HTTP

HTTP Request Methods

- HTTP request methods tell the server what action the client wants to perform. Each method has a different purpose and behavior. It defines the **action** the client wants to perform on the server. It helps manage **data retrieval, creation, modification, and deletion** efficiently.
- **GET** = Retrieves data from the server. No request body is allowed.
- **POST** = Sends new data to the server (e.g., submitting a form or creating new data).
- **PUT** = Updates existing data by replacing the entire resource.
- **PATCH** = Partially updates existing data, modifying only specific fields.
- **DELETE** = Removes data from the server. No request body is allowed.
- **HEAD** = Similar to GET, but only returns response headers, not the body.
- **OPTIONS** = Retrieves allowed methods for a resource, often used in CORS preflight requests.

Question

- **To create a search API for an e-commerce platform, which request method should be used?**
- **Answer:** Although GET seems appropriate, it has a character limit of 256 bytes. Since search APIs can have multiple filters, **POST** is a better choice.

HTTP Headers

What are HTTP Headers?

- HTTP headers are key-value pairs sent between the client (e.g., a browser) and the server to provide additional information about the request or response.
- **Request Headers**: Sent by the client to provide metadata about the request.
- **Response Headers**: Sent by the server to provide metadata about the response.

How HTTP Headers Work

Common Request Headers

- **Connection: Keep-Alive / Close**
  - Allows the HTTP connection to persist across multiple requests.
  - If `Keep-Alive` is set, subsequent requests do not need to establish a new connection.
  - If the server sends this header in the response, it can be used for **Server-Sent Events (SSE)** to enable real-time updates on the client side.
- **Accept**
  - Specifies the types of content the client can process.
  - Example: `Accept: text/html, application/json`.
- **Authorization**
  - Contains credentials for authenticating the client.
  - Example: `Authorization: Bearer <token>`.
- **Cookie**
  - Sends stored cookies from the client to the server.
  - Example: `Cookie: sessionId=abc123`.
- **User-Agent**
  - Provides information about the client making the request.

- Example: `User-Agent: Mozilla/5.0`.

Common Response Headers

- **Set-Cookie**
  - Used by the server to set cookies on the client.
  - Example: `Set-Cookie: sessionId=abc123; HttpOnly`.
- **Content-Type**
  - Specifies the type of content being returned.
  - Example: `Content-Type: application/json`.
- **ETag**
  - Used for **cache validation** and **content optimization**.
  - Helps browsers determine whether content has changed since the last request.

Custom Headers

- Developers can define custom headers for application-specific needs.
- Example: `X-Custom-Header: customValue`.

Why Are HTTP Headers Important?

- HTTP is a **stateless protocol**, meaning the server does not retain session information.
- Headers provide additional metadata to make requests **statefull**, enabling functionalities like:
  - Authentication
  - Caching
  - Content negotiation
  - Real-time communication

Status Codes

- **HTTP status codes** are **three-digit numbers** included in the **HTTP response** from a server to indicate the result of the request made by the client.
- **Status Codes**
  - 1xx - information server has sent but not data that client is using
  - 2xx - Success
  - 3xx - Redirection
  - 4xx - You are Wrong (page not found, unauthorized, forbidden)
  - 5xx - I am wrong (Bad Gateways, Internal server error)

Code	Meaning	Category
100	Continue	1xx (Informational)
101	Switching Protocols	1xx (Informational)
102	Processing	1xx (Informational)
200	OK	2xx (Success)
201	Created	2xx (Success)
202	Accepted	2xx (Success)
204	No Content	2xx (Success)
301	Moved Permanently	3xx (Redirection)
302	Found	3xx (Redirection)
303	See Other	3xx (Redirection)
304	Not Modified	3xx (Redirection)
307	Temporary Redirect	3xx (Redirection)
308	Permanent Redirect	3xx (Redirection)
400	Bad Request	4xx (Client Error)
401	Unauthorized	4xx (Client Error)

403	Forbidden	4xx (Client Error)
404	Not Found	4xx (Client Error)
405	Method Not Allowed	4xx (Client Error)
406	Not Acceptable	4xx (Client Error)
408	Request Timeout	4xx (Client Error)
409	Conflict	4xx (Client Error)
410	Gone	4xx (Client Error)
411	Length Required	4xx (Client Error)
412	Precondition Failed	4xx (Client Error)
413	Payload Too Large	4xx (Client Error)
414	URI Too Long	4xx (Client Error)
415	Unsupported Media Type	4xx (Client Error)
429	Too Many Requests	4xx (Client Error)
500	Internal Server Error	5xx (Server Error)
501	Not Implemented	5xx (Server Error)
502	Bad Gateway	5xx (Server Error)
503	Service Unavailable	5xx (Server Error)
504	Gateway Timeout	5xx (Server Error)
505	HTTP Version Not Supported	5xx (Server Error)

Cookies

- Cookies are a way to store data on the client side, similar to local storage, session storage, and Indexed DB, but they have some key differences:
  - **Storage Limit:** Not more than 4 KB
  - **Creation:** Can be created by both Client and Server (server instructs the client to create cookies)
  - **Format:** Key-Value Pair (Both are stored as strings)

- Cookies are sent with every HTTP request to the domain that set them, which can impact performance and increase request sizes.
- **Cookie Attributes:**
  - **HttpOnly (Server Side):** Cannot be accessed by Client-Side JavaScript. This helps prevent XSS attacks. To send them with requests, use `credentials: 'include'` in fetch or `withCredentials: true` in axios.
  - **SameSite:** Restricts how cookies are sent with cross-site requests to help prevent CSRF attacks.
    - `SameSite=Strict`: The cookie is **only sent** in requests originating from the same site (main domain and subdomains).
    - `SameSite=Lax`: The cookie is sent in same-site requests (main domain & subdomains) and some cross-site requests (like links or top-level navigation).
    - `SameSite=None`: The cookie is sent in **all requests (client and server can have different main domains)** but must also have the `Secure` flag set.
  - **Secure:** Ensures the cookie is only sent over HTTPS.
  - **Max-Age:** Defines the duration (in seconds) until the cookie expires.
  - **Expires:** Defines a specific expiration date and time for the cookie.
  - **Path:** Specifies the request path for which the cookie is valid. A cookie with `Path=/app` is sent with requests to `/app/*` but not `/dashboard`.
  - **Domain:** Specifies the domain for which the cookie is valid. To make it available for all subdomains, use `Domain=.codersgyan.com`.

Why we need cookies?

- **Authentication:** Cookies are widely used to store session tokens, allowing users to stay logged in.
- **User Preferences:** They can store user settings and preferences, such as language choices or theme settings.
- **Tracking:** Third-party cookies are often used for tracking users across different websites to improve ad targeting and user experience.

Etag

- **ETag** is a **Server-Side Header**.
- It identifies whether the version of a resource cached in the browser matches the resource on the web server.

How Etag Works?

1. When a client visits a web page, the server sends an **ETag** value.
2. **Weak ETags:** Indicate that a cached resource is **semantically equivalent** to the server version but may not be byte-for-byte identical.
3. **Strong ETags:** Ensure the resource in the browser cache and on the web server are **byte-for-byte identical**.
4. When the client makes a request, it may send an `If-None-Match` header with the previously received ETag value.
5. If this value matches the server's ETag, the server responds with a `304 Not Modified` status, meaning the client can use the cached resource.

Why we need Etag?

- **Caching and Performance Optimization**: Reduces bandwidth usage and improves resource management efficiency.

TLS

- **TLS (Transport Layer Security)** and **SSL (Secure Sockets Layer)** are cryptographic protocols used to secure communication over a network, most commonly the internet.
- They ensure **encryption** of data exchanged between a client and a server.
- **HTTPS** uses **TLS** for encryption, **not SSL**.

- **SSL** was the original protocol for secure communication but is now outdated and insecure.
- **TLS** was developed as the successor to SSL and is now the standard protocol used to secure HTTPS connections.
- Although HTTPS uses **TLS**, it is still sometimes referred to as **SSL** in everyday conversations.
- **HTTP/1.0 & 1.1:** Technically support TLS 1.0-1.3, but should use 1.2 or 1.3 due to security risks with older versions. 1.0 and 1.1 are deprecated.
- **HTTP/2:** Requires TLS 1.2 or 1.3.
- **HTTP/3:** Uses only TLS 1.3.
