

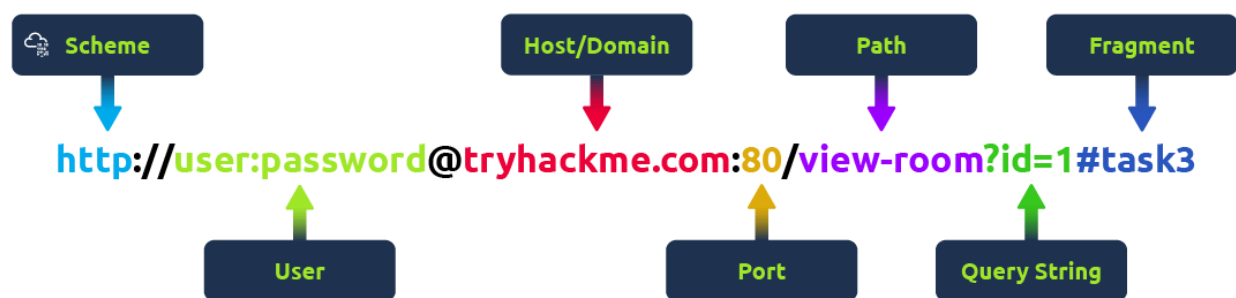
Web Basics

- **Front End** components like **HTML**, **CSS**, and **JavaScript** focus on the experience inside the browser.
- **Back End** components such as the **Web Server**, **Database**, or **WAF** are the engine under the surface that enable the web application to function.

Uniform Resource Locator

- URL is a unique web address that guides an individual to access all kind of online content It's a webpage , video , a photo or other media .

Anatomy of a URL



- **Scheme** : The **scheme** is the protocol used to access the website. (HTTP/HTTPs)
- **User** : Some URLs can include a user's login details (usually a username) for sites that require authentication, it's rare nowadays because putting login details in the URL isn't very safe
- **Host/Domain** : The **host** or **domain** is the most important part of the URL because it tells you which website you're accessing. 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.
- **Query-String**: It 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 Request: Request Line and Methods



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 authorized to make changes before accepting the request.
- **DELETE**
***Removes** something from the server. Reminder! Just like with PUT, make sure only authorized 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.

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

- **URL Encoded (application/x-www-form-urlencoded)**

A format where data is structured in pairs of key and value where (key=value). Multiple pairs are separated by an (&) symbol, such as key1=value1&key2=value2 . Special characters are percent-encoded.

```
POST /profile HTTP/1.1
Host: tryhackme.com
User-Agent: Mozilla/5.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 33
```

```
name=Aleksandra&age=27&country=US
```

- **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.**

```
POST /upload HTTP/1.1
Host: tryhackme.com
User-Agent: Mozilla/5.0
Content-Type: multipart/form-data; boundary=----
WebKitFormBoundary7MA4YWxkTrZu0gW

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

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

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

HTTP Response: Status Line and Status Codes

```
HTTP Request

POST /login HTTP/1.1
Host: tryhackme.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 43

username=aleksandra&password=securepassword
```

```
HTTP Response

HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 56
Date: Wed, 29 Aug 2024 12:00:00 GMT

{
  "message": "Login successful!",
  "status": "success"
}
```

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.

Security Headers

- Content-Security-Policy (CSP)
- Strict-Transport-Security (HSTS)
- X-Content-Type-Options
- Referrer-Policy

Content-Security-Policy (CSP)

- A CSP header is an additional security layer that can help mitigate against common attacks like **Cross-Site Scripting (XSS)**. Malicious code could be hosted on a separate website or domain and injected into the vulnerable website.
- 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.

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 specifies 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. Here's an example:

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

Here's a breakdown of the X-Content-Type-Options header by directives:

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

MIME-type sniffing is when a browser tries to guess the content type of a file instead of strictly following the `Content-Type` sent by the server.