Exercise: CI/CD with Jenkins

Exercises for the "Software Engineering and DevOps" course @ SoftUni.

1. Install Jenkins

Our first task is to install Jenkins on our machines.

In order to do that, follow this link: https://www.jenkins.io/download/ and chose the package that is suitable for you.
The installation for the different operating systems and their distributions are different. You can find the instructions that you need here: https://www.jenkins.io/doc/book/installing/. Simply chose your OS and follow the instructions.

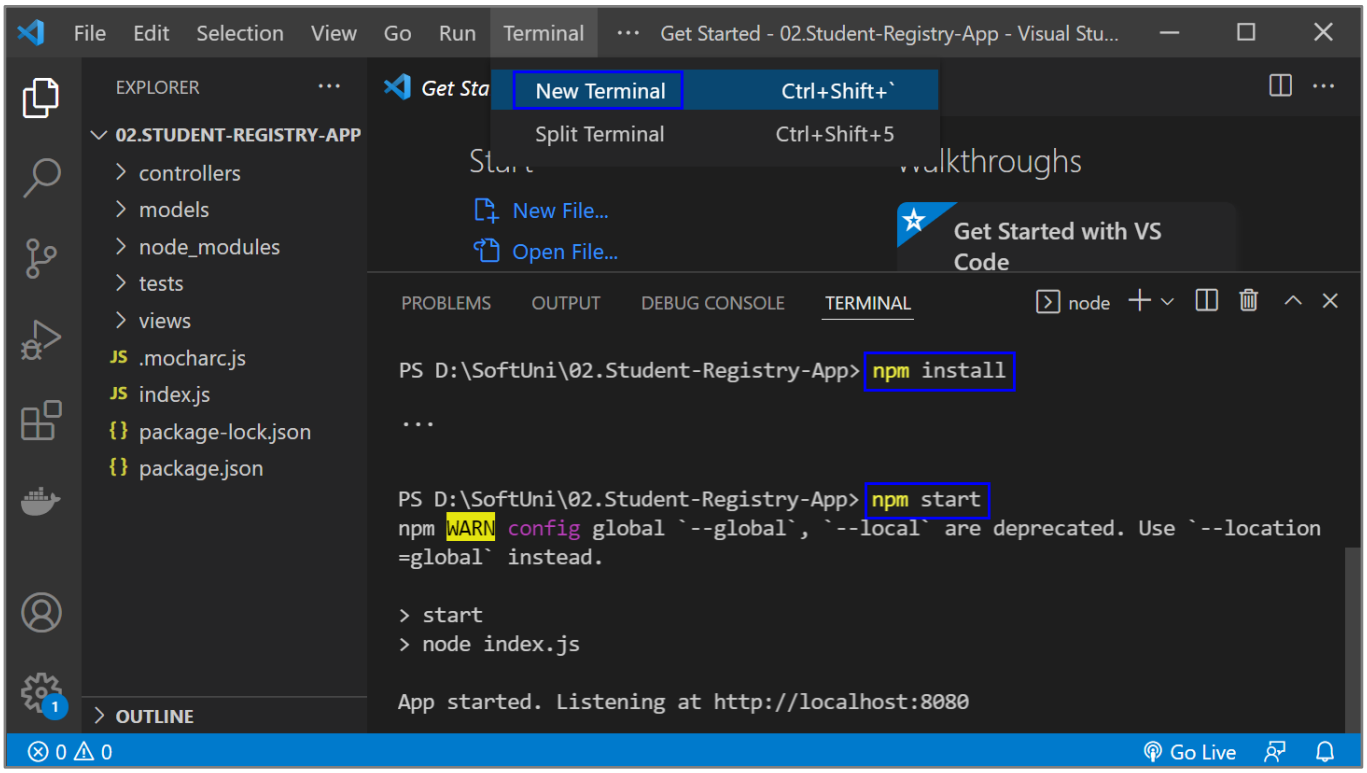
After you have installed Jenkins, follow the **Post-installation setup wizard** in order to **start** using Jenkins. Without completing the steps from it, you won't be able to use it. This is a one-time setup, so don't worry – you won't need to complete those steps each time you want to work with Jenkins.

2. CI Pipeline – "Student Registry" App

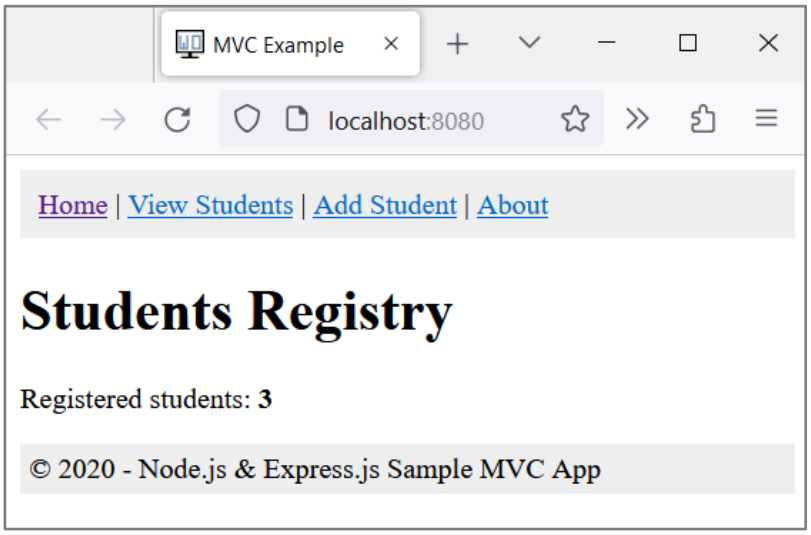
Step 1: Run the App Locally

We have the "**`Student Registry`**" **`Node.js` app** in the **resources**. Your task is to **create a CI workflow** with **Jenkins** to **start and test the app** on three different https://www.jenkins.io/doc/book/installing/versions:

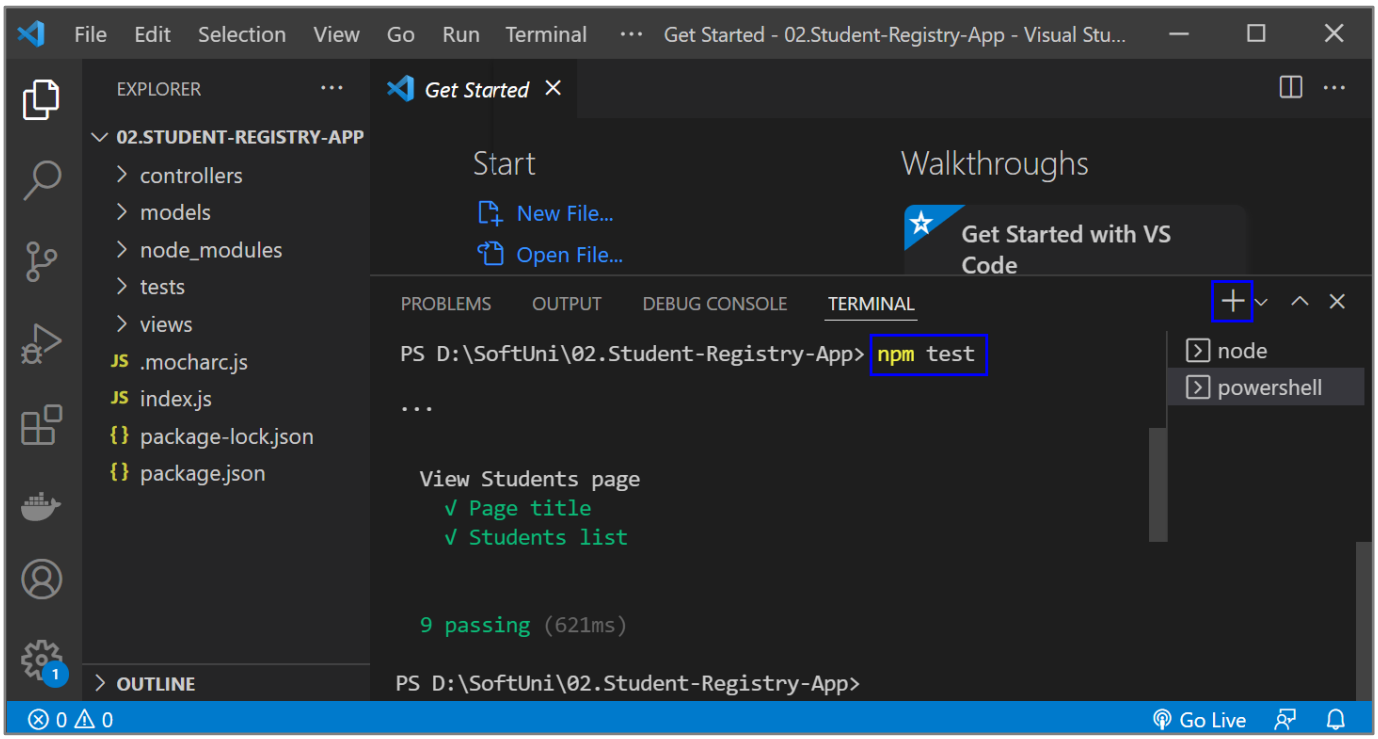
Let's first **start the app locally** in **`Visual Studio Code`**. To do this, you should **open the project**, open a **new terminal** from **`[Terminal]`** → **`[New Terminal]`** and **execute** the "**`npm install`**" and "**`npm start`**" **commands**:



The "**`npm install`**" **command installs app dependencies** from the **`package.json` file** and "**`npm start`**" **starts the app**. You can **look at the app** on http://localhost:3030:



Then, you can **return to `Visual Studio Code`**, open a **new terminal** with **`[+]`** and **run** "**`npm test`**" to **run the app tests**. They should be **successful**:



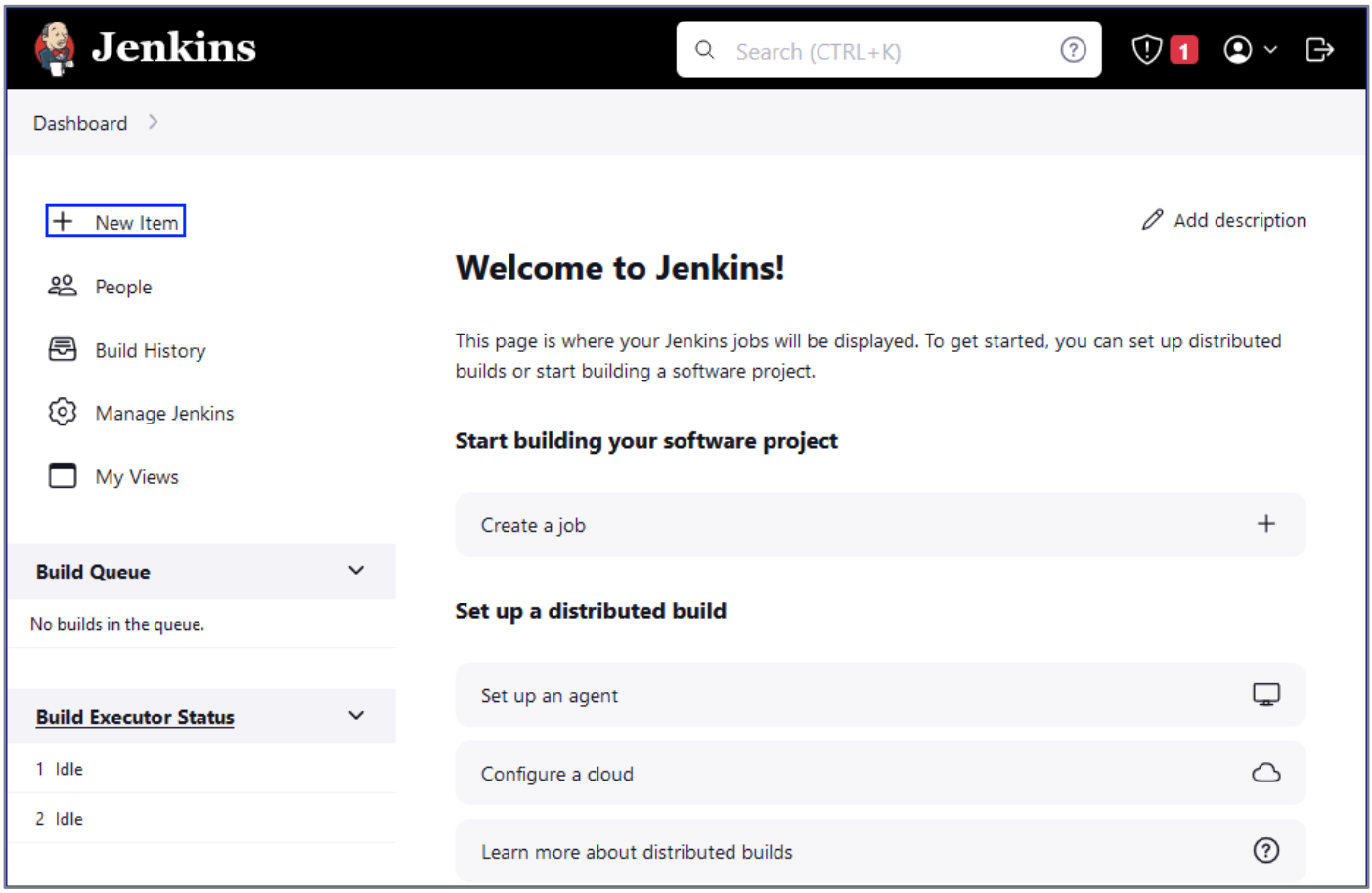
NOTE: if the **app was not started**, **tests would fail** because these are integration tests and are executed on the running app.

Step 2: Create a GitHub Repo

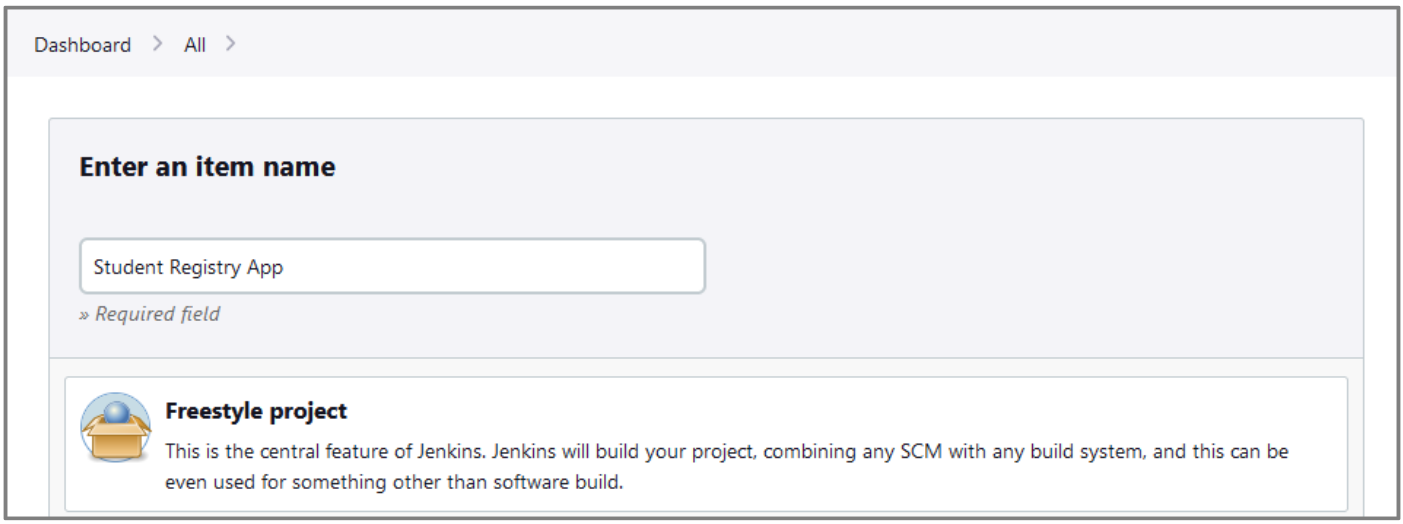
Now you should **upload the app code to `GitHub`**.

Step 3: Create a New Job

Now, let's access Jenkins. Open the Jenkins interface in a web browser. This is usually at **`http://localhost:8080`**. Let's create a new job by selecting **`[New Item]`** from the **Jenkins dashboard**.



We will enter a name for the job "**Student Registry App**", chose **[Freestyle Project]** and we should click on the **[OK]** button.

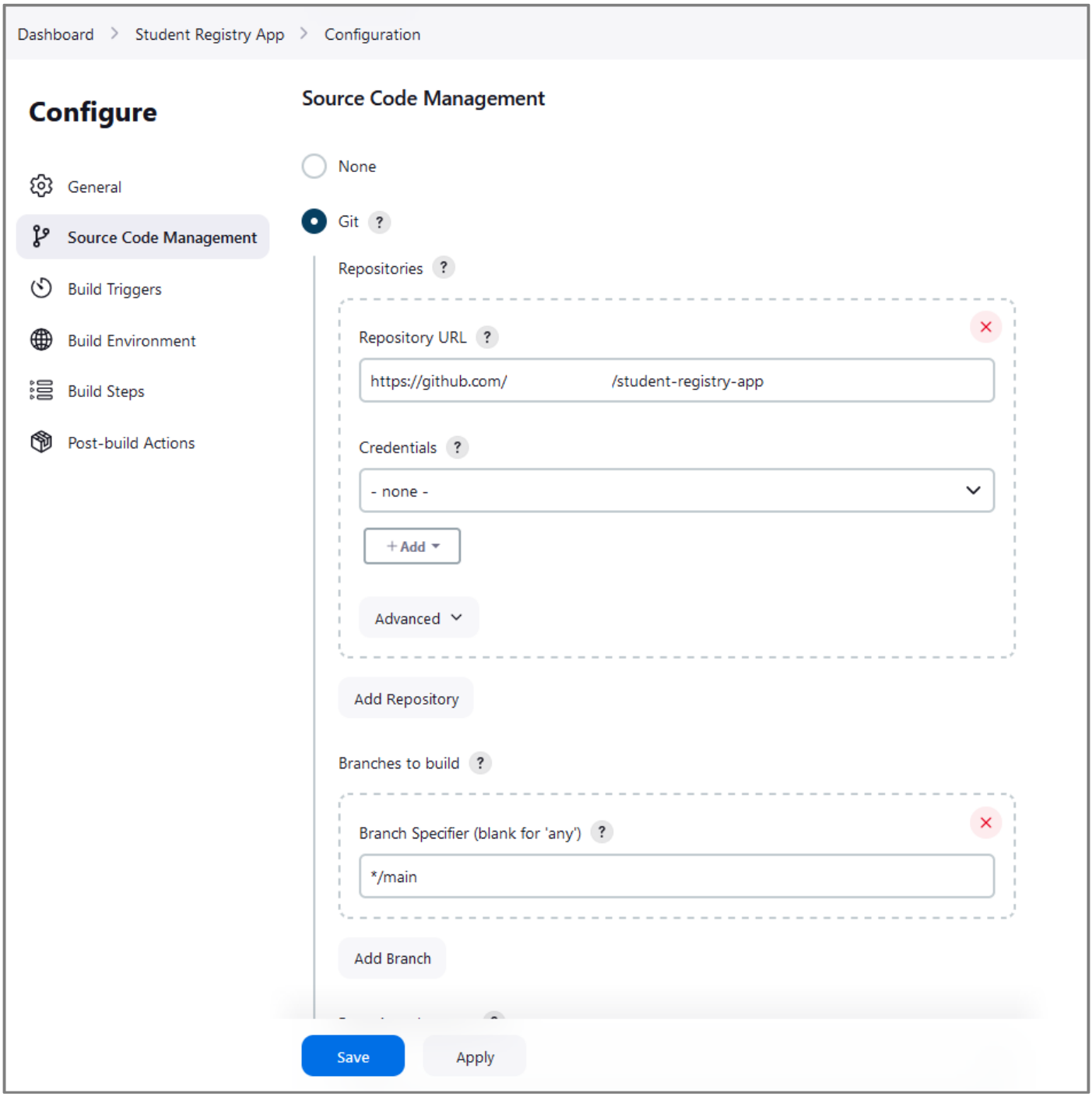


Step 4: Source Code Management

In the job configuration, go to the **Source Code Management** section.

Select **[Git]** and enter the repository URL.

After that, click on the **[Save]** button.



Step 5: Build Triggers

Setting up **build triggers** in **Jenkins** to initiate **builds on commits** to the GitHub repository involves configuring a webhook in GitHub. This **webhook** will **notify** Jenkins **each time a commit is pushed to the repository**, **triggering a build automatically**.

To do that, we have to configure webhooks in GitHub and configure the Jenkins job.

First, navigate to the GitHub repository that is used for the application. Click on the **`Settings`** tab in the GitHub repo. In the settings menu, find and click on **`Webhooks.`** Click the **`[Add webhook]`** button.

The webhook settings should be the following:

- **Payload URL**: Enter your Jenkins server's URL followed by **`/github-webhook/`**. For example, http://localhost:8080/github-webhook/.
- **Content type**: Choose **`application/json`**.

- **Secret**: Optionally, you can set a secret token for additional security (make sure to remember this as you will need it in Jenkins).
- **Which events would you like to trigger this webhook?**: Select **`Just the push event`**.
- **Active**: Ensure this checkbox is selected.

Finally, click on the **`[Add webhook]`** button to save the settings.

NOTE: For now, our Jenkins server is **not** on a **public** IP address, so we are going to use a tunneling service to expose our local Jenkins server to the Internet **temporarily**. Here's how to do it:

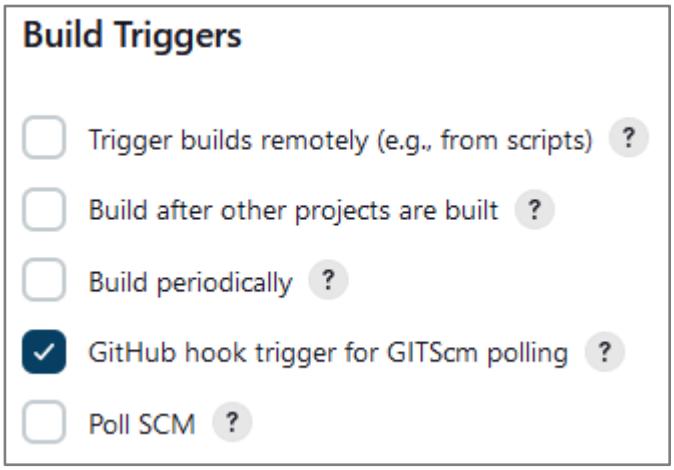
- Download and run **`ngrok`**:
  - Download **`ngrok`** and run it on your machine.
  - Use the command **`ngrok http 8080`**
  - **`ngrok`** will provide you with a public URL (e.g., **`http://abc123.ngrok.io`**).
- Update Webhook in GitHub:
  - Use the **`ngrok`** URL followed by **`/github-webhook/`** as the payload URL in the webhook settings on GitHub.
- Keep **`ngrok`** running:
  - Ensure that **`ngrok`** is running whenever you want GitHub to trigger Jenkins

With that, we have set up GitHub to notify Jenkins for each new commit.

Now, let's modify our Jenkins job to trigger on GitHub webhook notifications.

To do that, go back the Jenkins dashboard and open the job that we created for the application. Click on **`Configure`** and select **`Source Code Management`** again.

This time, in the **`Build Triggers`** section, select **`GitHub hook trigger for GitHub hook trigger for GITScm polling.`**



Step 6: Build Steps

Now it's time to add build steps to execute our commands. In our case, this will be the **`npm install`** **and** **`npm test`** **commands.**

Step 7: Configure Jenkins with Docker

Now let's modify our Jenkins's job to build and push Docker images.

Place the provided Dockerfile in the root of the directory of the repo. Then, go back to the **job configuration** and **add the following commands in order to**

```
docker build -t {your-dockerhub-username}/{app-name}:{tag} .
echo "$DOCKER_PASSWORD" | docker login --username {your-username} --password-stdin
docker push {your-username}/{app-name}:{tag}
```

The settings in the Jenkins dashboard should look like this:

NOTE: In order for Jenkins to successfully access your DockerHub account, you should create a DockerHub access token and use it for the script.

NOTE: Ensure that the Jenkins server has Docker installed and that the Docker daemon is running.

NOTE: The Jenkins user must have the necessary permissions to execute Docker commands.

Step 8: Test the CI Pipeline

After completing those steps, we are ready with the CI pipeline and it's time to test if it's working as expected.

First, make a minor change in the app code and commit and push this change to the repo, holding the application. This will trigger the Jenkins job and in the console output we can check if there are any errors.

If no errors have occurred, we can check the Docker Hub, too, to verify that the image is pushed with the correct tag.

3. CD Pipeline – "Student Registry" App

Setting up the CD Pipeline with Jenkins and Docker is pretty straightforward. However, we will need a docker-compose file for the app, we will have to configure the Jenkins job for deployment and last, we'll verify our setup.

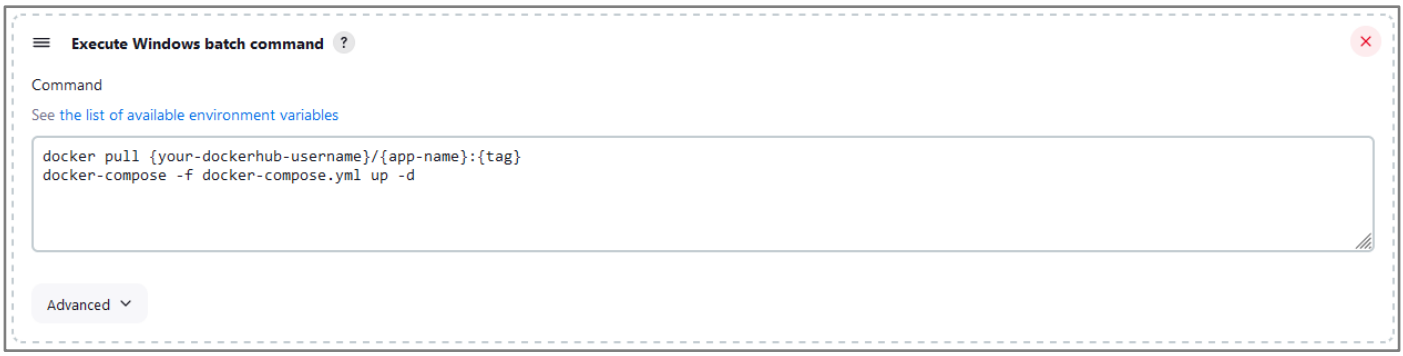
Step 1: Docker Compose Setup

Examine the **`docker-compose.yml`** file in the resources. Add to the placeholders your username, the name of the application and the tag name. They must be the same as the ones from the previous task.

Step 2: Jenkins CD Pipeline Configuration

Now we will create a new Jenkins job that is specifically for our deployment.

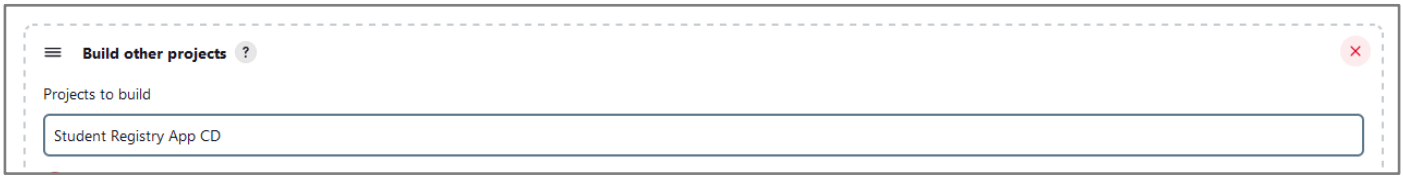
This time we will add deployment steps. We will add them the same way we added the build steps. The configuration should look something like this:



NOTE: We should add the GitHub repo again.

Step 2: Add Post-Build Actions

Now we have to set up the job to automatically deploy after a successful build. We will have to configure the CI job again – this time we will add a post-build action to trigger the CD job:



Choose the **`Trigger only if build is stable`** option as this will ensure that the CD job will only run if the CI job succeeds without any errors.

This way we linked our CI and CD jobs and whenever our CI job (build and test) completes successfully, it will automatically trigger our CD job, which takes care of deploying our application using Docker.