Contents

1 What the heck is a web server any way? 1

1.1 Parts of a Web Server: . . . 1

1.1.1 Navigating the World of Protocols: A Quick Overview . . . 2

1.1.2 The Relationship Between HTTP and TCP: Ensuring Reliable Web Communication . . . 3

1.1.3 1. Data Integrity and Order . . . 3

1.1.4 2. Acknowledgment Mechanism . . . 4

1.1.5 3. Complex Interactions . . . 4

1.1.6 4. Transmission Overhead . . . 4

1.2 Asking and Getting: How Web Servers Respond to Your Requests . . . 4

1.2.1 The Request: . . . 4

1.2.1.1 Your Request: . . . 4

1.2.1.2 Finding the Address: . . . 5

1.2.1.3 Resolving the Address: . . . 5

1.2.2 The Response: . . . 5

1.2.2.1 Return Address: . . . 5

1.2.2.2 Sending the Request: . . . 5

1.2.2.3 Preparing the Content: . . . 5

1.2.2.4 Sending the Response: . . . 5

1.2.2.5 Enjoying the Content: . . . 5

2 Your first `node.js` program 7

2.1 What exactly is node or nodejs? . . . 7

2.2 Your first node.js program . . . 8

2.2.1 How does `console.log()` work in Node.js? . . . 9

2.2.2 The `process` **Object**: . . . 10

2.2.3 The `stdout` **property of the** `process` **object**: . . . 11

3 Working with files 15

3.0.1 What will the logging library do . . . 15

3.0.2 How do you work with files anyway? . . . 15
3.0.3 Let's get back to `files` . . . 21
3.0.4 A little more about file descriptors . . . 22
3.0.5 Creating our first file . . . 22
3.0.6 `path` argument . . . 24
3.0.7 `flag` argument . . . 25
3.0.8 `mode` argument . . . 26
3.1 Reading from a file . . . 30
3.2 A small primer on `for..of` and `for await..of` in javascript . . . 34
3.2.1 `for..of` . . . 34
3.2.2 `for await..of` . . . 35
3.3 Reading the `json` file . . . 37
3.4 Buffers . . . 38
3.4.1 Parsing the `json` file . . . 39

4 `logtar` our own logging library **43**
4.1 Initializing a new project . . . 43
4.2 A little about `SemVer` . . . 44
4.3 Creating a `LogLevel` class . . . 45
4.4 The `Logger` class . . . 48
4.4.1 Encapsulation with `private` fields . . . 49
4.5 The `LogConfig` class . . . 53
4.6 Design patterns . . . 55
4.6.1 The `Builder` pattern . . . 56
4.7 Using `builder` pattern with the `LogConfig` class . . . 58
4.8 jsdoc comments . . . 60
4.9 The `RollingConfig` class . . . 62
4.9.1 The `RollingSizeOptions` class . . . 62
4.9.2 The `RollingTimeOptions` class . . . 63
4.10 Finishing up the `RollingConfig` class . . . 64
4.10.1 Let's recap . . . 65
4.11 Adding more useful methods in the `LogConfig` class . . . 66
4.11.1 Why `readFileSync` ? . . . 69
4.12 Refactoring the code . . . 69
4.12.1 **The Need for Refactoring** . . . 70
4.12.2 **Creating Separate Files** . . . 70
4.12.2.1 Explanation . . . 71
4.12.3 The `index.js` file . . . 71

4.12.4 The `lib/logtar.js` file . . . 72
4.12.5 The `lib/logger.js` file . . . 72
4.12.6 The `lib/config/log-config.js` file . . . 73
4.12.7 The `lib/config/rolling-config.js` file . . . 77
4.12.8 The `lib/utils/log-level.js` file . . . 79
4.12.9 The `lib/utils/rolling-options.js` class . . . 80
4.13 Writing logs . . . 82
4.13.1 1. Re-using the File Handle . . . 82
4.13.2 2. Log Rotation . . . 82
4.13.3 3. Asynchronous Logging . . . 83
4.13.4 4. Getting Caller Information (Module and Line Number) . . . 83
4.13.5 Testing our current API . . . 84
4.13.6 Implementing logging methods . . . 87
4.13.7 DRY (Don't Repeat Yourself) . . . 88
4.13.8 The `log` method . . . 89
4.13.9 Considering the `log_level` member variable . . . 91
4.13.10 Writing to a file . . . 92
4.13.10.1 A small primer on Regular Expressions . . . 93
4.13.10.2 Testing the log file creation . . . 95
4.13.11 Another gotcha . . . 96
4.13.12 Logs directory configuration . . . 97
4.13.12.1 Our first script . . . 98
4.13.13 The `require` object . . . 99
4.13.14 Adding a new helper to create log directory . . . 99
4.13.15 Updating the `init` method . . . 101
4.13.16 Completing the `log` method . . . 101
4.14 Capturing metadata . . . 103
4.14.1 What is a Stack? . . . 104
4.14.2 Examples of Stacks . . . 104
4.14.3 The Call Stack . . . 105
4.14.4 Getting the stack info . . . 107
4.14.5 Getting the `callee` name and the line number . . . 108
4.14.6 A more ergonomic way . . . 111
4.14.7 Using the `get_caller_info` function . . . 113
4.15 A small intro to `async` vs `sync` . . . 114
4.15.1 The Balance between Opposites . . . 115
4.15.2 Mixing Asynchronous and Synchronous Code . . . 115
4.15.3 Faster I/O out of the box . . . 115

4.15.4 Blocking Code . . . 116
4.15.5 Concurrency . . . 117
4.16 Adding Rolling File Support . . . 117
4.16.1 Rolling features . . . 117
4.16.1.1 `#time_threshold` . . . 118
4.16.1.2 `#size_threshold` . . . 118
4.16.2 The `rolling_check()` method . . . 118
4.16.3 `file_handle.stat()` . . . 119
4.16.4 Calling the `rolling_check` method . . . 121
4.16.5 A big gotcha! . . . 123
4.17 Stack traces across `await` points . . . 124
4.17.0.1 The culprit . . . 124
4.17.1 Testing the new Log file creation . . . 127

5 HTTP Deep dive **131**
5.1 A small web server . . . 132
5.1.1 Starting our web server . . . 133
5.1.2 Testing our web server . . . 134
5.1.3 Testing with `cURL` . . . 134
5.2 HTTP Verbs, Versioning and the benefits of `HTTP/1.1` . . . 137
5.2.1 `GET` - Retrieve data . . . 138
5.2.2 `POST` - Create something . . . 139
5.2.3 `PUT` - Replace or Create . . . 140
5.2.4 `HEAD` - Retrieve Metadata . . . 140
5.2.5 `DELETE` - Remove from existence . . . 141
5.2.6 `PATCH` - Partial updates . . . 141
5.2.7 A small recap . . . 141
5.2.8 The `/` path . . . 142
5.2.9 `HTTP/0.9` . . . 143
5.2.10 `HTTP/1.0` . . . 143
5.2.10.1 Introduction of the HTTP Header . . . 143
5.2.10.2 Versioning . . . 144
5.2.10.3 Status Codes . . . 144
5.2.10.4 Content-Type Header . . . 144
5.2.10.5 New Methods . . . 144
5.2.11 `HTTP/1.1` . . . 145
5.2.11.1 Persistent Connections . . . 145
5.2.11.2 Pipelining . . . 146

5.2.11.3 Chunked Transfer Encoding . . . 146
5.2.11.4 The `Host` header . . . 146
5.2.11.5 Caching improvements . . . 147
5.2.11.6 Range Requests . . . 148
5.2.11.7 New Methods: `PUT` , `DELETE` , `CONNECT` , `OPTIONS` , `TRACE` . . . 149
5.3 User agents . . . 152
5.3.1 `User-Agent` can be weird . . . 152
5.4 MIME Type and `Content-Type` . . . 154
5.4.1 Understanding the `Accept` Header . . . 154
5.4.1.1 Breaking Down the Line . . . 154
5.4.1.2 Why the Wildcard? . . . 154
5.4.1.3 Server Response . . . 154
5.4.2 Mime Type . . . 155
5.4.3 Anatomy of a MIME type . . . 155
5.4.4 But why the wildcard `*/*` ? . . . 156
5.4.5 The `Content-Type` header . . . 156
5.4.5.1 `Content-Type` on request header . . . 156
5.4.5.2 `Content-Type` on response header . . . 157
5.4.6 The `charset=UTF-8` : character encoding . . . 157
5.4.6.1 Universal Character Encoding . . . 157
5.5 Headers . . . 158
5.5.1 Header Name . . . 159
5.5.2 Colon (`:`) . . . 159
5.5.3 Header Value . . . 159
5.5.4 Whitespace . . . 159
5.5.5 Custom `X-` based headers . . . 159
5.6 Request headers . . . 160
5.6.1 Accept . . . 160
5.6.2 Referer . . . 162
5.6.3 Authorization . . . 163
5.6.4 Cookie . . . 163
5.6.5 Host . . . 164
5.6.6 Content-Type (request) . . . 164
5.7 Response Headers . . . 165
5.7.1 Content-Type (response) . . . 165
5.7.2 Cache-Control . . . 166
5.7.2.1 How Caches Work: . . . 166
5.7.2.2 Cache-Control Directives: . . . 167

5.7.3 Set-Cookie . . . 168
5.8 Response and Status Codes . . . 169
5.8.1 `Connection: close` in action . . . 172
5.8.2 Status Codes . . . 178

6 `Velocy` - Our backend framework 181
6.1 Why Velocy? . . . 182
6.2 What is a backend framework/library anyway? . . . 182
6.3 Core features of our backend framework . . . 182
6.3.1 Routing and URL Handling: . . . 182
6.3.2 Middlewares . . . 185
6.3.3 Building our own database . . . 185
6.3.3.1 Data Storage and Retrieval: . . . 185
6.3.3.2 Indexing: . . . 185
6.3.3.3 CRUD Operations: . . . 186
6.3.3.4 Querying: . . . 186
6.3.4 Caching . . . 186
6.3.5 Rate limiting . . . 186
6.3.6 Some other features that we will be implementing . . . 186
6.4 A basic `Router` implementation . . . 187
6.4.1 A Toy Router . . . 189
6.4.2 `Transfer-Encoding: chunked` . . . 193
6.4.3 Chunks, oh no! . . . 194
6.4.4 Specifying `Content-Length` . . . 195
6.4.5 Code reusability . . . 196
6.4.5.1 Separation of Concerns . . . 198
6.5 The `Router` class . . . 201
6.5.1 Using `Router` with an HTTP server . . . 204
6.6 `this` is not good . . . 207
6.6.0.1 Using `.bind()` . . . 208
6.6.0.2 Using `Arrow` function . . . 208
6.6.1 Lexical Context . . . 209
6.6.2 Arrow functions are not free . . . 211
6.6.3 Why should we care about memory? . . . 212
6.6.4 Testing the updated code . . . 213
6.7 Improving the `Router` API . . . 217
6.8 The Need for a `Trie` . . . 221
6.8.1 What is a `Trie` anyway? . . . 223

6.9 Exercise 1 - Implementing a `Trie` . . . 224
6.9.1 Root Node . . . 225
6.9.2 End of the word . . . 226
6.9.3 Challenge 1: Basic Trie with `insert` Method . . . 227
6.9.3.1 Requirements . . . 227
6.9.3.2 More details . . . 228
6.9.3.3 Solution . . . 229
6.9.3.4 Using a `for` loop instead of recursion . . . 232
6.9.4 Challenge 2: Implement `search` method . . . 233
6.9.4.1 Requirements . . . 233
6.9.4.2 More details . . . 233
6.9.4.3 Hints . . . 234
6.9.4.4 Solution . . . 234
6.10 Exercise 2 - Implementing our Trie based `Router` . . . 235
6.10.1 Challenge 1: Implementing the `addRoute` method . . . 235
6.10.1.1 Requirements . . . 235
6.10.1.2 More details . . . 236
6.10.1.3 Hints . . . 238
6.10.1.4 Solution . . . 238
6.10.1.5 Explanation . . . 241
6.10.2 Challenge 2: Implementing the `findRoute` method . . . 243
6.10.2.1 Requirements . . . 243
6.10.2.2 More details . . . 244
6.10.2.3 Starting Boilerplate . . . 245
6.10.2.4 Hints . . . 245
6.10.2.5 Solution . . . 245
6.10.2.6 Explanation . . . 246
6.11 Exercise 3 - Adding `HTTP` method support . . . 249
6.11.1 Requirements . . . 249
6.11.2 More details . . . 249
6.11.3 Example . . . 249
6.11.4 Hints . . . 250
6.11.5 Solution . . . 250
6.12 Adding HTTP methods to the Router . . . 254
6.12.1 Update the `TrieRouter` class . . . 254
6.12.1.1 Explanation . . . 255
6.13 Exercise 4 - Implementing Dynamic Routing . . . 256
6.13.1 Why Dynamic Routing? . . . 256

6.13.1.1 Flexibility 256
6.13.1.2 Better User Experience 257
6.13.1.3 Better Developer Experience 257
6.13.1.4 Better SEO 258
6.13.2 Anatomy of a dynamic route 258
6.13.3 Challenge: Enhance the `TrieRouter` Class to Support Dynamic Routing 258
6.13.3.1 Requirements 258
6.13.3.2 More Details 259
6.13.3.3 Solution 259
6.13.3.4 Explanation 263
6.13.3.5 Testing the implementation 266
6.13.4 Summary 268
6.14 Running Our Server 268
6.14.1 Refactoring the `TrieRouter` class 268
6.14.2 Type Aliases 270
6.14.3 The `run` function 271
6.15 Building our first web-server 274
6.15.1 More refactoring 274
6.15.1.1 `lib/router.js` 274
6.15.1.2 `lib/constants.js` 280
6.15.1.3 `lib/index.js` 280
6.15.1.4 `globals.js` 281
6.15.2 Your first web server 281

Chapter 1

What the heck is a web server any way?

Note: This chapter gives you a brief overview of the basics of web servers and HTTP. We'll learn about HTTP in more details in an upcoming chapter.

If you do not wish to read about the basics of web/http, you can safely jump to the coding section.

Before diving straight into writing JavaScript code to create web servers, it's essential to grasp the fundamental concepts that are the basic building blocks of web server. Web servers are like the traffic controllers of the internet. They manage requests from users (like you!) and send back the right information. But what makes up a web server, and how does it even work? Let's break it down into simple terms.

1.1 Parts of a Web Server:

Web servers are like friendly translators that help computers understand each other. Imagine if you and a friend speak different languages. To have a conversation, you'd need a common language that both of you understand. In the same way, web servers and computers need a common set of rules to talk to each other effectively. These rules are called protocols, which are like languages specifically designed for computers. When you type a website address in your browser and hit "Enter," your computer sends a message to the web server. This message follows the web's language rules, known as the HTTP protocol (Hypertext Transfer Protocol).

HTTP is like a code that tells the web server what you want (e.g., a webpage or an image) and how to respond. The web server reads your message, understands it because it knows the HTTP protocol too, and then sends back the information you requested using the same rules. This information could be a webpage, a picture, or any other content.

Just like you need to speak the same language to have a successful conversation with someone, computers and web servers need to use the same protocol to communicate effectively. This way, they can understand each other's requests and provide the right responses, allowing you to enjoy the content you're looking for on the internet. HTTP establishes a standardised set of rules for how your computer's request (like asking for a webpage) should be structured and how the server's response (the webpage itself) should be formatted. This ensures seamless communication between different devices, regardless of their underlying technologies.

Think of HTTP as a detailed script for a play. It outlines every step, from introducing characters (your request) to their dialogues (data transmission) and the grand finale (the server's response). This structured script eliminates misunderstandings and ensures that both sides know what to expect at each stage of the conversation.

But the protocol game isn't limited to HTTP. The web's secure version, HTTPS (Hypertext Transfer Protocol Secure), adds an extra layer of protection through encryption. This way, even if someone tries to eavesdrop on your conversation, they'll only hear garbled nonsense.

Protocols extend beyond web browsing too. Email, file sharing, and even the way your phone connects to Wi-Fi rely on various protocols to ensure reliable and efficient communication. Each protocol serves a specific purpose, just like different languages for different scenarios in real life.

1.1.1 Navigating the World of Protocols: A Quick Overview

Like I explained above, to re-iterate - Protocols are like the rules that enable devices to communicate effectively on the internet. They define how data is formatted, transmitted, and understood by different systems. Just as people follow social etiquette during conversations, devices follow protocols to ensure smooth communication. Here's a glimpse into some major types of protocols:

- **TCP/IP (Transmission Control Protocol/Internet Protocol):** a set of rules for exchanging data over a network.
- **HTTP (Hypertext Transfer Protocol):** a protocol for transmitting data between a web server and a web client.
- **HTTPS (Hypertext Transfer Protocol Secure):** an extension of HTTP that encrypts data in transit.
- **UDP (User Datagram Protocol):** a protocol for transmitting data between networked devices without requiring a connection or reliability guarantees.
- **FTP (File Transfer Protocol):** a protocol for transferring files between computers on a network.
- **SMTP (Simple Mail Transfer Protocol):** a protocol for sending email messages between servers.
- **POP3 (Post Office Protocol 3) and IMAP (Internet Message Access Protocol):** protocols for retrieving email messages from a server.
- **DNS (Domain Name System):** a protocol for translating domain names into IP addresses.
- **DHCP (Dynamic Host Configuration Protocol):** a protocol for automatically assigning IP addresses

to devices on a network.

In order to become a proficient backend engineer, it is important to have a solid understanding of different networking protocols. While HTTP(s) is the main focus of this guide, having knowledge of other protocols such as FTP, SMTP, and DNS can prove beneficial in the long run. FTP (File Transfer Protocol) is commonly used for transferring files between servers, while SMTP (Simple Mail Transfer Protocol) is used for sending emails. DNS (Domain Name System) is responsible for translating domain names into IP addresses.

If you're programming game servers, it's important to have a solid understanding of UDP. UDP is faster but less reliable than TCP, making it ideal for applications that can tolerate occasional data loss, such as video streaming or online gaming. Unlike TCP, UDP is a ”fire and forget” protocol, meaning data is sent without any error-checking or acknowledgment mechanisms.

1.1.2 The Relationship Between HTTP and TCP: Ensuring Reliable Web Communication

HTTP (Hypertext Transfer Protocol) and TCP (Transmission Control Protocol) form a strong partnership when it comes to web communication. The reason HTTP prefers TCP lies in the very nature of their roles and responsibilities within the world of networking.

1.1.3 1. Data Integrity and Order

HTTP is used to send web content, like web pages, images, and videos, from a server to a user's browser. Imagine if you requested a webpage and the images were missing or the text was scrambled. That wouldn't be a good experience, right? HTTP has to make sure that the data is delivered correctly and in order.

TCP helps with this. It was designed to make sure that data is delivered in the right order and without errors. TCP breaks up the data into small pieces called packets, sends them to the destination, and makes sure they arrive in the correct order. If any packet is lost during the process, TCP asks for it to be sent again. This is important for web pages because everything needs to be presented in a way that makes sense.

A packet is a small unit of data that is sent over a network. In the context of web communication, TCP breaks up the data into small pieces called packets, sends them to the destination, and makes sure they arrive in the correct order. If any packet is lost during the process, TCP asks for it to be sent again.

1.1.4 2. Acknowledgment Mechanism

HTTP is a way to request a webpage, and the server sends back the content you asked for. To make sure the data is received correctly, an acknowledgment mechanism is needed.

TCP provides this mechanism by waiting for your browser to confirm that it has received each packet of data sent from the server. If your browser does not confirm, TCP sends the packet again, so that both the server and browser can be sure that the data is being received properly.

1.1.5 3. Complex Interactions

HTTP transactions involve multiple steps, like requesting a webpage, receiving the HTML structure, fetching linked assets (images, stylesheets), and more. These interactions require precise data handling and sequencing.

TCP works well with HTTP for handling complex interactions. TCP's mechanisms guarantee that every piece of data reaches its intended destination and fits into the bigger interaction. For instance, when you visit a webpage, your browser makes several HTTP requests for different assets. TCP helps ensure that these requests and responses occur in an orderly and dependable manner.

1.1.6 4. Transmission Overhead

TCP adds some extra information to every message to make sure it gets to its destination without errors. This extra information includes acknowledgments, sequence numbers, and error-checking. Even though it adds a little more data to every message, it's still worth it because it helps make sure the data is accurate and in the right order. This is especially important when communicating over the web.

1.2 Asking and Getting: How Web Servers Respond to Your Requests

Imagine you're at home, sitting in front of your computer, and you decide to visit a website, let's say ”example.com.” This simple action initiates a series of events that highlight the ”Asking and Getting” process in web communication.

1.2.1 The Request:

1.2.1.1 Your Request:

You type ”example.com” into your browser's address bar and hit Enter. This is like you telling your computer, ”Hey, I want to see what's on this website!”

1.2.1.2 Finding the Address:

Your computer knows the basics of websites, but it needs the exact address of "example.com" to connect to it. So, it reaches out to a special helper called a DNS resolver.

1.2.1.3 Resolving the Address:

The DNS resolver is like a digital address book. It takes "example.com" and looks up the actual IP address associated with it. This IP address is like the specific coordinates of the website's location on the internet.

> A website URL like https://google.com also be referred to as a **domain name**

1.2.2 The Response:

1.2.2.1 Return Address:

The DNS resolver finds the IP address linked to "example.com" and sends it back to your computer. It's like the DNS resolver telling your computer, "The website is located at this IP address."

1.2.2.2 Sending the Request:

Now that your computer knows the IP address, it can send a request to the web server that holds the website's content. This request includes the IP address and a message saying, "Hey, can you please give me the content of your website?"

1.2.2.3 Preparing the Content:

The web server receives your request and understands that you want to see the content of "example.com." It then gathers the necessary files – HTML, images, stylesheets, scripts – to create the webpage.

1.2.2.4 Sending the Response:

The web server packages the content into a response and sends it back to your computer through the internet. It's like the server sending a digital package to your doorstep.

1.2.2.5 Enjoying the Content:

Your computer receives the response from the web server. Your browser interprets the HTML, displays images, and applies styles, creating a complete webpage. This is what you see on your screen – the final result of your request.

A quick disclaimer: our learning approach will prioritize clarity and thoroughness. I will introduce a topic, break it down, and if we come across any unfamiliar concepts, we will explore them until everything is fully understood.

Chapter 2

Your first `node.js` program

The following section assumes that you have nodejs installed locally and are ready to follow along. You can check whether you have nodejs installed by running this command on your terminal -

```
node --version

# Outputs
# v18.17.0
```

If you see a `Command not found` error, that means you do not have nodejs installed. Follow the instructions here to download and install it.

2.1 What exactly is node or nodejs?

From the official website -

> Node.js® is an open-source, cross-platform JavaScript runtime environment.

What does a "runtime" mean?

Simply put, when you write code in a programming language like JavaScript, you need something to execute that code. For compiled languages like C++ or Rust, you use a compiler. The runtime environment takes care of executing the code, ensuring that it works well with the computer's hardware and other software components.

For Node.js, being a JavaScript runtime environment means it has everything needed to execute JavaScript code ***outside of a web browser***. It includes the V8 JavaScript engine (which compiles and executes JavaScript

code), libraries, APIs for file, network, and other system-related tasks, and an event loop for asynchronous, non-blocking operations.

> We'll discuss what exactly an event loop means, and implement our own version of event loop to understand how it works, later on in the guide.

2.2 Your first node.js program

Let's begin by writing some code. Let's create a new folder, and name it whatever you wish. I've named it `intro-to-node`. Inside it, create a new file `index.js` and add the following content inside it.

```
// Write the string `Let's learn Nodejs` to the standard output.
process.stdout.write("Let's learn Node.js");
```

To execute the code, open your terminal and `cd` into the folder containing the `index.js` file, and run `node index.js` or `node index`. You may alternatively run the command by specifying the relative or absolute path of the `index.js` file -

```
node ../Code/intro-to-node/index.js

#or

node /Users/ishtmeet/Code/intro-to-node/index.js
```

This should output

```
Let's learn Node.js
```

> You might also see a trailing `%` at the end due to the absence of a newline character (`\n`) at the end of the string you're writing to the standard output (stdout). You can modify the code as `process.stdout.write("Let's learn Node.js\n");` to get rid of that trailing modulo.

What is the code above doing?

There's too much going on in the code above, and I simply chose it over `console.log()` to explain a huge difference between the Javascript API and the Nodejs API.

JavaScript and Node.js are closely related, but they serve different purposes and have different environments, which leads to some differences in their APIs (Application Programming Interfaces).

JavaScript was created to make web pages more interactive and dynamic. It was meant for creating user interfaces and responding to user actions on the client side, **inside the browser**. However, as web applications became more complex, relying only on client-side JavaScript was not enough. This led to the development of Node.js, which allows JavaScript to be executed on the server side. Node.js extends JavaScript's capabilities, introducing APIs for file system operations, network communication, creating web servers, and more. This means developers can use one programming language throughout the entire web application stack, making development simpler.

So let's jump back to the code above, and understand why did I use `process.stdout.write` instead of `console.log`.

Simply put, `console.log` is a method that outputs a message to the web console or the browser console. However, Node.js does not run on the web, which means it does not recognize what a console is.

But if you change your code inside `index.js` to this

```
console.log("Let's learn Node.js");

// Outputs → Let's learn Node.js
```

It works. However, isn't it the case that I just mentioned Node.js being unfamiliar with the concept of a browser console? Indeed, that's correct. However, Node.js has made it easier for developers who are only used to working with JavaScript in a web context. It has included all the important features of browser-based JavaScript in its framework.

Expanding upon this topic, it's important to understand that Node.js, despite its roots in server-side development, strives to bridge the gap between traditional web development and server-side scripting. By incorporating features commonly associated with browser-based JavaScript, Node.js has made it more accessible for developers who are already well-versed in the language but might be new to server-side programming.

2.2.1 How does `console.log()` work in Node.js?

The `node:console` module offers a wrapper around the standard console functionalities that javascript provides. This wrapper aims to provide a consistent and familiar interface for logging and interacting with the Node.js environment, just as developers would in a web browser's developer console.

The module exports two specific components:

- A `Console` class with methods like `console.log()`, `console.error()`, and `console.warn()`. These can be used to write to any Node.js **stream**.
- A global `console` instance that is set up to write to `process.stdout` and `process.stderr`.

(Note that `Console` is not `console` (lowercase). `console` is a special instance of `Console`)

> You can use the global `console` without having to call `require('node:console')` or `require('console')`. This global availability is a feature provided by the Node.js runtime environment. When your Node.js application starts running, certain objects and modules are automatically available in the global scope without the need for explicit importing.
> Here are some of the examples of globally available objects/modules in Node.js - `console`, `setTimeout`, `setInterval`, `__dirname`, `__filename`, `process`, `module`, `Buffer`, `exports`, and the `global` object.

As I mentioned earlier, Node.js provides the global `console` instance to output text to `process.stdout` and `process.stderr`. So if you're writing this

```
console.log("Something");
```

the above code is just an abstraction of the code below.

```
process.stdout.write("Something\n");
```

However, even after reading this, the code above may still be confusing. You may not yet be familiar with the `process` object, or with `stdout` and `stderr`.

2.2.2 The `process` Object:

The **`process`** object in Node.js tells you about the environment where the Node.js app is running. It has various properties, methods, and event listeners to help you work with the process and access info about the runtime environment.

These are some of the useful properties and functions that are provided by the `process` object. Copy paste the code below and paste it inside your `index.js` file. Try to execute it, using `node path/to/index/file`.

```
console.log(process.version);
// v18.17.0

console.log(process.platform);
// darwin

console.log(process.uptime());
// 0.023285791

console.log(process.cpuUsage());
```

```
// { user: 31466, system: 6772 }

console.log(process.resourceUsage().systemCPUTime);
// 6865

console.log(process.memoryUsage());
// {
//  rss: 39239680,
//  heapTotal: 6406144,
//  heapUsed: 5388408,
//  external: 425804,
//  arrayBuffers: 17694
// }

console.log(process.cwd());
// /Users/ishtmeet/Code/intro-to-node

console.log(process.title);
// node

console.log(process.argv);
// [
//  '/usr/local/bin/node',
//  '/Users/ishtmeet/Code/intro-to-node/index.js'
// ]

console.log(process.pid);
// 39328
```

> We will discuss most of these properties/functions further down the line when we talk about implementing our own framework.

2.2.3 The `stdout` property of the `process` object:

In Node.js, the `stdout` property is a part of the `process` object. This property represents the standard output ***stream***, which is used for writing data to the console or other output destinations. Anything written

to the `stdout` stream is displayed in the console when you run your program.

Now you may ask, what is a `stream`?

Streams are used in programming to efficiently handle data flow, especially when working with large datasets or network communication. A stream is a sequence of data elements that is made available over time. Instead of loading all the data into memory, streams allow you to process and transmit data in smaller, more manageable pieces.

Streams can also be classified as input streams and output streams. Input streams are used to take in data from a source, while output streams are used to send data to a destination.

Streams have an important advantage of supporting parallelism. Instead of processing data one after the other, streams can process data in parallel and concurrently. This is helpful when working with large datasets because it speeds up processing time significantly.

Node.js provides a comprehensive implementation of streams, which can be categorized into several types:

1. **Readable Streams**: These streams represent a source of data from which you can read. Examples include reading files, reading data from an HTTP request, or even generating data programmatically.
2. **Writable Streams**: Writable streams are destinations where you can write data. Examples include writing data to files, standard output (`stdout`), standard error output (`stderr`) and many more.
3. **Duplex Streams**: Duplex streams represent both a readable and a writable side. This means you can both read from and write to these streams concurrently. An example of a Duplex stream is a TCP socket. It can both receive data from the client and send data back to the client concurrently.
4. **Transform Streams**: These are a specific type of duplex stream that allow you to modify or transform data as it's being read or written. They are often used for data manipulation tasks, like compression or encryption.

> Streams are incredibly versatile and efficient because they work with small chunks of data at a time, which is particularly useful when dealing with data that doesn't fit entirely into memory or when you want to process data in real-time. They also make it possible to start processing data before the entire dataset is available, reducing memory consumption and improving performance.

Now you know what streams are, and what is the standard output (`stdout`), we can simplify the code below.

```
process.stdout.write("Hello from Node.js");
```

We're simply writing to **`stdout`** or the standard output stream which Node.js provides, which means that we're sending data or messages from our program to the console where you see the program's output. The data we write to **`stdout`** is displayed in the order it's written, giving us a way to communicate with developers or users and provide insights into the program's execution in real-time.

Working with `process.stdout` can be rather cumbersome, and in practice, you tend to use it sparingly. Instead, developers frequently opt for the more user-friendly `console.log` method. Instances of code employing `process.stdout` are typically encountered when there's a need for a greater level of control over output formatting or when integrating with more complex logging mechanisms.

> ***Warning***: The ways of the global console object are not always synchronous like the browser APIs they resemble, nor are they always asynchronous like all other Node.js streams. For more information, please see the note on process I/O.

Chapter 3

Working with files

Now that we've covered the basics of logging in Node.js, let's explore a real-world example. Let us understand the low level of files and how to interact with them. After that, we'll build a logging library logtar that writes logs to a log file. It also has a support for tracing and rolling file creation. We'll use this library as the central mechanism of logging for our web framework, that we build further into this guide.

3.0.1 What will the logging library do

- Log messages to a file
- Choose log location, or simply generate a new file
- Support for log levels (debug, info, warning, error, critical)
- Timestamps on log messages
- Customizable log message format
- Automatic log rotation based on file size or time interval
- Support for console output in addition to log files
- Simple and easy-to-use API for logging messages

3.0.2 How do you work with files anyway?

A file in Node.js is represented by a JavaScript object. This object has properties that describe the file, such as its name, size, and last modified date. The object also has methods that can be used to read, write, and delete the file.

In order to work with files and access file-related helper methods, you can import the `fs` ***module*** from the Node.js standard library.

Wait, what exactly is a `module`?

In Node.js, every JavaScript file is like a little package, called a module. Each module has its own space, and anything you write in a module can only be used in that module, unless you specifically share it with others.

When you make a **`.js`** file in Node.js, it can be a module right away. This means you can put your code in that file, and if you want to use that code in other parts of your application, you can share it using the **`module.exports`** object. On the other hand, you can take code from other modules and use it in your file using the **`require`** function.

This modular approach is important for keeping your code organized and separate, and making it easy to reuse parts of your code in different places. It also helps keep your code safe from errors that can happen when different parts of your code interact in unexpected ways.

Let's see an example by creating a module called `calculate`

Create a file `calculator.js` and add the following contents inside it

```
// calculator.js

function add(num_one, num_two) {
  return num_one + num_two;
}

function subtract(num_one, num_two) {
  return num_one - num_two;
}

function multiply(num_one, num_two) {
  return num_one * num_two;
}

function divide(num_one, num_two) {
  return num_one / num_two;
}

// Only export add and subtract
module.exports = {
  add,
  subtract,
};
```

By specifying the `exports` property on the global `module` object, we declare which specific methods or

properties should be publicly exposed and made accessible from all other modules/files during runtime.

Note, we haven't exported `multiply` and `divide` and we'll see in a moment what happens when we try to access them and invoke/call those functions.

> Note: Provide the relative path to `calculator.js`. In my case, it is located in the same directory and at the same folder level.

In your `index.js` file, you can import the exported functions as shown below.

```
const { add, divide, multiply, subtract } = require("./calculator");

// You may also write it this way, but it's preferred to omit the `.js` extension
const { add, divide, multiply, subtract } = require("./calculator.js");
```

Notice that we're importing the functions `multiply` and `divide` even though we're not exporting them from the `calculator` module. This won't cause any issues until we try to use them. If you run the code above with `node index`, it runs fine but produces no output. Let's try to understand why it doesn't fail.

The `module.exports` is basically a javascript `Object`, and when you `require` it from another file, it tries to evaluate the fields with the names provided (destructuring in short).

So, you can think of it as something like this:

```
const my_module = {
    fn_one: function fn_one() {...},
    fn_two: function fn_two() {...}
}

const { fn_one, fn_two, fn_three } = my_module;
fn_one;   // fn_one() {}
fn_two;   // fn_two() {}
fn_three; // undefined
```

This may clear up why we don't get an error if we try to include a function/property that is not being explicitly exported from a module. If that identifier isn't found, it's simply `undefined`.

So, the `multiply` and `divide` identifiers above are just `undefined`. However, if we try to add this line:

```
// index.js
```

```
let num_two = multiply(1, 2);
```

the program crashes:

```
/Users/ishtmeet/Code/intro-to-node/index.js:5
let num_two = multiply(1, 2);
              ^

TypeError: multiply is not a function
    at Object.<anonymous> (/Users/ishtmeet/Code/intro-to-node/index.js:5:15)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
```

We cannot invoke an `undefined` value as a function. `undefined()` doesn't make any sense.

Let's export all the functions from the `calculator` module.

```
// calculator.js

function add(num_one, num_two) {...}

function subtract(num_one, num_two) {...}

function multiply(num_one, num_two) {...}

function divide(num_one, num_two) {...}

// Only export add and subtract
module.exports = {
  add,
  subtract,
  multiply,
  divide,
};
```

In the `index.js` file, call all those functions to see if everything's working as expected.

```
// index.js

const { add, divide, multiply, subtract } = require("./calculator");

console.log(add(1, 2));
console.log(subtract(1, 2));
console.log(multiply(1, 2));
console.log(divide(1, 2));

// outputs
3 - 1;
2;
0.5;
```

Recall what was just stated above: `module.exports` is simply an object. We only add fields to that object that we wish to export.

So instead of doing `module.exports = { add, subtract, .. }`, you could also do this

```
// calculator.js

module.exports.add = function add(num_one, num_two) {
  return num_one + num_two;
};

module.exports.subtract = function subtract(num_one, num_two) {
  return num_one - num_two;
};

module.exports.multiply = function multiply(num_one, num_two) {
  return num_one * num_two;
};

module.exports.divide = function divide(num_one, num_two) {
  return num_one / num_two;
};
```

It's a matter of preference. But there's a big downside and nuance to this approach. You cannot use these

functions in the same file.

We'll use the term `file` and `module` interchangeably, even though they're not actually the same in theory

```
// calculator.js
module.exports.add = function add(num_one, num_two) {..}
module.exports.subtract = function subtract(num_one, num_two) {..}
module.exports.multiply = function multiply(num_one, num_two) {..}
module.exports.divide = function divide(num_one, num_two) {..}

divide(1,2)

// Outputs
/Users/ishtmeet/Code/intro-to-node/calculator.js:16
divide(1, 2);
^

ReferenceError: divide is not defined
    at Object.<anonymous> (/Users/ishtmeet/Code/intro-to-node/calculator.js:16:1)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Module.require (node:internal/modules/cjs/loader:1143:19)
    at require (node:internal/modules/cjs/helpers:110:18)
    at Object.<anonymous> (/Users/ishtmeet/Code/intro-to-node/index.js:1:45)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
```

This is because `divide` and all the other functions declared in this module are a part of `module.exports` object, and they're not available in the scope. Let's break it down into an easy example

```
let person = {};
person.get_age = function get_age() {...}

// `get_age` is not defined as it can only be accessed using
// `person.get_age()`
get_age();
```

I hope this makes it clear. Instead you could do something like this

```
// calculator.js

...

// Can do this
module.exports.add = add;
module.exports.subtract = subtract;
module.exports.multiply = multiply;
module.exports.divide = divide;

// Or this
module.exports = {
  add,
  subtract,
  multiply,
  divide,
};
```

The first method isn't the best way to create your library's API. The second option is more concise and easier to read. It clearly shows that you're exporting a group of functions as properties of an object. This can be particularly useful when you have many functions to export. Also, everything is nicely placed at a single place. You don't have to keep searching for `module.exports.export_name` to find out what this module exports.

3.0.3 Let's get back to `files`

In Node.js, a `file` is a way to interact with the data in a file. The **`fs`** module is used to handle file operations. It works by using unique identifiers assigned by the operating system to each file, called file descriptors.

With the **`fs`** module, you can perform several operations on files, such as reading, writing, updating, and deleting. Node.js provides both synchronous and asynchronous methods for these operations. The synchronous methods can slow down your application's responsiveness, while the asynchronous methods allow non-blocking execution.

Node.js interacts (indirectly, through) with the operating system's I/O subsystem to manage file operations, making system calls such as **`open`**, **`read`**, **`write`**, and **`close`**. When you open a file, Node.js requests the operating system to allocate a file descriptor, which is used to read or write data from the file. Once the operation is complete, the file descriptor is released.

A file descriptor is a way of representing an open file in a computer operating system. It's like a special number that identifies the file, and the operating system uses it to keep track of what's happening to the file. You can use the file descriptor to read, write, move around in the file, and close it.

3.0.4 A little more about file descriptors

When a file is opened by a process, the operating system assigns a unique file descriptor to that open file. This descriptor is essentially an integer value that serves as an identifier for the open file within the context of that process. File descriptors are used in various system calls and APIs to perform operations like reading, writing, seeking, and closing files.

In Unix-like systems, including Linux, file descriptors are often managed using a data structure called a file table or file control block. This table keeps track of the properties and status of each open file, such as the file's current position, permissions, and other relevant information. The file descriptor acts as an ***index*** or key into this table, allowing the operating system to quickly look up the details of the open file associated with a particular descriptor, which is more efficient, and more performant than to iterate over a vector/array of files and find a particular file.

When you interact with files or file descriptors, you're typically dealing with numeric values. For instance, in C, the **`open()`** system call returns a file descriptor, and other functions like **`read()`**, **`write()`**, and **`close()`** require this descriptor to operate on the corresponding file. In a runtime like Node.js, the **`fs`** module abstracts the direct use of file descriptors by providing a more user-friendly API, but it still relies on them behind the scenes to manage file operations.

A file descriptor is a small, non-negative integer that serves as an index to an entry in the process's table of open file descriptors. This integer is used in subsequent system calls (such as read, write, lseek, fcntl, etc.) to refer to the open file. The successful call will ***return the lowest-numbered file*** descriptor that is not currently open for the process.

3.0.5 Creating our first file

The `node:fs` module lets you work with the file system using standard POSIX functions. Node.js provides multiple ways to work with files. It exposes many flavours of its FileSystem API. A *promise-based asynchronous API*, a *callback-based API* and a *synchronous API.*

Let's create a new module, `files.js`, in the same folder where your `calculator` module and the `index.js` file lives. Let's import the `fs` module to start working with files.

```
// Promise based API
const fs = require("node:fs/promises");

// Sync/Callback based API
const fs = require("node:fs");
```

A general rule of thumb is - always prefer asynchronous API, unless you're dealing with a situation that specifically demands synchronous behaviour.

Asynchronous APIs have two main benefits: they make your code more responsive and scalable. These APIs let your code keep running while it waits for slow tasks like I/O operations or network requests. By not blocking other operations, these APIs allow your application to handle many tasks at once, which improves its overall performance.

Asynchronous code is better for managing multiple tasks happening at the same time than traditional callback-based approaches. With callbacks, it can be hard to keep track of what's going on, leading to a **callback hell**. Using promises and async/await helps make the code easier to read and manage, making it less likely to have issues with complex nested callbacks.

> I will be using the promise-based API of Node.js. However, you may use other options to see what issues arise when your code becomes more complex.

Inside `files.js` add this snippet of code

```
// files.js
const fs = require("node:fs/promises");

async function open_file() {
  const file_handle = await fs.open("calculator.js", "r", fs.constants.O_RDONLY);
  console.log(file_handle);
}

module.exports = open_file;
```

and in `index.js`

```
// index.js
const open_file = require("./files");
```

```
open_file();

/*
FileHandle {
  _events: [Object: null prototype] {},
  _eventsCount: 0,
  _maxListeners: undefined,
  close: [Function: close],
  ..
}
 */
```

Let's break this down.

```
const fs = require("node:fs/promises");
```

This line brings in the `fs` module from Node.js. It specifically imports the `fs/promises` sub-module, which provides file system operations that can be executed asynchronously and are wrapped in Promises.

```
fs.open("calculator.js", "r", fs.constants.O_RDONLY);
```

The `fs.open` function is used to open a file. It takes three arguments - file's `path`, `flag`, and a `mode`.

The `path` takes an argument of type `PathLike` which is a type that represents a file path. It's a concept used in Node.js API to indicate that a value should be a string representing a valid file path. Let's see the type definition of `PathLike`

```
export type PathLike = string | Buffer | URL;
```

3.0.6 `path` argument

1. String **Paths:** The most common way to represent file paths is as strings. A string path can be either a relative or an absolute path. It's simply a sequence of characters that specifies the location of a file on the computer.
   - Example of relative string path: `"./calculator.js"`
   - Example of absolute string path: `"/Users/ishtmeet/Code/intro-to-node/calculator.js"`
2. **Buffer Paths:** While strings are the most common way to represent paths, Node.js also allows you to use `Buffer` objects to represent paths. A `Buffer` is a low-level data structure that can hold binary data. In reality, using `Buffer` objects for paths is less common. Read about Buffers here
3. **URL Paths:** With the `URL` module in Node.js, you can also represent file paths using URLs. The URL

must be of scheme file. Example URL path:

```
const url_path = new URL("file:///home/user/projects/calculator.js");
```

3.0.7 `flag` argument

The `flag` argument indicates the mode (not to confused by `mode` argument) in which you wish to open the file. Here are the supported values as a `flag` -

- `'a'` : Open file for appending. The file is created if it does not exist.
- `'ax'` : Like `'a'` but fails if the path exists.
- `'a+'` : Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` : Like `'a+'` but fails if the path exists.
- `'as'` : Open file for appending in synchronous mode. The file is created if it does not exist.
- `'as+'` : Open file for reading and appending in synchronous mode. The file is created if it does not exist.
- `'r'` : Open file for reading. An exception occurs if the file does not exist.
- `'rs'` : Open file for reading in synchronous mode. An exception occurs if the file does not exist.
- `'r+'` : Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs+'` : Open file for reading and writing in synchronous mode. Instructs the operating system to bypass the local file system cache.
- `'w'` : Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` : Like `'w'` but fails if the path exists.
- `'w+'` : Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` : Like `'w+'` but fails if the path exists.

> You do not need to remember all of these, but it can be useful to write consistent APIs to ensure that no undefined behavior occurs.

Let's use `wx+` to show a small example. `wx+` will open a file for read and write, but fail to open a file if it already exists. If the file doesn't exists it will create a file and work just fine.

```
// calculator.js
const file_handle = await fs.open(
    "calculator.js",
    "wx+",
    fs.constants.O_RDONLY
  );
```

```
// Outputs
node:internal/process/promises:288
            triggerUncaughtException(err, true /* fromPromise */);
            ^

[Error: EEXIST: file already exists, open 'calculator.js'] {
  errno: -17,
  code: 'EEXIST',
  syscall: 'open',
  path: 'calculator.js'
}
```

It's a good practice to specify the `flag` argument.

3.0.8 `mode` argument

The `mode` argument specifies the permissions to set for the file when its created. `mode`s are always interpreted **in octal.** For example,

- **`0o400`** (read-only for the owner)
- **`0o600`** (read and write for the owner)
- **`0o644`** (read for everyone, write only for the owner)

You don't need to remember the octal representation. Simply use the `fs.constants.your_mode` to access it.

In our case, the permissions are specified as `fs.constants.O_RDONLY`. Here is a list of available `modes` that can be used. Notice the `O_` prefix, which is short for `Open`. This prefix tells us that it will only work when used with `fs.open()`.

Modes to use with `fs.open()`

```
/** Flag indicating to open a file for read-only access. */
const O_RDONLY: number;

/** Flag indicating to open a file for write-only access. */
const O_WRONLY: number;

/** Flag indicating to open a file for read-write access. */
const O_RDWR: number;
```

```
/** Flag indicating to create the file if it does not already exist. */
const O_CREAT: number;

/** Flag indicating that opening a file should fail if the O_CREAT flag is set and the file
 ↪  already exists. */
const O_EXCL: number;

/** Flag indicating that if the file exists and is a regular file, and the file is opened
 ↪  successfully for write access, its length shall be truncated to zero. */
const O_TRUNC: number;

/** Flag indicating that data will be appended to the end of the file. */
const O_APPEND: number;

/** Flag indicating that the open should fail if the path is not a directory. */
const O_DIRECTORY: number;

/** Flag indicating that the open should fail if the path is a symbolic link. */
const O_NOFOLLOW: number;

/** Flag indicating that the file is opened for synchronous I/O. */
const O_SYNC: number;

/** Flag indicating that the file is opened for synchronous I/O with write operations waiting for
 ↪  data integrity. */
const O_DSYNC: number;

/** Flag indicating to open the symbolic link itself rather than the resource it is pointing to.
 ↪  */
const O_SYMLINK: number;

/** When set, an attempt will be made to minimize caching effects of file I/O. */
const O_DIRECT: number;

/** Flag indicating to open the file in nonblocking mode when possible. */
const O_NONBLOCK: number;
```

Going back to the code we wrote in the `files` module

```
// files.js
const fs = require("node:fs/promises");

async function open_file() {
  const file_handle = await fs.open("calculator.js", "r", fs.constants.O_RDONLY);
  console.log(file_handle);
}

module.exports = open_file;
```

The return type of `fs.open()` is a `FileHandle`. A file handle is like a connection between the application and the file on the storage device. It lets the application work with files without worrying about the technical details of how files are stored on the device.

We previously discussed **file descriptors**. You can check which descriptor is assigned to an opened file.

```
// files.js

..

async function open_file() {
    const file_handle = await fs.open("calculator.js", "r", fs.constants.O_RDONLY);
    console.log(file_handle.fd); // Print the value of the file descriptor `fd`
}

..

// Outputs -> 20
```

You may get the same integer value for the file descriptor if you try to run the program multiple times. But if you try to create another file handle, it should have a different file descriptor

```
// files.js

..

async function open_file() {
    const file_handle     = await fs.open("calculator.js", "r", fs.constants.O_RDONLY);
```

```
    const file_handle_two = await fs.open("calculator.js", "r", fs.constants.O_RDONLY);
    console.log(file_handle.fd);
    console.log(file_handle_two.fd);
}

..

// Outputs ->
20
21
```

Note that if a `FileHandle` is not closed using the `file_handle.close()` method, it will try to automatically close the file descriptor and emit a process warning, helping to prevent memory leaks. It's always good practice to call `file_handle.close()` to explicitly close it. However, in our case, the program exits just after running the `open_file` function, so it doesn't matter in our case.

One import thing to note is, `open`ing a file can fail, and will throw an exception.

`fs.open()` can throw errors in various scenarios, including:

- `EACCES` : Access to the file is denied or permission is lacking, or the file doesn't exist and parent directory isn't writable.
- `EBADF` : The directory file descriptor is invalid.
- `EBUSY` : The file is a block device in use or mounted.
- `EDQUOT` : Disk quota for user is exceeded when creating a file.
- `EEXIST` : File already exists while trying to create it exclusively.
- `EFAULT` : Path is outside accessible memory.
- `EFBIG` / `EOVERFLOW` : File is too large to open.
- `EINTR` : Opening a slow device is interrupted by a signal.
- `EINVAL` : Invalid flags or unsupported operations.
- `EISDIR` : Attempting to write to a directory, or using `O_TMPFILE` on a version that doesn't support it.
- `ELOOP` : Too many symbolic links encountered.
- `EMFILE` : Process reached its limit of open file descriptors.
- `ENAMETOOLONG` : Pathname is too long.
- `ENFILE` : System-wide limit on open files is reached.
- `ENOENT` : File or component in path doesn't exist.
- `ENOMEM` : Insufficient memory for FIFO buffer or kernel memory.

- `ENOSPC`: No space left on device.
- `ENOTDIR`: Component in path is not a directory.
- `ENXIO`: File doesn't correspond to device, socket, or FIFO.
- `EOPNOTSUPP`: Filesystem doesn't support `O_TMPFILE`.
- `EROFS`: File is on read-only filesystem.
- `ETXTBSY`: File is being executed, used as swap, or read by kernel.
- `EPERM`: Operation prevented by file seal or mismatched privileges.
- `EWOULDBLOCK`: `O_NONBLOCK` specified, incompatible lease held on the file.

Make sure to handle errors gracefully. There may be cases where you don't need to handle the errors and want the program to fail, exit, or throw an error to the client. For example, if you're writing a CLI application that compresses an image using the `path/to/image` provided as an argument, you want it to fail to let the user know that there is an issue with the file/path provided.

To catch errors, wrap the code inside a `try/catch` block.

```
// files.js

..

async function open_file() {
  try {
    const file_handle = await fs.open("config", "r", fs.constants.O_WRONLY);
        // do something with the `file_handle`
  } catch (err) {
    // Do something with the `err` object
  }
}

..
```

3.1 Reading from a file

Too much of the theory. We'll work on a real example now. Let's try to read from a file. We'll create a `log_config.json` file, in the `config` folder. The directory structure will look something like this (get rid of the `calculator` module)

```
.
├── config
│   └── log_config.json
```

```
├── files.js
└── index.js
```

Add these content inside the `log_config.json` file

```
// log_config.json

{
  "log_prefix": "[LOG]: "
}
```

Node.js provides many utility methods for reading from a specific file using the `file_handle`. There are different ways to handle interactions with the files from the `node:fs` and the `node:fs/promises` modules. But we're specifically going to use a `file_handle` for now.

```
// files.js

const fs = require("node:fs/promises");

// This function asynchronously opens a file, reads it line by line
// and logs each line on the console.
async function read_file() {
  try {
    // open the file in read-only mode.
    const file_handle = await fs.open("./index.js", "r", fs.constants.O_RDONLY);

    // create a stream to read the lines of the file.
    let stream = file_handle.readLines({
      // start reading from the beginning of the file.
      start: 0,

      // read till the end of the file.
      end: Infinity,

      // specify the encoding to be utf8, or else the stream
      // will emit buffer objects instead of strings.
      encoding: "utf8",

      /**
```

```
       * If autoClose is false, then the file descriptor won't be closed,
       * even if there's an error. It is the application's responsibility
       * to close it and make sure there's no file descriptor leak. If
       * autoClose is set to true (default behavior), on 'error' or 'end' the
       * file descriptor will be closed automatically.
       */
      autoClose: true,

      /**
       * If emitClose is true, then the `close` event will be emitted
       * after reading is finished. Default is `true`.
       */
      emitClose: true,
    });

    // The 'close' event is emitted when the file_handle has been closed
    // and can no longer be used.
    stream.on("close", () => {
      console.log("File handle %d closed", file_handle.fd);
    });

    // The 'line' event be fired whenever a line is read from the file.
    stream.on("line", (line) => {
      console.log("Getting line -> %s", line);
    });
  } catch (err) {
    console.error("Error occurred while reading file: %o", err);
  }
}

module.exports = read_file;
```

This outputs

```
Getting line -> const open_file = require("./files");
Getting line ->
Getting line -> open_file();
File handle 20 closed
```

The code above has a function called `read_file` that does three things: open a file, read each line, and show each line on the console.

This function uses the `fs` module. It opens a read-only file and creates a stream to read it. The function can read only some lines using the `start` and `end` options. The function also needs to know the file's characters using the `encoding` option.

This function also sets two options to handle the file descriptor automatically when the reading is finished. Finally, this function creates two listeners to handle two events: `close` and `line`. The `close` event tells the function that the file handle has been closed. The `line` event tells the function that it has read a line from the file.

If there's an error while reading the file, the function shows an error message on the console.

One thing to note is that we used string substitution `%s` instead of template literals. When passing a string to one of the methods of the `console` object that accepts a string, you may use these substitution strings:

- `%o` or `%O`: Outputs a JavaScript object. Clicking the object name opens more information about it in the inspector (browser).
- `%d`: Outputs an integer. Number formatting is supported. For example, `console.log("Foo %d", 1)` will output the number as an number (will retain floating point value).
- `%i`: Outputs an integer. Number formatting is supported. For example, `console.log("Foo %i", 1.1)` will output the number as an integer (will truncate the floating point value).
- `%s`: Outputs a string.
- `%f`: Outputs a floating-point value. Formatting is supported. For example, `console.log("Foo %f", 1.1)` will output "Foo 1.1".

> Using `%o` to show the output on terminal, just prints the whole object as a string, this is something that the string substitution has an advantage over template literals.

We can simplify the above code. I included all possible option keys previously just to show that they exist, and you could use them if you want to have more control over what you're doing.

The simplified version looks like this

```
// files.js

...

async function read_file() {
  try {
```

```
    const file_handle = await fs.open("./index.js");
    const stream = file_handle.readLines();

    // we'll get to this syntax in a bit
    for await (const line of stream) {
      console.log("Reading line of length %d -> %s", line.length, line);
    }
  } catch (err) {
    console.error("Error occurred while reading file: %o", err);
  }
}

...
```

This outputs

```
Reading line of length 59 -> const { read_entire_file, read_file } = require("./files");
Reading line of length 0 ->
Reading line of length 12 -> read_file();
```

Notice that we get rid of all those options since they are already set to default values for our convenience. Only specify them if you wish to choose values other than the defaults.

3.2 A small primer on `for..of` and `for await..of` in javascript

3.2.1 `for..of`

The **`for..of`** loop is a JavaScript feature that provides an easy and straightforward way to go through elements in an array, string, or other iterable objects. It makes it simpler to iterate through each item without the need to manage the loop's index or length manually.

Let's look at the syntax:

```
for (const element of iterable) {
  // Code to be executed for each element
}
```

Here's an overview of how the **`for..of`** loop works:

1. **`for`** : This is the keyword that indicates the start of the loop structure.

2. **`element`**: This is a variable that you define to represent the current element of the iterable in each iteration of the loop. In each iteration, the **`element`** variable will hold the value of the current element in the iterable.
3. **`of`**: This is a keyword that signifies the relationship between the **`element`** variable and the **`iterable`** you're looping through.
4. **`iterable`**: This is the collection or object you want to iterate over. It can be an array, a string, a set, a map, or any other object that has a collection of items.

Here's an example of using **`for..of`** to loop through an array:

```
const fruits = ["apple", "banana", "orange", "grape"];

for (const fruit of fruits) {
  console.log(fruit);
}
```

The loop will iterate through each element in the **`fruits`** array, and in each iteration, the **`fruit`** variable will contain the value of the current fruit. The loop will log:

```
apple
banana
orange
grape
```

The **`for..of`** loop is particularly useful when you don't need to access the index of the elements directly. It provides a cleaner and more concise way to work with iterable objects.

Note that the **`for..of`** loop can't be used to directly loop over properties of an object. It's specifically designed for iterating over values in iterable collections. If you need to loop through object properties, the traditional **`for..in`** loop or using **`Object.keys()`**, **`Object.values()`**, or **`Object.entries()`** would be more appropriate.

3.2.2 `for await..of`

The **`for await..of`** loop is an extension of the **`for..of`** loop. It is used for asynchronous operations and iterables. It can iterate over asynchronous iterable objects like those returned by asynchronous generators or promises. The loop is useful when dealing with asynchronous operations like fetching data from APIs or reading from streams, just like we did above!

Here's how the **`for await..of`** loop works:

```
for await (const element of async_iterable) {
  // Asynchronous code to be executed for each element
}
```

Let's break down the key components:

1. `for await`: These keywords start the asynchronous loop structure.
2. `element`: This variable represents the current element of the asynchronous iterable in each iteration of the loop.
3. `async_iterable`: This is an asynchronous iterable object, such as an asynchronous generator, a promise that resolves to an iterable, or any other object that implements the asynchronous iteration protocol.

Here's an example of using `for await..of` to loop through an asynchronous iterable:

```
async function fetch_fruits() {
  const fruits = ["apple", "banana", "orange", "grape"];

  for await (const fruit of fruits) {
    console.log(fruit);

    // a dummy async operation simulation
    await new Promise((resolve) => setTimeout(resolve, 1000));
  }
}

fetch_fruits();
```

Here, the `fetchFruits` function uses the `for await..of` loop to iterate through the `fruits` array asynchronously. For each fruit, it logs the fruit name and then simulates an asynchronous operation using `setTimeout` to pause for a second.

The `for await..of` loop is a handy tool when working with asynchronous operations. It allows us to iterate over the results of promises or asynchronous generators in a more readable and intuitive way. It ensures that the asynchronous operations within the loop are executed sequentially, one after the other, even if they have varying completion times.

3.3 Reading the `json` file

However, reading a json file line by line isn't the best way. The `readLine` is a very memory-efficient way to read files. It does not load all the contents of the file into memory, which is usually what we want. But if the file is small, and you know before hand, that the file is not really big, it's usually quicker, and more performant to load the entire file at once into the memory.

> If you're dealing with large files, it's usually better to use a buffered version, i.e `createReadStream()` or `readLines()`

Let's update the code

```
...

async function read_file() {
  try {
    const file_handle = await fs.open("./config/log_config.json");
    const stream = await file_handle.readFile();

    console.log("[File contents] ->\n %s", stream);
  } catch (err) {
    console.error("Error occurred while reading file: %o", err);
  }
}

...
```

Outputs

```
[File contents] ->
 {
  "log_prefix": "[LOG]: "
}
```

Nice. But what happens, if we do not use the string substitution with `%s`?

```
console.log("[File contents] ->\n", stream);
```

Strangely, this outputs some weird looking stuff

```
[File contents] ->
 <Buffer 7b 0a 20 20 22 6c 6f 67 5f 70 72 65 66 69 78 22 3a 20 22 5b 4c 4f 47 5d 3a 20 22 0a 7d 0a>
```

Why is it so? And what is a `Buffer`? This is one of the most unvisited topics of programming. Let's take a minute to understand it.

3.4 Buffers

`Buffer` objects are used to represent a fixed-length sequence of bytes, in memory. **`Buffer`** objects are more memory-efficient compared to JavaScript strings when dealing with data, especially very large datasets. This is because strings in JavaScript are UTF-16 encoded, which can lead to higher memory consumption for certain types of data.

Q: But why does the `readLines()` method returned strings if it's not "efficient"?

Well turns out, they do indeed use buffers internally to efficiently read and process data from files or streams. `readLines()` is a special variant of `createReadStream()` which is designed to provide a convenient interface for working with lines of text content, making it easier for developers to interact with the data without needing to handle low-level buffer operations directly.

So, what you're looking at when you see the value of a buffer is just a raw representation of binary data in ***hexadecimal format***. This raw data might not make much sense to us as humans because it's not in a readable format like text.

To print the json file to the console, we have 3 ways.

First method

```
console.log("[File contents] ->\n", stream.toString("utf-8"));
```

Second method

String substitution to the rescue again

```
console.log("[File contents] ->\n %s", stream);
```

The second method is much more user friendly. They automatically serialize the binary content into a string. But, to use and manipulate the string contents, we'll have to fall back to the first method.

Third method

Set the `encoding` option to `utf-8`

```
const stream = await file_handle.readFile({ encoding: "utf-8" });
console.log("[File contents] ->\n", stream);
```

3.4.1 Parsing the `json` file

To read the `log_prefix` property that we specified into the `config/log_config.json` file, let's parse the contents of the file.

> Many people use the `require('file.json')` way, but there are several drawbacks to it. First, the entire file is loaded into memory when your program encounters the require statement. Second, if you update the json file during runtime, the program will still refer to the old data. It is recommended to use `require()` only when you expect the file not to change, and it is not excessively large; otherwise, it will always remain in memory.

```
// files.js

...

const stream = await (await fs.open("./config/log_config.json")).readFile();
const config = JSON.parse(stream);

console.log('Log prefix is: "%s"', config.log_prefix);

...

// Outputs →
// Log prefix is: "[LOG]: "
```

This looks fine, but it is not a very good practice to specify paths like this. Using **`"./config/log_config.json"`** assumes that the **`config`** directory is located in the same directory as the current working directory of the terminal. This might not always be the case, especially if your script is being run from a different working directory, eg. from the config folder. To test this behavior, `cd config` and run `node ../index.js`

```
Error occurred while reading file: [Error: ENOENT: no such file or directory, open
↪  './config/log_config.json'] {
  [stack]: "Error: ENOENT: no such file or directory, open './config/log_config.json'",
  [message]: "ENOENT: no such file or directory, open './config/log_config.json'",
  errno: -2,
  code: 'ENOENT',
  syscall: 'open',
  path: './config/log_config.json'
```

```
}
```

This expects the path is relative to the current working directory, hence not what we expect. We should be able to run the script from anywhere, no matter what folder we are in. This is very useful for large projects having folders multiple levels deep.

Update the code to include the `path` module in scope

```
// files.js

const path = require('path');

...

const log_path = path.join(__dirname , 'config' , 'log_config.json');
const stream = await (await fs.open(log_path)).readFile();

...
```

Using **`__dirname`** and the **`path`** module ensures that you are referencing the correct path regardless of the current working directory you're in.

`__dirname` is a special (module-level) variable that represents the absolute path of the directory containing the current JavaScript file. Isn't it magic?

`path.join()` method joins all given `path` segments together using the **platform-specific separator** as a delimiter, then normalizes the resulting path. Zero-length `path` segments are ignored. If the joined path string is a zero-length string then `'.'` will be returned, representing the current working directory.

The full code of `files.js` looks like this now.

```
const fs = require("node:fs/promises");
const path = require("path");
async function read_file() {
  try {
    const log_path = path.join(__dirname, "config", "log_config.json");
    const stream = await (await fs.open(log_path)).readFile();
    const config = JSON.parse(stream);

    console.log('Log prefix is: "%s"', config.log_prefix);
  } catch (err) {
```

```
    console.error("Error occurred while reading file: %o", err);
  }
}
```

Now you can run the code from whatever directory, no matter how much deeply nested it is, it is going to work fine unless you move the `files.js` file to a different location.

Chapter 4

`logtar` our own logging library

> Note: The entire code we write here can be found at the src/chapter_04.0 directory. This will be a single file, and we'll refactor in subsequent chapters.

Logging is an important part of creating robust and scaleable application. It helps developers find and fix problems, keep an eye on how the application is working, and see what users are doing.

4.1 Initializing a new project

Let's create a new project. Close your current working directory, or scrap it.

```
# Create a new directory, and cd into it
mkdir logtar && cd logtar

# Initializes a new package
npm init -y
```

This creates a new npm package/project, creates a `package.json` file and sets up some basic config inside it. This should be the contents of your `package.json`

```
{
  "name": "logtar",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
```

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Let's change the version from `1.0.0` to `0.0.1`.

4.2 A little about `SemVer`

Starting versioning from **`0.0.1`** is a good practice in software development because version numbers have semantic meaning. Using **`0.0.1`** as the initial version indicates that the software is in its initial development stages or that it's undergoing rapid changes and improvements. This convention aligns with Semantic Versioning (SemVer), which is a widely adopted versioning scheme that helps developers understand the compatibility and significance of changes in software releases.

Starting with **`0.0.1`** is particularly beneficial for a few reasons:

1. **Clarity of Development Stage**: Starting with **`0.0.1`** clearly communicates that the software is in its early stages of development. This helps other developers and users understand that the API and features might change more rapidly and might not yet be stable.
2. **Semantic Versioning**: Semantic Versioning consists of three numbers separated by dots: **`MAJOR.MINOR.PATCH`**. Starting from **`0.0.1`** indicates that you're in the process of making minor patches and potentially significant changes as you develop the software.
3. **Incremental Progress**: Starting with **`0.0.1`** allows for a clear sequence of version increments as development progresses. Each release can follow the rules of SemVer: incrementing the **`MAJOR`** version for backward-incompatible changes, the **`MINOR`** version for backward-compatible additions, and the **`PATCH`** version for backward-compatible bug fixes.
4. **User Expectations**: When users or other developers encounter a software version that starts with **`0.0.1`**, they'll understand that the software might not be feature-complete or entirely stable. This helps manage expectations and reduces confusion.
5. **Preventing Confusion**: If you started with version **`1.0.0`**, there might be an expectation of stability and feature completeness that could lead to confusion if the software is actually in an early stage of development.

4.3 Creating a `LogLevel` class

Log level is a basic concept in logging libraries. It helps control how much detail the application's log messages show. Developers use log levels to filter and manage the output. This is especially useful when debugging issues or dealing with complex systems.

Usually logging libraries have these 5 log levels (from least to most severe):

1. **Debug**: Detailed debugging information. Usually not used in production environments because it generates too much data.
2. **Info**: Informative messages about the regular flow of the application. Shows what the application is doing.
3. **Warning**: Indicates potential issues that might require attention. Warnings suggest that something might be going wrong.
4. **Error**: Reports errors or exceptional conditions that need to be addressed. These messages indicate that something has gone wrong and might affect the application's functionality.
5. **Critical/Fatal**: For severe errors that might crash the application or cause major malfunctions. These messages require immediate attention as they indicate critical failures.

> I prefer using `Class` over functions or objects to provide a better API. It's a powerful system in JavaScript, and I find it superior to factory functions. `Class` do have some draw backs but for our use case, they're the best possible solution. If you're un-aware of how classes work in javascript, just go through this and this page on my another javascript learning resource.

When building a complex system or anticipating scalability, it's best to start simply and refactor as necessary. You don't need to use best practices from day one.

The process should be: write code, make it work, test it, and then refactor it.

Let's create a new file `index.js` inside the `logtar` directory, and add a new class `LogLevel`

```
// index.js

class LogLevel {
  static Debug = 0;
  static Info = 1;
  static Warn = 2;
  static Error = 3;
  static Critical = 4;
}
```

```
module.exports = {
  LogLevel,
};
```

You might be wondering about the use of a class `LogLevel` instead of an object, or maybe some constants that can be easily exported, like this -

```
module.exports.LogLevel = {
    Debug: 0
    ...
}

// or

module.exports.Debug = 0
...
```

You could do something like this too, and that's totally fine. But instead, I chose a different method i.e using a utility class `LogLevel` which encapsulates all log levels within a class, you create a clear namespace for these constants. This helps avoid potential naming conflicts with other variables or constants in your application. There's more to this!

You will see another powerful feature by using this method, in a bit.

Let's add a helper method to our `LogLevel` class which checks and verifies whether the `LogLevel` provided by our client (user of our library) is correct and supported.

```
// index.js

class LogLevel {
    ...

    static assert(log_level) {
       if (
            log_level !== LogLevel.Debug ||
            log_level !== LogLevel.Info ||
            log_level !== LogLevel.Warn ||
            log_level !== LogLevel.Error ||
            log_level !== LogLevel.Critical
```

```
        ) {
            throw new Error(
            `log_level must be an instance of LogLevel. Unsupported param
            ↪ ${JSON.stringify(log_level)}`);
        }
    }
}
```

What is this `assert` method going to do? `assert` will be a method used inside our library, which verifies that the value of type `LogLevel` provided as an argument is valid and supported.

However, we can refactor the code above to look more readable, and not repeat too much.

```
// index.js

static assert(log_level) {
    if (![LogLevel.Debug, LogLevel.Info, LogLevel.Warn, LogLevel.Error,
    ↪ LogLevel.Critical].includes(log_level)) {
        throw new Error(`log_level must be an instance of LogLevel. Unsupported param
        ↪ ${JSON.stringify(log_level)}`);
    }
}
```

This way, if we wish to add more log levels, we can simply add them to the array. This is another powerful use-case of classes over normal object values. Everything is namespace'd. Even without typescript, we can tell the client (someone who uses our library) what arguments are we expecting. If they fail to provide an invalid argument, our `assert` method will throw an error.

Also, even if the user is unaware of the values we're using for each log level, it does not matter as long as they're using the `LogLevel.any_level` syntax. If we in future change the internals of the library, as long as the public API is consistent, everyone will be good. This is a key to build reliable APIs.

> I am going to use the terms `client` and `user` interchangeably. A `client` is someone who uses our library's API.

The `LogLevel` looks fine for now. Let's introduce a new class `Logger` which will be the backbone of our logging library.

4.4 The `Logger` class

```
// index.js

class LogLevel {...}

class Logger {
    /* Set the default value of `level` to be `LogLevel.Debug` for every
    * new instance of the `Logger` class
    */
    level = LogLevel.Debug;

    // Sets the log level to whatever user passed as an argument to `new Logger()`
    constructor(log_level) {
      this.level = log_level;
    }
}
```

There's an issue. The user may construct the `Logger` with whatever value they want. This can be some useless value like `100000` or "`Hello`" and that's not what we would expect.

For example

```
// makes no sense
const my_logger = new Logger("test");
```

Let's make use of the `LogLevel.assert()` static method that we just defined.

```
// index.js

class LogLevel {...}

class Logger {
    level = LogLevel.Debug;

    constructor(log_level) {
        // Throw an error if the `log_level` is an unsupported value.
        LogLevel.assert(log_level);
        this.level = log_level;
```

```
    }
}
```

Now if we try to pass an invalid argument to the construction of `Logger` we get an error. Exactly what we want

```
const logger = new Logger(6);

// outputs
Error: log_level must be an instance of LogLevel. Unsupported param "6"
    at LogLevel.assert (/Users/ishtmeet/Code/logtar/index.js:10:13)
    at new __Logger (/Users/ishtmeet/Code/logtar/index.js:86:14)
    at Object.<anonymous> (/Users/ishtmeet/Code/logtar/index.js:91:16)
    at Module._compile (node:internal/modules/cjs/loader:1256:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1310:10)
    at Module.load (node:internal/modules/cjs/loader:1119:32)
    at Module._load (node:internal/modules/cjs/loader:960:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
```

Everything looks good! No, not yet. What if we try to do

```
const logger = new Logger(LogLevel.Error);
logger.level = 1000;
```

Again, this breaks our whole library's functionality. How do we prevent this? Seems like Javascript has us covered.

4.4.1 Encapsulation with `private` fields

Class fields are public by default, which means anyone can change it from anywhere in the code. Even the clients of our library can change it too. However, you can create private class features by adding a hash `#` prefix. JavaScript enforces the privacy encapsulation of these class features.

Before this syntax existed, private members were not native to the language. In prototypical inheritance, their behavior can be emulated with `WeakMap` objects or closures. But using `#` syntax is more ergonomic than these methods.

`members` of a class mean any thing that is defined inside the `class` block. That includes variables, static variables, or even methods. Note that if a function is defined inside a class, it's referred to as a `method` and not a `function`

Let's update the code above by incorporating `encapsulation`.

```
// index.js

class LogLevel {...}

class Logger {
    // introduce a new `private` variable `#level`
    #level;

    constructor(log_level) {
        // You refer to private variables using the `#` prefix.
        this.#level = log_level;
    }
}

const logger = new Logger(LogLevel.Debug);
console.log(logger.#level); // Error: Private field '#level' must be declared in an enclosing
↪ class
logger.#level = LogLevel.Info; // Error: Private field '#level' must be declared in an enclosing
↪ class
```

This is looking good. We can refer to `#level` member variable only inside the class. No one can change it. But we do need to provide a way to know the current log level of our logger. Let's add a `getter` method.

```
// index.js

class LogLevel {...}

class Logger {
    ...

    get level() {
        return this.#level;
```

```
    }
}

const logger = new Logger(LogLevel.Debug);
console.log(logger.#level); // Error: Private field '#level' must be declared in an enclosing
↪ class
console.log(logger.level); // Good. Calls the `get level()` method
```

> Note: If you create a getter using `get()`, you do not need to specify the parenthesis after `level`. Javascript knows that we're referring to the `get level()` getter.

Now, add the `LogLevel.assert` method inside the constructor, to make sure the clients pass in a correct value for the `log_level` constructor parameter.

```
// index.js
class Logger {
    ..
    constructor(log_level) {
        LogLevel.assert(log_level);
        this.#level = log_level;
    }
    ..
}

const logger = new Logger(100); // Throws an error
const logger  = new Logger(3); // Good
const logger = new Logger(LogLevel.Debug); // Best practice
```

It's always a good practice to allow clients create an object without specifying a value in the constructor, in that case we should use some set defaults. The full code of the `index.js` should look like this

```
// index.js

class LogLevel {
  static Debug = 0;
  static Info = 1;
  static Warn = 2;
```

```
  static Error = 3;
  static Critical = 4;

  static assert(log_level) {
    if (![LogLevel.Debug, LogLevel.Info, LogLevel.Warn, LogLevel.Error,
    ↪  LogLevel.Critical].includes(log_level)) {
      throw new Error(`log_level must be an instance of LogLevel. Unsupported param
      ↪  ${JSON.stringify(log_level)}`);
    }
  }
}

class Logger {
  // set a default value for the log level
  #level = LogLevel.Info;

  constructor(log_level) {
    // only set/check the log level if the client has provided it
    // otherwise use the default value, i.e `LogLevel.Info`
    if (arguments.length > 0) {
      LogLevel.assert(log_level);
      this.#level = log_level;
    }
  }

  get level() {
    return this.#level;
  }
}

module.exports = {
  Logger,
  LogLevel,
};
```

Let's try to test this.

```
new Logger("OK"); // throws error
new Logger(LogLevel.Debug); // works fine
new Logger(); // works fine

let logger = new Logger(LogLevel.Warn);
logger.level; // returns the `level` because of the getter `level()`
logger.#level; // throws error
logger.#level = LogLevel.Warn; // throws error
logger.level = 10; // throws error
```

Perfect! This all looks really good. We are confident that neither clients nor our own library's code will affect the internals of the library. Please note that only the `#level` member variable can be changed from within the class Logger's scope, which is exactly what we want.

4.5 The `LogConfig` class

We have a bare bones `Logger` setup, which isn't helpful at all. Let's make it a bit more useful.

You see, setting the log level inside the logger is fine until we start adding a lot of config settings. For example, we may add a `file_prefix` member variable, as well as some other configuration related variables. In that case, the `Logger` class will get cluttered too much, and that's not what we want.

Let's start refactoring now. Create a new class `LogConfig` that will contain all the utility helpers to deal with log config. Everything that take cares about the configuration will live inside it.

```
// index.js

class LogConfig {
  /** Define necessary member variables, and make them private. */

  // log level will live here, instead of the `Logger` class.
  #level = LogLevel.Info;

  // We do not initialise it here, we'll do it inside the constructor.
  #rolling_config;

  // the prefix to be added to new files.
  #file_prefix = "Logtar_";
```

```
  /**
   * We're going to follow the convention of creating a static assert
   * method wherver we deal with objects. This is one way to write
   * safe code in vanilla javascript.
   */
  static assert(log_config) {
    // if there's an argument, check whether the `log_config` is an instance
    // of the `LogConfig` class? If there's no argument, no checks required
    // as we'll be using defaults.
    if (arguments.length > 0 && !(log_config instanceof LogConfig)) {
      throw new Error(`log_config must be an instance of LogConfig. Unsupported param
      ↪  ${JSON.stringify(log_config)}`);
    }
  }

  get level() {
    return this.#level;
  }

  get rolling_config() {
    return this.#rolling_config;
  }

  get file_prefix() {
    return this.#file_prefix;
  }
}
```

All looks okay. We have a `LogConfig` class setup. Now, instead of using `#level` for storing the log level inside the `Logger` class, let's replace it with `#config`. Also add a helper method `with_config`.

```
// before

class Logger {
    ...

    #level = LogLevel.Info;
```

```
    constructor(log_level) {
        // only set/check the log level if the client has provided it
        // otherwise use the default value, i.e `LogLevel.Info`
        if (arguments.length > 0) {
            LogLevel.assert(log_level);
            this.#level = log_level;
        }
    }

    ...
}

// now

class Logger {
    #config;
    ...

    constructor(log_config) {
        // we'll create the `with_defaults()` static method in just a bit.
        log_config = log_config || LogConfig.with_defaults();
        LogConfig.assert(log_config);
        this.#config = log_config;
    }

     static with_config(log_config) {
        return new Logger(log_config);
    }
}
```

Awesome. Let's pause for a moment before adding further functionality inside the `LogConfig` class. Let me quickly introduce you to a very important topic in software engineering.

4.6 Design patterns

Software design patterns are a solution to a common problem that software engineers face while writing code. It's like a blueprint that shows how to solve a couple problem that can be used in many different

situations. Those problems are - maintainability and organizing code.

This is a vast topic, and people have dedicated books for explaining the use of these patterns. However, we aren't going to explain each one of those. We'll use the one that suits best for our project. Always find the right tool for the right job. For us, the most reasonable design pattern to build our web framework, as well as our logging library will be the `Builder pattern`

4.6.1 The `Builder` pattern

Think of the Builder Pattern as a way to create complex objects step by step. Imagine you're building a house. Instead of gathering all the materials and putting them together at once, you start by laying the foundation, then building the walls, adding the roof, and so on. The Builder Pattern lets you do something similar. It helps you construct objects by adding parts or attributes one by one, ultimately creating a complete and well-structured object.

Just for a minute think that you're creating a web application where users can create personal profiles. Each profile has a `name`, an `age`, and a `description`. The Builder Pattern would be a great fit here because users might not provide all the information at once. Here's how it could work

```
const user = new ProfileBuilder().with_name("Alice").with_age(25).with_description("Loves hiking
↪  and painting").build();
```

Doesn't this look so natural? Having to specify steps, without any specific order, and you get what you desired. Compare this to a traditional way of building using an object

```
const user = create_profile({
  name: "Alice",
  age: 25,
  description: "Loves hiking and painting",
});
```

The object solution looks fine, and even has less characters. Then why the builder pattern? Imagine your library in a future update changes the `name` property to `first_name` and include a secondary `last_name` property. The code with object will fail to work properly. But in our builder pattern, it's obvious that the `name` means full name. That might not sound convincing. Let's see at a different example.

In a language like javascript (typescript solves this) you need to make sure that the params you pass as an argument are valid.

Here's a common way you'll write the function `create_profile`

```
function create_profile({ name, age, description }) {
    let profile = {
        name: Defaults.name,
        age: Defaults.age
        description: Defaults.description
    }
    if (typeof name === 'string') { profile.name = name }
    if (typeof age === 'number' && age > 0) { profile.age = age }
    if (typeof description === 'string') { profile.description = description }
}
```

Notice how cluttered this code becomes if there are 10 fields? The function `create_profile` should not be responsible for testing. Its role is to create a profile. We could group other functions, such as `validate_name` and `validate_email`, and call them inside the `create_profile` function. However, this code would not be reusable. I have made this mistake in the past and ended up with code that was difficult to refactor.

Instead, let's use the builder pattern to validate whether the fields are valid:

```
class ProfileBuilder {
    name = Defaults.name;
    age = Defaults.age;
    description = Defaults.description;

    with_name(name) {
        validate_name(name);
        this.name = name;
        return this;
    }

     with_age(age) {
        validate_age(age);
        this.age = age;
        return this;
    }

    with_description(description) {...}
}
```

Do you notice the difference? All of the related validations and logic for each field are separated and placed

in their respective locations. This approach is much easier to maintain over time, and reason about.

4.7 Using `builder` pattern with the `LogConfig` class

Here's what I'd like the API of `LogConfig` to look like

```
const config =
↪  LogConfig.with_defaults().with_file_prefix("LogTar_").with_log_level(LogLevel.Critical);
```

Our current `LogConfig` class looks like this

```
// index.js

class LogConfig {
  #level = LogLevel.Info;

  // We'll talk about rolling config in just a bit.
  #rolling_config = RollingConfig.Hourly;
  #file_prefix = "Logtar_";

  static assert(log_config) {
    if (arguments.length > 0 && !(log_config instanceof LogConfig)) {
      throw new Error(`log_config must be an instance of LogConfig. Unsupported param
      ↪  ${JSON.stringify(log_config)}`);
    }
  }
}
```

Add the required methods

```
// index.js

class LogConfig {
    ...
    // This can be called without a `LogConfig` object
    // eg. `LogConfig.with_defaults()`
    static with_defaults() {
        return new LogConfig();
    }
```

```
    // Validate the `log_level` argument, set it to the private `#level` variable
    // and return this instance of the class back. So that other methods can mutate
    // the same object, instead of creating a new one.
    with_log_level(log_level) {
        LogLevel.assert(log_level);
        this.#level = log_level;
        return this;
    }

    // We'll talk about rolling config in just a bit, bare with me for now.
    with_rolling_config(rolling_config) {
       this.#rolling_config = RollingConfig.from_json(config);
       return this;
    }

    with_file_prefix(file_prefix) {
        if (typeof file_prefix =/= "string") {
            throw new Error(`file_prefix must be a string. Unsupported param
            ↪  ${JSON.stringify(file_prefix)}`);
        }

        this.#file_prefix = file_prefix;
        return this;
    }
    ...
}
```

Static methods in JavaScript are methods that belong to a class instead of an instance of the class. They can be used without creating an instance of the class and are often used for utility functions or operations that don't need any state. To create a static method in a class, use the `static` keyword before the method definition. For example:

```
class MyClass {
  static my_static_method() {
    console.log("This is a static method.");
  }
}

MyClass.my_static_method();
```

One thing to keep in mind is that you cannot access the `this` keyword inside a static method. This is because `this` refers to the instance of the class, and static methods are not called on an instance.

Subclasses can also inherit static methods, but they cannot be used on instances of the subclass. They are useful for organizing code and providing a namespace for related utility functions.

You may notice a difference now. Every method that we added is only responsible to validate a single input/argument. It does not care about any other options, whether they are correct or not.

4.8 jsdoc comments

If you're writing vanilla javascript, you may have trouble with the auto-completion or intellisense feature that most IDE provide, when working with multiple files. This is because javascript has no types (except primitives). Everything is an object. But don't we deserve those quality of life features if we're writing vanilla JS? Of course, we do. That's where `jsdoc` saves us.

We will not cover the entire feature set of `jsdoc`, but only focus on what we need for this particular purpose. We are concerned with two things: the parameter and the return type. This is because if a function returns a type, our auto-completion feature will not work across multiple files and will not display other associated methods of that return type in the dropdown.

Let's fix it.

```
/**
  * @param {string} file_prefix The file prefix to be set.
  * @returns {LogConfig} The current instance of LogConfig.
  * @throws {Error} If the file_prefix is not a string.
```

```
     */
    with_file_prefix(file_prefix) {
        if (typeof file_prefix !== "string") {
            throw new Error(`file_prefix must be a string. Unsupported param
            ↪ ${JSON.stringify(file_prefix)}`);
        }

        this.#file_prefix = file_prefix;
        return this;
    }
```

We create `jsdoc` comments with multi-line comment format using `/** ... */`. Then specify a tag using `@`. In the code snippet above, we specified three tags - `@params`, `@returns` and `@throws`. The tags have the following syntax

```
@tag {Type} <argument> <description>
```

The `Type` is the actual type of the `argument` that you're referring to. In our case it's the argument `file_prefix` in the method `with_file_prefix`. The type for that is `string`. The description is the documentation part for that particular parameter.

Here's the `jsdoc` comments with `with_log_level` method

```
    /**
     * @param {LogLevel} log_level The log level to be set.
     * @returns {LogConfig} The current instance of LogConfig.
     */
    with_log_level(log_level) {
        LogLevel.assert(log_level);
        this.#level = log_level;
        return this;
    }
```

I'll be not including the `jsdoc` comments to make the code snippets short, and easier to read. However, if you're writing vanilla javascript, it's a good practice to start incorporating these into your work flow. They'll save you a lot of time! There's much more than this that `jsdoc` helps us with. You can go through the documentation of `jsdoc` here.

4.9 The `RollingConfig` class

The **`RollingConfig`** class is going to be a vital part of our logging system that helps manage log files. It does this by rotating or rolling log files based on a set time interval or the size of file. This ensures that log files don't become too large and hard to handle.

The **`RollingConfig`** class's main purpose is to define settings for the log file's rolling process. This includes how often log files should be rolled, the maximum size of log files before they are rolled, and other relevant settings. By doing this, it helps keep log files organized and manageable while still preserving the historical data needed for analysis, debugging, and monitoring.

The **`RollingConfig`** class typically includes the two key features:

1. **Rolling Time Interval:** This setting determines how frequently log files are rolled. For example, you might set the logger to roll log files every few minutes, hours, or days, depending on how detailed you need your logs to be.
2. **Maximum File Size:** In addition to time-based rolling, the **`RollingConfig`** class may also support size-based rolling. When a log file exceeds a certain size limit, a new log file is created, with a new prefix to let you distinguish between different log files.

Before creating the `RollingConfig` class. Let's create 2 utility helper classes - `RollingSizeOptions` and `RollingTimeOptions`. As the name suggests, they are only going to support the `RollingConfig` class.

4.9.1 The `RollingSizeOptions` class

```
// index.js

class RollingSizeOptions {
  static OneKB = 1024;
  static FiveKB = 5 * 1024;
  static TenKB = 10 * 1024;
  static TwentyKB = 20 * 1024;
  static FiftyKB = 50 * 1024;
  static HundredKB = 100 * 1024;

  static HalfMB = 512 * 1024;
  static OneMB = 1024 * 1024;
  static FiveMB = 5 * 1024 * 1024;
  static TenMB = 10 * 1024 * 1024;
  static TwentyMB = 20 * 1024 * 1024;
```

```
  static FiftyMB = 50 * 1024 * 1024;
  static HundredMB = 100 * 1024 * 1024;

  static assert(size_threshold) {
    if (typeof size_threshold =/= "number" || size_threshold < RollingSizeOptions.OneKB) {
      throw new Error(`size_threshold must be at-least 1 KB. Unsupported param
      ↪  ${JSON.stringify(size_threshold)}`);
    }
  }
}
```

I've set some defaults, to make it easy for the clients of our library to use them. Instead of them having to declare an extra constant, they can quickly use `RollingSizeOptions.TenKB` or whatever they wish. However, they can also specify a number as a value, and that's where our `RollingSizeOptions.assert()` helper is going to do the validation for us.

4.9.2 The `RollingTimeOptions` class

```
// index.js

class RollingTimeOptions {
  static Minutely = 60; // Every 60 seconds
  static Hourly = 60 * this.Minutely;
  static Daily = 24 * this.Hourly;
  static Weekly = 7 * this.Daily;
  static Monthly = 30 * this.Daily;
  static Yearly = 12 * this.Monthly;

  static assert(time_option) {
    if (![this.Minutely, this.Hourly, this.Daily, this.Weekly, this.Monthly,
    ↪  this.Yearly].includes(time_option)) {
      throw new Error(
        `time_option must be an instance of RollingConfig. Unsupported param
        ↪  ${JSON.stringify(time_option)}`
      );
    }
  }
```

```
}
```

4.10 Finishing up the `RollingConfig` class

It's time to create our `RollingConfig` class. Let's add some basic functionality in it for now.

```
// index.js
class RollingConfig {
  #time_threshold = RollingTimeOptions.Hourly;
  #size_threshold = RollingSizeOptions.FiveMB;

  static assert(rolling_config) {
    if (!(rolling_config instanceof RollingConfig)) {
      throw new Error(
        `rolling_config must be an instance of RollingConfig. Unsupported param
        ↪  ${JSON.stringify(rolling_config)}`
      );
    }
  }

  // Provide a helper method for the clients, so instead of doing `new RollingConfig()`
  // they can simply use `RollingConfig.with_defaults()` that too without specifying the
  // `new` keyword.
  static with_defaults() {
    return new RollingConfig();
  }

  // Utilizing the `Builder` pattern here, to first verify that the size is valid.
  // If yes, set the size, and return the current instance of the class.
  // If it's not valid, throw an error.
  with_size_threshold(size_threshold) {
    RollingSizeOptions.assert(size_threshold);
    this.#size_threshold = size_threshold;
    return this;
  }

  // Same like above, but with `time`.
```

```
  with_time_threshold(time_threshold) {
    RollingTimeOptions.assert(time_threshold);
    this.#time_threshold = time_threshold;
    return this;
  }

  // Build from a `json` object instead of the `Builder`
  static from_json(json) {
    let rolling_config = new RollingConfig();

    Object.keys(json).forEach((key) => {
      switch (key) {
        case "size_threshold":
          rolling_config = rolling_config.with_size_threshold(json[key]);
          break;
        case "time_threshold":
          rolling_config = rolling_config.with_time_threshold(json[key]);
          break;
      }
    });

    return rolling_config;
  }
}
```

The `RollingConfig` class is ready to be used. It has no functionality, and is merely a configuration for our logger. It's useful to add a suffix like `Config`, `Options` for things that have no business logic inside them. It's generally a good design practice to stay focused on your naming conventions.

4.10.1 Let's recap

- `RollingConfig` - A class that maintains the configuration on how often a new log file file should be rolled out. It is based on the `RollingTimeOptions` and `RollingSizeOptions` utility classes which define some useful constants as well as an `assert()` method for the validation.
- `LogConfig` - A class that groups all other configurations into one giant class. This has a couple of private member variables - `#level` which is going to be of type `LogLevel` and keeps track of what logs should be written and what ignored; `#rolling_config` which is going to store the `RolllingConfig` for

our logger; `#file_prefix` will be used to prefix log files.

  - `with_defaults` constructs and returns a new `LogConfig` object with some default values.
  - `with_log_level`, `with_file_prefix` and `with_rolling_config` mutates the current object after testing whether the input provided is valid. The example of what we learnt above - a `Builder` pattern.
  - `assert` validation for the `LogConfig` class.

- `Logger` - The backbone of our logger. It hardly has any functionality now, but this is the main class of our library. This is responsible to do all the hard work.

4.11 Adding more useful methods in the `LogConfig` class

The `LogConfig` class looks fine. But it's missing out on a lot of other features. Let's add them one by one.

Firstly, not everyone is a fan of builder pattern, many people would like to pass in an object and ask the library to parse something useful out of it. It's generally a very good practice to expose various ways to do a particular task.

We are going to provide an ability to create a `LogConfig` object from a json object.

```
// index.js

...

class LogConfig {
    ...

    with_rolling_config(config) {
        this.#rolling_config = RollingConfig.from_json(config);
        return this;
    }

    static from_json(json) {
        // Create an empty LogConfig object.
        let log_config = new LogConfig();

        // ignore the keys that aren't needed for our purposes.
        // if a key matches, let's set it to the provided value.
        Object.keys(json).forEach((key) => {
```

```
            switch (key) {
                case "level":
                    log_config = log_config.with_log_level(json[key]);
                    break;
                case "rolling_config":
                    log_config = log_config.with_rolling_config(json[key]);
                    break;
                case "file_prefix":
                    log_config = log_config.with_file_prefix(json[key]);
                    break;
            }
        });

        // return the mutated log_config object
        return log_config;
    }

    ...
}

...
```

Now we can call it like this -

```
const json_config = { level: LogLevel.Debug };
const config = LogConfig.from_json(json_config).with_file_prefix("Testing");

// or

const config = LogConfig.from_json({ level: LogLevel.Debug }).with_file_prefix('Test');

// or

const config = LogConfig.with_defaults().with_log_level(LogLevel.Critical);

// Try to add an invalid value
const config = LogConfig.from_json({ level: 'eh?' }); // fails
```

```
Error: log_level must be an instance of LogLevel. Unsupported param "eh?"
    at LogLevel.assert (/Users/ishtmeet/Code/logtar/index.js:251:19)
    at LogConfig.with_log_level (/Users/ishtmeet/Code/logtar/index.js:177:18)
    at /Users/ishtmeet/Code/logtar/index.js:143:45
```

The API of our library is already looking solid. But there's one last thing that we wish to have as a convenience method to build `LogConfig` from. It's from a config file. Let's add that method

```
// import the `node:fs` module to use the `readFileSync`
const fs = require('node:fs')

class LogConfig {
    ...

    /**
     * @param {string} file_path The path to the config file.
     * @returns {LogConfig} A new instance of LogConfig with values from the config file.
     * @throws {Error} If the file_path is not a string.
     */
    static from_file(file_path) {
        // `fs.readFileSync` throws an error if the path is invalid.
        // It takes care of the OS specific path handling for us. No need to
        // validate paths by ourselves.
        const file_contents = fs.readFileSync(file_path);

        // Send this over to our `from_json` method to do the rest
        return LogConfig.from_json(JSON.parse(file_contents));
    }

    ...
}
```

Do you notice how we reused the `from_json` method to parse the json into a `LogConfig` object? This is one thing you have to keep in mind while building good and maintainable APIs. Avoid code duplication, and make the methods/helpers re-usable. As much as you can.

4.11.1 Why `readFileSync`?

Loggers are usually initialized once when the program starts, and are not usually created after the initialization phase. As such, using `readFileSync` over the asynchronous version (readFile) can provide several benefits in this specific case.

`readFileSync` operates synchronously, meaning it blocks the execution of the code until the file reading is complete. For logger configuration setup, this is often desired because the configuration is needed to initialize the logger properly before any logging activity begins, since our application will be using the logger internally.

We cannot let the application start before initializing the logger. Using asynchronous operations like `readFile` could introduce complexities in managing the timing of logger initialization.

Let's test using a config file. Create a `config.demo.json` file with the following contents

```
{
  "level": 3,
  "file_prefix": "My_Prefix_",
  "rolling_config": {
    "size_threshold": 1024,
    "time_threshold": 3600
  }
}
```

Since we have added the support for files, the following code will work now

```
const config = LogConfig.from_file("./config.demo.json");
const logger = Logger.with_config(config);
```

Everything works as expected.

> Note: The entire code we write here can be found at the src/chapter_04.0 directory. This will be a single file, and we'll refactor in subsequent chapters.

4.12 Refactoring the code

> The code for the entire chapter can be found at the src/chapter_04.1 directory

In this section, we will explore how to improve the organization and maintenance of our library before introducing more features. Currently, all the code is in one file i.e **`index.js`**. `index.js` also serves as the entry point of our project. We will show how to move the code to multiple files without changing how it works.

4.12.1 The Need for Refactoring

The code has become too large and difficult to manage. This chapter covers the benefits of breaking it into smaller files.

Splitting the code into separate files creates a more organized and manageable codebase. Each part should have a clear responsibility, making it easier to work with and understand. This simplification lays the foundation for future improvements, ensures that the project remains consistent and easy to work with, and allows for the introduction of new features.

Some of the key benefits of organizing/splitting your code into smaller re-usable pieces:

1. **Modularity:** Breaking the code into smaller files helps us manage each component better. This way, the codebase is easier to understand and work with.
2. **Readability:** Smaller files are easier to read and understand. Even people who haven't written any code in your library can quickly understand each file's purpose and contents.
3. **Maintainability:** When the codebase is organized into separate files by functionality, it becomes easier to maintain and update. Changes are limited to specific modules, reducing the risk of unintended consequences.
4. **Testing:** Individual components can be tested separately when code is in separate files. This leads to more thorough testing and fewer interdependent tests. (We'll cover testing in a later chapter of this book.)

4.12.2 Creating Separate Files

Let's work together to split the code in **`index.js`** into separate files for each class and utility. First, create a new directory called `lib`. Inside the `lib` directory, create two folders named `utils` and `config`. Add a file `logtar.js` inside the root of the `lib` directory.

Inside the `utils` directory, create two files named `rolling-options.js` and `log-level.js`. Inside the `config` directory, create two files named `rolling-config.js` and `log-config.js`.

Finally, create a file named `logger.js` at the root of the directory, where the `index.js` and `package.json` files are located.

Your directory structure should now look something like this:

```
lib/
├── logtar.js
├── logger.js
├── utils/
│     ├── rolling-options.js
│     ├── log-level.js
├── config/
│     ├── rolling-config.js
│     ├── log-config.js
index.js (entry point)
package.json
```

4.12.2.1 Explanation

The `logtar.js` file serves as the key file that exports all the necessary functionality to the client.

The `logger.js` file exports our `Logger` class and some related functionality.

The `utils/rolling-options.js` file exports our `RollingSizeOptions` and `RollingTimeOptions` classes.

The `index.js` file only contains a single line of code:

```
module.exports = require("./lib/logtar");
```

The other files export functionality based on their names.

> Note: If you are not working with TypeScript and are using vanilla JavaScript, get into the habit of using `JSDoc` as much as possible. Use it for every function's argument and return type. Be explicit. This may take a bit of time, but it will be convenient in the long run. Using `JSDoc` will make your workflow much smoother as your project grows.

4.12.3 The `index.js` file

Here are the contents inside the `index.js` file

```
module.exports = require("./lib/logtar");
```

4.12.4 The `lib/logtar.js` file

```
module.exports = {
  Logger: require("./logger").Logger,
  LogConfig: require("./config/log-config").LogConfig,
  RollingConfig: require("./config/rolling-config").RollingConfig,
  LogLevel: require("./utils/log-level").LogLevel,
  RollingTimeOptions: require("./utils/rolling-options").RollingTimeOptions,
  RollingSizeOptions: require("./utils/rolling-options").RollingSizeOptions,
};
```

4.12.5 The `lib/logger.js` file

```
const { LogConfig } = require("./config/log-config");
const { LogLevel } = require("./utils/log-level");

class Logger {
  /**
   * @type {LogConfig}
   */
  #config;

  /**
   * @returns {Logger} A new instance of Logger with default config.
   */
  static with_defaults() {
    return new Logger();
  }

  /**
   *
   * @param {LogConfig} log_config
   * @returns {Logger} A new instance of Logger with the given config.
   */
  static with_config(log_config) {
    return new Logger(log_config);
  }
```

```
  /**
   * @param {LogConfig} log_config
   */
  constructor(log_config) {
    log_config = log_config || LogConfig.with_defaults();
    LogConfig.assert(log_config);
    this.#config = log_config;
  }

  /**
   * @returns {LogLevel} The current log level.
   */
  get level() {
    return this.#config.level;
  }
}

module.exports = { Logger };
```

4.12.6 The `lib/config/log-config.js` file

```
const fs = require("node:fs");

const { LogLevel } = require("../utils/log-level");
const { RollingConfig } = require("./rolling-config");

class LogConfig {
  /**
   * @type {LogLevel}
   * @private
   * @description The log level to be used.
   */
  #level = LogLevel.Info;

  /**
```

```
 * @type RollingConfig
 * @private
 */
#rolling_config;

/**
 * @type {string}
 * @private
 * @description The prefix to be used for the log file name.
 *
 * If the file prefix is `MyFilePrefix_` the log files created will have the name
 * `MyFilePrefix_2021-09-01.log`, `MyFilePrefix_2021-09-02.log` and so on.
 */
#file_prefix = "Logtar_";

constructor() {
  this.#rolling_config = RollingConfig.with_defaults();
}

/**
 * @returns {LogConfig} A new instance of LogConfig with default values.
 */
static with_defaults() {
  return new LogConfig();
}

/**
 * @param {string} file_path The path to the config file.
 * @returns {LogConfig} A new instance of LogConfig with values from the config file.
 * @throws {Error} If the file_path is not a string.
 */
static from_file(file_path) {
  const file_contents = fs.readFileSync(file_path);
  return LogConfig.from_json(JSON.parse(file_contents));
}

/**
```

```
 * @param {Object} json The json object to be parsed into {LogConfig}.
 * @returns {LogConfig} A new instance of LogConfig with values from the json object.
 */
static from_json(json) {
  let log_config = new LogConfig();
  Object.keys(json).forEach((key) => {
    switch (key) {
      case "level":
        log_config = log_config.with_log_level(json[key]);
        break;
      case "rolling_config":
        log_config = log_config.with_rolling_config(json[key]);
        break;
      case "file_prefix":
        log_config = log_config.with_file_prefix(json[key]);
        break;
    }
  });
  return log_config;
}

/**
 * @param {LogConfig} log_config The log config to be validated.
 * @throws {Error} If the log_config is not an instance of LogConfig.
 */
static assert(log_config) {
  if (arguments.length > 0 && !(log_config instanceof LogConfig)) {
    throw new Error(`log_config must be an instance of LogConfig. Unsupported param
    ↪ ${JSON.stringify(log_config)}`);
  }
}

/**
 * @returns {LogLevel} The current log level.
 */
get level() {
  return this.#level;
```

```
}

/**
 * @param {LogLevel} log_level The log level to be set.
 * @returns {LogConfig} The current instance of LogConfig.
 * @throws {Error} If the log_level is not an instance of LogLevel.
 */
with_log_level(log_level) {
  LogLevel.assert(log_level);
  this.#level = log_level;
  return this;
}

/**
 * @returns {RollingConfig} The current rolling config.
 */
get rolling_config() {
  return this.#rolling_config;
}

/**
 * @param {RollingConfig} config The rolling config to be set.
 * @returns {LogConfig} The current instance of LogConfig.
 * @throws {Error} If the config is not an instance of RollingConfig.
 */
with_rolling_config(config) {
  this.#rolling_config = RollingConfig.from_json(config);
  return this;
}

/**
 * @returns {String} The current max file size.
 */
get file_prefix() {
  return this.#file_prefix;
}
```

```
  /**
   * @param {string} file_prefix The file prefix to be set.
   * @returns {LogConfig} The current instance of LogConfig.
   * @throws {Error} If the file_prefix is not a string.
   */
  with_file_prefix(file_prefix) {
    if (typeof file_prefix !== "string") {
      throw new Error(`file_prefix must be a string. Unsupported param
      ↪  ${JSON.stringify(file_prefix)}`);
    }

    this.#file_prefix = file_prefix;
    return this;
  }
}

module.exports = { LogConfig };
```

4.12.7 The `lib/config/rolling-config.js` file

```
const { RollingTimeOptions, RollingSizeOptions } = require("../utils/rolling-options");

class RollingConfig {
  /**
   * Roll/Create new file every time the current file size exceeds this threshold in `seconds`.
   *
   * @type {RollingTimeOptions}
   * @private
   *
   */
  #time_threshold = RollingTimeOptions.Hourly;

  /**
   * @type {RollingSizeOptions}
   * @private
   */
```

```
#size_threshold = RollingSizeOptions.FiveMB;

/**
 * @returns {RollingConfig} A new instance of RollingConfig with default values.
 */
static with_defaults() {
  return new RollingConfig();
}

/**
 * @param {number} size_threshold Roll/Create new file every time the current file size exceeds
↪  this threshold.
 * @returns {RollingConfig} The current instance of RollingConfig.
 */
with_size_threshold(size_threshold) {
  RollingSizeOptions.assert(size_threshold);
  this.#size_threshold = size_threshold;
  return this;
}

/**
 * @param {time_threshold} time_threshold Roll/Create new file every time the current file size
↪  exceeds this threshold.
 * @returns {RollingConfig} The current instance of RollingConfig.
 * @throws {Error} If the time_threshold is not an instance of RollingTimeOptions.
 */
with_time_threshold(time_threshold) {
  RollingTimeOptions.assert(time_threshold);
  this.#time_threshold = time_threshold;
  return this;
}

/**
 * @param {Object} json The json object to be parsed into {RollingConfig}.
 * @returns {RollingConfig} A new instance of RollingConfig with values from the json object.
 * @throws {Error} If the json is not an object.
 */
```

```
  static from_json(json) {
    let rolling_config = new RollingConfig();

    Object.keys(json).forEach((key) => {
      switch (key) {
        case "size_threshold":
          rolling_config = rolling_config.with_size_threshold(json[key]);
          break;
        case "time_threshold":
          rolling_config = rolling_config.with_time_threshold(json[key]);
          break;
      }
    });

    return rolling_config;
  }
}

module.exports = { RollingConfig };
```

4.12.8 The `lib/utils/log-level.js` file

```
class LogLevel {
  static #Debug = 0;
  static #Info = 1;
  static #Warn = 2;
  static #Error = 3;
  static #Critical = 4;

  static get Debug() {
    return this.#Debug;
  }

  static get Info() {
    return this.#Info;
  }
```

```
  static get Warn() {
    return this.#Warn;
  }

  static get Error() {
    return this.#Error;
  }

  static get Critical() {
    return this.#Critical;
  }

  static assert(log_level) {
    if (![this.Debug, this.Info, this.Warn, this.Error, this.Critical].includes(log_level)) {
      throw new Error(`log_level must be an instance of LogLevel. Unsupported param
      ↪ ${JSON.stringify(log_level)}`);
    }
  }
}

module.exports = { LogLevel };
```

4.12.9 The `lib/utils/rolling-options.js` class

```
class RollingSizeOptions {
  static OneKB = 1024;
  static FiveKB = 5 * 1024;
  static TenKB = 10 * 1024;
  static TwentyKB = 20 * 1024;
  static FiftyKB = 50 * 1024;
  static HundredKB = 100 * 1024;

  static HalfMB = 512 * 1024;
  static OneMB = 1024 * 1024;
  static FiveMB = 5 * 1024 * 1024;
```

```
  static TenMB = 10 * 1024 * 1024;
  static TwentyMB = 20 * 1024 * 1024;
  static FiftyMB = 50 * 1024 * 1024;
  static HundredMB = 100 * 1024 * 1024;

  static assert(size_threshold) {
    if (typeof size_threshold !== "number" || size_threshold < RollingSizeOptions.OneKB) {
      throw new Error(`size_threshold must be at-least 1 KB. Unsupported param
      ↪  ${JSON.stringify(size_threshold)}`);
    }
  }
}

class RollingTimeOptions {
  static Minutely = 60; // Every 60 seconds
  static Hourly = 60 * this.Minutely;
  static Daily = 24 * this.Hourly;
  static Weekly = 7 * this.Daily;
  static Monthly = 30 * this.Daily;
  static Yearly = 12 * this.Monthly;

  static assert(time_option) {
    if (![this.Minutely, this.Hourly, this.Daily, this.Weekly, this.Monthly,
    ↪  this.Yearly].includes(time_option)) {
      throw new Error(
        `time_option must be an instance of RollingConfig. Unsupported param
        ↪  ${JSON.stringify(time_option)}`
      );
    }
  }
}

module.exports = {
  RollingSizeOptions,
  RollingTimeOptions,
};
```

See how we can still benefit from the strong jsdoc type completion for our classes, even if they exist in different files? This isn't something achievable with regular JavaScript – all credit goes to jsdoc.

The code for the entire chapter can be found at the src/chapter_04.1 directory

4.13 Writing logs

The code for the entire chapter can be found at the src/chapter_04.2 directory

We've covered how to build the core utility helpers that will help us construct our logging library in a more modular way. However, we haven't gotten to the fun part of actually writing logs yet.

You might be wondering how we can call this a logging library if we haven't even learned how to write logs with it. Don't worry, we're about to get to that!

Before we begin writing logs to files, let's first introduce some concepts related to file-based logging and how we can make it more efficient and faster.

4.13.1 1. Re-using the File Handle

When logging to a file, it's important to manage the file handle efficiently. The file handle is the connection between our code and the log file on disk. Opening and closing the file for every log entry can be slow and use up resources. To avoid this, we want to reuse the file handle throughout the logging process.

To do this, we open the file handle once when we start the logging library and keep it open until the logging is done. This means we don't need to keep opening and closing the file, which can speed up our logging and save resources.

4.13.2 2. Log Rotation

Log rotation is a critical strategy in file-based logging to manage log files over time. As your application generates more log data, log files can become large and unwieldy. Log rotation involves creating new log files periodically or based on certain conditions, and optionally archiving or deleting older log files. This helps keep the log files at a manageable size, ensures easier log analysis, and prevents running out of disk space.

4.13.3 3. Asynchronous Logging

In network based apps, especially those with high levels of concurrency or those that involve asynchronous operations, implementing asynchronous logging can be beneficial, and that's the only reason we're using `node:fs/promises` instead of `node:fs`. Asynchronous logging ensures that the act of writing logs doesn't block or slow down the main execution thread of your application. This can prevent performance bottlenecks and maintain the responsiveness of your application.

When logging is performed synchronously, each log entry is written immediately to the log file or output, which can introduce delays and impact the overall performance of your application. Asynchronous logging, on the other hand, involves buffering log messages and writing them to the log file in batches or on a separate thread or process.

> We'll need to do more optimization than just using asynchronous file writing. Specifically, we should store the entire log contents in memory and write them periodically. This will make the process extremely fast and ensure that it doesn't consume too much memory.

By decoupling the logging process from the main application logic, we can achieve several advantages:

- **Improved Performance:** Asynchronous logging allows the main application thread to continue executing without waiting for log writes to complete. This can be crucial for applications that require high responsiveness and throughput.
- **Reduced I/O Overhead:** Writing log messages to disk can be an I/O-intensive operation. By batching multiple log messages together and writing them in one go, you reduce the frequency of disk I/O operations, which can improve efficiency.
- **Better Resource Utilization:** Asynchronous logging allows you to optimize resource utilization, such as managing concurrent log writes, handling errors without disrupting the main flow, and efficiently managing file handles.
- **Enhanced Scalability:** Applications with multiple threads or processes benefit from asynchronous logging because it minimizes contention for resources like the log file. This is particularly valuable in scenarios where multiple components are concurrently generating log messages

4.13.4 4. Getting Caller Information (Module and Line Number)

Including caller information, such as the file name and line number from which a log message originated, can significantly enhance the effectiveness of our logging library. This feature provides contextual insight into where specific events occurred in the codebase, making it easier to identify the source of issues and troubleshoot them.

When an application encounters an error or unexpected behavior, having access to the module and line number associated with the log message allows developers to:

- Quickly locate the exact location in the code where the event occurred.
- Understand the sequence of events leading up to the issue.
- Make precise code adjustments to fix problems.

Implementing this feature might involve using techniques from the programming language's stack trace or introspection mechanisms. Here's a high-level overview of how you could achieve this:

1. **Capture Caller Information:** When a log message is generated, our logging library will retrieve the caller information, including the module and line number. We'll learn it in a bit that how can we do that.
2. **Format the Log Message:** Combine the captured caller information with the log message and other relevant details like timestamp and log level.
3. **Output the Log Message:** Write the formatted log message to the desired output destinations, ensuring that the caller information is included.

Enough of the theory, let's start writing logs.

4.13.5 Testing our current API

Let's start with building small features that we need to have on our `Logger` class. Before that, we're going to do some testing on whatever we've built till now.

To do the testing, we'll create a new file `test.js` and a `config.json`. `test.js` will hold our code that we write while making use of our `logtar` library. The `config.json` will used to store our config for `LogConfig` as a `json` object.

```
// test.js

const { Logger, LogConfig } = require("./index");

const logger = Logger.with_config(LogConfig.from_file("./config.json"));
```

The `config.json` file has the following contents. You may try to tweak the values as well. Try putting in the values that aren't supported by us, and see whether the `assert` methods that we created actually crash the program or not?

```
// config.json

{
```

```
    "level": 3,
    "file_prefix": "LogTar_",
    "rolling_config": {
        "size_threshold": 1024000,
        "time_threshold": 86400
    }
}
```

Try executing the `test.js` file.

```
$ node test.js
```

Nothing happens. But this proves that the whole setup for our `Logger` and other classes is working perfectly fine.

To check whether the config is being loaded properly, we can create a couple of getter methods on our `Logger` class.

```
// file: lib/logger.js

class Logger {
    ...
    get level() {
        return this.#config.level;
    }

    get file_prefix() {
        return this.#config.file_prefix;
    }

    get time_threshold() {
        return this.#config.rolling_config.time_threshold;
    }

    get size_threshold() {
        return this.#config.rolling_config.size_threshold;
    }
    ...
}
```

Now, in `test.js` print those out to the standard output.

```
$node test.js

### outputs
LogTar_
1024000
86400
3
```

Perfect! Everything works fine. But what happens if I try to put a value that's not supported by our Logger API? Let's change the `time_threshold` to `400` in `config.json` and re-run the app.

```
$node test.js

### outputs
Error: time_option must be an instance of RollingConfig. Unsupported param 400
    at Function.assert (/Users/ishtmeet/Code/logtard/lib/utils/rolling-options.js:36:19)
```

Why? Let's take a quick look at our `RollingTimeOptions` utility class

```
// file: lib/util/rolling-options.js

class RollingTimeOptions {
    static Minutely = 60; // Every 60 seconds
    ... // Other options

    // Throw an error when the user sets the value which isn't one of those
    static assert(time_option) {
        if (![this.Minutely, this.Hourly, this.Daily, this.Weekly, this.Monthly,
        ↪  this.Yearly].includes(time_option)) {
            throw new Error(
                `time_option must be an instance of RollingConfig. Unsupported param
                ↪  ${JSON.stringify(time_option)}`
            );
        }
    }
}
```

You might argue that this isn't the best dev experience. However, I think it is. We should always constrain

over how configurable our library is. You would hardly need any other duration other than we specified i.e `Minutely`, `Hourly`, `Daily` etc. and anything more than `Yearly`

Let's change the `time_threshold` in `config.json` back to `86400` which means 1 day.

4.13.6 Implementing logging methods

Since our logger supports 5 types of `log_level`s, let's write a public method for each one of those on our `Logger` class.

```
// file lib/logger.js

class Logger {
    ...

    debug(message) { console.log('Debug: %s', message) }

    info(message) { console.log('Info: %s', message) }

    warn(message) { console.log('Warn: %s', message) }

    error(message) { console.log('Error: %s', message) }

    critical(message) { console.log('Critical: %s', message) }

    ...
}
```

And in `test.js` call these methods

```
// file: test.js
const { Logger, LogConfig } = require('./index')

const logger = Logger.with_config(LogConfig.from_file('./config.json'));

console.log(logger.file_prefix);
console.log(logger.size_threshold);
console.log(logger.time_threshold);
console.log(logger.level);
```

```
logger.debug('Hello debug');
logger.info('Hello info');
logger.warn('Hello warning');
logger.error('Hello error');
logger.critical('Hello critical');


// outputs
LogTar_
1024000
86400
3
Debug: Hello debug
Info: Hello info
Warn: Hello warning
Error: Hello error
Critical: Hello critical
```

4.13.7 DRY (Don't Repeat Yourself)

The "Don't Repeat Yourself" (DRY) principle is a basic concept in software development that promotes code reusability and maintainability. The idea behind DRY is to avoid duplicating code or logic in multiple places within your codebase. Instead, you aim to create a single source for a particular piece of functionality, and whenever you need that functionality, you refer to that source.

DRY encourages developers to write clean, efficient, and modular code by:

- Reducing the chances of errors: Duplicated code increases the chances of mistakes or bugs when changes are made in one place but not in others.
- Simplifying maintenance: When a change is required, you only need to update the code in one place, making it easier to keep your codebase up-to-date and consistent.
- Enhancing readability: Code that is free from unnecessary duplication is easier to understand and follow, making it more accessible to other developers.

Although following the DRY principle is generally beneficial, there can be situations where duplication might not necessarily be a bad thing. Not every instance of code repetition needs to be eliminated, and striving for absolute **DRY**ness in all cases might lead to overcomplicated solutions or premature abstraction.

A guideline often mentioned is the "**Rule of Three**": If you find yourself repeating the same code or logic more than three times, it's a strong indication that you should consider refactoring that code into a reusable

function, class, or module. This threshold helps you strike a balance between reusability and pragmatic simplicity.

Trying too hard to avoid repeating code can make it so complicated that it's hard to understand. It's better to find a balance where your code can be used again and again, but it's still easy to read and work with.

4.13.8 The `log` method

We'll introduce a `private` member method called `log`. The clients don't need to know about what's going inside the library, so making the crucial/core methods private is a nice thing.

Update the code of your `Logger` class to include the `log` method. Also, update the code so that all other helper methods call the `log` method, thus avoiding the code duplication.

```
// lib/logger.js

const { LogLevel } = require("./utils/log-level");

class Logger {
    ...

    #log(message, log_level) {
        console.log('%s: %s', message, log_level)
    }

    debug(message) {
        this.#log(message, LogLevel.Debug);
    }

    info(message) {
        this.#log(message, LogLevel.Info);
    }

    warn(message) {
        this.#log(message, LogLevel.Warn);
    }

    error(message) {
        this.#log(message, LogLevel.Error);
    }
```

```
    critical(message) {
        this.#log(message, LogLevel.Critical);
    }
    ...
}
```

Executing the code gives us the desired output. The output includes the `LogLevel` as an integer. However, that's not helpful at all, we should be showing what the `LogLevel` is with a string representation of the level.

Let's introduce a new static method `to_string` inside the `LogLevel` class

```
// file: lib/utils/log-level.js

class LogLevel {
    ...
    static to_string(log_level) {
        const levelMap = {
            [this.Debug]: "DEBUG",
            [this.Info]: "INFO",
            [this.Warn]: "WARN",
            [this.Error]: "ERROR",
            [this.Critical]: "CRITICAL"
        };

        if (levelMap.hasOwnProperty(log_level)) {
            return levelMap[log_level];
        }

        throw new Error(`Unsupported log level ${log_level}`);
    }
    ...
}
```

Change the code inside the `log()` method of `Logger` class.

```
// lib/logger.js

const { LogLevel } = require("./utils/log-level");
```

```
class Logger {
    ...

    #log(message, log_level) {
        console.log('%s: %s', message, LogLevel.to_string(log_level))
    }
    ...
}
```

This outputs

```
Hello debug: DEBUG
Hello info: INFO
Hello warning: WARN
Hello error: ERROR
Hello critical: CRITICAL
```

Everything looks good.

4.13.9 Considering the `log_level` member variable

Notice that in our `config.json` we specified that the log level should be `3` that is `LogLevel.Error`. Specifying the log level means that we should only write logs that are equal to or above the specified level.

Imagine a production application, which is usually under a very heavy load. We'd like to specify the `level` as `LogLevel.Warn` or even `LogLevel.Info`. We don't care about the `LogLevel.Debug` logs at all. They might pollute the log files as the debug logs are usually very verbose.

We are going to add a small check to ignore the logs which are below the current `log_level`.

```
// file: lib/logger.js

class Logger {
    ...
    #log(message, log_level) {
        if (log_level < this.#config.level) {
            return;
        }

        console.log('%s: %s', message, LogLevel.to_string(log_level))
```

```
    }
    ...
}
```

On running, we get the following output.

```
Hello error: ERROR
Hello critical: CRITICAL
```

We only write logs based off the current logger's `log_level`

4.13.10 Writing to a file

We've been using `console.log()` to print the log messages to the standard output, or your terminal. However, there are many drawbacks to this approach.

1. **Lack of Persistence:** Console logs are ephemeral and disappear once the application session ends (when you close the terminal or exit out of an ssh connection). This makes it challenging to review logs for past sessions.
2. **Performance Impact:** Continuous console output can impact application performance, especially when generating a high volume of logs. It can slow down the application and interfere with its responsiveness. There are certain ways to get mitigate this, we'll talk about this in a later chapter.

Create a private `log_file_handle` member

```javascript
// file: lib/logger.js

const fs = require('node:fs/promises')

class Logger {
    ...

    /**
     * @type {fs.FileHandle}
     */
     #log_file_handle;

    ...
}
```

The `log_file_handle` member variable will be our opened file handle that we'll reference again and again

whenever we wish to write logs to a file. It will hold the opened log file, which will be created whenever the client initiates our logging library.

We'll expose a public method on the `Logger` called `init` and make that `async` so that it waits for the `init()` to finish initializing the file handle before moving further.

```
// file: lib/logger.js
const fs = require('node:fs/promises')

class Logger {
    ...

    #log_file_handle;

    async init() {
        const file_name = this.#config.file_prefix + new Date().toISOString().replace(/[\.:]+/g,
        ↪ "-") + ".log";
        this.#log_file_handle = await fs.open(file_name, "a+");
    }
}
```

There's a lot going on in this method. Let's break it down.

4.13.10.1 A small primer on Regular Expressions

1. The `init` method is responsible for initializing the logger, which includes creating or opening a log file with a dynamically generated name based on the current date and time.
2. `this.#config.file_prefix` is used to prefix the name of the log file, which is actually set in the `config.json` file or can be passed as a json object or utilizing the `Builder` pattern.
3. `new Date().toISOString()` generates a string representation of the current date and time in the ISO 8601 format, such as "2023-08-18T15:30:00.000Z".
4. `.replace(/[\.:]+/g, "-")` is a regular expression operation. Let's break down the regex:
    - the square brackets [] are used to define a character class. A character class is a way to specify a set of characters from which a single character can match. For example:
        - [abc] matches any single character that is either 'a', 'b', or 'c'.
        - [0-9] matches any single digit from 0 to 9.
        - [a-zA-Z] matches any single alphabetical character, regardless of case.

You can also use special characters within character classes to match certain character types, like `\d` for digits, `\w` for word characters, `\s` for whitespace, etc.

In this case we are looking for all the dot (`.`) characters and colon (`:`) characters in the string.

- `\.` matches a literal dot character. Since the dot (`.`) is a special character in regular expression, known as a wildcard. It matches any single character except for a newline character (`\n`). This is useful when you want to match any character, which is often used in pattern matching.

  However, in this case, we want to match a literal dot character in the string (the dot in the date-time format "00.000Z"). To achieve this, we need to escape the dot character by preceding it with a backslash (`\`). When you place a backslash before a special character, it indicates that you want to match the literal character itself, rather than its special meaning.

- `+` matches one or more of any character except newline. We match for all the characters following the dot (.) and the (:) character.

- `/g` is the global flag, indicating that the replacement should be applied to all occurrences of the pattern.

- So, the regex `[\.:]+` matches the dot or colon and all the repeated occurences of these 2 characters.

- The replacement `"-"` removes the dot and all characters after it and replaces it with a hyphen (`-`).

5. The result of the `replace` operation is a modified version of the ISO string, which now includes only the date and time portion, without the fractional seconds.

6. `.log` is appended to the modified date and time string to form the final log file name.

7. `await fs.open(file_name, "a+")` opens or creates the log file using the `fs.open` function with "a+" flag. We talked about the modes in a previous chapter

   - If the file doesn't exist, it will be created.
   - If the file exists, data can be appended to it.
   - The `"a+"` mode allows both reading and appending. Appending means, we begin writing to the end of the file instead of from the 1st line. However, if the file is empty, it starts from the beginning.

This code initializes the logger by creating or opening a log file with a name based on the current date. The regular expression is used to remove the fractional seconds from the ISO date string, and the resulting string is used as part of the log file name. This allows for creating a new log file every time the `init` method is called, typically representing logs for a specific time period.

Since regular expressions are one of the most important concepts which are usually neglected, we'll have an entire chapter dedicated to mastering regular expressions in case you'd like to get comfortable using them.

4.13.10.2 Testing the log file creation

Let us try to test the `init` method, and see if it creates a log file like we desire?

Before that, we'll look at a small nuance of calling an `async` function inside a `non-async` scope. Add the following log statement in the `init` method

```
// file: lib/logger.js

class Logger {
    ...

    async init() {
        const file_name = this.#config.file_prefix + new Date().toISOString().replace(/[\.:]+/g,
        ↪  "-") + ".log";
        this.#log_file_handle = await fs.open(file_name, "a+");
        console.log("File created.")
    }
    ...
}
```

And then call the `init` method from the `test.js`

```
// file: test.js

const { Logger, LogConfig } = require("./index");

const logger = Logger.with_config(LogConfig.from_file("./config.json"));
logger.init();
console.log("End of the file.");
```

However the program exits, without creating the log file.

```
$ node test.js
```

```
### Outputs
End of the file.
```

Why is it so? It is because we're using the `node:fs/promises` module, where all the functions are asynchronous. That means, they do not block any code and continue execution without waiting. This is the key to creating performant concurrent applications.

How do we fix it? Simply by `await`ing the `init()` method.

```
// file: test.js

const { Logger, LogConfig } = require("./index");

async function main() {
  const logger = Logger.with_config(LogConfig.from_file("./config.json"));
  await logger.init();
  console.log("End of the file");
}

main();
```

If you try to run now, the output is what you would've expected.

```
### Outputs
End of the file
```

You'll also notice that the log file has also been created! For me the log file was created in the root directory. The name of the log file is

`LogTar_2023-08-18T17-20-23-132Z.log`

`LogTar_` is the prefix that I specified in the `config.json` file. Following it is the timestamp of when the file was created. We also add a `.log` extension at the end. All working as expected.

4.13.11 Another gotcha

If you're paying attention, you'll have already figured out that the way we're providing path to a file is not a nice way. If we wanted to run the `test.js` file from a different directory, say `lib/config` we'll get an error.

```
$ cd lib/config && node ../../test.js
```

```
// outputs
Error: ENOENT: no such file or directory, open './config.json'
```

Let's fix this by using the `__dirname` global variable that we used earlier.

```
// file: test.js

const path = require("node:path");
const { Logger, LogConfig } = require("./index");

async function main() {
  const logger = Logger.with_config(LogConfig.from_file(path.join(__dirname, "config.json")));
  await logger.init();
  console.log("End of the file");
}

main();
```

Now if we try to run it, it works irrespective of the directory you're running the code from.

```
$ node ../../test.js
### Outputs
End of the file
```

4.13.12 Logs directory configuration

When we run our code using the `node ../../test.js` command, the log files are being created in the same place where we typed that command, which is the `lib/config` folder. This is not what we want. Our library is a tool that other people will use. We should have complete say over where the log files go. This is because we want to make sure our library works well no matter how others use it.

Normally, logs (like records of what our code is doing) are put in a special folder called "log." This folder sits at the main spot of the entire project.

By centralizing the log files within the `log` directory at the root of the project that includes our library, we achieve a more organized and modularized approach to logging. This ensures that the log files don't clutter the main project's structure and are conveniently located in one designated place.

4.13.12.1 Our first script

Inside the `package.json` file, add a script `start` with the value of `node test.js` and get rid of the `test` script.

```
// file: package.json

{
    ...
    "scripts": {
        "start": "node test.js"
    },
    ...
}
```

If you run `npm start` from any folder of the project, it runs the `node test.js` command from the directory the `package.json` file is located.

The logs output in the root of the directory. Let's create them inside the `log` directory, instead of the root directory.

Update the code inside `Logger.init` method:

```
// file: lib/logger.js

const path = require('node:path')

async init() {
    const file_name = this.#config.file_prefix + new Date().toISOString().replace(/[\.:]+/, "-")
    ↪  + ".log";
    this.#log_file_handle = await fs.open(path.join('logs', file_name), "a+");
}
```

If you do run `npm start`, it fails.

```
### Outputs
[Error: ENOENT: no such file or directory, open 'logs/LogTar_2023-08-18T19-14-46-162Z.log']
```

It's failing because it could not find the `log` directory. If we mention something like this in our library's readme:

```
Hey please create a `log` folder in order for this library to work
```

They'll simply ignore the library and use something better. Let's make the whole process streamlined. Irrespective of where our library's code live, whether it's inside the `node_modules` folder or in the same directory.

4.13.13 The `require` object

We've been using `require` throughout our code, which just imports a module, and exposes all it's exported functionality - whether it's a variable, a function or a class. You may think of it as a function since `require('node:path')` indeed looks like a function call.

But, require is not just a function!

`require` also acts like an object that has several useful properties and methods associated with it. When you use `require` in your code, you are utilizing both its function-like behavior to load modules and its object-like properties and methods to interact with modules and the module system.

For example:

- `require.resolve('module-name')`: Returns the path of the module without actually loading it.
- `require.cache`: Provides access to the cache of loaded modules.
- `require.main`: Provides access to the `Module` object representing the entry script loaded when the Node.js process launched. This is exactly what we need.

The reason `require` might feel like both an object and a function is because JavaScript allows functions to have properties. You can test this yourself

```
function my_fun() {
  console.log("hi");
}
my_fun.hey = "there";

console.log(my_fun.hey); // there
my_fun(); // 'hi'
```

4.13.14 Adding a new helper to create log directory

Create a new file called `helpers.js` inside the `lib/utils` folder.

```
// file: lib/utils/helpers.js

const fs_sync = require("node:fs");
```

```
const path = require("path");

/**
 * @returns {fs_sync.PathLike} The path to the directory.
 */
function check_and_create_dir(path_to_dir) {
  const log_dir = path.resolve(require.main.path, path_to_dir);
  if (!fs_sync.existsSync(log_dir)) {
    fs_sync.mkdirSync(log_dir, { recursive: true });
  }

  return log_dir;
}

module.exports = {
  check_and_create_dir,
};
```

Let's go through the `check_and_create_dir` function's code line by line.

1. The `path.resolve()` function creates an absolute path by combining a sequence of paths or path segments.

   It processes the sequence from right to left, with each subsequent path added to the beginning until an absolute path is created. For example, if the path segments are `/foo`, `/bar`, and `baz`, calling `path.resolve('/foo', '/bar', 'baz')` would return `/bar/baz` because `'/bar' + '/' + 'baz'` creates an absolute path, but `'baz'` does not.

   If, after processing all the segments in the given path, an absolute path hasn't been created yet, then the current working directory is used instead.

2. The `require.main.path` holds the absolute path to the directory of the entry point. The entry point is the `test.js` in our case. Or whatever file you specify while running `node file_name.js` command.

3. The `fs_sync.existsSync` function is checking if a given directory path exists in the file system. It's using the `log_dir` path (the result of path resolution) to see if the directory is already there.

4. If the directory doesn't exist, the `fs_sync.mkdirSync` function comes into play. It's used to create directories. The `{ recursive: true }` option ensures that not only the final directory is created, but also any necessary intermediate directories along the path. In case we wish to specify a log directory as `logs/new/today`

5. At the end, function returns the `log_dir`, which is the full path to the directory. Whether the directory existed before or was just created, this path is what you get as a result from the function.

4.13.15 Updating the `init` method

```
// file: lib/logger.js

class Logger {
    ...
    async init() {
        const log_dir_path = check_and_create_dir("logs")

        const file_name = this.#config.file_prefix + new
        ↪  Date().toISOString().replace(/[\.:]+/, "-") + ".log";

        // Open/Create the file at the specified log directory and
        // save the file handle to the `#log_file_handle` for further
        // use
        this.#log_file_handle = await fs.open(path.join(log_dir_path, file_name), "a+");
    }
    ...
}
```

4.13.16 Completing the `log` method

The current version of our `Logger.log` method looks like this:

```
#log(message, log_level) {
    if (log_level < this.#config.level) {
        return;
    }

    console.log('%s: %s', message, LogLevel.to_string(log_level))
}
```

Instead of logging to the standard output, let's write it to the log file that is created when the `Logger` is initialized.

```javascript
// file: lib/logger.js

class Logger {
    ...
    async #log(message, log_level) {
        // dont' write to the file if
        //     1. The `log_level` argument is less than the `#config.level` value
        //     2. If the `fd` (file descriptor) is either 0 or -1, which means the file
        //        descriptor is closed or not opened yet.
        if (log_level < this.#config.level || !this.#log_file_handle.fd) {
            return;
        }

        // write logs to the opened file
        await this.#log_file_handle.write(log_message);
    }
    ...
}
```

In the `test.js` file

```javascript
// file: test.js

const path = require("node:path");
const { Logger, LogConfig } = require("./index");

async function initialize_logger() {
  let logger = Logger.with_config(LogConfig.from_file(path.join(__dirname, "config.json")));
  await logger.init();

  return logger;
}
async function main() {
  let logger = await initialize_logger();
  logger.error("This is an error message");
}

main();
```

In `test.js` we create a new `async` function `initialize_logger` that creates a logger, `init`ializes it and returns it.

We call the `logger.error()` method to print the log to the file.

Let's run the code

```
$ npm start
```

A new log file will be created inside the `logs` directory. Open and see the contents -

```
This is an error message
```

Perfect! Everything seems to be working now. The logs are being saved, and we can be proud of it! But wait, the logs aren't useful. We don't know what are those logs, when did we write them and what function called the `logger.error()` method?

We'll take care of all this in the next chapter.

> The code for the entire chapter can be found at the src/chapter_04.2 directory

4.14 Capturing metadata

> The code for the entire chapter can be found at the src/chapter_04.3 directory

The Logger is writing to the file as expected, but there is an issue: it is not capturing any useful metadata. Here is an example of what our logs currently look like:

```
Hello this is a log of type ERROR
```

This is no different than using the standard console.log method. For example, we could write:

```
const logger =
  /* init logger */

  function create_subscription() {
    const my_log_message = "This is an error";
    logger.error(`[Error]: create_subscription() Line No. 69 ${my_log_message}`);
  };
```

This code appears to work. However, if you add more functionality above this function in the same file, the line number will change again and again, which is not ideal.

For a logging library, it's important to minimize the amount of unnecessary code that clients need to type out. We should do the heavy lifting for our clients, so they only need to call our logger.error (or other similar) method.

This will make our library easier to use and more user-friendly.

```
function my_deeply_nested_api_route() {
    logger.error('my error')
    logger.warn('my warning')
}
```

The output of the above code should look like this:

```
[2023-08-19T15:10:37.097Z] [ERROR]: my_deeply_nested_api_route
↪  (/Users/ishtmeet/Code/logtar/test.js:12) my error
[2023-08-19T15:10:37.097Z] [ERROR]: my_deeply_nested_api_route
↪  /Users/ishtmeet/Code/logtar/test.js:13 my warning
```

How cool would it be to achieve this functionality with a little bit of Javascript hack?

Before we dive into the details of how we can extract the function name or its line number, let's take a small break to understand what a call stack is. But before we understand what a call stack is, we need to understand what a stack is.

4.14.1 What is a Stack?

A stack is a data structure that is widely used in in programming. It is designed to store a collection of elements and is based on the Last In First Out (LIFO) principle. This means that the most recently added element to the stack is the first one to be removed.

Stacks are used in a variety of applications, from handling function calls to undoing/redoing actions in software applications. Stacks can also be implemented in different ways, such as using arrays or linked lists.

4.14.2 Examples of Stacks

Stacks are a common occurrence in everyday life, and here are some examples:

- A stack of books
- A stack of files
- A stack of pizzas

In each of these cases, the most recent item is placed on top of the stack while the oldest is located at the bottom. For example, to access the pizza at the bottom, you will need to remove all the pizzas above it in the stack.

4.14.3 The Call Stack

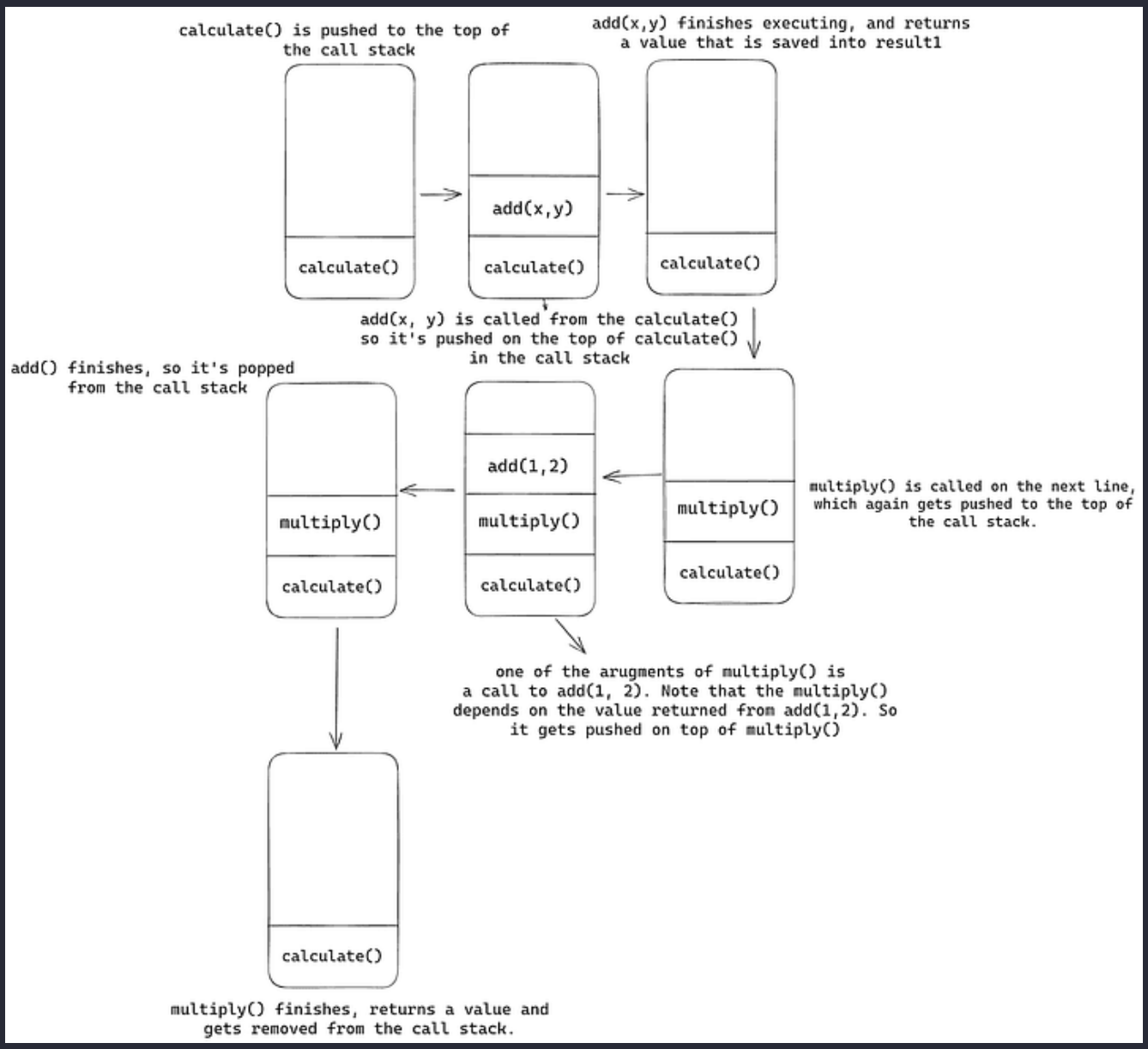
A call stack is a special type of stack that keeps track of function calls. Whenever a function is called, its information is pushed onto the call stack. When the function returns, its information is popped off the stack.

Here is an example of a call stack in JavaScript:

```
function add(a, b) {
  return a + b;
}

function multiply(a, b) {
  return a * b;
}

function calculate() {
  const x = 10;
  const y = 5;

  const result1 = add(x, y);
  const result2 = multiply(add(1, 2), result1);

  return result2;
}

calculate();
```

In the example provided, the `calculate` function calls two other functions (`add` and `multiply`). Each time a function is called, its information is added to the call stack. When a function returns, its information is removed from the stack.

To further illustrate this, consider the following graphic:



When the `calculate` function is called, its information is added to the top of the stack. The function then calls the `add` function, which adds its information to the top of the stack. The `add` function then returns, and its information is removed from the stack. The `multiply` function is then called, and its information is added to the top of the stack.

One important thing to note is, when multiply is called, the first argument is a call to `add(1, 2)`. That means we need to push the `add(..)` back to the top, above `multiply` in the call stack. When the `add` finishes executing, it's removed from the top.

This process continues, with each function call adding its information to the top of the stack and returning, with its information being removed from the stack.

This call stack is important because it allows the program to keep track of where it is in the execution of a program. If a function call is not properly removed from the stack, it can cause a stack overflow error, which

can crash the program.

> In compiled languages like C++ or Rust, the compiler is smart enough to inline the `add()` function above, and directly place the contents of the `add` function in the place of the function call. This can result in performance improvements by eliminating the overhead of a function call and return.

4.14.4 Getting the stack info

How do you even see the stack? Well, it's not that hard. When you `throw` an error, that error contains the information about the stack.

```
function main() {
  child();
}

function child() {
  grand_child();
}

function grand_child() {
  throw new Error("This is an error from grand_child()");
}

main();
```

If you run this file using `node your_file_name.js` it outputs this on the console

```
Error: This is an error from grand_child()
    at grand_child (/Users/ishtmeet/Code/logtard/test.js:10:11)
    at child (/Users/ishtmeet/Code/logtard/test.js:6:5)
    at main (/Users/ishtmeet/Code/logtard/test.js:2:5)
    at Object.<anonymous> (/Users/ishtmeet/Code/logtard/test.js:12:1)
    at Module._compile (node:internal/modules/cjs/loader:1198:14)
    ...
```

We only need to care about the top 5 lines. Rest of those are Node.js's internal mechanism of running and executing the files. Let's go through the error above line by line:

- The first line includes the error message `"This is an error from grand_child()"`, which is the cus-

tom message we provided when we used the `throw` statement in the `grand_child()` function.

- The second line of the stack trace indicates that the error originated in the `grand_child()` function. The error occurred at line 10, column 11 of the `test.js` file. This is where the `throw` statement is located within the `grand_child()` function. Therefore, `grand_child` was at the top of the stack when the error was encountered.
- The third line shows that the error occurred at line 6, column 5 of the `test.js` file. This line pinpoints where the `grand_child()` function is called within the `child()` function. That means, `child()` was the second top function on the call stack, below `grand_child()`.
- The fourth lines tells us that the `child()` function was called from within the `main()` function. The error occurred at line 2, column 5 of the `test.js` file.
- The line 5th tells that the `main()` function was called from the top-level of the script. This anonymous part of the trace indicates where the script execution starts. The error occurred at line 12, column 1 of the `test.js` file. This is where the `main()` function is called directly from the script.

This is called a stack trace. The throw new Error() statement prints the entire stack trace, which unwinds back through the series of function calls that were made leading up to the point where the error occurred. Each function call is recorded in reverse order, starting from the function that directly caused the error and going back to the initial entry point of the script.

This trace of function calls, along with their corresponding file paths and line numbers, provides developers with a clear trail to follow. It aids in identifying where and how the error originated.

This is exactly what we want to know where was the `logger.error` and the other methods are being called from.

4.14.5 Getting the `callee` name and the line number

How can we use the information above to obtain the line number from the client's code? Can you think about it?

Let's add the following in our `#log` method of the `Logger` class:

```
// file: lib/logger.js

class Logger {
    ...

    async #log(message, log_level) {
        if (log_level < this.#config.level || !this.#log.file_handle.fd) {
            return;
```

```
        }

        /* New code inserted */
        let stack_trace;
        try {
            throw new Error();
        } catch(error) {
            stack_trace = error.stack;
        }
        console.log(stack_trace)
        /* New code ends */

        await this.#log_file_handle.write(log_message);
    }

    ...
}
```

Try to execute the `test.js` file.

```
// file: test.js

const { Logger } = require("./index");

async function initialize() {
  const logger = Logger.with_defaults();
  await logger.init();
  return logger;
}

async function main() {
  let logger = await initialize();
  logger.critical("Testing");
}

main();
```

This outputs

```
Error
    at Logger.#log (/Users/ishtmeet/Code/logtard/lib/logger.js:98:19)
    at Logger.critical (/Users/ishtmeet/Code/logtard/lib/logger.js:141:18)
    at main (/Users/ishtmeet/Code/logtard/test.js:12:12)
```

Awesome. We now know who invoked the function. There are 4 lines. The important piece is on the last line.

```
at main (/Users/ishtmeet/Code/logtard/test.js:12:12)
```

This is where we called `logger.critical('Testing')`. However, you may think - "Yeah fine, it's always the last line". No, it is not. Let's add two nested functions in `test.js`

```
// file: test.js

...

async function main() {
    let logger = await initialize()
    nested_func(logger)
}

function nested_func(logger) {
    super_nested(logger)
}

function super_nested(logger) {
    logger.critical('Testing')
}

...
```

After executing `node test.js` we get the following output.

```
Error
    at Logger.#log (/Users/ishtmeet/Code/logtard/lib/logger.js:98:19)
    at Logger.critical (/Users/ishtmeet/Code/logtard/lib/logger.js:141:18)
    at super_nested (/Users/ishtmeet/Code/logtard/test.js:20:12)
    at nested_func (/Users/ishtmeet/Code/logtard/test.js:16:5)
```

```
    at main (/Users/ishtmeet/Code/logtard/test.js:12:5)
```

However, this time, there are two more lines. If you read the stack trace from the bottom to the top, you can understand the sequence of steps that led to the error. The most useful information is not actually at the very bottom, but rather on the fourth line (including the "Error" line at the top of the stack trace).

```
at super_nested (/Users/ishtmeet/Code/logtard/test.js:20:12)
```

The 4th line will always be the one that invoked the method. Which is the line directly below the call to `Logger.critical`. Isn't that what we need?

4.14.6 A more ergonomic way

Looking at the code we just wrote in our `Logger.#log` method, it seems poorly written, making it hard to understand our desired outcome. Additionally, why are we throwing an Error? Someone unfamiliar with our code might consider it redundant and remove it.

We can make it even better by creating a helper method that extracts essential information from our stack trace.

Add a new function in the `lib/utils/helpers.js` file

```
// file: lib/utils/helpers.js

function get_caller_info() {}

module.exports = {
  check_and_create_dir,
  get_log_caller, // Add this!
};
```

We are going to get our stack trace in a shorter and more efficient way. Here's what we're going to do

```
// file: lib/util/helpers.js

function get_caller_info() {
  const error = {};
  Error.captureStackTrace(error);

  const caller_frame = error.stack.split("\n")[4];
```

```
  const meta_data = caller_frame.split("at ").pop();
  return meta_data;
}
```

`Error.captureStackTrace(targetObject)` is a static method on the Error class that's used to customize or enhance the creation of stack traces when throwing errors. It doesn't throw an error itself, but rather it modifies the target object to include a custom stack trace.

It's designed specifically for capturing stack traces without generating a full error object, which can be helpful when you want to create your own custom error objects and still capture the stack trace efficiently. It directly associates the stack trace with the provided object.

```
const caller_frame = error.stack.split("\n")[4];
```

We are extracting the 5th line of the stack trace. Note, it's the 5th line and not 4th like we talked in the previous section. This is because, we introduce one more function `get_log_caller`, that will also live on the call stack. You can imagine a call stack like this:

```
get_caller_info
Logger.#log
Logger.critical/Logger.debug etc
user_function // that called `logger.critical`
```

On the top of the stack trace there's a line that says

```
Error
```

So the entire stack trace can be imagined like this:

```
Error
    at get_caller_info line:number
    at Logger.#log line:number
    at Logger.critical line:number
    at user_function line:number // that called `logger.critical`
```

Right, we only care about the `5th` line.

```
const meta_data = caller_frame.split("at ").pop();
```

In this line, we are retrieving the part of the string that follows the word ”at” followed by a space, as we do not need to include it in our output. Finally, on the last line, we return the necessary metadata to display in the logs.

4.14.7 Using the `get_caller_info` function

Update the code in the `Logger` class to use the info provided by the `get_caller_info` function.

```
// file: lib/logger.js

class Logger {
    ...
    async #log(message, log_level) {
        if (log_level < this.#config.level || !this.#log_file_handle.fd) {
            return;
        }

        const date_iso = new Date().toISOString();
        const log_level_string = LogLevel.to_string(log_level)

        // add additional info to the log messages.
        const log_message = `[${date_iso}] [${log_level_string}]: ${get_caller_info()}
        ↪  ${message}\n`;
        await this.#log_file_handle.write(log_message);
    }
    ...
}
```

Now, we're going to test whether everything works like we expect?

In the `test.js` file let's write a couple of logs.

```
// file: test.js

const {Logger} = require('./index')

async function initialize() { ... }

async function main() {
    let logger = await initialize()
    logger.critical('From the main() function')
    nested_func(logger)
}
```

```
function nested_func(logger) {
    logger.critical('From the nested_func() function')
    super_nested(logger)
}

function super_nested(logger) {
    logger.critical('From the super_nested() function')
}

main()
```

The log file shows the following -

```
[2023-08-19T19:11:51.888Z] [CRITICAL]: main (/Users/ihtmeet/Code/logtard/test.js:11:12) From the
↪  main() function
[2023-08-19T19:11:51.888Z] [CRITICAL]: nested_func (/Users/ishtmeet/Code/logtard/test.js:16:12)
↪  From the nested_func() function
[2023-08-19T19:11:51.888Z] [CRITICAL]: super_nested (/Users/ishtmeet/Code/logtard/test.js:21:12)
↪  From the super_nested() function
```

This all seems to work pretty well. We now have helpful logs. However, before we start using this logging library in our personal projects, there are a lot of things that need to be taken care of. This includes logging crashes, handling SIGINT and SIGTERM signals, as well as properly utilizing the file_handle.

We'll take care of this in the next chapter.

> The code for the entire chapter can be found at the src/chapter_04.3 directory

4.15 A small intro to `async` vs `sync`

Note: I am not going to explain the event loop just yet. We will have a dedicated chapter on it later in this book. Bear with me whenever I say "event-loop.".

> As an asynchronous event-driven JavaScript runtime, Node.js is designed to build scalable network applications

The line above is the very first line you'd see on the About Node.js page. Let's understand what do they mean by **asynchronous event-driven** runtime.

In Node.js, doing things the asynchronous way is a very common approach due to its efficiency in handling multiple tasks at once. However, this approach can be complex and requires a careful understanding of the interplay between asynchronous (async) and synchronous (sync) operations.

This chapter aims to provide a comprehensive explanation of how these operations work together in Node.js. We will delve into the intricacies of async and sync operations, including how buffers can be used to fine-tune file calls.

Also, we will explore Node.js's **smart** optimization techniques, which allow for increased performance and responsiveness. By understanding the interplay between async and sync operations, the role of buffers, and the trade-offs involved in optimization, you will be better equipped to write efficient and effective Node.js applications.

4.15.1 The Balance between Opposites

Node.js's architecture is designed to support asynchronous operations, which is consistent with JavaScript's event-driven nature. Asynchronous operations can be executed via callbacks, Promises, and async/await. This concurrency allows tasks to run in parallel (not exactly parallel like multi-threading), which enables your application to remain responsive even during resource-intensive operations such as file I/O.

However, synchronous operations disrupts this balance. When a synchronous operation is encountered, the entire code execution is halted until the operation completes. Although synchronous operations can usually execute more quickly due to their direct nature, they can also lead to bottlenecks and even application unresponsiveness, particularly under high I/O workloads.

4.15.2 Mixing Asynchronous and Synchronous Code

You must ensure consistency between asynchronous and synchronous operations. Combining these paradigms can lead to a lot of challenges. The use of synchronous operations within an asynchronous context can result in performance bottlenecks, which can derail the potential of your application.

Every operation affects how quickly it responds to requests. If you use both async and sync operations, it can make the application slower and less efficient.

4.15.3 Faster I/O out of the box

Node.js uses buffering to handle file operations. Instead of writing data directly to the disk, Node.js stores the data in an internal buffer in memory. This buffer combines multiple write operations and writes them to the disk as one entity, which is more efficient. This strategy has two benefits: it's much faster to write

data to memory than to disk, and batching write operations reduces the number of disk I/O requests, which saves time.

Node.js's internal buffering mechanism can make asynchronous write operations feel instantaneous, as they are merely appending data to memory without the overhead of immediate disk writes. However, it's important to note that this buffered data isn't guaranteed to be persisted until it's flushed to the disk.

4.15.4 Blocking Code

Blocking code refers to a situation where additional JavaScript execution in the Node.js process has to wait until a non-JavaScript operation finishes. This can occur because the event loop cannot continue running JavaScript when a blocking operation is in progress.

Consider the following example that reads a file synchronously:

```
const fs = require("node:fs");

// blocks the entire program till this finishes
const fileData = fs.readFileSync("/path/to/file");
console.log(fileData);
```

The `readFileSync` method is blocking, which means that the JavaScript execution in the Node.js process has to wait until the file reading operation is complete before continuing. This can cause performance issues, especially when dealing with large files or when multiple blocking operations are performed in sequence.

Fortunately, the Node.js standard library offers asynchronous versions of all I/O methods, which are non-blocking and accept callback functions. Consider the following example:

```
const fs = require("node:fs/promises");

async function some_function() {
    // blocks the execution of the current function
    // but other tasks are unaffected
    const data = await fs.readFile("test.js", "utf-8");
    console.log(data);
}

some_function();
```

The `readFile` method is non-blocking, which means that the JavaScript execution can continue running while the file reading operation is in progress. When the operation is complete, the next line is executed.

Some methods in the Node.js standard library also have blocking counterparts with names that end in `Sync`. For example, `readFileSync`, that we just saw, has a non-blocking counterpart called `readFile`.

It's important to understand the difference between blocking and non-blocking operations in Node.js to write efficient and performant code.

4.15.5 Concurrency

One key aspect of Node.js is that JavaScript execution is single-threaded, meaning that only one task can be executed at a time. This can be a challenge when dealing with tasks that require a lot of processing power or that involve I/O operations, which can cause the program to block and slow down its execution.

To address this issue, Node.js uses an event-driven, non-blocking I/O model. This means that instead of waiting for an I/O operation to complete before moving on to the next task, Node.js can perform other tasks while the I/O operation is taking place.

For example, suppose an average request for an API endpoint takes 60ms. Within that 60ms, 30ms are spent reading from a database, and 25ms are spent reading from a file. If this process were synchronous, our web server would be incapable of handling a large number of concurrent requests. However, Node.js solves this problem with its non-blocking model. If you use the asynchronous version of the file/database API, the operation will continue to serve other requests whenever it needs to wait for the database or file.

4.16 Adding Rolling File Support

> The code for the entire chapter can be found at the src/chapter_04.5 directory

Since the beginning of Chapter 4, where we introduced `logtar`, our own logging library, we have been discussing the rolling file support. This involves creating new files based on the rolling configuration set in `config.json` or provided via the `LogConfig.from_json` method. In this chapter, we will finish building this feature.

4.16.1 Rolling features

In our `RollingConfig` class, we defined 2 private variables - `#time_threshold` and `#size_threshold`

```
// file: lib/config/rolling-config.js

class RollingConfig {
    #time_threshold = RollingTimeOptions.Hourly;
```

```
    #size_threshold = RollingSizeOptions.FiveMB;

    ...
}
```

4.16.1.1 `#time_threshold`

This defines how often do we need to create a new file. For example, if the set value is `RollingTimeOptions.Hourly` that means, we'll be creating a new log file every hour.

4.16.1.2 `#size_threshold`

This property tells what should be the maximum size of a log file. If it exceeds the size, we'll simply create a new log file.

Let's start coding.

4.16.2 The `rolling_check()` method

We're going to introduce a new private method `Logger.rolling_check`, that is going to take care of the rolling logic.

```
// file: lib/logger.js

class Logger {
    ...
    /**
     * Checks if the current log file needs to be rolled over.
     */
    async #rolling_check() {
        const { size_threshold, time_threshold } = this.#config.rolling_config;

        const { size, birthtimeMs } = await this.#log_file_handle.stat();
        const current_time = new Date().getTime();

        if (size ≥ size_threshold || (current_time - birthtimeMs ≥ time_threshold * 1000)) {
            await this.#log_file_handle.close();
            await this.init();
        }
```

```
    }
    ...
}
```

4.16.3 `file_handle.stat()`

Let's go through the code line by line:

```
const { size_threshold, time_threshold } = this.#config.rolling_config;
```

This line above destructures the `size_threshold` and `time_threshold` properties from the `rolling_config` member object.

```
const { size, birthtimeMs } = await this.#log_file_handle.stat();
```

We're destructuring the `size` and the `birthtimeMs` properties from the `file_handle.stat()` method.

Node.js provides us with a `stat` method on the `FileHandle` class, that can be used to check various stats of a file. The stat method provides a lot of useful information about the current file/file_handle. Some of them are methods, and some are properties.

Let's take a look at some of the useful information provided by `file_handle.stat()`:

```
// Returns true if the `file_handle` is a file. Fails if the file handle is
// pointing to a directory instead
file_handle.stat().isFile();

// Returns true if this `file_handle` refers to a socket
file_handle.stat().isSocket();

// Numeric indentifier of of the device which contains this file.
file_handle.stat().dev;

// The size of the file in bytes.
file_handle.stat().size;

// Time since this file was last accessed, in `ms`
file_handle.stat().atimeMs;

// Time since the file was last modified, in `ms`
```

```
file_handle.stat().mtimeMs;

// Time since the last time it's file status changed.
// File status is a two-byte code that indicates how a file operation completed;
// either successfully, or with some form of error.
file_handle.stat().ctimeMs;

// Time since this file was created, in `ms`
file_handle.stat().birthtimeMs;

// Same methods as above, but the value is in `ns` (nanoseconds)
// 1 ns = 1,000,000 ms
file_handle.stat().atimeNs;
file_handle.stat().mtimeNs;
file_handle.stat().ctimeNs;
file_handle.stat().birthtimeMs;

// Same methods as above, but the type of this is `Date`. The actual javascript Date class
file_handle.stat().atime;
file_handle.stat().mtime;
file_handle.stat().ctime;
file_handle.stat().birthtime;
```

I've omitted the ones that are not of any use for us.

```
const current_time = new Date().getTime();
```

We are creating a new `Date` object and uses the `getTime()` method to obtain the current time in milliseconds since epoch.

```
if (size ≥ size_threshold || (current_time - birthtimeMs ≥ time_threshold)) {
```

We'll be creating a new file:

- If the `size` of the log file is greater than or equal to the `size_threshold`, or
- If the time difference between `current_time` and `birthtimeMs` (file creation time) is greater than or equal to the `time_threshold`.

```
await this.#log_file_handle.close();
```

Closes the file handle after waiting for any pending operation on the handle to complete. This is important. We should make sure if there are any pending operations, or the Node.js runtime hasn't flushed (written) the entire file, we're going to wait till then.

When everything is flushed, we're simply closing the file handle, so nothing in our application will access this file again.

```
await this.init();
```

We call the `Logger.init` method to create the new file again. Here's the code for `init()` just in case:

```
async init() {
    const log_dir_path = check_and_create_dir("logs");

    const file_name = this.#config.file_prefix + new Date().toISOString().replace(/[\.:]+/g, "-")
     ↪  + ".log";
    this.#log_file_handle = await fs.open(path.join(log_dir_path, file_name), "a+");
}
```

4.16.4 Calling the `rolling_check` method

Where are we going to call this method that we just created? Of-course, it's the `Logger.#log` method.

```
class Logger {
    ...
    async #log(message, log_level) {
        if (log_level < this.#config.level || !this.#log_file_handle.fd) {
            return;
        }

        await this.#rolling_check(); // Call the check method.

        // I've extracted all the writing functionality into a separate method
        await this.#write_to_handle(message, log_level);
    }
    ...
}
```

Here's the abstracted functionality in the `Logger.#write_to_handle` method.

```
class Logger {
    ...
     async #write_to_handle(message, log_level) {
        const date_iso = new Date().toISOString();
        const log_level_string = LogLevel.to_string(log_level);

        const log_message = `[${date_iso}] [${log_level_string}]: ${get_caller_info()}
         ↪  ${message}\n`;
        await this.#log_file_handle.write(log_message);
    }
    ...
}
```

Let's try to run the `test.js` file again.

```
// file: test.js

const { LogConfig } = require("./lib/config/log-config");
const { Logger } = require("./lib/logger");

async function init() {
  const logger = Logger.with_config(LogConfig.from_file("config.json"));
  await logger.init();
  return logger;
}

async function main() {
  const logger = await init();
  logger.info("Hello World!\n");
}

main();
```

Oops...

```
TypeError: Cannot read properties of undefined (reading 'split')
    at get_caller_info (/Users/ishtmeet/Code/logtard/lib/utils/helpers.js:24:36)
    at Logger.#write_to_handle (/Users/ishtmeet/Code/logtard/lib/logger.js:112:69)
```

```
    at Logger.#log (/Users/ishtmeet/Code/logtard/lib/logger.js:89:36)
```

4.16.5 A big gotcha!

You might have guessed what part of the code crashed. Yes, that's the `get_caller_info` function defined inside `lib/utils/helpers.js`. That function is in-charge of collecting the stack trace and returning the line that includes the caller. Here's how that function looks:

```
// file: lib/utils/helpers.js

function get_caller_info() {
  const error = {};
  Error.captureStackTrace(error);

  // this line fails.
  const caller_frame = error.stack.split("\n")[4];

  const meta_data = caller_frame.split("at ").pop();
  return meta_data;
}
```

But why? Shouldn't this code work properly? Well, there's a huge part of `async` in general that you should be aware of. For now let's add a `console.log` to check that the stack trace is.

```
function get_caller_info() {
    ...
    console.log(err.stack);
    const caller_frame = error.stack.split("\n")[4];
    ...
}
```

Here's what the stack trace looks like:

```
Error
    at get_caller_info (/Users/ishtmeet/Code/logtard/lib/utils/helpers.js:21:11)
    at Logger.#write_to_handle (/Users/ishtmeet/Code/logtard/lib/logger.js:113:69)
    at Logger.#log (/Users/ishtmeet/Code/logtard/lib/logger.js:90:36)
```

Where are the lines that refer to the `logger.info` and the `main()` function? They seem to be missing...

4.17 Stack traces across `await` points

An ”asynchronous point” or an ”await point” is a concept that refers to the moment when control is handed back to the JavaScript runtime during an asynchronous operation.

This point is marked by the `await` keyword or when a promise's `then()` or `catch()` method is called. At an `await` point, the runtime can execute other tasks while keeping track of the ongoing asynchronous operation. But this has something going behind the scenes.

When an `async` function is invoked and it encounters an `await` expression, the function execution is essentially paused and control is returned to the event loop. The event loop then continues processing other tasks in the queue. When the awaited asynchronous operation completes (e.g., a Promise resolves), the function is resumed from where it was paused.

The function kind of `yeilds` back to the runtime, saying - ”Hey, please continue with the event loop and continue this function from this line whenever you get that data”.

When an error occurs within an asynchronous function, the stack trace that is captured includes information up to the point where the asynchronous operation was initiated (i.e., where the `await` is placed). However, it does not include the full call stack leading up to the initial invocation of the asynchronous function. This is because the call stack gets unwound as control is returned to the event loop.

If an error occurs within an asynchronous function **before** the `await` expression, the stack trace will include information about where the error happened and all the calling functions leading up to that point.

If an error occurs **after** the `await` expression, the stack trace will only include information about the current function and the awaited function that was resumed after the asynchronous operation completed. It will not include the calling functions that led to the initial invocation of the asynchronous function.

This behavior is designed to make the program more efficient. But what's causing the issue in our code?

4.17.0.1 The culprit

```
// file: lib/logger.js

class Logger {
    ...
    async #log(message, log_level) {
        if (log_level < this.#config.level || !this.#log_file_handle.fd) {
            return;
        }
```

```
        // This is causing the issue
        await this.#rolling_check();
        await this.#write_to_handle(message, log_level);
    }

    ...
}
```

The `async` method `#rolling_check()` is the one that's causing the stack trace to trim all the required info.

When the runtime encounters the expression starting with `await`, it pauses the execution of that function and "unwind the stack" in a non-blocking manner. This means that the method will yield control back to the event loop, allowing other tasks to be processed while the awaited asynchronous operation completes.

So, all the information about who invoked the `#log` method is dropped. This is done to make the code more efficient. The more information the runtime needs to hold, the more memory it will occupy.

So how do we fix it? It's very simple. Write to the `file_handle` before we do the `rolling_check`. Here's the updated code:

```
// file: lib/logger.js

class Logger {
    ...
    async #log(message, log_level) {
        if (log_level < this.#config.level || !this.#log_file_handle.fd) {
            return;
        }

        await this.#write_to_handle(message, log_level);
        // Make sure we're doing the writing before checking for rolling creation
        await this.#rolling_check();
    }

    ...
}
```

Let's try to run `test.js` again. That works!

```
[2023-08-21T23:28:59.178Z] [INFO]: Logger.info
↪  (/Users/ishtmeet/Code/logtard/lib/logger.js:128:18) Hello World!
```

This is still not accurate. It's telling that the function which was responsible for writing to the file was `Logger.info`, however it is not correct.

Why is that? It's because we introduce a new method, `this.#write_to_handle()`, so the line which contains our actual invocation went more line down. To illustrate, let's print our stack trace.

```
function get_caller_info() {
    ...
    console.log(err.stack);
    const caller_frame = error.stack.split("\n")[4];
    ...
}
```

The output is:

```
1    Error
2        at get_caller_info (/Users/ishtmeet/Code/logtard/lib/utils/helpers.js:21:11)
->       at Logger.#write_to_handle (/Users/ishtmeet/Code/logtard/lib/logger.js:113:69)
4        at Logger.#log (/Users/ishtmeet/Code/logtard/lib/logger.js:89:36)
5        at Logger.info (/Users/ishtmeet/Code/logtard/lib/logger.js:128:18)
6        at main (/Users/ishtmeet/Code/logtard/test.js:13:12)
```

The line marked with `->` shows the extra method which was pushed on the stack, because we abstracted all the functionality into that method. To fix that, we're going to update the code to get the `6th` line instead of the `5th`

```
// file: lib/utils/helpers.js

function get_caller_info() {
    ...
    const caller_frame = error.stack.split("\n")[5];
    ...
}
```

Let's test whether it works one last time:

```
[2023-08-21T23:37:11.919Z] [INFO]: main (/Users/ishtmeet/Code/logtard/test.js:13:12) Hello World!
```

Everything is working as expected.

4.17.1 Testing the new Log file creation

Update our `RollingTimeOptions` class to include a new `FiveSeconds` as a temporary variable, which we are going to remove after testing. Also make sure to add it inside the `static assert()` method, or this method will reject a value it doesn't know.

```
// file: lib/utils/rolling-options.js
class RollingTimeOptions {
    static FiveSeconds = 5; // Every 5 seconds
    static Minutely = 60; // Every 60 seconds
    ...

    static assert(time_option) {
        // Add `this.FiveSeconds` for validation
        if (![this.FiveSeconds, this.Minutely, this.Hourly, this.Daily, this.Weekly,
        ↪  this.Monthly, this.Yearly].includes(time_option)) {
            throw new Error(
                `time_option must be an instance of RollingConfig. Unsupported param
                ↪  ${JSON.stringify(time_option)}`
            );
        }
    }
}
```

The `config.json` should look like this

```
// file: config.json

{
    "level": 0, // Log Debug messages and above
    "log_prefix": "LogTar_",
    "rolling_config": {
        "size_threshold": 1048576, // 1 MB:  1024 * 1024
        "time_threshold": 5 // Create a new file every 5 seconds
    }
```

```
}
```

We're going to test the `time_threshold` for now. I've set it to `5` seconds, so it creates a new log file every time the `file_handle.stat().birthtimeMs > 5000`.

Inside `test.js` run a log statement every 1 second:

```
// file: test.js
...

async function main() {
    const logger = await init()
    setInterval(() => {
        logger.critical("This is critical");
    }, 1000);
}

...
```

Run the program. You'll notice a new log file created every 5 seconds!

We are going to test the `size_threshold` rolling option now. Remove the newly created `FiveSeconds` static member from `RollingTimeOptions`.

Update your code in `test.js`:

```
// file: test.js

async function main() {
  const logger = await init();
  setInterval(() => {
    logger.critical("Hi there");
  }, 20); // run every 20 milliseconds
}
```

The `config.json` should specify the `size_threshold` as `20428` bytes (20 KB).

```
{
    "level": 0,
    "log_prefix": "LogTar_",
    "rolling_config": {
```

```
        // max size should be 20 KB (20 * 1024 bytes)
        "size_threshold": 20480,
        "time_threshold": 60
    }
}
```

Run the `test.js` script. You'll notice a new file created every now and then. Let's check the sizes of those log files.

```
$ ls -al ./logs

.rw-r--r--  10k ishtmeet 22 Aug 06:11 Logtar_2023-08-22T00-41-58-132Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-41-59-232Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-01-101Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-02-139Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-03-545Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-04-132Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-05-382Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-06-432Z.log
.rw-r--r--  10k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-07-109Z.log
.rw-r--r-- 3.6k ishtmeet 22 Aug 06:12 Logtar_2023-08-22T00-42-08-138Z.log
```

Each file is only 10KB, which is exactly what we need. However, the last file is not 10KB because I had to exit the infinite loop caused by `setInterval` by pressing `Ctrl + C`.

We now have an actual logging library that can be used with any type of project. However, there is still something we need to take care of: providing a middleware function that can be used.

For instance, many Node.js backend frameworks, such as express, utilize the middleware pattern. In this pattern, incoming requests are passed through a series of functions (the middlewares), which utilize the request, and either reject it or forward it to the next middleware.

To use a logging library, we will print out certain information, such as the IP of the request, the query, pathname, or anything else.

> We will ensure that our library is compatible with other frameworks. However, we will address this after we have built enough of our backend framework.

Chapter 5

HTTP Deep dive

This chapter gives an overview of how the web functions today, discussing important concepts that are fundamental for understanding the rest of the book. Although some readers may already be familiar with the material, this chapter provides a valuable opportunity to revisit the basics.

All the browsers, servers and other web related technologies talk to each other through HTTP, or Hypertext Transfer Protocol. HTTP is the common language used on the internet today.

Web content is stored on servers that communicate using the HTTP protocol, often called HTTP servers. They hold the Internet data and provide it when requested by HTTP clients.

Clients ask for data by sending HTTP requests to servers. Servers then respond with the data in HTTP responses.

The message that was sent from your browser is called a **Request** or an **HTTP request**. The message received by your browser, which was sent by a server is called an **HTTP Response** or just **Response**. The response and the request is collectively called an **HTTP Message**.

> Warning: Don't visit the URL mentioned above.

5.1 A small web server

Before introducing more components of HTTP, let's build a basic HTTP server using Node.js.

Create a new project, name it whatever you like. Inside the directory create a new file `index.js`

```
// file: index.js

// Import the 'node:http' module and assign it to the constant 'http'
const http = require("node:http");

// Define a function named 'handle_request' which takes two parameters: 'request' and 'response'
function handle_request(request, response) {
  // Send the string "Hello world" as the response content
  // Do nothing with the request, for now
  response.end("Hello world");
}

// Create an HTTP server using the 'createServer' method of the 'http' module.
// Pass the 'handle_request' function as the callback for handling incoming requests.
const server = http.createServer(handle_request);

// Start the server to listen for incoming requests on port 3000 and the host 'localhost'
server.listen(3000, "localhost");

// Alternatively, you can use the IP address '127.0.0.1' instead of 'localhost'
server.listen(3000, "127.0.0.1");
```

Let's go through the code above in more detail:

```
const http = require("node:http");
```

This line brings in Node.js's **`http`** module, that provides basic functionality to create HTTP servers.

```
function handle_request(request, response) {
  response.end("Hello world");
```

```
}
```

This function has two input parameters: **`request`** (it is the message that comes from the internet) and **`response`** (it is the message that we send back to the user). And we're simply sending back a response "Hello world".

```
const server = http.createServer(handle_request);
```

We're creating an HTTP server using the **`createServer`** method from the **`http`** module. The **`handle_request`** function is assigned as the callback function that will be called when the server receives a request.

So, whenever there's a new HTTP request, our `handle_request` function will be invoked, and two arguments `request` and `response` will be provided.

```
server.listen(3000, "localhost");
```

This line of code starts the server and makes it available at **`http://localhost:3000`**.

```
server.listen(3000, "127.0.0.1");
```

You can also use the IP address `'127.0.0.1'` instead of `'localhost'` to achieve the same result.

Both **`localhost`** and **`127.0.0.1`** refer to the local loopback address, which means that the server will only be accessible from the same machine it's running on. This is commonly used by developers for testing and debugging purposes.

In fact, it's a best practice to develop and test applications locally before deploying them to a production environment, as it allows for easier troubleshooting and debugging of any issues that may arise.

5.1.1 Starting our web server

To start the web server, we simply need to execute the file `index.js`. Let's try to run it:

```
$ node index


# no response, is there something wrong?
```

You might have noticed that the program didn't exit, like it used to do previously when we ran our logging library or previous code examples. This behavior can be a bit surprising, but it's expected due to the nature of how HTTP servers work.

When we run an HTTP server using `http.createServer()`, the server is designed to listen for incoming requests indefinitely. It doesn't terminate as soon as the script finishes executing, unlike some other scripts

that might perform a single task and then exit.

This is because the purpose of an HTTP server is to continuously listen for incoming HTTP requests and respond to them in real-time. If the server were to exit immediately, it wouldn't be able to fulfill its purpose of serving content to clients.

5.1.2 Testing our web server

To see whether our web server is working as we expected, and to see the "Hello world" response back, we need to make a request first. There are certain ways to make a request. We'll take the easy route.

Go to the browser of your choice, and visit the URL `http://localhost:3000` or `http://127.0.0.1` or `localhost:3000` or `127.0.0.1:3000`.

Yes, it seems to work correctly. Let's make a request to our server in a different way, to look at all the necessary parts that make up an HTTP request.

5.1.3 Testing with `cURL`

Open up your terminal, and enter the following `cURL` command:

```
$ curl http://localhost:3000
Hello world%
```

Perfect. `cURL` is a convenient and a quick way of testing HTTP endpoints. It is easy to use, and do not need to have other HTTP clients running, or your chrome's network tab opened.

Let's modify the `cURL` request:

```
$ curl http://localhost:3000 -v
```

We're specifying the `-v` argument (also `--verbose`) which displays more information about the HTTP connection lifecycle. This is the output you get after you execute the command above:

```
$ curl http://localhost:3000 -v
*   Trying 127.0.0.1:3000...
* Connected to localhost (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Wed, 23 Aug 2023 13:13:32 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Content-Length: 11
<
* Connection #0 to host localhost left intact
Hello world%
```

That's a lot of stuff to digest. One thing to notice is two different operators that are used - `>` and `<`. The `>` arrow indicates that this is being sent as a **request** from the client. Client here refers to the `cURL` or your terminal. `<` indicates that these lines have been received as a **response** from the server.

Let me walk through this line by line:

```
Trying 127.0.0.1:3000...
```

This line is indicating that `cURL` is attempting to establish a connection to the IP address 127.0.0.1. This IP address is also known as localhost and it refers to the local computer you are currently working on.

The connection is being made on port number 3000, which is a number used to identify a specific process to which data is being sent or received. We'll talk about PORTs in the next chapter.

```
Connected to localhost (127.0.0.1) port 3000 (#0)
```

This means the connection to the specified IP address and port has been successfully established. This is great news and means that you can proceed with your task without any further delay.

The (#0) in the message refers to the connection index. This is helpful information, especially if you're making multiple connections at the same time. By keeping track of the connection index, you can avoid any confusion or errors that could arise from mixing up different connections.

```
> GET / HTTP/1.1
```

`cURL` is sending an HTTP request to the server using the HTTP method **`GET`**. The **`/`** after the method indicates that the request is being made to the root path, or the root endpoint of the server.

```
> Host: localhost:3000
```

This line specifies that the `cURL` command is setting the **`Host`** header on the **request**, which tells the server the domain name and port of the request.

```
> User-Agent: curl/7.87.0
```

This line is also part of the HTTP request headers. It includes the **`User-Agent`** header, which identifies the client making the request. In this case, it indicates that the request is being made by **`curl`** version 7.87.0.

```
> Accept: */*
```

This line sets the **`Accept`** header, which tells the server what types of response content the client can handle. In this case, it indicates that the client accepts any type of content.

```
>
```

This line indicates the end of the HTTP request headers. An empty line like this separates the headers from the request body, which is not present in this case because it's a `GET` request. We'll talk about `POST` and other http verbs/methods in the upcoming chapters.

```
Mark bundle as not supporting multiuse
```

This is an internal log message from **`curl`** and doesn't have a direct impact on the request or response interpretation. It's related to how **`curl`** manages multiple connections in a session.

```
< HTTP/1.1 200 OK
```

This line is part of the HTTP response. It indicates that the server has responded with an HTTP status code **`200 OK`**, which means the request was successful. Again, we're going to understand HTTP status codes in the next chapter. For now, it's enough to think that any status code in the form `2xx`, where x is a number, means everything is fine.

```
< Date: Wed, 23 Aug 2023 13:13:32 GMT
```

This line is part of the HTTP response headers. Important thing to note, this is the header set by the server, and not the client. It includes the **`Date`** header, which indicates the date and time when the response was

generated on the server.

```
< Connection: keep-alive
```

This line is part of the response headers and informs the client that the server wants to maintain a persistent connection, and re-use the connection for potential future requests. This helps with the performance.

```
< Keep-Alive: timeout=5
```

This line, also part of the response headers, specifies the duration of time (5 seconds in this case) that the server will keep the connection alive if no further requests are made.

```
< Content-Length: 11
```

This line indicates the length of the response content in bytes. In this case, the response body has a length of 11 bytes.

```
<
```

This line marks the end of the response headers and the beginning of the response body.

```
Connection #0 to host localhost left intact
```

This line is a log message from **`curl`** indicating that the connection to the server is being left open (**`intact`**) and not closed immediately after receiving the response. It could potentially be reused for subsequent requests.

```
Hello world%
```

This is the actual response body returned by the server. In this case, it's a simple text string saying "Hello world". The `%` is nothing to worry about. It indicates that the response doesn't ends with a `\n` character.

Now that we understand what are the basic components of the `request` and the `response`, let's understand these components in more detail, in the next chapter

5.2 HTTP Verbs, Versioning and the benefits of `HTTP/1.1`

In the previous chapter, we made a simple request to `http://localhost:3000` with `cURL` and received the following output on the terminal:

```
$ curl http://localhost:3000 -v
*   Trying 127.0.0.1:3000...
```

```
* Connected to localhost (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Wed, 23 Aug 2023 13:13:32 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Content-Length: 11
<
* Connection #0 to host localhost left intact
Hello world%
```

Our focus of this chapter will be the first two lines of the request:

```
> GET / HTTP/1.1
> Host: localhost:3000
```

This starts with `GET` which is a **HTTP method** or commonly called as HTTP verb, because they describe an action. These HTTP methods are an essential part of the communication process between client applications, such as web browsers, and web servers. They provide a structured and standardized way for client applications to interact with web servers, ensuring that communication is clear and concise.

HTTP methods offer a set of instructions for the server to carry out, which can include retrieving a resource, submitting data, or deleting a resource. These methods are essential for ensuring that client applications can perform a variety of tasks in a streamlined and efficient way.

The most commonly used HTTP methods include `GET`, `POST`, `PUT`, `DELETE`, and `PATCH`, each with a specific purpose and set of rules. For example, the `GET` method is used to retrieve data from a server, while the `POST` method is used to submit data to a server for processing. By following these rules, client applications can communicate effectively with web servers, ensuring that data is transmitted correctly and that the server can respond appropriately.

5.2.1 `GET` - Retrieve data

The most basic example of the `GET` method is typing any URL in the browser and pressing enter. You're looking to `GET` the contents of a certain website.

Then, the browser sends a request to the server that hosts the website, asking for the content you want to see. The server processes the request and sends a response back to your browser with the content you requested.

The `GET` method helps digital content (like text, pictures, HTML pages, stylesheets and sounds) appear on your web browser.

When a client sends a GET request to a server, the server should only return data to the client and not modify or change its state. This means that the server should not alter any data on the server-side or perform any actions that could modify the system's state in any way.

This is important to ensure that the data requested by the client is accurate and consistent with the server-side data. By following this protocol, developers can ensure that their applications function smoothly and without any unexpected side effects that could cause problems in the future.

> Note: It's technically possible to modify anything on a GET request from the server side, but it's a bad practice and goes against standard HTTP protocol rules. Stick to the REST API guidelines for good API design.

5.2.2 `POST` - Create something

The `POST` method serves a different purpose compared to the `GET` method. While `GET` is used for retrieving data, `POST` is employed when you want to send data to a server, to create a new resource on the server.

Imagine you're filling out a form on a website to create a new account. When you submit the form, the website sends a `POST` HTTP request to send the information you provided (such as your username, password, and email) to the server. The server processes this data, typically validating it, and if everything is fine - create a new user account.

Just like with `GET`, it's crucial to follow proper HTTP API guidelines when using the `POST` method. Ensuring that your `POST` requests only create data and don't have unintended side effects helps maintain the integrity of your application and the server's data.

> Note: Again, it is technically possible to use the `POST` method to retrieve data from a server, or update data, it's considered non-standard and goes against conventional HTTP practices.

5.2.3 `PUT` - Replace or Create

The PUT method is used to change or replace data that already exists on the server, completely. It's important to remember that, according to HTTP API guidelines, the PUT request should not be used to update only part of the content.

You may ask, aren't `POST` and `PUT` same as they do the same thing - `CREATE` data if it doesn't exist? There's a difference.

The main difference between the `PUT` and `POST` methods is that PUT is considered **idempotent**. This means that calling it once or multiple times in a row will result in the same outcome, without any additional side effects.

In contrast, successive identical POST requests may have additional effects, such as creating multiple instances of a resource or submitting duplicate orders.

5.2.4 `HEAD` - Retrieve Metadata

The `HEAD` method shares a resemblance to the `GET` method in that it retrieves information from the server, but with one fundamental difference: it only retrieves the metadata associated with a resource, not the resource's actual content.

Metadata can include details like the resource's size, modification timestamp, content-type, and more.

When a `HEAD` request is made to a server, the server processes the request in a manner similar to a `GET` request, but instead of sending back the full content, it responds with just the metadata in the HTTP headers. This can be useful when a client needs information about a resource without the need to transfer the entire content, which can save bandwidth and improve performance.

For example, imagine a web page that lists links to various files for download. When a user clicks on a link, the client can initiate a `HEAD` request to gather information about the file, such as its size, without actually downloading the file itself.

By using the `HEAD` method when needed, we can optimize data retrieval, minimize unnecessary network traffic, and obtain crucial resource information efficiently and quickly.

> It's worth noting that servers are not required to support the `HEAD` method, but when they do, they should ensure that the metadata provided in the response headers accurately reflects the current state of the resource.

5.2.5 `DELETE` - Remove from existence

The `DELETE` method is used when you need to remove or delete a specific resource from the server. When using the `DELETE` method, it's important to know that the action is **idempotent**. This means that no matter how many times you execute the same `DELETE` request, the outcome remains the same – the targeted resource is removed.

We should be careful when using the DELETE method, as it's purpose is to permanently remove the resource from the server. To avoid accidental or unauthorized deletions, it's recommended to use proper authentication and authorization mechanisms.

After successfully deleting a resource, the server may respond with a `204 No Content` status code to indicate that the resource has been removed, or a 404 Not Found status code if the resource was not found on the server.

If you only need to remove part of something and not the whole thing, using the `DELETE` method might not be the best idea. To make things clear and well-organized, it's better to identify each part as a separate resource, like with the `PATCH` method for partial updates.

5.2.6 `PATCH` - Partial updates

The `PATCH` method is used to partially update an existing resource on the server. `PATCH` is different from the `PUT` method, which replaces the entire resource. With `PATCH`, you can modify only the specific parts of the resource's representation that you want to change.

So, if you're looking to update a document, or update a user, use the `PATCH` method.

PATCH requests provide instructions to the server on how to modify a resource. These instructions may include adding, modifying, or removing fields or attributes. The server executes the instructions and makes the changes to the resource accordingly.

5.2.7 A small recap

Method Name	Description
GET	Transfers a current representation of the target resource to the client.
HEAD	Same as GET, but doesn't transfer the response content - only the metadata.
POST	Creates new data.
PUT	Replaces the current representation of the target resource with the request content, if not found - crea
DELETE	Removes all current representations of the target resource.
PATCH	Modifies/Updates data partially.

There are other HTTP methods as well, but these are the most commonly used ones, and would suffice for most use-cases.

In the next chapter, we'll take a look at status codes.

5.2.8 The `/` path

The second component of the first line in the `cURL` output was `/`:

```
GET / HTTP/1.1
```

Let us take a moment to understand this

When it comes to browsing the internet, URLs like "http://localhost:3000/" or "http://localhost:3000" may appear to be simple at first glance, but these seemingly straightforward addresses actually hold a wealth of information that helps our browser locate specific resources on the web.

When we see "http://localhost:3000", it essentially means "http://localhost:3000/".

The trailing "/" serves as the root directory of a web server. Essentially, it points to the primary entry point or homepage of a website hosted at that specific address.

Think about these trailing slashes as separators - something that separates multiple different things.

For example, imagine you're in a library that houses various genres of books. The main entrance, equivalent to the root directory in a URL, guides you to these different sections. In the online world, the trailing slash "/" acts as a digital separator, ensuring that your browser understands the distinct paths to different resources, just like you'd follow separate signs to reach the Fiction, Science, or History sections in a library.

Let's illustrate this concept with an example involving a game website. Consider you're visiting a gaming platform at "http://games.com%22. Without any additional path, it's like arriving at the game site's main entrance. The trailing slash, though often omitted, signifies the root directory and helps organize the content.

Now, imagine you're interested in exploring racing games. You'd add a path to your URL: "http://games.com/racing%22. This is like following the sign to the Racing Games section in the library. The trailing slash, in this case, clarifies that you're entering a specific subdirectory for racing games.

Now, let's say you want to access a particular game, "VelocyX." You'd extend the URL to "http://games.com/racing/velocyx%22. This is similar to walking further into the Racing Games section to find the specific "VelocyX" game. The trailing slash here isn't necessary since you're indicating a specific resource (the game) rather than a directory.

Now, coming back to the main output from `cURL`: the `/` in `GET / HTTP/1.1` means we're trying to get to the base path. That is what we specified when we made a `cURL` request:

```
curl http://localhost:3000 -v
# is same as
curl http://localhost:3000/ -v
```

5.2.9 `HTTP/0.9`

Let's talk about the final component in the first line of the `cURL` response: `HTTP/1.1`

Let's go a little bit back than `HTTP/1.1` to understand why do we have `HTTP/1.1` in the first place.

The initial version of HTTP was very basic and didn't have a version number. It was later given the name "HTTP/0.9" to distinguish it from its successors. In its simplicity, HTTP/0.9 requests were composed of only one line and started with only one method, which was the `GET` method. And that was the reason it was called a **One-line protocol**.

The path to the resource followed the method. The full URL wasn't included in the request since once connected to the server, the protocol, server, and port were not necessary. Despite its simplicity, `HTTP/0.9` was a groundbreaking protocol that paved the way for the development of more advanced versions in the future.

There were no request/response headers, as well as no status codes. That doesn't sound too fun, is it?

5.2.10 `HTTP/1.0`

HTTP version 1.0 was a big improvement over HTTP 0.9. It added new features like metadata transmission through HTTP headers, explicit versioning, status codes that made it clear what happened with requests, the **Content-Type** field that allowed for different types of documents to be sent, and new methods that let clients and servers interact in more ways. These changes made HTTP 1.0 stronger and more flexible, which set the stage for even more improvements in later versions.

Let's talk about those in a bit more detail:

5.2.10.1 Introduction of the HTTP Header

The `HTTP/1.0` version brought significant improvements to the protocol, and one of the most pivotal enhancements was the introduction of the HTTP header.

In `HTTP/0.9`, HTTP requests were composed solely of a method (like `GET`) and the resource name (like `/about.html`). However, HTTP 1.0 expanded this approach by allowing for the inclusion of additional metadata and information alongside the request through the concept of the HTTP header. The HTTP header brought flexibility and extensibility to the protocol, enabling servers and clients to exchange important details about the request.

> *We'll talk about headers in an upcoming chapter. For now think of it as an extra meta-data.*

5.2.10.2 Versioning

Versioning became explicit, allowing each HTTP request sent to the server to include version information indicating that the request was being made using `HTTP/1.0`. This addition of version information in the request enabled more precise communication between clients and servers, ensuring compatibility and accurate interpretation of the request, whereas in `HTTP/0.9`, there was no way to explicitly indicate which version of the HTTP protocol was being used.

5.2.10.3 Status Codes

`HTTP/1.0` implemented status codes in response messages to inform clients about the outcome of their requested operation. These standardized codes indicated whether the request was successful, an error occurred, or further action was required. This enabled clients to better understand the status of their requests without solely relying on the response content.

Since `HTTP/1.1` is a successor to `HTTP/1.0`, it also has this concept of status codes. In our `cURL` example, you can see the status code on the very first line of the `response` sent by the server:

```
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK # 200 is the Status Code, which means OK
```

5.2.10.4 Content-Type Header

It also introduced a very important concept in the form of the HTTP header, specifically the **`Content-Type`** field. This field allowed servers to inform clients about the type of content being sent in the response, which was very helpful for transmitting various document types beyond just plain HTML files.

With this addition, it became easier to send images, videos, audio files, and more types of docs from the server, so the client could accurately interpret the received content and render it appropriately based on its type.

You'll notice that we don't have the `Content-Type` header in our `cURL` request, and we'll get back to that in an upcoming chapter.

5.2.10.5 New Methods

Two new HTTP methods: `POST` and `HEAD` were introduced as well. We talked about these headers in the previous section. This was particularly useful for checking things like the last modification date or the size of a resource before deciding whether to fully download it.

5.2.11 `HTTP/1.1`

HTTP 1.1 was the last version in the HTTP 1 series. After that, we got newer versions called `HTTP/2` and `HTTP/3`. We'll talk a very little about `HTTP/3` in this book, and will keep our focus towards `HTTP/2` and `HTTP/1.1`.

HTTP 1.1 made important improvements to the way web pages or clients communicate with servers. This was done by making it possible to keep a connection open and send multiple requests at once, as well as breaking up large pieces of data into smaller chunks. It also made it easier to store web page data on your computer, and added more ways for web pages to interact with servers. These changes helped make the web faster, more reliable, and more flexible.

Let's talk about the improvements in a bit more detail:

5.2.11.1 Persistent Connections

`HTTP/1.1` introduced persistent connections, also called **keep-alive** connections, which allowed multiple requests and responses to be sent and received over a single connection, unlike HTTP 1.0 which required a new connection for each request-response pair. This reduced the overhead of establishing new connections for each resource, leading to faster and more efficient communication between clients and servers.

You may ask, *what overhead did "creating new connections" cause*? I'll try my best to explain this.

- **Opening a new connection actually takes time, and system resources.** With `HTTP/1.0`, a new connection was opened for each resource request. Establishing a new connection requires several steps, including setting up a network connection, negotiating parameters, and implementing security measures if necessary. Once the request is fulfilled, the connection is closed.

  However, frequently opening and closing connections for each resource can result in inefficiencies due to the overhead involved in setting up and tearing down connections.

- **Connection limits**. To prevent issues with opening too many connections, web browsers and servers limit the number of simultaneous connections that can be open at a time. This helps prevent overloading the server and ensures that the browser can handle the responses efficiently.

  For example, in `HTTP/1.1`, browsers usually limit the number of simultaneous connections to a single domain to 6 to 8 connections. Opening too many connections can slow down performance and even cause crashes. These are useful when you want to download `CSS` files, or multiple images at once. Browsers limit the amount of open connections you could have.

  Limiting the number of connections encourages responsible resource usage and helps maintain a smooth browsing experience for users.

  `HTTP/1.1` introduced persistent connections, which send multiple requests and responses over the same connection, reducing the need to constantly open and close connections. This improves the

efficiency of web communication and reduces the impact of connection limits on performance.

> `HTTP/2` and `HTTP/3` tackle this issue even further with multiplexing and other techniques that allow multiple resources to be fetched over a single connection, minimizing the impact of connection limits and reducing latency even more.

5.2.11.2 Pipelining

This feature enables clients to send multiple requests to the server without waiting for the responses to previous requests. The server processes the requests in the order they were received and sends back the responses in the same order. Pipelining reduces communication latency by eliminating the need for the client to wait for each response before sending the next request.

> We'll understand and implement pipelining during the final chapters of building our framework.

5.2.11.3 Chunked Transfer Encoding

This allowed servers to send large amounts of data in smaller, manageable chunks. This was especially useful for dynamic content or cases where the total content length was unknown at the beginning of the response. It does so by sending a series of **buffers** with specific lengths. This makes it possible for the sender to maintain connection persistence and for the recipient to know when it has received the complete message.

5.2.11.4 The `Host` header

This was a game changing addition. It introduced the mandatory `Host` header in the request. This header specifies the domain name of the server the client wants to communicate with. This enhancement paved the way for the hosting of multiple websites on a single IP address (virtual hosting), as servers could now distinguish between different websites based on the Host header.

Prior to `HTTP/1.1`, the original `HTTP/1.0` protocol did not include the "Host" header, which meant that a single web server using a specific IP address and port number could only serve a single website. This limitation posed a significant challenge as the web began to grow rapidly.

The "Host" header made it possible for a web browser to tell a web server which website it wanted to access as part of an HTTP request. This was important because it allowed many websites to be hosted on a single server. Here are some benefits:

1. **Virtual Hosting**: A single physical server could be used to host multiple websites with different domain names. This was not possible before the "Host" header because the server couldn't determine which site the client wanted to access if multiple domain names were associated with the same IP address. With the "Host" header, the server could inspect the requested domain name and serve the appropriate content.
2. **Resource Efficiency**: Hosting multiple websites on a single server made more efficient use of resources. Without the "Host" header, separate servers or IP addresses would have been required for each hosted website, leading to wastage of resources.
3. **Cost Savings**: Hosting multiple websites on a single server reduced the cost of infrastructure, as fewer servers and IP addresses were needed. This was particularly important for smaller businesses and individuals who couldn't afford dedicated infrastructure for each website.
4. **Scalability**: The "Host" header made it easy to add new websites without needing additional physical hardware, allowing for scalable web hosting solutions.
5. **Easier Website Management**: Website owners and administrators could now manage multiple websites from a single server, simplifying maintenance and updates.
6. **Cloud Hosting**: The "Host" header helped develop cloud hosting platforms where virtualization allowed even more efficient use of resources across multiple clients and websites.
7. **Domain-based Routing**: The "Host" header also allowed for more advanced routing and load balancing strategies, as servers could make routing decisions based on the requested domain.

5.2.11.5 Caching improvements

`HTTP/1.1` refined the caching mechanisms, allowing more granular control over caching strategies. It introduced headers like `Cache-Control`, `ETag`, and `If-None-Match`, enabling servers and clients to communicate about cache validity and freshness. This led to improved cache management and reduced redundant data transfers.

Here's how:

1. **`Cache-Control Header`**: The Cache-Control header let servers and clients communicate about caching behavior of objects like images/html files etc. These instructions included:
   - **`public`**: Says the response can be cached by any intermediary (like a proxy server) and the client.
   - **`private`**: Says the response is meant for a single user and should not be cached by intermediaries.
   - **`max-age`**: Sets the maximum time a resource should be fresh in the cache.
   - **`no-cache`**: Says the resource shouldn't be used from the cache without checking with the server.
   - **`no-store`**: Tells caches, including browser caches, not to store a copy of the response under any circumstances.
2. **`ETag Header`**: The `ETag` (Entity Tag) header gave servers a way to identify versions of a resource. When a client cached a resource, it stored the `ETag` for that version. When the client wanted to check if its cached version was still valid, it could send the `ETag` in a request's `If-None-Match` header. If the server

found that the ETag was still valid, it could respond with a "304 Not Modified" status, showing that the client's cached version is still fresh.

3. **`If-None-Match` Header**: Like we just discussed, clients could use the `If-None-Match` header to give the server the `ETag` of a cached resource. The server would then compare the provided ETag with the current ETag of the resource. If they matched, the server could respond with a "304 Not Modified" status, saving bandwidth and time by not transferring the full resource.

These caching improvements had several benefits:

- **Reduced Data Transfers**: The ability to use `If-None-Match` and receive "304 Not Modified" status code meant that clients didn't have to download resources they already had cached when those resources hadn't changed on the server. This saved both bandwidth and time.
- **Faster Page Load Times**: Caching mechanisms reduced the need to retrieve resources from the server for every request. This led to faster loading times for websites as clients could use cached resources for subsequent visits.
- **Efficient Use of Network Resources**: By controlling how resources were cached and for how long, servers could use their network resources more efficiently and reduce the load on both their infrastructure and the clients'.
- **Dynamic Content Handling**: The ETag mechanism also allowed efficient caching for dynamically generated content. If the content of a page changed, the server could change the ETag, prompting clients to fetch the new content.

5.2.11.6 Range Requests

HTTP 1.1 introduced a feature called "range requests", which allows clients to request specific parts of a resource instead of the entire resource. This feature has several benefits. This made it possible to continue paused downloads, made it easier to stream media without interruptions, and saved internet data by allowing users to only download specific parts of a file. This improvement greatly enhanced the user's experience when downloading large files or streaming media on the internet. In short, here are the major benefits that `Range Request`s provided:

Resuming Interrupted Downloads: If your download is interrupted, range requests allow you to request only the missing parts, so you can resume the download from where it left off instead of starting from scratch.

Streaming Media Content: It enabled efficient media streaming by allowing clients to request parts of a video as needed. This way, the client can buffer and play parts of the video while simultaneously fetching the next parts, leading to smoother streaming.

Bandwidth Conservation: By transmitting only the necessary parts of a resource, range requests help conserve bandwidth and reduce data usage for both the client and server. Imagine having to download the entire 4K youtube video before watching it, how bad would it be? `HTTP/1.0` did not have built-in mechanisms

to efficiently support streaming or buffering for media content. When you requested a file using `HTTP/1.0`, the server would typically send the entire file in response, and the client (browser or media player) would wait until the entire file was received before it could start processing or rendering it

Implementation and Server Response: When a client sends a request with a "Range" header, it specifies the parts it needs from the resource. The server then responds with a "206 Partial Content" response, along with the requested data and additional headers indicating the content's range and length.

5.2.11.7 New Methods: `PUT`, `DELETE`, `CONNECT`, `OPTIONS`, `TRACE`

We talked about `PUT` and `DELETE` methods in the previous section. We'll focus on `CONNECT`, `OPTIONS` and `TRACE` HTTP methods.

- **`TRACE`**: method lets a client ask for a diagnostic loop-back (echo) of the request message. This means it sends back the same request back to the client. `TRACE` isn't often used in regular web applications, but it's useful for debugging and troubleshooting. It helps check how an intermediary, like a proxy server, handles and changes the request. It's a way to look at how the request changes as it goes through different nodes in the network.
- **`OPTION`**: method lets a client ask the server what it can do. When an `OPTIONS` request is sent, the server answers with a list of HTTP methods it supports for a specific resource, along with any extra options or features available. This method is helpful for figuring out what actions can be taken on a resource without actually changing the resource.

  It helps developers understand what they can do with the server, which makes it easier to create and design web applications. Very helpful because using the `OPTIONS` method allows you to gather vital information about the API's capabilities without triggering any actual actions.

  No rate limits, no authorization, nothing. It provides insights into how you can interact with the API properly, which methods you can use, and what kind of authentication you need. This is especially helpful for developers aiming to build robust and well-informed applications that interact seamlessly with APIs.

  Let's try to demo this by sending an `OPTIONS` request to stripe's endpoint and see the result. I recommend copy pasting the `cURL` command into your terminal:

```
# Make a request to `https://api.stripe.com/v1/charges` with the method `OPTIONS`
$ curl -X OPTIONS https://api.stripe.com/v1/charges \
    -H "Origin: https://anywebsite.com" \
    -H "Access-Control-Request-Method: POST" \
    -H "Access-Control-Request-Headers: authorization,content-type" \
    -v
```

```
# The backslash (\) is used to break the command into multiple lines, so it's easier to read.

# I trimmed out the request data to make it easier to read, we're only concerned with
↪ response.

* Connection state changed (MAX_CONCURRENT_STREAMS == 128)!
< HTTP/2 200
< server: nginx
< date: Fri, 25 Aug 2023 16:47:14 GMT
< content-length: 0
< access-control-allow-credentials: true
< access-control-allow-headers: authorization,content-type
< access-control-allow-methods: GET, POST, HEAD, OPTIONS, DELETE
< access-control-allow-origin: https://yourwebsite.com
< access-control-expose-headers: Request-Id, Stripe-Manage-Version,
↪ X-Stripe-External-Auth-Required, X-Stripe-Privileged-Session-Required
< access-control-max-age: 300
< strict-transport-security: max-age=63072000; includeSubDomains; preload
```

Let's analyze this:

- **`HTTP/2 200`**: The response was successful with a status code of 200. Note that Stripe is actually using `HTTP/2` instead of `HTTP/1.1`. These big companies usually upgrade their web servers and APIs to newer protocols like `HTTP/2` in this case. You should do it too!

  About `HTTP/3`, I don't think there's an actual requirement yet, but many clients don't support `HTTP/3` also migrating a code base from `HTTP/2` to `HTTP/3` is complex; might require a lot of work and require modifications to existing server configurations, load balancers, firewalls, etc.

- **`server: nginx`**: The web server being used is "nginx", probaby as a load balancer. We'll discuss about load-balancers in depth later on in this book.

- **`date: Fri, 25 Aug 2023 16:47:14 GMT`**: The server generated the response on this date and time.

- **`content-length: 0`**: The response has no content, with a length of 0 bytes.

- **`access-control-allow-credentials: true`**: The API supports credentials (such as cookies or HTTP authentication) for cross-origin requests.

- **`access-control-allow-headers: authorization,content-type`**: Only the specified headers are allowed for cross-origin requests.

  - **`access-control-allow-methods: GET, POST, HEAD, OPTIONS, DELETE`**: The listed HTTP methods are allowed for cross-origin requests.
  - **`access-control-allow-origin: <https://anywebsite.com>`**: Requests from the specified origin (domain) are allowed to make cross-origin requests. In this case, it's `https://anywebsite.com`
  - **`access-control-expose-headers: Request-Id, Stripe-Manage-Version, X-Stripe-External-Auth-Required,`** The listed headers can be exposed to the client in the response, which might contain additional information for the client.
  - **`access-control-max-age: 300`**: The client can cache the preflight response for the OPTIONS request for up to 300 seconds (5 minutes).
  - **`strict-transport-security: max-age=63072000; includeSubDomains; preload`**: The server enforces strict transport security, with the **`max-age`** value specifying how long the client should consider the server secure, **`includeSubDomains`** indicating that subdomains should also use HTTPS, and **`preload`** indicating that the domain is included in the browser's preload list for HSTS (HTTP Strict Transport Security).

  Notice how easy was it to know that what methods, headers, cors settings and other stuff does the `/v1/charges` stripe endpoint supports? We didn't even need to visit the stripe documentation for this.

- `CONNECT`: method's primary purpose is to facilitate the creation of a tunnel between the client and the destination server, often involving one or more proxies, with a **two-way communication channel**. This tunnel can then be secured using TLS (Transport Layer Security), allowing encrypted communication between the client and the destination server. Too much theory, let's take a look is it useful:

  One common use case for the `CONNECT` method is accessing websites that use `HTTPS`. When a client wants to access an HTTPS website through a proxy, it sends a CONNECT request to the proxy server. The proxy then establishes a tunnel to the destination server, allowing the client to communicate directly with the server over a secure connection.

  *Note: We'll talk about `HTTPS` in a later chapter in the course. For now you can think of it as a **secure** version of the **HTTP protocol**.*

> A proxy like a middle person between you and a website or online service. When you want to visit a website, instead of connecting directly, your request goes to the proxy first. The proxy then sends your request to the website and gets the response back. It's like the proxy is fetching the web page on your behalf.

We covered the first two lines of the `cURL` output in this chapter. In the next one, we'll take a look at **user agents**.

5.3 User agents

A user agent is any software client that sends a request for data or information. One of the most common type of user agent are the web browsers. There are many other types of user agents as well. These include crawlers (web-traversing robots), command-line tools, billboard screens, household appliances, scales, light bulbs, firmware update scripts, mobile apps, and communication devices in various shapes and sizes.

It's worth noting that the term "user agent" doesn't imply a human user is interacting with the software agent at the time of the request. In fact, many user agents are installed or configured to run in the background and save their results for later inspection or to only save a subset of results that might be interesting or erroneous. For instance, web-crawlers are typically given a starting URL and programmed to follow specific behavior while crawling the web as a hypertext graph.

In our `cURL` response, we have the user-agent listed on this line:

```
> User-Agent: curl/7.87.0
```

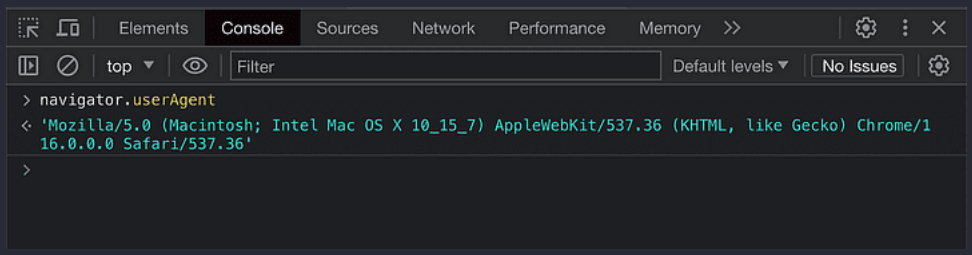
Here, the user-agent is the `cURL` command line program that we used to make the request to our server. But, it get's more strange when you make a request from a browser. Let's see that in action.

5.3.1 `User-Agent` can be weird

Open chrome or any browser of your choice, and open the web console using `Command + Option + J` on a mac or `Control + Shift + J` on windows or linux. Type the following command in the console and press enter:

```
navigator.userAgent;
```

This outputs:



Ha, such a weird looking string isn't it? What are 3 browsers doing there? Shouldn't it be only `Chrome`?

This phenomenon of "pretending to be someone else" is quite prevalent in the browser world, and almost all browsers have adopted this strategy for practical reasons. A great article that you should read if you're interested is the user-agent-string-history.

For instance, let's examine the example in question:

```
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko)
↪  Chrome/116.0.0.0 Safari/537.36
```

It is a representation of Chrome 116 on a Macintosh. But why is it showing Mozilla, even though we are not using it?

```
Why does a browser behave that way?
```

The answer is simple. Many websites and web pages are designed to identify which browser is visiting the page, and only deliver certain features if the browser is Chrome/Mozilla/... (although this is a BAD programming practice, (we'll get to it in a bit) it happens, especially in the early days of the Internet). In such cases, if a new browser is released and wants to provide the same features for that page, it must pretend to be Chrome/Mozilla/Safari

Does that mean that I cannot find the information I need from the `User-Agent` header?

Not at all. You can definitely find the information you need, but you must match the `User-Agent` precisely with the User-Agent database. This can be a daunting task because there are more than **219 million** user agent strings!

Like I told earlier, checking the type of browser to do things separately is a very bad idea. Some reasons include:

- Malicious clients can easily manipulate or spoof user agents, causing the information provided in the user agent string to be potentially inaccurate.
- Modern browsers often use similar user agent structures to maintain compatibility with legacy websites, making it difficult to accurately identify specific browsers or features based solely on user agents.
- The rise of mobile devices and Internet of Things (IoT) devices has led to a diverse range of user agents, making it challenging to accurately identify and categorize all types of devices.
- Some users might switch user agents to access restricted content or to avoid being tracked, resulting in unexpected behavior when making decisions based on user agent data.
- Many alternative and lesser-known browsers and user agents do not follow standard patterns, so relying solely on well-known user agents could exclude valid users.
- Tying application behavior too closely to specific user agents can hinder future updates or improvements and complicate the development process by requiring constant adjustments.

You can find a very detailed explanation on why this is bad here - Browser detection using the user agent and what alternate steps you could take in order to create a feature that's not supported on some browsers.

5.4 MIME Type and `Content-Type`

In this chapter, we're going to take a look at the next line of the `cURL` output:

```
> Accept: */*
```

This line is specifies a header named `Accept` which has the value of `*/*`. This is very crucial. Let's take a moment to understand, why.

5.4.1 Understanding the `Accept` Header

The **`Accept`** header is important in an HTTP request, especially when your client (in this case, `cURL`) wants to tell the server what types of media it can handle. This header tells the server the types of media or MIME types that the client wants in the response body. Essentially, it's like the client's way of saying, "I'm open to receiving content in these formats."

5.4.1.1 Breaking Down the Line

In the cURL output, the line **`> Accept: */*`** might look confusing, but it means something simple. Let's break it down:

- **`>`**: This symbol shows that the line is part of the request headers sent from the client (you, via cURL) to the server.
- **`Accept: */*`**: This is the actual **`Accept`** header value. The `*/*` part means "anything and everything." It shows that the client is willing to accept any type of media or MIME type in the response. In other words, the server has the freedom to choose the most suitable format to send back.

5.4.1.2 Why the Wildcard?

You might wonder why the client would use such a general approach. The reason is flexibility. By using **`*/*`**, the client is showing that it can handle many content types. This can be useful when the client doesn't care about the specific format or when it's okay with multiple formats. The server can then choose the most appropriate representation of the resource based on factors like its capabilities and available content types.

5.4.1.3 Server Response

Based on the **`Accept: */*`** header, the server should create a response that matches the client's willingness to accept any media type. The server chooses the most suitable **Content-Type** from its available options and include it in the response headers.

5.4.2 Mime Type

You are likely already familiar with MIME types if you have engaged in web development, particularly when including JavaScript files into HTML documents. For example, the line below might look familiar:

```
<script type="text/javascript"></script>
```

The `type` attribute on the script tag above is also a MIME type. It consists of two parts - a `type` i.e **text** and a `subtype` i.e **javascript**.

MIME type(s) are a critical part of how the web works. It stands for Multipurpose Internet Mail Extensions, and are often called "media types". They function like tags that are attached to the content shared across the internet, in order to provide information about the type of data contained within them. This information allows browsers and applications to properly process and display the content to the user.

For example, a plain text document may have a different MIME type than an image or an audio file. Additionally, even within the same category, such as images or audio files, there may be different formats that require different MIME types. This is because each file format has unique characteristics that need to be accounted for in order to ensure proper display and functionality.

For example, the MIME type `image/jpeg` means the file has a JPEG image and `audio/mp3` means the file has an MP3 audio. These labels are important so that web browsers can display content correctly and multimedia players can play the right kind of media file.

Without them, web pages would be confusing and multimedia files wouldn't work right. To make sure files work right, we include the right MIME type label when uploading to a website or sending them through email.

You can find a exhaustive list of all the MIME types on Media container formats (file types)

5.4.3 Anatomy of a MIME type

A MIME type has two parts: a "type" and a "subtype." These parts are separated by a slash ("/") and have no spaces. The "type" tells you **what category the data belongs** to, like "video" or "text." The "subtype" tells you exactly **what kind of data it is**, like "plain" for plain text or "html" for HTML source code. For example:

```
text/plain
image/png
image/webp
application/javascript
application/json
```

Each "type" has its own set of "subtypes." All MIME types have a "type" and a "subtype."

You can add more information with an optional "parameter." This looks like "type/subtype;parameter=value." For example, you can add the "charset" parameter to tell the computer what character set to use for the text. If you don't specify a "charset," the default is ASCII. To specify a UTF-8 text file, the MIME type "text/plain;charset=UTF-8" is used.

MIME types can be written in uppercase or lowercase, but lowercase is more common. The parameter values can be case-sensitive.

We won't be diving too much into MIME types just yet, we'll come back to these when we start working on our backend library.

5.4.4 But why the wildcard `*/*`?

The wildcard `*/*` approach is a versatile strategy. It's like telling the server, "I'm flexible. Show me what you've got, and I'll adapt." This can be handy when you're browsing web pages with a mix of images, videos, text, and more. Instead of specifying a narrow set of MIME types, you're leaving room for surprises.

So, when you see `> Accept: */*` in your cURL output or in any request header, remember that it's your browser's (or client's) way of embracing the diversity of the digital marketplace. It's a friendly nod to MIME types, indicating that you're ready to explore whatever content the server has to offer.

5.4.5 The `Content-Type` header

The `Content-Type` header tells what kind and how the data is sent in the request or response body. It helps the receiver understand and handle the content correctly. The `Content-Type` header can be set on response headers, as well as the request headers.

> Note: The value of the `Content-Type` header should be a valid MIME type.

5.4.5.1 `Content-Type` on request header

When a client sends an HTTP request to a server, the `Content-Type` header can be included to inform the server about the type of data being sent in the request body. For example, if you're submitting a form that includes file uploads, you would specify the `Content-Type` header to match the format of the uploaded file. This helps the server understand how to process the incoming data.

> We haven't reached the response part of the `cURL` request yet, but for now just bare with me.

Here's an example of including `Content-Type` in a request header:

```
POST /accounts HTTP/1.1
Host: github.com
Content-Type: application/json
```

5.4.5.2 `Content-Type` on response header

In a response by a server, the `Content-Type` header informs the client about the format of the content in the response body. This helps the client, such as a browser, to properly interpret and render the received data. For instance, when a server sends an HTML page to a browser, it specifies the `Content-Type` as `text/html`.

Here's an example of including `Content-Type` in a response header:

```
HTTP/1.1 201 CREATED
Content-Type: text/html; charset=UTF-8
Content-Length: 10
```

5.4.6 The `charset=UTF-8`: character encoding

The **`charset`** parameter in the **`Content-Type`** header tells which character encoding is used for text-based content. Character encoding specifies how characters are represented as binary data (bytes). Each character encoding supports different character sets and languages.

5.4.6.1 Universal Character Encoding

`UTF-8` stands for a character encoding that can represent almost all characters in the Unicode standard. Unicode contains many characters used in different languages and scripts all over the world.

Significance in HTML Content:

When you use **`charset=UTF-8`** in HTML, it means that the content is using the UTF-8 character encoding. This is important because it makes sure that characters from different languages and scripts will show up correctly in browsers and other apps.

For example:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>UTF-8 Example</title>
  </head>
```

```
  <body>
    <h1>Hello, □□, □□□□□</h1>
  </body>
</html>
```

In this HTML markup, the `<meta charset="UTF-8">` tag inside the `<head>` tag specifies that the document is encoded using UTF-8. This allows the browser to accurately render characters from multiple languages, such as English, Chinese, Tamil, and Japanese, all in the same document.

Universal Compatibility:

Using UTF-8 as the character encoding ensures universal compatibility, as it can represent characters from various languages without any issues. It is a popular choice for web content due to its versatility and support for a wide range of characters.

This should be enough for a basic understanding of the `Content-Type` header and the MIME type. We'll start talking about the response part of the `cURL` output in the next chapter.

5.5 Headers

Before diving into the response part of the original `cURL` request, let's talk a bit about headers in a HTTP Request and Response.

HTTP headers are important for communication between clients, usually browsers, and servers on the web. Headers are a way to provide more info than just the basic request or response.

One example of an HTTP header is the `Authorization` header, which is used to provide authentication information for a request. The header name is "Authorization" and the header value typically includes the authentication scheme and credentials.

For example, a basic authorization header might look like this: "Authorization: Basic dXNlcjpwYXNzd29yZA==". In this case, the authentication scheme is "Basic" and the credentials are the base64-encoded string "user:password".

The way this header would appear in an HTTP request would be something like this:

```
Authorization: Basic dXNlcjpwYXNzd29yZA==
```

Let's look at the syntax a bit more closely.

5.5.1 Header Name

The header name is case-insensitive, meaning that it doesn't matter whether you use uppercase, lowercase, or a mix of both when specifying the header name. This allows for more flexibility when working with HTTP headers. In our example, the header name is `Authorization`.

> Note: HTTP/2 enforces lower case header names, or it treats the header names with mixed casing or upper casing as malformed.

5.5.2 Colon (:)

The colon (`:`) separates the header name from its corresponding value. This helps to distinguish the name of the header from its value and makes it easier to read and understand. Without the colon, the header name and value would run together and be difficult to identify.

Think of headers somewhat similar to a JSON key-value pair, except the double quotes. For example `"key": "value"`.

5.5.3 Header Value

The value of the header follows the colon. This value provides specific information relevant to the context of the request or response. For instance, the `"Content-Type"` header might have a value like `"application/json"` to indicate that the content being sent is in JSON format.

Other examples of header values include timestamps, authentication tokens, and content length. We'll take a look at common header values in a bit.

5.5.4 Whitespace

Leading whitespace (spaces or tabs) before a header value is ignored to improve readability of transmitted data and ensure proper parsing by the recipient. However, trailing whitespace after the header value is not ignored and may cause issues if not removed.

5.5.5 Custom `X-` based headers

Developers and organizations used to add extra information to HTTP requests and responses using custom headers. These headers would start with `"X-"` to show that they were not part of the official HTTP standard.

For example, a website might use a custom header like `"X-Session-ID"` to send extra information about the user's session. But using custom headers had some problems.

First, some custom headers became so popular that they were added to the official HTTP standard, causing confusion.

Second, different developers used custom headers in different ways, making it hard to understand them.

In 2012, custom headers starting with `"X-"` were officially discouraged. Instead, developers were encouraged to use an official registry of HTTP headers managed by the Internet Assigned Numbers Authority (IANA).

This registry lists all the official HTTP headers, including ones that used to be custom. The IANA also has a registry for new headers that developers suggest. This helps ensure that new headers are considered carefully before becoming standard.

> In short, custom headers starting with `"X-"` are no longer recommended.

5.6 Request headers

Request headers are sent from the client to the server and contain additional information about the client's intent and the resources it wants to access. In the `cURL` output that we previously saw, we had a Request header:

```
Accept: */*
```

Request headers provide the server with guidance on how to handle the request and tailor the response accordingly. By including request headers, the client can inform the server about the preferred language and encoding, any authentication credentials, and more.

This additional information helps the server to better understand the client's needs and respond appropriately.

The request headers can also provide valuable context to intermediaries such as proxies and gateways, helping them to correctly route and handle the request as it travels through the network.

Therefore, it is essential to include accurate and relevant request headers to ensure successful communication between the client and server.

Let's take a look at some of the common request headers:

5.6.1 Accept

We previously talked about the `Accept` header in the ”Mime Type and Content Type” chapter. To re-iterate, it is used by the client to communicate the media types or content types that it can understand and handle.

This header allows the server to select an appropriate format for the response, taking into account the client's preferences. Without the "Accept" header, the server may send a response in a format that the client cannot interpret, resulting in a communication breakdown.

Additionally, the "Accept" header can include multiple media types, allowing the server to select the best format from the list of media types that the client can handle. This flexibility provides more options for the server to deliver the response, enhancing the user experience.

An example of `Accept` headers that include multiple media types is given below:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
```

It supports a lot of media types (or "content-type"s). One thing to note is the `q=0.9` and `q=0.8` parameter. It is called **relative quality factor** or in short **quality factor**. This says, the higher the value of `q` is the more the preference of that content-type is.

Let's break down the entire header value stp-by-step

1. `text/html`
    - This media type represents HTML content, which is used to structure and present web content.
    - There is no specified q-value, so it's assumed to be 1 (highest preference).
2. `application/xhtml+xml`
    - This media type represents XHTML content, which is an XML-based version of HTML.
    - There is no specified q-value, so it's assumed to be 1 (highest preference).
3. `application/xml;q=0.9`
    - This media type represents XML content, which is used for data interchange and structured documents.
    - The q-value is 0.9, indicating a slightly lower preference compared to HTML and XHTML.
4. `image/webp`
    - This media type represents WebP images, a modern image format that offers efficient compression.
    - There is no specified q-value, so it's assumed to be 1 (highest preference).
5. `image/apng`
    - This media type represents Animated Portable Network Graphics (APNG) images.
    - There is no specified q-value, so it's assumed to be 1 (highest preference).
6. `*/*;q=0.8`
    - `*/*` is a wildcard media type that matches any type of content.

- The q-value is 0.8, indicating a preference that is lower compared to specific formats.

5.6.2 Referer

The "Referer" specifies the URL of the page that led the client to the current page. This header is often used to understand the client's navigation context and referral sources.

By analyzing this information, websites can gain valuable insights into how users interact with their site and where they come from.

This header can be used to track user behavior and provide personalized recommendations or targeted advertising. It is important to note, however, that some browsers and privacy plugins may block or modify the Referer header, which can affect the accuracy of the data collected.

A very simple example is the github analytics. Open any of your open-source (public) repository and visit the following link:

```
https://github.com/ishtms/learn-nodejs-hard-way/graphs/traffic
```

Replace `ishtms` with your github username, and `learn-nodejs-hard-way` with your repository slug. On the bottom of this page, you'll see this section:

Referring sites

Site	Views	Unique visitors
github.com	4,409	700
reddit.com	2,522	539
Google	437	71
old.reddit.com	275	31
cnodejs.org	218	56

So, github is actually tracking the visitors based on the `Referer` request header! Isn't it cool to know where the users on your repository came from?

Here's an example of how the `Referer` header may look like:

```
Referer: https://www.reddit.com/r/sub-reditname/post-id
```

5.6.3 Authorization

The `Authorization` header is a vital component of secure communication between clients and servers. It carries authentication information, such as tokens or credentials, which are used to verify the identity of the user and grant access to protected resources.

Without this header, access to restricted content would be impossible, leaving sensitive data vulnerable to unauthorized access. Therefore, it is essential to ensure that the Authorization header is implemented correctly and that it is encrypted to protect against potential security threats.

> There are other ways to send the token/credentials, like the response body, but that is not considered standard. Prefer the `Authorization` header to send Authorization/Authentication specific data.

An example of how the `Authorization` header may look like is given below:

```
Authorization: Bearer eyJhcGciOiJIUzT1NiIsInR5cCI6IkpXVCJ9
# or
Authorization: Basic DWxcZGRpbjpPcGVuU2VzYW1l
```

5.6.4 Cookie

> This is a type of header which is both - sent by a client to the server, and from server to the client. We'll see one more, i.e the `Content-Type` header.

When a user requests a website, the server sends back a response with a `"Cookie"` header that contains previously stored cookies. These cookies are t**hen sent back** to the server with future requests, allowing the server to maintain state and session information across requests.

This information can include user preferences, login status, and other data that can be used to personalize the user's experience on the website.

Without cookies, the server wouldn't be able to recognize the user from one request to another, and the user would have to log in again with every new request. Therefore, cookies play a crucial role in enabling a seamless and personalized browsing experience for users.

An example of how you could send the `Cookie` header in a request is given below:

```
Cookie: session_id=b1dxc5QrdR; prefers_notif=no
```

1. **session_id=b1dxc5QrdR**

This cookie holds a session id. Sessions are a way to maintain user-specific data across different requests. The `session_id` value of `b1dxc5QrdR` is a unique identifier associated with a user's session on the server.

It allows the server to link requests to a particular user.

2. **`;`**

   Multiple session keys are separated using a semi colon (`;`).

3. **prefers_notif=no**

   - This cookie also holds the user's notification preference, which tells whether to show this user a browser notification or not. This is just a dummy example, and is not how you'd manage notification preferences.

5.6.5 Host

> We talked about the `Host` header in the chapter 5.1.

The `"Host"` header specifies the domain name of the server being addressed. This is particularly important in the cases where one physical server hosts multiple domains.

Without the `"Host"` header, the server would not know which domain the client is requesting, and would be unable to serve the appropriate content.

By including the `"Host"` header in the request, the client can ensure that it receives the correct response from the server. This simple yet essential piece of information helps to ensure the smooth functioning of the internet and the web as we know it.

Here's an example of the `Host` header in a request:

```
Host: www.eaxmple.com
```

5.6.6 Content-Type (request)

We talked about `Content-Type` in the previous section. This is another example of headers that are used both in a Request and the Response.

This header is typically found in response headers. However, it can also be included in requests, such as POST requests, to provide additional information regarding the format of data being transmitted in the request body.

This header is particularly important because it allows the receiving end to correctly interpret and process the data. This can be especially critical in situations where the data being sent is complex or requires specific handling instructions, such as in the case of multimedia files or other specialized file formats.

Thus, while the "Content-Type" header may seem like a small and unimportant piece of information, it is actually a vital component of many requests and responses, helping to ensure that data is transmitted correctly and interpreted accurately.

Here's an example of how the `Content-Type` header may be used in the requests:

```
Content-Type: application/json
```

5.7 Response Headers

When a client sends a request to a server, it is the server's responsibility to send back a response with the appropriate headers. These headers contain information such as the type of content that is being sent, the status of the response, and any caching instructions that the client should follow.

For example, the `Content-Type` header, like we just discussed, indicates the format of the data being sent back to the client, such as HTML, JSON, or XML. The `Cache-Control` header specifies how long the client should cache the response before requesting it again from the server.

In addition to these standard headers, servers can also send custom headers that provide additional information specific to the application or service. For instance, an API might include a custom header containing a unique authentication token that the client can use to make subsequent requests. We'll use this technique while building our backend library, in the upcoming chapters.

5.7.1 Content-Type (response)

This header specifically deals with indicating the format or MIME type of the content being sent or received. It plays a pivotal role in ensuring that the client and server understand how to process the exchanged data accurately.

In response headers, the `Content-Type` header informs the client about the format of the content being sent by the server. For example, if a server is sending an HTML page as the response, it includes the `Content-Type` header to specify that the content is in HTML format:

```
> HTTP/1.1 200 OK
> Content-Type: text/html; charset=UTF-8
```

Here, the `Content-Type` header indicates that the content is HTML, and the `charset` attribute is an extra parameter which specifies the character encoding used. We talked about the extra parameters in the previous

chapter (MIME Type and Content-Type)

5.7.2 Cache-Control

The Cache-Control header is a really important aspect of the HTTP protocol that web developers should not overlook. This header provides directives to both client and intermediary caches on how to handle the response from the server. It is highly recommended to use this header in every HTTP response because it plays a crucial role in optimizing performance and content freshness by controlling caching behavior.

An intermediary cache, is a special server that sits between the client and the original server, serving as a middleman for requests and responses. The main job of an intermediary cache is to keep copies of often requested resources, like web pages, images, and other content, to decrease waiting time and network traffic.

There are two main types of intermediary caches:

1. **Proxy Cache:** A proxy cache is like a middleman between employees' devices and the internet. When someone requests a web page, the proxy cache checks if it has a copy of that page and sends it straight to the person if it does. This saves time and internet data because the page doesn't have to be downloaded from the internet multiple times.
2. **CDN (Content Delivery Network):** A CDN is a network of distributed intermediary caches placed in various locations around the world. One of the examples is AWS Cloudfront. CDNs are primarily used to deliver web content (like images, stylesheets, scripts) to end-users more efficiently. When a user requests content from a website using a CDN, the request is routed to the nearest CDN server. If the requested content is cached on that server, it's delivered quickly.

5.7.2.1 How Caches Work:

When someone requests a web page, the cache checks if it already has a copy of that web page. If it does, and the copy is still fresh (based on the expiration time and the `Cache-Control` header), the cache sends that copy to the person right away. This saves time and internet data. If the cache doesn't have a copy of the web page, it asks the origin server for the page. Once the origin server responds, the cache stores a copy of the page so it can send it to someone else who requests it in the future without asking the origin server again.

By specifying directives such as `max-age` and `public`, the server can communicate with the cache to determine how long the response should be cached and whether it can be cached by public caches or not. This way, the server can ensure that visitors get the latest version of the content, while avoiding unnecessary requests and thereby, improving the overall performance of the website.

5.7.2.2 Cache-Control Directives:

The Cache-Control header is used to provide directives to caching systems, instructing them on how to handle a response. It can specify whether a response should be cached, for how long, and whether it can be reused for subsequent requests. There are several common directives that can be used with this header.

- `public`: Indicates that the response can be cached by both the client and intermediary caches. This is typically suitable for content that is meant to be shared publicly.
- `private`: Specifies that the response can be cached by the client but not by intermediary caches. This is useful for content that is specific to the individual user.
- `max-age`: Sets the maximum time (in seconds) for which the response can be cached. After this time elapses, the cached response becomes stale and should be revalidated with the server.
- `no-cache`: Instructs caches to revalidate the response with the server before using a cached copy, even if it appears to be fresh.
- `no-store`: Requires caches to not store the response at all. Each request/response cycle must involve contacting the origin server.

Let's see some of the examples on how we can use these directives with the `Cache-Control` header:

5.7.2.2.1 Always Cache (infrequent updates) Suppose a website has static assets like images, stylesheets, and JavaScript files that rarely change. To improve performance, the server can allow public caching of these assets by specifying the `Cache-Control` header:

```
Content-Type: image/webp
Cache-Control: public, max-age=604800
```

If the `max-age` is set, and has the `public` directive, this means that the image is marked as publicly cacheable for up to 7 days (`max-age=604800`).

5.7.2.2.2 Always Cache (private only) Consider a website that displays recommendations for each user based on their browsing history. These recommendations are relatively stable and can be cached on the user's device for a limited time to improve performance, and we do not wish the intermediate servers to cache this info. The `private` cache directive with the `max-age` directive can be used in this scenario.

```
Content-Type: application/json
Cache-Control: private, max-age=3600
```

The `max-age=3600` directive specifies that the cached response can be used for up to 3600 seconds (1 hour). This means that the user's device will display the cached recommendations for subsequent interactions within the next hour.

5.7.2.2.3 Never Cache (realtime data) For content that can change frequently, like social feed or your messages feed, the server might use the `no-cache` directive to ensure that clients revalidate the content on each request:

```
Content-Type: text/html
Cache-Control: no-cache, no-store, must-revalidate, max-age=0
```

When all these directives are combined as `'no-cache, no-store, must-revalidate, max-age=0'`, the goal is to prevent caching entirely and ensure that each request to the resource results in a revalidation with the origin server. This combination is useful for scenarios where content should always be dynamically generated and where no cached copy, regardless of freshness, is considered acceptable.

For example, if a web application displays real-time data, like stock prices or live news updates, this cache-control combination ensures that users always receive the most current information by fetching it directly from the origin server. It minimizes the risk of outdated data being displayed to users due to cached copies.

We looked at `no-cache`, `no-store` and `max-age` in the **Cache-Control directives** section. Let's take a look at the `must-revalidate` directive:

The **`must-revalidate`** directive emphasizes that the cached response must be revalidated with the origin server before being used, even if it's marked as fresh. If the cached response has expired, it must not be used without revalidation. It's an extra level of security to ensure up-to-date content.

5.7.3 Set-Cookie

The `Set-Cookie` header plays a key role in maintaining the state of a user's session. When a server sends a response to a client's browser, it can include instructions to store cookies, which are small pieces of data that are stored locally on the client's machine. These cookies can then be sent back to the server with subsequent requests, allowing the server to identify the client and maintain a record of their activity.

Cookies are widely used on the web for a variety of purposes, including tracking user sessions, personalizing user experiences, and more. For example, a website might use cookies to remember a user's login credentials so that they don't have to enter them every time they visit the site. Cookies can also be used to store user preferences, such as language, font size or theme (dark or light) for a more personalized browsing experience.

> We looked at the `Cookie` header earlier, which is sent as request header, on the contrary the `Set-Cookie` is sent as a response header.

Some of the most important components of the `Set-Cookie` header include expiration date (`expires`), the `path` attribute, the `domain` attribute, the `secure` attribute, and the `HttpOnly` attribute. These elements

work together to ensure that cookies are delivered to the correct domain, are only accessible via secure connections, and cannot be accessed by malicious scripts or third-party applications.

- **`expires`**: attribute allows you to specify the exact date and time when the cookie will expire. Once expired, the cookie will be considered invalid and will no longer be sent back by the client. By setting an expiration date, you can control how long the cookie will remain active on the user's computer.

  > Note: If the `expires` attribute is not specified, the cookie becomes a **session cookie**. A session finishes when the client shuts down, after which the session cookie is removed. Many web browsers have a *session restore* feature that will save all tabs and restore them the next time the browser is used. Session cookies will also be restored, as if the browser was never closed.

- **`domain`**: This attribute allows you to specify the domain for which the cookie is valid. You can set it to a specific domain or a subdomain.

- **`path`**: This attribute allows you to specify the URL path for which the cookie is valid. The cookie will only be sent to the server if the requested path matches the specified path. This means that you can control which pages on your website can access the cookie.

  Let's say you set the `path` attribute as `/blog`:

  - If you type `/blog`, `/blog/`, `/blog/post`, or `/blog/post/1` as the request path, it will work.
  - But if you type `/`, `/docs`, `/blogpost`, or `/v1/blog` as the request path, it won't work.

- **`Secure`**: This attribute instructs the client to send the cookie only over secure (HTTPS) connections. This ensures that the cookie is transmitted securely and cannot be intercepted by attackers. By using this attribute, you can protect sensitive information stored in the cookie.

- **`HttpOnly`**: This attribute prevents client-side scripts from accessing the cookie through JavaScript. This enhances security by protecting sensitive information from being accessed by malicious scripts. By using this attribute, you can ensure that the cookie is only accessible through the server-side code. This mitigates attacks against cross-site scripting (XSS).

An example response using `Set-Cookie`:

```
Set-Cookie: userprefs=language=en; currency=INR; expires=Thu, 31 Dec 2023 23:59:59 GMT; Path=/;
↪  Secure; HttpOnly
```

5.8 Response and Status Codes

It's time to discuss the response part of our `cURL` output, that is:

```
< HTTP/1.1 200 OK
< Date: Wed, 23 Aug 2023 13:13:32 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Content-Length: 11
<
* Connection #0 to host localhost left intact
Hello world%
```

This is the response sent by the server, which in case is our simple Node.js HTTP server. Let's go through this line by line:

```
< HTTP/1.1 200 OK
```

This line indicates the HTTP protocol version, i.e `HTTP/1.1` and the response status code (we'll look at status code in a bit). In this case, the response status code is `200`, which means `OK` or that the request was successful. The server has successfully processed the request and is sending back the requested resource.

Imagine you're requesting a webpage from a server, and you get a `200 OK` status code. It's like receiving a green light from the server, indicating that the page you requested has been found and is being sent back to you, and everything went OK.

```
< Date: Wed, 23 Aug 2023 13:13:32 GMT
```

This shows the date and time when the response was generated on the server.

```
< Connection: keep-alive
```

This line specifies the `Connection` HTTP header that is used to indicate that the connection between the client (our curl command) and the server will not be closed after this request-response cycle.

Instead, it will be kept alive so that subsequent requests can be made without the overhead of establishing a new connection each time. By keeping the connection open, the client and server can exchange data more efficiently and with less latency, which can lead to improved performance and faster response times.

The value of `keep-alive` is particularly useful for applications that require frequent requests to be made to a server, such as real-time web applications or streaming services. But for bi-directional communication we often use `Connection` header with the value `Upgrade`.

The `Connection` header can also have a value of `upgrade`. This means the server might want to change the way it communicates with the client. The new way might be better for certain types of actions, like sending a lot of data at once.

For example, we can upgrade from HTTP 1.1 to HTTP 2.0, or from an HTTP or HTTPS connection into a WebSocket.

> In HTTP/2 or HTTP/3, using the header Connection is not allowed. This header should only be used in HTTP/1.1.

```
< Keep-Alive: timeout=5
```

`Keep-Alive` is an another header, which is closely related to the `Connection` header that we talked about previously. It has a value, that has an attribute named `timeout` with a value of `5`.

It specifies that the server will close the connection if no new requests are made within 5 seconds. This is particularly useful in preventing idle connections from hogging server resources unnecessarily.

Imagine if a server had to keep a connection open indefinitely, even when no data is being transferred. This would lead to a significant waste of resources, which could have been better allocated to active connections. This ensures that the server can efficiently manage its resources, leading to optimal performance and a better user experience.

There is no "standard" value for the timeout duration, but here are some recommended guidelines that I've been following for my applications:

1. **Short-Lived Requests**: If the application involves short-lived requests and responses, where clients frequently make requests and receive responses within a short time frame, a relatively low timeout value might be suitable. A value between 1 to 10 seconds could work in this case.

   For example, a chat application where messages are sent frequently between users might benefit from a lower timeout value.

2. **Long-Polling or Streaming**: If the application involves long-polling or streaming, where clients maintain a connection for an extended period to receive real-time updates, a longer timeout value would be appropriate. Values between 30 seconds to a few minutes might be reasonable.

   For eg. a stock market monitoring application that provides real-time updates to traders might use a longer timeout to keep the connection open while waiting for market changes.

3. **Low Traffic and Resource Constraints**: If your server has limited resources and serves a low volume of requests, you might want to use a shorter timeout to free up resources more quickly. A value around 5 to 15 seconds could be considered.

4. **High Traffic and Scalability**: In scenarios with high traffic and the need to maximize server efficiency, a slightly longer timeout value might be chosen. This allows clients to reuse connections more frequently, reducing the overhead of connection establishment. Values around 15 to 30 seconds could

be appropriate.

Remember that the chosen timeout value should strike a balance between keeping connections open long enough to benefit from connection reuse and not tying up server resources unnecessarily.

> **Warning:** Headers that are specific to the connection, such as `Connection` and `Keep-Alive`, are not allowed in HTTP/2 and HTTP/3.

```
< Content-Length: 11
```

We talked about Content-Length earlier, to revisit: It's the length of the response body in bytes. The body in our case is the `Hello world` text.

```
<
```

This line represents a blank line that separates the response headers from the response body. It indicates the end of the header section and the start of the actual content being sent back in the response.

```
* Connection #0 to host localhost left intact
```

After the `cURL` completes the HTTP request and receives the response from the server, it keeps the network connection open and in a "keep-alive" state. This means that the TCP connection established for the request-response cycle is maintained and not terminated immediately. The connection is "left intact" to allow for the possibility of reusing it for subsequent requests to the same host.

This behavior aligns with the idea of connection reuse that we talked about previously, where keeping the connection open reduces the overhead of TCP's connection establishment and termination.

5.8.1 `Connection: close` in action

Here's the current code for our HTTP server, that we wrote in the chapter `5.0`:

```
// file: index.js

const http = require("node:http");

function handle_request(request, response) {
  response.end("Hello world");
}
```

```
const server = http.createServer(handle_request);

server.listen(3000, "localhost");
```

Let's add two more headers: `Content-Type` and `Connection` in our response.

```
// file: index.js

...

function handle_request(request, response) {
    /** Set the "Content-Type" header to "text/plain" */
    response.setHeader("Content-Type", "text/plain");

    /** Set the "Connection" header to "close" */
    response.setHeader("Connection", "close")

    /** Write the response body */
    response.end(`
Request method: ${request.method}
Request URL: ${request.url}
Request headers: ${JSON.stringify(request.headers, null, 2)}
HTTP Version: ${request.httpVersion}
HTTP Major Version: ${request.httpVersionMajor}
HTTP Minor Version: ${request.httpVersionMinor}
`);
}

...
```

Let's re-start our server using `node index`, and try to send an `HTTP` request again.

```
> curl http://localhost:3000 -v
*   Trying 127.0.0.1:3000...
* Connected to localhost (127.0.0.1) port 3000 (#0)
> GET / HTTP/1.1
> Host: localhost:3000
> User-Agent: curl/7.87.0
```

```
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Content-Type: text/plain
< Connection: close
< Date: Wed, 30 Aug 2023 01:16:14 GMT
< Content-Length: 156
<

Request method: GET
Request URL: /
Request headers: {
  "host": "localhost:3000",
  "user-agent": "curl/7.87.0",
  "accept": "*/*"
}
HTTP Version: 1.1 1.1
* Closing connection 0
```

That's too much to look at. Let's look at some key points.

Since we manually specified the `Content-Type` header in the `handle_request` function, it shows up as a part of the response, it did not previously.

The `headers` property of the `request` object provides access to all key-value pairs of the headers. Additionally, you can access the complete HTTP version, as well as the major and minor versions. This is useful if you want to upgrade an `HTTP/1.1` connection to a `WebSocket` connection or `HTTP/2`.

The last line now says `* Closing connection 0`. This is because we set the `Connection: close` header, which instructs the client to immediately close the connection and not use it again for subsequent requests.

It also indicates that the HTTP method is `GET`, which is somewhat strange since we did not specify it ourselves. We instructed `cURL` to make a request to the `http://localhost:3000` URL. Why was it automatically added?

Well turns out, by default every request is assumed to be a `GET` request even if we do not specify a method. Let's try to change the method to `POST`. With `cURL` you can set methods like this:

```
curl -X POST http://localhost:3000 -v
```

```
# or

curl --request POST http://localhost:3000 -v
```

By sending a `POST` request, our response output changed to:

```
...

Request method: POST
Request URL: /
Request headers: {
  "host": "localhost:3000",
  "user-agent": "curl/7.87.0",
  "accept": "*/*"
}
HTTP Version: 1.1 1.1
* Closing connection 0

...
```

The code seems to repeat the `setHeader` method call two times. What if we have 10-12 response headers that we wish to set? Is there a better way? Yes, there is: `response.writeHead()`

Let's look at the function signature:

```
writeHead(
    statusCode: number,
    headers?: OutgoingHttpHeaders | OutgoingHttpHeader[]
): this;
```

This is written in Typescript. Don't worry if you do not know TypeScript. I'll explain it to you in easy words.

In Typescript, you have to specify the `type` of every parameter for a function. In this case, the first parameter - `statusCode` argument has a type of number. The second parameter `header` has a type of either `OutgoingHttpHeaders` or `OutgoingHttpHeader[]`. Before looking at these types, let's talk about `?` in `headers?`.

The `?` specifies that the parameter is optional. So, if you do not provide any argument for `headers`, it will work fine and not throw any error.

`OutgoingHttpHeaders` is a type representing a collection of outgoing HTTP headers. Each header is defined

as a property with the header name as the property key and the header value as the property value. This type provides a structured way to specify headers.

`OutgoingHttpHeader[]` is an array of headers, where each header is defined as an object containing a header name (string) and one or more corresponding header values (string[]).

This provides a lot of flexibility in how you provide headers when calling the `writeHead` function. But the preferred way is to use an object with keys. Let's update our code snippet to use `response.writeHead` instead of `response.setHeader`

```
// file: index.js

...

function handle_request(request, response) {
    response.writeHead(200, {
        "Content-Type": "text/plain",
        "Connection": "close"
    })

    ...
}

...
```

Now the code looks much more easier to reason about. One more thing that could be an issue if you're working with any Node.js project that we talked about earlier.

The editor has no way to identify the type of `request` and `response` arguments in the `handle_request` method, and because of that you can not enjoy the intellisense or autocompletion while trying to access properties of `request` or `response`.

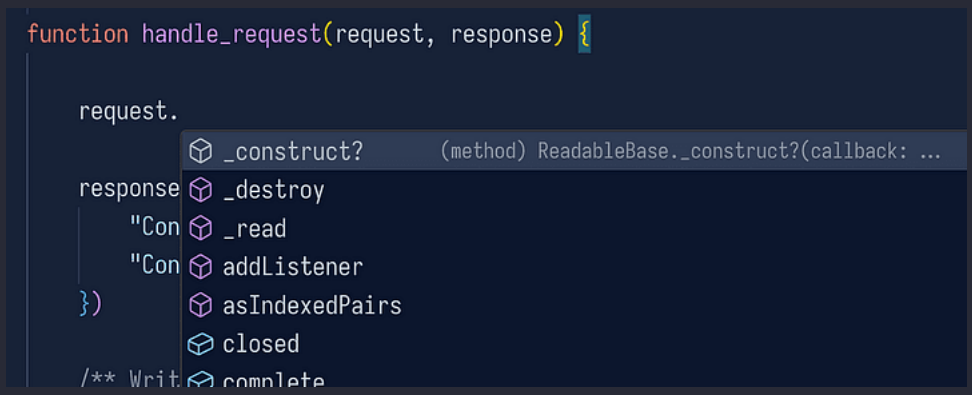
If we were using a callback approach, like this:

```
const server = http.createServer(function handle_request(request, response) {
   ...
});
```

you would've got the autocompletes when you tried to access any property or method of `request` or `response` object. This might look okay, but it's always better to have reusability in mind. To fix this, let's just use `jsdoc` styled comments:

```
// file: index.js

/**
 *
 * @param {http.IncomingMessage} request
 * @param {http.ServerResponse} response
 */
function handle_request(request, response) { .. }
```

Try to access a property of `request` by typing `request` followed by a period:



This works fine! But how do we actually find the type of the arguments in the first place?

It's quite simple.

1. In your code, locate the line where you're calling `http.createServer` and providing the `handle_request` function as a parameter.
2. On that line, hover your mouse cursor over the `createServer` function to highlight it.
3. With the `createServer` function highlighted, use the keyboard shortcut `Ctrl + Click` (or `Cmd + Click` on macOS) to jump to the function's declaration.

Doing this should take you to the definition of `createServer`. This is where things might look a bit complex due to the TypeScript type declarations. This is the type declaration for the method `http.createServer`:

```
function createServer<
  Request extends typeof IncomingMessage = typeof IncomingMessage,
  Response extends typeof ServerResponse = typeof ServerResponse
>(requestListener?: RequestListener<Request, Response>): Server<Request, Response>;
```

We only care about the following lines:

```
Request extends typeof IncomingMessage = typeof IncomingMessage
Response extends typeof ServerResponse = typeof ServerResponse
```

- `Request extends typeof IncomingMessage` means that `Request` is constrained to be a subtype of the `IncomingMessage` type. In simpler words, `Request` can only be a type that inherits from or is compatible with `IncomingMessage`.

So in short, the `IncomingMessage` can be used for the type of `request` parameter. Same for the response, we're using the type `ServerResponse`.

Let's talk about status codes now.

5.8.2 Status Codes

Status Codes are three-digit numeric codes generated by a web server in reply to a client's request made through an HTTP request. They provide crucial information about the status of the requested resource or the outcome of the client's request.

Status codes are grouped into five classes, each indicating a specific category of response. Let's look at a simple analogy on why do we need status codes.

When you ask a librarian for a specific book, they may respond in one of three ways: "Here's the book," "Sorry, we don't have that book," or "I'm not sure, let me go check." These responses provide information about the status of your request. Similarly, when your web browser sends a request to a web server, it's like asking for something in the vast digital library of the internet.

HTTP status codes work in a similar way. They're responses from the web server to your browser's request. When you click on a link, fill out a form, or make any kind of request on the internet, the web server replies with a status code to let your browser know what happened with your request.

These codes are grouped into classes to help you understand the general situation. Imagine you're playing a game, and you have four types of cards: success cards, redirection cards, client error cards, and server error cards. Each card type represents a different situation. The status code classes work like these card types, categorizing responses based on their nature.

- **1xx Informational Responses:** These are like hints that the web server is still working on your request. It's like the librarian saying, "I'm checking the back shelves for your book."
- **2xx Successful Responses:** These are success cards. They mean your request was understood and fulfilled. It's like the librarian handing you the book you asked for.
- **3xx Redirection Responses:** These are like directions the librarian gives you to find the book in a different section. Similarly, your browser might be directed to a different URL.

- **4xx Client Error Responses:** These are cards that say something is wrong with your request. Maybe you're asking for something that doesn't exist or you're not allowed to access.
- **5xx Server Error Responses:** These cards mean the library (web server) is having trouble. It's like the librarian apologizing for not being able to find the book due to a problem.

As server/API programmers we should make sure that we're sending valid and reasonable response codes back to the client. We'll learn about many status codes when we build our backend framework. But in-case you're curious you can check all the status codes here:

Code	Reason-Phrase	Defined in...
100	Continue	Section 6.2.1
101	Switching Protocols	Section 6.2.2
200	OK	Section 6.3.1
201	Created	Section 6.3.2
202	Accepted	Section 6.3.3
203	Non-Authoritative Information	Section 6.3.4
204	No Content	Section 6.3.5
205	Reset Content	Section 6.3.6
206	Partial Content	Section 4.1
300	Multiple Choices	Section 6.4.1
301	Moved Permanently	Section 6.4.2
302	Found	Section 6.4.3
303	See Other	Section 6.4.4
304	Not Modified	Section 4.1
305	Use Proxy	Section 6.4.5
307	Temporary Redirect	Section 6.4.7
400	Bad Request	Section 6.5.1
401	Unauthorized	Section 3.1
402	Payment Required	Section 6.5.2
403	Forbidden	Section 6.5.3
404	Not Found	Section 6.5.4
405	Method Not Allowed	Section 6.5.5
406	Not Acceptable	Section 6.5.6
407	Proxy Authentication Required	Section 3.2
408	Request Timeout	Section 6.5.7
409	Conflict	Section 6.5.8
410	Gone	Section 6.5.9
411	Length Required	Section 6.5.10

Code	Reason-Phrase	Defined in...
412	Precondition Failed	Section 4.2
413	Payload Too Large	Section 6.5.11
414	URI Too Long	Section 6.5.12
415	Unsupported Media Type	Section 6.5.13
416	Range Not Satisfiable	Section 4.4
417	Expectation Failed	Section 6.5.14
426	Upgrade Required	Section 6.5.15
500	Internal Server Error	Section 6.6.1
501	Not Implemented	Section 6.6.2
502	Bad Gateway	Section 6.6.3
503	Service Unavailable	Section 6.6.4
504	Gateway Timeout	Section 6.6.5
505	HTTP Version Not Supported	Section 6.6.6

Chapter 6

`Velocy` - Our backend framework

We have now reached the ultimate goal of this book - creating Velocy, a super-fast backend framework/library. We will also be creating an in-memory data store, like Redis alongside our backend framework to avoid using `npm install`.

Throughout the upcoming chapters, we will discuss every single line of code, design decision, and performance consideration that goes into building the framework. Since we are creating it alongside the book, the framework does not exist yet. We will build it together as I write the content to provide the best possible explanation.

> We will also implement a couple of data-structures, one example is the Trie for efficient route matching. Don't worry if you're not aware of what a Trie is, we'll write it from scratch.

Velocy will be designed with performance, efficiency, and scalability in mind. If you've ever been curious about how backend frameworks like Fastify or Express work under the hood, or if you've been on a hunt for a framework that truly fits your needs but couldn't find one — no worries, we're going to create it from scratch. Our goal is to make it production ready, and start using it for our upcoming projects.

Even if you're here just for learning Node.js, this experience will open up a world of understanding in web development, programming patterns, data management, and network communications that you may never have encountered otherwise.

While we were building our logging library, we looked at the **Builder pattern**. For the router of our framework, we will incorporate another popular software design pattern: the **dependency injection pattern**.

6.1 Why Velocy?

With many backend frameworks available, you might wonder why do we need another one. Velocy aims to focus on three core tenets that we believe are sometimes compromised in existing solutions:

1. **Speed:** Velocy is built from the ground up to be fast—not just in terms of handling requests but also in terms of development time.
2. **Efficiency:** Conventional frameworks often come loaded with features that not every project needs, leading to bloated applications. Velocy aims to be modular, letting you plug in only what you need.
3. **Scalability:** The architecture of Velocy is designed to easily adapt from a small, single-node application to a large, distributed system without requiring a complete overhaul.

6.2 What is a backend framework/library anyway?

A backend framework or library is a set of tools that helps developers create the server-side parts of web applications more easily and quickly. It's like a toolbox built on top of a programming language, and it simplifies the development process by hiding complex technical details and providing standard tools and methods to use.

For example, you could write performant server applications with Node.js, but that process is time consuming. Instead, you use a library which includes all the necessary features that you require for your server application.

However, in this book, we are going to take the hard route - build our own backend library.

> Note: I am going to use the terms library/framework interchangeably. In practice, they are somewhat different.

6.3 Core features of our backend framework

6.3.1 Routing and URL Handling:

Routing is a basic part of web apps. It tells how requests are linked to specific parts of the application. Routes are essential for structuring and organizing the endpoints of our API, making it easier for clients. Here are some examples of routes:

- `GET /api/users`: Retrieves a list of users.
- `GET /api/users/:id`: Retrieves a specific user by their ID.

- `POST /api/users`: Creates a new user.
- `PUT /api/users/:id`: Updates an existing user.
- `DELETE /api/users/:id`: Deletes a user.

> We'll take a look at what `:id` means, in a bit.

In a backend library, a strong routing system makes it simpler to create and manage routes. This helps developers who use our library, create well-organized applications. Here's why getting "routing and URL handling" done right is very important:

- **Endpoint Mapping:** We should make it easy for developers to create endpoints and associate them with the right actions or handlers. Developers should be able to specify which function or method will run when a specific URL is accessed.

  This is important in RESTful APIs where different HTTP methods (GET, POST, PUT, DELETE, etc.) must be linked to specific actions.

- **Parameter Extraction**: URLs sometimes have dynamic parameters, like IDs or slugs. A good routing system lets developers define placeholders in the URL and extract these parameters to use in the associated handler. This feature is important for making dynamic and data-driven applications.

  For example, let's take a look at an URL endpoint:

  `GET /api/games/:type`

  Developers who use our library should be able to configure the URLs like above, and we as a library should provide them with the ability to extract useful info for them, whenever someone makes a request like this:

  `https://ourapi.com/api/games/multiplayer?limit=10&order=asc`

  This should extract the `:type` "path" parameter, that is `multiplayer` and `limit`, `order` "query" parameters, which are `10` and `asc` respectively.

  > There are also other type of parameters, some of them are headers, body, cookies. We'll learn in-depth about those in the next chapter.

- **Route Hierarchies**: Modern applications often have complex route hierarchies. We should allow them to build specific part of the API separately, and then merge them altogether.

  For example: Usually the API endpoints are prefixed with `/api/version`, and typing them out in every

single route handler is quite cumbersome. What if our library offered a functionality to nest certain routes under a a specific pattern.

```
let v1_router = velocy.base_route("/api/v1");

let users_router = velocy.base_route("/users");

let add_user = velocy.get("/", add_user_callback);
let delete_user = velocy.delete("/:user_id", delete_user_callback);

// Nest `users_router` inside `v1_router`, and `add_user`, `delete_user` inside
↪ `users_router`
v1_router.nest(users_router.nest(add_user, delete_user));
```

This way, whatever requests that hit the endpoint `GET /api/v1/users/` will be forwarded to the `add_user_callback` function, and the requests hitting `DELETE /api/v1/users/some_id` will be forwarded to `delete_user_callback` function.

Wouldn't this be so cool?

- **Handling HTTP Verbs:** HTTP verbs, like `GET`, `POST`, `PATCH`, `DELETE`, `PUT` etc. play a crucial role in specifying the intended action for a request. Our library's routing system should allow developers to associate different HTTP verbs with appropriate route handlers. This ensures that the application responds correctly to different types of requests. We'll talk more about HTTP verbs in the next chapter.

- **Regex Support:** With regular expressions (regex), developers can create a flexible and powerful path matching mechanism that dynamically routes incoming requests to the appropriate handlers based on the URL structure.

  However, certain regex patterns may need to go through the input string an exponential number of times, taking O(2^n) time. This won't be an issue with small URLs, but an attacker might try to exploit this behavior by providing specially crafted input strings that trigger excessive backtracking, leading to a significant slowdown or even crashing of the application. This is known as aReDoS (Regex Denial of Service) attack.

- **Request and Response Handling:** Our routing system should provide an abstraction for handling incoming requests and generating appropriate responses. This could involve parsing request data, handling headers, and sending back structured responses.

6.3.2 Middlewares

Middleware is an important concept. It lets developers add their own code to the process of handling requests and responses. Middleware functions are like a middleman between the incoming request and the final response. They let developers do different things before and after the main application code runs.

For example,

```
let fetch_tweets = velocy.get("/tweets", rate_limiter, auth_middleware, fetch_tweets_handler);
```

Before executing the main function `fetch_tweets_handler` the request will go through a series of middlewares. In the case above, the middlewares are `rate_limiter` and `auth_middleware`. These middlewares can be reused. The request that hits the `/tweet` endpoint, first goes through the `rate_limiter` middleware function, it can either approve the request, or reject.

If the request is rejected, it does not go to the next middleware, or the main function.

6.3.3 Building our own database

We will also create a basic in-memory key-value database. This mini in-memory database will provide users with a lightweight and efficient solution for storing and retrieving data within their applications.

While it won't have the full range of capabilities found in dedicated databases, it will serve as a valuable tool for scenarios where a simple and fast data storage option is needed, without relying on other third party tools.

Our mini in-memory database with index support has the following key features:

6.3.3.1 Data Storage and Retrieval:

- Stores structured data in tables.
- Quickly retrieves data based on primary key or indexed fields.

6.3.3.2 Indexing:

- Supports indexing of key fields to speed up data retrieval.
- Has basic indexing mechanisms to optimize query performance.

> We won't focus on this target in the initial chapters, but we'll cover it as we approach the end of the book.

6.3.3.3 CRUD Operations:

- Performs Create, Read, Update, and Delete operations for managing data.
- Has a simplified interface for these operations.

6.3.3.4 Querying:

- Allows basic filtering and sorting operations to retrieve data based on specified criteria.
- Has support for simple filtering and sorting operations.

6.3.4 Caching

Caching means saving often-used data in a memory. When the data is needed again, the app can get it from the cache instead of doing the calculations or getting it from the original source. This makes things much faster and smoother for the user.

Caching can help lessen the work on the database server. Instead of asking the database for the same data many times, the application can get it from the cache. This not only makes things faster but also reduces the load on the database.

This also helps significantly when your web server is under huge load, caching improves your server's ability to handle many requests if the same piece of data is requested over and over again. However, caching also results in stale data. We'll address this in the later chapters of this book.

6.3.5 Rate limiting

API rate limiting is a way to control how often clients, like apps or users, can ask an API for things. This is important to stop people from using too much of the API and to keep the API and server working well.

6.3.6 Some other features that we will be implementing

- Shared state
- File uploads
- Static file serving
- Multi-part data
- Websockets
- Logging (using `logtar`)
- Monitoring

We will begin building our backend library/framework in the upcoming chapters. However, before doing so, we need to have a strong understanding of HTTP. Let's tackle that first in the next chapter.

6.4 A basic `Router` implementation

> A small note: we will be using the camelCase naming convention for our web framework. I plan to use this framework for my personal projects and encourage others developers to use it as well. For this reason, I need to ensure that my naming conventions align perfectly with the naming conventions used in Node.js's standard library.

We'll start building the fundamental building block of any web server framework - a `Router`. A router is an utility that determines how an application responds to different HTTP requests for particular URLs or in simple terms, it determines which HTTP requests should be handled by which part of the application.

Let's look at a normal `node:http` server, and understand why it is really cumbersome to manage different HTTP methods. Create a new project/directory, and create a new file: `index.js`

```
// file: index.js
const http = require("node:http");

const PORT = 5255;

const server = http.createServer((req, res) => {
  res.end("Hello World");
});

server.listen(PORT, () => {
  console.log(`Server is listening at :${PORT}`);
});
```

> We talked about what every line of server does in the chapter `5.0 - HTTP Deep Dive`, but to recap: the method `res.end()` tells the server that everything in the response, including headers and body, has been sent and the server should treat the message as finished.

If you try to send a `GET`, `POST`, `PUT` request to this server using `cURL`:

```
curl --request POST http://localhost:5255
# Hello World

curl -X PUT http://localhost:5255
# Hello World
```

How do we identify and differentiate between different HTTP methods? Fortunately, we have `req.method`:

```
// file: index.js

const http = require("node:http");

const PORT = 5255;

const server = http.createServer((req, res) => {
  res.end(`Hello from -> ${req.method} ${req.url}`); // Changed line
});

server.listen(PORT, () => {
  console.log(`Server is listening at :${PORT}`);
});
```

If you send a request again, using `cURL`:

```
curl -X PUT http://localhost:5255
# Hello from -> PUT /

curl -X PUT http://localhost:5255/hi/there/test
# Hello from -> PUT /hi/there/test
```

That should have already made you think, to write actual applications using the bare `node:http` module, there would be lot of `if` and `else` statements in our code base. You are correct. That is why we need a `Router` class that abstracts away all the functionality of handling different HTTP methods and URLs and provides a clean interface to the developer.

Before we implement our `Router` class, let's try to write a useful web server with what we have with us.

6.4.1 A Toy Router

```
// file: index.js

const http = require("node:http");

const PORT = 5255;

const server = http.createServer((request, response) => {
  const { headers, data, statusCode } = handleRequest(request);
  response.writeHead(statusCode, headers);
  response.end(data);
});

function handleRequest(request) {
  const { method, url } = request;

  let data = "";
  let statusCode = 200;
  let headers = {
    "Content-Type": "text/plain",
  };

  if (method === "GET" && url === "/") {
    data = "Hello World!";
    headers["My-Header"] = "Hello World!";
  } else if (method === "POST" && url === "/echo") {
    statusCode = 201;
    data = "Yellow World!";
    headers["My-Header"] = "Yellow World!";
  } else {
    statusCode = 404;
    data = "Not Found";
    headers["My-Header"] = "Not Found";
  }

  return { headers, data, statusCode };
}
```

```
server.listen(PORT, () => {
  console.log(`Server is listening at :${PORT}`);
});
```

Let's go through this line by line.

```
const { method, url } = request;
```

We passed the entire `request` object as an argument to `handleRequest`, we only care about two fields: `method` and `url`. So, instead of accessing the properties like `request.method` and `request.url`, we're using destructuring to get the values of `method` and `url` from the `request` object.

```
let headers = {
  "Content-Type": "text/plain",
};
```

Since Node.js's `ServerResponse.end` method does not implicitly sets the `Content-Type` header, we're manually setting it - as we know we're only returning plain text back to the client. It is a good practice to include all necessary headers following the HTTP guidelines.

```
if (method === "GET" && url === "/") {
    data = "Hello World!";
    headers['My-Header'] = "Hello World!";
} else if (method === "POST" && url === "/echo") {
    ...
} else {
    ...
}
```

We are using the `data` variable to keep track of the content that will be sent back to the client. If the HTTP method is `GET` and the URL is `/` (which means the request URL on `cURL` is either `http://localhost:5255/` or `http://localhost:5255`), we will send back `Hello World!` and set our custom header `My-Header`.

The same applies to other routes as well, where we change the data being sent, the status code, and the `My-Header` header according to the `method` and the `url`. If it doesn't matches our needs, we're simply sending a `404` status code, and `Not Found` back to the client.

> Sending the right headers is really important when sending data back to the client. Headers control how the client and server communicate and protect the data. If the headers are wrong or missing, bad things can happen like security problems or the application not working. So, getting the headers right is more important than the actual data being sent.
> In fact you can just ignore the data, and just send an empty response with 404 status code. Every developer knows what does a `404` code means.

```
return { headers, data, statusCode };
```

We're returning the information back to the caller function - which in case is this part:

```
const server = http.createServer((request, response) => {
  const { headers, data, statusCode } = handleRequest(request);
  response.writeHead(statusCode, headers);
  response.end(data);
});
```

Once the `handleRequest` function finishes executing, we get the appropriate headers, the data and the status code that needs to be sent back to the client in order to let them know "Okay we're done processing your request. Here is the result".

```
response.writeHead(statusCode, headers);
```

`writeHead` is a method on the `ServerResponse` class, which gives us the flexibility to set response headers, as well as status code. The headers should be an object with key being the header name, and value being the header value.

But what if we do not set these headers? Turns out, if you do not set headers before `res.end()`, Node.js implicitly sets most of the headers.

> The second argument can also be an array with multiple entries, but we're not going to use that.

```
res.end(data);
```

Tells the client - "I'm officially done now. Take this `data`".

Now, let's test our simple implementation of this "router" function.

```
$ curl http://localhost:5255 -v

*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> GET / HTTP/1.1
> Host: localhost:5255
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< My-Header: Hello World!
< Date: Fri, 01 Sep 2023 14:49:06 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Transfer-Encoding: chunked
<
* Connection #0 to host localhost left intact
Hello World!
```

We have our header `My-Header`, status code of `200` and the response body `Hello World!`. Let's test the `POST` endpoint too:

```
curl -X POST  http://localhost:5255/echo -v
*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> POST /echo HTTP/1.1
> Host: localhost:5255
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 201 Created
< My-Header: Yello World!
< Date: Fri, 01 Sep 2023 14:52:33 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Transfer-Encoding: chunked
```

```
<
* Connection #0 to host localhost left intact
Yellow World!%
```

Perfect! Everything looks fine. Now it's time to test an URL or method which isn't covered by our `handleRequest` function.

```
*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> POST /nope HTTP/1.1
> Host: localhost:5255
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< My-Header: Not Found
< Date: Fri, 01 Sep 2023 14:53:47 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Transfer-Encoding: chunked
<
* Connection #0 to host localhost left intact
Not Found%
```

Yes, the status code is `404` as expected. `My-Header` has the correct value, and the response body is `Not Found`. The implementation looks fine.

6.4.2 `Transfer-Encoding: chunked`

You may have noticed a new header `Transfer-Encoding` with the value of `chunked` in the output above. What is it?

In chunked transfer encoding, the data is split into separate pieces called "chunks". These chunks are sent and received without depending on each other. Both the sender and receiver only need to focus on the chunk they are currently working on, without worrying about the rest of the data stream.

This is because, we did not specify the `Content-Length` header, and that means we (as a server) **do not** know the size of the response payload, in bytes. However, one should only use `Transfer-Encoding: chunked` when you're sending a large payload and don't know about the exact size of payload.

There is a slight (but noticeable) performance overhead with chunked encoding when you're sending small payload to the client as chunks. In our case, we're just sending a small string `"Hello, world!"` back to the client.

6.4.3 Chunks, oh no!

The message "Hello, world!" is only 13 bytes long, so its length in hexadecimal is `D`. In fact we're only sending a simple looking string back to the client, but turns out the data being sent is almost 2x the size of the string `Hello, world!`.

Here is how it would look in chunked encoding:

```
D\r\n
Hello, world!\r\n
```

Each chunk in the response is preceded by its length in hexadecimal format, followed by `\r\n` (CRLF), and then another `\r\n` follows the data. This means that for each chunk, you have the extra bytes to represent the length and two `CRLF` sequences.

To indicate the end of all chunks, a zero-length chunk is sent:

```
0\r\n
\r\n
```

So, the full HTTP message body in chunked encoding would be:

```
D\r\n
Hello, world!\r\n
0\r\n
\r\n
```

Now, let's calculate the size of response with `Transfer-Encoding: chunked`:

- The size of the chunk in hexadecimal (`D`) = 1 byte
- The first CRLF (`\r\n`) after the size = 2 bytes
- The data ("Hello, world!") = 13 bytes
- The second CRLF (`\r\n`) after the data = 2 bytes
- The zero-length chunk to indicate the end (`0`) = 1 byte
- The CRLF (`\r\n`) after the zero-length chunk = 2 bytes
- The final CRLF (`\r\n`) to indicate the end of all chunks = 2 bytes

In total, it becomes `23 bytes`! You're actually sending back 2x the size of the payload, which is a lot of extra overhead. The bigger your response payload, the lesser the overhead.

But most of the text responses don't need chunked encoding. It's helpful for things which are large, and you may not know the size of the payload you're sending, like a file that lives on AWS S3, or a file that you're downloading from an external CDN. Chunked encoding would be a great candidate for that.

6.4.4 Specifying `Content-Length`

To get rid of `Transfer-Encoding: chunked`, we just have to specify the `Content-Length` header, with the value of the payload in bytes.

```
function handleRequest(request) {
  const { method, url } = request;

  let data = "";
  let statusCode = 200;
  let headers = {};

  if (method === "GET" && url === "/") {
    data = "Hello World!";
    headers["My-Header"] = "Hello World!";
  } else if (method === "POST" && url === "/echo") {
    statusCode = 201;
    data = "Yellow World!";
    headers["My-Header"] = "Yello World!";
  } else {
    statusCode = 404;
    data = "Not Found";
    headers["My-Header"] = "Not Found";
  }

  // set the content-length header to the value of length of `data` in bytes.
  headers["Content-Length"] = Buffer.byteLength(data);
  return { headers, data, statusCode };
}
```

The `Buffer` class provides a very useful helper method: `Buffer.byteLength`. If you try to make a request using `cURL`, you'll see the `Content-Length` header, and the `Transfer-Encoding: chunked` isn't there. Perfect.

```
curl http://localhost:5255 -v
```

```
# Omitted request output
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< My-Header: Hello World!
< Content-Length: 12
< Date: Fri, 01 Sep 2023 17:25:08 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
<
* Connection #0 to host localhost left intact
Hello World!
```

6.4.5 Code reusability

We are still one step away from implementing our `Router` class. Before doing so, I want to talk a little bit about code-maintainability. The current way is fine, but this will go out of hand when you're going to handle a lot of routes. We should always design our programs with scalability and reuse in mind.

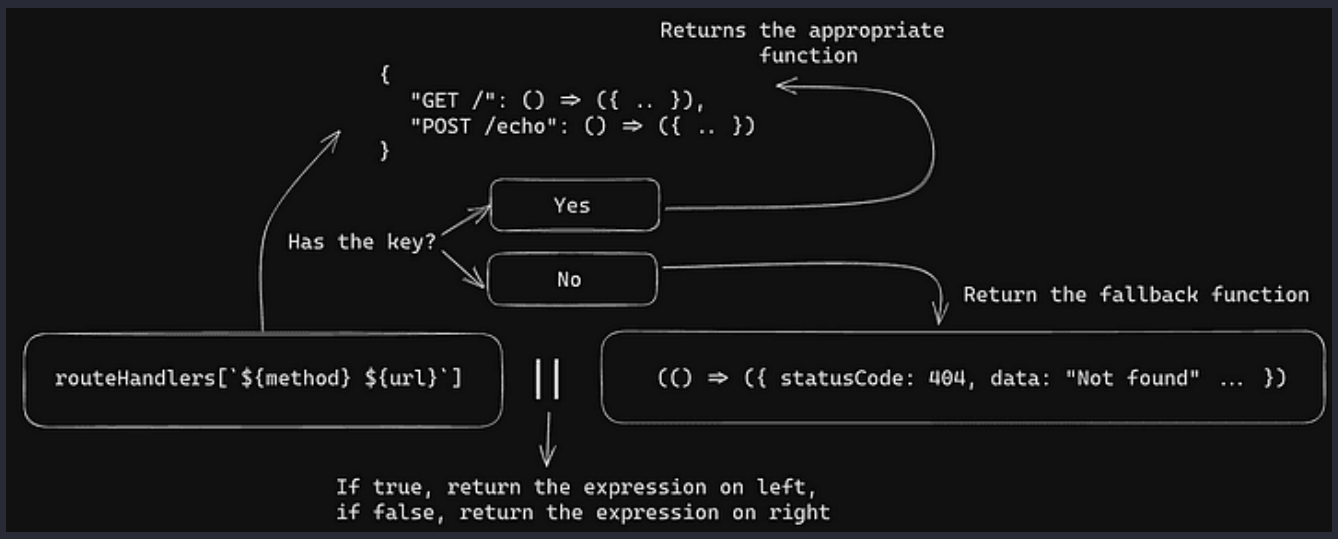
Let's include a global object `routeHandlers` and then update the code accordingly, in a way that we can easily add new routes without having to change the `handleRequest` function.

```javascript
// file: index.js

const http = require("node:http");

const PORT = 5255;

const server = http.createServer((request, response) => {
  const { headers, data, statusCode } = handleRequest(request);
  response.writeHead(statusCode, headers);
  response.end(data);
});

// The header that needs to be sent on every response.
const baseHeader = {
  "Content-Type": "text/plain",
};
```

```
const routeHandlers = {
  "GET /": () => ({ statusCode: 200, data: "Hello World!", headers: { "My-Header": "Hello World!"
  ↪ } }),
  "POST /echo": () => ({ statusCode: 201, data: "Yellow World!", headers: { "My-Header": "Yellow
  ↪ World!" } }),
};

const handleRequest = ({ method, url }) => {
  const handler =
    routeHandlers[`${method} ${url}`] ||
    (() => ({ statusCode: 404, data: "Not Found", headers: { "My-Header": "Not Found" } }));

  const { statusCode, data } = handler();
  const headers = { ...baseHeader, "Content-Length": Buffer.byteLength(data) };

  return { headers, statusCode, data };
};

server.listen(PORT, () => {
  console.log(`Server is listening at :${PORT}`);
});
```

Let's look at some of the weird looking parts:

```
// `handler` holds a function, based on the incoming HTTP `method` and the `URL`
const handler =
  routeHandlers[`${method} ${url}`] ||
  (() => ({ statusCode: 404, data: "Not Found", headers: { "My-Header": "Not Found" } }));
```



We're looking for a key in the `routeHandlers` object, which matches the incoming method and URL. If the key is available, we're going to use the value of that key from `routeHandlers`, which is in-fact a function. If the key is not found, means that we do not have a handler associated for a particular `method` and `URL` combination, we simply assign the `handler` variable with the value of a function that returns:

```
{ statusCode: 404, data: "Not Found", headers: { "My-Header": "Not Found" } }
```

Although we will not use the code written above in our `Router`, it is important to understand the significance of better design and planning ahead. As projects grow, it is easy to fall into the trap of the sunk cost fallacy. This fallacy leads us to stick with our initial approach simply because of the time and effort already invested.

Before you dive into coding, take some time to think about how your design can grow and adapt easily down the road. Trust me, it'll save you headaches later. Start off by making a basic prototype or a couple of features. Then, as things start to grow, that's the clue to dig into the design and structure of your program.

6.4.5.1 Separation of Concerns

Our route handlers are defined as properties of an object `routeHandlers`. If we need to add support for more HTTP methods, we can do that without changing any other parts of our code:

```
const routeHandlers = {
    "GET /": () => ({ statusCode: 200, data: "Hello World!", headers: { "My-Header": "Hello
    ↪  World!" } }),
    "POST /echo": () => ({ statusCode: 201, data: "Yellow World!", headers: { "My-Header":
    ↪  "Yellow World!" } }),
    "POST /accounts": () => ({ statusCode: 201, data: "Creating Account!", headers: { ... } }),
    // Add more HTTP methods
};
```

A general rule of thumb is - Your functions should only do what they are supposed to, or what their name says. It is called the Single-responsibility principle. Suppose you have a function with a signature of `function add(x, y)`, it should only add two numbers, nothing else. An example of bad code, that you shouldn't do:

```
function add(x, y) {
  // Not only adding x and y, but also writing to console, which is not expected.
  console.log(`Adding ${x} and ${y}`);

  // Performing a file operation, which is definitely not expected from an 'add' function.
  const fs = require("fs");
  fs.writeFileSync("log.txt", `Adding ${x} and ${y}\n`, { flag: "a+" });

  // Sending an HTTP request, which is out of scope for an 'add' function.
  const http = require("http");
  const data = JSON.stringify({ result: x + y });
  const options = {
    hostname: "github.com",
    port: 80,
    path: "/api/add",
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Content-Length": data.length,
    },
  };

  const req = http.request(options);
  req.write(data);
  req.end();

  // Finally adding, which is what the function is supposed to do.
  return x + y;
}
```

However, a good example would look something like:

```
// This function does only what it says: adds two numbers.
function add(x, y) {
```

```
  return x + y;
}

// A separate function to log the addition operation.
function logAddition(x, y) {
  console.log(`Adding ${x} and ${y}`);
}

// A separate function to write the addition operation to a file.
function writeFileLog(x, y) {
  const fs = require("fs");
  fs.writeFileSync("log.txt", `Adding ${x} and ${y}\n`, { flag: "a+" });
}

// A separate function to send the addition result to an API.
function sendAdditionToAPI(x, y) {
  const http = require("http");
  const data = JSON.stringify({ result: add(x, y) });
  const options = {
    hostname: "example.com",
    port: 80,
    path: "/api/add",
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Content-Length": data.length,
    },
  };
  const req = http.request(options);
  req.write(data);
  req.end();
}

// You can now compose these functions according to your needs.

logAddition(3, 5); // Logs "Adding 3 and 5"
writeFileLog(3, 5); // Writes "Adding 3 and 5" to 'log.txt'
```

```
sendAdditionToAPI(3, 5); // Sends a POST request with the result
console.log(add(3, 5)); // Logs "8", the result of the addition
```

While modularity and the Single Responsibility Principle are generally good practices, overdoing it can lead to its own set of problems. Extracting every tiny piece of functionality into its own function or module can make the codebase fragmented and difficult to follow. This is sometimes referred to as "over-engineering."

Here are some general considerations, that I tend to follow:

- **Too Many Tiny Functions**: If you find that you have a lot of functions that are used only once and consist of just one or two lines, that might be an overkill.
- **High Abstraction Overheads**: Excessive modularity might introduce unnecessary layers of abstraction, making the code less straightforward and harder to debug.
- **Reduced Performance**: While modern compilers and interpreters are good at making things faster, if you have a lot of little functions that you call many times, it can add extra overhead if the compiler decides to not inline them.

We'll take everything we've learned so far and apply it to our `Router` class in the next chapter.

6.5 The `Router` class

To begin with, let's create a basic version of the `Router` class for better understanding. We can gradually add more functionality to it as we move forward. The first implementation of the `Router` class will allow us to add routes and handle basic HTTP requests.

```
// file: index.js

class Router {
  constructor() {
    // Stores our routes
    this.routes = {};
  }
}
```

The `routes` member variable in our `Router` class serves as an internal data structure for managing the mapping between URL paths and corresponding handlers. This approach provides the benefits of encapsulation and efficient route lookups. However, this approach is not suitable for dynamic routes such as `/api/:version` or `/api/account/:account_id/transactions`, where there are infinite possible URLs that can match. This issue will be addressed later. For now, we will stick to static routes such as `/api/users`

or `/api/account/signup`.

Let's introduce a new helper method called `addRoute` in the `Router` class. This method will help us bind callback functions to execute whenever a request for a particular endpoint is made.

```
// file: index.js

class Router {
  constructor() {
    this.routes = {};
  }

  addRoute(method, path, handler) {
    this.routes[`${method} ${path}`] = handler;
  }
}
```

The first argument, `method` is the HTTP method - `GET`, `POST`, `PUT` etc. The second argument is the URL or the path of the request, and the third argument is the callback function that will be called for that particular `method` and `path` combination. Let's see an example on how will be it called.

```
// file: index.js

class Router { ... }

const router = new Router();

router.addRoute('GET', '/', () => console.log('Hello from GET /'));
```

Let us add a method inside our `Router` class which will print all the routes, for debugging purposes.

```
// file: index.js

class Router {
    constructor() { ... }

    addRoute(method, path, handler) { ... }

    printRoutes() {
        console.log(Object.entries(this.routes));
```

```
    }
}

const router = new Router();
router.addRoute('GET', '/', () => console.log('Hello from GET /'));
router.addRoute('POST', '/', () => console.log('Hello from POST /'));

router.printRoutes()
```

Now, let us give it a try:

```
$ node index.js
// Outputs
[
  [ 'GET /', [Function (anonymous)] ],
  [ 'POST /', [Function (anonymous)] ]
]
```

The `Object.entries` method converts an object, with key-value pairs, into a tuple array with only two elements. The first element is the key, and the second element is the corresponding value.

The `function (anonymous)` signature isn't helpful at all. Let's change it:

```
// file: index.js

...

const router = new Router();
router.addRoute('GET', '/', function handleGetBasePath() { console.log(...) });
router.addRoute('POST', '/',function handlePostBasePath() { console.log(...) });

router.printRoutes()
```

Outputs:

```
[
  [ 'GET /', [Function: handleGetBasePath] ],
  [ 'POST /', [Function: handlePostBasePath] ]
]
```

Much better. I generally tend to avoid anonymous functions as much as I can, and give them a proper name instead. The above can also be written as:

```
// file: index.js

function handleGetBasePath() { ... }
function handlePostBasePath() { ... }

router.addRoute("GET", "/", handleGetBasePath)
router.addRoute("POST", "/", handlePostBasePath)
```

Let us try to hook our router with a real HTTP server. We'll have to import the `node:http` module, as well as add an utility method inside the `Router` class that redirects the incoming request to their appropriate handlers.

6.5.1 Using `Router` with an HTTP server

```
// file: index.js

class Router {
    ...
    handleRequest(request, response) {
        const { url, method } = request;
        this.routes[`${method} ${url}`](request, response);
    }
}
```

Since we only care about the `url` and `method` of an HTTP request, we're destructuring it out of the `request` object. There's a weird looking syntax, that might be foreign to you if you're new to javascript.

```
this.routes[`${method} ${url}`](request, response);
```

The above is a short way to write:

```
let functionToExecute = this.routes[`${method} ${url}`];
functionToExecute(request, response);
```

You need to be careful with this syntax though. If `path` and `method` combination isn't registered, it will return `undefined`. And the above syntax would resolve to `undefined()`, that in fact does not make any sense. Javascript will throw a beautiful error message -

```
undefined is not a function
```

To take care of it, let's add a simple check:

```
class Router {
    ...
    handleRequest(request, response) {
        const { url, method } = request;
        const handler = this.routes[`${method} ${url}`];

        if (!handler) {
            return console.log('404 Not found')
        }

        handler(request, response)
    }
}
```

Now let's forward every request to this method.

```
// file: index.js

const http = require('node:http')
const PORT = 5255;

class Router { ... }

const router = new Router();
router.addRoute('GET', '/', function handleGetBasePath() { console.log(...) });
router.addRoute('POST', '/', function handlePostBasePath() { console.log(...) });

let server = http.createServer(function serveRequest(request, response) {
    router.handleRequest(request, response)
})

server.listen(PORT)
```

We've imported the `node:http` module, to create an HTTP server, and used the `http.createServer` method, providing it a callback that takes 2 arguments - first one is the request, and the second one is the response

object.

We can still make our code a little better. Instead of passing a callback that has only one job, i.e. calls another method; we can directly pass the target method as an argument:

```
// file: index.js

class Router {...}

/** __ Add routes __ **/

let server = http.createServer(router.handleRequest);
server.listen(PORT)
```

Or even shorter, in case you do not wish to access any methods of the `http.Server` object returned by `http.createServer`.

```
http.createServer(router.handleRequest).listen(PORT);
```

Let us test it using `cURL`, after starting the server using `node index` on a different terminal:

```
$ curl http://localhost:5255 -v
*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> GET / HTTP/1.1
> Host: localhost:5255
> User-Agent: curl/7.87.0
> Accept: */*
>
* Empty reply from server
* Closing connection 0
curl: (52) Empty reply from server
```

An empty reply from server? Let's head over to the node program's console. Oops, a crash!

```
TypeError: Cannot read properties of undefined (reading 'GET /')
        at Server.handleRequest (/Users/ishtmeet/Code/velocy/index.js:16:36)
```

It's saying we tried to access the key `GET /` of an undefined value. It's pointing at this line:

```
const handler = this.routes[`${method} ${url}`];
```

It is saying that `this.routes` is undefined. Weird, isn't it? No. If you have written some code with javascript previously, you would've already figured out the issue. The culprit is this line:

```
http.createServer(router.handleRequest);
```

Let us try to print what the value of `this` is, in the `handleRequest` method:

```
class Router {
    ...
    handleRequest(request, response) {
        console.log(this.constructor.name);
        ...
    }
}
```

Send another `cURL` request:

```
# Prints
Server
```

How?

6.6 `this` is not good

Let's take a moment to understand a huge part of Javascript programming in general: the `this` keyword. We won't go deep into the nitty gritty of `this` but we'll understand enough to not fall into this weird semantic bug in our programs.

When we tried to log `this.constructor.name`, it printed `Server` which has nothing to do with our code. We don't have a class or function named `Server` in our code. It means that the `this` context inside `handleRequest` is an instance of Node's native HTTP Server class, not our `Router` class.

The reason is how `this` works in JavaScript when passing a method as a callback. When we originally had:

```
// we're passing the router.handleRequest method here as an argument
let server = http.createServer(router.handleRequest);

// The `handleRequest` method is defined as a normal function, not an arrow function
class Router {
```

```
    handleRequest(request, response) {
        this.routes; // Looks good but not good.
        ...
    }
}
```

The `this` value in the `handleRequest` method will not be the original object (`router` in this case), but will be determined by how it's called — which, in the case of `http.createServer` , won't be the `router` object. That's why `this.routes` is `undefined` .

It turns out that there is nothing wrong with the method definition inside the `Router` class. However, there is an issue with the way it is being invoked.

The method `handleRequest` is passed as a callback to `http.createServer()` . When this callback gets invoked by the Node.js HTTP Server, the context (`this`) inside `handleRequest` is bound to that Server instance, not to the `Router` instance.

We're passing a reference to the `handleRequest` method, but it loses its context (`this`), i.e. it gets dissociated from the `router` instance of the `Router` class. When the `handleRequest` method is invoked by the HTTP server, this is set to the HTTP Server object, not the `router` instance.

How do we fix this? There are two ways: an old way and a modern one. Let's see the old way first:

6.6.0.1 Using `.bind()`

```
let server = http.createServer(router.handleRequest.bind(router));
```

The `.bind` method returns a new function that is a "bound" version of the `handleRequest` method, such that the `this` context within that method is set to the `router` instance.

So, `.bind()` ensures that when `handleRequest` is called, the value of `this` inside of it will be our `router` object. Before `ES6` or `EcmaScript 2015` , this was the standard way of solving issues with the **this** keyword.

Let's take a look at a more convenient way, i.e use an `Arrow` function:

6.6.0.2 Using `Arrow` function

```
let server = http.createServer((req, res) => router.handleRequest(req, res));

// or if you prefer named functions
const handleRequest = (req, res) => router.handleRequest(req, res);
const server = http.createServer(handleRequest);
```

Unlike normal functions, arrow functions don't have their own `this`. Instead, they inherit the `this` value from the surrounding lexical context where they were defined. This lexical scoping for `this` is one of the most useful features of arrow functions.

I'll explain what I mean by the **lexical** context.

6.6.1 Lexical Context

Lexical context (or lexical scope) is the area where a certain variable is accessible or has meaning. When a variable is defined, it's confined to a particular scope, and it can't be accessed from outside of that scope.

```
Global Scope
|
|-- const global = "I'm global";
|
|-- function outerFunction() {
|     |
|     |-- const outer = "I'm in the outer function";
|     |
|     |-- function innerFunction() {
|     |     |
|     |     |-- const inner = "I'm in the inner function";
|     |     |
|     |     |-- // Can access inner, outer, and global
|     |}
|     |
|     |-- // Can access outer and global, but NOT inner
|}
|
|-- // Can access global, but NOT outer or inner
```

Let's look at another example using classes:

```
class Person {
  constructor() {
    this.name = "Ishtmeet";
  }
```

```
  regularFunction() {
    setTimeout(function () {
      // `this` here is not the Person instance, it's either the window object
      // in browsers or `global` in Node.js
      console.log(this.name); // Undefined or error
    }, 1000);
  }

  arrowFunction() {
    setTimeout(() => {
      // `this` here is lexically bound, it's the Person instance
      console.log(this.name); // Outputs: "Ishtmeet"
    }, 1000);
  }
}
```

One more example using classes:

```
class Player {
  constructor() {
    this.health = 52;
  }

  regularFunction() {
    // Will output 52 if called as an instance method
    console.log(this.myValue);
  }

  arrowFunction = () => {
    // Will also output 52 if called as an instance method
    console.log(this.myValue);
  };
}
```

For example:

```
const mainCharacter = new Player();
```

```
mainCharacter.regularFunction(); // Outputs 42
mainCharacter.arrowFunction(); // Outputs 42
```

But, let's see what happens if we extract these methods and call them independently of the class instance:

```
const extractedRegularFn = mainCharacter.regularFunction;
const extractedArrowFn = mainCharacter.arrowFunction;

extractedRegularFn(); // Outputs undefined or throws an error
extractedArrowFn(); // Outputs 42
```

In the code above, `extractedRegularFn()` outputs `undefined` or throws a TypeError depending on strict mode because it loses its original `this` context.

On the other hand, `extractedArrowFn()` still outputs `42`, because the arrow function doesn't have its own `this`; it uses the `this` from the lexical scope where it was defined (inside the `Player` constructor, because it is bound to `mainCharacter` as we specifically called it on `mainCharacter` using `mainCharacter.arrowFunction`).

6.6.2 Arrow functions are not free

Keep in mind, arrow functions come with a slight performance penalty. It is usually negligible for most applications, but can be heavy on memory if we're creating a lot of objects of a certain class. This won't be an issue with our `Router`, but it's worth knowing this.

When arrow functions are defined as class methods/properties, a new function object is created for each instance of the class, rather than each time the function is invoked.

For example let's look at the an example:

```
class Monster {
  regularMethod() { ...  }
  arrowMethod = () => { ... };
}

const boss = new Monster();
const creep = new Monster()
```

In this case, both `boss` and `creep` will have their own copy of `arrowMethod`, because it's defined as an instance property using arrow syntax. Each time a new `MyClass` object is created, new memory is allocated for `arrowMethod`.

On the other hand, `regularMethod` is defined on `Monster.prototype`, meaning that all instances of `MyClass` share the same `regularMethod` function object. This is generally more memory-efficient.

In a game, one can spawn thousands, if not millions of monsters. Or imagine another hypothetical example of a photo editing application, that stores every pixel on the screen as an object of a `Pixel` class. There are going to be millions of pixels on the screen, and every extra allocation for the function body may be slight overwhelming for the memory constraints.

6.6.3 Why should we care about memory?

We are focusing on building a high-performance backend framework, it is important to consider the impact of memory allocations. While the `Router` class may only have 10-15 instances, we may introduce our custom `Response` or `Request` class in the future. If we create every function as an arrow function, our framework will allocate unnecessary memory for applications receiving high load, such as those receiving thousands of requests per second. An easy way to picture arrow functions in mind is as follows:

```
class Response {
  constructor() {
    // If these arrow functions are created new for every instance of Response,
    // and the site is currently in a heavy load situation, receiving 5k requests per second
    // we'd be creating 5,000 * num_of_arrow_functions new function instances per second
    this.someMethod = () => {
      /*... */
    };
    this.anotherMethod = () => {
      /* ... */
    };
    // ... more arrow functions
  }
}
```

Creating separate function objects for each instance is usually not a concern for most applications. However, in cases where you have a very large number of instances or if instances are frequently created and destroyed, this could lead to increased memory usage and garbage collection activity.

It's also worth noting that if we're building a library or a base class that other developers will extend, using prototype methods (regular methods) allows for easier method overriding and usage of the `super` keyword.

> Note: Unless we're in a very performance-critical scenario or creating a vast number of instances, the difference is likely negligible. Most of the time, the decision between using arrow functions or regular methods in classes comes down to semantics and specific requirements around `this` binding.

6.6.4 Testing the updated code

Our code in the `index.js` file should look something like this:

```
// file: index.js

const http = require("node:http");

const PORT = 5255;

class Router {
  constructor() {
    this.routes = {};
  }

  addRoute(method, path, handler) {
    this.routes[`${method} ${path}`] = handler;
  }

  handleRequest(request, response) {
    const { url, method } = request;
    const handler = this.routes[`${method} ${url}`];

    if (!handler) {
      return console.log("404 Not found");
    }

    handler(request, response);
  }

  printRoutes() {
    console.log(Object.entries(this.routes));
  }
```

```
}

const router = new Router();
router.addRoute("GET", "/", function handleGetBasePath() {
  console.log("Hello from GET /");
});

router.addRoute("POST", "/", function handlePostBasePath() {
  console.log("Hello from POST /");
});

// Note: We're using an arrow function instead of a regular function now
let server = http.createServer((req, res) => router.handleRequest(req, res));
server.listen(PORT);
```

Let us try to execute this code, and send a request through `cURL`:

We see the output `Hello from GET /` on the server console. But, there's still something wrong on the client (`cURL`):

```
$ curl http://localhost:5255/ -v
*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> GET / HTTP/1.1
> Host: localhost:5255
> User-Agent: curl/7.87.0
> Accept: */*
>
```

The request headers are shown, means the request was made successfully, although server did not send any response back. Why is it so?

We're observing this behavior because the server has not indicated to the client (in this case, `cURL`) that the request has been fully processed and the response has been completely sent. Well, how do we indicate that?

We do that using `.end()` method on the response object. But how can we get access to that inside our callback functions `handlePostBasePath()` and `handleGetBasePath()`? Turns out, they're already supplied to these functions when we did this:

```
// pass the `request` as the first argument, and `response` as the second.
let server = http.createServer((req, res) => router.handleRequest(req, res));
```

The `http.createServer` method requires a callback function, and provides request object as the first argument, and the response object as the second.

On updating the code:

```
// file: index.js

...

router.addRoute("GET", "/", function handleGetBasePath(req, res) {
    console.log("Hello from GET /");
    res.end();
});

router.addRoute("POST", "/", function handlePostBasePath(req, res) {
    console.log("Hello from POST /");
    res.end()
});

...
```

Now if you try to make a request to any endpoint, the server will respond back with appropriate response body:

```
$ curl http://localhost:5255/ -v
*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> GET / HTTP/1.1
> Host: localhost:5255
> User-Agent: curl/7.87.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Date: Thu, 07 Sep 2023 13:04:39 GMT
< Connection: keep-alive
```

```
< Keep-Alive: timeout=5
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

We also set up a 404 handler, in case a route is not configured. We're also going to add a `response.end()` to indicate the client that the request has been processed.

```
class Router {
    ...
    handleRequest(request, response) {
        ...
        if (!handler) {
            console.log("404 Not found");
            response.writeHead(404, { 'Content-Type': 'text/plain' })
            return response.end('Not found');
        }
        ...
    }
}
```

Let's check whether it returns `404`, if the route is not registered?

```
$ curl http://localhost:5255/not/found -v
*   Trying 127.0.0.1:5255...
* Connected to localhost (127.0.0.1) port 5255 (#0)
> GET /not/found HTTP/1.1
... request body trimmed ...
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< Content-Type: text/plain
... response body trimmed ...
* Connection #0 to host localhost left intact
Not found
```

Great! In the next section, we will explore how to make our router API even more elegant by eliminating the need to specify the name of the HTTP method every time we define a new endpoint handler. So, instead of writing:

```
router.addRoute("POST", "/", function handlePostBasePath(req, res) {
  console.log("Hello from POST /");
  res.end();
});
```

We could do something like:

```
router.get(/", function handlePostBasePath(req, res) {
    console.log("Hello from POST /");
    res.end();
});
```

This way, we'll provide a clean and clear interface for our clients.

6.7 Improving the `Router` API

The utility method on the `Router` class - `addRoute` is a bit too verbose. You need to specify the HTTP method as a string. It would get tedious when there are suppose hundreds of API routes in an application. Also, devs might not know whether the HTTP methods should be sent in lower-case or upper-case without looking at the source.

Let's abstract that functionality away from the developer, making sure the developers only need to worry about the important pieces.

Current way to add routes:

```
// file: index.js

class Router {
    constructor() {
        this.routes = {};
    }

    addRoute(method, path, handler) {
        this.routes[`${method} ${path}`] = handler;
    }
    ...
}
```

Let's add two new methods named `get` and `post`, and add some type checks in the `addRoute` method:

```
// file: index.js

const HTTP_METHODS = {
    GET: "GET",
    POST: "POST",
    PUT: "PUT",
    DELETE: "DELETE",
    PATCH: "PATCH",
    HEAD: "HEAD",
    OPTIONS: "OPTIONS",
    CONNECT: "CONNECT",
    TRACE: "TRACE",
};

class Router {
    constructor() {
        this.routes = {}
    }

    #addRoute(method, path, handler) {
        if (typeof path !== "string" || typeof handler !== "function") {
            throw new Error("Invalid argument types: path must be a string and handler must be a
            ↪ function");
        }

        this.routes.set(`${method} ${path}`, handler);
    }

    get(path, handler) {
        this.#addRoute(HTTP_METHODS.GET, path, handler);
    }
    post(path, handler) {
        this.#addRoute(HTTP_METHODS.POST, path, handler);
    }
    put(path, handler) {
        this.#addRoute(HTTP_METHODS.PUT, path, handler);
    }
```

```
    delete(path, handler) {
        this.#addRoute(HTTP_METHODS.DELETE, path, handler);
    }

    patch(path, handler) {
        this.#addRoute(HTTP_METHODS.PATCH, path, handler);
    }

    head(path, handler) {
        this.#addRoute(HTTP_METHODS.HEAD, path, handler);
    }

    options(path, handler) {
        this.#addRoute(HTTP_METHODS.OPTIONS, path, handler);
    }

    connect(path, handler) {
        this.#addRoute(HTTP_METHODS.CONNECT, path, handler);
    }

    trace(path, handler) {
        this.#addRoute(HTTP_METHODS.TRACE, path, handler);
    }

    ...
}
```

Let's go through the new additions in our code:

```
get(path, handler) {
    this.#addRoute(HTTP_METHODS.GET, path, handler);
}

post(path, handler) {
    this.#addRoute(HTTP_METHODS.POST, path, handler);
}
/** rest HTTP method handlers **/
```

We've created new utility methods on the `Router` class. Each one of these methods call the `addRoute` method by passing in required parameters. You'd notice that we've also made the `addRoute` method private. Since we wish to use it internally in our library and not expose it, it's a good practice to hide it from any external use.

```
const HTTP_METHODS = { ... }
```

We've created an object of all the HTTP methods, so that we can use their names with the `HTTP_METHODS` namespace, instead of directly passing in strings as an argument, for example:

```
this.#addRoute("GET", path, handler);
```

There's nothing wrong with this approach too, but I prefer to avoid raw strings. `"GET"` can mean many things, but `HTTP_METHODS.GET` gives us the actual idea of what it is all about.

Let's update our testing code to call the newly created http methods instead:

```
// file: index.js

...

router.get("/", function handleGetBasePath(req, res) {
    console.log("Hello from GET /");
    res.end();
});

router.post("/", function handlePostBasePath(req, res) {
    console.log("Hello from POST /");
    res.end()
});

...
```

If we do a quick test on both the endpoints, every thing seems to be working alright:

```
$ curl -X POST http://localhost:5255/ -v
# Success

$ curl -X POST http://localhost:5255/foo -v
```

```
# Not found

$ curl -X POST http://localhost:5255/foo/bar -v
# Not found

$ curl http://localhost:5255/ -v
# Success

$ curl http://localhost:5255/foo -v
# Not found

$ curl http://localhost:5255/foo -v
# Not found
```

Great! This looks much better than the previous implementation.

6.8 The Need for a `Trie`

Until now, we've been using a straightforward object to store our routes. While this is easy to understand, it's not the most efficient way to store routes, especially when we have a large number of them or when we introduce dynamic routing capabilities like `/users/:id`. It's a simple and readable approach but lacks efficiency and the capability for dynamic routing. As we aim to build a robust, scalable, and high-performance backend framework, it is crucial to optimize our routing logic.

As long as you don't need dynamic parameters, or query parameters, you'd be good enough with a javascript object (like we do now), or a `Map`. But a backend framework that doesn't supports dynamic parameters, or query parsing is as good as a social media site without an ability to add friends.

In this chapter, we'll explore a new data-structure that you may not have heard of before - **Trie**. We'll also look at how we can utilize it to enhance our router's performance.

For example, imagine we have the following four routes:

```
GET /api/v1/accounts/friend
GET /api/v1/accounts/stats
GET /api/v1/accounts/upload
GET /api/v1/accounts/blocked_users
POST /api/v1/accounts/friend
POST /api/v1/accounts/stats
```

```
POST /api/v1/accounts/upload
POST /api/v1/accounts/blocked_users
```

Our current implementation will have them stored as separate keys in the object:

```
{
    "GET /api/v1/accounts/friend": function handle_friend() { ... },
    "GET /api/v1/accounts/stats": function handle_stats() { ... },
    "GET /api/v1/accounts/upload": function handle_upload() { ... },
    "GET /api/v1/accounts/blocked_users": function handle_blocked_users() { ... },
    "POST /api/v1/accounts/friend": function handle_friend() { ... },
    "POST /api/v1/accounts/stats": function handle_stats() { ... },
    "POST /api/v1/accounts/upload": function handle_upload() { ... },
    "POST /api/v1/accounts/blocked_users": function handle_blocked_users() { ... }
}
```

That is not efficient. For most of the applications this is nothing to worry about, but there's a better way. Also with this approach it becomes impossible to extend our router with the other functionalities we mentioned - dynamic routes, queries etc. There's a way to do some regex sorcery to achieve it, but that method will be way way slower. You don't need to sacrifice performance in order to support more features.

A better way to store the routes could be the following:

```
{
    "/api": {
        "/v1": {
            "/accounts": {
                "friend": function handle_friend() { ... },
                "stats": function handle_stats() { ... },
                "upload": function handle_upload() { ... },
                "blocked_users": function handle_blocked_users() { ... }
            }
        }
    }
}
```

This is an easy way to think of how a `Trie` stores the paths.

6.8.1 What is a `Trie` anyway?

A `Trie`, which is also known as a prefix tree, is a specialized tree structure used for storing a mapping between keys and values, where the keys are generally strings. This structure is organized in such a way that all the child nodes that stem from a single parent node have a shared initial sequence of characters, or a "common prefix." So the position of a node in the Trie dictates what key it corresponds to, rather than storing the key explicitly in the node itself.

Imagine we have the following routes:

```
'GET /users'
'GET /users/id'
'POST /users'
```

With our current implementation, the routes object would look like:

```
{
    "GET /users": handler,
    "GET /users/id": handler,
    "POST /users": handler
}
```

But, with a Trie, it will look like the following:

```
    [root]
      |
     GET
      |
    users
     / \
   POST GET
          \
          id
```

Every node, including `root` will be an object that contain some necessary information with it.

1. `handler`: The function to be executed when the route represented by the path to this node is accessed. Not all nodes will have handlers, only the nodes that correspond to complete routes.
2. `path`: The current route segment in string, for example - `/users` or `/id`
3. `param` and `paramName`: If the current path is `/:id` and the client makes a request at `/xyz`, the `param` will be `xyz` and the `paramName` will be `id`.

4. `children`: Any children nodes. (We'll get more deep into this in the upcoming chapters)

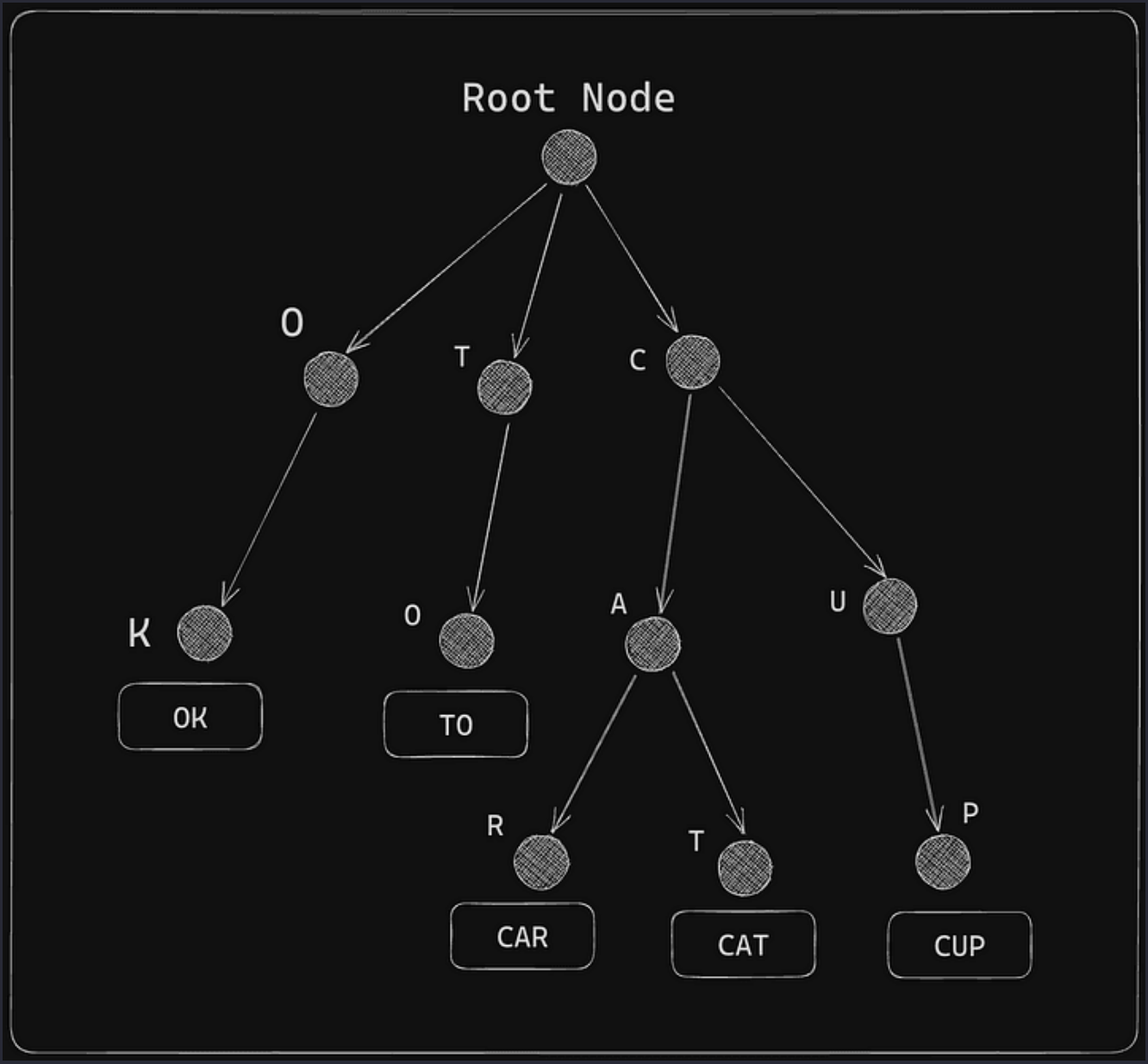
Enough with the theory. In the next chapter, we'll dive into our very first exercise for this book: **implementing a Trie**.

6.9 Exercise 1 - Implementing a `Trie`

> This exercise will motivate you to work on implementing your solution independently. Once you have completed the exercise, you can move on to the next challenge or read the solution to find a different approach.
> In these exercises, we are not focusing on performance, so it's important to focus on making your solution work correctly the first time you attempt to solve a problem.

To re-iterate, Trie (pronounced "try") is a tree-like data structure that stores a dynamic set of strings, typically used to facilitate operations like searching, insertion, and deletion. Tries are particularly useful for tasks that require quick lookups of strings with a common prefix, such as in text autocomplete or in a Router implementation to find the matching paths.

Here's an illustration that shows how does a `Trie` look like in theory:



Here's how you can visualize the Trie above based on the words "OK", "TO", "CAR", "CAT", and "CUP":

6.9.1 Root Node

The Trie starts with a root node that doesn't hold any character. It serves as the starting point of the Trie.

```
    Root
   /  |  \
  T   O   C
```

- **Level 1**: You have the characters "O", "T", and "C" branching from the root node.
- **Level 2 and Beyond**: These nodes further branch out to form the words.

  - ”O” branches to ”K”, completing the word ”OK”.
  - ”T” branches to ”O”, completing the word ”TO”.
  - ”C” branches to ”A” and ”U”:
    * ”A” further branches to ”R” for ”CAR” and ”T” for ”CAT”.
    * ”U” further branches to ”P”, completing the word ”CUP”.

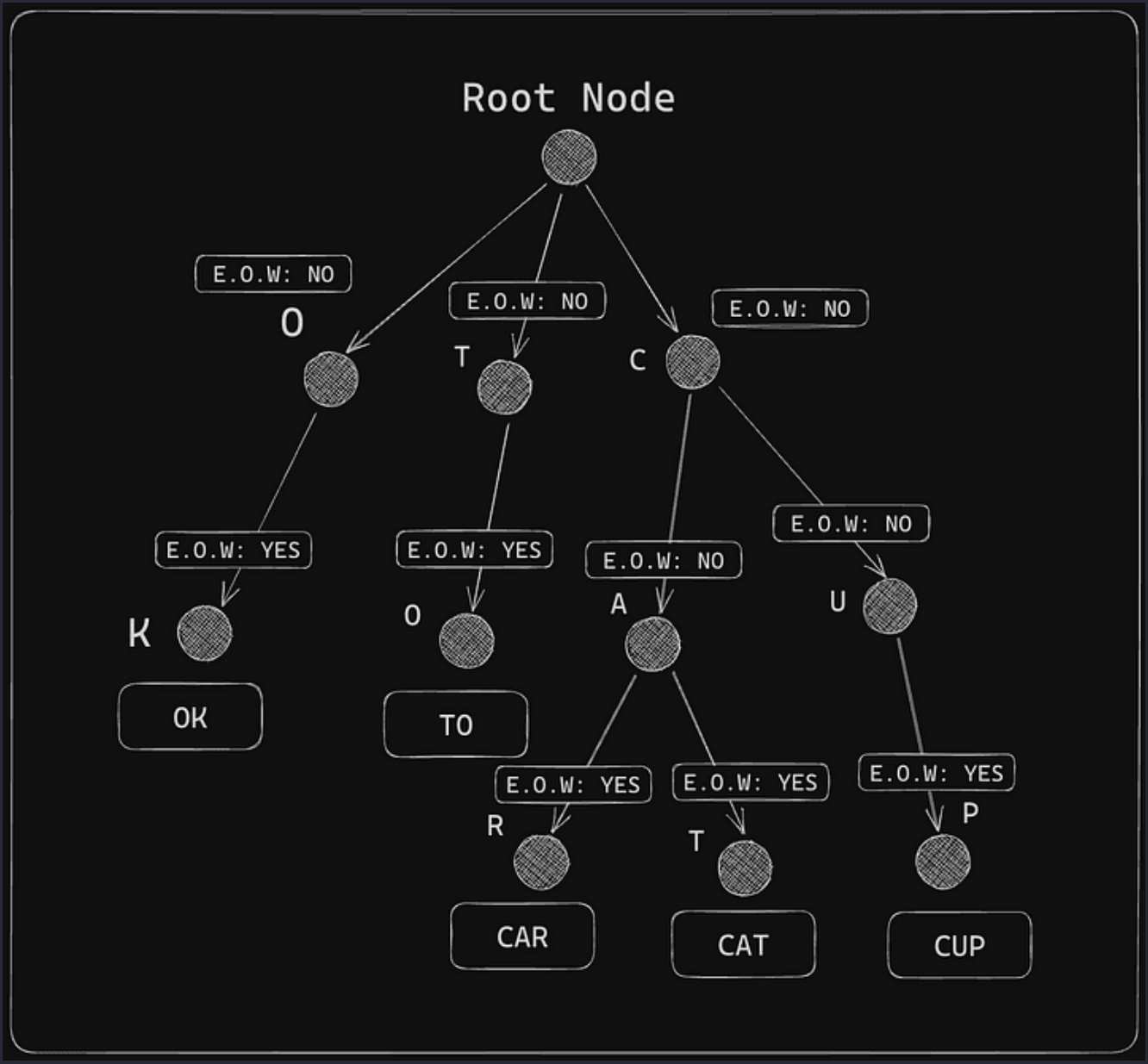
6.9.2 End of the word

The ”end of the word” is often represented by a boolean flag at a node to signify that the path from the root of the Trie to that node corresponds to a complete word. This flag helps distinguish between a string that is merely a prefix and one that is a full word in the Trie.

For example, consider a Trie that stores the words ”car”, ”cat”, and ”cup”. The node corresponding to the last 't' in ”cat” and the last 'p' in ”cup” would have the end-of-word marker, indicating that they are complete words, as opposed to just prefixes. Same for 'k' in ”ok” and 'o' in ”to”

By doing so, if someone searches for ”ca” it should not return true, since we only stored ”cat” and ”car” where as ”ca” is just a prefix.

Here's an another illustration to explain the ”end-of-word” (EOW):



6.9.3 Challenge 1: Basic Trie with `insert` Method

In this first challenge, your task is to implement a Trie data structure with only one functionality: inserting a word into the Trie.

6.9.3.1 Requirements

1. Create a class called `Trie`.
2. Implement an `insert(word)` method that takes a string `word` and inserts it into the Trie.

6.9.3.2 More details

1. **Initialization**: You'll begin with a root node. This node will be the starting point for all word insertions, and it won't store any character itself.
2. **Traversal**: For each character in the word you want to insert, you'll traverse the Trie from the root node, going as far down as the current character sequence allows.
3. **Node Creation**: If a character in the word doesn't match any child node of the current node:
   - Create a new node for that character.
   - Link this new node to the current one.
   - Move down to this new node and continue with the next character in the word.
4. **End-of-Word**: When you've inserted all the characters for a particular word, mark the last node in some way to indicate that it's the end of a valid word. This could be a boolean property in the node object, for example.

Here's the boilerplate to get you started.

> Note: If you wish, you may code everything from scratch, without using the boilerplate below. I recommend doing it that way if you're comfortable.

```
class TrieNode {
  constructor() {
    this.children = new Map(); // To store TrieNode children with char keys
    // this.children = new Map(); You may also use a Map instead.
    this.isEndOfWord = false; // To mark the end of a word
  }
}

class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  insert(word) {
    // Your code here
  }
}
```

Once implemented, your code should allow operations like:

```
const trie = new Trie();
trie.insert("hello");
```

Go ahead and implement the `insert` method, and then share your code to help others or to receive feedback in the Github discussions section. I'll try to review all the code submissions and provide feedback if required.

Great. You just implemented a `Trie` which is a Tree data structure. You've also wrote code to traverse a tree which is generally called "tree traversal".

> In case you were not able to figure out what to do, I would still like you to scrap the code you've written and start again from scratch. Get a pen and paper, and visualize it. That way you can convert hard problems into easier ones.

6.9.3.3 Solution

```
class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  insert(wordToInsert, node = this.root) {
    let length = wordToInsert.length;
    if (length == 0) return;

    const letters = wordToInsert.split("");

    const foundNode = node.children.get(wordToInsert[0]);

    if (foundNode) {
      this.insert(letters.slice(1).join(""), foundNode);
    } else {
      let insertedNode = node.add(letters[0], length == 1);
      this.insert(letters.slice(1).join(""), insertedNode);
    }
  }
```

```
}

class TrieNode {
  constructor() {
    /**
     * Children will be Map<key(String), node(TrieNode)>
     */
    this.isEndOfWord = false;
    this.children = new Map();
  }

  add(letter, _isLastCharacter) {
    let newNode = new TrieNode();
    this.children.set(letter, newNode);

    if (_isLastCharacter) newNode.isEndOfWord = true;
    return newNode;
  }
}

const trie = new Trie();
trie.insert("node");
trie.insert("note");
trie.insert("not");
```

Let's take a look at the code:

```
class TrieNode {
  constructor() {
    this.isEndOfWord = false;
    this.children = new Map();
  }
}
```

Initializes an instance of the `TrieNode` class. A TrieNode has two properties:

- `isEndOfWord`: A boolean flag that denotes whether the node is the last character of a word in the Trie. Initially set to `false`.
- `children`: A Map to store the children nodes. The keys are letters, and the values are TrieNode objects.

```
add(letter, _isLastCharacter) {
        let newNode = new TrieNode();
        this.children.set(letter, newNode);

        if (_isLastCharacter) newNode.isEndOfWord = true;
        return newNode;
}
```

I've created a utility method on `TrieNode` to extract some logic from the `Trie.insert` method. This adds a new `TrieNode` as a child of the current node, corresponding to the given letter.

```
class Trie {
  insert(wordToInsert, node = this.root) {
    let length = wordToInsert.length;

    // Exit condition: If the word to insert is empty, terminate the recursion.
    if (length == 0) return;

    // Convert the string into an array of its individual characters.
    const letters = wordToInsert.split("");

    // Attempt to retrieve the TrieNode corresponding to the first letter
    // of the word from the children of the current node.
    const foundNode = node.children.get(wordToInsert[0]);

    if (foundNode) {
      // The first letter already exists as a child of the current node.
      // Continue inserting the remaining substring (sans the first letter)
      // starting from this found node.
      this.insert(letters.slice(1).join(""), foundNode);
    } else {
      // The first letter doesn't exist in the children of the current node.
      // Create a new TrieNode for this letter and insert it as a child of the current node.
      // Also, set the node's 'isEndOfWord' flag if this is the last character of the word.
      let insertedNode = node.add(letters[0], length == 1);

      // Continue inserting the remaining substring (without the first letter)
      // starting from this new node.
```

```
      this.insert(letters.slice(1).join(""), insertedNode);
    }
  }
}
```

The previous code looks fine. However, as our backend library/framework prioritizes performance, we will optimize the code we write in the upcoming exercises to improve efficiency.

The code above is acceptable, and not a bad implementation. But we can do better.

6.9.3.4 Using a `for` loop instead of recursion

I am not a big fan of recursion, and I prefer using loops over recursion most of the time. For loop is much easier to reason about and as someone who's reading the code, it's easier to understand what's going on.

Let's update our submission to use a `for` loop instead of recursion.

```
/** this class remains identical **/
class TrieNode {
  isEndOfWord = false;
  children = new Map();
}

class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  insert(word, node = this.root) {
    const wordLength = word.length;
    if (wordLength === 0) return;

    for (let idx = 0; idx < wordLength; idx++) {
      let char = word[idx];

      // Check if the current node has a child node for the current character.
      if (!node.children.has(char)) {
        // If not, create a new TrieNode for this character and add it to the children of the
        ↪ current node.
        node.children.set(char, new TrieNode());
```

```
      }

      // Move to the child node corresponding to the current character.
      node = node.children.get(char);
    }

    node.isEndOfWord = true;
  }
}
```

6.9.4 Challenge 2: Implement `search` method

Now that we have a Trie with insertion capabilities, let's add a `search` method.

6.9.4.1 Requirements

1. Add a `search(word)` method to the `Trie` class.
2. The method should return `true` if the word exists in the Trie and `false` otherwise.

6.9.4.2 More details

1. **Start at the Root**: Begin your search at the root node.
2. **Traversal**: For each character in the word, traverse down the Trie, going from one node to its child that corresponds to the next character.
3. **Word Existence**: If you reach a node that is marked as the end of a word (`isEndOfWord = true`), and you've exhausted all the characters in the word you're searching for, then the word exists in the Trie.

Once implemented, your code should allow:

```
const trie = new Trie();
trie.insert("code");
trie.insert("coding");

let found = trie.search("code");
console.log(found); // true

found = trie.search("cod");
console.log(found); // false
```

Go ahead and implement the `Trie.search` method. Don't read anything below before implementing it yourself.

If you are having trouble or are stuck, here are some hints to help you with the implementation -

6.9.4.3 Hints

1. **Starting Point**: Similar to the `insert` method, you'll start at the root node and traverse the Trie based on the characters in the word you're searching for.
2. **Character Check**: For each character in the word, check if there's a child node for that character from the current node you're at.
   - **If Yes**: Move to that child node.
   - **If No**: Return `false`, as the word can't possibly exist in the Trie.
3. **End-of-Word Check**: If you've reached the last character of the word, check the `isEndOfWord` property of the current node. If it's `true`, the word exists in the Trie; otherwise, it doesn't.
4. **Recursion or Loop**: You can choose to implement this method either recursively or iteratively.
   - **Recursion**: If you opt for recursion, you might want to include an additional parameter in the `search` method for the current node, similar to how you did it for the `insert` method.
   - **Loop**: If you prefer loops, you can use a `for` loop to go through each character in the word, updating your current node as you go along.
5. **Return Value**: Don't forget to return `true` or `false` to indicate whether the word exists in the Trie.

Good luck!

6.9.4.4 Solution

Again, I chose to implement tree traversal using a for loop instead of recursion.

```
search(word) {
    // Initialize 'currentNode' to the root node of the Trie.
    let currentNode = this.root;

    // Loop through each character in the input word.
    for (let index = 0; index < word.length; index++) {

        // Check if the current character exists as a child node
        // of the 'currentNode'.
        if (currentNode.children.has(word[index])) {
```

```
            // If it does, update 'currentNode' to this child node.
            currentNode = currentNode.children.get(word[index]);
        } else {

            // If it doesn't, the word is not in the Trie. Return false.
            return false;
        }
    }

    // After looping through all the characters, check if the 'currentNode'
    // marks the end of a word in the Trie.
    return currentNode.isEndOfWord;
}
```

Awesome work. Now you know the basics of the `Trie` data structure and how to implement it. In the next exercise, we'll implement our `Router` from scratch! The next exercise will be more challenging and exhaustive.

6.10 Exercise 2 - Implementing our Trie based `Router`

> This challenge is designed to push your boundaries and is considered to be more advanced than usual. It's completely okay if you don't crack it on your first attempt. The key is to persist, revisit your logic, and don't hesitate to iterate on your solutions.

Since we just built a Trie data structure that can efficiently insert and search words, we can further enhance its capabilities by extending it to implement a Trie-based router for matching URL patterns. This powerful application of `Trie` data structure is commonly utilized in web frameworks, where it plays a crucial role in efficiently routing incoming HTTP requests to their respective handler functions.

By building a `Trie`-based router, our framework can achieve optimal performance and scalability, ensuring that each request is efficiently directed to the appropriate handler for processing.

6.10.1 Challenge 1: Implementing the `addRoute` method

6.10.1.1 Requirements

Create a new class, `TrieRouter`, similar to what we had earlier - `Trie`. Add a method, `addRoute`, that takes in a URL pattern (like `/home` or `/user/status/play`) as a first parameter and a handler function as second

parameter. Then insert the URL pattern into the `TrieRouter`, associating the handler function with the last node of that pattern.

6.10.1.2 More details

1. **Class Definition**: Define a class named `TrieRouter`. This class should contain:
    - A root node, which is the starting point of the Trie.
    - A method called `addRoute`.
2. **Route Node**: Define a class named `RouteNode` which will represent all the nodes.
    1. `RouteNode` should contain the handler function, which will be `null` or `undefined` for all nodes except the end nodes of each URL pattern.
    2. `RouteNode` should also contain a `Map` to store its children nodes, where the key will be the URL segment eg "home" or "user", and the value will be another `RouteNode`
3. **Root Node**: The root node is an empty node of the type `RouteNode`, that serves as the starting point for inserting new URL patterns into the Trie. Initialize it in the constructor of `TrieRouter`.
4. **Method - `addRoute`**: This method takes in two parameters:
    - `path`: A string representing the URL pattern to add to the Trie. The URL pattern will be segmented by forward slashes `/`.
    - `handler`: A function that should be called when the URL pattern is matched.
    - Remove the trailing slash `/` from the `path` if it exists.
    - The method should insert the `path` into the `TrieRouter`, associating the `handler` function with the last node of that pattern.
5. **Trailing forward-slashes**: You should treat routes that end with a forward slash `/` the same as those that don't, so that `/home/` and `/home` point to the same handler.
6. **Repeated forward-slashes**: You should remove all the repeated `/` in the path.
    1. `/user//hello////` should resolve as `/user/hello`
    2. `/user//////////` should resolve as `/user`
7. **Reject URLs** that contain any kind of white space. For example `/ user/ node /` should throw an error.
8. **Reject URLs** that do not start with `/`
    1. If someone uses `trieRouter.addRoute("hi/something")`, your code should throw an error.

Once implemented, we should be able to do something like this:

```
const trieRouter = new TrieRouter();

function ref() {}

trieRouter.addRoute("/home/", ref);
trieRouter.addRoute("/  user/  status/play", function inline() {});

// /home -> valid
// /user/status/play -> valid
// /user/status -> invalid
// /user -> invalid
// /home/ -> valid
// /user/status/play/ -> invalid
```

You don't need to worry about making HTTP requests just yet. A correctly implemented `TrieRouter` should look like this after adding both the routes mentioned above -

```
                (Root)
                  |
      -----------------------
      |           |          |
    "home"      "user"     (other segments)
      |           |
function ref   "status"
                  |
                  |
               "play"
                  |
         function inline
```

Go ahead and implement your version of the `TrieRouter`, `RouteNode` and `addRoute`. Here's a starting boilerplate for the challenge. You may proceed without using the boilerplate code if you're comfortable.

You may then share your code to help others or to receive feedback in the Github discussions section. I'll try to review all the code submissions and provide feedback if required.

```
class TrieRouter {
  constructor() {
    this.root = new RouteNode();
```

```
  }

  addRoute(path, handler) {
    /* Add route code goes here */
  }
}

class RouteNode {
  constructor() {
    /** Define handler and children map **/
  }
}
```

6.10.1.3 Hints

1. Remember that a Trie is a tree-like structure where each node represents a piece/segment of a URL. Understanding the hierarchy can simplify the process.
2. Before diving into implementing all the conditions like removing trailing slashes or spaces, make sure your Trie works with the simplest case, such as adding a single route.
3. Consider breaking the URL path into segments using `split("/")` and loop through the segments to traverse the Trie.
4. Keep in mind that the handler function is associated with the end node of the URL pattern. Make sure you place the handler only at the right node.
5. Use the `Map` in each node to store its children. When adding a new route, check if a node for a segment exists; if it does, traverse to it. Otherwise, create a new node.
6. To deal with trailing slashes, and repeated slashes, you could write utility functions that normalize the path before processing it.
7. You should throw an error if the path contains any whitespace.

6.10.1.4 Solution

Kudos to those who successfully implemented the `addRoute` function in the `TrieRouter` class. You've just completed the first difficult exercise in this book, showcasing not only your coding abilities but also your problem-solving skills.

For those who found this challenge particularly challenging, don't get discouraged. The complexities you faced are what deepen your understanding and enhance your coding skills. Consider revisiting this exercise

after looking at the solution or scraping and starting again from scratch.

```
class RouteNode {
  constructor() {
    this.children = new Map();
    this.handler = null;
  }
}

class TrieRouter {
  constructor() {
    this.root = new RouteNode();
  }

  addRoute(path, handler) {
    if (typeof path !== "string" || path[0] !== "/") throw new Error("Malformed path provided.");
    if (typeof handler !== "function") throw new Error("Handler should be a function");

    let currentNode = this.root;
    let routeParts = path.split("/").filter(Boolean);

    for (let idx = 0; idx < routeParts.length; idx++) {
      const segment = routeParts[idx].toLowerCase();
      if (segment.includes(" ")) throw new Error("Malformed path parameter");

      let childNode = currentNode.children.get(segment);
      if (!childNode) {
        childNode = new RouteNode();
        currentNode.children.set(segment, childNode);
      }

      currentNode = childNode;
    }
    currentNode.handler = handler;
  }
}

const trieRouter = new TrieRouter();
```

```
function ref() {}

trieRouter.addRoute("/home/", ref);
trieRouter.addRoute("/user/status/play", function inline() {});

trieRouter.printTree();
```

Let's visualize our tree. I've included a new `printTree` method inside the `TrieRouter` class prints all the nodes of our `TrieRouter` recursively:

```
class TrieRouter {
  ...

  printTree(node = this.root, indentation = 0) {
    const indent = "-".repeat(indentation);

    node.children.forEach((childNode, segment) => {
      console.log(`${indent}${segment}`);
      this.printTree(childNode, indentation + 1);
    });
  }

  ...
}
```

To check our output, let's execute our file:

```
const trieRouter = new TrieRouter();

function ref() {}

trieRouter.addRoute("/home/", ref);
trieRouter.addRoute("/user/status/play", function inline() {});
trieRouter.printTree();
```

Output:

```
$node trie_router.js

## OUTPUT
-home
-user
--status
---play
```

Looks perfect. Let's go through the code and understand what's going on.

6.10.1.5 Explanation

```
class RouteNode {
  constructor() {
    // Create a Map to store children nodes
    this.children = new Map();

    // Initialize the handler to null
    this.handler = null;
  }
}
```

In the `RouteNode` class, each node is initialized with a `handler` set to `null`. This handler will hold a reference to the function we want to execute when a route matching the URL pattern is requested. Alongside the handler, we created a `children` Map. This Map will contain references to the next nodes in the Trie, allowing us to navigate through the Trie using URL segments as keys.

```
class TrieRouter {
  constructor() {
    // Create a root node upon TrieRouter instantiation
    this.root = new RouteNode();
  }
}
```

The `TrieRouter` class acts as a manager for the Trie data structure. When an instance of this class is created, a `root` node is initialized. This root node acts as the entry point for any operation that needs to traverse the Trie, essentially representing the root of the Trie structure.

```
addRoute(path, handler) {
    // Validate input types
    if (typeof path ≠ "string" || path[0] ≠ "/") throw new Error("Malformed path provided.");
    if (typeof handler ≠ "function") throw new Error("Handler should be a function");
}
```

The `addRoute` method is responsible for adding URL patterns and their corresponding handlers to the Trie. The method starts by validating the inputs, ensuring that `path` is a string that starts with a slash and `handler` is a function. If either of these conditions isn't met, an error is thrown.

```
addRoute(path, handler) {
    ...
    // Split the path into segments and filter out empty segments
    let routeParts = path.split("/").filter(Boolean);
}
```

The next part of the `addRoute` method preprocesses the path. The path is split into its segments (parts between slashes), and any empty segments are filtered out.

```
addRoute(path, handler) {
    ...
    // Start at the root node of the Trie
    let currentNode = this.root;

    // Loop through all segments of the route
    for (let idx = 0; idx < routeParts.length; idx++) {
        const segment = routeParts[idx].toLowerCase();
        if (segment.includes(" ")) throw new Error("Malformed path parameter");

        // Attempt to find the next node in the Trie
        let childNode = currentNode.children.get(segment);
        ...
}
```

The `addRoute` method continues by setting `currentNode` to the `root` of the Trie. A `for` loop then iterates through each segment in the `routeParts` array. For each segment, the code checks if a child node with that segment as the key already exists in the `children` Map of the current node. If the segment contains spaces, an error is thrown.

We're also converting the segment to lowercase to ensure case-insensitivity in our router. This means that `/home` and `/Home` will be treated as the same route.

```
addRoute(path, handler) {
    ...

    // If the next node doesn't exist, create it
    if (!childNode) {
        childNode = new RouteNode();
        currentNode.children.set(segment, childNode);
    }

    // Move to the next node for the next iteration
    currentNode = childNode;
}
```

If a child node for the current segment does not exist, a new `RouteNode` is instantiated, and it's added to the `children` Map of the current node with the segment as the key. The current node is then updated to this new node, ready for the next iteration or to end the loop.

```
addRoute(path, handler) {
    ...
    // Assign the handler to the last node
    currentNode.handler = handler;
}
```

After the loop completes, the handler function is associated with the last node in the path.

That's it. We now have a working implementation of our router that supports adding routes. The next challenge involves finding a route and returning the handler associated with it.

6.10.2 Challenge 2: Implementing the `findRoute` method

6.10.2.1 Requirements

You've successfully implemented the `addRoute` method to build our `Trie`-based router. Now, lets extend our `TrieRouter` class by adding another method, `findRoute`. This method should take a URL pattern (e.g., `/home` or `/user/status/play`) as its parameter. Search the `TrieRouter` and find the handler function associated with the last node that matches the pattern.

6.10.2.2 More details

1. **Method - `findRoute`**: Add a method to your `TrieRouter` class called `findRoute`.
   - This method should take a single parameter, `path`, which is a string representing the URL pattern to find in the Trie.
   - Return the handler function associated with the last node of the matching URL pattern.
   - If the URL pattern is not found, return `null` or some indication that the route does not exist.
2. **Path Normalization**: We do not need to worry about path normalization. We are searching for a route, not adding one. So, we expect the client to make a request with a path identical to the one configured in the router. For checks like - Whether the path starts with `/`? We do't need to worry about these is because Node.js's HTTP server will provide the URL path in this format.
3. **Traversal**: Start from the root node and traverse the Trie based on the URL segments. Retrieve the handler function from the last node if the path exists.
4. **Route Matching**: The Trie should now allow for a partial match. For instance, if a handler is set for `/user/status`, a request for `/user/status/play` should return null if `/user/status/play` has not been set!
5. **Case Sensitivity**: Make sure to convert the url paths into lower-case before matching. So `/AbC` and `/abc` should result to the same handler.

Once implemented, we should be able to do something like this:

```
const trieRouter = new TrieRouter();

function homeHandler() {}
function userHandler() {}

trieRouter.addRoute("/home", homeHandler);
trieRouter.addRoute("/user/status", userHandler);

console.log(trieRouter.findRoute("/home")); // Should return homeHandler
console.log(trieRouter.findRoute("/user/status")); // Should return userHandler
console.log(trieRouter.findRoute("/user/status/play")); // Should return null
```

Feel free to share your implementation or ask for feedback in the Github discussions section. I'll try to review all code submissions and provide feedback if required.

6.10.2.3 Starting Boilerplate

Feel free to use the starting boilerplate below. If you are comfortable, you may proceed without it.

```
class TrieRouter {
  constructor() {
    this.root = new RouteNode();
  }

  addRoute(path, handler) {
    /* Your addRoute code */
  }

  findRoute(path) {
    /* Your findRoute code goes here */
  }
}
```

6.10.2.4 Hints

1. When traversing the Trie, you may find it beneficial to break down the URL pattern into segments just like you did while inserting the route.
2. Be careful about the return values. Ensure you return the handler function if a match is found and a suitable indicator (like `null`) if no match exists.

6.10.2.5 Solution

Here's the solution that I came up with:

```
class Router {
    constructor() {
        this.root = new RouteNode();
    }

    addRoute(path, handler) {
        ...
        /** Nothing changed **/
    }
```

```
    findRoute(path) {
        let segments = path.split("/").filter(Boolean);
        let currentNode = this.root;

        for (let idx = 0; idx < segments.length; idx++) {
            const segment = segments[idx];

            let childNode = currentNode.children.get(segment);
            if (childNode) {
                currentNode = childNode;
            } else {
                return null;
            }
        }

        return currentNode.handler;
    }


    printTree(node = this.root, indentation = 0) {
       /** Nothing changed **/
    }
}

class RouteNode {
    /** same as before **/
}
```

6.10.2.6 Explanation

```
    findRoute(path) {
       // Split the path into segments and filter out empty segments
       let segments = path.split("/").filter(Boolean);

       // Start at the root node of the Trie
```

```
        let currentNode = this.root;
        ...
    }
```

We've initialized two key variables. The `segments` variable stores the differet parts of the URL split by `/`. That is, if the path is `/home/users`, the segments array will contain `["home", "users"]`. The `currentNode` variable keeps track of our current position in the Trie and is initialized to the root node.

```
findRoute(path) {
    ...

    // Traverse the Trie based on the URL segments
    for (let idx = 0; idx < segments.length; idx++) {
        // Retrieve the current URL segment
        let segment = segments[idx];

        // Retrieve the child node corresponding to the current URL segment
        let childNode = node.children.get(currPart);
    ...
}
```

We loop through each segment of the `segments` array. Within the loop, `segment` holds the current URL segment, and `childNode` is obtained from the `children` map of the current `node` based on this segment. This part is crucial because we're determining if a child node exists for the current URL segment in our Trie.

```
findRoute(path) {
    ...
     // If the next node exists, update the current node
     if (childNode) {
        currentNode = childNode;
     } else {
    // If the next node doesn't exist, exit the function with null to indicate no matching route
        return null;
     }

     // Since we've reached the end of the URL, return the handler if found, otherwise `null`
     return currentNode.handler;
    ...
```

```
}
```

First, this portio of the code checks whether `childNode` exists. If it doesn't, the function immediately returns `null`. This means that the Trie doesn't contain a matching route for the given URL, and there's no need to continue searching.

This allows us to move deeper into the Trie as we iterate through each URL segment. After the loop, the method returns the `handler` that was found. If no handler is found during the Trie traversal, the return value will be `null`.

Let's test our code:

```
const trieRouter = new TrieRouter();

function ref() {}
function refs() {}

trieRouter.addRoute("/home/", ref);
trieRouter.addRoute("/user/status/play", function inline() {});
trieRouter.addRoute("/home/id", refs);
// trieRouter.addRoute("/throw/   error", function throwError() {}) // Should not let the program
 ↪  to run.

console.log(trieRouter.findRoute("/home/"));
console.log(trieRouter.findRoute("/home"));
console.log(trieRouter.findRoute("/home/id/"));
console.log(trieRouter.findRoute("/home/id/1"));
console.log(trieRouter.findRoute("/user/status/play"));
console.log(trieRouter.findRoute("/user/status/play/"));
```

This outputs:

```
[Function: ref]
[Function: ref]
[Function: refs]
null
[Function: inline]
[Function: inline]
```

Everything seems to be working well. This is it for the `findRoute` method. This was much easier than our

`addRoute` implementation, since we only cared about searching. Excellent, we've grasped the basics well! Now let's move on to the more advanced features in the next chapter, ie Implementing HTTP methods with our router.

6.11 Exercise 3 - Adding `HTTP` method support

So far, we've built a router capable of matching URL paths to specific handlers. This is a good starting point, but as of now, our router does not differentiate between different HTTP methods like GET, POST, PUT, DELETE, etc. In real-world applications, the same URL path can behave differently based on the HTTP method used, making our current router almost useless for such scenarios.

6.11.1 Requirements

To make our router more useful and versatile, we need to extend the existing `TrieRouter` and `RouteNode` classes to support different HTTP methods (GET, POST, PUT, DELETE, etc.). This means that each node in the Trie could potentially have multiple handler functions, one for each HTTP method.

6.11.2 More details

1. Continue with the existing router class `TrieRouter`. Add new functionalities to it.
2. The key in the `handler` `Map` will be the HTTP method as a string (like "GET", "POST") and the value will be the handler function for that HTTP method.
3. Modify the `addRoute` method of the `TrieRouter` class to take an additional parameter `method`.
   - `method`: A string representing the HTTP method. This could be "GET", "POST", "PUT", "DELETE", etc.
4. Also update the `findRoute` method. Now it will have another parameter - `method`, to search for routes based on the HTTP method as well as the path.
5. If a handler for a specific path and HTTP method is already present, the new handler should override the old one.

6.11.3 Example

Once implemented, the usage might look like this:

```
const trieRouter = new TrieRouter();

function getHandler() {}
function postHandler() {}
```

```
trieRouter.addRoute("/home", "GET", getHandler);
trieRouter.addRoute("/home", "POST", postHandler);

console.log(trieRouter.findRoute("/home", "GET")); // -> fn getHandler() {..}
console.log(trieRouter.findRoute("/home", "PATCH")); // -> null or undefined
console.log(trieRouter.findRoute("/home", "POST")); // -> fn postHanlder() {..}
```

Go ahead and add the functionality to our `TrieRouter` class. This will involve making a lot of changes to the previous code. Feel free to share your implementation or ask for feedback in the Github discussions section.

6.11.4 Hints

1. When you're adding or searching for a route, make sure to consider both the path and the HTTP method.
2. Take care to handle the HTTP method case-insensitively (prefer uppercase). It's common to receive HTTP method names in different cases.
3. Be careful with your error-handling logic to correctly manage the situation where the client does not provide a valid HTTP method.
4. As with Challenge 1, start by making sure the Trie works for a simple case before diving into the more complex functionalities.
5. Don't forget to update your utility functions and other methods to be compatible with these new requirements.

6.11.5 Solution

Here's the solution I came up with:

```
const HTTP_METHODS = {
  GET: "GET",
  POST: "POST",
  PUT: "PUT",
  DELETE: "DELETE",
  PATCH: "PATCH",
  HEAD: "HEAD",
  OPTIONS: "OPTIONS",
  CONNECT: "CONNECT",
```

```
  TRACE: "TRACE",
};

class RouteNode {
  constructor() {
    this.children = new Map();
    this.handler = new Map();
  }
}

class TrieRouter {
  constructor() {
    this.root = new RouteNode();
  }

  addRoute(path, method, handler) {
    if (typeof path !== "string" || path[0] !== "/") throw new Error("Malformed path provided.");
    if (typeof handler !== "function") throw new Error("Handler should be a function");
    if (!HTTP_METHODS[method]) throw new Error("Invalid HTTP Method");

    let currentNode = this.root;
    let routeParts = path.split("/").filter(Boolean);

    for (let idx = 0; idx < routeParts.length; idx++) {
      const segment = routeParts[idx].toLowerCase();
      if (segment.includes(" ")) throw new Error("Malformed `path` parameter");

      let childNode = currentNode.children.get(segment);
      if (!childNode) {
        childNode = new RouteNode();
        currentNode.children.set(segment, childNode);
      }

      currentNode = childNode;
    }
    currentNode.handler.set(method, handler); // Changed this line
  }
```

```
  findRoute(path, method) {
    let segments = path.split("/").filter(Boolean);
    let currentNode = this.root;

    for (let idx = 0; idx < segments.length; idx++) {
      const segment = segments[idx];

      let childNode = currentNode.children.get(segment);
      if (childNode) {
        currentNode = childNode;
      } else {
        return null;
      }
    }

    return currentNode.handler.get(method); // Changed this line
  }

  printTree(node = this.root, indentation = 0) {
    /** Unchanged **/
  }
}
```

The new HTTP method implementation introduces only some minor key changes to extend the existing router implementation to support HTTP methods. Below are the details of what was changed and why:

```
const HTTP_METHODS = {
  GET: "GET",
  POST: "POST",
  PUT: "PUT",
  DELETE: "DELETE",
  PATCH: "PATCH",
  HEAD: "HEAD",
  OPTIONS: "OPTIONS",
  CONNECT: "CONNECT",
  TRACE: "TRACE",
};
```

Firstly, we've defined a constant object named `HTTP_METHODS` to represent the different HTTP methods. This serves as a reference for the HTTP methods that our `TrieRouter` class will support. We might even do some validation, but that is not necessary (we'll look at it in a later chapter why validation isn't required here)

```
class TrieRouter {
    addRoute(path, method, handler) { ... }
    ...
}
```

In our `TrieRouter` class, we updated the `addRoute` method. It now takes an additional argument, `method`, which specifies the HTTP method for the route.

```
addRoute(path, method, handler) {
  if (typeof path !== "string" || path[0] !== "/") throw new Error("Malformed path provided.");
  if (typeof handler !== "function") throw new Error("Handler should be a function");

  // New check for HTTP method
  if (!HTTP_METHODS[method]) throw new Error("Invalid HTTP Method");
  ...
}
```

The error handling has been updated to ensure the `method` is a valid HTTP method.

```
this.handler = new Map();
```

The `handler` in `RouteNode` has changed from a single function reference to a `Map`. This allows you to store multiple handlers for the same path but with different HTTP methods.

```
addRoute(path, method, handler) {
  ...
  // Previous -> currentNode.handler = handler;
  currentNode.handler.set(method, handler);
}

findRoute(path, method) {
  ...
  // Previous -> return currentNode.handler;
  return currentNode.handler.get(method); // Changed this line
}
```

In both the `addRoute` and `findRoute` methods, we've updated the line that sets and gets the handler for a specific path. Now, the handler is stored in the `handler` map of the current node, with the HTTP method as the key.

6.12 Adding HTTP methods to the Router

In this exercise, we will improve our `TrieRouter` API by implementing support for HTTP verbs (GET, POST, PUT, DELETE, etc.) directly instead of using raw strings. Let's look at our current implementation of the `TrieRouter`:

```
trieRouter.addRoute("GET", "/users", () => {});
trieRouter.addRoute("GET", "/", () => {});
```

As you could already feel, this approach is not very flexible and uses raw strings, which can lead to typing errors, and has no auto-completion support unfortunately. Let's improve this by adding support for HTTP methods directly to the `TrieRouter` API.

6.12.1 Update the `TrieRouter` class

```
const HTTP_METHODS = { ... };

class TrieRouter {
    ...

    #addRoute(path, method, handler) {
    ...
    }

    get(path, handler) {
        this.#addRoute(path, HTTP_METHODS.GET, handler);
    }

    post(path, handler) {
        this.#addRoute(path, HTTP_METHODS.POST, handler);
    }

    put(path, handler) {
```

```
        this.#addRoute(path, HTTP_METHODS.PUT, handler);
    }

    delete(path, handler) {
        this.#addRoute(path, HTTP_METHODS.DELETE, handler);
    }

    patch(path, handler) {
        this.#addRoute(path, HTTP_METHODS.PATCH, handler);
    }

    head(path, handler) {
        this.#addRoute(path, HTTP_METHODS.HEAD, handler);
    }

    options(path, handler) {
        this.#addRoute(path, HTTP_METHODS.OPTIONS, handler);
    }

    connect(path, handler) {
        this.#addRoute(path, HTTP_METHODS.CONNECT, handler);
    }

    trace(path, handler) {
        this.#addRoute(path, HTTP_METHODS.TRACE, handler);
    }
    ...
}
```

6.12.1.1 Explanation

Firstly, we've added dedicated methods for each HTTP method in the `TrieRouter` class. This allows users to define routes more intuitively using method-specific calls like `trieRouter.get('/home', handler)` for the GET method and `trieRouter.post('/home', handler)` for the POST method.

In each of these methods, we call the existing `addRoute` method, passing the appropriate HTTP method from the `HTTP_METHODS` object.

This change allows for a consistent and clear way to find routes based on the HTTP method.

Sedcondly, we've made the `addRoute` method private by prefixing it with a `#`. This means that the `#addRoute` method can now only be accessed from within the `TrieRouter` class and not from outside.

Now, to test the new API, let's update our previous example:

```
const trieRouter = new TrieRouter();

trieRouter.get("/users", function get1() {});
trieRouter.post("/users", function post1() {});
trieRouter.put("/users", function put1() {});
trieRouter.delete("/users", function delete1() {});

console.log(trieRouter.findRoute("/users/e", HTTP_METHODS.GET)); // null
console.log(trieRouter.findRoute("/users", HTTP_METHODS.POST)); // function post1() {}
console.log(trieRouter.findRoute("/users", HTTP_METHODS.PUT)); // function put1() {}
console.log(trieRouter.findRoute("/users", HTTP_METHODS.TRACE)); // undefined
```

Looks good, and now we have a more intuitive way to define routes based on HTTP methods. Let's move on to the next exercise to add support for route parameters.

6.13 Exercise 4 - Implementing Dynamic Routing

When we're building a server application, dynamic routing is an essential feature for creating flexible and scalable applications. To fully grasp its significance and how we can enhance our router to support dynamic routes like `/users/:id`, let's delve into the concept of dynamic routing.

6.13.1 Why Dynamic Routing?

At its core, dynamic routing refers to the ability of a web application (not just server) to handle requests to URLs that are not predetermined but rather, are defined by patterns. It allows developers to create routes that can match a range of URL structures, often using parameters. For instance, a URL like `/users/:user_id` can handle requests for any user ID, where `:user_id` is a variable part of the URL.

6.13.1.1 Flexibility

Dynamic routing introduces a level of flexibility that static routing cannot match. As applications grow and become more complex, new routes often need to be added. With dynamic routing, you can handle a vast number of routes with just a few route patterns, making the application more scalable and easier to maintain. For example, imagine having to define a route for every kind of asset your website serves.

You may do something like this

```
// images
app.get("/static/imgs/img1.png", img_handler);
app.get("/static/imgs/img2.png", img_handler);
app.get("/static/imgs/img3.png", img_handler);
app.get("/static/imgs/img4.png", img_handler);

// javascript
app.get("/static/js/main.js", script_handler);
app.get("/static/js/third_party.js", script_handler);

/** And so on**/
```

This is quite tedious, and you cannot expect an application like this to scale. What if we had a dynamic route to serve all the assets?

```
app.get("/static/img/:image_file_name", img_handler);
app.get("/static/js/:javascript_file_name", img_handler);
```

This is somewhat better than the previous.

> However, this is still not the best way to handle assets. You may have subdirectories - i.e `/img/compressed/webps/img.webp`. You will get a route not found while doing the method above. To solve this issue, we have a concept of wildcards. You don't need to worry about wildcards just yet. We'll cover them in an upcoming challenge.

6.13.1.2 Better User Experience

Dynamic routes allow for creating more personalized user experiences. For example, in a blog application, a dynamic route like `/posts/:postId` can display a specific post based on the ID in the URL. This approach makes it straightforward to link to specific content, improving the navigability and user engagement.

6.13.1.3 Better Developer Experience

By using dynamic routes, developers of our framework can avoid the tedium of defining every possible URL path in their application. This not only saves time but also reduces the risk of errors. A single dynamic route can replace dozens, if not hundreds, of static routes, streamlining the development process.

6.13.1.4 Better SEO

Dynamic routing can also contribute to better Search Engine Optimization (SEO). With the ability to generate clean and meaningful URLs (e.g., `/game/dota2` instead of `/game?uid=dota2`), dynamic routes make URLs more understandable to both users and search engines, potentially improving search rankings.

6.13.2 Anatomy of a dynamic route

A dynamic route follows a structure where certain parts of the URL path are variable, known as dynamic segments.

```
/[Static Path Segment]/[Dynamic Segment]/[More Static or Dynamic Segments]
```

Example URL: https://github.com/:user_id/repos

1. Static Path Segment: 'repos'
    - A fixed part of the URL path that doesn't change.
2. Dynamic Segment: `:user_id`
    - A variable part of the URL. The `user_id` can be any value, representing a specific user.

6.13.3 Challenge: Enhance the `TrieRouter` Class to Support Dynamic Routing

In this challenge, we will modify the existing `TrieRouter` class to support dynamic routing. Dynamic routes are identified by segments of the URL that begin with a colon (`:`), allowing for variable parameters in the route paths. This enhancement will enable the router to extract these dynamic parameters and make them accessible to the corresponding route handler.

6.13.3.1 Requirements

1. **Dynamic Route Identification**: Modify the `#addRoute` method to recognize dynamic route segments (e.g., `/users/:userId`) and store them appropriately within the `RouteNode`.
2. **Parameter Extraction**: Update the `findRoute` method to extract dynamic parameters from the URL path and return them alongside the route handler.
3. **Data Structure Update**: Enhance the `RouteNode` class to include a property for storing dynamic parameter names.
4. **Error Handling**: Ensure that the router continues to validate paths and handlers, throwing appropriate errors for malformed paths or invalid HTTP methods.

6.13.3.2 More Details

1. **Dynamic Parameter Storage**: In the `RouteNode` class, add a `params` property to store the names of dynamic parameters for each route.
2. **Route Addition Logic**: In the `#addRoute` method, check if a segment starts with a colon (`:`). If it does, treat it as a dynamic parameter and store its name in the `params` array of the current node.
3. **Route Finding Logic**: In the `findRoute` method, when a dynamic segment is encountered, extract the corresponding value from the URL and store it somewhere for later use.
4. **Example**:
   - Input: `"/users/:userId"`
   - Output: When the path `"/users/123"` is requested, the router should return `{ params: { userId: '123' }, ha`

6.13.3.3 Solution

Here's the updated implementation of the `TrieRouter` class with dynamic routing capabilities:

```
const HTTP_METHODS = {
  GET: "GET",
  POST: "POST",
  PUT: "PUT",
  DELETE: "DELETE",
  PATCH: "PATCH",
  HEAD: "HEAD",
  OPTIONS: "OPTIONS",
  CONNECT: "CONNECT",
  TRACE: "TRACE",
};

class RouteNode {
  constructor() {
    this.children = new Map();
    this.handler = new Map();
    this.params = [];
  }
}

class TrieRouter {
  constructor() {
```

```
    this.root = new RouteNode();
  }

  #verifyParams(path, method, handler) {
    if (typeof path !== "string" || path[0] !== "/") throw new Error("Malformed path provided.");
    if (typeof handler !== "function") throw new Error("Handler should be a function");
    if (!HTTP_METHODS[method]) throw new Error("Invalid HTTP Method");
  }

  #addRoute(path, method, handler) {
    this.#verifyParams(path, method, handler);

    let currentNode = this.root;
    let routeParts = path.split("/").filter(Boolean);
    let dynamicParams = [];

    for (const segment of routeParts) {
      if (segment.includes(" ")) throw new Error("Malformed `path` parameter");

      const isDynamic = segment[0] === ":";
      const key = isDynamic ? ":" : segment.toLowerCase();

      if (isDynamic) {
        dynamicParams.push(segment.substring(1));
      }

      if (!currentNode.children.has(key)) {
        currentNode.children.set(key, new RouteNode());
      }

      currentNode = currentNode.children.get(key);
    }

    currentNode.handler.set(method, handler);
    currentNode.params = dynamicParams;
  }
```

```
findRoute(path, method) {
  let segments = path.split("/").filter(Boolean);
  let currentNode = this.root;
  let extractedParams = [];

  for (let idx = 0; idx < segments.length; idx++) {
    const segment = segments[idx];

    let childNode = currentNode.children.get(segment.toLowerCase());
    if (childNode) {
      currentNode = childNode;
    } else if ((childNode = currentNode.children.get(":"))) {
      extractedParams.push(segment);
      currentNode = childNode;
    } else {
      return null;
    }
  }

  let params = Object.create(null);

  for (let idx = 0; idx < extractedParams.length; idx++) {
    let key = currentNode.params[idx];
    let value = extractedParams[idx];

    params[key] = value;
  }

  return {
    params,
    handler: currentNode.handler.get(method),
  };
}

get(path, handler) {
  this.#addRoute(path, HTTP_METHODS.GET, handler);
}
```

```
post(path, handler) {
  this.#addRoute(path, HTTP_METHODS.POST, handler);
}

put(path, handler) {
  this.#addRoute(path, HTTP_METHODS.PUT, handler);
}

delete(path, handler) {
  this.#addRoute(path, HTTP_METHODS.DELETE, handler);
}

patch(path, handler) {
  this.#addRoute(path, HTTP_METHODS.PATCH, handler);
}

head(path, handler) {
  this.#addRoute(path, HTTP_METHODS.HEAD, handler);
}

options(path, handler) {
  this.#addRoute(path, HTTP_METHODS.OPTIONS, handler);
}

connect(path, handler) {
  this.#addRoute(path, HTTP_METHODS.CONNECT, handler);
}

trace(path, handler) {
  this.#addRoute(path, HTTP_METHODS.TRACE, handler);
}

printTree(node = this.root, indentation = 0) {
  const indent = "-".repeat(indentation);

  node.children.forEach((childNode, segment) => {
```

```
      console.log(`${indent}(${segment}) Dynamic: ${childNode.params}`);
      this.printTree(childNode, indentation + 1);
    });
  }
}
```

Let's go through the updated code and understand the changes made to support dynamic routing in the `TrieRouter`.

6.13.3.4 Explanation

```
class RouteNode {
  constructor() {
    this.children = new Map();
    this.handler = new Map();

    // Initialize an array to store dynamic parameter names
    this.params = [];
  }
}
```

In the `RouteNode` class, we added a `params` array to store the names of dynamic parameters. This allows us to keep track of which segments of the route are dynamic (e.g., `:userId`) and helps in extracting their values when a matching route is found.

> Note, we're going to store the `params` on the leaf node of the route. This is because the dynamic parameters are only relevant when we reach the end of the route.

```
class TrieRouter {
  constructor() {
    // Create a root node upon TrieRouter instantiation
    this.root = new RouteNode();
  }
}
```

The `TrieRouter` constructor remains unchanged, still initializing a `root` node that serves as the entry point for route management.

```
#verifyParams(path, method, handler) {
  if (typeof path !== "string" || path[0] !== "/") throw new Error("Malformed path provided.");
  if (typeof handler !== "function") throw new Error("Handler should be a function");
  if (!HTTP_METHODS[method]) throw new Error("Invalid HTTP Method");
}
```

We introduced a new private method `#verifyParams` to validate the parameters passed to the `addRoute` method. This method checks that the `path` is a valid string starting with a slash, that the `handler` is a function, and that the `method` is a valid HTTP method. This way we separate the validation logic from the main route handling logic.

```
#addRoute(path, method, handler) {
  this.#verifyParams(path, method, handler);

  let currentNode = this.root;
  let routeParts = path.split("/").filter(Boolean);
  let dynamicParams = [];
}
```

In the `#addRoute` method, we call `#verifyParams` to validate the inputs. We also initialize an array `dynamicParams` to keep track of any dynamic segments in the route.

```
for (const segment of routeParts) {
  if (segment.includes(" ")) throw new Error("Malformed `path` parameter");

  const isDynamic = segment[0] === ":";
  const key = isDynamic ? ":" : segment.toLowerCase();

  if (isDynamic) {
    dynamicParams.push(segment.substring(1));
  }
```

As we iterate through each segment of the route, we check if it is dynamic by looking for a leading colon (`:`). If it is dynamic, we store the parameter name (without the colon) in the `dynamicParams` array. We also determine the key to use in the `children` Map: if it's dynamic, we use `":"` as the key; otherwise, we convert the segment to lowercase.

```
if (!currentNode.children.has(key)) {
  currentNode.children.set(key, new RouteNode());
```

```
}
```

If a child node for the current segment does not exist, we create a new `RouteNode` and add it to the `children` Map of the current node.

```
currentNode.handler.set(method, handler);
currentNode.params = dynamicParams;
```

After processing all segments, we set the handler for the route in the `handler` Map and store the dynamic parameters in the `params` array of the current node.

```
findRoute(path, method) {
  let segments = path.split("/").filter(Boolean);
  let currentNode = this.root;
  let extractedParams = [];

  ...
}
```

In the `findRoute` method, we initialize an array `extractedParams` to store values for any dynamic parameters. Note that we store the "keys" of the dynamic parameter in the `addRoute` method and the "values" in the `findRoute` method.

```
for (let idx = 0; idx < segments.length; idx++) {
  const segment = segments[idx];

  let childNode = currentNode.children.get(segment.toLowerCase());
  if (childNode) {
    currentNode = childNode;
  } else if ((childNode = currentNode.children.get(":"))) {
    extractedParams.push(segment);
    currentNode = childNode;
  } else {
    return null;
  }
}
```

As we iterate through the segments, we first check for a direct match in the `children` Map. If no match is found, we check for a dynamic route (using the `":"` key). If we find a dynamic route, we push the current

segment's value into `extractedParams`.

Note the syntax `(childNode = currentNode.children.get(":"))`. This is a common pattern in JavaScript to both check if a value is truthy and assign it to a variable in a single line. So,

```
let x = obj.get("key");
if (x) {
  // do something with x
}
```

can be written as

```
if ((x = obj.get("key"))) {
  // do something with x
}
```

Now, moving on to extracting the dynamic parameters:

```
let params = Object.create(null);

for (let idx = 0; idx < extractedParams.length; idx++) {
  let key = currentNode.params[idx];
  let value = extractedParams[idx];

  params[key] = value;
}
```

After traversing the route, we create an object `params` to store the extracted dynamic parameters. We loop through `extractedParams` and map each value to its corresponding key from `currentNode.params`.

```
return {
  params,
  handler: currentNode.handler.get(method),
};
```

Finally, we return an object containing the extracted parameters and the corresponding handler for the route.

6.13.3.5 Testing the implementation

I've also updated the `printTree` method to include dynamic parameters in the output. You can use this method to visualize the trie structure and verify that dynamic routes are correctly stored.

```
const trieRouter = new TrieRouter();
trieRouter.get("/users/:id/hello/there/:some/:hello", function get1() {});
trieRouter.post("/users/:some/hello/there/:id/none", function post1() {});

console.log("Printing Tree:");
trieRouter.printTree();

console.log("Finding Handlers:");

console.log(trieRouter.findRoute("/users/e/hello/there/2/3", HTTP_METHODS.GET));
console.log(trieRouter.findRoute("/users/1/hello/there/2/none", HTTP_METHODS.GET));
console.log(trieRouter.findRoute("/users", HTTP_METHODS.PUT));
console.log(trieRouter.findRoute("/users", HTTP_METHODS.TRACE));
```

This outputs:

```
Printing Tree:
(users) Dynamic:
-(:) Dynamic:
--(hello) Dynamic:
---(there) Dynamic:
----(:) Dynamic:
-----(:) Dynamic: id,some,hello
-----(none) Dynamic: some,id

Finding Handlers:
{
  params: [Object: null prototype] { id: 'e', some: '2', hello: '3' },
  handler: [Function: get1]
}
{
  params: [Object: null prototype] { some: '1', id: '2' },
  handler: undefined
}
{ params: [Object: null prototype] {}, handler: undefined }
{ params: [Object: null prototype] {}, handler: undefined }
```

That's it! We now have a `TrieRouter` that supports dynamic routing, allowing us to define routes with

variable segments and extract their values when a route is matched.

6.13.4 Summary

In this challenge, we successfully enhanced the `TrieRouter` class to support dynamic routing. The router can now handle routes with dynamic parameters, extract their values from the URL, and provide them to the corresponding route handler. This functionality is essential for building flexible and scalable web applications.

6.14 Running Our Server

We have a working implementation of our `TrieRouter` class, which has enough functionality to handle routing for our server. But, we don't have a way to run our server or listen for requests yet.

In this chapter, we will implement a `run` function that accepts a router (`TrieRouter`) and a port (`number`) argument. This function will serve as the entry point of our server code, allowing us to handle incoming requests and route them to the appropriate handlers defined in our router.

But before that, let's refactor our code a little bit. We're going to rename the `TrieRouter` class to `Router`. Secondly, we're going to add JSDoc comments to all the methods in the `Router` class, to make our lives easier with the auto-completion and documentation.

6.14.1 Refactoring the `TrieRouter` class

```
const HTTP_METHODS = { ... } // Remains unchanged

class RouteNode {
  constructor() {
    /** @type {Map<String, RouteNode>} */
    this.children = new Map();

    /** @type {Map<String, Function>} */
    this.handler = new Map();

    /** @type {Array<String>} */
    this.params = [];
  }
}
```

```
class Router {
  constructor() {
    /** @type {RouteNode} */
    this.root = new RouteNode();
  }

  /**
   * @param {String} path
   * @param { 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT' |
↪  'TRACE' } method
   * @param {Function} handler
   */
  #verifyParams(path, method, handler) { ... }


  /**
   * @param {String} path
   * @param { 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT' |
↪  'TRACE' } method
   * @param {Function} handler
   */
  #addRoute(path, method, handler) { ... }

  /**
   * @param {String} path
   * @param { 'GET' | 'POST' | 'PUT ' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT ' |
↪  'TRACE ' } method
   * @returns { { params: Object, handler: Function } | null }
   */
  findRoute(path, method) { ... }

   /**
   * @param {String} path
   * @param {Function} handler
   */
  get(path, handler) {
```

```
    this.#addRoute(path, HTTP_METHODS.GET, handler);
  }

  /** For POST, PUT, DELETE, PATCH, HEAD, OPTIONS, CONNECT, TRACE, we are going to re-use the
   ↪  same JSDoc comment as `get` method */

  /**
   * @param {RouteNode} node
   * @param {number} indentation
   */
  printTree(node = this.root, indentation = 0) { ... }
}
```

As you can see, we're repeating this JSDoc comment for a couple of methods:

```
/**
 * @param {String} path
 * @param { 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT' |
 ↪  'TRACE' } method
 * @param {Function} handler
 */
```

This is not a good practice. Instead we can make use of something called **Type Aliases**.

Type aliases are a way to give a type a name, just like a variable. Each type **should be** distinct. They are exactly the same as the original type, but with a different name. This is useful when you want to refer to the same type multiple times and don't want to repeat the same type definition.

6.14.2 Type Aliases

Let's add a type alias for the HTTP methods:

```
/**
 * @typedef { 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT' |
 ↪  'TRACE' } HttpMethod
 */
```

And, to use this type alias in our JSDoc comments:

```
/**
* @param {String} path
* @param {HttpMethod} method
* @param {Function} handler
*/
#verifyParams(path, method, handler) { ... }

/**
 * @param {String} path
 * @param {HttpMethod } method
 * @param {Function} handler
 */
#addRoute(path, method, handler) { ... }

/** And so on... */
```

But there's a small issue. We have defined the `HttpMethod` type alias in the same file as the `Router` class. This will make it impossible to use the `HttpMethod` type alias in other files. To fix this, we can create a new file `globals.js` and move all the global type aliases here.

```
// file: globals.js

/**
 * @typedef { 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT' |
 ↪  'TRACE' } HttpMethod
 */
```

Now you'd be able to use the `HttpMethod` type alias in any file. That's enough refactoring for now.

6.14.3 The `run` function

```
const { createServer } = require("node:http");

/**
 * Run the server on the specified port
 * @param {Router} router - The router to use as the main request handler
 * @param {number} port - The port to listen on
 */
```

```
function run(router, port) {
  if (!(router instanceof Router)) {
    throw new Error("`router` argument must be an instance of Router");
  }

  if (typeof port !== "number") {
    throw new Error("`port` argument must be a number");
  }

  createServer(function _create(req, res) {
    const route = router.findRoute(req.url, req.path);

    if (route?.handler) {
      req.params = route.params;
      route.handler(req, res);
    } else {
      res.writeHead(404, null, { "content-length": 9 });
      res.end("Not Found");
    }
  }).listen(port);
}
```

Let's go through the code line by line:

```
const { createServer } = require("node:http");
```

We're importing the `createServer` function that the `node:http` module provides. This function is used to create an HTTP server that listens for requests on a specified `port`. We already created a demo server using `http.createServer()` in HTTP Deep Dive.

```
function run(router, port) { ... }
```

This is going to be the main entry point of our library. The `run` function is accepts two arguments: `router` and `port`. The `router` will be an instance of the `Router` class that we defined earlier, and the `port` will be the port number on which the server will listen for incoming requests.

```
if (!(router instanceof Router)) {
  throw new Error("`router` argument must be an instance of Router");
}
```

```
if (typeof port !== "number") {
  throw new Error("`port` argument must be a number");
}
```

Some basic type checking to ensure that the `router` argument is an instance of the `Router` class and the `port` argument is a number.

```
createServer(function _create(req, res) { ... }).listen(port);
```

We're creating an HTTP server using the `createServer` function. To re-iterate, the `createServer` function takes a single argument*, which is a callback function. The callback function will receive two arguments: `req` (the `Http.IncomingMessage` object) and `res` (the `Http.ServerResponse` object).

```
const route = router.findRoute(req.url, req.path);

if (route?.handler) {
  req.params = route.params;
  route.handler(req, res);
} else {
  res.writeHead(404, null, { "Content-Length": 9 });
  res.end("Not Found");
}
```

We're calling the `findRoute` method on the `router` object to find the route that matches the incoming request. The `findRoute` method will return an object with two properties: `handler` and `params`. If a route is found, we'll call the `handler` function with the `req` and `res` objects. If no route is found, we'll return a `404 Not Found` response.

Inside the `if` statement, we're attaching a new property `req.params` to the `req` object. This property will contain the parameters extracted from the URL. The client can easily access the parameters using `req.params`.

You might have noticed that we're using a hard-coded `Content-Length` header with a value of `9`. This is because, if we do not specify the `Content-Length` header, the response headers will include a header `Transfer-Encoding: chunked`, which has a performance impact. We discussed about this in a previous chapter - Chunks, oh no!

That's it! We have implemented the `run` function, which will allow us to run our server and listen for incoming requests. In the next chapter, we'll implement a simple server using our `Router` class and the `run` function.

*The callback function has multiple overloads, i.e it has a couple more function signatures. But for now, we're only interested in the one that takes a single callback function.

6.15 Building our first web-server

Our `Router` implementation has enough functionality to handle basic HTTP requests. In this chapter, we're going to spin up our first web server with this little toy `Router` and put it to test.

6.15.1 More refactoring

Till now, our entire `Router` implementation and the helper function stayed in a single file. As we're going to build a web server, it's a good idea to separate the `Router` implementation into its own module, as well as the helper functions.

Here's the updated file structure:

```
🗁./
├── </>lib/                # Our library code
│   ├── JS constants.js    # Constants used in our library
│   ├── JS index.js        # Entry point of our library
│   └── JS router.js       # Router implementation
├── JS globals.js          # Global Typedefs
└── JS test.js             # We'll write our code for testing here.
```

6.15.1.1 `lib/router.js`

```
const { HTTP_METHODS } = require("./constants");

class RouteNode {
  constructor() {
    /** @type {Map<String, RouteNode>} */
    this.children = new Map();

    /** @type {Map<String, RequestHandler>} */
    this.handler = new Map();
```

```
    /** @type {Array<String>} */
    this.params = [];
  }
}

class Router {
  constructor() {
    /** @type {RouteNode} */
    this.root = new RouteNode();
  }

  /**
   * @param {String} path
   * @param {HttpMethod} method
   * @param {RequestHandler} handler
   */
  #verifyParams(path, method, handler) {
    if (typeof path !== "string" || path[0] !== "/") throw new Error("Malformed path provided.");
    if (typeof handler !== "function") throw new Error("Handler should be a function");
    if (!HTTP_METHODS[method]) throw new Error("Invalid HTTP Method");
  }

  /**
   * @param {String} path
   * @param {HttpMethod } method
   * @param {RequestHandler} handler
   */
  #addRoute(path, method, handler) {
    this.#verifyParams(path, method, handler);

    let currentNode = this.root;
    let routeParts = path.split("/").filter(Boolean);
    let dynamicParams = [];

    for (const segment of routeParts) {
      if (segment.includes(" ")) throw new Error("Malformed `path` parameter");
```

```
      const isDynamic = segment[0] === ":";
      const key = isDynamic ? ":" : segment.toLowerCase();

      if (isDynamic) {
        dynamicParams.push(segment.substring(1));
      }

      if (!currentNode.children.has(key)) {
        currentNode.children.set(key, new RouteNode());
      }

      currentNode = currentNode.children.get(key);
    }

    currentNode.handler.set(method, handler);
    currentNode.params = dynamicParams;
  }

  /**
   * @param {String} path
   * @param {HttpMethod} method
   * @returns { { params: Object, handler: RequestHandler } | null }
   */
  findRoute(path, method) {
    let segments = path.split("/").filter(Boolean);
    let currentNode = this.root;
    let extractedParams = [];

    for (let idx = 0; idx < segments.length; idx++) {
      const segment = segments[idx];

      let childNode = currentNode.children.get(segment.toLowerCase());
      if (childNode) {
        currentNode = childNode;
      } else if ((childNode = currentNode.children.get(":"))) {
        extractedParams.push(segment);
        currentNode = childNode;
```

```
    } else {
      return null;
    }
  }

  let params = Object.create(null);

  for (let idx = 0; idx < extractedParams.length; idx++) {
    let key = currentNode.params[idx];
    let value = extractedParams[idx];

    params[key] = value;
  }

  return {
    params,
    handler: currentNode.handler.get(method),
  };
}

/**
 * @param {String} path
 * @param {RequestHandler} handler
 */
get(path, handler) {
  this.#addRoute(path, HTTP_METHODS.GET, handler);
}

/**
 * @param {String} path
 * @param {RequestHandler} handler
 */
post(path, handler) {
  this.#addRoute(path, HTTP_METHODS.POST, handler);
}

/**
```

```
 * @param {String} path
 * @param {RequestHandler} handler
 */
put(path, handler) {
  this.#addRoute(path, HTTP_METHODS.PUT, handler);
}

/**
 * @param {String} path
 * @param {RequestHandler} handler
 */
delete(path, handler) {
  this.#addRoute(path, HTTP_METHODS.DELETE, handler);
}

/**
 * @param {String} path
 * @param {RequestHandler} handler
 */
patch(path, handler) {
  this.#addRoute(path, HTTP_METHODS.PATCH, handler);
}

/**
 * @param {String} path
 * @param {RequestHandler} handler
 */
head(path, handler) {
  this.#addRoute(path, HTTP_METHODS.HEAD, handler);
}

/**
 * @param {String} path
 * @param {RequestHandler} handler
 */
options(path, handler) {
  this.#addRoute(path, HTTP_METHODS.OPTIONS, handler);
```

```
  }

  /**
   * @param {String} path
   * @param {RequestHandler} handler
   */
  connect(path, handler) {
    this.#addRoute(path, HTTP_METHODS.CONNECT, handler);
  }

  /**
   * @param {String} path
   * @param {RequestHandler} handler
   */
  trace(path, handler) {
    this.#addRoute(path, HTTP_METHODS.TRACE, handler);
  }

  /**
   * @param {RouteNode} node
   * @param {number} indentation
   */
  printTree(node = this.root, indentation = 0) {
    const indent = "-".repeat(indentation);

    node.children.forEach((childNode, segment) => {
      console.log(`${indent}(${segment}) Dynamic: ${childNode.params}`);
      this.printTree(childNode, indentation + 1);
    });
  }
}

module.exports = Router;
```

6.15.1.2 lib/constants.js

```
const HTTP_METHODS = Object.freeze({
  GET: "GET",
  POST: "POST",
  PUT: "PUT",
  DELETE: "DELETE",
  PATCH: "PATCH",
  HEAD: "HEAD",
  OPTIONS: "OPTIONS",
  CONNECT: "CONNECT",
  TRACE: "TRACE",
});

module.exports = {
  HTTP_METHODS,
};
```

6.15.1.3 lib/index.js

```
const { createServer } = require("node:http");
const Router = require("./router");

/**
 * Run the server on the specified port
 * @param {Router} router - The router to use for routing requests
 * @param {number} port - The port to listen on
 */
function run(router, port) {
  if (!(router instanceof Router)) {
    throw new Error("`router` argument must be an instance of Router");
  }
  if (typeof port !== "number") {
    throw new Error("`port` argument must be a number");
  }

  createServer(function _create(req, res) {
```

```
    const route = router.findRoute(req.url, req.method);

    if (route?.handler) {
      req.params = route.params || {};
      route.handler(req, res);
    } else {
      res.writeHead(404, null, { "content-length": 9 });
      res.end("Not Found");
    }
  }).listen(port);
}

module.exports = { Router, run };
```

6.15.1.4 `globals.js`

```
/**
 * @typedef { 'GET' | 'POST' | 'PUT' | 'DELETE' | 'PATCH' | 'HEAD' | 'OPTIONS' | 'CONNECT' |
 ↪  'TRACE' } HttpMethod
 */

/**
 * @typedef {import("http").RequestListener} RequestHandler
 */
```

We've added a new typedef, i.e `RequestHandler` in `globals.js`. This typedef is used to define the type of the handler function that we pass to the `Router` instance. By default, we're using the `RequestListener` type from the `http` module, which is the type of the handler function that the `http.createServer` function expects.

6.15.2 Your first web server

Now that we've refactored our code, it's time to jump in and put our little project to test. Let's create a simple web server that listens on port `3000` and has a couple of endpoints.

```
"GET /"                          → Hello from the root endpoint
"GET /hello/:name"               → Hello, {name}!
"GET /user/:age/class/:subject"  → You're {age} years old and you're studying {subject}.
```

Here's the code for the web server:

```
// Get the `Router` and `run` function from our library
const { Router, run } = require("./lib");

// Create a new instance of the `Router` class
const router = new Router();

// Define the routes
router.get("/", (req, res) => {
  res.end("Hello from the root endpoint");
});

router.get("/user/:name", (req, res) => {
  res.end(`Hello, ${req.params.name}!`);
});

router.get("/user/:age/class/:subject", (req, res) => {
  res.end(`You're ${req.params.age} years old, and you're studying ${req.params.subject}.`);
});

// Start the server at port 3000
run(router, 3000);
```

To test our server, we'll make some cURL requests from the terminal.

```
$ curl http://localhost:3000
Hello from the root endpoint

$ curl http://localhost:3000/user/Ishtmeet
Hello, Ishtmeet!

$ curl http://localhost:3000/user/21/class/Mathematics
You're 21 years old, and you're studying Mathematics.
```

Everything looks good! Our server is up and running, and it's handling the requests as expected.