Exercise: CI/CD with Jenkins

Exercises for the "DevOps for Developers" module @ 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 and your machine.
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.

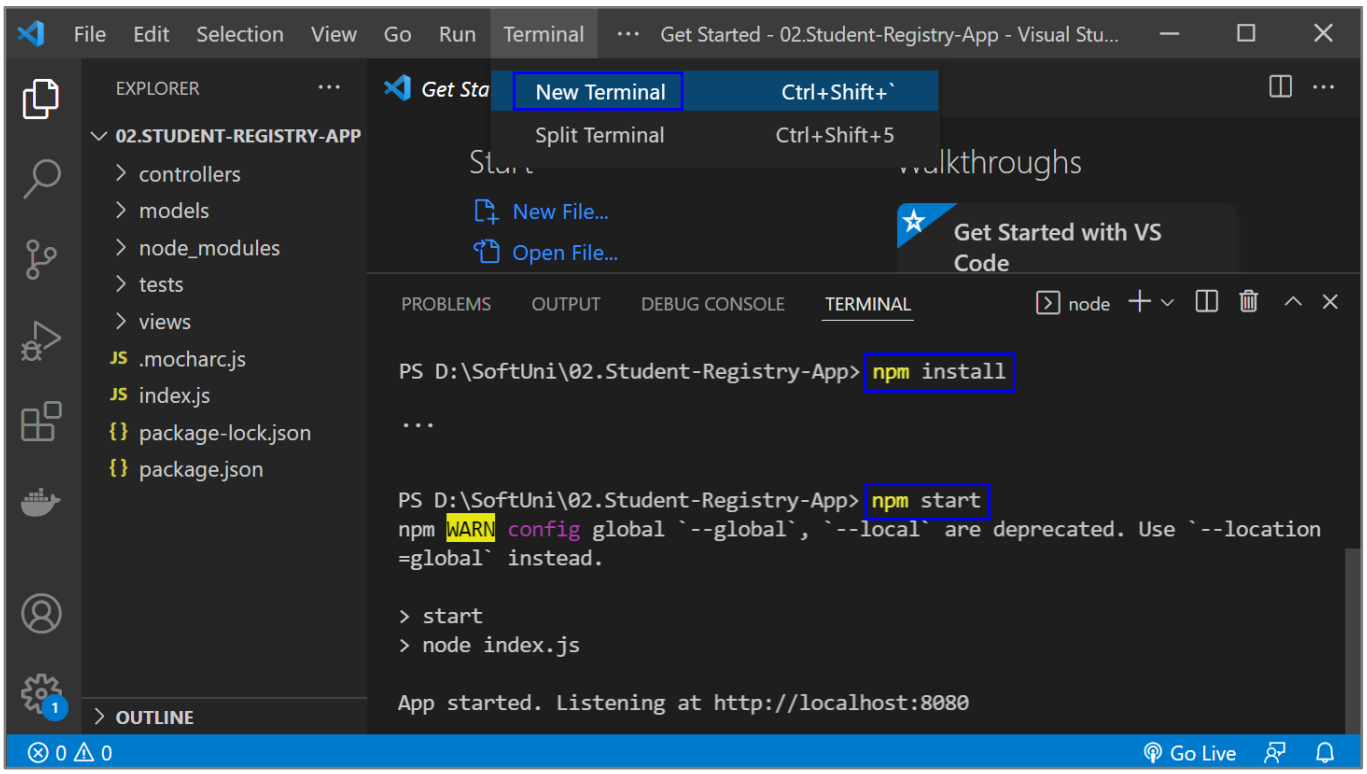
2. Configuring Jenkins with Docker

CI Pipeline – "Student Registry" App

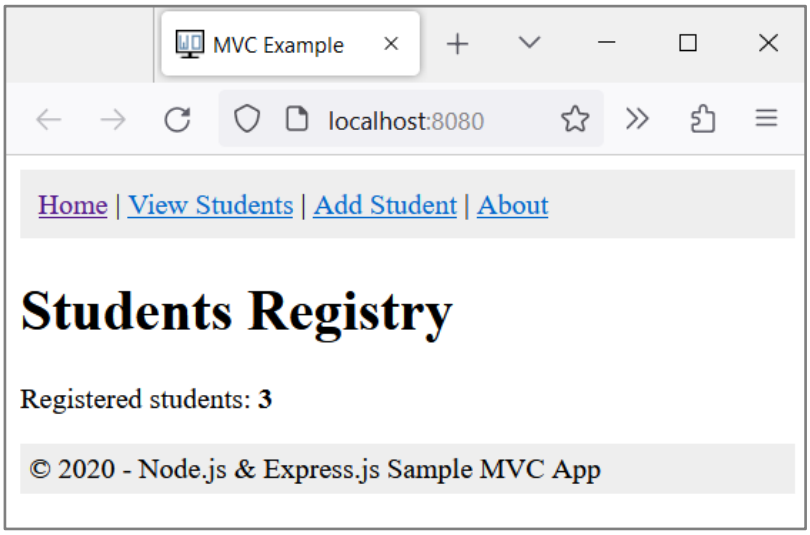
Step 1: Run the App Locally

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 versions:

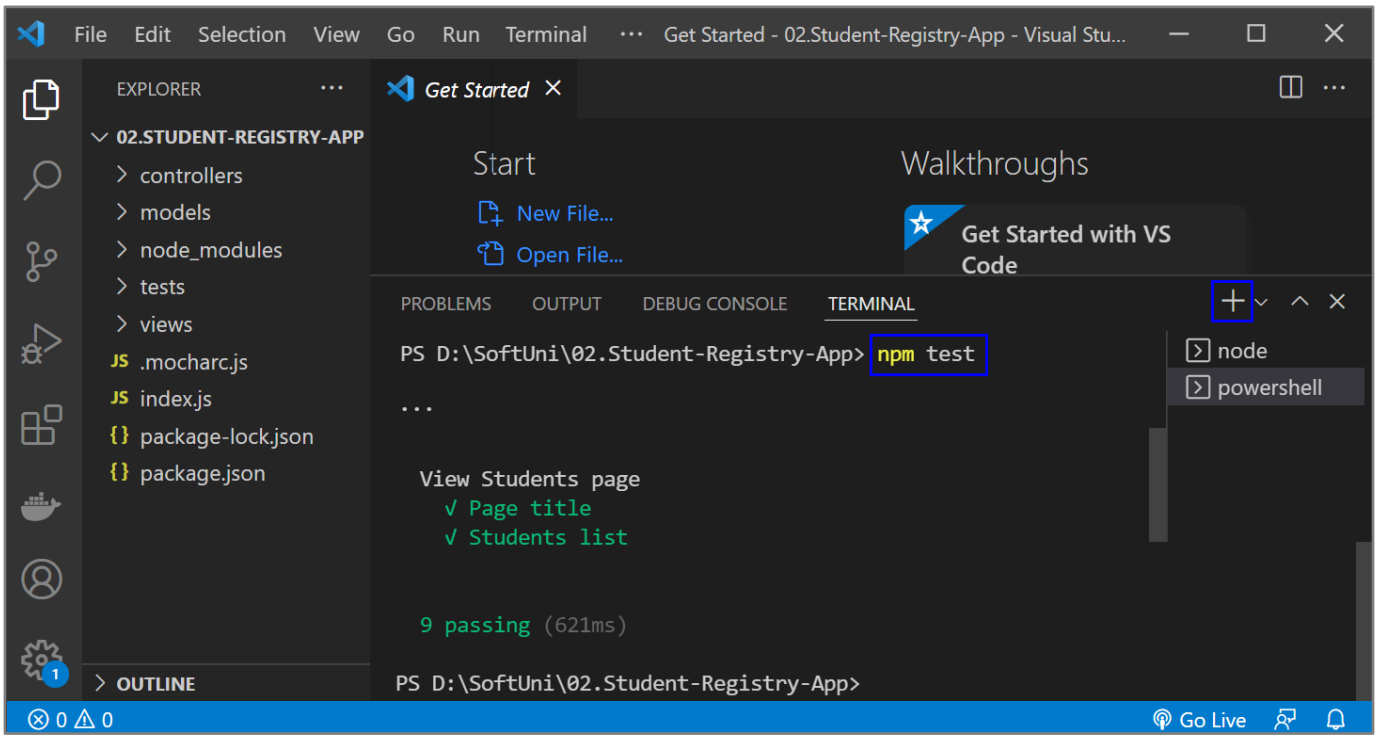
Let's first **start the app locally** in **`Visual Studio Code`**. To do this, you should **open the project**, open a **new terminal** from **`[Terminal]`** → **`[New Terminal]`** and **execute** the "**`npm install`**" and "**`npm start`**" **commands**:



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**:



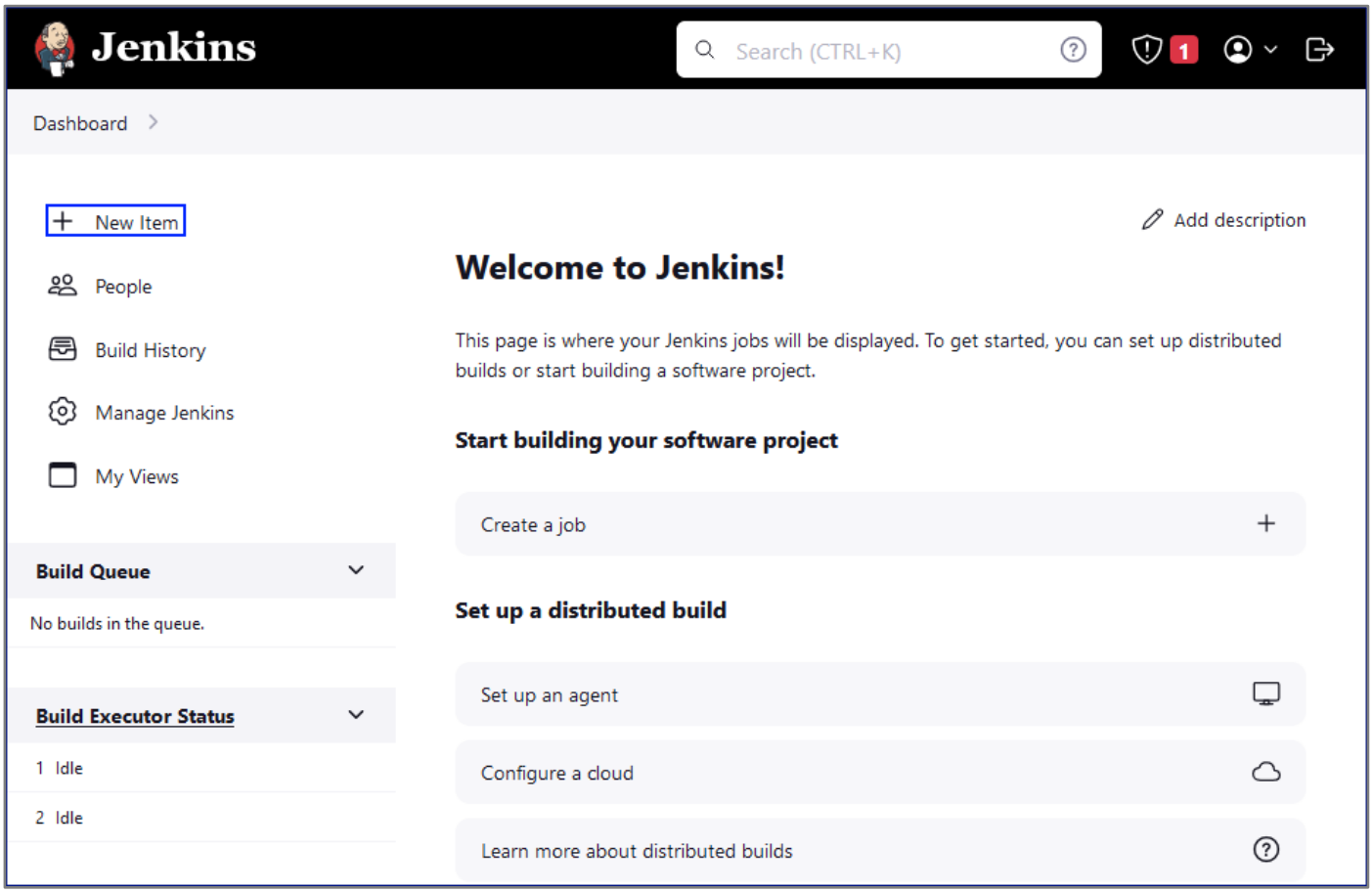
NOTE: if the **app was not started**, **tests would fail** because these are integration tests and are executed on the running app.

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`**, but it depends on the **port that you had set up during the installation.** Let's create a new job by selecting **`[New Item]`** from the **Jenkins dashboard**.



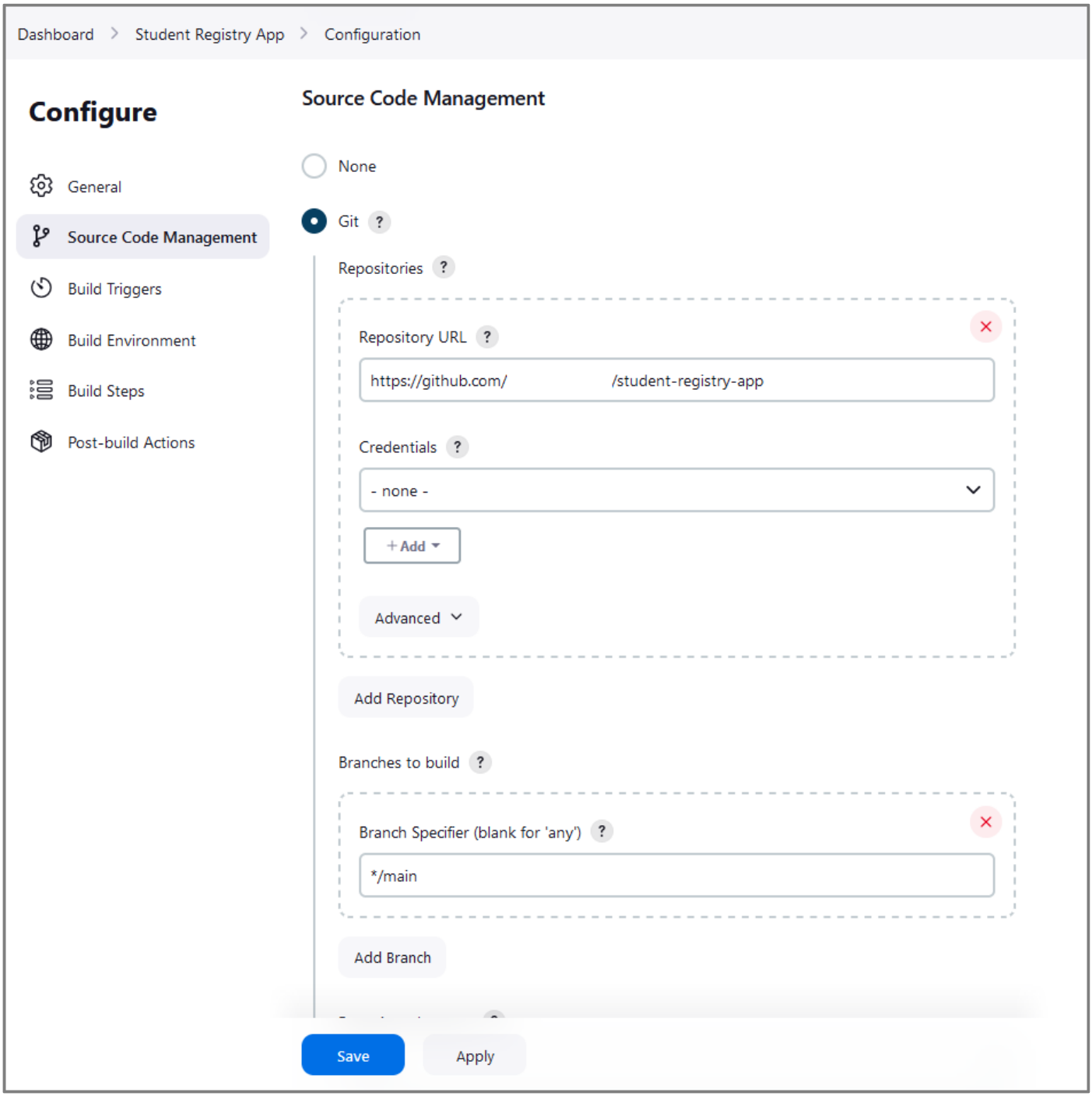
We will enter a name for the job "**Student Registry App**", chose **[Freestyle Project]** and we should click on the **[OK]** button.

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:

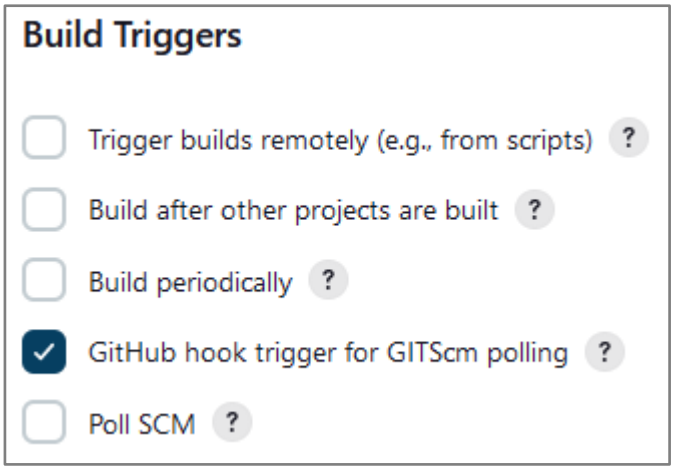
- Download and run **ngrok**:
  - Download **ngrok** and run it on your machine.
  - Use the command **ngrok `http 8080`**
  - **ngrok** will provide you with a public URL (e.g., **`http://abc123.ngrok.io`**).
- Update Webhook in GitHub:
  - Use the **ngrok** URL followed by **`/github-webhook/`** as 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.

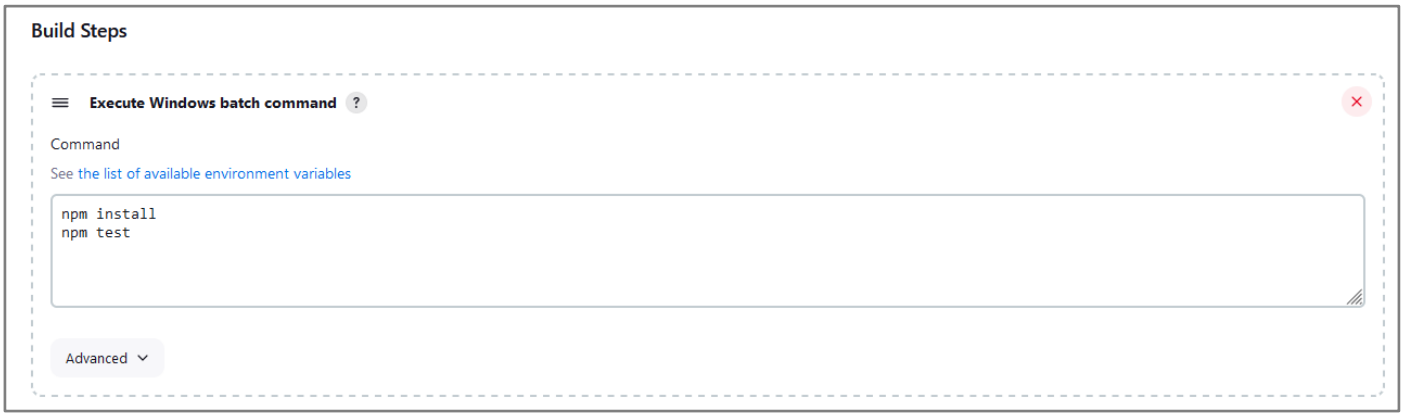
To do that, go back the Jenkins dashboard and open the job that we created for the application. Click on **`Configure`** and select **`Source Code Management`** again.

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.

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.

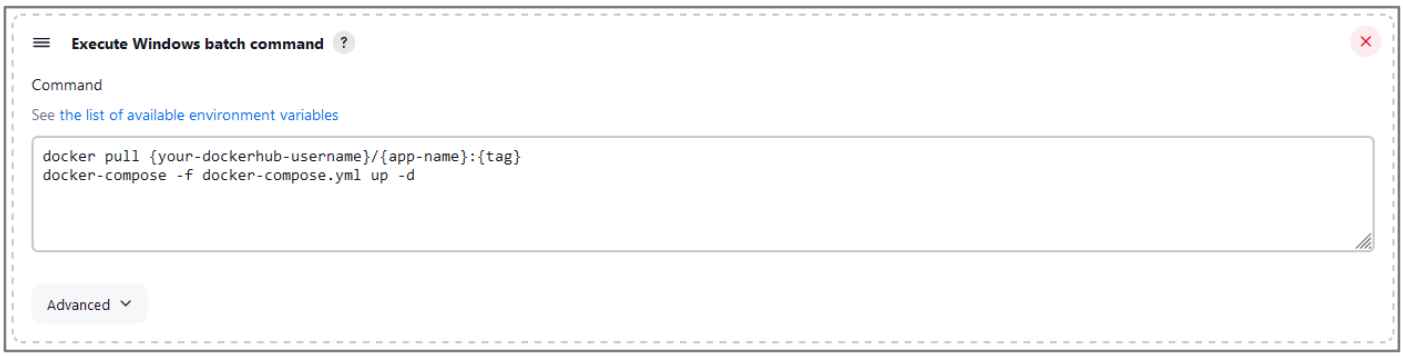
Step 1: Docker Compose Setup

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.

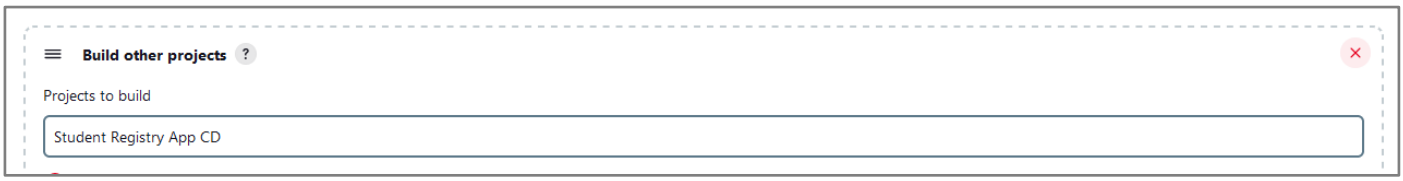
This time we will add deployment steps. We will add them the same way we added the build steps. The configuration should look something like this:



NOTE: We should add the GitHub repo again.

Step 3: 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.

3. Configuring Jenkinsfile

CI Pipeline – "Student Registry" App

Step 1: Run the App Locally

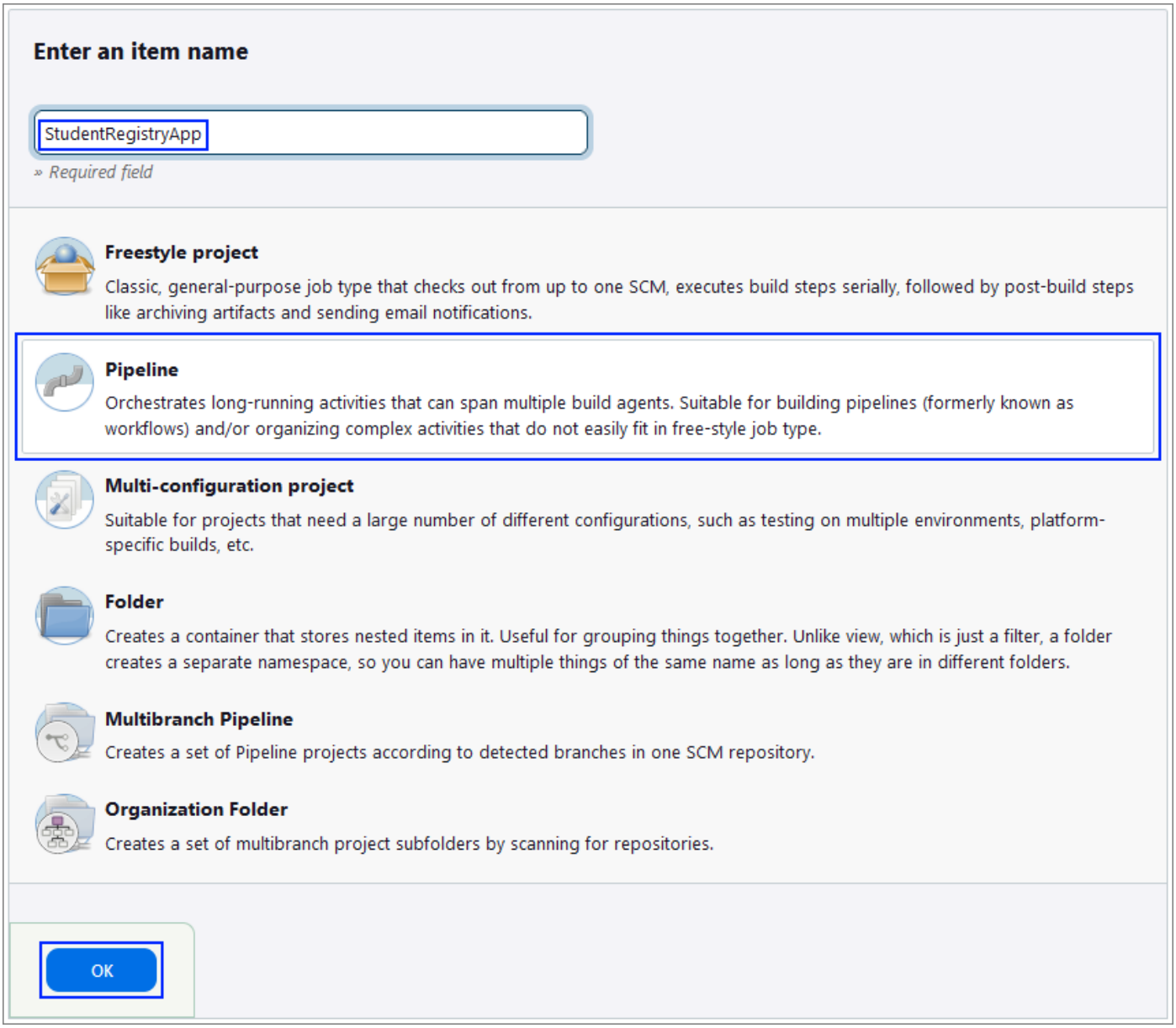
As always, you should run the app locally to ensure that everything is working correctly.

Step 2: Create a GitHub Repo

Create a new GitHub repo and **upload the app code to it**.

Step 3: Create a New Job

Now, let's create a new job by selecting **`[New Item]`** from the **Jenkins dashboard**. Choose **`Pipeline`** and give it a **meaningful** name, after that click on the **`[OK]`** button.



Step 4: Create the Jenkinsfile

Best practice for using a Jenkinsfile is to keep it **within your source control repository**.

This approach has several advantages like version control and branch specific pipelines. Placing the Jenkinsfile in the repository, means that it will be versioned alongside your application code and the versions can later be reviewed. Also, you can have different Jenkinsfile versions in different branches, which allows for testing changes to the build process in a feature branch before merging them to your main branch.

The Jenkinsfile should contain **steps** for:

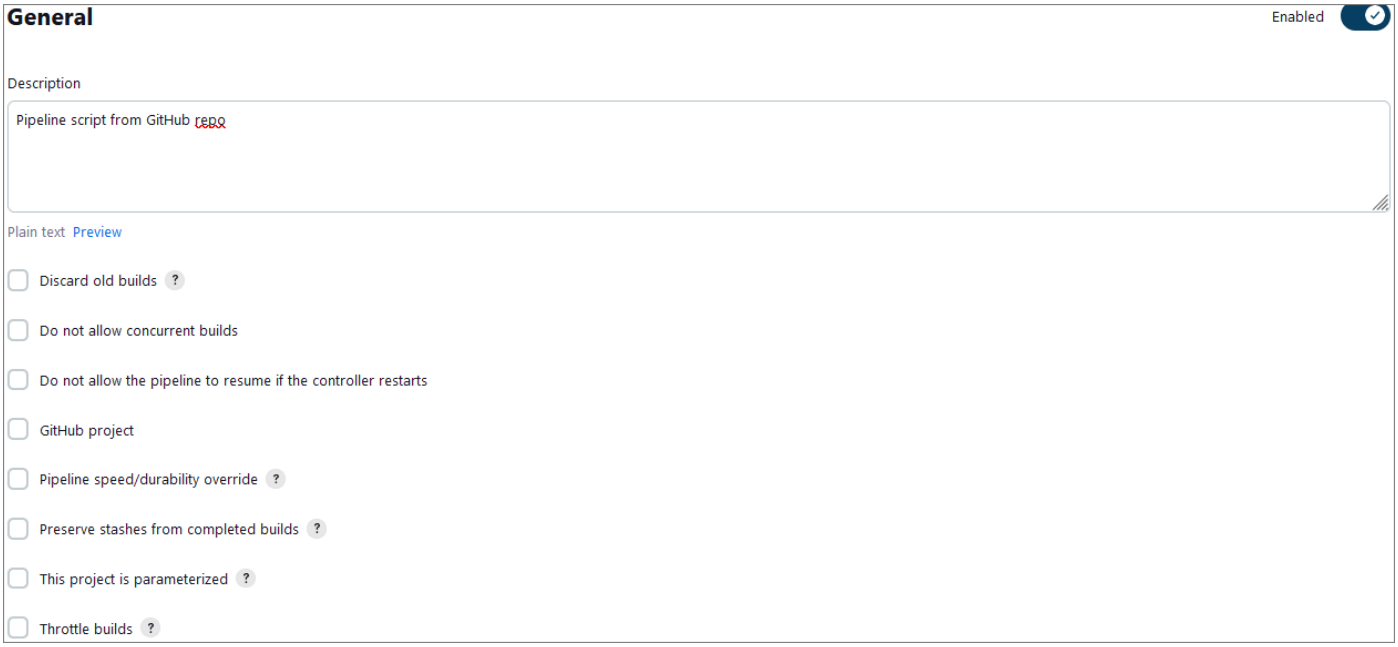
- Checking out the code
- Setting up Node.js
- Installing dependencies
- Starting the application
- Running tests

Create your file and upload it to your GitHub repository, containing the code for the application.

Step 5: Configure the Job

Now, let's **go back** to **Jenkins** to finish **configuring** your **job**.

First, in the **`General`** section give a **`Description`** for the job.



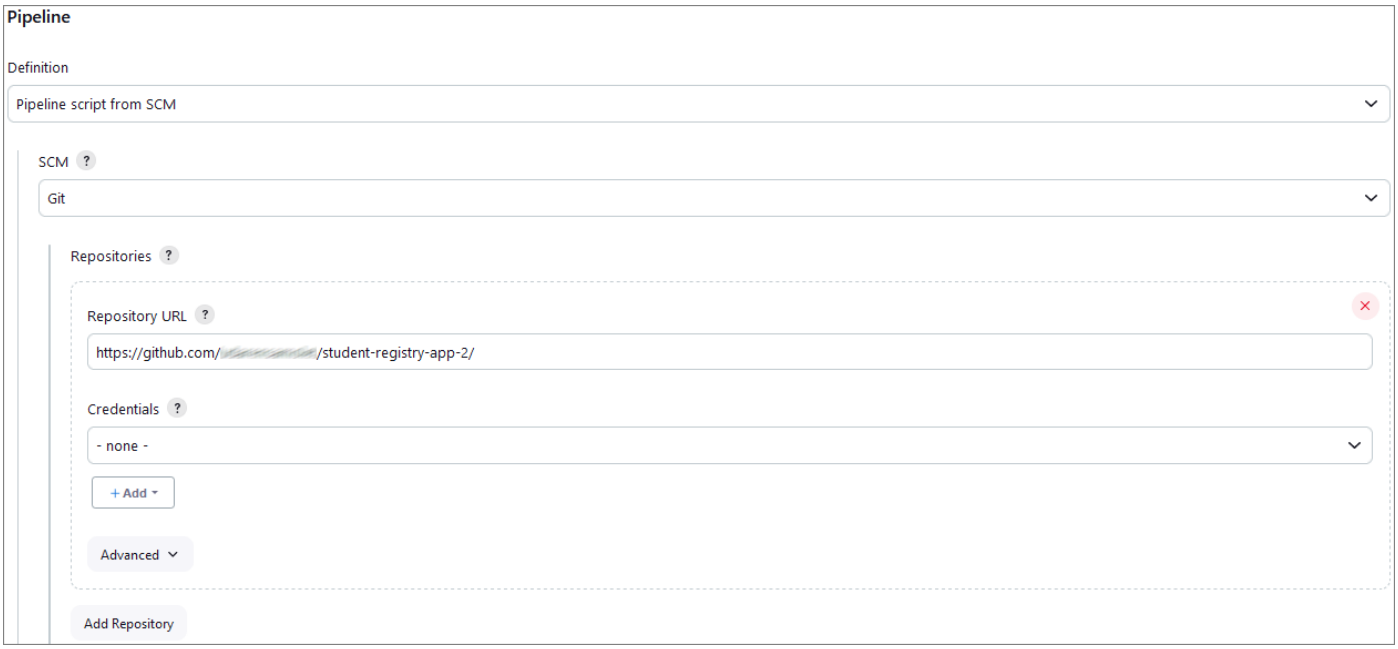
Then, scroll down to the **`General`** section In the job configuration, and from the **`Definition`** dropdown menu, select the **`Pipeline script from SCM`** option.

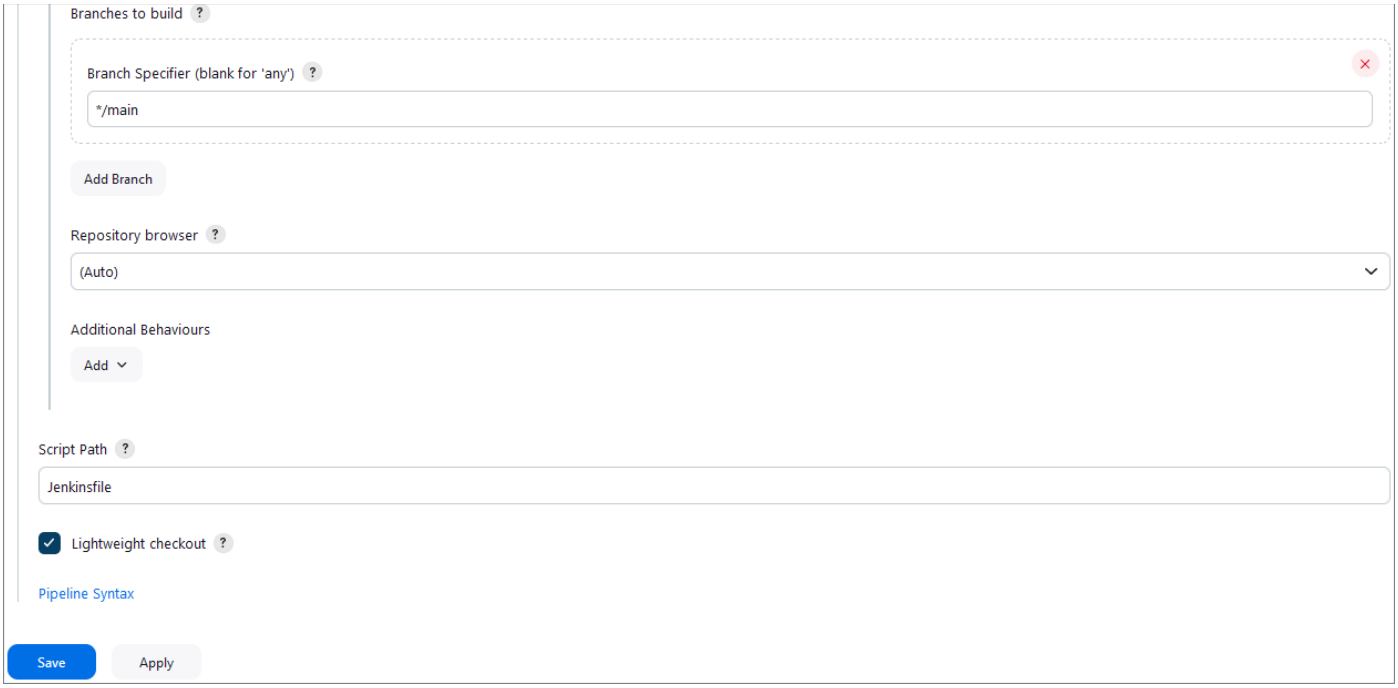
After that, select **Git** as the **SCM** and enter **your GitHub repository URL**.

Under **`Branches to build`**, enter the **branch name** that contains your **Jenkinsfile**.

Under **`Script Path`**, ensure it points to your **Jenkinsfile** (for example, type in **Jenkinsfile** if it's in the repository root).

Your configuration should look like the images below:





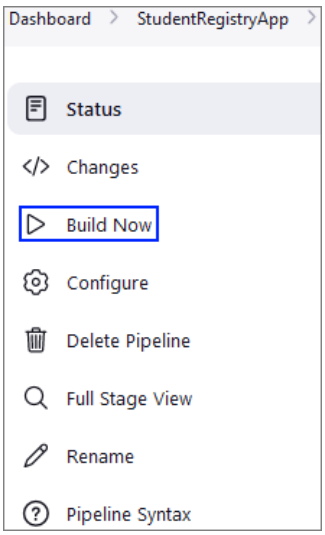
Finally, click on the **[Save]** button.

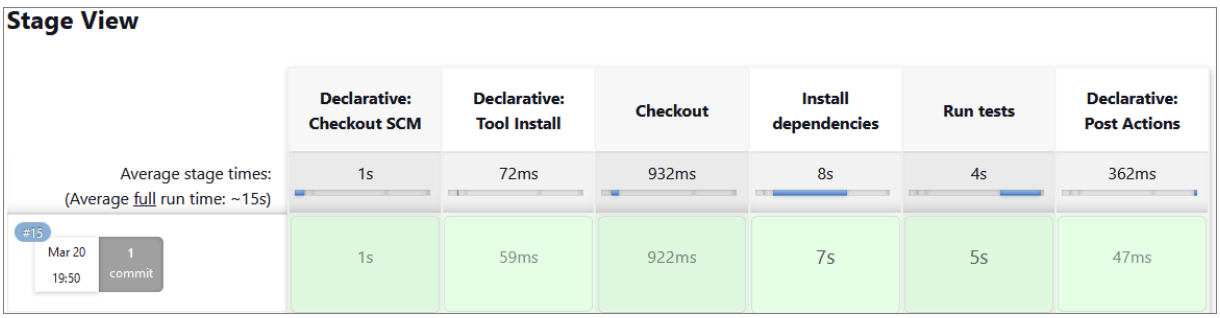
Step 6: Test the CI Pipeline

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

First, click on the **`Build Now`** option to start a new build manually.

You can monitor the build progress by clicking on the build number and then **`Console Output`**.





You can try and set up yourself Webhooks for automatic triggers just like we did in the previous task, so that **each new commit triggers Jenkins** to **build automatically** the **pipeline**.

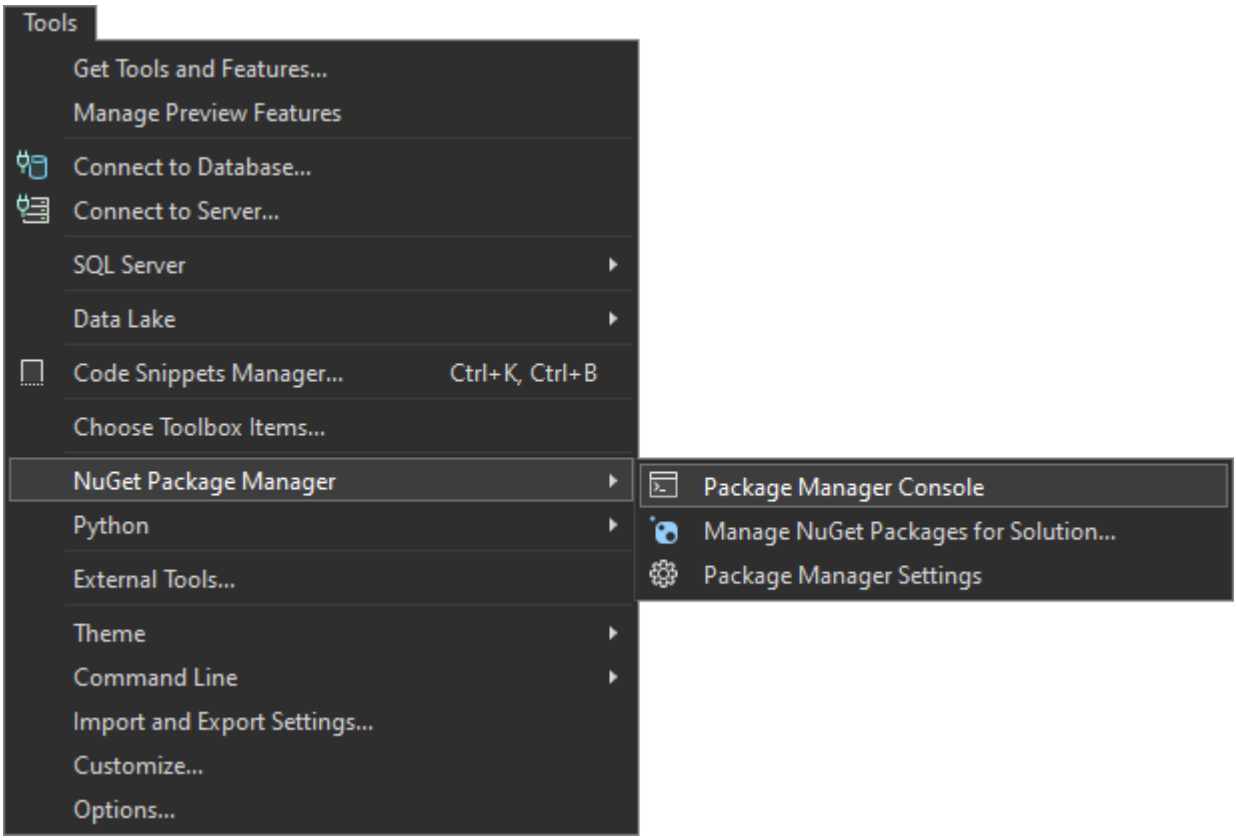
4. "HouseRentingSystem" App – ASP.NET Core MVC app

Step 1: Run the App Locally

We have the "**`HouseRentingSystem`**" **ASP.NET Core MVC app** in the **resources which has some unit and integration tests already**. Your task is to **create a CI workflow** with **Jenkins** to **start and test the app.**

It's a good practice to first **start the app locally** in **`Visual Studio`**, in order to be sure everything works properly and as expected.

Open **Visual Studio** and from there navigate to the **Tools** menu. Select **NuGet Package Manager** and select **Package Manager Console**:



Let's first build the application by using the **`dotnet build`** command:

```
PM> dotnet build
MSBuild version 17.8.3+195e7f5a3 for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
  HouseRentingSystem.Services -> C:\Users\[redacted]\HouseRentingSystem\HouseRentingSystem.Services\bin\Debug\net6.0\HouseRentingSystem.Services.dll
  HouseRentingSystem.Web -> C:\Users\[redacted]\Desktop\HouseRentingSystem\HouseRentingSystem.Web\bin\Debug\net6.0\HouseRentingSystem.Web.dll
  HouseRentingSystem.Tests -> C:\Users\[redacted]\Desktop\HouseRentingSystem\HouseRentingSystem.Tests\bin\Debug\net6.0\HouseRentingSystem.Tests.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.96
```

After you have **ensured** that the **build** was **successful**, you can **run** the **tests**, too, by using the **`dotnet test`** command:

```
PM> dotnet test
  Determining projects to restore...
  All projects are up-to-date for restore.
Test run for C:\Users\[redacted]\Desktop\HouseRentingSystem.Tests
Microsoft (R) Test Execution Command Line Tool Version 17.8.0 (x64)
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:    29, Skipped:     0, Total:    29
```

NOTE: Visual Studio has built-in test runners that allow you to run your tests directly from the IDE. This is the simplest way to execute tests if you're already working within Visual Studio. However, it's **better** to get used **using** the **console**.

After we have ensured that the **tests run successfully**, we can proceed with the next step.

Step 2: Create a GitHub Repo

Now you should **upload the app code to `GitHub`**. Try using the **CLI** and the **commands** from the previous task to add the code to the repo and commit it.

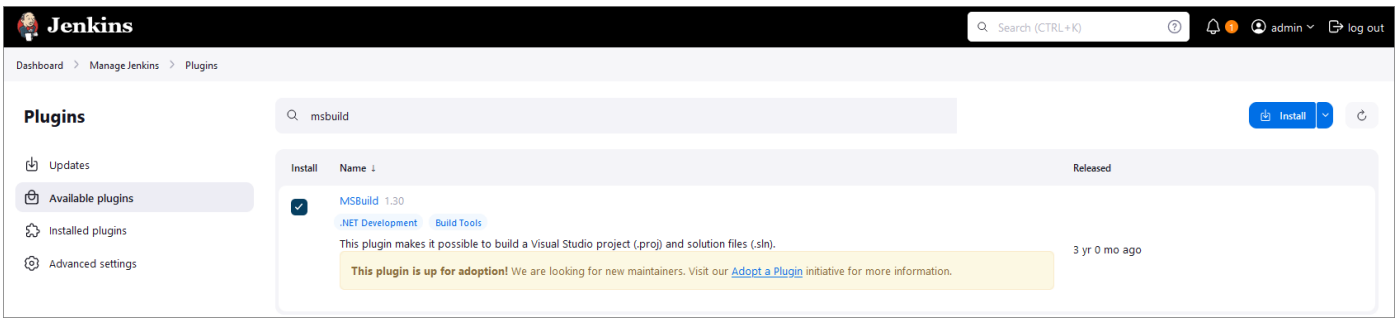
Step 3: Configure Tools in Jenkins

To run an **ASP.NET Core MVC app** in Jenkins, you need **two** plugins: **Git** and **MSBuild**.

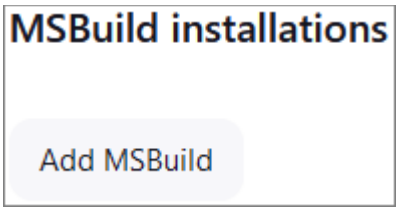
Usually, **Git** is being **installed** when you are **configuring** your **Jenkins** installation and we **already** used it in the previous task.

Let's focus on configuring the **MSBuild** plugin.

Go to **Manage Jenkins** menu and select **Plugins**. From the menu on the left, select **Available plugins** and type **MSBuild** in the search field. Select the plugin and click on the **`[Install]`** button:



Once you have the needed plugin installed, go back to **Manage Jenkins** and select **Tools.** Scroll down to find the **MSBuild installations** section and click on **`[Add MSBuild]`** button:

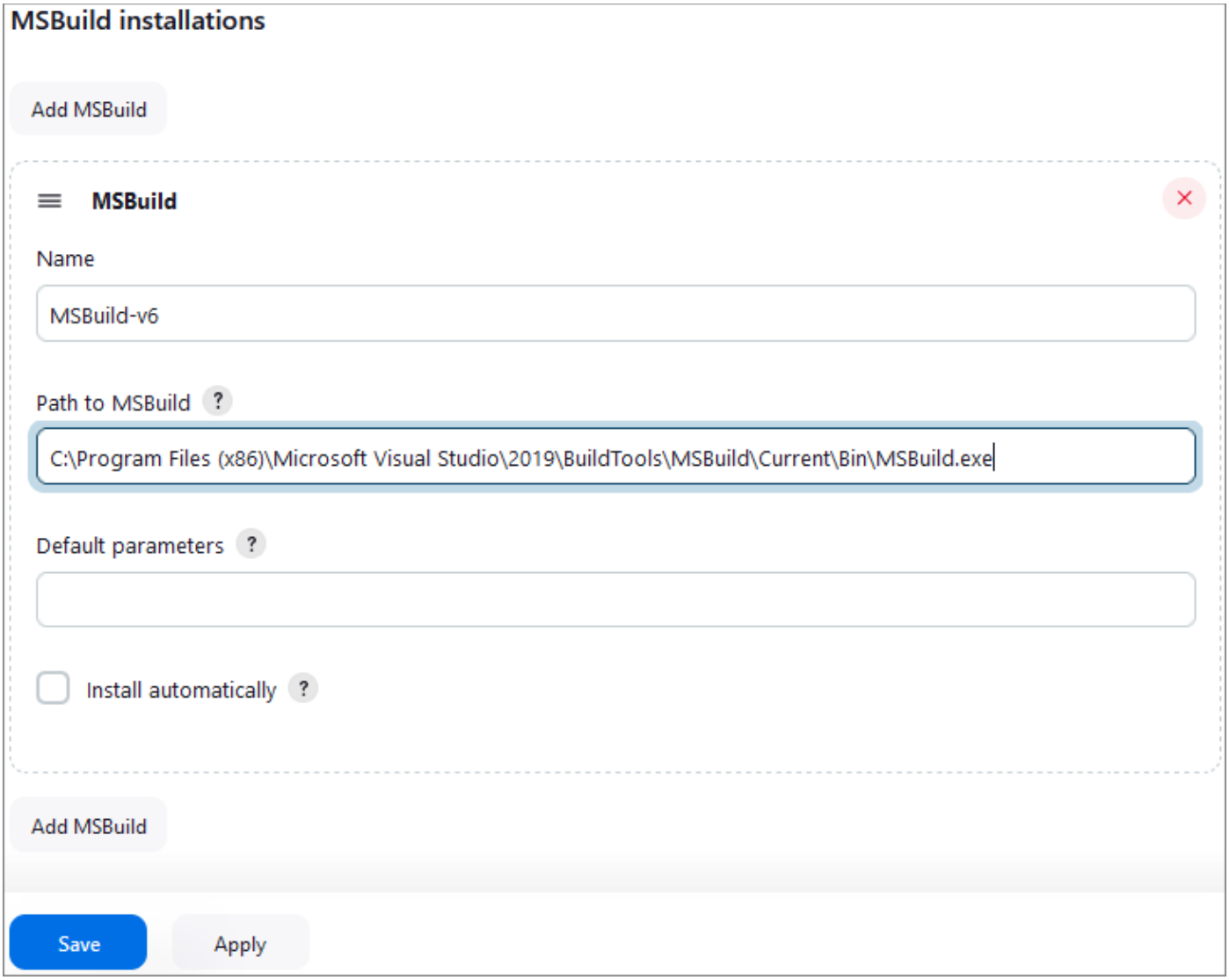


Give a **meaningful name** to your MSBuild and provide the path to your MSBuild.exe file.

NOTE: MSBuild.exe is the **command-line tool** for **Microsoft Build Engine**, which is used to **build applications**. This engine uses **XML-based** project **files** to **compile** and **link** the **code**, manage **project dependencies**, and **execute**

other **build tasks**. It's a vital **component** of the **.NET framework development process** and is also used in building software projects in other languages. **MSBuild** comes **included** with several **Microsoft** products, including **Visual Studio**. Usually, the path to your MSBuild.exe file is something like **`C:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools\MSBuild\Current\Bin\MSBuild.exe`**.

The configuration should look like the image below:



Finally, click on the **[Save]** button.

Step 4: Create and Configure a New Job

Open the **Jenkins interface** in a **web browser**.

Create a new job by selecting **`[New Item]`** from the **Jenkins dashboard**. Choose **`Pipeline`** and give it a **meaningful** name, after that click on the **`[OK]`** button.

Next, on the **General** section, type in a proper description.

Select **GitHub project** as the **Source Code Management** option and input the **URL** of your **repository**.

If you want, you can play around a little bit and add a **build trigger**, as you already know how to do that.

Go to the **Pipeline** section and select **Pipeline script from SCM** as you already know this is the **best practice** for where to keep the **Jenkinsfiles**. Configure the path to the repository and to the Jenkinsfile. The steps are the same as in the previous task.

Step 5: Create the Jenkinsfile

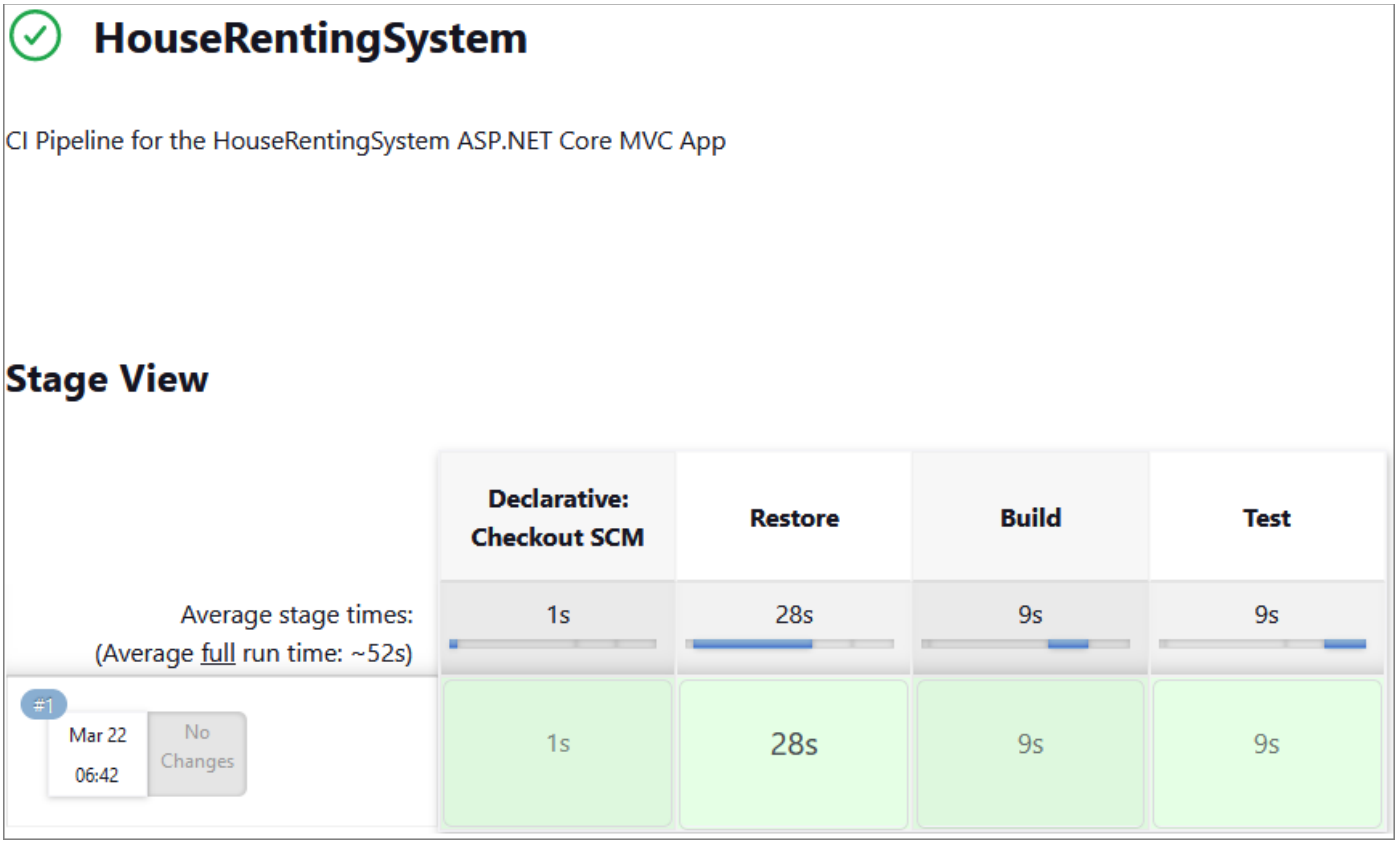
The Jenkinsfile should contain **steps** for:

- **Restore**
  - Restore the NuGet packages needed for the project to ensure all dependencies are downloaded and available during the build process.
- **Build**
  - Build the project to check for compilation errors.
- **Test**
  - Execute the tests to ensure that they're running properly

Create your **file** and **upload** it to your GitHub **repository**, containing the code for the application.

Step 6: 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, click on the **`Build Now`** option to start a new build manually (in case you haven't configured the build triggers).

You can monitor the build progress by clicking on the build number and then **`Console Output`**.

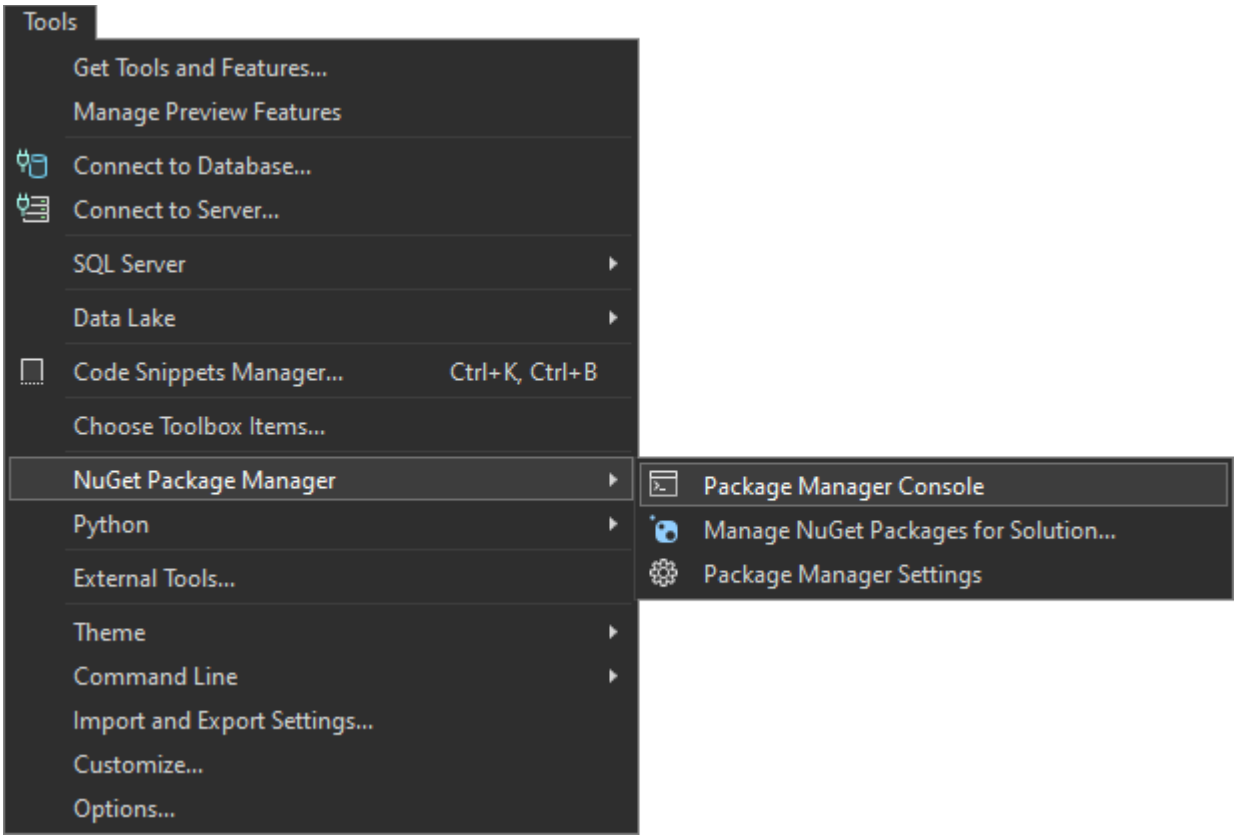
5. Selenium IDE

Step 1: Run the App Locally

We have the "**`SeleniumIde`**" **solution** in the **resources which has one test projects already**. Your task is to **create a CI workflow** with **GitHub Actions** to **run the tests automatically.**

It's a good practice to **build the solution locally** in **Visual Studio**, in order to be sure everything works properly and as expected.

Open **Visual Studio** and from there navigate to the **Tools** menu. Select **NuGet Package Manager** and select **Package Manager Console**:



Let's first build the application by using the following command:

```
dotnet build
```

After you have **ensured** that the **build** was **successful**, you can **run** the **tests**, too, by using the command below or just by clicking on the **`[Run All Tests in View]`** button in the **`Text Explorer`**.

```
dotnet test
```

After we have ensured that the **tests run successfully**, we can proceed with the next step.

⚠	You have to be sure that the **Chrome** and **ChromeDriver** installed on your local **machine** are one and the **same major version**. For example, ChromeDriver v.125 won't work with Chrome v. 127!

Step 2: Create a GitHub Repo

Now you should **upload the solution to `GitHub`**.

It's a **good** practice to start using the **console** and not the interface of GitHub, in case you haven't started doing so yet.

If you **don't have Git** already **installed** on your machine, follow the **provided installation instructions** from the **resources**.

Try using the **following commands** in order to initialize a repository in your project directory, add the code to the repo, commit and push:

```
git init
git add .
git commit -m "Initial commit"
git remote add origin https://github.com/{name-of-your-repository}
git push -u origin main
```

After running the commands, **check you GitHub repo** – the application code should be visible.

Step 3: Add Changes to Test Files

Before creating the workflow file, we have to make some adjustments in the **.cs** files. This is needed due to the fact that the default GitHub runner does not have Chrome installed. We will take care of this in the workflow, but we also need the prepare the tests to run Chrome in a headless mode within the CI environment.

In order to do that, go to the **`SetUp()`** method of the project and modify it so it looks like below:

```
[SetUp]
0 references
public void SetUp()
{
    ChromeOptions options = new ChromeOptions();
    options.AddArguments("headless");
    options.AddArguments("no-sandbox");
    options.AddArguments("disable-dev-shm-usage");
    options.AddArguments("disable-gpu");
    options.AddArguments("window-size=1920x1080");

    driver = new ChromeDriver(options);
    js = (IJavaScriptExecutor)driver;
    vars = new Dictionary<string, object>();
}
```

Don't forget to **commit** and **push** the changes from the file.

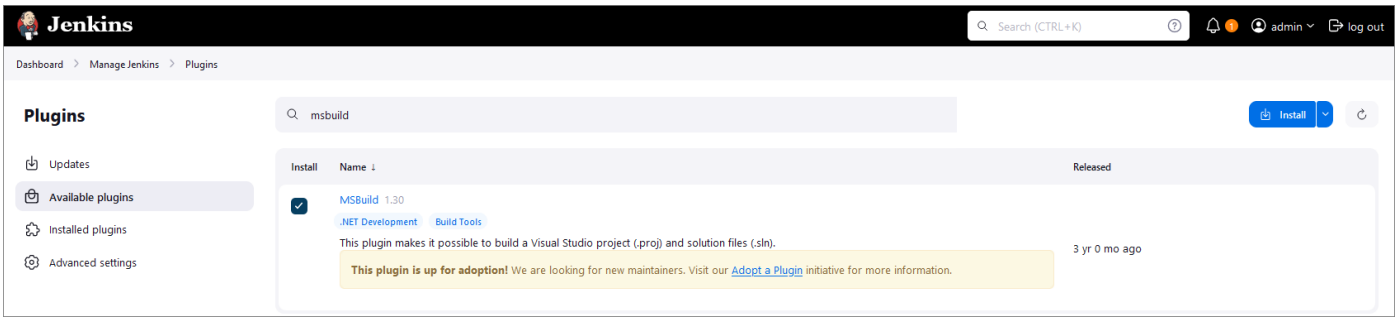
Step 4: Configure Tools in Jenkins

To run an **ASP.NET Core MVC app** in Jenkins, you need **two** plugins: **Git** and **MSBuild**.

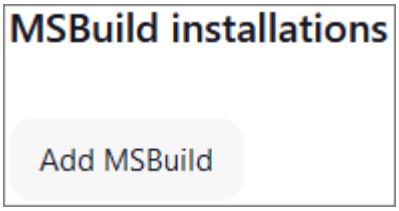
Usually, **Git** is being **installed** when you are **configuring** your **Jenkins** installation and we **already** used it in the previous task.

Let's focus on configuring the **MSBuild** plugin.

Go to **Manage Jenkins** menu and select **Plugins**. From the menu on the left, select **Available plugins** and type **MSBuild** in the search field. Select the plugin and click on the **`[Install]`** button:



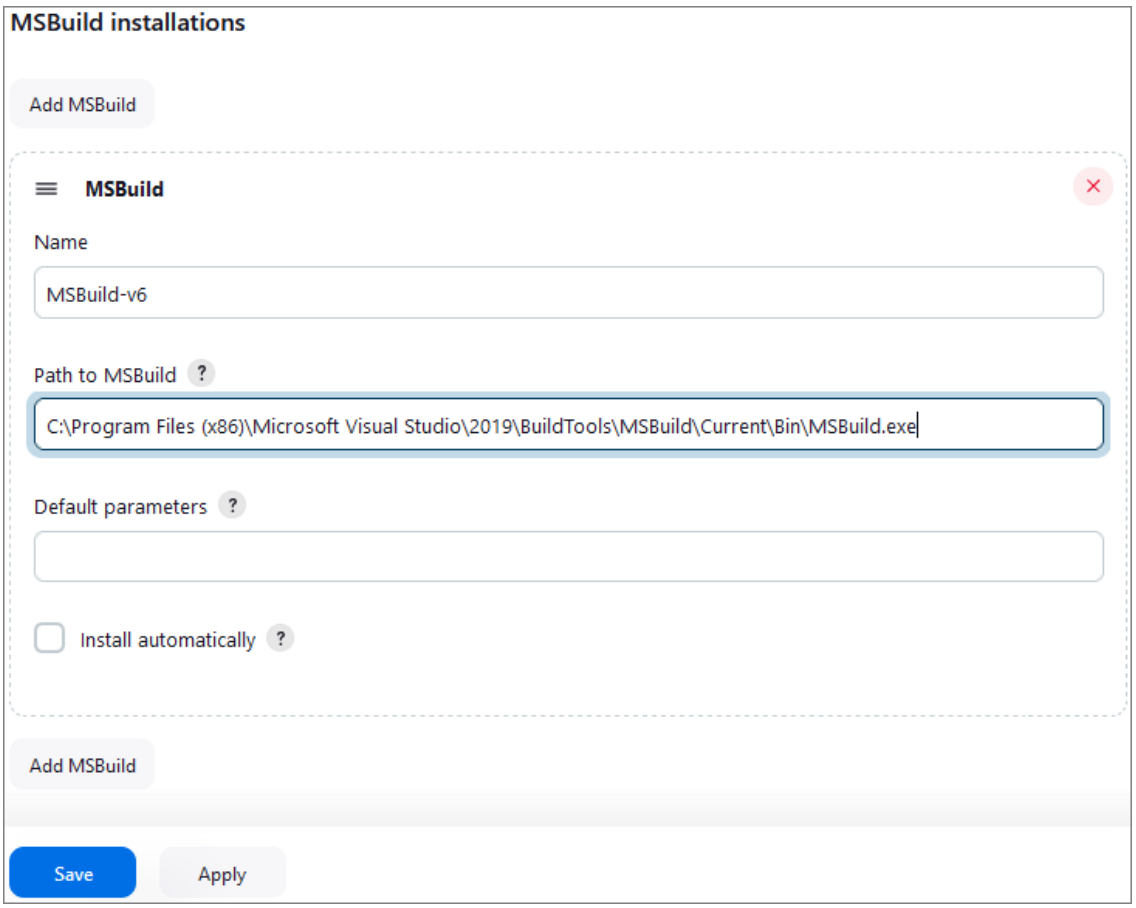
Once you have the needed plugin installed, go back to **Manage Jenkins** and select **Tools.** Scroll down to find the **MSBuild installations** section and click on **`[Add MSBuild]`** button:



Give a **meaningful name** to your MSBuild and provide the path to your MSBuild.exe file.

NOTE: **MSBuild.exe** is the **command-line tool** for **Microsoft Build Engine**, which is used to **build applications**. This engine uses **XML-based** project **files** to **compile** and **link** the **code**, manage **project dependencies**, and **execute** other **build tasks**. It's a vital **component** of the **.NET framework development process** and is also used in building software projects in other languages. **MSBuild** comes **included** with several **Microsoft** products, including **Visual Studio**. Usually, the path to your MSBuild.exe file is something like **`C:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools\MSBuild\Current\Bin\MSBuild.exe`**.

The configuration should look like the image below:

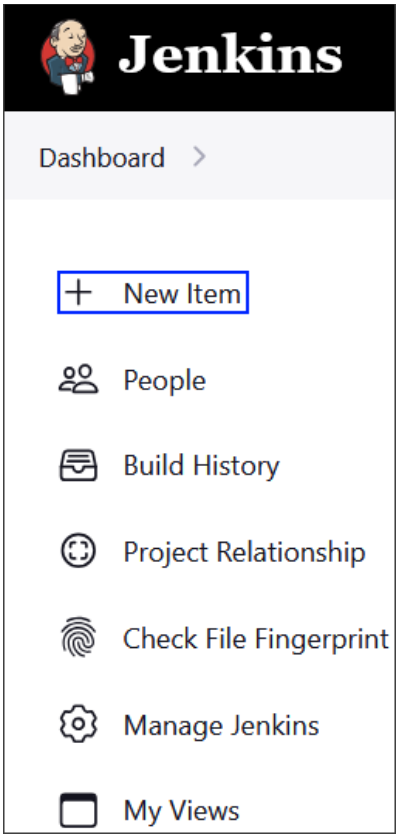


Finally, click on the **[Save]** button.

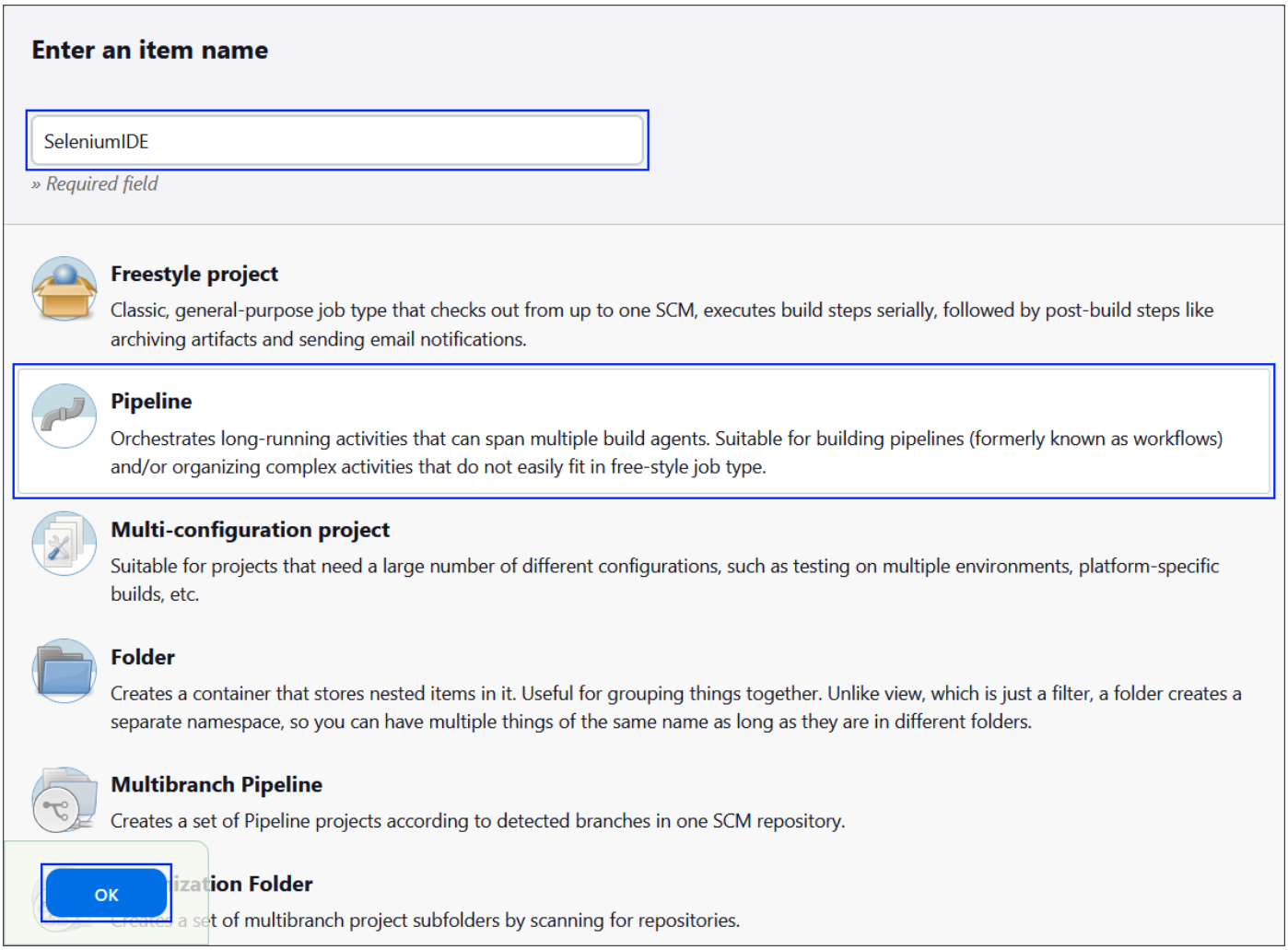
Step 5: Create a New Job

Now, let's access Jenkins. Open the Jenkins interface in a web browser. This is usually at **`http://localhost:8080`**, but it depends on the **port that you had set up during the installation.**

Let's create a new job by selecting **`[New Item]`** from the **Jenkins dashboard**.



Choose **Pipeline** and give it a **meaningful** name, after that click on the **[OK]** button.



Step 6: Create the Jenkinsfile

Best practice for using a **Jenkinsfile** is to keep it **within your source control repository**.

This approach has several advantages like version control and branch specific pipelines. Placing the **Jenkinsfile** in the repository, means that it will be versioned alongside your application code and the versions can later be reviewed. Also, you can have different **Jenkinsfile versions** in **different** branches, which allows for testing changes to the build process in a feature branch before merging them to the main branch.

The Jenkinsfile should contain **steps** for:

- Checkout the code
- Set up .NET Core
- Uninstall current chrome
- Install specific version of Chrome
- Download and install ChromeDriver
- Restore dependencies
- Build
- Run tests

Pipeline Configuration

Let's start with the pipeline configuration.

We have to specify that the pipeline can run on any available Jenskins agent and declare the environmental variables to be used within it:

- **CHROME_VERSION**: The version of **Google Chrome** to be installed
- **CHROMEDRIVER_VERSION**: The version of **ChromeDriver** to be installed
- **CHROME_INSTALL_PATH**: The installation path for **Google Chrome**
- **CHROMEDRIVER_PATH**: The installation path for **ChromeDriver**

```
pipeline {
    agent any

    environment {
        CHROME_VERSION = '127.0.6533.73'
        CHROMEDRIVER_VERSION = '127.0.6533.72'
        CHROME_INSTALL_PATH = 'C:\\Program Files\\Google\\Chrome\\Application'
        CHROMEDRIVER_PATH = '"C:\\Program Files\\Google\\Chrome\\Application\\chromedriver.exe"'
    }
```

Checkout Code Stage

Next step is to define a stage for checking out the source code.

```
    stages {
        stage('Checkout code') {
            steps {
                // Checkout code from GitHub and specify the branch
                git branch: 'main', url: 'https://github.com/[redacted]/SeleniumIDE.git'
            }
        }
```

Set up .NET Core Stage

After that, we have to define the stage for setting up .NET Code SDK.

```
stage('Set up .NET Core') {
    steps {
        bat '''
        echo Installing .NET SDK 6.0
        choco install dotnet-sdk -y --version=6.0.100
        '''
    }
}
```

* Uninstall Current Chrome Stage

This step is optional, in case you are not sure how to install the proper Google Chrome version.

```
stage('Uninstall Current Chrome') {
    steps {
        bat '''
        echo Uninstalling current Google Chrome
        choco uninstall googlechrome -y
        '''
    }
}
```

* Uninstall Current Chrome Stage

This step is optional and is used in combination with the previous step.

```
stage('Install Specific Version of Chrome') {
    steps {
        bat '''
        echo Installing Google Chrome version %CHROME_VERSION%
        choco install googlechrome --version=%CHROME_VERSION% -y --allow-downgrade --ignore-checksums
        '''
    }
}
```

* Download and Install ChromeDriver Stage

This step is optional and is used in combination with the previous two previous steps.

Use the code below, as this is a pretty long command:

```
stage('Download and Install ChromeDriver') {
            steps {
                bat '''
                echo Downloading ChromeDriver version %CHROMEDRIVER_VERSION%
                powershell -command "Invoke-WebRequest -Uri
https://chromedriver.storage.googleapis.com/%CHROMEDRIVER_VERSION%/chromedriver_win3
2.zip -OutFile chromedriver.zip -UseBasicParsing"
                powershell -command "Expand-Archive -Path chromedriver.zip -
DestinationPath ."
                powershell -command "Move-Item -Path .\\chromedriver.exe -
Destination '%CHROME_INSTALL_PATH%\\chromedriver.exe' -Force"
                '''
            }
        }
```

Restore Dependencies Stage

Now we have to define a stage for restoring the project's dependencies.

```
        stage('Restore dependencies') {
            steps {
                // Restore dependencies using the solution file
                bat 'dotnet restore SeleniumIde.sln'
            }
        }
```

Build Stage

Now let's define a stage for building the project.

```
        stage('Build') {
            steps {
                // Build the project using the solution file
                bat 'dotnet build SeleniumIde.sln --configuration Release'
            }
        }
```

Run Tests Stage

Finally, after we have set everything needed, we can define a stage for running the tests.

```
        stage('Run tests') {
            steps {
                // Run tests using the solution file
                bat 'dotnet test SeleniumIde.sln --logger "trx;LogFileName=TestResults.trx"'
            }
        }
    }
```

* Post Stage

This is an optional stage.

Now, let's define a post-build actions that are always executed. In our case, we will archive the test results and publish them to Jenkins.

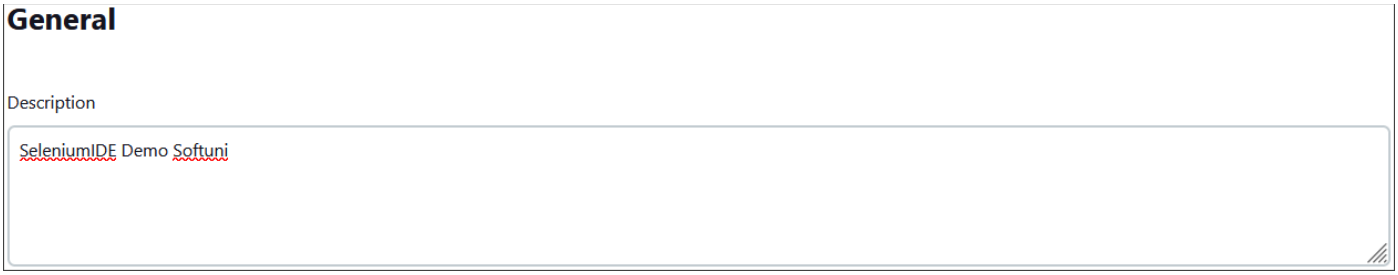
```
    post {
        always {
            archiveArtifacts artifacts: '**/TestResults/*.trx', allowEmptyArchive: true
            junit '**/TestResults/*.trx'
        }
    }
}
```

Create your file and upload it to your GitHub repository, containing the code for the application.

Step 7: Configure the Job

Now, let's **go back** to **Jenkins** to finish **configuring** your **job**.

First, in the **General** section give a **Description** for the job.



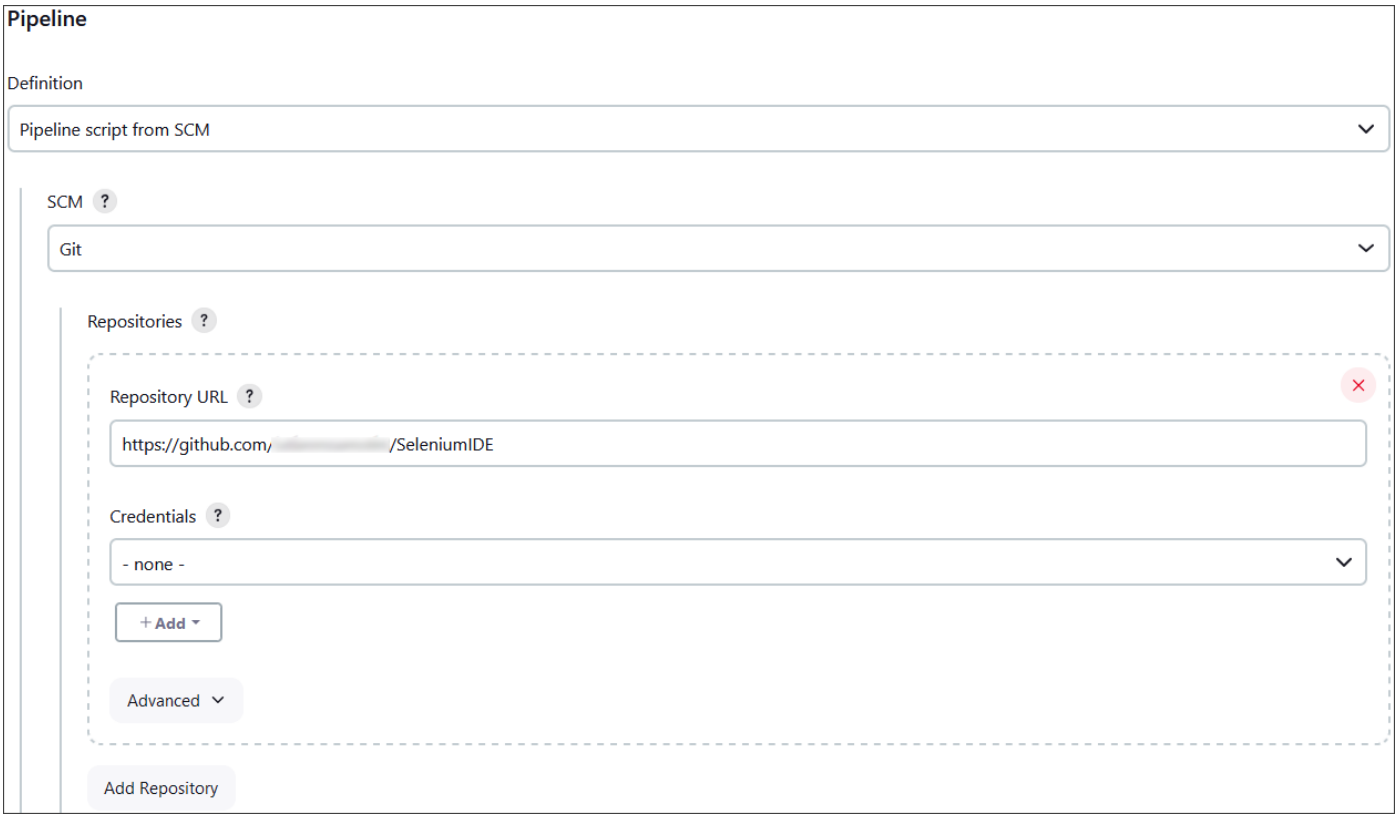
Then, scroll down to the **`Pipeline`** section in the job configuration, and from the **`Definition`** dropdown menu, select the **`Pipeline script from SCM`** option.

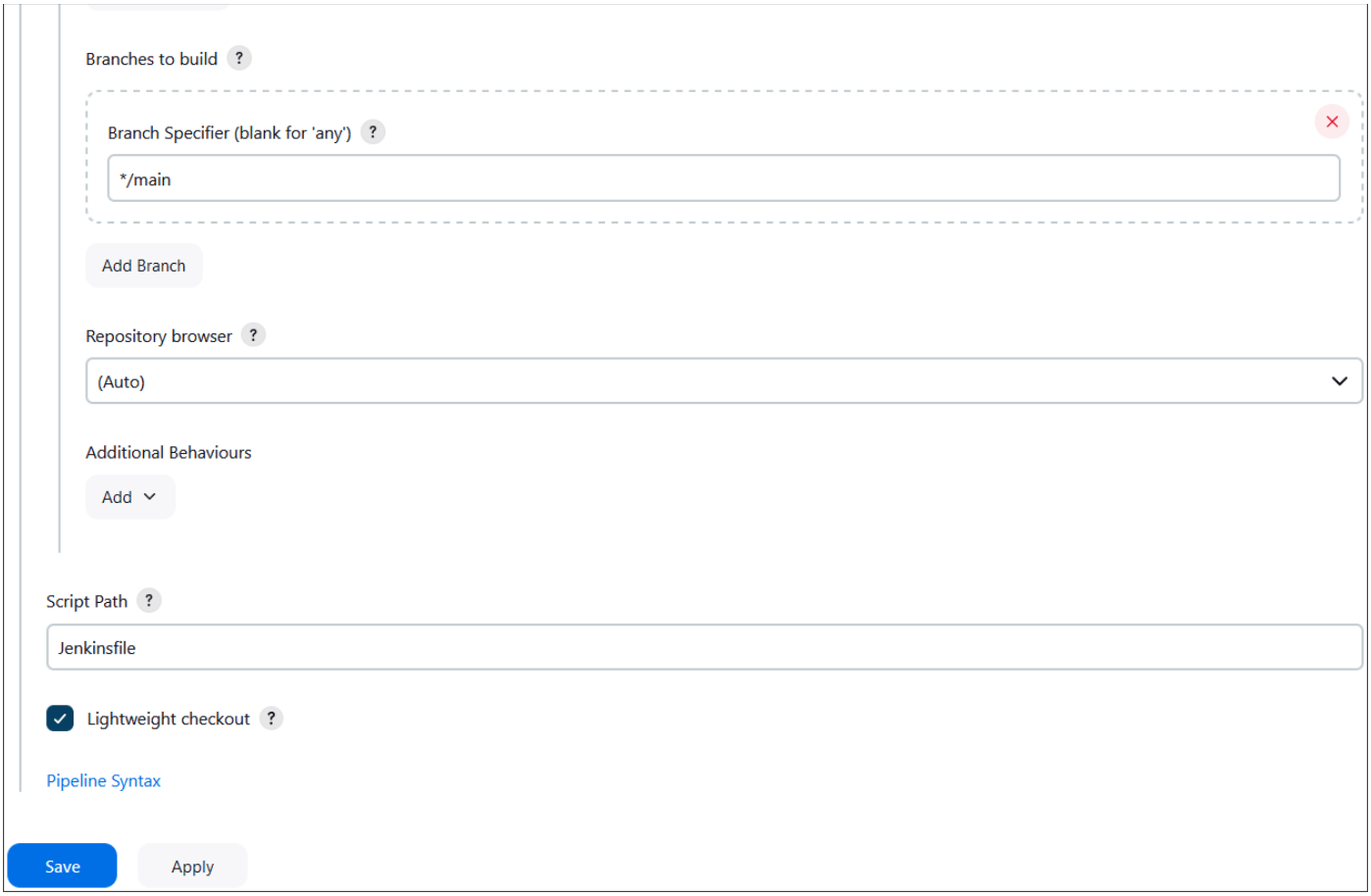
After that, select **Git** as the **SCM** and enter **your GitHub repository URL**.

Under **`Branches to build`**, enter the **branch name** that contains your **Jenkinsfile**.

Under **`Script Path`**, ensure it points to your **Jenkinsfile** (for example, type in **Jenkinsfile** if it's in the repository root).

Your configuration should look like the images below:





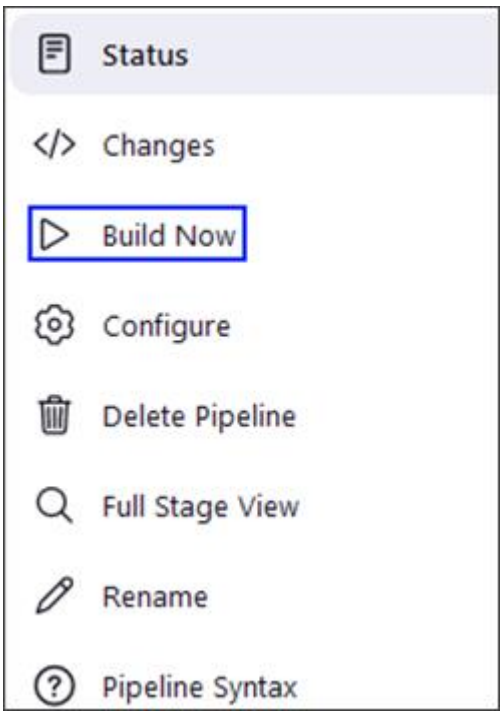
Finally, click on the **[Save]** button.

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, click on the **`Build Now`** option to start a new build manually.

You can monitor the build progress by clicking on the build number and then **`Console Output`**.



Declarative: Checkout SCM	Checkout code	Set up .NET Core	Uninstall Current Chrome	Install Specific Version of Chrome	Download and Install ChromeDriver	Restore dependencies	Build	Run tests	Declarative: Post Actions
1s	946ms	1s	6s	18s	6s	3s	1s	3s	91ms
1s	919ms	1s	7s	16s	1s	15s	2s	7s	82ms

6. Selenium Web Driver

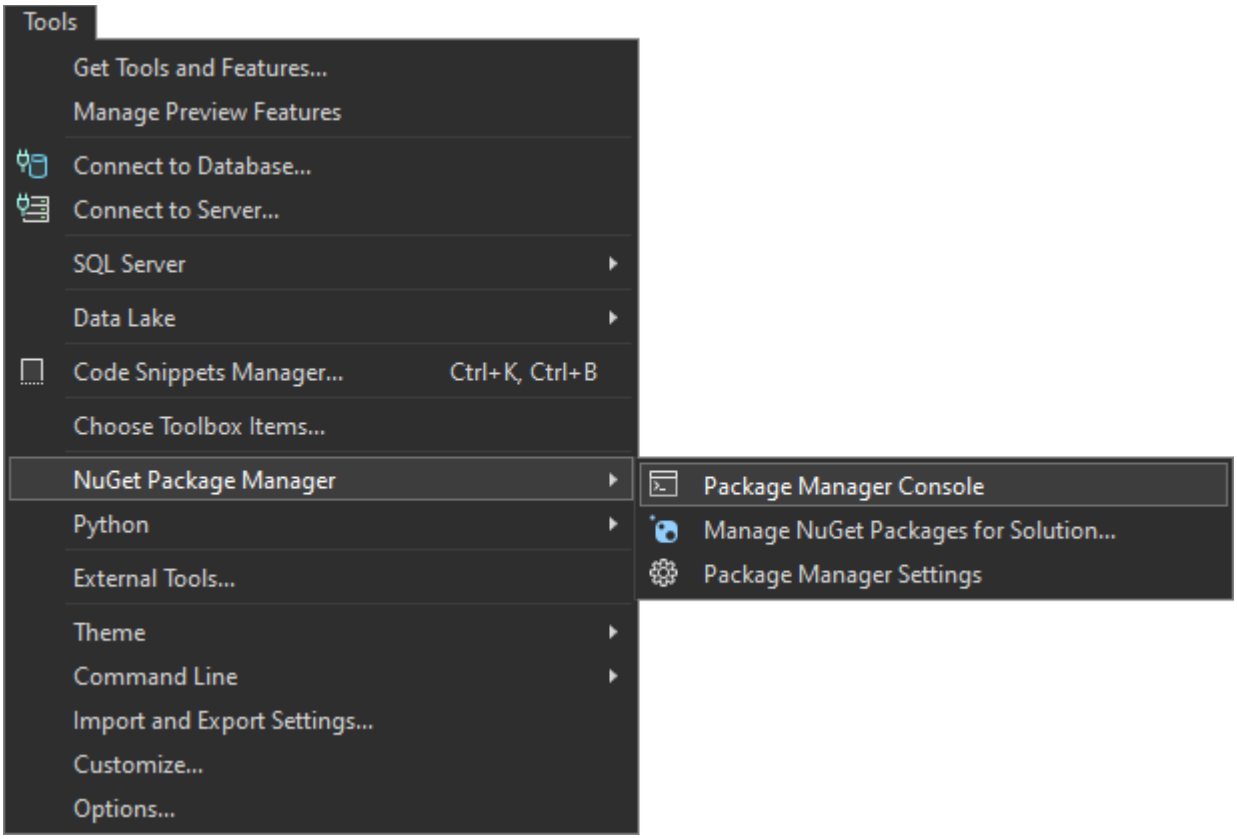
Our second task will be to create a CI for using Selenium to automate several test projects, combined in one solution.

Step 1: Run the App Locally

We have the "**`SeleniumBasicExercise`**" **solution** in the **resources which has four test projects already**. Your task is to **create a CI workflow** with **GitHub Actions** to **run the tests automatically.**

It's a good practice to **build the solution locally** in **Visual Studio**, in order to be sure everything works properly and as expected.

Open **Visual Studio** and from there navigate to the **Tools** menu. Select **NuGet Package Manager** and select **Package Manager Console**:



Let's first build the application by using the **`dotnet build`** command:

```
Package Manager Console
Package source: All        Default project: HTMLElements01
PM> dotnet build
MSBuild version 17.8.3+195e7f5a3 for .NET
  Determining projects to restore...
  All projects are up-to-date for restore.
  HTMLElements01 -> C:\Users\[redacted]\Desktop\SeleniumBasicExercise\HTML_Elements_01\bin\Debug\net6.0\HTMLElements01.dll
  DataDriven -> C:\Users\[redacted]\Desktop\SeleniumBasicExercise\DataDriven\bin\Debug\net6.0\DataDriven.dll
  HTMLElements02 -> C:\Users\[redacted]\Desktop\SeleniumBasicExercise\HTML_Elements_02\bin\Debug\net6.0\HTMLElements02.dll
  HTMLElements03 -> C:\Users\[redacted]\Desktop\SeleniumBasicExercise\HTMLElements03\bin\Debug\net6.0\HTMLElements03.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:04.51
121 %
```

After you have **ensured** that the **build** was **successful**, you can **run** the **tests**, too, by using the **`dotnet test`** command or just by clicking on the **`[Run All Tests in View]`** button in the **`Text Explorer`**.

After we have ensured that the **tests run successfully**, we can proceed with the next step.

Step 2: Create a GitHub Repo

Now you should **upload the solution to `GitHub`**.

It's a **good** practice to start using the **console** and not the interface of GitHub, in case you haven't started doing so yet.

If you **don't have Git** already **installed** on your machine, follow the **provided installation instructions** from the **resources**.

Try using the **following commands** in order to initialize a repository in your project directory, add the code to the repo, commit and push:

```
C:\Users\[redacted]\Desktop\CI-Demo>git init
Initialized empty Git repository in C:/Users/[redacted]/Desktop/CI-Demo/.git/
```

```
C:\Users\[redacted]\Desktop\CI-Demo>git add .
```

```
C:\Users\[redacted]\Desktop\CI-Demo>git commit -m "initial commit"
[main (root-commit) 9dc6adf] initial commit
 13 files changed, 455 insertions(+)
```

```
C:\Users\[redacted]\Desktop\CI-Demo>git remote add origin https://github.com/[redacted]/CI-Demo
```

```
C:\Users\[redacted]\Desktop\CI-Demo>git push -u origin main
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 16 threads
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 5.34 KiB | 1.78 MiB/s, done.
Total 15 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/[redacted]/CI-Demo
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

After running the commands, **check you GitHub repo** – the application code should be visible.

Step 3: Add Changes to Test Files

Before creating the workflow file, we have to make some adjustments in the **.cs** files. This is needed due to the fact that the default GitHub runner does not have Chrome installed. We will take care of this in the workflow, but we also need the prepare the tests to run Chrome in a headless mode within the CI environment.

In order to do that, go to the **`SetUp()`** method of each project and add the following code:

```
ChromeOptions options = new ChromeOptions();
// Ensure Chrome runs in headless mode
options.AddArguments("headless");
// Bypass OS security model
options.AddArguments("no-sandbox");
// Overcome limited resource problems
options.AddArguments("disable-dev-shm-usage");
// Applicable to Windows OS only
options.AddArguments("disable-gpu");
// Set window size to ensure elements are visible
options.AddArguments("window-size=1920x1080");
// Disable extensions
options.AddArguments("disable-extensions");
// Remote debugging port
options.AddArguments("remote-debugging-port=9222");
```

Then, we need to pass the **`ChromeOptions`** to the **`ChromeDriver`** constructor:

```
driver = new ChromeDriver(options);
```

Don't forget to **commit** and **push** the changes to each one of the files.

Step 4: Create and Run Workflow

Now, it's time to set up the Jenkins file.

Try doing this on your own. The only difference here is that here we have to run three test projects, not just one. Think how you can achieve running the three test projects separately.