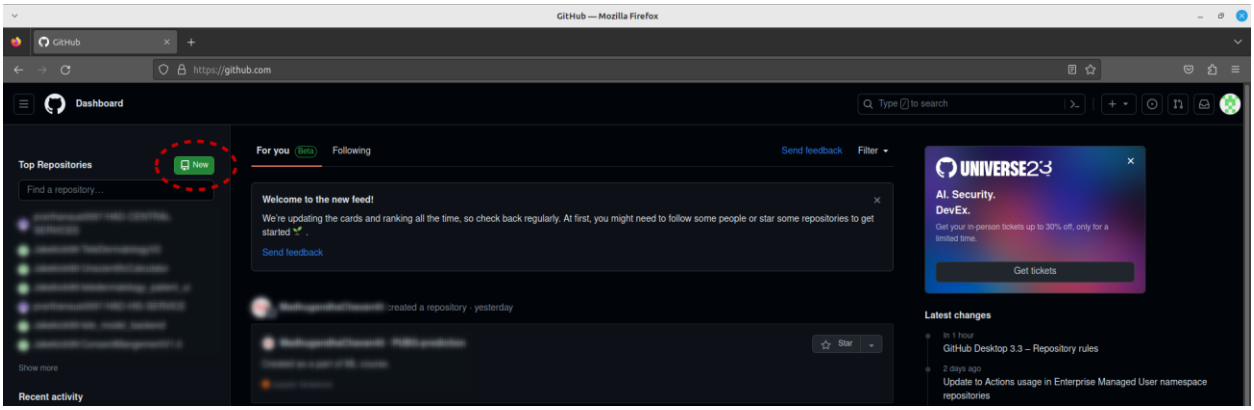

Fig 3.23 Dashboard of your GitHub

In the repository creation page, we are going to name our repository **Calculator** with not description for the repository. We are also going to leave our repository **Public**. *Note that we are not selecting any other options like Readme or license.*

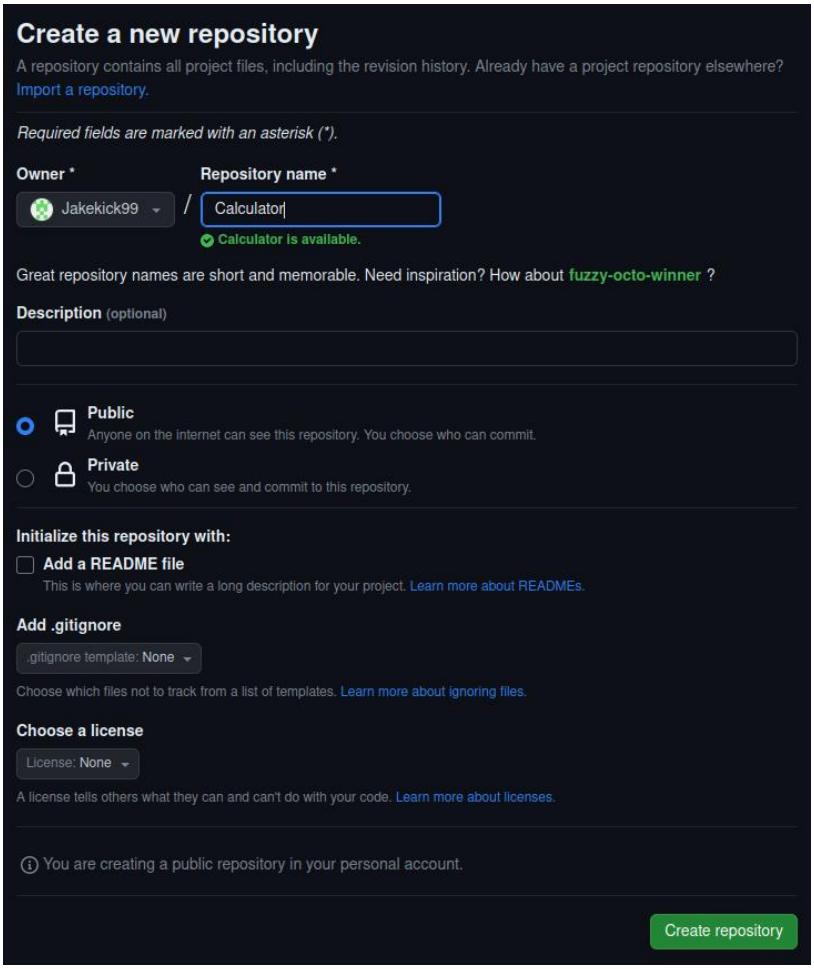

Fig 3.24 Repository creation

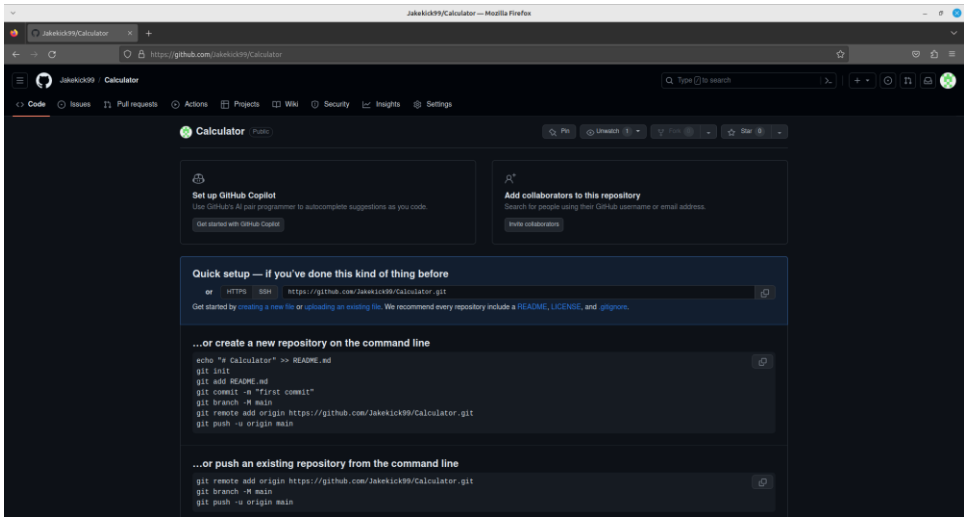

Fig 3.25 Created Repository

Once we have created the remote repository, we are switching to the IDE and in the terminal, we are going to run the following command in the same directory that we initialised git in:

```
git add .
```

This command will stage all the files that in the root directory for commit. This will however ignore the files that are listed in the .**gitignore file**. This is to ensure that compiled files and library inclusions are not pushed into the remote repository to save space as well as avoid conflicts with other developers. This was created when we ran the command **git init**. Before we do git add, if we perform git status we will get the following output:

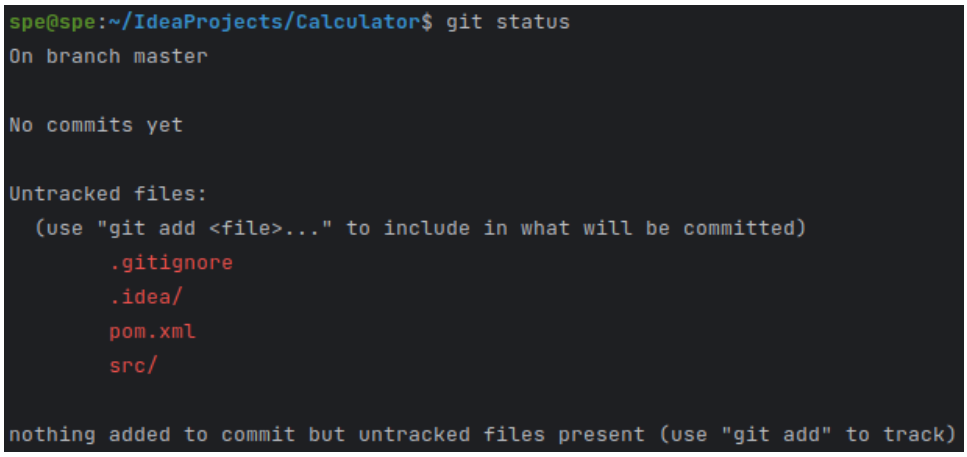

Fig 3.26 Git status before staging files

Once we run the git add command, we will see that all the files become green, which indicates that the files are all staged and ready to be committed.

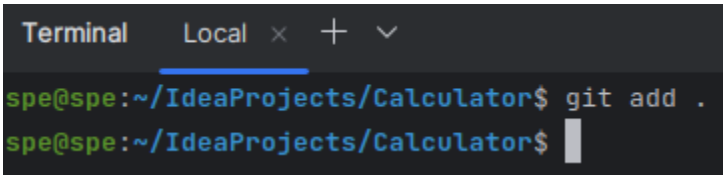

Fig 3.27 Executing git add command

```
spe@spe:~/IdeaProjects/Calculator$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   .idea/.gitignore
        new file:   .idea/encodings.xml
        new file:   .idea/misc.xml
        new file:   .idea/vcs.xml
        new file:   pom.xml
        new file:   src/main/java/org/example/Main.java

spe@spe:~/IdeaProjects/Calculator$
```

Fig 3.28 Seeing all the files staged

Once the files are staged and show up as green when performing git status, we can now commit the changes using the git commit command. However, the first time you try to run git commit you will likely face the following error:

```
spe@spe:~/IdeaProjects/Calculator$ git commit -m "Initial Commit"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'spe@spe.(none)')
spe@spe:~/IdeaProjects/Calculator$
```

Fig 3.29 Mail address not set

In case you don't it is still advised to manually set up your email and username to match the ones in your GitHub account. You can set it using the following commands:

```
git config --global user.email "Your_Email_ID@Domain.com"
git config --global user.name "Your_Username_Here"
```

```
spe@spe:~/IdeaProjects/Calculator$ git config --global user.email "[redacted]"
spe@spe:~/IdeaProjects/Calculator$ git config --global user.name "Jakekick99"
spe@spe:~/IdeaProjects/Calculator$
```

Fig 3.30 Setting your username and email

Now that your email and name is setup, we can go ahead and commit using the git commit command. The syntax of the command is as follows:

```
git commit –m <Your commit message here>
```

The commit message will be visible when you check the commit history. Keep it meaningful to what changes you are committing.

```
spe@spe:~/IdeaProjects/Calculator$ git commit -m "Initial Commit"
[master (root-commit) b6a035c] Initial Commit
 7 files changed, 127 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 .idea/.gitignore
 create mode 100644 .idea/encodings.xml
 create mode 100644 .idea/misc.xml
 create mode 100644 .idea/vcs.xml
 create mode 100644 pom.xml
 create mode 100644 src/main/java/org/example/Main.java
spe@spe:~/IdeaProjects/Calculator$
```

Fig 3.31 Committing your changes in git

Now we have a local repository where we have committed changes. We also have a remote repository in GitHub where we want to store these as well. So, we now need to push our local commits to the remote repository.

To do this, initially all you had to do was use the git push command and then supply your remote repository account username and password. However, for the sake of security, GitHub has stopped accepting passwords in command line and instead require you to use a **Personal Access Token**. So, we are going to create one now.

To do this, we click on our profile in the top right from anywhere in GitHub. This will show a drop down from where we can click on Settings

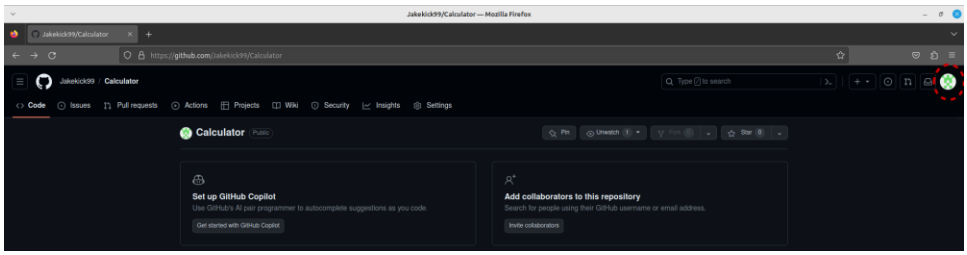

Fig 3.32 Profile Dropdown location

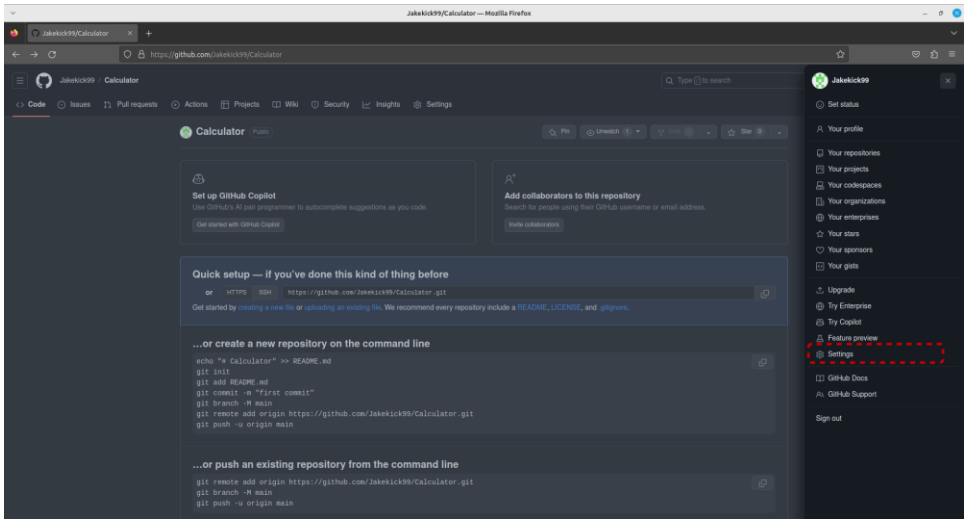

Fig 3.33 Profile Settings

Once in the profile settings, scroll down and select the **Developer Options** from the Left menu:

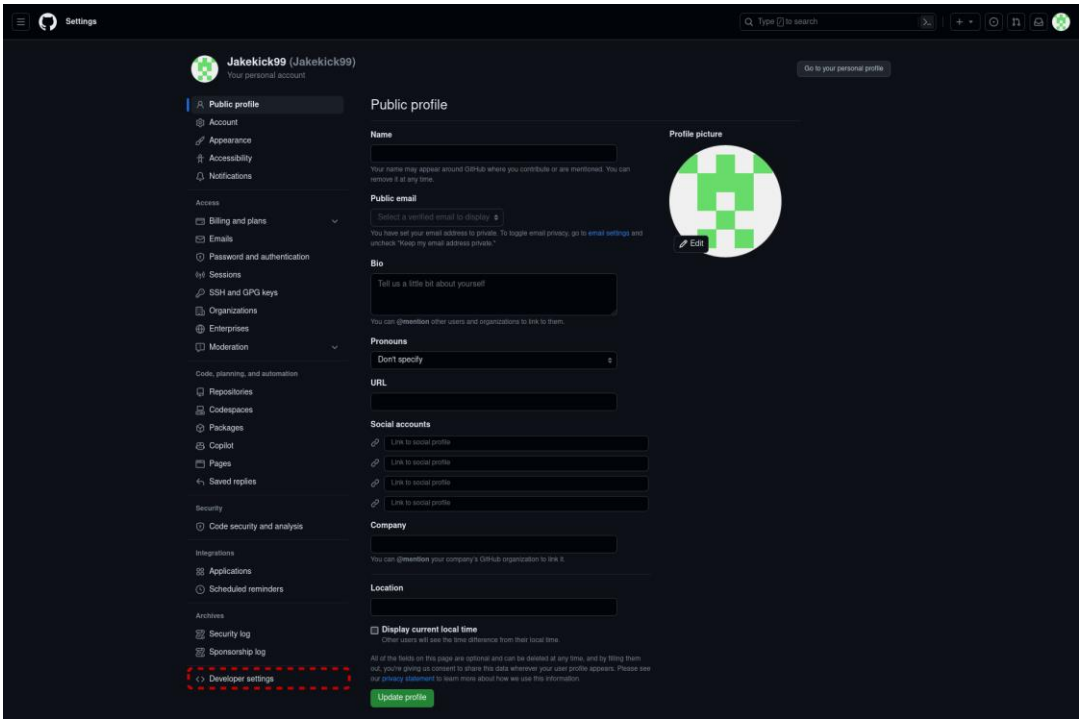

Fig 3.34 Developer Settings

In developer settings, click on **Personal Access Tokens** and then on **Tokens (classic)**

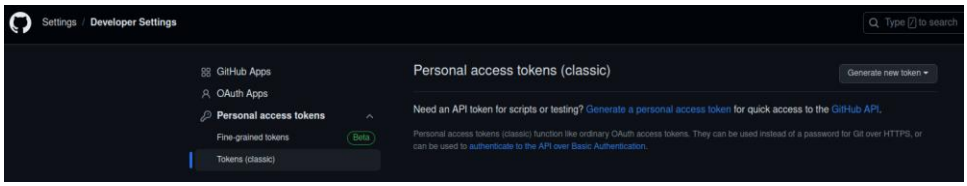

Fig 3.35 Personal Access Tokens

In the top right drop down, click on Generate New Token and choose the **Generate new Token (classic)** from the drop down:

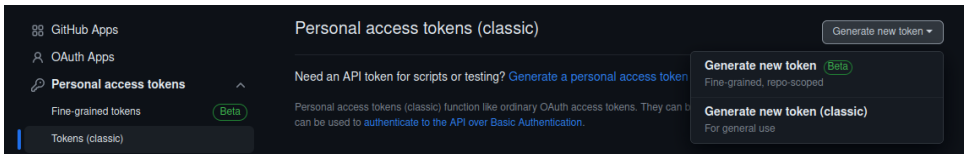

Fig 3.36 Generate new classic token

Now you could generate a new token from the beta option or even set specific permissions in the classic token too. Ideally, we should specify what all permissions we want our token to have. However, for the sake of the demo we are going to create a token will all access permissions. This is not recommended from a security standpoint but keeping the spirit of the demo in mind, we will keep this brief and simple.

We give a note to our token saying calculator and leave expiration to the default 30 days. Then under select scopes we will check all the options. *Checking the parent checkbox will check all child checkboxes*

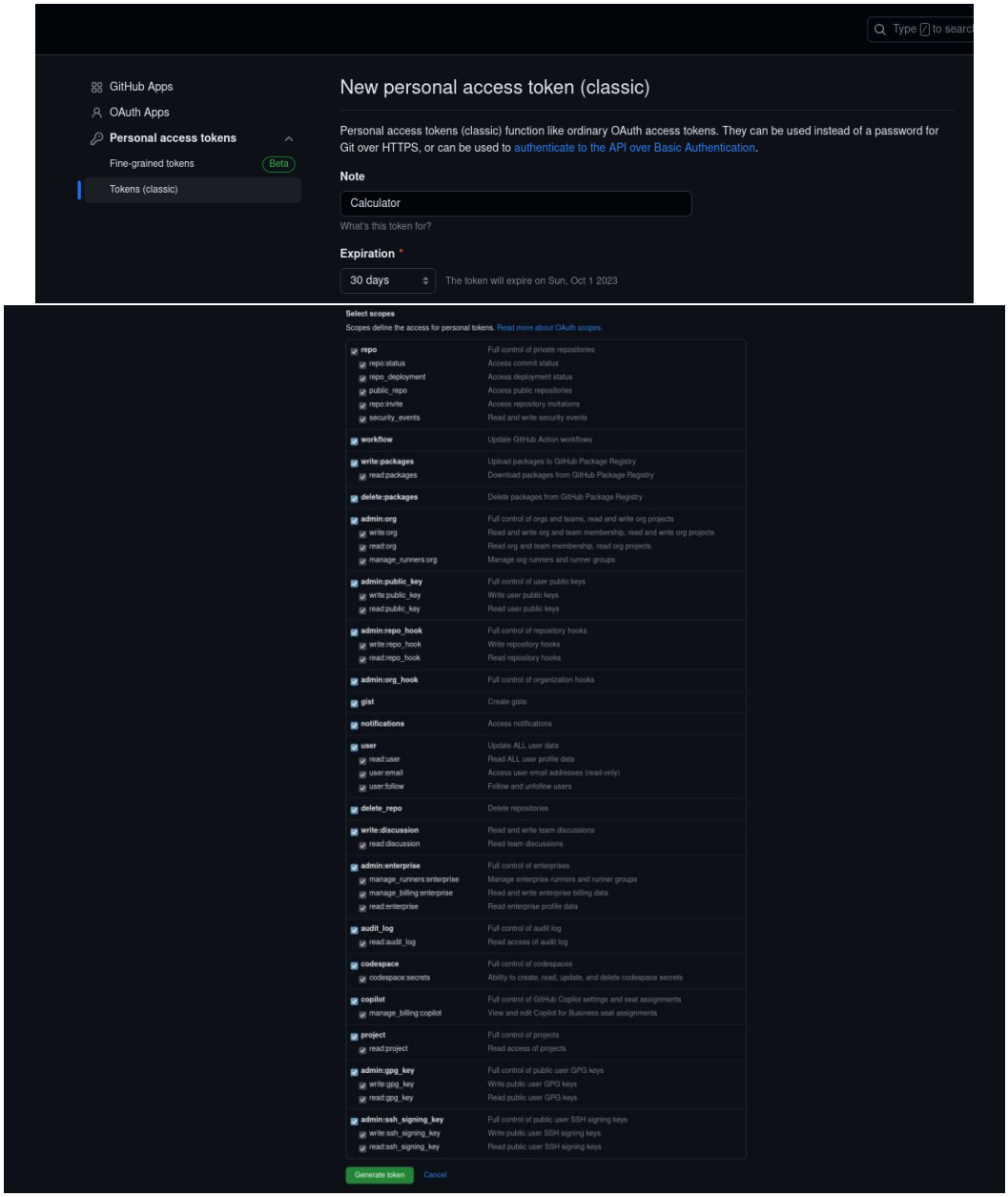

Fig 3.37 Personal Access Token details

Once we have given the note and necessary permissions have been selected, we will click on generate token. This will show the Personal Access Token that has been created. Note that this will be shown only once and needs to be saved. After you navigate out of this page it will not be shown again. If forgotten the only way is to recreate another token.

On a security note, make sure not to disclose your personal access token to anyone. The one used by the book is an expired token. If someone gets a hold of your token and it has sufficient rights, then they can use it malicious reasons.

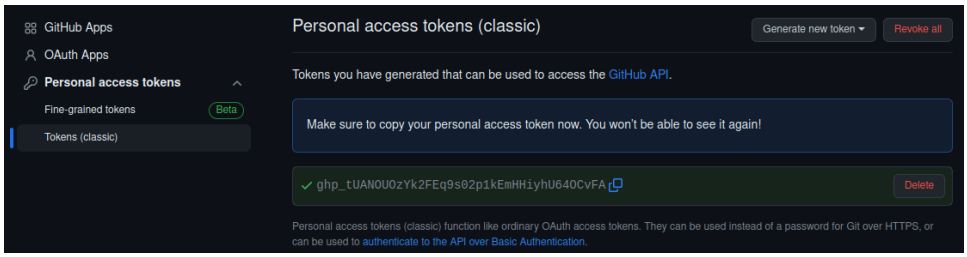

Fig 3.38 Newly created personal access token

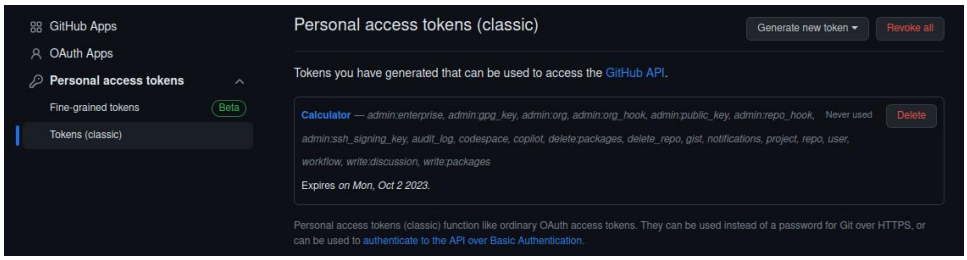

Fig 3.39 When you visit this page again, token is no longer visible

Now that we have the token, we can now push our local changes to the remote repository. We will first link our local repository to the remote repository by adding it as the **origin** of your project. This is done using the following command:

```
git remote add origin <repo-url>
```

We need to next find the branch that we are working on. To do this, we will use the following commands:

```
git branch
```

This shows the current branch with a star (*) next to it.

Once we know the branch as well, we will execute the following command:

```
git push –u origin <branch name>
```

When running, it will ask us for the username which can be the mail id you signed up with on GitHub or your GitHub username. Then it will also ask for your password where you will need to copy paste the personal access token (this might not show up as some editors don't show the password text for security reasons)

```
spe@spe:~/IdeaProjects/Calculator$ git remote add origin https://github.com/Jakekick99/Calculator.git
spe@spe:~/IdeaProjects/Calculator$ git branch
* master
```

Fig 3.40 Add remote origin and checking which branch to push

```
spe@spe:~/IdeaProjects/Calculator$ git push -u origin master
Username for 'https://github.com':  @gmail.com
Password for 'https://  @gmail.com@github.com':
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 4 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (15/15), 2.43 KiB | 1.21 MiB/s, done.
Total 15 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Jakekick99/Calculator.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
spe@spe:~/IdeaProjects/Calculator$
```

Fig 3.41 Pushing the branch to remote repository

Now when we go and check the repository on GitHub, we will see our changes present on the branch named master:

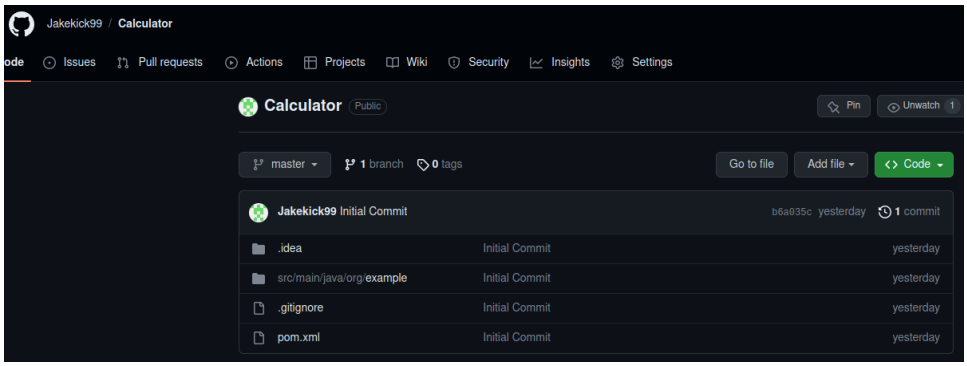

Fig 3.42 Remote repository with pushed changes

With this, we have a functioning local and remote repository system that we can sync our changes with and can be extended for any number of developers that are part of the project.

Creating & Integrating with Jenkins pipeline

Now that our repositories are set up, we are going to create the CI/CD pipeline in Jenkins that will allow us to pull changes and build the project whenever we push changes to the repository. To start off, we will login to Jenkins.

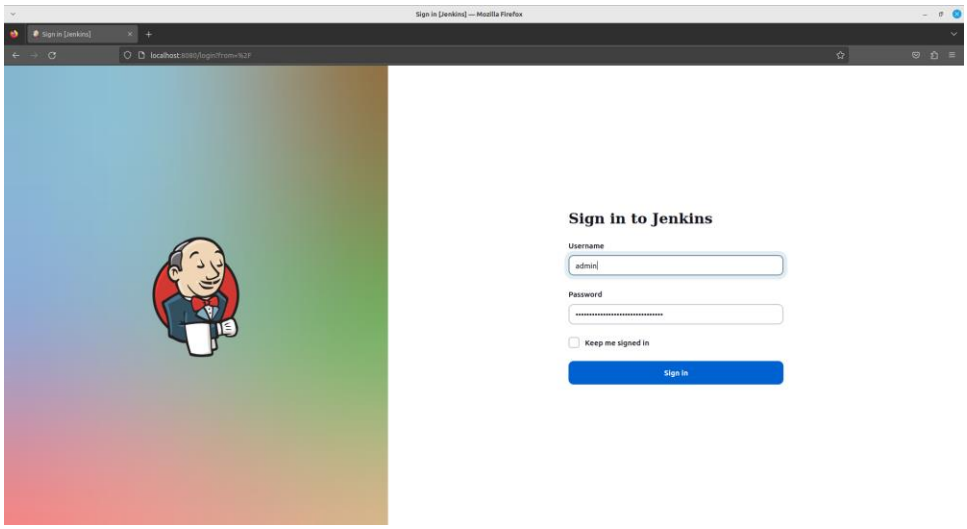

Fig 3.43 Log in to Jenkins

Once we login, we will be greeted with the dashboard. From here we can click on the **New Item** to create our new pipeline.

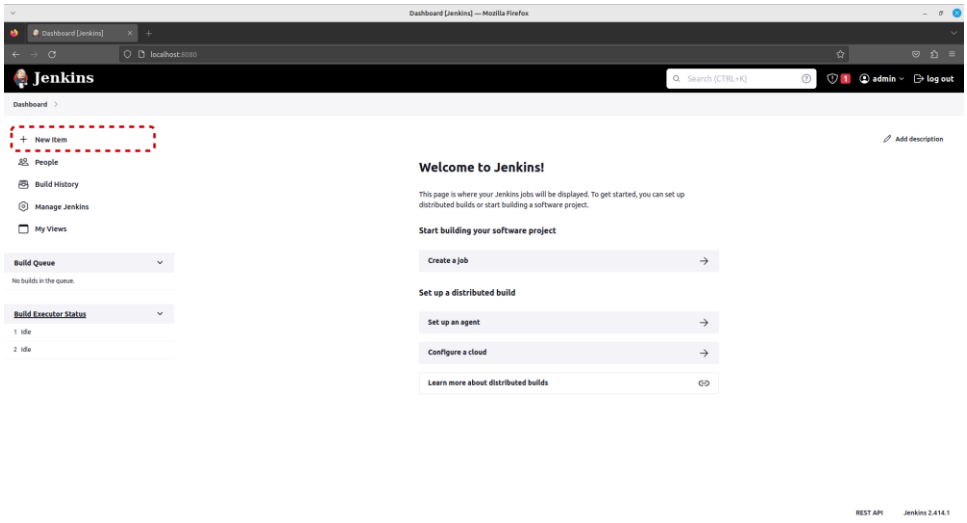

Fig 3.44 Jenkins Dashboard

In the new item creation screen we will name our item as **Calculator** and select the **Pipeline** option.

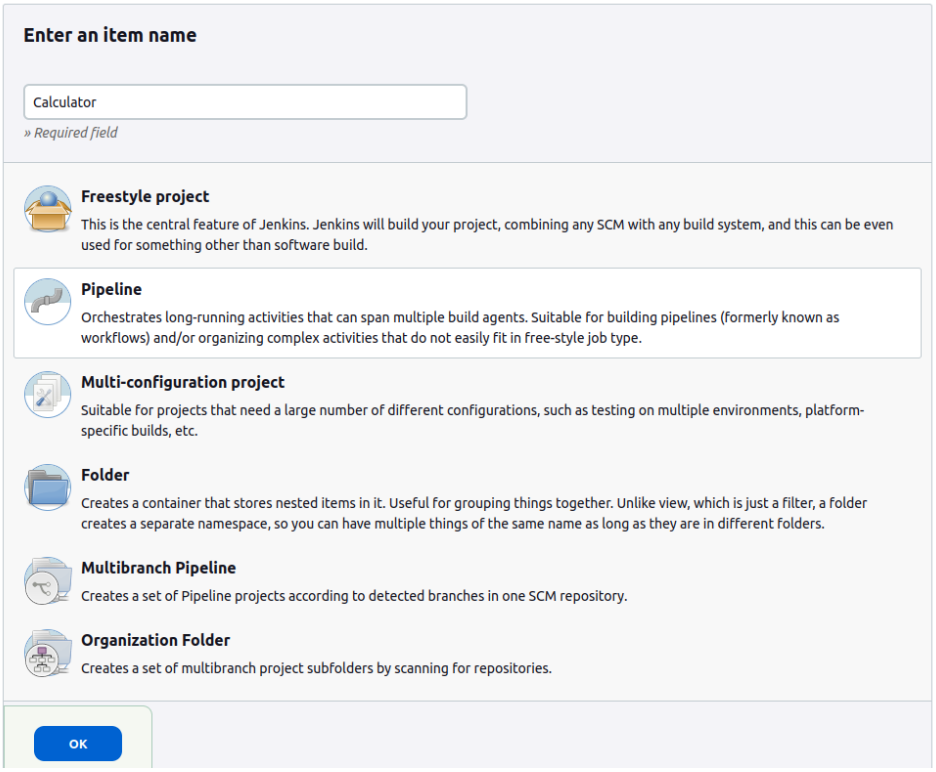

Fig 3.45 New Jenkins Item

Once created, it will ask us to configure the pipeline. We will click on save and keep the defaults for now. This will take us to the dashboard of the pipeline which is quite barren.

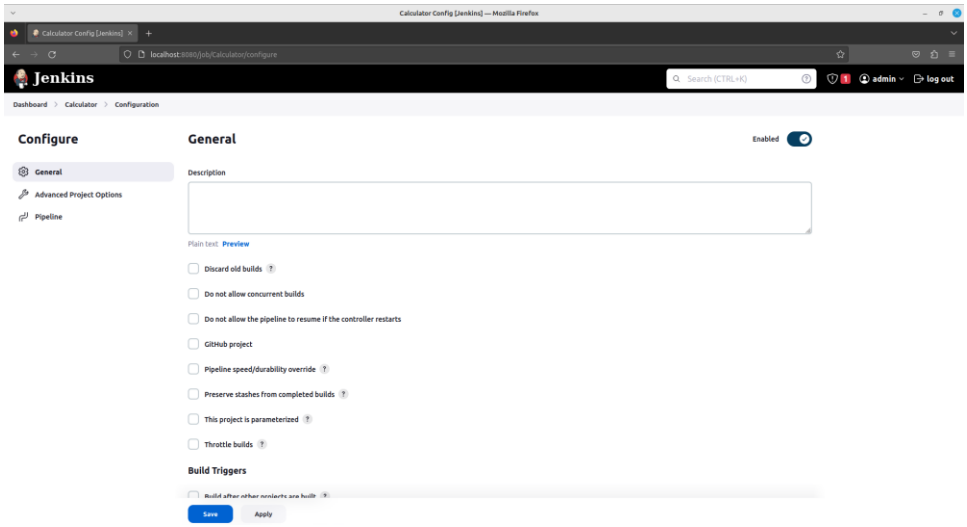

Fig 3.46 Configuring pipeline – keep defaults & save

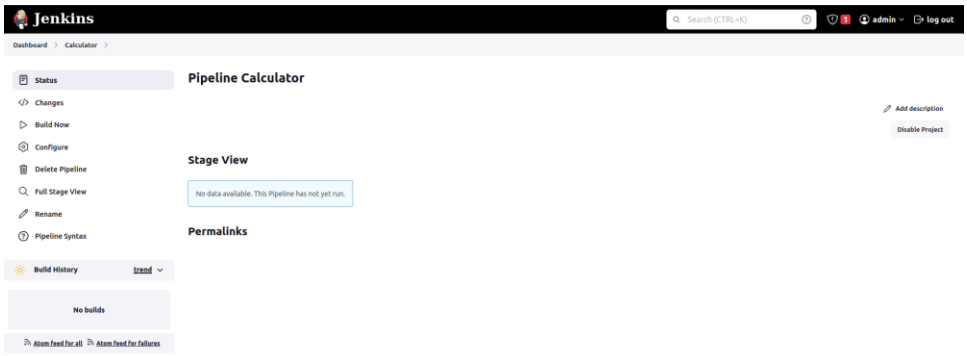

Fig 3.47 Dashboard of your pipeline

Once the pipeline has been created, we will come back here to add the pipeline syntax for every stage that is there. However, there will be unmet dependencies for each stage, so we are going to resolve that by adding all the required plugins in one go. We will go to Dashboard > Manage Jenkins > Manage Plugins and add the necessary plugins by searching them in Available Plugins. The list of plugins that need to be present for this mini project to work apart from the default plugins are:

- Ansible
- Docker
- Docker Pipeline
- Git Client
- Git plugin
- GitHub
- JUnit

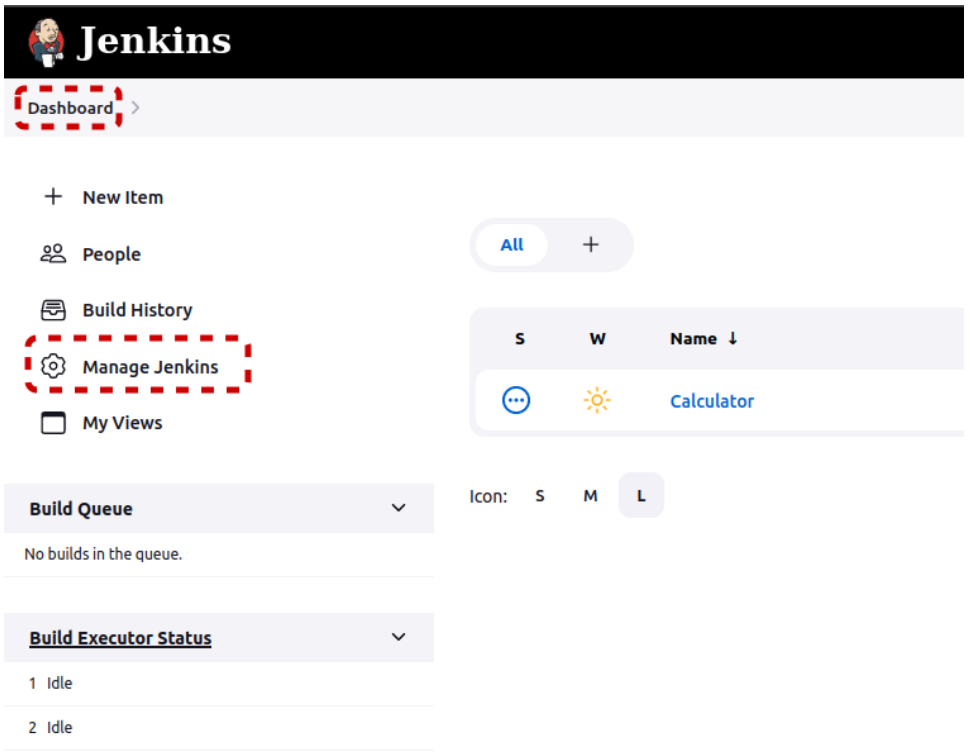

Fig 3.48 Dashboard and Manage Jenkins options

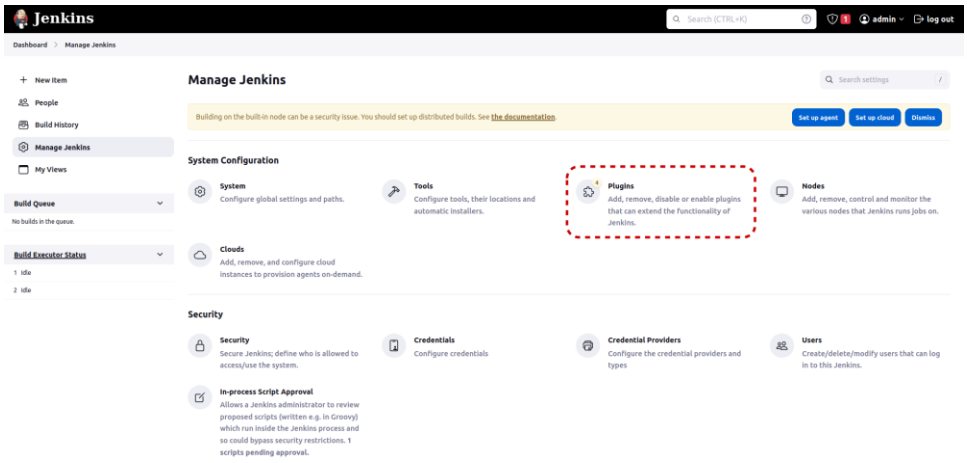

Fig 3.49 Plugins option

Once here, we will search for the above-mentioned Plugins and make sure that they are installed. If they aren't then we will install them. Once they are installed, we can proceed to setting up the pipeline.

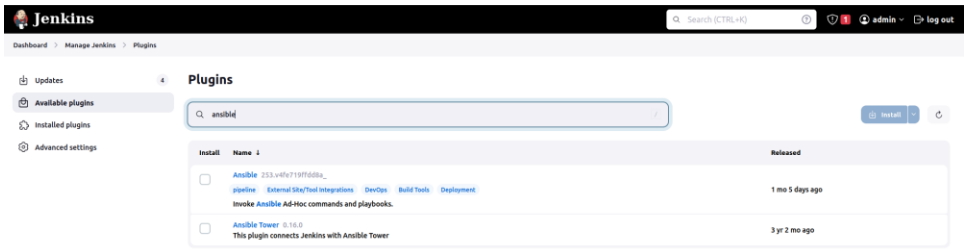

Fig 3.50 Ensuring all plugins are installed

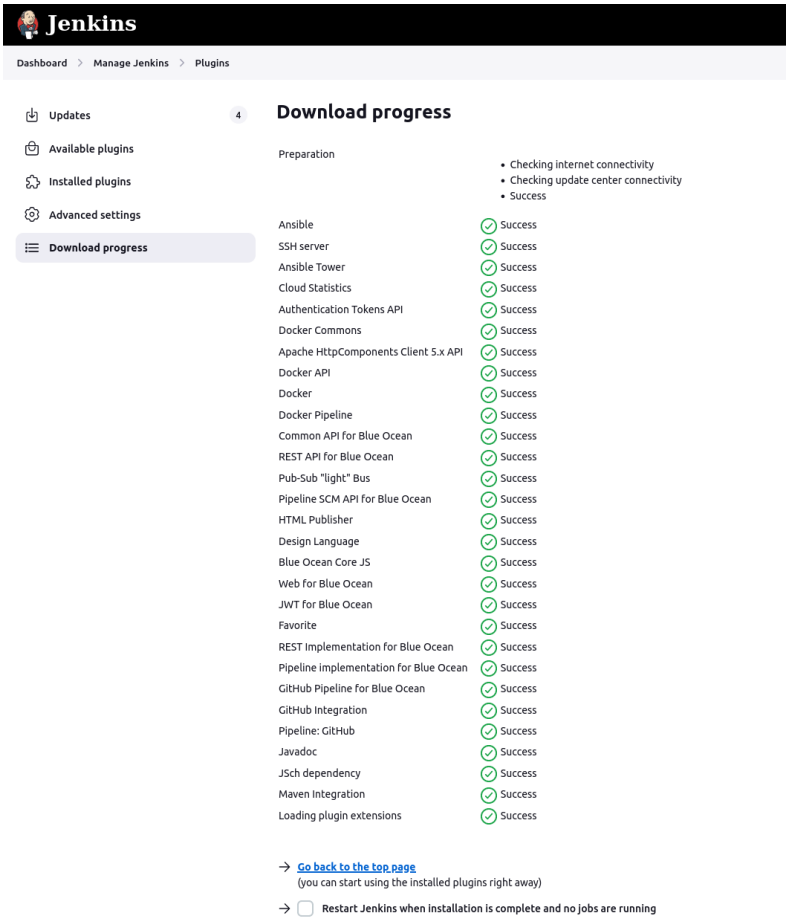

Fig 3.51 Installing missing plugins

Stage 1: Cloning project from GitHub

With the plugins out of the way, we can proceed with creating each step of the pipeline. The first stage of the pipeline deals with cloning the project master branch. Since the pipeline will be triggered by commits to the project, the latest changes are cloned in this step. To set this stage up, we will go to Dashboard > Calculator Pipeline > Configure

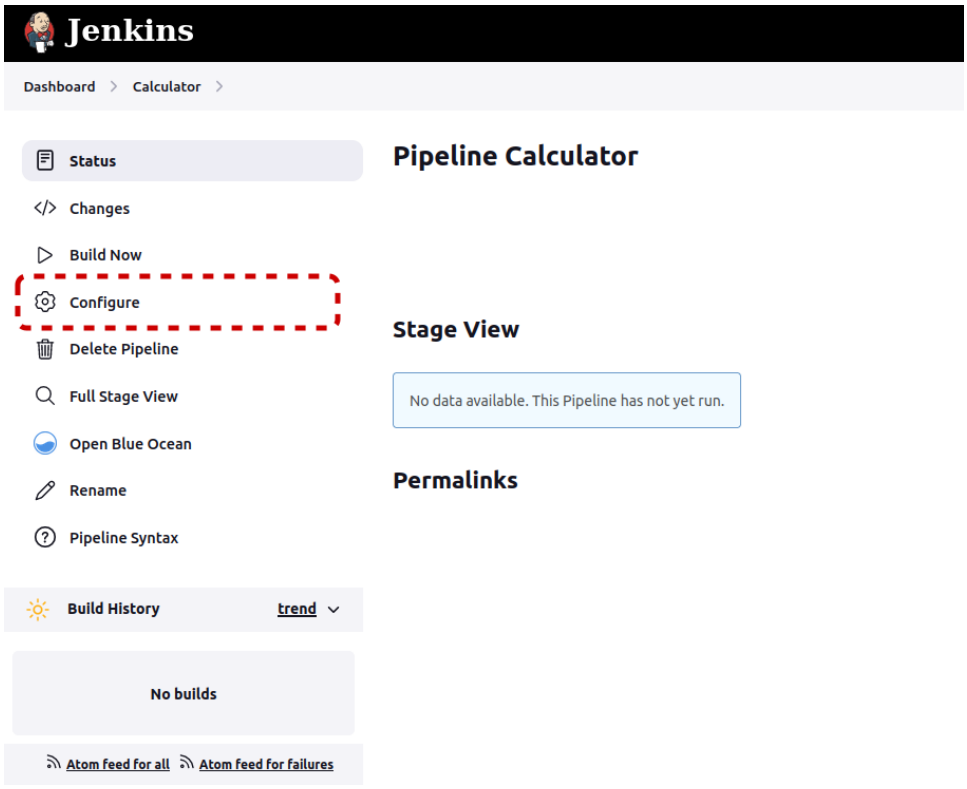

Fig 3.52 Configure option in the pipeline

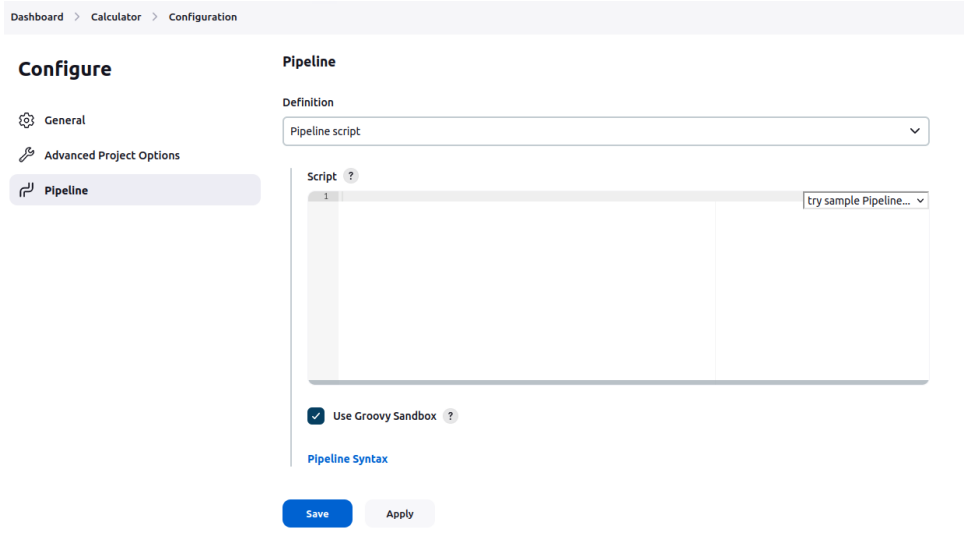

Fig 3.53 Pipeline script edit field

In the pipeline script, we are going to add the first stage of the pipeline with the following script:

```
pipeline{
    environment{
        docker_image = ""
```

```
    }
    agent any
    stages{
        stage('Stage 1: Git Clone'){
            steps{
                git branch: 'master',
url:'https://github.com/Jakekick99/Calculator.git'
            }
        }
    }
}
```

In the above script, the git branch must match the branch in GitHub and the URL must be the git URL to the project from GitHub.

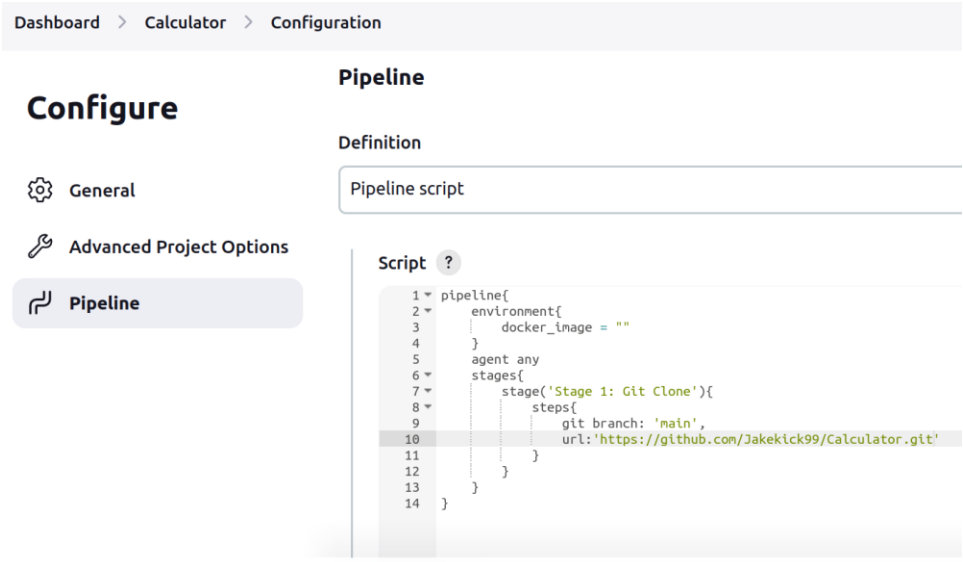

Fig 3.54 Pipeline script after adding stage 1

The script starts with the pipeline object. Within it we specify an environment variable which will be used by the Docker stage later. We also mention as any agent which will run the pipeline on any Jenkins agent that is free. Then we have a stages object where we will specify each stage individually. We add the first stage script into it.

Now if we try and execute the pipeline, but if you get the following error (using the **Build Now)**:

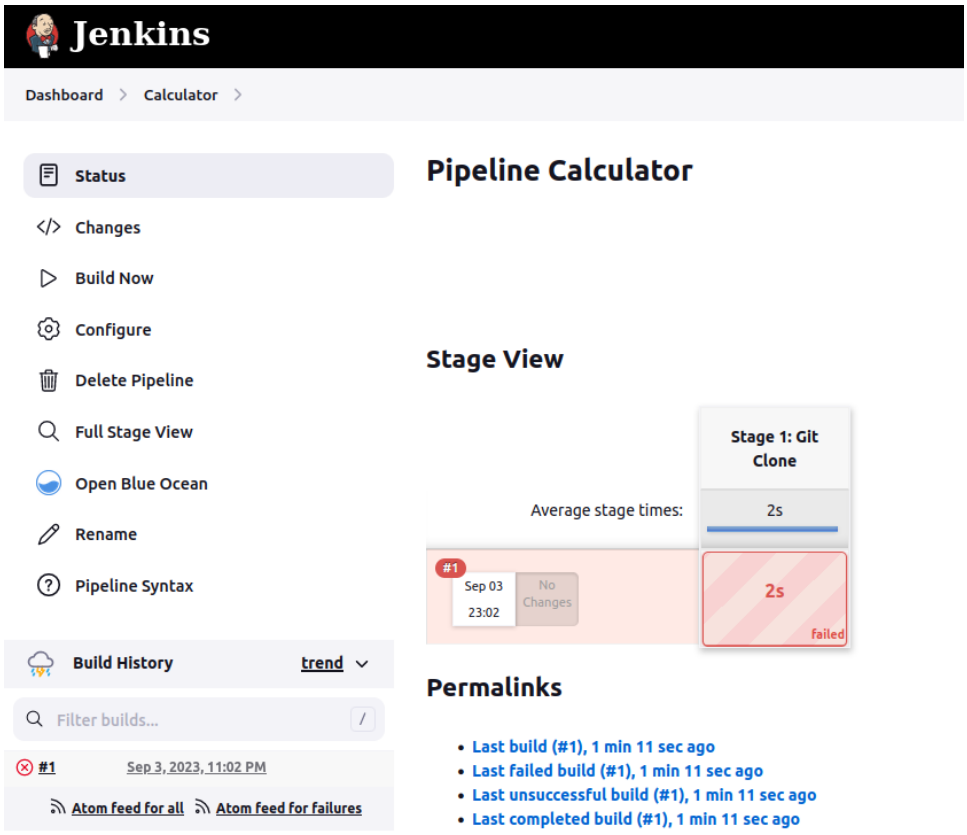

Fig 3.55 Attempting to run the pipeline

If we hover over the stage, we can see that it shows the option to see the logs

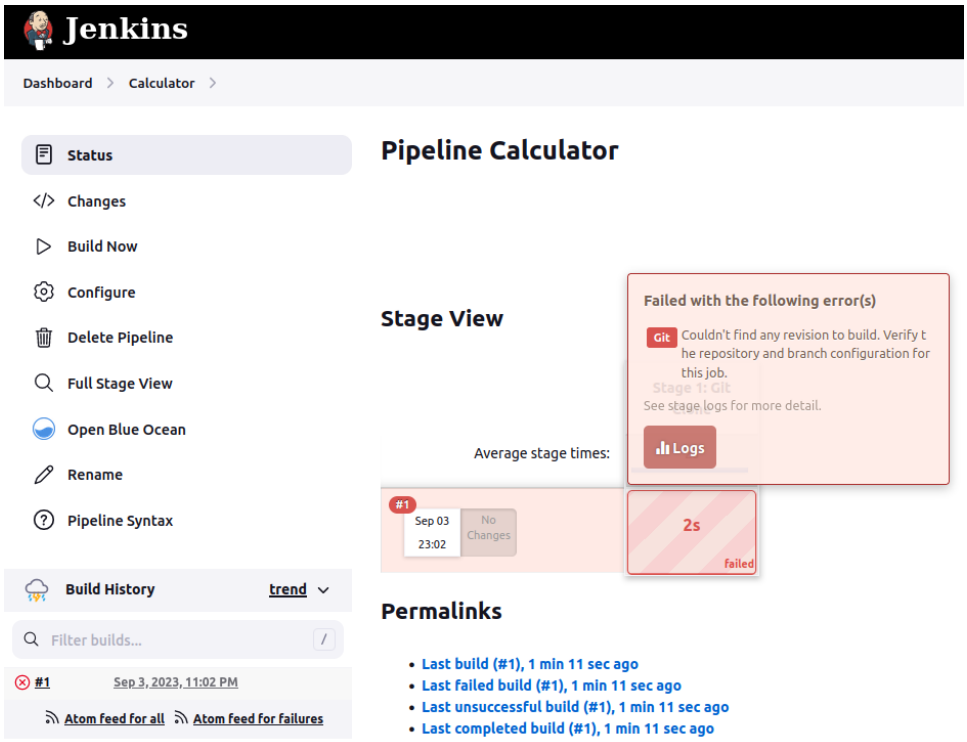

Fig 3.56 Option to see stage logs

When we click on the logs, we can see the console output with the following error:

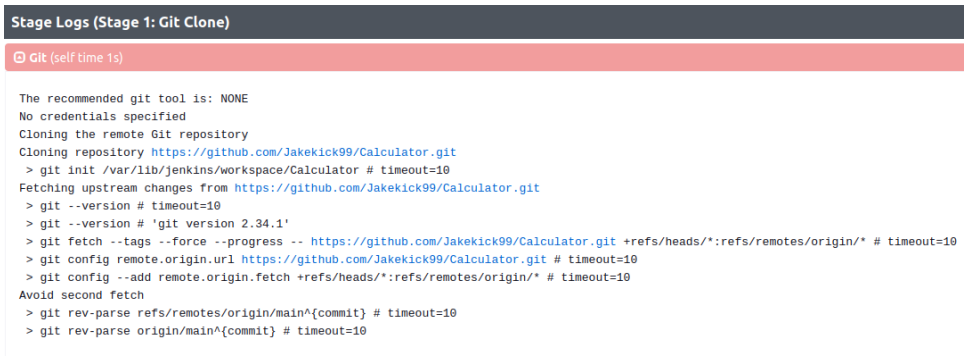

Fig 3.57 Console output for the stage

Double checking, we can see that the script said main while our branch in GitHub is master. This mismatch can lead to the above error. So, make sure that the branch name that you are giving matches the branch name in Git and GitHub.

In case you didn't get an error then congratulations, your first pipeline was created and executed.

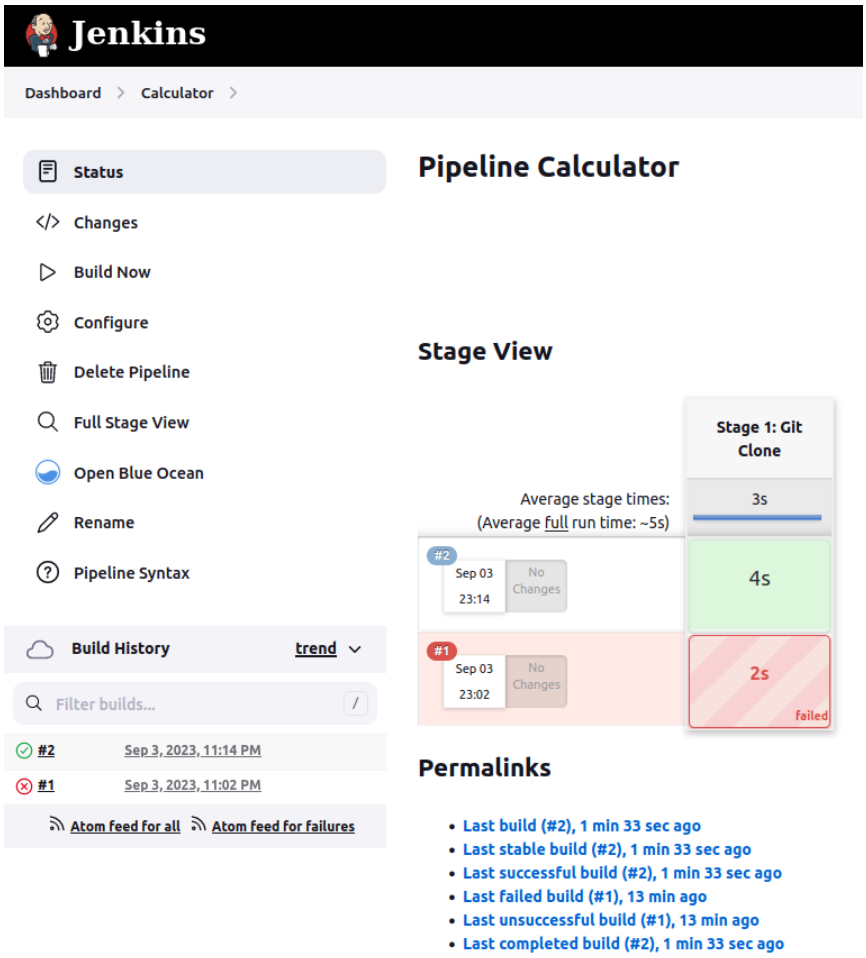

Fig 3.58 Successful execution of the pipeline in the second try

Stage 2: Building the project using Maven

Our first stage is now able to clone the project repository and make a local copy of it. The next stage should now be able to run the local repository and check if the project is compiling or not. Ideally it will run test cases and see if the test cases aresassing or failing but for now, we just want the project to be able to run the main method. To achieve this, we are going to use Maven in the second stage. The script will be as follows:

```
stage('Step 2: Maven Build'){
    steps{
        sh 'mvn clean install'
    }
}
```

Add this code within the stages object of the pipeline script we already have.

3.59 Pipeline script after adding stage 2

Now when we try and build the pipeline, we will see the following output:

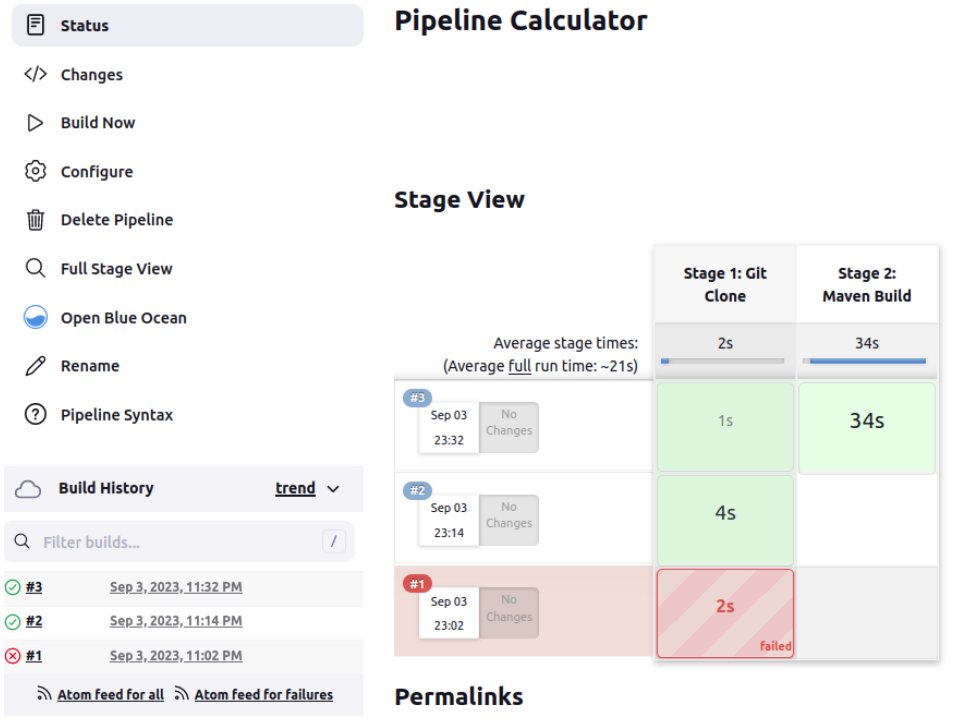

Fig 3.60 Successful execution of stage 2

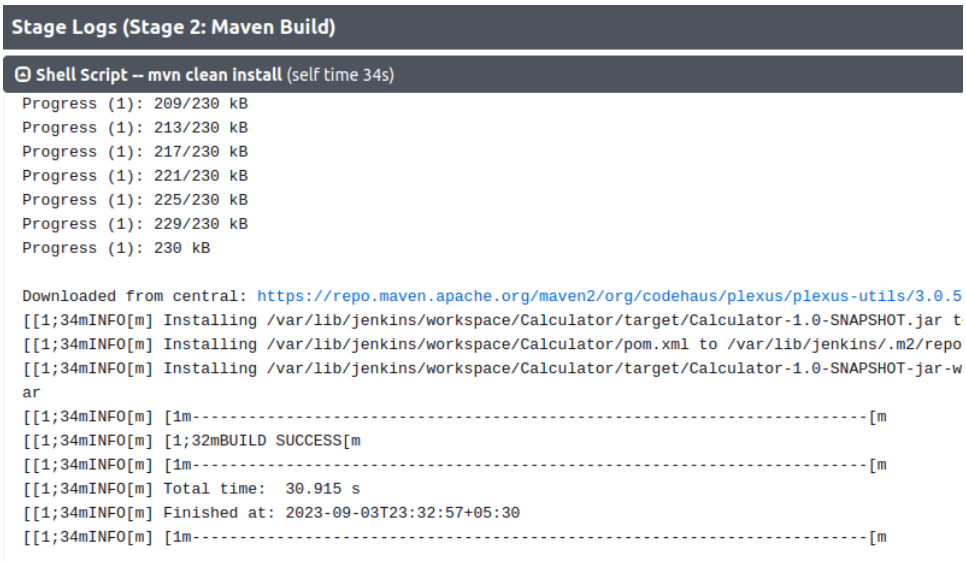

Fig 3.61 Console output of after pipeline execution

Stage 3: Creating Docker container of the project

Now that we can clone and build the project, we are going to containerise the project into a Docker container. This will wrap all dependencies needed to run the project into a single container that in later stages can be used to deploy remotely.

To start, we need to add a **Dockerfile** to our project. This will contain the steps and script that will be run to convert our project into a Docker container. We create this file in the root directory of the project as follows:

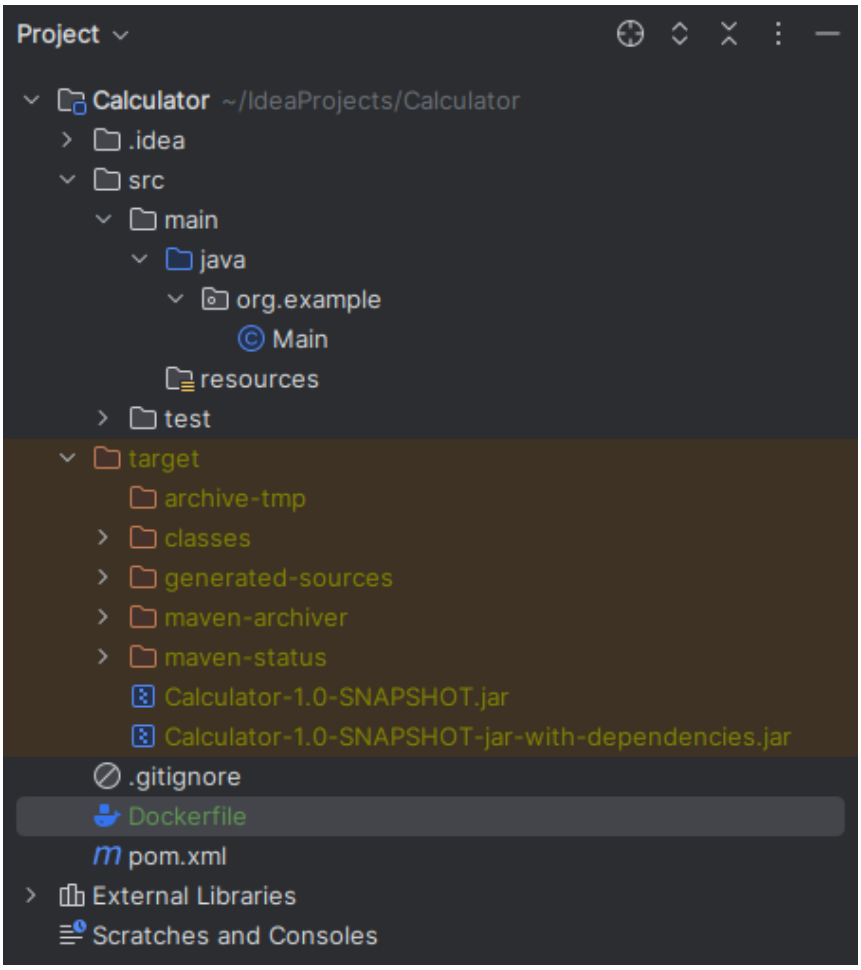

Fig 3.62 Adding Dockerfile to the project

When you create the file, the IDE may ask you if you want to add the file to Git. If you click add, it will get added to your repository. In the above figure, you can see that the Dockerfile is green, signifying that it is added to Git. However, in case the option doesn't pop up for your or you clicked cancel, then you can manually add the file to git using the **git add . c**ommand in the root directory of the project.

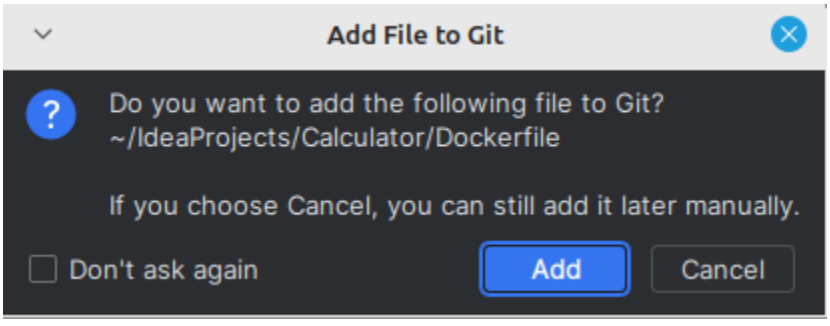

Fig 3.63 Adding Dockerfile to git by the IDE

Once you add the Dockerfile, we will edit it and add the following content:

```
FROM openjdk:11
```

```
COPY ./target/Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar
./

WORKDIR ./

CMD [“java”,”-cp”,” Calculator-1.0-SNAPSHOT-jar-with-
dependencies.jar”,”org.example.Main”]
```

The purpose of the above code is to generate a Docker container with OpenJDK version 11 as the base image which acts like a JVM. This will ensure that the container will have java allowing us to execute java programs. Other dependencies will be read from pom.xml file of the project. Then we are using the copy command to copy the modified JAR file from the target directory to the root directory of the container. Then, using the workdir command, we are making the root directory of the container as the working directory. Finally, using the CMD command we are executing the jar file. The CMD command takes a string array with each comma separated value representing a regular terminal command that is space separated. The only difference is that we are additionally specifying the Main class location as a command line parameter. If this class is not specified properly then it can lead to *ClassNotFound* Exception.

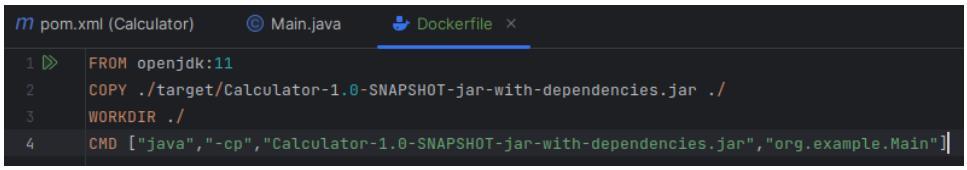

Fig 3.64 Dockerfile contents

Before we build the pipeline stage for creating the Docker image, we are going to run git commit and commit the Dockerfile

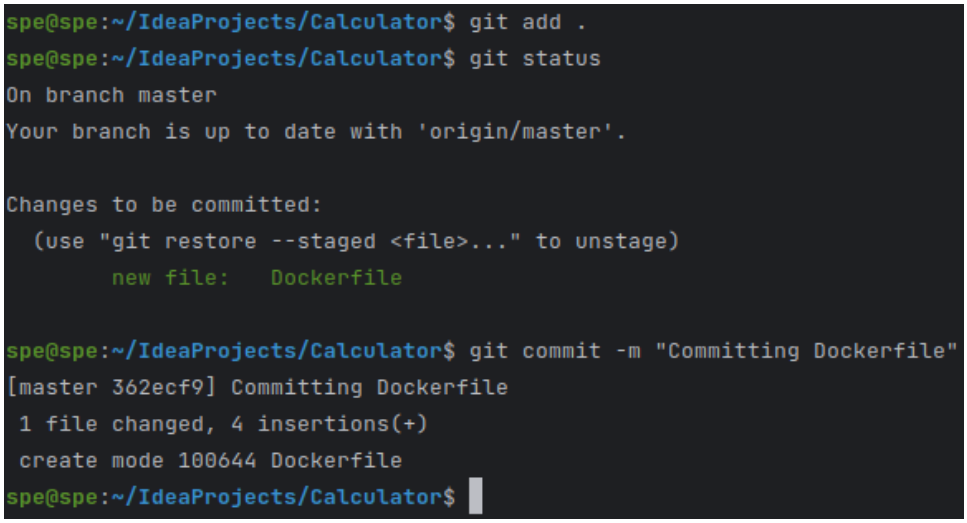

Fig 3.65 Committing Dockerfile

Finally, we will perform **git push**, pass the username and personal access token, and the commits will be pushed to the local repository. Now, when we run the pipeline and clone the remote repository, it will have the Dockerfile.

Now, we will add the following stage to the pipeline script:

```
stage('Stage 3: Build Docker Image'){
    steps{
        script{
            docker_image = docker.build
"jacobkick/calculator:latest"
        }
    }
}
```

The docker build command means *<Docker Hub username>/<name of the container>:<tag>.* The tag latest is used to identify the most recently built image.

The pipeline looks as follows:

Fig 3.66 Pipeline with all 3 stages up til now

If we run the pipeline now, we should see the following:

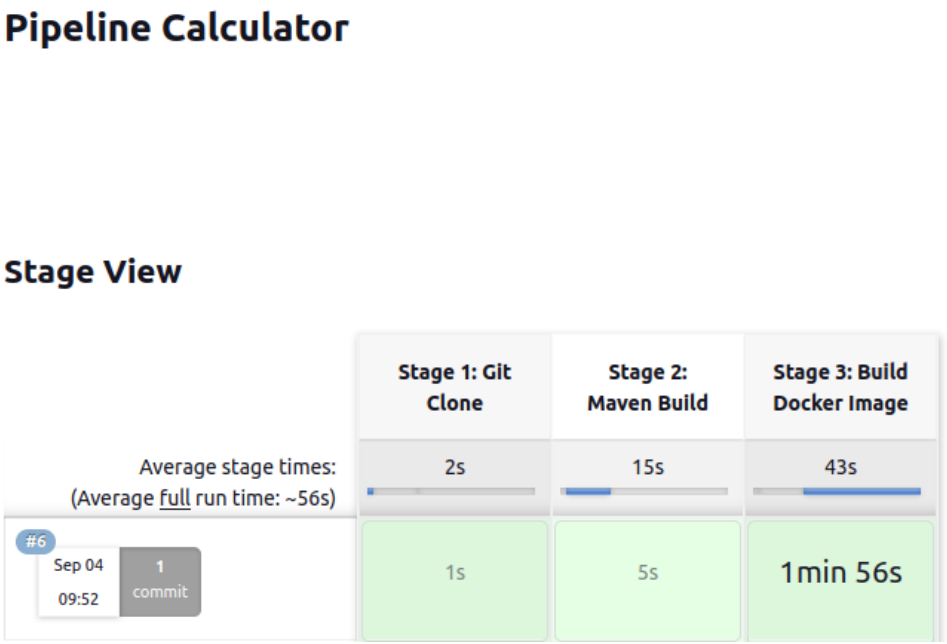

Fig 3.67 Successful pipeline with 3 stages

We can verify that the docker file was created by opening the terminal and running the following command:

```
sudo docker images
```

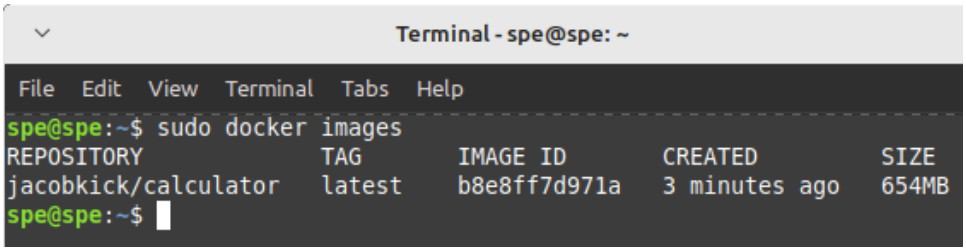

Fig 3.68 List of Docker images

The reason we are using sudo is that we added Docker to Jenkin's user group, which is the sudo user. If we don't use sudo, we will get permission denied error.

Stage 4: Pushing Docker image to Docker Hub

Once we have the docker image, we are going to push it Docker Hub. We start by storing our Docker Hub credentials in Jenkins. We navigate to Dashboard > Manage Jenkins > Credentials. Within that, you want to select System under **Stores scoped to Jenkins.**

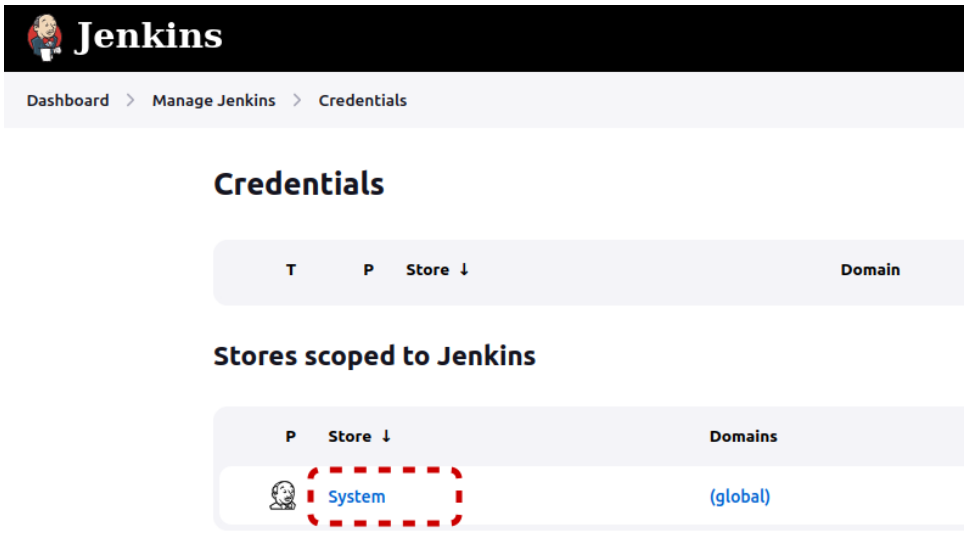

Fig 3.69 System option in Credentials

In System, click on Global Credentials:

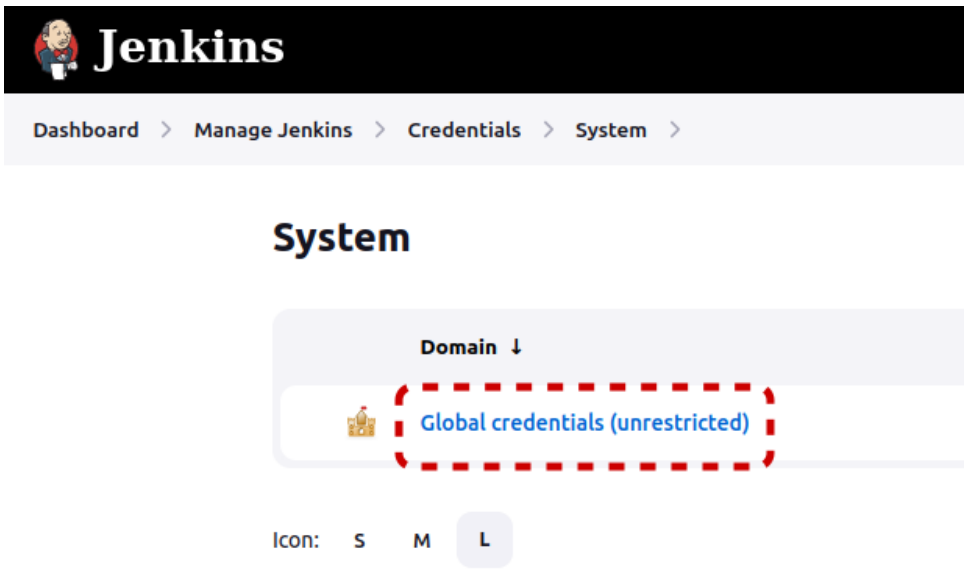

Fig 3.70 Global Credentials option

In Global Credentials, click on add credentials:

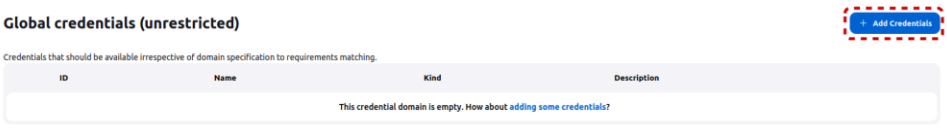

Fig 3.71 Add Credentials Option

In the New Credentials form, we are going to fill the following:

KIND: Username with password

SCOPE: Global

USERNAME: <Your Docker Hub Username>

PASSWORD: <Your Docker Hub Password>

ID: DockerHubCred

DESCRIPTION: Docker Hub Credentials

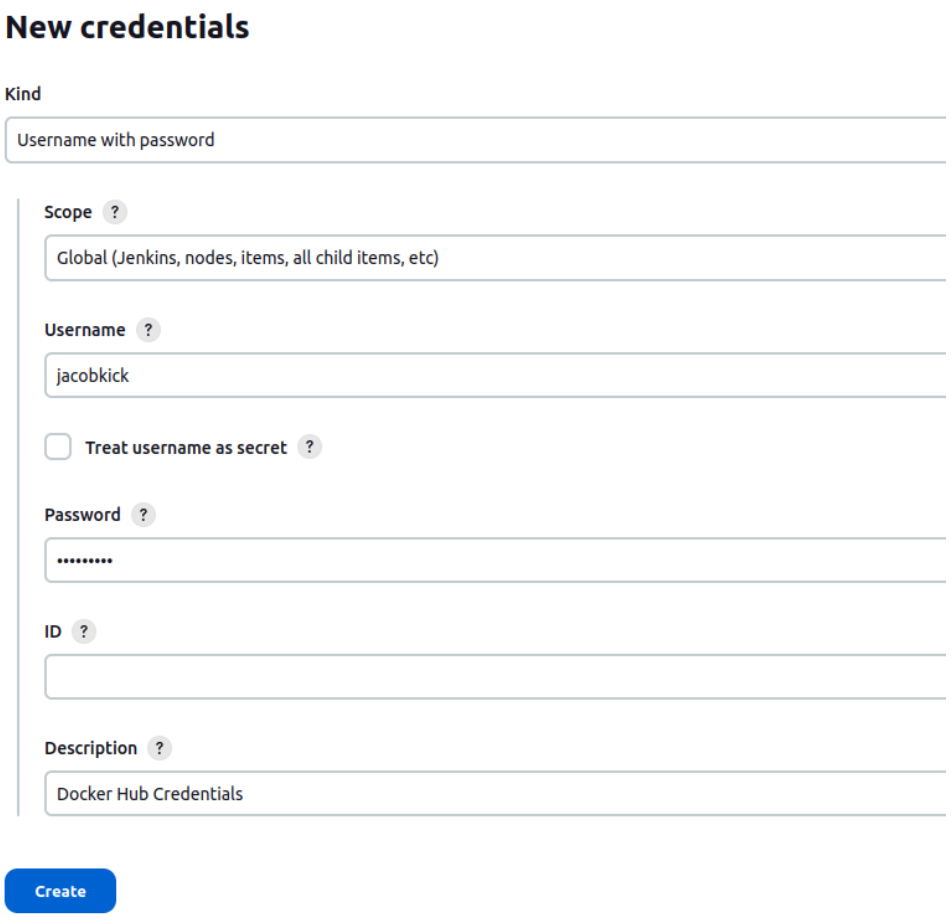

Fig 3.72 Creating Docker Hub credentials

Fig 3.73 Successfully created the credentials

With the credentials successfully created, we can work on the 4th stage of the pipeline. We will add the following to the pipeline script:

```
stage('Stage 4: Push docker image to hub') {
        steps{
            script{
                docker.withRegistry('', 'DockerHubCred'){
                    docker_image.push()
                }
            }
        }
    }
```

This script will push the docker image that we created into Docker Hub using the credentials that we created. We can run the pipeline and see the result.

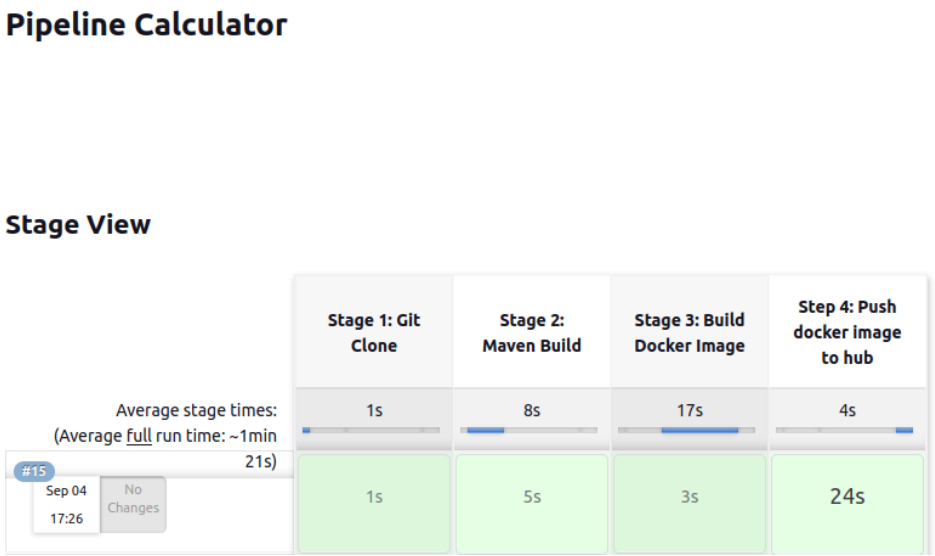

Fig 3.74 Running pipeline with 4 stages

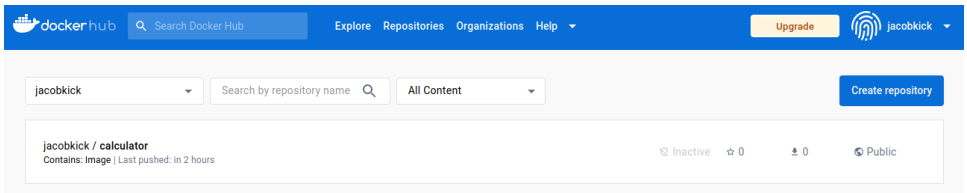

Fig 3.75 Image pushed to Docker Hub

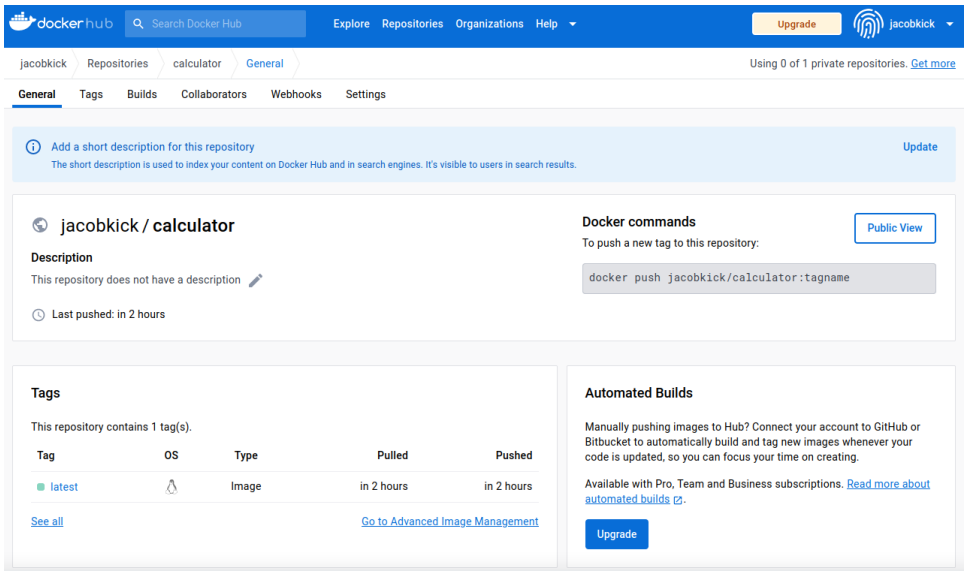

Fig 3.76 Details about the Docker image that was pushed

Stage 5: Cleaning up Docker images

Every time we run the pipeline a new image gets created. Only one image will have tag **latest** and hence the latest one alone gets pushed to Docker Hub. We want to remove these images to ensure that down the pipeline when we pull images there is no errors due to file name already existing.

```
spe@spe:~$ sudo docker images
[sudo] password for spe:
REPOSITORY              TAG       IMAGE ID       CREATED          SIZE
jacobkick/calculator    latest    f11917185ad0   11 minutes ago   654MB
<none>                  <none>    58e756bb1bd1   13 minutes ago   654MB
<none>                  <none>    c3a0b6211aaa   16 minutes ago   654MB
<none>                  <none>    0884458b42c8   22 minutes ago   654MB
<none>                  <none>    3b54264c673c   27 minutes ago   654MB
<none>                  <none>    99e114c332d1   42 minutes ago   654MB
<none>                  <none>    abe9bb6980fa   45 minutes ago   654MB
<none>                  <none>    b8e8ff7d971a   8 hours ago      654MB
spe@spe:~$
```

Fig 3.77 All Docker images

To remove these images except the most recently generated one, we are going to add the following commands into the pipeline script:

```
stage('Stage 5: Clean docker images'){
    steps{
        script{
            sh ‘docker container prune -f’
            sh ‘docker image prune -f
        }
```

```
        }
    }
```

This will remove any dangling images and containers that are present.

After running this stage, if you were to run *sudo docker images* you will see that there is only one image:

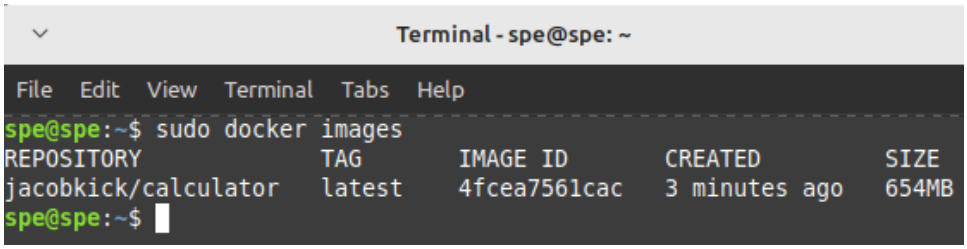

Fig 3.78 No more copies of Docker images

Stage6: Pulling Docker Images using Ansible

Now that we have pushed our image to docker hub, we are going to pull them and create a container using Ansible and execute it.

In this demo, we are going to pull the images into our machine itself. Hence, we are not going to add any hosts into ansible. Else we need to modify **/etc/ansible/host** file and add the appropriate IP addresses.

The first step towards deployment is to store the details of how we want to deploy the docker image. We are going to store this within the project directory itself under a sub directory called **Deployment**. The folder will contain two files: **inventory** and **deploy.yml**.

The inventory file will contain the list of clients that will deploy the project. The deploy yaml file will contain the specifications of the image that is being pulled and deployed.

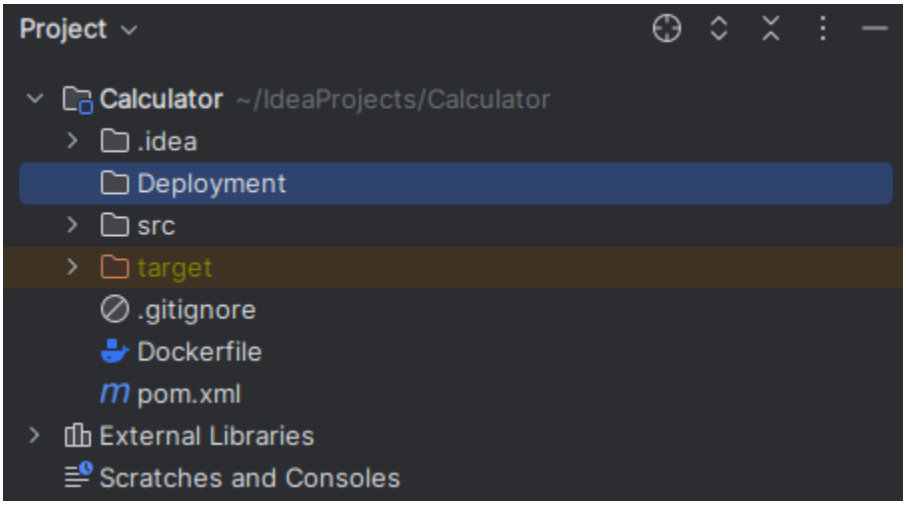

Fig 3.79 Structure of project root directory

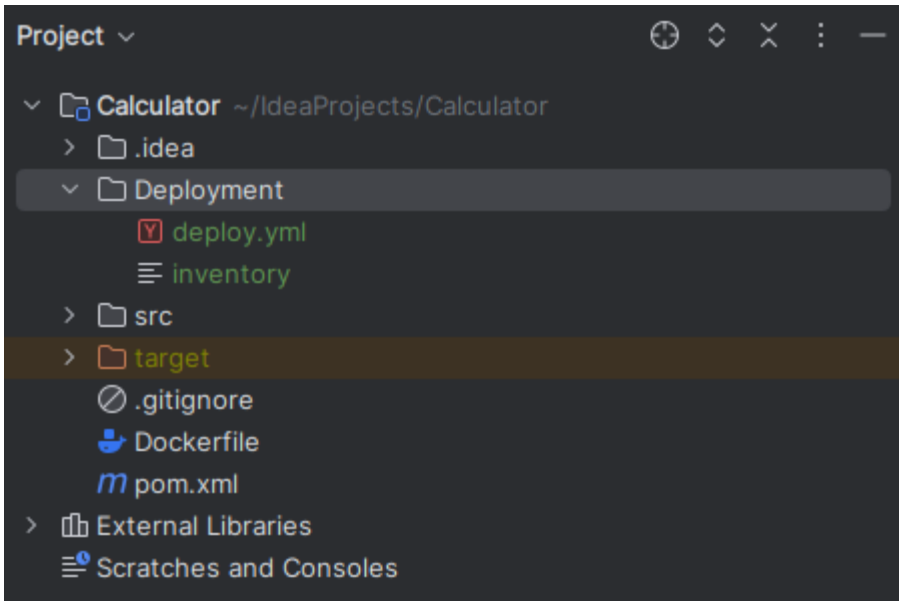

Fig 3.80 Adding inventory and deploy yaml file under Deployment sub directory

Since we are deploying the project in our computer and user, we are going to put the following in the inventory file:

```
localhost ansible_user = <username>
```

Here, spe is the name of the user account that we are currently in.

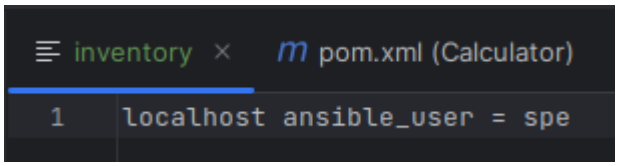

Fig 3.81 Inventory file content

In the deploy yaml file we will add the following:

```
---
- name: Pull Docker Image of Calculator
  hosts: all
  vars:
    ansible_python_interpreter: /usr/bin/python3
  tasks:
    - name: Pull image
      docker_image:
        name: jacobkick/calculator:latest
        source: pull
    - name: Start docker service
      service:
        name: docker
```

```
      state: started
  - name: Running container
    shell: docker run -it -d --name Calculator
jacobkick/calculator
```

Note that yaml files are sensitive to the indentation of script blocks, like python. Moreover, your IDE might also show everything having a warning which is normal. Just be sure to follow the indentation as show in the below figure:

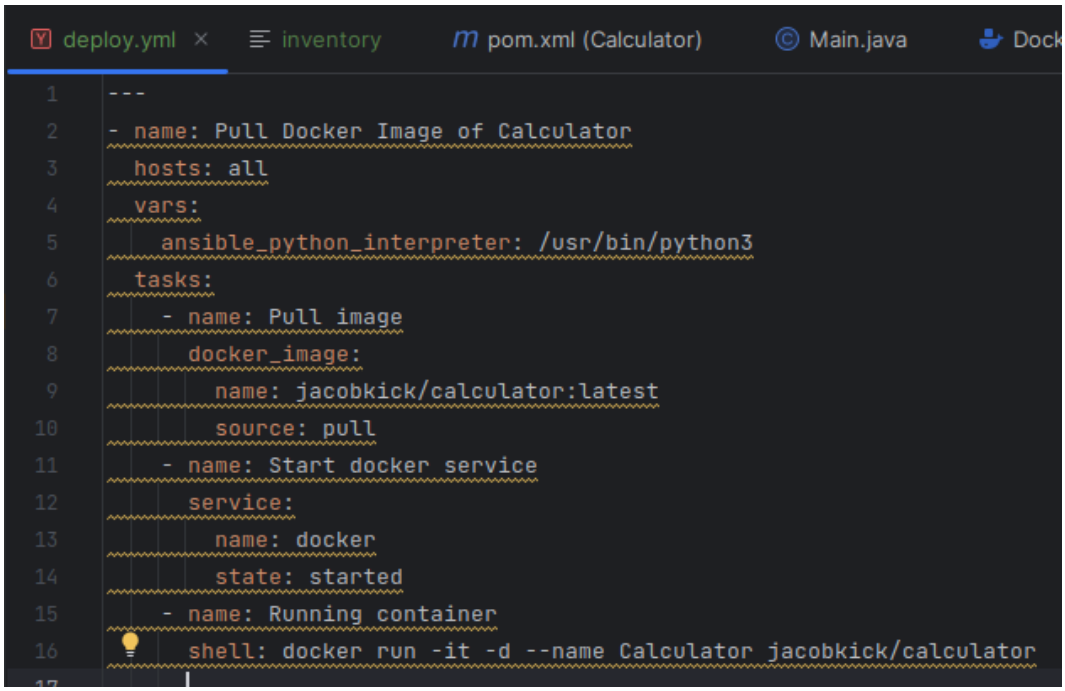

Fig 3.82 Deploy yaml file contents

Ensure that the <username>/<image name>:<tag> matches the one that you pushed to Docker Hub. Also ensure that the path to Python 3.x is given properly as Ansible is based on Python and needs the Python interpreter's path to be given to it. Python is by default installed with most flavours of linux. You can get the path using the command **whereis python3**:

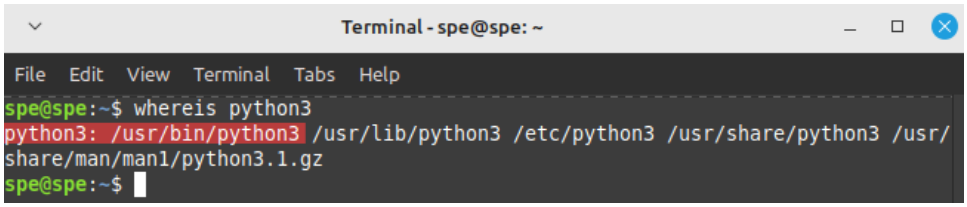

Fig 3.83 Python 3 path

With this, we are ready to add the files to git, commit and push them to the local repo. After that, we will add the 6th and final stage to the pipeline using the following script:

```
stage('Step 6: Ansible Deployment'){
```

```
        steps{
              ansiblePlaybook becomeUser: null,
              colorized: true,
              credentialsId: 'localhost',
              disableHostKeyChecking: true,
              installation: 'Ansible',
              inventory: 'Deployment/inventory',
              playbook: 'Deployment/deploy.yml',
              sudoUser: null
        }
  }
```

Here, we make sure to mention **credentialsId: 'localhost'** as we are deploying to our own system. Otherwise, you will get a permission denied error. Then, we need to specify our localhost username and password in Jenkin's Global credentials, the same way we added Docker Hub credentials. The username and password will be the localhost user's Linux login username and password. The ID of the credentials will be localhost, to match what we are setting in the pipeline script:

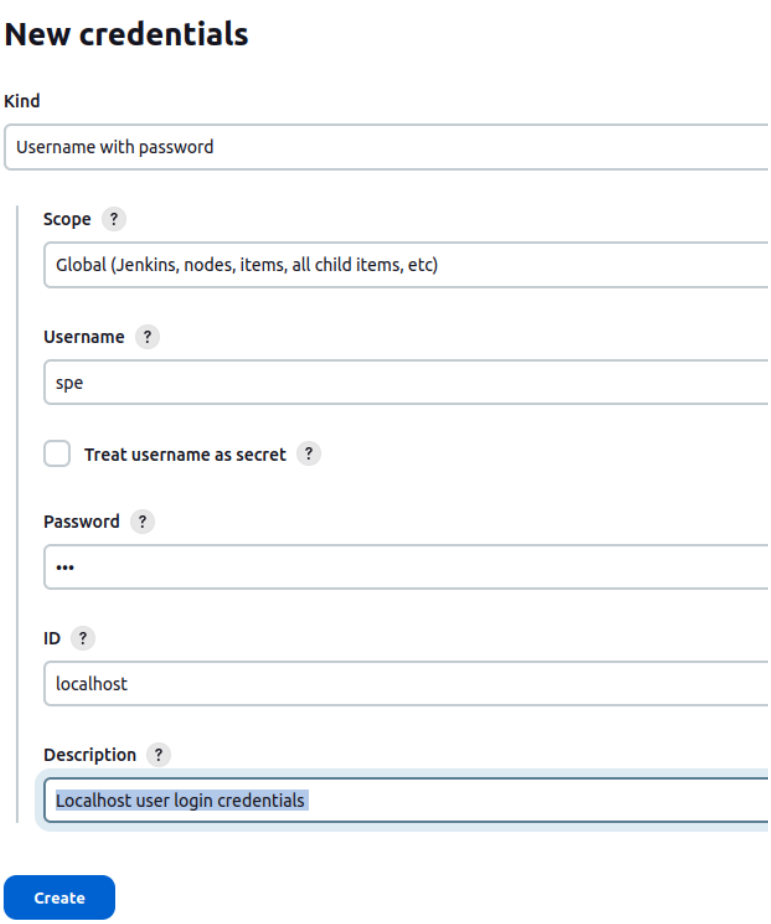

Fig 3.84 Localhost credentials details

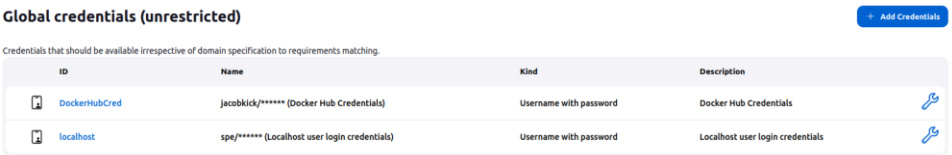

Fig 3.85 Localhost credentials saved

With this, if we now run the pipeline:

Pipeline Calculator

Stage View

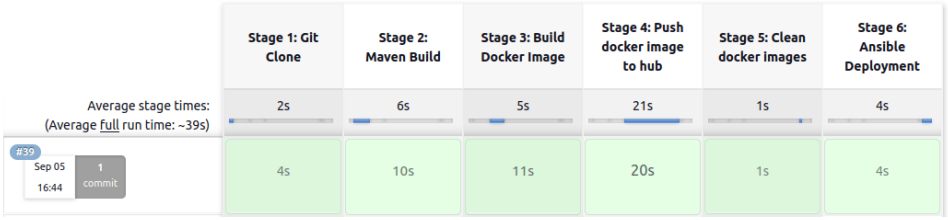

Fig 3.86 Pipeline successfully executed

To see the list of containers, we can use the following command:

```
sudo docker container ls --all
```

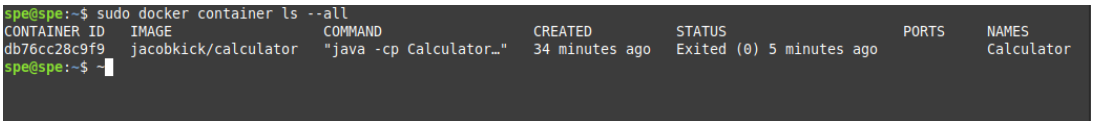

Fig 3.87 List of containers

We can see that the container status is Exited. To run the container, we use the following command:

```
sudo docker start -a -i Calculator
```

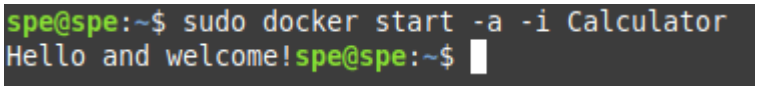

Fig 3.88 Output of running the container

Congratulations. We can finally see the output of the main program after such convoluted steps. But the project was not Hello World program, it was calculator program. So, we can now finally write the calculator program.

Implementation of Calculator program

We start by including Log4j2 dependency in our pom.xml file. The required xml for that is:

```
<!--
https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j
-core -->

<dependency>

    <groupId>org.apache.logging.log4j</groupId>

    <artifactId>log4j-core</artifactId>

    <version>2.20.0</version>
```

```
    </dependency>
```

```
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core -->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.20.0</version>
        </dependency>
    </dependencies>
</project>
```

Fig 3.89 Dependency for log4-core with the link commented out

The <dependencies>...</dependencies> tag is a direct child of the <project>...</project> tag. Within it, each of the library or dependency we are going to use will have its own <dependency>...</dependency> tag.

We are using Log4j for the ability to generate log files that we will use in later sections to analyse not only if something goes wrong but also to measure performance parameters as well. However, Log4j decided to split its API and implementation into separate abstractions and implementations. Log4j core is the implementation. We therefore need the abstraction as well, which is log4j-api.

```
    <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.20.0</version>
    </dependency>
```

```
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.apache.
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.20.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-api</artifactId>
            <version>2.20.0</version>
        </dependency>
    </dependencies>
</project>
```

Fig 3.90 Both Logj4 core and api dependencies satisfied in pom.xml

With the dependency satisfied, we can now instantiate the logger in our calculator project using the following code.

```
private static final Logger logger =
LogManager.getLogger(Main.class);
```

```
deploy.yml   inventory   pom.xml (Calculator)   Main.java   Dockerfile
1  package org.example;
2
3  import org.apache.logging.log4j.LogManager;
4  import org.apache.logging.log4j.Logger;
5
6  import java.util.Scanner;
   Jakekick99 *
7  public class Main {
       no usages
8      private static final Logger logger = LogManager.getLogger(Main.class);
       Jakekick99 *
9      public static void main(String[] args) {
10         System.out.println("-----Welcome to Calculator-----\n" +
```

Fig 3.91 Creating the logger instance within the Main class

Now that we have the logger instantiated, we are going to add some code for all the calculator functionality we are going to implement. Each of the 4 operations is going to have their own methods.

This is because we want to include logging into each of the operation's functions so that we can demonstrate how log messages can be used to track the flow of execution and later measure performance how long each calculation took by analysing the time the program spent within each function.

Within this the most important lines of code are the ones that generate the logs. There are a few predefined levels of logs: Trace, Debug, Info, Warn, Error and Fatal. They can all be accessed using the logger object. Example is as follows:

```
logger.info("Start of Execution");

logger.warn("Invalid input");
```

To enable the generation of logs, we need to an xml file that describes how we want the log file to look like. For the we add an xml file called **log4j2.xml** which will contain the following XML data:

```
<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="INFO">

    <Appenders>

        <Console name="ConsoleAppender" target="SYSTEM_OUT">

            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-
5level %logger{36} - %msg%n" />

        </Console>

        <File name="FileAppender"
fileName="calculator_devops.log" immediateFlush="false"
append="true">

            <PatternLayout pattern="%d{yyy-MM-dd HH:mm:ss.SSS}
[%t] %-5level %logger{36} - %msg%n"/>

        </File>

    </Appenders>

    <Loggers>

        <Root level="debug">

            <AppenderRef ref="FileAppender"/>

        </Root>

    </Loggers>

</Configuration>
```

The log4j2.xml file should be placed under the main subdirectory, in a directory called **Resources** as follows:

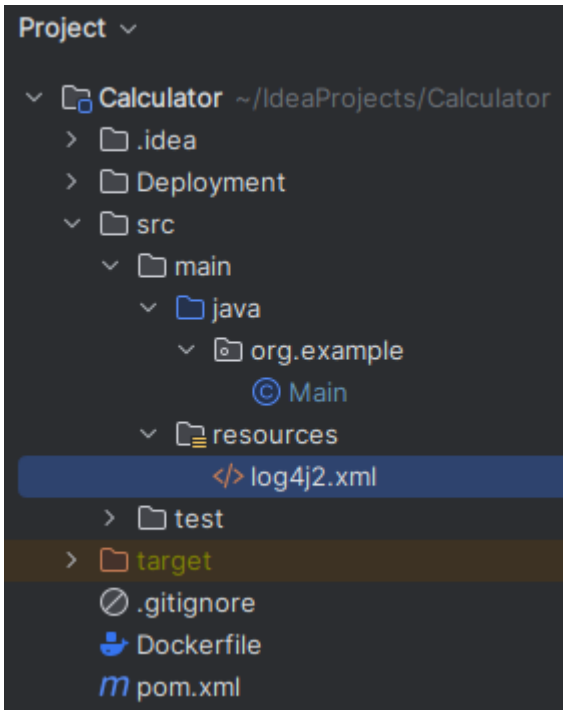

Fig 3.92 Location of log4j2.xml file

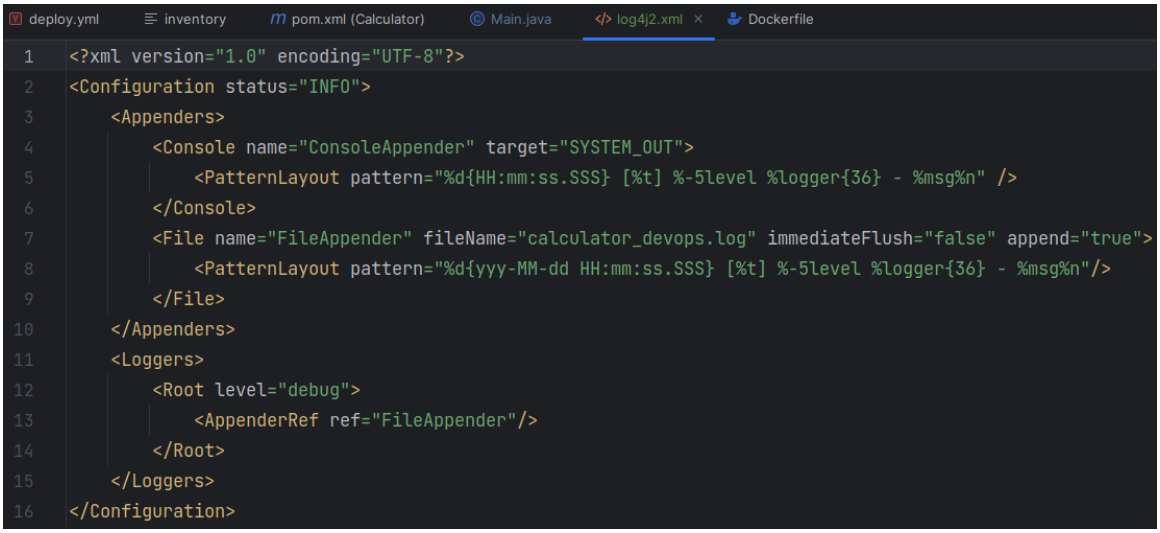

Fig 3.93 Contents of log4j2.xml file

Now if you run the program and give the necessary input, you can see the file appear in the project explorer. (*If you add it to git then it will be green otherwise it will be red)*

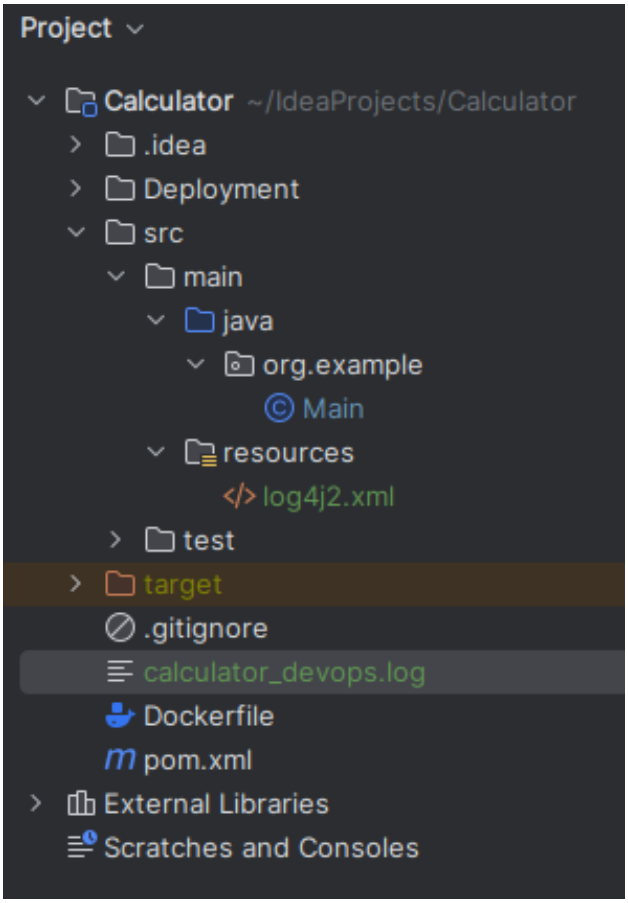

Fig 3.94 Project Explorer showing the log file

```
/usr/lib/jvm/default-java/bin/java -javaagent:/home/
-----Welcome to Calculator-----
Choose your operation
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit
1
Enter 1st operand:
1
Enter 2nd operand:
2
--------Result is--------
1 + 2 = 3

Process finished with exit code 0
```

Fig 3.95 The input given during execution

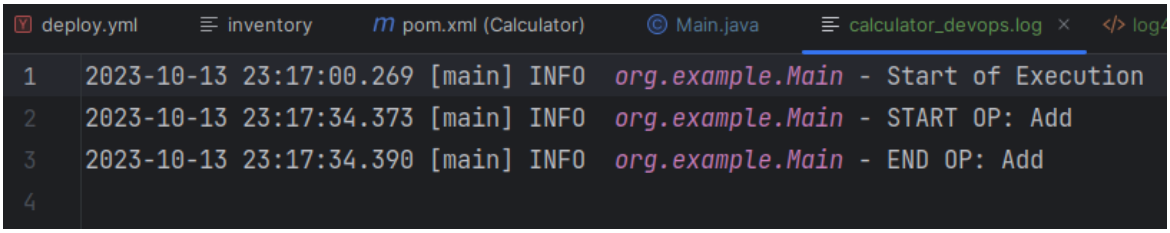

Fig 3.96 Contents of the log file

Once we have logging, we are also going to add some test cases to the project. We add test cases that are required to pass when run on maven. This ensures that any changes made to the project does not result in unintended behaviour change. We are going to write and execute tests using **JUnit**. We add the dependency for JUnit by adding the following xml in pom.xml:

```
<dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13.2</version>
        <scope>test</scope>
</dependency>
```

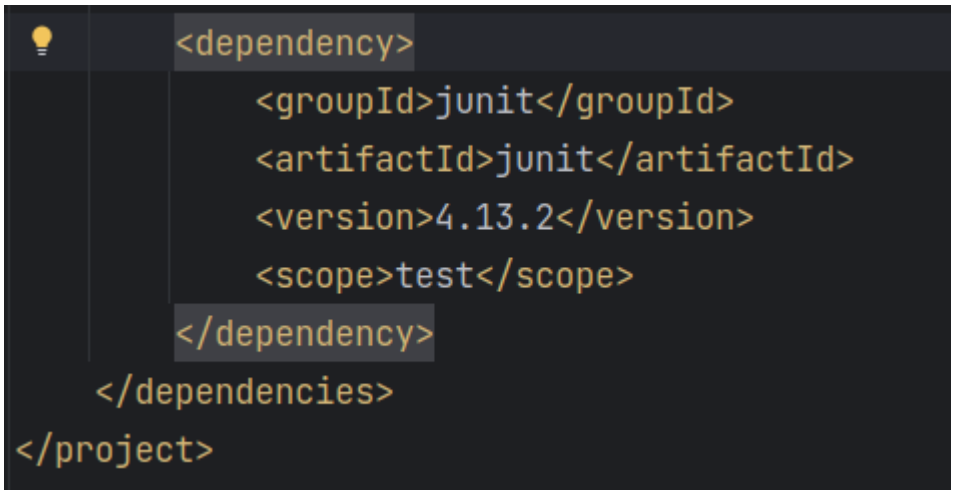

Fig 3.97 JUnit XML in pom.xml

We then create a new java class file called **CalculatorTest.java** under the test subdirectory:

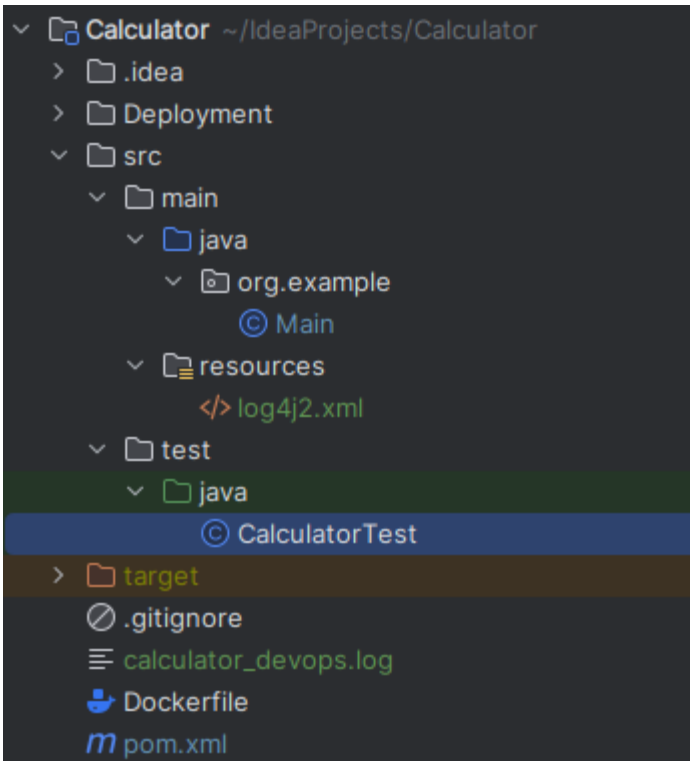

Fig 3.98 Test file location & name

We then need to create a global instance of the program that we are going to test:

Fig 3.99 @Before of the test file

Once we have created a global instance of the Main class then we can add test cases like the following:

```
@Test
```

```
    public void test_add_positive(){
        int a = 1;
        int b = 2;
        int expectedResult = 3;
        Assert.assertEquals(expectedResult, calculator.add(a,
b));
    }
    @Test
    public void test_add_negative(){
        int a = 1;
        int b = 2;
        int expectedResult = 0;
        Assert.assertNotEquals(expectedResult, calculator.add(a,
b));
    }
```

This is an example of a positive and negative test case for the add() function. In a similar way we will add for the other operations as well.

```java
    @Test
    public void test_add_positive(){
        int a = 1;
        int b = 2;
        int expectedResult = 3;
        Assert.assertEquals(expectedResult, calculator.add(a, b));
    }
    new *
    @Test
    public void test_add_negative(){
        int a = 1;
        int b = 2;
        int expectedResult = 0;
        Assert.assertNotEquals(expectedResult, calculator.add(a, b));
    }
    new *
    @Test
    public void test_sub_positive(){
        int a = 2;
        int b = 2;
        int expectedResult = 0;
        Assert.assertEquals(expectedResult, calculator.sub(a, b));
    }
    new *
    @Test
    public void test_sub_negative(){
        int a = 2;
        int b = 2;
```

Fig 3.100 Example test cases

Now if we were to run *mvn clean install* command from the terminal we will see the following in the console output:

```
[INFO] Surefire report directory: /home/spe/IdeaProjects/Calculator/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running CalculatorTest
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.776 sec

Results :

Tests run: 8, Failures: 0, Errors: 0, Skipped: 0

[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ Calculator ---
[INFO] Building jar: /home/spe/IdeaProjects/Calculator/target/Calculator-1.0-SNAPSHOT.jar
```

Fig 3.101 Test cases being run when executing mvn clean install command

With this, the programming is complete. We will move onto wrapping up the pipeline execution and then focus on deploying the container, extracting the log file from the container and using ELK Stack for analysis of the log file.

Executing the pipeline automatically using Git SCM Polling & ngrok

We start by running ngrok on port 8080 with the following command:

```
ngrok http 8080
```

Fig 3.102 Command to run ngrok

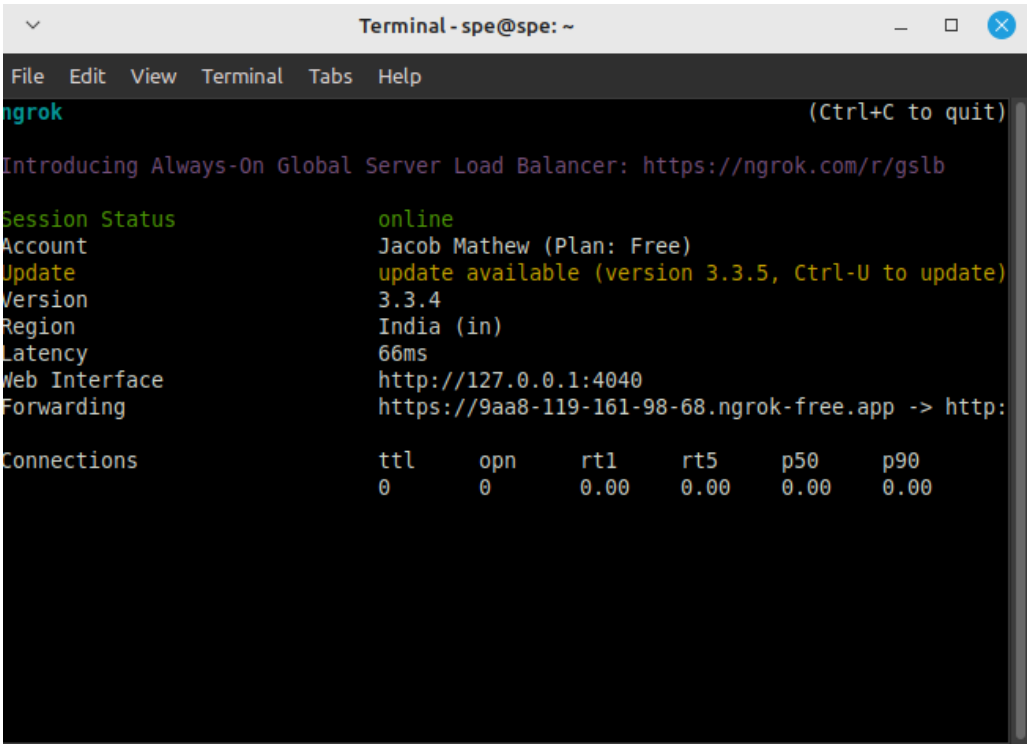

Fig 3.103 Ngrok running and shows the forwarding URL

The forwarding URL is https://9aa8-119-161-98-68.ngrok-free.app . Note that this will change every time we close and launch ngrok (since this is the free version). So, make sure not to close this for now. We then go to our project repository in GitHub > Settings > Webhooks option of our repository,

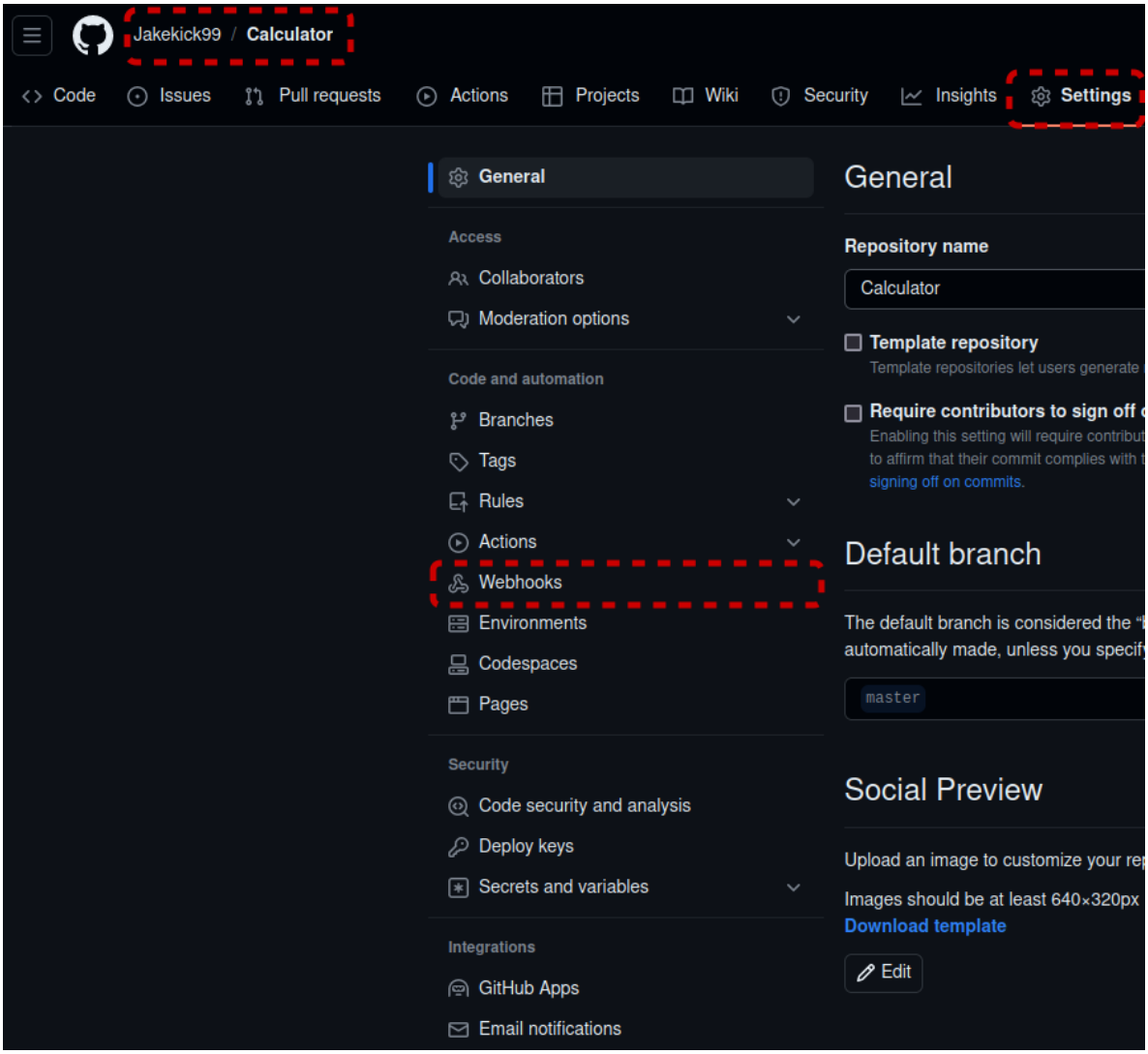

Fig 3.104 Web Hook section of our repository

In the webhooks section, we click on add webhook and then specify the following details:

Payload URL: `<ngrok forwarding url>/github-webhook/`

Considering the above forwarding address, it would be:

https://9aa8-119-161-98-68.ngrok-free.app/github-webhook/

The rest of the options are left to default, and we save the webhook.

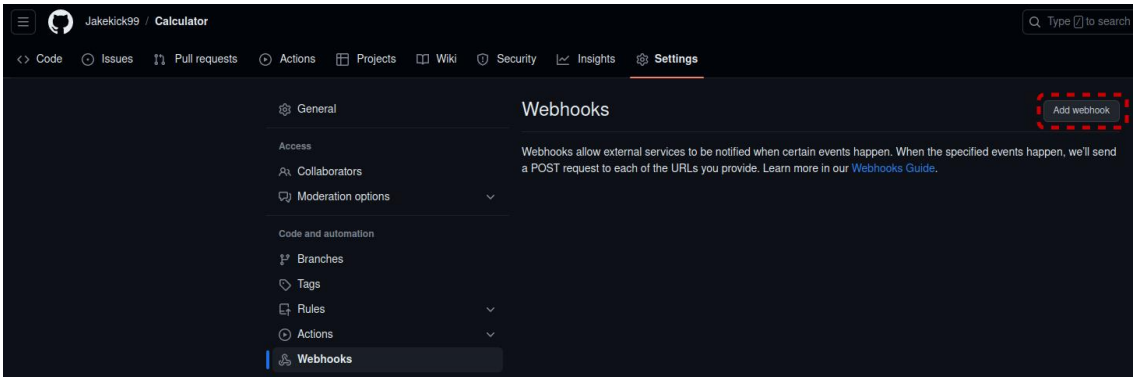

Fig 3.105 Adding webhook

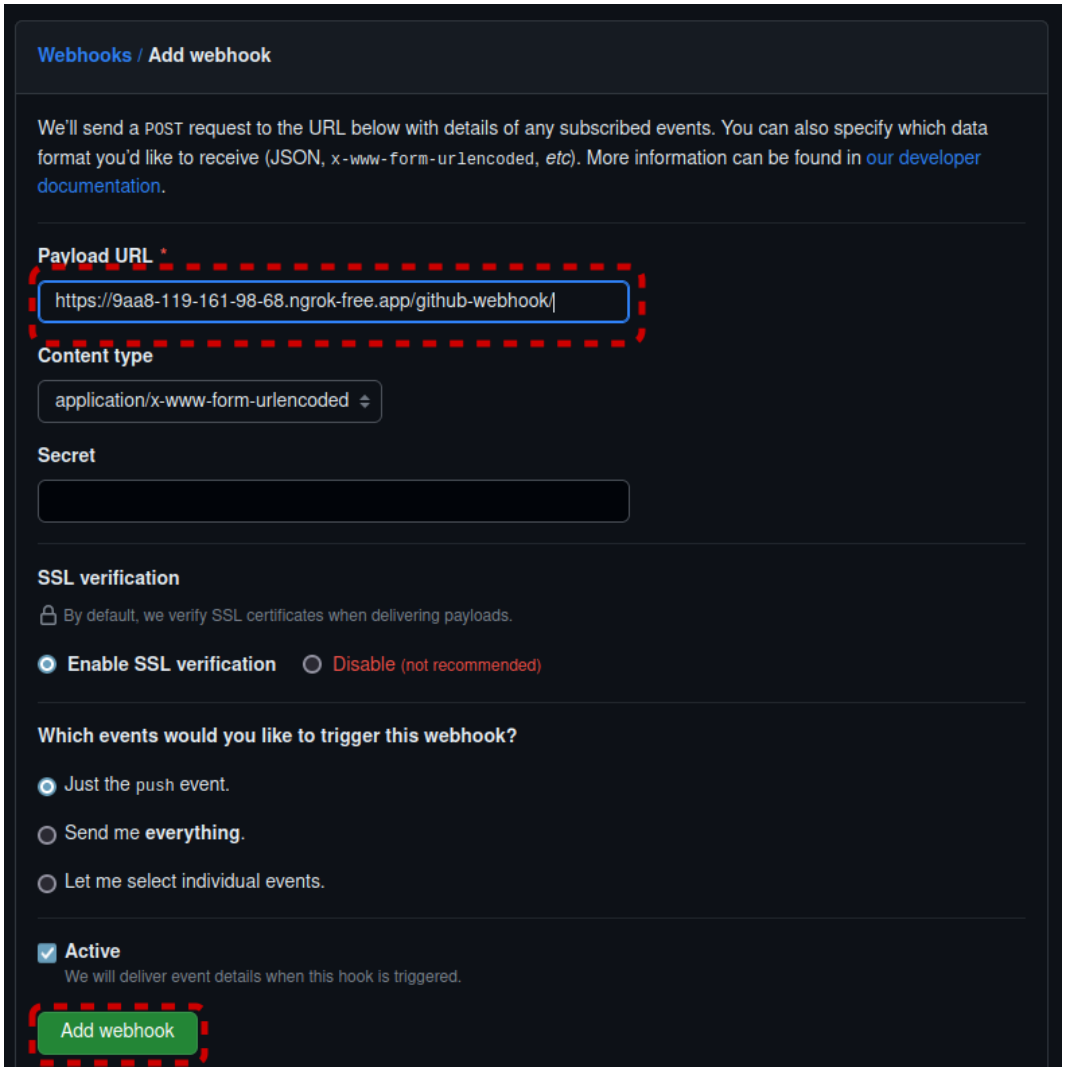

Fig 1.106 Webhook details

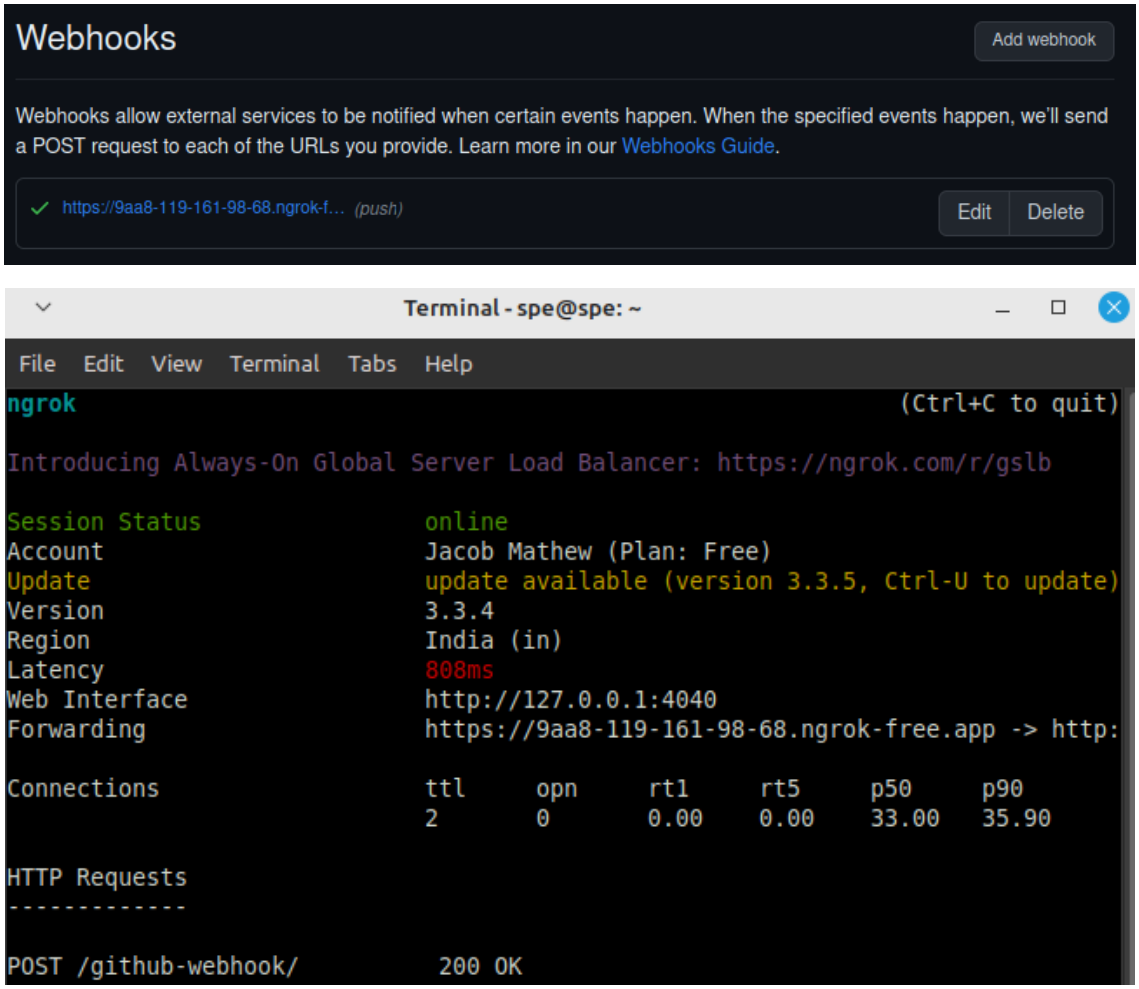

Fig 3.107 Seeing in GitHub & ngrok that it has received a test forward from GitHub

Now that the webhook is reaching the port forwarding address, we are going to make Jenkins recognise this port forwarding address by changing the Jenkin's from localhost to the forwarding address of ngrok. We go to Jenkins Dashboard > Manage Jenkins > System > Jenkins URL:

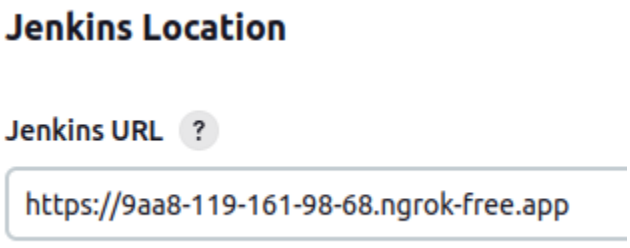

Fig 3.108 Setting Jenkins URL to ngrok forwarding address

(When you close ngrok and want Jenkins to work, you must change the Jenkins URL back to the default which is http://localhost:8080. This will also be visible on the URL bar of the browser showing you where Jenkins is running from, in this case from port 8080 on localhost)

In the same page, we also add the URL for GitHub API along with the personal access token saved as Jenkins Credentials as follow:

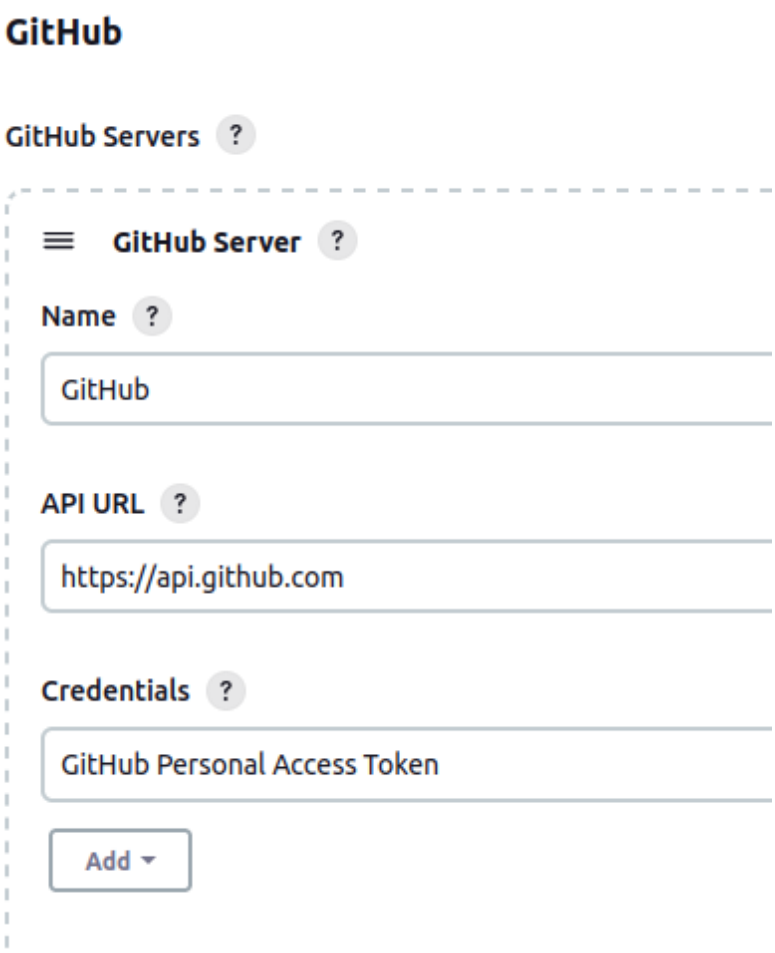

Fig 3.109 GitHub API details

The last step is to allow the pipeline to be triggered by the webhook. For this we go from Jenkins Dashboard > Calculator pipeline > Configure pipeline > Build Triggers and allow GitHub hook trigger for GITScm polling as follows:

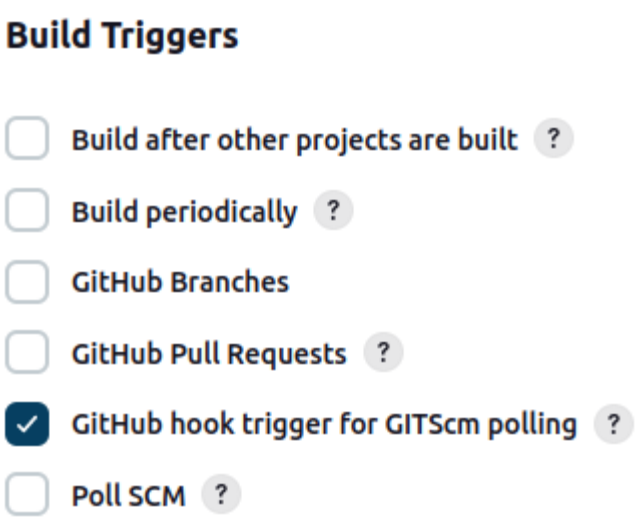

Fig 3.110 Allow Git SCM Polling

Once we save this, we can now demonstrate that if we commit any changes from the IDE, the pipeline will start and execute by itself:

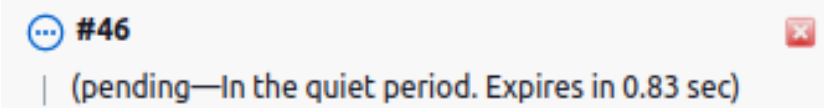

Fig 3.111 Jenkins getting the webhook about the push

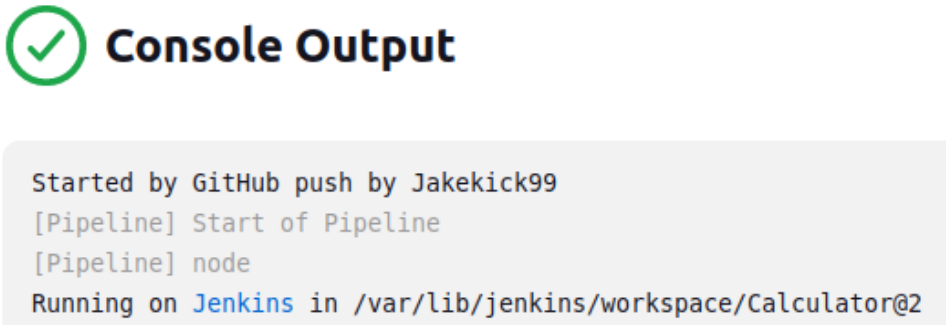

Fig 3.112 Console output showing that it was a GitHub push that started the pipeline execution

With that our entire pipeline is working seamlessly without any manual intervention. We can continue to make changes to the project in the IDE. As soon as we commit those changes, the push event gets broadcasted to Jenkins through the webhook. From there, Jenkins pulls the repo, builds it and run tests using Maven and then creates a docker container and pushes to docker hub. Finally, the image is pulled, and the container is run on the machine using Ansible. The only thing left now is to open the running container, perform some calculations using the calculator, extract the log file of the calculator and analyse it using ELK Stack.

Extracting log file from running container

Now that we have the project running, we are going to access the container and run the program so that we can perform a few calculations. Since our Ansible playbook already has the necessary commands to run the docker image, the container should already be running as the pipeline should automatically execute the Ansible playbook.

To access the container, we can use the following command:

```
sudo docker start -a -i Calculator
```

The name of the container must match what we have given in the Playbook which was:

```
shell: docker run -it -d --name Calculator jacobkick/calculator
```

We can then give our desired input like the following example:

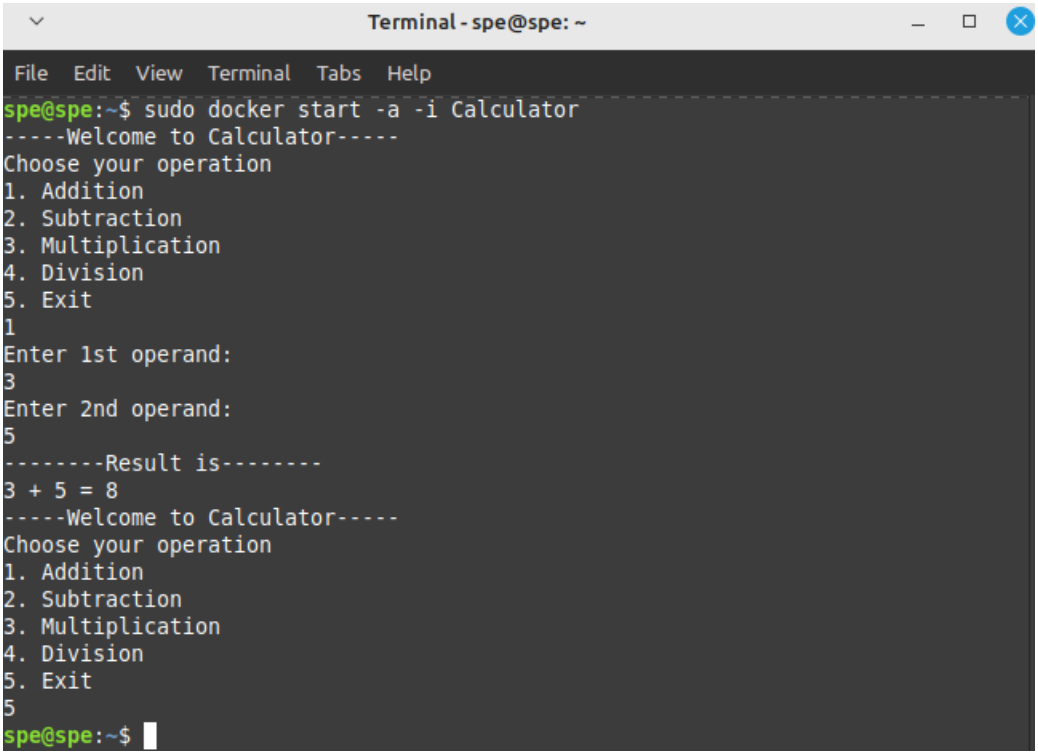

Fig 3.113 Running our calculator project

Now, to access the log file, we must open the container like a file directory. We can do that with the following command:

```
sudo docker start Unscientific
```

```
sudo docker exec -it Unscientific /bin/bash
```

This will change our terminal's pwd to the root directory of the container as shown below:

```
spe@spe:~$ sudo docker start Calculator
sudoCalculator
spe@spe:~$ sudo docker exec -it Calculator /bin/bash
root@b16bfe69ca18:/# pwd
/
root@b16bfe69ca18:/#
```

Fig 3.114 Entering the file directory of the project container

Once we are in the container's file directory, we need to copy the log fie out of it. We first use the **ls** command to see the list of files, which also shows the log file.

```
root@b16bfe69ca18:/# ls
Calculator-1.0-SNAPSHOT-jar-with-dependencies.jar  dev   lib64  proc  srv  var
bin                                               etc   media  root  sys
boot                                              home  mnt    run   tmp
calculator_devops.log                             lib   opt    sbin  usr
root@b16bfe69ca18:/#
```

Fig 3.115 Log file within the container

Once we verify the log file exists, we use the cp command as follows to copy the log file out of the container:

`exit` *(to exit the container's file directory)*

`docker cp Calculator:calculator_devops.log <Destination Directory>/calculator.log`

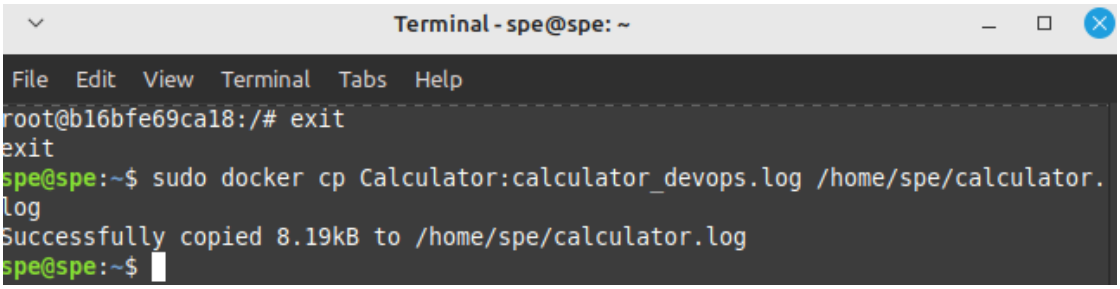

Fig 3.116 Copying log file to our user's home directory

Now if we go to the destination location i.e /home/spe we can see the log file. Open it and we can see the contents as well:

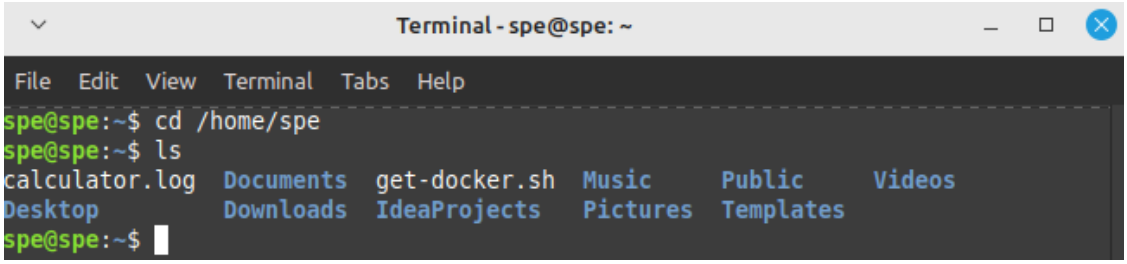

Fig 3.117 Log file destination

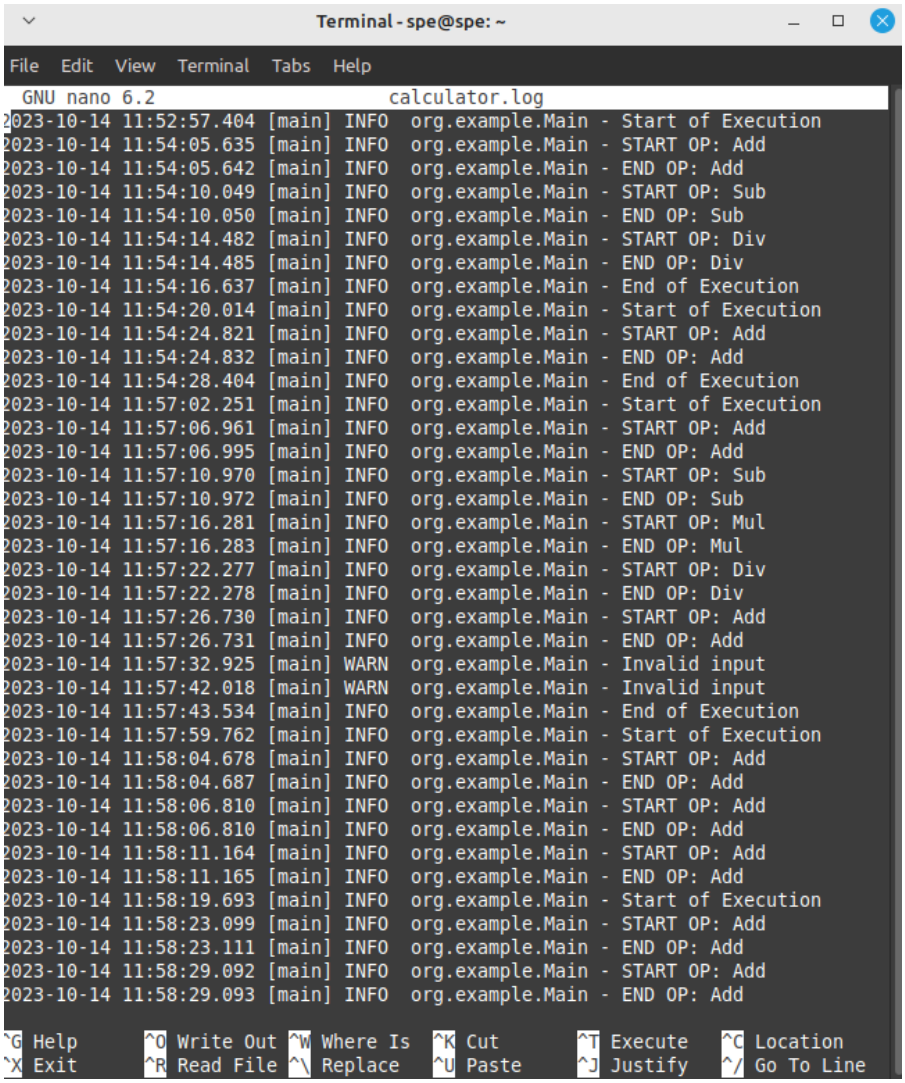

Fig 3.118 Log file contents

With this, we have the log file that we can now use to analyse using ELK Stack. This completes the programming and deployment aspect of the project.

Using ELK Stack

We go to http://localhost:5601/ to enter the Kibana dashboard. In the dashboard, we click on **Upload a file** and choose the file that we want to upload, in this case the log file from the container we copied out:

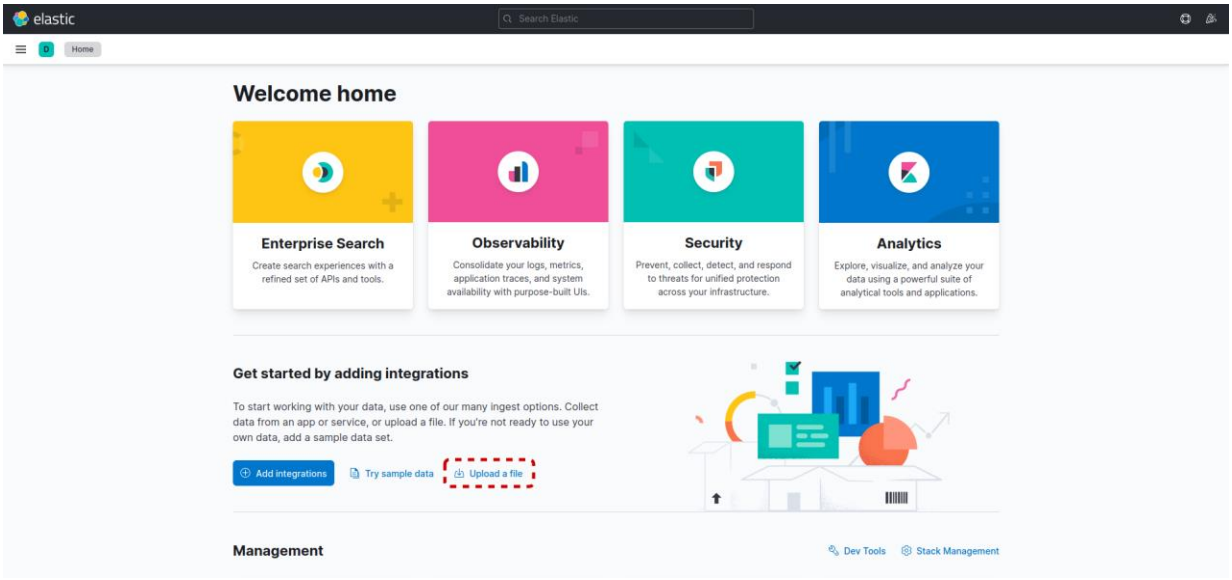

Fig 3.119 Uploading log file to Kibana

Once the file shows up, we can see the file show up along with some details like the number of lines of code. More importantly, we can see that there are few settings that are already pre applied by Kibana like identifying the GROK pattern and the time pattern. If you followed how along as above then good news, this was the default GROK pattern that we set in the log4j2.xml file in our project. In case you set something else, them you can click on override settings to change the GROK pattern and other parameters.

Kibana has also identified for us the columns which in this case are loglevel, message and time stamp. Again, depending on the log pattern that you set, you can change this as necessary, adding or dropping the fields.

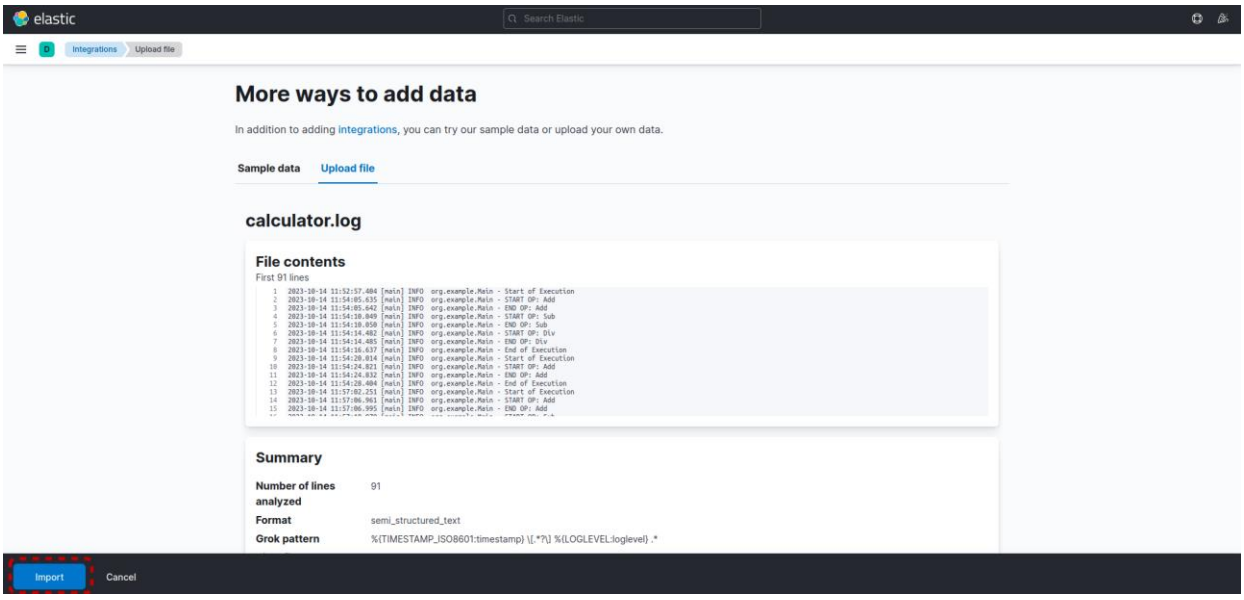

Fig 3.120 Importing the log file

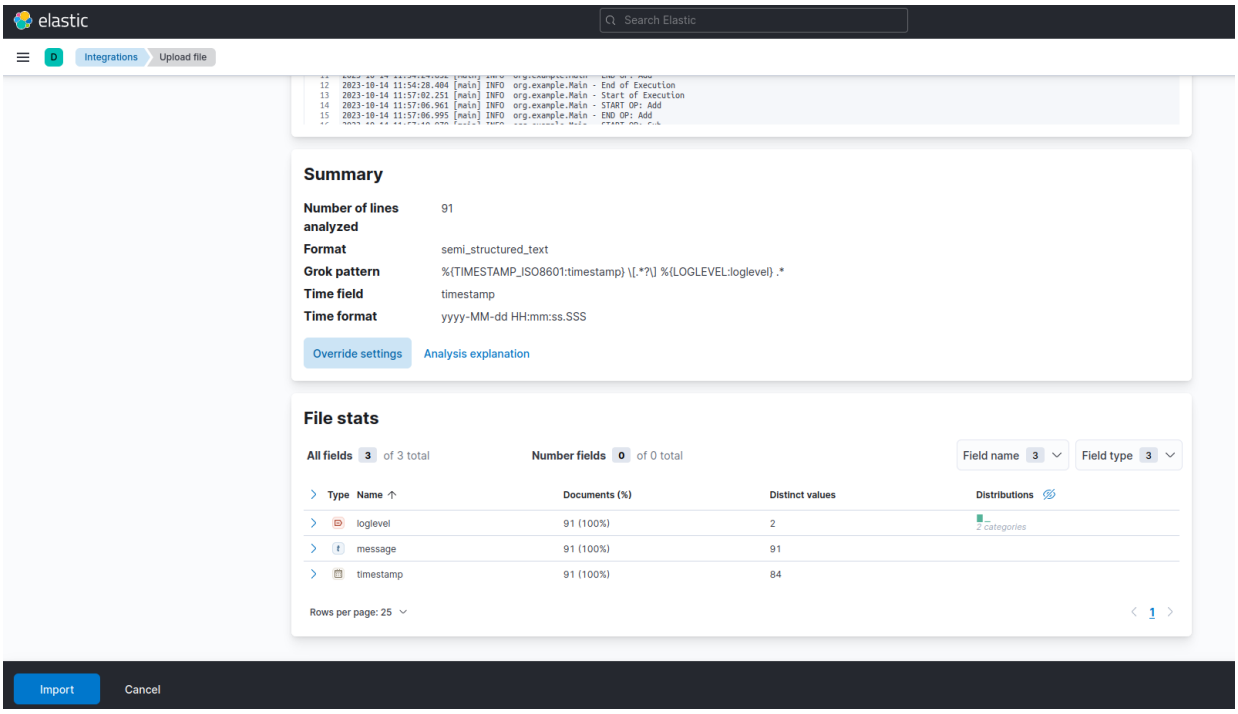

Fig 3.121 Options to change parameters about the log file

Once you import the log file, Kibana will then ask you to create an index name. The option index pattern will also be checked. This is for using multiple log files across multiple nodes, which our project doesn't concern about so we will create a dummy index called index and leave index pattern checked and click on import.

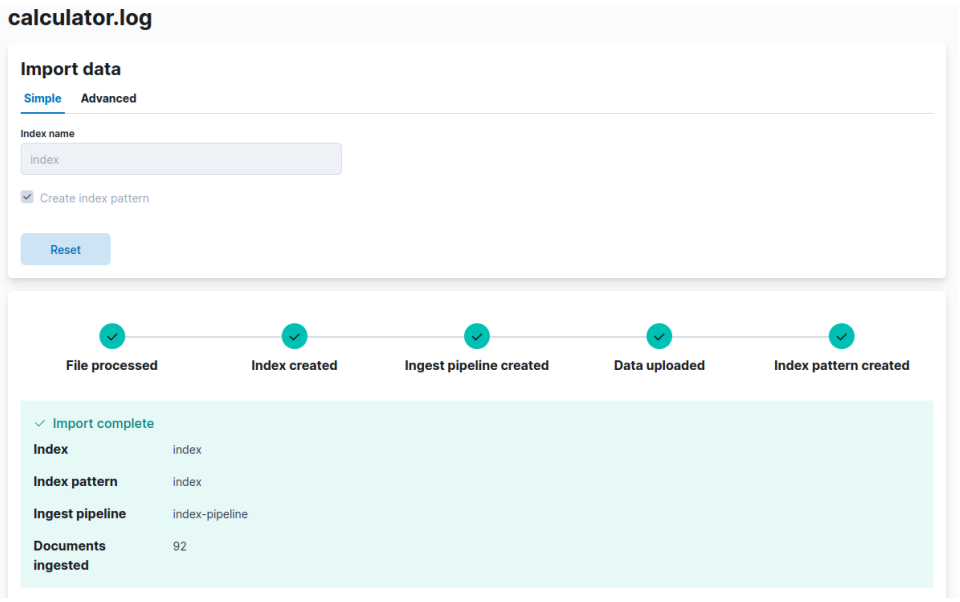

Fig 3.122 Index created and imported

If we scroll down, there will be an option called **View index in Discover** which will allow you to see our log file and analyse it.

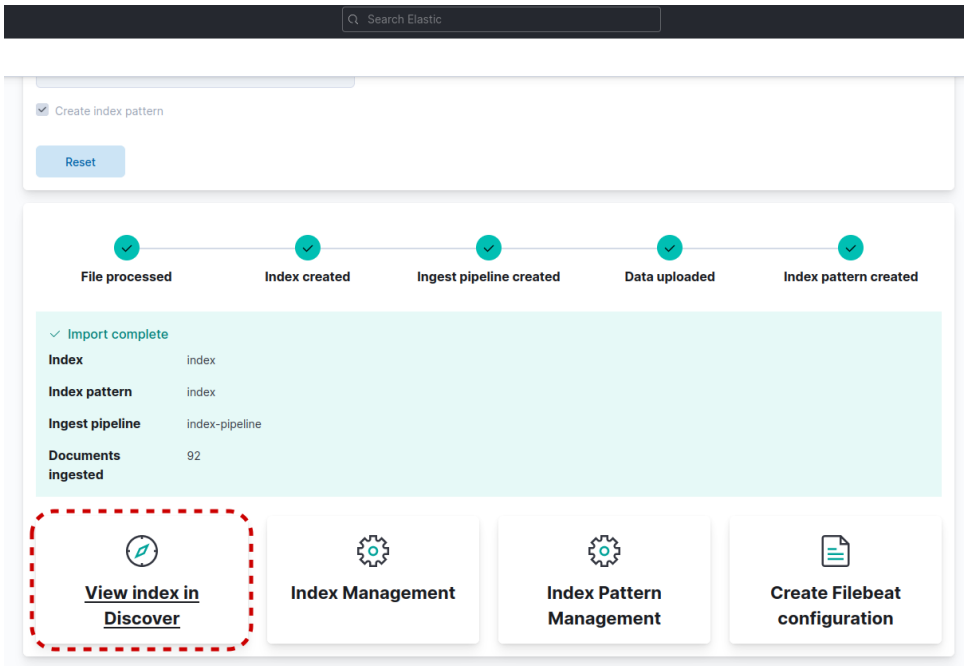

Fig 3.123 Discover index option

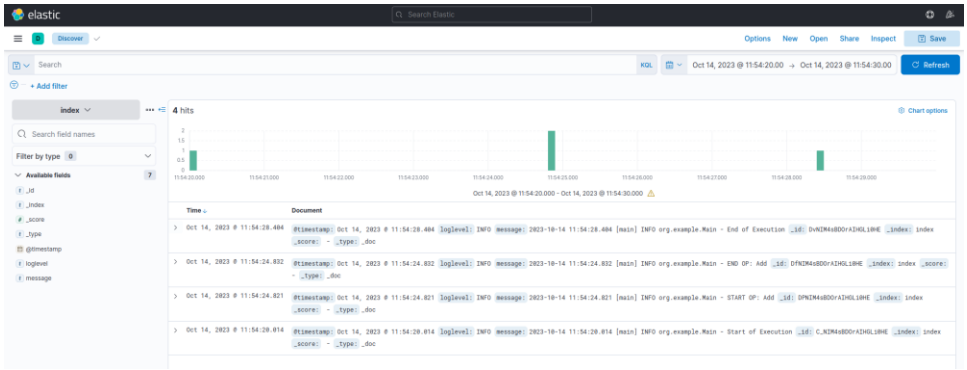

Fig 3.124 All our logs in one place with filtering options

From here, it is a matter of filtering the logs according to various parameters. You can filter based on the time stamp or on the fields that it identified. After you have filtered down the logs, you can generate visualisations that match your requirements and analysis. We will go through a few example ones:

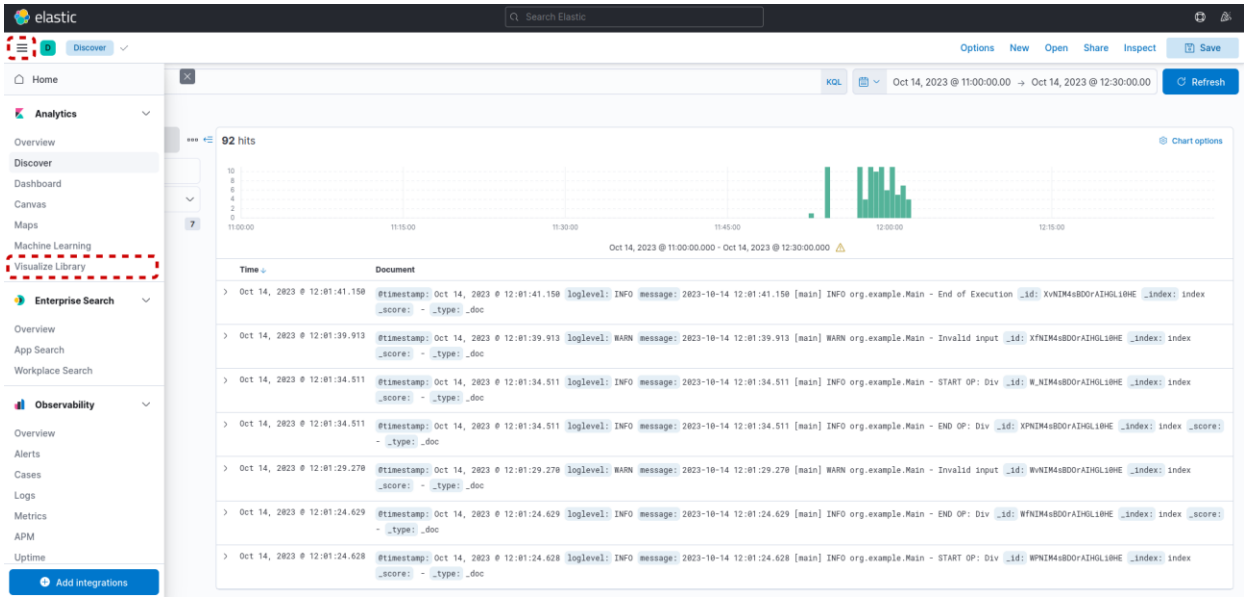

Fig 3.125 Visualise menu option in top left hamburger menu

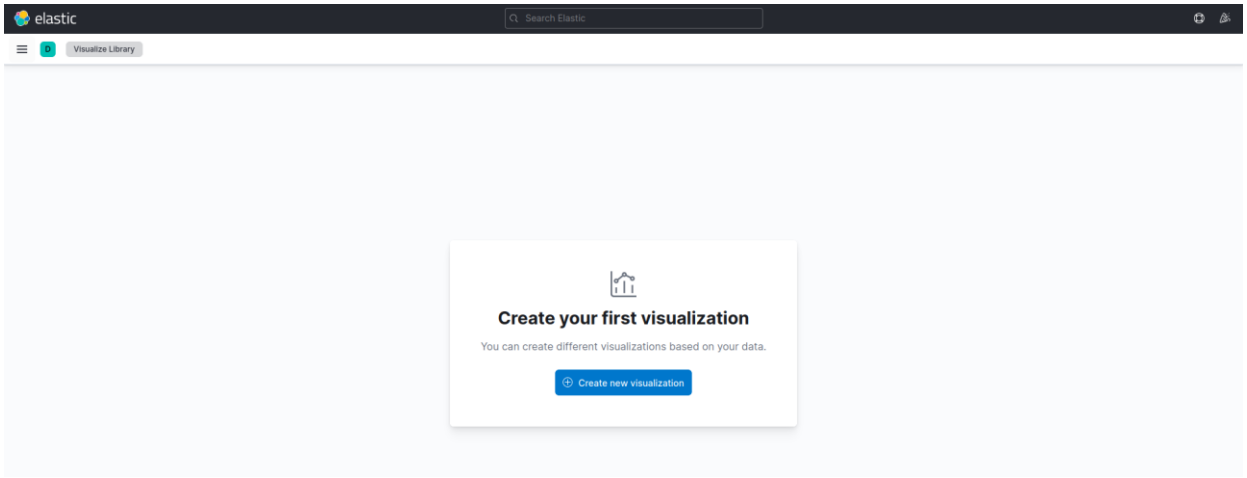

Fig 3.126 Creating new visualisation page

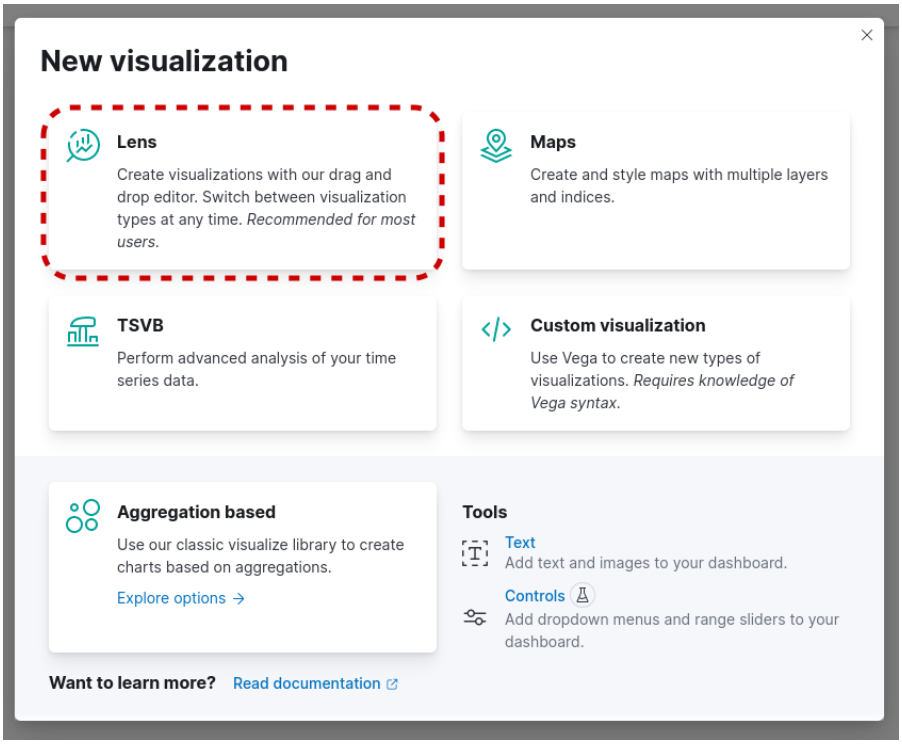

Fig 3.127 Choose lens visualisation

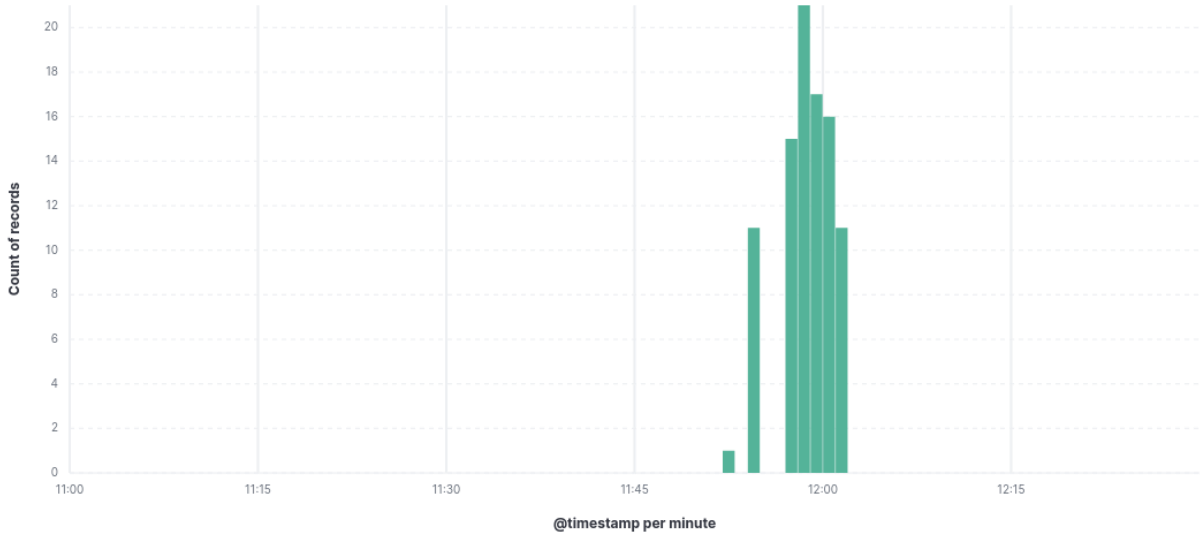

Fig 3.128 Bar graph showing how many log entries were created per minute

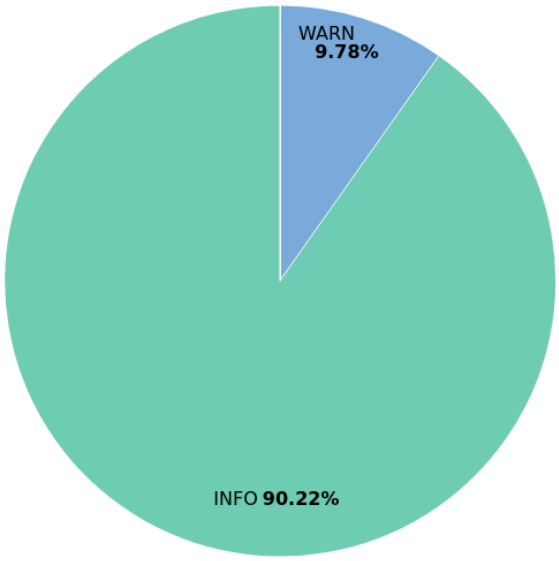

Fig 3.129 Pie chart of how much info versus warns were logged

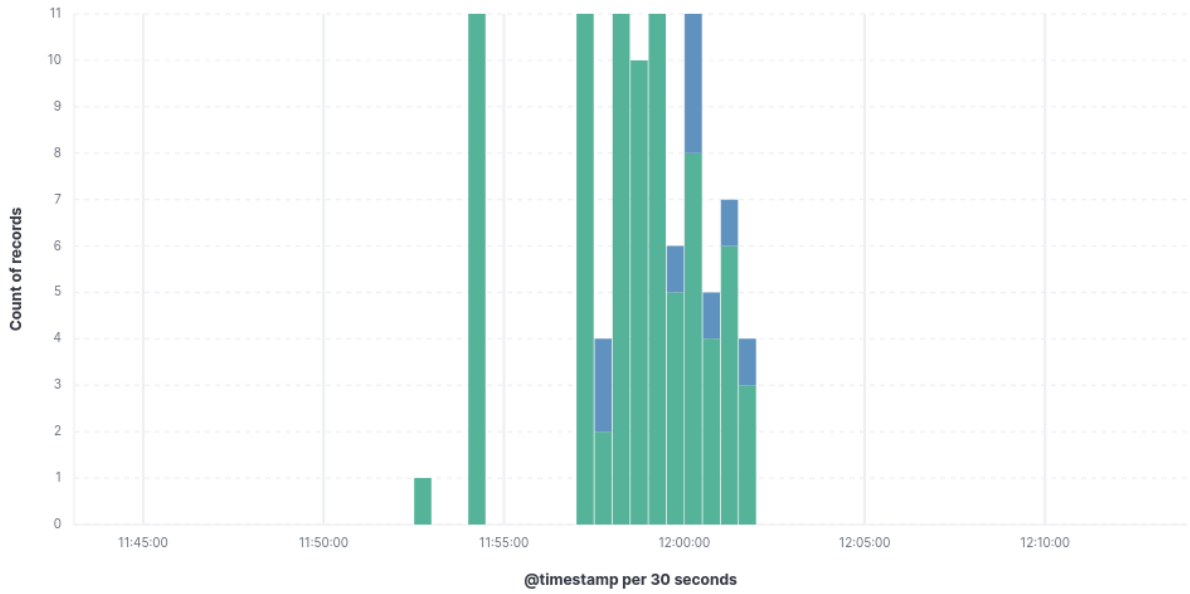

Fig 3.130 Bar graph showing how many warns and info were logged per 30 seconds

Future Roadmap

And there you have it. A simple program with a bare bones pipeline and visualisation of log files. This is by no means an extensive project but serves as a jump pad for all your future learnings. Want to have multiple containers instead of one? Want to automate the copying of log files from the container and into ELK? Want to write and automate tests based on the platform? All of these and much more form your exploration into the subject and a few of them are demonstrated down the line in upcoming case studies. But for now, congratulations on a job well done on your first project.