Web Application Basics

Web Application Overview

Front end

The **Front End** the parts that a user can see and interact with and use HTML, CSS, and JavaScript to do this.

HTML (Hypertext Markup Language) is a foundational aspect of web applications. It is a set of instructions or code that instructs a web browser on what to display and how to display it

CSS (Cascading Style Sheets) in web applications describes a standard appearance, such as certain colours, types of text, and layouts.

JS (JavaScript) is part of a web application front end that enables more complex activity in the web browser. Whereas HTML can be considered a simple set of instructions on what to display, JavaScript is a more advanced set of instructions that allows choices and decisions to be made on what to display.

Back End

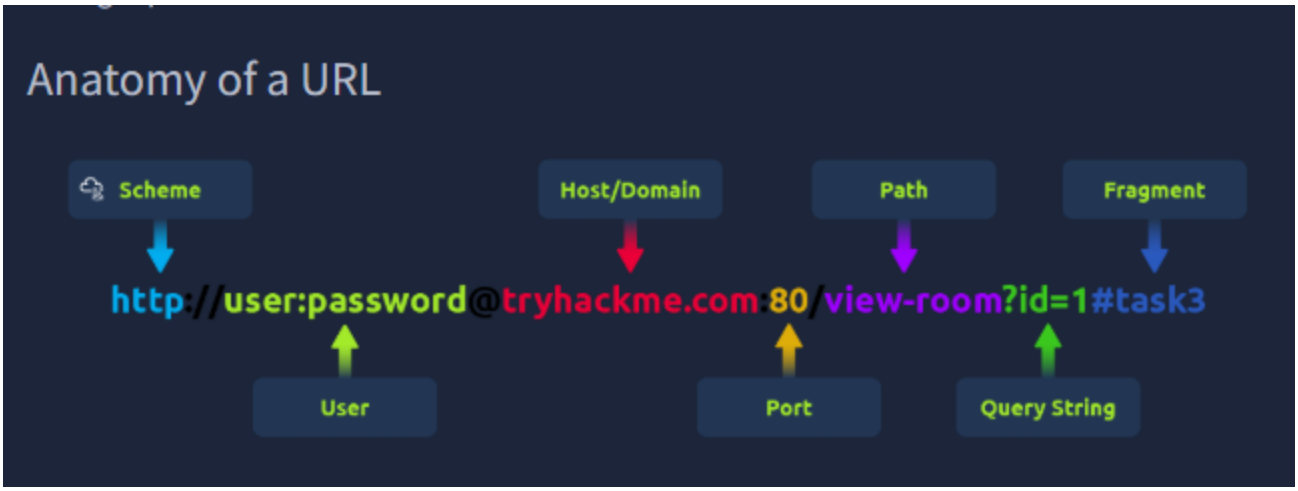
The **Back End** of a web application is things you don’t see within a web browser but are important for the web application to work.

A **Database** is where information can be stored, modified, and retrieved.

There are many other **Infrastructure** components underpinning Web Applications, such as web servers, application servers, storage, various networking devices, and other software that support the web application.

WAF (Web Application Firewall) is an optional component for web applications. It helps filter out dangerous requests away from the Web Server and provides an element of protection.

Uniform Resource Locator (URL)



Scheme
The **scheme** is the protocol used to access the website. The most common are **HTTP** (HyperText Transfer Protocol) and **HTTPS**

User
Some URLs can include a user's login details (usually a username) for sites that require authentication.

Host/Domain
The **host** or **domain** is the most important part of the URL because it tells you which website you're accessing.

Port
The **port number** helps direct your browser to the right service on the web server.

Path
The **path** points to the specific file or page on the server that you're trying to access.

Query String
The **query string** is the part of the URL that starts with a question mark (?). It's often used for things like search terms or form inputs. handle them securely to prevent attacks like **injections**.

Fragment

The **fragment** starts with a hash symbol (#) and helps point to a specific section of a webpage —like jumping directly to a particular heading or table.

HTTP Messages

HTTP messages are packets of data exchanged between a user (the client) and the web server.

There are two types of HTTP messages:

- **HTTP Requests**: Sent by the user to trigger actions on the web application.
- **HTTP Responses**: Sent by the server in response to the user's request.

Each message follows a specific format that helps both the user and the server communicate smoothly.

Start Line
The start line is like the introduction of the message. It tells you what kind of message is being sent—whether it's a request from the user or a response from the server.

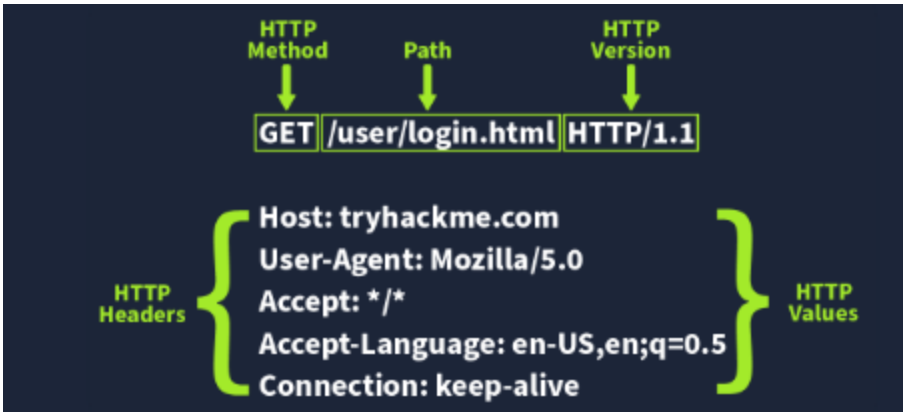
Headers
Headers are made up of key-value pairs that provide extra information about the HTTP message. They give instructions to both the client and the server handling the request or response.

Empty Line
The empty line is a little divider that separates the header from the body. It's essential because it shows where the headers stop and where the actual content of the message begins.

Body
The body is where the actual data is stored. In a request, the body might include data the user wants to send to the server (like form data).

HTTP Request: Request Line and Methods

An **HTTP request** is what a user sends to a web server to interact with a web application and make something happen.



Request Line

The **request line** (or start line) is the first part of an HTTP request and tells the server what kind of request it's dealing with. It has three main parts: the **HTTP method**, the **URL path**, and the **HTTP version**.
Example: `METHOD /path HTTP/version`

HTTP Methods

The **HTTP method** tells the server what action the user wants to perform on the resource identified by the URL path.

GET

Used to **fetch** data from the server without making any changes. Reminder! Make sure you're only exposing data the user is allowed to see. Avoid putting sensitive info like tokens or passwords in GET requests since they can show up as plaintext.

POST

Sends data to the server, usually to create or update something. Reminder! Always validate and clean the input to avoid attacks like SQL injection or XSS.

PUT

Replaces or **updates** something on the server. Reminder! Make sure the user is authorised to make changes before accepting the request.

DELETE

Removes something from the server. Reminder! Just like with PUT, make sure only authorised users can delete resources.

Besides these common methods, there are a few others used in specific cases:

PATCH

Updates part of a resource. It's useful for making small changes without replacing the whole thing, but always validate the data to avoid inconsistencies.

HEAD

Works like GET but only retrieves headers, not the full content. It's handy for checking metadata without downloading the full response.

OPTIONS

Tells you what methods are available for a specific resource, helping clients understand what they can do with the server.

TRACE

Similar to OPTIONS, it shows which methods are allowed, often for debugging. Many servers disable it for security reasons.

CONNECT

Used to create a secure connection, like for HTTPS. It's not as common but is critical for encrypted communication.

URL Path

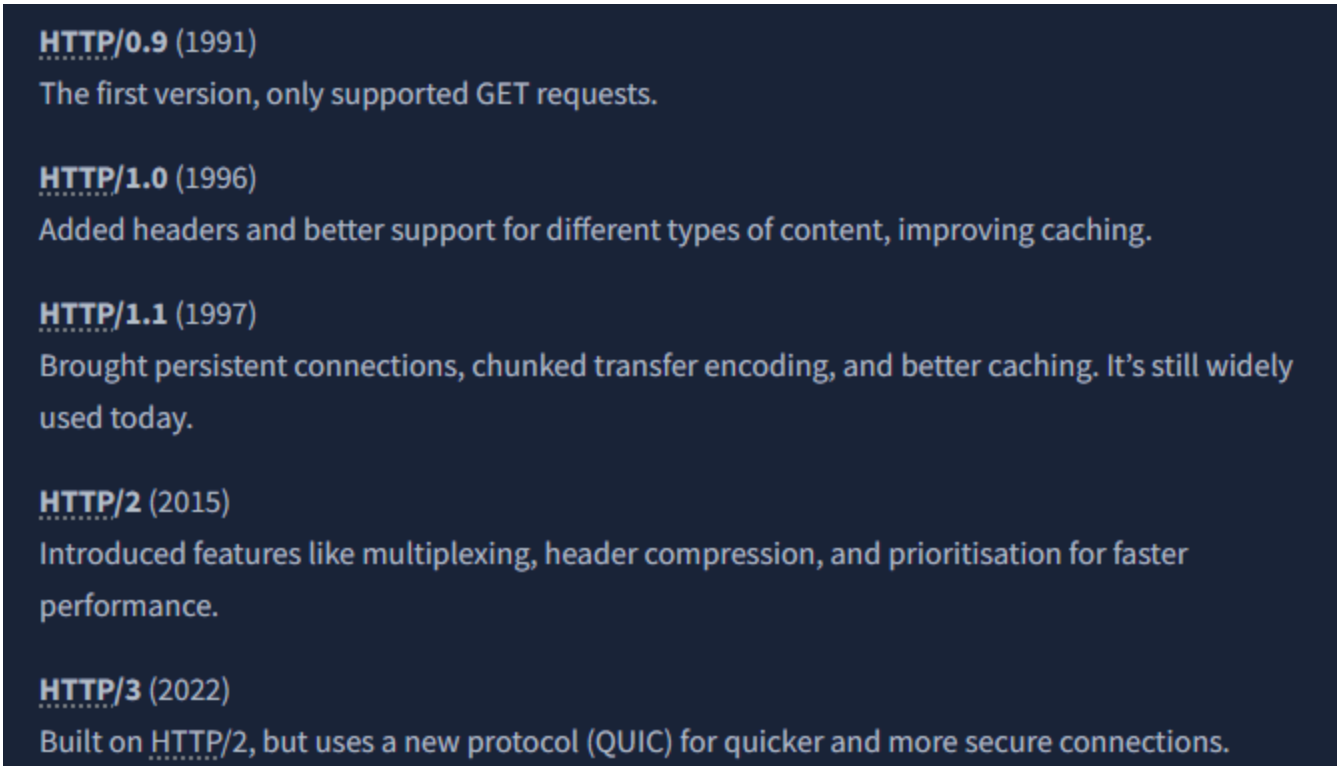
The **URL path** tells the server where to find the resource the user is asking for

Attackers often try to manipulate the URL path to exploit vulnerabilities, so it's crucial to:

- Validate the URL path to prevent unauthorised access
- Sanitise the path to avoid injection attacks
- Protect sensitive data by conducting privacy and risk assessments

HTTP Version

The **HTTP version** shows the protocol version used to communicate between the client and server.



HTTP Request: Headers and Body

Request Headers

Common Request Headers

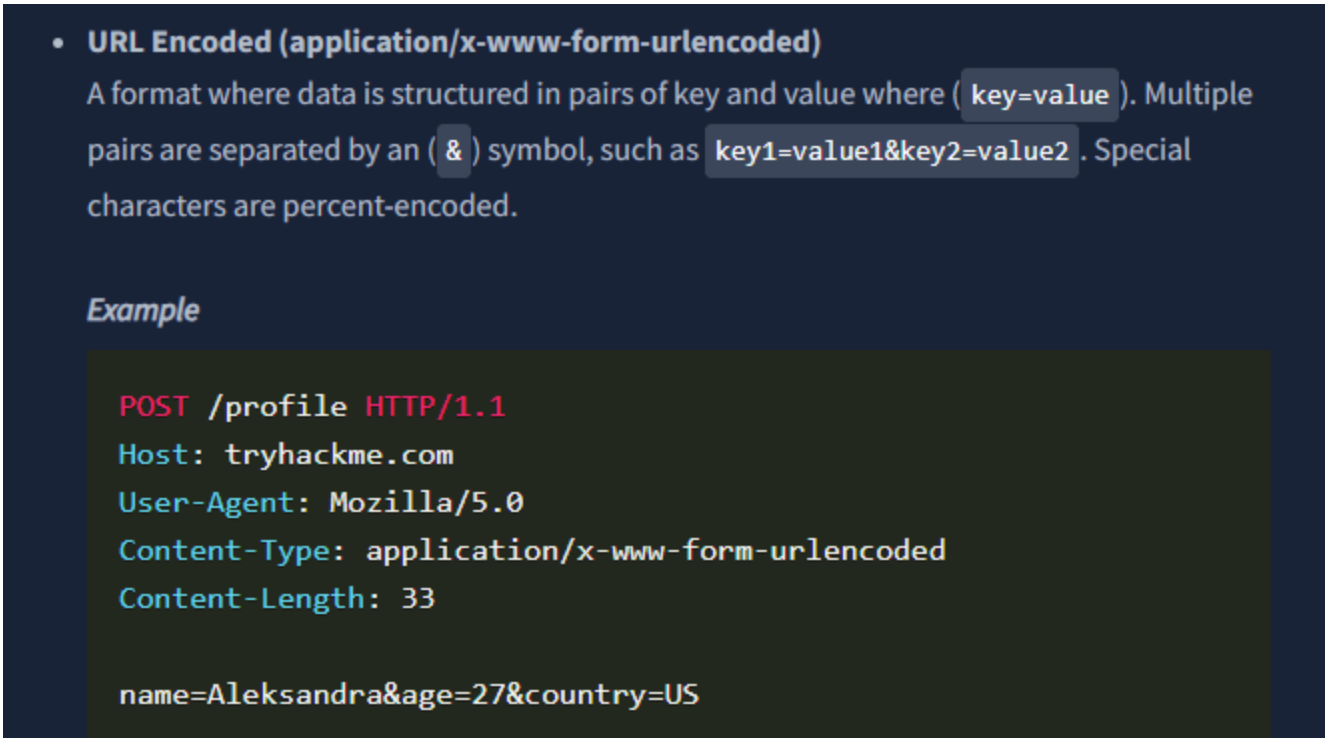
Request Header	Example	Description
Host	`Host: tryhackme.com`	Specifies the name of the web server the request is for.
User-Agent	`User-Agent: Mozilla/5.0`	Shares information about the web browser the request is coming from.

Referer	`Referer: https://www.google.com/`	Indicates the URL from which the request came from.
Cookie	`Cookie: user_type=student; room=introtowebapplication; room_status=in_progress`	Information the web server previously asked the web browser to store is held in cookies.
Content-Type	`Content-Type: application/json`	Describes what type or format of data is in the request.

Request Body

In HTTP requests such as POST and PUT, where data is sent to the web server as opposed to requested from the web server, the data is located inside the HTTP Request Body

The formatting of the data can take many forms, but some common ones are `URL Encoded`, `Form Data`, `JSON`, or `XML`.



- **Form Data (multipart/form-data)**
  Allows multiple data blocks to be sent where each block is separated by a boundary string. The boundary string is the defined header of the request itself. This type of formatting can be used to send binary data, such as when uploading files or images to a web server.

  Example

```
POST /upload HTTP/1.1
Host: tryhackme.com
User-Agent: Mozilla/5.0
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary

----WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="username"

aleksandra
----WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="profile_pic"; filename="alek
Content-Type: image/jpeg

[Binary Data Here representing the image]
----WebKitFormBoundary7MA4YWxkTrZu0gW--
```

- **JSON (application/json)**
  In this format, the data can be sent using the JSON (JavaScript Object Notation) structure. Data is formatted in pairs of name : value. Multiple pairs are separated by commas, all contained within curly braces { }.

  Example

```
POST /api/user HTTP/1.1
Host: tryhackme.com
User-Agent: Mozilla/5.0
Content-Type: application/json
Content-Length: 62

{
    "name": "Aleksandra",
    "age": 27,
    "country": "US"
}
```

- **XML (application/xml)**
  In XML formatting, data is structured inside labels called tags, which have an opening and closing. These labels can be nested within each other. You can see in the example below the opening and closing of the tags to send details about a user called Aleksandra.

  Example

```
POST /api/user HTTP/1.1
Host: tryhackme.com
User-Agent: Mozilla/5.0
Content-Type: application/xml
Content-Length: 124

<user>
    <name>Aleksandra</name>
    <age>27</age>
    <country>US</country>
</user>
```

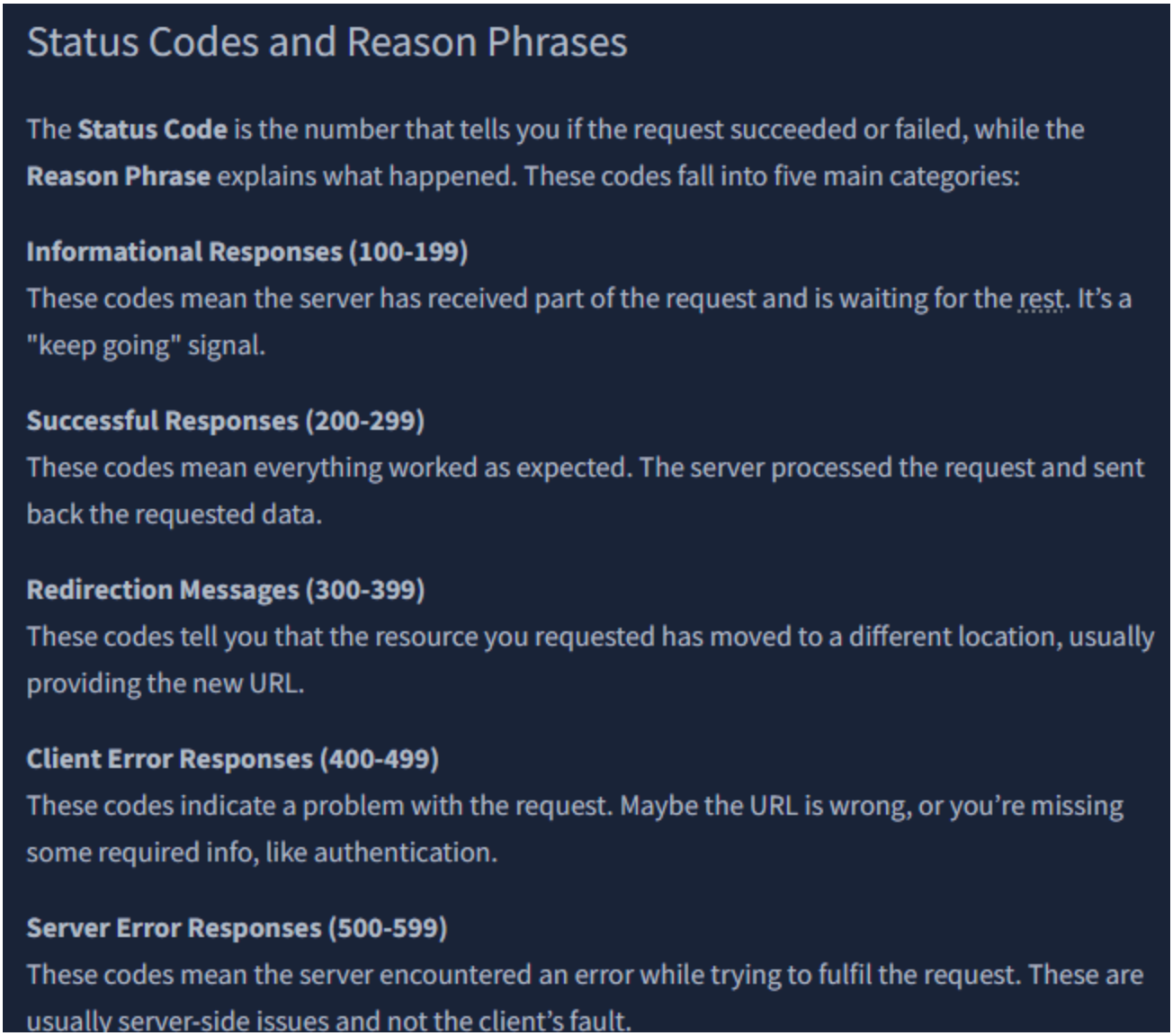
HTTP Response: Status Line and Status Codes

When you interact with a web application, the server sends back an **HTTP response** to let you know whether your request was successful or something went wrong.

Status Line

The first line in every HTTP response is called the **Status Line**. It gives you three key pieces of info:

1. **HTTP Version**: This tells you which version of HTTP is being used.
2. **Status Code**: A three-digit number showing the outcome of your request.
3. **Reason Phrase**: A short message explaining the status code in human-readable terms.



Common Status Codes

Here are some of the most frequently seen status codes:

100 (Continue)
The server got the first part of the request and is ready for the rest.

200 (OK)
The request was successful, and the server is sending back the requested resource.

301 (Moved Permanently)
The resource you're requesting has been permanently moved to a new URL. Use the new URL from now on.

404 (Not Found)
The server couldn't find the resource at the given URL. Double-check that you've got the right address.

500 (Internal Server Error)
Something went wrong on the server's end, and it couldn't process your request.

HTTP Response: Headers and Body

When a web server responds to an HTTP request, it includes **HTTP response headers**, which are basically key-value pairs.

Required Response Headers

Some response headers are crucial for making sure the HTTP response works properly. They provide essential info that both the client and server need to process everything correctly

- **Date**:
  Example: `Date: Fri, 23 Aug 2024 10:43:21 GMT`
  This header shows the exact date and time when the response was generated by the server.
- **Content-Type**:
  Example: `Content-Type: text/html; charset=utf-8`
  It tells the client what kind of content it's getting, like whether it's HTML, JSON, or something else. It also includes the character set (like UTF-8) to help the browser display it properly.
- **Server**:
  Example: `Server: nginx`
  This header shows what kind of server software is handling the request. It's good for debugging, but it can also reveal server information that might be useful for attackers, so many people remove or obscure this one.

Other Common Response Headers

- **Set-Cookie**:
  Example: `Set-Cookie: sessionId=38af1337es7a8`
  This one sends cookies from the server to the client, which the client then stores and sends back with future requests. To keep things secure, make sure cookies are set with the `HttpOnly` flag (so they can't be accessed by JavaScript) and the `Secure` flag (so they're only sent over HTTPS).
- **Cache-Control**:
  Example: `Cache-Control: max-age=600`
  This header tells the client how long it can cache the response before checking with the server again. It can also prevent sensitive info from being cached if needed (using `no-cache`).
- **Location**:
  Example: `Location: /index.html`
  This one's used in redirection (3xx) responses. It tells the client where to go next if the resource has moved. If users can modify this header during requests, be careful to validate and sanitise it—otherwise, you could end up with open redirect vulnerabilities, where attackers can redirect users to harmful sites.

Response Body

The **HTTP response body** is where the actual data lives—things like HTML, JSON, images, etc., that the server sends back to the client. To prevent **injection attacks** like Cross-Site Scripting (XSS), always sanitise and escape any data (especially user-generated content) before including it in the response.

Security Headers

HTTP Security Headers help improve the overall security of the web application by providing mitigations against attacks like Cross-Site Scripting (XSS), clickjacking, and others

- Content-Security-Policy (CSP)
- Strict-Transport-Security (HSTS)
- X-Content-Type-Options
- Referrer-Policy
  https://securityheaders.io/ to analyse the security headers of any website

Content-Security-Policy (CSP)

A CSP header is an additional security layer that can help mitigate against common attacks like Cross-Site Scripting (XSS).

A CSP provides a way for administrators to say what domains or sources are considered safe and provides a layer of mitigation to such attacks.

Within the header itself, you may see properties such as `default-src` or `script-src` defined and many more

Looking at an example CSP header:

```
Content-Security-Policy: default-src 'self'; script-src 'self'
https://cdn.tryhackme.com; style-src 'self'
```

We see the use of:

- **default-src**
  - which specifies the default policy of self, which means only the current website.
- **script-src**
  - which specifics the policy for where scripts can be loaded from, which is self along with scripts hosted on `https://cdn.tryhackme.com`
- **style-src**
  - which specifies the policy for where style CSS style sheets can be loaded from the current website (self)

Strict-Transport-Security (HSTS)

The HSTS header ensures that web browsers will always connect over HTTPS. Let's look at an example of HSTS:

```
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
```

Here’s a breakdown of the example HSTS header by directive:

- **max-age**
  - This is the expiry time in seconds for this setting
- **includeSubDomains**
  - An optional setting that instructs the browser to also apply this setting to all subdomains.
- **preload
  - This optional setting allows the website to be included in preload lists. Browsers can use preload lists to enforce HSTS before even having their first visit to a website.

X-Content-Type-Options

The X-Content-Type-Options header can be used to instruct browsers not to guess the MIME type of a resource but only use the Content-Type header.

```
X-Content-Type-Options: nosniff
```

- **nosniff**
  - This directive instructs the browser not to sniff or guess the MIME type.

Referrer-Policy

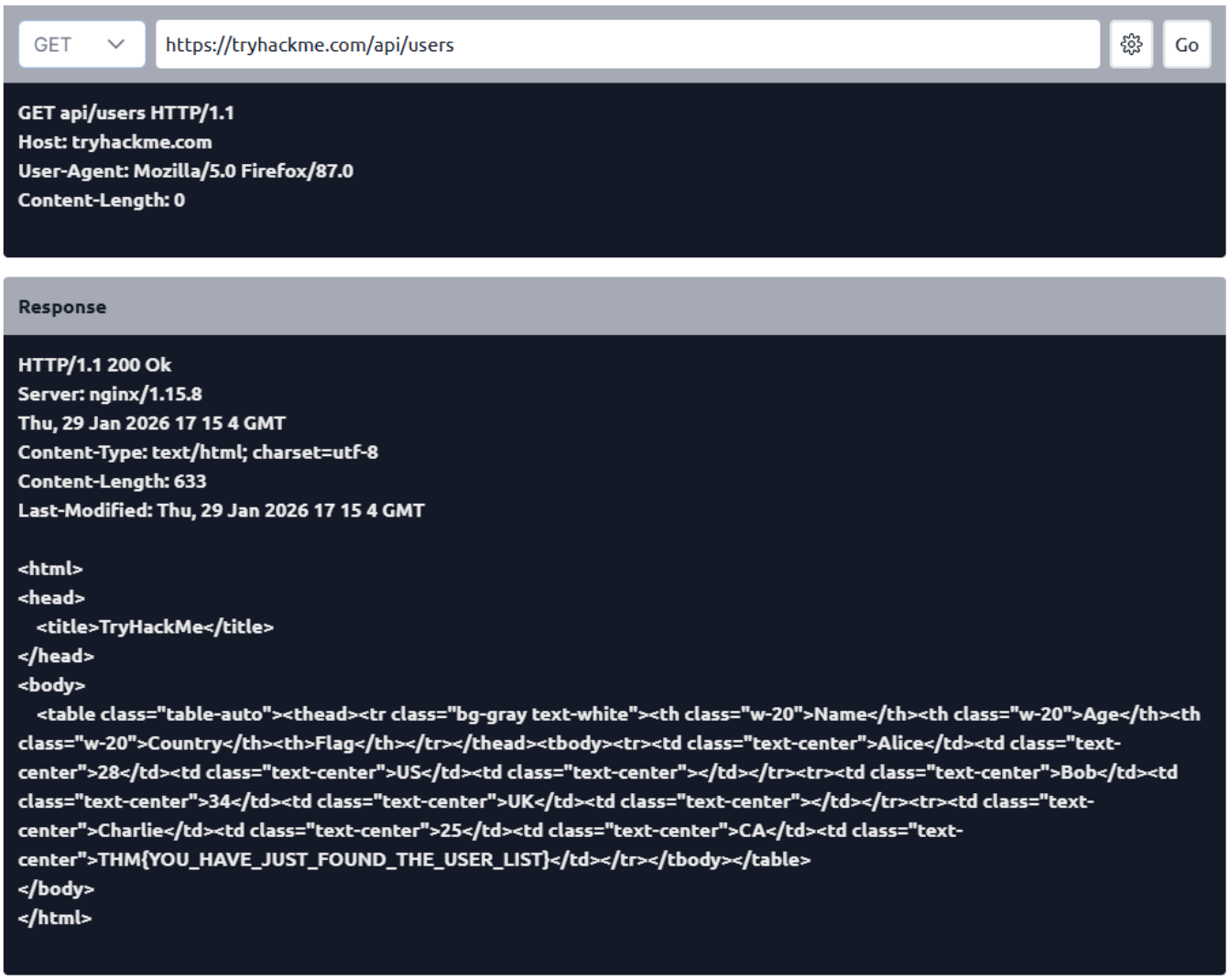
This header controls the amount of information sent to the destination web server when a user is redirected from the source web server, such as when they click a hyperlink

- `Referrer-Policy: no-referrer`
- `Referrer-Policy: same-origin`
- `Referrer-Policy: strict-origin`
- `Referrer-Policy: strict-origin-when-cross-origin`
- **no-referrer**
  - This completely disables any information being sent about the referrer
- **same-origin**
  - This policy will only send referrer information when the destination is part of the same origin. This is helpful when you want referrer information passed when hyperlinks are within the same website but not outside to external websites.
- **strict-origin**

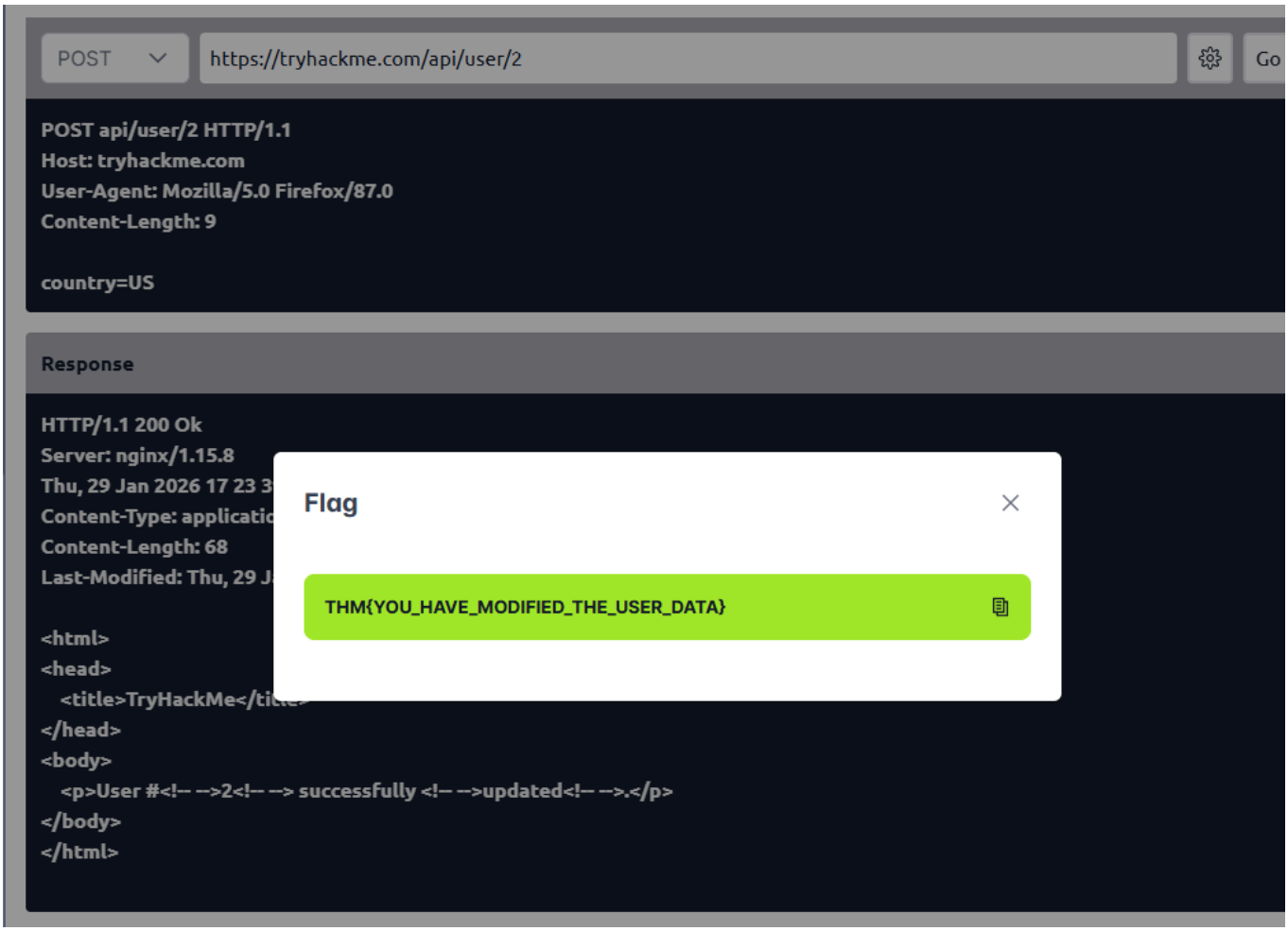
    - This policy only sends the referrer as the origin when the protocol stays the same. So, a referrer is sent when an HTTPS connection goes to another HTTPS connection.
- **strict-origin-when-cross-origin**
    - This is similar to strict-origin except for same-origin requests, where it sends the full URL path in the origin header.

Practical: Making HTTP Requests

I made a get request to the /api/users path and found a flag



I then edited user 2 for a post request to submit the country as US



I made a delete request for user 1