Contents

Chapter 1

The Great Network

1.1 Before we begin

Modern web development seems daunting at first. A brave wanderer is expected to learn a lot of things in order to qualify as a knowledgeable web developer. And even more if they are after the notorious "full-stack" role.

The term "full-stack developer" is so relevant in web development for these very reasons: the "arsenal" or the "stack" of things needed to be put on top of one another in order to make a web app work is so large, just "developer" doesn't cut it. You have to know or at least understand several layers.

With full stack comes the full responsibility.

The speed of progress and innovation in this field makes things somewhat worse. For one technology conquered, you face three new ones next year. This hydra appears to be indestructible, but seeing it as a constant opponent is actually the wrong thing to do. Don't fight the hydra, but rather learn to harness and ride it. Learn the underlying principles that don't change for decades. Once you dig deep enough, things become clearer and much more stable. And simple, for that matter.

When I started learning web development, I felt lost and confused for a long time. Books and courses were teaching me to make websites

or interact with databases or make requests, but the more techniques I learned, the more questions I had left with.

Here are some of the questions I had:

- does it **always** work like that or was it some special case?
- who decides how HTTP works?
- is "server" a special thing or… what is it?
- is "database" a program? a file? what makes it a "base"?
- what the heck is DNS?!
- okay, this works on my computer? how do I make it work on some domain?

This book is created to help you see the big picture. To clear the fog of war, rise above the minutiae and understand what the hell is going on.

It is written for two groups of people:

1. **Absolute beginners** who are about to start their long and fascinating journey into web programming. You will see the landscape ahead from 50 000 feet.
2. **Novice web developers** who feel like they don't see the whole picture or struggle to connect the dots.

In other words, this is an attempt to lay the foundation. I hope you'll like it.

1.2 The Internet vs The Web

The Internet is simply a collection of computers connected with each other. The web emerges when this mega-network is used to deliver content: text, video, music, chat messages, etc. Web browsers like Chrome or Firefox are apps that communicate with other apps — web servers — to send and request information. And the internet is the underlying infrastructure that makes it possible.

Think of the internet as the system of roads, highways and traffic lights. Think of the web as the postal service. The postal service uses the roads. It would stop working if all roads suddenly vanished. But you can imagine them finding other ways to deliver mail, if needed.

For all practical purposes, many use the words "internet" and "web" interchangeably, but they are not the same thing. The good news is that

the responsibility of web developers usually don't go into the land of the Internet.

Computer networking is a separate, complex topic. It is not essential to learn all the ins and outs right now, it will be enough to understand what networks are and what do people mean when they say "computers are connected".

I have a laptop on my desk, and there's another computer nearby. I want to read the document that's stored on that other computer. What are my options? Well, I can use something like USB thumb drive, copy the file and transfer to my laptop. This was a common problem from the very beginning of computing. So engineers decided from early on to come up with a way to connect computers using wires.

It turned out to be really handy: you can transfer documents and all kinds of information, including commands, instructions for the other computer to run. For example, if my computer is not powerful enough but is connected to another more powerful computer, I can use its computational resources and accomplish something I couldn't have done myself.

Many organizations in the 60s started connecting their computers into networks. And then someone said "hey, why don't we connect our networks into a bigger network. A network of networks!" And that is exactly what Internet is: an **inter**-connected collection of networks.

Your computer, phone or tablet is most probably connected to the Internet right now. It means that it is a part of some small network that's connected to a bigger network that's connected to multiple bigger networks. That small network might be your local house network, your Internet provider's neighborhood network, your city and country networks, etc.

While we enjoy wireless connections all the time, most of the Internet is still a bunch of wires. There are thousands of kilometers of undersea cables that connect continents. It's quite amazing! And it's funny to think that connection problems on the Internet sometimes occur because of hungry fish.

Each computer connected to a network, be it a small local network or a global Internet, has a unique address called **an IP address**. Users don't deal with IP addresses, because `172.217.21.174` is not as convenient as `google.com`, but in order for computers to connect to each other, IP

addresses are required. This is why there are special computer applications that run constantly on the internet and help other apps convert words like `google.com` into corresponding IP-addresses. You actually have at least a few of these applications running on your devices.

They are called DNS servers — Domain Name System servers, and you have one in your computer, one in your smartphone and probably one in your wifi-router. There are millions of DNS servers on the internet, and they communicate with each other in a specific way, which, again, goes beyond the scope of this book and course. But in the end of the day, they made it possible for us to forget about IP-addresses and only think in terms of human-readable, nice addresses like `codexpanse.com` or `nasa.gov`.

As you can imagine, it takes quite a bit of engineering to make it all work, especially considering how fast everything must be. During a normal day you can easily get millions and millions of pages of information from five continents without leaving your living room. But as web developers, we don't have to think about this too much. When we create web applications we treat the Internet as a magical portal that somehow connects the client and the server.

In order to understand how the web works, we will explore the simplest web setup possible: two computers talking to each other via Internet.

Chapter 2

HTTP

2.1 HTTP and web clients

The first thing to understand is that when people say client or server they just mean a computer program that is capable of using the magical portal of the Internet. The most common example of a web client is a web browser: Google Chrome, Firefox, Safari etc. Nowadays quite often a web client is built into other appreciations. For example, a game you play on your phone can show you the global leader board by connecting to some server via the Internet.

You can think of a web client as a human client in a restaurant. The waiters in this restaurant are a bit shy, and they never talk to you first, but diligently wait for your requests. So, on the web everything starts with the client.

While in a restaurant, you obey multiple rules of communication and interaction. Both you and the waiter speak the same language and know these rules. If you say something that doesn't make sense in the context of a restaurant, like "let me dance with the cats while you watch", the waiter will probably tell you that they don't understand you. But if you follow the rules and pick something from the menu, the waiter will bring it to you.

The situation is similar when a web client is interacting with the web server. There is a set of rules, a language of sorts that both client and

Figure 2.1: Tom's Restaurant in New York City

the server speak. Don't confuse this language with a programming language. Programming languages are the languages in which the programs are written. But this web language is the language that programs *speak*. Similar to programming languages, the web language must be strict and simple, so that it's equally effective for programs to interpret and humans to read or write.

Languages like that are called protocols. To avoid further confusion with programming languages, we shall only call them protocols from now on.

The protocol which the web browser uses to communicate with the server is called HTTP: Hypertext Transfer Protocol. Hypertext is a fancy word for "interactive, rich text and multimedia", in other words, all that cool stuff we see in the browser: pages, images, videos, animations.

HTTP consists of methods (also called 'verbs' sometimes) and some additional information. There are just a few of possible methods. The one we are interested in right now is called GET. It makes sense, since we want to **get** a web page.

So, assuming the client knows how to address the server, all it needs to do is to write down its request following the rules of HTTP. It is literally some text written down and sent via that magical portal. This is how it

looks like for the simplest kind of request — a GET request that, as the name suggests, asks the server to get some information.

```
GET /article.html HTTP/1.1
Host: coolsite.com
```

The first thing after the GET verb is the name of the thing that we are interested in. In this case, it's a file called `article.html`. Then the rules require us to specify the version of the protocol. The client and the server need to make sure that they speak the same language, and just like human languages, the HTTP protocol changes over time. When a change occurs, it is marked as a new version. So, here, by saying `HTTP 1.1` the client signals to the server that its request follows the rules of a HTTP version `1.1`.

It would be interesting if we did the same in our everyday lives. Imagine someone approaching you on the street and asking for directions by first declaring the version of the language they speak:

— Hello, I'm speaking NYC slang. Yo, I'm mad lost, which way to Daly avenue?

The second line of the request must specify the host or in other words a website. This is needed because on the other side of the portal is a computer, and the Web server on that computer might be running multiple websites at the same time. By specifying the host the client makes sure that its request is delivered to the particular website. And that's it. A small computer program capable of sending this text into the portal is considered to be a web client. Of course, useful with client will also be capable of processing whatever response the server sends back. Which brings us to the second piece of the puzzle.

2.2 Server response

The second piece of the puzzle is the server. Just like a server at a restaurant, the Web server diligently awaits clients' instructions. It too speaks the HTTP protocol. When client sends out the verb to the server it is called HTTP request. The job of the server is to respond to each request. You might've seen the response request timeout error in the browser. It appears when the server is so busy and overwhelmed that it cannot respond in the reasonable amount of time. With clients like your browser usually don't wait forever, thus saying timeout as in "I waited

and waited but it doesn't respond so I bailed".

So, a good behaving server sends out an HTTP response back to the client. If everything goes well in the requested thing actually exists, then the response will include a cheerful success message, some rather boring technical information, and, more importantly, the text that the client wanted.

At this point the server completely forgets about the client. It doesn't even try to remember who the client is, what kind of things it's interested in, or whether it's going to request more things. Continuing our analogy with a restaurant, imagine that the waiter is never assigned to a table. You can always raise your hand and ask for something, but every time a new waiter comes. It's completely fine if you are asking for particular things every time, but it requires additional explanation from you if you want to maintain common context. So, if you had a glass of red wine, you can't just ask for another glass. You have to specify a glass of red wine since the waiter has no idea what you had.

As a user you have a different experience. Once you login to Gmail you usually stay logged in for months. The Gmail server does indeed remember you, even after you reload your computer. In order to achieve this illusion of consistency and memory the Web server and a web browser do lots of smart tricks. you will learn these tricks later but for now I encourage you to acknowledge the underlying simplicity of the web. Every single thing that happens in your browser, starting from simple articles on Wikipedia and all the way to interactive web applications, is the result of hundreds and hundreds of primitive requests and responses from an amnesiac.

The final piece of the puzzle is the pipe between the client and the server. As I said before, the Internet is the thing that allows pieces of the web to talk to each other. The client and the server speak HTTP, but the underlying Internet speaks a different protocol. As web developers we don't have to think about this too much, but it's useful to understand that in reality HTTP makes sense only on the ends of the pipe, and while in the pipe requests and responses are wrapped inside a whole different protocol. The cool thing about software engineering and computer science is that it provides lots of abstractions: things that hide the way complex systems and make it so that you only have to think in terms of something that's relevant to you. Just like with paper mail you don't have to think about how your letter goes through multiple stations vehicles

and roads, you only have to specify the address and the name of the recipient. Everything else is taken care of and is hidden from you.

So when the client sends the request via the pipe, it thinks about the pipe as a magical portal that goes directly into the server. The server does the same.

Chapter 3

Building a client

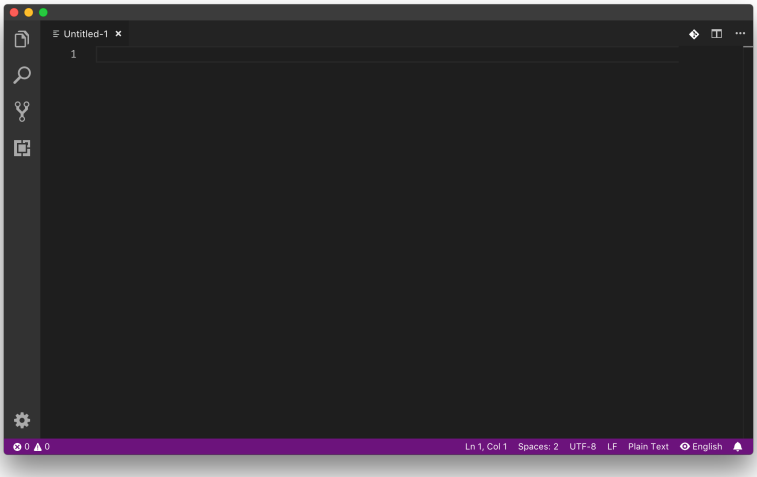
3.1 Get a text editor

Let's try and actually build what we discussed. Since the pipe is taken care of, we only need to focus on the client and the server. I know I said there are two computers in the scenario, nothing stops us from running both the client application and the server application on the same computer. Neither the client nor the server recognizes this. Nor do they care. Web developers often have multiple servers and multiple clients running on the same machine, this is just the way we work.

Before we start, we need to learn a bit about the terminal. If you already have some experience working with command line utilities, feel free to skip the next section.

You also need a text editor for writing code. You probably have one already, like Notepad on Windows or TextEdit on macOS. These might work, but they are painful to use when writing code. It's much better to use a specialized text editor, created for programmers.

I recommend Microsoft VS Code. It's free, it works on any operating system and it works nicely out of the box. So, go ahead, download and install it.



3.1.0.1 VS Code Tips

- If you don't like the default dark style of VS Code, you can change themes by simultaneously hitting `Cmd Shift P` (macOS) or `Ctrl Shift P` (Windows) and typing `Theme`, then hitting Enter and choosing from a list.
- You can increase font size by hitting `Cmd +` and `Cmd -` (macOS) or `Ctrl +` and `Ctrl -` (Windows)

3.2 The terminal

We won't be building a full-blown web HTTP client application. Instead, we will use an existing app capable of communicating with servers via the Internet. This application can establish a connection (i.e. create a portal) and send some text back and forth. It doesn't know anything about HTTP.

Our goal is to use this app and manually do things that web clients do automatically.

This existing app is called `netcat` (often abbreviated to `nc`), and it is not graphical: it doesn't have an icon to double click and it won't show

any nice windows and buttons when you start it. In order to run and interact with `nc`, you have to work with a Terminal.

You've probably seen this already: a (usually) black screen with white text, and lots of lines of seemingly undecipherable stuff. This is how hackers are portrayed in movies. This is where many developers spend their days. We will talk in detail about terminals, command line utilities and Bash shell in future courses and books, but for now – here is a short intro, just enough to get you up to speed.

First computers didn't have graphics. Well, *first* first computers didn't even have screens, but those that had screens didn't have any on-screen graphics like buttons, images, text fields or menus. They looked like glorified typewriters that "talk back": you type something, hit "Enter" and it types something in response.

Modern computers are billions of times faster and more capable, but this sort of typewriter-like textual interaction is still in the core of programming and system administration. Like it or not, you will have to learn to deal with it.

Imagine you have an app that shows the contents of documents. How would it look like? Maybe, it's just a small window with some sort of a file selector and a button that says "Show". Select a file, hit the button and the contents of the file is displayed on the screen.

It is possible to accomplish the same feature **without** windows and buttons.

First, you need to somehow launch the app, without having any icons or menu items. This app has a name – `fileviewer`. This is how you run an app via textual terminal: by typing its name:

```
fileviewer
```

Okay, assuming this works, there are different ways to proceed. First way is hit "Enter", which will just launch the app. It might greet us and tell us what to do next, something like this:

```
Welcome to fileviewer!
Please, enter the path to a file:
```

Or maybe this `fileviewer` app works differently (its developer decided so), and says:

```
Error! No file specified.
```

Which means it probably expected us to type the address of a file on the same line with the app name, like so:

```
fileviewer /Users/sam/Desktop/file.txt
```

The way a text-based app works is defined by its creator, the developer. There are no strict rules, but there are certain conventions and expectations. Many command line apps work like the latter example: they expect some "parameters" to be specified after the their name, on the same line.

The `nc` app works like this: it expects you to enter some information right after its name and then hit Enter.

3.2.0.1 Getting `nc`

- If you're on Windows, please follow our guide on setting up a good programming environment. Then, use chocolatey to install nc by typing `chocolatey install netcat`
- If you're on macOS, you already have `nc` installed, since it comes built in.

3.3 HTTP request with netcat

We are now ready to use `nc`.

Launch a terminal:

On Windows (assuming you've completed our guide):

1. Press `WinKey+R`
2. Type `cmder`
3. Press `Enter`

On macOS:

1. Launch Spotlight (`Cmd+Space` by default)
2. Type `Terminal.app`
3. Press `Enter`

Now, type

```
nc 127.0.0.1 8080
```

We've put three things separated by spaces:

1. The name of the app.
2. First parameter that `nc` requires: the address of the machine we want to connect to.
3. Second parameter that `nc` requires: the port.

`127.0.0.1` is a special IP-address that corresponds to "this computer". It's like a pointer that always points to the current place: if I run a program on my laptop, then `127.0.0.1` means "my laptop". If I run the same program on my mom's PC, `127.0.0.1` will mean "my mom's PC". We use this address here because we will run both the client app and the server app on the same physical computer.

I know we talked about things as if the client and the server must be located on different computers, but hey, these are just apps, and we can certainly launch multiple apps at the same time on a single machine.

The second parameter for `nc` is a **port**. There can be multiple different server applications running on a single machine. In addition to a web server, we could run a database server, a chat server, an email server and any number of other usual or unusual servers. All of them communicate with the world via the Internet, but how can we make sure that HTTP requests go to the web server, but not to the database server?

This is similar to having an apartment building: if a postal courier comes with a package for Mr Jekyll, they **must** know the apartment number. The computer is like that building, and different apps are running behind different doors or ports. In our example above we put `8080` as port number. Our assumption is that the web-server is indeed running behind door `8080`.

By the way, `port` literally means "door" in some languages.

When you hit `Enter` after typing `nc 127.0.0.1 8080`, nothing will happen. Because there is no app currently running on the same machine listening for connecting on port `8080`.

XIt's time to build this app.

Chapter 4

Building a server

4.1 Get Python

We will use a programming language called Python to create our server. Keep in mind that we're jumping around and skipping important things right now. The idea is to launch something functioning as soon as possible, and then, having that sweet satisfaction as our motivator, look back and understand the things we've skipped over.

First, download and install Python. I know this might be confusing, because you probably don't know what does it mean to "download Python": download a language? Is it an app then? Or… what is it?

We'll talk about programming languages in the future books and courses, but for now you can think of it this way: we're downloading a program that is capable of reading code and creating applications from it.

Go to Python downloads page and download the version for your operating system (Windows or macOS). Make sure to download the latest version — 3.7.1 or higher. Launch the downloaded file to install Python.

Important note to Windows users: make sure to check "Add Python 3.7 to PATH" checkbox on the first screen of the installer!

Now go back to the terminal and type

- On macOS: `python3 --version`

- On Windows: `python --version`

and hit Enter. You should see `Python 3.7.1` printed on the screen. Success! We have a functioning **Python interpreter** on our computer! It will **interpret** code we write and do things.

4.2 DIY web server

Here comes the first challenge. On one hand, you want to make something that works as soon as possible, prove yourself you can really create a computer application from nothing. On the other hand, even a simple HTTP server, which consists of just 4 lines of code, is a pretty complex program. Each line includes tens and tens of different concepts.

One way to go — a classic, "proper" way — is to start with the basics of Python and make our way up the ladder of difficulty. We'd need to learn about variables, modules, importing, TCP, handlers, `with` -statements and command line arguments. It's quite a list, and each concept requires you to at least gave some idea about a dozen more. It'll take us a few days or even weeks!

The rabbit's hole is very, very deep.

Another way to go is to just give you those 4 lines to copy and paste and hope it works. You'll have no idea what's happening, but the HTTP server will probably be launched successfully.

I believe there's a third way. I will indeed give you 5 lines of code to copy and paste. I will also describe them, but won't go deep. But this is a loan, not a gift. You'll have to pay back!

In the future courses and books we will look back and cover the missing parts. I call this a **"learning credit card"**. We'll always jump ahead and do something cool, but then at some point we'll come back and "pay the debt". As long as we're disciplines and don't take too many "knowledge loans", we'll be fine and it'll be fun!

So, here comes the first loan :-) Go to your code editor (VS Code), click File -> New File and create a new file called `server.py`. It doesn't matter where, but it'll be easier if you create a folder somewhere first. Call the folder as you wish, e.g. "webserver".

Copy and paste the following code into the file:

```
import http.server
import socketserver

with socketserver.TCPServer(("", 8080), http.server.SimpleHTTPRequestH
    httpd.serve_forever()
```

This is how it looks like in my VS Code:

The first two lines **import** some code from two other files that come build-in with Python itself. This term is used in many programming languages and it means what you think it means: some useful code is stored somewhere else, and you want to "get" it here, in your file. `import` tells the Python interpreter to "take" it, essentially, copy and paste into your program.

The final two lines use some that imported code to create a new server, specify its port 8080 and makes sure that requests are processed with `SimpleHTTPRequestHandler`: a special function that comes with Python which knows how to accept requests and create responses.

Save the file, then go back to the terminal and navigate to that folder.

On macOS: the easiest way is to enter `cd` in the Terminal, hit Spacebar and then drag-n-drop the folder into the Terminal. This will save you from typing the full path by hand.

But if you prefer, you can do it by hand. For example, if your folder is called `server` and is located on your desktop, then type `cd /Users/YOUR_USERNAME_HERE/Desktop/server`. Don't forget to change `YOUR_USERNAME_HERE` with your actual username.

(`cd` stands for "change directory", by the way).

On Windows: assuming your folder is saved on the Desktop, in the Terminal type `cd C:\Users\YOUR_USERNAME_HERE\Desktop\server`. Don't forget to change `YOUR_USERNAME_HERE` with your actual username.

Now run the server by typing:

- **On macOS:** `python3 server.py`
- **On Windows:** `python server.py`

(Windows might ask you to "Allow access" since your web server is a new application that tries to use the network; press "Allow access").

If everything is okay, the Terminal will kind of freeze and stop on a new line.

4.3 Back to the client

Remember, our client couldn't do anything because we didn't have a server running. Now we do! Open a new Terminal window (on macOS hit `Command + N`, on Windows just run another `cmd`), and repeat the same command:

```
nc 127.0.0.1 8080
```

This time the `nc` program will not quit and the cursor will move to a new line. Now the connection is established, we have a portal running! It's time to write that GET request. Type it:

```
GET /article.html HTTP/1.1
Host: coolsite.com
```

You have to hit Enter after `HTTP/1.1`. Then, after the second line, hit Enter two more times. You should get a response from the server similar to this:

```
HTTP/1.0 404 File not found
Server: SimpleHTTP/0.6 Python/3.7.1
Date: Mon, 26 Nov 2018 20:44:56 GMT
Connection: close
Content-Type: text/html;charset=utf-8
Content-Length: 469

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
        "http://www.w3.org/TR/html4/strict.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html;charset=utf
        <title>Error response</title>
    </head>
    <body>
        <h1>Error response</h1>
        <p>Error code: 404</p>
```

```
        <p>Message: File not found.</p>
        <p>Error code explanation: HTTPStatus.NOT_FOUND - Nothing matc
    </body>
</html>
```

Woah! Read the first line and you should get an idea of what's happened: the thing you've requested is not found. Do you remember creating a file "article.html"? Me neither. Our server runs fine, but a file with that name doesn't exist.

The server we've built has access to the same folder it's located in, and it "serves" the files from it. When the request asks for "article.html", the server looks into the folder and tries to find the requested file. Let's create that file and save it next to `server.py`. Go to VS Code and create a new file, put the following code in it:

```
<!doctype html>
<html lang=en>
  <body>
    <h1>Article title</h1>
    <p>Some very important text.</p>
  </body>
</html>
```

Once again, we're jumping ahead and writing some HTML without understanding what it means. We'll talk about HTML in the future, of course. For now, just save that file as `article.html` in the same folder where your server is located.

Now repeat the GET request.

This is what you should get back from the server:

```
HTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.7.1
Date: Mon, 26 Nov 2018 21:07:37 GMT
Content-type: text/html
Content-Length: 122
Last-Modified: Mon, 26 Nov 2018 21:07:25 GMT

<!doctype html>
<html lang=en>
  <body>
```

```
    <h1>Article title</h1>
    <p>Some very important text.</p>
  </body>
</html>
```

Yes! Even though it says "HTTP/1.0" instead of "1.1", the important part is that the request was successful and we get a "200 OK" status code. This means everything is okay. After a few lines of technical information (these are called "headers"), we see the contents of that HTML file.

If this was your browser, and not `nc`, at this point it would've saved that HTML file somewhere and opened for you to see.

Well, that's it! The fundamental layer is ready to be built upon! The next steps are to learn about further layers to explore the wonderfully complex world of modern web development.