HTTP Request

An **HTTP (HyperText Transfer Protocol) request** is a message sent by a client (e.g., browser, mobile app) to a server to request some data or perform an action.

4 Types of Requests
GET = get something from the backend
POST = create something
PUT = update something
DELETE = delete something

Method	Description
GET	Requests data from a server (e.g., webpage, API data)
POST	Sends data to the server (e.g., form submission)
PUT	Updates an existing resource
DELETE	Removes a resource from the server
PATCH	Partially updates an existing resource
HEAD	Similar to `GET`, but only returns headers (no body)

- **GET** is an **HTTP request method** used to retrieve data from a server. **No body** is sent in a GET request.

```
// get request
const xhr = new XMLHttpRequest();
xhr.addEventListener('load', ()=> {
    console.log(xhr.response);
});
xhr.open('GET', 'https://supersimplebackend.dev/greeting');
xhr.send();
```

- **POST** is an **HTTP request method** used to **send data** to a server. **Data is sent in the request body**, not in the URL. Commonly used in **forms, APIs, authentication, and data submission**.

```
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json' // Specifies JSON format
  },
  body: JSON.stringify({
    name: "John Doe",
    email: "john@example.com"
  })
})
.then(response => response.json())  // Convert response to JSON
.then(data => console.log("Success:", data))  // Handle the response
.catch(error => console.error("Error:", error));
```

Here, we send user data (name, email) to the server.
The server processes it and **creates a new user**.

POST request in HTML form

```
<form action="https://example.com/register" method="POST">
  <input type="text" name="username" placeholder="Enter Username">
  <input type="password" name="password" placeholder="Enter Password">
  <button type="submit">Register</button>
</form>
```

When the user submits the form, the **data is sent to the server** in the request body.

```
// post request
async function postGreeting() {
    const response = await fetch('https://supersimplebackend.dev/greeting', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            name: 'Priyanshi'
        })
    });
    const order = await response.text();
    console.log(order);
}
postGreeting();
```

fetch?

fetch() is a JavaScript function used to send HTTP requests. It supports all request methods (GET, POST, PUT, DELETE, etc.).

It returns a **Promise** that resolves to a Response object.

Fetch is same as xmlhttprequest that is used to make backend request but fetch uses promises

```
// fetch request
fetch('https://supersimplebackend.dev/greeting')
.then((response) => {
    return response.text();
}).then((text) => {
    console.log(text);
});
```

```
// fetch using async await
async function getGreeting() {
    const response = await fetch('https://supersimplebackend.dev/greeting');
    const op = await response.text();
    console.log(op);
}
getGreeting();
```

fetch() = better way to make HTTP requests

headers gives the backend more information about our request.

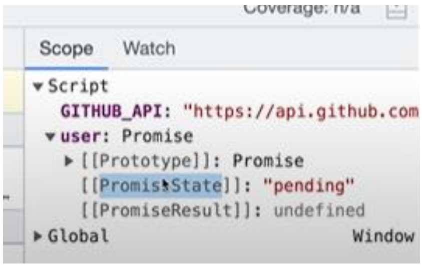
What type of data we are sending to our json

```
method: 'POST',
headers: {
  'Content-Type': 'application/json'
}
)
```

Fetch? It is an API given by browsers to us to make external calls. In below example we will use fetch function to make an API call to github servers and we will get a user info with us.

```
const GITHUB_API = "https://api.github.com/users/akshaymarch7"

const user = fetch(GITHUB_API);
```



XSS (Cross-Site Scripting)

It is a **security vulnerability** where an attacker injects **malicious scripts (JavaScript)** into a trusted website.

When other users visit that page, **the malicious script runs in their browser** — without their permission.

How does XSS happen?

Suppose there is a website with a **comment section.**

When a user submits a comment, the website **directly adds the comment** to the page without cleaning it. (FYI, this is a dom based XSS)

```html
<!-- comment.html -->
<form id="commentForm">
  <input type="text" id="userComment" placeholder="Write your comment">
  <button type="submit">Post Comment</button>
</form>

<div id="comments"></div>

<script>
  document.getElementById('commentForm').addEventListener('submit', function(e) {
    e.preventDefault();
    var comment = document.getElementById('userComment').value;
    document.getElementById('comments').innerHTML += `<p>${comment}</p>`;
  });
</script>
```

Now, when a user writes something and clicks "Post Comment", the comment is inserted into the page using **innerHTML. No filtering** or **escaping** is done.

Attacker enter this as a comment: **<script>alert('You are hacked!')</script>**
Result on the page: <p><script>alert('You are hacked!')</script></p>
Browser sees `<script>` tag and executes it immediately.

Instead of just alert, attackers can: Steal cookies, Redirect users to phishing sites, Install malware, Steal sessions.

How to fix/prevent XSS?
Instead of innerHTML, use **textContent**:

```javascript
document.getElementById('comments').textContent += comment;
```

Or use a **library like DOMPurify** to sanitize the input

```javascript
var cleanComment = DOMPurify.sanitize(comment);
document.getElementById('comments').innerHTML += `<p>${cleanComment}</p>`;
```

Sanitize the input: Remove or escape any potentially dangerous characters.
Set HTTP Headers like Content-Security-Policy (CSP) to restrict what scripts can run.
Use innerText/textContent instead of innerHTML if you don't need HTML interpretation.
Validate input on both client and server side

Types of XSS attacks:

1. **Stored XSS**

Malicious script is stored permanently in the server/database (like in a comment, profile info, etc).
When **other users** visit the page, the malicious script **automatically executes**.
Example: Imagine a comment section: A user submits a blog comment.
The comment is saved to the database **without sanitization.** When another user views the blog, the malicious script runs.

```php
// comment.php (stores comment to DB)
$comment = $_POST['comment'];
mysqli_query($conn, "INSERT INTO comments (text) VALUES ('$comment')");
```

Frontend rendering the comment

```html
<!-- comments.html -->
<div id="comments">
  <p><?php echo $comment['text']; ?></p>
</div>
```

Malicious input

```html
<!-- Attacker submits a comment like this -->
<script>alert('You are hacked!')</script>
```

The script is saved in the DB and runs every time someone views the comment section

Stored XSS is the most dangerous because it affects **many users** without them doing anything.

2. **Reflected XSS**
Occurs when malicious scripts are embedded in a URL, which executes when the URL is visited.
Example: Attack link → http://site.com/?name=<script>alert('hacked')</script>

Imagine a website shows the `search` keyword from the URL:

```
<h1>Search results for: <span id="searchTerm"></span></h1>

<script>
  const params = new URLSearchParams(window.location.search);
  document.getElementById('searchTerm').innerHTML = params.get('q');
</script>
```

Now, attacker shares a **fake URL** like:

```
https://example.com/search?q=<script>alert('Hacked!')</script>
```

When user **clicks the link**, the script **executes immediately**.

Reflected XSS needs social engineering — user must click on a bad link

3. **DOM-based XSS**
Happens purely **on the client-side (browser)**, using **JavaScript DOM manipulation**.
Example: A JS code that trusts location.search and directly inserts it into innerHTML.

```
<!-- HTML -->
<div id="output"></div>

<script>
  // Directly taking URL hash and putting it into DOM
  document.getElementById('output').innerHTML = location.hash.substring(1);
</script>
```

Now if someone opens:

```
https://example.com/#<script>alert('DOM XSS')</script>
```

As soon as page loads, the script **runs**.

In DOM XSS, **attack happens inside browser memory**, no request goes to server.

CSRF (Cross-Site Request Forgery)

CSRF is an attack where a **trusted user's browser** is **tricked into making an unwanted request** to a web application **where the user is already authenticated** (logged in).
It **forces the user** to perform actions they didn't intend to, like changing their email, making a purchase, or even transferring money.

How CSRF Happens (Step-by-Step)

User logs into a trusted website (like their bank account) — and their browser saves the login session via a **cookie**.
While still logged in, the user visits a malicious website (attacker's site).
That malicious site **secretly sends a request** (like a form submission) **to the bank's server**, using the **user's session cookie** (because browser sends cookies automatically).
The bank's server **trusts the session** (because it sees the valid cookie) and **executes the malicious action**, thinking the request is legitimate.

In short,
CSRF = attacker tricks your browser into sending a request **you didn't mean** to send. Browser **blindly attaches** your **login cookie** to the request. Bank thinks it's a **genuine** request from you.

Example: Imagine you're logged into your bank website.
The attacker's website contains:

```
<img src="https://bank.com/transfer?amount=1000&to=attackerAccount">
```

As soon as the user opens the malicious site, **this request fires automatically**!
Browser **sends cookies** with the request, so bank server **thinks** it's a real transfer request from the user.

How to Prevent CSRF?

Use CSRF Tokens
Every sensitive request (like form submit) should include a unique **token**.
Server verifies this token — if it's missing or incorrect, it rejects the request.

SameSite Cookies
Set cookies with the `SameSite` attribute (`Strict` or `Lax`).
This prevents browsers from sending cookies along with cross-site requests.

Check Referer or Origin Headers
Server can check if the incoming request comes from its own domain.

Double Submit Cookies (advanced)
Send CSRF token both in cookie and in request body and validate on the server.

User Confirmation
For sensitive actions, ask for password re-entry or OTP (one-time password).

Problem	Solution
Browser sends cookies automatically	**Use CSRF Token** — a random string must be sent along with the form, checked on the server.
Browser allows cross-site cookie sending	**Use SameSite cookie** (SameSite=Strict or Lax) so cookies are not sent with cross-site requests.
No check where request comes from	Server can **check Origin or Referer header** to ensure the request is from its own website.

Feature	CSRF	XSS
What happens?	Attacker tricks **user** to send malicious request.	Attacker injects malicious **script** into website.
Target?	**User's actions**	**Website content**
Goal?	Perform action as user	Steal data or hijack session

CORS (stands for **Cross-Origin Resource Sharing)**

It's a way for your browser to **allow** or **block** web pages from **requesting resources** (like data, APIs, etc.) from **another domain (origin).**
By default, **browsers** don't allow requests **between two different domains.** That's called the **Same-Origin Policy** (SOP).
Same Origin → same protocol (http/https) + same domain + same port.
If anything is different → it's a **Cross-Origin** request.

Page URL	Request URL	Allowed?
https://mywebsite.com	https://mywebsite.com/api/data	✅ (Same origin)
https://mywebsite.com	https://anotherwebsite.com/api/data	❌ (Cross origin)

Why was CORS Introduced?

Because in real life:
You may want your **frontend** (like React app) on `http://localhost:3000` to call an **API server** running at `http://localhost:5000`.
OR Your site on `https://shop.com` wants to request data from `https://api.shop.com`.

Without CORS, browser will block these.
With CORS, the server can say: "It's okay, I allow this other domain to access my data."

How Does CORS Work?

When browser detects a cross-origin request, it **sends an extra request first** → called a **preflight request** (HTTP `OPTIONS` method).
This checks **"Is it okay if I access this resource?"**
Server responds with **special CORS headers** like:

```
Access-Control-Allow-Origin: https://yourfrontend.com
Access-Control-Allow-Methods: GET, POST
Access-Control-Allow-Headers: Content-Type
```

If server **allows**, browser proceeds. If server **denies**, browser blocks the response.

Another example: Suppose your frontend on `http://abc.com` wants data from http://xyz.com

```
fetch('http://xyz.com/api/data')
  .then(res => res.json())
  .then(data => console.log(data));
```

If xyz.com server does not send Access-Control-Allow-Origin header, browser **blocks** this fetch call.

Why Browser Blocks It?
To **protect** users from malicious sites trying to steal data (hijacking sessions, etc.)
CORS is a security feature enforced by browsers, **not by servers**.

In short,
CORS is a browser security feature that restricts web pages from making requests to a different domain, unless that domain explicitly allows it through specific HTTP headers.

Common CORS Errors
No 'Access-Control-Allow-Origin' header present.
The CORS policy does not allow access from this origin.

CSP (stands for **Content Security Policy**)

It's a **security mechanism** that helps prevent attacks like: XSS (Cross Site Scripting), Data injection attacks.

CSP is like a rulebook you give to the browser, saying: "Hey B**rowser,** only load scripts, images, styles, fonts, etc. from these trusted places."
If anything tries to load **outside** your trusted sources- Browser blocks it immediately.

Why CSP is Important?
Stops hackers from injecting bad scripts into your page. Even if someone somehow injects malicious code, **CSP can block its execution**. It **reduces the risk of XSS attacks massively**.

How CSP Works?
Server sends a special **HTTP Response Header**:

```
Content-Security-Policy: rules
```

These rules tell the browser: From where it can load **scripts, images, CSS, fonts, etc.**

Example: Server sends:

```
Content-Security-Policy: script-src 'self' https://apis.google.com
```

'self' → allow scripts from **the same domain**.
https://apis.google.com → allow scripts from Google APIs.
Block all other scripts from random sites!

How to Implement CSP?
You can set CSP:
From **HTTP Headers** (best way for real sites) OR inside your HTML using `<meta>` tag:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self';">
```

But using HTTP headers is safer because HTML <meta> can itself be modified by attackers.

Without CSP	With CSP
Anyone can inject scripts easily.	Only trusted sources are allowed.
Big XSS attack risk.	XSS risk becomes very low.
No control over 3rd party stuff.	Full control over what loads.