11

PRODUCTION-READY C#: FROM DEVELOPMENT TO DEPLOYMENT

OVERVIEW

In the previous chapters, you learned several new concepts and ideas, from C# basics such as variables and loops to the creation of web applications. To wrap everything up, this chapter will take you through some basic concepts that you need to know to push an application live. It begins with Git, a distributed version control system that is used to keep track of changes in code and teaches you how to use GitHub to keep a remote backup of your code. Finally, you will use these tools to enable Continuous Deployment (CD) and deployment from your code to the cloud. That means that you will create a pipeline to continuously build, test, and deploy your code to a cloud infrastructure for each new change you make to your code repository.

INTRODUCTION

This chapter aims to give life to the concepts you have learned previously in this book, such as creating web APIs and implementing unit tests. These concepts will be used to showcase code versioning and the deployment of an application to a production server (that is, to make it available to its intended users). The main server on which an application lives and is accessed is called a **production server**.

The concept of deployment is intertwined with code versioning. Following a strict definition, to **version** your code means to track changes made throughout the code history. The tracking of these changes allows you to create different versions of an application. You can use this tracking to not only continuously **integrate** changes to your code base but also continuously **deploy** these changes.

This chapter will teach you how to do this in practice using the core concepts you learned throughout the book, such as unit and integration testing, as well as clarifying how they can be used to build safe and trustable deployment processes.

GIT AS A CHANGE TRACKER AND BACKUP KEEPER

In *Chapter 7*, *Creating Modern Web Applications with ASP.NET*, you built your own to-do list with Razor Pages. It was an interactive chapter, as you started developing your application from scratch until it had a nice interactive board where you could move tasks across. For every exercise and activity, you went through, you had to make some changes in the code. So, what you did was make and save changes to a file, keeping your work up to date. However, there was no way to track the changes. That is where Git comes into play. Git is a **Version Control Software** (**VCS**) that allows you to track work back and forth. It was created by the creator of Linux, Linus Torvald. You might be asking yourself now what version control means.

To know this, imagine that you make a change to your code, and you want to record this change and keep on recording changes as long as your code or project grows. What Git does behind the scenes is take a **snapshot** for each change you want to record that is, one-change-one-snapshot, another-change-another-snapshot. Eventually, you end up with a series of snapshots that reflect all your work history within your code base and that also allow you to navigate back and forth through these changes. This is a powerful tool as it gives you some useful abilities, such as rolling back on work that was done incorrectly or simply seeing how the code base has evolved over time.

INSTALLING GIT

In order to use Git, you must first install the **Source Control Manager** (**SCM**) on your machine. Git is multiplatform and can be easily downloaded for any operating system by accessing Git (https://packt.link/CoJCj) and downloading the relevant installation, depending on your OS.

Once you proceed through the installation steps of your corresponding OS version, you will install Git as an SCM on your machine. The Git CLI will be your most precious ally as you go through this chapter. The CLI has everything you need to move back and forth in any Git repository.

Fortunately, **Visual Studio** (**VS**) Code (the code editor that you have been using throughout this book) has built-in support for Git that makes it easy to keep track of changes in a visual way. This can come in handy quite often. Throughout this chapter, you will learn how to use Git in both ways (VS Code and the CLI) so that you can tailor your approach to suit your project's specific requirements. As a starting point in the next section, you'll first initialize your repository using Git.

COMMITS—THE SOURCE OF TRUTH

Commits are the most important component of a Git repository, as they identify the changes made in the repository at different points in time. Here is an example log:

```
commit cbc3892d7bfe4bd69bf35d2cc1dee920045737aa (HEAD -> master)
Author: Mateus Viegas <mateuscviegas@gmail.com>
Date:   Sat Dec 19 19:39:49 2020 +0000
```

You can see the following information in the preceding snippet:

- A commit identifier is displayed on the first line, which is a long hexadecimal number. With this number, Git makes it possible to navigate throughout the repository history, meaning that you can go back and forth between each change tracked by the repository. It allows you to check out a specific point in time in Git history.
- Next, the name of the author is displayed, with both the name and email configured during repository initialization.
- Finally, the date is displayed, containing the time zone of the timestamp.

Every time a new file is created under a Git repository, it is said to be **untracked**. In order to track changes to this file, you need to **stage** it. Once this file is under Git staging, a new **commit** can be created with it. From this point on, it will be possible to track any changes made to this file. Once it is committed, Git knows that this file is **unmodified**. If this file is edited, then Git places it under the **modified** status. To commit these changes, however, the file must first be moved from modified to staged, after which the file can be committed. All of this may seem a little complex at first, but as you progress through this chapter, you will develop a clearer grasp of this process. The following figure breaks it down:

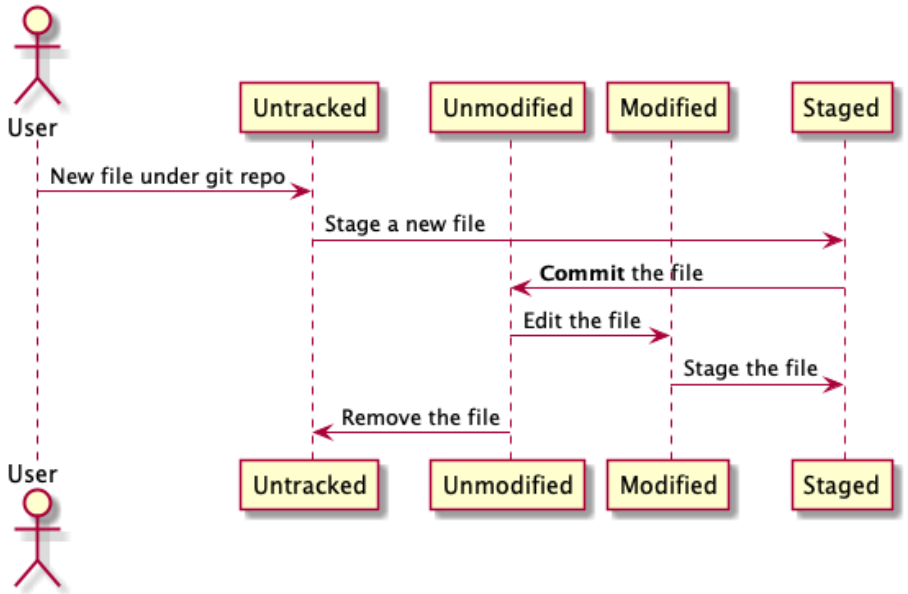

Figure 11.1: File status inside a Git repository

To check whether a repository has any files in either the modified or staged state, run the following command on the integrated terminal of VS Code:

```
git status
```

If there are no changes, you will receive the following output:

```
On branch master
nothing to commit, working tree clean
```

Note the **`tree`**, **`branch`**, and **`master`** keywords in the preceding output. You will learn about these shortly in this chapter's *Commits as Graphs* section.

Throughout this chapter, you will learn how to use Git while developing a web API that converts temperature values to different units (Celsius, Fahrenheit, and Kelvin). While creating this simple application in the following exercise, you will also implement many of the concepts you learned in previous chapters.

EXERCISE 11.01: INITIALIZING A REPOSITORY

By this point, you should be well acquainted with *The C# Workshop* GitHub repository (https://packt.link/sezEm). In this exercise, you will create a new Git repository that you will later use to develop your application.

The following steps will help you complete this exercise:

1. Open **`VS Code`**.
2. Click on **`File`** and then choose the **`Open Folder`**... option:

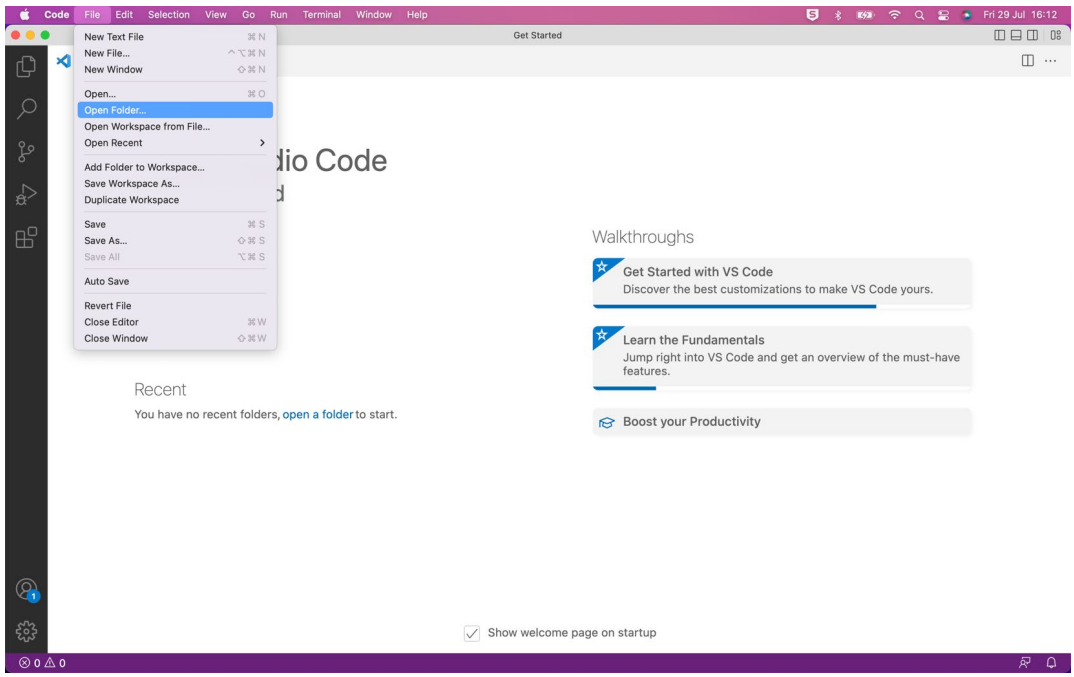

Figure 11.2: Opening a folder in VS Code

3. Create a new folder named **`temperature-converter`** and open it.

4. Now, navigate to the **`SOURCE CONTROL`** icon on the left. You'll see that there are two options—Initialize Repository and Publish to GitHub—as shown in the following figure:

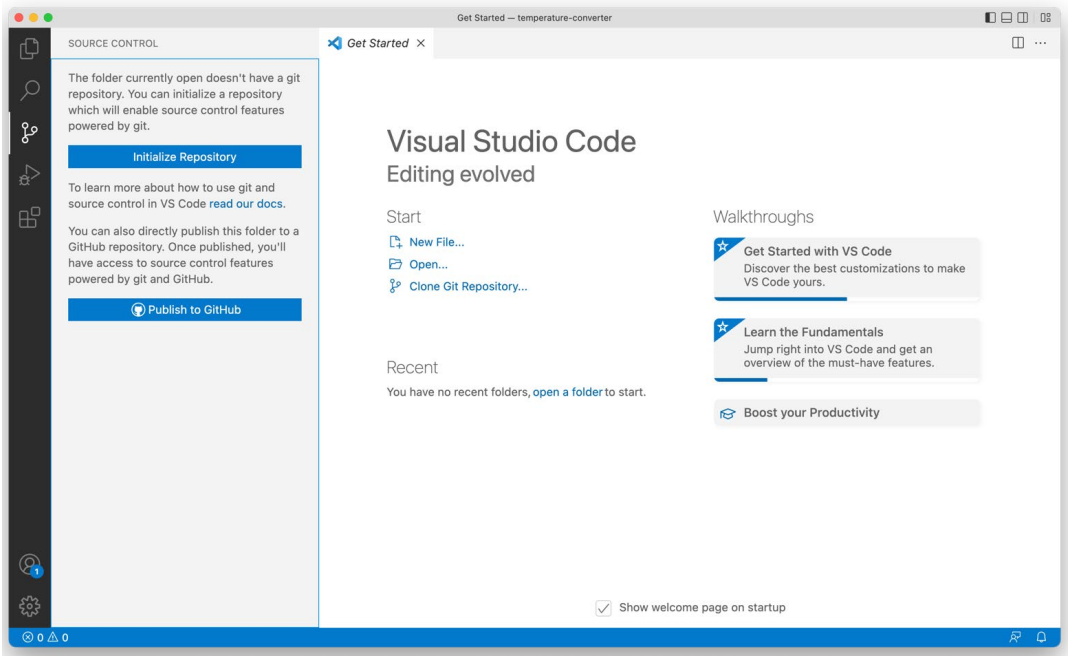

Figure 11.3: Initializing a Git repository with VS Code

The first option here is to initialize a Git repository, which is what you are going to do now. The second option is to publish this repository to GitHub, which will be covered later in this chapter.

5. For now, click on the **`Initialize Repository`** option. You will notice that the screen will be displayed as follows:

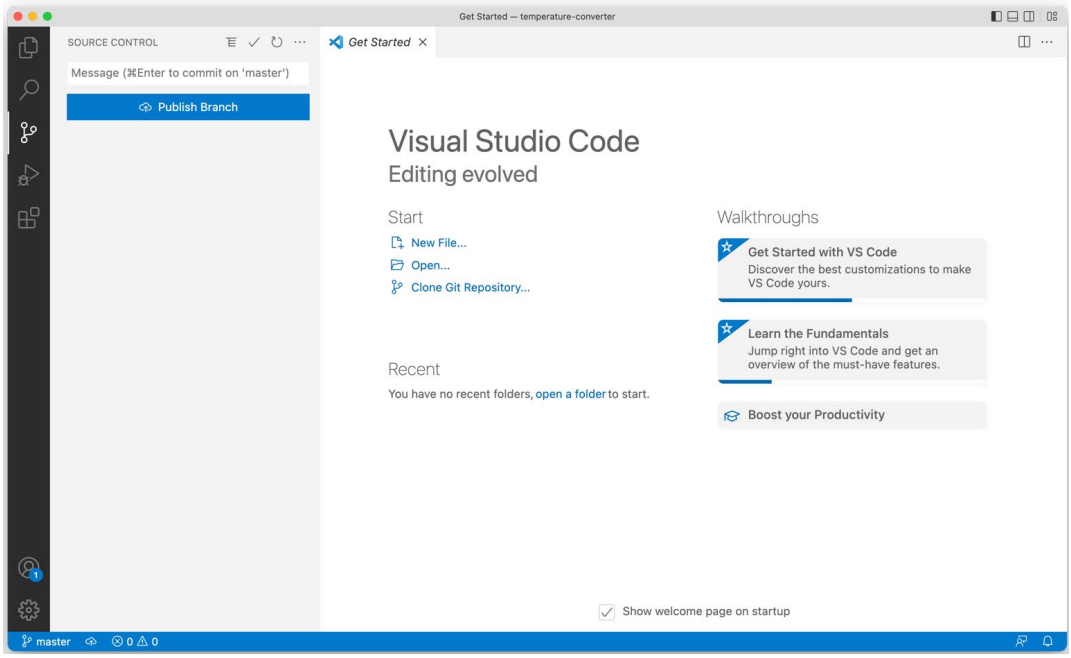

Figure 11.4: A Git repository initialized with VS Code

This means that the repository was successfully initialized. However, it does not have anything tracked yet. In the next step, you will create a file and commit it to the repository to start this tracking.

6. Open the VS Code integrated terminal and create a solution file for the API you are going to build by running the following command:

```
dotnet new sln -n TemperatureConverter
```

7. Now go back to the **SOURCE CONTROL** tab and you'll see the solution file beneath the **Changes** section.

   Here, Git has detected that a new file is present in the folder. However, it is not tracking this file yet, which means that the file is unstaged.

8. To make Git track this file, click on the plus (**+**) icon on the left of the filename to stage this change into the repository:

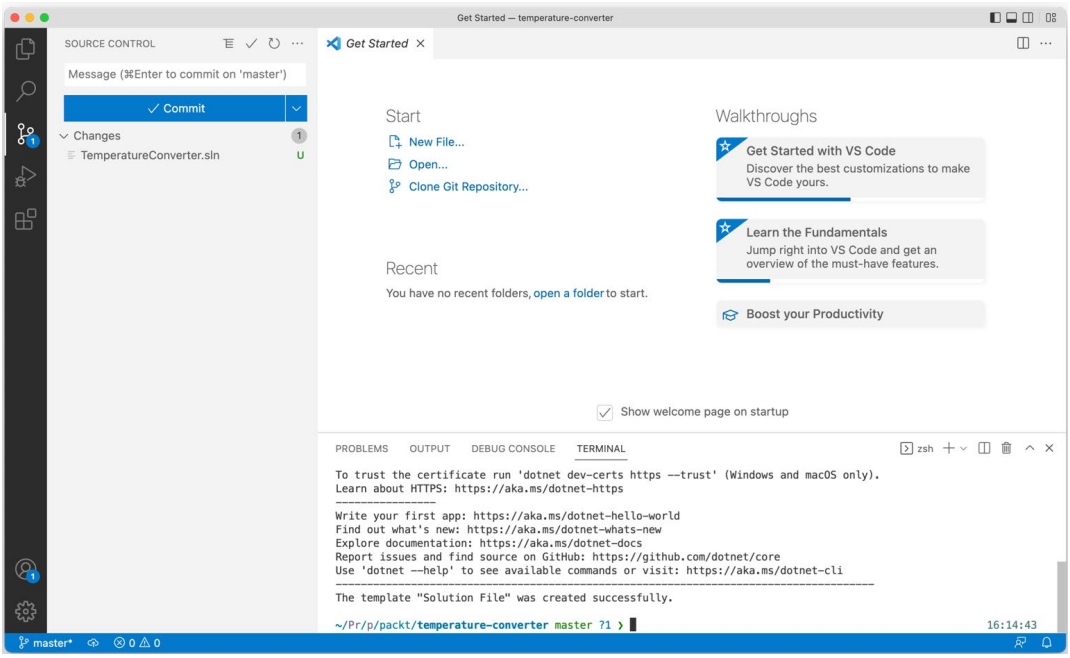

Figure 11.5: Staging a file with VS Code

After staging the file with the previous command, you will now see that the file will be displayed under a new tab called Staged Changes. A **staged file** is a file in which changes are tracked by Git:

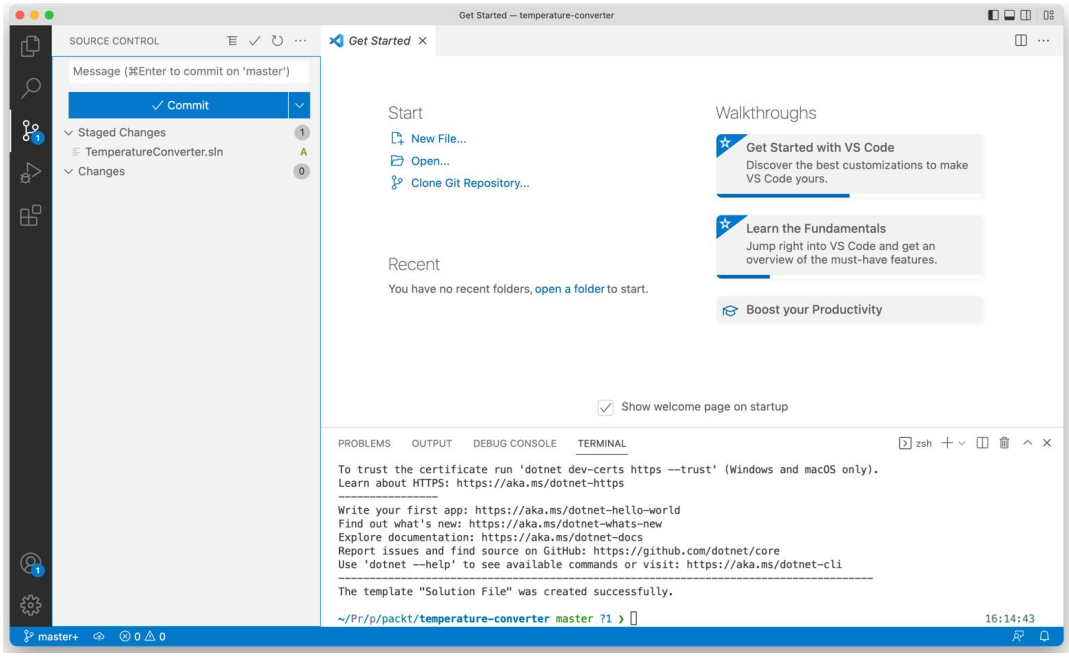

Figure 11.6: Staged file with VS Code

Once you have this file in this new section, you will commit the first change. A **commit** is like a snapshot of a certain point of time that you can navigate to inside your repository. Each commit has a commit message to identify it.

9. So, in VS Code, type **`First Commit`** in the message field.

10. Then click on the **`Commit`** button with the checkmark icon located under the message field to commit the file, as shown in the following figure:

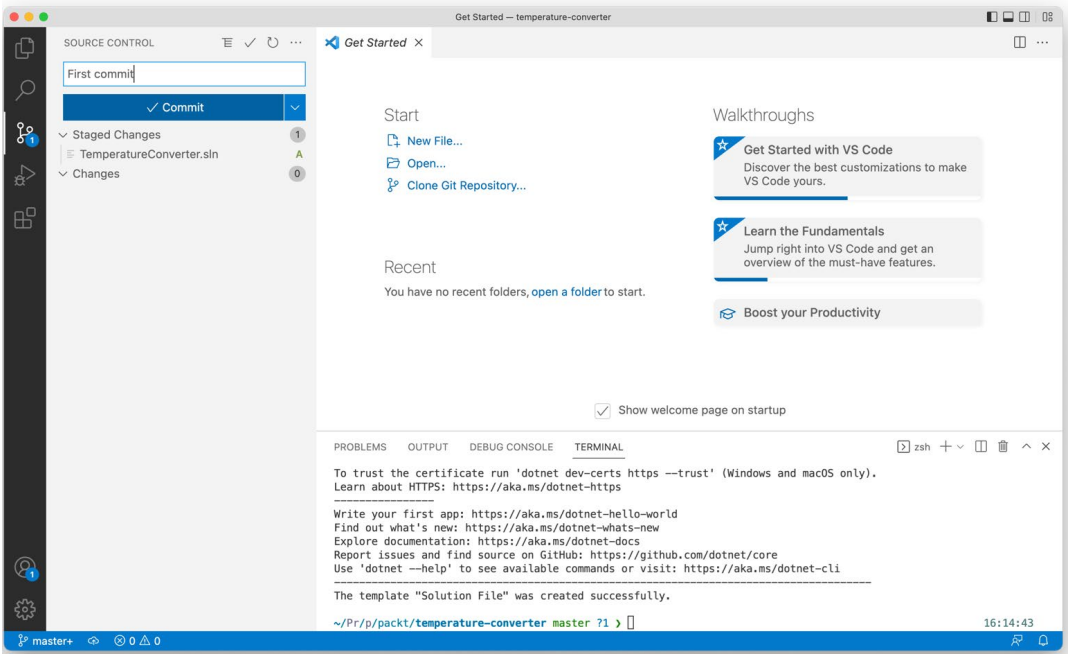

Figure 11.7: First commit with VS Code

11. Finally, go to the integrated terminal and type the following command:

```
git log
```

You should see your commit, along with some extra information, as follows:

```
commit b67a22604275fd88d7bb138fc0303b0c462dc563 (HEAD -> master)
Author: Mateus Viegas <mateus.viegas@luxclusif.com>
Date:   Fri Jul 29 16:16:17 2022 -0300

    First commit
(END)
```

> **NOTE**
>
> You can find the code used for this exercise at https://packt.link/OlNYV.

This section showed you the basic flow of files inside Git. The untracked files are the ones that Git knows that it might track if prompted by the user to do so. However, there are some files that you might never want to track, such as files containing secrets and database passwords (for obvious security-related reasons) or files that contain build outputs, such as DLLs.

In order to prevent the tracking of sensitive data, such as passwords and build outputs, Git has a special type of file named **`.gitignore`**. It is a simple text file that Git reads to determine whether there are any files at the repository level that shouldn't even be in the **untracked** state, avoiding unwanted movement of sensitive files during repository commits.

You saw how VS Code provides an easy way to some basic Git commands to initialize a repository, stage files to Git, and commit changes. In the next section, you will see how the same result can be achieved using CLI commands.

COMMON CLI COMMANDS

If you prefer to use the CLI rather than the VS Code interface for creating a file and committing it to the repository to start this tracking, read through this section.

To initialize a new Git repo (like *Step 5* of *Exercise 11.01*), use the following command in the integrated terminal:

```
git init
```

To make Git track the file, use the following command (like *Step 7* of *Exercise 11.01*):

```
git add TemperatureConverter.sln
```

To add a file inside a folder in the integrated terminal, simply type:

```
git add NAME_OF_FILE
```

Here, you can also replace the **`NAME_OF_FILE`** with a dot (**`.`**) to specify all files inside the current folder of the terminal, or with a full directory/ file path.

Similarly, to remove a file from staging use the following command:

```
git restore –staged NAME_OF_FILE
```

You can also use a dot (`.`) to denote all files in the current folder or specify the full directory/ file path, as applicable.

To remove all files in the current folder from staging, use the following command:

```
git restore –staged
```

To commit the first change, the following command would have the same effect as seen in *Step 9* of *Exercise 11.01*:

```
git commit FILENAME -m 'First Commit'
```

The **`-m`** flag in the preceding command denotes the commit message. Here, you could use the filename that you want to commit.

If you want to commit all staged files, type in the following command:

```
git commit -am 'Commit Message'
```

Note the usage of the **`-am`** flag, which indicates all files being committed.

To commit all staged files inside a specific path, use the following command:

```
git commit /some_path/* -m 'Commit Message'
```

You now learned about some basic Git commands to initialize a repository, stage files to Git, and commit changes using common CLI commands. VS Code has a useful extension that helps you to set up **`.gitignore`** files according to the kind of project/ programming language that you are using: https://marketplace.visualstudio.com/items?itemName=codezombiech.gitignore. Install this extension before proceeding with *Exercise 11.02*.

EXERCISE 11.02: IGNORING FILES DURING COMMITS

In this exercise, you will stage and commit the **`.gitignore`** file to the repository you created in the previous exercise. Perform the following steps to do so:

1. Open the extensions tab and search for the **`codezombiech.gitignore`** extension.
2. Click on the **`Install`** button to execute the extension installation:

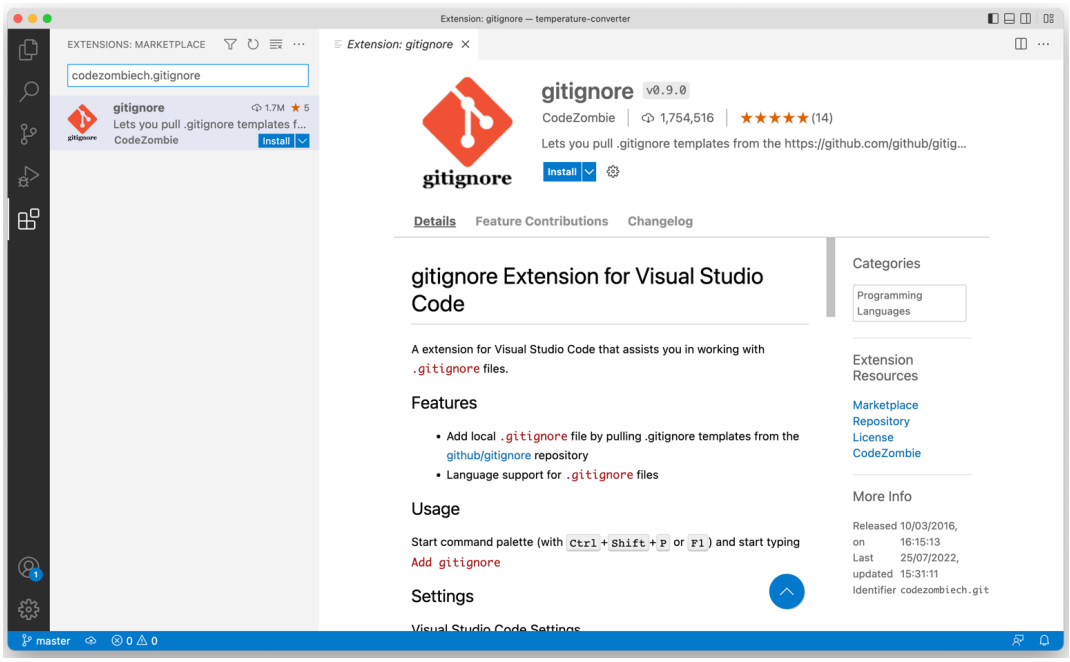

Figure 11.8: Installing the codezombiech.gitignore extension

3. Open the VS Code **`Command Palette`**.
4. Type **`Add gitignore`** and execute the command:

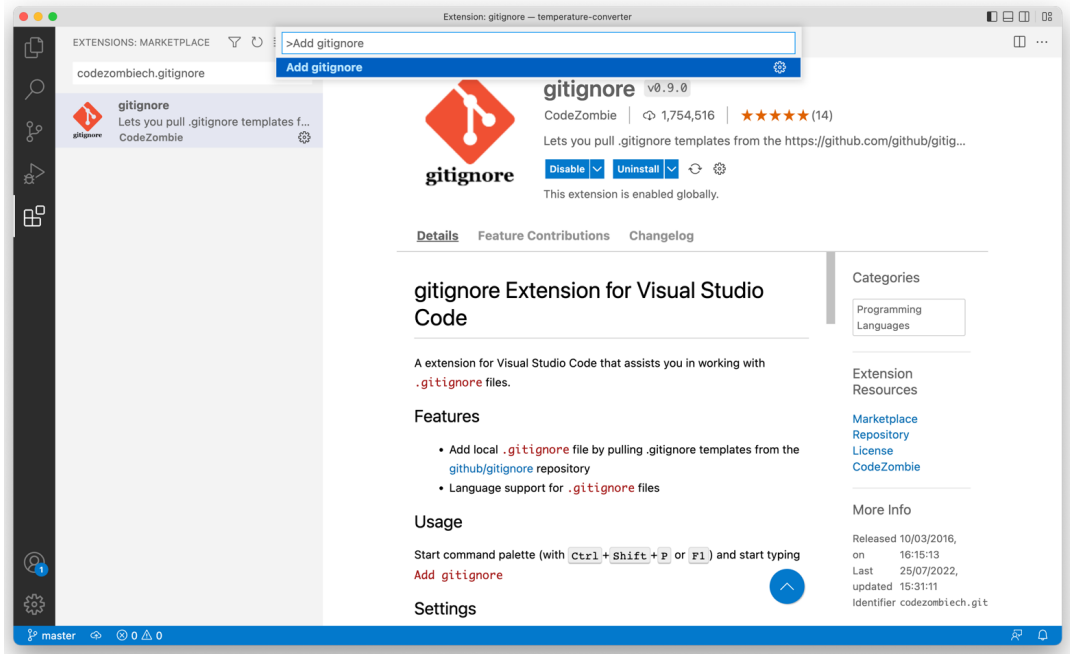

Figure 11.9: Adding a .gitignore file

5. Now, you will be prompted to choose a type. So, choose the **`VisualStudio gitignore`** option as it contains the default ignore files for a C# solution/project:

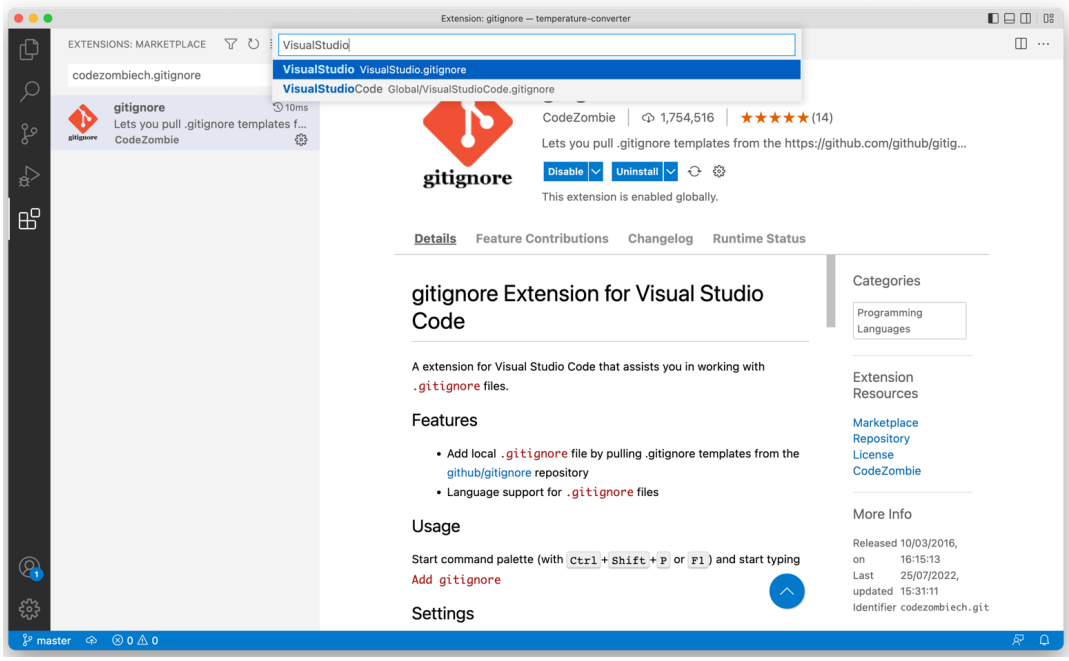

Figure 11.10: Choosing the .gitignore template

The **`VisualStudio gitignore`** option gives you a file that ignores all C# default objects, such as build outputs.

> **NOTE**
>
> Do not choose Visual Studio Code.

6. Now go to the **`Source Control`** tab.
7. Add and stage **`.gitignore`** and create a commit with the **`Added gitignore`** message:

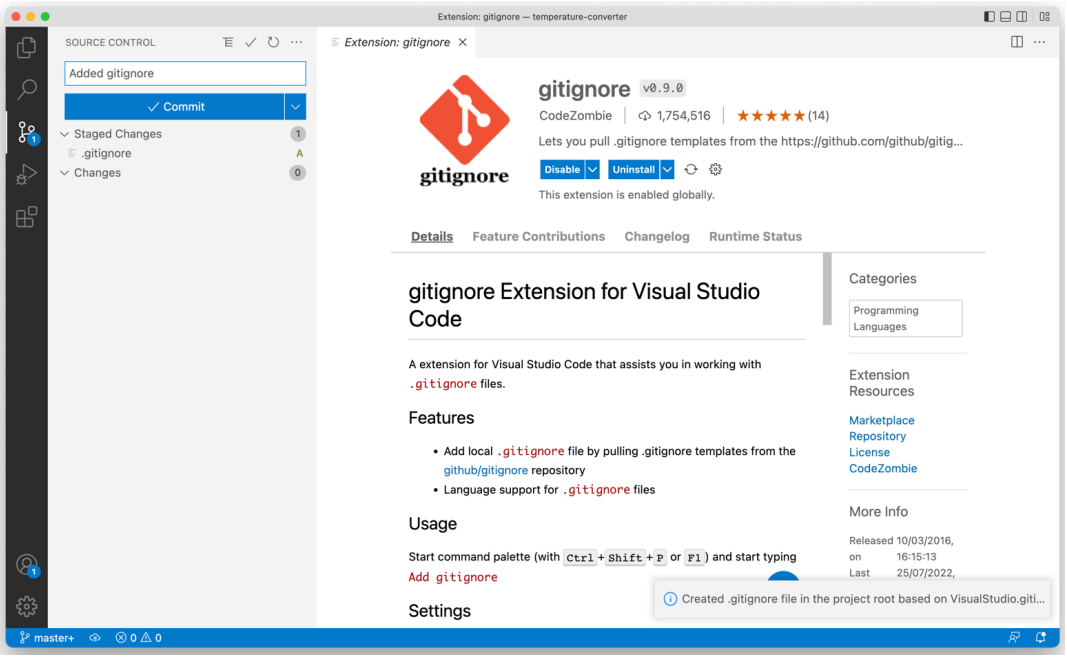

Figure 11.11: Committing the .gitignore file

8. On the integrated terminal, enter the following command:

```
git log
```

You will see an output as follows:

```
commit 43c285d274337916e072795085faa7fbf9d0bfe1 (HEAD -> master)
Author: Mateus Viegas <mateus.viegas@luxclusif.com>
Date:   Fri Jul 29 16:26:30 2022 -0300

    Added gitignore
```

```
commit b67a22604275fd88d7bb138fc0303b0c462dc563
Author: Mateus Viegas <mateus.viegas@luxclusif.com>
Date:   Fri Jul 29 16:16:17 2022 -0300

    First commit
```

With this exercise, you have staged and committed the .gitignore file to the repository you created.

> **NOTE**
>
> You can find the code used for this exercise at https://packt.link/bUaK9.

An important thing to note here is that you could add as many files as you want to the **`.gitignore`** file. It doesn't have to be autogenerated as shown in *Step 2* of this exercise. If you want to ignore an already committed file, use the following command:

```
git rm –cached FILENAME
```

If you want to commit a previously ignored file, perform the following steps:

1. Remove the filename from the *.gitignore* file.
2. Use the following command to forcefully add the file to the staging:

```
git add –force FILENAME
```

3. Use the standard command to create a commit with it:

```
git commit
```

With this section, you learned how to create a new Git repository and stage and commit the **`.gitignore`** file to the repository you created. Now proceed to know about a visual representation of the commit.

COMMITS AS GRAPHS

Another useful VS Code extension that can come in handy is Git Graph. It provides a visual representation of the commits inside a Git repository. Perform the following steps to see this in action:

1. Install the **`Git Graph`** extension on your system by clicking the **`Install`** button:

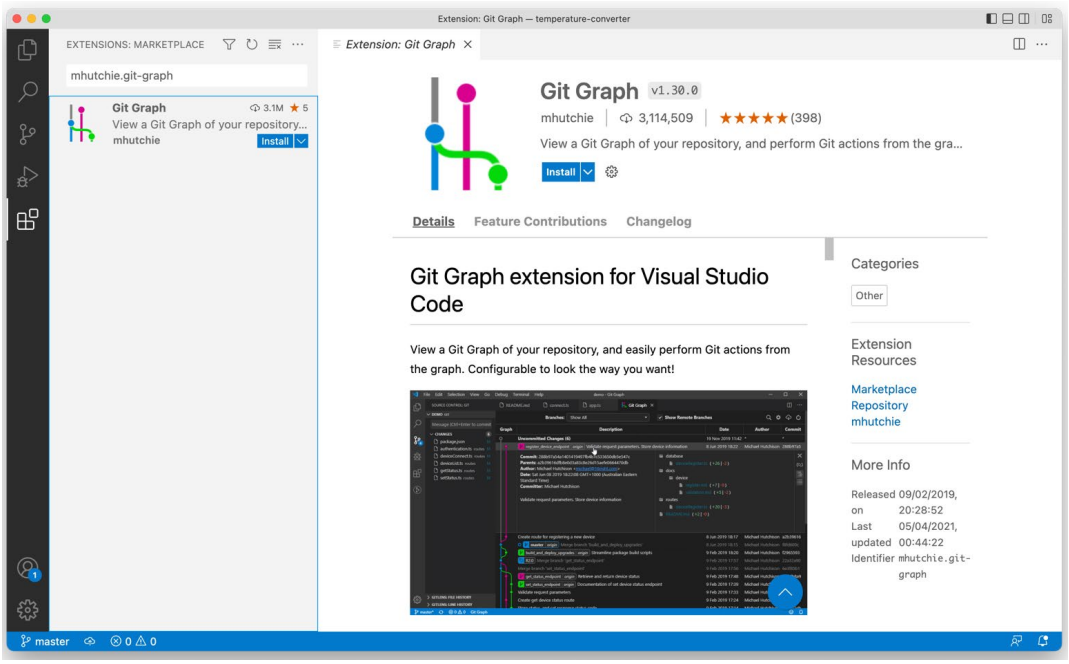

Figure 11.12: Installing the Git Graph extension

> **NOTE**
>
> You can find the installation link for Git Graph for VS Code at https://marketplace.visualstudio.com/items?itemName=mhutchie.git-graph.

2. Once you have installed Git Graph, go to the **`Source Control`** tab of the Git Graph.

3. Then click on the **`Git Graph`** icon at the top of the panel. You will see something similar to what is seen in *Figure 11.13*:

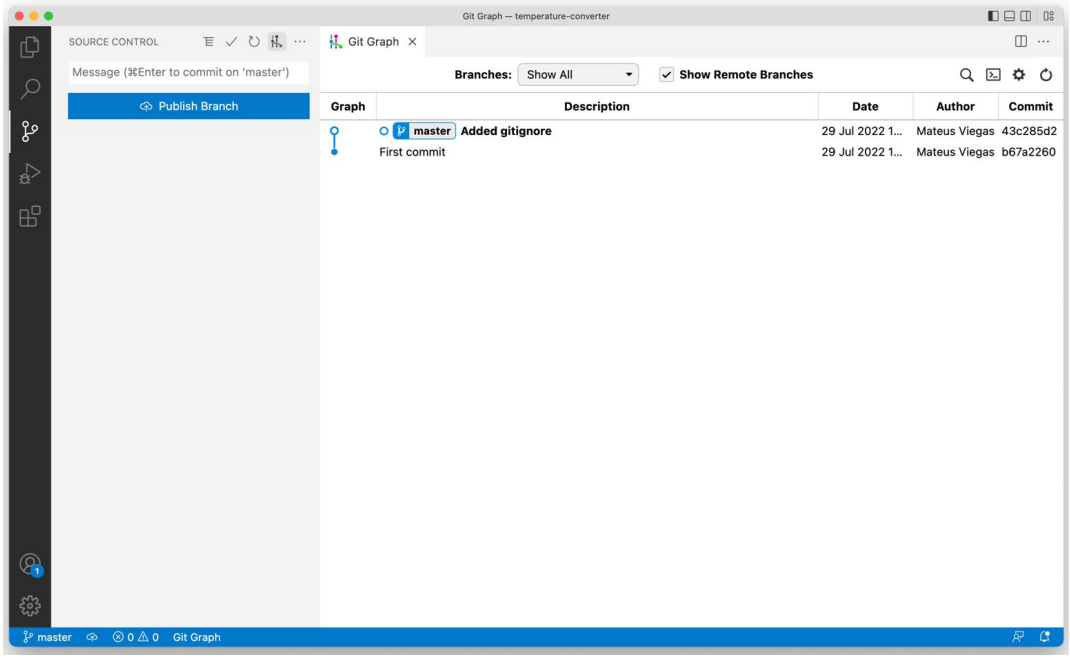

Figure 11.13: Git Graph extension showing commit history

The way that Git stores commits is by placing them as edges on a graph. This is what allows you to see the relationship between commits and navigate along them throughout their entire history.

In the preceding screenshot, you see one word that shows up frequently—**Branches**. A branch is simply a pointer to a specific commit. More practically, you can think of branches as part of a repository history. When a branch points to a commit, you can navigate from this commit all the way back inside this branch. That happens because, as you have seen, Git stores commits in a graph form that he has its edges connected. This will become clearer in the following exercises, as you work with multiple branches.

EXERCISE 11.03: CREATING AND MERGING NEW BRANCHES

In this exercise, you will create a new branch separate from your main one to create the project structure. You will learn about branching, how to create new branches, and how to switch between them using an important command called **`checkout`**.

Perform the following steps to complete this exercise:

1. In the integrated terminal, enter the following command:

```
git checkout -b feature/project-structure
```

   Here, the **`checkout`** command allows you to switch between branches. When you use the **`-b`** flag before the branch name, you create a new branch with the name provided (in this case it is **`feature/project-structure`**), instead of simply switching branches. It is important to note that Git does not allow the creation of a new branch with an already existing name.

   The console will output the following message:

```
Switched to a new branch 'feature/project-structure'
```

2. Now create your solution projects (a web API project as well as a test project). For this, run the following commands on the terminal:

```
dotnet new webapi -n API
dotnet new xunit -n API.Tests
dotnet sln add API
dotnet sln add API.Tests
cd API.Tests
dotnet add reference ../API
```

   With the projects created, stage all the changes as shown in *Exercise 11.01*:

   - With the VS Code Source Control tab (*Step 7* in *Exercise 11.01*).
   - Or by using the **`git add`** command on the terminal, as displayed in *Common CLI Commands* section.

3. Commit these files from the VS Code Source Control tab (*Step 9* in *Exercise 11.01*) or by entering the following command:

```
git commit -am  'Created API and test project and added to solution'
```

   To push the changes of a new branch to the master branch, you must first perform a checkout to go back to the master branch, and then use the merge command to bring these changes to the current branch.

4. So, first return to the master branch by writing the following command on the terminal:

```
git checkout master
```

5. Then, enter the following command to merge the changes from the feature branch to the master branch:

```
git merge feature/project-structure
```

Now check the Git Graph. You will have two branches that point to the same commit:

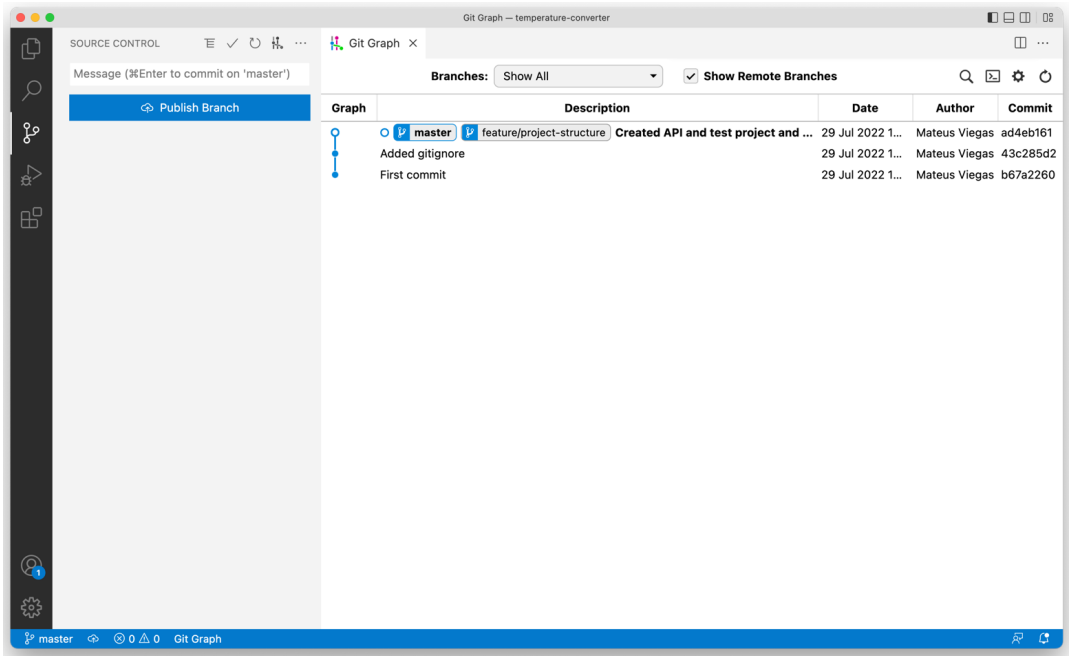

Figure 11.14: Git Graph extension showing commit history

> **NOTE**
>
> You can find the code used for this exercise at https://packt.link/JPNyR.

Branches allow you to create modifications to a project without affecting the main tree of the graph. Another utility of branches comes when you are working in teams and many people are collaborating on the same branch. This allows every member of the team to do their work without interrupting others and simply merge the changes when the work is done. In this way, everyone can have their work synchronized.

By now, you've learned a lot about Git and how it works as a VCS. You have probably heard about GitHub as well. The next section and the exercise will highlight how they can work together to enhance productivity.

GITHUB—A GIT HOSTING REPO

While Git is a VCS itself, GitHub is a Git hosting repository. It has built-in tools that are linked to Git data (such as commits) with a focus on improving the development process for both individuals and collaborating teams. But first, you must have an account on GitHub.

Perform the following steps to create one:

1. Navigate to https://packt.link/EDLC6.
2. Then click on the **`Sign-up`** option.
3. Fill in the relevant details. You should receive an email confirming your account, at which point you're ready to go.

Now you will move to the following exercise, where you will learn how to link a local Git repository to GitHub.

EXERCISE 11.04: ADDING A REMOTE GITHUB REPOSITORY TO YOUR LOCAL GIT

In this exercise, you will push your changes by adding a **remote** Git repository linked to your local Git. That way, you can synchronize everything that's local to be online, have an online backup, and be able to collaborate with others on the project, if required.

Specifically, you will be using three commands to complete this exercise:

- **`fetch`**: It identifies changes made on the remote repository without retrieving the files with the changes.
- **`pull`**: It is used to retrieve the remote changes.
- **`push`**: It is used to push your local changes to the remote repository.

Perform the following steps to complete this exercise:

1. After creating your GitHub account, log into the account.
2. Once you have logged in, look to the top left of the black navbar. You will see a plus (+) icon.
3. Click the Plus (**+**) icon, and you will see the **`New repository`** option.

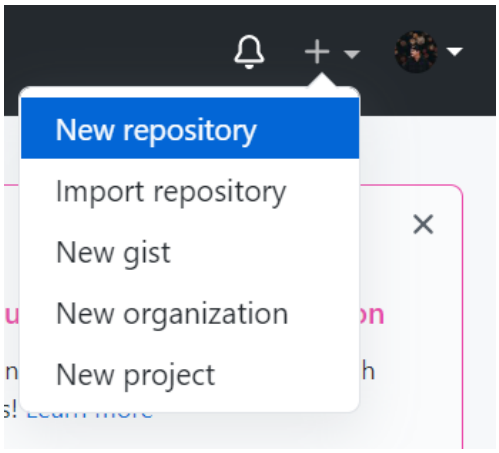

Figure 11.15: Choosing the New repository option in GitHub

4. Click on the **`New repository`** option. Right after that, you'll see a screen similar to *Figure 11.16*.
5. Leave the **`Add a README file`** checkbox unchecked.
6. Choose **`None`** under the **`Add .gitignore`** option.
7. Select None for the **`Choose a license`** option.

8. Finally, click on the **`Create repository`** button:

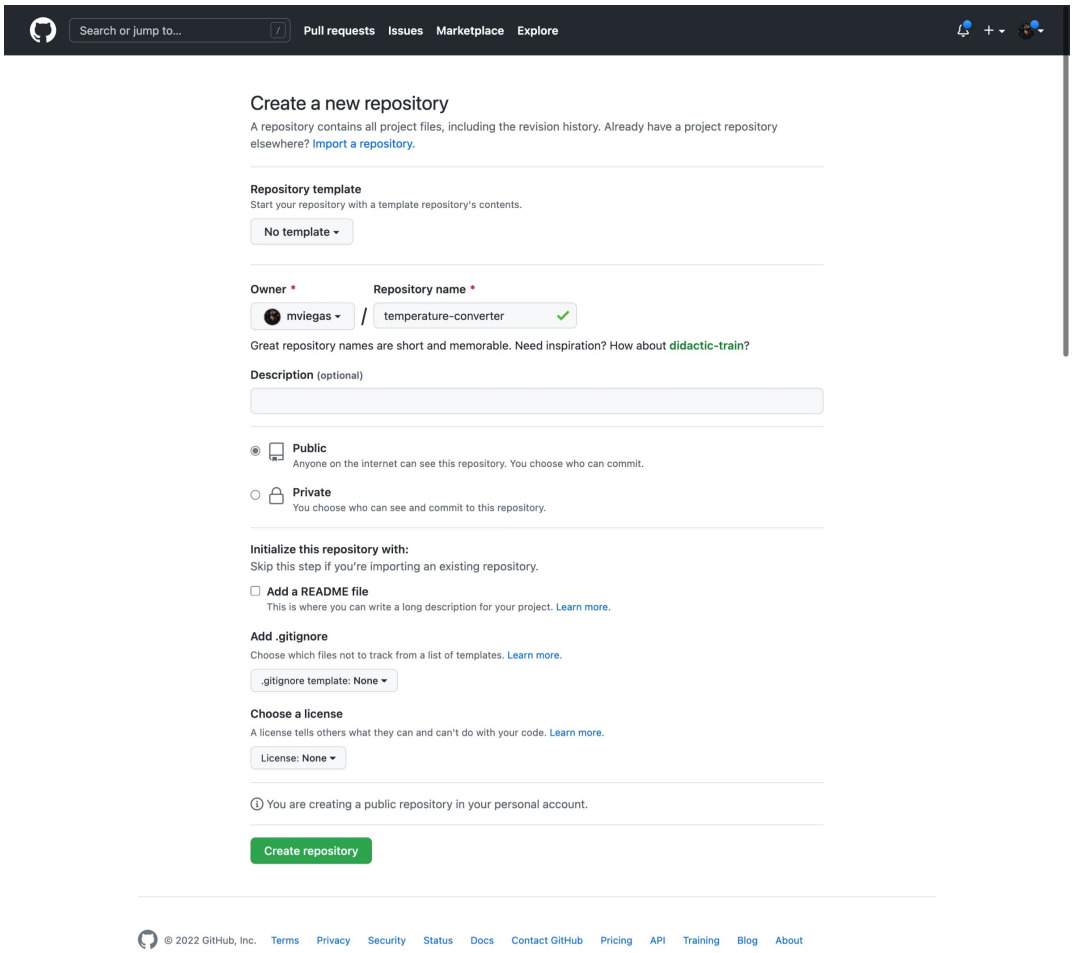

Figure 11.16: Creating a new repository on GitHub

You have the option of creating a new repository on the command line or pushing an existing one as shown in the following figure:

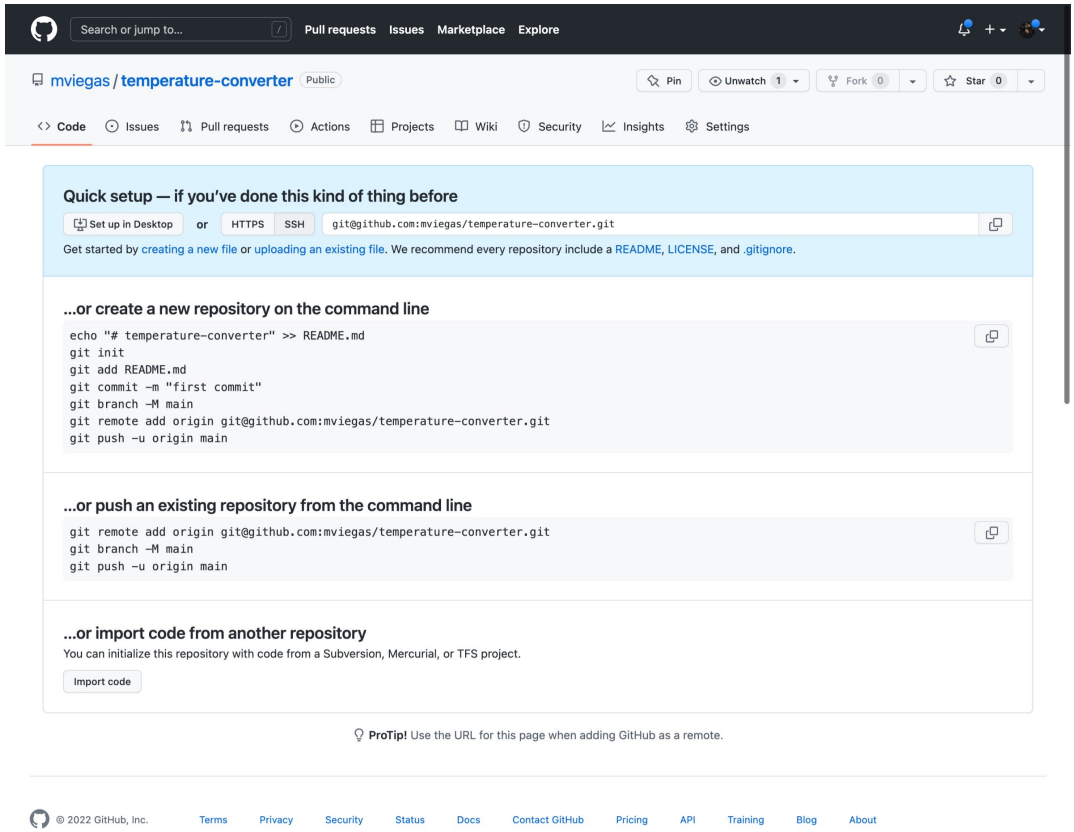

Figure 11.17: GitHub showing option for creating new or pushing existing repo

9. Push an existing repository by entering the following commands on the VS Code integrated terminal:

```
git remote add origin https://github.com/<YOUR_USERNAME>/temperature-converter.git
git branch -M main
git push -u origin main
```

In the preceding snippet, there are a few new commands you need to be familiar with:

- The **`remote`** is a version of your repository hosted somewhere on the internet.
- The **`git remote add origin <PATH_TO_REPO>`** command is used to add a reference to a remote repository to your local one.
- Directly following this, you've used the **`git branch -M main`** command to rename the local branch from **`master`**, which is the name GitHub chooses by default, to **`main`**.
- You've used the **`-u`** flag within the push command to link your local branch to the **`main`** branch on GitHub.
- The **`push`** command is a command that literally pushes your local changes to the remote repository. However, in order to do so, your local branch has to have a remote upstream branch to push these changes. That's why you've used the **`-u`** flag here. This flag sets the branch named **`main`** located at origin as an upstream branch from your current, local branch (**`main`**).
- Finally, remember to replace the **`<YOUR_USERNAME>`** part of the command with your actual GitHub username.

10. Now, check the remote repository by running the following command in the terminal:

```
git remote -v show
```

You should see the following output:

```
 git remote -v show
origin  git@github.com:mviegas/temperature-converter.git (fetch)
origin  git@github.com:mviegas/temperature-converter.git (push)
```

At this point you have created a repository on GitHub and set it as a remote repository to the local one. Now, two more important commands are enabled for you to work with—that is, **`fetch`** and **`pull`**.

11. Open your repo page:

```
https://github.com/<YOUR_USERNAME>/temperature-converter
```

Remember to replace the **`<YOUR_USERNAME>`** tag with your actual GitHub username.

12. Above the file list, click on **`Add file`** button.

13. Then click on the **`Create new file`** button:

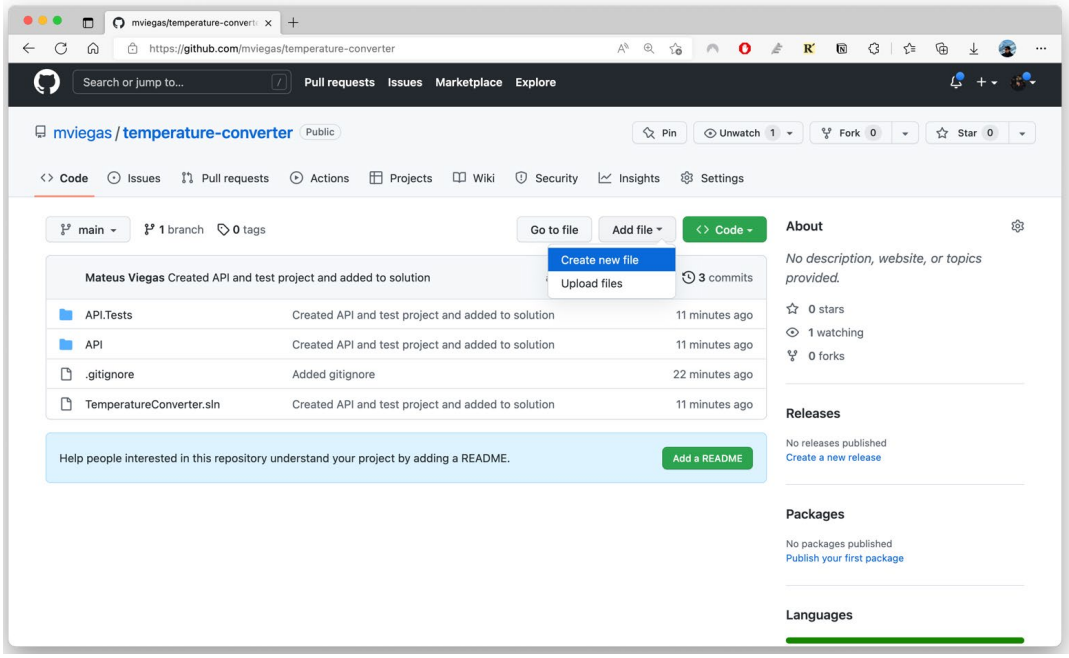

Figure 11.18: Creating a new file via GitHub interface

14. Name the file **`README.md`**.

    This file is commonly used with Git repositories to add useful information about your repository as documentation. It might contain, for instance, build instructions or a description of what your repository is about. It is called a **Markdown** type of file.

15. Now, place the following contents in it:

```
# This is a simple Temperature Converter built for The C# Workshop
```

16. Go to the bottom of the page and click on the **`Commit new file`** option. Leave all the other options with the default values:

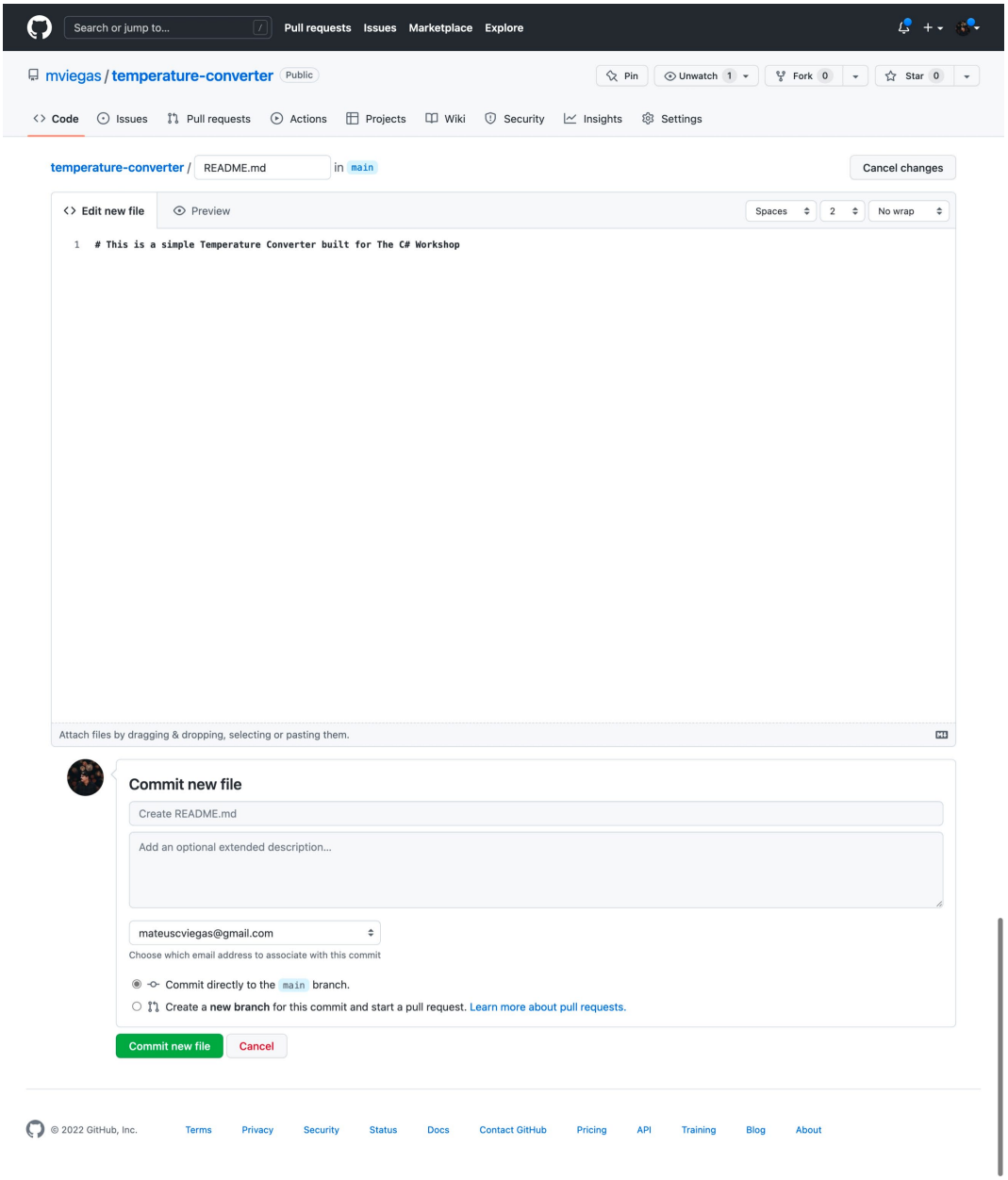

Figure 11.19: Commit of a new file via GitHub interface

17. In the VS Code integrated terminal, run the **`git fetch`** command.

18. Open **`Git Graph`** on the VS Code Source Control tab, and you will see something like the following screen:

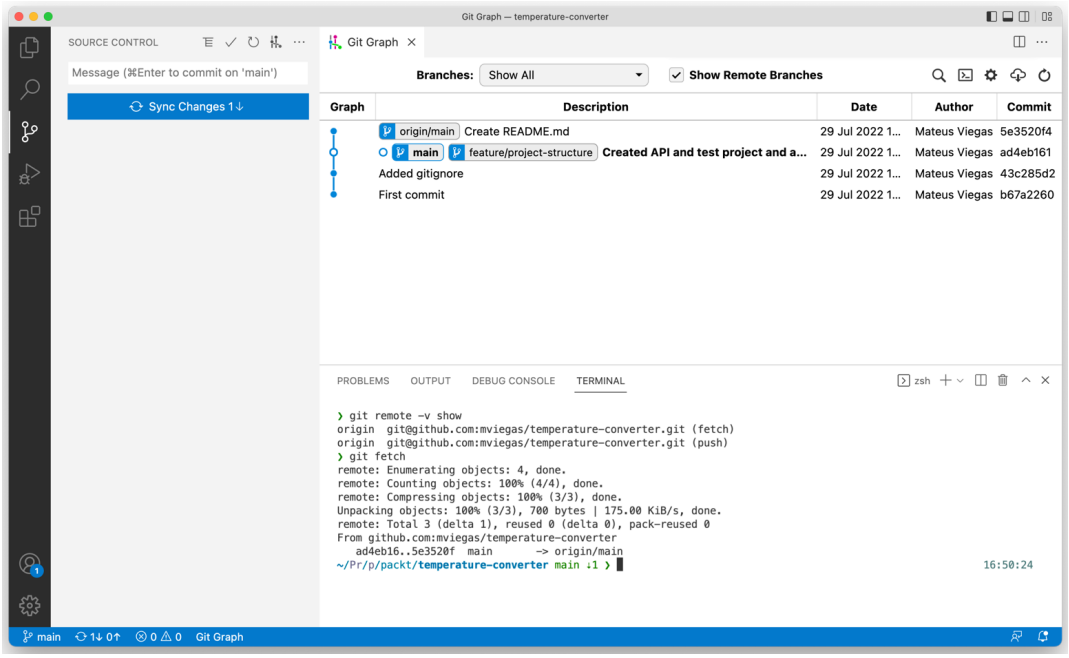

Figure 11.20: Local Git after the fetch command

As you can see, the **`fetch`** command downloads the information about a new commit on your main branch in the remote repository. However, it does not merge the changes to your local main. What **`fetch`** does is a **harmless** download, as it brings the remote information but does not manipulate anything locally on your branches.

19. To update your current branches with remote changes, you should use the **`pull`** command. So, type the following command on the integrated terminal:

```
git pull
```

Your Git Graph should now look like this:

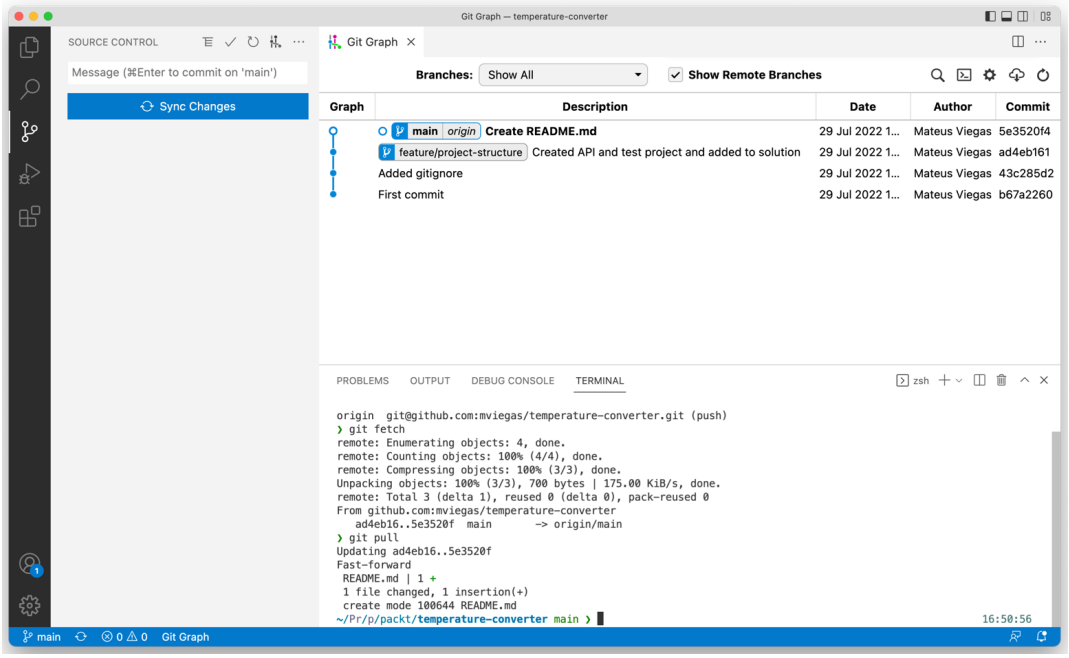

Figure 11.21: Git Graph after a pull command retrieving remote changes

The **`pull`** command also downloads the information, like **`fetch`**, but it also merges any changes on your remote repository to your local one.

20. To see what the **`push`** command does, first open the **`README.md`** file in VS Code and replace it with the following content:

```
# This is a simple Temperature Converter built for The C# Workshop

## Built with .NET 6.0 by YOUR NAME HERE
```

21. Now commit your file with the following command:

```
git commit README.md -m 'Updated README'
```

Your Git Graph will look as follows:

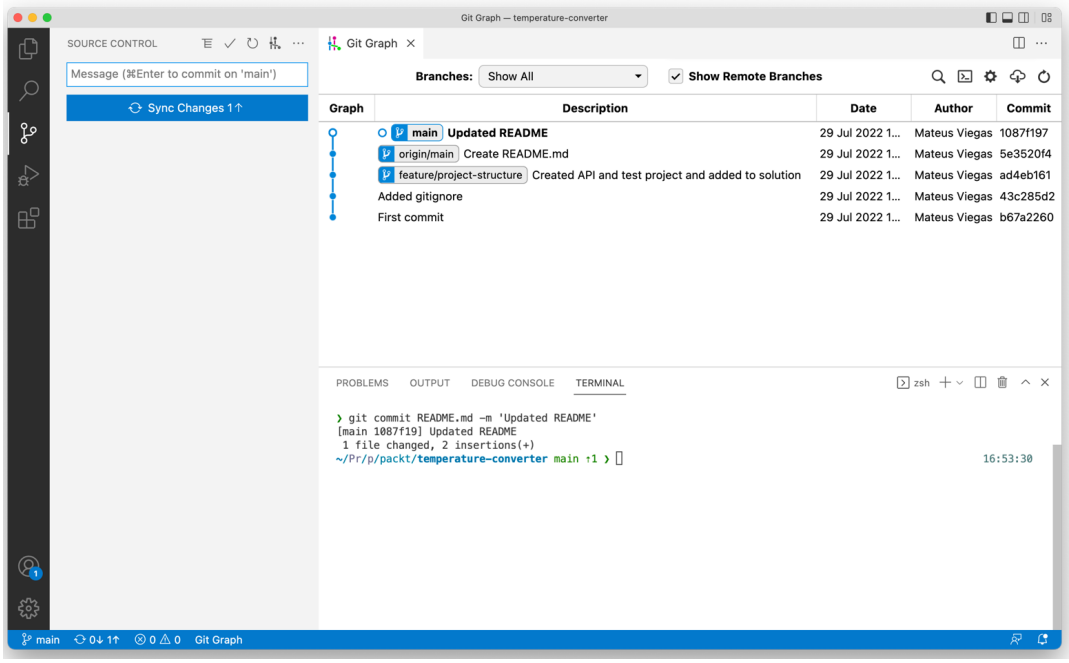

Figure 11.22: Local Git after committing before push

22. The new commit is only local, and you need to push it to the remote repository. To do this, run the following command:

```
git push
```

In the Git, the head branch is part of the **`git`** command output and is simply a reference to a commit. The reference to the current commit within the current branch is aliased as **HEAD** with uppercase letters. After the last command, your Git Graph will show that both repositories (**`main`** and **`origin/main`**) are synchronized with their HEAD at the same commit, as shown in *Figure 11.23*:

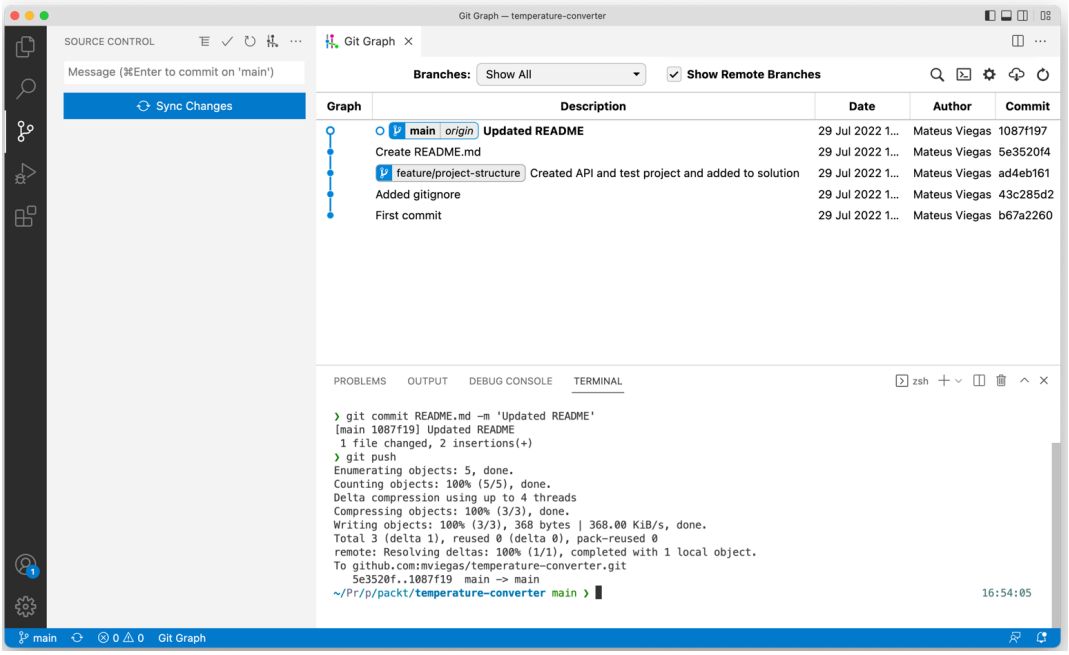

Figure 11.23: Local Git after commit and after push

In this exercise, you synchronized your local changes with the remote ones when a remote repository was connected to your local branch. You also used the Git **`fetch`**, **`pull`**, and **`push`** commands, which are a must-have in your Git toolbox as a developer.

> **NOTE**
>
> GitHub does not carry *Exercise 11.04*, as this exercise simply creates a **remote.remote branch**.

GitHub is one of the many options available for remote Git hosting. There are also some other websites, such as Bitbucket, GitLab, and Azure DevOps. Each of these provides a set of tools as a product that helps developers increase productivity while working with Git. Which one to choose is a matter of evaluating necessities and trade-offs.

> **NOTE**
>
> A full-fledged discussion on which option to choose for remote Git hosting is beyond the scope of this chapter, but those of you who are interested can refer to https://sourceforge.net/software/compare/Azure-Repos-vs-Bitbucket-vs-GitLab/ for more information.

The next section will focus on teaching you how to continuously integrate and deploy your code to a production environment using Git.

ENSURING QUALITY WITH CI

Practicing **Continuous Integration** (popularly abbreviated as **CI**) consists of making new changes, which could be either adding new features or bug fixes, in already existing software. As you have seen with Git, it all starts with a checkout from a branch. From there, a developer can start building new changes into the code. These changes can be integrated into the VCS mainline by either creating commits directly or merging a new branch into the mainline. An advantage of creating new branches and merging these branches after successful builds is that the mainline is protected from broken commits.

An important pillar of CI is **automated builds**. That means that the moment a commit touches a branch, a build runs. **Running a build** means compiling the code and, if this code has tests, also running the automated tests. There are several possibilities within CI pipelines. Nowadays, CI providers have options to run integration tests against real databases, for instance. Each system may have its own requirements regarding the type of builds and tests that it will run, and which components it might require.

The advantages of CI are as follows:

- It maintains the integrity of the mainline, as the **automated builds** must run successfully for the changes to be integrated. It allows the whole team to have a safe starting point to work.
- It enables a second step named **Continuous Deployment** (**CD**), which means continuously pushing changes to production. You will be covering this concept in detail in the upcoming *Continuous Deployment* section.

GITHUB ACTIONS

Automated builds are a central piece for implementing CI. One of the most useful features that GitHub delivers is GitHub Actions, which allows you to create automated CI builds with **workflows**, also commonly known as pipelines. It is possible to configure workflows to run on any event triggered by GitHub, such as **`push`**. The next exercise will be a construction of a workflow to automate your application build.

EXERCISE 11.05: CREATING A WORKFLOW FOR CI

In this exercise, you will implement CI. For this, you will first create a workflow that will be executed on GitHub each time a commit touches a branch in your repo. This workflow will build and test your application for every single commit. If everything runs well, you will get a message that your workflow succeeded. If not, you'll get a message that it has failed.

Perform the following steps to complete this exercise:

1. In the root (parent directory) of your repository, create a folder named **`.github`**.
2. Then create another folder inside **`.github`** named **`workflows`**.
3. Inside the **`workflows`** folder, create a file named **`ci.yaml`**. YAML files are files that provide a hierarchical structure and are commonly used for configurations.
4. Next, a workflow must be created inside **`ci.yaml`**. So, first name the workflow **`ci`** and configure it to be triggered after every **`push`** command. For that, add the following content to the file:

```
name: ci
on: [push]
```

Now, you will create a **job** (in other words, a set of steps that run sequentially inside the same isolated environment) for your workflow. Your job will consist of **three** steps:

- Check your Git repository with the application code.
- Build the application.
- Finally, run tests for it.

If every step is executed correctly, then your workflow will run with success.

5. Set your job to run on a Linux machine with the latest version available of Ubuntu. This will be the operating system of a virtual machine hosted in GitHub workflows system that your pipeline will run against.

> **NOTE**
>
> There's no specific requirement here for an operating system, but it might be an effective practice to choose a similar operating system to where you will be hosting your application.

6. Now, complete your file with the following code:

```
name: ci
on: [push]
jobs:
    build:
        runs-on: ubuntu-latest
        steps:
            - name: Checkout git repository
              uses: actions/checkout@v2
            - name: Setup dotnet
              uses: actions/setup-dotnet@v1
              with:
                dotnet-version: '6.0.x'
            - name: Build solution
              run: dotnet build
            - name: Test solution
              run: dotnet test
```

7. In the VS Code integrated terminal, run the following commands to commit the **`workflows`** folder and push it to the repository:

```
git add .
git commit -am 'Created github workflow for CI'
git push
```

8. Now click the **`Actions`** tab on your repository page on GitHub. You should see the following screen:

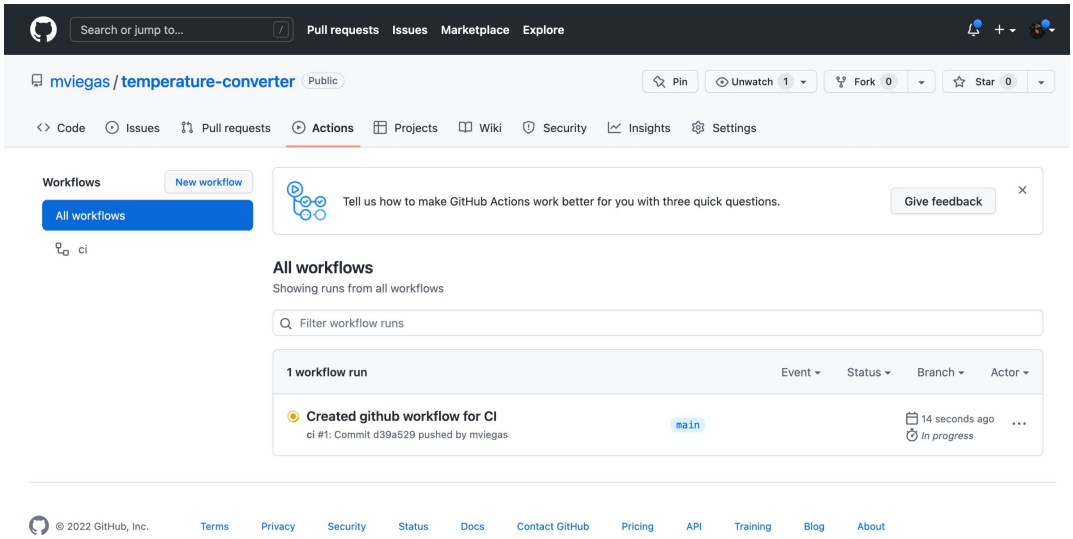

Figure 11.24: CI workflow running on GitHub Actions

9. Click on the **`ci`** workflow, and you'll be able to see the successful job details:

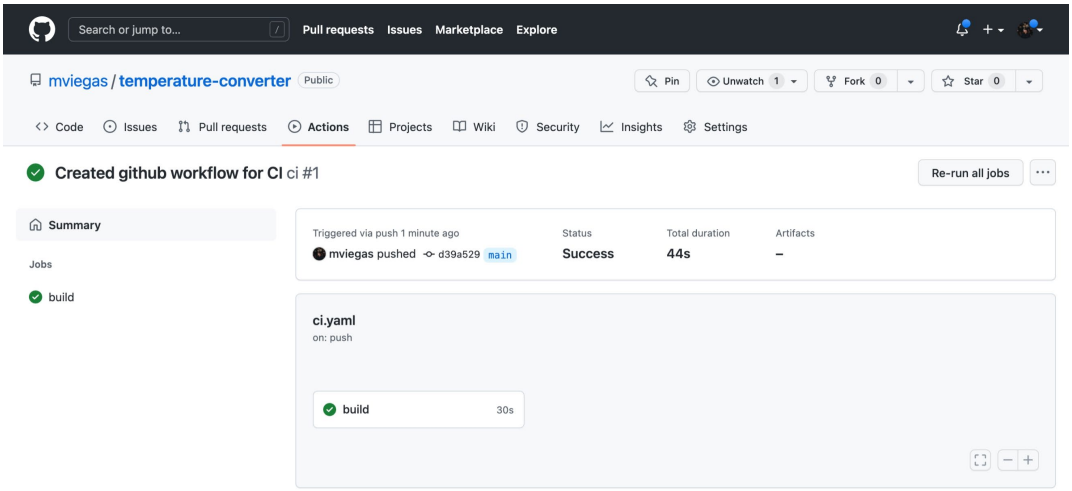

Figure 11.25: CI workflow successful on GitHub Actions

Through this exercise, you implemented CI by first creating a workflow and then executing it on GitHub each time a commit touched a branch in your repo. The workflow was built and run successfully. Note that all the steps that you set up on the workflow file are displayed in *Figure 11.26*:

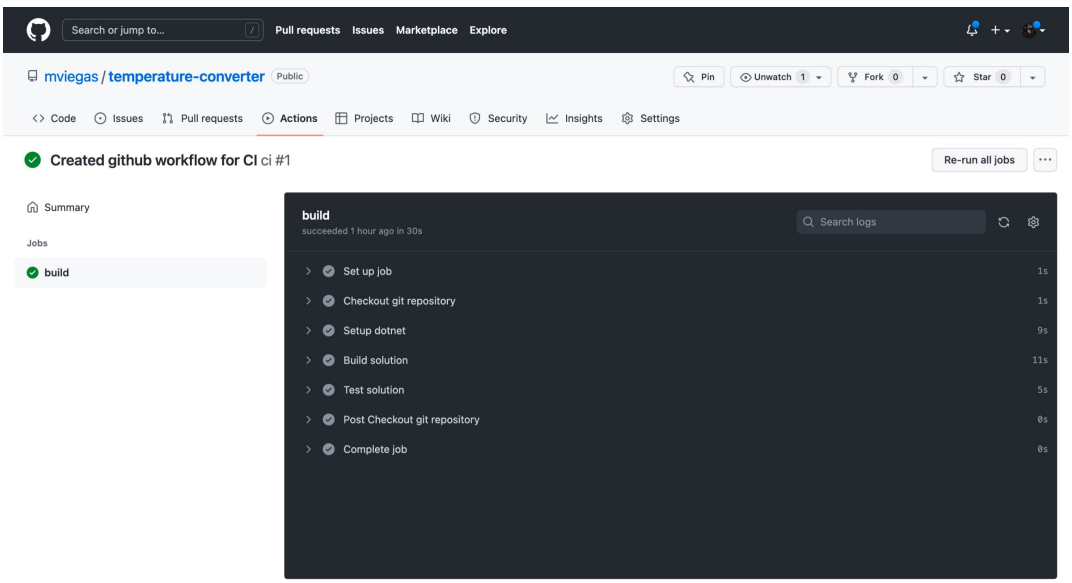

Figure 11.26: Steps of the successful CI workflow

> **NOTE**
>
> You can find the code used for this exercise at https://packt.link/G4sJx.

CI is an important subject with a lot of ground to cover. As you must have noticed, it gives visibility to every change that is done in the repository. If the application is well tested and the CI workflow is well configured (you will also hear the word pipeline a lot), it is an admirable step toward continuously evolving your application with no fear of breaking changes. In the next exercise, you will create an API with CI configured.

EXERCISE 11.06: CONTINUOUSLY INTEGRATING FEATURES

Now that you have the CI configured, in this exercise, you will create the temperature converter API. You will then introduce a new feature to the API and write tests to validate this feature. Finally, you will check the execution of the CI workflow.

The following steps will help you complete this exercise:

1. Create a new branch locally called **`feature/converter-endpoint`** and push this branch to GitHub with the following command:

```
git push -u origin feature/converter-endpoint
```

2. Delete **`WeatherForecastController`** inside the **`Controllers`** folder, which is autogenerated by the .NET CLI when creating a new web API project.
3. Create a new static class called **`Converter`** and implement the following conversions:
   - From Fahrenheit to both Celsius and Kelvin
   - From Celsius to both Fahrenheit and Kelvin
   - From Kelvin to both Fahrenheit and Celsius

   Using these formulas, you should be able to perform all the conversions:
   - From Fahrenheit to Celsius = 5/9 * (F – 32)
   - From Kelvin to Celsius = K – 273
   - From Celsius to Fahrenheit = (C * 1.8) + 32
   - From Kelvin to Fahrenheit = (K – 273.15) * 1.8 + 32

- From Celsius to Kelvin = C + 273
- From Fahrenheit to Kelvin = (F – 32) * 5/9 + 273

The code for this should look as follows:

Converter.cs

```
namespace API;

public class Converter
{
    public static int ToCelsius(ETemperatureUnit @from, double value) => @from
switch
    {
        ETemperatureUnit.Fahrenheit => (int)(5 * (value - 32) / 9),
        ETemperatureUnit.Kelvin => (int)value - 273,
        _ => (int)value,
    };

    public static int ToFahrenheit(ETemperatureUnit @from, double value) => @
from switch
    {
        ETemperatureUnit.Celsius => (int)((1.8 * value) + 32),
        ETemperatureUnit.Kelvin => (int)((value - 273) * 1.8 + 32),
```

You can find the complete code here: https://packt.link/tubjo.

4. Now, create a new file named **ConverterController**.
5. Inside this new controller, place an **HttpGet** method/ action on the converter route that receives two parameters, from and to, which should be a unit such as Kelvin, Fahrenheit, or Celsius, and a value of type **double**. All of these parameters are required, and the action should return an **int** value:

ConverterController.cs

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc;

namespace API.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class ConverterController : ControllerBase
    {
        public ConverterController()
        {
        }

        [HttpGet()]
        public int ToCelsius(
```

You can find the complete code here: https://packt.link/MxdAv.

6. In the test project, create a class called **`ConverterTests`** and place some theories for each type of conversion as shown in *Step 3*.

7. Here, you use some random numbers to specify the matching temperatures in both Fahrenheit and Celsius:

`ConverterTests.cs`

```
using Xunit;
using API;

namespace API.Tests
{
    public class ConverterTests
    {
        [Theory]
        [InlineData(1, -17)]
        [InlineData(50, 10)]
        [InlineData(95, 35)]
        [InlineData(73, 22)]
        public void Should_ReturnProperlyConverted_When_Converting_
FromFahrenheit_ToCelsius(double from, double result)
        {
            Assert.Equal(result, Converter.ToCelsius(ETemperatureUnit.
Fahrenheit, from));
```

You can find the complete code here: https://packt.link/v6e8M.

8. Stage your changes with the following command in the VSCode integrated terminal:

```
git add .
```

9. Now, commit your changes with the following command:

```
git commit -am 'Added converter controller'
```

10. Next push the commit to the remote repository to GitHub:

```
git push
```

11. Navigate to the **`Actions`** tab in the GitHub repository to check the execution of the CI workflow. On success, you will see a screen similar to *Figure 11.27*:

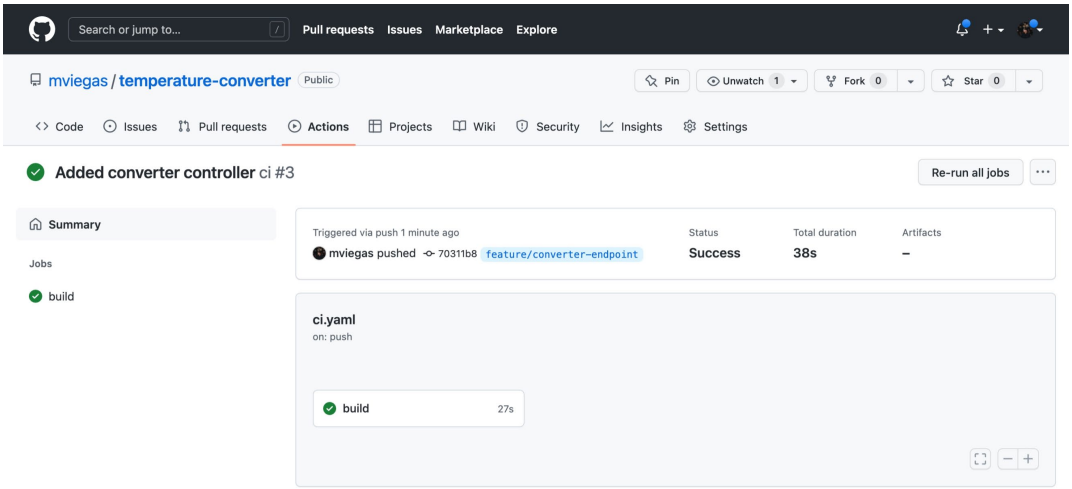

Figure 11.27: CI workflow successful execution

In this exercise, you saw a GitHub action being executed after merging a change you implemented to your application. The success of this execution relied on tests running to validate the new feature. Since all the tests had passed and the build was successful, you can say your changes were continuously integrated.

> **NOTE**
>
> You can find the code used for this exercise at https://packt.link/xm3Q5.

In the next exercise, you will learn to perform pull requests.

EXERCISE 11.07: USING THE PULL REQUESTS FEATURE

Pull requests are another feature from GitHub that allows a developer to review the changes in code before merging this code to the main or any other branch. In this exercise, you will learn how to make use of this feature by integrating changes from the previous activity into the mainline branch.

Perform the following steps to complete this exercise:

1. On your repository page on GitHub, click on the **`Pull requests`** tab.
2. Then click the **`New pull request`** button:

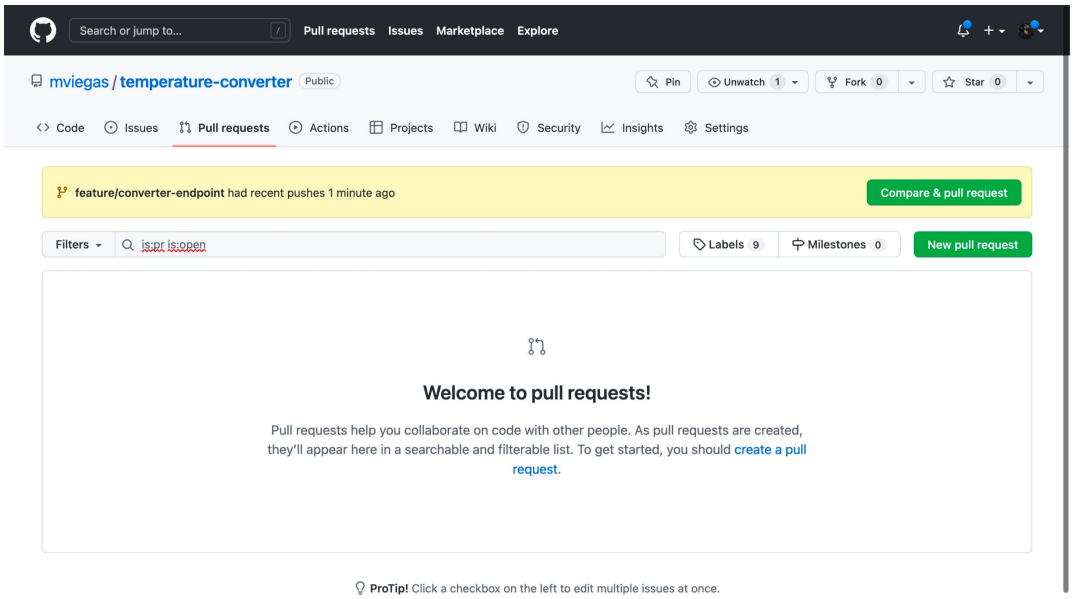

Figure 11.28: Creating a New pull request

3. Set the compare branch as **`feature/converter-endpoint`**. This will be the branch that you are going to compare with your main branch in order to merge the changes.

4. Finally, click on the **`Create pull request`** button:

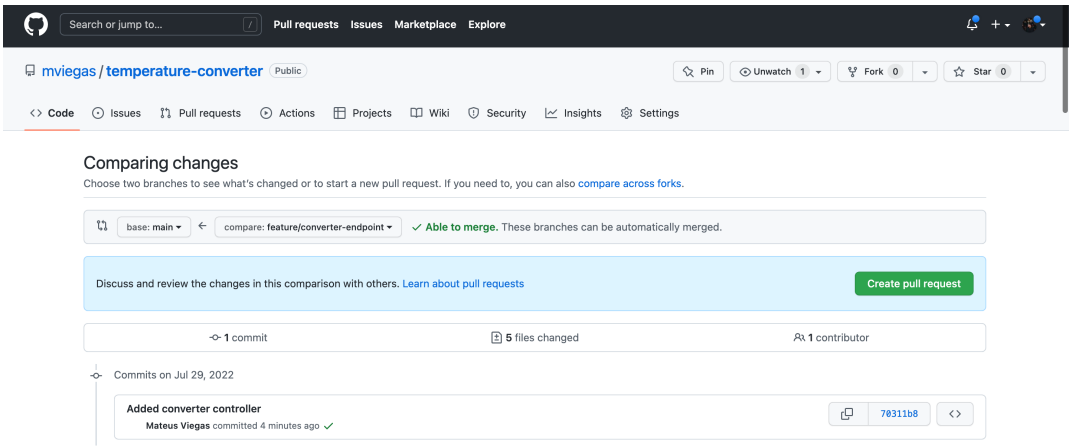

Figure 11.29: Creating a pull request

Here, you can add a lot of details pertinent to the pull request, such as information about reviewers, assignees, and labels.

5. For now, simply add a title and a description to the pull request:

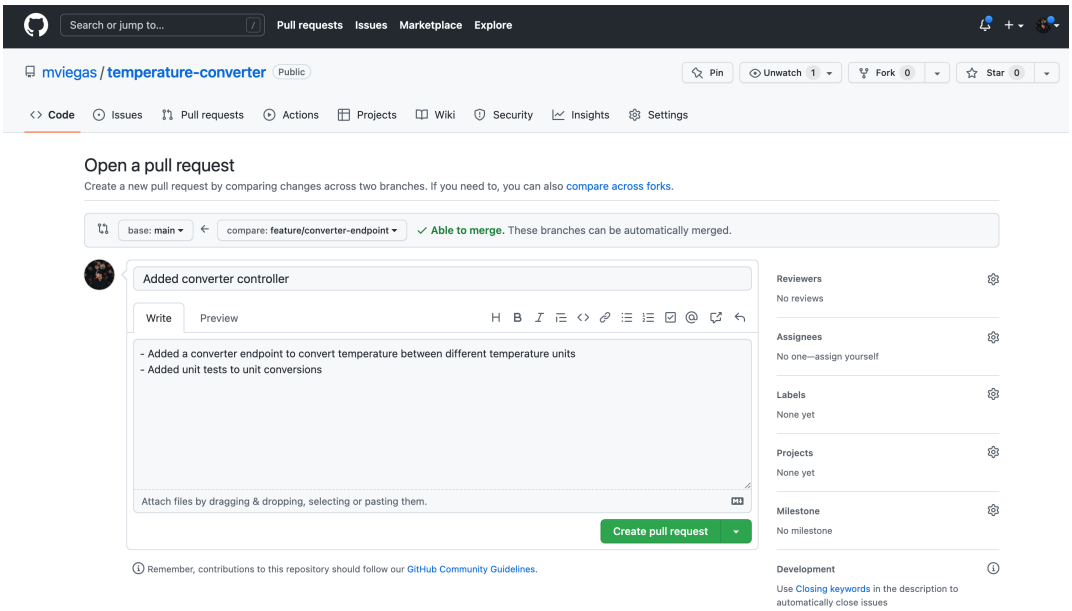

Figure 11.30: Describing a pull request

There might be some guidelines that vary from system to system, but a rule of thumb when working in a team is to be as descriptive as possible in these fields. Try to summarize the changes in the title and describe them in detail in the description field.

Since you have a CI workflow set, the **`pull`** request will show the results of the already run workflow. In this case, the workflow will have been completed successfully, as you can see from the **`Checks`** section.

6. Click on the **`Merge pull request`** button:

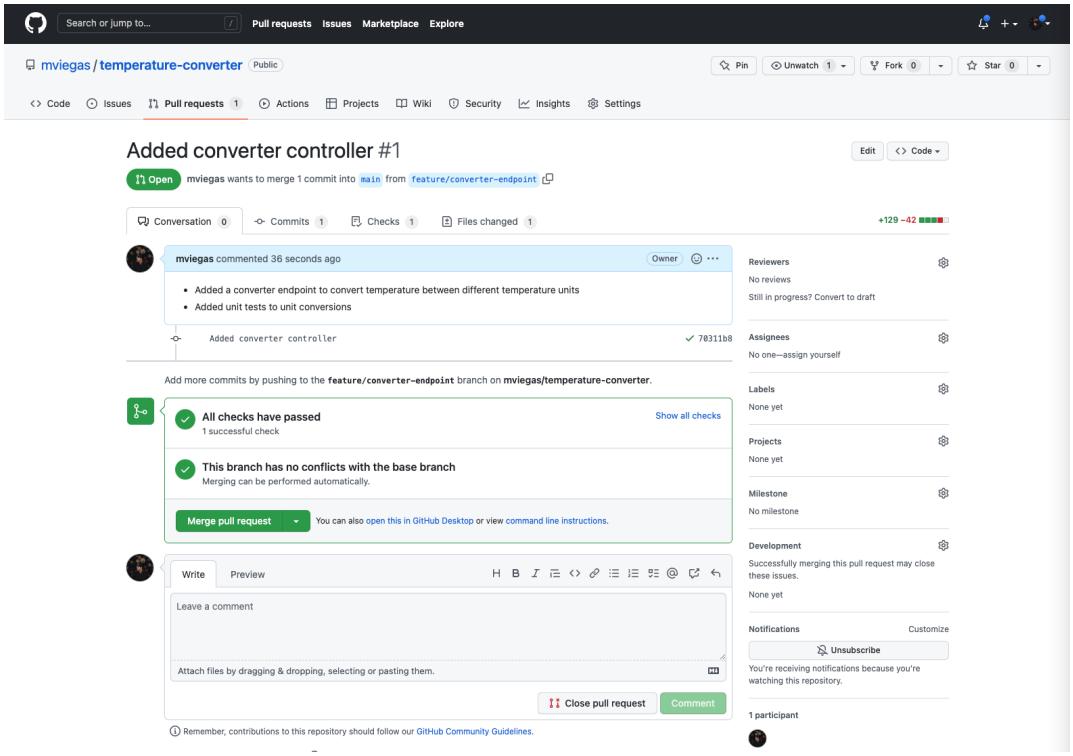

Figure 11.31: Checking a pull request

On clicking the Merge pull request button, the green **`Open`** status will change to a purple one named **`Merged`**:

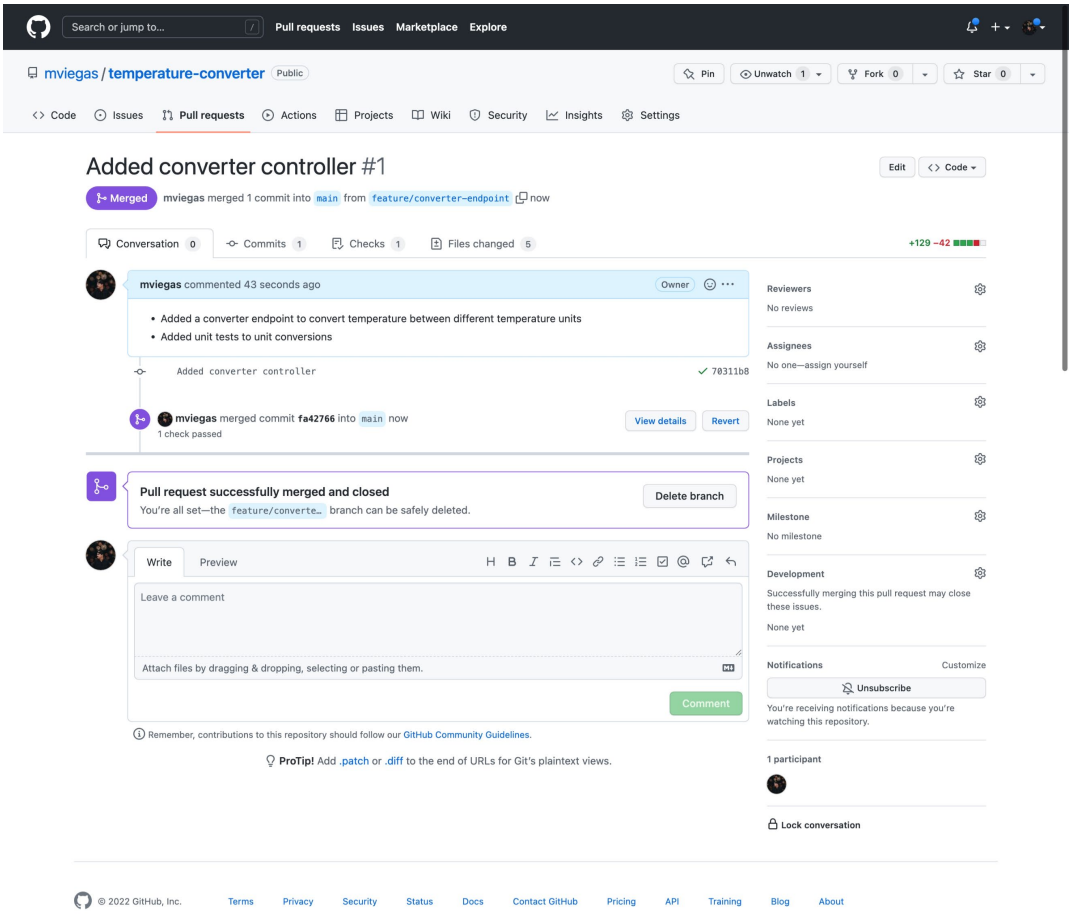

Figure 11.32: Open badge status changing to purple Merged pull request

7. If you run the **`git pull`** locally, you will be able to retrieve the new merged changes:

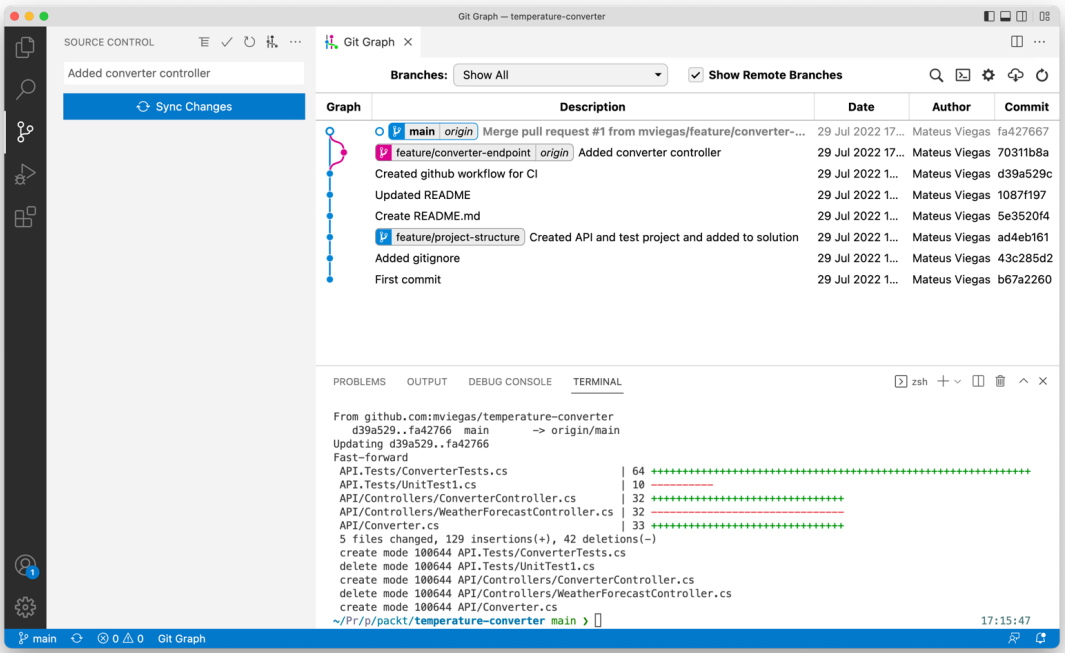

Figure 11.33: Git Graph showing remote and local changes after pull command

In this exercise, you learned how to use the pull request feature in GitHub to merge changes between branches.

Till this point, you have grasped the basics of Git and GitHub and learned how to keep a copy of your repository by properly integrating changes into your code base. Now proceed to the next section and learn how to use these concepts to create a live application in the cloud.

UNVEILING THE CLOUD

Nowadays, there is a lot of buzz around the term cloud, especially regarding web applications. But what is the cloud? A **cloud** is nothing more than a set of data centers with computers managed by some companies, such as Microsoft Azure, and Google, that allows you to quickly deploy applications and infrastructure via a set of services through the internet without having to handle the physical machines themselves. Each of these providers has its own set of services for various types of applications. Choosing which one to use is a matter of a cost-benefit analysis for the issue you are trying to solve.

For this chapter, you will be using Microsoft Azure to host the application you've been building so far, as well as integrating it with your GitHub actions that you created in the last section. There are a lot of other cloud providers, such as AWS, Heroku, DigitalOcean, and Google Cloud Platform. You will use Microsoft Azure here only for the sake of familiarity since you are already within their ecosystem with C# and .NET.

To start using Azure, you must first create an account. Microsoft makes it possible to create a free account with some credits to use in the first month of usage. What is nice here is that once the first month ends or the free credits are used, you are not charged anymore without your explicit consent, so you can use it to really explore the services provided by Microsoft. To create your free account, navigate to https://azure.microsoft.com/free.

In short, a cloud allows you to quickly deploy applications and infrastructure via a set of services through the internet. Let's practice creating a web application resource on Azure through this exercise.

EXERCISE 11.08: CREATING A WEB APPLICATION ON AZURE

In this exercise, you will create a new web application resource on Azure and deploy the API you've been building and hosting on GitHub there.

Perform the following steps to complete this exercise:

1. Create a free Microsoft Azure account by providing your card details.
2. After creating your account, navigate to http://portal.azure.com/.
3. Click on the **`Create a resource`** (**+**) button on the landing page:

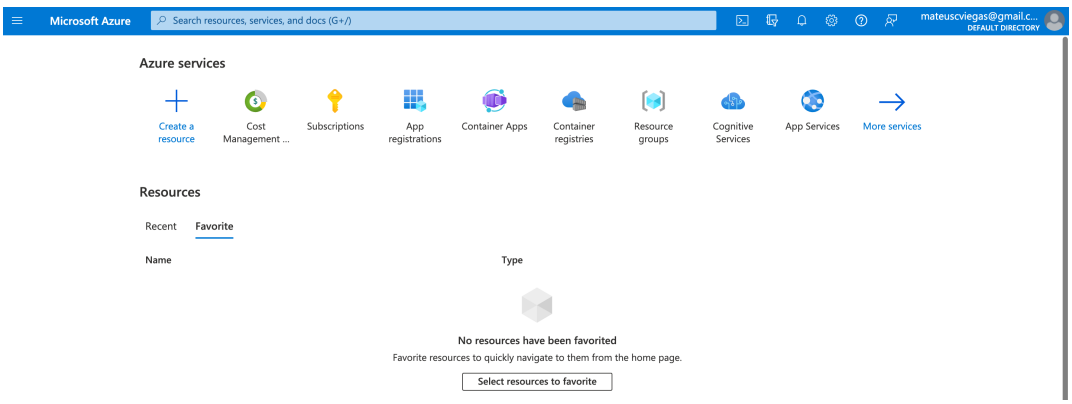

Figure 11.34: Creating a resource

4. Navigate to the **`Web`** category on the menu on the left side.
5. Click on the **`Web App`** option from the resulting menu:

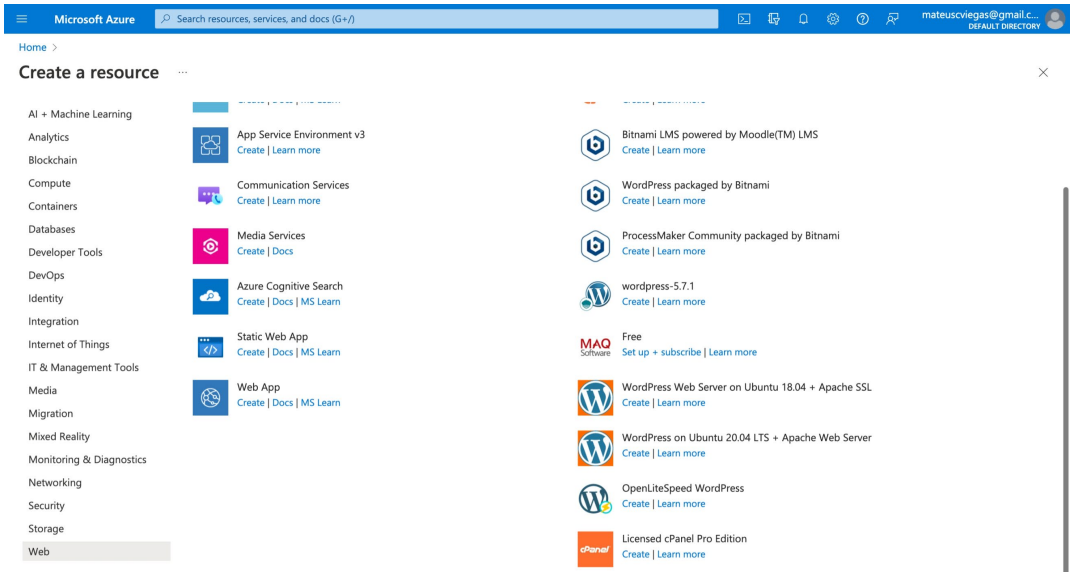

Figure 11.35: Creating a Web App

After choosing Web App, review the screen displayed (*Figure 11.36*). You will need to fill in some of these form fields in the next steps.

6. Fill in the **`Subscription`** details. This will be the trial subscription you will be creating.
7. Next fill the **`Resource Group`** which holds a set of resources created in Azure. So, click **`Create new`** option and type **`c-sharp-workshop`** to create a resource group for your application.
8. Type a **`Name`**. This will be both your application name and part of the DNS that you're going to use to access your application.
9. Select what you want to publish by choosing one option from **`Publish`**. In this case, it will be **`Code`**.
10. Choose **`.NET 6 (LTS)`** as the **`Runtime stack`**.

11. Choose from the **`Operating Systems`** options. For this exercise, the **`Windows`** option is chosen.

12. Choose the **`Region`** closest to where you are based (West US, East US, Central US, Europe, or South America) to reduce the latency. Here the **`Central US`** region is chosen:

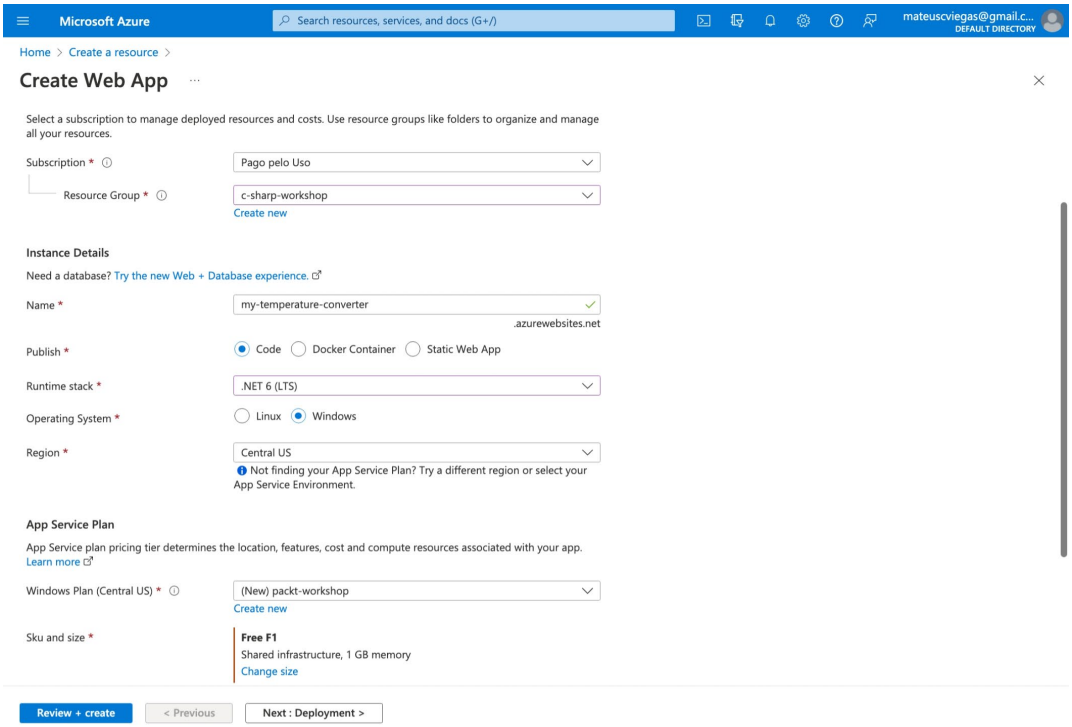

Figure 11.36: Creating an Azure web app

13. The **`App Service Plan`** is used to select the server that will host your application on. For this exercise, choose the **`packt-workshop`** option.

14. Once all the preceding fields are set, click on the **`Review + create`** button. Once your application is created, a summary of the Web App is displayed.

15. Click on the **`Create`** button:

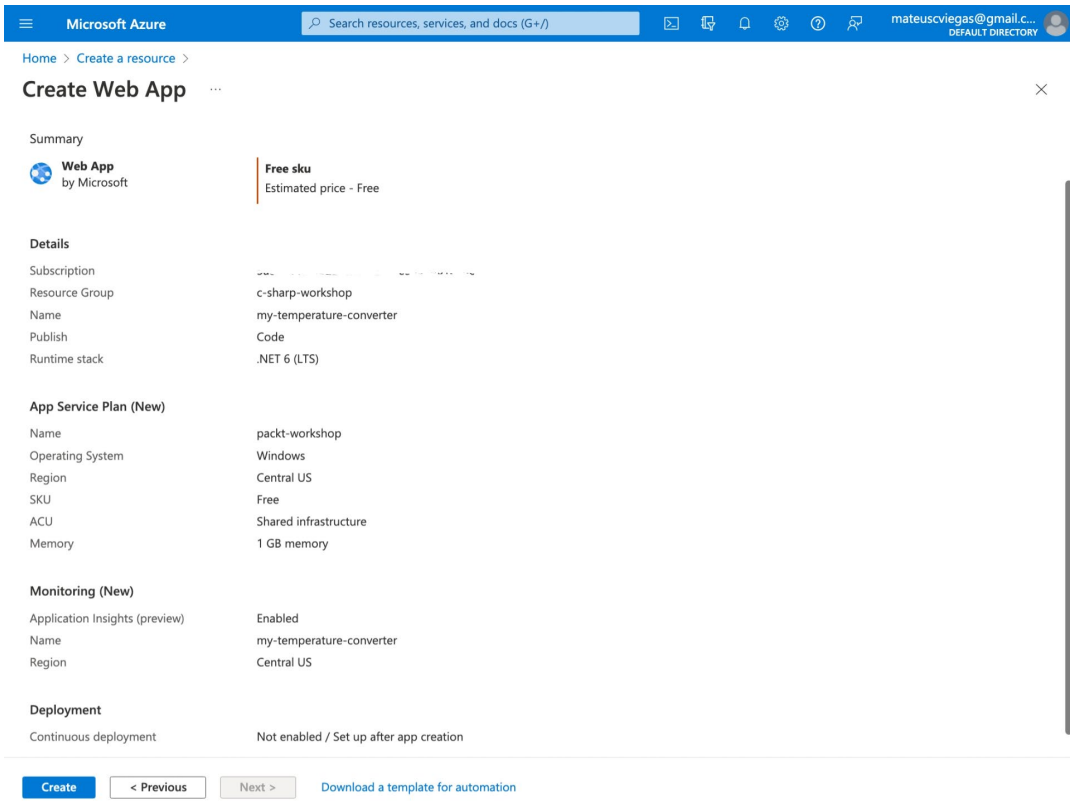

Figure 11.37: A summary review of the Web App

After the deployment gets completed, you'll see a screen similar to *Figure 11.38*:

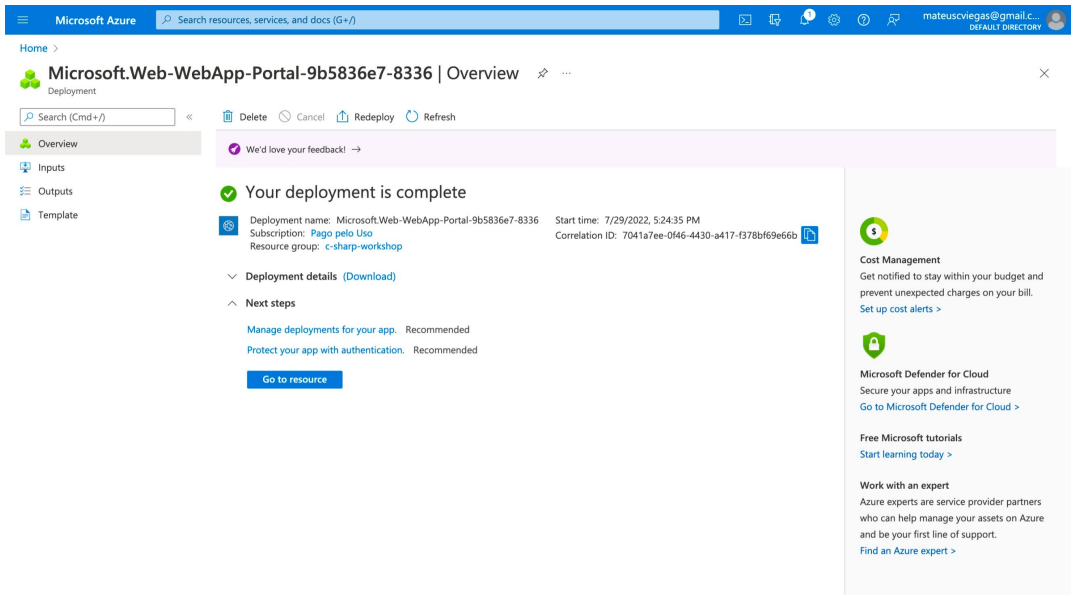

Figure 11.38: Azure deployment completed

16. Now click on the *Go to resource* button to access your application:

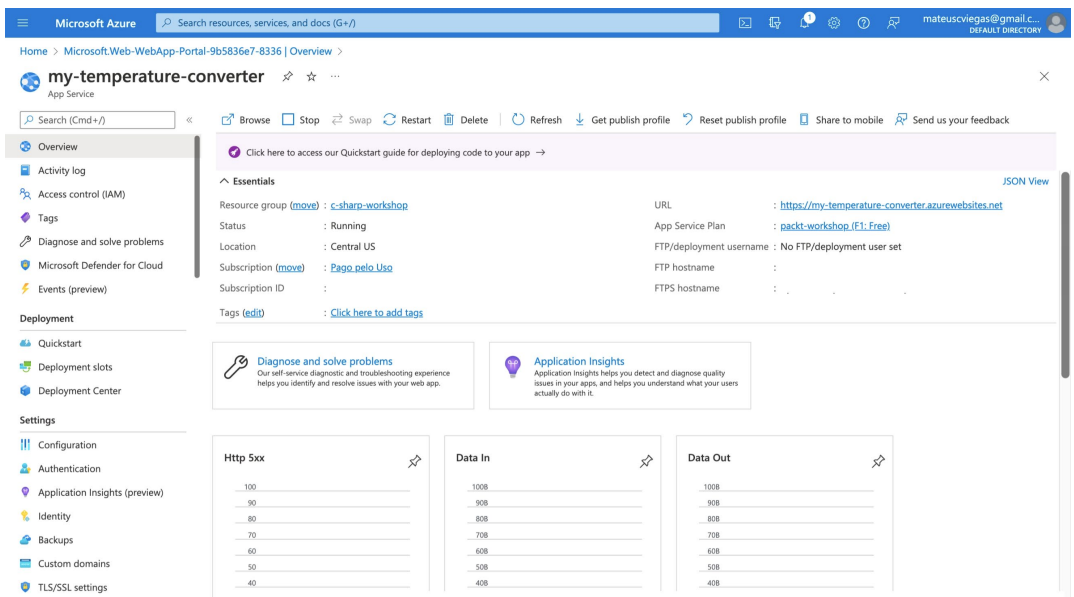

Figure 11.39: Details of the Azure web app created

The **`URL`** is how you will be able to access your app. Since you have yet to perform the deployment, you will notice that the deployment page will be empty:

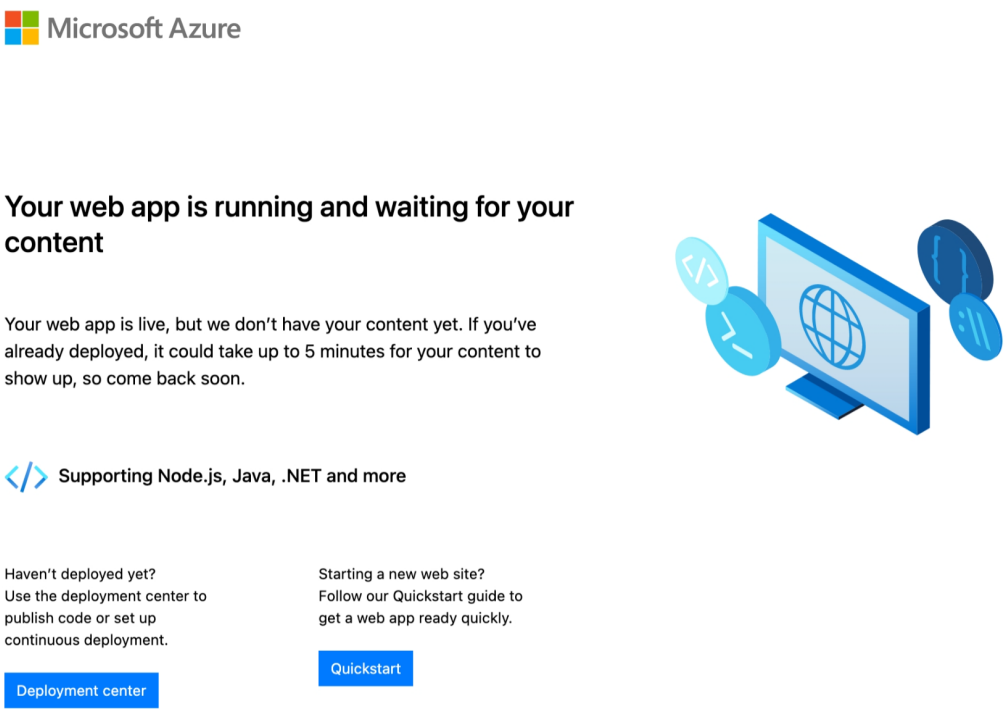

Figure 11.40: Azure web app running with no deployment

In this exercise, you have learned how to create a new web application resource on Azure. You will now learn to deploy.

Before you proceed to learn about the deployment in *Activity 11.01*, grasp the concept behind the continuous deployment.

CONTINUOUS DEPLOYMENT

As you might have noticed, the deployed application is not the application you have been developing. It is actually a default sample app from Microsoft. In order to deploy your application on GitHub, you will create a CD workflow in the following activity.

Before that, however, you first need to know what CD means. CD refers to a workflow in which every change in the mainline code base goes directly into production. That usually results in multiple deployments per day in the real world.

CD is built on top of CI. Since you are deploying changes live directly, you need to ensure the integrity of these changes. The importance of a proper CI pipeline is to automatically integrate new changes into a live environment for a correct build by passing tests. In other words, you can always test your code before it reaches the end user. This gives more reliability to such automation.

In the following activity, you will create a CD workflow with GitHub Actions to automatically deploy new changes to your recently created Azure web app.

ACTIVITY 11.01: CD WORKFLOW WITH GITHUB ACTIONS AND AZURE

This activity will use the same basis that you used in the CI exercise with one extra step; you will now deploy the application you created to Azure automatically. The deployment should happen every time a commit reaches the master branch and successfully builds and tests the application inside the workflow. So, if the build or the tests fail, the application won't be published. With the previously created CI workflow, you were publishing it anyway, only considering a new commit on the master branch and not considering the tests result.

The following steps will help you complete this activity by testing it:

1. Inside the **`.github/workflows`** directory, create a file named **`cd.yaml`**.
2. Here, copy the **`ci.yaml`** file as a base because you need to change only a few parts of it.
3. Pass **`cd`**, instead of **`ci`**, as a name for the file.
4. To ensure this workflow runs only after commits on the master branch exclusively, replace the **`on`** clause with the specification that this workflow will be restricted to the master branch. So, paste the following code there:

```
on:
    push:
        branches:
            - 'main'
            - 'master'
```

5. Maintain the steps of the CI pipeline just as they were.

6. Add an extra step, after the test step of your pipeline, which will publish your application to an output directory so that you can deploy the content of this directory to Azure:

```
- name: Publish API
  run: dotnet publish -c Release -o ${{env.DOTNET_ROOT}}/app
```

7. Add the deploy step of your pipeline, which will push your changes to your Azure web app, by placing the following at the end of your job:

```
- name: Deploy to Azure
  uses: azure/webapps-deploy@v2
  with:
    app-name: ${{ secrets.AZURE_WEBAPP_NAME }}
    publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE  }}
    package: ${{env.DOTNET_ROOT}}/app
```

8. To set the secrets that you see in the preceding snippet, navigate to your repository on GitHub.
9. Go to the **`Settings`** option and choose the **`Secrets`** option.
10. Click on the **`New Repository Secret`** button.
11. Place **`AZURE_WEBAPP_NAME`** as the secret name.
12. Navigate to the web application overview page on the Azure portal.
13. Enter the application name as a value for the preceding secret.
14. On the GitHub secrets page, create a new repository secret with the name **`AZURE_WEBAPP_PUBLISH_PROFILE`**.
15. On the Azure portal, still on the web application overview page, click on **`Get publish profile`**. A file will be downloaded.
16. Open the file and place its contents as the value of the **`AZURE_WEBAPP_PUBLISH_PROFILE`** variable you created in the previous step.
17. Commit your changes to the application repository.

18. Push it to the remote repository on GitHub and watch the action run.
19. After the action succeeds, run the following command in the VSCode integrated terminal to see the output of a **`GET`** request on your deployed API:

```
curl -X GET 'https://YOUR_APP_NAME.azurewebsites.net/
converter?from=0&to=2&value=0'
```

    This request will convert the value of **`0`** from the corresponding entry of index **`0`** in the **`ETemperatureUnit enum`**—**`Celsius`**—to the entry of index **`2`** in the same **`enum`**—**`Kelvin`**. It should provide **`273`** as an output. Remember to replace **`YOUR_APP_NAME`** in the URL with the actual name of your application.

In this activity, you used the power of CI and CD to automatically integrate and deploy new changes in an application to the production environment through GitHub Actions.

> **NOTE**
>
> The solution to this activity can be found at https://packt.link/qclbF.

SUMMARY

In this chapter, you learned the definition and function of a version control system and explored one (Git) in detail. You learned about its basic commands, including **`fetch`**, **`push`**, and **`pull`**, using which you can keep track of your work record as your project progresses. You have also learned how to use a remote repository on GitHub to maintain a distributed copy of your work.

With this chapter, you learned what upstream branches are and how to link your local branches to upstream ones. Finally, you studied the cloud, using Azure, and learned how to use GitHub Actions to perform CI and CD to push application changes live in production.