What Does a Browser Do?

In this lesson, we'll dive a bit deeper into how web browsers work.

We'll cover the following

Long story short, a browser's job consists of:

- DNS resolution
- HTTP exchange
- Rendering
- Rinse and repeat

DNS resolution

DNS resolution makes sure that once the user enters a URL, the browser knows which server it should connect to. The browser contacts a DNS server to find that `google.ae` translates to `216.58.207.110` which is an IP address the browser can connect to.

HTTP exchange

Once the browser has identified which server is going to serve our request, it will initiate a TCP connection and begin the **HTTP exchange**. This is nothing but a way for the browser to communicate to the server what it wants, and for the server to reply back.

HTTP is simply the name of the most popular protocol for communication on the web. Browsers mostly talk via HTTP when communicating with servers. An HTTP exchange involves the client, our browser, sending a **request** and the server

exchange involves the client, our browser, sending a **request** and the server replying back with a **response**.

Parsing a sample request

For example, after the browser has successfully connected to the server behind `google.com`, it will send a request that looks like the following:

```
GET / HTTP/1.1
Host: google.com
Accept: */*
```

Let's break the request down, line by line:

- `GET / HTTP/1.1`: with the *start line*, the browser asks the server to retrieve the document at the location `/`, adding that the rest of the request will follow the `HTTP/1.1` protocol. It could also have used `1.0` or `2`.
- `Host: google.com`: this is **the only HTTP header mandatory in HTTP/1.1**. Since the server might serve multiple domains (`google.com`, `google.co.uk`, etc.) the client here mentions that the request was for that specific host.
- `Accept: */*`: an optional header, where the browser is telling the server that it will accept any kind of response back. The server could have a resource that is available in JSON, XML, or HTML formats, so it can pick whichever format it prefers.

Parsing a sample response

In this example, the browser, acting as a **client**, is done with its request; now it's the server's turn to reply:

```
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=1234; expires=Fri, 18-Jan-2019 18:25:04 GMT; path=/; domain=.g
oogle.com; HttpOnly

<!doctype html><html">
...
...
```

```
</html>
```

Whoa, that's a lot of information to digest! Let's break it down.

The server lets us know that the request was successful (`200 OK`) and adds a few headers to the **response**;

- For example, it advertises which server processed our request (`Server: gws`), what the `X-XSS-Protection` policy of this response is, and so forth.

You do not need to understand each and every piece of information right now, as we will go over the HTTP protocol, its headers, and so on in their dedicated chapters. For now, all you need to understand is that the client and the server are exchanging information and that they do so via HTTP.

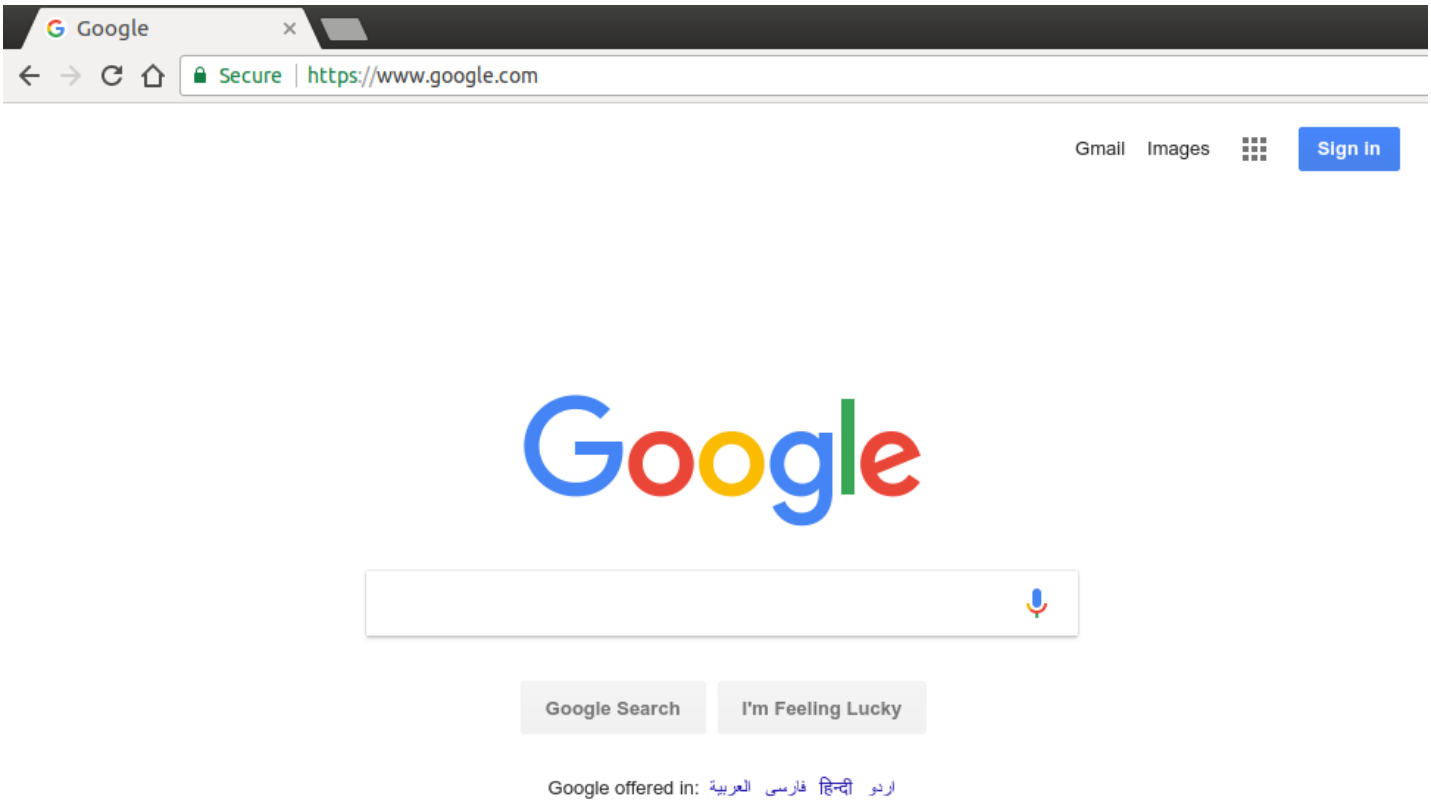
How a browser renders an HTTP response

Last but not the least, let's discuss the **rendering** process. How good would a browser be if the only thing it showed to the user was a list of funny characters?

```
<!doctype html><html>
...
...
</html>
```

In the **body** of the response, the server includes the representation of the response according to the `Content-Type` header. In our case, the content type was set to `text/html`, so we are expecting HTML markup in the response, which is exactly what we find in the body. This is where a browser truly shines: it parses the HTML, loads additional resources included in the markup (for example, there could be JavaScript files or CSS documents to fetch), and presents them to the user as quickly as possible.

Once more, the end result is something the average Joe can understand:



google.com rendered by the Chrome browser

For a more detailed version of what really happens when we hit enter in the address bar of a browser, I would suggest reading "What happens when...", an elaborate explanation of the mechanics behind the process.

Since this course is focused on security, I am going to drop a hint about what we've just learned, attackers easily make a living out of vulnerabilities in the HTTP exchange and rendering process. Vulnerabilities and malicious users lurk in other places as well, but a better security approach on those levels already allows you to make strides in improving your security posture.

In the next lesson, we'll study some popular browser vendors.