



1ST EDITION

# Test Automation Engineering Handbook

Learn and implement techniques for building  
robust test automation frameworks

MANIKANDAN SAMBAMURTHY



# Chapter 1

## Figures

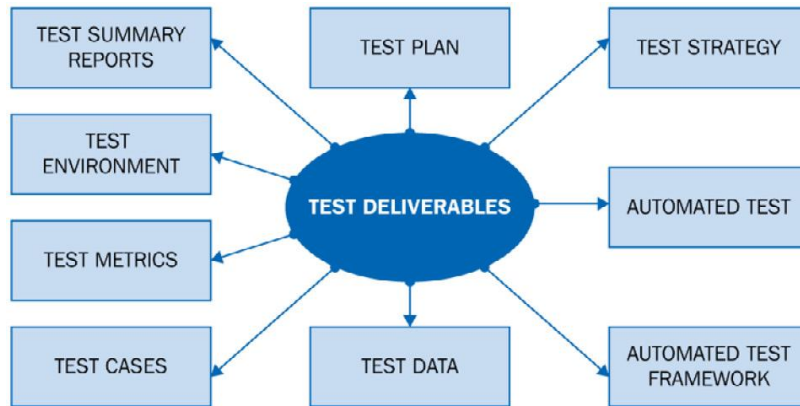


Figure 1.1 – Testing deliverables

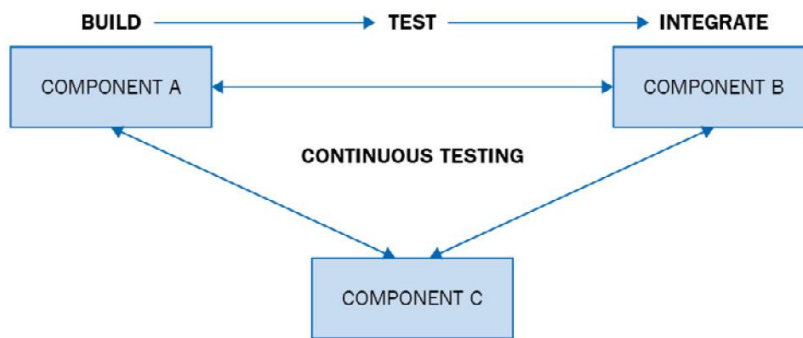


Figure 1.2 – Continuous testing

## Tables

TEST EARLY	TEST OFTEN
<ul style="list-style-type: none"><li>• More time to fix bugs</li></ul>	<ul style="list-style-type: none"><li>• Drastically increases chance of catching bugs faster</li></ul>
<ul style="list-style-type: none"><li>• Cost of fixing bugs is low early on</li></ul>	<ul style="list-style-type: none"><li>• Exposes bugs throughout the development life cycle</li></ul>
<ul style="list-style-type: none"><li>• Fewer surprises in the later stages of product development</li></ul>	<ul style="list-style-type: none"><li>• Easy to make design changes upfront</li></ul>

Table 1.1 – Importance of testing early and testing often

<b>Test automation engineer</b>	<b>SDET</b>
Creates and executes automated and manual tests	Creates and maintains the test automation framework
Collaborates with the product and implementation teams	Collaborates with software engineers and DevOps teams
Highly skilled in programming with testing skills	Experts in testing either manually or by automation
Develops test automation tools	Uses test automation tools

**Table 1.2 – Test automation versus SDET**

# Chapter 2

## Technical requirements

In the later part of this chapter, we will be looking at some Python code to understand a simple implementation of design patterns. You can refer to the following GitHub URL for the code in the chapter: [https://github.com/PacktPublishing/B19046\\_Test-Automation-Engineering-Handbook/blob/main/src/test/design\\_patterns\\_factory.py](https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook/blob/main/src/test/design_patterns_factory.py).

This Python code is provided mainly for understanding the design patterns and the readers don't have to execute the code. But if you are interested, here is the necessary information to get it working on your machine. First, readers will need an **integrated development environment (IDE)** to work through the code. **Visual Studio Code (VS Code)** is an excellent editor with wide support for a variety of programming languages.

The following URL provides a good overview for using Python with VS Code:

<https://code.visualstudio.com/docs/languages/python>

You will need software versions Python 3.5+ and the **Java Runtime Environment (JRE)** 1.8+ installed on your machine to be able to execute this code. **pip** is the package installer for Python, and I would recommend installing it using <https://pip.pypa.io/en/stable/installation>. Once you have PIP installed, you can use the **pip install -U selenium** command to install Selenium Python bindings.

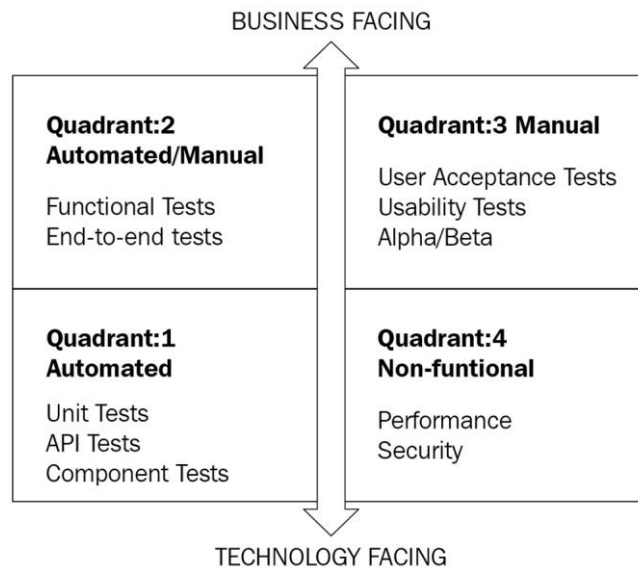
Next is to have the driver installed for your browsers. You can do this by going to the following links for your particular browser:

- Chrome: <https://chromedriver.chromium.org/downloads>
- Firefox: <https://github.com/mozilla/geckodriver/releases>
- Edge: <https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/>

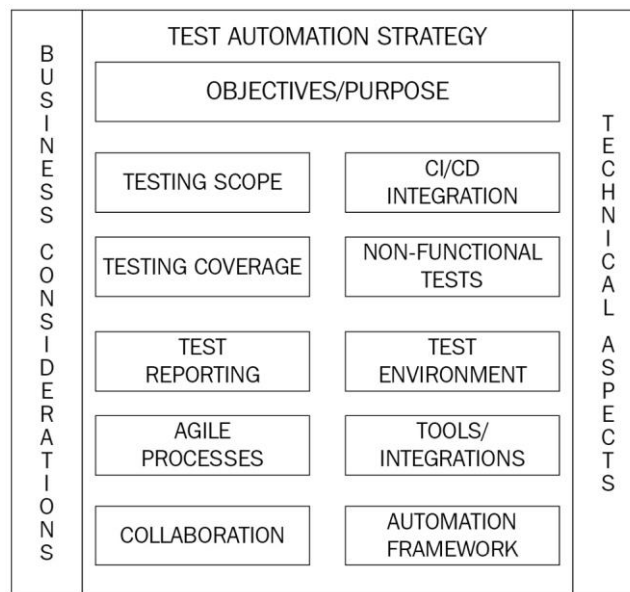
Make sure the driver executables are in your **PATH** environment variable.



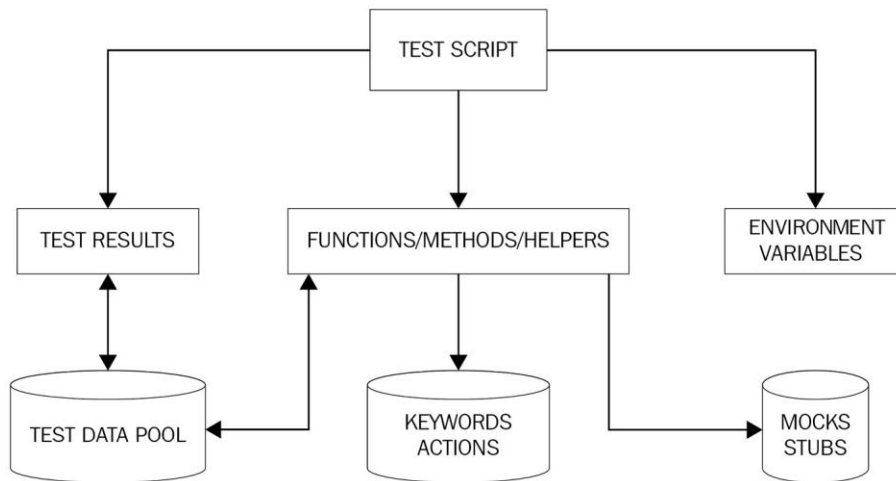
# Figures



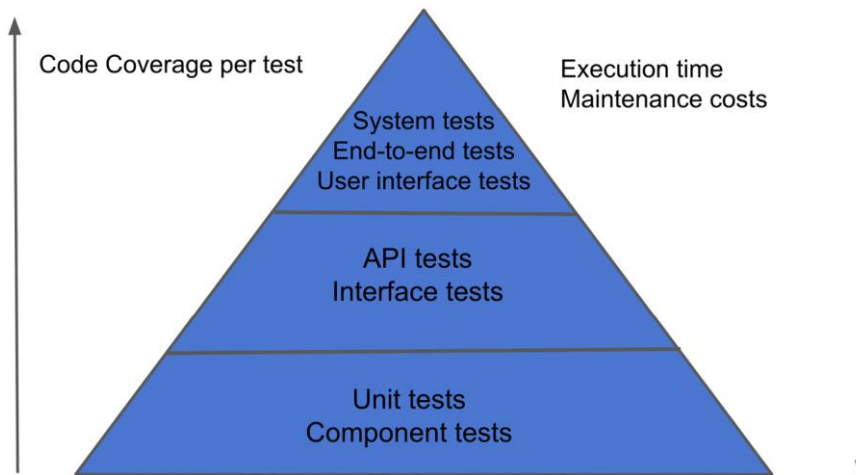
**Figure 2.1 – Agile testing quadrants**



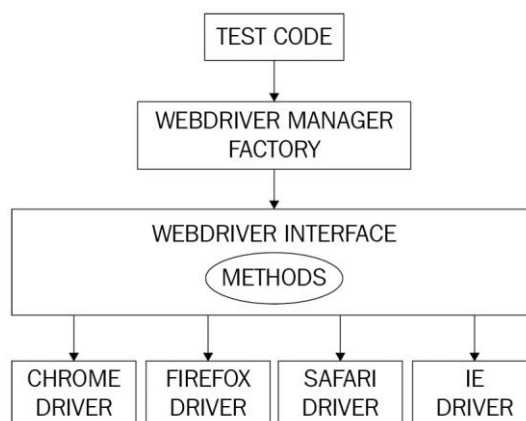
**Figure 2.2 – Test automation strategy breakdown**



**Figure 2.3 – Essential components of a test automation framework**



**Figure 2.4 – Test automation pyramid**



**Figure 2.5 – Selenium WebDriver architecture**

# Hands-On Section

## POM

Object repositories, in general, help keep the objects used in a test script in a central location rather than having them spread across the tests. POM is one of the most used design patterns in test automation; it aids in minimizing duplicate code and makes code maintenance easier. A page object is a class defined to hold elements and methods related to a page on the UI, and this object can be instantiated within the test script. The test can then use these elements and methods to interact with the elements on the page.

Let us imagine a simple web page that serves as an application for various kinds of loans (such as personal loans, quick money loans, and so on). There may be multiple business flows associated with this single web page, and these can be set up as different test cases with distinct outcomes. The test script would be accessing the same UI elements for these flows except when selecting the type of loan to apply for. POM would be a useful design pattern here as the UI elements can be declared within the page object class and utilized in each of the different tests running the business flows. Whenever there is an addition or change to the elements on the UI, the page object class is the only place to be updated.

The following code snippet illustrates the creation of a simple page object class and how the **test\_search\_title** test uses common elements on the UI from the **Home\_Page\_Object** page object class to perform its actions:

```
import selenium.webdriver as webdriver
from selenium.webdriver.common.by import By
class WebDriverManagerFactory:
def getWebdriverForBrowser(browserName):
if browserName=='firefox':
return webdriver.Firefox()
elif browserName=='chrome':
return webdriver.Chrome()
elif browserName=='edge':
return webdriver.Edge()
else:
return 'No match'
```

The **WebDriverManagerFactory** class contains a method to select the driver corresponding to the browser being used, as illustrated in the following code snippet:

```
class Home_Page_Object:
def __init__(self, driver):
self.driver = driver
def load_home_page(self):
```

```

self.driver.get("https://www.packtpub.com/")

return self

def load_page(self, url):
self.driver.get(url)

return self

def search_for_title(self, search_text):
self.driver.find_element(By.ID, 'search').send_keys(search_text)

search_button=self.driver.find_element(By.XPATH,
'//button[@class="action search"]')

search_button.click()

def tear_down(self):
self.driver.close()

```

The POM paradigm usually has a base class that contains methods for identifying various elements on the page and the actions to be performed on them. The **Home\_Page\_Object** class here in this example has methods to set up the driver, load the home page, search for titles, and close the driver:

```

class Test_Script:
def test_1(self):
driver = WebDriverManagerFactory
.getWebdriverForBrowser("chrome")

if driver == 'No match':
raise Exception("No matching browsers found")

pageObject = Home_Page_Object(driver)
pageObject.load_home_page()
pageObject.search_for_title('quality')
pageObject.tear_down()

def main():
test_executor = Test_Script()
test_executor.test_1()

if __name__ == "__main__":
main()

```

The **test\_search\_title** method sets up the Chrome driver and the page object to search for the **quality** string.

We will look in further detail about setting up and using POM in [Chapter 5, Test Automation for Web](#).

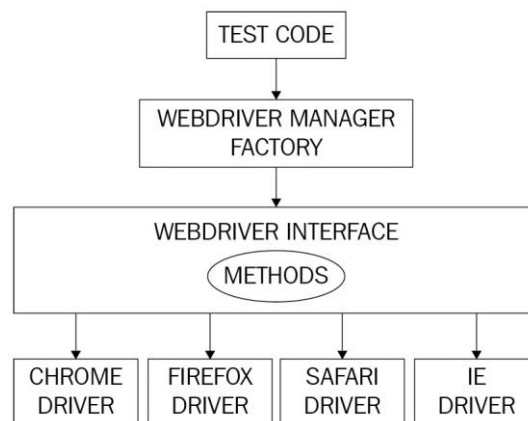
Now, let us investigate how the factory design pattern is helpful in test automation.

## The factories pattern

The factory pattern is one of the most used design patterns in test automation and aids in creating and managing test data. The creation and maintenance of test data within a test automation framework could easily get messy, and this approach provides a clean way to create the required objects in the test script, thereby decoupling the specifics of the factory classes from the automation framework. Separating the data logic from the test also helps test engineers keep the code clean, maintainable, and easier to read. This is often achieved in test automation by using pre-built libraries and instantiating objects from classes exposed by the libraries. Test engineers can use the resulting object in their scripts without the need to modify any of the underlying implementations.

A classic example of a factory design pattern in test automation would be how **Selenium WebDriver** gets initialized and passed around in a test. Selenium WebDriver is a framework that enables the execution of cross-browser tests.

The following diagram breaks down how a piece of test script can exercise Selenium WebDriver to make cross-browser calls:



**Figure 2.5 – Selenium WebDriver architecture**

[Figure 2.5](#) shows how the test code utilizes a factory method to initialize and use the web drivers.

Please refer to the code snippet in the previous section for a simple implementation of the factory pattern. Here, the **WebDriverManagerFactory** class returns an instance of the web driver for the requested browser. The **Test\_Search\_Choose\_Title** class can use the factory method to open a Chrome browser and perform additional validations. Any changes to how drivers are being created are encapsulated from the test script. If Selenium WebDriver supports additional browsers in the future, the factory method will be updated to return the corresponding driver.

## Business layer pattern

This is an architectural design pattern where the test code is designed to handle each layer of the application stack separately. The libraries or modules that serve the test script are intentionally broken down into UI, business logic/API, and data handling. This kind of design is immensely helpful

when writing E2E tests where there is a constant need to interface with the full stack. For example, the steps involved may be to start with seeding the database with the pre-requisite data, make a series of API calls to execute business flows, and finally validate the UI for correctness. It is critical here to keep the layers separate as this reduces the code maintenance nightmare. Since each of the layers is abstracted, this design pattern promotes reusability. All the business logic is exercised in the API layer, and the UI layer is kept light to enhance the stability of the framework.

Design patterns play a key role in improving the overall test automation process and they should be applied after thoroughly understanding the underlying problem. We need to be wary of the fact that if applied incorrectly, these design patterns could lead to unnecessary complications.

# Chapter 3

## Technical requirements

In this chapter, we will be looking at working examples of the CLI and Git. We will be using the **Terminal** software on macOS for our examples in both sections. Windows users can use the in-built **PowerShell** to execute these commands.

## Hands-On Sections

### The basic tools for every automation engineer

One of the primary tasks of a test automation engineer is to create, edit, or delete code daily. Test engineers will often also have to interact with the shell of the system under development to tweak their test environments or the underlying test data. In this section, we will be covering a few basic commands that test engineers will need to be able to access the source code and navigate the system under test. This section is a quick refresher for readers who are already experienced in the software engineering space, and can help to build a good foundation for beginners.

Let us start by looking at the CLI.

#### The CLI

The CLI is a means to interact with the shell of the system under test. A lot of the tasks performed through the graphical user interface can be done through the CLI too. But, the real might of the CLI lies in its ability to programmatically support the simulation of these tasks. Let's try and get familiar with a few basic CLI commands. The CLI commands covered in this section can be run on Terminal software on macOS, or PowerShell on Windows:

- The **ls** command lists all the files and directories in the current folder:

```
→ls
```

The output to the preceding command should be as follows:

```
test.py  test.txt  testing_1 testing_2 testing_3 testing_4  
→
```

- The **cd** command stands for change directory and is used to switch to another directory. The **cd ..** command navigates to the parent directory.

The syntax is as follows:

```
cd [path_to_directory]
```

The command and output should be as follows:



```
→cd testing_1→ testing_1 cd ..
```

- The **mkdir** command creates a new directory under the current directory.

The syntax is as follows:

```
mkdir [directory_name]
```

The command and output should be as follows:

```
→mkdir testing_5
→ls
test.py  test.txt  testing_1 testing_2
testing_3  testing_4 testing_5
→
```

- The **touch** command creates a new file in the current directory without a preview.

The syntax is as follows:

```
touch [file_name]
```

The command and output should be as follows:

```
→ touch testing.txt
→ls
test.py  test.txt  testing.txt testing_1  testing_2
testing_3  testing_4  testing_5
```

## Note

Windows PowerShell users can use **ni** as **touch** is not supported.

- The **cat** command allows the user to view file contents on the CLI.

The syntax is as follows:

```
cat [file_name]
```

The command and output should be as follows:

```
→ cat vim_file
This is a new file
→ cli_demo
```

So far, we have looked at how to create and modify files. Next, let us look at the commands for deleting files and folders:

- The **rm** command can be used to delete folders and files. Let us look at some specific examples of how to go about this deletion.

To remove a directory and all the contents under that directory, use the **rm** command with the **-r** option.

The syntax is as follows:

```
rm -r [directory_name]
```

The command and output should be as follows:

```
→ ls
test.py      test.txt      testing.txt  testing_1    testing_2
testing_3    testing_4     testing_5    vim_file
→ rm -r testing_1
→ls
test.py      test.txt      testing.txt  testing_2    testing_3
testing_4    testing_5     vim_file
→
```

- To delete the file(s), the same **rm** command can be used followed by the filename.

The syntax is as follows:

```
rm [file_name]
```

The command and output should be as follows:

```
→ cli_demo ls
test.py      test.txt      testing.txt  testing_2    testing_3
testing_4    testing_5     vim_file
→ cli_demo rm test.txt
→ cli_demo ls
test.py      testing.txt  testing_2    testing_3
testing_4    testing_5     vim_file
→ cli_demo
```

Next, let us quickly look at **Vim**, which is a commonly used file-handling tool for the CLI.

### Working with Vim

Vim is an in-built editor that allows you to modify the contents of a file. Vim aims to increase efficiency when editing code via the CLI and is supported across all major platforms, such as macOS,

Windows, and Linux. Vim also supports creating custom keyboard shortcuts based on your typing needs. Let's look at a basic example of editing and saving a file. This editor supports a wide range of commands and can be referenced at <http://vimdoc.sourceforge.net>. To edit and save a file, you need to do the following:

1. To execute the editor, the user has to type **vi**, followed by a space and the filename:

```
workspace vi test.txt
```

1. Then, type **i** to switch to **INSERT** mode and type in the contents of the file.
2. Press the **Esc** key to quit **INSERT** mode.
3. Next, type **:wq** to save and exit. This command is a combination of **:w** to write the contents of the file to the disk and **q** to quit the file.
4. Then, press **i** to enter **INSERT** mode and type the required contents in the file.

The CLI commands we have looked at so far should serve as a good starting point for new users. Now, let us familiarize ourselves with flags in the CLI.

### Flags in the CLI

Flags are add-ons to enhance the usage of a command in the CLI. For example, the **-l** flag can be applied to the **ls** command to alter the displayed list of files and folders in a long format.

The syntax is as follows:

```
ls -l
```

The command and output should be as follows:

```
→ls -l
total 8
-rw-r--r--  1 packt  staff    0 Jun 25 09:33 test.py
-rw-r--r--  1 packt  staff    0 Aug 14 18:14 testing.txt
drwxr-xr-x  3 packt  staff   96 Jun 26 10:20 testing_2
drwxr-xr-x  2 packt  staff   64 Jun 25 09:32 testing_3
drwxr-xr-x  2 packt  staff   64 Jun 25 23:16 testing_4
drwxr-xr-x  2 packt  staff   64 Aug 14 18:13 testing_5
-rw-r--r--  1 packt  staff   19 Aug 14 18:26 vim_file
→
```

There are thousands of flags that can be attached to various CLI commands, and it is impossible to know all of them. This is where the **man** command comes in handy. **man** can be used with any CLI command, and it gives all the options and an associated description for each command. There are usually multiple pages of help content and you are encouraged to browse through them.

For example, to learn all the information associated with the **ls** commands, you just have to run the following command:

```
man ls
```

There are a few tips/tricks to keep in mind regarding CLI usage, such as the following:

- All the CLI commands are case sensitive
- The **pwd** command lists the current working directory
- The **clear** command clears the contents on the current shell window
- The **up/down** arrow keys can be used to navigate through the history of the CLI commands
- The **Tab** key can be used to get autocomplete suggestions based on the string typed so far
- The **cd -** and **cd ~** commands can be used to navigate to the last working directory and home directory, respectively
- Multiple CLI commands can be run in a single line using the **;** separator

### The power of shell scripting

The ultimate utility of the CLI lies in writing automated scripts that perform repeatable tasks. Shell scripting can be used to achieve this and can save you a great deal of time. Users are encouraged to refer to the full documentation at <https://www.gnu.org/software/bash/manual/bash.html> to learn more about commands and their syntax. To understand the power of the CLI, let us look at an example of a shell script in this section. This script creates a folder, named **test\_folder**, and then creates a text file, named **test\_file**, within it. The script then uses the **curl** command to download a web resource that is passed as an argument and stores its output in **test\_file.txt**. Now, **\$1** refers to the first argument used when invoking this file for execution. **-o** is used to override the contents of the file. Then, it reads the file using the **cat** command and stores it in a variable named **file\_content**. Finally, this file is removed using the **rm** command:

```
#!/bin/bash
mkdir test_folder
cd test_folder
touch test_file.txt
curl $1 -o test_file.txt
file_content=`cat test_file.txt`
echo $file_content
rm test_file.txt
```

This script can be executed using the **bash sample\_bash\_script.sh https://www.packt.com/** command, where **sample\_bash\_script.sh** is the name of the file. Please note that the web resource here can be downloaded at <https://www.packt.com/> and that it is being passed as the first argument to the script.

We have just gotten a bird's eye view of the CLI, and I strongly encourage you to dive deeper into CLI commands to increase your proficiency. Some major advantages of using the CLI include the following:

- **Speed and security:** CLI commands are faster and more secure to use than the corresponding actions being done through the graphical user interface.
- **Scripting on the CLI:** The CLI lets users write scripts to perform repetitive actions by combining them into a single script file. This is much more stable and efficient than a script run on a graphical user interface.
- **Resource efficient:** CLI commands use much fewer system resources and therefore provide better stability.

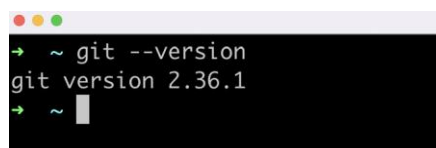
Now that we have familiarized ourselves with the CLI, let us look at another tool that is an absolute necessity for the maintenance of a software project of any size.

## Git

Git is a modern distributed version control system that allows tracking changes to the source code and is a versatile tool to enable collaboration in the engineering team. Git primarily helps in synchronizing contributions to source code by various members of the team, by keeping track of the progress over time.

Every software application is broken down into code repositories and production code is stored on a branch called master on the repository. When an engineer is ready to begin working on a feature, they can clone the repository locally and create a new branch to make their changes. After the code changes are complete, the engineer creates a pull request that is then peer-reviewed and approved. This is when they are merged into the master branch. Subsequently, the changes are deployed to the staging and production environments. There are various hosting services, such as GitHub, that provide a user interface to maintain, track, and coordinate contributions to the code repositories. Now, let us look at some of the common Git commands that test engineers might have to use frequently:

- **git --version** shows the version of Git software installed on the machine:



```
→ ~ git --version
git version 2.36.1
→ ~
```

**Figure 3.1 – git version**

- **git init** initializes the project folder into a GitHub repository:

```
+ git_demo git:(master) * git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/priya/Documents/git_demo/.git/
+ git_demo git:(master) |
```

Figure 3.2 – git init

- **git clone [repository\_URL]** creates a local copy of the remote repository:

```
+ git_demo git:(master) git clone https://github.com/PacktPublishing/B19046_Test-Automation-En
gineering-Handbook.git
Cloning into 'B19046_Test-Automation-Engineering-Handbook'...
remote: Enumerating objects: 116, done.
remote: Counting objects: 100% (116/116), done.
remote: Compressing objects: 100% (90/90), done.
remote: Total 116 (delta 20), reused 89 (delta 9), pack-reused 0
Receiving objects: 100% (116/116), 3.73 MiB | 5.49 MiB/s, done.
Resolving deltas: 100% (20/20), done.
+ git_demo git:(master) * |
```

Figure 3.3 – git clone

- **git push** pushes all of the committed local changes to the remote GitHub repository:

```
+ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) * git push origin mani/
update_readme
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 390 bytes | 390.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'mani/update_readme' on GitHub by visiting:
remote:   https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook/pul
l/new/mani/update_readme
remote:
To https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook.git
 * [new branch]   mani/update_readme -> mani/update_readme
+ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) * |
```

Figure 3.4 – git push

- **git pull** pulls all the latest code from the remote branch and merges them with the local branch:

```
+ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) * git pull
Already up to date.
+ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) * |
```

Figure 3.5 – git pull

- **git log** lists the entire commit history:

```

git log
commit 1efeb6a2f799cb91f75d1805f735a97cea9638ed (HEAD -> mani/update_readme, origin/mani/update_readme)
Author: Mani S <manikandan.sambamurthy@gmail.com>
Date:   Tue Nov 15 22:43:37 2022 -0800

    Update readme for chapter 3 content

commit d0cc13896ced2c37fe4a79d64a2c20eda0d8340c (origin/main, origin/HEAD, main)
Merge: cb0317c 7d8870c
Author: manisam <manikandan.sambamurthy@gmail.com>
Date:   Tue Sep 27 08:03:27 2022 -0700

    Merge pull request #12 from PacktPublishing/mani/move_ch3_src

    Moved ch3 under src

commit 7d8870c1c284cdd9df0c686774eb74edfdce1d43 (origin/mani/move_ch3_src, mani/move_ch3_src)
Author: Mani S <manikandan.sambamurthy@gmail.com>
Date:   Tue Sep 27 08:03:00 2022 -0700

    Moved ch3 under src
  
```

**Figure 3.6 – git log**

- **git branch [branch\_name]** creates a new branch in the local Git repository:

```

B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git branch mani/git-demo-branch
B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x
  
```

**Figure 3.7 – git branch [branch\_name]**

- **git branch** lists all the local branches created so far. \* indicates the branch that is currently checked out:

```

ch4/git_a
ch4/git_amend
ch4/git_commit_multiline
ch4/merge_conflict_branch_1
main
man/ch4_js_snippets
mani/ch3_bash_script
mani/ch5_contents
mani/ch6_appium_webdriverio
mani/check_push_access
mani/design_patterns_factory
mani/git-demo-branch
mani/move_ch3_src
mani/rename_search_method
* mani/update_readme
(END)
  
```

**Figure 3.8 – git branch**



- **git branch -a** lists all the local and remote branches created so far:

```

git branch -a
ch4/git_a
ch4/git_amend
ch4/git_commit_multiline
ch4/merge_conflict_branch_1
main
man/ch4_js_snippets
mani/ch3_bash_script
mani/ch5_contents
mani/ch6_appium_webdriverio
mani/check_push_access
mani/design_patterns_factory
mani/git-demo-branch
mani/move_ch3_src
mani/rename_search_method
* mani/update_readme
remotes/origin/HEAD -> origin/main
remotes/origin/ch4/git_a
remotes/origin/ch4/git_amend
remotes/origin/ch4/git_commit_multiline
remotes/origin/ch4/merge_conflict_branch_1
remotes/origin/main
remotes/origin/man/ch4_js_snippets
remotes/origin/mani/ch3_bash_script

```

Figure 3.9 – git branch -a

- **git checkout [branch\_name]** switches between local Git branches:

```

B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git checkout ma
ni/rename_search_method
Switched to branch 'mani/rename_search_method'
B19046_Test-Automation-Engineering-Handbook git:(mani/rename_search_method) x

```

Figure 3.10 – git checkout

- **git status** displays the modified files and folders in the current project repository:

```

B19046_Test-Automation-Engineering-Handbook git:(main) x git status
On branch main
Your branch is up to date with 'origin/main'.

You are currently bisecting, started from branch 'main'.
(use "git bisect reset" to get back to the original branch)

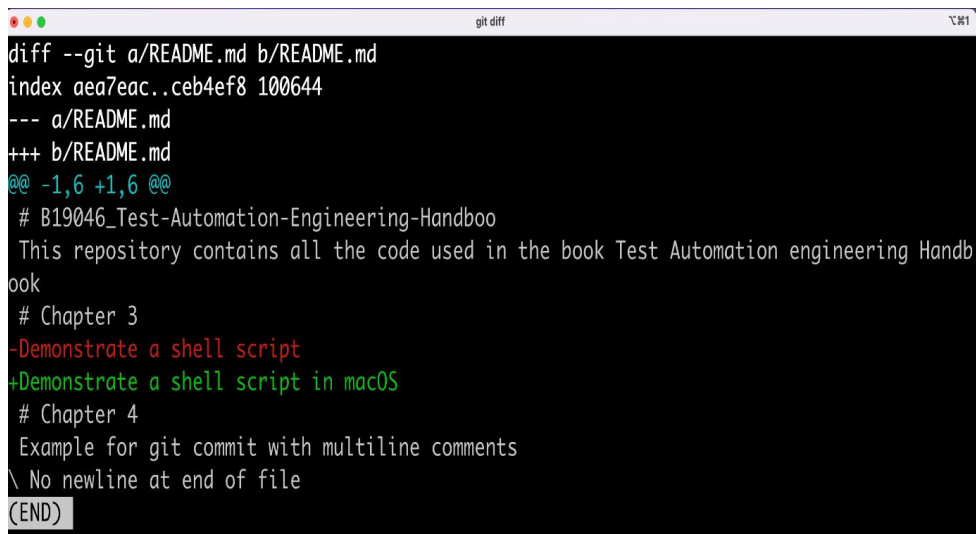
Untracked files:
(use "git add <file>..." to include in what will be committed)
src/ch5/cypress.config.js
src/ch5/cypress/
src/ch8/

nothing added to commit but untracked files present (use "git add" to track)
B19046_Test-Automation-Engineering-Handbook git:(main) x

```

Figure 3.11 – git status

- **git diff** shows the difference between files in the staging area and the working tree:



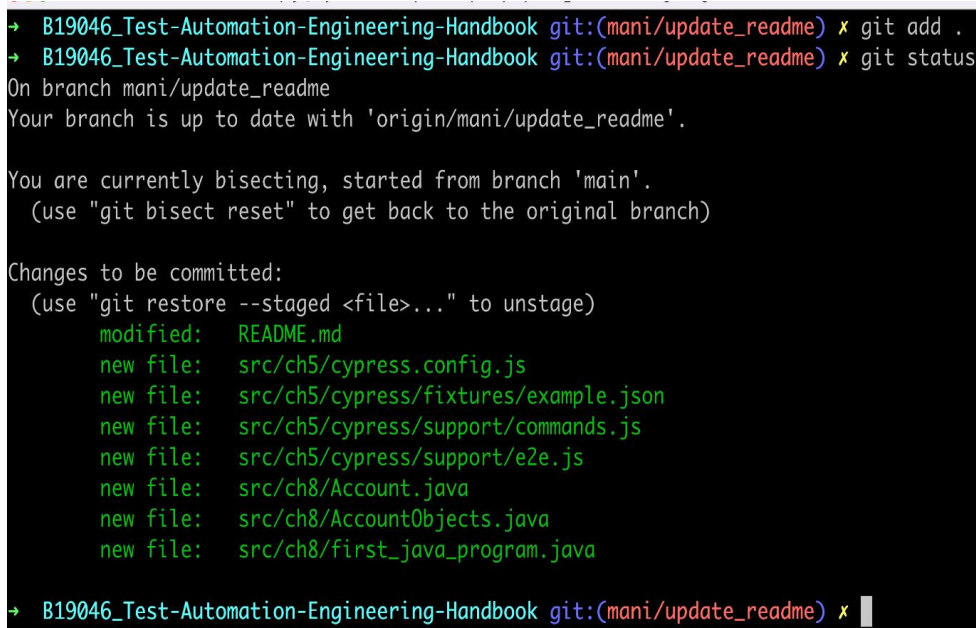
```

diff --git a/README.md b/README.md
index aea7eac..ceb4ef8 100644
--- a/README.md
+++ b/README.md
@@ -1,6 +1,6 @@
 # B19046_Test-Automation-Engineering-Handbook
 This repository contains all the code used in the book Test Automation engineering Handbook
 # Chapter 3
-Demonstrate a shell script
+Demonstrate a shell script in macOS
 # Chapter 4
 Example for git commit with multiline comments
 \ No newline at end of file
(END)

```

**Figure 3.12 – git diff**

- **git add .** adds all the modified files to the Git staging area:



```

→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git add .
→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git status
On branch mani/update_readme
Your branch is up to date with 'origin/mani/update_readme'.

You are currently bisecting, started from branch 'main'.
(use "git bisect reset" to get back to the original branch)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.md
    new file:   src/ch5/cypress.config.js
    new file:   src/ch5/cypress/fixtures/example.json
    new file:   src/ch5/cypress/support/commands.js
    new file:   src/ch5/cypress/support/e2e.js
    new file:   src/ch8/Account.java
    new file:   src/ch8/AccountObjects.java
    new file:   src/ch8/first_java_program.java

→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x

```

**Figure 3.13 – git add**

- **git commit -m "commit description"** saves the changes to the local repository with the provided description:

```

→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git commit -m "
GIT Demo commit"
[mani/update_readme 6960800] GIT Demo commit
8 files changed, 100 insertions(+), 1 deletion(-)
create mode 100644 src/ch5/cypress.config.js
create mode 100644 src/ch5/cypress/fixtures/example.json
create mode 100644 src/ch5/cypress/support/commands.js
create mode 100644 src/ch5/cypress/support/e2e.js
create mode 100644 src/ch8/Account.java
create mode 100644 src/ch8/AccountObjects.java
create mode 100644 src/ch8/first_java_program.java
→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) █

```

Figure 3.14 – git commit

- **git branch -D [branch\_name]** force deletes the specified local branch.
- **git stash** temporarily removes the changes on the local branch. Use **git stash pop** to apply the changes back onto the local branch:

```

→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git stash
Saved working directory and index state WIP on mani/update_readme: 6960800 GIT Demo commi
t
→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) git stash pop
On branch mani/update_readme
Your branch is up to date with 'origin/mani/update_readme'.

You are currently bisecting, started from branch 'main'.
(use "git bisect reset" to get back to the original branch)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (2f6325fb0bab45d2de37bf59e80a01f74af80a35)
→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x █

```

Figure 3.15 – git stash

- **git remote -v** gives the name, as well as the URL, of the remote repository:

```

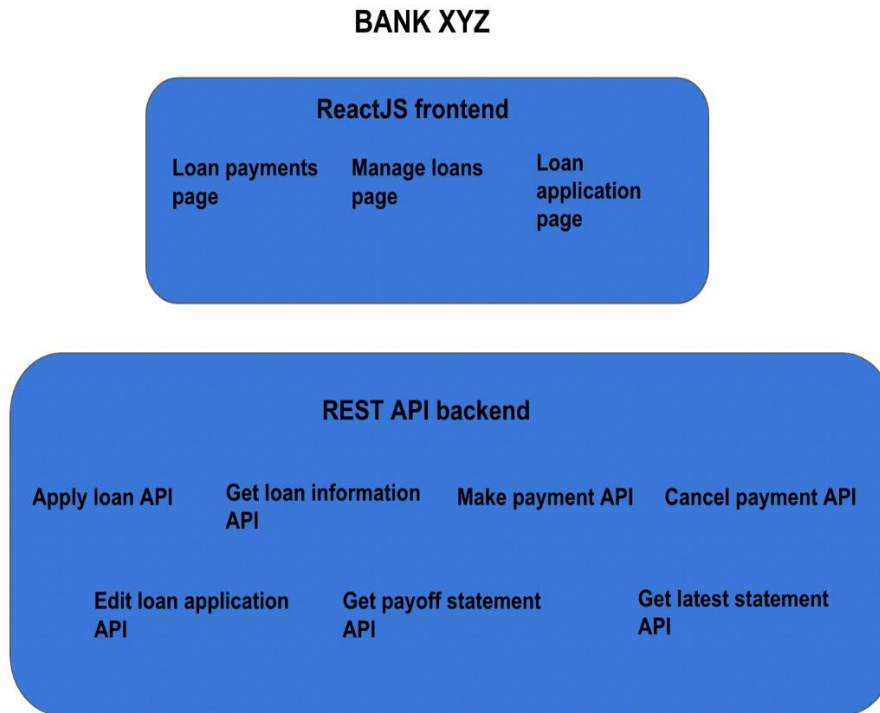
→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x git remote -v
origin https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook.gi
t (fetch)
origin https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook.gi
t (push)
→ B19046_Test-Automation-Engineering-Handbook git:(mani/update_readme) x █

```

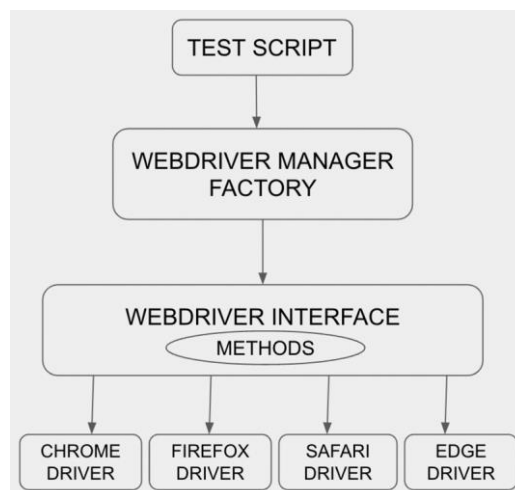
Figure 3.16 – git remote

This overview provides you with a healthy introduction to Git and its most commonly used commands. You can explore additional commands and their usage here at <https://git-scm.com/docs/git>. Next, let us dive into some of the most commonly used test automation frameworks.

## Figures



**Figure 3.17 – Example loan application**



**Figure 3.18 – Visual representation of how a test script utilizes a factory method to initialize and use the drivers**

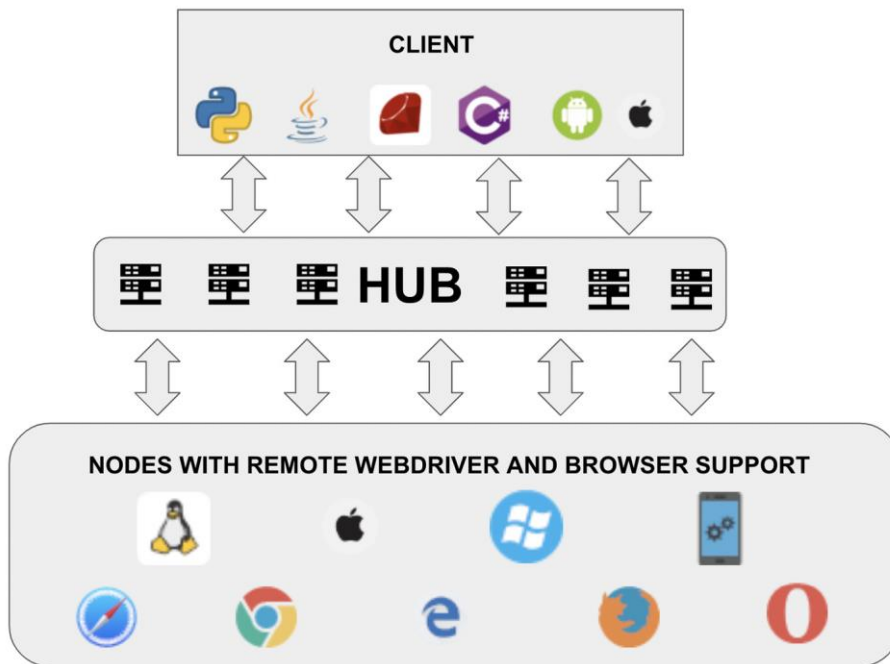


Figure 3.19 – Components of Selenium Grid

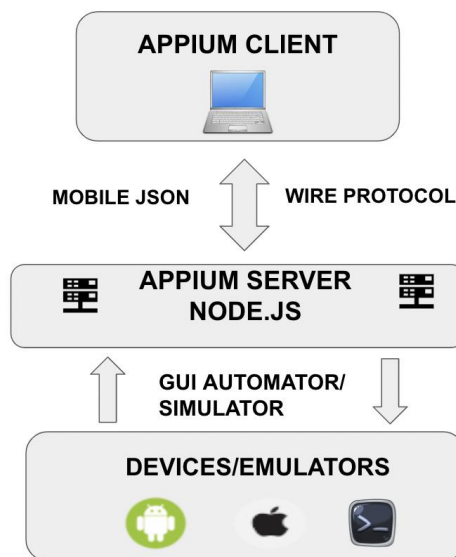


Figure 3.20 – Appium architecture/components

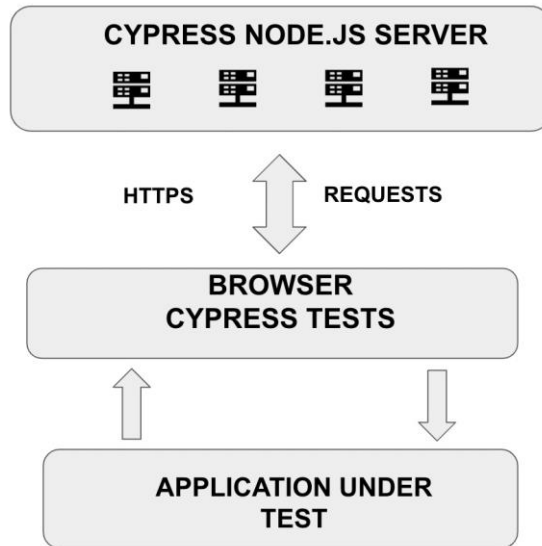


Figure 3.21 – Cypress architecture/components

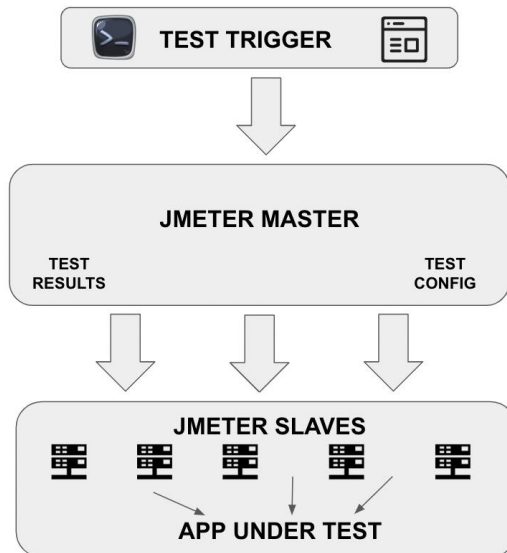


Figure 3.22 – JMeter architecture/components

## Table

Tool	Popularly used for	Applications tested	Supported platforms	Supported programming languages
Selenium	Web browser automation	Web, mobile (with external integrations)	Windows/macOS/Linux	JavaScript, Java, Python, C#, PHP, Ruby, Perl
Appium	Native and hybrid mobile application automation	Mobile	iOS, Android, macOS, Windows	JavaScript, Java, Python, C#, PHP, Ruby

Tool	Popularly used for	Applications tested	Supported platforms	Supported programming languages
Cypress	E2E testing for web applications	Web	Windows/macOS/Linux	JavaScript
JMeter	Performance testing of web applications	Web	Windows/macOS/Linux	Java, Groovy script
AXE	Accessibility testing and associated compliance	Web, mobile	Windows, macOS, iOS, Android	JavaScript, Java, Python, C#, PHP, Ruby

## Links

This is the high-level architecture of the Appium tool. In [Chapter 6, Test Automation for Mobile](#), we will look at a test case's implementation using Appium. In the meantime, you can refer to Appium's official documentation at <https://appium.io/docs/en/about-appium/intro/>.

The Selenium project can be found at <https://github.com/SeleniumHQ>.

For further reading on Cypress, you can refer to the documentation at <https://docs.cypress.io>

The official documentation for JMeter can be found at <https://jmeter.apache.org/usermanual/index.html>

You can further explore the capabilities of the AXE tool by referring to <https://www.deque.com/axe/core-documentation/api-documentation/>.



# Chapter 4

## Technical requirements

In this chapter, we will continue looking at working examples of Git through the **Command-Line Interface (CLI)**. We will be using the **Terminal** software on the Mac for our examples. Windows users can use the built-in **PowerShell** to execute these commands. We will also be downloading and exploring VS Code, which is an IDE. Please check this page for the download requirements: <https://code.visualstudio.com/docs/supporting/requirements>. We also expect you to know the basics of HTML to follow along with the next section on JavaScript.

All the code snippets can be found in the GitHub repository:

[https://github.com/PacktPublishing/B19046\\_Test-Automation-Engineering-Handbook](https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook) in the `src/ch4` folder.

## Figures

```
→ B19046_Test-Automation-Engineering-Handbook git:(main) git checkout -b ch4/git_commit_multiline
Switched to a new branch 'ch4/git_commit_multiline'
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_commit_multiline) git status
On branch ch4/git_commit_multiline
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_commit_multiline) x git add README.md
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_commit_multiline) x git commit -m "readme file: add chapter 4 header"
-m "readme file: add a note for git commit multiline comment"
[ch4/git_commit_multiline a858d24] readme file: add chapter 4 header
1 file changed, 3 insertions(+), 1 deletion(-)
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_commit_multiline) git push origin ch4/git_commit_multiline
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 437 bytes | 437.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'ch4/git_commit_multiline' on GitHub by visiting:
remote:   https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook/pull/new/ch4/git_commit_multiline
remote:
To https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook.git
 * [new branch]      ch4/git_commit_multiline -> ch4/git_commit_multiline
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_commit_multiline) █
```

Figure 4.1 – git commit multiline comment

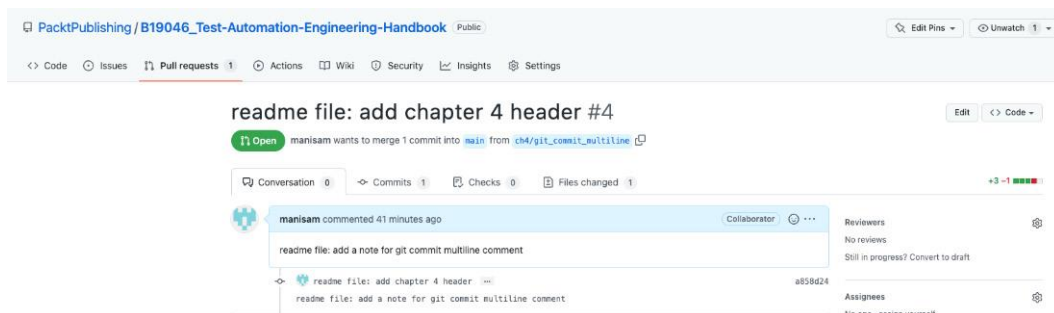


Figure 4.2 – GitHub multiline commit view

```

→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) ✖ git status
On branch ch4/git_amend
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    src/ch4/

nothing added to commit but untracked files present (use "git add" to track)
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) ✖ git add src/ch4/
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) ✖ git commit -m "Add ch4 folder"
[ch4/git_amend 03f2f9e] Add ch4 folder
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 src/ch4/git_amend_1.txt
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) git status
On branch ch4/git_amend
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    src/ch4/git_amend_2.txt

nothing added to commit but untracked files present (use "git add" to track)
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) ✖ git add src/ch4/git_amend_2.txt
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) ✖ git commit --amend --no-edit
[ch4/git_amend 6d88963] Add ch4 folder
Date: Sat Aug 20 21:54:38 2022 -0700
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 src/ch4/git_amend_1.txt
create mode 100644 src/ch4/git_amend_2.txt
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_amend) █

```

Figure 4.3 – git commit with the --amend flag

```

→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_a) git status
On branch ch4/git_a
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   src/ch4/git_amend_1.txt
    deleted:    src/ch4/git_amend_2.txt

no changes added to commit (use "git add" and/or "git commit -a")
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_a) ✖ git commit -a -m "Modified git_amend_1.txt" -m "Deleted git_amend_2.txt"
[ch4/git_a 2f40807] Modified git_amend_1.txt
2 files changed, 1 insertion(+)
delete mode 100644 src/ch4/git_amend_2.txt
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_a) git status
On branch ch4/git_a
nothing to commit, working tree clean
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_a) █

```

Figure 4.4 – git commit with the -a flag

## Hands-On Section

### Resolving merge conflicts

Merge conflicts happen when changes have transpired in the same region of a file and Git cannot automatically merge the changes in the file. It is possible that two different engineers are working on the same file and tried to push their changes to the remote repository. In such cases, Git fails the merge processes and forces manual resolution of the merge conflict. Without an IDE, this process can get really messy easily and might end up consuming a lot of the programmer's time. Resolving merge conflicts without an IDE usually involves viewing/editing multiple files through a CLI editor and identifying and fixing the parts of the file that are in conflict. This is a tedious process, but IDEs

provide an interface to deal with conflicts and it is usually completed with a few clicks after manual file inspection.

Let us now look at how to resolve a merge conflict step by step. **Figure 4.5** illustrates a **ch4/merge\_conflict\_branch\_1** branch where the **git\_amend\_2.txt** file was updated and this change was pushed to the remote repository and merged with main through a pull request.

```
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/merge_conflict_branch_1) git status
On branch ch4/merge_conflict_branch_1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   src/ch4/git_amend_2.txt

no changes added to commit (use "git add" and/or "git commit -a")
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/merge_conflict_branch_1) x git add src/ch4/git_amend_2.txt
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/merge_conflict_branch_1) x git commit -m "Update git_amend_2.txt"
[ch4/merge_conflict_branch_1 ed13825] Update git_amend_2.txt
1 file changed, 1 insertion(+)
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/merge_conflict_branch_1) git push origin ch4/merge_conflict_branch_1
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 445 bytes | 445.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'ch4/merge_conflict_branch_1' on GitHub by visiting:
remote:   https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook/pull/new/ch4/merge_conflict_branch_1
remote:
To https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook.git
```

**Figure 4.5 – The git\_amend\_2.txt file updated and merged**

In another branch, let us try to merge the changes from **Figure 4.4**, where the **git\_amend\_2.txt** file was deleted. It is evident that these two changes contradict each other. **Figure 4.6** shows the merged changes being fetched from the **main** branch:

```
priya@Priyas-MacBook-Air:~/Documents/workspace/B19046_Test-Automation-Engineering-Handbook
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_a) git checkout main
Switched to branch 'main'
Your branch is behind 'origin/main' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
→ B19046_Test-Automation-Engineering-Handbook git:(main) git pull
Updating 2f2ed33..6e8dcf8
Fast-forward
 src/ch4/git_amend_2.txt | 1 +
 1 file changed, 1 insertion(+)
→ B19046_Test-Automation-Engineering-Handbook git:(main) git checkout ch4/git_a
Switched to branch 'ch4/git_a'
```

**Figure 4.6 – Fetched merged changes from main**

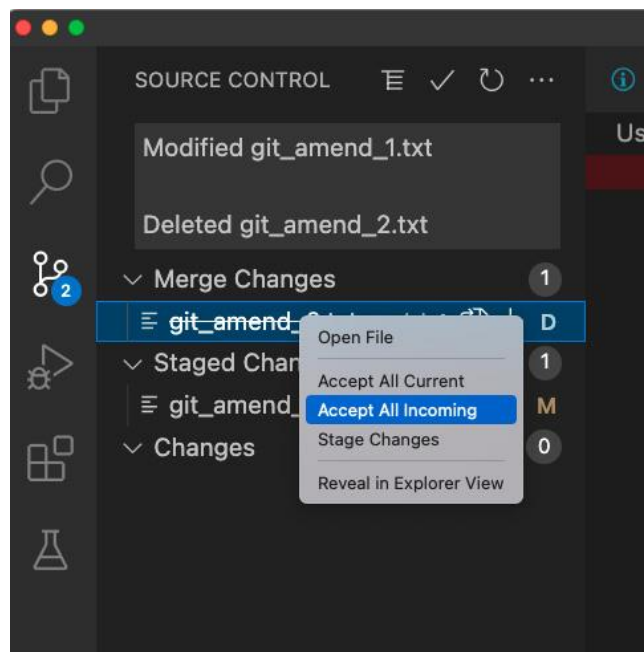
**Figure 4.7** shows the result when the conflicting branch is being rebased with the main. Rebasing is the process of combining a chain of commits and applying it on top of a new base commit. Git automatically creates the new commit and applies it on the current base. Frequent rebasing from the main/master branch helps keep a sequential project history. At this point in the process, the conflicts have to be resolved manually. The engineer has to look through the file and accept or reject others' changes based on the project's needs.

-

```
→ B19046_Test-Automation-Engineering-Handbook git:(ch4/git_a) git rebase main
CONFLICT (modify/delete): src/ch4/git_amend_2.txt deleted in 2f40807 (Modified git_amend_1.txt) and modified in HEAD. Version
HEAD of src/ch4/git_amend_2.txt left in tree.
error: could not apply 2f40807... Modified git_amend_1.txt
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply 2f40807... Modified git_amend_1.txt
→ B19046_Test-Automation-Engineering-Handbook git:(6e8dcf8) x
```

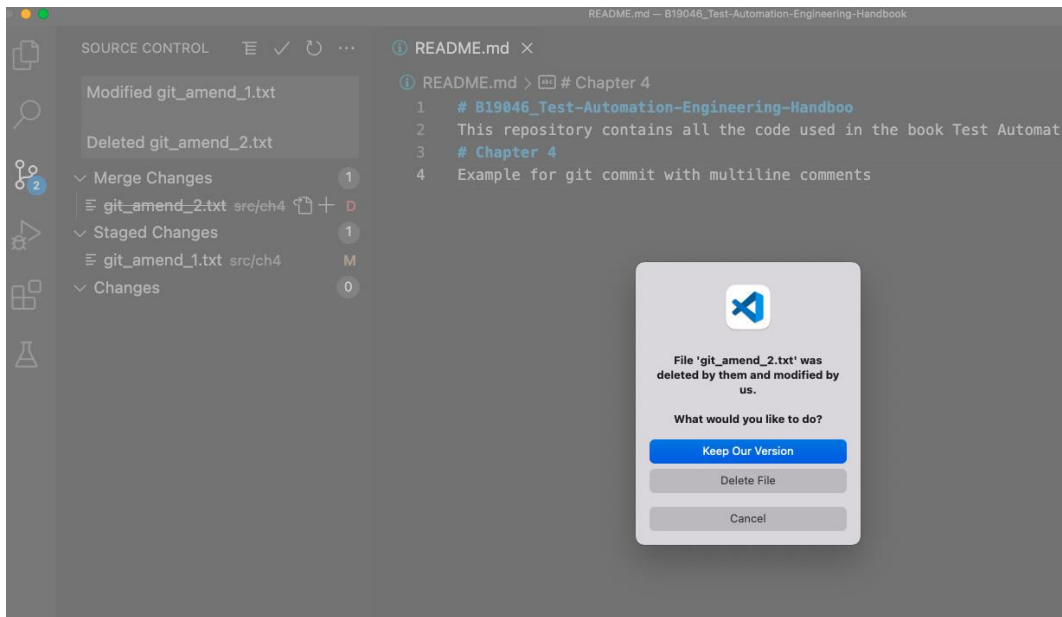
**Figure 4.7 – Merge conflict message**

In this case, let's resolve the merge conflict by accepting the incoming changes from the remote rather than pushing the delete. This is simpler when done through an IDE, as shown in [Figure 4.8](#). On navigating to the source control pane in the IDE and by right clicking the file, the user is shown an option to accept the incoming changes:



**Figure 4.8 – Accept incoming changes**

[Figure 4.9](#) shows the result of staging the accepted changes, which results in the deleted file being retained with the modified contents from the main branch:



**Figure 4.9 – Stage the accepted changes**

Now that the merge conflict is resolved, the rebase can be continued using **git rebase --continue** to complete the commit and merge process subsequently. It is important to remember to pull from remote or other branches (if necessary) before beginning any new work on the local code base. This keeps the branch updated, thereby reducing merge conflicts. It is also vital to have continued communication with the rest of the team when deciding which changes to accept/reject when resolving merge conflicts.

## Downloading and setting up VS Code

In this section, we will be going over the process of downloading VS Code, which can run on macOS, Linux, and Windows. All the code examples cited in the rest of this book will use VS Code. You are free to use an IDE of your choice. Let us now go through the steps for manually installing VS Code on macOS. At the end of this section, you are provided with a shell script to perform this installation via the CLI:

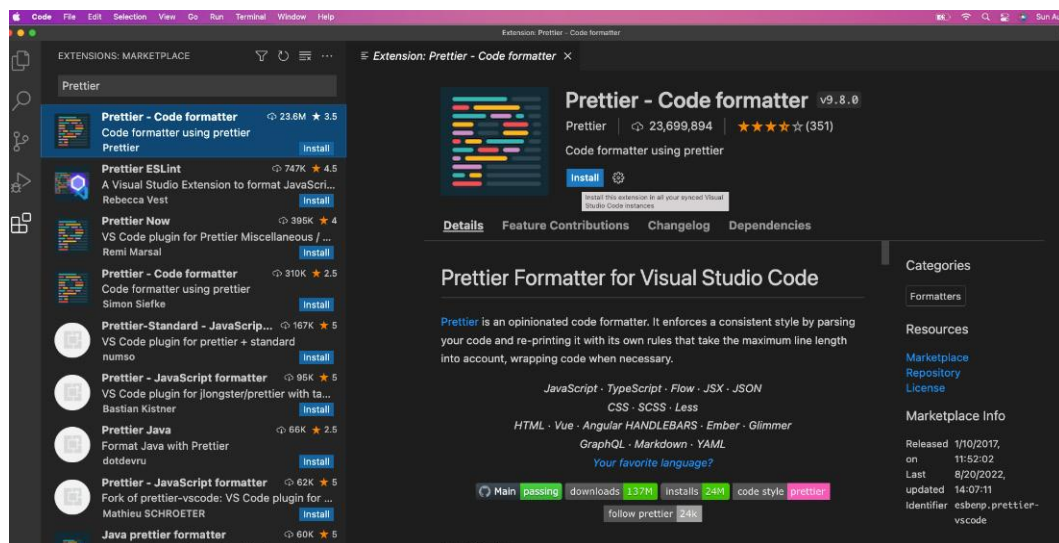
1. Review this link for the necessary system requirements to download and set up VS Code: <https://code.visualstudio.com/docs/supporting/requirements>.
2. Click on the download link on the installation page to download the executable file:
  - macOS and Mac: <https://code.visualstudio.com/docs/setup/mac>
  - Linux: <https://code.visualstudio.com/docs/setup/linux>
  - Windows: <https://code.visualstudio.com/docs/setup/windows>
3. Extract the downloaded archive file and move the **Visual Studio Code** application file to the **Applications** folder.
4. Double-click the **VS Code** icon in the **Applications** folder to open the VS Code IDE.



5. Use the **Settings** option in the **Preferences** menu for additional configuration. VS Code has inherent support for Git provided Git is installed on the machine.

VS Code comes as a lightweight installation and in most cases, engineers would need additional components installed through the **Extensions** option in the **Preferences** menu.

6. Click the **Extensions** option and search for **Prettier**, which is a particularly useful tool for code formatting.
7. Select the **Prettier - Code formatter** extension and click on the **Install** button, as shown in **Figure 4.10**. This should complete the installation of the extension from the Marketplace.



**Figure 4.10 – Installing Prettier - Code formatter**

There are hundreds of helpful extensions available for installation from the Marketplace. These extensions have a wide community of users supporting them, thereby creating a strong ecosystem. Users are strongly encouraged to browse through the available extensions and install them as necessary.

If you prefer using the CLI for installation, the following shell script can be used for installing VS Code on macOS. This example uses Homebrew (<https://brew.sh>) for CLI installation:

```
#!/bin/sh

/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"

brew tap homebrew/cask

brew install --cask visual-studio-code
```

This concludes our section on setting up the IDE. Let us now move on to learning the basics of JavaScript.

## Running a JavaScript program

In this section, let us explore how to run a basic JavaScript program from within our IDE. We will start by installing the **Node.js** runtime environment.

### Installing Node.js

One of the easiest ways to execute a JavaScript program is to run it using Node.js. Node.js is a JavaScript runtime environment created to execute JavaScript code outside of a browser. Use <https://nodejs.org/en/> to download Node.js and navigate through the wizard to complete the installation of the latest stable version of Node.js. Check the installation using the **node --version** command. Alternatively, you can use **Homebrew** on macOS or **winget** on Windows to download it via the CLI. At the time of writing this book, the latest stable node version is **18.12.1**:

```
→ cli_demo node --version
v18.12.1
→ cli_demo
```

Now that we have Node.js installed, let us execute our first JavaScript program.

### Executing the JavaScript program

We will be using the VS Code IDE to execute the programs for the rest of this book. Let us go through step by step to execute a simple **Hello World** program written in JavaScript:

1. The **console.log** command outputs the message in the parenthesis to the console. Create a new **hello\_world.js** file and save it with the contents, **console.log('hello world');**.
2. Open a new Terminal window by selecting the **New terminal** option from the **Terminal** menu. In the Terminal window, navigate to the folder in which the **hello\_world.js** file exists and run the **node hello\_world.js** command. This prints out the text **hello world** in the console:

```
→ B19046_Test-Automation-Engineering-Handbook git:(main) X
cd src/ch4
→ ch4 git:(main) X node hello_world.js
hello world
→ ch4 git:(main) X
```

We have set up our IDE and are able to execute a simple JavaScript program using Node.js. It is now time to get started with the basics.

## Getting to know the JavaScript objects

In JavaScript, an object is a collection of properties. Objects are first initialized using a variable name and assigned to a set of properties. Objects further provide methods to update these properties. Properties can be of any data type, sometimes even other objects. This enables building complex objects in JavaScript. Let us start learning about objects with arrays in the next section.



## Using JavaScript arrays

Arrays are one of the most frequent data structures and are built-in objects in JavaScript. Arrays are nothing but variables that can hold multiple values. The first element of an array is indexed by **0** and the subsequent indices are incremented by 1. The size of the arrays can be changed by adding or deleting elements, and they can contain a mix of data types. Arrays can be initialized by enclosing the elements in square brackets, `[]`. Subsequently, the elements can be accessed by plugging the index within `[]`. Let's look at some commonly used array methods:

- **push(element)**: Adds an element at the end of the array.
- **unshift(element)**: Adds an element at the beginning of the array.
- **pop()**: Removes the last element of the array.
- **indexOf(element)**: Returns the index of the element in the array. Returns **-1** if the element is not found in the array.
- **length()**: Returns the number of elements in the array.

**Figure 4.16** shows these array operations in action:

```
JS arrays.js U X
src > ch4 > JS arrays.js > ...
1 //Arrays
2 const cities = ['San Francisco', 'Los Angeles', 'San Diego', 'Irvine'];
3 console.log(cities);
4 console.log(cities[1]);
5 cities.push('Oakland');
6 console.log(cities);
7 cities.unshift('Sunnyvale');
8 console.log(cities);
9 cities.push(1);
10 console.log(cities);
11 cities.pop();
12 console.log(cities);
13 console.log(cities.indexOf('San Diego'));
14 console.log(cities.indexOf(1));
15 console.log(cities.length);
```

**Figure 4.16 – Array operations**

**Figure 4.17** shows the corresponding outputs. We begin by creating the array and printing it to the console. Then, we add elements to the end and beginning of the array. Subsequently, we work with the indices of the array, and finally, get the length of the array:

```
TERMINAL
→ ch4 git:(main) x node arrays.js
[ 'San Francisco', 'Los Angeles', 'San Diego', 'Irvine' ]
Los Angeles
[ 'San Francisco', 'Los Angeles', 'San Diego', 'Irvine', 'Oakland' ]
[
  'Sunnyvale',
  'San Francisco',
  'Los Angeles',
  'San Diego',
  'Irvine',
  'Oakland'
]
[
  'Sunnyvale',
  'San Francisco',
  'Los Angeles',
  'San Diego',
  'Irvine',
  'Oakland',
  1
]
[
  'Sunnyvale',
  'San Francisco',
  'Los Angeles',
  'San Diego',
  'Irvine',
  'Oakland'
]
3
-1
6
```

Figure 4.17 – Array operation outputs

Unlike a lot of other programming languages, arrays in JavaScript do not throw an **Array Out of Bounds** error when trying to access an index greater than or equal to the length of the array. JavaScript simply returns **undefined** when trying to access the non-existent index array. Arrays come with a wide variety of built-in methods, and I would strongly encourage you to browse through them at [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array). Next, let us look at work with object literals.

### Working with object literals

Object literals allow properties to be defined as key-value pairs. Values of the properties can be other objects as well. Dot (.) or square bracket ([]) notations can be used to retrieve the value of a property. The code snippet demonstrated in [Figure 4.18](#) shows an object and array nested within the **movie** object. In such cases, the object name can be chained subsequently with the call to a nested data structure. Adding an extra property to the object is very simple and looks like a variable assignment. To further our example, it would be **movie['producer']='Danny DeVito'**:

```
JS objects_declare.js U x
src > ch4 > JS objects_declare.js > [x] movie
1 //Objects Declaration and access
2 const movie = { name: 'Pulp Fiction',
3               director: 'Quentin Tarantino',
4               year_of_release: 1994,
5               cast: {
6                 'Vincent Vega': 'John Travolta',
7                 'Jules': 'Samuel Jackson',
8                 'Mia': 'Uma Thurman'
9               },
10              awards: ['Academy Awards 1995', 'Golden Globes 1995', 'Cannes 1994', 'SAG 1995']
11            };
12 console.log(movie);
13 console.log(movie.name, movie.year_of_release);
14 console.log(movie.awards[0]);
15 console.log(movie.cast.Jules)

TERMINAL
→ ch4 git:(main) x node objects_declare.js
{
  name: 'Pulp Fiction',
  director: 'Quentin Tarantino',
  year_of_release: 1994,
  cast: {
    'Vincent Vega': 'John Travolta',
    Jules: 'Samuel Jackson',
    Mia: 'Uma Thurman',
    Butch: 'Bruce Willis'
  },
  awards: [
    'Academy Awards 1995',
    'Golden Globes 1995',
    'Cannes 1994',
    'SAG 1995'
  ]
}
Pulp Fiction 1994
Academy Awards 1995
Samuel Jackson
```

Figure 4.18 – Objects

**const** prevents reassigning the variable but does not prevent modifying values within an object:

```
const a = { message: "hello" };
a.message = "world"; // this will work
```

Having learned the basics of JavaScript objects, let us now look at how to destructure one.

### Destructuring an object

Object destructuring is used in JavaScript to extract property values and assign them to other variables. It has various advantages such as assigning multiple variables in a single statement, accessing properties from nested objects, and assigning a default value when a property doesn't exist. We use the same example as in the previous section but as shown in [Figure 4.19](#), we destructure the **movie** object by specifying the **name** and **awards** variables within **{}**. On the right-hand side of the expression, we specify the **movie** object. We could also assign them to a variable within the curly brackets to fetch data from nested objects. Object destructuring was introduced in ECMAScript 6 and prior to this, extracting and assigning properties in such a way required a lot of boilerplate code.

```
JS objects_destructure.js U X
src > ch4 > JS objects_destructure.js > awards
1 //Objects Declaration and access
2 const movie = { name: 'Pulp Fiction',
3               director: 'Quentin Tarantino',
4               year_of_release: 1994,
5               cast: {
6                 'Vincent Vega': 'John Travolta',
7                 'Jules': 'Samuel Jackson',
8                 'Mia': 'Uma Thurman'
9               },
10              awards: ['Academy Awards 1995', 'Golden Globes 1995', 'Cannes 1994', 'SAG 1995']
11            }
12 const {name, awards} = movie;
13 console.log(name, awards);
14 const {cast: {Jules}} = movie;
15 console.log(Jules);

TERMINAL
→ ch4 git:(main) x node objects_destructure.js
Pulp Fiction [
  'Academy Awards 1995',
  'Golden Globes 1995',
  'Cannes 1994',
  'SAG 1995'
]
Samuel Jackson
→ ch4 git:(main) x
```

Figure 4.19 – Object destructuring

Let us next work with an array of objects.

### Arrays of objects

Working with arrays of objects is crucial for quality engineers as a lot of times, the API response payloads in JSON format have multiple objects embedded within an array and their format is identical to JavaScript object literals. Let us consider an example where multiple **movie** objects are embedded within the **movies** array. We could use the **JSON.stringify** method to create a JSON string. The code snippet in [Figure 4.20](#) demonstrates how to access nested elements in an array of objects and how to create a JSON string from a JavaScript object:



```
JS loops_for.js U X
src > ch4 > JS loops_for.js > [e]i
1   for (let i=0; i<10; i++){
2     console.log(`For loop iteration: ${i}`);
3   }

TERMINAL

→ ch4 git:(main) x node loops_for.js
For loop iteration: 0
For loop iteration: 1
For loop iteration: 2
For loop iteration: 3
For loop iteration: 4
For loop iteration: 5
For loop iteration: 6
For loop iteration: 7
For loop iteration: 8
For loop iteration: 9
→ ch4 git:(main) x
```

Figure 4.21 – A simple for loop

### Note

One of the common pitfalls while looping through an array is to accidentally exceed the last index of the array. The condition to check the array should be `i<array.length` or `i<=array.length-1`.

The **while** loop operates similarly to the **for** loop, but we set the variable outside of the loop. It is a common mistake to miss the increment or incorrectly specify the condition. Doing so would result in an infinite loop. [Figure 4.22](#) shows the same logic in the **while** loop:

```
JS loops_while.js U X
src > ch4 > JS loops_while.js > ...
1   let i=0;
2   while (i<10){
3     console.log(`While loop iteration: ${i}`);
4     i++;
5   }

TERMINAL

→ ch4 git:(main) x node loops_while.js
While loop iteration: 0
While loop iteration: 1
While loop iteration: 2
While loop iteration: 3
While loop iteration: 4
While loop iteration: 5
While loop iteration: 6
While loop iteration: 7
While loop iteration: 8
While loop iteration: 9
→ ch4 git:(main) x
```

Figure 4.22 – A simple while loop

Now, let us loop through the array of objects we created in [Figure 4.20](#). For this purpose, we will use the **for..of** loop, which is much more readable than the regular **for** loop. In [Figure 4.23](#), we have the code snippet that iterates over each of the objects in the **movies** array and prints the name and director. We create a temporary variable to hold the current entry of the array in the loop and use that variable to print the properties:

```
JS for_of.js U X
src > ch4 > JS for_of.js > ...
1  const movies = [
2    {
3      id:1,
4      name: 'Pulp Fiction',
5      director: 'Quentin Tarantino'
6    },
7    {
8      id:2,
9      name: 'Inception',
10     director: 'Christopher Nolan'
11   },
12   {
13     id:3,
14     name: 'The Shawshank Redemption',
15     director: 'Frank Darabont'
16   }
17 ];
18 for(let movie of movies){
19   console.log(`Director of ${movie.name} is ${movie.director}`);
20 }
21
TERMINAL
→ ch4 git:(main) x node for_of.js
Director of Pulp Fiction is Quentin Tarantino
Director of Inception is Christopher Nolan
Director of The Shawshank Redemption is Frank Darabont
→ ch4 git:(main) x █
```

Figure 4.23 – A for..of loop

We looked at some very useful examples of loops in this section. Let us move on next to conditional statements.

### Using the conditional statements

Conditional statements are used to separate the logic into different code blocks based on one or more conditions. The most common conditional statement is the **if...else** statement. This is better understood by referring to the code snippet in [Figure 4.24](#). Here, we use conditional statements to assign a grade based on the student's score. We start with the **if** statement and check for the highest grade and then use a series of **else if** statements followed by the **else** statement to check for any score less than **60**. The **else if** statements are useful to extend the logic to include additional conditions. It is important to remember that in the absence of an **else** statement, JavaScript ignores the conditional code block when the **if** condition is not **true**:



```

JS conditionals.js U X
src > ch4 > JS conditionals.js > ...
1 //if...else if...else
2 const score = 79
3 let grade;
4 if (score>90){
5     grade = 'A';
6 }
7 else if (score> 80){
8     grade = 'B';
9 }
10 else if (score>70){
11     grade = 'C';
12 }
13 else if (score>60){
14     grade = 'D';
15 }
16 else {
17     grade = 'F';
18 }
19 console.log(`Student's score is ${grade}`);
20
TERMINAL
→ ch4 git:(main) x node conditionals.js
student's score is C
→ ch4 git:(main) x █

```

**Figure 4.24 – Conditional statements**

**Table 4.1** summarizes the most common conditional operators in JavaScript:

Operator	Description
==	Equal to
===	Equal value and equal type
!=	Not equal to
!==	Not equal value and not equal type
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

**Table 4.1 – Conditional operators**

In the next section, let us learn about JavaScript functions and how to use them to make the code more reusable.



## Figures

```
JS comments.js U X
src > ch4 > JS comments.js
1 // Single Line comment
2 /* Multi
3 Line
4 Comment*/
```

Figure 4.11 – Adding comments in a JavaScript file

```
JS var_data.js U X
src > ch4 > JS var_data.js > ...
1 //let, const
2
3 let temperature=75;
4 const city = 'San Francisco';
5 console.log(city);
6 console.log(temperature);
7 temperature=80;
8 console.log(temperature);
9 city = 'San Diego';
10
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
75
80
/Users/priya/Documents/workspace/B19046_Test-Automation-Engineering-Handbook/src/ch4/var_data.js:
9
city = 'San Diego'
  ^
TypeError: Assignment to constant variable.
    at Object.<anonymous> (/Users/priya/Documents/workspace/B19046_Test-Automation-Engineering-Handbook/src/ch4/var_data.js:9:6)
    at Module._compile (node:internal/modules/cjs/loader:1126:14)
    at Object.Module._extensions..js (node:internal/modules/cjs/loader:1180:10)
    at Module.load (node:internal/modules/cjs/loader:1004:32)
    at Function.Module._load (node:internal/modules/cjs/loader:839:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:17:47
→ ch4 git:(main) x []
```

Figure 4.12 – Variable types, let and const

```
JS primitives.js U X
src > ch4 > JS primitives.js > ...
1  const city = 'San Francisco'; //string
2  const population = 815000; //number
3  const crime_rate = 50.72; //number
4  const isWarm = false; //boolean
5  const x = null; //null
6  let y; //undefined
7  console.log(typeof(city))
8  console.log(typeof(population))
9  console.log(typeof(crime_rate))
10 console.log(typeof(isWarm))
11 console.log(typeof(x))
12 console.log(typeof(y))

TERMINAL
→ ch4 git:(main) x node primitives.js
string
number
number
boolean
object
undefined
→ ch4 git:(main) x
```

Figure 4.13 – Data types

```
JS strings.js U X
src > ch4 > JS strings.js > ...
1  const city = 'San Francisco';
2  const population = 815000;
3  //concatenation
4  sfo = `${city} has a population of ${population}`;
5  console.log(sfo);
6

TERMINAL
→ ch4 git:(main) x node strings.js
San Francisco has a population of 815000
→ ch4 git:(main) x
```

Figure 4.14 – String concatenation

```
functions_area.js — B19046_Test-Automatio
JS functions_area.js U X
src > ch4 > JS functions_area.js > compute_area
1  function compute_area() {
2      if (arguments.length==1){
3          area = arguments[0]*arguments[0];
4      }
5      else if (arguments.length==2){
6          area = arguments[0]*arguments[1];
7      }
8      else {
9          area = 'Invalid number of arguments';
10     }
11     return area;
12 }
13 console.log(compute_area(10));
14 console.log(compute_area(10,20));
15 console.log(compute_area());

TERMINAL
→ ch4 git:(main) x node functions_area.js
100
200
Invalid number of arguments
→ ch4 git:(main) x
```

Figure 4.25 – Functions

## Links

The MDN docs for JavaScript, located at <https://developer.mozilla.org/en-US/docs/Web/JavaScript>, can be used as a standard reference to get additional details on any concepts.

You might have noticed from [Figure 4.13](#) that `null` has a `typeof` object. This is considered a bug in JavaScript. The explanation for this can be found in this link: [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof#typeof\\_null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/typeof#typeof_null).

# Chapter 5

## Technical requirements

In this chapter, we will continue using Node.js (version 16.14.2), which we installed in [Chapter 4, Getting Started with the Basics](#). We will also be using **node package manager (npm)** to install Cypress version 11.2.0. All the code examples illustrated in this chapter can be found under the **ch5** folder at <https://github.com/PacktPublishing/B19046-Test-Automation-Engineering-Handbook/tree/main/src/ch5>.

## Hands-on Sections

### Installing and setting up Cypress

Let us now run through a detailed step-by-step installation and setup process for Cypress:

1. In [Chapter 4](#), we installed Node.js, which is a runtime environment for JavaScript. Node.js installation comes with a default and extremely useful package manager called npm. [Figure 5.1](#) shows how to check the version of npm installed on your machine:

```
→ ch5 npm -v
8.15.0
→ ch5 █
```

**Figure 5.1 – Checking the installed npm version**

2. Let us next create an empty project to install Cypress and further explore its features. Run **npm init -y** in an empty folder (preferably named **app**) in your local directory to create a **package.json** file. [Figure 5.2](#) shows the corresponding output with the contents of the file:

```
→ B19046-Test-Automation-Engineering-Handbook git:(main) cd src/ch5/app
→ app git:(main) npm init -y
Wrote to /Users/priya/Documents/workspace/B19046-Test-Automation-Engineering-Handbook/src/ch5/app/package.json:

{
  "name": "app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}

→ app git:(main) █
```

**Figure 5.2 – npm init**

Note

**npm init <initializer>** is used to set up new or existing packages. If **<initializer>** is omitted, it will create a **package.json** file with fields based on the existing dependencies in the project. The **-y** flag is used to skip the questionnaire.

**package.json** is the primary configuration file for npm and can be found in the root directory of the project. It helps to run your application and handle all the dependencies.

- Execute **npm install cypress** in the root of our **src/ch5/app** project. This creates a **node\_modules** folder, which contains a chain of dependencies required by the package being installed (Cypress). **Figure 5.3** shows the output of this step, with **package.json** showing Cypress installed. It is generally considered good practice to save the testing libraries in the **devDependencies** section of the **package.json** file using the **npm install cypress --save-dev** command:

```
1 {
2   "name": "app",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \\"Error: no test specified\\" && exit 1"
8   },
9   "keywords": [],
10  "author": "",
11  "license": "ISC",
12  "devDependencies": {
13    "cypress": "^11.2.0"
14  }
15 }
```

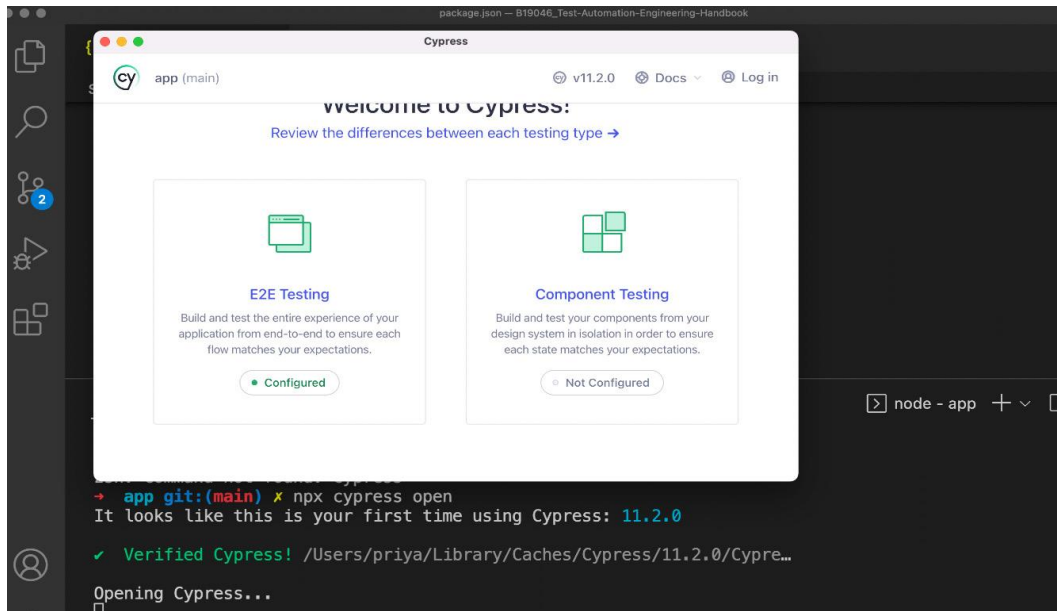
**Figure 5.3 – npm install cypress**

- Create an **index.html** file in the root, as shown in **Figure 5.4**, to serve as the primary loading page for our application. Also, create an empty **index.js** file:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <meta charset="utf-8">
5     <title>Cypress</title>
6   </head>
7   <body>
8     <h1>Learning Cypress through Packt</h1>
9   </body>
10 </html>
```

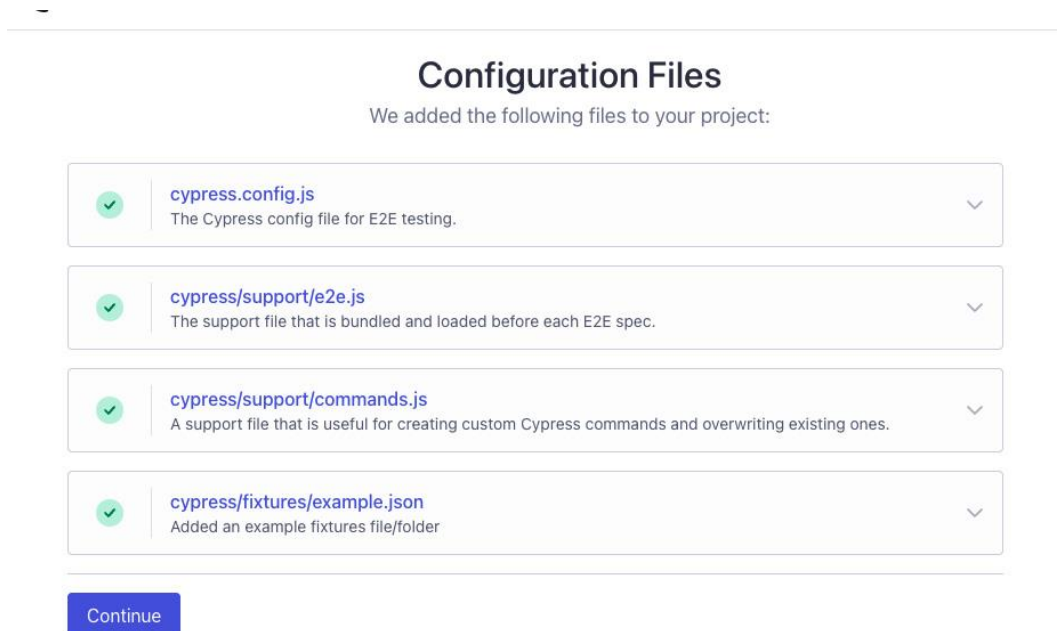
**Figure 5.4 – Creating an index.html file**

5. Execute `npx cypress open` to open Cypress. This command opens the executable file from the `node_modules/bin` directory. **Figure 5.5** illustrates the output where the in-built browser is opened:



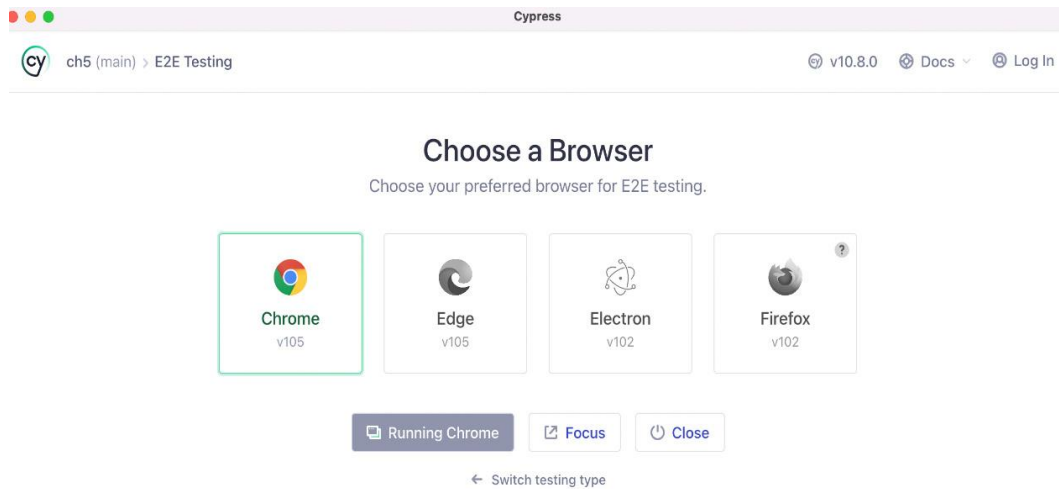
**Figure 5.5 – npx cypress open command**

6. Now, click on the **E2E Testing** option, which adds some configuration files to the repository, and hit **Continue** in the next modal, as shown in **Figure 5.6**:



**Figure 5.6 – Cypress config modal**

7. In the next modal, select the preferred browser for E2E testing. I have selected **Chrome** in this case, as shown in **Figure 5.7**, and it opens the browser in a new window:



**Figure 5.7 – Choosing a preferred browser**

This completes the installation of Cypress and gets it ready to a point where we can start writing our own tests. In the next section, let us start working on our first test and review some additional configurations.

## Creating your first test in Cypress

A test in Cypress is commonly referred to as a **spec**, which stands for **specification**. We will be referring to them as specs for the remainder of this chapter. Let us begin by understanding how to write arrow functions and callback functions in JavaScript.

### Creating arrow functions in JavaScript

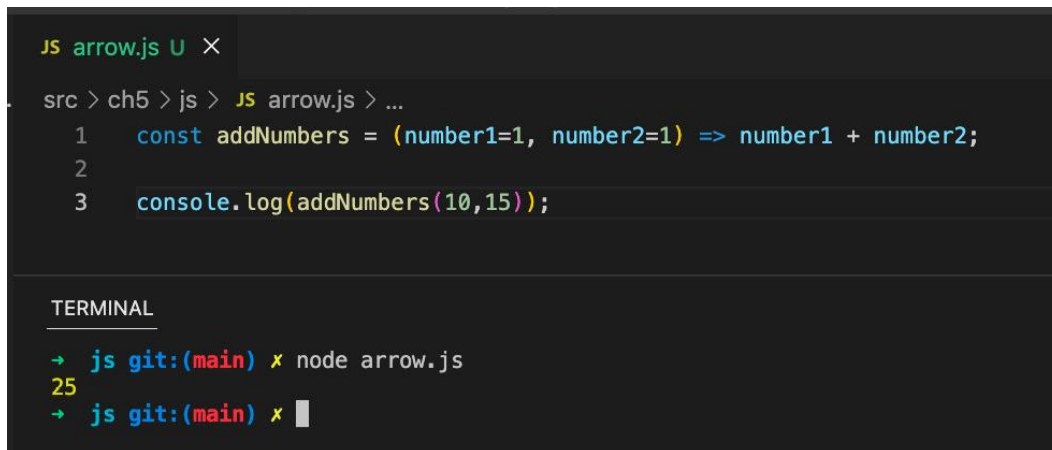
Arrow functions are extremely handy, and they clean things up quite a bit. They were introduced in the **ECMAScript 6 (ES6)** version. The code snippet in **Figure 5.8** shows a simple function to add two numbers. It takes two parameters and returns the sum. Let us turn this into an arrow function:

```
JS arrow.js U X
src > ch5 > js > JS arrow.js > addNumbers
1 function addNumbers(number1=1, number2=1) {
2   return number1 + number2;
3 }
4
5 console.log(addNumbers(10,15))

TERMINAL
→ js git:(main) x node arrow.js
25
→ js git:(main) x
```

**Figure 5.8 – Function to add two numbers**

Instead of using the `function` keyword, we name it like a variable and use an equals sign to assign it to the body of the function. After the parameters, we use a symbol called `fat arrow` (`=>`). In the case of one-liner functions, we can further simplify them by removing the curly braces surrounding the function body. We could also remove the `return` keyword, and it still returns the computed value. If we have only one parameter, we could lose the parentheses around the parameters as well. It would look like this: `const addNumbers = number1 => number1 + 5`. An example is shown in [Figure 5.9](#). This works very neatly in the case of array iterations. Let's say we have an array of movies, and we would like to iterate over them and print the names of all the movies. This can be neatly done in a single line by using `movies.forEach(movie) => console.log(movie, name)` arrow functions:



```
JS arrow.js U X
src > ch5 > js > JS arrow.js > ...
1  const addNumbers = (number1=1, number2=1) => number1 + number2;
2
3  console.log(addNumbers(10,15));

TERMINAL
→ js git:(main) x node arrow.js
25
→ js git:(main) x
```

**Figure 5.9 – Arrow function with two parameters**

Let us next learn about callback functions in JavaScript.

### Creating callback functions in JavaScript

In JavaScript, since functions are set up as objects, we can pass and call other functions within a function. A function that is passed as a parameter to another function is called a **callback** function.

Let us use the `setTimeout()` function to understand callback functions. The `setTimeout()` function calls a method after a specified wait in milliseconds. For example, `setTimeout(() => console.log('hello!'), 5000)` would print the message after a wait of 5 seconds. Let us now create an arrow function to accept and print a message to the console, as shown in [Figure 5.10](#). Let us call this function `printMessage()`, with a delay of 5 seconds by passing it as a parameter to the `setTimeout()` function, making it a callback function:



```
JS callback.js U X
src > ch5 > js > JS callback.js > ...
1  const printMessage = message => console.log(message);
2  printMessage('hello!');
3  setTimeout(printMessage, 5000, 'hello delayed...');

TERMINAL
→ js git:(main) x node callback.js
hello!
hello delayed...
→ js git:(main) x
```

Figure 5.10 – Callback functions

We could also pass in the whole body of the arrow function instead of the name, as shown in [Figure 5.11](#). These are called anonymous functions since they do not have a name and are declared at runtime:

```
JS callback.js X
src > ch5 > js > JS callback.js > ...
1  console.log('hello!');
2  setTimeout(message => console.log(message), 5000, 'hello delayed...');

TERMINAL
→ js git:(mani/update_readme) x node callback.js
hello!
hello delayed...
→ js git:(mani/update_readme) x
```

Figure 5.11 – Anonymous callback functions

A key advantage of using callback functions is that it enables the timing of function calls and assists in writing and testing asynchronous JavaScript code. There are many instances in the modern web application where there is a need to make external API calls and resume the current task rather than wait for the external call to complete. Callback functions come in handy here to unblock the execution of the main block of code. It is important to use callbacks only when there is a need to bind your function call to something potentially blocking, to facilitate asynchronous code execution.

With this additional knowledge about functions in JavaScript, let us now commence writing our first spec.

### Writing our first spec

It is a good practice to organize all tests under a single folder in your repository. If there are more tests, then they can be categorized under a parent test folder. Create a folder named **e2e** under **src/ch5/app/cypress**. Now, create a test file, as shown in [Figure 5.12](#):

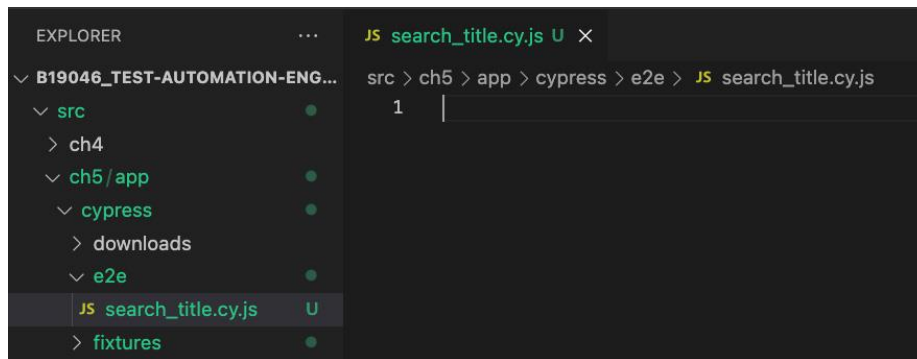


Figure 5.12 – Creating a test file

Our first test searches for the string **quality** in the search box on the home page of <https://www.packtpub.com>. Then, it verifies the search result page by looking for the **Filter Results** string. Copy and paste the code from the [https://github.com/PacktPublishing/B19046\\_Test-Automation-Engineering-Handbook/blob/main/src/ch5/app/cypress/e2e/search\\_title.cy.js](https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook/blob/main/src/ch5/app/cypress/e2e/search_title.cy.js) GitHub link into the test file.

Let us now examine the structure of a Cypress spec.

### Becoming familiar with the spec structure

Every test framework requires its tests to be written in a specific language and format. Cypress is no different, and as we already know, it uses JavaScript. Cypress comes packed with its own set of functions under the global **cy** object. It also utilizes the **describe-it-expect** format using bundled libraries from **Mocha** and **Chai** frameworks. Additionally, an assertions framework using **expect** with command chaining is also supported to complete granular validations. The **describe** block captures the high-level purpose of the spec, and the **it** block adds specific implementation details of the test. Note that both the **describe** and **it** blocks accept callback functions as their second parameter, and they are defined as arrow functions. This is a common syntax, and you will see this more often in modern JavaScript code. Please be wary of braces, semicolons, and parentheses. It is recommended to use an extension such as **Prettier** to assist with the formatting as it could get messy pretty quickly.

We have started with a comment that describes what is being achieved in this spec. Cypress internally uses the **TypeScript** compiler, and the reference tag is used to equip autocompletion with only Cypress definitions. The **beforeEach** block, as the name suggests, runs before every **it** block. It usually contains the prerequisite steps to execute the individual **it** blocks. Here, we use the **visit** command to access the **Packt Publishing** website within the **beforeEach** block. Then, the **it** block drills down to which action is performed in the spec. If we end up adding more **it** blocks to this spec, the **visit** command would be executed before the beginning of each **it** block. This is a simple spec but it captures the necessary structure of a spec written in Cypress.

Next, let us examine how to execute our first spec.

### Executing our first spec

Cypress comes packed with a powerful visual runner tool to assist in test execution. This can be used when users have a need to inspect tests visually during runtime. Another option is to execute tests via the CLI for quicker results and minimal test execution logs. In this section, we will survey both ways to execute tests in Cypress.

## Using the command line

Using the command line to execute tests is always a quick and easy option. It usually helps when you are not interested in looking at the frontend aspects of the test execution. The `npm run cypress/e2e/search_title.cy.js` command can be used to execute an individual spec in Cypress. The `-s` flag stands for spec, followed by the name of the file. Without the `-s` flag, the `npm run cypress` command would execute all the specs found in the current project. [Figures 5.13](#) and [5.14](#) illustrate the output of the command-line execution of our first spec. [Figure 5.13](#) shows the output of the CLI, with a listing of actions performed on the UI:

```
→ app git:(main) ✗ npx cypress run -s cypress/e2e/search_title.cy.js

=====

(Run Starting)

| Cypress:      11.2.0
| Browser:      Electron 106 (headless)
| Node Version: v18.12.1 (/usr/local/bin/node)
| Specs:         1 found (search_title.cy.js)
| Searched:     cypress/e2e/search_title.cy.js
|
|-----|

Running: search_title.cy.js (1 of 1)

Vist packt home page,
  ✓ search for a title and and click submit (11003ms)
  ✓ by intercepting the recent items GET call (1802ms)

2 passing (14s)
```

Figure 5.13 – CLI test execution

[Figure 5.14](#) shows a summary of the tests executed, with a breakdown of the results:

```
=====

(Run Finished)

Spec                                Tests  Passing  Failing  Pending  Skip
ped
|-----|
| ✓ search_title.cy.js              00:13   2        2        -        -
|-----|
| ✓ All specs passed!              00:13   2        2        -        -
|-----|

→ app git:(main) ✗
```

Figure 5.14 – CLI test execution (continued)

Next, let us next explore the visual test runner for executing our spec.

## Using the visual test runner

Cypress comes with an extremely insightful and detailed test runner and provides quite a bit of debugging data for tests being executed. To utilize this mode, we can start with the `npm run cypress:open` command, which opens up a series of modals. The first modal requires the selection of the type of test, as previously shown in [Figure 5.5](#). The second modal, as seen earlier in [Figure 5.7](#), provides an option to select a browser against which the test can be run. The third modal lists all specs in the project and shows some additional metadata about the specs and their recent runs:

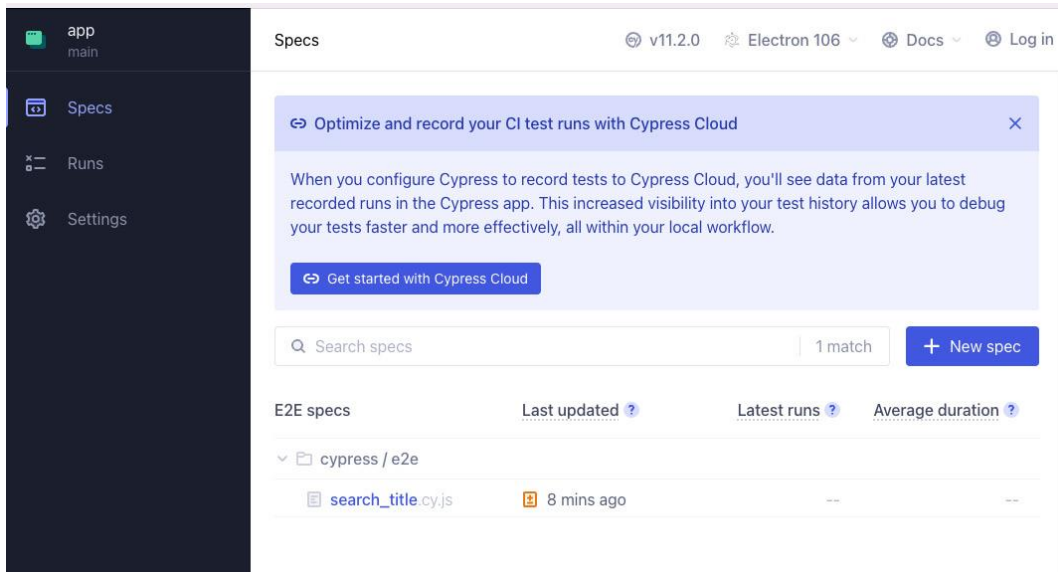


Figure 5.15 – Test selection modal

Test execution begins when the user clicks on the test, as shown in [Figure 5.15](#). This opens a new browser that shows the actual steps being executed. The left pane shows the various frontend and backend calls being made while executing this test. [Figure 5.16](#) shows a view of the test execution. Cypress offers a live-reloading feature out of the box using the `cypress-watch-and-reload` package. Whenever there is a change in the code, the test is rerun automatically and the view, as shown in [Figure 5.16](#), reloads live:

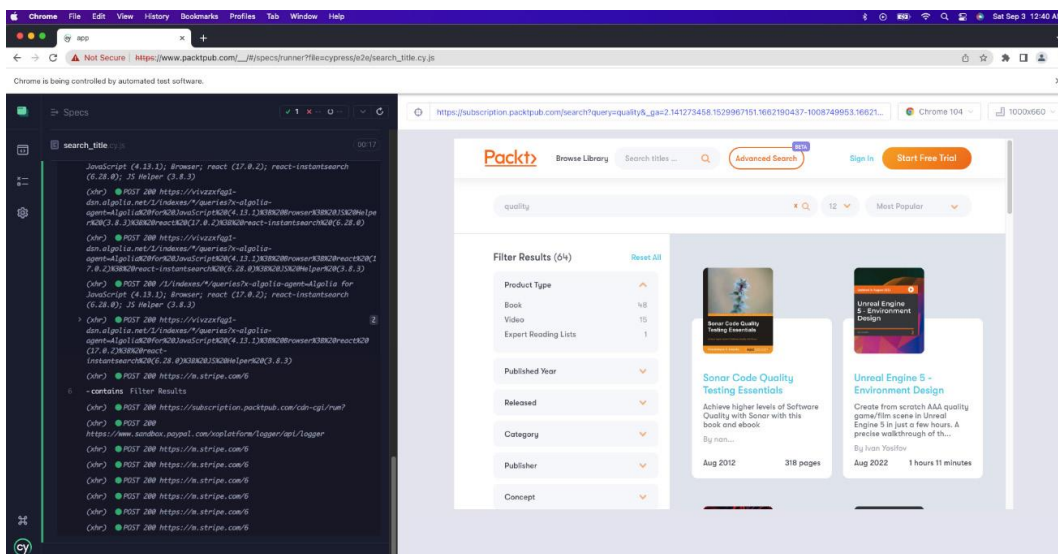


Figure 5.16 – Visual test runner

This view also allows users to view the stack trace of errors and provides options to navigate between test runs and settings. The browser on the right pane can be used like any other browser window to inspect and debug using the developer tools. Users are strongly encouraged to further explore the features that this test runner has to offer.

In the next section, let us gain a deeper understanding of using selectors in Cypress.

## Employing selectors and assertions

Selectors are identifiers for elements in the **Document Object Model (DOM)**. We have various ways to identify elements, such as using their class, name, type, and so on. Every test framework has its own custom commands to make the code clear and concise. Cypress provides users with an efficient interface to look for selectors and comes with standard support for all selectors. Let us continue using our first spec to dig deeper into utilizing selectors.

### Working with selectors

**cy.get** is the primary function to search for elements in the DOM. In our **search\_title.cy.js** test file, we have used **.input-text**, which identifies the element with the **input-text** class name and sets a value in it. We have also used **[aria-label="Search"]** to look for the **Search** button. This is an example of an attribute search. We are essentially finding an element with the value of the **aria-label Search** attribute and clicking on it. **id** and **data** are other reliable attributes for identifying elements in the DOM. It is important to remember to use square brackets when employing attributes in selectors. This raises the question of what kind of selector to use in each case. The answer would be to employ the simplest one that uniquely identifies the required element on the DOM.

Cypress assists users here by providing a selector playground feature that automatically populates the selector. Let us rerun our first spec using a visual test runner and reach the execution page, as shown in **Figure 5.16**. Now, refer to **Figure 5.17** and click the circular toggle icon. This opens the selector playground where the user can type the selector or use the arrow icon for Cypress to automatically populate it. Now, the user can use the browser to click on the required UI element and get the unique selector right away. The user can also play around with other options and validate their correctness by plugging them into the textbox:

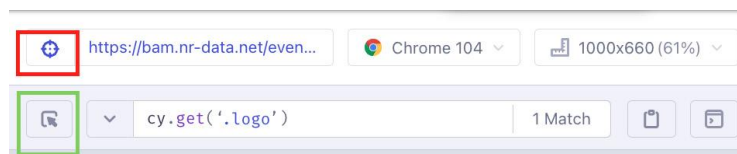


Figure 5.17 – Selecting a playground

To write efficient automation scripts, it is vital to know which selectors are reliable and perform better in a given situation. Imagine a test automation project with 5,000 test cases and all of them find a link using the worst-performing selector, which has a lag of 50 milliseconds relative to the best-performing selector. That would make the test suite slower by 250,000 milliseconds for every run. This would impact feedback times immensely when considering hundreds of CI pipeline runs over a few days.

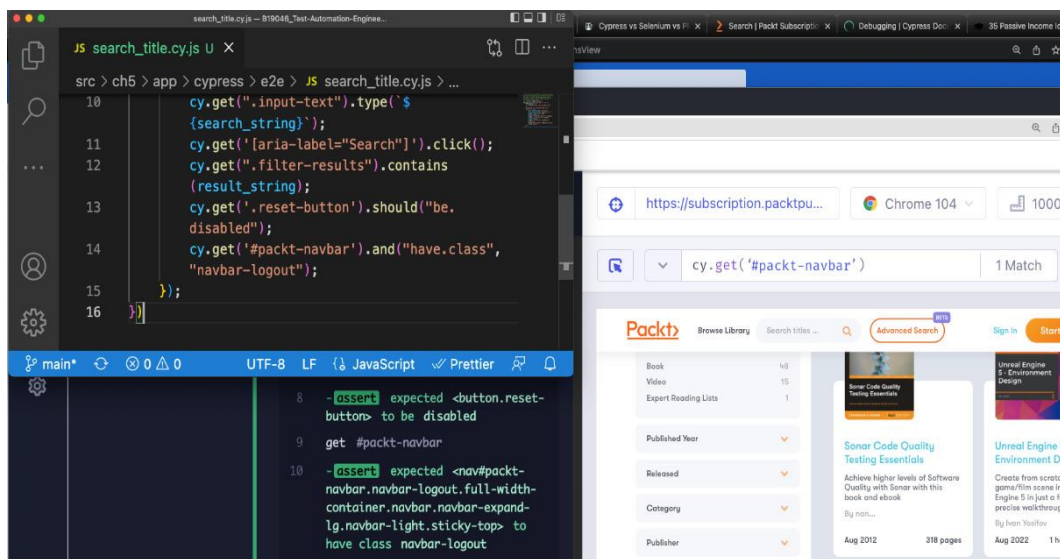
XPath selectors identify the target element by navigating through the structure of the HTML document, while CSS selectors use a string to identify them. Direct CSS selectors using an element's ID or class usually perform better than XPath selectors. Using an ID selector is often the most reliable way of selecting an HTML element. It helps to analyze the elements to understand whether they are dynamic and which selectors would be supported across different browsers, and based on that, decide on a selector strategy. It usually takes a bit of troubleshooting to arrive at an efficient pattern of selectors working for a specific application and a team.

Let us now learn about the available assertion options.

## Asserting on selectors

Assertions give us a way to perform validations on various UI elements. They are usually chained to the command with selectors and work together to identify an element and verify it. **should** is the primary function utilized on assertions, and it works with a myriad of arguments.

Let us update our first spec to add some assertions. We have earlier used the **contains** function in our spec to assert a partial string in the search results page. **Figure 5.18** shows the assertions in action. Next, we add an assertion on the **Reset** button to validate that it is disabled. In the following line, we get the navigation bar element by the **id** attribute and chain it with an assertion that validates the class name:



**Figure 5.18 – Assertions for the navbar and Reset button**

Let us add another assertion before entering the search string to validate it is empty using the **have.value** parameter. **Figure 5.19** demonstrates this assertion:



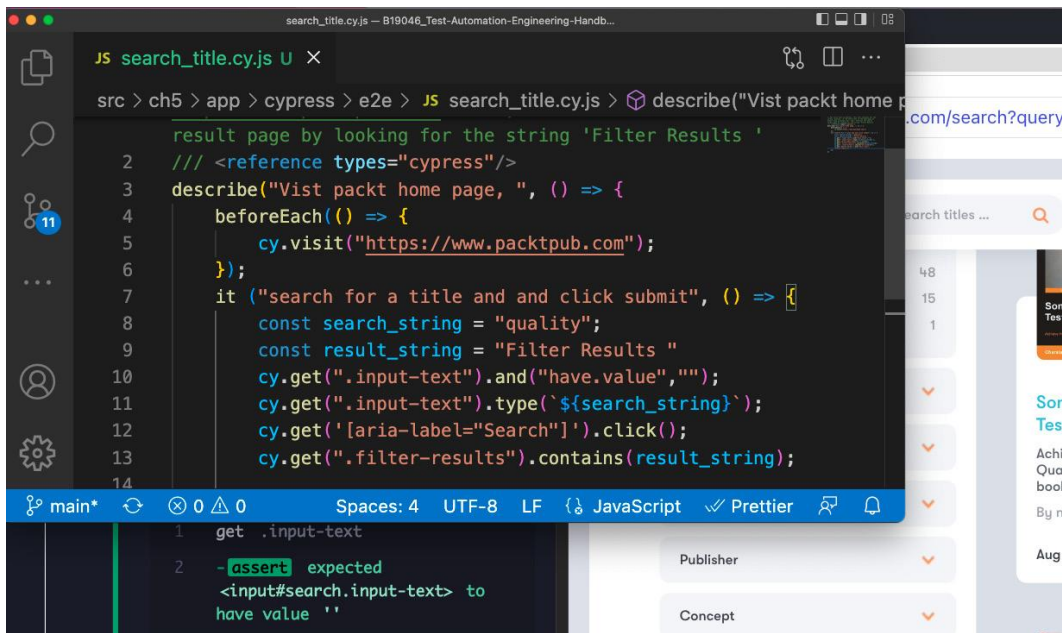


Figure 5.19 – Asserting empty value in a textbox

Cypress comes with very good documentation (<https://docs.cypress.io/api/table-of-contents>) and users are encouraged to use it as a reference to be aware of the various available options. So far, we've worked on identifying and asserting UI elements. In the next section, let us work with API calls in Cypress.

## Intercepting API calls

Cypress lets users work with underlying API requests and stub responses where necessary. Let us analyze the API calls when loading the **Packt Publishing** home page and try to stub one of the responses. **cy.intercept()** is the command used to work with API calls, and it offers a wide variety of parameters. For this example, we will be using the **routeMatcher** and **staticResponse** arguments. We add a second **it** block to intercept the underlying API call and specify the type of HTTP call, URL, and a predefined response as parameters, as shown in **Figure 5.20**:

```
40  });
41  cy.intercept(
42    "GET",
43    "https://subscription.packtpub.com/api/subscription/getrecentitems?offset=0&limit=4",
44    staticResponse
45  );
```

Figure 5.20 – cy.intercept call

The value of the static response parameter can be obtained using the **Network** tab of the developer tools to get the actual response for the API call. This is illustrated in **Figure 5.21**. By passing this in as the **staticResponse** parameter, the **GET** call on this URL will always return the stubbed response instead of the original:

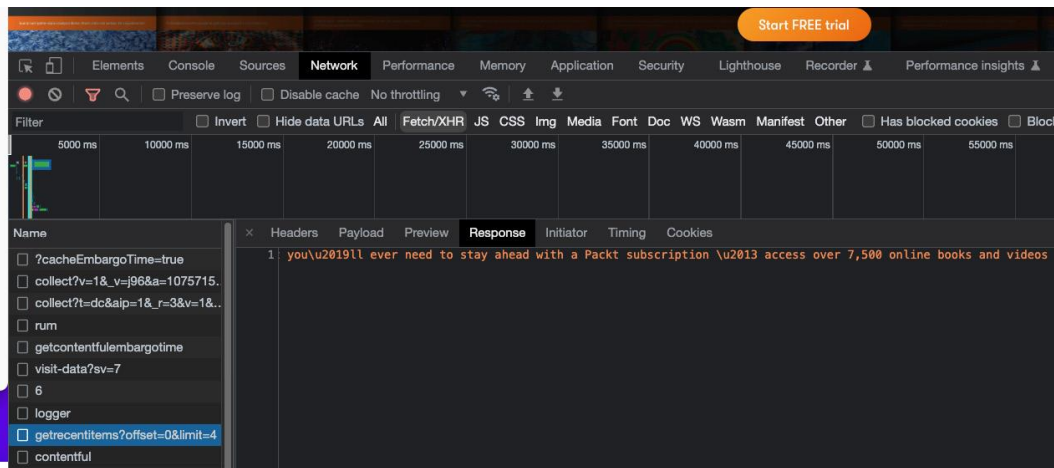


Figure 5.21 – API payload for stubbing

Figure 5.22 demonstrates the result of the intercept command in action:

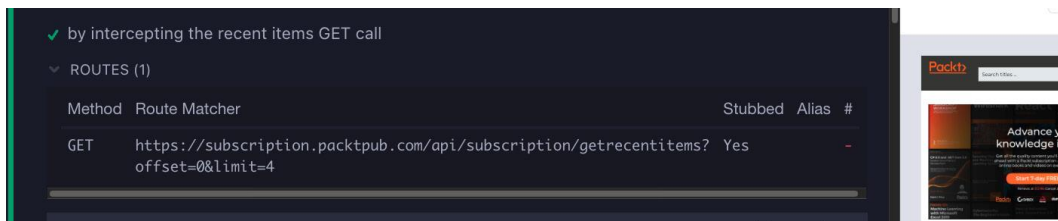


Figure 5.22 – Intercept results

This empowers the user to test the underlying API calls for different payloads and validate the application behavior in each case. This also saves resources in cases where some of these are expensive API calls. This is just one way to handle API calls with Cypress, and there are a variety of options available to explore. In the next section, let us quickly review some additional configurations that might be helpful with setup and validation.

## Additional configurations

Let us review a few additional configurations in this section to build stable and efficient specs:

- The first configuration is a Git feature and not specific to Cypress. Adding a **.gitignore** file is a general necessity for all projects. Add it to the **src/ch5/app** root folder to ignore files we don't want Git to track on our local directory. As shown in Figure 5.23, let's add a **node\_modules** folder so that we don't have to check in and keep track of all dependencies:



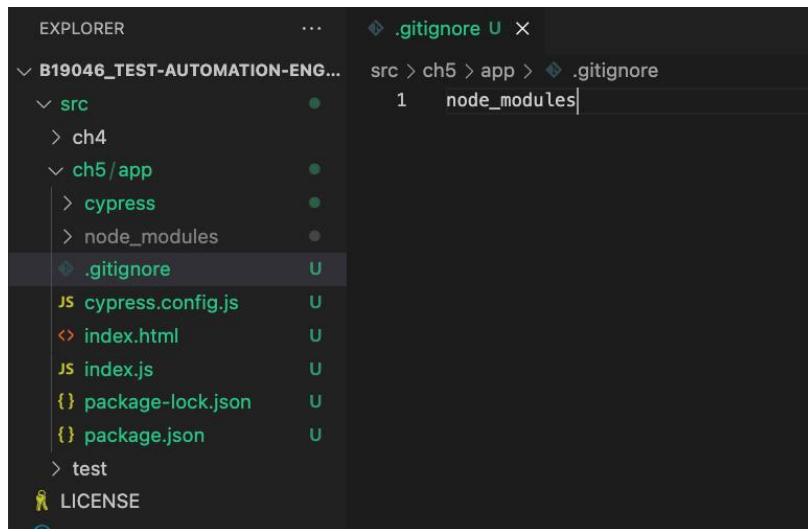


Figure 5.23 – .gitignore file

- Cypress comes with a default timeout of 4 seconds (4,000 milliseconds), and the **get** method accepts a second parameter to set a custom timeout. For example, in our spec, we can add extra wait after searching for the string and waiting for the **Reset** button to appear with **cy.get('.reset-button', {timeout:10000}).should("be.disabled")**. This line waits for 10 seconds for the **Reset** button to appear and then runs the assertion.
- Cypress provides a delay feature when performing actions on the DOM. For example, when typing an entry in the textbox, it has a default delay of 10 milliseconds for each key press. In our spec, this delay can be customized as **cy.get(".input-text").type(`\${search\_string}`, {delay:500 })** to fit the need of the application. In this case, there will be a half-second delay before typing the next character in the textbox.

We have secured a solid understanding of the major features of Cypress and are well set to explore its capabilities further. Before we close this chapter, let us review some valuable considerations for web automation.

## Links

There are also options to extend the behavior of Cypress using innumerable plugins, available at <https://docs.cypress.io/plugins/index>

# Chapter 5

## Technical requirements

We will continue using Node.js, which we installed in [Chapter 4](#). We need the **Java Development Kit (JDK)** on our machines (which can be found at this link: <https://adoptium.net/temurin/releases>) to download and install the compatible version with the operating system. Please remember to set up the **JAVA\_HOME** path in your **.zshrc** or **.bashrc** files. **JAVA\_HOME** should point to the directory where the JDK was installed.

Next, we will be needing Android Studio, which can be installed using the following link: <https://developer.android.com/studio>. Similar to the **JAVA\_HOME** path, the **ANDROID\_HOME** variable should be set up in your **.zshrc** or **.bashrc** files and point to the directory where the Android SDK is installed. Also, make sure to append the **PATH** variable to include the **platform-tools** and **tools** folders within the SDK. We will be using a demo Android application from [github.com/appium/android-apidemos](https://github.com/appium/android-apidemos) for our automated testing. This is Google's demo application used for testing Appium. All the code and setup used in this chapter can be found at <https://github.com/PacktPublishing/B19046-Test-Automation-Engineering-Handbook/tree/main/src/ch6>.

## Hands-On Section

### Setting up Appium and WebdriverIO

Combining Appium with WebdriverIO helps build an extremely scalable and customizable mobile automation framework. Let us start with Appium installation on macOS.

#### Appium installation

Next, let us go ahead and install Appium. This can be done using the **npm install -g appium@next** command to install a version above 2.0 globally. The use of **@next** here is to force an installation of a beta version (2.0.0). If by the time you install this, 2.0 is available as the latest stable version, you do not have to use **@next**. With the **appium -v** command, double-check the version after installation.

Appium involves multiple installations working in tandem, and it would be convenient to have a tool that could provide us with constant feedback on the health of our setup. **appium-doctor** does just that. Let us install it using the **npm install -g appium-doctor** command, and the installation can be checked using **appium -version**.

**appium-doctor** can be executed for the Android platform through the **appium-doctor --android** command. It primarily checks for the presence of all dependencies and their path. Users should be on the lookout for any errors in the output from executing **appium-doctor** and fix them before proceeding further.

There is one more installation before we can wrap up this section. Let us now install the necessary Appium drivers for iOS and Android using the following commands:

```
appium driver install xcuitest
```

```
appium driver install uiautomator2
```

The installations can be verified using **appium driver list**, as shown in [Figure 6.1](#):

```
→ ch6 git:(main) ✘ appium driver install xcuitest
info Appium Setting NODE_PATH to '/usr/local/lib/node_modules'
✓ Installing 'xcuitest' using NPM install spec 'appium-xcuitest-driver'
i Driver xcuitest@4.11.1 successfully installed
- automationName: XCUITest
- platformNames: ["iOS","tvOS"]
→ ch6 git:(main) ✘ appium driver install uiautomator2
info Appium Setting NODE_PATH to '/usr/local/lib/node_modules'
✓ Installing 'uiautomator2' using NPM install spec 'appium-uiautomator2-driver'
i Driver uiautomator2@2.6.0 successfully installed
- automationName: UiAutomator2
- platformNames: ["Android"]
→ ch6 git:(main) ✘ appium driver list
info Appium Setting NODE_PATH to '/usr/local/lib/node_modules'
✓ Listing available drivers
- xcuitest@4.11.1 [installed (NPM)]
- uiautomator2@2.6.0 [installed (NPM)]
- youiengine [not installed]
- windows [not installed]
- mac [not installed]
- mac2 [not installed]
- espresso [not installed]
- tizen [not installed]
- flutter [not installed]
- safari [not installed]
- gecko [not installed]
→ ch6 git:(main) ✘
```

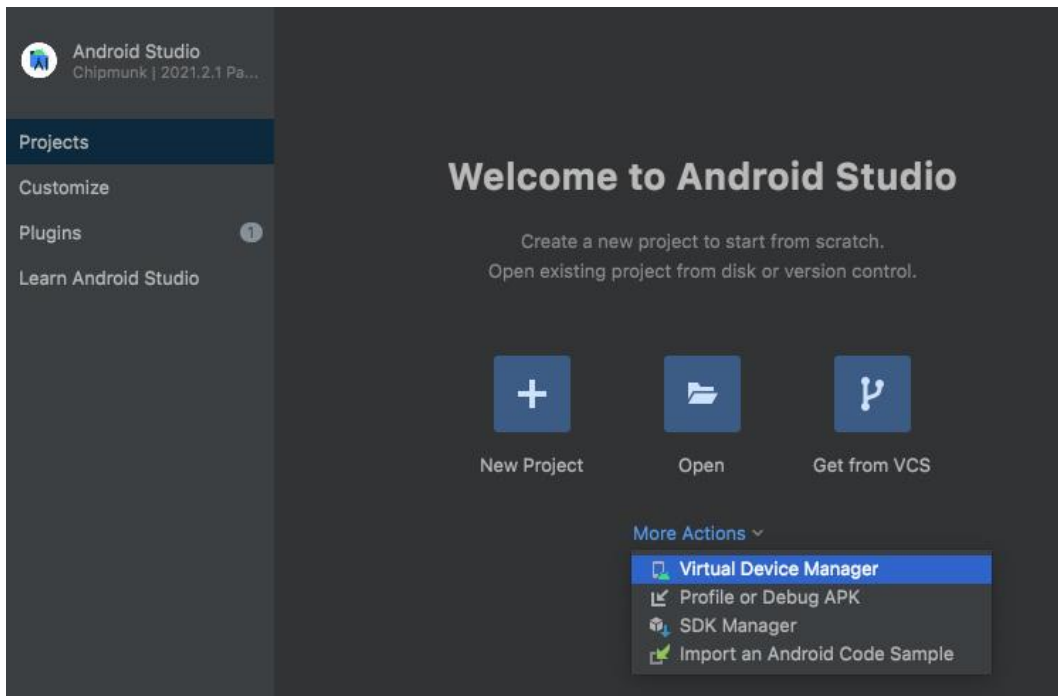
**Figure 6.1 – Appium driver installation**

This completes the Appium setup for macOS, and we have a couple more steps before we can start writing our first tests. Before diving into the WebDriverIO setup, let us look at how to configure an Android emulator.

### Configuring an Android emulator

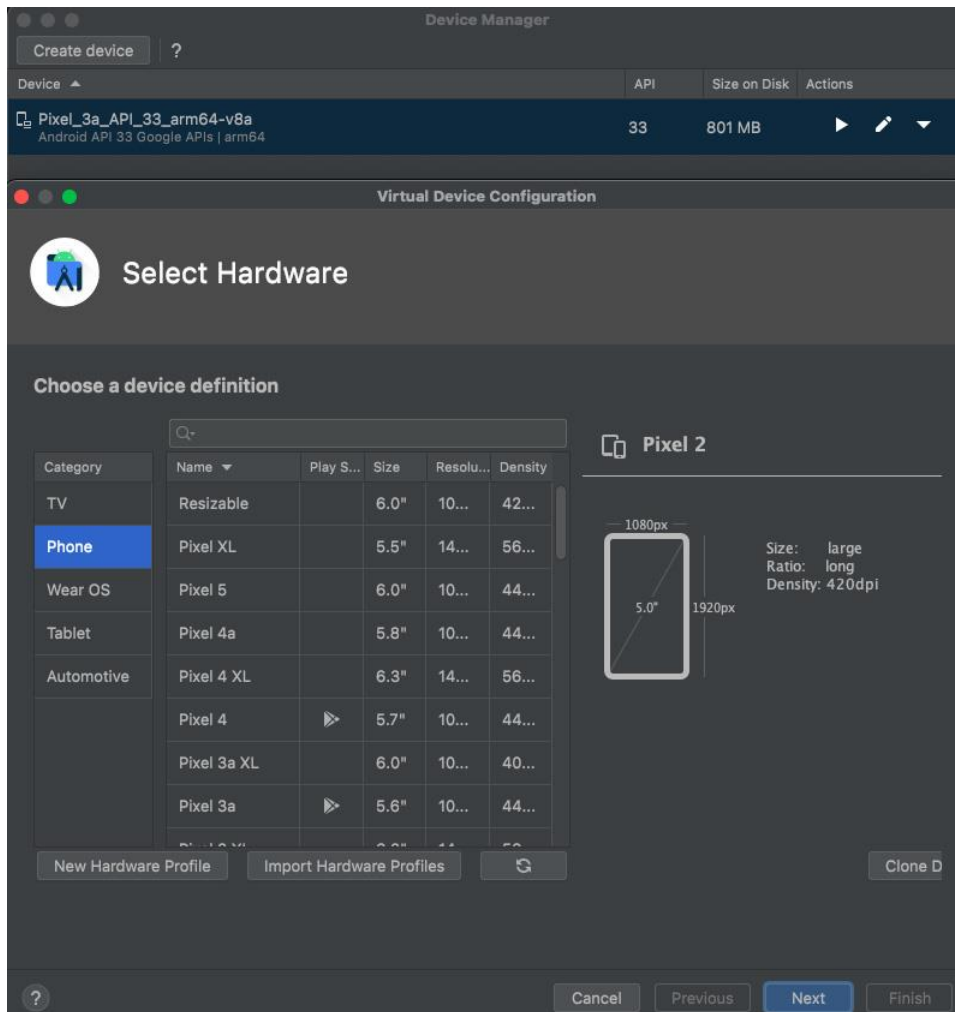
Emulators are virtual devices that let engineers set up and test against a variety of devices on the computer. Let us go over how to set up an Android emulator in this section:

1. In the home dialog of Android Studio, click on the **Virtual Device Manager** option, as shown in [Figure 6.2](#):



**Figure 6.2 – Android Studio Virtual Device Manager**

2. The Android Studio installation usually comes with an emulator, as shown in **Figure 6.3**. You can add another device by clicking on the **Create device** button at the top left of the dialog box. This opens an additional dialog to set up the new hardware:



**Figure 6.3 – Android Studio device selection**

3. For our example, let us select **Pixel 4a** and hit **Next**, which will take us to the **System Image** selection dialog.
4. Please make sure to download two different versions here. We will be running the Appium tests on one version and connecting the Appium Inspector on the other. In our case, we will be downloading the Android API versions 32 and 33, as shown in **Figure 6.4**:

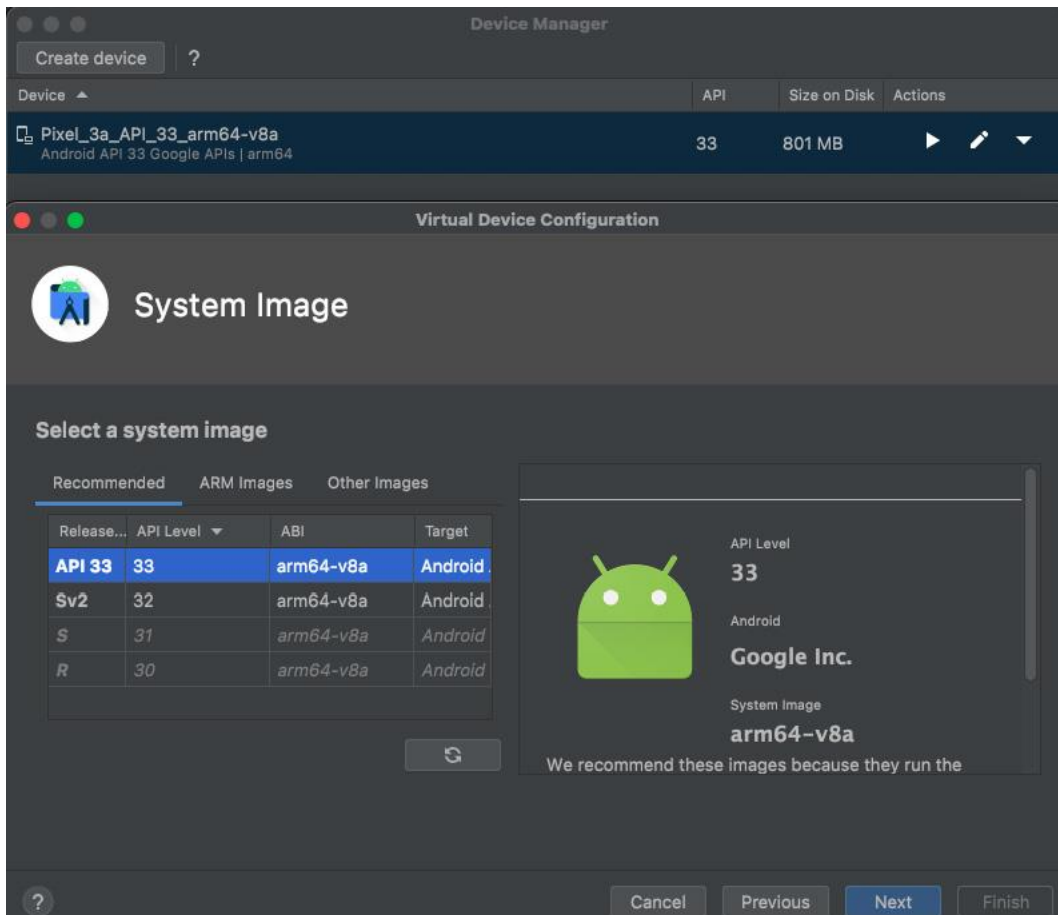


Figure 6.4 – Android Studio system images

- Hit the **Next** button and complete the emulator setup with the **Finish** button.
- On the **Device Manager** dialog, hit the play button against one of the devices, as highlighted in green in **Figure 6.5**. This will open the device emulator:

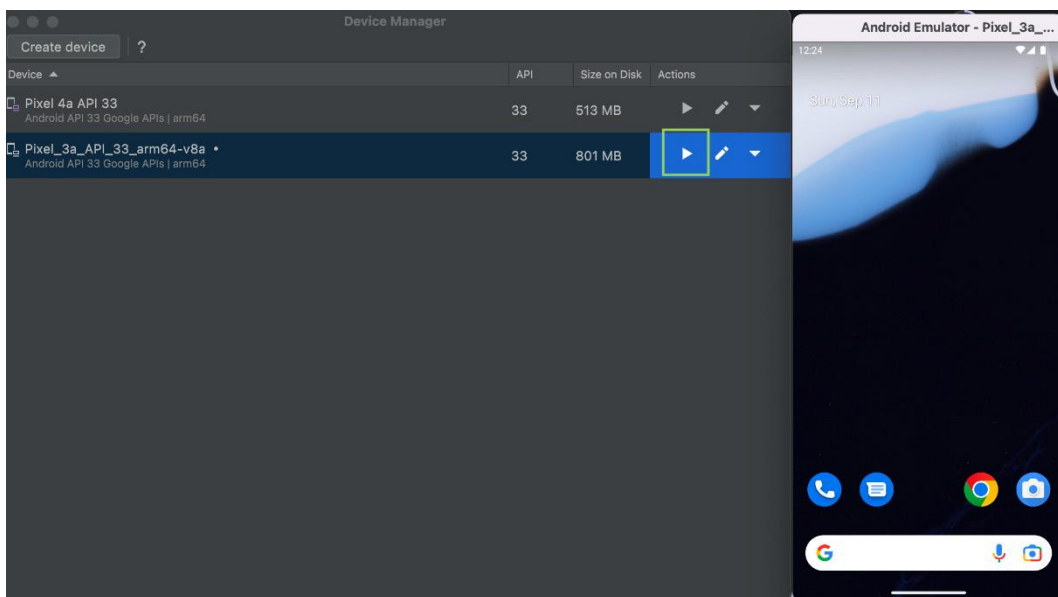


Figure 6.5 – Opening the Android device emulator

We are now set up with Appium and an emulated device to test on. Let us now dive into WebdriverIO installation and setup.

## Configuring WebdriverIO with Appium

Let us begin with the WebdriverIO setup by creating and initializing a new directory named **webdriverio-appium**. You could also name it based on whichever project name suits your needs. Once we are inside the directory, let us run the **npm init -y** command to set up our project. This should create a **package.json** file within the directory.

Next, let us install the CLI for WebdriverIO using the **npm install @wdio/cli** command. This is used for setting up the configuration for WebdriverIO using the CLI. Let us now run **npx wdio config**. This **npx** command goes inside the **node\_modules** folder to find WebdriverIO, which we just installed, and runs the **config** command using the CLI.

On running this command, we go through a series of steps, with each of them requiring a selection. Let us look at them one by one. Refer to **Figure 6.6** for a summary of the options selected for configuring WebdriverIO through the CLI:

```
→ webdriverio-appium git:(main) ✕ npx wdio config

=====
WDIO Configuration Helper
=====

? Where is your automation backend located? On my local machine
? Which framework do you want to use? mocha
? Do you want to use a compiler? No!
? Where are your test specs located? ./test/specs/**/*.js
? Do you want WebdriverIO to autogenerate some test files? No
? Which reporter do you want to use? spec
? Do you want to add a plugin to your test setup?
? Do you want to add a service to your test setup? appium
? What is the base url? http://localhost
? Do you want me to run `npm install`? Yes
```

**Figure 6.6 – WebdriverIO configuration**

Here are the steps

1. In the first step, we will be selecting the location of the automation backend. Since we are not running the tests externally yet, we will be selecting the first option, which is the local machine.

We will be using the **Mocha** framework for our tests, and that option should be selected in the next step.

2. In the next step, we will be selecting the **No** option as we will not be using any compiler in our tests.
3. We will go with the default location for our test specs in the next step.
4. We will type **No** in the next step as we will be starting all our tests from scratch and do not need the autogenerate option.
5. We will stick with the default **spec** reporter in the next step.

6. Let us skip adding any plugins at this step.
7. In the next step, since we will be running the tests from Appium, we will not be needing any additional drivers. So, let us select the **appium** option.
8. We can ignore the base URL step as we will not be running a test on the web, and move on to the next step.
9. In the last step, we can select **Yes** for running the **npm install**.

The use of the CLI to configure WebdriverIO is productive as we circumvent the need for installing these packages manually. WebdriverIO does a lot of heavy lifting for us to configure the fundamental requisites for our project.

Before proceeding, please make sure to install Appium and its drivers again within the **webdriverio-appium** folder as this is where we will be storing and executing our tests.

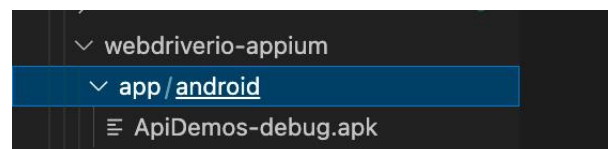
In the next section, let us look at additional WebdriverIO configuration for Android.

### WebdriverIO Android configuration

As part of the WebdriverIO setup through the CLI, you will notice that a new **wdio.conf.js** file has been created. This is the primary file where we will be making changes to get WebdriverIO working with Appium. Let us now go ahead and look at how it's set up to start making changes.

Quickly browsing through this file shows the customizable port number (**4723**) where the Appium server will spin up. All configurations that we did through the CLI should also be reflected in this file. The important change to be done in this file is in the **capabilities** section. It shows the browser as Chrome by default. Here, we will be specifying the Android settings to connect to the Appium server and run it via WebdriverIO.

Let us now copy our test application within our project by creating a new **app/android** folder structure, as shown in **Figure 6.7**:



**Figure 6.7 – Copying test Android app**

We are now ready to update the capabilities section of the config file. Refer to **Figure 6.8** for the values to be used here. We have added the platform name and platform version, which are Android and 13.0, respectively, in this case. Then, we added the device name, which should be the same as the one set up for the emulator in Android Studio. The automation name is the name of the driver used for Android automation.

For the app path, we use the path library to dynamically create a complete path to the test app within our project, as shown in the following screenshot. This library is built-in in Node.js and doesn't need to be installed separately. The path library must be initialized at the beginning of this config file using **const path = require('path')**. This completes the preliminary customization of the config file for the Android application:



```
JS wdio.conf.js X
src > ch6 > webdriverio-appium > JS wdio.conf.js > [E] config > capabilities
50 //
51 // If you have trouble getting all important capabilities together, check out the
52 // Sauce Labs platform configurator - a great tool to configure your capabilities:
53 // https://saucelabs.com/platform/platform-configurator
54 //
55 capabilities: [{
56   platformName: "Android",
57   "appium:platformVersion": "13.0",
58   "appium:deviceName": "Pixel 4a",
59   "appium:automationName": "UIAutomator2",
60   "appium:app": path.join(process.cwd(), "app/android/ApiDemos-debug.apk")
61 }
```

Figure 6.8 – wdio config: capabilities section

Before we try to run the app with WebdriverIO, let us create a test folder and an empty test file, as shown in [Figure 6.9](#). Also, launch the specified emulator from within Android Studio:

```
EXPLORER JS first_spec.js X
src > ch6 > tests > specs > JS first_spec.js > ...
1 describe("First Android Spec", () => {
2   it("using Appium and WebdriverIO", () => {
3
4   });
5 });
```

Figure 6.9 – Test folder and file creation

We are now ready to run this spec. For running this spec, use the **npm wdio** command. This command by default uses the **wdio.config.js** file to spin up the Appium server, load the test app, and execute the test. Results from the empty test can be seen in [Figure 6.10](#):





In the next section, let's understand how an **async** function works in JavaScript and then write a test to validate mobile elements.

## Figures

```
JS async_await.js X
src > ch6 > functions > JS async_await.js > ...
1  async function demo_async_await() {
2
3      let simple_promise = new Promise((resolve, reject) => {
4          setTimeout(() => resolve("Promise Fulfilled!"), 5000)
5      });
6      // wait for the promise to be resolved
7      let result = await simple_promise;
8      console.log(result);
9  }
10
11  demo_async_await();

TERMINAL
→ functions git:(main) X node async_await.js
Promise Fulfilled!
→ functions git:(main) X
```

Figure 6.14 – JavaScript function with async/await

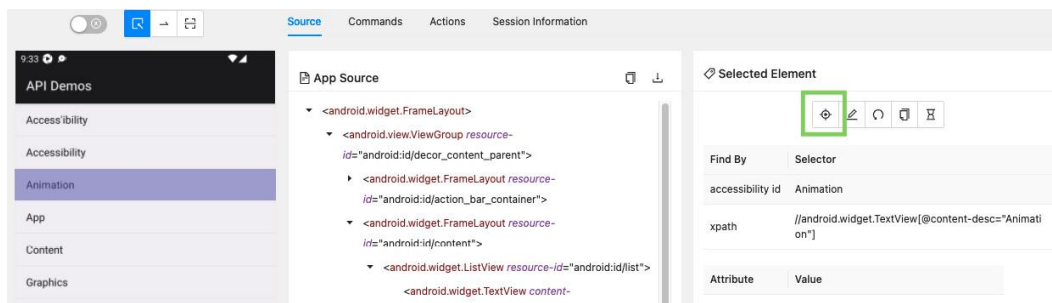


Figure 6.15 – Inspecting an element via Appium Inspector

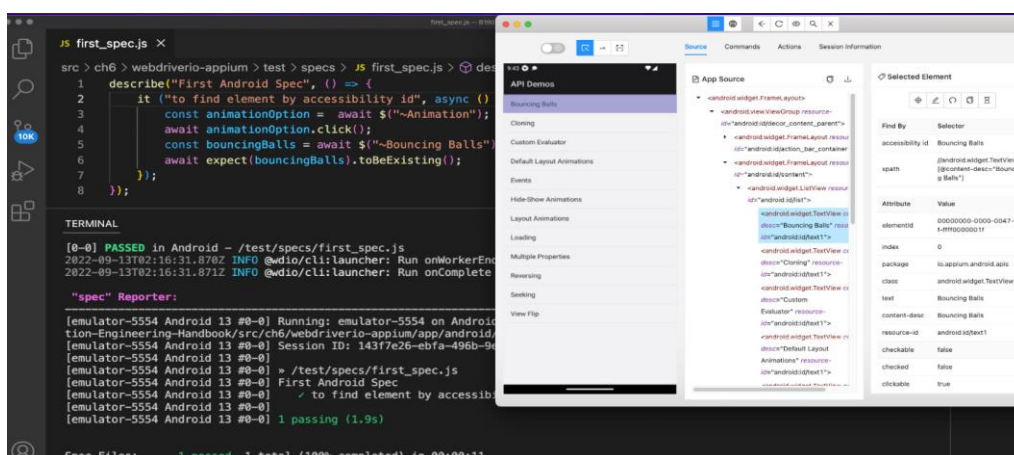


Figure 6.16 – Appium test execution

```
JS wdio.shared.conf.js JS wdio.android.conf.js X JS wdio.ios.conf.js
src > ch6 > webdriverio-appium > config > JS wdio.android.conf.js > ...
1  const { config } = require('./wdio.shared.conf')
2  const path = require('path');
3  config.port = 4723
4  config.specs = [
5    '<<folder where the Android tests live>>'
6  ]
7  config.capabilities = [
8    {
9      platformName: "Android",
10     "appium:platformVersion": "13.0",
11     "appium:deviceName": "Pixel 4a",
12     "appium:automationName": "UIAutomator2",
13     "appium:app": path.join(process.cwd(), "/app/android/ApiDemos-debug.apk")
14   }
15 ]
16 exports.config = config
```

Figure 6.18 – WebdriverIO Android config file

```
JS wdio.shared.conf.js JS wdio.android.conf.js JS wdio.ios.conf.js X
src > ch6 > webdriverio-appium > config > JS wdio.ios.conf.js > ...
1  const { config } = require('./wdio.shared.conf')
2  const path = require('path');
3  config.port = 4723
4  config.specs = [
5    '<<folder where the iOS tests live>>'
6  ]
7  config.capabilities = [
8    {
9      platformName: "iOS", |
10     "appium:platformVersion": "15.0",
11     "appium:deviceName": "iPhone 13",
12     "appium:automationName": "XCUITest",
13     "appium:app": path.join(process.cwd(), "/app/iOS/Test-iOS.app")
14   }
15 ]
16 exports.config = config
```

Figure 6.19 – WebdriverIO iOS config file

## Code

### Code 6.1

```
describe ("First Android Spec", () => {
  it ("to find element by accessibility id", async () => {
    const animationOption = await $("~Animation");
```

```
await animationOption.click();  
const bouncingBalls = await $("~Bouncing Balls");  
await expect(bouncingBalls).toBeExisting();  
});  
});
```

### **Code 6.2: An example of the capabilities section of the wdio.config.js file for iOS:**

```
platformName: "iOS",  
"appium:platformVersion": "15.0",  
"appium:deviceName": "iPhone 13",  
"appium:automationName": "XCUITest",  
"appium:app": path.join(process.cwd(), "/app/iOS/Test-iOS.app")
```

## **Links**

For further exploration of the various features that Appium offers, refer to the documentation at <https://appium.io/docs/en/about-appium/api>.



# Chapter 7

## Technical requirements

We will continue using Node.js along with JavaScript in this chapter. We will also download the Postman application (version **9.3.15**) and the Newman command-line tool. We will need Docker installed locally to run our Postman collections on a Docker container. Docker installation instructions can be found at <https://www.docker.com>.

## Figures

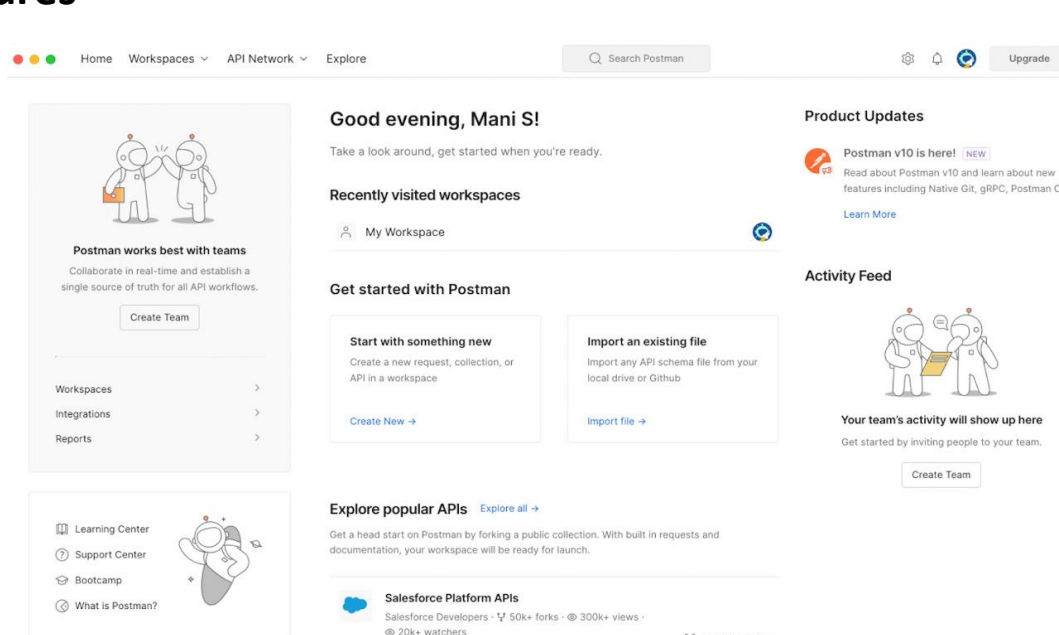


Figure 7.1 – Postman application

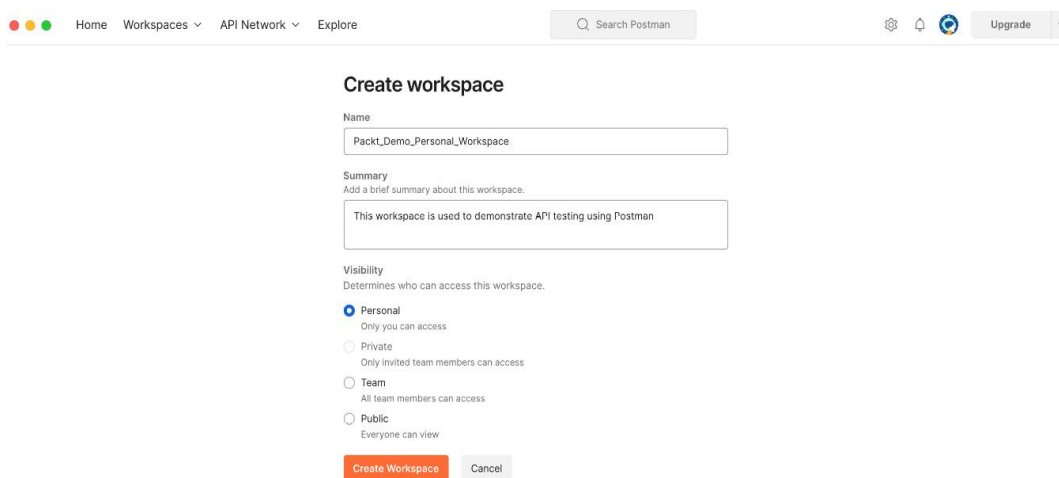




Figure 7.2 – Creating a workspace

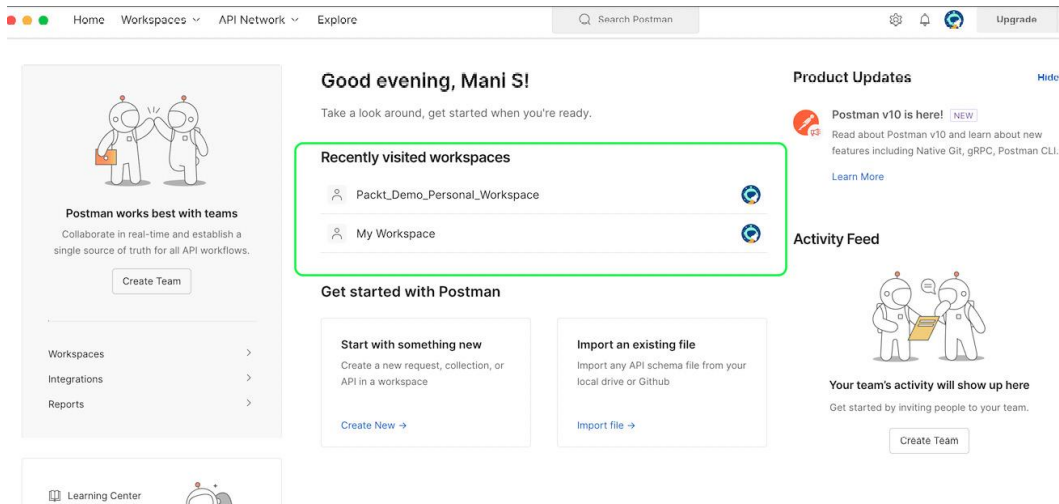


Figure 7.3 – Home page with multiple workspaces

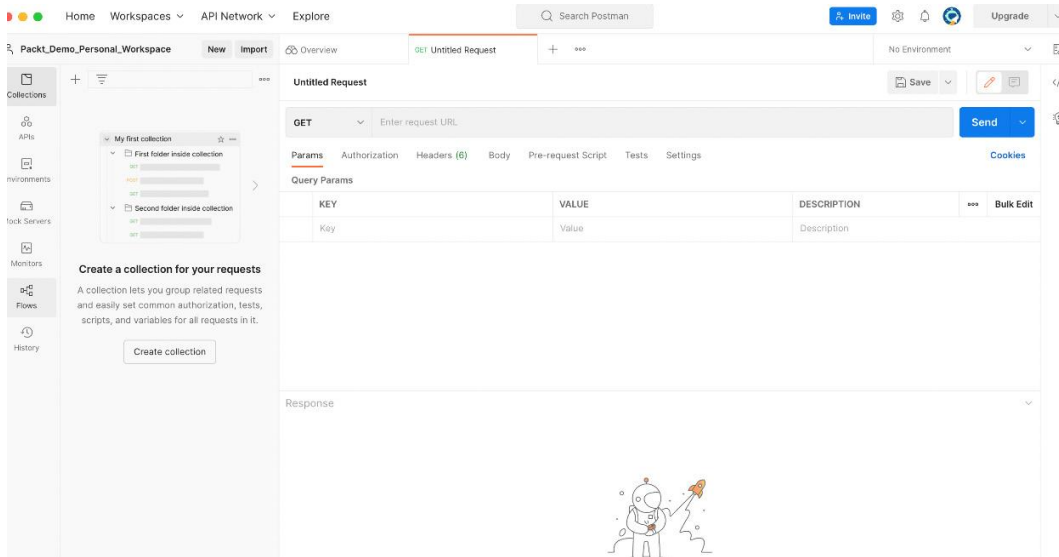
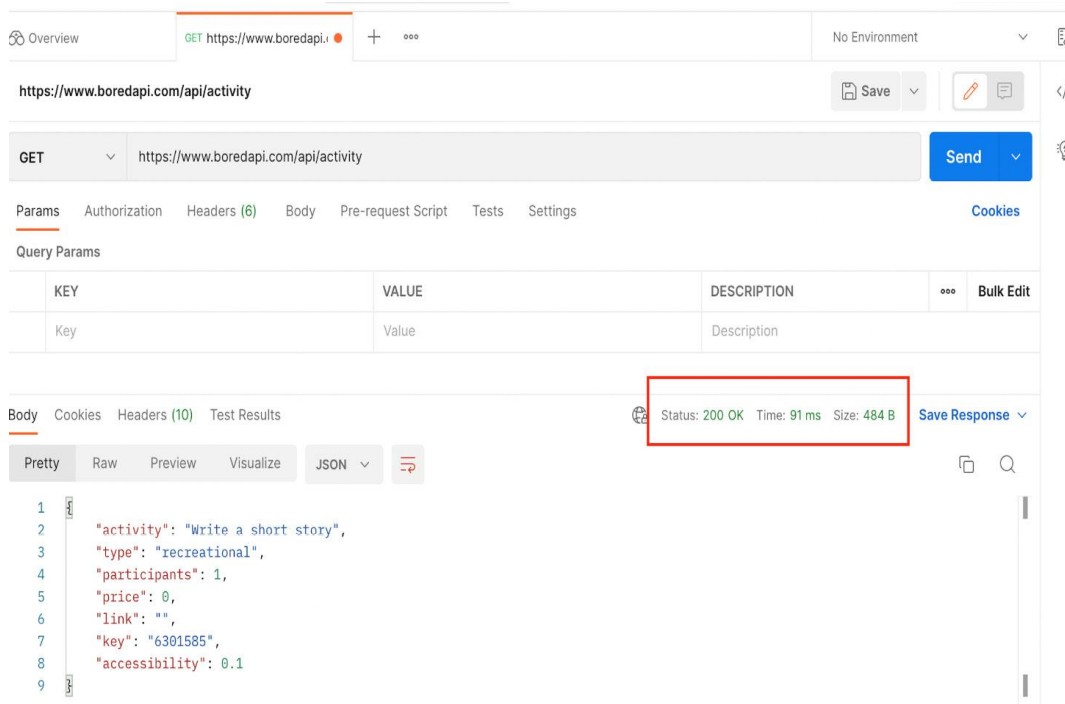


Figure 7.4 – New API request

## Hands-On Section

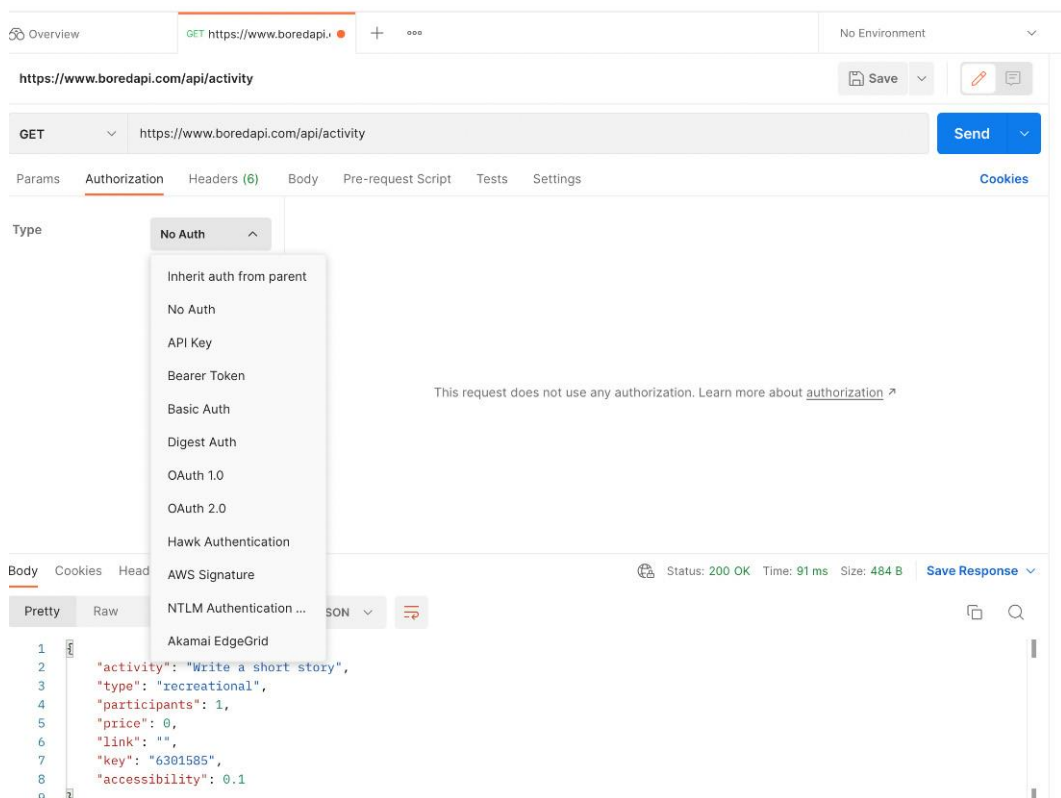
### Making a GET API request

We will be using the <https://www.boredapi.com/api/activity> API to get an understanding of **GET** requests. Bored is a free API that returns some random activities to do when bored. Postman makes it easy to get a simple **GET** API request without any authorization working. Just paste the API URL in the URL section of the request window and hit the **Send** button next to it. Every request to the server must be made with a URL to fetch the required response. **Figure 7.5** shows the **GET** request with the response. Here, the **Status** section of the response says **200 OK**, which means that the server responded to the request without any errors. The server returns the response in a JSON format, which can be validated for accuracy based on the business logic. In our case, we see an activity being returned with a bunch of other information:



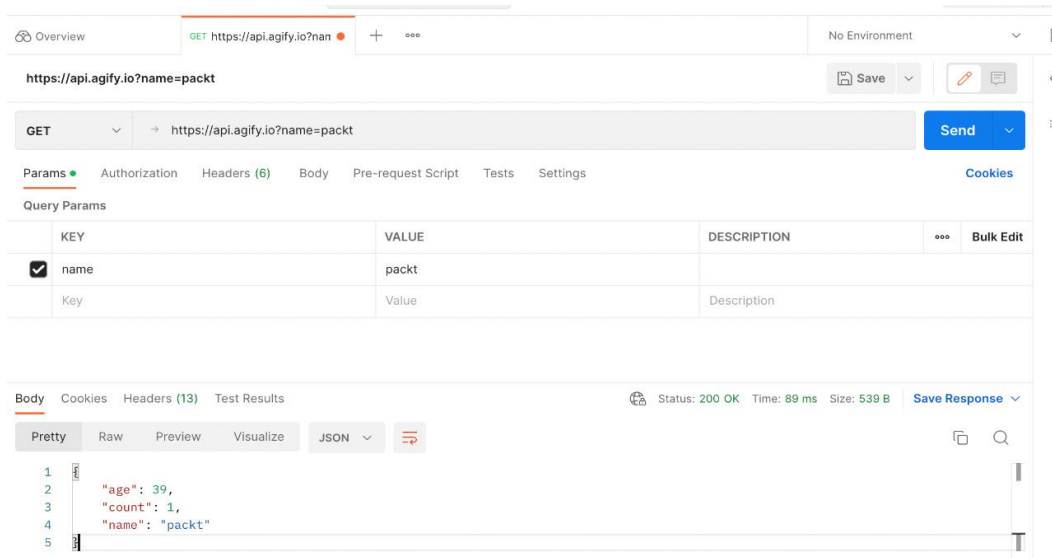
**Figure 7.5 – GET API request**

In most cases, the API will have certain authorization to be added for the request to work. Postman supports a wide variety of authorization mechanisms that can be accessed via the **Authorization** tab, as shown in **Figure 7.6**:



**Figure 7.6 – Request authorization support in Postman**

Postman identifies the applicable headers for a given API call, but in cases where there is specific metadata that must be sent as part of the header, this can be done using the **Headers** tab. Postman automatically identifies any query parameters that are added as part of the URL. For the case in **Figure 7.7**, <https://api.agify.io?name=packt>, a name is sent as a query parameter as part of the URL. Postman creates a parameter in the **Params** tab, and this can be modified to feed the request with different test data. New parameters can be manually added here as well:

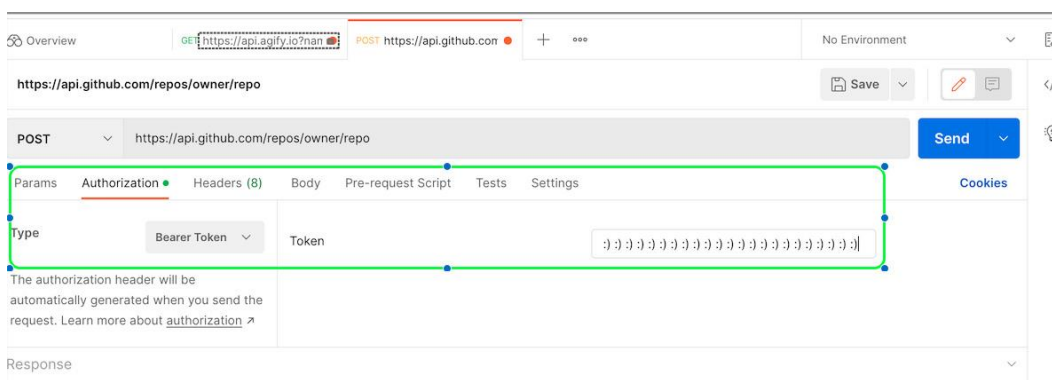


**Figure 7.7 – API request with a query parameter**

Users are encouraged to check out the various other features available within the request and response windows. Let us now learn to make a **POST** API request.

## Making a POST API request

A **POST** request creates a new resource on the server and requires content to be sent in the body of the request. Postman supports different body types for a **POST** call, and in this section, we will review how to make a **POST** call. Create a new API request and click on the dropdown to the left of the URL section to select a **POST** request type. We will be using GitHub’s **create a new repository** API call (<https://api.github.com/user/repos>) for our example here. GitHub provides a lot of public APIs, but it requires the generation of an access token. This token can be generated in the personal access tokens section of your GitHub profile. Please remember to copy and save this secure token for future use in Postman. As shown in **Figure 7.8**, use this as a bearer token in the **Authorization** tab of the new **POST** request:

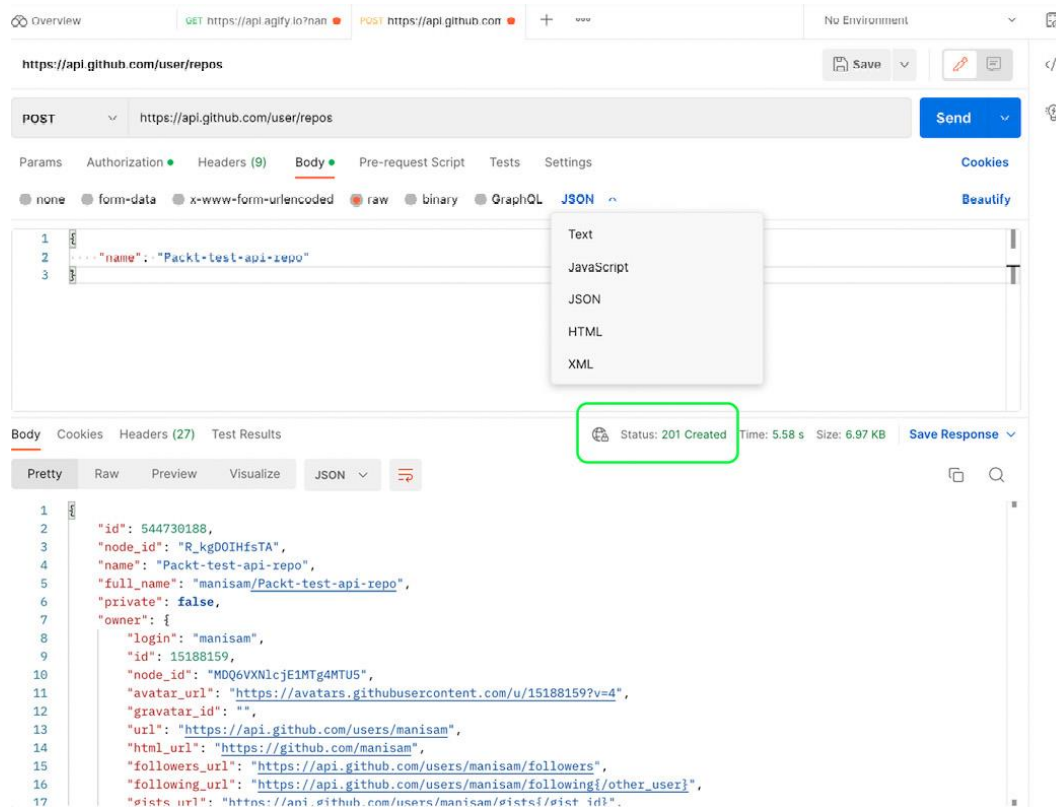


**Figure 7.8 – POST API request authorization**

Now, moving on to the **Body** section of the request, this API requires a name as a mandatory key for the new repository being created. An example of this request can be found at this link:

<https://docs.github.com/en/rest/repos/repos#create-a-repository-for-the-authenticated-user>.

We will be using the raw body type with JSON from the dropdown, as shown in **Figure 7.9**. Postman supports a wide variety of request body formats, and the one supported by the API being tested should be used. On hitting the **Send** button, we complete the **POST** call to the GitHub server to create a new repository with the name **Packt-test-api-repo**. **Figure 7.9** shows the response status of **201 Created** with all the metadata in the response body section. Users may notice that the status code is different from the **GET** call as **201** indicates that in addition to the call being successful, a new resource was created by the server:



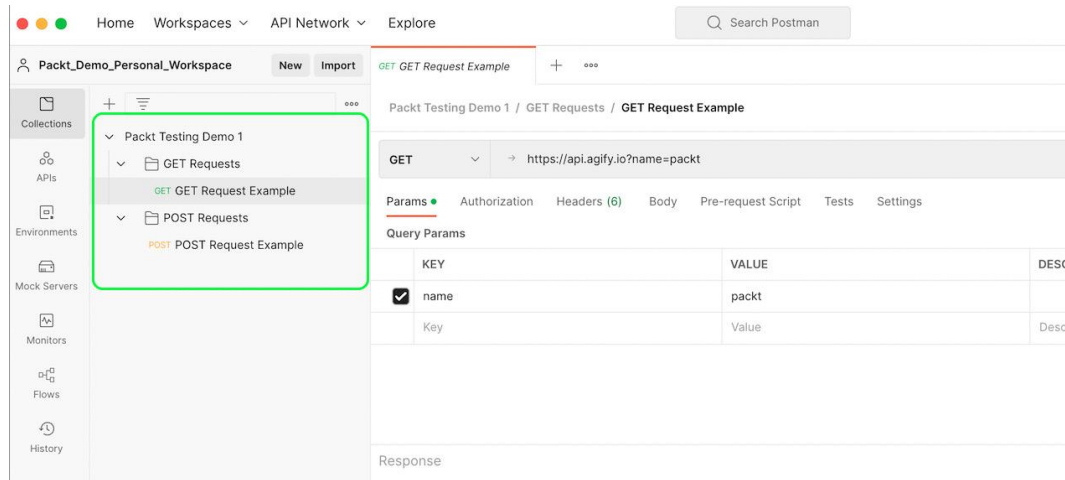
**Figure 7.9 – POST API request body type**

This completes our section on creating a new resource using a **POST** call. In the next section, let's learn about collections and how they help structure API requests in Postman.

## Organizing API requests using collections

Postman provides a way to group the API requests using collections. It helps organize a workspace by breaking it down, and a workspace can also be sorted into multiple collections. Apart from this, collections can also be published as documentation as well as run together in an automated fashion. In this section, let us review how to create a collection and add requests to it. Collections can be created in multiple ways within a workspace, and a simple way is to use the **+** button next to the **Collections** icon in the left pane of the **Workspaces** window. A name must be provided for the

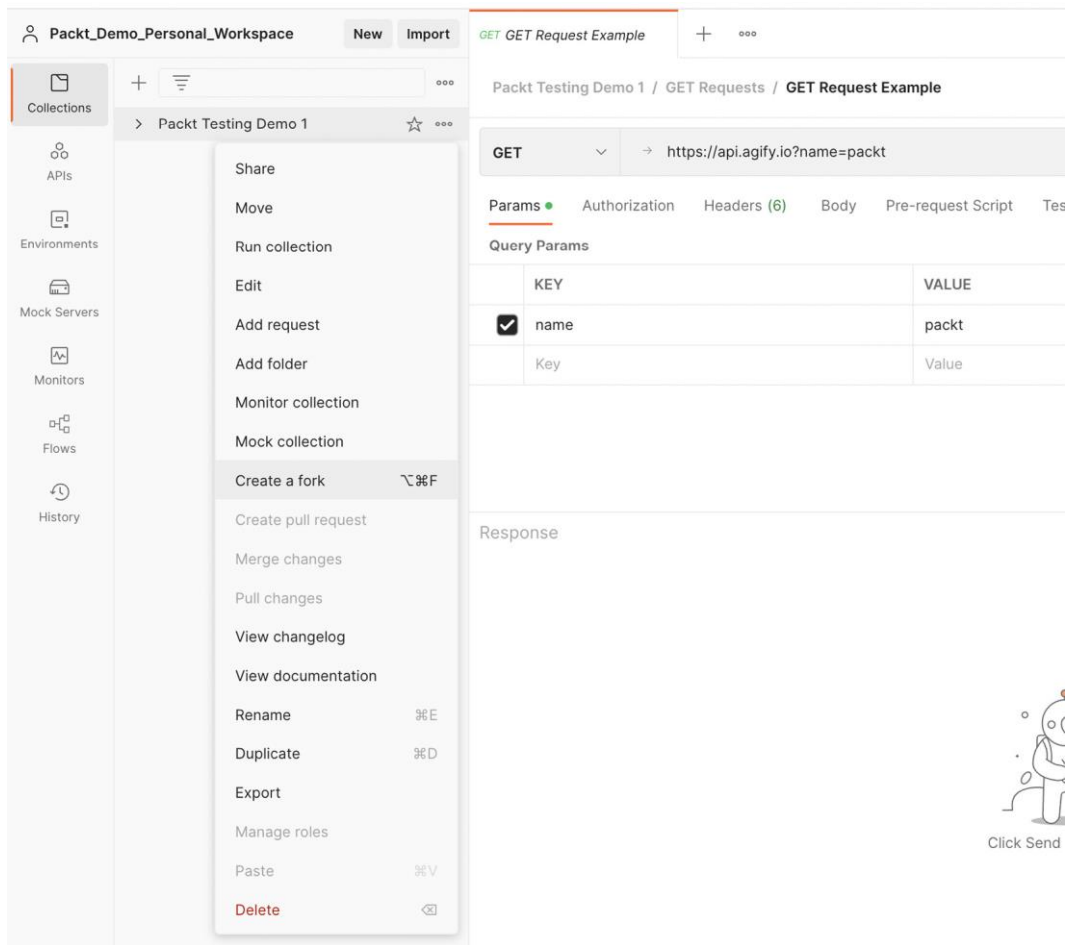
collection, and both existing and new requests can be saved to this collection. **Figure 7.10** shows a new collection holding the two API requests we have created so far in the previous sections:



**Figure 7.10 – Collections**

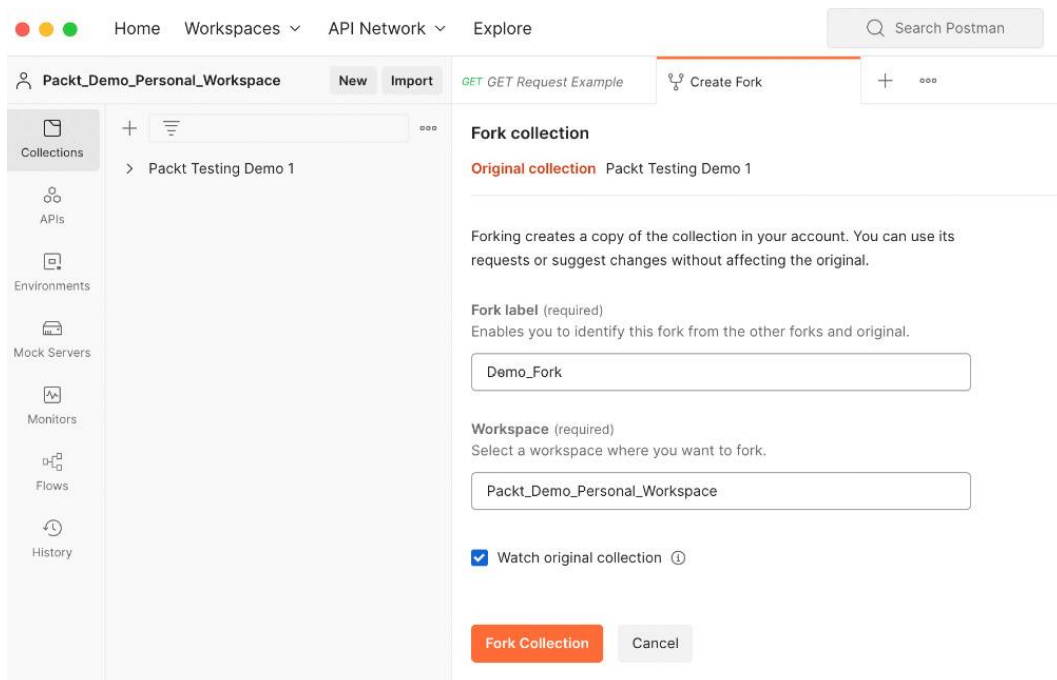
Collections can thus help organize API requests in a meaningful way under a given workspace. This helps immensely when there are a high number of requests, which is usually the case when testing enterprise applications. Various other actions can be performed on collections, such as **Export**, **Monitor**, **Mock**, and so on. Let us now look at one more feature of a collection that promotes collaboration within the team.

Postman allows users to create a fork of a collection and merge it after making some modifications to the forked collection. These changes can be shared with other members, just like how Git pull requests work. This can be done via the **Collections** drop-down menu, as shown in **Figure 7.11**:



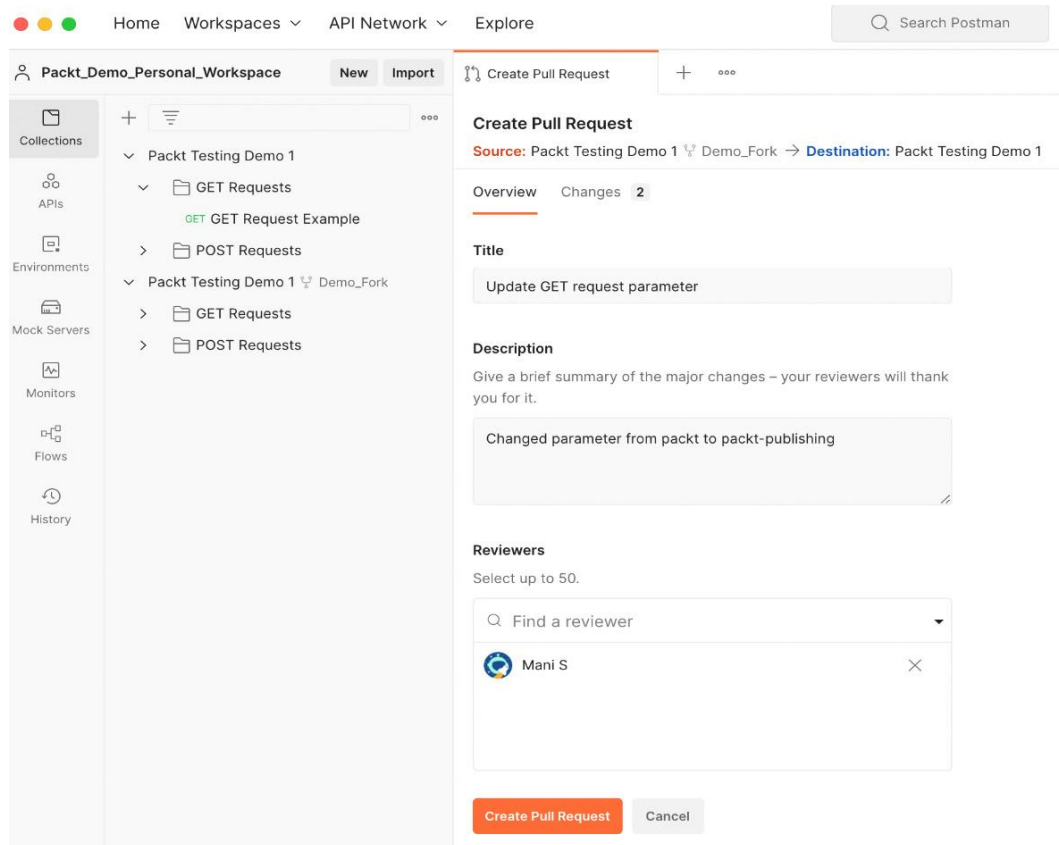
**Figure 7.11 – Collections drop-down menu**

On clicking the **Create a fork** option, the user will be required to enter a label for the fork and to which workspace this collection is to be forked, as shown in **Figure 7.12**:



**Figure 7.12 – Forking a collection**

Once the required changes have been made to the forked collection, a pull request can be created using the **Collections** drop-down menu. **Figure 7.13** shows a snippet of a pull request where the user can provide a title, description, and select reviewers:



**Figure 7.13 – Creating a pull request**

The changes can be reviewed and merged using the merge option within the pull request or through the **Collections** drop-down menu. This is a neat way to keep track of changes to your API requests and nurture collaboration within the team while making these changes.

So far, we have looked at the basics of the Postman tool and how to make requests manually. This works well for testing new API features but falls short when it comes to regression testing. In the next section, let's learn to write automated tests to validate API responses.

## Writing automated API tests

Postman allows us to add tests that run automatically after an API response is returned from the server. This can be done through the **Tests** tab in the Postman request dashboard. We will be using the following three GitHub API requests in this section to help us set up and understand automated API response validation:

- Create a new repository
- Get a repository by name
- Delete a repository by name

In the next section, let us review how to use snippets to speed up our test automation process.



## Using snippets for asserting an API response

Postman comes with pre-defined JavaScript test scripts in the form of code snippets that can directly be used in our tests. Let us start by adding snippets for some of the basic checks performed on an API response. Snippets are shown on the right pane next to the various tabs on the request dashboard.

Every API test requires the validation of the status of an API response, and Postman provides a readymade code snippet for this. On selecting the **Status code: Code is 200** snippet, the following code is populated onto the **Tests** tab, as shown in **Figure 7.14**:

```
pm.test("Status code is 200", function () {  
  pm.response.to.have.status(200);  
});
```

**pm** represents a Postman object, and it contains all information pertaining to the request and response body. The **pm** object comes with a lot of properties and methods built in. Here, we use the **test** method, which accepts a description and a function within which an assertion can be defined. These assertions are defined using the **chai** library, and readers can refer to their documentation at <https://www.chaijs.com/api/bdd/> to get familiarized with more assertions.

Let us now add the **Status code: Code name has string** snippet, which adds the following code snippet. In the case of a **GET** request, this string is **OK**, and for the **POST** request, it is **Created**, as we have seen before:

```
pm.test("Status code name has string", function () {  
  pm.response.to.have.status("Created");  
});
```

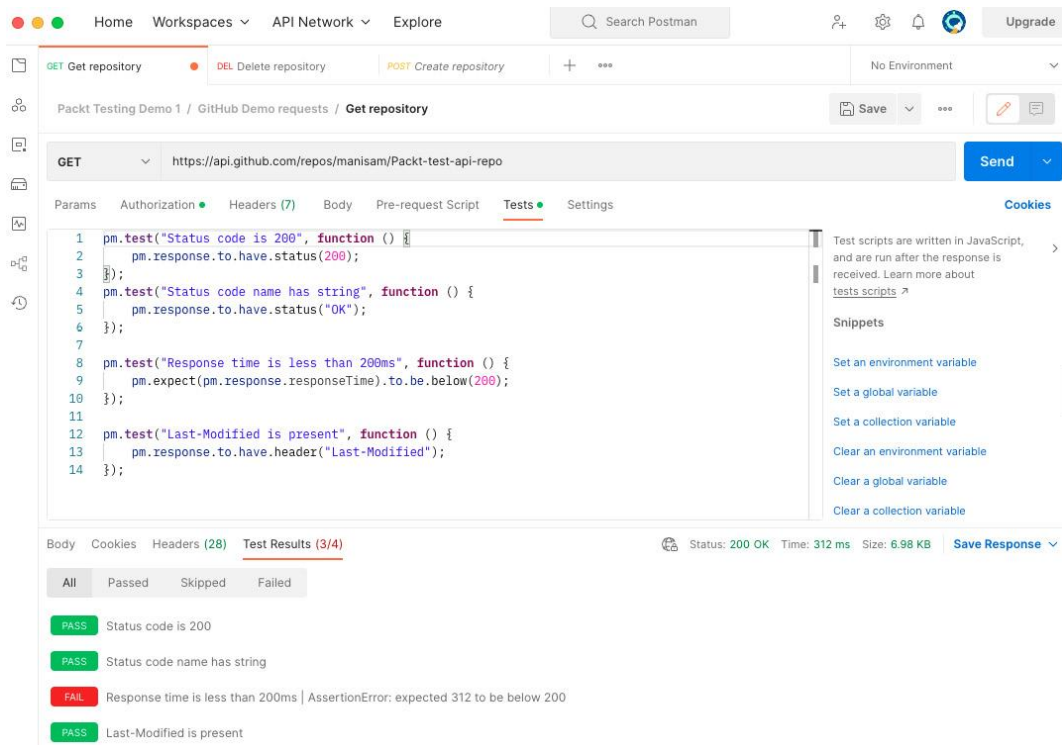
Postman provides a snippet to validate the response time of the API. On adding the **Response time is less than 200ms** snippet, we see the following code added. Note that we use the **expect** assertion here, which operates on the **responseTime** property and checks its value range:

```
pm.test("Response time is less than 200ms", function () {  
  pm.expect(pm.response.responseTime).to.be.below(200);  
});
```

Let us now add an assertion on the response header using the **Response headers: Content-Type header check** snippet. This can be further modified to check the presence of any header in the response, as shown in the following code snippet:

```
pm.test("Last-Modified is present", function () {  
  pm.response.to.have.header("Last-Modified");  
});
```

**Figure 7.14** shows a summary of the test results for the **GET** repository API call with a test status for each of the snippets we have added:

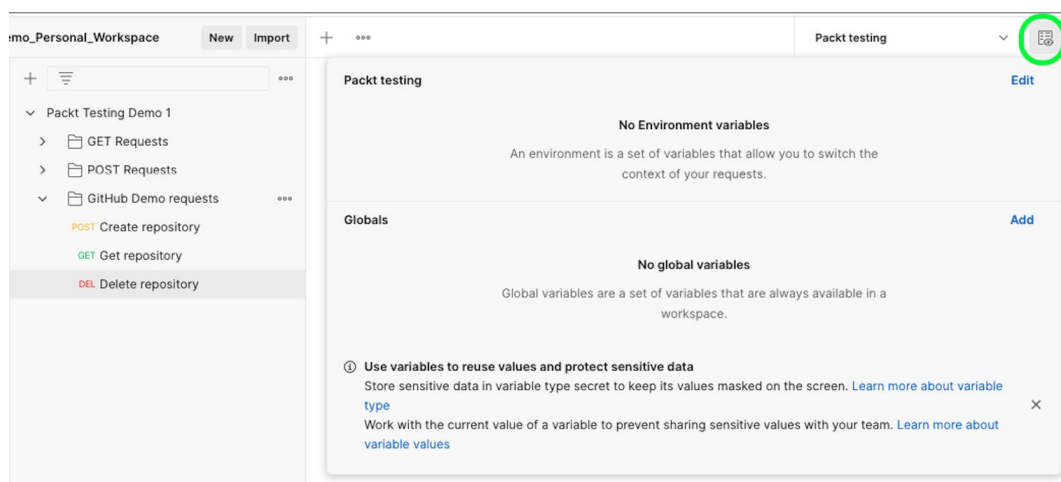


**Figure 7.14 – Automated API response validation**

These are some of the generic tests we can add for API responses. In the next section, let us learn how to add assertions on specific values such as the repository name on the response.

## Understanding Postman variables

Let us understand what an environment means in Postman before jumping into looking at variables. An environment is an assembly of variables that can be used in API requests. For example, multiple environments can be added in Postman, each with its own collection of variables. A new environment can be created and added using the **Environments** drop-down menu and the icon from the top-right section of the workspace window, as shown in **Figure 7.15**. Here, we have created a new **Packt testing** environment:



**Figure 7.15 – Environments drop-down menu and icon**

Postman has five types of built-in variables, which are the following:

- **Global variables:** Variables with the broadest scope and can be accessed anywhere in a workspace
- **Collection variables:** Variables scoped to be available for all the requests within a collection
- **Environment variables:** These variables are accessible only within an environment and primarily used to manipulate access—for example, in staging versus production
- **Data variables:** These are variables created when importing external data and are scoped for a single execution
- **Local variables:** Used on tests in a single request and lose scope as soon as the execution ends

Let us consider the API for creating a new GitHub repository to understand how variables can be used in Postman to remove static context from API requests. For example, we must validate that the repository name matches between request and response. It is also important to remember that GitHub repositories cannot have duplicate names. So, in order for this to work, we should provide a randomly generated name in our **POST** request body and validate the presence of this name in the corresponding response body. Dynamic variables come in handy to achieve this in Postman. Variables are defined using double curly braces: `{{<variable name>}}`. The request body for the create repository **POST** call will look like this:

```
{
  "name": "{{repository_name}}",
  "private": false
}
```

Now, we need this value to be new for every **POST** request we send, and the best place to do this is in the **Pre-request Script** tab of the request dashboard. This represents a pre-condition to the test case, and it is run before the request is sent to the server. We accomplish random value generation by defining a variable with static and dynamic parts. Here, we use the **variables** property of the **pm** object. Then, we set this as an environment variable. [Figure 7.16](#) shows a new **repository\_name** environment variable created in the current environment, **Packt testing**:

```
let repository_name = "test_packt_api_" +
pm.variables.replaceIn('{{${randomInt}}');
pm.environment.set("repository_name", repository_name);
```

The following figure shows the **repository\_name** environment variable:

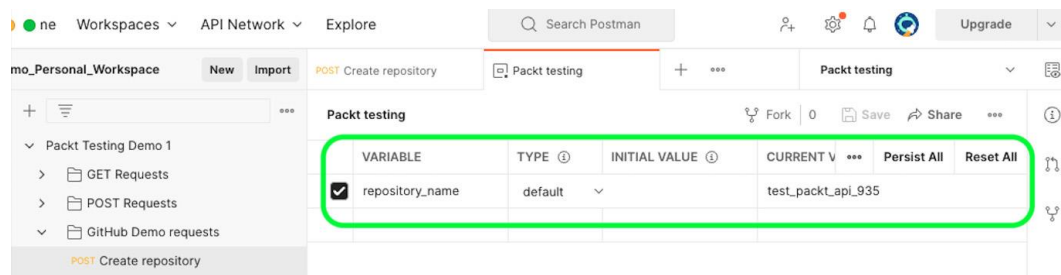


Figure 7.16 – A new environment variable created

#### Note

Pre-requisite scripts are run before the API request is executed, while the test scripts are run after the server returns a response.

So far, we have created an environment variable in the **Pre-request Script** tab and updated the request body to use that variable. Let us next add an assertion on this variable in the response body. This can be done in the **Tests** tab via the **Response body: JSON value check** snippet. This snippet helps check a specific value in the API response. Note that the value of an environment variable can be fetched using **pm.environment.get("repository\_name")**:

```
pm.test("Check repository name", function () {  
  var jsonData = pm.response.json();  
  pm.expect(jsonData.name).to.eql(pm.environment.get("repository_name"));  
});
```

In the next section, let us learn to chain a series of API requests by passing data from one API to the next.

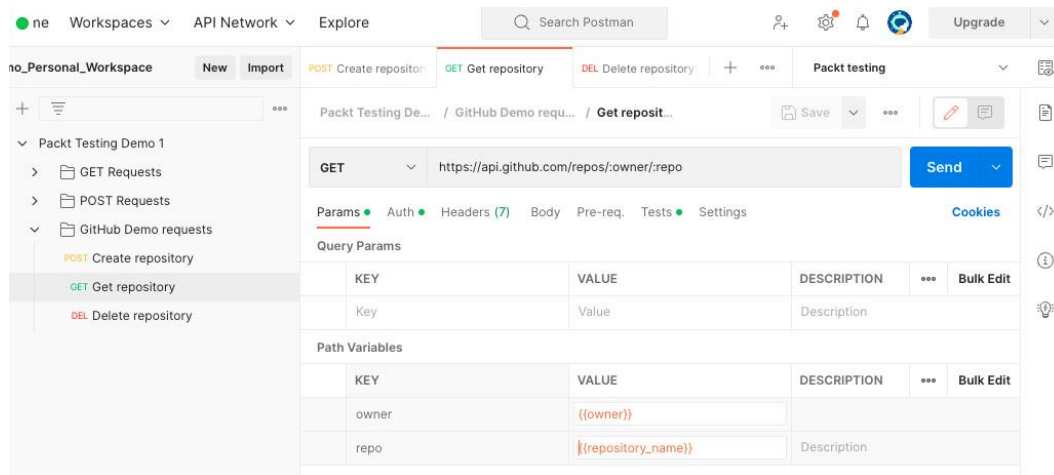
### Chaining API requests

Postman allows us to use variables to enable the chaining of a series of API requests. A variable created from the response of an API can be used in the subsequent request. Let us take the example of the create repository call where a new repository is created for every request. The name of this repository can be captured and used in a subsequent **GET** call.

For our understanding here, let us chain the create repository call with **GET** and **DELETE** calls. These calls require the name of the repository and owner. In the create repository call from the previous section, we have the **repository\_name** variable. Let us now capture the **owner** variable from the response body using the following code:

```
let json = JSON.parse(responseBody);  
pm.environment.set("owner", json.owner.login);
```

We use the **JSON.parse()** method to convert **responseBody** into a **JSON** object and then create an environment variable using the **login** key from the response. **GET** and **DELETE** calls both use the <https://api.github.com/repos/:owner/:repo> route. We create requests for each of these API requests with this route, and on saving, Postman automatically creates a new **Path variables** section in the **Params** tab of the request dashboard. We can now substitute the captured environment variables as values, as shown in **Figure 7.17**. In this way, we are chaining the response of the create repository call to the **GET** and **DELETE** calls:



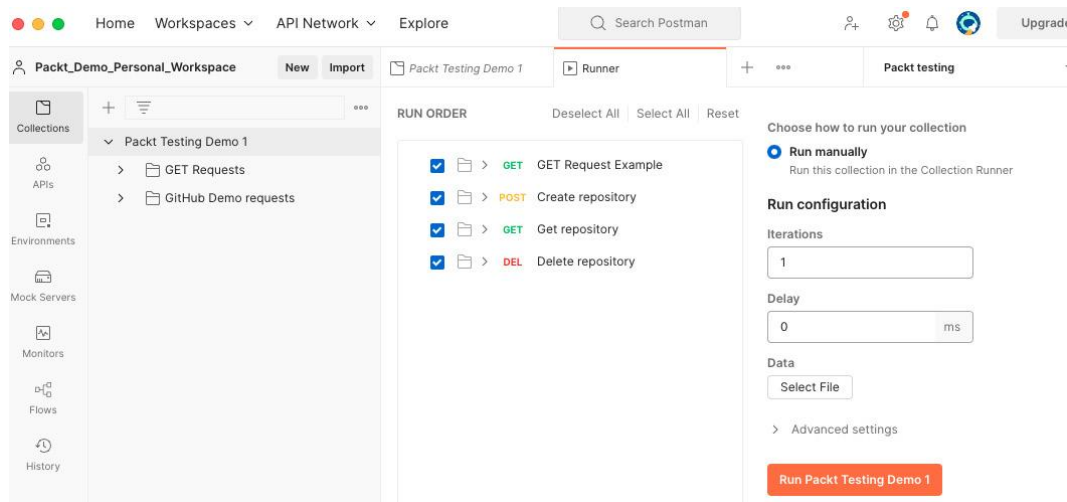
**Figure 7.17 – Chaining API requests**

On executing these requests one after another, we notice that the values from the request are being passed on to the next one, seamlessly eliminating static data transfer. This behavior makes Postman an effective tool for testing complex API workflows.

We now have a simple collection with multiple API requests, and it is not feasible to run each of them manually for every test cycle. In the next section, let us survey a few ways of executing tests in Postman.

### Various ways to execute tests

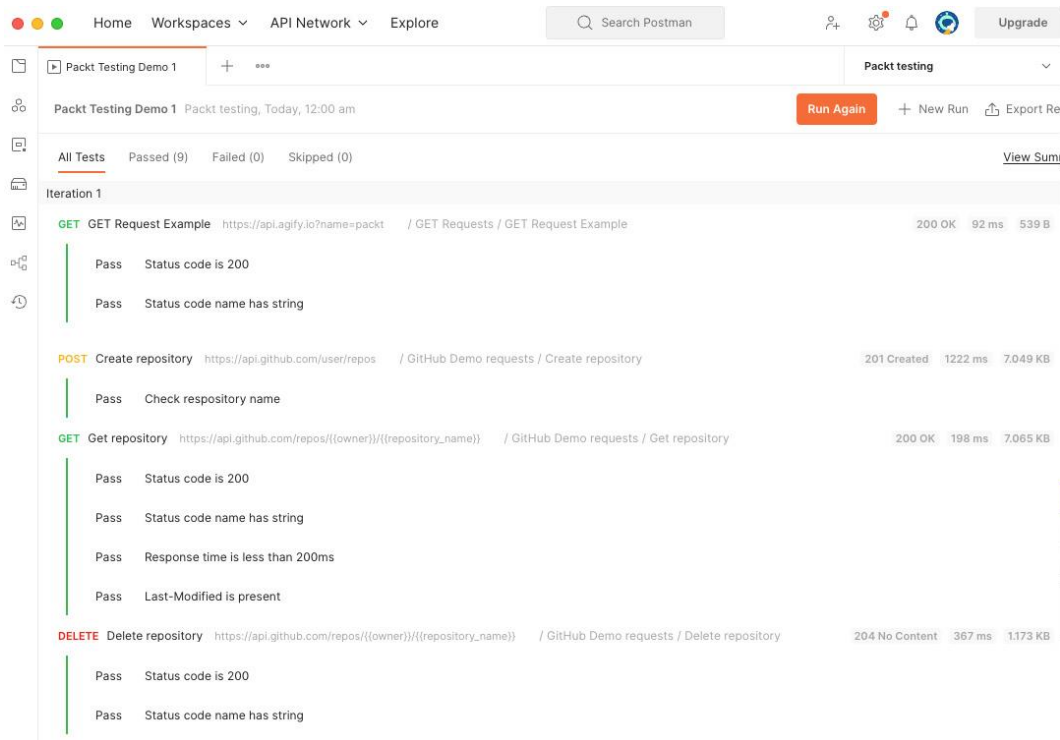
Let us first look at how to run our tests using the **Collection** runner. This is helpful when all tests in a collection must be run sequentially in an automated fashion. The **Collection** runner window can be launched by using the **Run** button from the **Overview** tab on clicking the collection name. This opens a new tab that displays all the requests in the collection and some additional run parameters. **Figure 7.18** shows this window and the associated options:



**Figure 7.18 – Collection runner window**

Using the **Iterations** option, users can specify the number of times the requests would be run. The **Delay** option helps to add a specified wait between subsequent requests. This is very useful in case of long-running requests. There is also an option to upload an external data file and use that data in

the form of variables within the request. On clicking the **Run** button, all requests in the collection are run, and results with a clear breakdown are populated in the same window, as shown in **Figure 7.19**:



**Figure 7.19 – Test results summary**

Postman also supports running a collection from the command line through a tool called **Newman**. Newman can be installed using the **npm** command: **npm install -g newman**. The installation can be verified using the **newman -v** command. To run the collection, we will first export the collections and the associated environment variables. A collection can be exported and saved to the local filesystem using the **Export** option from the **Collections** drop-down menu. Similarly, environment variables can be downloaded by using the **Export** option within **Environments** on the left navigation bar. Note that these files are downloaded in a JSON format. Now that we have the collection and its necessary variables, we can run it using Newman:

```
newman run packt_testing_collection.json -e packt_testing_environment.json
```

The output from the **run** command line is demonstrated in **Figure 7.20**:

```

→ packt_postman_downloads ls
packt_testing_collection.json packt_testing_environment.json
→ packt_postman_downloads clear
→ packt_postman_downloads ls
packt_testing_collection.json packt_testing_environment.json
→ packt_postman_downloads newman run packt_testing_collection.json -e packt_testing_environment.json
newman

Packt Testing Demo 1
├─ GET Requests
├─ GET Request Example
  │ GET https://api.agify.io?name=packt [200 OK, 539B, 620ms]
  │ ✓ Status code is 200
  │ ✓ Status code name has string
├─ GitHub Demo requests
├─ Create repository
  │ POST https://api.github.com/user/repos [201 Created, 7.09kB, 1237ms]
  │ ✓ Check repository name
├─ Get repository
  │ GET https://api.github.com/repos/manisam/test_packt_api_131 [200 OK, 7.11kB, 208ms]
  │ ✓ Status code is 200
  │ ✓ Status code name has string
  │ 1. Response time is less than 200ms
  │ ✓ Last-Modified is present
├─ Delete repository
  │ DELETE https://api.github.com/repos/manisam/test_packt_api_131 [204 No Content, 1.17kB, 377ms]
  │ ✓ Status code is 200
  │ ✓ Status code name has string

```

	executed	failed
iterations	1	0
requests	4	0
test-scripts	4	0
prerequisite-scripts	1	0
assertions	9	1

```

total run duration: 2.5s
total data received: 11.39kB (approx)
average response time: 610ms [min: 208ms, max: 1237ms, s.d.: 390ms]

# failure      detail
1. AssertionError Response time is less than 200ms
   expected 208 to be below 200
   at assertion:2 in test-script
   inside "GitHub Demo requests / Get repository"
→ packt_postman_downloads

```

Figure 7.20 – Collection CLI run

Postman provides integration with Docker for executing tests within a Docker container. Docker is a platform that assists in building, deploying, and testing your application code on units called containers, irrespective of the underlying operating system. It provides great portability for developing and testing applications. Running a collection on Postman’s Docker container involves just a couple of commands. Once you have Docker installed on your machine, run the **docker pull postman/newman** command. This command pulls the latest image of the Postman **docker/newman** runner from Docker Hub and sets up the container. Next, we need the URL of the collection to be able to run it externally. This can be obtained using the **Share** option from the **Collections** drop-down menu. Now, run the following Docker command:

```
run -t postman/newman run "<<Collection URL>>"
```



This brings us to the end of a basic exploration of API testing with Postman. The Postman tool has so much more to offer, and its capabilities can be referenced at <https://learning.postman.com/docs>. In the next section, let us review the considerations that go into API automation testing.

## Links

Postman's downloads page (<https://www.postman.com/downloads/>)

We will be using the <https://www.boredapi.com/api/activity> API to get an understanding of **GET** requests

For the case in *Figure 7.7*, <https://api.agify.io?name=packt>, a name is sent as a query parameter as part of the URL

# Chapter 8

## Technical requirements

To get functional with JMeter, we need Java installed on our machine. Currently, JMeter works with JDK 8 and JRE 8 or higher.

## Hands-on Sections

### Installing JMeter

Let's look at the steps involved in the installation of JMeter:

1. The first step in the installation of JMeter is to check the Java version on your machine. This can be done using the `java -version` command. As shown in [Figure 8.1](#), this command prints out the JDK version:

```
→ app git:(main) * java -version
openjdk version "17.0.4.1" 2022-08-12
OpenJDK Runtime Environment Temurin-17.0.4.1+1 (build 17.0.4.1+1)
OpenJDK 64-Bit Server VM Temurin-17.0.4.1+1 (build 17.0.4.1+1, mixed mode)
→ app git:(main) * █
```

**Figure 8.1 – Checking the Java version**

2. The next step is to download the JMeter binary. The file for download can be found on the JMeter website, [https://jmeter.apache.org/download\\_jmeter.cgi](https://jmeter.apache.org/download_jmeter.cgi). In this case, I am downloading the binaries zip file for version 5.5, as shown in [Figure 8.2](#).

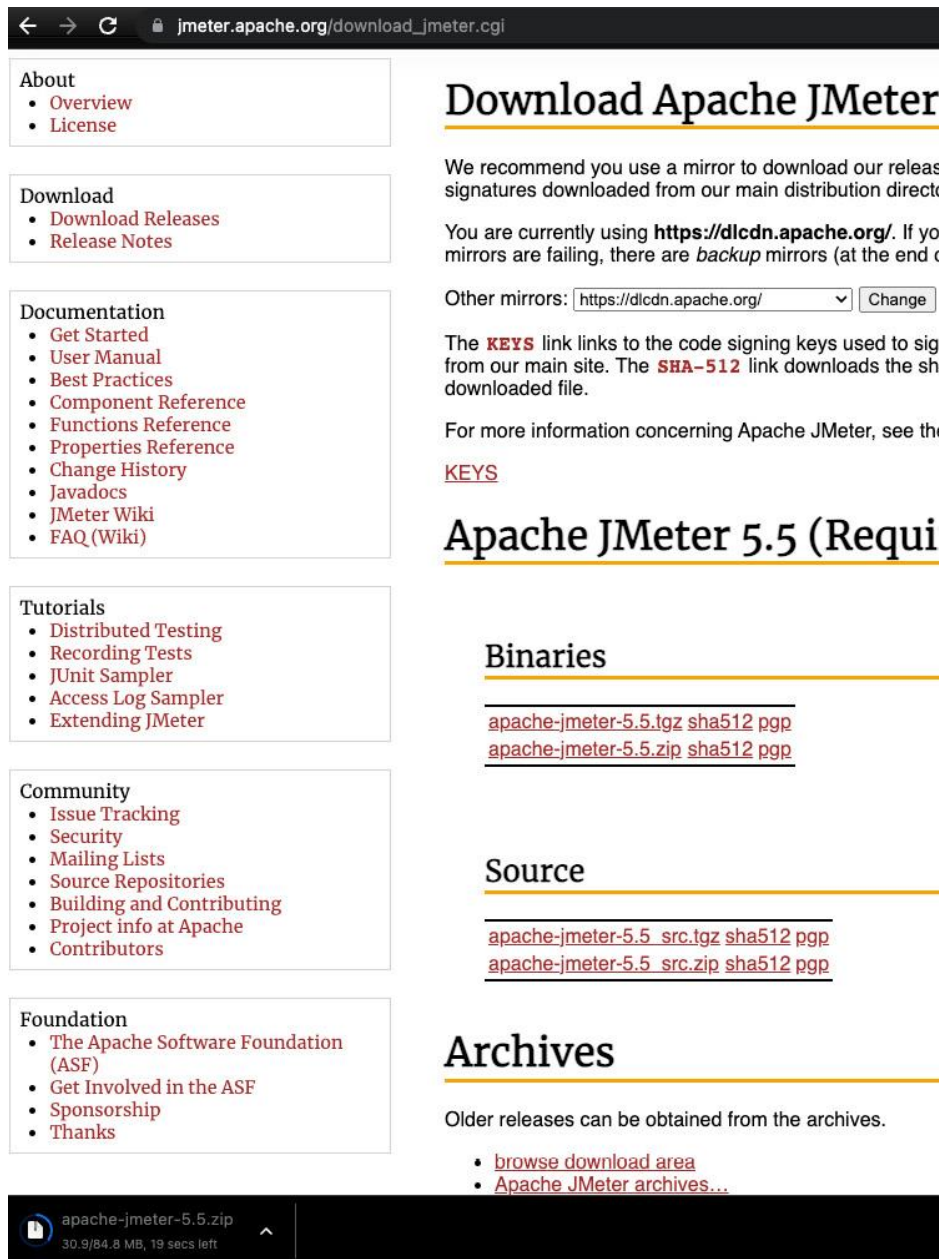
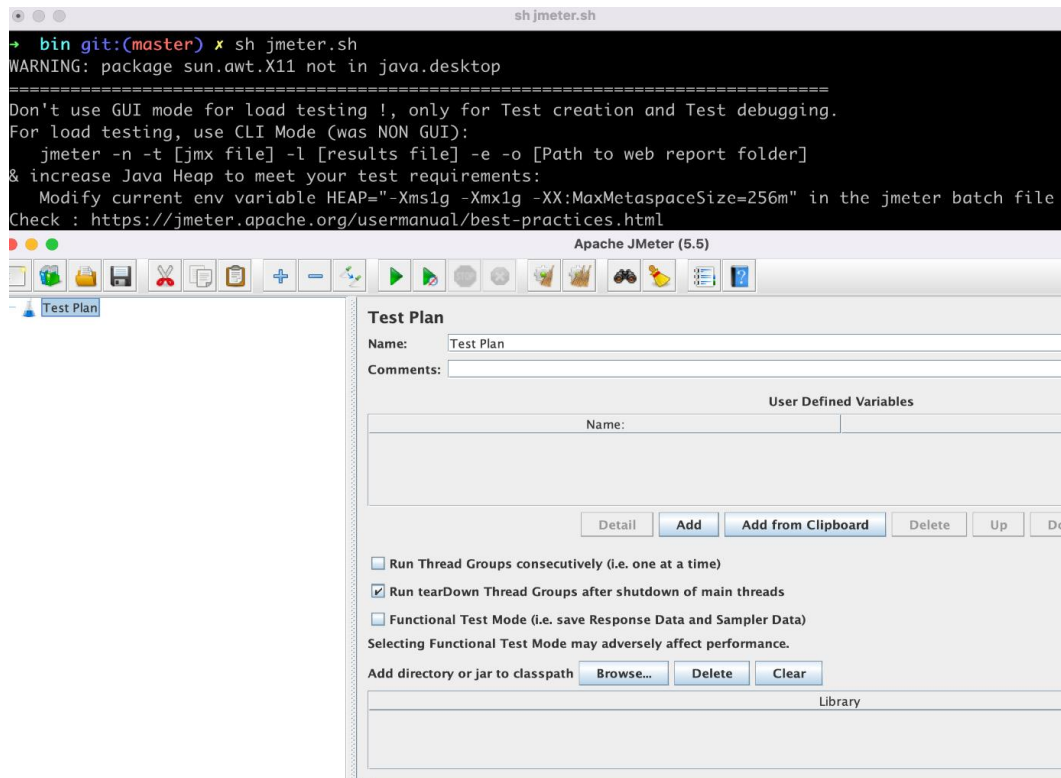


Figure 8.2 – JMeter downloads page

3. Once the download is complete, move the zipped file to the desired local folder and unzip it. This should create a new folder in the same location within which all the contents are extracted.

JMeter can now be started with the `sh jmeter.sh` command from the `bin` folder of the application. This brings up the application in a separate window, as shown in [Figure 8.3](#).



**Figure 8.3 – Starting JMeter**

JMeter comes with a simple GUI that contains the following components:

- Menu bar: Contains a collection of high-level options to set up and configure various aspects of the tool
- Tool bar: Contains frequently used tools
- Test plan tree view: Groups all components that are added within a test plan
- Editor section: Provides options to edit the selected component from the test plan view

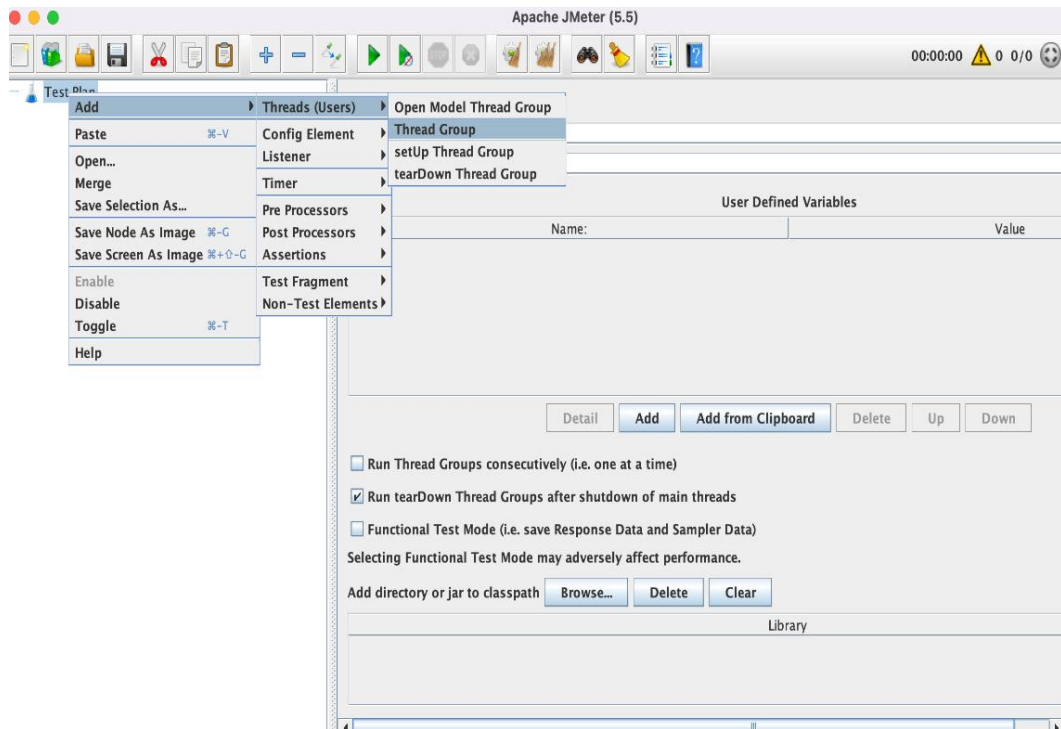
In the next section, let's look at how to create our first performance test in JMeter.

## Automating a performance test

JMeter provides an intuitive GUI that we can use to create and configure performance tests. The test plan is at the core of a performance test, and we start by creating one. We can do this by either using the **New** option from the menu bar or the tool bar. We looked at the new **Test Plan** window in [Figure 8.3](#) when launching the JMeter application.

## Building and running our first performance test

One of the primary focuses of a performance testing tool is its ability to simulate multiple users. This is accomplished in JMeter by configuring a thread group. As shown in [Figure 8.4](#), this is done via the **Thread Group** option under **Test Plan**.

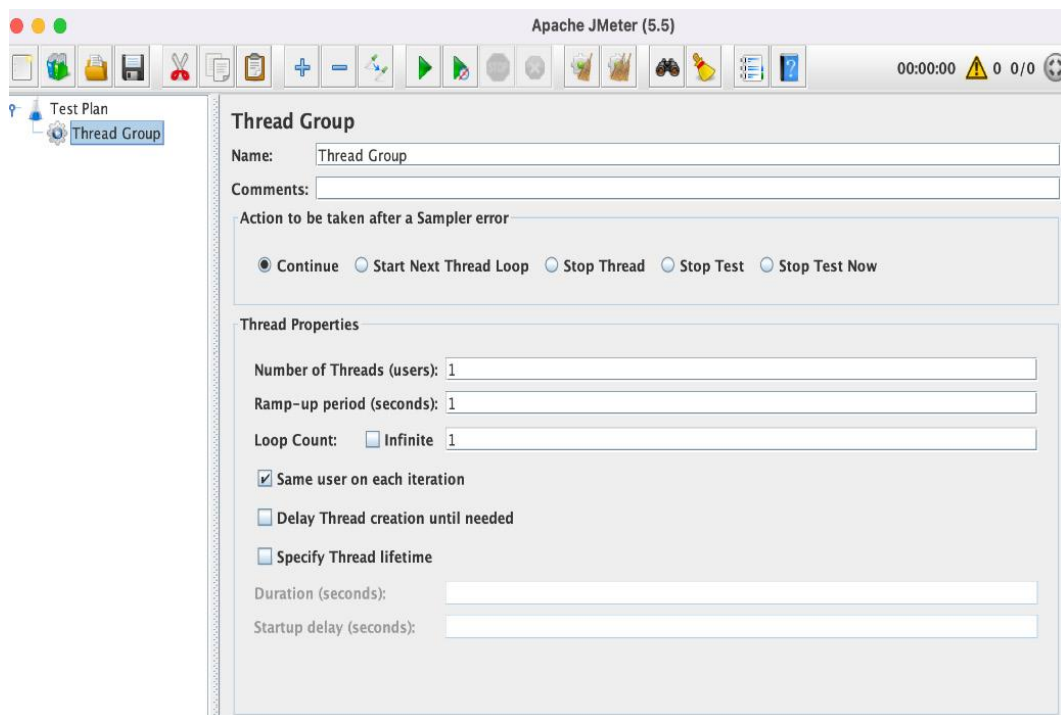


**Figure 8.4 – New thread group**

We use a combination of three parameters to achieve the required pacing for our performance test:

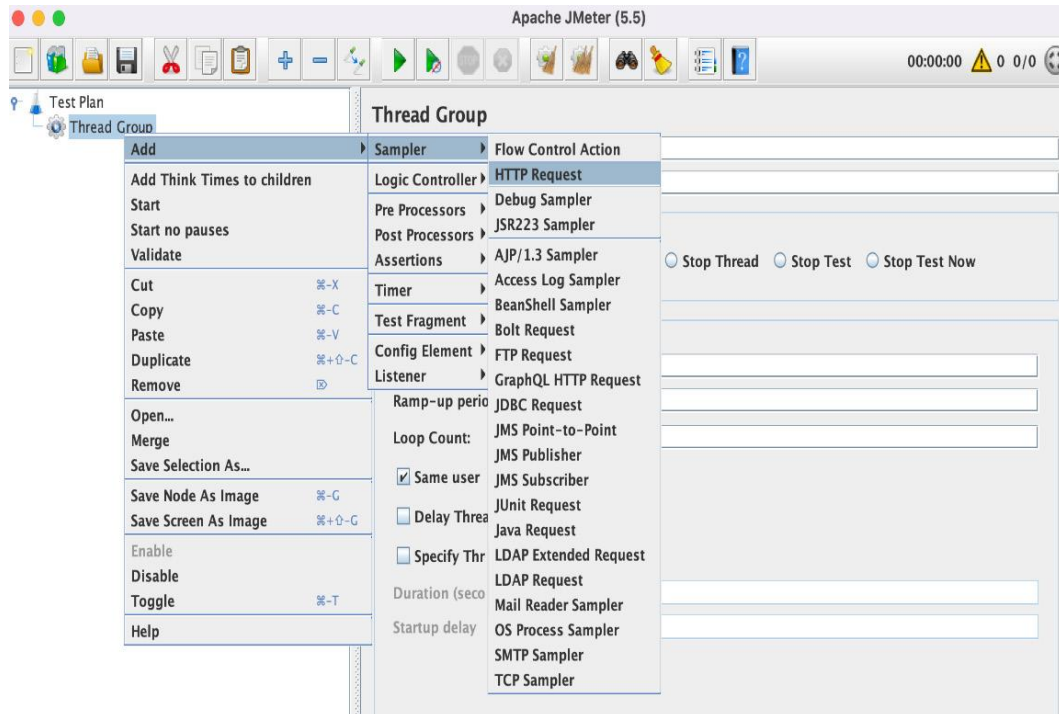
- **Number of Threads:** The number of parallel users to be simulated in this test
- **Ramp-up period:** The time taken to simulate the specified number of users
- **Loop Count:** The number of iterations to be executed as part of the current test

There are additional settings to configure and fine-tune the load on the test, as shown in **Figure 8.5**.



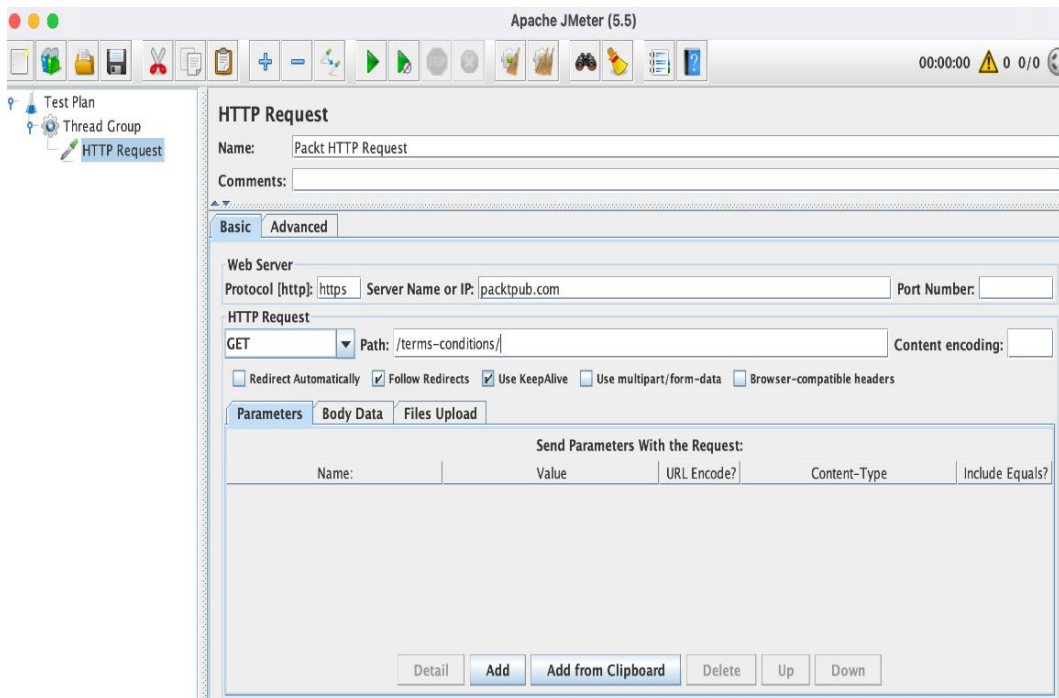
**Figure 8.5 – Thread Group configuration**

The next step is to add a sampler to the test plan. A sampler is nothing but a test added in JMeter. **Figure 8.6** shows the list of the samplers supported by JMeter. Let's now add a simple HTTP sampler for our test. We can then use the HTTP editor to configure our test.



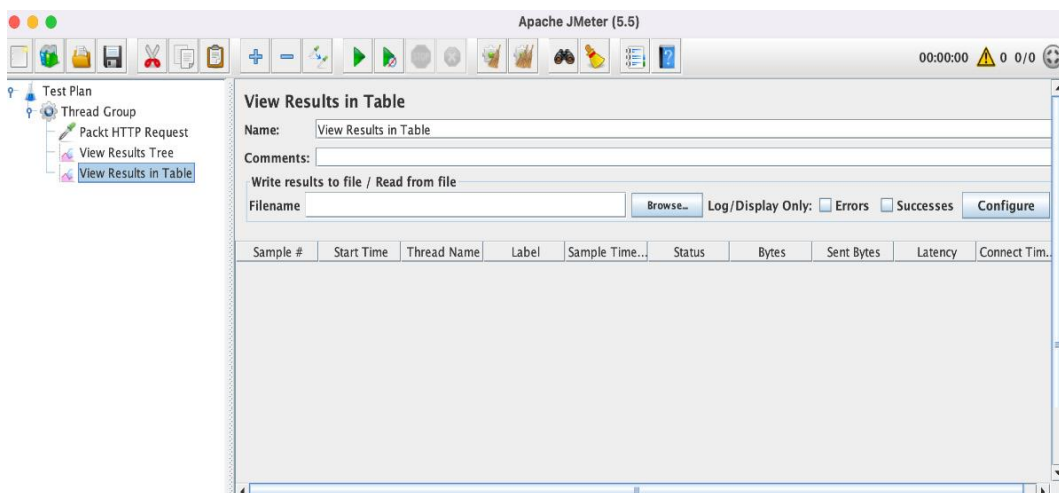
**Figure 8.6 – Adding an HTTP sampler**

In our example, we will be load-testing the Packt Publishing website at <https://packtpub.com>. The URL is split between the fields, protocol, and server name in the editor. Then we specify the path, **/terms-conditions**, in the **Path** field. We will be testing the GET request, but there are in-built options to support other types of requests, along with request body and file uploads, as shown in **Figure 8.7**.



**Figure 8.7 – Configuring an HTTP sampler**

The test plan can now be saved to a local directory using the **Save** option from the menu bar. The next step is to add a **listener**, which helps us view the test results. A listener is a component within a test plan that stores and allows us to view results. Let's add the **View Results Tree** and **View Results in Table** listeners to our test plan. JMeter provides a variety of listener options in the **Add | Listener** menu. **Figure 8.8** shows our test plan with the listeners added.



**Figure 8.8 – Listeners in a test plan**

After saving the test plan, we are ready to execute our first test. This is done using the **Start** button on the menu bar. We can see the results being populated in the listeners as soon as the test begins. **Figure 8.9** shows a breakdown of the test run stats by the thread group within the **View Results in Table** listener.



View Results in Table

Name: View Results in Table

Comments:

Write results to file / Read from file

Filename:   Log/Display Only:  Errors  Successes

Sample #	Start Time	Thread Name	Label	Sample Time...	Status	Bytes	Sent Bytes	Latency	Connect Tim...
1	00:30:59.688	Thread Grou...	Packt HTTP ...	2270	✓	8077	430	1384	1349
2	00:31:02.648	Thread Grou...	Packt HTTP ...	1882	✓	8077	430	143	105
3	00:31:05.648	Thread Grou...	Packt HTTP ...	1494	✓	8076	430	131	79
4	00:31:08.650	Thread Grou...	Packt HTTP ...	1811	✓	8077	430	89	60
5	00:31:11.645	Thread Grou...	Packt HTTP ...	1308	✓	8077	430	111	72
6	00:31:14.649	Thread Grou...	Packt HTTP ...	1284	✓	8076	430	108	80
7	00:31:17.649	Thread Grou...	Packt HTTP ...	1249	✓	8077	430	102	58
8	00:31:20.646	Thread Grou...	Packt HTTP ...	1296	✓	8077	430	91	59
9	00:31:23.646	Thread Grou...	Packt HTTP ...	1141	✓	8077	430	120	76
10	00:31:26.644	Thread Grou...	Packt HTTP ...	1664	✓	8076	430	88	54

Scroll automatically?  Child samples? No of Samples 10 Latest Sample 1664 Average 1539 Deviation 341

**Figure 8.9 – Test run results**

JMeter provides options to configure the fields in the results. Some important fields to look out for in the results are **Sample Time(ms)**, **Latency**, and **Status**, as they specify the status of the test and the time taken to get a response from the server. JMeter offers a convenient option to save and view the test results in CSV or XML format.

### Sample time versus latency

Latency is the time taken by the server to return the first byte of the response, whereas sample time is the total time taken by the server to return the complete response. Sample time is always greater than or equal to latency.

### Working with assertions

Assertions, as we have seen in previous chapters, are the checks performed on the request and response. JMeter provides options to perform checks on an array of options, such as response size, response time, the structure of the response, and so on. An important thing to note about assertions is that they can be added at all levels. For example, an assertion added at the test plan level will apply to every sampler within it. For our example, let's add response and duration assertions for the HTTP sampler, as shown in **Figure 8.10**.

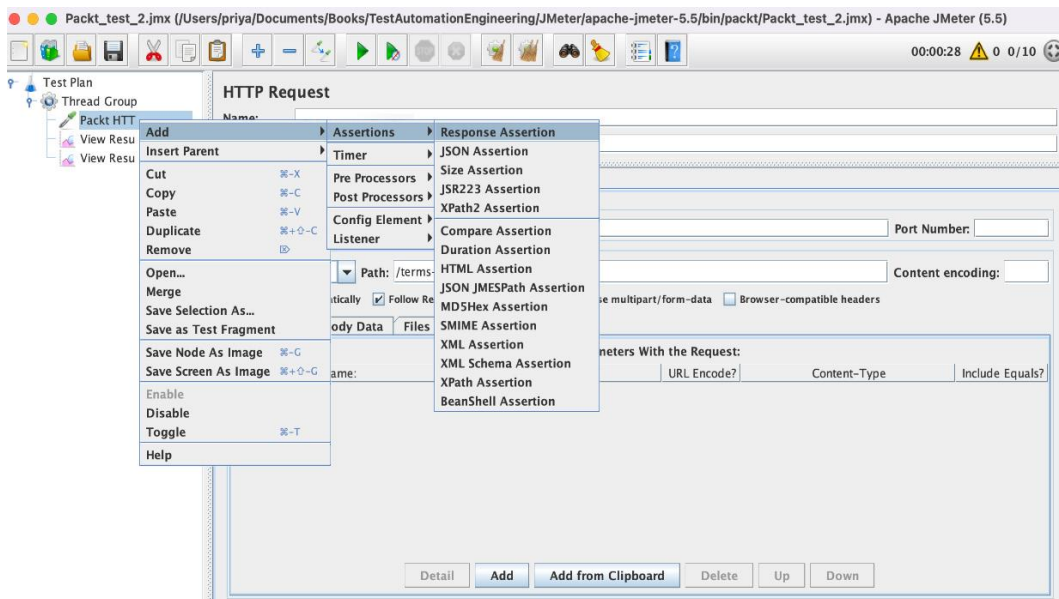


Figure 8.10 – Adding assertions

Let's update the **Response Assertion** to look for the response code 200 and the **Duration Assertion** to flag responses over 1,000 ms. These conditions are checked after every iteration of the HTTP sampler, and the results are flagged accordingly. **Figure 8.11** demonstrates the execution of the **Duration Assertion** where some of the responses took over a second to complete.

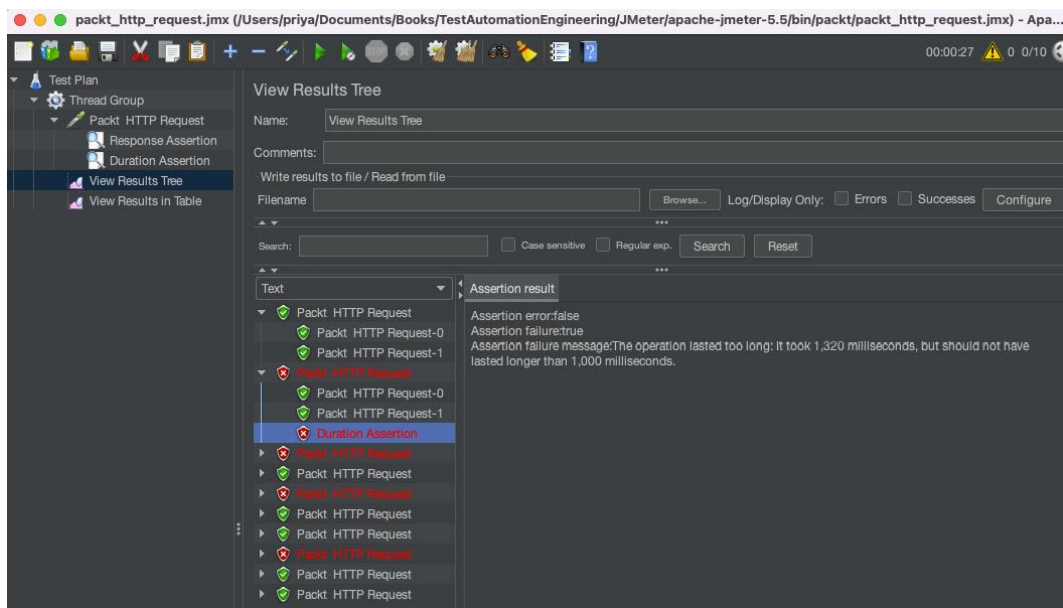
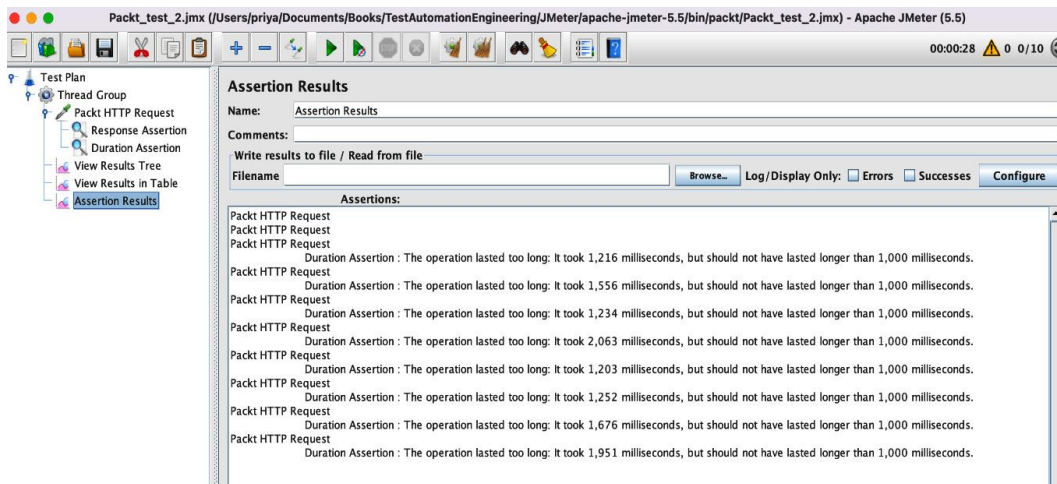


Figure 8.11 – Assertion results

The **Assertion Results** listener is an effective component that collates the responses from all the assertions so you can view them in one place. This listener can be added at the test plan level, as illustrated in **Figure 8.12**. It combines the results from the **Response Assertion** and the **Duration Assertion**.



**Figure 8.12 – Assertion results listener**

Let's now look at how to use the command line to handle JMeter's tests.

## Working with tests via the command line

Performance tests are often long-running and tend to be heavy on system resource consumption. GUI mode consumes a lot of memory, especially when running pre-recorded scripts, and execution via the command line alleviates this pain by reducing the memory footprint of the tool. Another significant benefit is the ability of the command line to integrate easily with external processes, such as continuous integration systems. In this section, we will learn how to configure and run a JMeter test from the command line.

Let's reuse the test plan from the previous section for execution via the command line. Navigate to JMeter's **bin** folder in your command line and run the following command:

```
sh jmeter -n -t "<location of the .jmx test plan>" -l "<location to log the results>"
```

Here, **-n** stands for non-GUI mode, **-t** specifies the location of the test plan, and **-l** is the location of the result logs. The command line supports various other parameters, but these are the minimum required parameters to trigger the execution. **Figure 8.13** shows the execution of a command line run.

```
Starting standalone test @ 2022 Nov 7 17:21:47 PST (166787057589)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary + 5 in 00:00:13 = 0.4/s Avg: 1029 Min: 728 Max: 1287 Err: 5 (100.00%) Active: 1 Started: 5 Finished: 4
summary + 5 in 00:00:15 = 0.3/s Avg: 959 Min: 860 Max: 1166 Err: 5 (100.00%) Active: 0 Started: 10 Finished: 10
summary = 10 in 00:00:28 = 0.4/s Avg: 994 Min: 728 Max: 1287 Err: 10 (100.00%)
Tidying up ... @ 2022 Nov 7 17:22:15 PST (1667870535505)
... end of run
+ bin
```

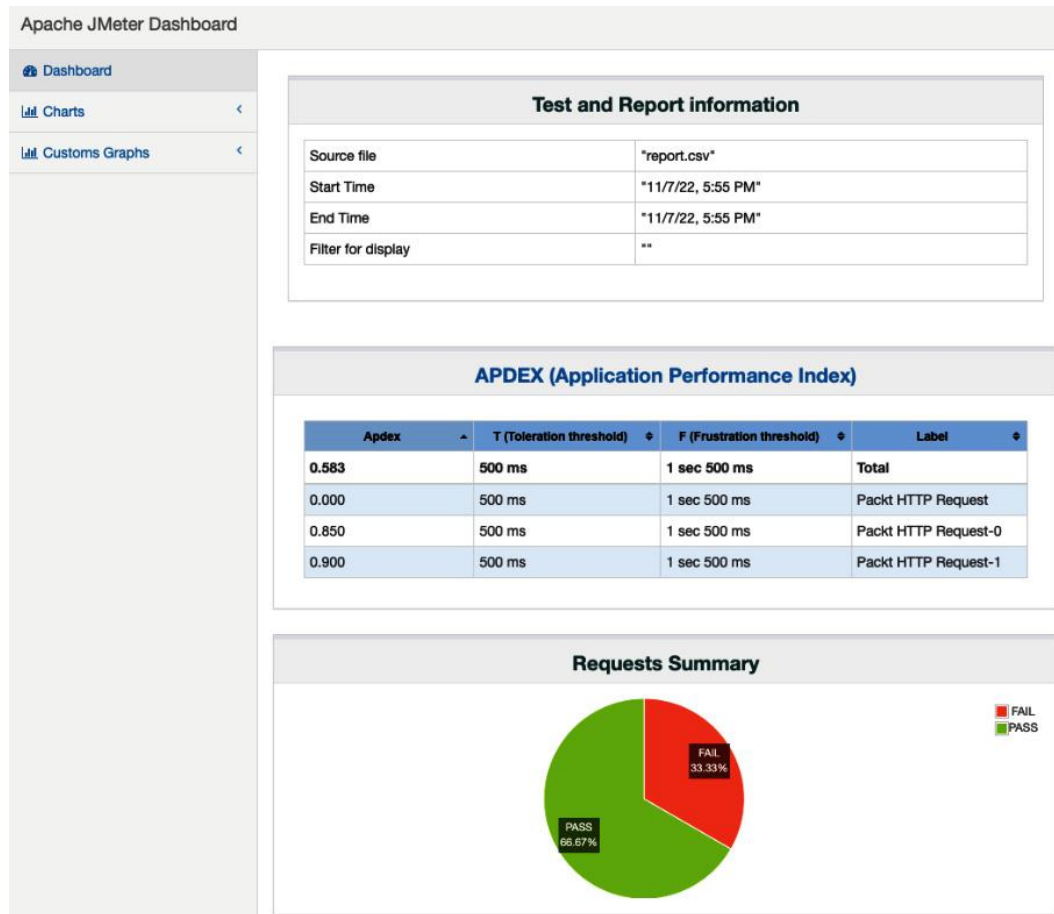
**Figure 8.13 – JMeter command line run**

Additionally, the **sh jmeter -h** command can be used to review all the available command-line options.

Performance test results can get voluminous, and it is always necessary to produce a clear and concise report. It will be hard to understand the test results with just the results shown on the command line and it necessitates a better report. This is achieved by using the **-e** option, which

generates a dashboard report, and the `-o` option to specify the location of the results folder. **Figure 8.14** shows a part of the HTML report generated when using these parameters. By default, this is produced as an `index.html` file within the results folder specified as part of the command-line option. The full command to achieve this is as follows:

```
sh jmeter -n -t "./packt/packt_http_request.jmx" -l
"./packt/report.csv" -e -o "./packt/dashboard_report"
```



**Figure 8.14 – JMeter Dashboard report**

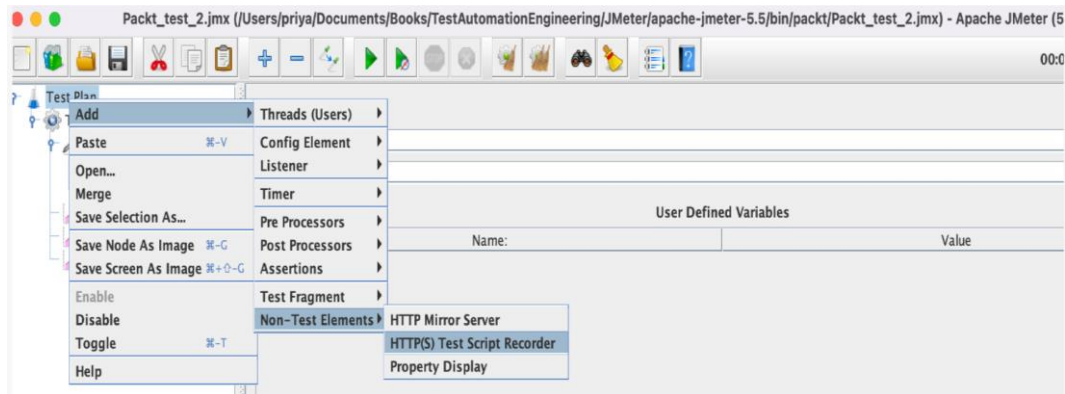
Another powerful feature that the JMeter command line provides is the use of built-in functions to send dynamic parameters when running a test plan. For example, in our test plan, we have hardcoded the path as `/terms-conditions`. In real time, we would be testing different paths from the command line and would not have to update the test plan for every run. The test plan can be updated with a function in this field to be able to receive this parameter via the command line using the format `$_P(VariableName)`. The path can now be sent through the command line by prefixing `J` to the variable name:

```
sh jmeter -n -t "./packt/packt_http_request.jmx" -Jpath=/terms-conditions
```

In the next section, let's look at how to use the **HTTP(S) Test Script Recorder** component in JMeter.

## Using the HTTP(S) Test Script Recorder

The **HTTP(S) Test Script Recorder** is a component that records requests from the browser. Previously, we manually added the HTTP request, but this component adds them automatically by recording the transactions. This option can be added directly under the test plan, as shown in **Figure 8.15**.



**Figure 8.15 – Adding the HTTP Test Script Recorder component**

We will also need a recording controller to be added to the test plan to categorize the recording by the traffic or per page. For simplicity, let's use a single controller here, but in real-world scenarios where the user flows involve multiple pages, we might need a separate controller per page. The **Target Controller** property should be set to point to the right controller within the **HTTP Test Script recorder**. Another notable feature within the **HTTP Test Recorder** component is **Request Filtering**. A lot of resources are exchanged when recording HTTP requests, and not all of them will be applicable for load testing. URL patterns that need to be included or excluded can be specified using the **Request Filtering** option.

The next step is to configure the proxy on our browser so that only the desired traffic flows through the port. This is done by specifying the default JMeter port **8888** within the browser's proxy configuration. **Figure 8.16** shows this configuration on the Chrome browser.

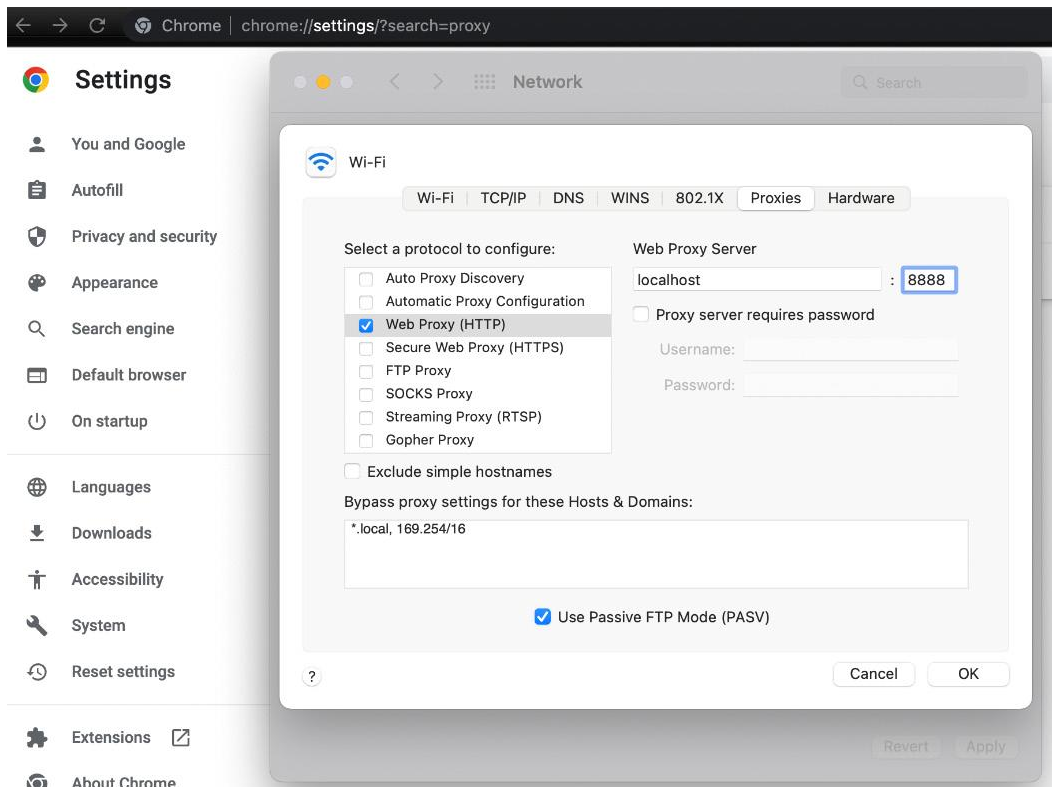


Figure 8.16 – Chrome proxy configuration

There is one more step before we can start recording, and that is to add the JMeter certificate to the browser. This file (**ApacheJMeterTemporaryRootCA.crt**) can be found in JMeter’s **bin** folder, and it needs to be added to the browser certificates via settings. Once this is done, we can use the **Start** button on the recorder component to commence the recording. When the recording is complete, the HTTP requests are stored under the corresponding controller. These requests can then be played back with the simulated load.

We have gained foundational knowledge on how JMeter operates, and we recommend you to further explore the tool using the user manual at <https://jmeter.apache.org/usermanual/index.html>. Let’s move on to the next section to gain a basic understanding of the Java programming language and how to use it to write custom code within JMeter.

## Java essentials for JMeter

There may be instances where the features that come out of the box with JMeter are not sufficient and custom scripts are needed to perform specific tasks. JSR233 and Beanshell assertions/samplers can be utilized in cases such as these to get the job done. Both these components support Java code, and hence it is important to acquire basic Java knowledge. In this section, let’s go through a quick introduction to the Java programming language.

### A quick introduction to Java

Java is a platform-independent compiled programming language. Java code gets compiled into bytecode, which can then be executed on any OS. The **Java Virtual Machine (JVM)** is the OS-specific

architectural component that sits between the compiled bytecode and the OS to make it work on any platform. Let's now create our first Java program, compile it, and run it. Any Java program comes with a boilerplate code, as follows:

```
package ch8;

public class first_java_program {
    public static void main(String[] args) {
    }
}
```

Let's familiarize ourselves with these keywords, to begin with. Whenever a new class is created in Java, the very first line is usually the package name, followed by the class definition. The **public** keyword is an access modifier that denotes the access level of this class. This is followed by the **class** keyword and the name of the class. Within the class, there is always a **main** method with a **public** access modifier.

This is followed by another keyword, **static**, which signifies that this method can be invoked directly without the need to create an instance of the class. The **main** method is always called by the JVM at the beginning of program execution, and that is why we do not need an instance of the class to call this method. Next is **void**, which represents the return type of this method. In this case, we do not return anything and hence leave it as **void**. We could return a string or an integer depending on what is being done within the method. The values within the parentheses after the method name mark the arguments accepted by the method.

Let's add a simple **print** statement, **System.out.println("My first java program")** within the **main** method and run it via the IDE. This should print the text specified within the **println** method. This completes our first program in Java. Java is a strong object-oriented language, so let's learn how to create classes and objects in Java.

Object-oriented programming primarily helps us model real-world information in our programs. Let's take an example of a bank account and see how it can be modeled in Java using object-oriented techniques. To start with, let's create an **Account** class, as follows:

```
package ch8;

public class Account {
    String account_holder_name;
    int age;
    float account_balance;
    boolean direct_deposit_enabled;
    Boolean maintains_minimum_balance;
    public void test_minimum_balance() {
    if (account_balance > 5000) {
    maintains_minimum_balance = true;
    }
}
```



```
}  
}
```

This class would act as a template for all the accounts that are created. Each account created from this template would be an object, or an instance, of this class. We have used different variable types to model real-world information. We have also used the **test\_minimum\_balance** method to derive and set the value of a variable called **maintains\_minimum\_balance** within the class.

Let's now go ahead and create another class that holds these objects:

```
package ch8;  
  
public class AccountObjects {  
    public static void main(String[] args) {  
        Account johns_account = new Account();  
        Account davids_account = new Account();  
        johns_account.account_holder_name = "John Doe";  
        johns_account.age = 32;  
        johns_account.account_balance = 10000;  
        johns_account.direct_deposit_enabled = true;  
        johns_account.test_minimum_balance();  
        tims_account.account_holder_name = "Tim Sim";  
        tims_account.age = 35;  
        tims_account.account_balance = 1000;  
        tims_account.direct_deposit_enabled = true;  
        tims_account.test_minimum_balance();  
    }  
}
```

We have created two objects in our second class, which represent two different people's bank accounts. This example demonstrates how we can use classes to model information.

### JDK versus JRE versus JVM

**JDK:** The **Java Development Kit** is an environment for developing, compiling, and running Java applications.

**JRE:** The **Java Runtime Environment** is an environment for running Java applications. Users of Java applications just need the JRE.

**JVM:** The **Java Virtual Machine** is an interpreter for executing Java programs.

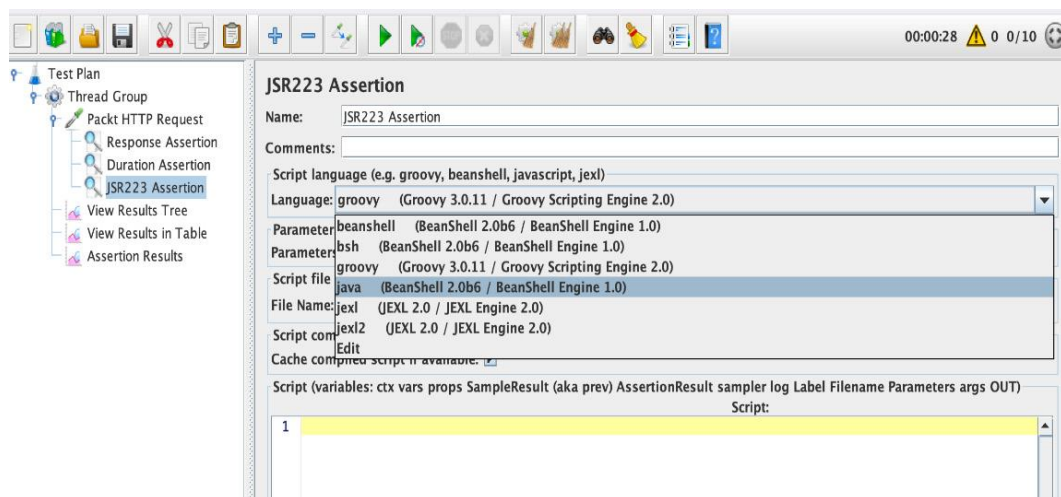
This section was meant to quickly inform you what the Java programming language is and how to write a basic program. You are encouraged to refer to the official Java documentation at <https://docs.oracle.com/javase/tutorial/getStarted/index.html> to further your knowledge. Let's now get back to writing custom scripts in JMeter.

## Using the JSR233 assertion

JMeter comes with a JSR233 assertion/sampler that can interpret and execute Java code. JSR233 is a scripting API for languages that can work on the JVM. Apache Groovy, Python, and Ruby are some of the supported languages, and we will be using Groovy for our example as it provides better performance.

Another advantage of using Groovy is that it is an extension of the JDK and accepts Java code. In fact, it supports all the features of Java and provides additional dynamic features, whereas Java is a strongly typed language. Groovy's official documentation can be found at <https://groovy-lang.org/documentation.html>. Since the Groovy engine is part of JMeter, no additional installation is required to get it working. Let's now look at how to employ a Groovy script within a JMeter test plan using the JSR233 sampler/assertion.

To start, let's add the JSR233 assertion to the HTTP request in our existing test plan. By default, Groovy is selected as the language for this assertion, but there are other options, as shown in **Figure 8.17**.



**Figure 8.17 – Adding a JSR233 sampler**

One of the top uses of employing custom scripting within JMeter is to enhance the logging capabilities wherever needed. This helps tremendously in reducing debugging effort. For example, the statement **log.info("Output from the log message")** can be used to print additional logging messages. Now consider the following code block, which can be run as part of the JSR233 assertion:

```
int thread_run_time = SampleResult.getTime();
int thread_latency = SampleResult.getLatency();
int response_threshold = 1000;
if ((thread_run_time+thread_latency)>response_threshold) {
    AssertionResult.setFailure(true);
}
```

```
AssertionResult.setFailureMessage("Threshold exceeded");  
}
```

**SampleResult** is a built-in JMeter object through which various properties of the test result can be accessed. Here, we are getting the run time and latency of the HTTP response and using an **if** statement to perform an assertion. Custom scripting thus extends JMeter's ability to perform specific validations.

Another area where custom scripting can be used is with getting and setting values of variables and properties. It might be necessary to dynamically change the value of a variable based on the test result. This can be performed with the following statements:

```
failure_count = vars.get("failure_count");  
Failure_count++;  
vars.put("failure_count", String.valueOf(failure_count));
```

We are getting the value of the **failure\_count** variable and incrementing it before writing it out. As you can see, custom scripting opens up various ways to extend our tests to address project-specific needs. This is as far as we can go here; it's up to you to explore it further.

In the next section, let's explore some considerations for performance testing.

# Chapter 9

## Technical requirements

We will be working on GitHub Actions in the last part of this chapter to implement a CI job. The repository used will be <https://github.com/PacktPublishing/B19046-Test-Automation-Engineering-Handbook>. It is advised to possess a basic familiarity with the GitHub UI and how it works to follow along.

## Figures

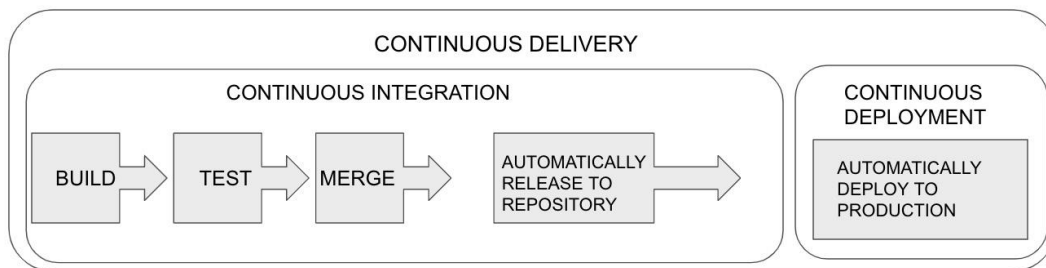


Figure 9.1 – CI/CD

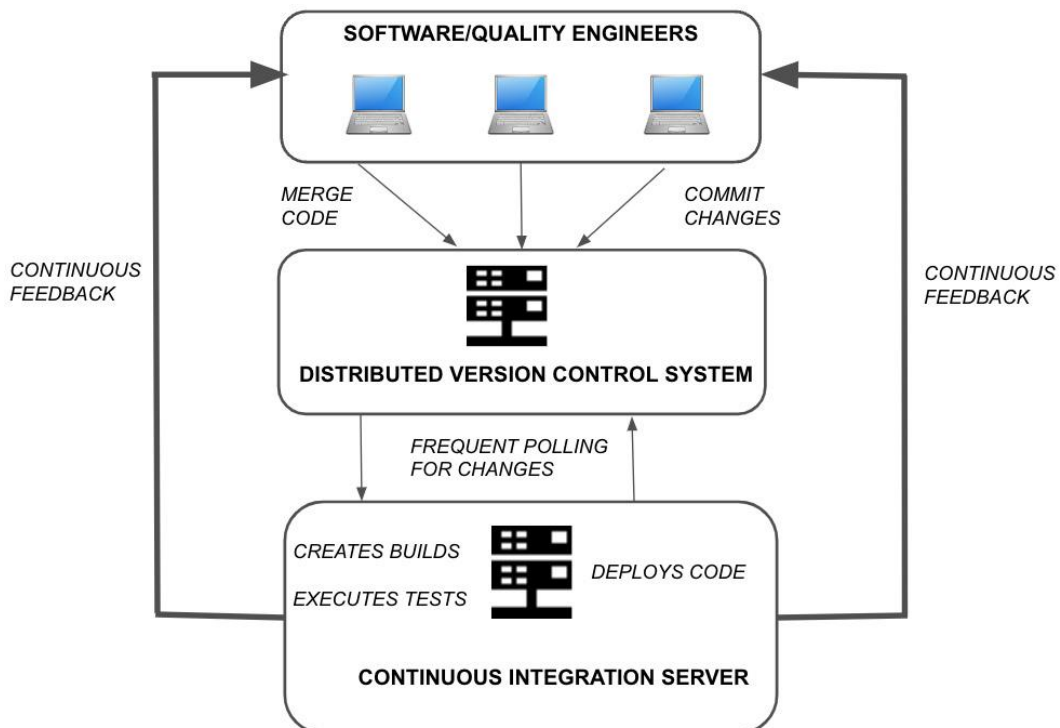
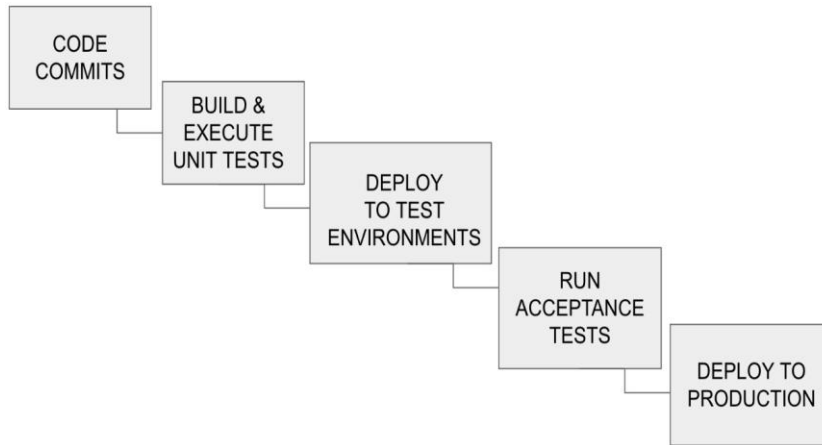
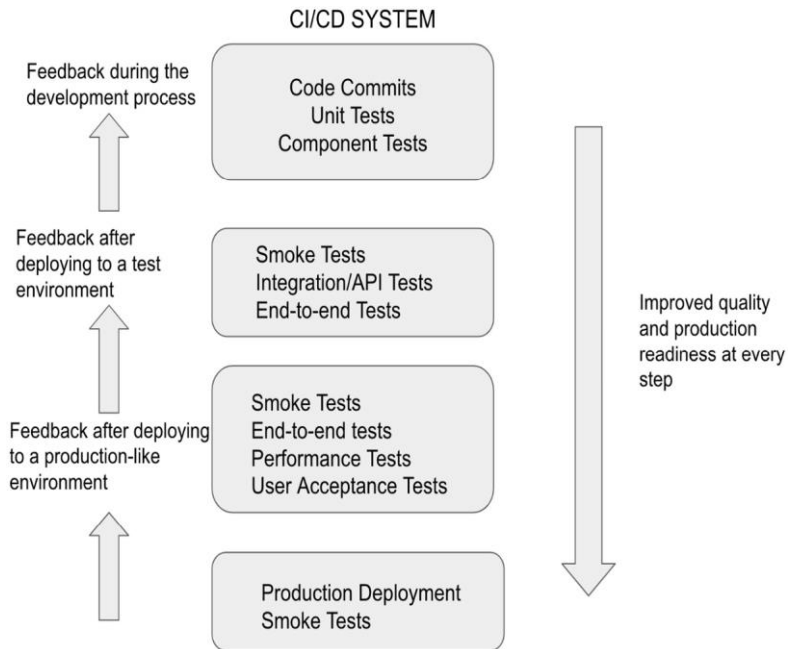


Figure 9.2 – Parts of a CI system



**Figure 9.3 – Components of a deployment pipeline**



**Figure 9.4 – Feedback loop in a CI/CD system**

## Table

Type of Test	Recommended CI/CD Strategy
Unit/component tests	Tests with minimal dependencies and the quickest feedback cycle to be run on every commit and every merge to master
API tests	Tests that verify the functional correctness of the API endpoints to be run on every merge to master

Type of Test	Recommended CI/CD Strategy
E2E API tests	Long-running tests involving sequential API calls to test business workflows to be run on every deployment to test environments
E2E UI tests	Long-running tests involving user actions to test business workflows to be run on every deployment to test environments
Smoke tests	A subset of tests selected to be run on every deployment to an environment

**Table 9.1 – CI/CD strategies for various test types**

## Hands-On Sections

### GitHub Actions CI/CD

GitHub Actions is a CI/CD platform that enables the automation of building, testing, and deployment of application code. It is the built-in CI/CD tool for GitHub. In this section, let us go over all the concepts we need to know to understand the GitHub Actions workflow. We will also learn to implement a GitHub action to run syntax checks against our code to make sure it meets specific criteria. Let us start with the necessary terms to help us understand the GitHub Actions workflow file.

The workflow **.yaml** file contains all the information used to initiate and drive the CI pipeline to completion. YAML is a data-serialization language commonly used for building configuration files. It is in human-readable format and compatible with all the major programming languages. The workflow **.yaml** file at a high level specifies the following:

- **Events:** An event is a trigger for a workflow
- **Jobs:** Jobs are high-level actions performed as part of the workflow
- **Runners:** A runner is a platform where the action is performed
- **Steps:** A job can be broken down into multiple steps
- **Actions:** Each step performs a specific action in an automated fashion

For illustration, we will be using code commits and merges, which are common events that occur in every repository. In this example, we will be configuring our workflow file to be triggered when someone pushes code to our repository. When this push event occurs, all jobs within the workflow will be run. This is demonstrated by the YAML code snippet shown next. In this configuration file, we use the **on** parameter to specify the trigger for the workflow. When the **push** event occurs, it will run all jobs within this workflow. We have a single job here that comprises multiple steps and actions. Under the steps, two actions will be run in this case. The first action will check out the latest version of our code from the main/master branch, and the next one will run the super-linter against it. Linters are tools to evaluate that our code conforms to certain standards. The super-linter supports multiple languages and automatically understands and checks any code in the specified repository. The **runs-on** parameter is used to specify the runner. This is the container environment where GitHub will run this job. There are additional options to host your own container; however, we will be sticking to the default container offered by GitHub in this case:

```

name: Packt CICD Linter Demo

on: [push]

jobs:
  super-lint:
    name: Packt CICD Lint Job
    runs-on: ubuntu-latest

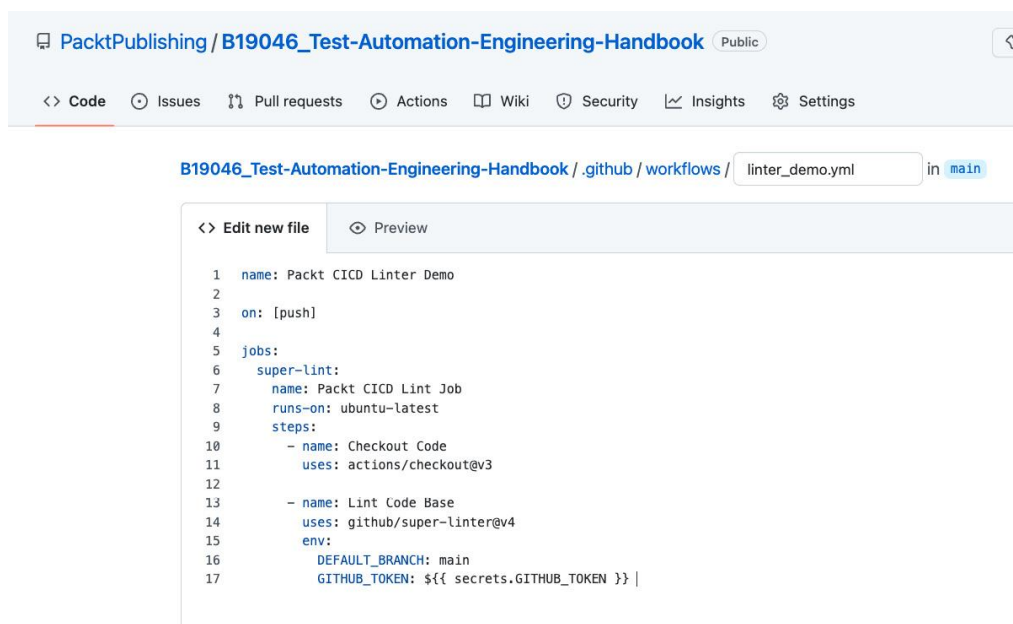
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Lint Code Base
        uses: github/super-linter@v4

    env:
      DEFAULT_BRANCH: main
      GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }

```

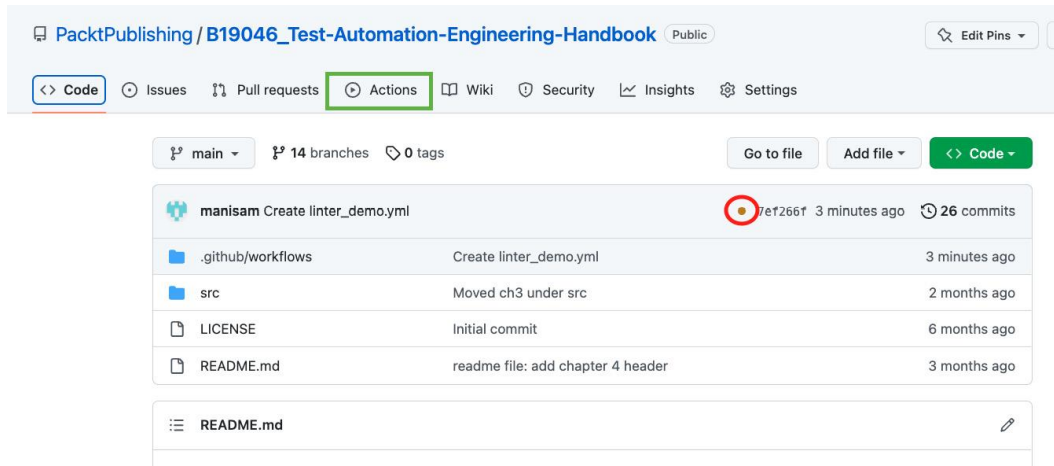
Let us now go to GitHub to set up a workflow in our repository ([https://github.com/PacktPublishing/B19046\\_Test-Automation-Engineering-Handbook](https://github.com/PacktPublishing/B19046_Test-Automation-Engineering-Handbook)). First, we create the right folder structure for our workflow file. We use the **Add File** option on the home page of our repository. We create a **linter\_demo.yml** file with a **.github/workflows** structure under the root folder of the project and copy the code into the editor below, as shown in **Figure 9.5**. Then, this file can be committed through a new branch or to the main branch directly. It is mandatory to follow this folder structure to save the workflow file:



**Figure 9.5 – Creating a GitHub workflow file**

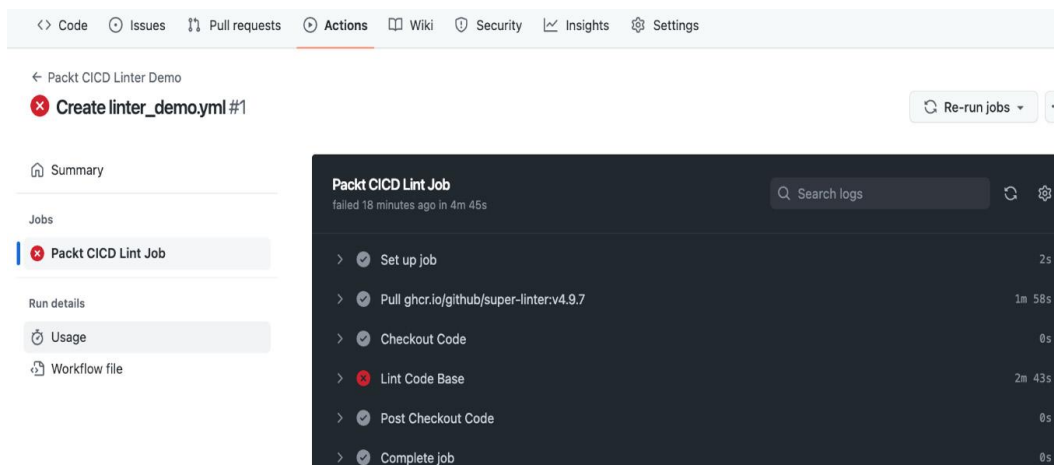


On navigating back to the home page of the repository, we notice a yellow status icon now, as shown in **Figure 9.6**. This signifies that the workflow is being run now and the code is being checked. This status icon turns green or red based on whether the checks pass or fail. This is particularly helpful when you are viewing a new repository and it aids to know that the repository is in a healthy state with all the tests passing. The results of the workflow can be viewed by clicking on the status icon or visiting the **Actions** tab:



**Figure 9.6 – Workflow status**

We can view the execution results of a specific job by following the link within the **Actions** tab. This provides a neat breakdown of the steps executed within the job and how long each one took. You could open each step to view the run logs. **Figure 9.7** shows the view for a failed job and its individual steps executed as part of the workflow:



**Figure 9.7 – Workflow results**

The **Actions** tab is where all the CI/CD information is shown within a GitHub repository. It shows a history of all our workflow jobs and their statuses, with options to look through each one further in detail. We can have as many workflows as we need within a single hub repository. For example, we could have one workflow that runs only Cypress tests and another to lint the entire code repository.

On fixing the suggestions from the linter and pushing the code to the repository, the CI job should automatically be triggered based on our setting in our workflow file.

The following is a sample snippet to invoke Cypress tests for reference. Placing these contents in a workflow file at the root of the project under the recommended directory structure triggers Cypress tests on every commit to the repository:

```
name: Packt Cypress Tests

on: [push]

jobs:
  cypress-run:
    name: Packt Cypress CI/CD Demo
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3
      - name: Cypress.io
        uses: cypress-io/github-action@v4.2.0
    env:
      DEFAULT_BRANCH: main
      GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```

GitHub has an extensive marketplace (<https://github.com/marketplace>) where you can grab pre-written workflows for different use cases. We can download and modify them to use in our repository. Readers are advised to take a look at these extensions to get an idea of the tremendous community surrounding CI/CD systems.

This brings us to the end of this chapter. In the next section, let us quickly summarize what we learned in this chapter and peek into our explorations in the final chapter of this book.

# Chapter 10

## Figures

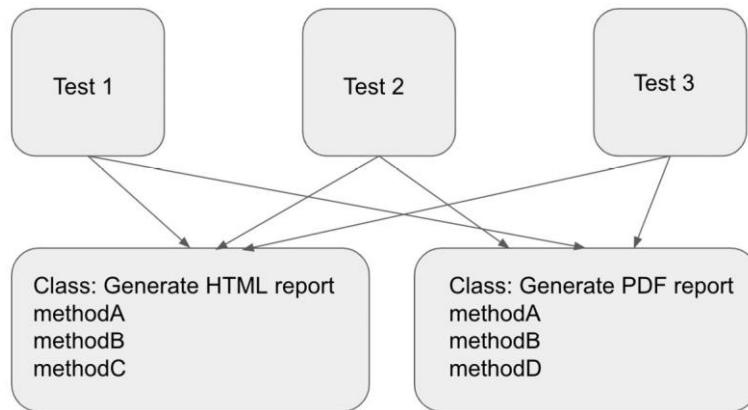


Figure 10.1 – Code duplication

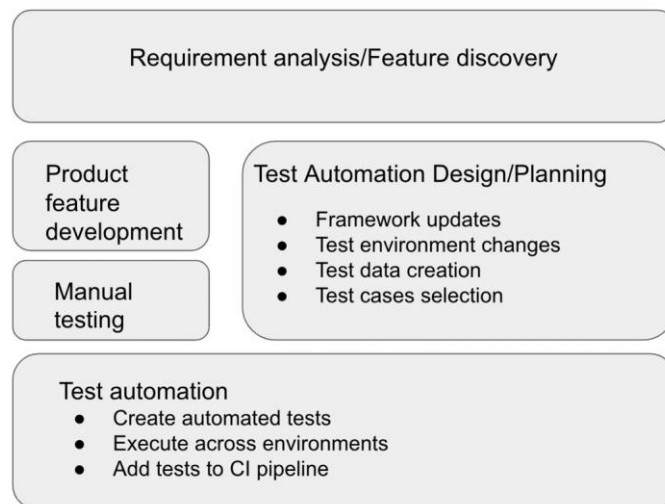


Figure 10.2 – Test automation tasks

## Table

Story points	Test type	The complexity of the task	Dependencies	The effort required in days
1	API integration	Very minor	Nothing	Less than 3 hours

Story points	Test type	The complexity of the task	Dependencies	The effort required in days
2	API integration	Simple	Some	Half a day
3	UI end-to-end	Medium	Some	Up to 2 days
5	UI end-to-end	Difficult	More than a few	3 to 5 days
Split into smaller tasks	API/UI	Very complex	Unknown	More than a week

**Table 10.1 – Sample test automation effort matrix**

## Code

### Code 10.1: Lengthy complex tests

```
describe("Visit packt home page, ", () => {
  beforeEach(() => {
    cy.visit("https://www.packtpub.com");
  });
  it("search, terms and contact pages", () => {
    const search_string = "quality";
    const result_string = "Filter Results ";
    cy.get('#__BVID__324').and("have.value", "");
    cy.get('#__BVID__324').type(`${search_string}`, { delay: 500 });
    cy.get('.form-inline > .btn-parent > .btn > .fa').click();
    cy.get(".filter-results").contains(result_string);
    cy.get(".reset-button", { timeout: 10000 })
      .should("be.disabled");
    cy.get("#packt-navbar").and("have.class", "navbar-logout");
    //test term-conditions page
    cy.visit("https://www.packtpub.com/terms-conditions");
    cy.get('.form-inline > .btn-parent > .btn > .fa').click();
  });
});
```

```
cy.get(".terms-button", { timeout: 10000 }).should("be.enabled");
//test contact us page
cy.visit("https://www.packtpub.com/contact");
cy.get('.form-inline > .btn-parent > .btn > .fa').click();
cy.get(".send-button", { timeout: 10000 }).should("be.disabled");
});
```

### Code 10.2: A case where no assertions are being used

```
function compute_product() {
...Test logic...
  if (product==10){
    console.log('product is 10');
  } else if (product==20){
    console.log('product is 20');
  }
  else {
    console.log('product is unknown');
  }
  return product;
}
```

### Code 10.3: Use of multiple assertions in a single test

```
cy.get('[data-testid="user-name"]').should('have.length', 7)
cy.get('[data-testid="bank_name"]').should('have.text', 'BOA Bank')
cy.get('[data-testid="form_checkbox"]')
  .should('be.enabled')
  .and('not.be.disabled')
```

### Code 10.4: The use of the same type of assertions for multiple UI elements

```
cy.get('#about').contains('About')
cy.get('.terms').contains('terms-conditions')
cy.get('#home').contains('Home')
```

## Code 10.5: Mishandling data in automation

- **Functional test data:** This drives the application logic and is seeded within the framework or comes from a test environment.
- **Dynamic test suite data:** This is data required by the test scripts for execution, such as secrets:

```
node test-script.js -secret='HAGSDH' -timeout=30000
```

- **Global data:** This is configuration data specific to particular environments, stored in **config** files and the CI system:

```
DEV_URL= //test-development.com  
STAGING_URL=https://test-staging.com  
AWS_KEY=test-aws-key
```

- **Framework level constants:** These are constant values required by the tests and stored within the framework in a non-extendable base class:

```
const swift_code = 111222333,  
      routing_number = 897654321;  
class BankConstants {  
  static get swift_code () {  
    return swift_code;  
  }  
  static get routing_number () {  
    return routing_number;  
  }  
}
```

# Appendix A: Mocking API Calls

## Figures

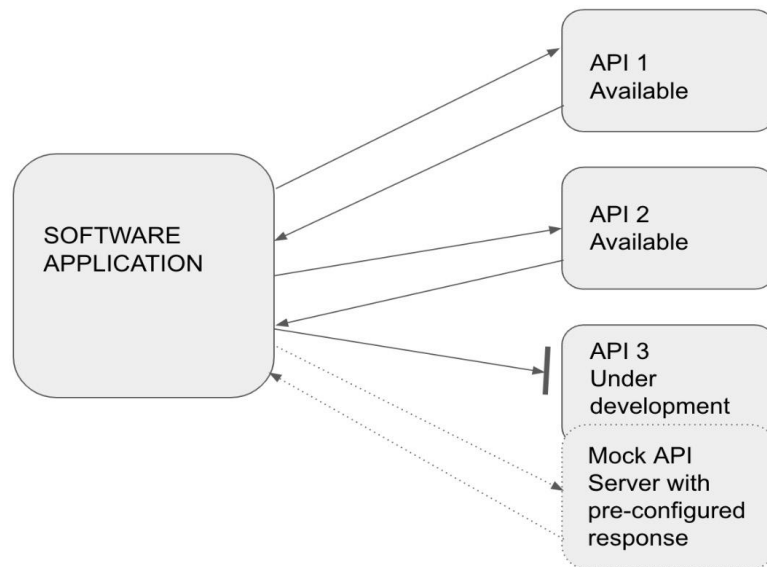


Figure A.1 – API mocking

## Hands-On section

### Mocking API calls using Postman

Postman provides an interactive GUI to set up mocks for API calls. Let us now review how to set one up step by step:

1. Create or use an existing Postman collection as seen earlier in [Chapter 7, Test Automation for APIs](#).
2. Set up a request as shown in [Figure A.2](#). There are two scenarios where we end up mocking an API request:
3. The first is when we have a sample response from the API call, but subsequent requests cannot be made to the API. We could save the response as an example in Postman and use it for mocking.
4. The other one is when the API call does not exist or we do not have a sample. In this case, we will have to build the response from the scratch. We will be simulating this scenario in our example:



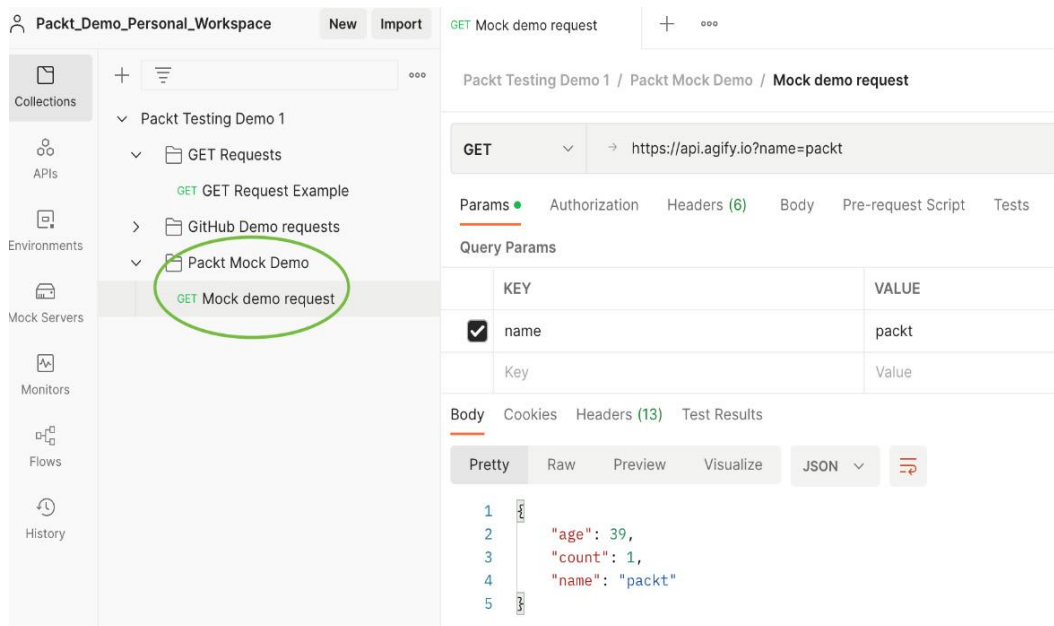


Figure A.2 – Postman request

- The next step is to create the mock server for our request as shown in **Figure A.3**. This can be done by using the **Mock Servers** option on the left-hand pane and selecting the collection to associate it with. Once this is done, a mock server URL is generated, to which a request can be made. Postman also provides an option to make the mock server private by generating an API key to authorize requests.

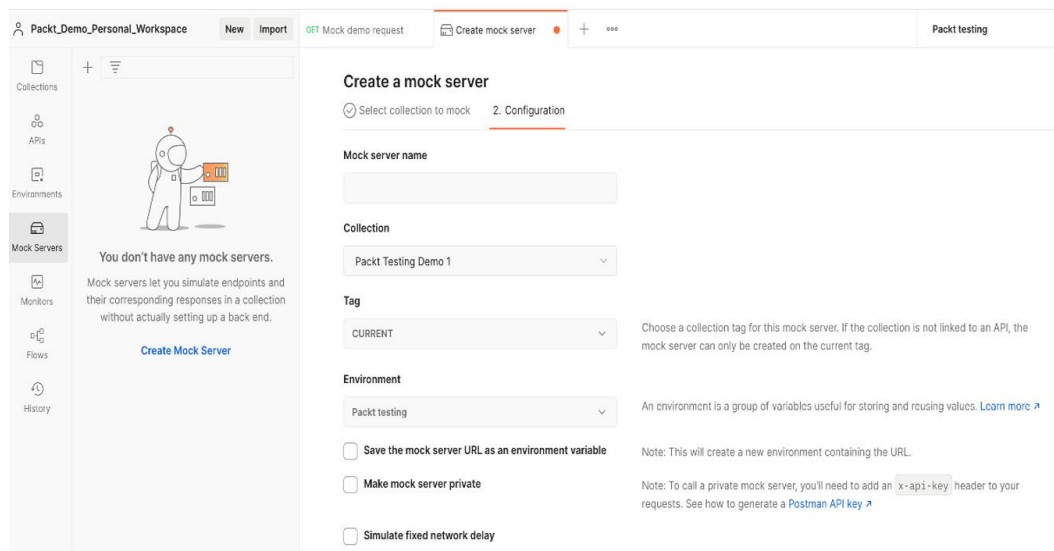


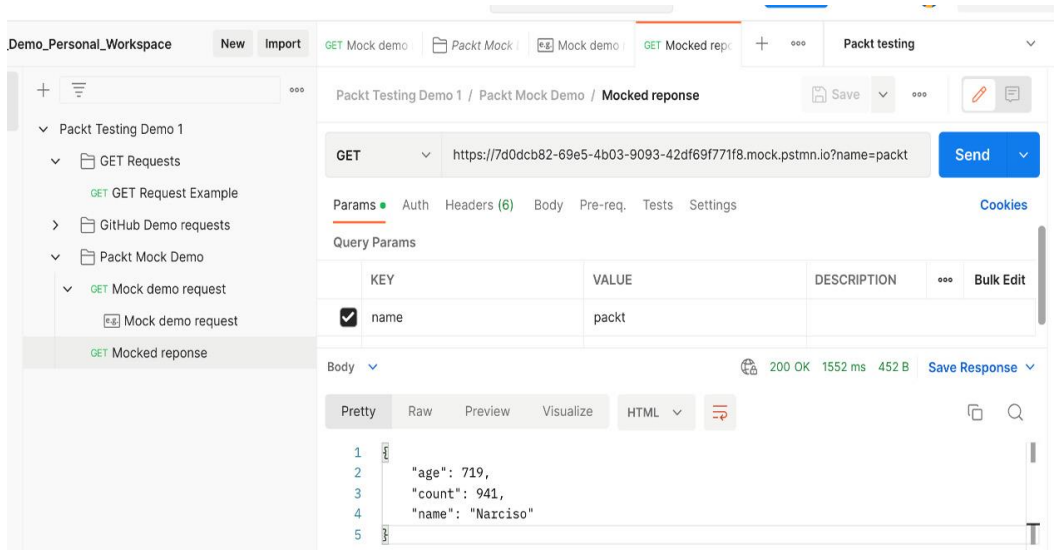
Figure A.3 – Creating a mock server

- The next step is to add an example to the request using the **Options** menu. We will be using the dynamic variables in Postman to generate random values in our response. This can be a static response as well. We will be using the following response in our example:

```
{  "age":  {{${randomInt}}},
```

```
"count": {{$randomInt}},  
"name": "{{$randomLastName}}"  
}
```

7. The final step is to create a new request and use the mock server URL to make the API call. **Figure A.4** shows this in action.



**Figure A.4 – Mocked response**

By using a combination of environment and dynamic variables, it is possible to simulate almost any kind of API response in Postman. Let us now look at a few important considerations when employing mocks.