# Workshop: CI System with Selenium Appium Tests – Part II

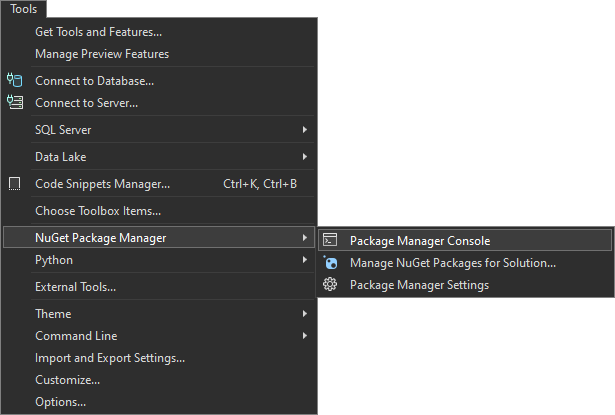
## Selenium IDE

### 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.

|  |  |
| --- | --- |
| Icon  Description automatically generated | 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:

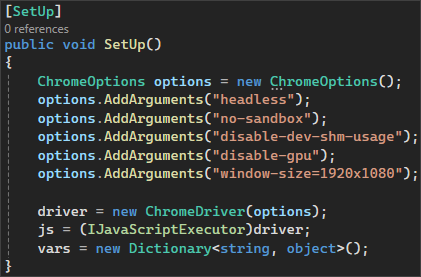
|  |
| --- |
| **git init**  **git add .**  **git commit -m "Initial commit"**  **git remote add origin** [**https://github.com/{name-of-your-repository}**](https://github.com/%7bname-of-your-repository%7d)  **git push -u origin main** |

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:



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

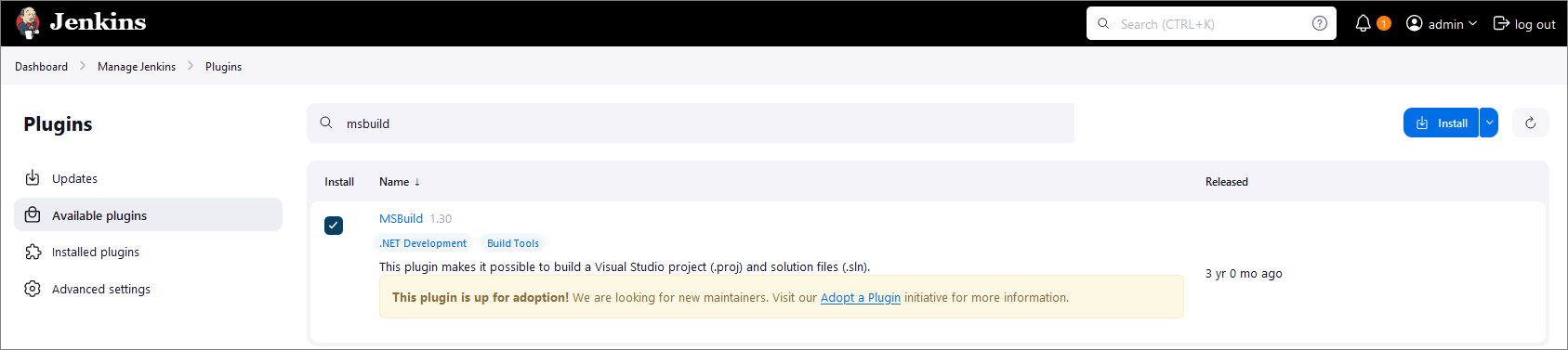
### Step 4: Configure Tools in Jenkins

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

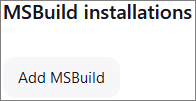
Usually, **Git** is being **installed** when you are **configuring** your **Jenkins** installation and we **already** used it in the previous task.

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:



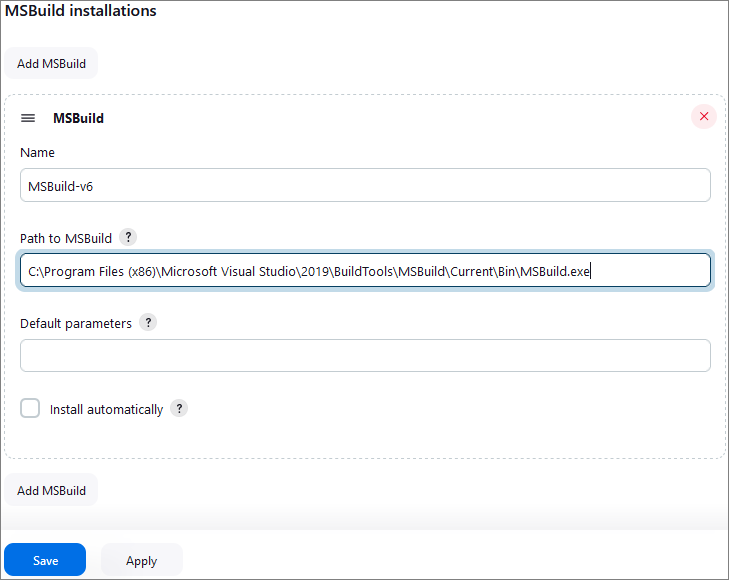
Once you have the needed plugin installed, go back to **Manage** **Jenkins** and select **Tools.** Scroll down to find the **MSBuild installations** section and click on **[Add MSBuild]** button:



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:

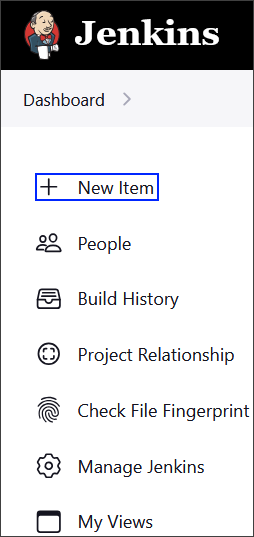


Finally, click on the **[Save]** button.

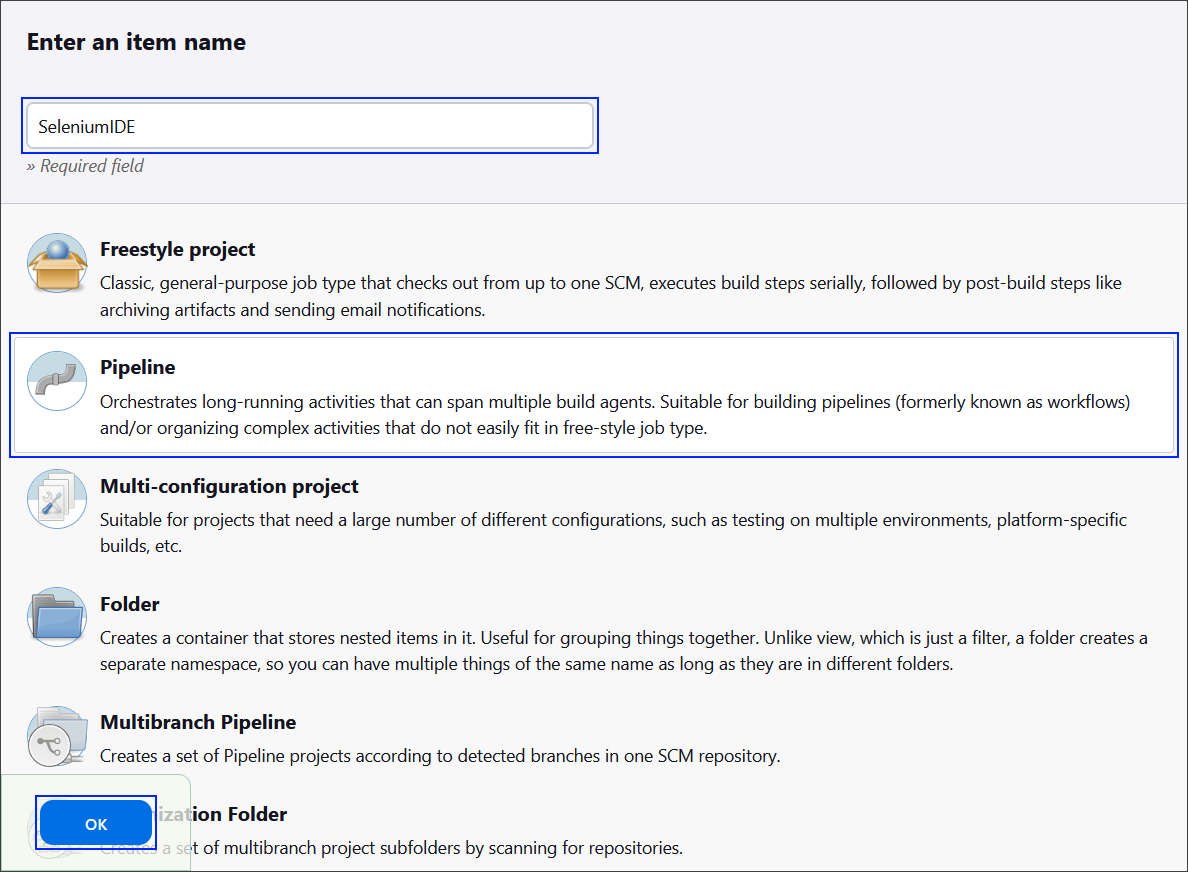
### Step 5: Create a New Job

Now, let's access Jenkins. Open the Jenkins interface in a web browser. This is usually at [**http://localhost:8080**](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.

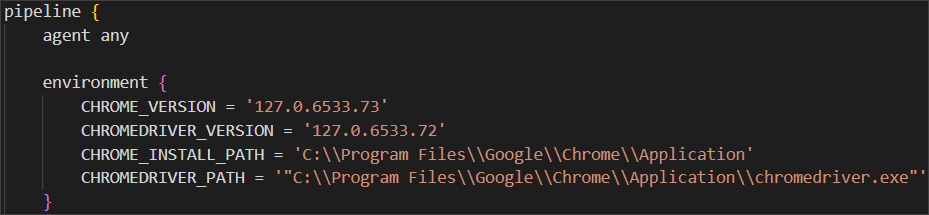
The Jenkinsfile should contain **steps** for:

* Checkout the code
* Set up .NET Core
* 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**



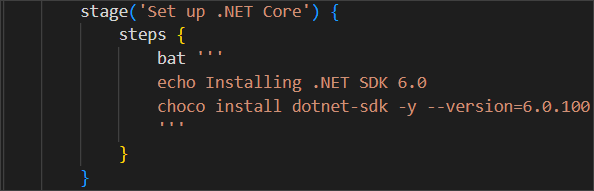
**Checkout Code Stage**

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

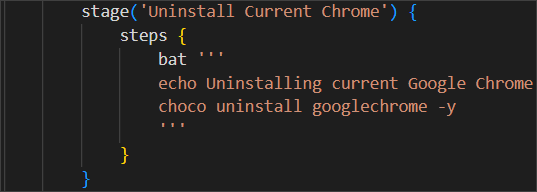


**Set up .NET Core Stage**

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

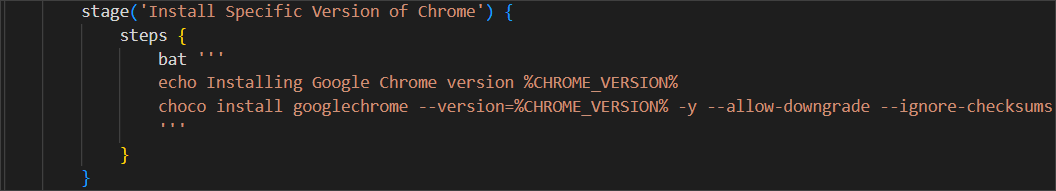


**\* Uninstall Current Chrome Stage**

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

**\* Uninstall Current Chrome Stage**

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



**\* Download and Install ChromeDriver Stage**

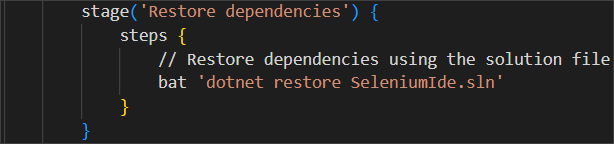
This step is optional and is used in combination with the previous two previous steps.

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\_win32.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"**  **'''**  **}**  **}** |

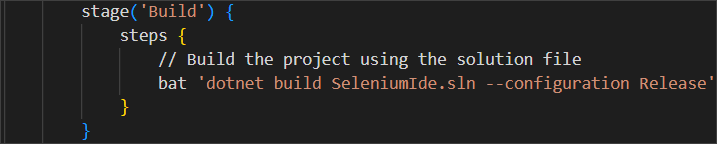
**Restore Dependencies Stage**

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



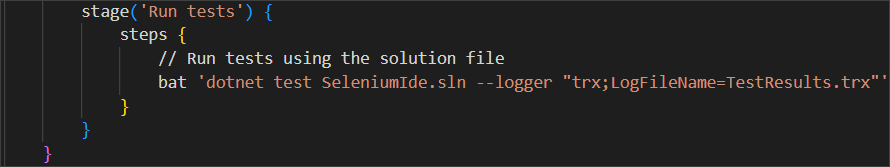
**Build Stage**

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



**Run Tests Stage**

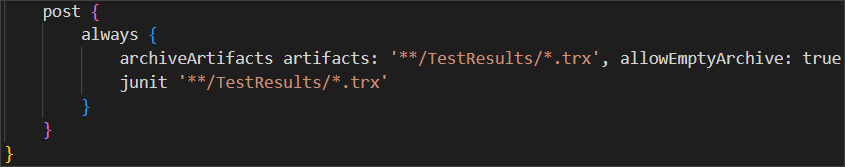
Finally, after we have set everything needed, we can define a stage for running the tests.



**\* 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.

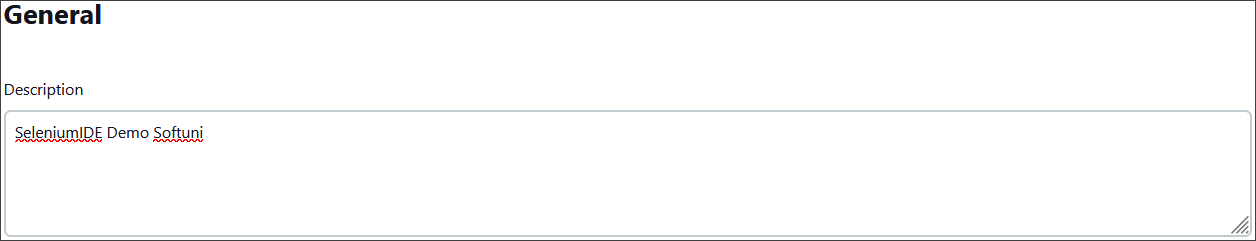


Create your file and upload it to your GitHub repository, containing the code for the application.

### 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:  


****

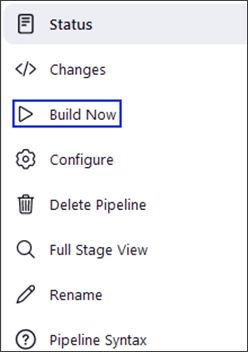
Finally, click on the **[Save]** button.

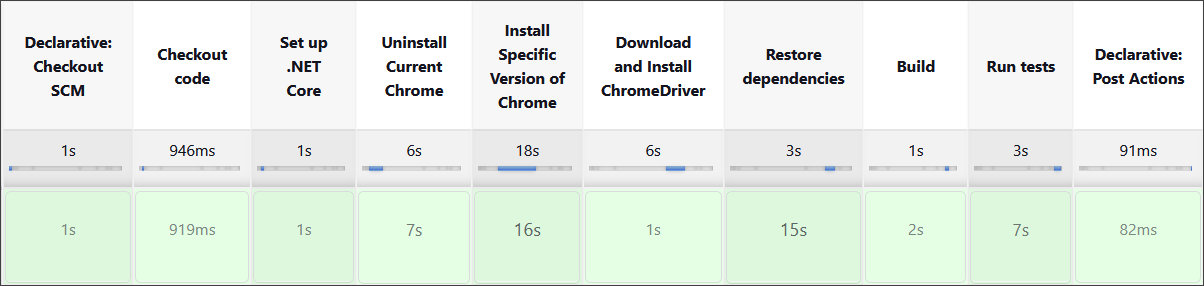
### Step 8: Test the CI Pipeline

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**.





## Selenium Web Driver

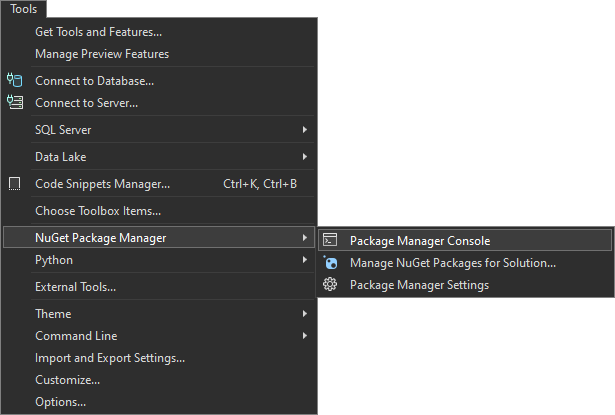
Our second task will be to create a CI for using Selenium to automate several test projects, combined in one solution.

### Step 1: Run the App Locally

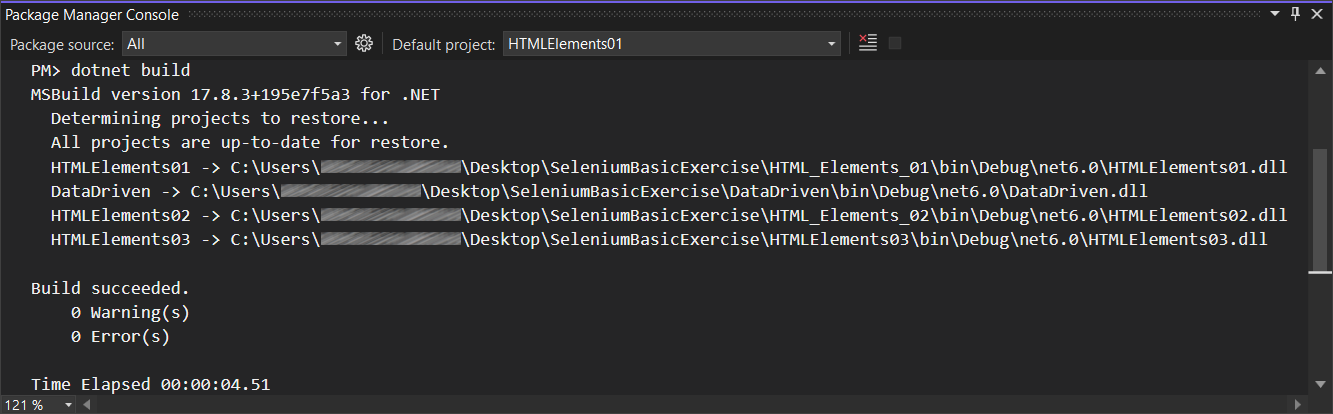
We have the "SeleniumBasicExercise" solution in the **resources which has four test projects already**. Your task is to **create a CI workflow** with **GitHub Actions** to **run the tests automatically.**

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:



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.

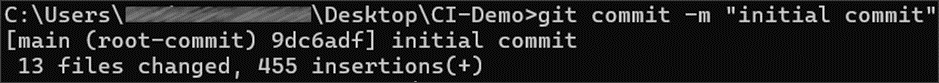
It's a good practice to start using the console and not the interface of GitHub, in case you haven't started doing so yet.

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

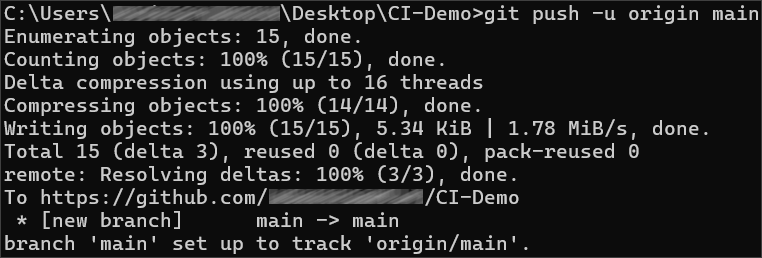
Try using the following commands in order to initialize a repository in your project directory, add the code to the repo, commit and push:







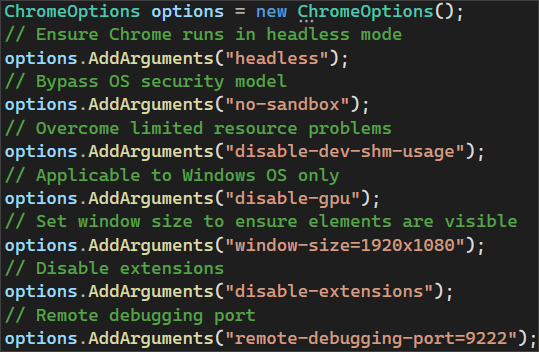




After running the commands, check you GitHub repo – the application code should be visible.

### 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:  


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



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.