HTTP

The objective of this lessonis for you to get comfortable with the main concepts of HTTP. HTTP is the underlying protocol used by the World Wide Web. It's essential knowledge for developers who work with the web. At the end of it, you'll be able to identify common HTTP verbs and status codes, as well as demonstrating how HTTP is used by setting up a simple server.

When you finish, you should be able to

- match the header fields of HTTP with a bank of definitions.
- matching HTTP verbs (GET, PUT, PATCH, POST, DELETE) to their common uses.
- match common HTTP status codes (200, 302, 400, 401, 402, 403, 404, 500) to their meanings.
- send a simple HTTP request to `google.com`
- write a very simple HTTP server using ‘http’ in node with paths that will result in the common HTTP status codes.

HTTP Basics

In the late 1980s, a computer scientist named Tim Berners-Lee proposed the concept of the "WorldWideWeb", laying the foundation for our modern Internet. A critical part of this concept was *HTTP*, the *Hypertext Transfer Protocol*.

We're going to to dive into what makes HTTP such an important part of Web browsing and learn how to leverage it in our applications.

We'll cover:

- the vocabulary of the Web,
- how stateless connections work,
- and HTTP request & response types.

First, some context

"If you want to build a ship, don't drum up the men and women to gather wood, divide the work, and give orders. Instead, teach them to yearn for the vast and endless sea."

-- *Antoine de Saint-Exupéry (paraphrased)*

So far, you've written code that runs in isolation on your own system. Now it's time to set sail into the "vast and endless" Internet! Before we can do so, we need to review the fundamentals: what makes the Web a "web"?

We're going to share a lot of vocabulary here, and it may be a little dry at times, but remember that these are the principles upon which the rest of your journey will be built! You'll find these concepts missing from most programming tutorials, so you'll be ahead of the game if you lay a strong foundation now.

Breaking it down...

Like many disciplines, computer science is built around a shared vocabulary. Let's demystify the acronym "HTTP" to understand it better.

HT-: HyperText

*Hypertext*is simply "content with references to other content". This term is used specifically to refer to content in computing, and may include text, images, video, or any other digital content. If "hypertext" sounds familiar, that's because you've heard it before: *HTML*stands for "**H**yper**T**ext**M**arkup **L**anguage".

Hypertext is what makes the Web a "web", and it's the most fundamental part of how we interact online. We refer to references between hypertext resources as*hyperlinks*, though you're probably used to hearing them referred to as*links*. Without links, the Internet would resemble a massive collection of separate books: each blog, news report, and social media site would exist in total isolation from each other. The ability to link these pages is what makes the kind of interactivity you're learning to build possible, and it was a revolutionary concept when it was introduced!

-TP: Transfer Protocol

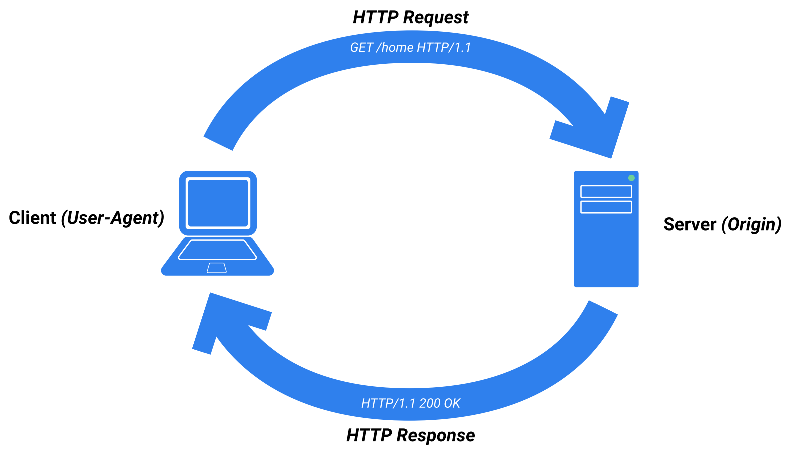
A *protocol*in computer science is a set of guidelines surrounding the transmission of data. Protocols define the process of exchanging data, but don't define exactly what that data must be. Think of it like a multi-course meal: we expect the appetizer, then the entree, then the dessert, but we could have any type of food for each of those courses! As long as the plates arrive in the particular order we expect, protocol is being followed.

HTTP acts as a *transfer protocol*. It defines the expectations for both ends of the transfer, and it defines some ways the transfer might fail. More specifically, HTTP is defined as a *request/response*protocol. An HTTP exchange is more like a series of distinct questions & answers than a conversation between two systems.

...and bringing it back together

HTTP defines the process of exchanging hypertext between systems. Specifically, HTTP works between *clients*and *servers*. A *client*(sometimes called the *user agent*) is the data consumer. This is usually your web browser. A *server*(sometimes referred to as the *origin*) is the data provider, often where an application is running. In a typical HTTP exchange, the client sends a *request*to the server for a particular *resource*: a webpage, image, or application data. The server provides a *response*containing either the resource that the client requested or an explanation of why it can't provide the resource.

Here's a high-level overview of the exchange:



We'll look more closely at the *request*and *response*in separate lessons.

Properties of HTTP

There are a few important properties of HTTP that we need to understand in order to use it effectively.

Reliable connections

Let's consider the example of two friends passing a note. If the note contains important information, the sender will want to make sure that it gets to its destination. They'll likely take a little extra time to deliver it carefully, and they'll expect confirmation once it's been received. In computing, we'd refer to this as a *reliable connection*: messages passed between a client & server sacrifice a little speed for the sake of trust, and we can rest assured that each message will be confirmed.

HTTP doesn't work well if messages aren't received in the correct order, so it's critical that the connection your hypertext is crossing is reliable! Tim Berners-Lee chose *TCP*, another transmission protocol, as HTTP's preferred connection type. We'll discuss TCP in greater detail when we get into network models in a future lesson.

Stateless transfer

HTTP is considered a *stateless*protocol, meaning it doesn't store any information. Each request you send across an HTTP connection should contain all its own context. This is unlike a *stateful*protocol, that might include specifications for storing data between requests.

This can be nice because we only ever need to read a single HTTP request to understand its intent, but it can cause headaches when it comes to things like maintaining your login status or the contents of your shopping cart!

To help us with this, HTTP supports *cookies*, bits of data that a client sends in with their request. The server can examine this data and look up a *session*for your account, or it can act on the info in the cookie directly. Note that neither the cookie nor the session are part of HTTP. They're just workarounds we've created due to the protocol's stateless nature.

Intermediaries

The Web is a big place, and it's unlikely that your request will go directly to its destination! Instead, it will pass through a series of *intermediaries*: other servers or devices that pass your request along. These intermediaries come in three types:

- *proxies*, which may modify your request so it appears to come from a different source,
- *gateways*, which pretend to be the resource server you requested,
- and *tunnels*, which simply pass your request along.

Here's an idea of how these intermediaries might be laid out:

Notice that these are interchangeable depending on the flow of data. When the response is sent back, "Their Router" is acting as a proxy and "Your Router" is acting as a gateway! This is an important part of HTTP: a single server may act as any of the intermediary types, depending on the needs of the HTTP message it's transmitting.

We'll discuss some of these intermediaries more in later lessons. For now, the takeaway is that HTTP isn't limited to your browser & application server. Lots of devices support HTTP in their own special way.

Digging deeper with the HTTP spec

We're just scratching the surface of how HTTP works. If you're interested in learning more, you can go straight to the source: the HTTP spec. A *spec*(short for *specification*) describes a protocol in great detail. It's the document generated by an idea's founders, and it's reviewed and carefully edited before being adopted by the IETF(*Internet Engineering Task Force*).

Specs are intended to be exhaustive, so they can be overwhelming at first! This is definitely not light reading but any question you have about a particular protocol can likely be answered from its spec.

What we've learned

Whew, that's a lot of jargon! Hopefully the fundamental aspects of HTTP are clearer to you now. Next up, we'll look at an HTTP request & response, and we'll cover how to generate each.

After completing this lesson, you should have a clear understanding of:

- HTTP's origin & purpose,
- special properties of HTTP,
- and how to learn more from the HTTP spec.

HTTP Requests

Without a query, there wouldn't be a need for a response! Let's take a look at the *request*: the client-initiated portion of an HTTP exchange.

We'll cover:

- what an HTTP request looks like,
- fields that make up a request,
- and how to send a request of your own!

Retrieving hypertext

Years ago, daily shopping looked very different. Instead of walking the aisles and picking up what they wanted, customers would approach a counter and ask a clerk to retrieve the items on their list. The clerk was responsible for knowing where those items were located and how best to get them to the customer.

While the retail industry has changed dramatically since that time, the Web follows that old tried-and-true pattern. You tell your browser which website you would like to access, and your browser hands that request off to a server that can get you what you've asked for. At the simplest level, the Web is just made up of computers asking each other for things!

Your browser's part in this transaction is called the *request*. Since the browser is acting on your behalf, we sometimes refer to it as the *user-agent*(you being the *user*). You might also hear this referred to more generically as the *client* in the exchange.

Structure of an HTTP request

Your browser is designed to be compliant with the HTTP specification, so it knows how to translate your instructions into a well-formatted HTTP request. An important part of the HTTP spec is that it's simple to read, so let's take a look at an example.

Here's what the HTTP request looks like for visiting `appacademy.io`:

```
GET / HTTP/1.1
Host: appacademy.io
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_5) AppleWebKit/537.36 (K
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/ap
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
```

Let's break it down!

Request-line & HTTP verbs

The first line of an HTTP request is called the **request-line**, and it sets the stage for everything to come. It's made up of three parts, separated by spaces:

- the *method*, indicated by an *HTTP verb*,
- the *URI (Uniform Resource Indicator)*that identifies what we've requested,
- and the *HTTP version*we expect to use (usually `HTTP/1.1`or `HTTP/2`).

In our `appacademy.io`example, we can see that our version matches the most common HTTP version (`1.1`) and that our URI is `/`, or the *root*resource of our target. That first word, `GET` is the HTTP verb we're using for this request.

HTTP verbs are a simple way of declaring our intention to the server. We do the same thing with English verbs when asking for help: "Can you **get** me that?", "Should I **remove**this?", etc. HTTP has a small handful of verbs available, but we're going to look at the five most common: `GET`, `POST`, `PUT`, `PATCH`, and `DELETE`.

- `GET`is used for direct requests. A `GET`request is generally how websites are retrieved, and they only require that the server return a resource. These types of requests will never have a body. Any data you need to send in a `GET`request must be shared via the URI.
- `POST`is typically used for creating new resources on the server. Most of the time, when you submit a form a `POST`request is generated. These types of requests can have a *body*containing any data the server might need to complete your request, like your username & password or the contents of your shopping cart.
- `PUT`requests are used to update a resource on the server. These will contain the whole resource you'd like to update. For example: when updating your name on a website, a `PUT`request will be generated containing not just your new name but also your user ID, email, etc.
- `PATCH`requests are very similar to `PUT`requests, but do not require the whole resource to perform the update. Keeping with our example of updating your name: a `PATCH`request would only require your new name, not the rest of your account details, to succeed.
- `DELETE`requests destroy resources on the server. These might be saved database records, like removing a product that's sold out, or more ephemeral resources, like logging a user out of their current session.

Ultimately, how these verbs get acted upon is up to the server. You could write an application that totally ignores these rules and uses a `DELETE`request to log in, but that's only going to confuse your teammates and frustrate you in the future! It's best to use them as the spec intends.

Headers

The *request-line* sets the table, but it's the headers that describe the menu!*Headers*are key/value pairs that come after the *request-line*. They each appear on separate lines and define metadata needed to process the request. Here are some common request headers you'll see:

- `Host`: The root path for our URI. This is typically the *domain*we'd like to request our resource from. As you can see above, our `Host`header for`appacademy.io`is, appropriately, `appacademy.io`.!
- `User-Agent`: This header displays information about which browser the request originated from. It's generally formatted as `name/version`. You can see in the `User-Agent`header above that we're using `Chrome/76.0`

  Our `User-Agent`has much more content, including references to Mozilla, makers of the popular Firefox browser, and Safari, Apple's default browser of choice. What gives?

  There is some interesting historybehind those additional references, and you can use www.useragentstring.comfor additional details about your current browser's `user-agent`.
- `Referer`: This defines the URL you're coming from. There's none in our example since we navigated directly to the App Academy website, but if we click any link on the page, the resulting HTTP request will have `Referer: https://appacademy.io/`in its headers. Also, you're not reading it wrong - this header is misspelled! It should be "referrer", but it was written incorrectly in the original specification and the typo stuck. Let this be a lesson: your poorly-written code might still be around in 20 years, too!
- `Accept`: "Accept-" headers indicate what the client can receive. When we go to most websites, our `Accept`header will be long to ensure we get all the various types of content that site might include. However, we can modify this

header in our requests to only get back certain types of data. One common use is setting `Accept: application/json`to get a response in JSON format instead of HTML. You may see variations of this header like `Accept-Language`for internationalized websites or `Accept-Encoding`for sites that support alternative compression formats.

- `Content-*`: Content headers define details about the body of the request. The most common content header is `Content-Type`, which lets the server know what format we're sending our body data as. This might be `application/json`from a JavaScript app or `application/x-www-form-urlencoded`for info submitted from a web form. Content headers will only show up on requests that support content in the body, so `GET`requests should never have this!

There are LOTS of other header keys! MDNhas an exhaustive reference list with examples.

Body

When we need to send data that doesn't fit in a header & is too complex for the URI, we can place it in the *body* of our HTTP request. The body comes right after the headers and can be formatted a few different ways.

The most common way form data is formatted is *URL encoding*. This is the default for data from web forms and looks a little like this:

```
name=claire&age=29&iceCream=vanilla
```

Alternatively, you might format your request body using JSON or XML or some other standard. What's most important is that you remember to set the appropriate `Content-Type`header so the server knows how to interpret your body.

Sending an HTTP request from the command line

We've discussed HTTP requests mostly in the context of your web browser, but that's not the only way. There are lots of HTTP clients out there you can use to send requests.

Let's stay close to the exchange itself with a lightweight tool that requires us to do most of the work ourselves. We'll use `netcat`(also known as `nc`), a utility that comes as part of Unix-like environments such as Ubuntu and macOS.

`netcat`allows you to open a direct connection with a URL and manually send HTTP requests. Let's see how this works with a quick `GET`request to App Academy's homepage.

From your command line, type `nc -v appacademy.io 80`. This will open a connection to `appacademy.io`on port 80 (the port most-often used for web connections). Once the connection is established, you'll be able to type out a simple HTTP request by hand! Let's copy the *request-line* and `Host:`header from our request above:

```
GET / HTTP/1.1
Host: appacademy.io
```

Now hit "Return" on your keyboard twice. This will send the request and display the server's response. You should see something similar to this:

```
HTTP/1.1 301 Moved Permanently
Date: Thu, 03 Oct 2019 04:17:23 GMT
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: max-age=3600
Expires: Thu, 03 Oct 2019 05:17:23 GMT
Location: https://www.appacademy.io/
Server: cloudflare
CF-RAY: 51fc1b0f8b98d304-ATL
```

Congratulations! You've sent your first manual HTTP request. We'll discuss the parts of the HTTP response you received in an upcoming lesson.

Try it one more time, this time typing `nc -v neverssl.com 80` and making the same HTTP request with the command `GET / HTTP/1.1`and the header `Host: neverssl.com`. Don't forget to hit Enter twice. Look! That's the HTML coming back from the server! Neat-o!

You can read much more about `netcat`by invoking the manual: `man nc`. We'll also use it in an upcoming project for extra practice.

What we've learned

HTTP requests are the first step to getting what you want on the web. Having completed this lesson, you should be able to recount:

- what an HTTP request is,
- some common HTTP request verbs,
- a rough outline of the HTTP request format,
- and how to use `netcat`to send HTTP requests from your command line.

HTTP Responses

A web server delivers content via *responses*, the second part of the HTTP's request/response cycle. Let's dive into how a response is structured and what your client can expect from the server.

We'll cover:

- HTTP response structure,
- differentiating errors & successful transfers,
- and how to use a server to generate your own responses.

Hypertext delivered

An HTTP response contains either the content we requested or an explanation of why that content couldn't be delivered. It's just like ordering at a restaurant: you place your order and receive either a plate of delicious food or an apology from the chef. In a good restaurant, the apology will include some extra help: "I'm sorry, we're out of broccoli. Can we get you something else? How can we make this right?".

When designing your own HTTP responses, remember that restaurant example. It's important to note that there's a problem, but it's equally important to provide reliable, helpful details. We'll look at some examples of this when we build our own HTTP server in a later lesson.

Structure of a Response

Responses are formatted similarly to requests: we'll have a *status-line*(instead of a request-line), headers that provide helpful metadata about the response, and the response body: a representation of the requested resource.

Here's what the HTTP response looks like when visiting `appacademy.io`:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: close
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Cache-Control: max-age=0, private, must-revalidate
Set-Cookie: _rails-class-site_session=BAh7CEkiD3Nlc3Npb25faWQGOgZFVEkiJTM5NWM5YTV
X-Request-Id: cf5f30dd-99d0-46d7-86d7-6fe57753b20d
X-Runtime: 0.006894
Strict-Transport-Security: max-age=31536000
Vary: Origin
Via: 1.1 vegur
Expect-CT: max-age=604800, report-uri="https://report-uri.cloudflare.com/cdn-cgi/
Server: cloudflare
CF-RAY: 51d641d1ca7d2d45-TXL

<!DOCTYPE html>
<html>
...
...
</html>
```

Oof! That's a lot of unfamiliar stuff. Let's walk through the important bits together.

Status

Like the request, an HTTP response's first line gives you a high-level overview of the server's intention. For the response, we refer to this as the*status-line*.

Here's the status line from our `appacademy.io`response:

```
HTTP/1.1 200 OK
```

We open with the HTTP version the server is responding with. `1.1` is still the most commonly used, though you may occasionally see `2`or even `1.0`. We follow this with a `Status-Code`and `Reason-Phrase`. These give us a quick way of understanding if our request was successful or not.

HTTP status codes are a numeric way of representing a server's response. Each code is a three-digit number accompanied by a short description. They're grouped by the first digit (so, for example, all "Informational" codes begin with a `1`:`100` - `199`).

Let's take a look at the most common codes in each group.

Status codes 100 - 199: Informational

Informational codes let the client know that a request was received, and provide extra info from the server. There are very few informational codes defined by the HTTP specification and you're unlikely to see them, but it's good to know that they exist!

Status codes 200 - 299: Successful

Successful response codes indicate that the request has succeeded and the server is handling it. Here are a couple common examples:

- **200 OK**: Request received and fulfilled. These usually come with a `body`that contains the resource you requested.
- **201 Created**: Your request was received and a new record was created as a result. You'll often see this response to `POST`requests.

Status codes 300 - 399: Redirection

These responses let the client know that there has been a change. There are a few different ways for a server to note a redirect, but the two most common are also the most important:

- **301 Moved Permanently**: The resource you requested is in a totally new location. This might be used if a webpage has changed domains, or if resources were reorganized on the server. Most clients will automatically process this redirect and send you to the new location, so you may not notice this response at all.
- **302 Found**: Similarly, to *301 Moved Permanently*, this indicates that a resource has moved. However, this code is used to indicate a temporary move. It's not often that you see temporary moves online, but this code may be used to indicate a permanent move where the old domain should still be valid too. Clients will usually follow this redirect automatically as well, but you shouldn't necessarily update your links until the server returns a `301`.

301 Moved Permanentlyand 302 Foundoften get confused. When might we want to use a 302 Found`? The most common use case today is for the transition from HTTP to HTTPS. _HTTPSis secure HTTP messaging, where requests & responses are encryptedso they can't be read by prying eyes while en route to their destinations.

This is a much safer way of communicating online, so most websites require access via `https://`before the domain. However, we don't want to ignore folks still trying to access our content from the older `http://`approach!

In this case, we'll return a 302 Foundresponse to the client, letting them know that it's okay to keep navigating to `http://our-website.com`, but we're going to redirect them to `https://our-website.com`for their protection.

Status codes 400 - 499: Client Error

The status codes from 400 to 499, inclusive, indicate that there is a problem with the client's request. Maybe there was a typo, or maybe the resource we requested is no longer available. You'll see lots of these as you're learning to format HTTP requests. Here are the most common ones:

- **400 Bad Request**: Whoops! The server received your request, but couldn't understand it. You might see a *400 Bad Request*in response to a typo or accidentally truncated request. We often refer to these as *malformed*requests.
- **401 Unauthorized**: The resource you requested may exist, but you're not allowed to see it without authentication. These type of responses might mean one of two things: either you didn't log in yet, or you tried to log in but your credentials aren't being accepted.
- **403 Forbidden**: The resource you requested may exist, but you're not allowed to see it *at all*. This response code means this resource isn't accessible to you, even if you're logged in. You just don't have the correct permission to see it.
- **404 Not Found**: The resource you requested doesn't exist. You may see this response if you have a typo in your request (for example: going to`appaccccademy.io`), or if you're looking for something that has been removed.

403 Forbiddenrequests let the client know that a valid resource was requested. This can be a security risk! For example: if I guess that you havepasswords.htmlon your website because you just want to be hacked, a 403 Forbiddenresponse tells me I'm correct. For this reason, some sites will return a 404 Not Foundfor resources that exist but aren't accessible.

A well-known example is GitHub. If you try to open a repository you don't have permission to access, GitHub will return a 404 Not Foundeven if your URL is correct! This protects you from random users guessing the names of your projects.

Status codes 500 - 599: Server Error

This range of response codes are the Web's way of saying "It's not you, it's me." These indicate that your request was formatted correctly, but that the server couldn't do what you asked due to an internal problem.

There are two common codes in this range you'll see while getting started:

- **500 Internal Server Error**: Your request was received, and the server tried to process it, but something went awry! As you're learning to write your own servers, you'll often see a *500 Internal Server Error*as your code fails unexpectedly.
- **504 Gateway Timeout**: Your request was received but the server didn't respond in a reasonable amount of time. Timeout errors can be tricky: your first instinct may be that your own connection is bad, but this code means the problem is likely on the server's side. You'll often see these when a server is no longer reachable (maybe due to an unexpected outage or power failure).

Headers

Headers on HTTP responses work identically to those on requests. They establish metadata that the receiving client might need to process the response. Here are a few common response headers you'll see:

- **Location**: Used by the client for redirection responses. This contains the URL the client should redirect to.
- **Content-Type**: Lets the client know what format the body is in. Your client will display different types of response content in different ways, to setting the *Content-Type*is important! Notice that this header can be present on responses **and**requests. It's a generic header for any HTTP interaction involving content.
- **Expires**: When the response should be considered *stale*, or no longer valid. The *Expires*header lets your client *cache*responses (that is: save them locally to prevent having to repeatedly re-download them). The client may ignore requests to that same resource until after the date set in the*Expires*header.
- **Content-Disposition**: This header lets the client know how to display the response, and is specifically devoted to whether the response should be visible to the client or delivered as a download. Think about your own experience online: sometimes you click a button and get an immediate download, while in other cases you click a button and get to "preview" the content before you download it. This is controlled by the *Content-Disposition*header.
- **Set-Cookie**: This header sends data back to the client to set on the*cookie*, a set of key/value pairs associated with the server's domain. Remember how HTTP is *stateless*? Cookies are one way to get around that!*Set-Cookie*may send back information like a unique ID for the user you've logged in as or details about other resources you've requested on this domain.

Remember - this isn't an exhaustive list of headers! Sites and intermediate gateways/proxies can define their own custom headers, so you'll see many more than these. If you're unsure what a header does, the MDN HTTP Header documentationis a great place to start searching.

Body

Assuming a successful request, the *body*of the response contains the resource you've requested. For a website, this means the HTML of the page you're accessing.

The format of the body is dictated by the *Content-Type*header. This is an important detail! If you accidentally configure your server to send "Content-Type: application/json" along with a body containing HTML, your HTML won't be rendered properly and your users will see plain text instead of beautifully-rendered elements. In the same way, API responses should be clearly marked so that other applications know how to manage them.

We can see in our `appacademy.io`response above that the body begins with `!<DOCTYPE html>`and ends with `</html>`. If you inspect the source of the page in your browser, you'll see that this is exactly what's being rendered. Headers may change **how** the browser handles the body, but they won't **modify**the body's content.

Using a custom server to generate responses

At its most basic, a web server is just a tool to generate HTTP responses. Therefore, the best way to practice is to build your own webserver!

We'll walk through building our own server from scratch using Node.js in an upcoming video lesson.

What we've learned

Like HTTP requests, HTTP responses involve lots of new lingo and details. Hang in there - we'll start doing practical work with this new vocabulary in the projects & video demos coming up.

After this reading, you should:

- Understand the parts of an HTTP response,
- be able to identify common status codes & their meanings,
- recognize common response headers,
- and be prepared to build your own server!