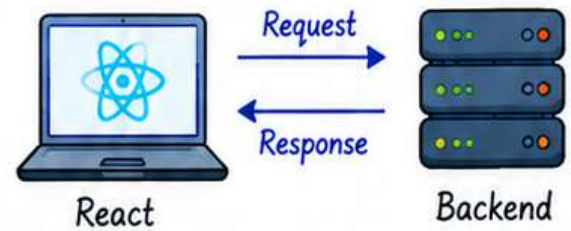


Chapter 10: React API Integration

10.1 Frontend-Backend Communication

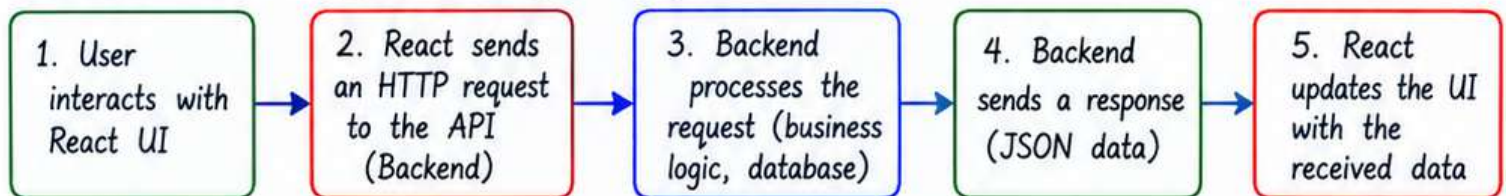
10.1.1 What is API?

- * API (Application Programming Interface) is a **set of rules** that allows different software applications to **communicate** with each other.
- * It acts as a **bridge** between the frontend (React) and backend (server / database).
- * Using APIs, the frontend can **send data** to the backend and **receive data** from the backend.
- * APIs make it possible to build **dynamic** and **interactive** web applications.



10.1.2 How Frontend-Backend Communication Works?

The communication happens in the following flow:



Key Points:

- ✓ Frontend and backend communicate using **HTTP/HTTPS**.
- ✓ Data is sent in **JSON** format (most common).
- ✓ APIs can be **RESTful APIs** or **GraphQL APIs**.
- ✓ This helps in building **scalable**, **secure** and **maintainable** applications.



10.1.3 Types of APIs

There are different types of APIs used in web development:

1. REST APIs

- * Most popular type of API.
- * Uses standard HTTP methods (GET, POST, PUT, DELETE).
- * Simple, stateless and easy to use.
- * Used widely in modern web applications.

2. GraphQL APIs

- * Allows client to request only the data they need.
- * More flexible and efficient than REST.
- * Reduces over-fetching and under-fetching of data.
- * Growing in popularity with modern applications.

Key Takeaway:

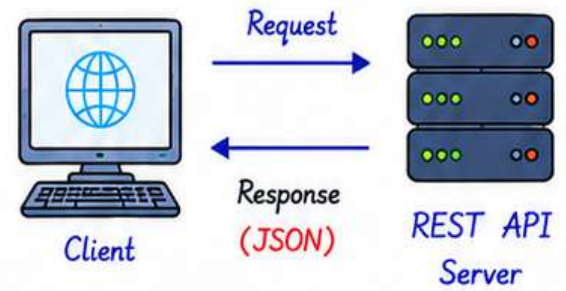


APIs are the **backbone** of modern web applications. They help React communicate with the backend, receive data, and render **dynamic** content in a smooth and **efficient** way.

10.2 REST APIs

10.2.1 What is REST API?

- * REST (Representational State Transfer) is an **architectural style** for designing networked applications.
- * REST APIs use **HTTP methods** to perform **CRUD** operations.
- * It is **stateless**, meaning each request from client to server must contain all the information needed to understand the request.
- * Data is exchanged in standard formats like **JSON** or **XML**.
- * It is lightweight, scalable and easy to integrate.



Key Points:

- ✓ REST uses standard HTTP methods.
- ✓ It is stateless.
- ✓ Data is exchanged in JSON format.
- ✓ It follows a client-server architecture.

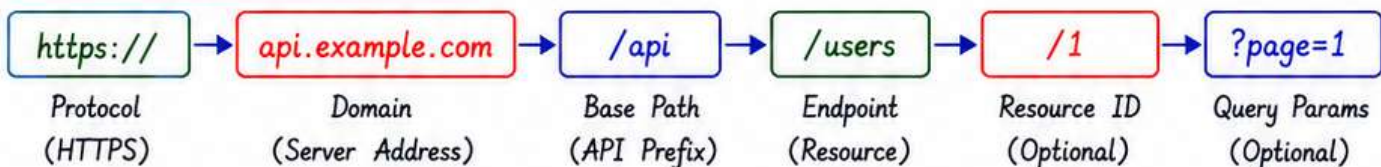


10.2.2 HTTP Methods in REST APIs

Method	Purpose	Description	Example
GET	Read	Fetch data from the server	GET /api/users
POST	Create	Send data to server to create a new resource	POST /api/users
PUT	Update	Update an existing resource completely	PUT /api/users/1
PATCH	Partial Update	Update an existing resource partially	PATCH /api/users/1
DELETE	Delete	Delete a resource from the server	DELETE /api/users/1



10.2.3 REST API Structure

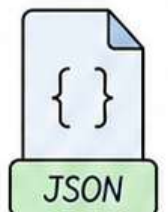


10.2.4 Example: GET Request

- * When we send a GET request to fetch users.
- * Request URL: **https://api.example.com/api/users**
- * Response (JSON):

```
[ { "id": 1, "name": "Alice", "email": "alice@gmail.com" },  
  { "id": 2, "name": "Bob", "email": "bob@gmail.com" } ]
```

} JSON data
returned by
the server



Key Takeaway:

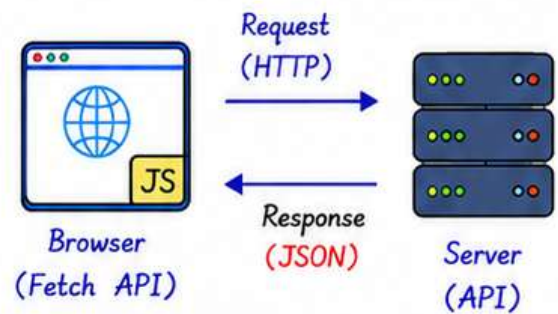


REST APIs allow React applications to perform **CRUD** operations by sending HTTP requests and receiving data in **JSON** format.

10.3 Fetch API

10.3.1 What is Fetch API?

- * Fetch API is a modern JavaScript API used to make **HTTP** requests from the browser.
- * It provides an easy and powerful way to **send** and **receive** data from a server.
- * Fetch returns **Promises**, making it easy to handle asynchronous operations.
- * It is supported by all modern browsers.



Key Points:

- ✓ Fetch API is used to make HTTP requests.
- ✓ It returns Promises.
- ✓ It can handle JSON, text, files and more.
- ✓ It is easy to use and widely supported.



10.3.2 Basic Syntax

```
fetch(url, options)
  .then(response => response.json())
  .then(data => {
    // use data
  })
  .catch(error => {
    // handle error
  });
```

- **url** → The API endpoint.
- **options** → (Optional) method, headers, body, etc.
- **response** → The response from the server.
- **response.json()** → Converts response to JSON.
- **data** → The data received from the server.
- **catch()** → Handles errors.

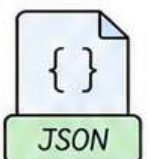
10.3.3 Example: GET Request

- * Let's fetch a list of users from an API.
- * API: <https://jsonplaceholder.typicode.com/users>

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => response.json())
  .then(data => {
    console.log(data); // display users in console
  })
  .catch(error => {
    console.error('Error:', error); // handle error
  });
```

Output (JSON Data)

```
[
  { "id": 1, "name": "Leanne Graham",
    "email": "leanne@gmail.com", ... },
  { "id": 2, "name": "Ervin Howell",
    "email": "ervin@gmail.com", ... }
  ...
]
```



10.3.4 Request Methods with Fetch

Method	Use	Example in Fetch
GET	Get data from the server	<code>fetch('/api/data')</code>
POST	Send data to create a resource	<code>fetch('/api/data', { method: 'POST', body: data })</code>
PUT	Update an existing resource	<code>fetch('/api/data/1', { method: 'PUT', body: data })</code>
DELETE	Delete a resource	<code>fetch('/api/data/1', { method: 'DELETE' })</code>



Key Takeaway:



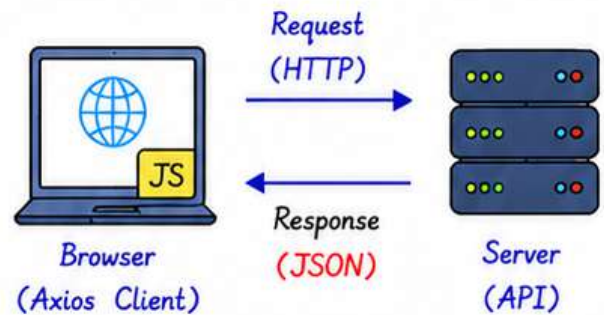
Fetch API makes it simple to communicate with APIs. It is **promise-based**, supports various HTTP methods and works smoothly with **JSON** data.



10.4 Axios

10.4.1 What is Axios?

- * Axios is a popular JavaScript library used to make **HTTP** requests from the browser or Node.js.
- * It is **promise-based**, easy to use and feature-rich.
- * Axios works in both the browser and Node.js environment.
- * It automatically transforms **JSON** data.
- * It has better **error handling** and request/response **interceptors**.



Key Points:

- ✓ Axios is a promise-based HTTP client.
- ✓ It is easy to send async requests.
- ✓ It automatically handles JSON data.
- ✓ It provides better error handling.
- ✓ It supports request & response interceptors.

axios()
makes HTTP
requests simple



10.4.2 Basic Syntax

```
axios(config)
  .then(response => {
    // handle success
  })
  .catch(error => {
    // handle error
  });
```

- **axios(config)** → Sends the request with the given configuration.
- **response** → The response object from the server.
- **response.data** → Contains the actual response data.
- **response.status** → Contains the HTTP status code.
- **catch(error)** → Handles any error that occurs.

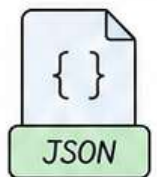
10.4.3 Example: GET Request

- * Let's fetch a list of users using Axios.
- * API: <https://jsonplaceholder.typicode.com/users>

```
axios.get('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    console.log(response.data); // display users
  })
  .catch(error => {
    console.error('Error:', error); // handle error
  });
```

Output (JSON Data)

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "email": "leanne@gmail.com", ...
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "email": "ervin@gmail.com", ...
  }
  ...
]
```



10.4.4 HTTP Methods with Axios

Method	Use	Example in Axios
GET	Get data from the server	<code>axios.get('/api/users')</code>
POST	Send data to create a resource	<code>axios.post('/api/users', data)</code>
PUT	Update an existing resource	<code>axios.put('/api/users/1', data)</code>
PATCH	Partially update an existing resource	<code>axios.patch('/api/users/1', data)</code>
DELETE	Delete a resource	<code>axios.delete('/api/users/1')</code>



Key Takeaway:



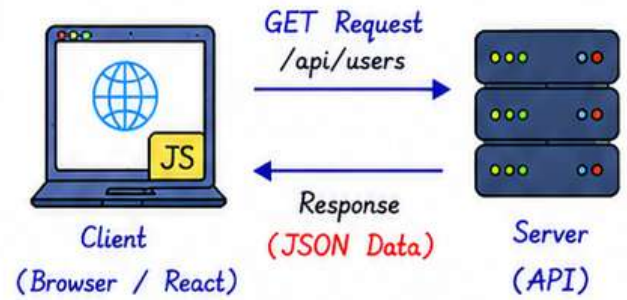
Axios makes it **easy** to communicate with APIs. It is simple, has powerful features and improves code **readability** and **maintainability**.



10.5 GET Requests

10.5.1 What is GET Request?

- * GET is an HTTP method used to **retrieve data** from the server.
- * It does **not** send data in the body.
- * Data is sent in the URL as **query parameters**.
- * It is **safe** and **idempotent**, meaning it does not change data on the server.
- * It is used to read data from resources.



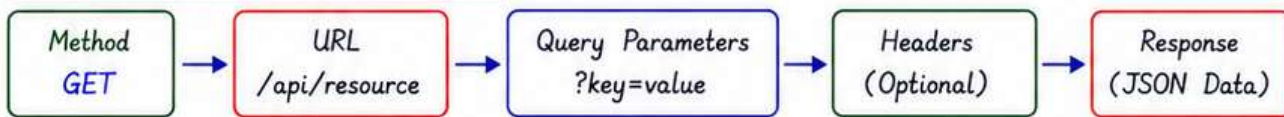
Key Points:

- ✓ GET is used to fetch data from the server.
- ✓ Data is sent in the URL, not in the request body.
- ✓ It is safe (does not modify data).
- ✓ It is idempotent (same request gives same result).
- ✓ Used for retrieving lists, details, or information.

GET
is used
to Read
Data



10.5.2 GET Request Structure



10.5.3 Example: GET Request

- * Let's fetch a list of users from an API.
- * API: <https://jsonplaceholder.typicode.com/users>

Using Fetch API

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => response.json())
  .then(data => {
    console.log(data); // display users
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Using Axios

```
axios.get('https://jsonplaceholder
.typicode.com/users')
  .then(response => {
    console.log(response.data);
    // display users
  })
  .catch(error => {
    console.error('Error:', error);
  });
```

Output (JSON Data)

```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "email": "leanne@gmail.com"
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "email": "ervin@gmail.com"
  },
  ...
]
```

JSON

10.5.4 GET Request with Query Parameters

Example: Get posts of user with id = 1

URL with Query Parameters
<https://jsonplaceholder.typicode.com/posts?userId=1>

Fetch API Example

```
fetch('https://jsonplaceholder.typicode.com/posts?userId=1')
  .then(res => res.json())
  .then(data => console.log(data));
```

Query
Parameters
are added
after ? in
the URL



10.5.5 Common Use Cases

Get all users
GET /api/users



Get user by ID
GET /api/users/1



Get posts of a user
GET /api/posts?userId=1



Get product details
GET /api/products/10



Key Takeaway:



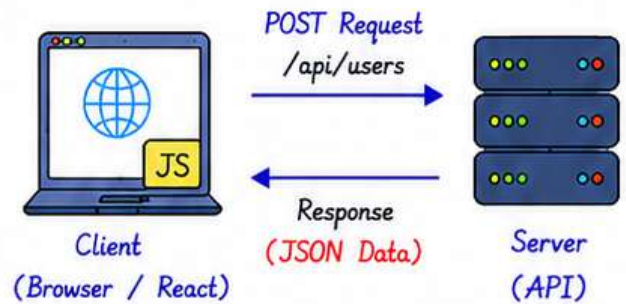
GET requests are used to **retrieve data** from the server. They are **simple, safe**, and send data in the URL as **query parameters**.



10.6 POST Requests

10.6.1 What is POST Request?

- * POST is an HTTP method used to **send data** to the server.
- * It sends data in the **request body**.
- * Data is usually used to **create** a new resource.
- * The server processes the data and returns a response.
- * It is **not idempotent**, meaning multiple requests may create multiple resources.
- * It is commonly used for forms, registrations, logins, etc.

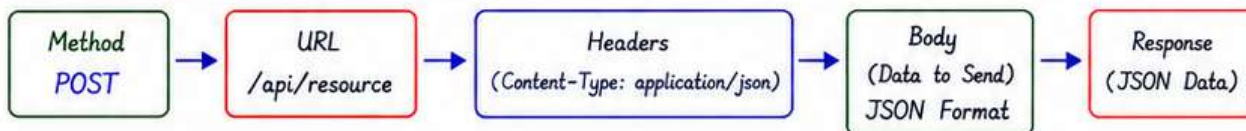


Key Points:

- ✓ POST is used to send data to the server.
- ✓ Data is sent in the request body.
- ✓ It is used to create new resources.
- ✓ It is not idempotent (multiple requests may create multiple resources).
- ✓ Used for submitting forms, registrations, logins, uploading data, etc.



10.6.2 POST Request Structure



10.6.3 Example: POST Request

- * Let's send user data to create a new user.
- * API: <https://jsonplaceholder.typicode.com/users>

Using Fetch API

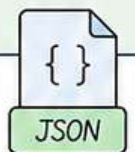
```
fetch('https://jsonplaceholder.typicode.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
.then(response => response.json())
.then(data => console.log(data));
```

Using Axios

```
axios.post('https://jsonplaceholder.typicode.com/users', {
  name: 'John Doe',
  email: 'john@example.com'
}, {
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => console.log(response.data));
```

Sample Response (JSON Data)

```
{
  "id": 101,
  "name": "John Doe",
  "email": "john@example.com",
  "createdAt": "2024-05-14T10:30:00Z"
}
```



10.6.4 POST Request with Form Data

HTML Form

```
<form id="userForm">
  <input type="text" name="name" placeholder="Name" />
  <input type="email" name="email" placeholder="Email" />
  <button type="submit">Submit</button>
</form>
```

JavaScript (Using Fetch)

```
document.getElementById('userForm').addEventListener('submit',
function(e) {
  e.preventDefault();
  const formData = new FormData(e.target);
  fetch('/api/users', {
    method: 'POST',
    body: formData
  })
  .then(res => res.json())
  .then(data => console.log(data));
});
```

FormData automatically handles form fields and sends them in the request body.



10.6.5 Common Use Cases

User Registration
POST /api/register



User Login
POST /api/login



Create New Post
POST /api/posts



Upload File / Data
POST /api/upload



Key Takeaway:



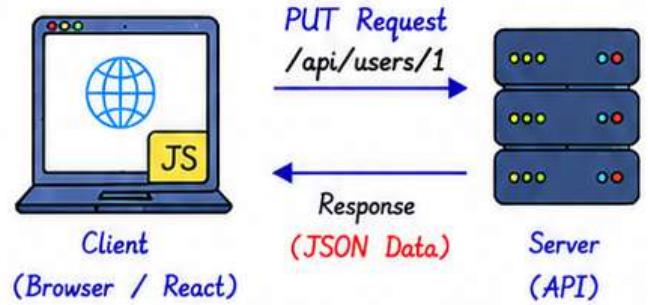
POST requests are used to **send data** to the server to **create** new resources. Data is sent in the **request body**, and the response usually contains the created resource.



10.7 PUT Requests

10.7.1 What is PUT Request?

- * PUT is an HTTP method used to **update an existing resource**.
- * It sends data in the **request body**.
- * It **replaces** the **entire resource** with the new data.
- * If the resource does not exist, the server may **create** it.
- * It is **idempotent**, meaning multiple identical requests have the same effect as a single request.
- * It is commonly used to update user profiles, records, etc.



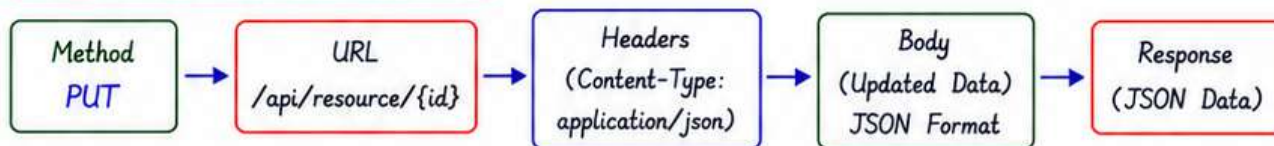
Key Points:

- ✓ PUT is used to update an existing resource.
- ✓ Data is sent in the request body.
- ✓ It replaces the entire resource.
- ✓ It is idempotent (same result for multiple identical requests).
- ✓ Used to update records like user profiles, settings, etc.

PUT
is used
to Update
Existing
Resource



10.7.2 PUT Request Structure



10.7.3 Example: PUT Request

- * Let's update user with id = 1.
- * API: <https://jsonplaceholder.typicode.com/users/1>

Using Axios

Using Fetch API

```
fetch('https://jsonplaceholder.typicode.com/users/1', {
  method: 'PUT',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    id: 1,
    name: 'John Doe Updated',
    email: 'john.updated@example.com',
    username: 'johndoe'
  })
})
.then(response => response.json())
.then(data => console.log(data));
```

```
axios.put('https://jsonplaceholder.typicode.com/users/1', {
  id: 1,
  name: 'John Doe Updated',
  email: 'john.updated@example.com',
  username: 'johndoe'
}, {
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => {
  console.log(response.data);
})
.catch(error => console.error(error));
```

Sample Response (JSON Data)

```
{
  "id": 1,
  "name": "John Doe Updated",
  "username": "johndoe",
  "email": "john.updated@example.com",
  "address": {
    "street": "Updated Street 123",
    "city": "New York"
  },
  "phone": "123-456-7890"
}
```



JSON

10.7.4 Common Use Cases

Update User Profile
PUT /api/users/1



Update Account Settings
PUT /api/settings/1



Update Product Details
PUT /api/products/10



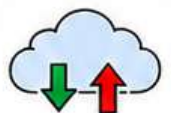
Update Order Status
PUT /api/orders/5



Key Takeaway:



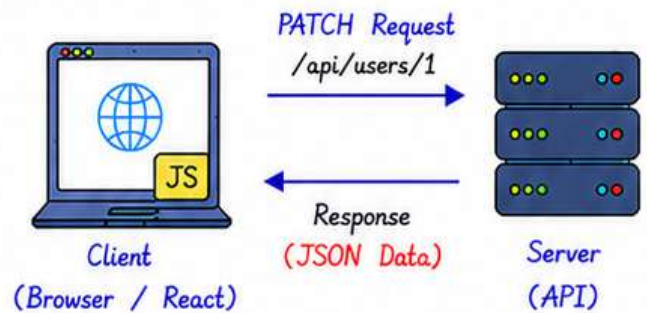
PUT requests are used to **update** existing resources by **replacing** them with new data. Data is sent in the **request body**, and the operation is **idempotent**.



10.8 PATCH Requests

10.8.1 What is PATCH Request?

- * PATCH is an HTTP method used to **partially update** an existing resource.
- * It sends data in the **request body**.
- * It **only modifies** the fields that are provided.
- * Other fields **remain unchanged**.
- * It is **not idempotent**. Multiple PATCH requests may lead to different results.
- * It is used when only a **small portion** of a resource needs to be updated.



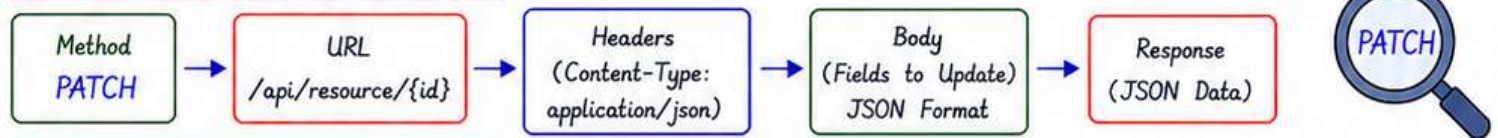
Key Points:

- ✓ PATCH is used to partially update an existing resource.
- ✓ Data is sent in the request body.
- ✓ Only specified fields are updated.
- ✓ Other fields are not affected.
- ✓ It is not idempotent.
- ✓ Used when small changes are needed, like updating a few fields.

PATCH is used to Partially Update an Existing Resource



10.8.2 PATCH Request Structure



10.8.3 Example: PATCH Request

- * Let's partially update the email of user with `id = 1`.
- * API: <https://jsonplaceholder.typicode.com/users/1>

Using Fetch API

```
fetch('https://jsonplaceholder.typicode.com/users/1', {
  method: 'PATCH',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    email: 'new.email@example.com'
  })
})
.then(response => response.json())
.then(data => console.log(data));
```

Using Axios

```
axios.patch('https://jsonplaceholder.typicode.com/users/1', {
  email: 'new.email@example.com'
}, {
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => console.log(response.data))
.catch(error => console.error(error));
```

Sample Response (JSON Data)

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "new.email@example.com",
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "...": "..."
}
```

JSON

10.8.4 Common Use Cases

Update User Email
PATCH `/api/users/1`



Update User Phone
PATCH `/api/users/1`



Update Product Price
PATCH `/api/products/10`



Update Order Status
PATCH `/api/orders/5`



10.8.5 PATCH vs PUT

Feature	PUT	PATCH
Purpose	Updates the entire resource	Updates only specified fields
Data Sent	Full resource data	Partial data (only fields to update)
Idempotent	Yes (Same result for multiple requests)	No (May lead to different results)
Use Case	Replace complete resource	Modify only some fields

Use **PATCH** when you need to update only a few fields. It saves bandwidth and avoids overwriting unchanged data.



Key Takeaway:



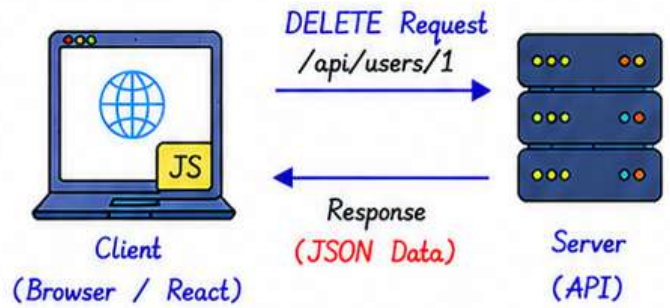
PATCH requests are used to **partially update** existing resources. Only the provided fields are updated, and other fields **remain unchanged**.



10.9 DELETE Requests

10.9.1 What is DELETE Request?

- * DELETE is an HTTP method used to remove a resource.
- * It sends a request to the server to **delete** the specified resource.
- * It does **not** send data in the request body.
- * The server removes the resource and returns a response.
- * If the resource does not exist, the server may return an error (like **404 Not Found**).
- * It is **idempotent**, meaning multiple DELETE requests have the same effect as a single request.
- * Used to delete data like user accounts, posts, files, etc.



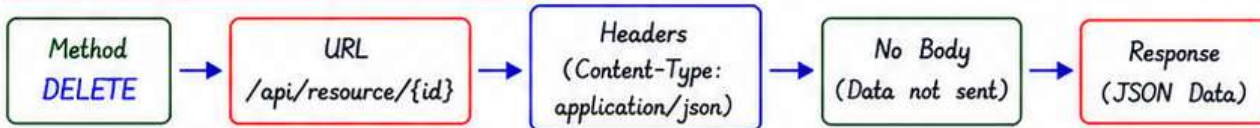
Key Points:

- ✓ DELETE is used to remove a resource.
- ✓ Data is **not** sent in the request body.
- ✓ The resource is permanently removed from the server.
- ✓ If the resource does not exist, it may return an error.
- ✓ It is idempotent (same result for multiple identical requests).
- ✓ Used to delete records like users, posts, files, etc.

DELETE
is used
to Remove
a Resource



10.9.2 DELETE Request Structure



10.9.3 Example: DELETE Request

- * Let's delete a user with id = 1.
- * API: <https://jsonplaceholder.typicode.com/users/1>

Using Fetch API

```
fetch('https://jsonplaceholder.typicode.com/users/1', {
  method: 'DELETE',
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => {
  if (response.ok) {
    console.log('User deleted successfully');
  } else {
    console.error('Failed to delete user');
  }
})
.catch(error => console.error(error));
```

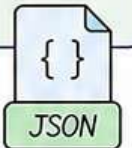
Using Axios

```
axios.delete('https://jsonplaceholder
.typicode.com/users/1', {
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => {
  console.log('User deleted
  successfully');
})
.catch(error => {
  console.error('Failed to delete user');
});
```

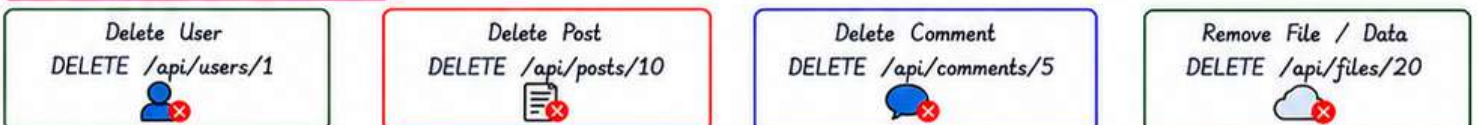
Sample Response (JSON Data)

```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "leanne@gmail.com"
}
```

Note: Some APIs return the deleted resource. Others may return 204 No Content.



10.9.4 Common Use Cases



10.9.5 DELETE vs Other Methods

Method	Purpose	Data Sent	Idempotent	Use Case
GET	Retrieve data	No	Yes	Read data
POST	Create new resource	Yes (in body)	No	Create data
PUT	Update entire resource	Yes (in body)	Yes	Replace resource
PATCH	Update part of resource	Yes (in body)	No	Partial update
DELETE	Delete resource	No	Yes	Remove resource

Use **DELETE** when you want to permanently remove a resource from the server.



Key Takeaway:



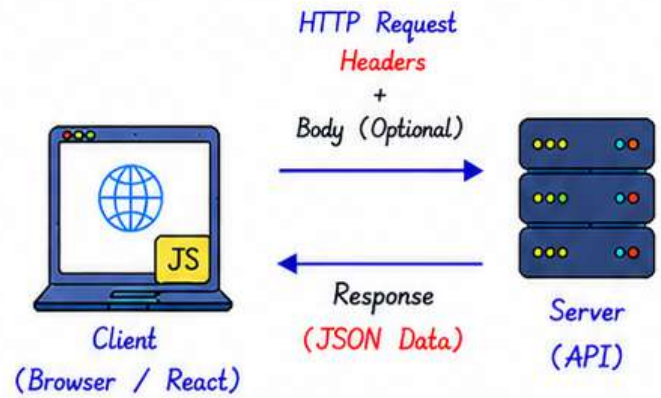
DELETE requests are used to **remove** resources from the server. They do not send data in the body and are **idempotent** (same effect for multiple requests).



10.10 Request Headers

10.10.1 What are Request Headers?

- * Request headers provide **additional information** about the request to the server.
- * They are sent along with the HTTP request.
- * Headers help the server understand the client, the type of data, authentication, preferences, etc.
- * Headers are sent in **key-value pairs**.
- * They are **not** visible in the URL.
- * Headers are **optional**, but some are very important.



Key Points:

- ✓ Headers send extra information about the request.
- ✓ They are in the format: Key: Value
- ✓ They are not part of the request body.
- ✓ They help in content negotiation, authentication, caching, and security.
- ✓ Common headers include: Content-Type, Authorization, Accept, User-Agent, etc.

Headers act like a **note** from the client to the server.



10.10.2 Common Request Headers

Header Name	Description	Example Value	Used For
Accept	Specifies the type of response the client can handle	application/json	Tells server the expected response format
Accept-Language	Specifies the preferred language(s)	en-US,en;q=0.9	Content localization
Content-Type	Specifies the type of data in the request body	application/json	Tells server the format of data being sent
Authorization	Provides authentication credentials	Bearer <token>	Access protected resources
User-Agent	Identifies the client (browser, OS, etc.)	Mozilla/5.0 ...	Analytics, logging, compatibility
Cache-Control	Controls caching behavior	no-cache	Caching instructions
Origin	Specifies the origin (domain) of the request	https://example.com	CORS (Cross-Origin Resource Sharing)
Referer	Shows the URL of the page making the request	https://site.com/page	Analytics, validation, logging
Custom Headers	User-defined headers	X-Client-ID: 12345	Send extra custom info to the server

Note:

Not all headers are mandatory. Some headers are added automatically by the browser.



10.10.3 Example: Request with Headers

Using Fetch API

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'GET',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json',
    'Authorization': 'Bearer YOUR_TOKEN_HERE',
    'User-Agent': 'MyReactApp/1.0'
  }
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error(error));
```

Using Axios

```
axios.get('https://jsonplaceholder.typicode.com/posts', {
  headers: {
    'Accept': 'application/json',
    'Authorization': 'Bearer YOUR_TOKEN_HERE',
    'User-Agent': 'MyReactApp/1.0'
  }
})
.then(response => {
  console.log(response.data);
})
.catch(error => {
  console.error(error);
});
```

Request Example (Headers)

```
GET /posts HTTP/1.1
Host: jsonplaceholder.typicode.com
Accept: application/json
Content-Type: application/json
Authorization: Bearer YOUR_TOKEN
User-Agent: MyReactApp/1.0
Accept-Language: en-US,en;q=0.9
Cache-Control: no-cache
Origin: https://example.com
...
```

HTTP

10.10.4 Best Practices

Send only necessary headers.



Never expose sensitive tokens in the frontend.



Use Authorization header securely.



Always set proper Content-Type.



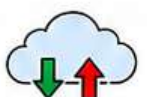
Follow CORS rules when sending requests from frontend.



Key Takeaway:



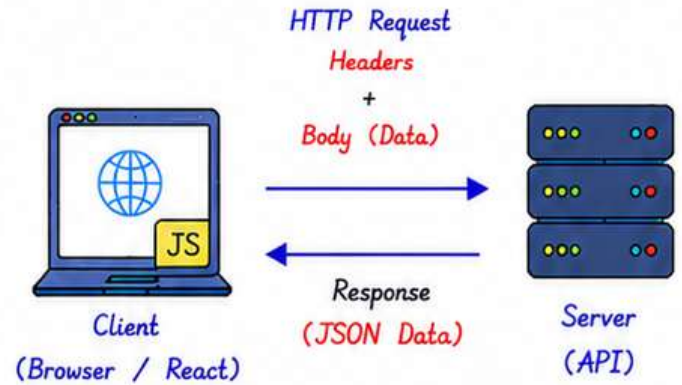
Request headers carry important information from the client to the server. They help in **authentication**, **content negotiation**, **security**, and **better communication**.



10.11 Request Body

10.11.1 What is Request Body?

- * The request body contains the data that the client sends to the server.
- * It is mainly used in methods like **POST, PUT, PATCH**.
- * **GET** and **DELETE** requests usually do not have a body.
- * The body carries the **actual content (data)** of the request.
- * The server reads the body to perform the required operation.
- * The body is **optional** in some requests but **mandatory** in others depending on the API.



Key Points:

- ✓ Request body contains the data sent to the server.
- ✓ Used in POST, PUT, PATCH to create or update data.
- ✓ Data is usually in JSON or other formats.
- ✓ Content-Type header tells the server the format of the body.
- ✓ GET and DELETE requests do not typically have a body.
- ✓ Body is not visible in the URL.

Body is the message/content you send to the server.



10.11.2 Common Formats of Request Body

Format	Content-Type	Example	Description	Used In
JSON	application/json	<pre>{ "name": "John", "email": "john@example.com" }</pre>	Most popular format. Easy to read and write.	APIs (Web, Mobile, Frontend)
Form Data	application/x-www-form-urlencoded	<code>name=John&email=john%40example.com</code>	Key-value pairs. Used in HTML forms.	HTML Forms, Legacy Systems
Text / Plain	text/plain	This is a plain text body	Plain text data.	Logging, Debugging
XML	application/xml	<pre><user> <name>John</name> <email>john@example.com</email> </user></pre>	Structured data in XML format.	Enterprise Applications
Multipart Form Data	multipart/form-data	(Used for file upload along with other fields)	Used to upload files with form fields.	File Upload APIs

10.11.3 Example: Request with Body (POST)

Using Fetch API

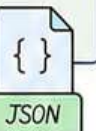
```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    title: 'My First Post',
    body: 'This is the content',
    userId: 1
  })
})
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.error(err));
```

Using Axios

```
axios.post('https://jsonplaceholder.typicode.com/posts', {
  title: 'My First Post',
  body: 'This is the content',
  userId: 1
}, {
  headers: {
    'Content-Type': 'application/json'
  }
})
.then(response => console.log(response.data))
.catch(error => console.error(error));
```

Sample Request Body (JSON)

```
{
  "title": "My First Post",
  "body": "This is the content",
  "userId": 1
}
```



10.11.4 Best Practices

Always set the correct Content-Type header.



Send only necessary data in the body.



Validate data before sending to the server.



Do not send sensitive information in body.



Keep request body size small.



Key Takeaway:



The request body carries the **actual data** you send to the server. It is mainly used in **POST, PUT, and PATCH** requests in formats like JSON, Form Data, etc.



10.12 Query Parameters

10.12.1 What are Query Parameters?

- * Query parameters are **key-value pairs** added to the URL after a **?** symbol.
- * They are used to send **additional data** to the server.
- * They help in **filtering, searching, sorting** and **pagination**.
- * They are part of the URL and are separated by **&** when there are multiple parameters.
- * Query parameters **do not** change the request method (e.g., GET, POST, etc.).

Key Points:

- ✓ Query parameters are appended to the URL after **?**.
- ✓ Key-value pairs are separated by **&**.
- ✓ Used to send **optional data** to the server.
- ✓ Commonly used for **filtering, searching, sorting, pagination**.
- ✓ Only the query string is sent, **not the request body**.
- ✓ Visible in the browser URL and can be bookmarked.

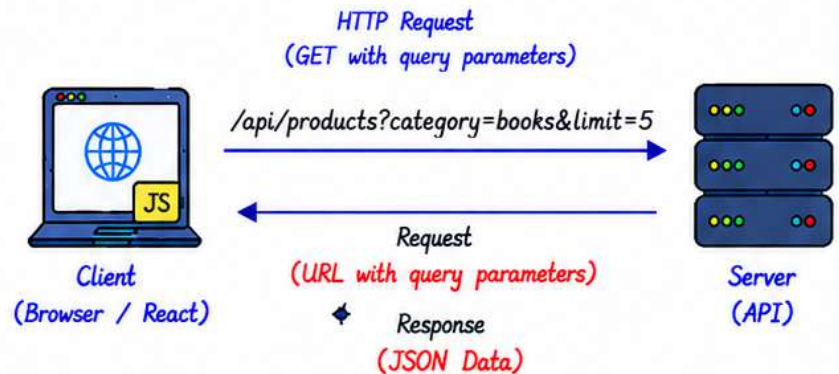
Query parameters are like filters in a search bar — they help you get the right data faster!

URL Structure:

base_url ? key1=value1
& key2=value2
& key3=value3

Example:

/api/products?category=books&limit=5



10.12.2 Example: GET Request with Query Parameters



10.12.3 Example Code (Fetch API)

```
const url = "https://api.example.com/api/products?" +  
  "category=books&limit=5";  
  
fetch(url)  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error("Error:", error));
```

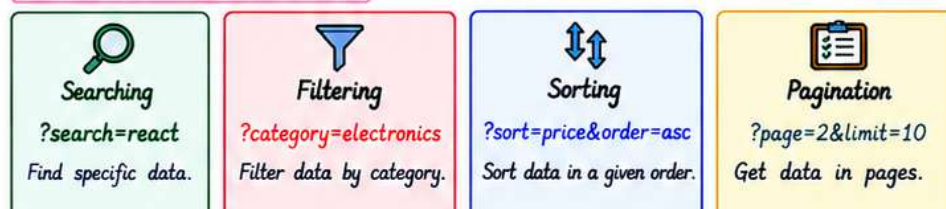
10.12.4 Example Code (Axios)

```
axios.get("https://api.example.com/api/products",{  
  params: {  
    category: "books",  
    limit: 5  
  }  
})  
  .then(response => {  
    console.log(response.data);  
  })  
  .catch(error => console.error(error));
```

10.12.5 Sample Response (JSON)

```
{  
  "total": 12,  
  "limit": 5,  
  "category": "books",  
  "products": [  
    { "id": 1, "name": "Clean Code", "price": 499 },  
    { "id": 2, "name": "Design Patterns", "price": 599 },  
    { "id": 3, "name": "Python Basics", "price": 399 },  
    { "id": 4, "name": "JavaScript Guide", "price": 499 },  
    { "id": 5, "name": "System Design", "price": 699 }  
  ]  
}
```

10.12.6 Common Use Cases



10.12.7 Key Takeaways

- * Query parameters are simple but powerful.
- * They help in filtering, searching, sorting and pagination.
- * Use **?** to start the query string and **&** to add more parameters.
- * They are visible in the URL and do not affect the request method.
- * Widely used in real-world APIs (e.g., GitHub, Google APIs, e-commerce APIs).



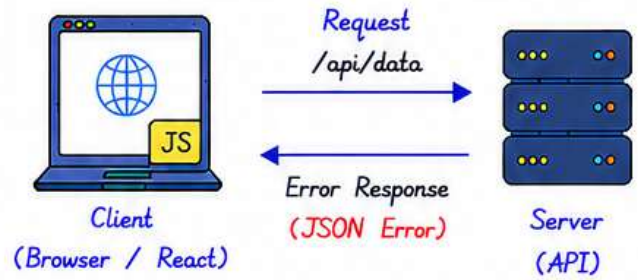
Remember: Query parameters make your API more **flexible, efficient** and **user-friendly**!



10.13 API Error Handling

10.13.1 What is API Error Handling?

- * Errors can occur while making API requests.
- * They may be due to **network issues**, **invalid input**, **server errors**, or **authentication problems**.
- * Proper error handling helps in providing a **better user experience**.
- * It helps in identifying issues and fixing them easily.
- * Always handle errors **gracefully** in your application.



Key Points:

- ✓ APIs may return success or error responses.
- ✓ Errors are returned with appropriate HTTP status codes.
- ✓ Always handle errors in `.catch()` block (Axios) or `try...catch`.
- ✓ Provide meaningful messages to users.
- ✓ Log errors for debugging.
- ✓ Do not expose sensitive error details to users.

Good error handling makes your application **reliable**.



10.13.2 Common HTTP Status Codes (Errors)

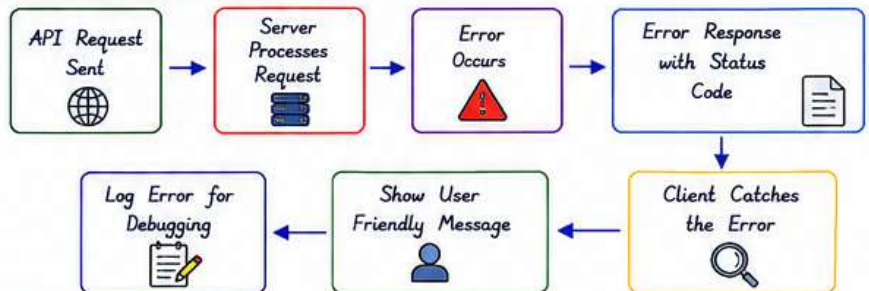
Status Code	Name	Meaning	Example
400	Bad Request	The request is invalid or cannot be understood.	Invalid input, missing parameters
401	Unauthorized	Authentication is required or has failed.	Missing or invalid token
403	Forbidden	The client does not have permission to access.	Accessing restricted resource
404	Not Found	The requested resource does not exist.	Wrong API endpoint
408	Request Timeout	The server timed out waiting for the request.	Slow network or long request
409	Conflict	Conflict with the current state of resource.	Duplicate data, version conflict
422	Unprocessable Entity	The request was well-formed but cannot be processed.	Validation errors
500	Internal Server Error	Something went wrong on the server.	Unexpected server error
503	Service Unavailable	The server is not ready to handle the request.	Server down or overloaded

10.13.3 Example: Error Response (JSON)

```
{
  "success": false,
  "message": "User not found",
  "error": "RESOURCE_NOT_FOUND",
  "statusCode": 404,
  "timestamp": "2024-05-12T10:30:00Z",
  "path": "/api/users/123"
}
```

JSON

10.13.4 Error Handling Flow



10.13.5 Example: Error Handling using Axios

```
import axios from 'axios';
axios.get('https://jsonplaceholder.typicode.com/users/9999')
  .then(response => {
    console.log('Data:', response.data);
  })
  .catch(error => {
    if (error.response) {
      // Server responded with a status code outside 2xx
      console.error('Status:', error.response.status);
      console.error('Message:', error.response.data.message);
    } else if (error.request) {
      // No response received
      console.error('No response received from server.');
```

Explanation

- * `error.response` => Server responded with an error status code. You can access status, data, and headers.
- * `error.request` => Request was made but no response received. Usually a network issue.
- * `error.message` => Other errors (e.g., invalid URL, code errors).

10.13.6 Best Practices

Always handle errors in `.catch()`.



Show meaningful messages to users.



Do not expose internal error details.



Log errors for debugging.



Use appropriate HTTP status codes.

404

Handle network errors gracefully.



Key Takeaway:



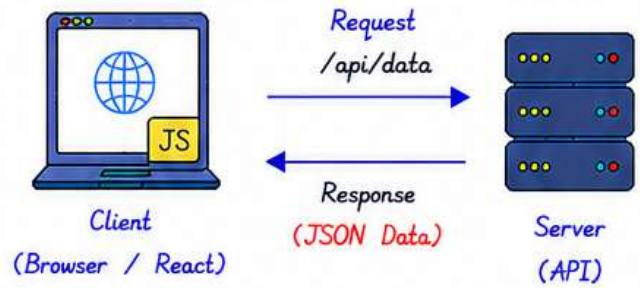
API errors are unavoidable, but proper error handling makes your application **robust**, **user-friendly**, and **easy to maintain**.



10.14 Loading States

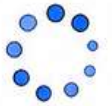
10.14.1 What are Loading States?

- * Loading states indicate that an operation is **in progress**.
- * They are shown while waiting for a response from the server (API call).
- * They improve user experience by providing visual **feedback** and preventing confusion.
- * They **prevent multiple requests** by disabling buttons or inputs.
- * Loading states are **temporary** and are replaced with success or error states.



Why Use Loading States?

- ✓ Provides feedback that something is happening.
- ✓ Reduces user anxiety while waiting.
- ✓ Improves perceived performance.
- ✓ Prevents duplicate submissions.



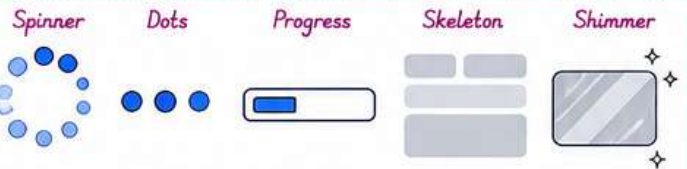
Key Points:

- ✓ Show loading indicator while waiting for API response.
- ✓ Disable buttons/inputs during loading.
- ✓ Hide loader and show data on success.
- ✓ Hide loader and show error message on failure.
- ✓ Keep the UI responsive and informative.
- ✓ Use appropriate loader type for better UX.

Good loading states make users happy!



Common Types of Loaders



10.14.2 When to Show Loading States?

Situation	Description	Example
API Call	While fetching data from server	Fetching user list, product details
Form Submission	While submitting form data	Register, login, update profile
File Upload	While uploading files	Uploading profile picture, documents
Data Processing	While processing or generating data	Generating report, exporting data
Navigation / Routing	While navigating to another page	Route change with data fetching

Note:

Do not show loaders for operations that take less than 200ms. It may cause flicker.



10.14.3 Example: Loading State with Fetch API

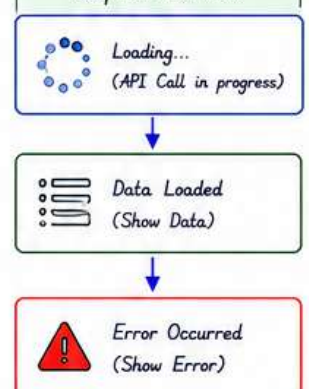
```
const [data, setData] = useState(null);
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);

const fetchData = async () => {
  setLoading(true);
  setError(null);
  try {
    const res = await fetch('https://jsonplaceholder.typicode.com/posts');
    const json = await res.json();
    setData(json);
  } catch (err) {
    setError('Failed to fetch data');
  } finally {
    setLoading(false);
  }
};
```

10.14.4 Example: UI with Loading State (React)

```
if (loading) {
  return (
    <div className="loader-container">
      <div className="spinner"></div>
      <p>Loading data, please wait...</p>
    </div>
  );
}
if (error) {
  return <p className="error">{error}</p>;
}
return (
  <ul>
    {data.map(item => (
      <li key={item.id}>{item.title}</li>
    ))}
  </ul>
);
```

Output / UI Flow



10.14.5 Best Practices

Show loading states immediately after request is made.



Do not block the entire UI, only the necessary part.



Use meaningful text along with loaders.



Disable buttons to prevent multiple requests.



Hide loader after success or error.



Keep loader simple and lightweight.



Key Takeaway:



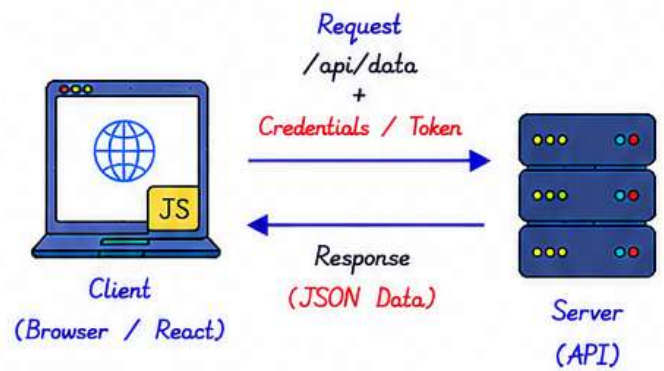
Loading states provide feedback to users while data is being fetched or processed. They improve **user experience**, prevent duplicate actions, and make your application feel fast and reliable.



10.15 API Authentication

10.15.1 What is API Authentication?

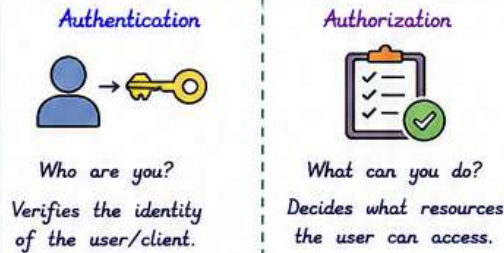
- * Authentication is the process of **verifying the identity** of a user or client before allowing access to an API.
- * It ensures that only **authorized users** can access protected resources.
- * APIs use **tokens, keys, or credentials** to authenticate requests.
- * It is different from **authorization** (which decides what an authenticated user can do).



Key Points:

- ✓ Authentication verifies who you are.
- ✓ Done using API keys, tokens, or credentials.
- ✓ Token-based authentication is most common.
- ✓ Tokens are usually sent in headers.
- ✓ Always use HTTPS to protect credentials.
- ✓ Do not expose sensitive keys in client-side code.

Authentication vs Authorization



10.15.2 Common API Authentication Methods

Method	How it Works	Sent In	Example	Use Cases
API Key	Client sends a unique key with each request.	Query Params or Headers	?api_key=12345 or x-api-key: 12345	Public APIs, simple apps, rate limiting
Bearer Token	Client sends a token (JWT or random string).	Authorization Header	Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...	Most modern APIs, secure access
Basic Auth	Client sends username and password encoded in base64.	Authorization Header	Authorization: Basic dXNlcjpwYXNz	Internal APIs, simple applications
OAuth 2.0	Uses access tokens issued by an authorization server.	Authorization Header	Authorization: Bearer access_token	Third-party login, delegated access

10.15.3 Example: Sending Token with Fetch API

```
const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...';
fetch('https://api.example.com/protected', {
  method: 'GET',
  headers: {
    'Authorization': `Bearer ${token}`,
    'Content-Type': 'application/json'
  }
})
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.error(err));
```



10.15.4 Example: API Key in Query Parameter

```
fetch('https://api.example.com/data?api_key=12345', {
  method: 'GET'
})
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.error(err));
```



10.15.6 Example: Basic Authentication

```
const user = 'admin';
const pass = 'password123';
const encoded = btoa(`${user}:${pass}`); // base64 encode
fetch('https://api.example.com/data', {
  method: 'GET',
  headers: {
    'Authorization': `Basic ${encoded}`
  }
})
.then(res => res.json())
.then(data => console.log(data))
.catch(err => console.error(err));
```



*** --

10.15.5 Example: Using Axios with Token

```
const token = 'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...';
axios.get('https://api.example.com/protected', {
  headers: {
    Authorization: `Bearer ${token}`
  }
})
.then(response => console.log(response.data))
.catch(error => console.error(error));
```



Note:

- * Never hardcode tokens or keys in public repositories.
- * Store tokens securely (e.g., environment variables).
- * Always use HTTPS for authenticated requests.



10.15.7 Best Practices

Use HTTPS for all API requests.



HTTPS

Do not expose API keys or tokens in client code.



Use short-lived tokens and refresh tokens.



Store tokens securely (localStorage with care / cookies).



Validate and verify tokens on the server side.



Implement rate limiting and log auth failures.



Key Takeaway:



API Authentication verifies the identity of the client using keys, tokens, or credentials. It protects your APIs from unauthorized access and ensures secure communication.



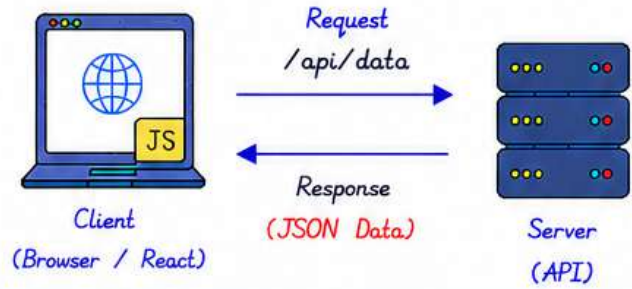
10.16 Environment Variables

10.16.1 What are Environment Variables?

- * Environment variables are **key-value pairs** stored outside of your code.
- * They store **sensitive** or **environment-specific** information securely.
- * They help keep your code **clean, secure**, and portable across environments.
- * **Examples:** API keys, database URLs, passwords, secrets, config values.

Key Points:

- ✓ Store configuration and secrets outside code.
- ✓ Use `.env` file during development.
- ✓ Access variables using `process.env` in Node.js.
- ✓ Never commit `.env` files to version control.
- ✓ Each environment has its own set of variables.



Why Use Environment Variables?

- ✓ Keeps **sensitive data** out of source code.
- ✓ Different values for **dev, test, and production**.
- ✓ **Easy** to manage and update without code changes.
- ✓ Improves **security** and **portability**.



10.16.2 Common Environment Variables

Variable Name	Example Value	Description	Used In
PORT	5000	Port number for the server	All Node.js apps
NODE_ENV	development / production	Environment type	All environments
API_KEY	sk_live_abc123xyz	Third-party API key	Payment / Maps / AI APIs
DB_HOST	localhost	Database host	Database connection
DB_USER	root	Database username	Database connection
DB_PASS	mypassword@123	Database password	Database connection
JWT_SECRET	my_jwt_secret_key	Secret key for JWT	Authentication
BASE_URL	https://api.example.com	Base URL of API	Frontend / Backend

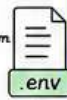
10.16.3 Example: .env File

```
# Server Configuration
PORT=5000
NODE_ENV=development

# Database Configuration
DB_HOST=localhost
DB_USER=root
DB_PASS=mypassword@123
DB_NAME=myapp

# API Keys
API_KEY=sk_live_abc123xyz
JWT_SECRET=my_jwt_secret_key_123

# Other Config
BASE_URL=https://api.example.com
```



10.16.4 Accessing in Node.js

```
require('dotenv').config(); // Load .env file

const express = require('express');
const app = express();

const port = process.env.PORT;
const dbHost = process.env.DB_HOST;
const apiKey = process.env.API_KEY;
const jwtSecret = process.env.JWT_SECRET;

app.get('/', (req, res) => {
  res.send('Server is running...');
});

app.listen(port, () => {
  console.log(`Server running on port ${port}`);
});
```



10.16.5 Accessing in Frontend (React)

```
// In React (Vite)
const apiBase = import.meta.env.VITE_API_BASE_URL;
const apiKey = import.meta.env.VITE_API_KEY;
```



Note:

- * In Vite, environment variables must start with `VITE_`.
- * In Create React App, use `REACT_APP_` prefix.



Do's and Don'ts

Do's	Don'ts
✓ Use <code>.env</code> for secrets. ✓ Use different <code>.env</code> files for dev, test, prod. ✓ Keep <code>.env</code> out of git.	✗ Never hardcode secrets. ✗ Never commit <code>.env</code> files. ✗ Never expose secrets in client-side code.

10.16.6 Multiple Environment Files

File	Purpose	Example Variables
<code>.env</code>	Default (local development)	PORT, DB_HOST, API_KEY
<code>.env.development</code>	Development environment	DB_HOST=localhost
<code>.env.test</code>	Testing environment	DB_HOST=test-db
<code>.env.production</code>	Production environment	DB_HOST=prod-db

Example Folder Structure

```
myapp/
├── .env
├── .env.development
├── .env.production
├── .env.test
└── server.js
```

10.16.8 Best Practices

Use environment variables for all sensitive data.



Use different `.env` files for each environment.



Use meaningful variable names.



Keep `.env` files out of version control.



Validate required env variables at startup.



Regularly rotate keys and secrets.



Key Takeaway:



Environment variables help you manage configuration and secrets securely. They keep your code **clean, safe**, and **easy** to deploy across environments.



10.17 CORS

10.17.1 What is CORS?

- * CORS stands for **Cross-Origin Resource Sharing**.
- * It is a security feature implemented by browsers.
- * It allows or restricts web pages from making requests to a **different domain (origin)**.
- * CORS helps servers specify who can access their resources.
- * It prevents unauthorized access from other websites.

Client (Origin 1)

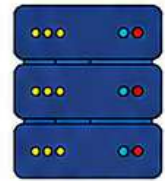


https://app.com
(Frontend)

Cross-Origin
Request

Response
(CORS Headers)

Server (Origin 2)



https://api.com
(Backend API)

Key Points:

- ✓ Same-Origin Policy (SOP) blocks requests across different origins by default.
- ✓ CORS uses HTTP headers to allow such requests safely.
- ✓ Server must explicitly allow the requested origin.
- ✓ If not allowed, the browser blocks the response.

What is an Origin?

An origin is defined by:

- Protocol (http / https)
- Domain (example.com)
- Port (80, 443, 3000, etc.)

Origin = `https` :// `example.com` : `443`
 ↓ ↓ ↓
 Protocol Domain Port

Change in any of these = Different Origin

10.17.2 How CORS Works?

1 Browser Sends Request

Frontend on Origin 1 sends request to Origin 2



2 Server Checks Origin

Server checks if the Origin is allowed in CORS policy



3 Server Sends Response with CORS Headers

If allowed, server sends response with **Access-Control-Allow-*** headers



4 Browser Allows or Blocks

Browser allows the response if headers are valid, otherwise blocks it



Common CORS Headers

- Access-Control-Allow-Origin
- Access-Control-Allow-Methods
- Access-Control-Allow-Headers
- Access-Control-Allow-Credentials
- Access-Control-Max-Age

10.17.3 Example

Without CORS (Blocked)

Request from: `https://app.com`
To: `https://api.com/data`

Server Response Headers:
(No CORS headers)

Result:
Browser blocks the response.
Error in Console:
"Access to fetch at 'https://api.com/data' from origin 'https://app.com' has been blocked by CORS policy."



With CORS (Allowed)

Request from: `https://app.com`
To: `https://api.com/data`

Server Response Headers:
Access-Control-Allow-Origin: `https://app.com`
Access-Control-Allow-Methods: `GET, POST, PUT, DELETE`
Access-Control-Allow-Headers: `Content-Type, Authorization`
Access-Control-Allow-Credentials: `true`

Result:
Browser allows the response.
Data is accessible in the frontend.



10.17.4 Simple CORS Example in Node.js

```
const express = require('express');
const cors = require('cors');
const app = express();

// Enable CORS for all origins
app.use(cors());

// Or allow specific origin
app.use(cors({
  origin: 'https://app.com',
  methods: ['GET', 'POST', 'PUT', 'DELETE'],
  allowedHeaders: ['Content-Type', 'Authorization'],
  credentials: true
}));

app.get('/data', (req, res) => {
  res.json({ message: 'Hello from API!' });
});

app.listen(5000, () => console.log('Server running...'));
```

10.17.5 Best Practices

Allow only trusted origins, not * in production.



Use specific HTTP methods and headers required.



Enable credentials only if necessary.



Set appropriate Access-Control-Max-Age to reduce preflight requests.



Always test CORS in development before deploying.



Key Takeaway:



CORS allows **secure cross-origin communication** between frontend and backend. Configure it properly on the server to control who can access your API.

