Fundamental Skills - Understanding HTTP Requests

Category	Experience Level
Web	Complete Beginner

Contents

What is a HTTP Request?

When computers communicate with each other over the internet, they require a set of rules to govern how this communication should take place.

These rules are called protocols, and different protocols exist for many different communication methods and purposes. Hypertext Transfer Protocol (HTTP) is one of the most common protocols, and is used to display and interact with websites.

Web Servers

When a computer *hosts* a website, it will use some sort of HTTP Server technology to do so. There are many different pieces of software that can do this:

- an extremely simple server can be hosted with Python's `SimpleHTTPServer` class
- an Apache server, which includes various features such as error handling and logging
- a framework such as Laravel can create a complex server that hosts PHP files

Whatever software is chosen, a port on the computer will be exposed to the public internet, and the computer becomes known as a *webserver*.

We will talk more about addresses, ports, and protocols more in a later lesson - for now, you just need to know that HTTP can run as a *service* on a *webserver*, and we can communicate with that webserver either using its IP address or a domain name for the website.

HTTP Requests

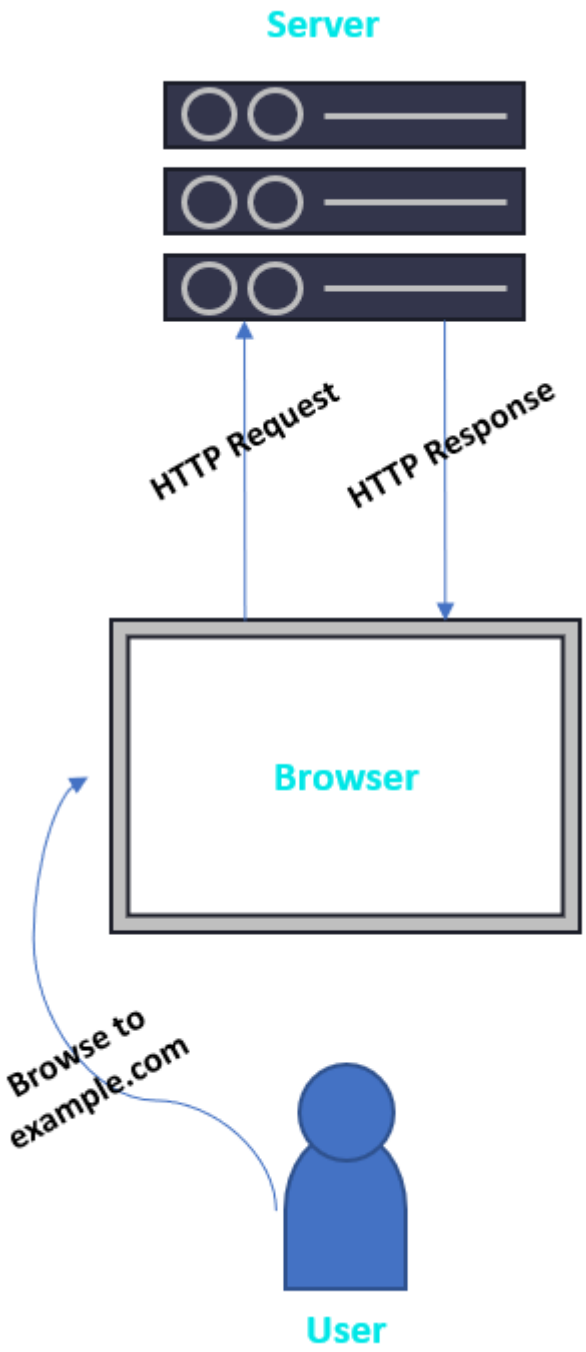
When we have a server running and we have its address, we can send it a HTTP request.

A HTTP Request can be used to interact with a website in many different ways. It can:

- request a specific page's content
- request a change, e.g. the creation, modification, or deletion of a resource
- attempt to upload a file

Similarly, the server can respond in several different ways. It can:

- cause a redirect to a new page
- mark an action as forbidden
- request authorisation for an action
- indicate that an error has occurred



In practical terms, whenever you visit a website (say `example.com`) your browser makes a HTTP request to that server for the specific page on that domain that you requested (e.g. `http://example.com/info`). The server will respond to this request, likely with some HTML code that your browser will then render.

When you submit data to this website, for example by logging in at `http://example.com/login`, your browser will make a different type of request, and the webserver may respond in a different way.

HTTPS

Hypertext Transfer Protocol Secure (HTTPS) is a similar protocol to HTTP - the main difference is that HTTPS traffic (i.e. requests and responses) is encrypted.

This means that the contents of a request (for example the specific file that is being requested, or login details that are being submitted) cannot be read by people who are monitoring traffic on a network.

Encryption is usually done by an algorithm called TLS, but may also use the older deprecated SSL algorithm.

Types of HTTP Request

A request can have many methods, depending on its purpose. These methods are often referred to as *verbs*.

GET requests are usually to request a resource, such as the contents of a webpage. For example, when you visit `http://example.com` in your browser, you are actually making a HTTP `GET` request for the file `/` under the hood (where `/` is just shorthand for `http://example.com/`, the site's *index*)

POST requests are usually used to submit data. For example, you may submit a POST request to `http://example.com/login` with the data `username=janedoe&password=securepassword`. The server will then process this data and decide whether to let you in or not. It may respond with a new page if you're successful (for example redirecting to `/profile`), or with some HTML code telling you your password is wrong.

PUT requests are used to write new data to the webserver - for example, rather than simply using a **POST** request to ask the server a question (like "are my login details correct?"), you might use a **PUT** request to write some new data, like a database entry or a file. **POST** requests can actually do a lot of the things that **PUT** requests can do, and due to their versatility are used more commonly.

DELETE requests have a similar use case to **PUT** requests - they can remove data from a server, but their purpose can also be achieved by a **POST** request with some backend processing. For example, a **POST** request to `/delete` on a server could trigger to server to make an equivalent action.

There are many other HTTP methods that have some other important uses. However, the most important ones to remember are **GET** (for retrieving information) and **POST** (for sending information).

Elements of a HTTP Request

A simple HTTP request can be represented like so:

```
GET /home HTTP/1.1
Host: example.com
```

```
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101
Firefox/78.0
```

The first line is known as the *start line* - it describes the request method (`GET`), the file that's being requested (`/home`), and the protocol (`HTTP/1.1`).

The other lines are *headers*, which are used to pass extra information to the server. The `Host` header, for example, specifies the domain that the HTTP request targets (combined with the filename this gives `example.com/home`).

GET requests may also have a request body (containing parameters or data), but more commonly will have parameters inside the file name, indicated by `?` characters and delimetered by `&`; for example:

```
GET /profile?id=123456&show_secret_info=true HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101
Firefox/78.0
```

This GET request has two parameters in the URL: `id`, which is equal to `123456`; and `show_secret_info`, equal to `true`.

POST requests have a similar structure, but more commonly have information stored in the request body:

```
POST /update-profile HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:78.0) Gecko/20100101
Firefox/78.0

{"JSON":"{\"name\":\"Jane Doe\"}"}
```

Here we are POSTing a JSON, which is a structure of keys and values in pairs, that might be used by the server to update our profile name.

Common Headers

Common *headers* include:

- `User-Agent` - this header indicates the type of device or software that is making the HTTP request. It is used by browsers and tools like Curl to identify themselves, and can be used by the server to warn users about outdated browsers or reject requests from suspicious users. However, it can be arbitrarily set, so is not a good defence mechanism on its own

- `Content-Type` - this indicates the type of data that is being sent. It is commonly equal to `text/html`, but may also be set to `application/x-www-form-urlencoded` (when submitting form data) or `application/json` when submitting data to an API. Getting this header wrong is a common reason an exploit may fail
- `Authorization` - this header is often used to store authorisation information, such as cookies. However, cookies could be stored under any arbitrary name, such as `PHPSESSID`, so being able to recognise what *may* be a cookie is a good skill
- `PHPSESSID` - a common header used in PHP-based applications for tracking user sessions

Response

A HTTP response looks similar to a HTTP request:

```
HTTP/1.1 200 OK
Server: example.com
Date: Tue, 21 Sep 2021 20:28:59 GMT
Content-Type: text/html; charset=utf-8
Set-Cookie: XSRF-TOKEN=ghTVo....b7ivy

<p>Hello World!</p>
```

The first line contains the protocol and a status code, in this case `200 OK` - this indicates whether the request was successful.

Common responses include:

- `301` - a redirect to another page
- `403` - forbidden (i.e. the server has denied access to the resource)
- `404` - the resource was not found
- `500` - an internal server error - this is often a good sign when testing for malicious user input

The response may also contain headers such as `Set-Cookie`, which is then used by the browser to store a cookie locally, or response data such as some JSON or HTML code.

Making a HTTP Request

It isn't just browsers that can make HTTP requests - there are many tools capable of this as well.

Curl is a tool made for making requests in the command line, and has a multitude of options including the ability to set headers (e.g. user agents and cookies), proxy

requests, POST data, follow redirects, and more.

Feroxbuster is a tool for making *lots* of HTTP requests in order to discover a site's contents. It is only to be used against sites you have explicit permission to test, as it creates a lot of network traffic.

Burp Suite is a tool for viewing HTTP requests as you make them, but it can also edit and resend HTTP requests with its Repeater function.

Netcat is a tool for making various network connections, and it can be used to manually make a HTTP request by sending a start line when a connection is opened. Netcat is sometimes useful for determining whether an obscure port is a webserver or not.

We will cover all of these tools in more detail in later lessons.

Going Further

To learn a bit more about the structure of a HTTP request, see IBM's guide or Mozilla's Articles

To read more about the mechanics of a HTTP request on a lower level, read this: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Response_syntax

HTTP requests are just one layer of the OSI model - to learn about this crucial piece of networking theory, read this: https://www.networkworld.com/article/3239677/the-osi-model-explained-and-how-to-easily-remember-its-7-layers.html

Worksheet

1. Make a request to shefesh.com and view it with your Developer Tools' Network tab. Can you identify any important headers that tell you anything about the site?
2. Can you find any requests that the server responds to with non-HTML content?