Exercise: CI/CD in GitHub Actions

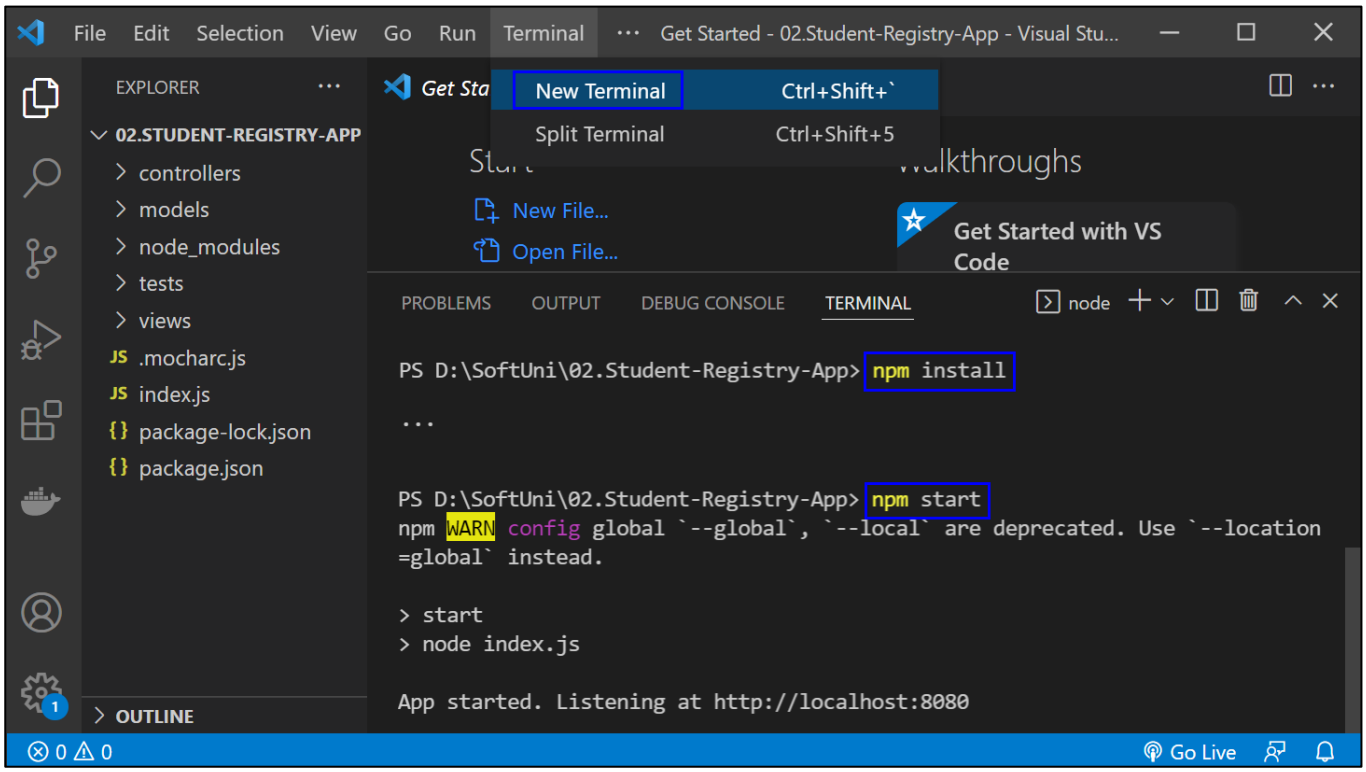
Exercises for the "DevOps for Developers" module @ SoftUni

1. CI Workflow – "Student Registry" App

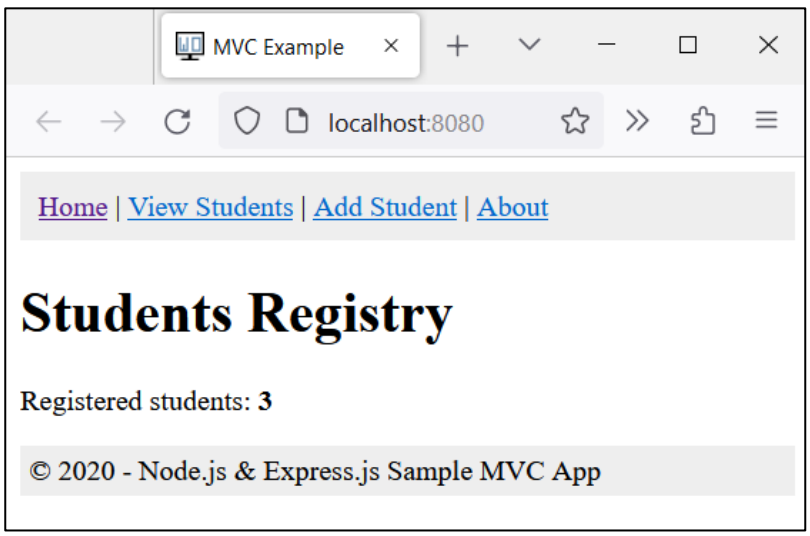
Step 1: Run the App Locally

We have the "**`Student Registry`**" **`Node.js` app** in the **resources**. Your task is to **create a CI workflow** in **`GitHub Actions`** to **start and test the app** on three different Node.js versions:

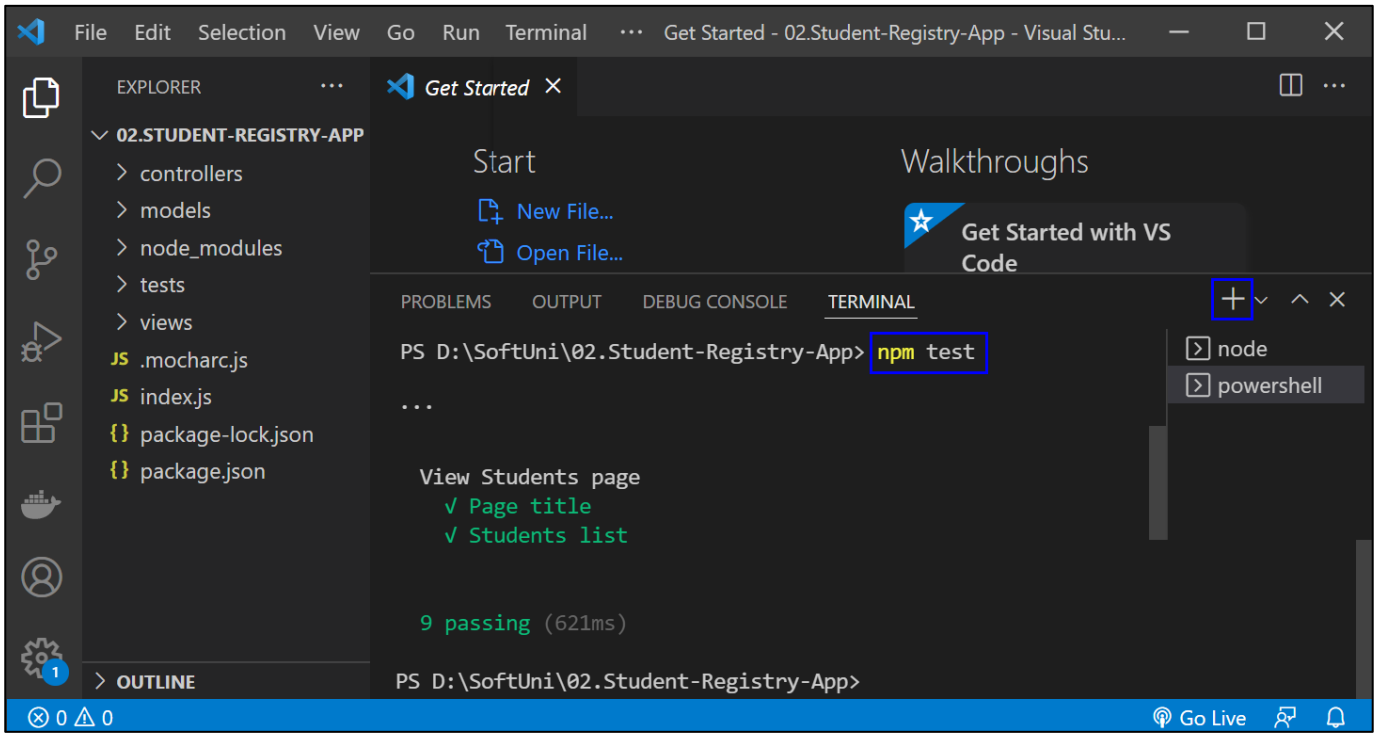
Let's first **start the app locally** in **`Visual Studio Code`**. To do this, you should **open the project**, open a **new terminal** from **`[Terminal]`** → **`[New Terminal]`** and **execute** the "**`npm install`**" and "**`npm start`**" **commands**:



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:8080:



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`**.

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

```
C:\Users\[redacted]\Desktop\HRS-app>git add .
```

```
C:\Users\[redacted]\Desktop\HRS-app>git commit -m "initial commit"
[main (root-commit) 52bc0bb] initial commit
```

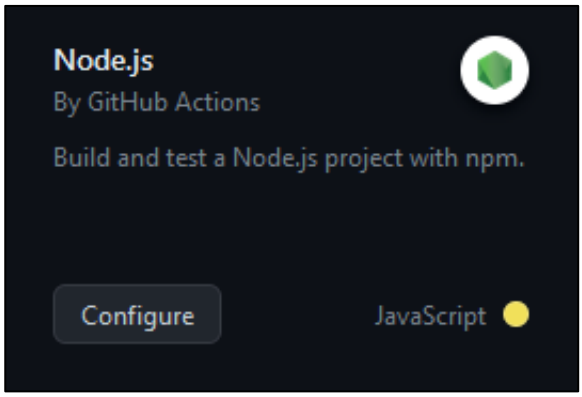
```
C:\Users\[redacted]\Desktop\HRS-app>git remote add origin https://github.com/[redacted]/HRS-app
```

```
C:\Users\[redacted]\Desktop\HRS-app>git push -u origin main
Enumerating objects: 236, done.
Counting objects: 100% (236/236), done.
Delta compression using up to 16 threads
Compressing objects: 100% (221/221), done.
Writing objects: 100% (236/236), 938.09 KiB | 5.58 MiB/s, done.
Total 236 (delta 62), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (62/62), done.
To https://github.com/[redacted]/HRS-app
 * [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: Create and Run Workflow

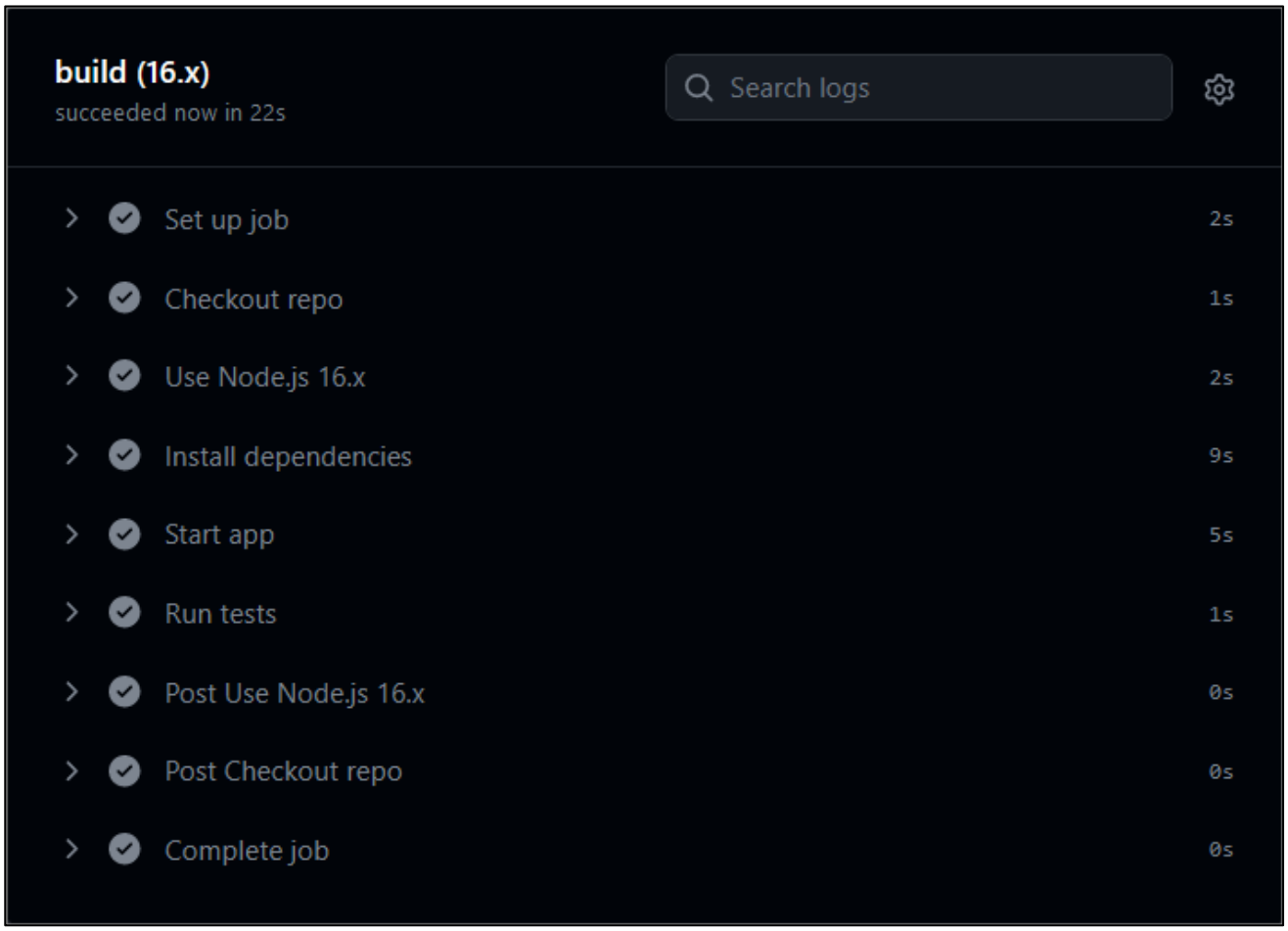
Now you should **upload the app code to `GitHub`** and **create a `GitHub Actions CI` workflow** to **start and test the app**. You can use the **following template**:



Before you commit the **generated YAML workflow file**, you should:

- **Change the YAML file name** to something more meaningful
- **Examine the workflow**, the **job** you have and its **steps**
- **Run the job** on the **last `Node.js` versions**: **`18.x`**
- **Change the workflow name**
- **Modify workflow job steps**: you should **use the three commands** which we used above to **start and test the app**, not the ones you have in the generated YAML file or **your workflow won't be successful**
- **Add names for each step** in your workflow job

Finally, **run the workflow job** and make sure that **it is successful**:



2. CD Workflow – "Student Registry" App

Now, let's **create a CD workflow** for the "**`Student Registry`**" **`Node.js` app** to **deploy it to `Render.com`**.

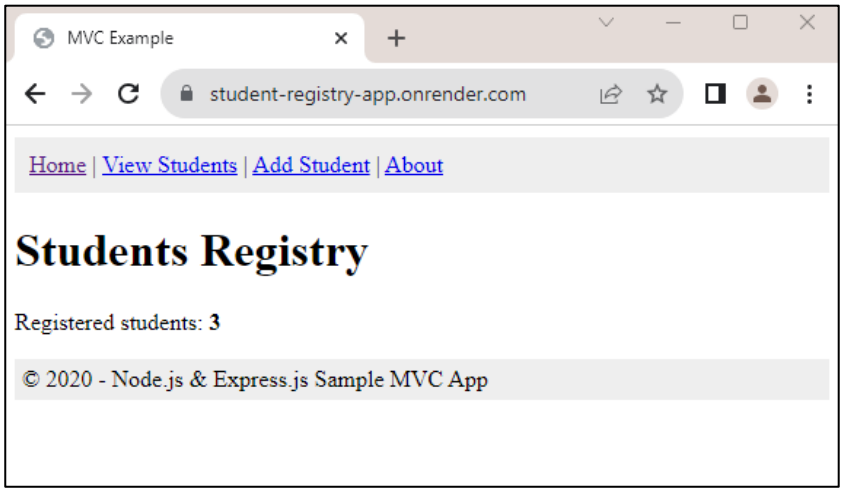
We will continue working on the file that we created for the **CI** workflow.

To do this, you should **fulfill the following steps**:

- Create a free **`Render.com` account**
- **Generate an API Token:**
  - Navigate to the "**`API Keys`**" section in your **`Render.com`** Account settings;
  - Generate an API token by clicking on "**`Create API Key`**";
  - Give it a meaningful name (e.g., "**`GitHub Actions Token`**");
  - Click on "**`Create Token`**" to generate it.
- Add a new **Web Service:**
  - Connect your **GitHub account** to the service;
  - Connect your **GitHub repository** holding the application;
  - Give your **service a unique** and **meaningful** name;
- Add Render **Service ID as a GitHub Secret**:
  - Go to the **Settings menu** of your web service in Render.com and find the **`Deploy Hook`**;
  - **Copy the value that matches the pattern from the red square:**

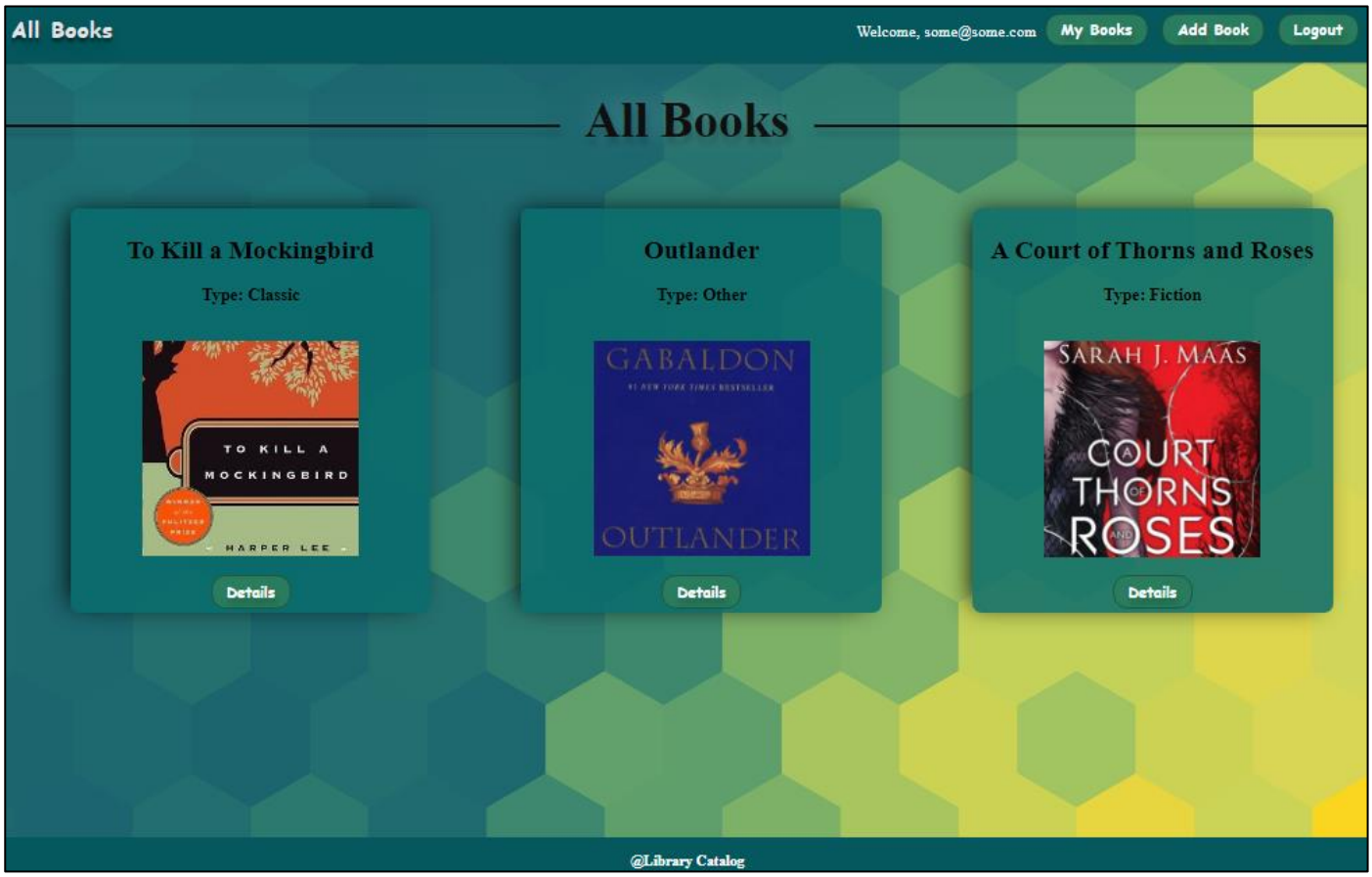
    - Go to your GitHub repository, click on "**`Settings`**," then select "**`Secrets and variables`**" from the left sidebar;
    - Click on "**`Actions`**" and then click on "**`New repository secret`**" and add a new secret with the following details:
        - Name: **`SERVICE_ID`**
        - Value: The service id that you copied from Render.com
    - Click "**`Add secret`**" to save it.
- Add **Render.com API Token as a GitHub Secret:**
    - Go to your GitHub repository, click on "**`Settings`**," then select "**`Secrets and variables`**" from the left sidebar;
    - Click on "**`Actions`**" and then click on "**`New repository secret`**" and add a new secret with the following details:
        - Name: **`RENDER_TOKEN`**
        - Value: The API token you generated on Render.com
    - Click "**`Add secret`**" to save it.
- **Create and define the CD workflow:**
    - Set the **job** to be **dependent** of the **`test`** job from the **CI workflow**
    - In the **YAML** file that we used for the CI workflow, use the **custom** GitHub action **`johnbeynon/render-deploy-action@v0.0.8`** to deploy the application to Render;
    - Use the Render service ID and API key, which are stored as secrets in the repository.

GitHub Actions will execute the CD workflow, which involves installing Node.js, installing dependencies, and deploying the app to **`Render.com`**. The workflow will log in to **`Render.com`** using the API token you provided as a secret and then deploy your app.



3. * CI/CD Workflow – "Library Catalog" App

We have the "**`Library Catalog`**" **app** in the **resources**. Your task is to **create a CI/CD workflow** in **`GitHub Actions`** to **start, test** and **deploy** the app to Render.com following the steps from the previous tasks.



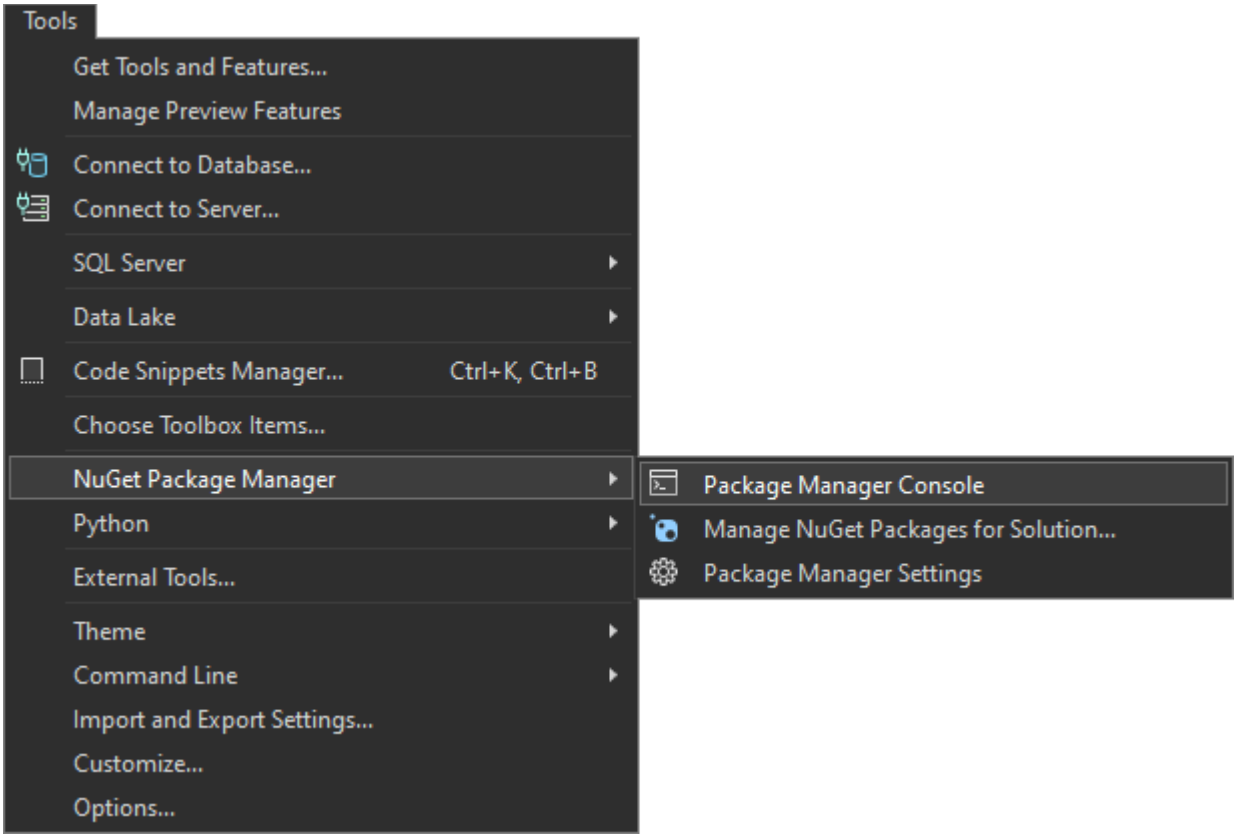
4. "HouseRentingSystem" App – ASP.NET Core MVC app

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 **GitHub Actions** 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.

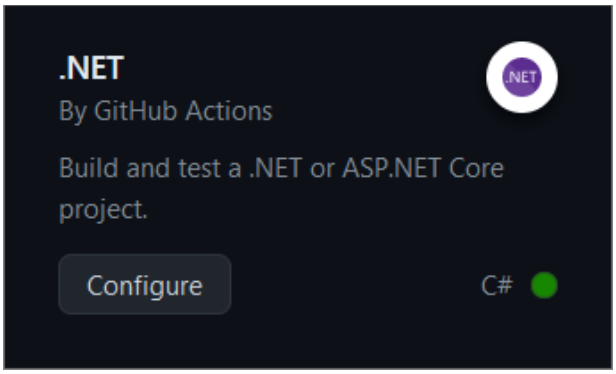
Step 2: Create a GitHub Repo

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: Create and Run Workflow

Now you should **create a `GitHub Actions CI` workflow** to **start and test the app**.

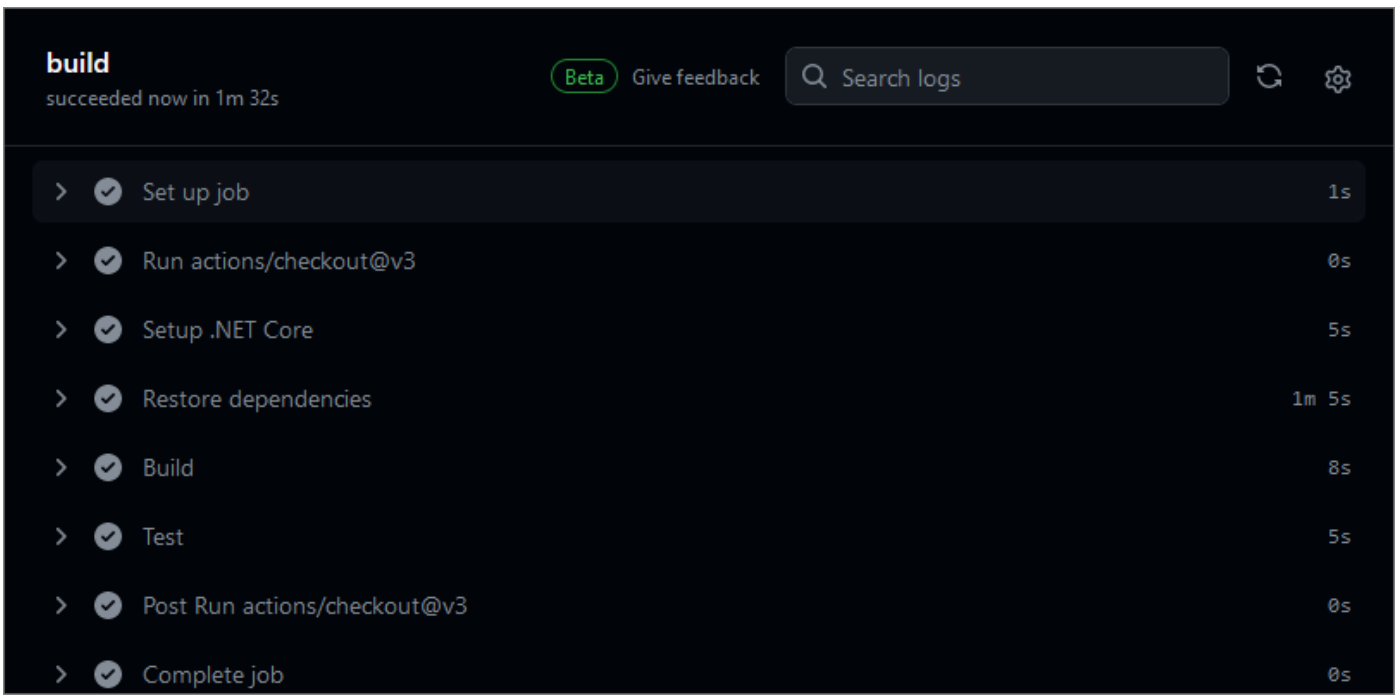
You can use the **following template**:



Before you commit the **generated YAML workflow file**, you should:

- **Change the YAML file name** to something more meaningful
- **Examine the workflow**, the **job** you have and its **steps**
- **Run the job** on .NET version **`6.0`**
- **Change the workflow name**
- **Modify workflow job steps**: you should **have jobs for**
  - **Setting up .NET Core**
  - **Restoring dependencies**
  - **Building the app**
  - **Running the tests**
- **Add names for each step** in your workflow job

Finally, **run the workflow job** and make sure that **it is successful**:



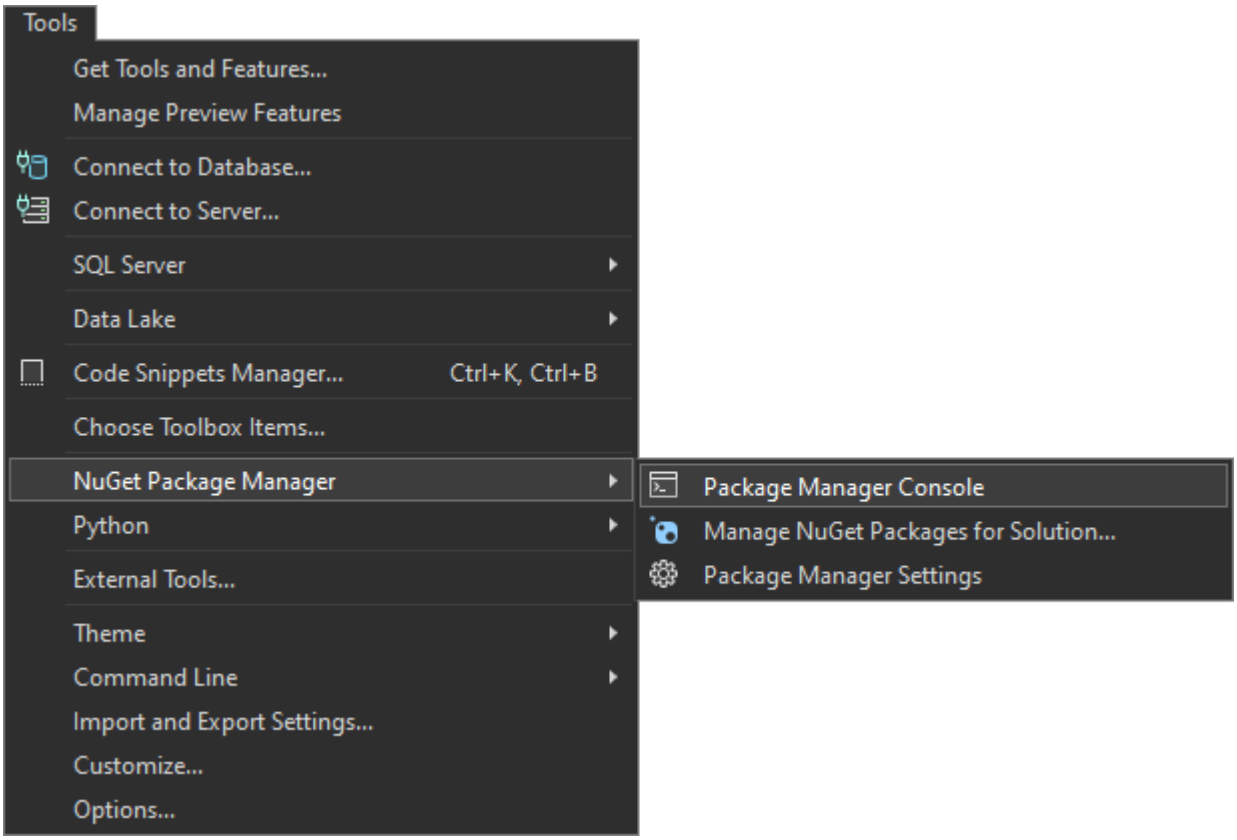
5. Selenium IDE

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.

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: Create and Run Workflow

Now you should **create a `GitHub Actions CI` workflow** to **start and test the app**.

In the root directory of the repository, create a new folder **`.github`** and in it create another one, called **`workflows`**. Then, inside this new folder, create a **YAML file**, which will hold the workflow definition.

Now, let's define our workflow file.

We have to give it a meaningful name and specify the event which will trigger the workflow. In our case, this will be the push and pull request events on the **`main`** branch:

```yaml
name: Selenium IDE CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

Then, we have to specify the **job** and the **environment**:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
```

After that, we start defining the **steps.** You have to create several **steps** for the **job**:

- **Checkout code**
  - Give the step a meaningful name
  - Checkout the repository code

```yaml
    steps:
    - name: Checkout code
      uses: actions/checkout@v3
```

- **Set up .NET Core**
  - Give the step a meaningful name
  - Use the appropriate action to set up the .NET Core SDK
    - Specify the .NET Core version

```yaml
    - name: Set up .NET Core
      uses: actions/setup-dotnet@v3
      with:
        dotnet-version: '6.0.x'
```

- **Install Chrome**
  - Give the step a meaningful name
  - Executes commands to update the package list and install Google Chrome

```yaml
    - name: Install Chrome
      run: |
        sudo apt-get update
        sudo apt-get install -y google-chrome-stable
```

- **Install dependencies**
  - Give the step a meaningful name
  - Run the appropriate command to restore the dependencies specified in the solution file

```yaml
    - name: Install dependencies
      run: dotnet restore
```

- **Build the solution**
  - Give the step a meaningful name
  - Run the appropriate command to build the solution without restoring the dependencies again

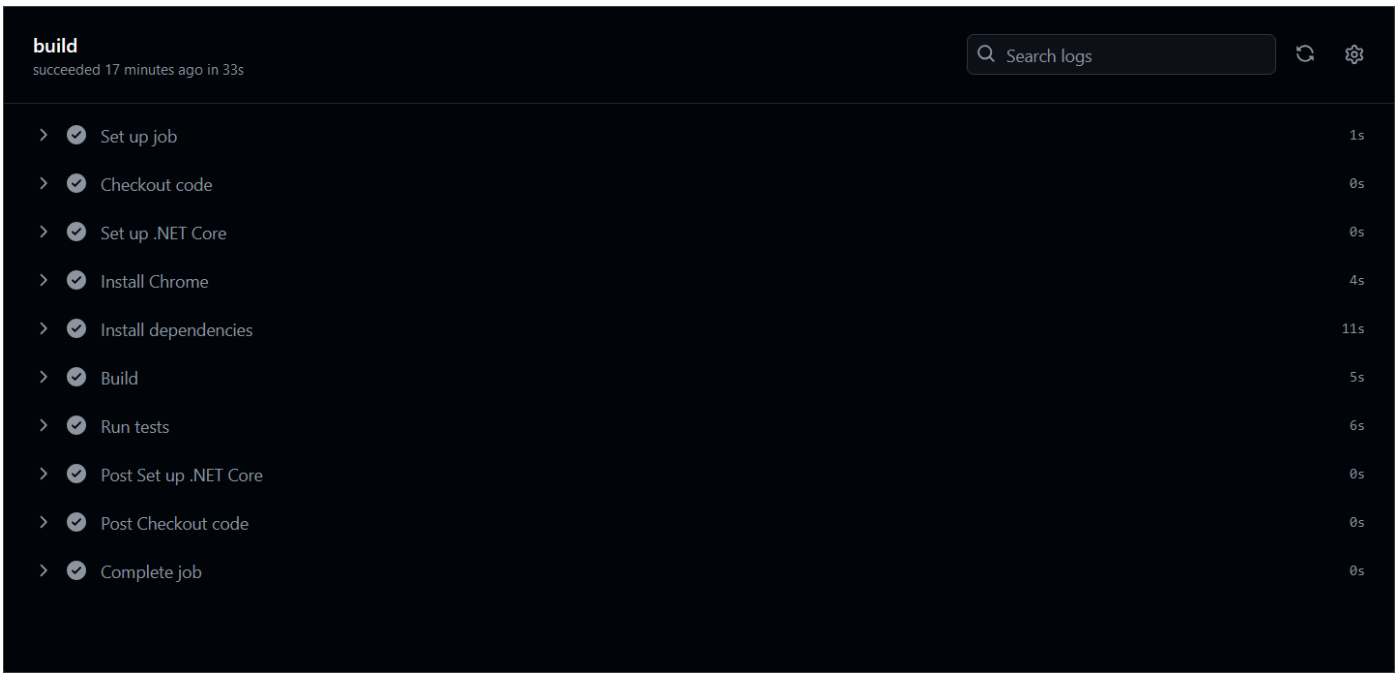
```
  - name: Build
    run: dotnet build --no-restore
```

- **Run the test project**
  - Give the step a meaningful name
  - Sets the environment variable **`CHROMEWEBDRIVER`** to the path of the Chrome executable
  - Run the tests in the project with normal verbosity

```
  - name: Run tests
    env:
      CHROMEWEBDRIVER: /usr/bin/google-chrome
    run: dotnet test --verbosity normal
```

Now, commit the changes to the main branch of the repository.

Finally, **the workflow job should run after the commit.** Make sure that it is **successful**:



6. Selenium Web Driver

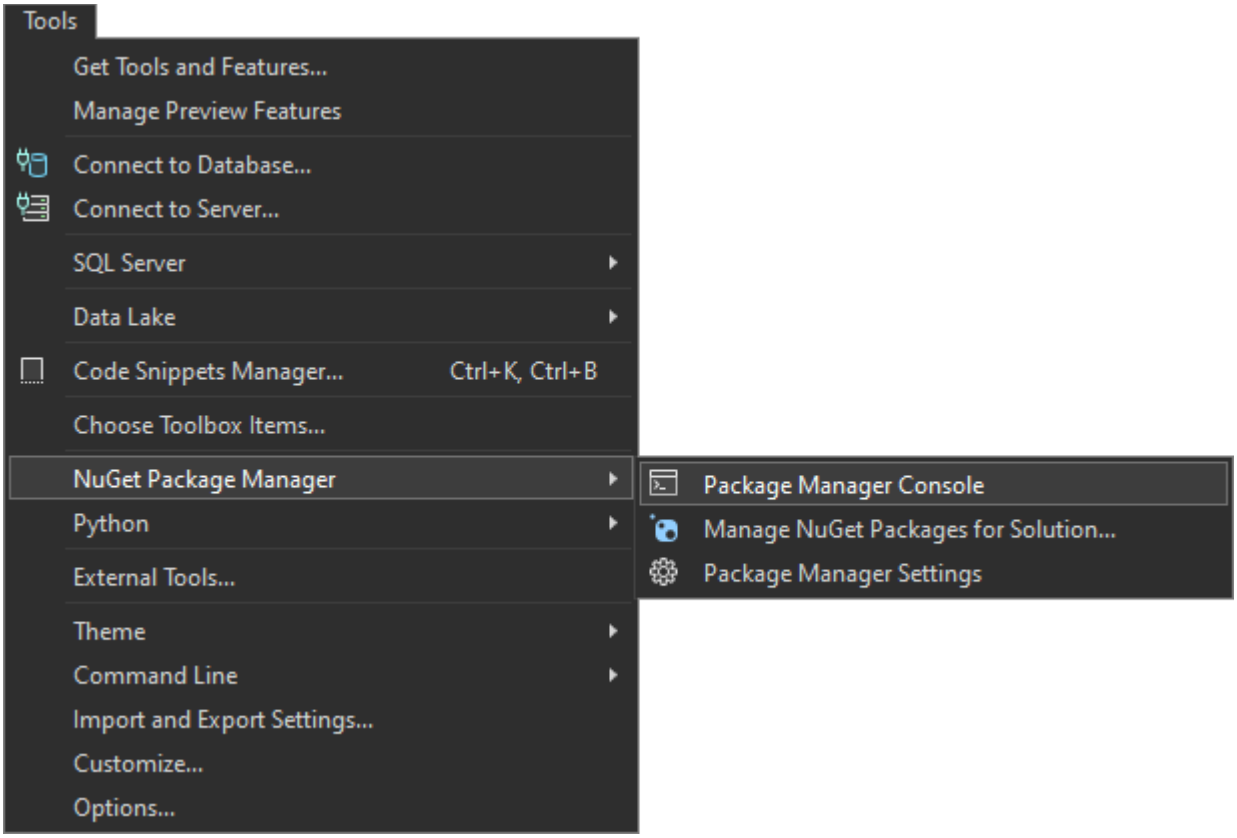
Our second task will be to create a CI for using Selenium to automate several test projects, combined in one solution.

Step 1: Run the App Locally

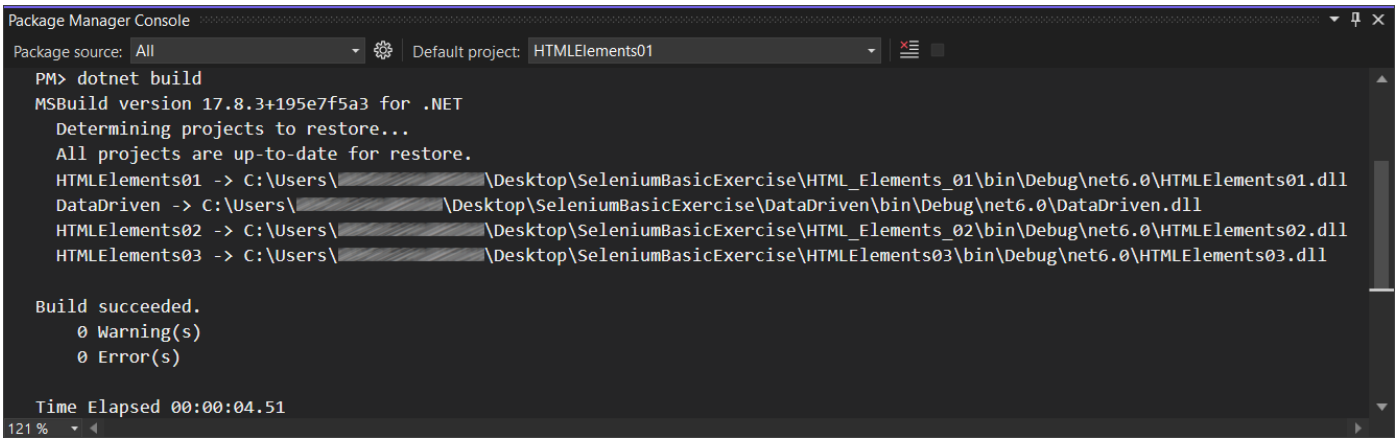
We have the "**`SeleniumBasicExercise`**" **solution** in the **resources which has four test projects already**. Your task is to **create a CI workflow** with **GitHub Actions** to **run the tests automatically.**

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:



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 you should **create a `GitHub Actions CI` workflow** to **start and test the app**.

In the root directory of the repository, create a new folder **`.github`** and in it create another one, called **`workflows`**. Then, inside this new folder, create a **YAML file**, which will hold the workflow definition.

Now, let's define our workflow file.

We have to give it a meaningful name and specify the event which will trigger the workflow. In our case, this will be the push and pull request events on the **`main`** branch:

```yaml
name: Selenium WebDriver CI

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
```

Then, we have to specify the **job** and the **environment**:

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
```

After that, we start defining the **steps.** You have to create several **steps** for the **job**:

- **Checkout code**
  - Give the step a meaningful name
  - Checkout the repository code

```yaml
    steps:
    - name: Checkout code
      uses: actions/checkout@v3
```

- **Set up .NET Core**
  - Give the step a meaningful name
  - Use the appropriate action to set up the .NET Core SDK
    - Specify the .NET Core version

```yaml
    - name: Set up .NET Core
      uses: actions/setup-dotnet@v3
      with:
        dotnet-version: '6.0.x'
```

- **Install Chrome**
  - Give the step a meaningful name
  - Executes commands to update the package list and install Google Chrome

```yaml
    - name: Install Chrome
      run: |
        sudo apt-get update
        sudo apt-get install -y google-chrome-stable
```

- **Install dependencies**

- Give the step a meaningful name
- Run the appropriate command to restore the dependencies specified in the solution file

```
- name: Install dependencies
  run: dotnet restore SeleniumBasicExercise.sln
```

- **Build the solution**
  - Give the step a meaningful name
  - Run the appropriate command to build the solution without restoring the dependencies again

```
- name: Build
  run: dotnet build SeleniumBasicExercise.sln --no-restore
```

- **Run each test project separately**
  - Give each step appropriate and meaningful name, describing which test project is being executed
  - Sets the environment variable **`CHROMEWEBDRIVER`** to the path of the Chrome executable
  - Run the tests in each project with normal verbosity

```
- name: Run TestProject1 tests
  env:
    CHROMEWEBDRIVER: /usr/bin/google-chrome
  run: |
    echo "Running TestProject1 tests"
    dotnet test TestProject1/TestProject1.csproj --verbosity normal

- name: Run TestProject2 tests
  env:
    CHROMEWEBDRIVER: /usr/bin/google-chrome
  run: |
    echo "Running TestProject2 tests"
    dotnet test TestProject2/TestProject2.csproj --verbosity normal

- name: Run TestProject3 tests
  env:
    CHROMEWEBDRIVER: /usr/bin/google-chrome
  run: |
    echo "Running TestProject3 tests"
    dotnet test TestProject3/TestProject3.csproj --verbosity normal
```

Now, commit the changes to the main branch of the repository.

Finally, **the workflow job should run after the commit.** Make sure that it is **successful**: