Web Application Basics

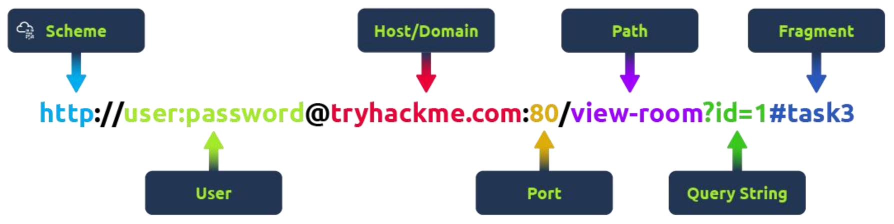
Front End

The **Front End** can be considered similar to the surface of the planet, the parts that an astronaut can see and interact with based on the laws of nature. A web application would have a user interact with it and use a number of technologies such as HTML, CSS, and JavaScript to do this.

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. On a planet, these are the non-visual things: the structures that keep a building standing, the air, and the gravity that keeps feet on the ground.

Uniform Resource Locator



Scheme

The **scheme** is the protocol used to access the website. The most common are HTTP (HyperText Transfer Protocol) and **HTTPS** (Hypertext Transfer Protocol Secure). HTTPS is more secure because it encrypts the connection, which is why browsers and cyber security experts recommend it. Websites often enforce HTTPS for added protection.

User

Some URLs can include a user's login details (usually a username) for sites that require authentication. This happens mostly in URLs that need credentials to access certain resources. However, it's rare nowadays because putting login details in the URL isn't very safe—it can expose sensitive information, which is a security risk.

Host/Domain

The **host** or **domain** is the most important part of the URL because it tells you which website you're accessing. Every domain name has to be unique and is registered through domain registrars. From a security standpoint, look for domain names that appear almost like real ones but have small differences (this is called **typosquatting**). These fake domains are often used in phishing attacks to trick people into giving up sensitive info.

Port

The **port number** helps direct your browser to the right service on the web server. It's like telling the server which doorway to use for communication. Port numbers range from 1 to 65,535, but the most common are **80** for HTTP and **443** for HTTPS.

Path

The **path** points to the specific file or page on the server that you're trying to access. It's like a roadmap that shows the browser where to go. Websites need to secure these paths to make sure only authorised users can access sensitive resources.

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. Since users can modify these query strings, it's important to handle them securely to prevent attacks like **injections**, where malicious code could be added.

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. Users can modify this too, so like with query strings, it's important to check and clean up any data here to avoid issues like injection attacks.

HTTP Messages

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.

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. This line also gives important details about how the message should be handled.

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. These headers cover all sorts of things, like security, content types, and more, making sure everything goes smoothly in the communication.

Empty Line

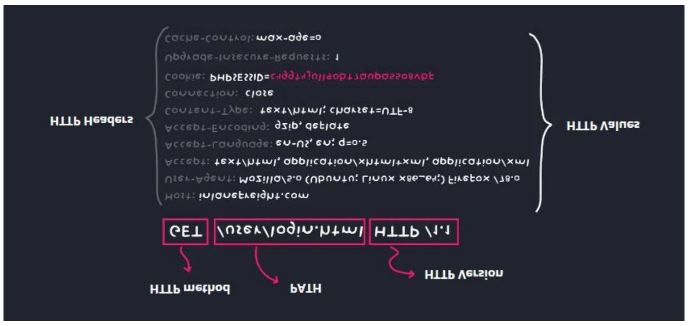
The empty line is a little divider that separates the header from the body. It's essential because it shows where the headers stop and where the actual content of the message begins. Without this empty line, the message might get messed up, and the client or server could misinterpret it, causing errors.

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). In a response, it's where the server puts the content that the user requested (like a webpage or API data).

HTTP Request: Request Line and Methods

Request Line



```
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Cookie: PHPSESSID=c4ggt4jull1obt7aupa55o8vbf
Connection: close
Content-Type: text/html; charset=UTF-8
Accept-Encoding: gzip, deflate
Accept-Language: en-US, en; q=0.5
Accept: text/html, application/xhtml+xml, application/xml
User-Agent: Mozilla/5.0 (Ubuntu; Linux x86_64;) Firefox /78.0
Host: inlanefreight.com
GET /user/login.html HTTP/1.1
```

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. Here are some of the most common methods and their possible security issue:

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.

Each of these methods has its own set of security rules. For example, PATCH requests should be validated to avoid inconsistencies, and OPTIONS and TRACE should be turned off if not needed to avoid possible security risks.

URL Path

The **URL path** tells the server where to find the resource the user is asking for. For instance, in the URL `https://tryhackme.com/api/users/123`, the path `/api/users/123` identifies a specific user.

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

Following these practices helps protect your web application from common attacks.

HTTP Version

The HTTP **version** shows the protocol version used to communicate between the client and server. Here's a quick rundown of the most common ones:

HTTP/**0.9** (1991)
The first version, only supported GET requests.

HTTP/**1.0** (1996)
Added headers and better support for different types of content, improving caching.

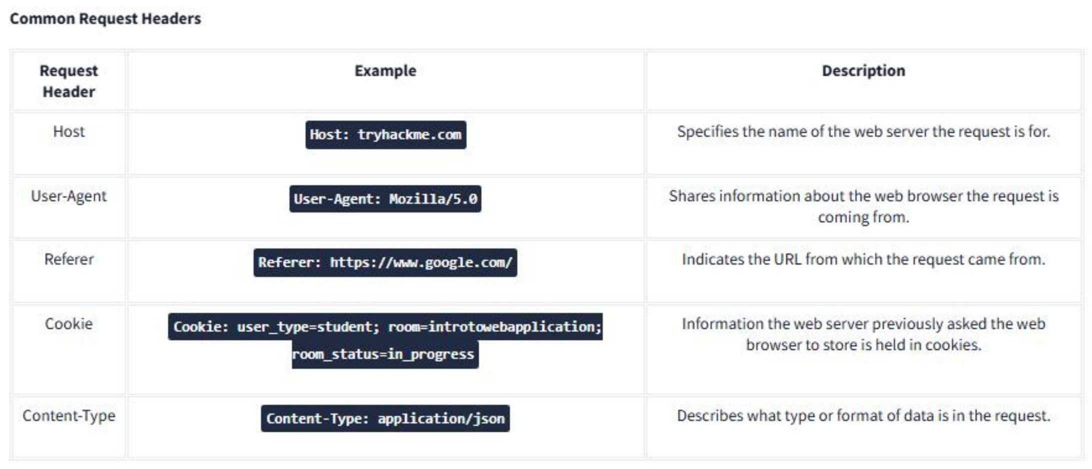
HTTP/**1.1** (1997)
Brought persistent connections, chunked transfer encoding, and better caching. It's still widely used today.

HTTP/**2** (2015)
Introduced features like multiplexing, header compression, and prioritisation for faster performance.

HTTP/3 (2022)
Built on HTTP/2, but uses a new protocol (QUIC) for quicker and more secure connections.

Although HTTP/2 and HTTP/3 offer better speed and security, many systems still use HTTP/**1.1** because it's well-supported and works with most existing setups. However, upgrading to HTTP/2 or HTTP/3 can provide significant performance and security improvements as more systems adopt them.

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`.

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.

Since we already covered HTTP Versions in Task 5, let's focus on the **Status Codes** and **Reason Phrases** here.

Status Codes and Reason Phrases

The **Status Code** is the number that tells you if the request succeeded or failed, while the **Reason Phrase** explains what happened. These codes fall into five main categories:

Informational Responses (100-199)
These codes mean the server has received part of the request and is waiting for the rest. It's a "keep going" signal.

Successful Responses (200-299)
These codes mean everything worked as expected. The server processed the request and sent back the requested data.

Redirection Messages (300-399)
These codes tell you that the resource you requested has moved to a different location, usually providing the new URL.

Client Error Responses (400-499)
These codes indicate a problem with the request. Maybe the URL is wrong, or you're missing some required info, like authentication.

Server Error Responses (500-599)
These codes mean the server encountered an error while trying to fulfil the request. These are usually server-side issues and not the client's fault.

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.

Response Headers

When a web server responds to an HTTP request, it includes HTTP **response headers**, which are basically key-value pairs. These headers provide important info about the response and tell the client (usually the browser) how to handle it.

Picture an example of an HTTP response with the headers highlighted. Key headers like `Content-Type`, `Content-Length`, and `Date` give us important details about the response the server sends back.

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. Here are a few important ones:

-

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

Besides the essential ones, there are other common headers that give additional instructions to the client or browser and help control how the response should be handled.

-

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.