Creating a demo Java application and Push it to the Dockerhub using Github and Jenkins

To start, let’s create a new Maven project with any Java IDE.

In src/main/java path, let’s create a class called Main. It will contain the code for our simple console application.

In the Main class, write a small program to check if an input is even or odd

Here is the final code for the Main class:

public class Main {

public static void main(String[] args) {

System.out.println(checkIfInputIsAnEvenNumber(122)); // Testing in the main method

}

public static boolean checkIfInputIsAnEvenNumber(int number){

return number % 2 == 0;

}

}

If you run the above code, the output will be true.

In the code snippet above, we are creating a static method so that we can write unit tests. We want to see how Jenkins will automate testing.

* If the input int is even or odd, the method will return true or false respectively.

Now, let’s write a unit test to test our checkIfInputIsAnEvenNumber method. First, in the src/test/java path, let’s create a test class TestMain to test the method.

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.assertTrue;

public class TestMain {

@Test

public void testInputIsEven(){

assertTrue(Main.checkIfInputIsAnEvenNumber(23)); // Assertion

}

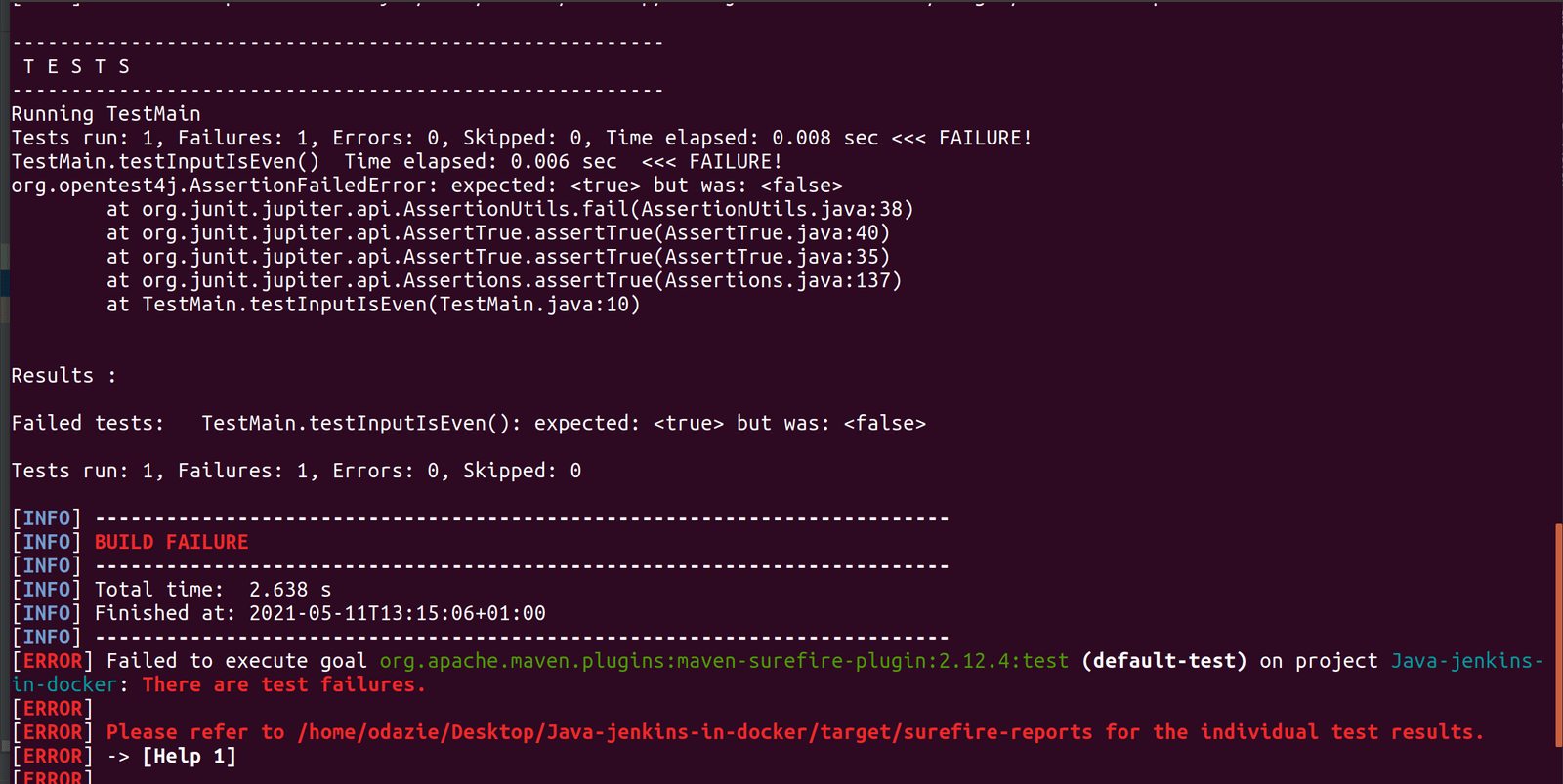
}

You can run the test above in your IDE.

Alternatively, we can use a Maven command to run all our unit tests in the command line, as shown below:

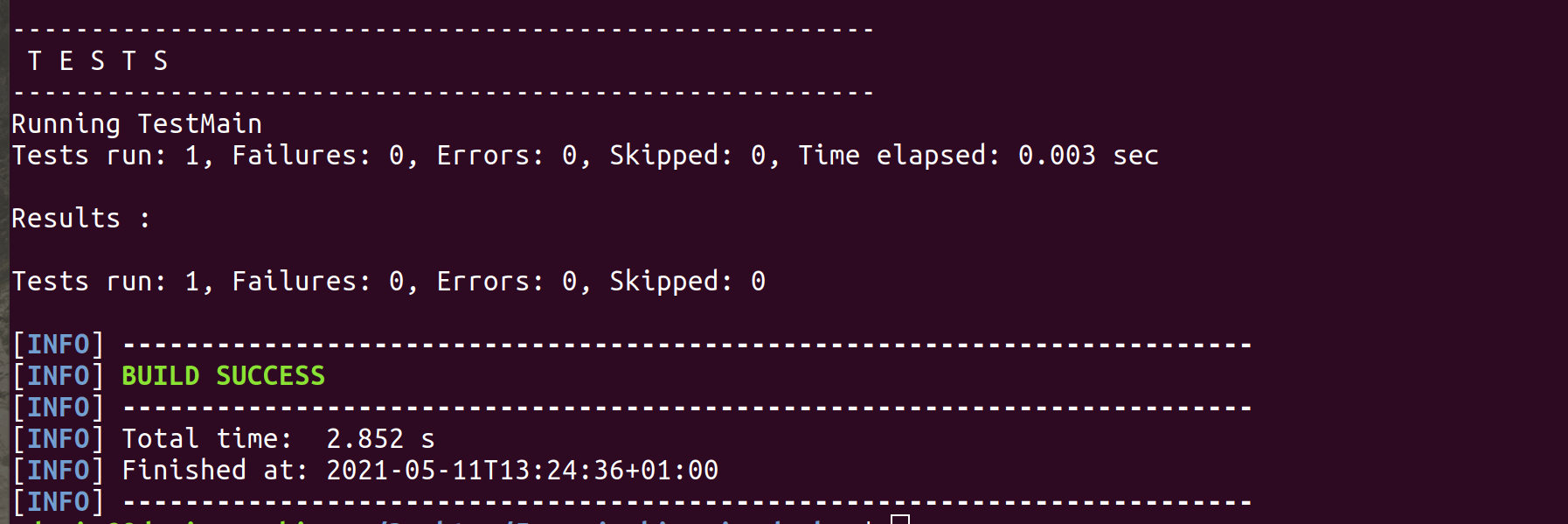
$ mvn test

When we use 23 as our input data, the test fails:



Let’s change the test input data to 22 and run the Maven command:

assertTrue(Main.checkIfInputIsAnEvenNumber(22)); // Assertion



The test passes. In a few steps, we will see how Jenkins can automate this process.

Hosting the demo application on GitHub

We are going to push our Java application code to GitHub. When we make any change (commit) to our application on GitHub, Jenkins will trigger a post-commit build process remotely.

* open up the terminal.
* Navigate to the directory of our demo application and run:

$ git init

* We will add all our application files using the command below:

$ git add .

* We can now commit our files:

$ git commit -m "Added java demo application files"

* Copy the created repository clone URL on GitHub.
* Then add the remote URL where we will push the local repository:

$ git remote add origin <REMOTE\_URL>

Verify the remote URL and push the changes of our local repository to Github:

$ git remote -v

$ git push origin main

Setting up Jenkins in Docker

Docker-in-Docker

As we set up Jenkins in Docker, we need to remember the goal of our setup: dockerizing of an application. For this to happen, we need to execute docker commands, as well as access other containers.

To achieve this functionality, we need a Dockerfile that configures a Jenkins environment. It will be capable of running Docker commands and managing docker containers.

Copy a Dockerfile in any directory and go to the directory from powershell

Run below command to create jenkins-docker image using the above Dockerfile :

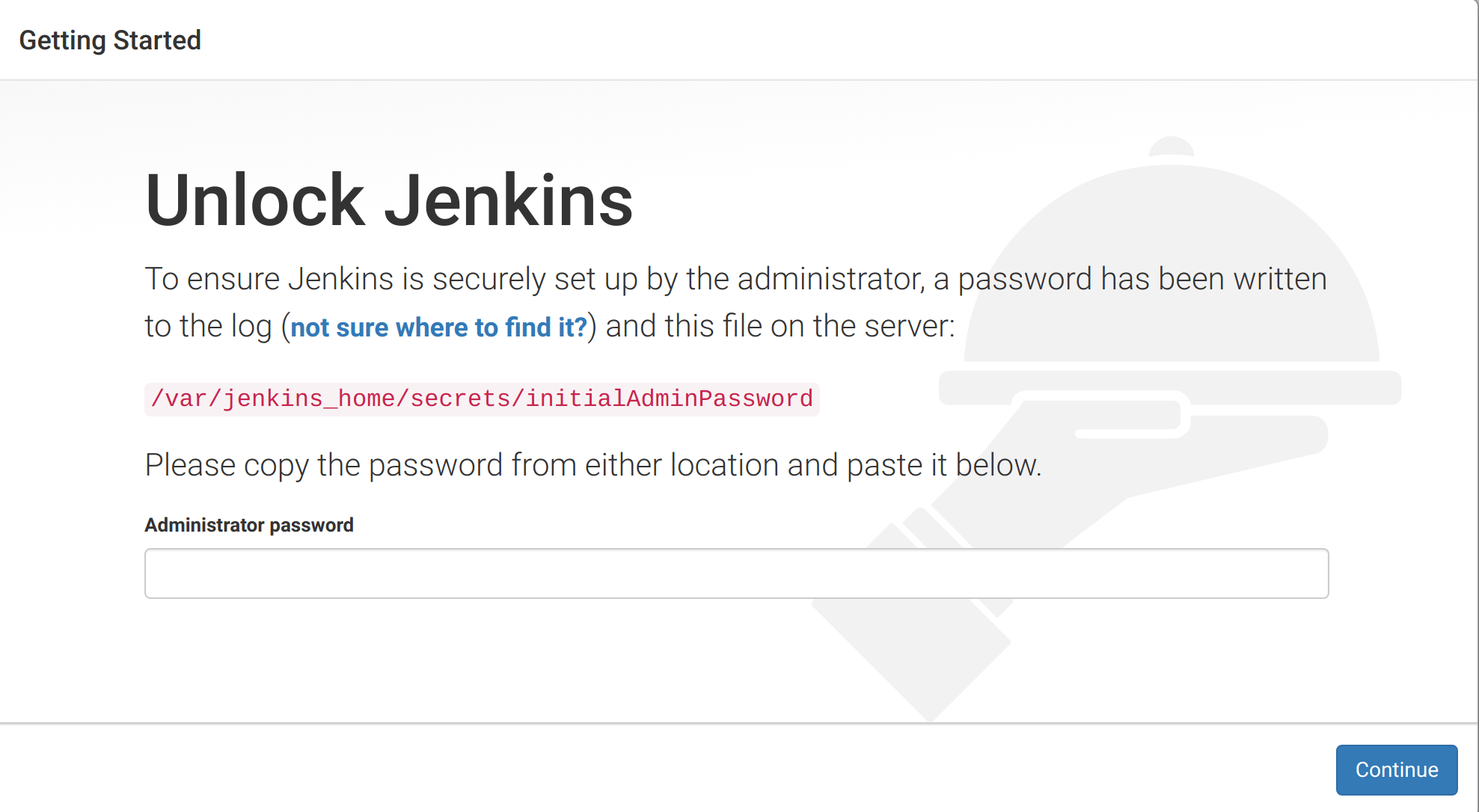
$ docker image build -t jenkins-docker .

To run our Jenkins-docker container in the command line, we use the code below:

$ docker run -it -p 8080:8080 -p 50000:50000 -v jenkins\_home:/var/jenkins\_home -v /var/run/docker.sock:/var/run/docker.sock --restart unless-stopped jenkins-docker

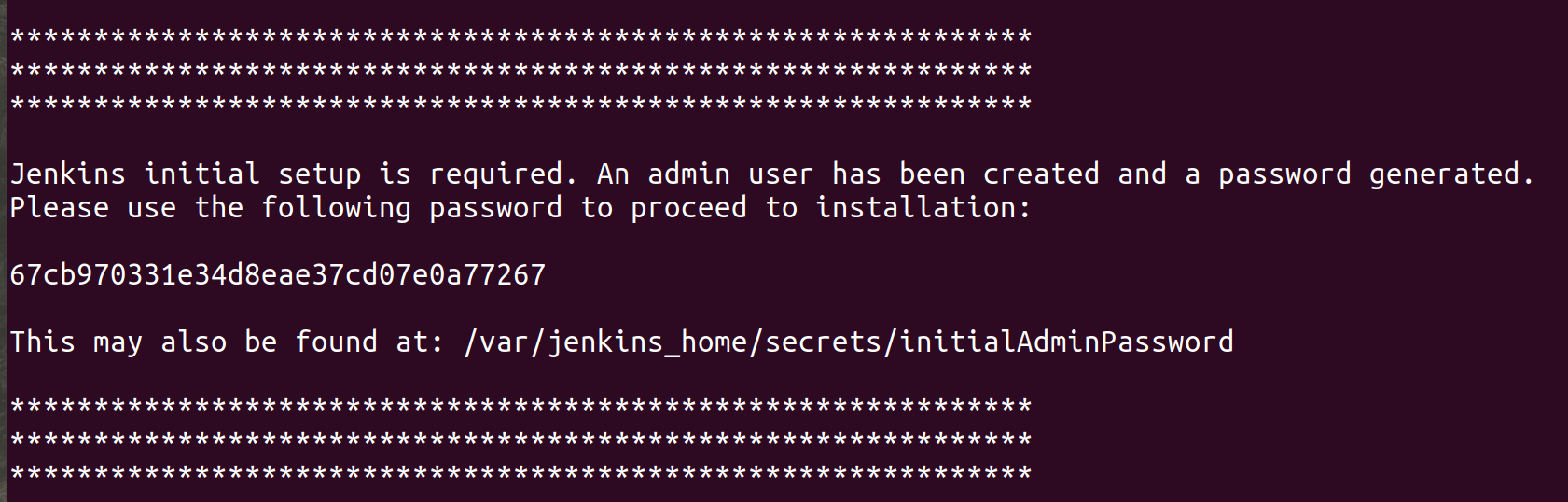
* The above command runs our pre-built jenkins-docker image. The -p command publishes the container’s ports 8080 and 50000 to the host machine.
* We should run Docker commands in our Jenkins container. However, there is only one Docker daemon running in our machine at a time. So what we need to do is to [bind mount](https://docs.docker.com/storage/bind-mounts/) our container to our host machine daemon while we run the container using this argument: -v /var/run/docker.sock:/var/run/docker.sock
* -v jenkins\_home:/var/jenkins\_home argument creates an explicit volume on our host machine. Why? During our initial setup, we will configure Jenkins and download plugins. When we stop/restart/delete our container, we need to have our initial setup configuration intact. We wouldn’t want to be doing those set ups every time we stop/restart/delete our container.
* --restart unless-stopped ensures that the container always restarts unless stopped using the docker stop <container\_name/container\_id> command.

After running the above command, visit localhost localhost:8080 to set up Jenkins.



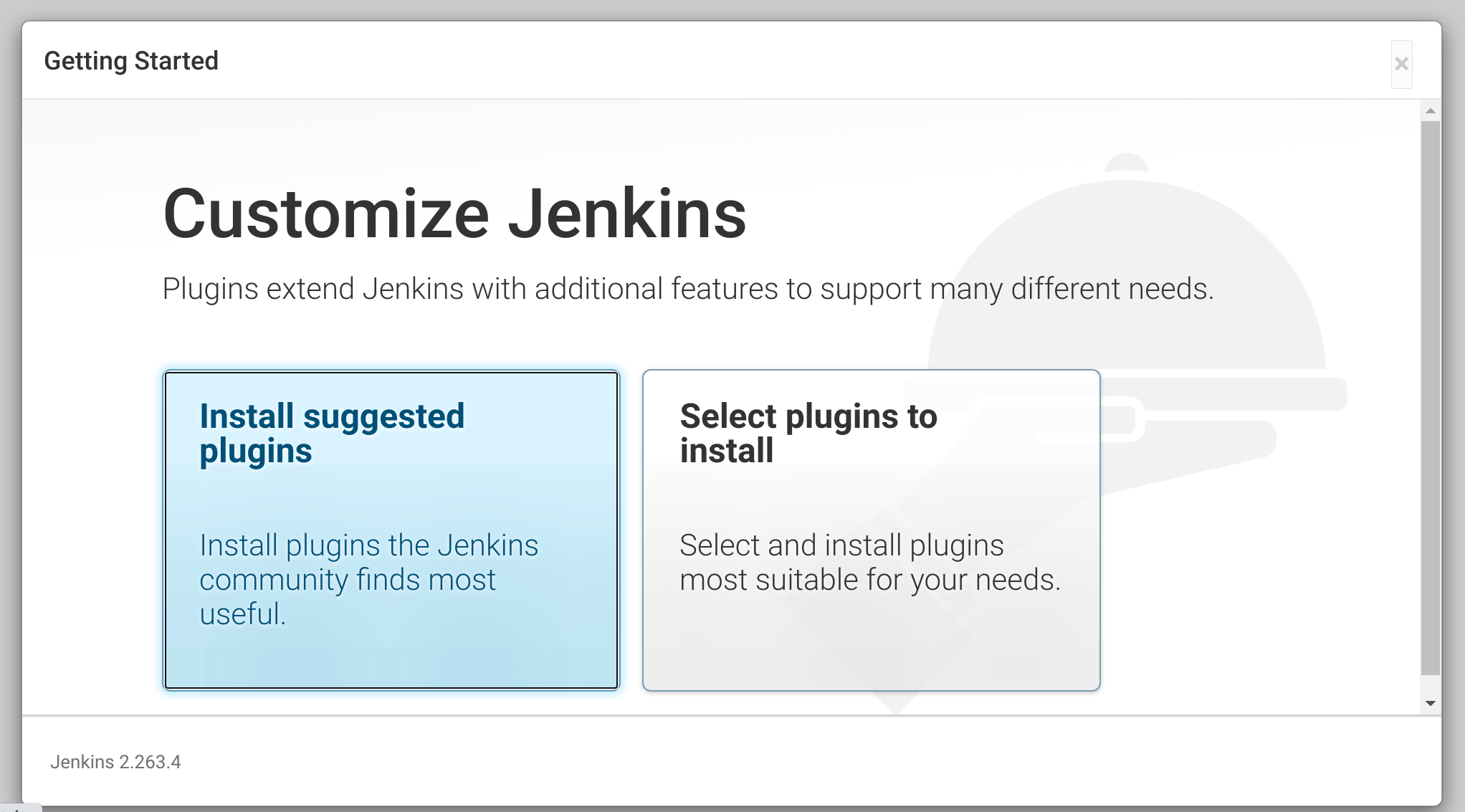
We can get the admin password from what command returns.

See what is looks like:



Next, we select Install suggested plugins.

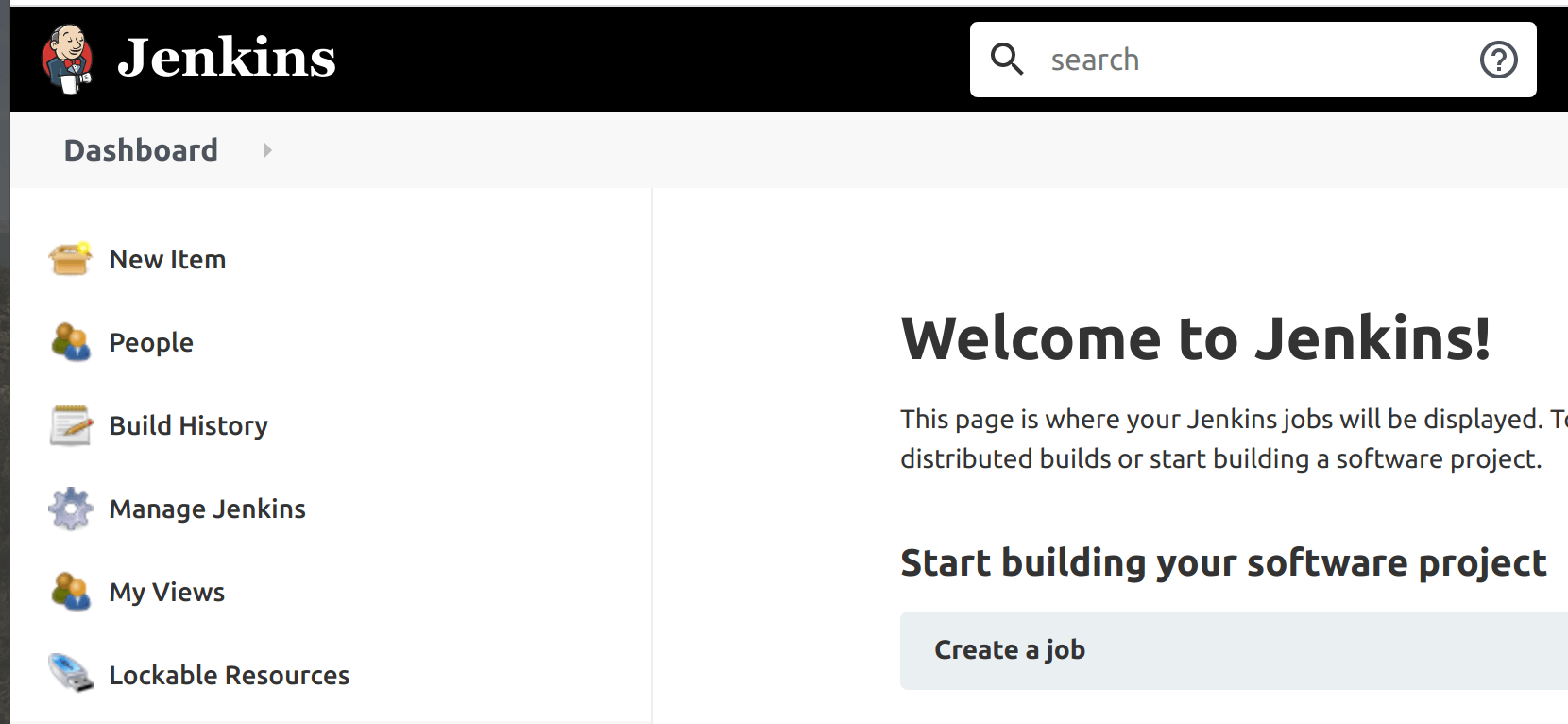
Jenkins will automatically download essential plugins:



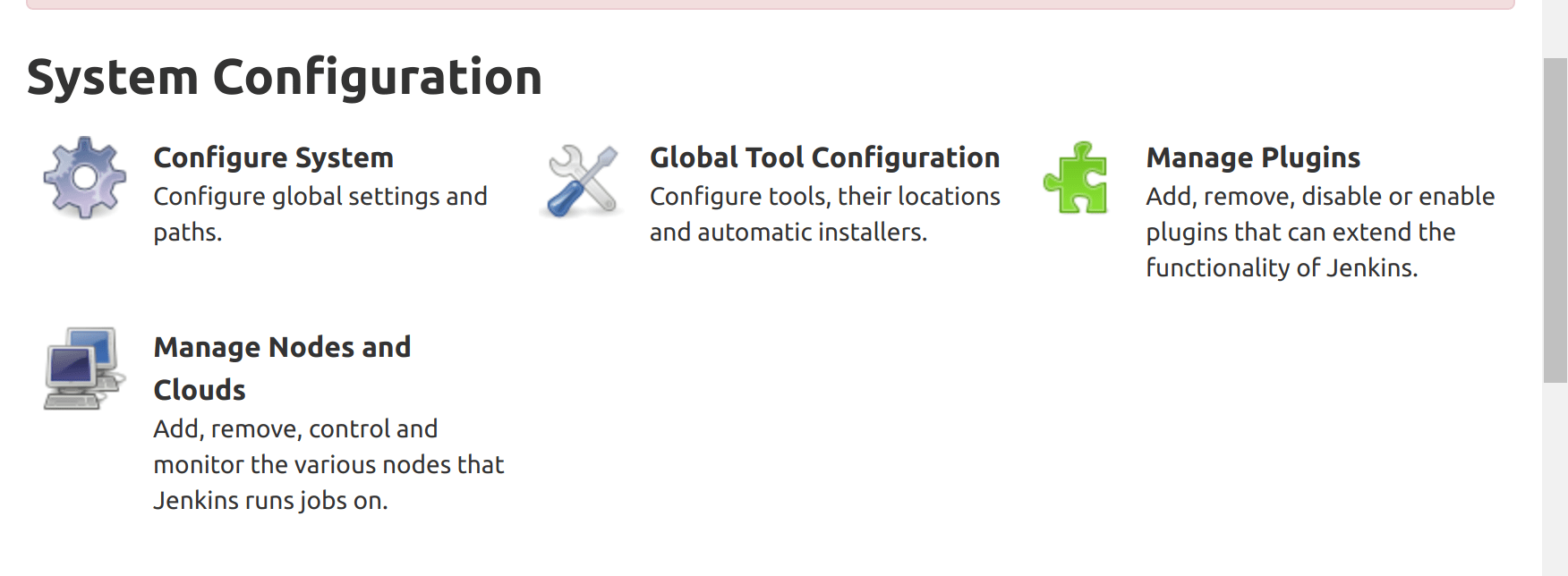
**Jenkins global configurations**

First, we will configure the JDK, Maven, and Git on our Jenkins console to enable Jenkins to clone our repository and build our application.

In our Jenkins console, go to Manage Jenkins.



Under System Configurations, click on Global Tool Configuration.



**JDK config**

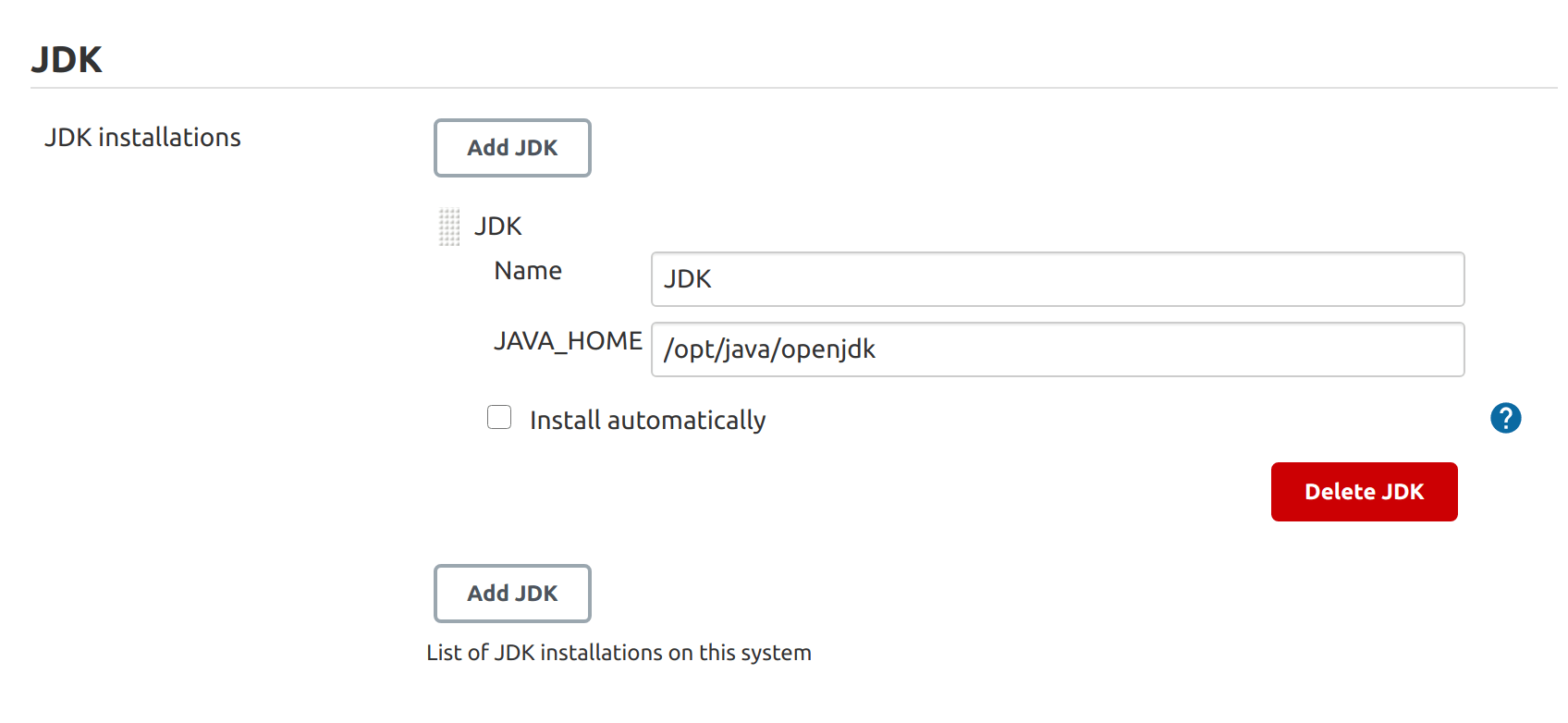
Our Jenkins container comes with an OpenJDK. To find it, we need to enter into the container’s bash shell to get the JAVA\_HOME path.

To get the bash shell of the container run below command in another powershell window, don’t close the earlier powershell windo because Jenkins is running

$ docker exec -it <container\_name/container\_id> /bin/bash

Then if we’re using either macOS or Linux, we run:

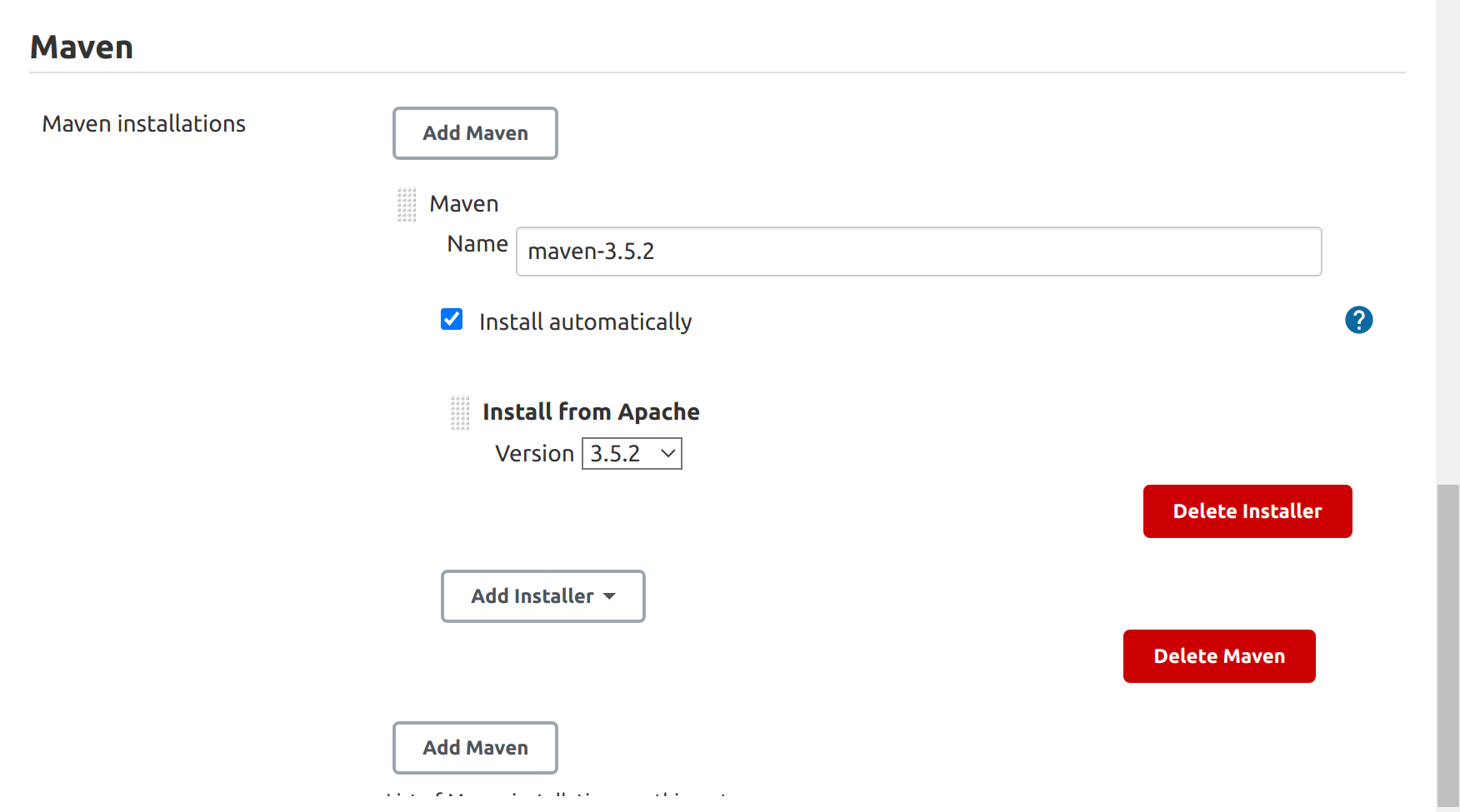
echo $JAVA\_HOME



Maven config

We can direct Jenkins to download Maven from Apache servers instead of the Maven directory on our system.

Follow the guideline shown in the image below:



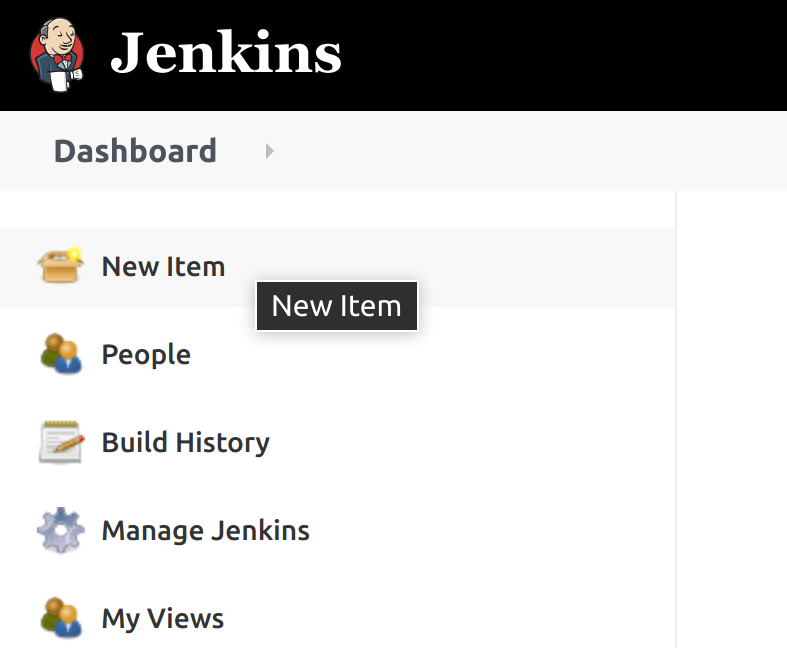
Make sure to save the configurations before exiting the page.

Putting it all together

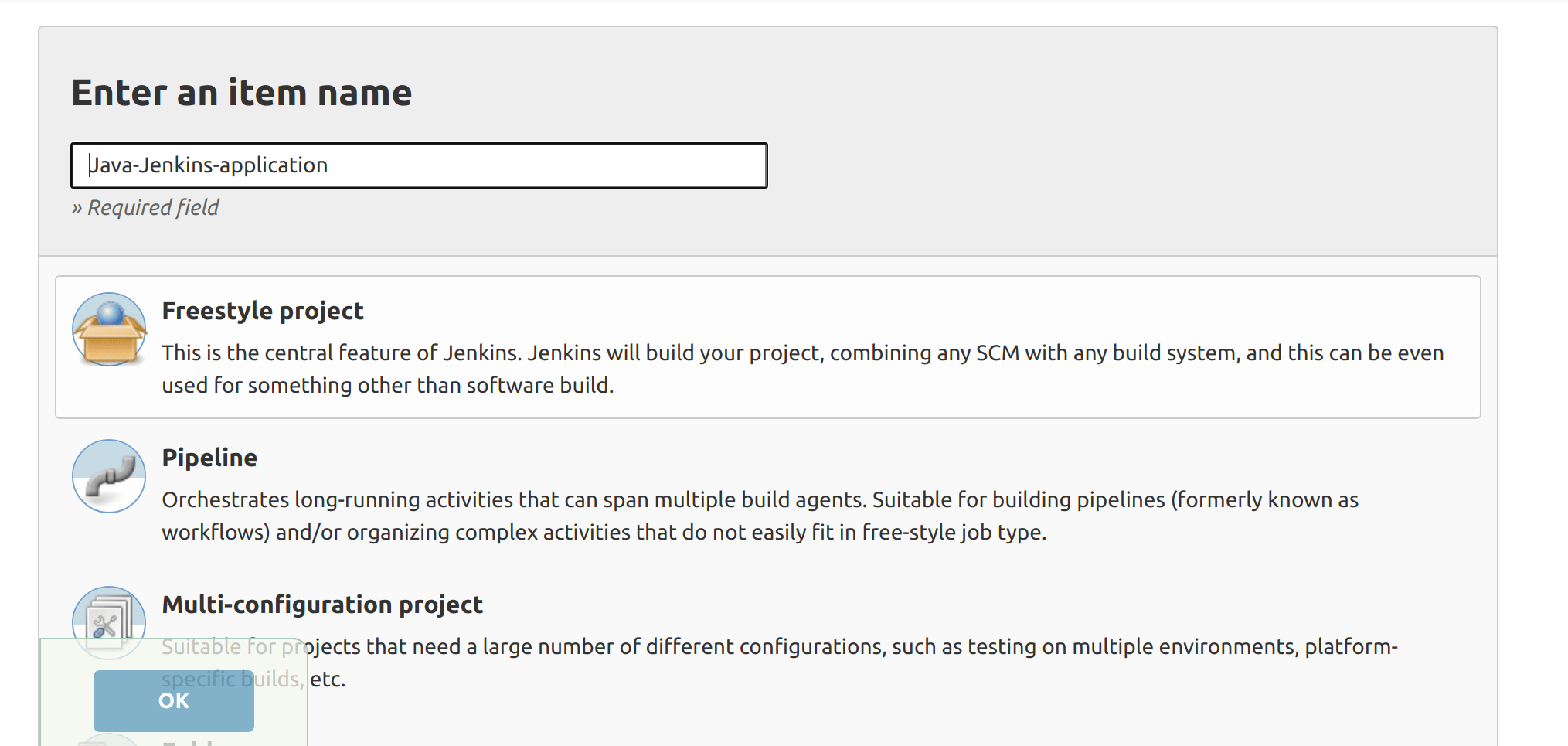
So far, we’ve built a simple demo Java console application, hosted our application code on Github, and set up Jenkins in Docker.

Now let’s put it all together by using Jenkins to automate the building, testing, dockerizing, and deploying our application Docker image to Docker Hub after every commit made to our application repository hosted on GitHub.

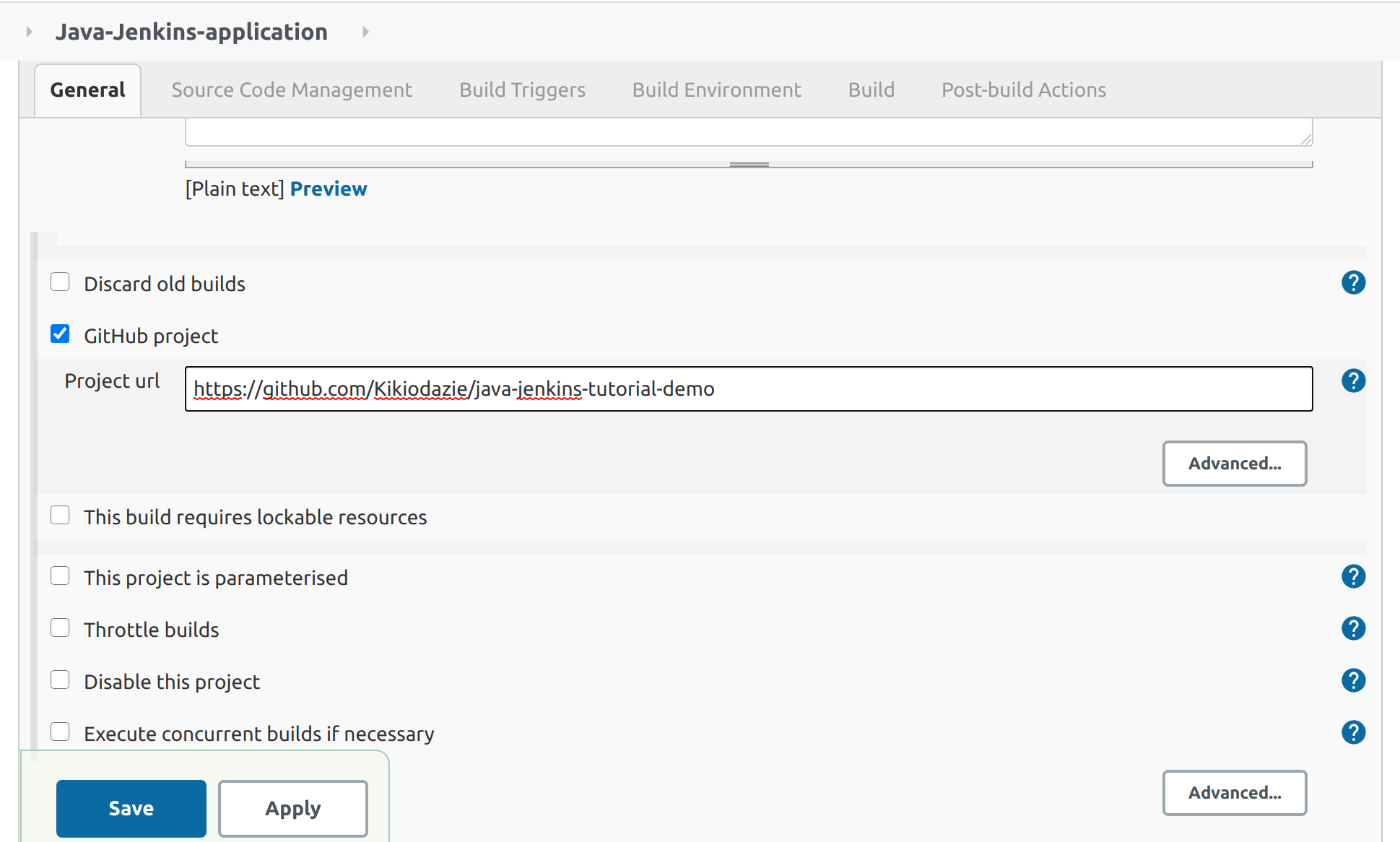
To start, let’s create a new Jenkins item:



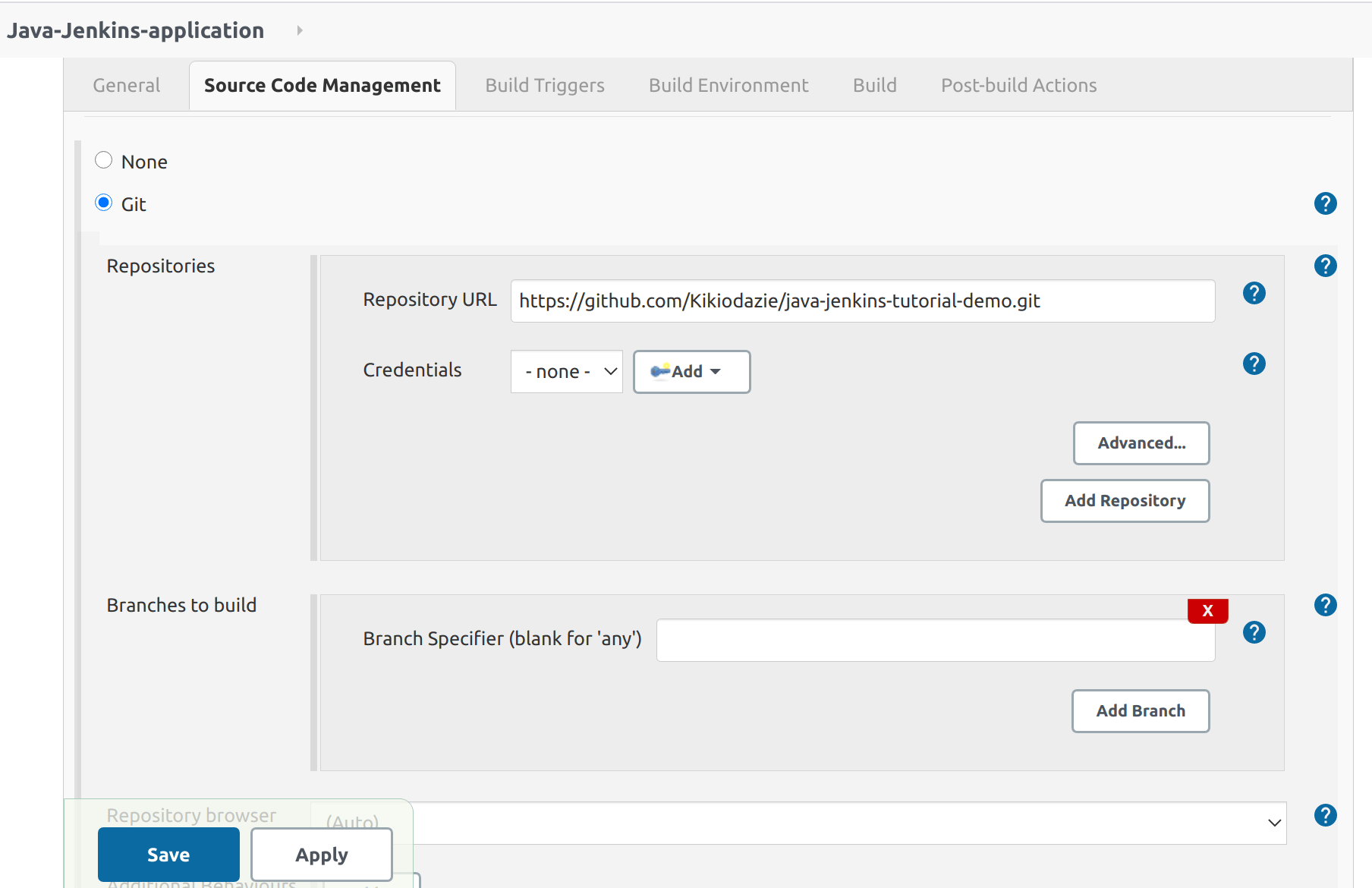
Then select Freestyle project:



To configure our Freestyle project, select GitHub project and add the project URL:



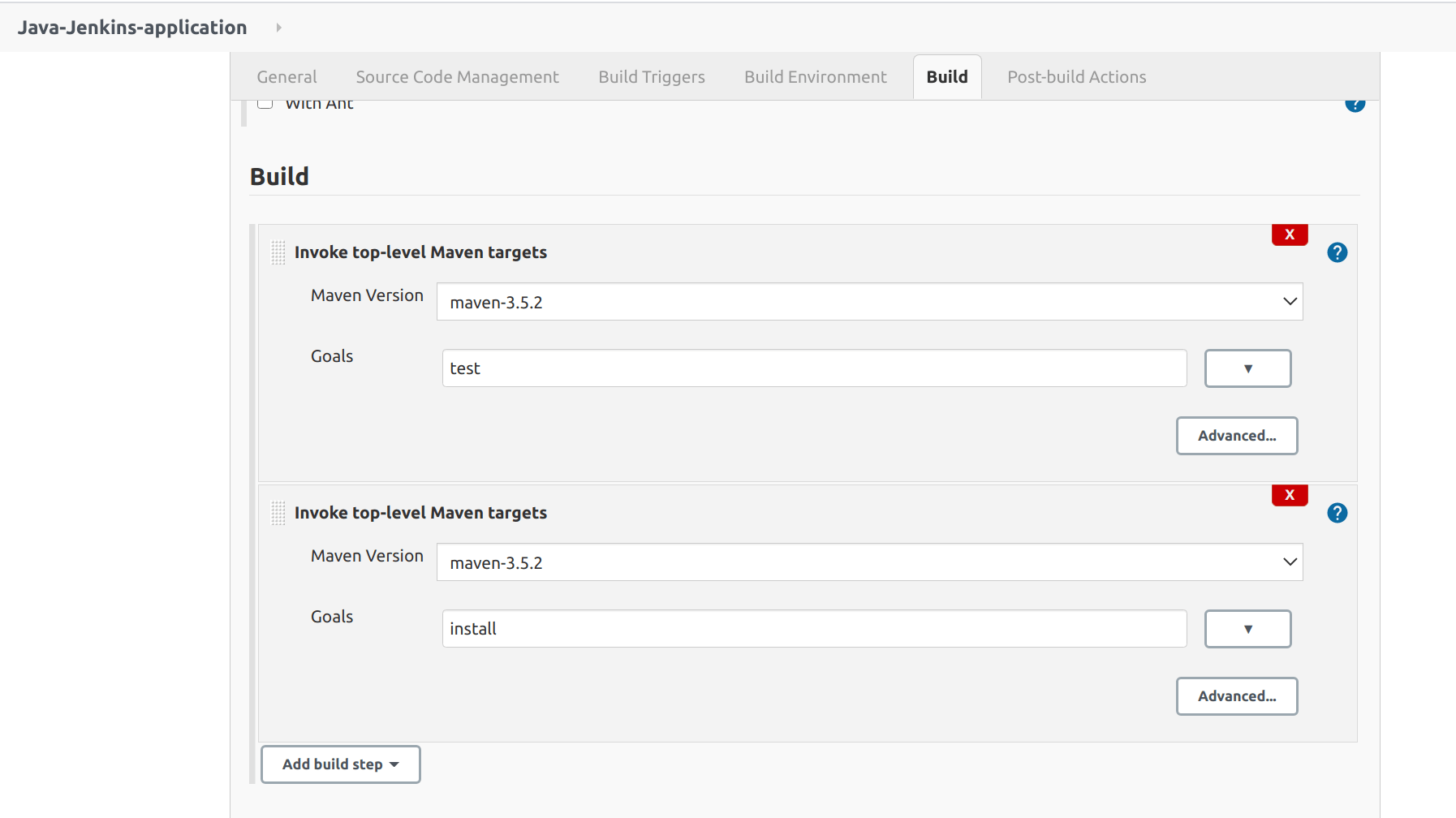
For our Source Code Management (or **SCM** for short), select Git, add the remote Git repository URL of the project and leave the branch field empty so any commit made to any branch triggers our entire Jenkins process:



For Build Triggers, select Poll SCM, which checks whether we made changes (i.e. new commits) and then rebuilds our project. Poll SCM periodically checks the SCM even if nothing has changed in the repository.

Next, we skip the Build Environment tab. In the Build window, we will add two Invoke top-level Maven targets steps.

Finally, we click on apply and save our Freestyle project configuration.

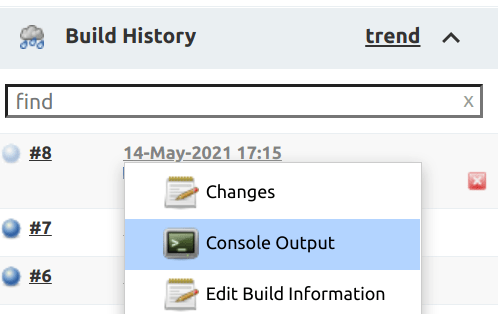


The above build steps run $ mvn test and $ mvn install commands automatically. If you recall our previous steps, we manually ran the test command for our unit test.

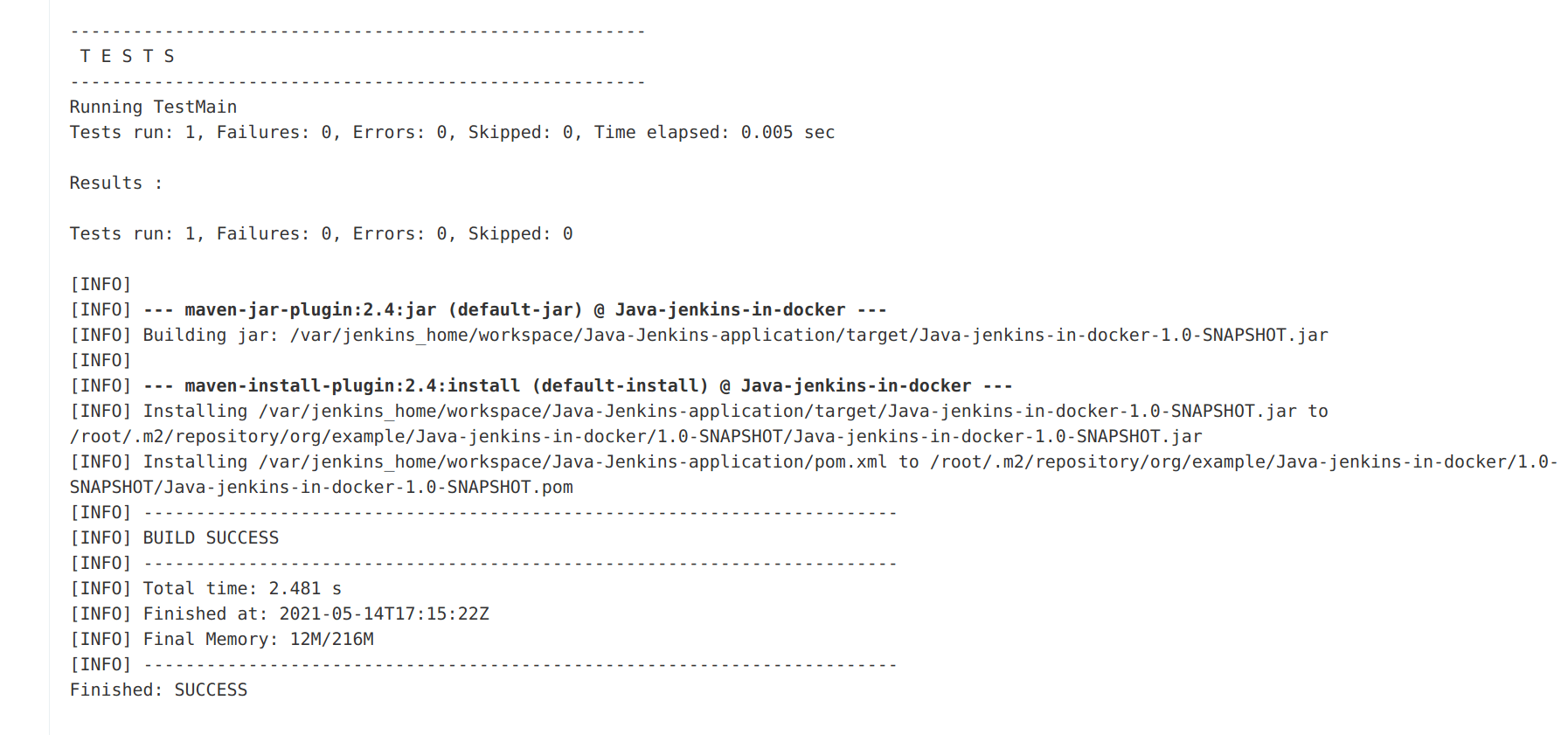
For testing purposes, let’s build our project to see if the current configuration works. Click on Build Now.



We can view the console output in the Build History:



Our console output should look a lot like the image below:



If we commit changes, we don’t need to manually click Build Now. Jenkins will automatically build our Freestyle project.

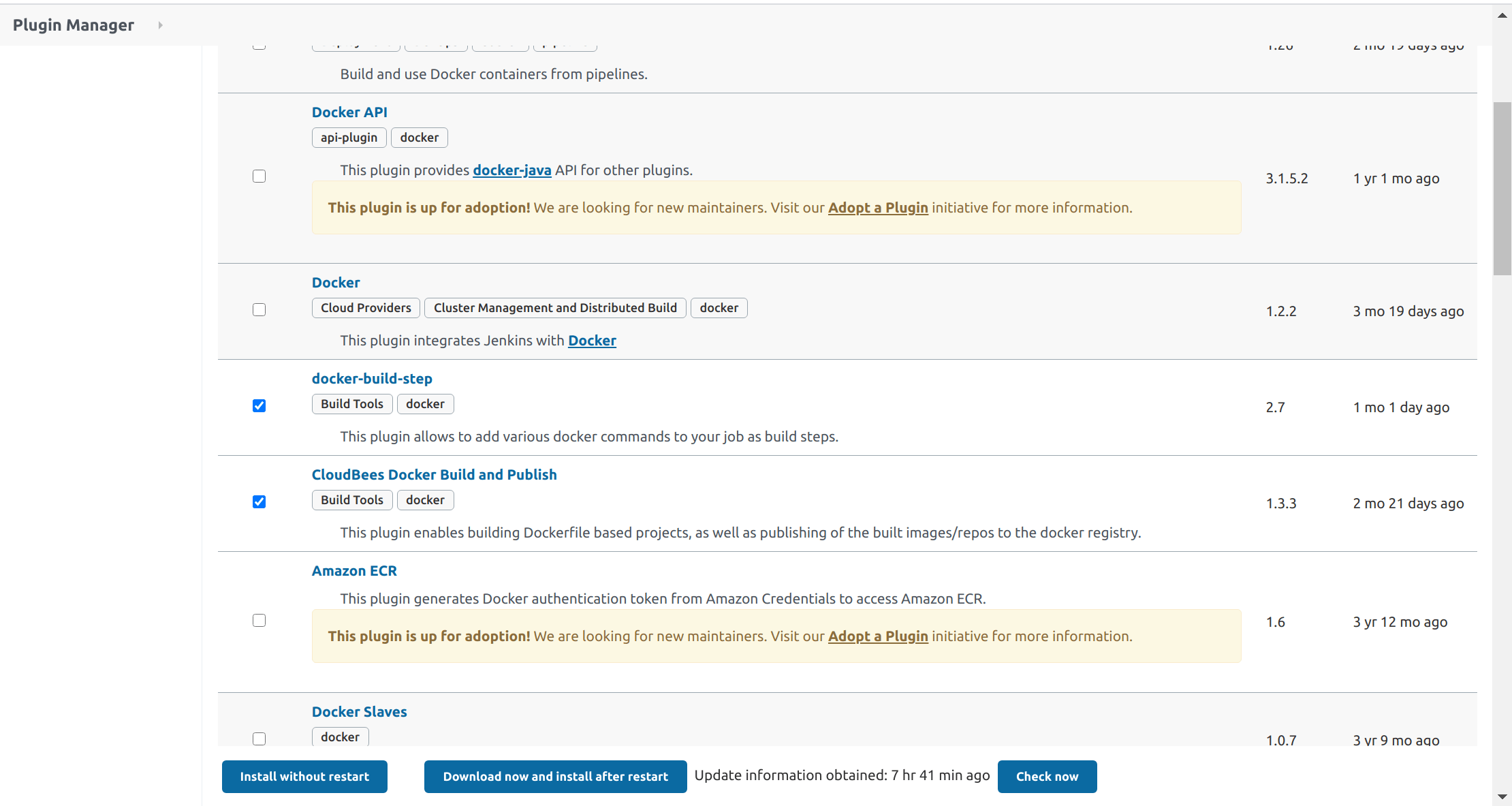
Building and deploying our Docker image to Docker Hub

We are almost there. What’s left is for us to configure Jenkins to build the Docker image of our Java application and deploy that image to Docker Hub.

To achieve this, we need a few Jenkins plugins installed.

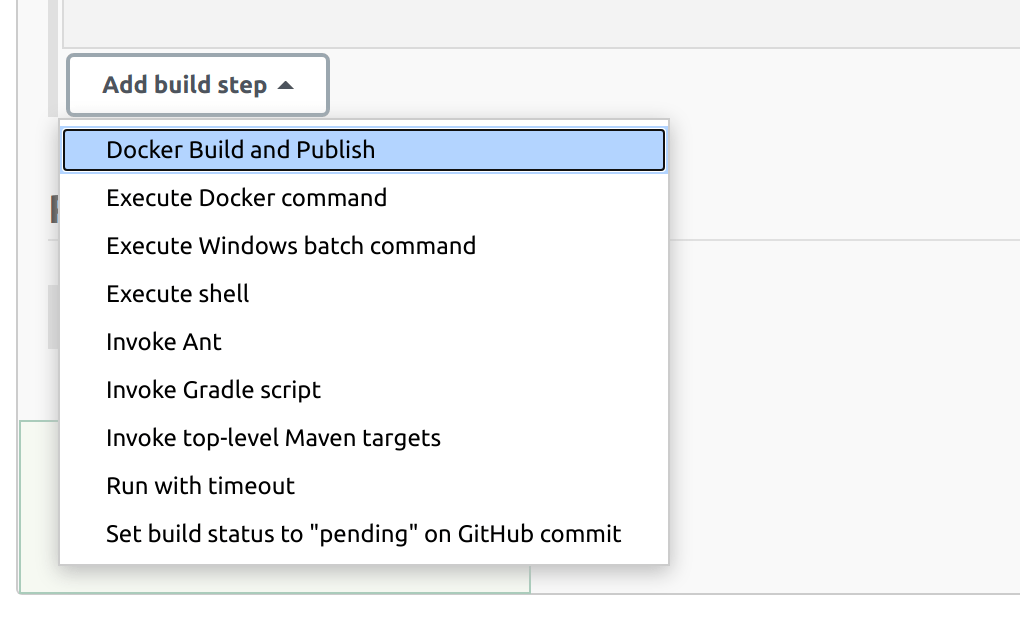
In Manage Jenkins, select Manage Plugins under System Configurations, search and install the following plugins:

* docker-build-step
* CloudBees Docker Build and Publish



To check if the plugins have been installed, let’s go back to our Freestyle project configuration and in the Build tab, click on Add build step.

We will see the Docker Build and Publish option:



To build a Docker image, we need a Dockerfile to notify docker which base image to build our image from and other Java-related configurations. We also need to generate a JAR (Java ARchive) file.

In the build profile, navigate to the pom.xml file and add a [finalName](https://kb.novaordis.com/index.php/Maven_pom.xml" \l ":~:text=finalName%20modifies%20the%20name%20of,named%20artifacts%20in%20the%20repository.).

This finalname will be our JAR name:

<build>

<finalName>java-jenkins-docker</finalName>

</build>

Now let’s create our Dockerfile into the docker container running in another powershell window

Go to the Jenkins directory path

Open the terminal and navigate to our Java application directory:

$ touch Dockerfile

And in our Dockerfile:

FROM openjdk:8

ADD target/java-jenkins-docker.jar java-jenkins-docker.jar

ENTRYPOINT ["java", "-jar","java-jenkins-docker.jar"]

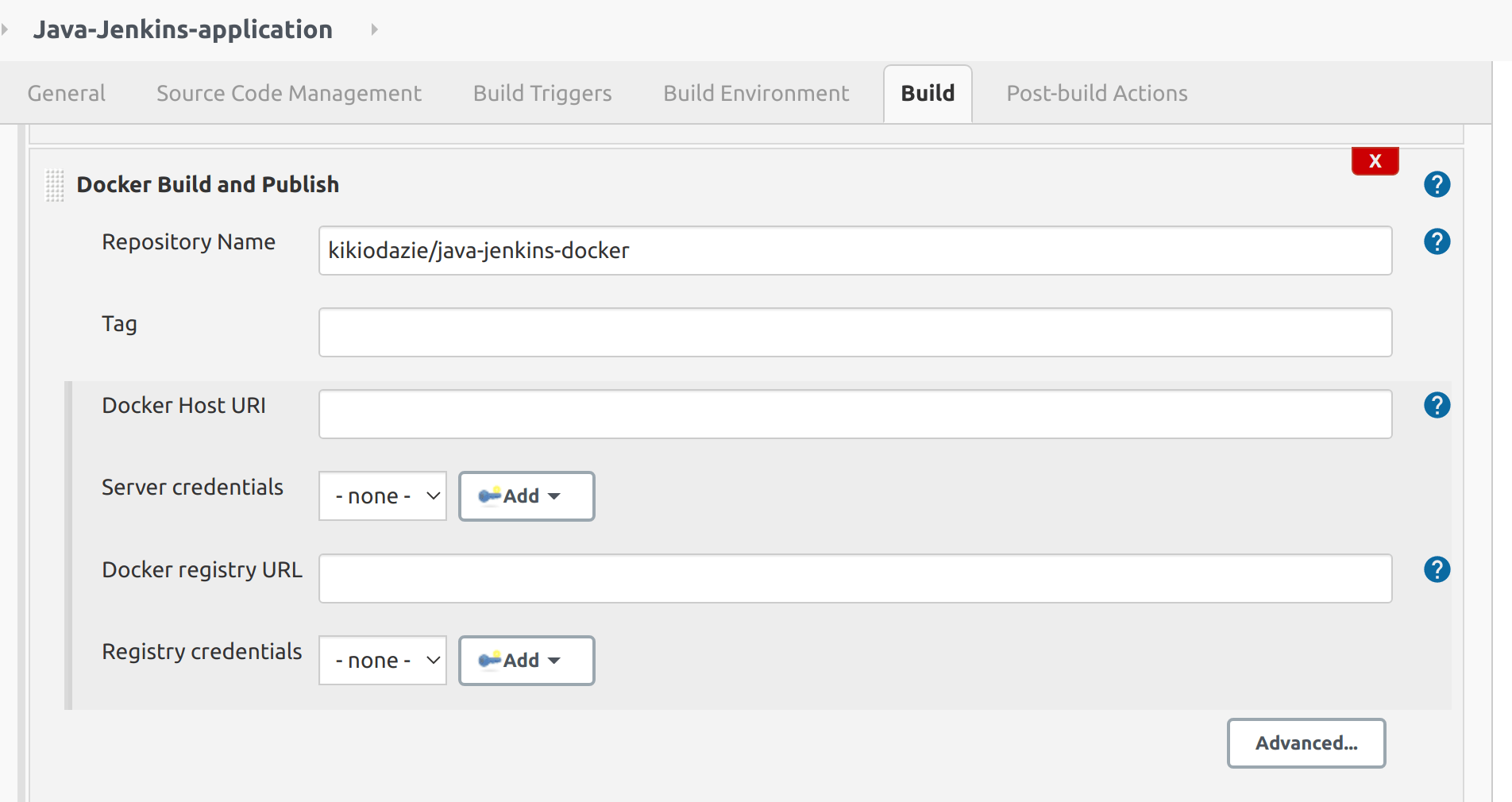
EXPOSE 8080

Add the new files and then commit the changes to the GitHub repository. This will trigger a Jenkins post-commit build process as we configured.

Now we can add our build steps to build and deploy our Java application’s Docker image. For this, we will need a Docker Hub account. You can create one [here](https://hub.docker.com/signup).

Then, in the build step set:

* Repository name: Docker\_id/jar\_name example kikiodazie/java-jenkins-docker
* For this demo, we will leave the rest of the fields empty then Apply and save.



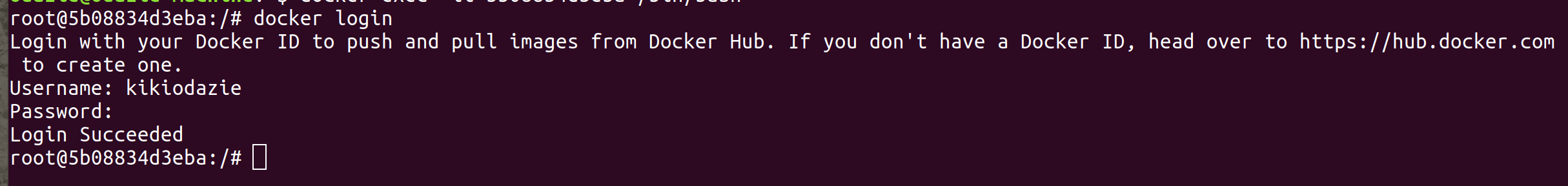
To give Jenkins access, we need to login to our Docker Hub account inside our Jenkins container through the command line, as shown below:

$ docker exec -it <container\_name/container\_id> /bin/bash

Then inside the container, run the Docker login command:

$ docker login

To complete this process, input your login credentials:



Go back to your project and click Build Now, then navigate to the console output. The output should look, as shown in the image below.

This means that our image has been successfully built and pushed to Docker Hub:

