EXPERT INSIGHT

API Testing and Development with Postman

API creation, testing, debugging, and management made easy

Second Edition

Dave Westerveld

packt

API Testing and Development with Postman

Second Edition

API creation, testing, debugging, and management made easy

Dave Westerveld

‹packt›

API Testing and Development with Postman
Second Edition

Senior Publishing Product Manager: Denim Pinto
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Meenakshi Vijay
Senior Development Editor: Elliot Dallow
Copy Editor: Safis Editing
Technical Editor: Anjitha Murali
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Presentation Designer: Rajesh Shirsath
Developer Relations Marketing Executive: Vipanshu Parashar

First published: April 2020
Second edition: June 2024

Production reference: 1130624

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN: 978-1-80461-790-8

www.packt.com

Contributors

About the author

Dave Westerveld is passionate about sharing his knowledge and expertise to help testers stay relevant in the ever-changing world of software. He has helped many software testers through his several popular video courses and has shared his expertise at conferences, online talks, and podcasts. He has also worked in the software industry for many years and in many different roles. He has worked, among other roles, as an exploratory tester, a test automation developer, and an API integrations developer.

I would like to thank my wife, Charlene, for her unwavering support of me in everything that I do. You are the rock that makes it possible for me to show up in the world in the ways that I do. You are always excited for me to share my expertise with the world. Thank you for your love and support.

About the reviewers

Christina Thalayasingam is a software engineering people manager at Northwestern Mutual with 9 years of industry experience. She holds a bachelor's degree in computer science engineering and has worked for companies such as Sysco and Dassault Systèmes®. She has a passion for testing, and beyond her daily work, she has been an active speaker at conferences and meetups such as Star East, Women in Tech Global Conference, and TestCon Europe.

Neil McCormick is a quality assurance manager who has worked in the QA space since 1998. He began API testing in the early 2000s. He helped pioneer the testing methodologies of his company, AAA, and in 2020 was promoted to his current position. He believes testers should never stop learning, exploring, and most importantly growing – both professionally and personally.

I am grateful to the author, Dave Westerveld, and the Packt Publishing team, my bosses Jerry and Kristi, my team, and too many people in the realm of family and friends to call out.

To my wife, Nandi, thank you for the weekends and nights you sacrificed for me to be able to partake in this opportunity. You smiling at my boyish excitement when I really had fun reviewing a chapter and listening to me speak endless technobabble to you has been greatly appreciated!

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://discord.com/invite/nEN6EBYPq9`

Table of Contents

Chapter 9: Running API Tests in CI with Newman 175

Chapter 10: Monitoring APIs with Postman 199

Chapter 14: API Security Testing 275

Chapter 15: Performance Testing an API 293

Preface

In a world of fast food, fast fashion, and fast-to-market strategies, does quality even matter? The pressures of the world we live in seem to push us towards taking shortcuts, even when it comes to producing high-quality software. I am doing my part to push back against that world. I think quality matters. We have enough easily broken junk in our lives. It's time for more quality.

This book is one small stake that I've put in the ground in an attempt to help the world see more high-quality software. I hope that whether you are a professional tester, or a developer looking to learn more about testing, you will be able to join me in my attempt to improve the world through quality software applications.

APIs are becoming the backbone of the internet. They help companies to communicate with each other externally and also provide the communication infrastructure for many internal pieces of modern software systems. A marriage is held together by good communication, and so it is with the internet too. Good communication between different services is important to well functioning applications. For that reason, API testing matters for producing good-quality software.

On the surface, this book is primarily about the API testing tool Postman, but I have also tried to weave in examples and teachings that will help you to to use that tool in a way that will have a real impact on quality. If you work through this book, you will gain an in-depth grasp of how Postman works, and you will also have a solid foundation in how to think about API testing in general. I want you to have more than just the ability to manipulate Postman to do what you want it to. I also want you to be able to know when and how to use it so that you can be an effective part of creating high-quality APIs.

Who this book is for

The first person I write for is myself. A lot of the ideas I talk about in this book are things I was learning myself a few years ago. In fact, Postman comes out with new capabilities so often that some of what is in this second edition are new things I learned while writing the book.

I'm always growing and learning and I love sharing what I've learned with others to help them along on their journey.

Getting started with API testing can be overwhelming, It's a huge topic and it can be intimidating to get started, so I wrote this book primarily for those software testers and developers who find themselves needing to test an API and not knowing where to start. Throughout this book I try not to assume much in-depth programming experience although an understanding of some of the basics of programming will certainly help with some sections of the book.

If you are working as a software tester and you are interested in expanding your skills into API testing, this book is certainly for you. If you are a developer who is looking to enhance your skills around testing and quality, congratulations, you are setting yourself up for a successful career! Developers that know and understand how to produce good quality software will always be in high demand. Whatever, your background, you may be able to skim through some parts of this book, but if you spend some time with it, you will find that you come away knowing how to use Postman and how to design and write good API tests.

What this book covers

Chapter 1, *API Terminology and Types*, gets you started with some of the basic API terminology and introduces you to the different types of APIs.

Chapter 2, *API Documentation and Design*, covers the design principles that apply to creating and testing APIs, and both *how* and *why* to create useful documentation.

Chapter 3, *Open API and API Specifications*, explains what API specifications are and how to get started with using them in Postman.

Chapter 4, *Considerations for Good API Test Automation*, teaches you how to create and execute valuable and long-lasting API tests in Postman.

Chapter 5, *Understanding Authorization Options*, walks through how to use many of the API authorization methods available in Postman.

Chapter 6, *Creating Test Validation Scripts*, explains how to create and use test scripts in Postman.

Chapter 7, *Data-Driven Testing*, discusses what data-driven testing is and how to use it to create scalable tests in Postman.

Chapter 8, *Workflow Testing*, explains what workflow tests are and how to create flows in Postman.

Chapter 9, *Running API Tests in CI with Newman*, shows how to run Postman API tests at the command line with the Newman runner.

Chapter 10, *Monitoring APIs with Postman*, explains how to monitor product usage of APIs with Postman monitoring.

Chapter 11, *Testing an Existing API*, works through a hands-on example that shows what kind of tests to create when testing an existing API.

Chapter 12, *Creating and Using Mock Servers in Postman*, explains what mock servers are and how to set up and use them in Postman.

Chapter 13, *Using Contract Testing to Verify an API*, discusses what contract testing is and shows how to create and use contract testing in Postman.

Chapter 14, *API Security Testing*, gives a brief introduction to security testing and gives an example of setting up fuzz testing in Postman.

Chapter 15, *Performance Testing an API*, explains the different types of performance testing and walks through some of the features in Postman that can be used to assess API performance.

To get the most out of this book

This book is intended to equip you with skills that you can use immediately in your work as a tester or developer. If you want to get the most value that you can out of this book, put the things that you learn into practice as soon as you possibly can. Work through all the exercises in this book, but also try to take the ideas that you learn and put them into practice in the "real world" as well.

This book does not assume a lot of prior knowledge of APIs, or even development and testing principles. As long as you have a basic grasp of web technology and what software development looks like in general, you should be able to follow along with this book and pick up everything that you need. Some of the test scripts in Postman use Javascript, but you don't need to know much about how that works in order to follow along, although a basic understanding would be helpful. There are examples and challenges throughout the book. They are an important part of the book and in order to get the most out of it, you should take the time to work through them.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/gbp/9781804617908`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "The `/product` endpoint gives information about the products accessed by this API."

A block of code is set as follows:

```
openapi: 3.0.1
info:
    title: ToDo List API
    description: Manages ToDo list Tasks
    version: "1.0"
servers:
    -url: https://localhost:5000/todolist/api
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
/carts:
    post:
    get:
        queryParameter:
        username:
    /{cartId}:
        get:
        put:
```

Any command-line input or output is written as follows:

```
npm install -g newman
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Click on the **Import** button and choose the **OpenAPI** option."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share your thoughts

Once you've read *API Testing and Development with Postman*, *Second Edition,* we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don’t worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don’t stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

`https://packt.link/free-ebook/9781804617908`

2. Submit your proof of purchase.
3. That’s it! We’ll send your free PDF and other benefits to your email directly.

1

API Terminology and Types

Learning something new can feel a little like falling over the side of a ship. Everything is moving and you can barely keep your head above water. You are just starting to feel like you understand how something works and then a new piece of knowledge comes out of nowhere, and your whole world feels topsy-turvy again. Having something solid to hold on to gives you the chance to look around and figure out where you are going. This can make all the difference in the world when learning something new.

In this chapter, I want to give you that solid foundation. As with almost any specialty, API testing and development has its own terminology. There are many terms that have specialized meanings when you are working with APIs. I will be using some of those terms throughout this book, and I want to make sure that you and I share a common understanding of what they mean.

As much as possible, I will use standard definitions. Some terms, however, do not have clearly agreed-on definitions, and for those terms, I'll share how I intend to use and talk about them in this book. Be aware that as you read or listen to things on the internet (or even just interact with teammates), you may come across others who use the terms in slightly different ways.

This book is not a dictionary, so I don't intend to just write down a list of terms and their definitions. That would be boring and probably not all that instructive. Instead, I'll spend a bit of time on the theory of what an API is and how you test it. I will weave in explanations and definitions of important terminology throughout the text.

This chapter will cover the following main topics:

- What is an API?
- Types of API calls

- Installing Postman
- The structure of an API request
- Considerations for API testing
- Different types of APIs

By the end of this chapter, you will be able to use Postman to make API requests and have a good grasp of basic API terminology. You will also have the opportunity to work through an exercise that will help you cement what you are learning, allowing you to start to use these skills in your day-to-day work.

What is an API?

A 1969 NASA publication entitled *Computer Program Abstracts* contains a summary of a real-time display control program sold by IBM (only $310! Plus $36 if you want the documentation). The advertisement says that this program was designed as an operator-application programming interface – in other words, an API.

Application Programming Interfaces (APIs) have been around for about as long as computer code has. Conceptually, it is just a way for two different pieces of code (or a human and some code) to interface with each other. A class that provides certain public methods that other code can call has an API. A script that accepts certain kinds of input has an API. A driver on your computer that requires programs to call it in a certain way has an API.

However, as the internet grew, the term *API* narrowed in focus. Almost always now, when someone talks about an API, they are talking about a web API. That is the context I will use in this book. A web API takes the concept of an interface between two things and applies it to the client/server relationship that the internet is built on. In a web API, a client is on one side of the interface and sends requests, while a server (or servers) is on the other side of the interface and responds to the request.

Over time, the internet has changed and evolved, and web APIs have changed and evolved along with it. Many early web APIs were built for corporate use cases, with strict rules in place as to how the two sides of the interface could interact with each other. A type of API called the **Simple Object Access Protocol (SOAP)** was developed for this purpose. However, in the early 2000s, the web started to shift toward becoming a more consumer-based place. Some of the e-commerce sites, such as eBay and Amazon, started to publish APIs that were more public and flexible. This was followed by many social media sites, including Twitter, Facebook, and others. Many of these APIs were built using a pattern called **Representational State Transfer (REST)**.

This approach is more flexible than SOAP and is built directly on the underlying protocols of the internet.

The internet continued to change though, and as mobile applications and sites grew in popularity, so did the importance of APIs. Some companies faced challenges with the amount of data they wanted to transfer on mobile devices, so Facebook created yet another type of API called GraphQL. This type of API defines a query language that helps to reduce the amount of data that gets transferred, while also introducing a slightly more rigid structure to the API. Each of these different API types works well in some scenarios, and I will explain more about what they are later in the chapter. However, before I get into the details of each of these types of APIs, it is important to first understand some of the concepts that underpin all web API calls.

Types of API calls

Some calls to APIs can change things on the server, while others return data without changing anything. The terms **safe** and **idempotent** are used to describe the different ways that API calls can affect data. These terms might sound a bit intimidating, so in order to better understand them, let's look at an illustration that uses something we can all understand: LEGO pieces.

Imagine that there is a table with a couple of LEGO pieces on it and I'm sitting by the table. I represent an API, while the table represents a server, and the LEGO pieces represent objects. If you come along and want to interact with the LEGO, you must do so through me. In this illustration, the LEGO pieces represent objects on a server, I represent an API, and you represent a client. In picture form, it looks something like this:

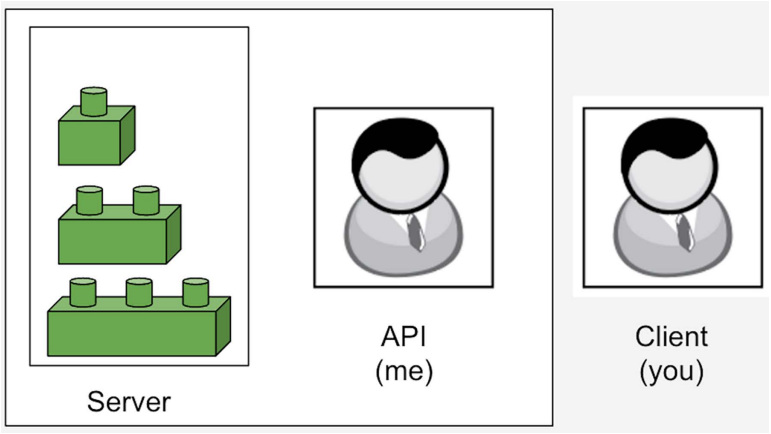

Figure 1.1: Representation of a server and a client connected by an API

You are going to be the client in this imaginary relationship. This means you can ask me to do things with the LEGO. You ask me to tell you what size the top LEGO piece is. I reply that it has a size of one. This is an example of an API request and response that is **safe**. A safe request is one that does not change anything on the server. By asking me for information about what is going on in the server, you have not changed anything on the server itself.

There are other kinds of API calls though. Imagine that you gave me a brick with a size of two and asked me to replace the top brick on the stack with the one you gave me. I do that, and in doing so I have changed the server state. The brick stack is now made up of a brick with a size of three, followed by two bricks with a size of two, as shown in the following diagram:

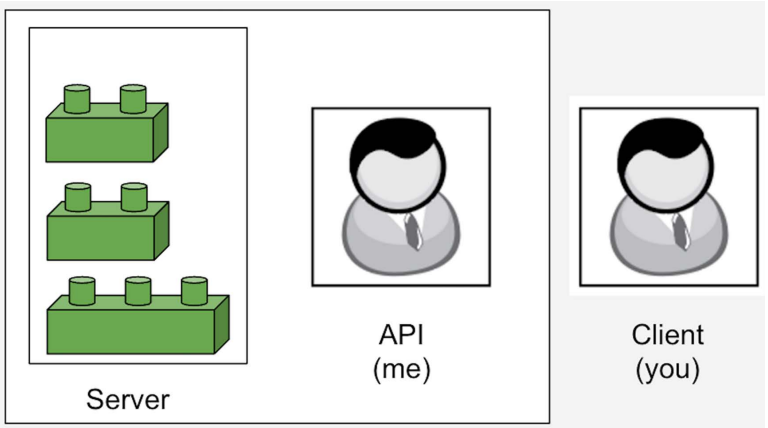

Figure 1.2: New server state

Since the server has changed, this is not a safe request. However, if you give me another brick with a size of two and ask me to once again replace the top brick with the brick you just gave me, nothing will change on the server. The stack will still be made up of a brick with a size of three followed by two bricks with a size of two. This is an example of an **idempotent** call. API calls that return the same result no matter how many times you call them are known as idempotent.

Let's imagine one more type of call. In this case, you give me a brick and ask me to add it to the top of the stack. I do that and now we have a stack of four bricks. This is clearly not a safe call, since the server has changed, but is it idempotent?

The answer is no, but take a second to think about it, and make sure you understand why this call would not be idempotent.

If you are struggling to see why it is not idempotent, think of what happens if you repeat the same request. You give me another brick and you ask me to add it to the top of the stack. If I do that a second time, is the brick stack still the same as it was after the first time you added it? No, of course not! It now has five bricks and every additional brick you give to me to add to the stack will change it. An idempotent call is one that only changes things the first time you execute it and does not make any changes on subsequent calls. Since this call changes something every time, it is not idempotent.

Safety and idempotency are important concepts to grasp, especially when it comes to testing APIs. For example, if you are testing calls that are safe, you can run tests in parallel without needing to worry about them interfering with each other. But if you are testing calls that are not safe or idempotent, you may need to be a little more careful about what kinds of tests you run and when you run them.

There are a few more important terms that we need to learn about, and we also need to dig into the structure of API requests. However, it will be a lot easier to do that if we have something concrete to look at, so at this point, let's take a brief pause to install Postman and send our first request.

Installing Postman

Postman can be run on the web or as a desktop application. The functionality and design are similar between the web and desktop applications, but in this book, I will mostly use the desktop application, so I would recommend you install it too. The app is available for Windows, Mac, and Linux, and installing it is the same as pretty much any other program you've installed. However, I would highly recommend creating a Postman account if you don't already have one. Creating an account is totally free, and it makes it a lot easier to manage and share your work. The free account of Postman is very generous in what functionality it enables. Postman does have some enterprise or advanced-level features that require a paid account, but all the examples in this book will work with the free features of Postman. However, if you don't have an account at all, it will be difficult to follow along with some of the examples, so I would strongly recommend that you register for one:

1. Go to `https://postman.com`.
2. Choose the **Sign Up for Free** option.
3. Create an account, and when asked how you want to use Postman, choose the **Download Desktop App**, and then install it as you would any program on your computer.

If you already have a Postman account but don't yet have the desktop app, you can skip steps two and three and just download it directly from the Postman home page.

I will be using the Mac version of Postman, but other than the occasional screenshot looking a bit different, everything should be the same regardless of which platform you are running Postman on.

I will primarily use the desktop application in this book, but there are times when the web application is helpful. I would recommend that you set it up as well. During the sign-up process, you can download the Desktop Agent. This agent allows the web version of Postman to get around some of the cross-origin restrictions that web browsers put on web requests. These restrictions are very important for security on the web, but when testing, we often need to be able to work around them, and this agent will allow you to do that. Once again, you can download and install this as you would any other program.

Starting Postman

Once you have Postman installed, open the application. The first time you open Postman, it will ask you to sign in. Once you've signed in, you will see the main screen with a bunch of different options for things that you can do with Postman. Don't worry about all these options right now. We will cover all of them (and more) as we go through this book. For now, just note that they are there and that you can do a lot of cool stuff with Postman. Maybe even get a little bit excited about how much you are going to learn as you go through this book!

Setting up a request in Postman

It's time to set up an API call so that we can dissect it and see how it all works:

1. To set up a new request, you can click on the + near the top of the application to add a new tab.

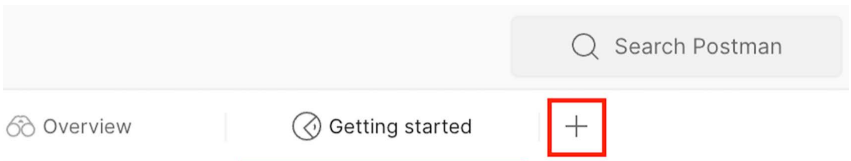

Figure 1.3: Create a new request tab

   For this first example, I will use the Postman Echo API. This is an API Postman provides that can be used to do some simple testing.

2. In the field that says **Enter URL or paste text**, type in the following URL: `https://postman-echo.com/get`.
3. Click on the **Send** button to the right of the URL field, and you should see a response from the server. Don't worry about the actual data of the response; for now, just congratulate yourself for having sent your first request with Postman!

Saving a request

Now that you have created a request, let's look at how to save requests in Postman. Postman requires that requests be saved into something called Collections. Collections are a way to collect multiple requests that belong together:

1. Click on the **Save** button above and to the right of the request URL.

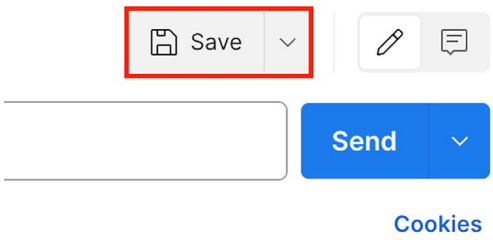

Figure 1.4: Save a request

2. In the pop-up dialog, give the request a name. Call it `Postman Echo GET`.
3. Click on the **New Collection** option at the bottom of the popup.
4. Name the collection something like `Postman Echo Requests` and click **Create**.
5. Click the **Save** button on the dialog. This will create the collection for you and save the request into that collection.

Now that you have saved a request, let's start digging into the basic structure of an API request.

The structure of an API request

The request tab in Postman provides a lot of information about the various pieces that make up an API request. Each of these pieces plays an important part in sending and receiving data with an API, so I will walk you through each one in turn. Some parts of an API request are optional, depending on what kind of request it is and what you are trying to do with it, but there are three pieces that are required for every API request. Every API request needs an endpoint, headers, and an action; let's look at those next.

API endpoints

Every web-based API request must specify an **endpoint**. In the **Postman requests** tab, you are prompted to enter the request URL. Postman asks you to enter a URL because an API endpoint is just a URL. We use the term *URL* so much that we can sometimes forget what it stands for. URL is an acronym for **Uniform Resource Locator**. The endpoint of an API call specifies the resource, or the "R" of the URL. In other words, an API endpoint is a uniform locator for a particular resource that you want to interact with on the server. URLs help you to locate resources on a server, so they are used as the endpoints in an API call.

In order to understand this, let's set up a concrete example:

1. Create a second collection by clicking on the **New** button above the navigation panel and choosing **Collection** on the popup.
2. Name the collection something like `GitHub API Requests`.
3. If you expand the collection, you will see a prompt telling you that the collection is empty, and a link reading **Add a request** is provided. Click on that link.

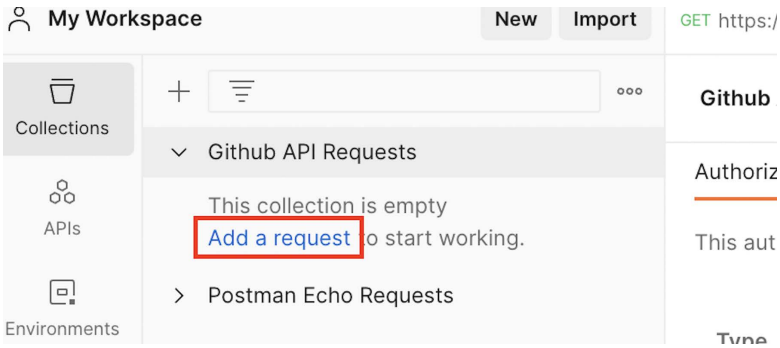

Figure 1.5: Add a request to a collection

4. Name the request `Get Repos` and fill in the request URL field with the following URL: `https://api.github.com/users/djwester/repos`.
5. Click **Send** to see the response.

   This endpoint will give you information about my public repositories on GitHub. If you have a GitHub account of your own, you can enter your username in the part of the URL where it says `djwester` and get back data for your own repositories.

You will often see an API endpoint specified without the base part of this API. So, for example, if you look at the GitHub API documentation, it will report the endpoint for this as `/users/:username/repos`. All the GitHub API calls start with the same base URL (in other words, `https://api.github.com`), so this part of the endpoint is often left out when talking about an endpoint. If you see API endpoints listed that start with a `/` instead of with `http` or `www`, just remember that you need to go and find the base API URL for the endpoint in order to call it.

API actions

Every API call needs to specify a resource that we work with. This resource is the endpoint, but there is a second thing that every API call needs. An API needs to do something with the specified resource. We specify what we want an API to do with API **actions**. These actions are sometimes called **verbs**, and they tell the API call what we expect it to do with the resource that we have given it. For some resources, only certain actions are valid, while for others, there can be multiple different valid API actions.

In Postman, you can select the desired action using the drop-down menu beside the textbox where you entered the URL. By default, Postman sets the action to **GET**, but if you click on the dropdown, you can see that there are many other actions available for API calls. Some of these actions are specialized for particular applications, so you won't run into them very often. In this book, I will only use **GET**, **POST**, **PUT**, and **DELETE**. Many APIs also use **PATCH**, **OPTIONS**, and **HEAD**, but using these is very similar to using the four that I will use, so you will be able to easily pick up on how to use them if you run into them. The rest of the actions in this list are not often used, and you will probably not encounter them much in the applications that you test and create.

The four actions (**GET**, **POST**, **PUT**, and **DELETE**) are sometimes summarized with the acronym **CRUD**. This stands for **C**reate, **R**ead, **U**pdate, and **D**elete. In an API, the **POST** action is used to create new objects, the **GET** action is used to read information about objects, the **PUT** action is used to modify (or update) existing objects, and (surprise, surprise) the **DELETE** action is used to delete objects. In practice, having an API that supports all aspects of CRUD gives you the flexibility to do almost anything you might need to, which is why these four actions are the most common ones you will see.

API actions and endpoints are required for all web APIs, but there are several other important pieces to API requests that we will consider.

API parameters

API parameters are used to create structure and order in an API. They organize similar things together. For example, in the API call that we looked at earlier, we get the repositories for a particular user in GitHub. There are many users in GitHub, and we can use the same API endpoint to get the repository list for any of them by merely changing the username in the endpoint. The part of the endpoint that accepts different usernames is a **parameter**.

Request parameters

The username parameter in the GitHub repositories API endpoint is known as a **request parameter**. You can think of a request parameter as a replacement string in the API endpoint. They are very common in web APIs. You will see them represented in different ways in the documentation of different APIs. For example, the GitHub documentation uses a colon in front of the request parameter to indicate that it is a request parameter and not just another part of the endpoint. You will see endpoints specified like this in the GitHub documentation:

```
  /users/:username/repos.
```

In other APIs, you will see request parameters enclosed in curly braces instead. In that case, the endpoint would look like this:

```
/users/{{username}}/repos.
```

Whatever the format used, the point of request parameters is to get information about different objects that are all the same type. We have already seen how you can do that with this endpoint by replacing my username with your username (or any other GitHub user's name).

Query parameters

There is another kind of parameter that you can have in an API endpoint. This kind of parameter is known as a **query parameter**, and it is a little bit trickier to deal with. A query parameter often acts like a kind of filter or additional action that you can apply to an endpoint. It is represented by a question mark in the API endpoint and is specified with a key, which is the item you are querying for, and a value, which is what you want the query to return.

That's all very abstract, so let's look at it with the GitHub request we sent earlier. This endpoint supports a couple of different query parameters. One of them is the `type` parameter. In order to add parameters to an API endpoint in Postman, make sure you have the **Params** tab selected, and then enter the name of the query parameter into the **Key** field and the value into the **Value** field. In this case, we will use the `type` parameter, so enter that word into the **Key** field.

For this endpoint, the type parameter allows us to filter based on whether you are the owner of a repository or just a member. By default, the endpoint will return only those repositories that I am the owner of, but if I want to see all the repositories that I am a member of, I can put `member` in the **Value** field for this. At this point, the request should look something like this:

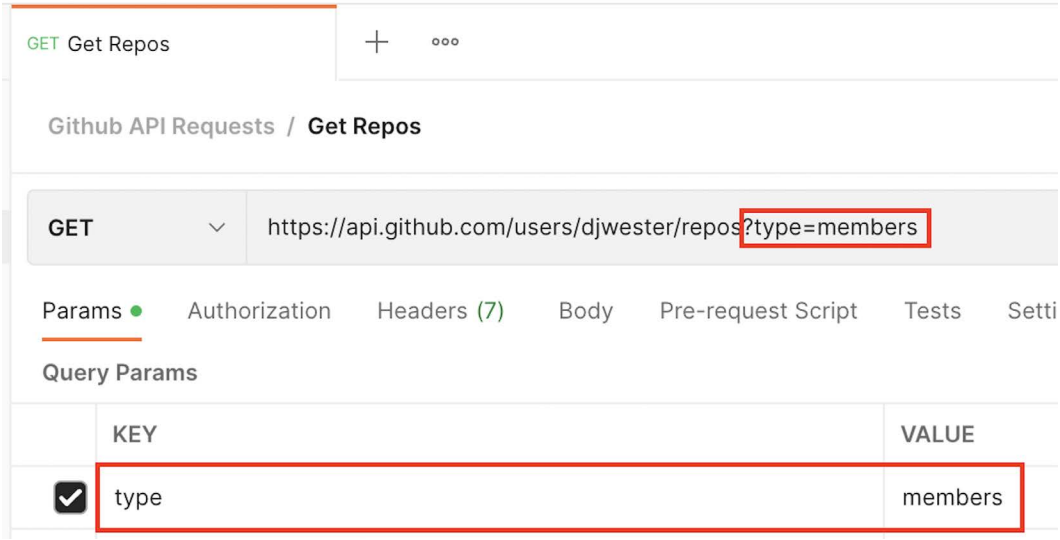

Figure 1.6: Query parameter type in an API call

You can see that when you specify query parameters in the **Params** tab, Postman automatically adds them to the request URL. If I send this request, I get back all the repositories that I am a member of, as opposed to just the ones that I own. Parameters are a powerful API paradigm, but there are still a few more fundamental pieces of the API structure that I haven't talked about yet. The next thing we will look at are API headers.

API headers

Every API request needs to include some **headers**. Headers include some of the background information that is often not that important to human users, but they help the server have some information about the client that is sending the request. Sometimes, we will need to modify or add certain headers in order to get an API to do what we want, but often, we can just let the tool that we are using send the default headers that it needs to send without worrying about it.

In Postman, you can see what headers will be sent with your request by using the **Headers** tab. When you first go to that tab, it might look like there aren't any headers specified. In that case, you probably need to toggle the view to show the auto-generated headers:

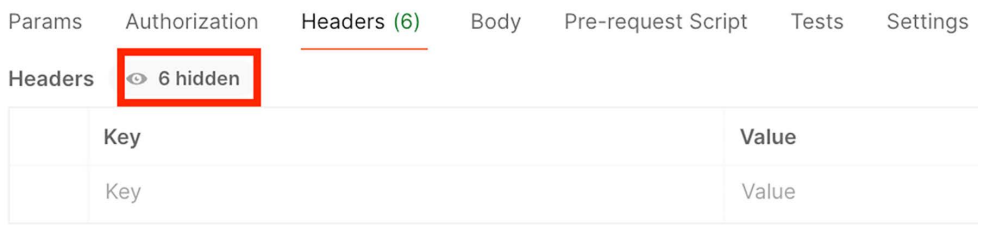

Figure 1.7: Headers hidden

Once the headers are revealed, you should see a list that looks something like this:

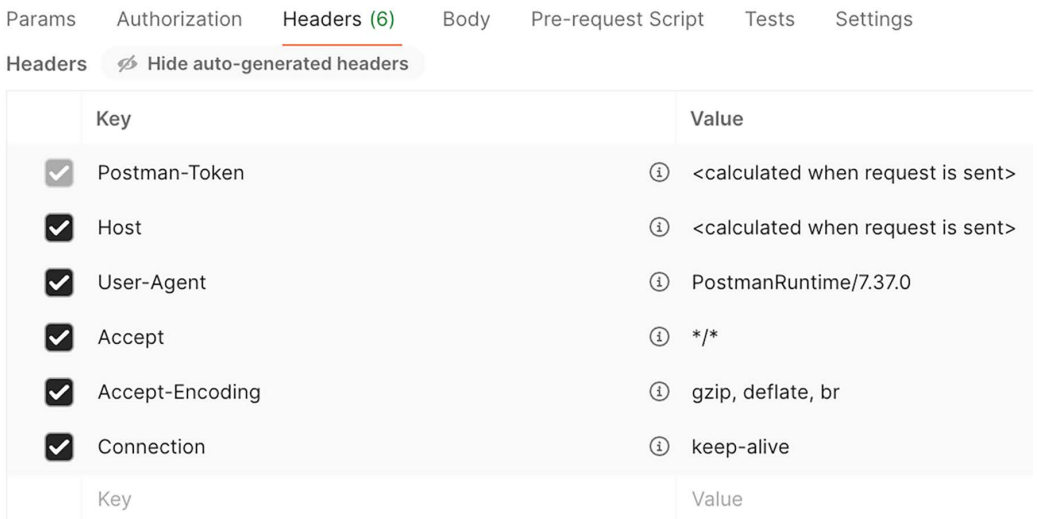

Figure 1.8: List of headers

You can modify the auto-generated headers and add additional ones here as needed. I will go into more detail on how headers work and how to use them in future chapters, so for now, you don't need to worry about them too much. The point of mentioning them here is just to make sure you know the terminology. Let's turn our attention instead to the body of an API request.

API body

If you want to create or modify resources with an API, you will need to give the server some information about what kind of properties you want the resource to have. This kind of information is usually specified in the **body** of a request.

The request body can take on many forms. If you click on the **Body** tab in the Postman request, you can see some of the different kinds of data that you can send. You can send form-data, encoded form data, raw data, binary data, and even GraphQL data. As you can imagine, there are a lot of details that go into sending data in the body of a request. For example, if you were to try and make a new repository using the GitHub API, you might enter a body that looks something like this:

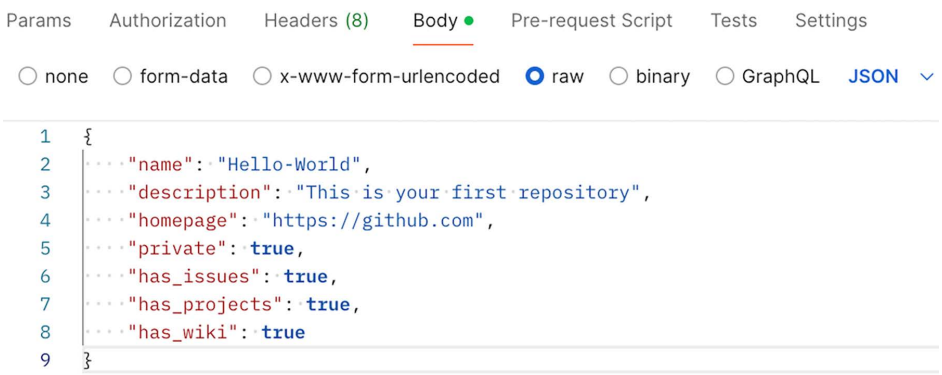

Figure 1.9: Body of a request

Most of the time, `GET` requests do not require you to specify a body. Other types of requests, such as `POST` and `PUT`, which do require you to specify a body, often require some form of authorization, since they allow you to modify data. We will learn more about authorization in *Chapter 5*, *Understanding Authorization Options*. Once you can authorize requests, there will be a lot more examples of the kinds of things you might want to specify in the body of an API request.

API response

So far, we have spent a lot of time talking about the various pieces that make up an API request, but there is one very important thing that we have been kind of ignoring. An API is a two-way street. It sends data to the server in the request, but then the server processes that request and sends back a response.

The default view that Postman uses displays the response at the bottom of the request page. You can also modify the view to see the request and the response in side-by-side panels. You can change to this view if you so wish by clicking on the **Two pane view** icon at the bottom of the application, as shown in the following screenshot:

Figure 1.10: Switching views

There are a few different aspects to the response. The most obvious one is the body of the response. This is usually where most of the information that you are looking for will be included. In the GitHub repositories requests that you have made, the lists of repositories will show up in the body of the response, and Postman will display them in that tab.

An API response can also include a few other things, such as cookies and headers. These kinds of things can be very important hints as to what is going on when testing or creating APIs, and I will talk about them more as we go through the book.

We have covered a lot of ground when it comes to how API requests work. We have seen how an API request can consist of a lot of different pieces. These simple pieces all come together to create a powerful tool. You now have a grasp of the basic pieces that make up an API call and how to use them in Postman. It's almost time to talk about how to use this to test APIs, but before we get into that, I want to pause for a minute so that you can put into practice all this theory that I've just gone over.

Learning by doing – making API calls

Books are a great way to grow and learn. You are reading this book, so I don't think I need to convince you of that! However, reading a book (or even three or four books) on a topic does not mean you understand that topic. There is theory and then there is putting it into practice. These are two very different things, and if all you do is read about a topic, you will feel like you know that topic, but that will be more of a feeling than a reality.

If you want that to be a reality and you don't just want this book to be another piece of theoretical knowledge bouncing around inside your head, you need to put into practice the things that you are learning.

Hands-on exercises might feel like they are slowing down your reading, but working through them will actually help you learn faster and get more out of this book. But enough talking about it. Let's get to some exercises!

In order to help you with getting some hands-on experience, I have created a simple API application that you can interact with. Keep in mind that this application is just a "toy" application, meaning that it might do some things more simply than a "real" API. However, I hope it will be helpful for you as you are learning the topics in this book.

Setting up the test application

In order to use this application, you will need to do a bit of setup. There are two options to do this. You can run a version of it on GitPod, or you can run it on your local machine. I would recommend using GitPod, as that approach should have all the dependencies set up for you automatically. You can do that with the following steps:

1. First of all, you will need a GitHub account. If you don't yet have one, you can sign up at `https://github.com`.
2. Once you have that account, navigate to `https://gitpod.io/#https://github.com/djwester/todo-list-testing` in your browser.
3. Click to **Continue with GitHub**.
4. You can accept the default workspace settings and continue.
5. Once it has finished loading, go to the terminal, type in the command `make run-dev`, and hit *Enter*.
6. The service should start, and you should see a toast message with a few options. Click on the **Make Public** option.
7. Click on the **Ports** tab, and you can see the public URL of the test site available to copy.

Note that GitPod will shut this site down after a few minutes of inactivity, so if you haven't been using it for a while and you get an error when making API calls, you may have to come back and repeat these steps to restart the site. Also, every time you restart the site, it will have a slightly different URL, so don't forget to update the URL of any API calls pointing at this site.

If you want to instead run the site locally on your own computer, you will need to make sure you have a few pre-requisites in place. First of all, you will need to have Python version 3.11 installed:

1. Go to `https://www.python.org/downloads/`.
2. You will want to download Python 3.11, so click on the link to your platform below the **Download Python** button.

3. On the resulting page, find version 3.11 (the subversion doesn't matter) and download and install it.

You will also need to install the poetry package manager. You can do this by running the following command in the command prompt:

```
curl -sSL https://install.python-poetry.org | python3 -
```

Once you have those pre-requisites, you will need to download the site from GitHub. Make sure you have followed the instructions to install Git from `https://github.com/git-guides/install-git`. You can then clone the repository from GitHub by going to the command prompt, navigating to the folder you want to download it into, and calling this command:

```
git clone https://github.com/djwester/todo-list-testing.git
```

After it has downloaded, navigate to the folder:

```
cd todo-list-testing
```

Start the application by calling `make run-dev`. You can now access the application by going to `https://localhost:8000`.

Now that you have this application running, let's look at sending a call to it.

Making a call to the test application

Let's practice some of the theory we have covered so far. First, open the application in your web browser. You should see a page like this:

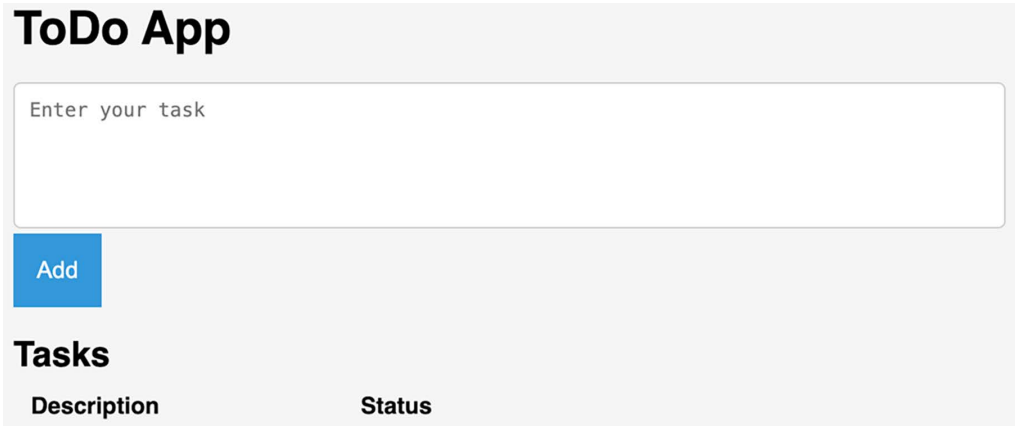

Figure 1.11: Todo list application

Type a task into the box and click on Add to add it to the list. You should see it appear in the list below the box. Now that there is something on the site, let's look at accessing it through the API:

1. Create a new collection in Postman and name it `TodoList`.
2. Add a request to the collection and name it `Get Todo List Item`.
3. In the URL field, enter the URL of the site that you built and then add /tasks to the end of it.
4. Send the request.

You should get back a response showing the item you created on the website.

A challenge

I've given you a lot of step-by-step instructions on how to make API calls, but now I have a challenge for you. I want you to try to do this without me giving you the exact steps to do it.

The documentation for this todo list site says that there is an endpoint like this:

```
/tasks/{task_id}
```

Can you call that endpoint in Postman for this todo list item that you've made? Don't forget that the curly braces around `task_id` indicate that it is a request parameter.

I will use the site in a few more examples and challenges as we go through this book, but for now, feel free to play around with the API and try out some of the things that we have covered already.

Considerations for API testing

We have started with the mechanics of how to make API requests, but this book is about API testing, so now that you know some of the basics of how an API request works, let's look at some of the things to consider when you test an API. One important aspect of testing is exploratory testing.

Beginning with exploration

I can still clearly remember the first time I saw a modern web API in action. The company I was working at was building a centralized reporting platform for all the automated tests, and I was assigned to help test the reporting platform. One of the key aspects of this platform was the ability to read and manipulate data through a web API. As I started testing this system, I quickly realized how powerful this paradigm was.

Another part of my job at that time was to work with a custom-built test automation system. This system was quite different from the more standard test automation framework that many others in the company were using.

However, the fact that the new reporting platform had an API meant that my custom test automation system could put data into this reporting platform, even though it worked very differently from the other test automation systems. The test reporting application did not need to know anything about how my system, or any other one, worked. This was a major paradigm shift for me and was probably instrumental in leading me down the path to writing this book. However, something else I noticed as I tested this was that there were flaws and shortcomings in the API.

It can be tempting to think that all API testing needs to be done programmatically, but I would argue that the place to start is with exploration. When I tested the API for that test reporting platform, I barely knew how to use an API, let alone how to automate tests for it, and yet I found many issues that we were able to correct. If you want to improve the quality of an API, you need to understand what it does and how it works. You need to explore it.

But how do you do that?

Thankfully, Postman is one of the best tools out there for exploring APIs. With Postman, you can easily try out many different endpoints and queries, and you can get instant feedback on what is going on in the API. Postman makes it easy to play around with an API and to go from one place to another. Exploring involves following the clues that are right in front of you. As you get results back from a request, you want to think of questions that those results bring to mind and try to answer them with further calls. This is all straightforward with Postman. To illustrate, I will walk you through a case study of a short exploratory session using Postman.

For this section, I will use the `https://swapi.dev/` API. This is a fun little API that exposes data about the Star Wars movies. Don't worry if you aren't into Star Wars. No specialized knowledge is needed!

Exploratory testing case study

Let's try this out. First, create a new collection called *Star Wars API* and add a request to it called *Get People*. Put `https://swapi.dev/api/people/1/` into the URL field and send that request. You should get back a response with some data for the character *Luke Skywalker*:

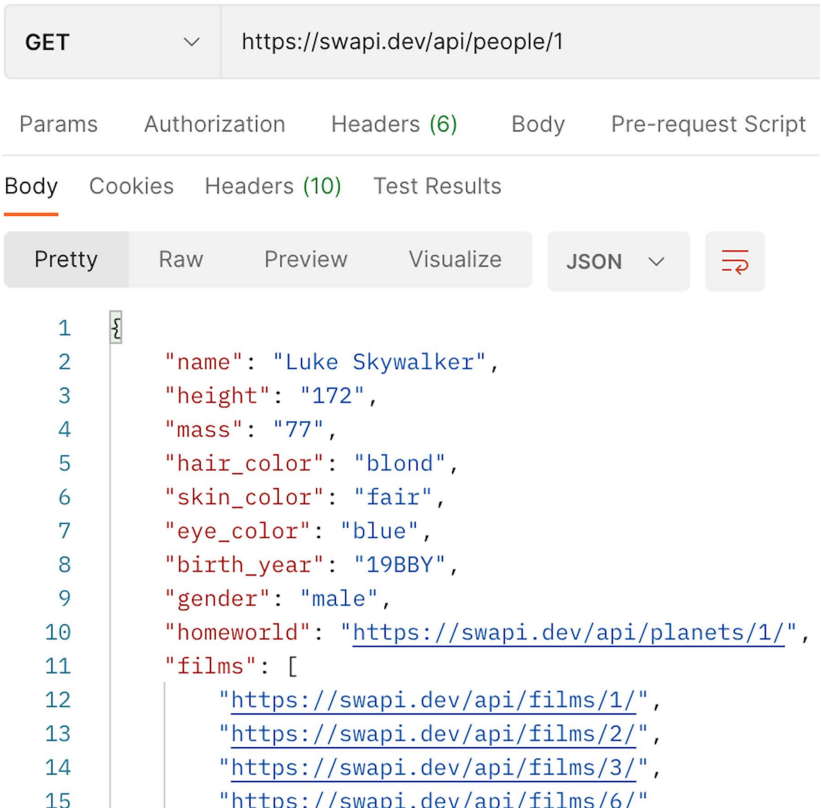

Figure 1.12: Response for the Star Wars API request

In the response, there are links to other parts of the API. A response that includes links to other relevant resources is known as a **Hypermedia** API. The first link in the films list will give us information about the film with ID 1. Since we are exploring, let's go ahead and look at that link and see what it does. You can just click on it and Postman will open a new requests tab with it already loaded. At this point, you know what to do: just click on **Send** and Postman will give you data about that first film. Under the characters in that film, you can see that there are several different characters, including, of course, a link to `/people/1`.

Seeing this triggers a thought that there might be more things to check in the `/people` API, so let's go back and explore that part of the API a bit more. Click on the **Get People** tab to go back to that request, and change the URL to remove the `/1` from the end of it. You can now click **Send** to send a request to the endpoint, `https://swapi.dev/api/people`. This response gives back a list of the different people in the database. You can see at the top of the response that it says there is a count of `82`.

We are in exploration mode, so we ask the question, "I wonder what would happen if I tried to request person number 83?" This API response seems to indicate that there are only 82 people, so perhaps something will go wrong if we try to get a person who is past the end of this dataset. In order to check this, add `/83` to the end of the URL and click **Send** again. Interestingly (if the API hasn't changed since I wrote this), we get back data for a Star Wars character. It seems like either the count is wrong, or perhaps a character has been removed somewhere along the line. Probably, we have just stumbled across a bug!

Whatever the case may be, this illustrates just how powerful a little bit of exploration can be. We will get to some powerful ways to use Postman for test automation later in this book, but don't rush right past the obvious. API testing is testing. When we are testing, we are trying to find out new information or problems that we might have missed. If we rush straight to test automation, we might miss some important things. Take the time to explore and understand APIs early in the process.

Exploration is a key part of any testing, but it takes more than just trying things out in an application. Good testing also requires the ability to connect the work you are doing to business value.

Looking for business problems

When considering API testing and design, it is important to consider the business problem that the API is solving. An API does not exist in a vacuum. It is there to help the company meet business needs. Understanding what those needs are will help to guide the testing approaches and strategies that you take. For example, if an API is only going to be used by internal consumers, the kinds of things that it needs to do and support are very different from those needed if it is going to be a public API that external clients can access.

When testing an API, look for business problems. If you can find problems that prevent the API from doing what the business needs it to do, you will be finding valuable problems and enhancing the quality of the application. Not all bugs are created equal.

Trying weird things

Not every user of an API is going to use it in the way that the API was written for. We are all limited by our own perspectives on life, and it is hard to get into someone else's mind and see things from their perspective. We can't know every possible thing that users of our system will do, but there are strategies that can help you improve seeing things in a different light. Try doing some things that are just weird or strange. Try different inputs and see what happens. Mess around with things that seem like they shouldn't be messed with. Do things that seem weird to you. Often, when you do this, nothing will happen, but occasionally, it will trigger something interesting that you might never have thought of otherwise.

Testing does not need to be the mindless repetition of the same test cases. Use your imagination. Try strange and interesting things. See what you can learn. The whole point of this book is for you to learn how to use a new tool. The fact that you have picked up this book shows that you are interested in learning. Take that learning attitude into your testing. Try something weird and see what happens.

There is obviously a lot more to testing than just these few considerations that I have gone over here. However, these are some important foundational principles for testing. I will cover a lot of different ways to use Postman for testing in this book, but most of the things that I talk about will be examples of how to put these strategies into practice. Before moving on to more details on using Postman though, I want to give you a brief survey of some of the different types of APIs that you might encounter.

Different types of APIs

There are several types of APIs commonly used on the internet. Before you dive too deep into the details of using Postman, it is worth knowing a bit about the different kinds of APIs and how to recognize and test them. In the following sections, I will provide brief explanations of the three most common types of APIs that you will see on the internet.

REST APIs

We'll start with what is probably the most common type of API you'll come across on the modern web, the **RESTful API**. REST stands for **R**epresentational **S**tate **T**ransfer and refers to an architectural style that guides you in terms of how you should create APIs. I won't go into the details of the properties that a RESTful API should have (you can look them up on Wikipedia if you want, at `https://en.wikipedia.org/wiki/Representational_state_transfer`), but there are a few clues that can let you know that you are probably testing a RESTful API.

Since RESTful APIs are based on a set of guidelines, they do not all look the same. There is no official standard that defines the exact specifications that a response must conform to. This means that many APIs that are considered to be RESTful do not strictly follow *all* the REST guidelines. REST in general has more flexibility than a standards-based protocol such as **SOAP** (covered in the next section), but this means that there can be a lot of diversity in the way REST APIs are defined and used.

So how do you know whether the API that you are looking at is RESTful?

Well, in the first place, what kind of requests are typically defined? Most REST APIs have `GET`, `POST`, `PUT`, and `DELETE` calls, with perhaps a few others. Depending on the needs of the API, it may not use all these actions, but those are the common ones that you will likely see if the API you are looking at is RESTful.

Another clue is in the types of requests or responses that are allowed by the API. Often, REST APIs will use JSON data in their responses (although they could use text or even XML). Generally speaking, if the data in the responses and requests of the API is not XML, there is a good chance you are dealing with a REST-based API of some sort. There are many examples of REST APIs on the web, and in fact, all the APIs that we have looked at so far in this book have all been RESTful.

SOAP APIs

Before REST, there was **SOAP**. SOAP has been around since long before Roy Fielding came up with the concept of REST APIs. It is not as widely used on the web now (especially for smaller applications), but for many years, it was the default way to make APIs, so there are still many SOAP APIs around.

SOAP is an actual protocol with a **W3C** standards definition. This means that its usage is much more strictly defined than REST, which is an architectural guideline as opposed to a strictly defined protocol.

If you want a little light reading, check out the w3 primer on the SOAP protocol (`https://www.w3.org/TR/soap12-part0/`). It claims to be a:

> *non-normative document intended to provide an easily understandable tutorial on the features of SOAP Version 1.2.*

OK, so the reading might not be quite as light as I promised, but looking at some of the examples in there may help you understand why REST APIs have become so popular. SOAP APIs require a highly structured XML message to be sent with the request. Being built in XML, these requests are not so human-readable and require a lot of complexity to build up. There are, of course, many tools (such as SoapUI) that can help with this, but in general, SOAP APIs tend to be a bit more complex to get started with. You need to know more information (such as the envelope structure).

But how do you know whether the API you are looking at is a SOAP API?

The most important rule of thumb here is, does it require you to specify structured XML in order to work? If it does, it's a SOAP API. Since these kinds of APIs are required to follow the W3C specification, they must use XML, and they must specify things such as `env:Envelope` nodes inside the XML. If the API you are looking at requires XML to be specified, and that XML includes the **Envelope** node, you are almost certainly dealing with a SOAP API.

Another indicator that you are dealing with a SOAP API is the existence of a **Web Services Description Language** (**WSDL**) file. This file is used to define the contract for how the API works. It is meant to be used programmatically by the consumer to help with calling the API. Not every SOAP API provides one, but many do, and if there is one, you can be sure that you are working with a SOAP API.

SOAP API example

Let's look at an example of what it would look like to call a SOAP API. This is a little bit harder than just sending a `GET` request to an endpoint, but Postman can still help us out with this. For this example, I will use the country info service to get a list of continents by name. The base page for that service is here: `http://webservices.oorsprong.org/websamples.countryinfo/CountryInfoService.wso`. In order to call this API in Postman, we will need to set up a few things. You will, of course, need to create a request in Postman. However, in this case, instead of having the request method as a `GET` request, you will need to set the request method to `POST` and then enter the URL specified above. SOAP requests are usually sent with the `POST` method rather than the `GET` method, as they have to send XML data in the body of the request. You usually don't send body data in a `GET` request, so most SOAP services require the requests to be sent using the `POST` protocol.

However, **don't** click **Send** yet. Since this is a SOAP API, we need to send some XML information as well. We want to get the list of continents by name, so if you go to the `CountryInfoServices` web page, you can click on the first link in the list, which will show you the XML definitions for that operation. Use the SOAP 1.2 example on that page and copy the XML for it.

In Postman, you will need to set the input body type to **raw**, choose **XML** from the dropdown, and then paste in the `Envelope` data that you copied from the documentation page. It should look something like this:

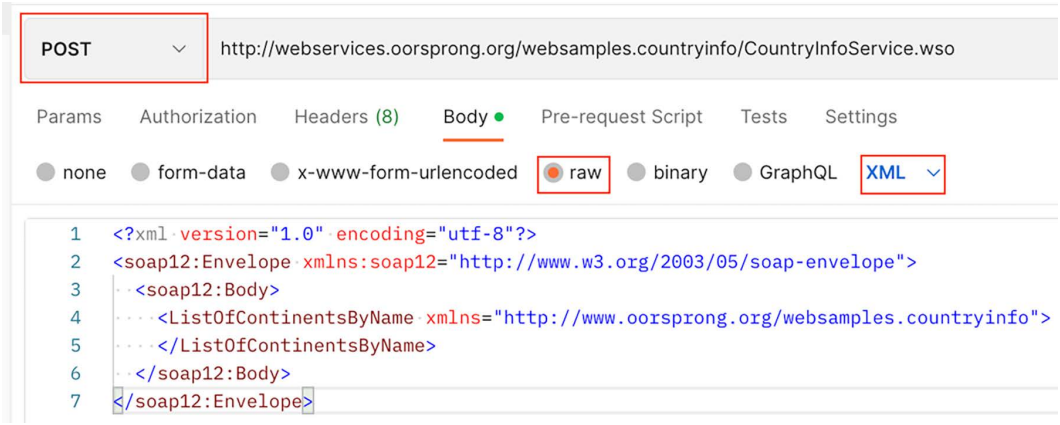

Figure 1.13: SOAP API information

For this particular API, we also need to modify the `Content-Type` header. Go to the **Headers** tab and click on the button labeled **hidden**. You will see that Postman has automatically added a Content-Type header with a value of `application/xml`. However, this API needs the content type set to `application/soap+xml`. We can't directly modify the automatically created header, but we can add one that replaces it. At the bottom of the list, type `Content-Type` into the **Key** field and set the value to `application/soap+xml`. Postman will automatically put a strikethrough in the autogenerated header, indicating that it won't use it and will instead use the one you specified. To be extra sure, you can deselect it from the list as well.

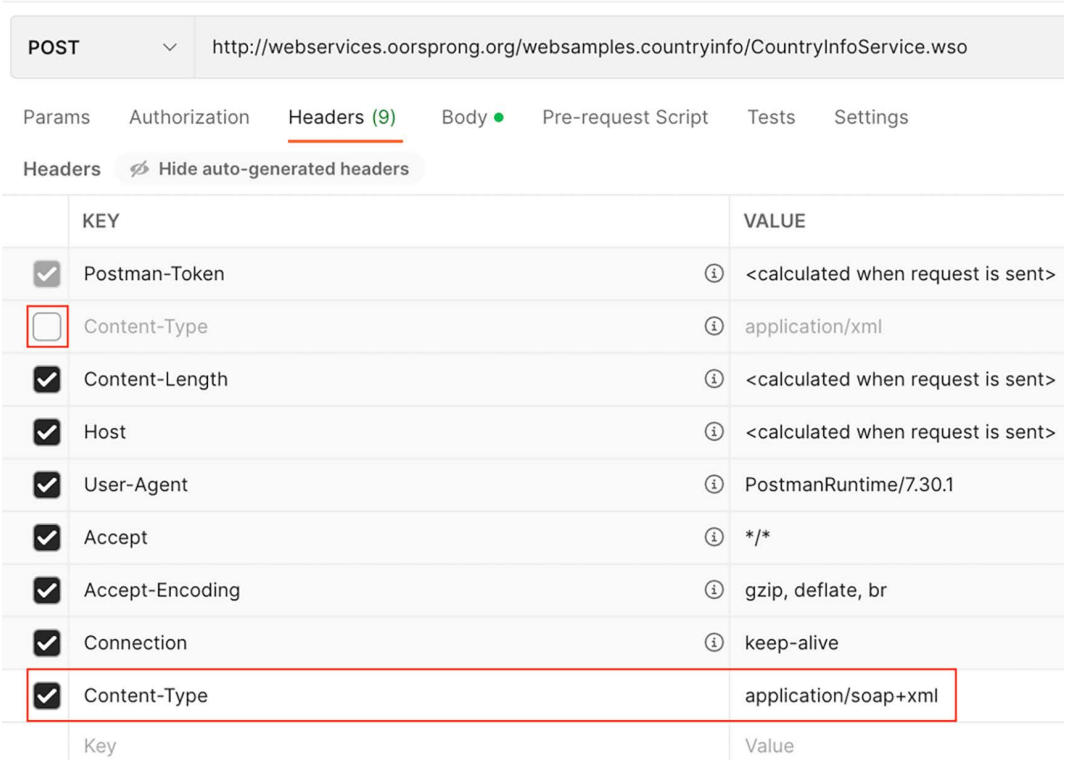

Figure 1.14: Content-Type header set to application/soap+xml

Now, you are finally ready to click on **Send**. You should get back a list of the continents. As you can see, there is a lot more complexity to calling SOAP APIs. This complexity is one of the contributors to the fall in popularity of these kinds of APIs, and it also indicates to us what type of API this is. REST APIs can, of course, have complex bodies specified as well, but the requirement to do this in XML and the existence of the **Envelope** node in this indicates that this API is indeed a SOAP API.

GraphQL APIs

SOAP came before REST, and in many ways, REST was designed to deal with some of the shortcomings of SOAP. Of course, in software, we are never done making things better, so we now have a type of API known as GraphQL. GraphQL is a query language that was designed to deal with some of the situations where REST APIs have shortcomings. RESTful APIs are not aware of what specific information you might be looking for, so when you call a REST API endpoint, it gives you all the information it has. This can mean that you can receive extra information that you don't need, or it can mean that you don't receive all the information you need and that you must call multiple endpoints to get what you want. Either of these cases can slow things down, and for big applications with many users, that can become problematic. GraphQL was designed by Facebook to deal with these issues.

GraphQL is a query language for APIs, so it requires you to specify in a query what you are looking for. With REST APIs, you will usually only need to know what the different endpoints are in order to find the information you are looking for, but with a GraphQL API, a single endpoint will contain most or all of the information you need, and you will use queries to filter down that information to only the bits that you are interested in. This means that with GraphQL APIs, you will need to know the schema or structure of the data so that you know how to properly query it, instead of needing to know what all the endpoints are.

How do you know whether the API you are looking at is a GraphQL API?

Well, if the documentation tells you about what kinds of queries you need to write, you are almost certainly looking at a GraphQL API. In some ways, a GraphQL API is similar to a SOAP API, in that you need to tell the service some information about what you are interested in. However, a SOAP API will always use XML and follow a strict definition in the calls, whereas GraphQL APIs are usually a bit simpler and not defined in XML. Also, with GraphQL, the way the schema is defined can vary from one API to another, as it does not need to follow a strictly set standard.

GraphQL API example

Let's look at a real-life example of calling a GraphQL API to understand it a bit better. This example will use a version of the countries API that you can find hosted at `https://countries.trevorblades.com/`. You can find information about the schema for it on GitHub: `https://github.com/trevorblades/countries`. Now, you could just create queries in the playground provided, but for this example, let's look at setting it up in Postman.

Similar to calling a SOAP API, we will need to specify the service we want and do a `POST` request rather than a `GET` request. GraphQL queries can be done with `GET`, but it is much easier to specify the query in the body of a `POST` request, so most GraphQL API calls are sent with the `POST` method. In fact, you will notice that when you choose the GraphQL option for the body of your request in Postman, it automatically changes the type to POST for you in anticipation that it is what a GraphQL request will need. So go ahead and do that; choose the **GraphQL** option on the **Body** tab and enter the query that you want:

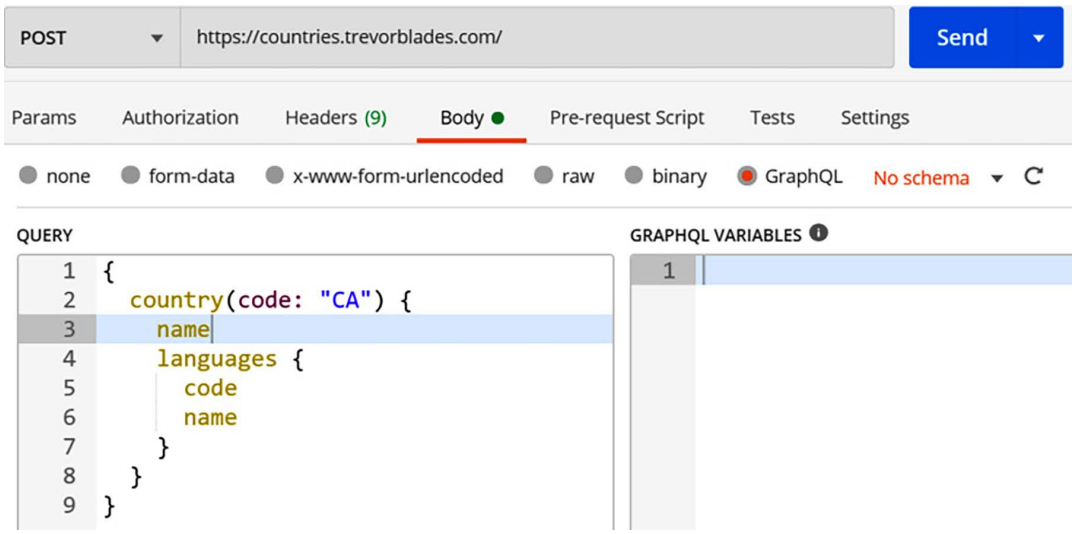

Figure 1.15: GraphQL query

As you type in your query, you might notice that Postman offers some autocomplete options. It can do this because it knows what structure the data has. Postman will automatically read the GraphQL schema for you and use that to help with constructing queries.

As you can see, in this example, I have requested the name and languages of Canada. Once I have specified this information, I can click **Send**, and I get back some JSON with the country name and a list of the official languages. If I wanted additional information (say the name of the capital city), I could just modify the query to include a request for that information and send it off again using the same endpoint, and I would get back the new set of information that I requested.

At this point, I have obviously only been able to give a very quick introduction to each of these types of APIs. If you are working with a GraphQL or SOAP API, you may need to spend a bit of time making sure you understand a little more about how they work before proceeding with this book. Most of the examples through the rest of this book will use RESTful APIs.

However, the concepts of API testing will largely stay the same, regardless of the type of API testing that you need to do. You should be able to take the things that you will learn in this book and put them to use, regardless of the type of API you work with in your day job.

Summary

Let's pause for a minute to consider everything we have gone over in this chapter. You've installed Postman and already made several API requests. You've learned how API requests work and how to call them. I explained some basic testing considerations and gave you strategies that you can start to use right away in your day-to-day work. You also got to make calls to GraphQL, SOAP, and REST APIs and learned a ton of API terminology.

You now have something firm to hold onto as we proceed through the rest of this book. I will take you deep into a lot of API testing and design topics and help you get the most out of Postman, but in order to get the most out of it and not feel frustrated, it would be good to make sure you understand the topics covered in this chapter.

Take a minute to ask yourself the following questions:

- Would I feel comfortable reading an article on API testing? Could I follow along with the terminology used?
- What are some basic strategies and approaches that I can use in API testing?
- If I was given an API endpoint, could I send a request to it in Postman? What things would I need to do to send that request?
- If I was given some API documentation, could I figure out what kind of API it was and send requests to it?

If you can answer these questions, you certainly have the grounding that you need to move on in this book. If you are not totally sure about some of them, you might want to review some of the relevant sections in this chapter and make sure you have a solid grip on them.

You've learned a lot already! In the next chapter, we will dive into some of the principles of API design and look at how to use Postman to help put those principles into practice when creating an API.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://discord.com/invite/nEN6EBYPq9`

2

API Documentation and Design

I believe that design, testing, and documentation are just as much a part of creating a quality product as development is. Perhaps they are even more important. You might find it surprising that a book on testing is spending so much time talking about documentation and design, but both of these practices are also key contributors to quality.

Good design can make the life of a developer or tester much easier. Designing something isn't so much about making it look good as it is about ensuring that it brings satisfaction. We, humans, enjoy things that look nice, so designs can sometimes be focused on the look and feel of something, but even in the realm of APIs, where there isn't much to "see," good design can make it much more enjoyable to use and thus improve the quality.

Good documentation also makes the life of a developer and tester much easier. I have struggled through working with APIs that have missing or incorrect API documentation. It makes it much harder to work with and test the API. You can spend hours trying to figure out how to make one call simply because there is one missing parameter that you need to include for the call to work, but with good documentation, you could have figured that out in minutes.

This chapter will go over the principles of API documentation and design and cover the following main topics:

- Figuring out the purpose of an API
- Creating usable APIs
- Documenting your API (with Postman)
- API design example (a practical exercise)

The material in this chapter can feel a bit abstract and theoretical, but I have included a few exercises to help you figure out how to use these concepts in practice. I would encourage you to spend the time to work through those exercises and, by the end of this chapter, you will be able to use these design principles to come up with great insights for APIs that you are currently working on, as well as having a strong foundation that you can use if you need to create a new API. This chapter will also help you get started with API documentation in Postman and show you how to use the RAML specification language to design and model an API and to automatically add requests into Postman.

Technical requirements

The code used in this chapter can be found at `https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter02`.

Start with the purpose

We don't just build an API because it is a fun thing to do. We build APIs to achieve a purpose. They help us or our users solve a problem. This might seem obvious, but let's be honest: we forget this far too often.

Design is a difficult domain. Perhaps your company has designers (as mine does). Technical people such as testers and developers can sometimes dismiss what designers do as "just making things look pretty," but the reality is that good design is very hard. Good design makes something that is suited for its purpose. This book is not a book on designs (if you want to read more about the theory of design, check out books such as *The Design of Everyday Things* by Don Norman). However, if you are interested in good-quality APIs, you need to think for a few minutes about the purpose of your API and how you will design it to meet that purpose.

Knowing that you need to do something is an important first step. Unfortunately, many talks, articles, and books stop there. But what good does that do you? You are convinced that it makes sense to design an API with the purpose in mind, but what is the purpose? What do you do if you don't know? How do you figure out the purpose of your API?

Figuring out the purpose of an API

You want to figure out the purpose of your API, but how do you do that? There are a few simple strategies that you can use for this. The first thing is to ask questions. If you or your team is writing an API, you were probably asked to do it by someone. Talk to that person! Why do they want the API? What do they want to do with it? What problems do they think it will solve? Ask them these questions and see if you can figure out some of what the purpose of the API is.

Personas

Another simple strategy that you can use is personas. A persona is simply a made-up person that you use to help you think through who might be using your API. For example, you might have a persona representing a user in your company who will be using the API to develop user interfaces, or you might have a persona of a software engineer who works for one of your clients and is using the API to develop a custom application. Thinking through different kinds of users and considering how they might interact with your API will help you understand the purpose that it serves.

When creating a persona, think about things that go beyond just the technology that the person might use or understand. Think about things like the goals and motivations they might have and the things that might frustrate them. It's also helpful to think of some personal details that might make them a bit more relatable to you. For example, you could think about whether they are a dog person or whether they have kids. Is there some other thing that makes them unique? In other words, what kind of person are they? Writing down details like this can seem a bit silly at times, but it helps you better empathize with this persona, and as you are able to do that, you will be more able to put yourself in their shoes and understand the kinds of things that are important to them. The more you understand this, the better you will be able to understand the purpose of your API.

The why

At the heart of figuring out the purpose is the question of why. Why are we making this API? The why is almost always about solving a problem. A great way to figure out the purpose of an API is to figure out what problem it solves. Does it make it easier to write UI elements? Does it enable clients to customize the application? Does it enable third-party developers to use your platform? Does it simplify integrations with other applications? What problem is your API solving? Answer these questions and you will be well on your way to knowing the purpose of your API.

NOTE:

This exercise of figuring out the purpose of an API doesn't just apply to new APIs. If you are working with an existing API, there is a lot of value in understanding the purpose. It may be too late to radically alter the design of the API, but there is no more important threat to quality than not actually helping people solve the problems they need to solve. If nothing else, understanding the purpose of an existing API that you are testing will help you figure out which bugs are important and which ones might not matter as much. It takes some skill to figure out which bugs are urgent and which are not, and understanding the purpose of the API helps with that.

Try it out

I have just given you some practical advice, but it won't do you much good if you don't use it. This book isn't just about filling your head with theory. It is about helping you get better at testing APIs. A little later in this book, I will show you some of the tools that Postman has for helping with API design, but right now, I want you to pause and try out what we have just been talking about.

Take an existing API that you have been working on and see if you can write down the purpose in two or three sentences. Use the following steps to work through the process:

1. Identify at least two key stakeholders for the API. Do this by asking the question "Who wants (or wanted) this API built?" Write down these stakeholder names.
2. If possible, talk to those stakeholders and ask them what they think the purpose of this API should be and why they want to build it. Write down their answers.
3. Create preferably two (but at least one) personas that list out the kinds of people that you think will be using the API. What skill level do they have? What work are they trying to accomplish? How will your API help them?
4. Write down what problem(s) you think the API will solve.
5. Now, take all the information that you have gathered and look through it. Distill it down into two or three sentences that explain the purpose of the API.

Once you've completed this exercise, you will have a good grasp of the purpose of the API you are investigating. The result of that might look like something like this:

"This API was built so that other services in our company could communicate with our service. It will be primarily consumed by other developers in the company. It is an internal API that will only be accessible inside of our internal company network.

Its purpose is to let other parts of our system know about the status of certain events in our service and so some of the endpoints will be polled intermittently, but overall, it isn't expected to have heavy usage rates."

It is important to understand the purpose of an API; however, to fulfill that purpose, an API needs to be usable. Let's move on to look at what makes a usable API.

Creating usable APIs

Usability is about the balance between exposing too many controls and too few. This is a very tricky thing to get right. On the extremes, it is obvious when things are out of balance. For example, the Metropolitan Museum of Art has an API that gives you information about various art objects in their possession. If all the API did was provide one call that gave you back all that data, it would be providing too few controls. You would need to do so much work after getting the information that you might as well not use the API at all. However, if, on the other hand, the API gave you a separate endpoint for every piece of metadata in the system, you would have trouble finding the endpoint that gave you the particular information you wanted. You would need to comprehend too much in order to use the system.

You need to think carefully about this if you want to get the balance right. Make sure your API is providing users with specific enough data for the things they need (this is where knowing the purpose comes in handy) without overwhelming them. In other words, keep it as simple as possible.

Usable API structure

One thing that can help create a usable API is to use only **nouns** as endpoints. If you want users to be able to understand your API, structure it according to the objects in your system. For example, if you want to let API users get information about the students in a learning system, don't create an endpoint called `/getAllStudents`. Create one called `/students` and call it with the `GET` method. One of the reasons for doing this has to do with creating new students. If your endpoint is called `/students`, you can easily call it with a `POST` method to create a new student. However, if you had named it `/getAllStudents`, it would feel very unnatural to call that with a method to create new items. You can think about it in terms of saying it out loud. Saying "I want to create a new student object" sounds natural, but saying "I want to create a get all student object" does not.

Creating endpoints based on nouns will also help you more naturally structure your data. For example, if you have `/students` as an endpoint, you can easily add an endpoint for each student at `/students/{studentId}`. This kind of categorization structure is another helpful API design principle to keep in mind.

Creating a structure like this maps the layout of the API to the kinds of things that the API user needs information about. This makes it much easier to know where to find the relevant information.

A structure like this works nicely, but does it really match up with how users will interact with the API? If I am looking for information about a student, am I going to know what their ID is in the API? Perhaps, but more likely I will know something like their name. So, should we modify the structure to have an additional endpoint like `/students/name`? But what if we are looking at all the students of a certain age? Should we add another endpoint, `/students/age`? You can see where I am going with this. It can get messy quickly.

This is where **query parameters** are helpful. A query parameter is a way of getting some subset of the category based on a property that it has. So, in the examples that I gave earlier, instead of making "name" and "age" endpoints under the "students" category, we could just create query parameters. We would call `/students?name='JimJones'` or `/students?age=25`. Query parameters help keep the endpoints simple and logical but still give the users the flexibility to get the information they are interested in effectively.

Good error messages

A usable API helps the users when they make mistakes. This means that you give them the correct **HTTP codes** when responding to a call. If the request is badly formatted, the API should return a 400 error code. I won't list all the HTTP codes here as they are easily searchable online, but ensuring that your API is returning codes that make sense for the kind of response you are getting is an important quality consideration.

In addition to the HTTP codes, some APIs may return messages that let you know what possible steps you can take to correct this issue. This can be very helpful, although you will want to be careful that you don't reveal too much information to those who might be bad actors.

Documenting your API

One often-overlooked yet vitally important aspect of a good-quality API is proper documentation. Sometimes, testers and developers can overlook documentation as something that is outside of their area of expertise. If you have a team that writes documentation for your API, it is certainly fine to have them write it, but don't treat documentation as a second-class citizen! Documentation is often the first exposure people have to your API, and if it does not point people in the direction they need to go, they will get confused and frustrated. No matter how well you have designed your API, users will need some documentation to help them know what it can do and how to use it.

Documenting with Postman

Postman allows you to create documentation directly in the application. In *Chapter 1*, *API Terminology and Types*, I showed you how to create a request to the GitHub API. Of course, GitHub has its own API documentation, but let's look at how you can create documentation in Postman using that API request. If you did not create the GitHub collection, you can import the collection from the GitHub repo for this chapter:

1. Download the `GitHub API Requests.postman_collection.json` collection file from `https://github.com/PacktPublishing/API-Testing-and-Development-with-Postman-Second-Edition/tree/master/Chapter02`.
2. In Postman, click on the **Import** button at the top of the navigation tree:

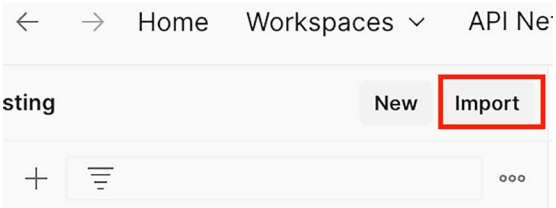

Figure 2.1: The Import button

3. On the resulting dialog, click on the **Choose Files** button, browse to where you downloaded the collection file, and select it.
4. Click on the **Import** button and Postman will import the collection for you.

Once you have the collection set up, you can create some documentation with the following steps:

1. The first thing to do is navigate to the **GitHub API Requests** collection and click on it.
2. Click on the **View complete documentation** link.

This brings up the **Documentation** panel.

In the top section, you can enter documentation for the collection itself, if you want. There might be times when you want to do this but, in this case, you only have one request so there probably isn't too much documentation to put in here. Instead, you can scroll down to the next section to enter documentation for the **Get User Repos** request.

Note that you can also jump to a request by clicking on it in the right-hand panel, as shown in the following figure:

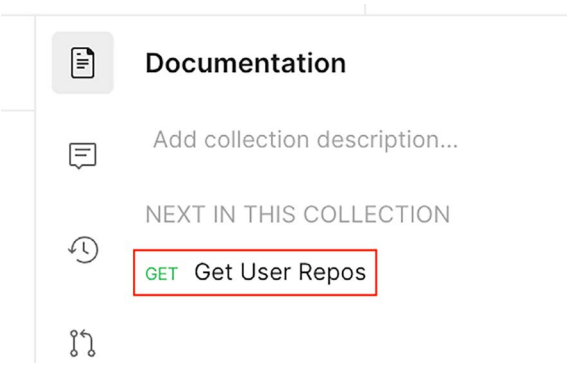

Figure 2.2: Going to the request documentation

In this case, we currently only have one request in this collection, so it is just as easy to scroll. However, if you have many requests, the jump navigation will let you quickly navigate to the one you are interested in. In order to edit the documentation, simply click on the area below the URL for the request and you can start typing your documentation for this request directly into the provided textbox.

NOTE:

Postman API descriptions support Markdown. If you are not familiar with Markdown, it is a simple set of rules that can be used to format text. A great resource showing the kinds of commands available is the Markdown cheat sheet, which you can find at `https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet`.

Let's write down some documentation for this endpoint:

1. At this point, just type something simple in here so that you can try it out. Type in something like this: `This endpoint will get information about what repos the given user has.`
2. When you click away from the editing panel, Postman will auto-save your changes. However, this documentation is currently only available in this collection. You can make it more public by publishing it.

3. In order to publish, click on the **Publish** icon near the top of the page, as shown in the following figure:

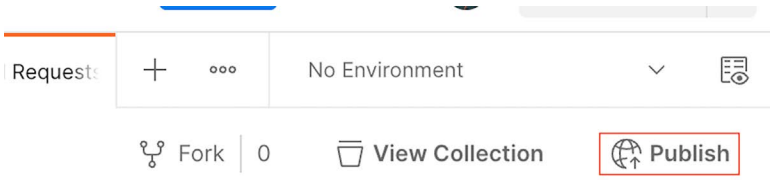

Figure 2.3: Publish documentation

4. This opens a web page where you can choose various styling options for your documentation. Once you have set up the options that you want on this page, you can scroll down and click on the **Publish Collection** button to make this documentation available to others.
5. The published page tells you the URL from which you can see the documentation that you made. Click on that and you will see a nice documentation page that you can share with anyone.

When looking at this documentation, it seems a bit sparse. It could be improved with an example request showing what a response looks like. You can easily add examples in Postman with the following steps:

1. Return to Postman and go to the `Get User Repos` request.
2. Replace the `{{username}}` variable in the request URL with a valid username (you could use your own or just put in mine, which is `djwester`). In future chapters, I will show you how to work with variables, but for now, just manually replace it.
3. Click on the **Send** button for the request to send it off. If you get back an empty array, it could be that your GitHub has privacy settings that don't allow these results to be returned. In that case, use my username (`djwester`) instead.
4. Now go down to the response and click on the **Save as example** button.

Figure 2.4: Adding examples

5. Postman will automatically add an example to this request. Return to your API documentation page and refresh it. You will now see an example request and response:

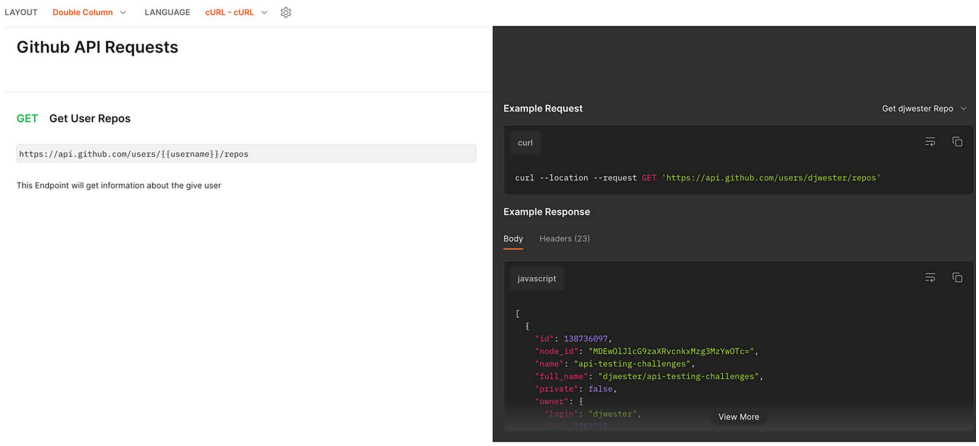

Figure 2.5: Example API request and response in documentation

It is worth spending more time exploring how to document your API well, but these features make it straightforward to do in Postman. Whether you are a tester or developer and whether you are working on a new API or one that has been around for some time, I would encourage you to take the time to properly document your API. Good documentation will make a world of difference to your users and will also lead to you and your team having a better understanding of how it works. It is well worth the time that it takes, especially when working with a tool like Postman that makes it so easy to do. In fact, although this is not a book on API documentation, it is worth spending a few minutes on some of the theory of how to write good API documentation.

Good practices for API documentation

One key practice to keep in mind when writing API documentation is **consistency**. In this chapter and the previous one, I have been giving you a lot of API-related terminology. There is always going to be some variation in how terms are used but, as much as possible, try to use terms in a way that is consistent with how other APIs use them. You should also try to have as much internal consistency as possible in your documentation. This means things like using the same terms to mean the same thing throughout the documentation. It also means that, as much as possible, you want your API structure to be consistent so that you don't have to explain similar things in different ways throughout your documentation.

Writing API documentation can benefit you and not just those reading it. Sometimes, in reading or creating documentation, things will jump out at you. You might see how two things that are quite similar are laid out in very different ways. This can help you see where you might need to fix or change things in the API itself, which is another benefit of good API documentation.

Another very important thing to keep in mind when it comes to API documentation is the importance of examples. We already saw how easy it is to make examples in Postman documentation. Take advantage of that! It has been said that a picture is worth a thousand words. I think that an example in an API doc is worth nearly as many. Some things are difficult to explain well in words and it takes seeing it done in an example for people to grasp how it works. Do not underestimate the power of a well-crafted example.

One final thing to keep in mind with API documentation is a problem that happens with all documentation. It gets out of date. Code changes, and that applies to API code as well. Over time, what your API does and the exact ways that it does it will change. If you don't keep your documentation up to date to match those changes, the documentation will soon cease to be useful.

You have already seen how you can keep your documentation right with your tests and how you can generate examples directly from your requests in Postman. Take advantage of that and make sure your documentation stays up to date. Postman will take care of a lot of the work for you by updating all the documentation automatically every time you publish.

There are some specification tools that can help automatically generate documentation. These kinds of tools can help keep documentation and tests up to date with the latest code changes. I will take you through some of those in more detail in *Chapter 4*, *Considerations for Good API Test Automation*. In that chapter, I will particularly focus on the Open API specification, as it is a powerful and popular API specification tool, but there are other API specification tools that we can use.

There is an ongoing debate in the API development community that mirrors broader debates in the software development community as a whole. The debate boils down to how much time you should spend upfront on design. Some will argue that since the only good software is software that helps someone solve a problem, we need a "ship first" mentality that gets our ideas out there. We then fill out the design based on the kinds of things clients actually need and care about. Others will argue for a "design first" approach, where you rigorously define the API behavior before you write any code. As with most things, you are probably best off avoiding either ditch and finding the middle ground between them.

Modern tooling can help us do this. For example, there are tools that will allow you to create simple designs and then use those designs to get feedback from clients. One tool that is helpful for API design is **specification languages**. These are sets of rules that can be used to define the way an API is expected to work. As we look into how to design APIs, let's look at a specification language that is meant to help shift the focus away from merely documenting an API and toward helping with API design.

RESTful API Modeling Language

RAML, which stands for **RESTful API Modeling Language**, is an API specification language that, as the name implies, helps with modeling APIs. You can read more about it on the RAML website, `https://raml.org/`. RAML has some tools that can help with API design but getting into those is beyond the scope of what I want to cover in this book. For now, I just want to introduce you to this specification and let you see how you can use it to design an API that meets the design criterion I've talked about in this chapter.

Getting started with RAML is as easy as opening a text editor and typing in some text. RAML is meant to be human-readable and so the specification is written in a simple text-based format. RAML is also structured hierarchically, which makes it easy to create the kind of usable API structures that I've talked about. In the next section, I will walk you through an example of using this modeling language to design an API and leverage the power of that design in Postman. You will then get to try it out on your own as well!

API design example

I have talked about a lot of the theory of API design, so now I want to look at how you can use Postman to help you out with practically designing an API. API design does not only apply to new APIs that you create. In fact, using the principles of API design when testing an existing API is a great way to find potential threats to the value of that API, but for the sake of understanding this better, let's look at how you can design an API from scratch. If you understand the principles through this kind of example, you should be able to use them on existing APIs as well.

Case study – Designing an e-commerce API

Let's imagine that we want to design an API for a very simple e-commerce application. This application has a few products that you can look at. It also allows users to create a profile that they can use when adding items to their cart and purchasing them. The purpose of this API is to expose the data in a way that can be used by both the web and mobile application user interfaces. Your team has just been given this information and you need to come up with an API that will do this.

So, let's walk through this and see how to apply the design principles we've covered. I will start with a simple RAML definition of the API. The first thing we need is to create a file and tell it what version of RAML we are using. I did this by creating a text file in Visual Studio Code (you can use whatever text editor you prefer) called `E-Commerce_API-Design.raml`. I then added a reference to the top of the file to let it know that I want to use the 1.0 version of the RAML specification:

```
#%RAML 1.0
---
```

I also needed to give the API a title and set up the base URI for this API, so, next, I defined those in the file:

```
title: E-Commerce API
baseUri: https://api.ecommerce.com/{version}
version: v1
```

Defining the endpoints

This is a made-up API so that base URI reference does not point to a real website. Notice also how the version has been specified. Now that I have defined the root or base of the API, I can start to design the actual structure and commands that this API will have. I need to start with the purpose of this API, which is to enable both a website and a mobile app. For this case study, I am not going to dive too deep into things like creating personas. However, we do know that this API will be used by the frontend developers to enable what they can show to the users. With a bit of thought, we can assume that they will need to be able to get product information that they can show the users. They will also need to be able to access a user's account data and allow users to create or modify that data. Finally, they will need to be able to add and remove items from the cart.

With that information in hand about the purpose of the API, I can start to think about the usability and structure of this API. The frontend developers will probably need a `/products` endpoint to enable the display of product information. They will also need a `/users` endpoint for reading and maintaining the user data and a `/carts` endpoint that will allow the developers to add and remove items from a cart.

These endpoints are not the only way that you could lay out this API. For example, you could fold the **carts** endpoint into the **users** one. Each cart needs to belong to a user, so you could choose to have the cart be a property of the user if you wanted. It is exactly because there are different ways to lay out an API that we need to consider things like the purpose of the API. In this case, we know that the workflow will require adding and removing items from a cart regularly.

Developers will be thinking about what they need to do in those terms, and so to make them call a "users" endpoint to modify a cart would cause extra data to be returned that they do not need in that context and could also cause some confusion.

Now that I have picked the endpoints I want to use in this API, I will put them into the RAML specification file. That is simply a matter of typing them into the file with a colon at the end of each one:

```
/products:
/users:
/carts:
```

Defining the actions

Of course, we need to be able to do something with these endpoints. What actions do you think each of these endpoints should have? Take a second to think about what actions you would use for each of these endpoints.

My initial thought was that we should have the following actions for each endpoint:

```
/products:
    get:
/users:
    get:
    post:
    put:
/carts:
    get:
    post:
    put:
```

Think about this for a minute, though. If I only have one endpoint, `/carts`, for getting information about the carts, I need to get and update information about every cart in the system every time I want to do something with any cart in the system. I need to take a step back here and define this a little better. The endpoints are plural here and should represent collections or lists of objects. I need some way to interact with individual objects in each of these categories:

```
/products:
    get:
    /{productId}:
        get:
```

```
/users:
    post:
    get:
    /{username}:
        get:
        put:
/carts:
    post:
    get:
    /{cartId}:
        get:
        put:
```

Here, I have defined **URI parameters** that enable users to get information about a particular product, user, or cart. You will notice that the `POST` commands stay with the collection endpoint, as sending a `POST` action to a collection will add a new object to that collection. I am also allowing API users to get a full list of each of the collections as well if they want.

Adding query parameters

In *Chapter 1*, *API Terminology and Types*, you learned about query parameters. Looking at this from the perspective of the users of the API, I think it would be helpful to use a query parameter in the carts endpoint. When a user clicks on a product and wants to add that product to their cart, the developer will already know the product ID based on the item the user clicked. However, the developer might not have information about the cart ID. In this case, they would have to do some sort of search through the carts collection to find the cart that belonged to the current user. I can make that easier for them by creating a query parameter. Once again, I am using the design principles of usability and purpose to help create a good model of how this API should work.

In RAML, I just need to create a query parameter entry under the action that I want to have a query parameter:

```
/carts:
    post:
    get:
        queryParameter:
            username:
    /{cartId}:
        get:
        put:
```

I have designed the structure of the API using the principles I laid out, and hopefully, you can see how powerful these principles are in helping you narrow down a broad space into something manageable and useful. When it comes to creating a RAML specification for an API, you would still need to specify the individual attributes or properties of each of the endpoints and query parameters that you want to create. I won't go into all those details here. You can look at the RAML tutorials (`https://raml.org/developers/raml-100-tutorial`) to learn more about how to do this. I didn't give enough information about this imaginary API that we are designing to fill out all the properties. We don't know what information each product or user has, for example. In real life, you would probably get this information based on what is in the database and then build out the examples and attributes based on that.

Using the RAML specification in Postman

This might not be a fully-fledged API specification, but it is enough for us to use in Postman! Click on the **Import** button in Postman and browse to the `.raml` file from this case study. Before importing it, check the **View Import Settings** section:

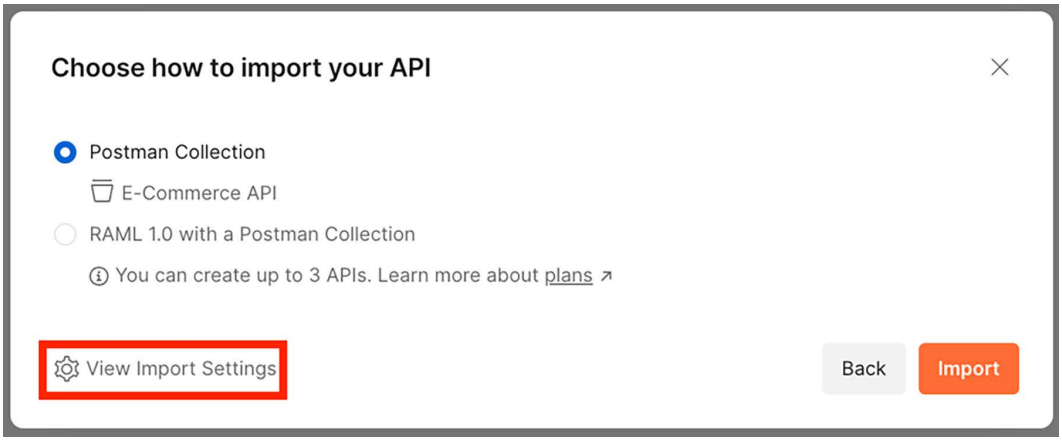

Figure 2.6: Import Settings

Ensure that both the request and response parameter generation options are set to use **Schema**. Return to the Import panel and click on **Import**. Postman will parse the specification for you and automatically create a collection that has calls defined for each of the endpoints and action combinations that you created in the specification file. Pretty cool, isn't it?

Once you have learned about some more Postman features, I will show you how to use these concepts to dive a lot deeper into the design process for APIs. For now, though, I want you to be able to think about how API design intersects with the quality of the API. In fact, I want to give you a little challenge to try out.

Modeling an existing API design

You have been learning how to think through API design. I gave you some principles for this and then walked you through putting the principles into practice in a case study. The case study we just went through involved designing a new API from scratch. Often, though, you will be working with existing APIs. These same design principles can be used to help you find places to improve your existing APIs. In order to learn these principles, you should try this exercise out for yourself. See if you can use these principles on an API that you are currently working on. If the company you are working at does not have an API that you can use, you can, of course, just find a public API to try out this exercise with.

Using the API you have selected, work through the following steps to apply the principles of API design:

1. Add each of the endpoints to a RAML file. Make sure to follow the hierarchy of the API in that file.
2. Spend a bit of time thinking about what the purpose of the API is and reflecting on whether the structure that you see here fits with that purpose. In what other ways might you design the API? Could you improve the layout to better fit the purpose?
3. If you were designing this API, what actions and query parameters would you give to each endpoint? Create a copy of the file and fill it in with what you think you would do with this API.
4. In the original file, add the actual actions and query parameters that the API has. How do they compare to the ones that you made in the copy of the file?

If you want, you can import the file into Postman and, as you learn more about testing and other features that you can use in Postman, you will already have a collection of requests ready to use.

Summary

Designing an API takes careful thought. An API is software, and the whole reason we write software is to help people solve problems. APIs need to solve a problem, and the better they solve that problem, the better quality they are. One thing that can help with API quality is to have a well-designed API. A well-designed API is one that is designed to fulfill the purpose for which it is used. In this chapter, you learned how to think through the purpose of an API by coming up with personas. You also learned some questions that you can ask to get to the heart of why an API needs to exist in the first place.

This chapter also showed you some ways to structure and document APIs. You learned about API specification and how you can use RAML to create design-driven specifications for an API. And, of course, you also got to try these things out and put them into practice! With a firm grasp of the principles of API design, you are ready to dig a little deeper into API specification languages and how to use them in creating good-quality APIs. We will dive into that in the next chapter.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

`https://discord.com/invite/nEN6EBYPq9`