Production grade APIs

Production Grade APIs

How to understand, build and maintain APIs for production

fredrik.no@gmail.com

This book is for sale at http://leanpub.com/theultimateguidetoapis

This version was published on 2022-09-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Acknowledgements

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Chapter One - Foundations

The hypertext transfer protocol (HTTP)

One of the most fundamental concepts we'll learn about in this book is the hypertext transfer protocol.

It is foundational to the core of not only the discussion we'll have on APIs later, but to the core of the internet itself.

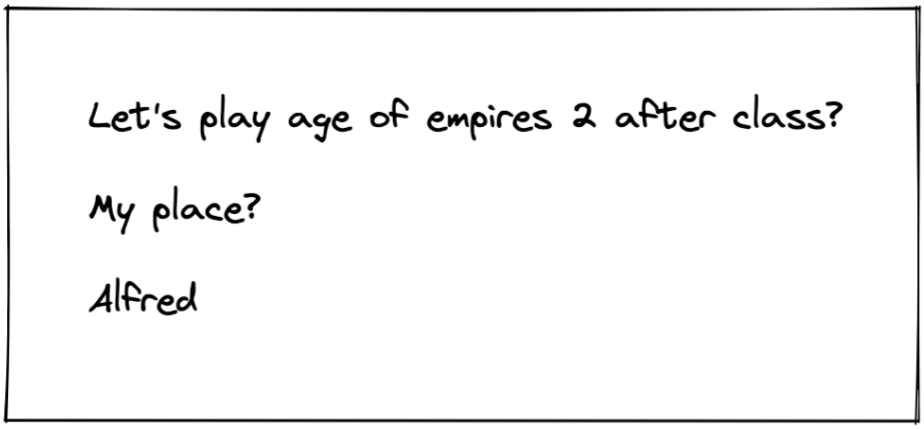
HTTP is about how communication happens on the internet. Let's explore what that means.

HTTP Basics

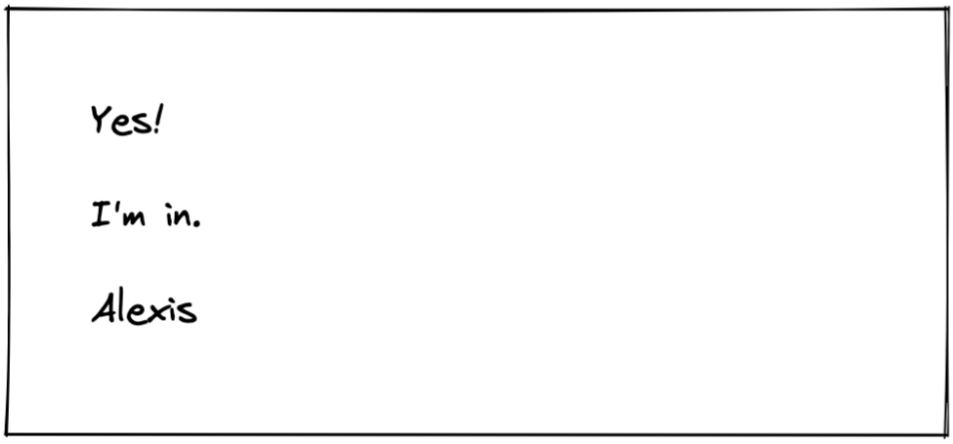
> If you're a visual learner here is an accompanying video[a]
>
> [a] https://youtu.be/0ykAOzJb-U8

Did you ever pass notes back to each other back in school class? You know, when your attention was supposed to be on the lecturer, but you were more interested in knowing if your friend wanted to play games later?

Maybe you slipped your friend a note asking if he or she wanted to meet up and play computer games after class. Maybe it looked something like this:



And maybe they responded like this:

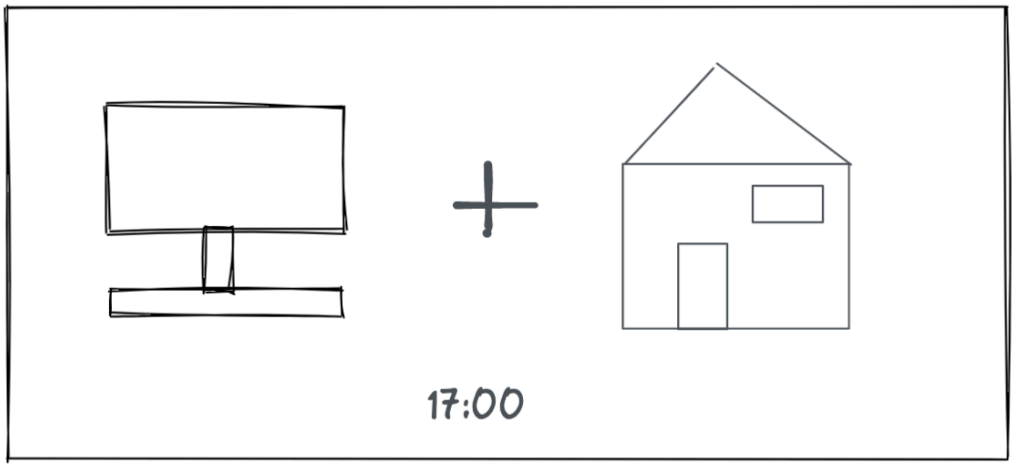


Awesome. Now you have a pattern for inviting your friends to play video games. Everything is fun and games for a while, but then your parents decide to move to Japan.

Your new class doesn't understand your English handwriting, and you don't know how to write or

speak in Japanese yet. Damn.

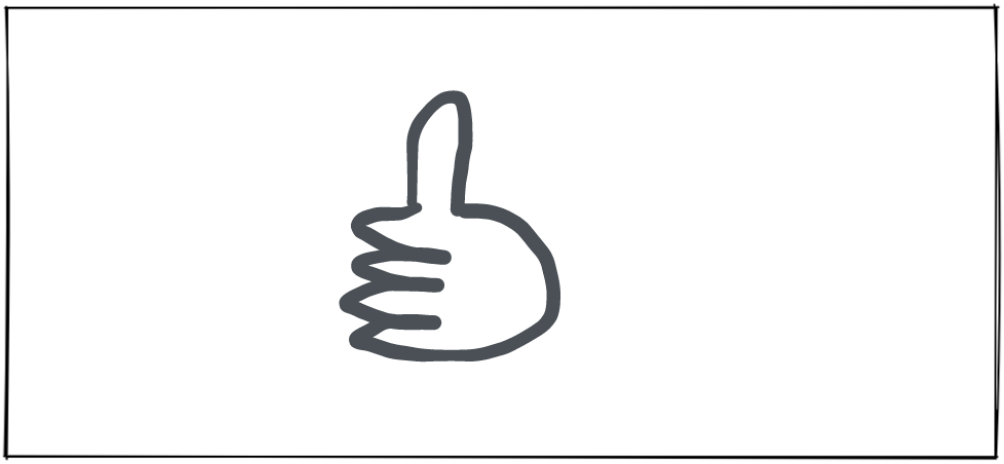
After some time, you come up with an idea. Instead of language, maybe you can draw something that can represent playing games. You come up with something like this:



You then spend the rest of the day using google translate to explain the concept of the note to one of your classmates.

He enthusiastically nods and helps to explain the concept to the rest of the class.

In your next class, you pass a bunch of these notes out and you receive notes like this in return:



Fantastic. You have created your own protocol. You sent a request and you received a response. This is important, and we'll get back to why. But first, take a moment to reflect over what it means to have established a protocol.

The protocol is a set of rules that governs your communication. Having the protocol is useful because once people know the protocol, everyone can communicate on the same terms.

If your protocol became a global phenomenon, you'd never have to worry about moving again.

If everywhere you went, people used your video game hangout protocol (VGHP) you'd never have any trouble inviting friends for video gaming sessions ever again.

The protocol would allow you to communicate with anyone, even if you didn't know the language.

Alright. So back to HTTP. The above example is contrived, but there's still something valuable to take away from this example.

One of the most important things to take away is that a protocol will allow us to communicate if we just learn the protocol, it will be the same in London, Tokyo, USA, Norway, Sweden and any other country in the world.

We don't need to know the people who implement the protocol, we just need to know the protocol itself.

The other important take-away from this example is the realization that the internet is nothing but HTTP text messages flying back and forth implementing the HTTP protocol.

Yes. Text messages.

HTTP messages across the internet

The anatomy of a HTTP Message

Realizing that the internet is nothing but text messages flying back and forth might be disturbing. After all, what do these text messages look like, and how come we never see them?

We'll get to that.

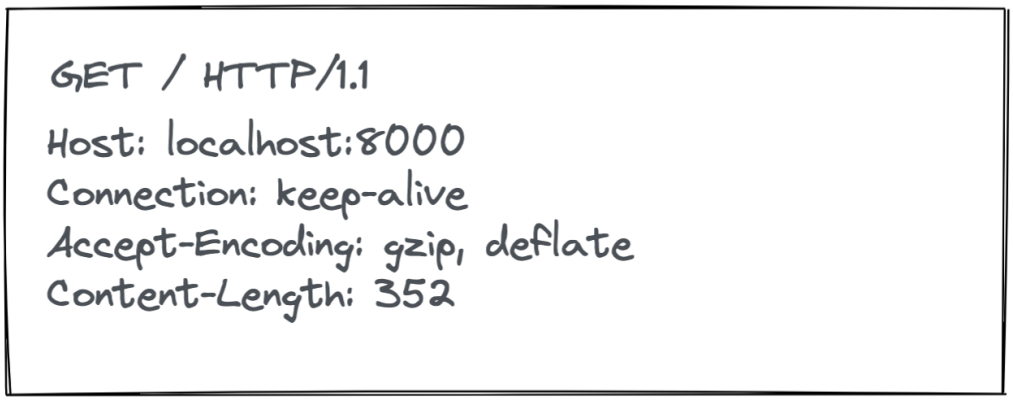
But before we do, let's take a look at the anatomy of a HTTP message. There are two types of messages. One is an HTTP Request message, and the other is an HTTP Response message.

HTTP Request message

If you're a visual learner, here is an accompanying video[a]

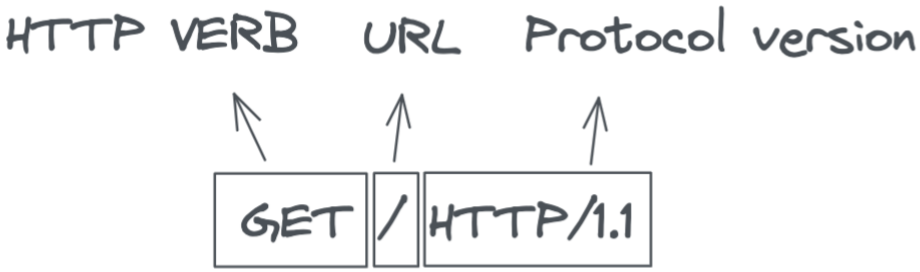
[a] https://youtu.be/8QfaJudvpmo

The HTTP Request message looks like this:



HTTP Request

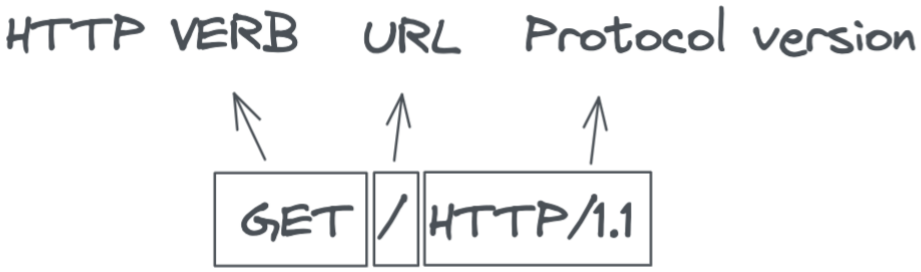
The first line, also known as the start-line, starts with an HTTP Verb. In this case the HTTP verb GET, then the verb is followed by an URL location and finally the protocol version.



HTTP Request status line

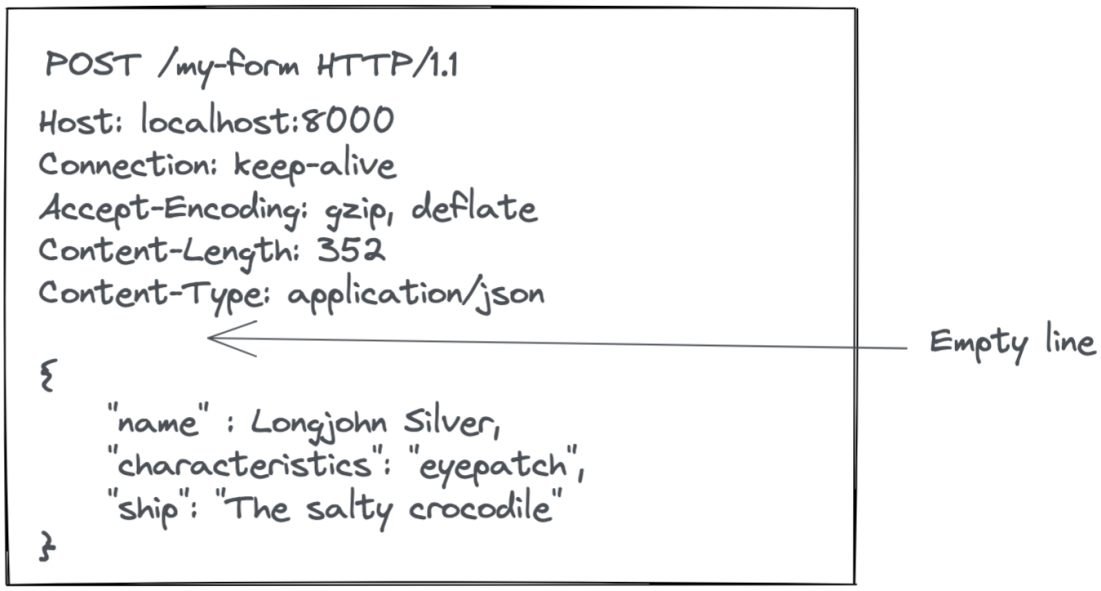
The next part of the message consists of HTTP headers. These are optional and can enrich the HTTP message with information that can be beneficial to the receiver.

HTTP headers consist of a case-insensitive string, followed by a colon, and the value of the HTTP header constricted to a single line.



HTTP headers

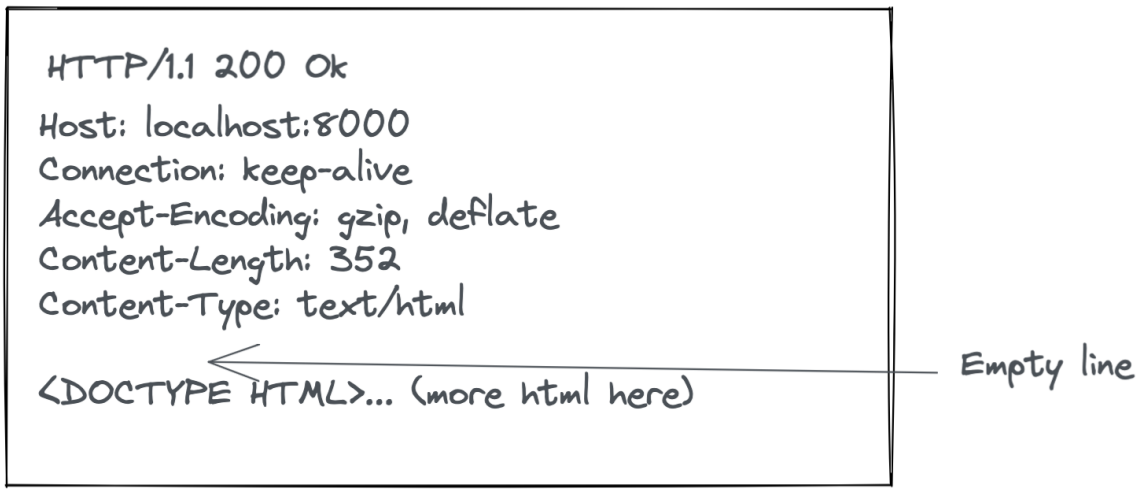
The final part of the HTTP request message might include an optional body. The body is separated from the rest of the form with an empty space.



HTTP body

HTTP Response message

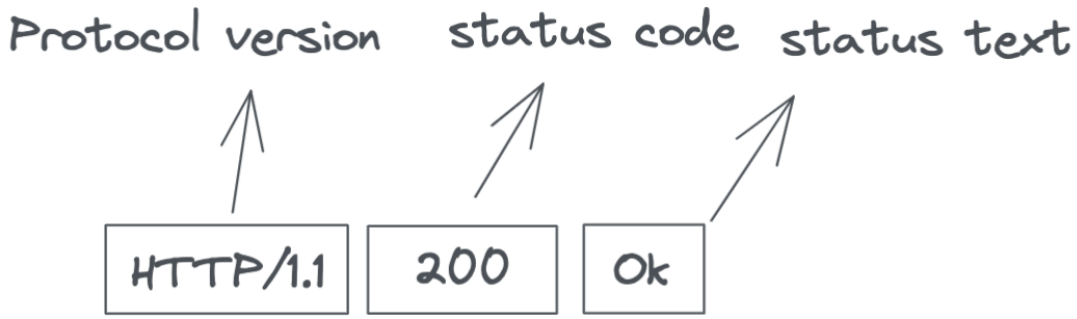
The HTTP response message also has a strict structure. It looks like this:



HTTP response

The status line in the HTTP request message is represented by the first line and consists of the protocol version, status code and status text:

The message then contains optional headers and body, with the body also separated by an empty line.



HTTP response status line

The HTTP request response cycle

Now that we've seen the structure of the HTTP request and response messages, take another second to consider that these messages represent the backbone of the internet.

It's simply this. You could write these yourself, just as you write a text message on your phone and send it. So why don't you try?

Using a tool called netcat we can open a connection to a server and send raw HTTP messages. First, make sure you have netcat installed.

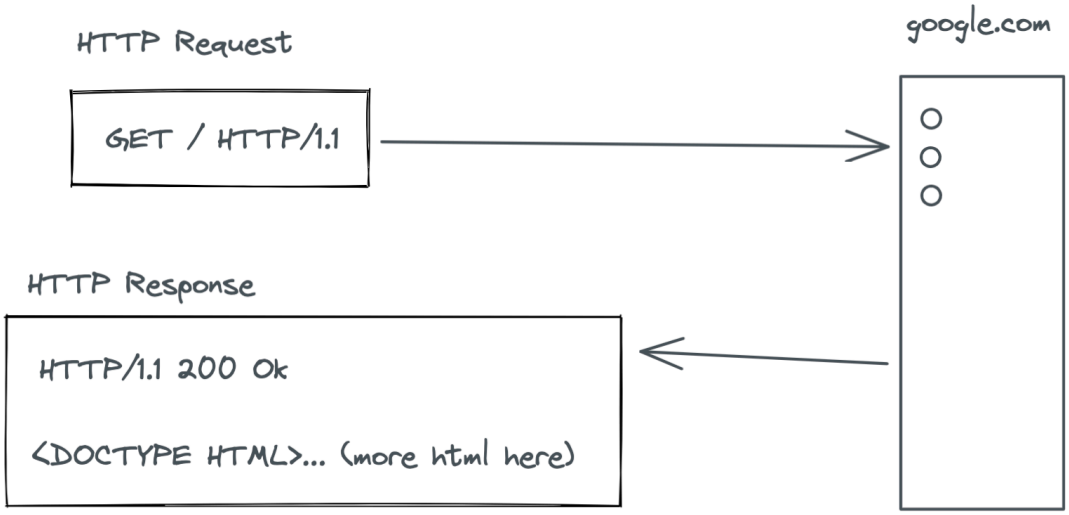
If you are on MacOS, netcat will be installed by default. Otherwise you might need to install it (on windows, you can use ncat[1])
Follow these steps:

- Open your terminal
- Type in `nc google.com 80`
- Type in `GET / HTTP/1.1`
- Hit enter twice

You should see the returned HTTP response message from google.com including the HTML body content that represents Google's homepage.

IT'S JUST TEXT. IT'S LAUGHABLY SIMPLE.

That is all. You send text messages, and you get text messages in return. Yet the abstractions on top rarely show us this part, you have to discover it on your own.



HTTP response status line

[1] https://nmap.org/ncat/

The value of predictability

According to what you've just learned (or perhaps already knew) about HTTP messages, they follow a strict format.

The messages will be the same every time, with the optional content subject to change (headers and body). This gives us predictability, and predictability is a very very good thing in computer programming.

It allows us to know ahead of time what the format is and exactly what we receive, and with that knowledge we can write code that parses an HTTP message into a format we can work with in code.

As they say, clear concise and clean code speaks more than a thousand words, so let's look at how we could structure such a function to parse an incoming HTTP request.

```
const parseRequest = (message) => {
    // validate HTTP message
    const validMessage = validateMessage(message)

    if (!validMessage) return;

    const status = getStatus(message);
    const headers = getHeaders(message);
    const body = getBody(message);

    return {
      ...status,
      headers,
      body
    }
}

const getStatus = (message) => {
    const fragments = message.split("\n");
    const statusLine = fragments[0];

    const statusLineFragments = statusLine.split(" ");

    return {
       method: statusLineFragments[0],
       path: statusLineFragments[1],
       protocol: statusLineFragments[2],
    }
}
```

The above function will yield an object that extracts the values of the HTTP message status line. We can do the same with the HTTP headers section and body, and once we have that information in a data structure we can work with within our programming language - then we can act on this information. We'll see how later in this chapter.

HTTP Methods

We'll briefly mention HTTP methods (or verbs) here. It will become clearer later in this book why these are useful. Simply put, a HTTP verb provides information about what type of action the HTTP request would like to perform.

This verb is always a part of the status line of the HTTP request, and you have already seen one of these verbs. The GET verb:

```
1 GET / HTTP/1.1
```

- GET

Should be used when you want to retrieve data

- POST

Should be used when you want to send data or change something on the server

- PUT

Should be used when you want to change or replace data on the server

- DELETE

Should be used when you want to delete data from the server

- PATCH

Should be used when you want to partially update data on the server

We also have:

- CONNECT
- OPTIONS
- HEAD
- TRACE

But we'll skip these for now, you can read more about them at mozilla developer network[2]

[2] https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods

HTTP Response codes

Where HTTP methods helps give context to the action you want to perform on the request. HTTP response codes helps provide information about the result of the action in the HTTP response. We have already seen once such code in the first line of the HTTP response:

```
1 HTTP/1.1 200 OK
```

These codes give information back to the sender of the HTTP request about the result of the request. We have five main classes of response codes:

- Informational responses (100-199)
- Successful responses (200-299)
- Redirection responses (300-399)
- Client error responses (400-499)
- Server error responses (500-599)

We won't go through all of these codes, but we should know some of them, because they (1) give us valuable information when sending HTTP requests about what potentially went wrong, (2) it will help us correctly write our own API once, and (3) we will have a fuller understanding of how the request/response cycle works if we understand the responses.

With that said, here are some good status codes to know. The below descriptions include the status code and the status text:

- 200 OK - Request was successful
- 201 Created - Request was successful and a resource was created
- 204 No Content - Request was successful, but there was no content to return
- 301 Moved Permanently - The resource you are trying to access has moved, and the new URL should be returned in the response
- 304 Not Modified - The resource you are trying to access has not changed, and you can continue using the cache
- 400 Bad Request - The incoming HTTP request includes errors and the server won't process the request
- 401 Unauthorized - The incoming HTTP request is not authenticated
- 403 Forbiddden - The incoming HTTP request caller is authenticated, but is not allowed to access this resource
- 404 Not Found - The resource could not be found
- 418 I'm a teapot - The server refuses the attempt to brew coffee with a teapot. Yes, this is a real status code. Who said devs don't have a sense of humor?
- 500 Internal Server Error - The server has encountered a situation it doesn't know how to handle

You can view all HTTP status codes here on MDN[3]

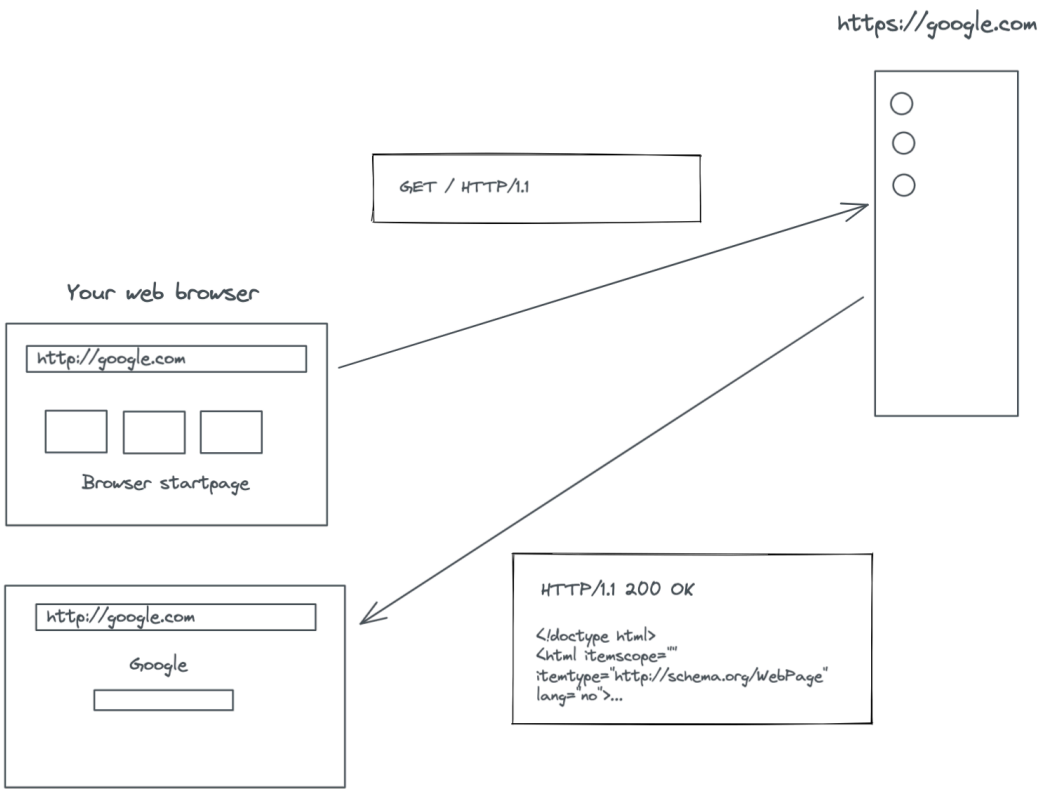
[3] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

The internet postman

Now that we have a foundation for understanding the way we communicate on the internet. We'll need to understand two more pieces of the puzzle. Who sends HTTP requests and who responds with HTTP responses?

There are many sources of HTTP requests, but one of the most common and easiest to understand might be a tool that you use every time you browse the internet. The web browser.

Let's have a look at what happens when you visit a page on the internet.



Browser communication

When you type in a URL in your web browser, the browser takes that URL and creates a HTTP request on your behalf. This HTTP request is then sent to the recipient web server, which will read the request and formulate a response message.

In this case, the request is accepted and we get an HTTP response which includes HTML in the response body. The browser reads the response, extracts the HTML and generates the page you know as the google homepage. Simple right?

Everyone who uses the internet today uses this modern postman to pass along a huge number of

HTTP messages every day. It's abstracted away from us as end users, but nevertheless it's there, making sure we can access the information we need on the world wide web.

It's a useful abstraction, no doubt the internet would not have as successful as it has been without it. But under the surface of this abstraction lies a wealth of valuable knowledge.

And if we only dive a little deeper, we'll access a level of understanding that will enable you not only to write production ready APIs, but understand what an API is at it's core. And that is a powerful thing.

The webserver

We'll close this chapter with most valuable information. If you only take one concept away from this book. Let it be this one.

This is the key.

Before we get started let's sum up what we already know:

- Internet communication is performed over HTTP
- HTTP is made up of requests and responses
- These messages follow a strict format
- A client typically sends a request
- A webserver typically receives the request and formulates a response

So far we've explored the 4 preceding steps. We know the format of HTTP messages, we know that browsers create these messages for us and dispatches them to the webserver. In this section we'll take deep dive into step 5.

And what better way to do that than to code a simple webserver of our own?

Here is an example of how to setup a basic web server with express.js[4], a popular javascript web server framework:

```
1  const express = require('express')
2  const app = express()
3  const port = 3000
4
5  app.get('/', (req, res) => {
6    res.send('Hello World!')
7  })
8
9  app.listen(port, () => {
10   console.log(`Example app listening on port ${port}`)
11 })
```

[4] https://expressjs.com/

In the following section we'll aim to create something similar to this from scratch, incorporating the concepts of the previous sections as we go along.

For reference, the full project can be found here:

- https://github.com/FredrikOseberg/simple-webserver

Getting the HTTP request message content

Previously we saw how to extract the status from a HTTP request message. Let's expand on that and see how we could parse a full message:

```
class HTTPParser {
  parse = (message) => {
    // todo: validate message

    const status = this.getStatus(message);
    const headers = this.getHeaders(message);
    const body = this.getBody(message);

    return {
      ...status,
      headers,
      body,
    };
  };

  getStatus = (message) => {
    // validated message incoming
    const fragments = message.split('\n');
    const statusLine = fragments[0];

    const statusLineFragments = statusLine.split(' ');

    return {
      method: statusLineFragments[0],
      path: statusLineFragments[1],
      protocol: statusLineFragments[2],
    };
  };

  getHeaders = (message) => {
    const fragments = message.split('\n');
```

```
32
33      const index = fragments.findIndex((elem) => elem === '');
34      let headerFragments;
35
36      if (index > 0) {
37        headerFragments = fragments.slice(1, index);
38      } else {
39        headerFragments = fragments.slice(1);
40      }
41
42      const headers = headerFragments.reduce((acc, curr) => {
43        const [key, value] = curr.split(':');
44
45        if (key && value) {
46          acc[key.trim()] = (value && value.trim()) || '';
47        }
48        return acc;
49      }, {});
50
51      return headers;
52    };
53
54    getBody = (message) => {
55      const fragments = message.split('\n');
56
57      const index = fragments.findIndex((elem) => elem === '');
58      if (index > 0) {
59        return fragments
60          .slice(index + 1)
61          .join('\n')
62          .trim();
63      }
64
65      return '';
66    };
67  }
68
69  module.exports = HTTPParser;
```

Creating the web server

> If you're a visual learner here is an accompanying video[a]
>
> [a] https://youtu.be/d41DbbeYe2M

Now we have something that can break down a HTTP request message into a javascript object containing the HTTP method, path, protocol, headers, and body.

With this in place, we should be able to take an incoming message a parse it into something we can work with in the programming language. Similar to the `req` (short for request) object in the express framework.

Remember how that looked like in the express framework?

```
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})
```

What is happening here is that we are instructing the webserver that when it receives an incoming HTTP request that matches the `GET` verb on the root path `/` we want to execute a function. In other words, when we receive a HTTP message like this:

```
GET / HTTP/1.1
```

The function we define will be called to handle the request. This function receives two arguments:

```
(req, res) => {
  res.send('Hello World!')
}
```

The request object is similar to the one returned by our HTTP parser. It's just an object containing the information that the HTTP message contained.

The response object here contains methods that will be used to construct a HTTP response message. In this instance the send method will add `'Hello world!'` to the response body.

We will get back to this shortly, for now let's have a look at what our main code will look like:

```
const Router = require('./router');
const HTTPParser = require('./http-parser');
const Response = require('./response');
const Request = require('./request');

const net = require('net');

class SimpleServer extends Router {
  constructor() {
    super();
    this.httpParser = new HTTPParser();
    this.server = net.createServer((connection) => {
      connection.on('data', (data) => {
        const httpMessage = data.toString();
        const requestObject = this.httpParser.parse(httpMessage);

        const req = new Request(requestObject);
        const res = new Response(connection, 'HTTP/1.1');

        const handler = this.getRoute(req.method, req.path);

        if (typeof handler === 'function') {
          handler(req, res);
        } else {
          connection.write(`HTTP/1.1 404 Not found\r\n`);
          connection.end();
        }
      });
    });
  }

  listenAndServe = (port) => {
    this.server.listen(port, () => {
      console.log(`Listening for connections on ${port}`);
    });
  };
}

module.exports = SimpleServer;
```

Alright, so there are a few things going on here. Note that that we are instantiating two properties here: (1) is the HTTPParser we just saw, that we'll use to parse the incoming message into a workable javascript object and (2) is the server property.

The server property uses the NodeJS package `net` to create a TCP connection. Inside the `createServer` call we set up an event listener that will trigger whenever the server receives data.

In the event listener, the data will be converted to a string and we pass this data to the parse method of the HTTPParser, getting back a request object.

We then create a new Response and Request with this data, and retrieve the route from our router. If the router has a registered function, we'll call it - otherwise we return a HTTP message that says `404 Not found`.

With the overview in place, let's take a look at the Response, Request and Router classes.

The request object

This class is simply a collection of the data we retrieved from parsing the HTTP message:

```
const Headers = require('./headers');

class Request {
  constructor(req) {
    this.headers = new Headers(req.headers);
    this.method = req.method;
    this.path = req.path;
    this.body = req.body;
    this.protocol = req.protocol;
  }
}

module.exports = Request;
```

We also have another class here to store the headers:

```
class Headers {
  constructor(headers = {}) {
    this.store = headers;
  }

  set = (key, value) => {
    this.store[key] = value;
  };

  get = (key) => {
    return this.store[key];
  };
```

```
}

module.exports = Headers;
```

This abstraction allows us to use the same functionality for headers in the Response object.

The response object

The response object is a little more involved than the request object. We'll use this to modify the HTTP response that we'd like to send back to the client:

```
const Headers = require('./headers');
// not complete in order to save space
const statusTexts = {
  200: 'Ok',
  404: 'Not found',
  500: 'Internal server error',
  401: 'Unauthorized',
  204: 'No content',
};

class Response {
  constructor(connection, protocol) {
    this.status = 200;
    this.headers = new Headers();
    this.protocol = protocol;
    this.connection = connection;
  }

  setProtocol = (protocol) => {
    this.protocol = protocol;
  };

  setStatus = (status) => {
    this.status = status;
    return this;
  };

  send = (body) => {
    const response = `${this.protocol} ${this.status} ${
      statusTexts[this.status]
    }\r\n${this.formatHeaders()}\r\n${this.formatBody(body)}`;
```

```
32
33      this.connection.write(response);
34      this.connection.end();
35    };
36
37    formatHeaders = () => {
38      let result = '';
39      for (const key in this.headers.store) {
40        result += `${key}: ${this.headers.store[key]}\r\n`;
41      }
42
43      return result.length > 0 ? result : '';
44    };
45
46    formatBody = (body) => {
47      if (body) {
48        return `${body}\r\n`;
49      }
50
51      return '';
52    };
53  }
54
55  module.exports = Response;
```

The response object takes in the connection, which allows us to send responses to the client. We also expose methods to set the status and return the class object, and create some methods to format the response properly according to the HTTP spec. This allows us to write a response like this:

```
1  const res = new Response(connection)
2
3  res.setStatus(200).send(`<!DOCTYPE html>
4  <html lang="en">
5    <head>
6      <meta charset="utf-8" />
7      <meta http-equiv="x-ua-compatible" content="ie=edge" />
8      <meta name="viewport" content="width=device-width, initial-scale=1" />
9      <title></title>
10     <link rel="stylesheet" href="css/main.css" />
11     <link rel="icon" href="images/favicon.png" />
12   </head>
13   <body>
14     <h1>Hello from webserver</h1>
```

```
  </body>
</html>`);
```

Resulting in a HTTP response like this:

```
HTTP/1.1 200 Ok

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title></title>
    <link rel="stylesheet" href="css/main.css" />
    <link rel="icon" href="images/favicon.png" />
  </head>
  <body>
    <h1>Hello from webserver</h1>
  </body>
</html>
```

We can also set headers using the headers property:

```
res.headers.set('Content-Type', 'text/html');
res.headers.set('Authorization', 'none');
res.status(200).send(`<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title></title>
    <link rel="stylesheet" href="css/main.css" />
    <link rel="icon" href="images/favicon.png" />
  </head>
  <body>
    <h1>Hello from webserver</h1>
  </body>
</html>`);
```

Resulting in:

```
HTTP/1.1 200 Ok
Content-Type: text/html
Authorization: none

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title></title>
    <link rel="stylesheet" href="css/main.css" />
    <link rel="icon" href="images/favicon.png" />
  </head>
  <body>
    <h1>Hello from webserver</h1>
  </body>
</html>
```

Creating the Router

The final piece of our server is the router. This will take care registering which HTTP messages we'd like our web server to respond to, and the function we'd like to execute when the server receives a HTTP message that matches the path and the method.

```
class Router {
  constructor() {
    this.routes = {
      GET: {},
      POST: {},
      PUT: {},
      PATCH: {},
      OPTIONS: {},
      DELETE: {},
    };
  }

  get = (path, handler) => {
    this.setRoute(this.routes['GET'], path, handler);
  };

  post = (path, handler) => {
```

```
18      this.setRoute(this.routes['POST'], path, handler);
19    };
20
21    put = (path, handler) => {
22      this.setRoute(this.routes['PUT'], path, handler);
23    };
24
25    delete = (path, handler) => {
26      this.setRoute(this.routes['DELETE'], path, handler);
27    };
28
29    patch = (path, handler) => {
30      this.setRoute(this.routes['PATCH'], path, handler);
31    };
32
33    options = (path, handler) => {
34      this.setRoute(this.routes['OPTIONS'], path, handler);
35    };
36
37    getRoute = (method, path) => {
38      return this.routes[method][path];
39    };
40
41    setRoute = (currentPath, path, handler) => {
42      currentPath[path] = handler;
43    };
44  }
45
46  module.exports = Router;
```

The router consists of an object that holds an internal record over the different HTTP methods (a few are omitted for simplicity), and exposes helper methods that allow you to register a route connected to a HTTP verb and a path.

For example, registering a root path becomes:

```
// router
const router = new Router()

router.get('/', (req, res) => {
    const html = `...html here`
    res.setStatus(200).send(html)
})

// Will answer this http message
GET / HTTP/1.1
```

Or maybe you want to register and endpoint to receive a POST request:

```
const router = new Router()

router.post('/article', (req, res) => {
    if (req.headers.get('Content-Type') === 'application/json') {
        let data;
        try {
            data = JSON.parse(req.body);
            // Create article
            createArticle(data);

            res.setStatus(200).send()
        } catch (e) {
            console.log(e);
            // respond with bad request if we can't parse JSON
            res.setStatus(400).send()
        }
    }

})

// will answer this http message
POST /article HTTP/1.1
Content-Type: application/json

{ "name": "Learn to build a webserver", description: "Build a webserver to deepen yo\
ur understanding on how data is exchanged on the internet" }
```

Summing up

With that, we have our very basic HTTP server in place. Since our `SimpleServer` extends the router, it will assume the same methods and properties through the javascript object prototype. This allows us to write code that looks like this:

```
const SimpleServer = require('./simple-server');

const rootHandler = (req, res) => {
  res.headers.set('Content-Type', 'text/html');
  res.headers.set('Authorization', 'none');
  res.send(`<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="x-ua-compatible" content="ie=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title></title>
    <link rel="stylesheet" href="css/main.css" />
    <link rel="icon" href="images/favicon.png" />
  </head>
  <body>
    <h1>Hello from webserver</h1>
  </body>
</html>`);
};

const receiveData = (req, res) => {
  let data;
  try {
    data = JSON.parse(req.body);
  } catch (e) {
    console.log(e);
  }

  res.headers.set('Content-Type', 'text/plain');
  res.headers.set('Authorization', 'none');
  res.send(`Hello from webserver, ${data.name}`);
};

const server = new SimpleServer();

server.get('/', rootHandler);
```

```
38 server.post('/receive', receiveData);
39
40 server.listenAndServe(5100);
```

I encourage you to try this for yourself. Starting this server will allow you to visit `localhost:5100` in your local browser and see the HTML result parsed in the browser. But you can also use `netcat` to send HTTP requests by hand and see what the server returns:

```
1 nc localhost 5100
2 GET / HTTP/1.1
```

Try playing around with setting up your own endpoints and change the path and verbs around.

That concludes this foundational chapter. The knowledge described here will serve you well in subsequent chapters on APIs, because you now posess a deep understanding on how the HTTP protocol work, how requests are made, and how requests are configured to be answered.

Chapter Two - Introduction to APIs

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

API use cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Web Services

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

The server client relationship

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Web service architecture patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

REST API by example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Exercises

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Chapter Three - Writing a REST API

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Tools

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Initial structure

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Setting up the product endpoints

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Error handling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Data validation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Logging

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Environment variables

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Chapter Four - Testing your API

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

A is for automation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Implementing automated testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Chapter Five - Persistant storage

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Storage options

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Chapter Five - API Documentation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.

Chapter seven - API Performance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/theultimateguidetoapis.