

A decorative background consisting of a large number of red dots of varying sizes. These dots are arranged in a circular pattern, with the density of the dots increasing towards the right side of the image, creating a sense of depth and movement.

WEB API

ONE LOVE. ONE FUTURE.

Motivation

- **Explain** what an API is and how it works in modern web applications.
- **Describe** JSON structure and use it to exchange data.
- **Understand** REST principles
- **Send and handle API requests** using Fetch and Axios.
- **Use Promises and async/await** to manage asynchronous operations.
- **Build simple client-server interactions** (consuming public APIs).
- **Debug and troubleshoot** common API call errors (CORS, network, status codes).

Content

1. Web API Fundamentals
2. Fetch API
3. Axios Library
4. Asynchronous JavaScript



1. Web API Fundamentals

1.1. API

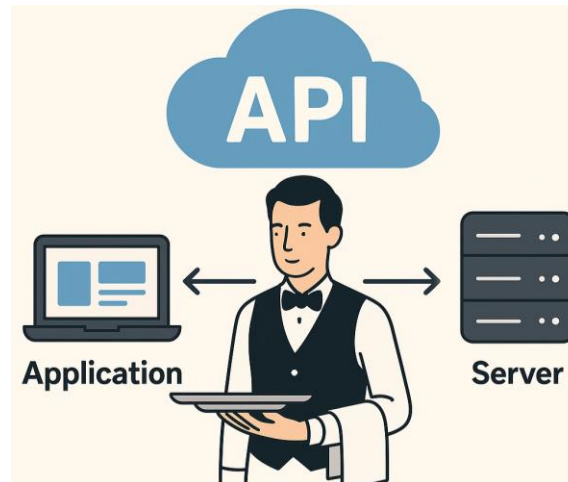
1.2. JSON

1.3. REST



1.1. API

- An API (application programming interface) is a set of rules or protocols that enables software applications to communicate with each other.
- An API is like a **restaurant waiter** — you tell the waiter what you want, the waiter communicates with the kitchen, and brings the result back to you.



Why Do We Need APIs?

- Modularity
 - breaks large, complex systems into smaller, independent services
 - enables easier maintenance, faster development, and better scalability.
- Security (Access Control):
 - exposes only the necessary endpoints and actions
 - protects sensitive data from unauthorized access
- Interoperability (Integration):
 - provides a communication standard for different technologies and systems
 - allows systems to integrate easily with each other.

Main Types of APIs

- **1. Web APIs (most common)**
 - Used by browsers, mobile apps, servers
 - Use HTTP/HTTPS
- **2. Library APIs**
 - Function calls inside programming languages
 - Example: DOM API, Node.js API
- **3. Operating System APIs**
 - Windows API, macOS API
- **4. Third-party Service APIs**
 - Google API, OpenAI API, Stripe API, Firebase API
- **5. Internal/Private APIs**
 - Used inside organizations for internal systems

Real-World Examples of APIs

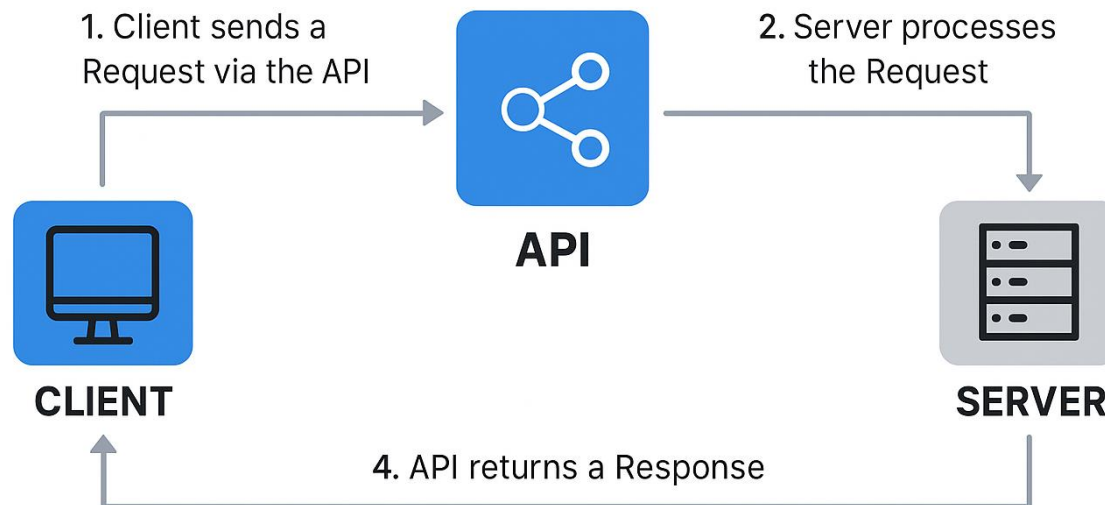
- Daily Apps
 - Google Maps API → show maps inside mobile apps
 - YouTube API → embedd video, search results
 - Gmail API → send email
 - Facebook API → login with Facebook
 - Message API → send message
 - Weather API → display current weather
- Finance
 - VNPay API, Momo API, ZaloPay API → payment
 - Striple API, PayPal API → payment
- E-commerce
 - Tiki Open API → product, order
 - Amazon API → product, order

4 Core Components of a Web API

Component	Meaning	Purpose
URL	Resource address	Identify target
Method	Action type	Define operation
Headers	Metadata	Context, auth, format
Body	Data payload	Send content

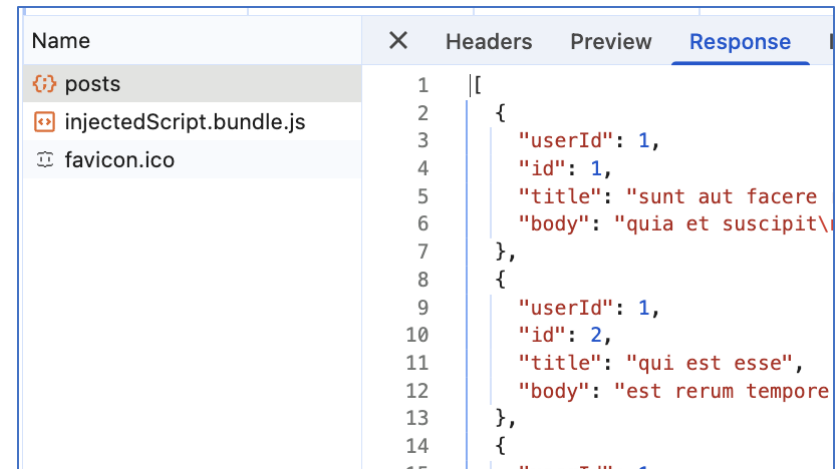
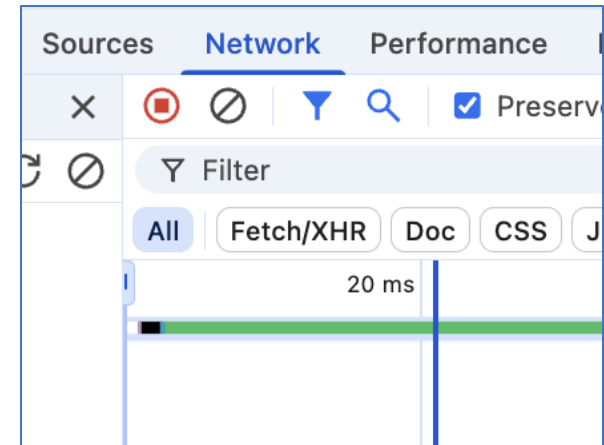
How Web APIs Work (Simple Flow)

- APIs act as a communication bridge between a **client** (your app, browser, or system) and a **server** (the backend that stores data or performs actions).
- Flow
 - 1. Client sends a Request via the API
 - 2. API receives the Request
 - 3. Server processes the Request
 - 4. API returns a Response



Inspecting Web APIs with DevTools

- Open DevTools
 - Choose **Inspect**
 - Choose Tab **Network**
- Filter Request
 - XHR: AJAX/Fetch/Axios
 - Doc/JS/CSS/Img
- Inspect a Request
 - Response body
 - Status code
 - Response header
- <https://jsonplaceholder.typicode.com/posts>



1.2. JSON

- **JSON (JavaScript Object Notation)** is a lightweight, text-based data format used to store and exchange data between systems.
- Features
 - Human-readable
 - Easy for machines to parse
 - Language-independent
- Structure
 - Objects (key-value pairs)
 - Arrays (ordered lists)
- Note
 - No comments allowed
 - UTF-8 by default

```
{  
  "name": "John",  
  "age": 25,  
  "isStudent": false  
}
```

```
{  
  "user": {  
    "id": 101,  
    "name": "Alice"  
  },  
  "roles": ["admin", "editor"]  
}
```

JSON Structure: Object & Array

- Two main structures
 - Objects → key–value pairs
 - Arrays → an ordered list of values
- JSON Object: collection of **key–value pairs** enclosed in { }
- Format: "key": value
- Keys are always strings
- Values can be any JSON data type
- Key–value pairs are separated by commas
- Unordered

```
{  
  "name": "John",  
  "age": 25,  
  "isStudent": false  
}
```

JSON Structure: Object & Array

- JSON Array: an ordered list of values enclosed in []
 - Ordered (index start at 0)
 - Used for lists: items, users, posts, etc

```
[  
  "apple",  
  "banana",  
  "orange"  
]
```

```
{  
  "user": {  
    "id": 101,  
    "name": "Alice"  
  },  
  "hobbies": ["reading", "music", "travel"],  
  "scores": [  
    { "subject": "Math", "score": 95 },  
    { "subject": "English", "score": 88 }  
  ]  
}
```

JSON ↔ JavaScript: parse() and stringify()

- Server sends JSON to Client => converts JSON to JS object
- Client sends data to Server => converts JS object to JSON
- JSON.parse(): converts JSON string → JS object
 - `const json = '{"name": "Alice"}';`
 - `const obj = JSON.parse(json);`
 - `console.log(obj.name); // "Alice"`
- JSON.stringify(): converts JS object → JSON String
 - `const user = {name: "Alice"};`
 - `const jsonString = JSON.stringify(user);`
 - `console.log(jsonString); // '{"name": "Alice"}'`

JSON Nested

- Nested JSON
 - Objects contain objects
 - Objects contain arrays
 - Arrays contain objects
- Example
 - `data.user.profile.city;`
`// "Hanoi"`
 - `data.orders[0].items[1]`
`.price; // 20`

```
{
  "user": {
    "id": 101,
    "name": "Alice",
    "profile": {
      "age": 25,
      "city": "Hanoi"
    }
  },
  "orders": [
    {
      "orderId": "A001",
      "items": [
        { "name": "Laptop", "price": 1500 },
        { "name": "Mouse", "price": 20 }
      ]
    }
  ]
}
```

JSON Problems

- JSON
 - supports: String, Number, Boolean, Null, Object, Array
 - not support: Date, Function, undefined, binary data, regular experssion, NaN
- Date → JSON → Date
 - `const obj = {created: new Date() };`
 - `JSON.stringify(obj);`
`//{ "created": "2025-10-10T05:00:00.000Z" }`
 - `const data = JSON.parse(json);`
 - `data.created = new Date(data.created);`
- `JSON.stringify({ x: undefined }); // "{}"`
- `JSON.stringify({ fn: () => {} }); // "{}"`

1.3. REST

- REST = Representational State Transfer: is a set of principles used to design Web APIs.
- Restful API: is an API that follows REST principles

Principle	Meaning
Client-Server	UI separate from backend
Stateless	No session on server
Cacheable	Responses can be cached
Uniform Interface	Consistent API design
Layered System	Multiple transparent layers
Code on Demand	Server sends code (optional)

REST: Resource Naming Rules

- Resource: any entity
- Use nouns, not a verb, e.g., users, orders, posts
 - Good: `/users`, `/orders/123`
 - Bad: `/user`, `/getUsers`, `/createOrder`
- Use hierarchical structure:
 - `/users/10/orders`
 - `/orders/5/items`
- Use Path for Resources, Query for Filters
 - `/products` → all products
 - `/products?category=phone` → filtered list
- Use Lowercase & Hyphens: `/user-profiles`
- Avoid Trailing Slashes: `/users` insted of `/users/`

CRUD Mapping

CRUD	HTTP Method	Example	Description
Create	POST	/users	Create new user
Read	GET	/users/5	Retrieve user
Update	PUT	/users/5	Replace entire user
	PATCH	/users/5	Update partially
Delete	DELETE	/users/5	Delete user

Parameters: Identifier (Path) vs Filter (Query)

- Path Parameters = Identifier
 - represent a **specific resource**
 - identify resource by ID/code
 - order is important
 - GET `/users/10`
 - GET `/orders/A001/items/5`
- Query Parameters = Filters/Sorting/Pagination
 - use for **query** (e.g., filtering, sorting, pagination)
 - order doesn't matter
 - GET `/products?category=phone`
 - GET `/posts?sort=desc&page=2&limit=10`
 - **✗** `/products/category/phone` (not RESTful)

HTTP Status Code

Category	Meaning	Examples
1xx	Info	100
2xx	Success	200, 201, 204
3xx	Redirect	301, 302, 304
4xx	Client error	400, 401, 403, 404
5xx	Server error	500, 502, 503, 504

2xx Success Codes (200, 201, 204)

- 200 OK
 - Request succeeded and returns a response body.
 - Return JSON data
 - e.g., GET /users
- 201 Created
 - A new resource was **successfully created**
 - May return newly created object
 - e.g., POST /users (create user)
- 204 No Content
 - Request succeeded **but no response body**
 - e.g., DELETE /users/10

4xx Client Errors (400, 401, 403, 404)

- 400 — Bad Request
 - server **cannot process** the request because of invalid request
 - e.g., POST /users with missing "name" field
- 401 - Unauthorized
 - authentication is **required** but missing or invalid.
 - e.g, GET /users without JWT token
- 403 — Forbidden
 - Client is authenticated but **not allowed** to access
 - e.g., insufficient permissions
- 404 — Not Found
 - requested resource **does not exist**
 - e.g., wrong path, resource not found

5xx Server Errors (500, 503)

- 500 — Internal Server Error
 - **server itself** encountered an error.
 - e.g., backend bugs, database errors, exception
- 503 — Service Unavailable
 - server is **reachable**, but **temporarily unable** to handle the request.
 - e.g., server overload
- 502 — Bad Gateway (Gateway: Nginx, load balancer)
 - gateway received a bad **response** from server.
 - e.g, server returned invalid response
- 504 — Gateway Timeout
 - gateway **did not receive a response in time** from server
 - e.g., slow backend processing

Gateway (Nginx, Apache, AWS, Azure...)

- Gateway is the smart middle layer that controls how the Client communicates with the Server
 - protect backend server
 - filter request, block attacks
 - direct traffics
 - track errors

```
upstream backend {  
    server 127.0.0.1:3000;  
    server 127.0.0.1:3001;  
}  
  
server {  
    listen 80;  
    location / { proxy_pass http://backend; }  
}
```



CORS: When the Browser “Asks Permission”

- CORS is a security mechanism in browsers that controls whether a web page can call APIs from a different origin.
 - Origin = protocol + domain + port
 - e.g., <http://app.com> and <http://api.com> are different origins
- Browser sends Preflight request for non-simple requests (e.g., DELETE, PUT, PATCH)
 - OPTIONS /api HTTP/1.1
 - Origin: <https://api.com>
- Server must reply as follows to allow browser to call API
 - HTTP/1.1 204 No Content
 - Access-Control-Allow-Origin: <https://api.com>
 - Access-Control-Allow-Methods: GET, POST, PUT
 - Access-Control-Allow-Headers: Content-Type

Simple vs Non-Simple Requests

- Simple requests: three conditions
 1. Allowed Methods: GET, POST, HEAD
 2. Allowed Headers (Only simple headers)
 - Accept
 - Accept-Language
 - Content-Language
 - Content-Type (but only 3 specific types below)
 3. Content-Type must be one of:
 - text/plain
 - multipart/form-data
 - application/x-www-form-urlencoded

Browser sends the **request immediately** (no OPTIONS).



HUST

2. Fetch API

Introduction

- A modern interface for fetching resources
- Native in all modern browsers
- Advantages
 - simplicity and modernity (promise-based)
 - clear request/response objects for data handling
 - excellent JSON handling
- Disadvantages
 - Error handling complexity (manual status check)
 - Lack of timeout mechanism
 - Manual cookie and credentials handling

Fetch API: Syntax with Async Await

- Writes code like traditional synchronous (blocking) code.
 - `async`: declares an asynchronous function
 - `await`: pauses the execution of the async function until the Promise settles
- Two-step process for data access
 - `fetch()`: return a response object
 - `response.json()` or `response.text` to access the data

```
async function fetchData() {  
  try {  
    const response = await fetch('https://api.example.com/data');  
    if (!response.ok) {  
      throw new Error(`HTTP error! status: ${response.status}`);  
    }  
    const data = await response.json();  
    console.log(data);  
  } catch (error) {  
    console.error('Fetch error:', error);  
  }  
}
```

Handling Response

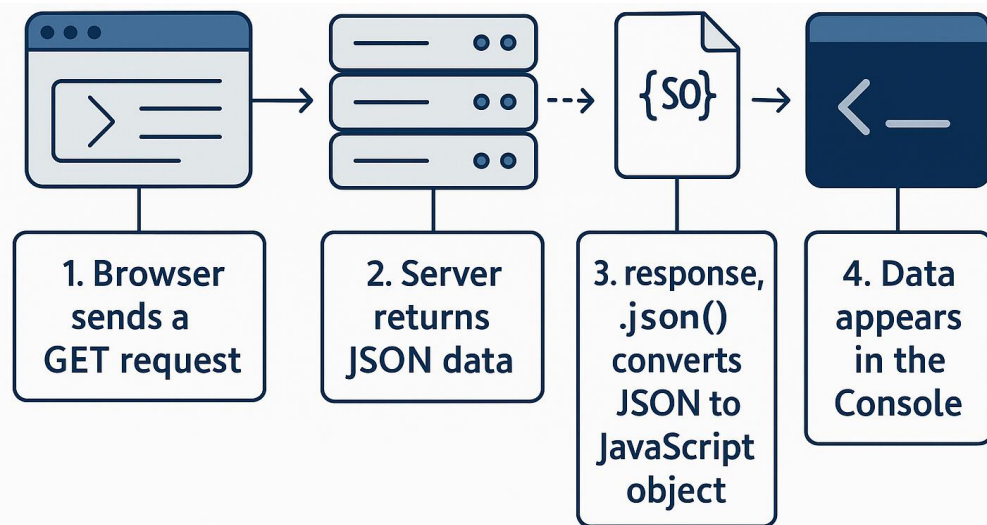
- `fetch()` considers a successful HTTP response to be any response received from the server, including 404 (Not Found), 500 (Internal Server Error).
- Developers must manually check the HTTP status code
 - `response.status`: number, e.g., 200, 404
 - `response.statusText`: string, e.g., OK, Not Found
 - `response.ok`: boolean
 - True if the status code is in the range 200-299;
 - False otherwise -> Error handling

response.json

- A method of the Fetch **Response** object
 - Reads the response body
 - Parses it as **JSON**
 - Returns a **Promise** that resolves to a JavaScript object
- Notes
 - Works only if the response body is valid JSON
 - Non-JSON data, use: `response.text()`, `response.blob()`, `response.formData()`

```
const response = await fetch(url);  
const data = await response.json();  
console.log(data);
```

Fetch - GET



```
fetch("https://jsonplaceholder.typicode.com/users")  
  .then(response => response.json())  
  .then(data => console.log(data));
```

```
async function loadUsers() {  
  const response = await fetch(  
    "https://jsonplaceholder.typicode.com/users"  
  );  
  const data = await response.json();  
  console.log(data);  
}  
loadUsers();
```

Fetch - POST

```
fetch("https://dummyjson.com/products/add", {  
  method: "POST",  
  headers: { "Content-Type": "application/json" },  
  body: JSON.stringify({  
    title: "New Product",  
    price: 150  
  })  
})  
  
.then(res => res.json())
```

```
async function addProduct() {  
  const response = await fetch(  
    "https://dummyjson.com/products/add",  
    {  
      method: "POST",  
      headers: { "Content-Type": "application/json" },  
      body: JSON.stringify({ title: "New Product" })  
    }  
  );  
  const data = await response.json();  
  console.log(data);  
}
```

Authorization Token (Bearer)

- Client sends a token in the **Authorization** header:
- Authorization: Bearer <token> → browser sends a Preflight

```
fetch("https://api.example.com/profile", {  
  headers: {  
    "Authorization": "Bearer YOUR_TOKEN_HERE"  
  }  
})  
  
.then(res => res.json())  
.then(data => console.log(data));
```

```
async function loadProfile() {  
  const response = await fetch("https://api.example.com/profile", {  
    headers: {  
      "Authorization": "Bearer YOUR_TOKEN_HERE"  
    }  
  });  
  
  const data = await response.json();  
  console.log(data);  
}
```

Timeout with Promise.race()

- Fetch API has no built-in timeout.
 - If the server hangs, the request waits forever
 - Promise.race(): runs two promises in parallel i.e., Fetch and Timeout

```
function timeout(ms) {  
  return new Promise( (_, reject) =>  
    setTimeout(() => reject(new Error("Timeout")), ms)  
  );  
}  
  
Promise.race([  
  fetch("https://api.example.com/data"),  
  timeout(5000)  
)  
  .then(res => res.json())  
  .then(console.log)  
  .catch(err => console.error(err.message));
```

AbortController: Active Request Cancellation

- A browser API that allows you to cancel (abort) an ongoing Fetch request.
- Useful for stopping slow, unnecessary, or duplicated requests.
- Does AbortController cancel the Promise?

```
function fetchWithTimeout(url, timeout = 3000) {  
  const controller = new AbortController();  
  const signal = controller.signal;  
  
  // Auto-cancel after timeout  
  setTimeout(() => controller.abort(), timeout);  
  
  return fetch(url, { signal });  
}  
  
// Use it  
fetchWithTimeout("/api/data", 3000)  
  .then(res => res.json())  
  .then(data => console.log(data))  
  .catch(err => {  
    if (err.name === "AbortError") {  
      console.log("Request timed out.");  
    } else {  
      console.error("Error:", err);  
    }  
  })  
};
```



HUST

3. Axios Library

Introduction

- Axios is one of the most widely HTTP client libraries.
- Automatic Error Handling
 - automatically rejects promises for all HTTP errors
 - `axios.get("/api/users").catch(err => console.error("Error:", err));`
- Built-in Timeout, Cancelation
 - Easy timeout configuration (no need AbortController).
 - `axios.get(url, { timeout: 5000 });`
- Automatic JSON Handling & Safer Defaults
 - Automatically parses JSON into `res.data`
 - `const res=await axios.post(url,{title: "A"});`
 - `console.log(res.data);`

Axios vs Fetch

- Use Axios for: large apps, interceptors, timeouts, clean error handling.
- Use Fetch for: simple requests, modern browsers, lightweight usage without libraries.
- Axios
 - Better error handling (treats non-2xx as errors)
 - Has built-in timeout
 - Cleaner syntax for complex calls
- Fetch
 - Native browser API (no installation)
 - No built-in timeout
 - Requires manual JSON parsing
 - Treats non-2xx responses as success (must check res.ok)

Install Axios

- Install

- Project: `npm install axios`
- Browser via CDN:

```
<script  
src="https://cdn.jsdelivr.net/npm/axios/dist/  
axios.min.js">  
  
</script>
```

- Quick check

- `import axios from "axios";`
- `axios.get("/api/test")
.then(res => console.log("Axios Ready:",
res.data));`

Axios GET: Clean Syntax & Destructuring

Axios GET Demo

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    }
  },
  ...
]
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Axios GET Demo</title>
  <script
    src="https://cdn.jsdelivr.net/npm/axios/dist/
    axios.min.js">
  </script>
</head>
<body>
  <h2>Axios GET Demo</h2>
  <pre id="output"></pre>
  <script>
    axios.get("https://jsonplaceholder.typicode.c
om/users")
      .then(response => {
        document.getElementById("output").textContent
        = JSON.stringify(response.data, null, 2);
      })
      .catch(err => console.error(err));
  </script>
</body>
</html>
```

Axios POST/PUT/DELETE

POST (create data)

```
axios.post("https://dummyjson.com/products/add", {  
  title: "New Product",  
  price: 150  
})  
  .then(res => console.log(res.data))  
  .catch(err => console.error(err));
```

PUT (update entire data)

```
axios.put("https://dummyjson.com/products/1", {  
  title: "Updated Title",  
  price: 200  
})  
  .then(res => console.log(res.data))  
  .catch(err => console.error(err));
```

DELETE (remove data)

```
axios.delete("https://dummyjson.com/products/1")  
  .then(res => console.log("Deleted:", res.data))  
  .catch(err => console.error(err));
```

Axios Instance: Global Configuration

- A pre-configured Axios object contains shared settings (base URL, headers, timeout, interceptors).
- Allow reuse it across your entire application

```
api.get("/users").then(res => console.log(res.data));
```

```
const api = axios.create({
  baseURL: "https://api.example.com",
  timeout: 5000,
  headers: {
    "Content-Type": "application/json"
  }
});
```

```
api.post("/login", { username, password });
```

Axios - Demo

Axios Instance Demo

Run GET

Run POST

Run PUT

```
// 1. Create Axios Instance
const api = axios.create({
  baseURL: "https://dummyjson.com",
  timeout: 5000,
  headers: { "Content-Type": "application/json" }
});

// 2. Demo GET
async function demoGet() {
  const { data } = await api.get("/products");
  document.getElementById("output").textContent =
    "GET /products\n\n" + JSON.stringify(data, null, 2);
}
```

GET /products

```
{
  "products": [
    {
      "id": 1,
      "title": "Essence Mascara Lash",
      "description": "The Essence Mas",
      "category": "beauty",
      "price": 9.99,
      "discountPercentage": 10.48,
      "rating": 2.56,
      "stock": 99,
      "tags": [
        "beauty",
        "mascara"
      ],
      "images": [
        "https://cdn.dummyjson.com",
        "https://cdn.dummyjson.com",
        "https://cdn.dummyjson.com",
        "https://cdn.dummyjson.com",
        "https://cdn.dummyjson.com"
      ]
    }
  ]
}
```

PUT /products/1

```
{
  "id": 1,
  "title": "Update Title for Product 1",
  "price": 9.99,
  "discountPercentage": 10.48,
  "stock": 99,
  "rating": 2.56,
  "images": [
    "https://cdn.dummyjson.com",
    "https://cdn.dummyjson.com",
    "https://cdn.dummyjson.com",
    "https://cdn.dummyjson.com",
    "https://cdn.dummyjson.com"
  ]
}
```

POST /products/add

```
{
  "id": 195,
  "title": "New Product from Axios",
  "price": 100
}
```

Request Interceptor

- allows you to run code BEFORE every request is sent.
 - add headers
 - inject Authorization tokens
 - log request information

```
const api = axios.create({
  baseURL: "https://dummyjson.com",
  timeout: 5000
});

api.interceptors.request.use(config => {
  console.log("Request →", config.method, config.url);

  config.headers.Authorization = "Bearer 123456";
  return config;
});
```

Response Interceptor

- allows you to handle response and error BEFORE conducting application logic.
 - handle API errors
 - auto-refresh tokens
 - standardize error messages
 - ...

```
api.interceptors.response.use(  
  response => {  
    // Successful response (2xx)  
    return response;  
  },  
  error => {  
    console.error("API Error:", error);  
    return Promise.reject(error);  
  }  
);
```

👉 Catch all errors from every request in one place.

👉 No need to write try/catch everywhere.

Axios Error Handling

- When Axios receives an HTTP error (4xx / 5xx), it returns an Error Object.
 - `error.response.status`: HTTP status code, e.g., 400, 401, 403, 404, 500
 - `error.response.data`: error body from the server
 - `error.response.headers`: headers from the server
 - `error.response.config`: request configuration

```
api.get("/auth/me")
  .catch(error => {
    if (error.response) {
      console.log("Status:", error.response.status);
      console.log("Message:", error.response.data);
    } else {
      console.log("Network error or no response");
    }
  });
```

Canceling Requests

- use **AbortController**

```
const controller = new AbortController();

axios.get("/api/data", {
  signal: controller.signal
})
.catch(err => {
  if (err.name === "CanceledError") {
    console.log("Request cancelled");
  }
});

// Cancel it
controller.abort();
```

Retry Pattern: Automatic Retries on 503/429

- When to Retry
 - 503 Service Unavailable → server is overloaded or restarting
 - 429 Too Many Requests → hit rate limit, need to slow down

```
async function fetchWithRetry(url, retries = 3) {  
  try {  
    return await axios.get(url);  
  } catch (err) {  
    if (retries > 0 && [429, 503].includes(err.response?.status)) {  
      return fetchWithRetry(url, retries - 1);  
    }  
    throw err;  
  }  
}
```



HUST

4. Asynchronous JavaScript

async/await

- async makes a function return a Promise
- await pauses execution until the Promise resolves/rejects

```
async function getUser() {  
  try {  
    const res = await fetch("/api/user");  
    if (!res.ok) throw new Error("Bad response");  
    return await res.json();  
  } catch (err) {  
    console.error("Error:", err);  
  }  
}
```

Promise.all: Boost Performance with Parallel Requests

- Promise.all() allows you to run multiple asynchronous operations in parallel

- Total time = *time of the slowest request*

```
const [users, posts] = await
Promise.all([
    api.get("/users"),
    api.get("/posts")
]);
```

- If any Promise fails:
 - Promise.all() rejects immediately
 - All results are discarded
 - Error goes to catch

Promise.allSettled: Safe Parallel Execution

- Runs multiple promises in parallel and waits for all of them to finish, regardless of success or failure
- Ideal when you **need all results**, even failed ones

```
const results = await Promise.allSettled([
  fetch("/api/user"),
  fetch("/api/products"),
  fetch("/api/orders")
]);

results.forEach(r => console.log(r.status, r.value || r.reason));
```

```
[
  {
    status: "fulfilled",
    value: Response { ... }
  },
  {
    status: "fulfilled",
    value: Response { ... }
  },
  {
    status: "rejected",
    reason: Error("Network error")
  }
]
```

Exercise

Build a CRUD web application using Fetch or Axios to interact with <https://jsonplaceholder.typicode.com/users>

Required Features:

- Read: show a table of users with name, email, phone
- Create: form to add new user
- Update/Edit: edit or popup form
- Delete: remove user from table
- Search: filter users by name
- Pagination
- Note:
 - use async/await, not .then()
 - update UI manually after POST/PUT/DELETE
 - handle errors

