

HTTP

HTTP (Hypertext Transfer Protocol)

This chapter takes our exploration further, focusing on the client-server model, which we've touched upon in previous lessons, and remote procedure calls (RPC). A special emphasis is placed on two particular application layer protocols – HTTP and Web Sockets. Both of these protocols are crucial in system design interviews and thus warrant a comprehensive understanding.

Client-Server Model

The client-server model, as we've discussed so far, often presents the client as a user utilizing a web browser to issue a request to a server. The server, typically another computer, responds to these requests. However, it's crucial to understand that the client in a client-server model doesn't necessarily have to be a web browser.

1. **Client:** The client is an application or system that accesses a service made available by a server. The client can be a web browser, an email software, an app on your phone, or any other software that needs to access some service. The term "client" can also refer to the device running this application. For example, your computer or smartphone can be a client when you're browsing the internet. The client typically initiates communication, sending a request to the server for specific information or services.
2. **Server:** The server is a computer, device, or software that provides resources, data, services, or functionality to the client or other servers on a network. Servers wait for incoming requests from clients and respond by fulfilling those requests. For instance, a web server delivers web pages to clients, an email server handles sending and receiving emails, and a database server provides clients with access to database services.

Based on context, the roles of client and server can interchange. For instance, a single machine can function both as a client, requesting resources, and a server, providing resources. This is observable when you enter a search query into Google, where google.com may request data from a third party, thereby assuming the role of a client. Peer-

to-peer (P2P) networks offer another example, where a machine can simultaneously act as both a client and a server.

The client is often labeled the “caller” as it initiates requests, while the server, being the entity called upon to deliver services or process requests, is considered the “callee”.

RPCs (Remote Procedure Call)

A remote procedure call (RPC) provides the ability for a program to perform functions on a separate machine, making it a simple and efficient solution for task management in distributed systems, where programs operate across multiple computers. Within the client – server model, both the client and the server maintain their connection via a network.

Consider a practical scenario: suppose we input “Jahong is cool” into the YouTube search bar, upon which the browser displays a list of all the Jahong’s videos. The code responsible for this listing operation doesn’t reside within our browser. This is because the browser isn’t where the videos are stored; instead, this code is situated on YouTube’s servers. The operation may utilize a function such as `listVideos(`Jahong`)`. Even though it may seem as if the code is executing on the client side, it is actually making a call to YouTube’s servers in the background to retrieve the relevant information.

HTTP

The Hypertext Transfer Protocol is built on top of IP and TCP. This is the protocol that we as developers have some control over. Recall that at the basic level, IP is responsible for delivering packets of data, from the source to the destination but doesn’t worry about the order in which packets arrive. TCP, which is built on top of IP, ensures that the packets are delivered in the order. HTTP sits on top of TCP. At its core, it is a request/response protocol. It is a set of rules for how data should be formatted and transmitted over the web, along with how the servers and the browsers should respond to different commands. HTTP is used every time we make a call through the browser.

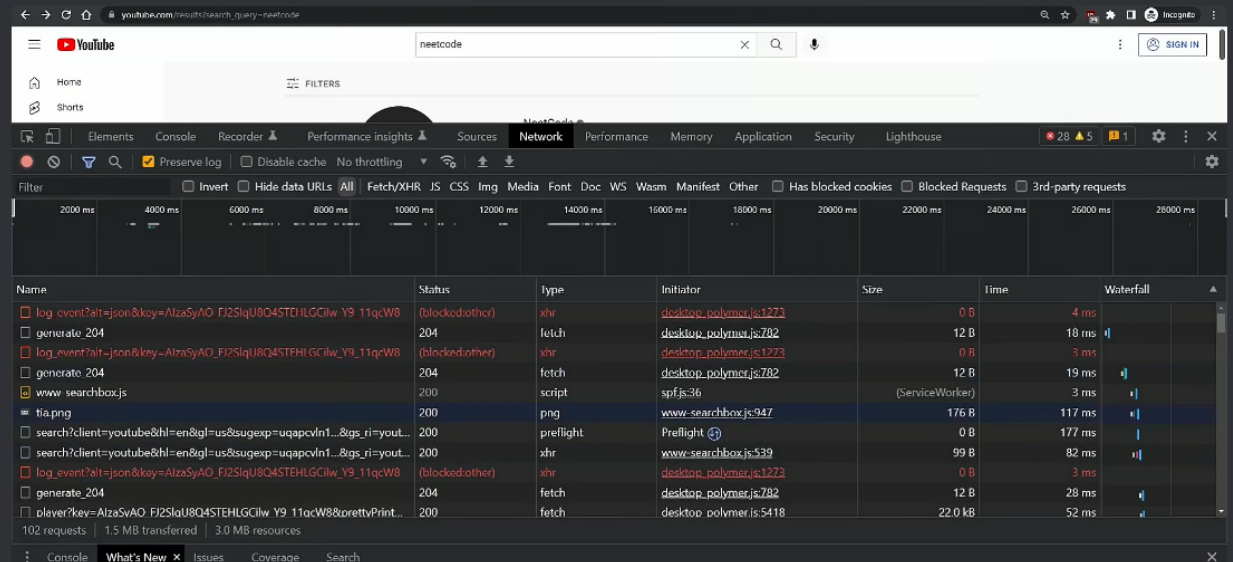
Let’s look at an example:

Understanding HTTP through Browser Developer Tools

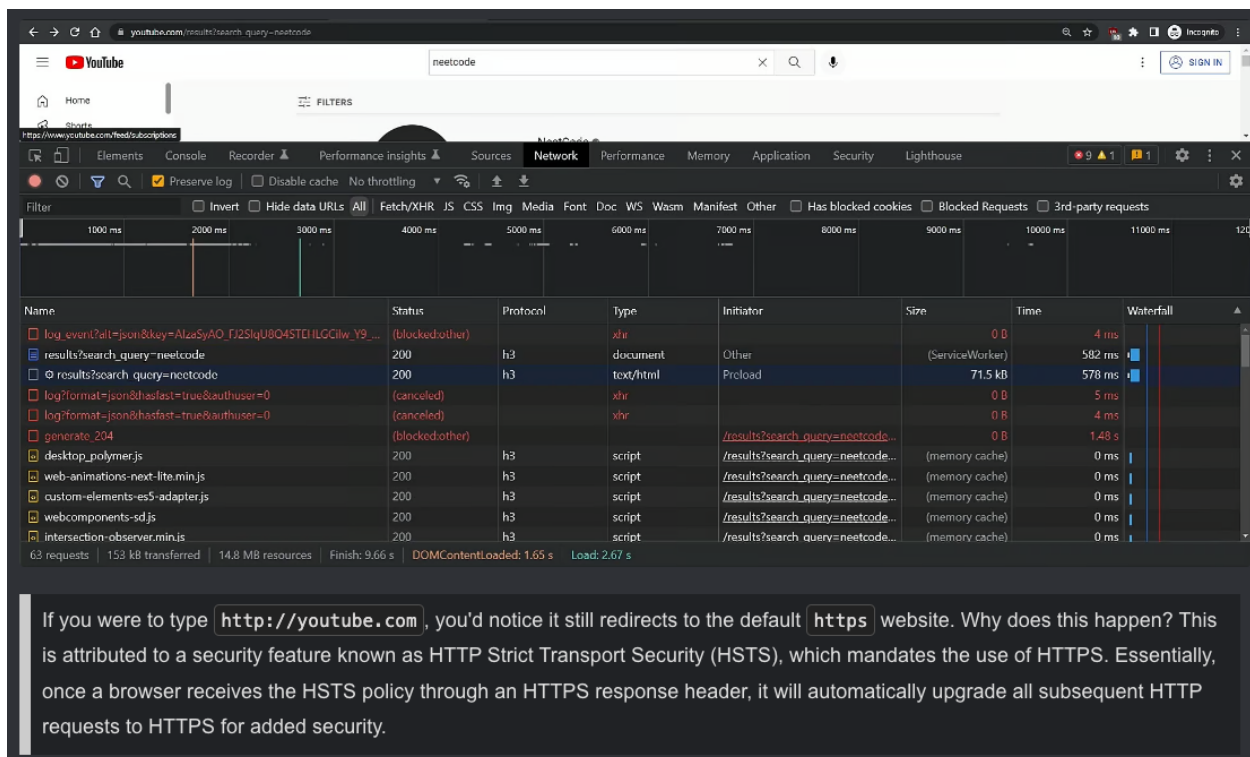
Entering `https://youtube.com` into your web browser takes you to YouTube's homepage. You'll find that most websites, including YouTube, default to `https` as their scheme. The element that we're particularly interested in is the developer tools. You can access these by right-clicking anywhere on the page and selecting `inspect`, which will open a new window or pane in your browser.

This may seem a bit daunting if it's your first encounter with developer tools. The panel we're focusing on is the network tab. Designed to monitor and present all network requests transmitted to and from the browser while active, this tab covers a range of request types, including HTML, CSS, JavaScript, images, APIs, and more.

Opening the Network tab reveals key details such as the `Name` of the request, the status code, the request type, the size of the response, and the time it took to fulfill the request. This is visually conceptualized in the figure below.



The diagram above does not explicitly show the scheme being utilized. However, you can reveal this by right-clicking within the developer tools and selecting 'protocol'. This action will reveal `h3` in the updated display, indicating the usage of HTTP/3. This is the most recent version of the protocol.



If you were to type `http://youtube.com`, you'd notice it still redirects to the default `https` website. Why does this happen? This is attributed to a security feature known as HTTP Strict Transport Security (HSTS), which mandates the use of HTTPS. Essentially, once a browser receives the HSTS policy through an HTTPS response header, it will automatically upgrade all subsequent HTTP requests to HTTPS for added security.

HTTP Methods

When we interact with an endpoint URL like <https://youtube.com>, different methods allow us to perform various operations. HTTP provides four core methods that HTTPS also employs:

1. **GET:** A GET method retrieves a resource. It is defined as idempotent, meaning that making the same GET request multiple times should always return the same result.
2. **POST:** A POST method will send data to the server to create a new resource. A POST request will have a request body (payload) which indicates the data being sent over the internet. This payload is not passed as part of the URL to prevent Man in the middle attacks (which we will discuss soon). The POST method is not idempotent
3. **PUT:** The PUT method is used to update a resource with the provided data. An important characteristic of the PUT method is that if the same update operation is performed multiple times using the same data, the state of the resource will remain unchanged. This property is known as idempotence. Why is it considered idempotent? When the same PUT request is repeated, the outcome remains consistent—either the resource will be created or updated to the same state. For example, updating a user's email multiple times with the same value would still be

considered idempotent since sending the request repeatedly would result in the same state.

4. **DELETE:** A delete method takes parameters and deletes the specified data. Delete is idempotent. The first call may return a status code 200 (ok) and any additional delete calls will return a 404 (not found). So, the response might be different but there is no change of state.

HTTP Status Codes

HTTP status codes are a way for the server to inform the client about the state of their requests. Status codes are divided into categories, which help developers understand the state of their request. Below are the network tiers:

1. Information responses (100 - 199)

Informational responses are utilized to acknowledge that the client's request has been received and is currently being processed. Some important informational response codes include:

- **100 Continue** : This response indicates that everything is in order so far, and the client should proceed with the request or disregard it if it has already been completed.
- **101 Switching Protocols** : This response indicates that the server is switching to a different protocol as specified in the upgrade request header received from the client.

These informational response codes play a crucial role in the communication between the client and the server, ensuring smooth and effective request processing.

2. Successful responses (200 - 299)

Any code at the **200** level depicts a successful response.

- **200 OK** : This response code indicates that the request has succeeded. It signifies that the request has been processed and the server is returning the requested data.
- **201 Created** : This response code indicates that the request has succeeded, resulting in the creation of a new resource. It is typically sent after POST or PUT requests, confirming that the resource has been successfully created or updated.

3. Redirection messages (300 - 399)

Redirection messages are used when the requested resource has been assigned a new permanent URL or is temporarily available at a different URL.

- **300 Multiple Choices** : This response code means that the request has more than one possible response. The client should choose one of them.
- **301 Moved Permanently** : This response code means that the requested resource has been permanently moved to a new location, and the server is redirecting the client to this new location.

4. Client error responses (400 - 499)

- **400 Bad Request** : This response code is used when the server encounters a client request that is invalid or cannot be understood. It often occurs when incorrect parameters are passed in the request, leading to a bad request.
- **401 Unauthorized** : This response code is returned when a client attempts to access a protected resource without proper authentication or authorization. For example, if you try to delete a video that you are not authorized to delete, the server will respond with a 401 Unauthorized status code.

5. Server error responses (500 - 599)

Server error responses in the range of 500-599 indicate that an error has occurred on the server side while processing the client's request. These response codes generally indicate issues or failures within the server that prevented it from fulfilling the request.

SSL/TLS

Let's now delve into SSL/TLS and explore how it operates in conjunction with HTTPS. TLS, or Transport Layer Security, facilitates secure communication over a network. We will only be discussing this on a high level because it is not super important for system design.

The reason we need SSL and TLS is because HTTP on its own is vulnerable to man-in-the-middle (MITM) attacks. 'Normal' HTTP requests are analagous to a bag of money in a transparent box; anyone can see the contents inside. HTTPS, however, prevents this from happening.

Together, TLS and SSL ensure that web data remains inaccessible and unalterable by unauthorized parties. HTTPS is essentially the combination of HTTP with TLS.

While SSL is often used interchangeably with TLS, it's worth noting that SSL is technically an outdated term, superseded by TLS.

So, in the grand scheme of things, the protocol that ultimately proves most useful for developers is HTTPS. This is the most important takeaway from this section.