GET vs POST

In this lesson, we'll compare and contrast GET and POST requests.

We'll cover the following

As we've seen earlier, an HTTP request starts with a peculiar start line:

```
GET / HTTP/1.1
```

First and foremost, a client tells the server what verbs it is using to perform the request. Common HTTP verbs include `GET`, `POST`, `PUT`, and `DELETE`, but the list could go on with less common verbs like `TRACE`, `OPTIONS`, or `HEAD`.

In theory, no method is safer than others; in practice, it's not that simple.

`GET` requests usually don't carry a body, so parameters are included in the URL (i.e., `www.example.com/articles?article_id=1`) whereas `POST` requests are generally used to send ("post") data which is included in the body. Another difference is in the side effects that these verbs carry with them. `GET` is an idempotent verb, meaning no matter how many requests you will send, you will not change the state of the webserver. `POST`, instead, is not idempotent; for every request you send you might be changing the state of the server. Think of, for example, POSTing a new payment, now you probably understand why sites ask you not to refresh the page when executing a transaction.

To illustrate an important difference between these methods we need to have a look at webservers' logs, which you might already be familiar with:

Sending data in the body is safer than in URL

```
192.168.99.1 - [192.168.99.1] - - [29/Jul/2018:00:39:47 +0000] "GET /?token=12
34 HTTP/1.1" 200 525 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.3
6 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36" 404 0.002 [example-]
```

```
ocal] 172.17.0.8:9090 525 0.002 200
192.168.99.1 - [192.168.99.1] - - [29/Jul/2018:00:40:47 +0000] "GET / HTTP/1.
1" 200 525 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, lik
e Gecko) Chrome/65.0.3325.181 Safari/537.36" 393 0.004 [example-local] 172.17.
0.8:9090 525 0.004 200
192.168.99.1 - [192.168.99.1] - - [29/Jul/2018:00:41:34 +0000] "PUT /users HTT
P/1.1" 201 23 "http://example.local/" "Mozilla/5.0 (X11; Linux x86_64) AppleWe
bKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.181 Safari/537.36" 4878 0.01
6 [example-local] 172.17.0.8:9090 23 0.016 201
```

As you can see, webservers log the request path. This means that if you include sensitive data in your URL, it will be leaked by the web server and saved somewhere in your logs. Your secrets are going to be somewhere in plaintext, something we need to avoid at all costs. Imagine an attacker being able to gain access to one of your old log files, which could contain credit card information, access tokens for your private services, and so on. That would be a disaster.

Webservers do not log HTTP headers or bodies, as the data to be saved would be too large. This is why sending information through the request body, rather than through the URL, is generally safer.

From here we can derive that `POST` and other, non-idempotent methods are safer than `GET`, even though it's more a matter of how data is sent when using a particular verb rather than a specific verb being intrinsically safer than others. If you were to include sensitive information in the body of a `GET` request, you'd face no more problems than when using a `POST`, even though the approach would be considered unusual.

Avoid storing durable and sensitive information in URLs

URLs are usually logged by web servers, so any information stored there could potentially be leaked. If you need to, consider using one-time/expiring secrets that are of no use in the long run. Amazon S3 signed URLs are a brilliant example of using expiring secrets to grant temporary access to a resource.

In HTTP headers we trust

In this chapter we looked at HTTP, its evolution, and how its secure extension integrates authentication and encryption to let clients and servers communicate integrates authentication and encryption to let clients and servers communicate through a safe(r) channel. This is not all HTTP has to offer in terms of security, as

we will see shortly. HTTP security headers offer a way to improve our application's security posture, and the next chapter is dedicated to understanding how to take advantage of them.

Take a quiz to test your knowledge of HTTP in the next lesson!