# JavaScript Fundamentals

1. Variables & Data Types

Why It’s Important:

Variables hold data so your program can remember and work with information. Data types define what kind of data you’re working with (text, numbers, lists, etc.).

Real-World Example:

Storing a user’s name, their age, or a list of their favorite products.

Concepts:

• let: Allows reassignment of variables.

• const: Creates a constant variable (cannot be reassigned).

• Primitive types: string, number, boolean, null, undefined.

• Reference types: arrays, objects.

Example:

const name = "Sofia";

let age = 7;

const isStudent = true;

const hobbies = ["drawing", "dancing"];

const profile = { name: "Sofia", age };

console.log(name, age, isStudent, hobbies, profile);

Practice:

1. Create a variable for your favorite color and log it.

2. Make an array of your 3 favorite foods.

3. Create an object representing a car with properties: brand, model, year.

2. Reference Variables

Why It’s Important:

Understanding reference types helps prevent bugs when working with arrays and objects. Changing one variable might unexpectedly change another if they share the same reference.

Real-World Example:

If you store a shopping cart object in multiple places in your app, you need to know if updates in one place affect the other.

Concepts:

• Primitive types are copied by value.

• Reference types (arrays, objects) are copied by reference.

Examples:

Primitive Copy:

let a = 5;

let b = a;

b = 10;

console.log(a); // 5

console.log(b); // 10

Reference Copy:

const person1 = { name: "Sofia" };

const person2 = person1;

person2.name = "Luna";

console.log(person1.name); // Luna

Safe Copy with Spread:

const person3 = { name: "Sofia" };

const person4 = { ...person3 };

person4.name = "Luna";

console.log(person3.name); // Sofia

console.log(person4.name); // Luna

Practice:

1. Create two variables x and y with the value 5. Change y to 10. What happens to x?

2. Create an object user1 with a name property. Assign it to user2. Change user2.name. What happens to user1.name?

3. Copy an object safely using the spread operator.

3. Functions

Why It’s Important:

Functions re-use code and break down tasks into smaller pieces. This makes your code cleaner and easier to maintain.

Real-World Example:

A function could handle calculating a total price every time a user adds items to their cart.

Concepts:

• Functions can be written as declarations or arrow functions.

• Parameters: Inputs to the function.

• Return values: Output from the function.

Examples:

// Function declaration

function greet(name) {

return `Hello, ${name}!`;

}

// Arrow function

const add = (a, b) => a + b;

console.log(greet("Sofia")); // Hello, Sofia!

console.log(add(3, 4)); // 7

Practice:

1. Write a function that returns the square of a number.

2. Write an arrow function that takes two numbers and returns their sum.

3. Write a function that takes a name and returns “Good morning, {name}!”

4. Conditionals

Why It’s Important:

Conditionals let your program make decisions. They’re essential for controlling what your app does based on user actions or data.

Real-World Example:

If a user is logged in, show their dashboard. If not, show the login page.

Concepts:

• if / else if / else control flow.

• Comparison operators: ===, !==, <, >, <=, >=.

Examples:

const age = 18;

if (age < 13) {

console.log("Child");

} else if (age < 20) {

console.log("Teenager");

} else {

console.log("Adult");

}

Switch Example:

const day = "Monday";

switch (day) {

case "Monday":

console.log("Start of the week!");

break;

case "Friday":

console.log("Almost weekend!");

break;

default:

console.log("Just another day.");

}

Practice:

1. Write a function that checks if a number is positive, negative, or zero.

2. Use a switch statement to print a message based on the day of the week ("Monday", "Tuesday", etc.).

5. Loops

Why It’s Important:

Loops repeat actions. They let you process lists of data without writing the same code over and over.

Real-World Example:

Display a list of products on an e-commerce page by looping through an array of product data.

Concepts:

• For loop: Repeats a set number of times.

• While loop: Repeats as long as a condition is true.

• For…of loop: Loops over items in an array.

Examples:

// For loop

for (let i = 1; i <= 5; i++) {

console.log(i);

}

// While loop

let count = 5;

while (count >= 1) {

console.log(count);

count--;

}

// For...of loop

const fruits = ["apple", "banana", "cherry"];

for (let fruit of fruits) {

console.log(fruit);

}

Practice:

1. Use a for loop to print numbers 1 to 5.

2. Use a while loop to count down from 5 to 1.

3. Use a for…of loop to print each item in an array of fruits.

6. Arrays & Objects

Why It’s Important:

Arrays and objects help you organize and store collections of data. They are the building blocks of data in any application.

Real-World Example:

• Array: A list of blog posts.

• Object: A user profile with a name, email, and preferences.

Concepts:

• Arrays: Lists of data.

• Methods: push, pop, length.

• Objects: Key-value pairs.

Examples:

// Array

const todoList = ["Learn JS"];

todoList.push("Practice coding");

console.log(todoList); // ["Learn JS", "Practice coding"]

todoList.pop();

console.log(todoList); // ["Learn JS"]

// Object

const user = { name: "Sofia", age: 7 };

console.log(user.name); // Sofia

Practice:

1. Create an array of 3 hobbies and add a new one using push.

2. Remove the last item from that array using pop.

3. Create an object for a book with properties: title, author, pages.

7. Array Methods: map & reduce

Why It’s Important:

These methods help process arrays efficiently without writing loops manually.

Real-World Example:

• Use map to create a new list of product prices with a discount applied.

• Use reduce to calculate the total cost in a shopping cart.

Concepts:

• map: Transforms each item in an array.

• reduce: Combines all items into a single value.

Examples:

// map

const numbers = [1, 2, 3];

const doubled = numbers.map(num => num \* 2);

console.log(doubled); // [2, 4, 6]

// reduce

const sum = numbers.reduce((total, num) => total + num, 0);

console.log(sum); // 6

Practice:

1. Use map to double the numbers in an array [1, 2, 3].

2. Use reduce to sum all numbers in an array [4, 5, 6].

8. Basic Error Handling

Why It’s Important:

Errors happen—good apps handle them gracefully. try...catch ensures your app doesn’t crash when something goes wrong.

Real-World Example:

Catching errors from API requests and showing friendly error messages to users.

Concepts:

• try…catch block handles errors.

Examples:

// Handling division by zero

try {

const result = 10 / 0;

console.log(result); // Infinity

} catch (error) {

console.error("Error:", error);

}

// Manual error

try {

throw new Error("This is a custom error");

} catch (error) {

console.error(error.message);

}

Practice:

1. Use try...catch to handle an error when dividing a number by zero.

2. Manually throw an error with a custom message.

9. Console for Debugging

Why It’s Important:

The console helps you understand what your code is doing. It’s the best tool for debugging.

Real-World Example:

Print API responses or variable values to ensure data is correct before displaying it.

Concepts:

• console.log: Print general info.

• console.error: Print error messages.

• console.table: Display objects/arrays as a table.

Examples:

const user = { name: "Sofia", age: 7 };

console.log("User:", user);

console.error("Something went wrong!");

console.table(user);

Practice:

1. Use console.log to print a greeting.

2. Use console.error to print an error message.

3. Use console.table to print an object.

This completes the JavaScript Fundamentals section.

# JavaScript in the Browser (DOM Manipulation)

Overview:

Why It’s Important:

The DOM (Document Object Model) lets JavaScript interact with the webpage—changing text, adding/removing elements, or responding to user actions. It’s how your app comes alive for users.

Real-World Example:

• Clicking a “Submit” button to send a form.

• Showing/hiding a popup.

• Dynamically updating a to-do list when adding items.

1. HTML & CSS Basics

Concepts:

• HTML is the structure of the page (headings, buttons, forms).

• CSS is the style (colors, fonts, layouts).

• JavaScript adds behavior (interactivity).

Example:

<!-- HTML -->

<button id="clickMe">Click Me!</button>

<p id="message"></p>

<!-- CSS -->

<style>

#clickMe {

background-color: blue;

color: white;

padding: 10px;

}

</style>

2. Selecting Elements

Why It’s Important:

You need to select elements on the page to interact with them using JavaScript.

Concepts:

• document.querySelector(): Selects the first matching element.

• document.querySelectorAll(): Selects all matching elements.

• document.getElementById(): Selects an element by its ID.

Example:

const button = document.querySelector("#clickMe");

const message = document.getElementById("message");

3. Event Listeners

Why It’s Important:

Event listeners respond to user actions like clicks, typing, or hovering.

Concepts:

• addEventListener("click", callbackFunction)

Example:

button.addEventListener("click", () => {

message.textContent = "Button clicked!";

});

4. Modifying the DOM

Why It’s Important:

Modifying the DOM lets you change the page in real-time—add new items, remove them, or update text and styles.

Concepts:

• textContent: Change or read text inside an element.

• innerHTML: Set HTML content.

• classList.add/remove/toggle: Change CSS classes.

Example:

message.textContent = "Hello!";

message.classList.add("highlight");

5. Forms and User Input

Why It’s Important:

Forms collect user data. JavaScript handles the form submission, prevents page reload, and processes input.

Concepts:

• input.value: Get the value from a form input.

• event.preventDefault(): Stop the form from refreshing the page.

Example:

<form id="nameForm">

<input type="text" id="nameInput" placeholder="Enter your name" />

<button type="submit">Submit</button>

</form>

<p id="greeting"></p>

const form = document.getElementById("nameForm");

const input = document.getElementById("nameInput");

const greeting = document.getElementById("greeting");

form.addEventListener("submit", (e) => {

e.preventDefault();

greeting.textContent = `Hello, ${input.value}!`;

input.value = ""; // Clear the input

});

Hands-On Practice:

1. Selecting Elements

• Question: Select the <button> with the ID "myButton".

Answer:

const button = document.getElementById("myButton");

2. Event Listener

• Question: Add a click event to the button that logs "Button was clicked!".

Answer:

button.addEventListener("click", () => {

console.log("Button was clicked!");

});

3. Modify the DOM

• Question: Select a <p> tag with the ID "status" and change its text to "Loading...".

Answer:

const status = document.getElementById("status");

status.textContent = "Loading...";

4. Toggle Class

• Question: Toggle a class "active" on a <div> when a button is clicked.

Answer:

const div = document.querySelector(".box");

button.addEventListener("click", () => {

div.classList.toggle("active");

});

5. Form Handling

• Question: Prevent a form from refreshing the page and log the input value.

Answer:

form.addEventListener("submit", (e) => {

e.preventDefault();

console.log(input.value);

});

Real-World Mini Project: Build a Simple To-Do List

HTML:

<input type="text" id="todoInput" placeholder="Add a task" />

<button id="addTodo">Add</button>

<ul id="todoList"></ul>

JavaScript:

const todoInput = document.getElementById("todoInput");

const addTodo = document.getElementById("addTodo");

const todoList = document.getElementById("todoList");

addTodo.addEventListener("click", () => {

const task = todoInput.value.trim();

if (task) {

const li = document.createElement("li");

li.textContent = task;

todoList.appendChild(li);

todoInput.value = ""; // Clear input

}

});

# Modern JavaScript (ES6+)

Overview:

Why It’s Important:

Modern JavaScript (ES6+) introduced cleaner syntax and powerful features that make writing code easier, faster, and less error-prone. These tools are standard in all React and backend projects today.

Real-World Example:

Every modern React or Node.js app uses arrow functions, destructuring, async/await, and array methods like map and reduce.

1. Arrow Functions

Why It’s Important:

Arrow functions are shorter to write and are common in React, especially for event handlers and callbacks.

Concepts:

• Shorter syntax for functions.

• Don’t bind their own this (important in React).

Example:

// Regular function

function add(a, b) {

return a + b;

}

// Arrow function

const addArrow = (a, b) => a + b;

console.log(add(2, 3)); // 5

console.log(addArrow(2, 3)); // 5

2. Destructuring

Why It’s Important:

Destructuring lets you pull out specific values from arrays or objects, which is useful when working with API responses or props in React.

Concepts:

• Object destructuring: Pull out properties.

• Array destructuring: Pull out elements.

Examples:

Object Destructuring:

const user = { name: "Sofia", age: 7 };

const { name, age } = user;

console.log(name); // Sofia

console.log(age); // 7

Array Destructuring:

const colors = ["red", "blue"];

const [firstColor, secondColor] = colors;

console.log(firstColor); // red

console.log(secondColor); // blue

3. Spread & Rest Operators

Why It’s Important:

The spread operator copies arrays/objects or combines them.

The rest operator collects multiple values into an array.

Concepts:

• Spread (...) for copying or combining.

• Rest (...) for collecting multiple arguments.

Examples:

Spread:

const arr1 = [1, 2];

const arr2 = [...arr1, 3, 4];

console.log(arr2); // [1, 2, 3, 4]

Rest:

function sum(...numbers) {

return numbers.reduce((total, num) => total + num, 0);

}

console.log(sum(1, 2, 3)); // 6

4. Template Literals

Why It’s Important:

Template literals make string building easier by allowing variables inside strings without needing + operators.

Concepts:

• Use backticks (`).

• Insert variables with ${}.

Example:

const name = "Sofia";

const message = `Hello, ${name}!`;

console.log(message); // Hello, Sofia!

5. Promises & Async/Await

Why It’s Important:

Promises handle asynchronous tasks (like fetching data from an API).

async/await makes working with promises look like normal code.

Concepts:

• Promise: Represents a future value.

• async/await: Makes asynchronous code easier to read.

Example:

Promise:

const fetchData = () => {

return new Promise((resolve) => {

setTimeout(() => resolve("Data loaded!"), 1000);

});

};

fetchData().then(data => console.log(data)); // Data loaded!

async/await:

const fetchAsync = async () => {

const data = await fetchData();

console.log(data);

};

fetchAsync(); // Data loaded!

6. Array Methods: map, filter, reduce (Review)

Why It’s Important:

These methods process arrays without writing manual loops.

• map: Creates a new array by transforming items.

• filter: Creates a new array by selecting items.

• reduce: Combines items into a single value.

Examples:

map:

const nums = [1, 2, 3];

const doubled = nums.map(num => num \* 2);

console.log(doubled); // [2, 4, 6]

filter:

const even = nums.filter(num => num % 2 === 0);

console.log(even); // [2]

reduce:

const sum = nums.reduce((total, num) => total + num, 0);

console.log(sum); // 6

Hands-On Practice:

1. Arrow Functions

• Question: Write an arrow function that multiplies two numbers.

Answer:

const multiply = (a, b) => a \* b;

console.log(multiply(3, 4)); // 12

2. Destructuring

• Question: Destructure the name and age from { name: "Luna", age: 5 }.

Answer:

const user = { name: "Luna", age: 5 };

const { name, age } = user;

console.log(name); // Luna

console.log(age); // 5

3. Spread & Rest

• Question: Combine [1, 2] and [3, 4] into one array.

Answer:

const arr1 = [1, 2];

const arr2 = [3, 4];

const combined = [...arr1, ...arr2];

console.log(combined); // [1, 2, 3, 4]

• Question: Write a function that sums any number of arguments using rest.

Answer:

const sum = (...nums) => nums.reduce((total, num) => total + num, 0);

console.log(sum(1, 2, 3)); // 6

4. Template Literals

• Question: Create a greeting that says "Hi, {name}!" using template literals.

Answer:

const name = "Luna";

const greeting = `Hi, ${name}!`;

console.log(greeting); // Hi, Luna!

5. Promises & Async/Await

• Question: Write a function that waits 1 second and logs "Done!" using async/await.

Answer:

const waitOneSecond = () => {

return new Promise(resolve => setTimeout(resolve, 1000));

};

const run = async () => {

await waitOneSecond();

console.log("Done!");

};

run();

Git & GitHub Basics

Overview:

Why It’s Important:

Git is a version control system that tracks changes to your code over time.

GitHub is a cloud platform where you store your Git repositories and collaborate with others.

Real-World Example:

• Working on a React app and accidentally break something? Git lets you go back to a previous version.

• Sharing your projects online or collaborating with a team? Use GitHub.

1. What is Version Control?

Concepts:

• Tracks history of code changes.

• Allows undoing mistakes.

• Supports collaboration between developers.

Real-World Example:

Save different versions of your website as you build, so you can rollback if something breaks.

2. Git Basics (Local Version Control)

Concepts:

• Repository (repo): A project folder tracked by Git.

• Commit: A snapshot of your changes.

Commands:

git init # Start tracking a folder

git status # Check what’s changed

git add . # Stage all changes

git commit -m "Message" # Save changes with a message

3. GitHub (Remote Version Control)

Concepts:

• Remote repo: Your project stored on GitHub.

• Push: Send local changes to GitHub.

• Pull: Get changes from GitHub.

Workflow:

1. Create a repo on GitHub.

2. Connect your local repo to GitHub:

git remote add origin https://github.com/your-username/your-repo.git

git push -u origin main

4. Common Git Workflow

1. Make changes in your code.

2. git status (see what changed).

3. git add . (stage changes).

4. git commit -m "Describe changes" (save).

5. git push (send to GitHub).

Hands-On Practice:

1. Initialize a Git Repository

• Question: How do you turn a folder into a Git repository?

Answer:

git init

2. Stage and Commit Changes

• Question: How do you stage all changes and commit them with a message "Initial commit"?

Answer:

git add .

git commit -m "Initial commit"

3. Push to GitHub

• Question: How do you push your local repository to GitHub after connecting the remote?

Answer:

git push -u origin main

4. Check Status

• Question: How do you check which files have been changed but not committed yet?

Answer:

git status

5. Clone a Repository

• Question: How do you copy an existing GitHub repo to your local machine?

Answer:

git clone https://github.com/username/repo.git

Mini Project: Create and Push a Project to GitHub

1. Create a folder for a new project (my-first-project).

2. Inside, create an index.html file with basic content.

3. Run:

git init

git add .

git commit -m "Initial commit"

4. Create a new repo on GitHub (without a README).

5. Connect and push:

git remote add origin https://github.com/your-username/my-first-project.git

git push -u origin main

React Basics

Overview:

Why It’s Important:

React is a JavaScript library for building user interfaces. It lets you create reusable components that update efficiently when data changes.

Real-World Example:

• Building a to-do list, weather app, or dashboard where the page updates automatically when data changes.

1. What is React?

Concepts:

• Component-based: Break your UI into small, reusable pieces (components).

• Declarative: Describe what you want to show, and React handles the updates.

2. Create React App

Why It’s Important:

This sets up a ready-to-go React project with all the tools you need.

Command:

npx create-react-app my-app

cd my-app

npm start

• Visit http://localhost:3000 to see your app running.

3. Components (Function Components)

Why It’s Important:

Components are the building blocks of React. Each one represents a piece of the UI.

Concepts:

• Function component: A simple function that returns JSX.

Example:

function Welcome() {

return <h1>Hello, world!</h1>;

}

• JSX: A syntax extension that looks like HTML inside JavaScript.

const element = <h1>Hello!</h1>;

4. Props (Passing Data)

Why It’s Important:

Props let you pass data to components, making them dynamic.

Concepts:

• Props are read-only data passed from parent to child.

Example:

function Welcome(props) {

return <h1>Hello, {props.name}!</h1>;

}

<Welcome name="Sofia" />

5. State (useState Hook)

Why It’s Important:

State lets components remember data and re-render when it changes.

Concepts:

• useState is a React hook that adds state to function components.

Example:

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0); // state: count, function: setCount

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

6. Event Handling in React

Why It’s Important:

Lets you respond to user actions like clicking buttons or submitting forms.

Concepts:

• Use camelCase event names (onClick, onSubmit).

• Use functions as event handlers.

Example:

<button onClick={() => alert("Clicked!")}>Click Me</button>

7. Conditional Rendering

Why It’s Important:

Show different UI based on data (like showing a login button when logged out).

Concepts:

• Use if statements or ternary operators.

Example:

function Greeting({ isLoggedIn }) {

return (

<div>

{isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}

</div>

);

}

Hands-On Practice:

1. Create a Component

• Question: Create a component called Hello that renders <h1>Hello, React!</h1>.

Answer:

function Hello() {

return <h1>Hello, React!</h1>;

}

2. Use Props

• Question: Create a component Greet that takes a prop name and displays "Hello, {name}!".

Answer:

function Greet(props) {

return <h1>Hello, {props.name}!</h1>;

}

<Greet name="Luna" />

3. Add State (useState)

• Question: Create a counter component with a button that increments the count.

Answer:

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increment</button>

</div>

);

}

4. Conditional Rendering

• Question: Show "Welcome!" if isLoggedIn is true, else show "Please log in".

Answer:

function Greeting({ isLoggedIn }) {

return (

<div>

{isLoggedIn ? <p>Welcome!</p> : <p>Please log in.</p>}

</div>

);

}

Mini Project: Build a Simple Click Counter

Steps:

1. Create a Counter component.

2. Add state to track the count.

3. Add a button to increment the count.

Code:

import { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<h2>Click Counter</h2>

<p>You've clicked {count} times.</p>

<button onClick={() => setCount(count + 1)}>Click me</button>

</div>

);

}

export default Counter;

React Advanced Concepts

Overview:

Why It’s Important:

This section helps you control component behavior beyond just rendering. You’ll learn how to handle side effects, manage forms, and share state between components—essential for real-world apps.

Real-World Example:

• Fetch data from an API when a component loads.

• Handle form inputs for login/signup.

• Share state between multiple components (like parent and child).

1. useEffect (Side Effects)

Why It’s Important:

useEffect runs code after rendering—for things like fetching data, setting up subscriptions, or updating the DOM outside React.

Concepts:

• Runs after every render by default.

• Can limit when it runs (with dependency array).

Example:

import { useState, useEffect } from 'react';

function Timer() {

const [seconds, setSeconds] = useState(0);

useEffect(() => {

const interval = setInterval(() => {

setSeconds(prev => prev + 1);

}, 1000);

return () => clearInterval(interval); // cleanup

}, []);

return <p>Seconds: {seconds}</p>;

}

2. Forms & Controlled Components

Why It’s Important:

Handling forms in React means keeping inputs in sync with state—this gives you full control over user input.

Concepts:

• Controlled component: Input’s value is managed by React state.

• onChange: Updates state as the user types.

Example:

import { useState } from 'react';

function NameForm() {

const [name, setName] = useState("");

return (

<form onSubmit={(e) => {

e.preventDefault();

alert(`Hello, ${name}!`);

}}>

<input

type="text"

value={name}

onChange={(e) => setName(e.target.value)}

placeholder="Enter your name"

/>

<button type="submit">Submit</button>

</form>

);

}

3. Lifting State Up

Why It’s Important:

When multiple components need access to the same state, lift the state to their common parent.

Concepts:

• Move state to the nearest shared parent.

• Pass data via props.

Example:

import { useState } from 'react';

function Parent() {

const [message, setMessage] = useState("");

return (

<div>

<ChildInput setMessage={setMessage} />

<ChildDisplay message={message} />

</div>

);

}

function ChildInput({ setMessage }) {

return (

<input onChange={(e) => setMessage(e.target.value)} placeholder="Type a message" />

);

}

function ChildDisplay({ message }) {

return <p>Message: {message}</p>;

}

Hands-On Practice:

1. useEffect

• Question: Use useEffect to log "Component mounted!" when a component renders.

Answer:

import { useEffect } from 'react';

function Logger() {

useEffect(() => {

console.log("Component mounted!");

}, []);

return <p>Check the console!</p>;

}

2. Controlled Form

• Question: Build an input that displays live text as the user types.

Answer:

import { useState } from 'react';

function LiveInput() {

const [text, setText] = useState("");

return (

<div>

<input

value={text}

onChange={(e) => setText(e.target.value)}

placeholder="Type here"

/>

<p>You typed: {text}</p>

</div>

);

}

3. Lifting State Up

• Question: Share a color state between two child components—one selects the color, the other displays it.

Answer:

import { useState } from 'react';

function ColorApp() {

const [color, setColor] = useState("red");

return (

<div>

<ColorPicker setColor={setColor} />

<ColorDisplay color={color} />

</div>

);

}

function ColorPicker({ setColor }) {

return (

<select onChange={(e) => setColor(e.target.value)}>

<option value="red">Red</option>

<option value="blue">Blue</option>

</select>

);

}

function ColorDisplay({ color }) {

return <p>The selected color is {color}</p>;

}

Mini Project: To-Do List with Form

Steps:

1. Use useState for the list of tasks and input value.

2. Use useEffect to log when a new task is added.

3. Lifting state isn’t needed here, but practice managing the input.

Code:

import { useState, useEffect } from 'react';

function TodoApp() {

const [tasks, setTasks] = useState([]);

const [input, setInput] = useState("");

const addTask = (e) => {

e.preventDefault();

if (!input) return;

setTasks([...tasks, input]);

setInput("");

};

useEffect(() => {

if (tasks.length > 0) {

console.log("New task added!");

}

}, [tasks]);

return (

<div>

<h2>To-Do List</h2>

<form onSubmit={addTask}>

<input

value={input}

onChange={(e) => setInput(e.target.value)}

placeholder="New task"

/>

<button type="submit">Add</button>

</form>

<ul>

{tasks.map((task, index) => <li key={index}>{task}</li>)}

</ul>

</div>

);

}

Backend Basics with Node.js and Express

Overview:

Why It’s Important:

The backend is the brain of your app—it handles data storage, business logic, and communicates with the frontend.

Node.js lets you run JavaScript on the server, and Express is a lightweight framework to build REST APIs.

Real-World Example:

• The frontend React app makes requests to the backend API (Node.js + Express) to get or save data (like user accounts or to-do items).

1. What is Node.js?

Concepts:

• Node.js runs JavaScript outside the browser.

• Allows building servers, APIs, and backend logic.

Real-World Example:

Building a to-do API that saves tasks to a database.

2. What is Express?

Concepts:

• Express is a framework for building web servers in Node.js.

• Handles HTTP requests (GET, POST, etc.).

• Makes building REST APIs easier.

Real-World Example:

Create an endpoint /tasks that lets the frontend get or add tasks.

3. Setting Up Node.js & Express

Steps:

1. Initialize a Node.js project:

mkdir my-api

cd my-api

npm init -y

2. Install Express:

npm install express

3. Create index.js:

const express = require('express');

const app = express();

const PORT = 3000;

app.get('/', (req, res) => {

res.send('Hello, API!');

});

app.listen(PORT, () => {

console.log(`Server running on http://localhost:${PORT}`);

});

4. Run the server:

node index.js

• Visit http://localhost:3000 to see "Hello, API!".

4. REST API Basics (GET, POST, PUT, DELETE)

Why It’s Important:

REST APIs use HTTP methods to manage resources.

HTTP Method What it does Example

GET Read data Get all tasks

POST Create data Add a new task

PUT Update data Update a task

DELETE Delete data Remove a task

Example:

const tasks = []; // In-memory data

app.use(express.json()); // Parse JSON

// GET all tasks

app.get('/tasks', (req, res) => {

res.json(tasks);

});

// POST a new task

app.post('/tasks', (req, res) => {

const task = req.body.task;

tasks.push(task);

res.status(201).json({ message: 'Task added!', task });

});

5. Middleware

Why It’s Important:

Middleware are functions that run between the request and response—used for parsing data, logging, or authentication.

Concepts:

• express.json(): Parses incoming JSON data.

Hands-On Practice:

1. Setup Server

• Question: Create a simple Express server that responds "API is running" at /.

Answer:

const express = require('express');

const app = express();

const PORT = 3000;

app.get('/', (req, res) => {

res.send('API is running');

});

app.listen(PORT, () => {

console.log(`Server running at http://localhost:${PORT}`);

});

2. GET Endpoint

• Question: Create a GET endpoint /greet that responds with { message: "Hello!" }.

Answer:

app.get('/greet', (req, res) => {

res.json({ message: "Hello!" });

});

3. POST Endpoint

• Question: Create a POST endpoint /echo that takes { message } from the request body and responds with it.

Answer:

app.use(express.json()); // Middleware to parse JSON

app.post('/echo', (req, res) => {

const { message } = req.body;

res.json({ message });

});

4. Build a Simple In-Memory Task API

1. Add an array tasks to hold task strings.

2. Create:

• GET /tasks (returns all tasks).

• POST /tasks (adds a new task).

Answer:

const tasks = [];

app.use(express.json());

app.get('/tasks', (req, res) => {

res.json(tasks);

});

app.post('/tasks', (req, res) => {

const { task } = req.body;

tasks.push(task);

res.status(201).json({ message: 'Task added!', task });

});

Mini Project: Basic Task Manager API

1. Setup an Express server.

2. Add GET and POST endpoints for tasks.

3. Use Postman or curl to test the API.

MongoDB & Mongoose Basics

Overview:

Why It’s Important:

MongoDB is a NoSQL database—it stores data as flexible documents (JSON-like).

Mongoose is a library that makes working with MongoDB in Node.js easier by adding schemas and validation.

Real-World Example:

Store users, tasks, or products for your app in MongoDB.

The backend (Express) connects to MongoDB to save and retrieve data.

1. What is MongoDB?

Concepts:

• NoSQL database: Uses collections (like tables) and documents (like rows).

• Stores data in JSON-like format.

Example document:

{

"name": "Sofia",

"age": 7

}

2. What is Mongoose?

Concepts:

• Mongoose is an ODM (Object Data Modeling) library.

• Adds schemas, models, and validation to MongoDB.

3. Setting Up MongoDB & Mongoose

1. Install MongoDB locally or use MongoDB Atlas (cloud).

• MongoDB Atlas is easier for beginners:

• https://www.mongodb.com/cloud/atlas

2. Install Mongoose:

npm install mongoose

3. Connect to MongoDB:

const mongoose = require('mongoose');

mongoose.connect('mongodb://localhost:27017/mydb', {

useNewUrlParser: true,

useUnifiedTopology: true,

})

.then(() => console.log('MongoDB connected!'))

.catch((err) => console.error('MongoDB connection error:', err));

• For MongoDB Atlas, replace the connection string:

mongoose.connect('your-mongodb-atlas-connection-string', { ... });

4. Mongoose Schemas & Models

Why It’s Important:

Schemas define structure for your data.

Models let you create, read, update, delete (CRUD) documents.

Concepts:

• Schema: Blueprint for a document.

• Model: Interface for interacting with the collection.

Example: Create a Task Schema and Model

const mongoose = require('mongoose');

const taskSchema = new mongoose.Schema({

name: { type: String, required: true },

completed: { type: Boolean, default: false },

});

const Task = mongoose.model('Task', taskSchema);

5. CRUD Operations with Mongoose

Create:

const newTask = new Task({ name: 'Learn Mongoose' });

await newTask.save();

Read:

const tasks = await Task.find();

Update:

await Task.findByIdAndUpdate(taskId, { completed: true });

Delete:

await Task.findByIdAndDelete(taskId);

6. Integrate with Express

Example: Basic Task API with MongoDB

// Connect to MongoDB

mongoose.connect('mongodb://localhost:27017/mydb');

// Define Schema & Model

const taskSchema = new mongoose.Schema({

name: { type: String, required: true },

completed: { type: Boolean, default: false },

});

const Task = mongoose.model('Task', taskSchema);

// Express Endpoints

app.use(express.json());

// GET tasks

app.get('/tasks', async (req, res) => {

const tasks = await Task.find();

res.json(tasks);

});

// POST task

app.post('/tasks', async (req, res) => {

const { name } = req.body;

const task = new Task({ name });

await task.save();

res.status(201).json(task);

});

Hands-On Practice:

1. Connect to MongoDB

• Question: How do you connect Mongoose to mongodb://localhost:27017/mydb?

Answer:

mongoose.connect('mongodb://localhost:27017/mydb', {

useNewUrlParser: true,

useUnifiedTopology: true,

});

2. Create a Mongoose Schema & Model

• Question: Create a schema for a User with fields username (string) and email (string).

Answer:

const userSchema = new mongoose.Schema({

username: String,

email: String,

});

const User = mongoose.model('User', userSchema);

3. Add CRUD with Mongoose

• Question: Write code to add a new user to the database.

Answer:

const newUser = new User({ username: 'Luna', email: 'luna@example.com' });

await newUser.save();

Mini Project: Task Manager with MongoDB

1. Setup MongoDB and Mongoose.

2. Define a Task schema.

3. Create GET and POST routes to manage tasks.

NestJS Backend Framework

Overview:

Why It’s Important:

NestJS is a backend framework built on Node.js and Express, designed for scalable and maintainable server-side applications.

It uses TypeScript and decorators to organize code into modules, controllers, and services—helping you write clean, structured APIs.

Real-World Example:

Build a production-grade API (like for an e-commerce store or social media app) that connects to a database, handles authentication, and scales easily.

1. What is NestJS?

Concepts:

• Modular: Code is organized into modules (features).

• Dependency Injection: Services are injected where needed.

• Built-in support for Express and TypeScript.

2. Setting Up NestJS

Steps:

1. Install NestJS CLI:

npm install -g @nestjs/cli

2. Create a new project:

nest new my-nest-app

3. Run the server:

cd my-nest-app

npm run start

• Visit http://localhost:3000/ to see "Hello World!".

3. NestJS Architecture

Concepts:

Component Purpose

Module Groups related components (feature area).

Controller Handles HTTP requests and responses.

Service Handles business logic (e.g., database).

4. Create a Basic Task Module

Generate a Module, Controller, and Service:

nest generate module tasks

nest generate controller tasks

nest generate service tasks

This creates:

src/tasks/tasks.module.ts

src/tasks/tasks.controller.ts

src/tasks/tasks.service.ts

Controller Example:

import { Controller, Get, Post, Body } from '@nestjs/common';

import { TasksService } from './tasks.service';

@Controller('tasks')

export class TasksController {

constructor(private tasksService: TasksService) {}

@Get()

getTasks() {

return this.tasksService.getTasks();

}

@Post()

createTask(@Body() body: { name: string }) {

return this.tasksService.createTask(body.name);

}

}

Service Example:

import { Injectable } from '@nestjs/common';

@Injectable()

export class TasksService {

private tasks = [];

getTasks() {

return this.tasks;

}

createTask(name: string) {

const newTask = { id: Date.now(), name };

this.tasks.push(newTask);

return newTask;

}

}

5. Connecting NestJS to MongoDB (Mongoose)

1. Install Mongoose package:

npm install @nestjs/mongoose mongoose

2. Configure Mongoose in app.module.ts:

import { MongooseModule } from '@nestjs/mongoose';

@Module({

imports: [

MongooseModule.forRoot('mongodb://localhost:27017/mydb'),

],

})

export class AppModule {}

3. Define a Task Schema:

import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';

import { Document } from 'mongoose';

@Schema()

export class Task extends Document {

@Prop({ required: true })

name: string;

@Prop({ default: false })

completed: boolean;

}

export const TaskSchema = SchemaFactory.createForClass(Task);

4. Register the Model in tasks.module.ts:

import { MongooseModule } from '@nestjs/mongoose';

import { Task, TaskSchema } from './task.schema';

@Module({

imports: [

MongooseModule.forFeature([{ name: Task.name, schema: TaskSchema }]),

],

})

export class TasksModule {}

Hands-On Practice:

1. Create a Controller

• Question: Create a GET /hello route that returns "Hello from NestJS!".

Answer:

@Controller()

export class AppController {

@Get('hello')

getHello() {

return 'Hello from NestJS!';

}

}

2. Create a Service

• Question: Create a service with a getGreeting() function that returns "Hi there!", and call it in a controller.

Answer:

@Injectable()

export class AppService {

getGreeting() {

return 'Hi there!';

}

}

@Controller()

export class AppController {

constructor(private appService: AppService) {}

@Get('greet')

greet() {

return this.appService.getGreeting();

}

}

3. Build a Simple Task API

1. Create tasks module, controller, and service.

2. Add:

• GET /tasks to list tasks.

• POST /tasks to add a task.

Mini Project: Task Manager API with NestJS

1. Setup NestJS project.

2. Connect to MongoDB with Mongoose.

3. Define a Task schema with name and completed.

4. Implement GET and POST routes.

Full Stack Integration (React + NestJS)

Overview:

Why It’s Important:

Connecting your React frontend to your NestJS backend lets the frontend display real data and send data back to the server. This is how full-stack applications work together.

Real-World Example:

Your React to-do app fetches tasks from the NestJS API and displays them. When you add a task in React, it sends a POST request to the backend to save it in MongoDB.

1. How Frontend and Backend Communicate

Concepts:

• Frontend (React) makes HTTP requests (GET, POST, etc.).

• Backend (NestJS) listens to those requests and sends responses.

• Use fetch or Axios in React to make requests.

2. Enable CORS in NestJS

Why It’s Important:

CORS (Cross-Origin Resource Sharing) lets your frontend talk to your backend when they run on different ports (React on 3000, NestJS on 3001).

Example (main.ts in NestJS):

import { NestFactory } from '@nestjs/core';

import { AppModule } from './app.module';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.enableCors(); // Enable CORS

await app.listen(3001);

}

bootstrap();

3. Backend API (NestJS)

• GET /tasks: Returns all tasks.

• POST /tasks: Adds a new task.

NestJS Controller Example:

@Controller('tasks')

export class TasksController {

constructor(private tasksService: TasksService) {}

@Get()

getTasks() {

return this.tasksService.getTasks();

}

@Post()

createTask(@Body() body: { name: string }) {

return this.tasksService.createTask(body.name);

}

}

4. Fetching Data in React

Concepts:

• Use fetch or Axios to request data from the backend.

• Use useEffect to fetch data when the component mounts.

Example:

import { useState, useEffect } from 'react';

function TaskList() {

const [tasks, setTasks] = useState([]);

useEffect(() => {

fetch('http://localhost:3001/tasks')

.then(res => res.json())

.then(data => setTasks(data));

}, []);

return (

<ul>

{tasks.map(task => <li key={task.\_id}>{task.name}</li>)}

</ul>

);

}

5. Sending Data from React (POST Request)

Concepts:

• Use fetch with method: 'POST' to send data.

• Include headers and body.

Example:

function AddTask({ onTaskAdded }) {

const [task, setTask] = useState('');

const handleSubmit = async (e) => {

e.preventDefault();

await fetch('http://localhost:3001/tasks', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ name: task }),

});

setTask('');

onTaskAdded(); // Refresh the task list

};

return (

<form onSubmit={handleSubmit}>

<input value={task} onChange={(e) => setTask(e.target.value)} placeholder="New task" />

<button type="submit">Add</button>

</form>

);

}

6. Putting It All Together (React + NestJS)

1. NestJS backend runs on http://localhost:3001.

2. React frontend runs on http://localhost:3000.

3. Enable CORS in NestJS to allow communication.

4. Use fetch or Axios in React to interact with NestJS.

Hands-On Practice:

1. Enable CORS

• Question: How do you enable CORS in a NestJS project?

Answer:

const app = await NestFactory.create(AppModule);

app.enableCors();

2. Fetch Tasks in React

• Question: How do you fetch tasks from http://localhost:3001/tasks when the component mounts?

Answer:

useEffect(() => {

fetch('http://localhost:3001/tasks')

.then(res => res.json())

.then(data => setTasks(data));

}, []);

3. Send Task (POST Request)

• Question: How do you send a new task { name: "New Task" } to the backend?

Answer:

fetch('http://localhost:3001/tasks', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ name: 'New Task' }),

});

Mini Project: Full Stack To-Do App

1. Backend (NestJS):

• Create a Task model (Mongoose).

• Add GET and POST routes.

2. Frontend (React):

• Display the list of tasks.

• Add a form to create tasks.

• Fetch tasks on mount and refresh the list after adding.

Authentication with JWT (JSON Web Tokens)

Overview:

Why It’s Important:

Authentication ensures that only logged-in users can access certain parts of your app (like personal dashboards or admin features).

JWT (JSON Web Tokens) are used to securely verify a user’s identity between the frontend and backend.

Real-World Example:

A user logs in through React, gets a JWT token from NestJS, and uses that token for protected routes (like viewing their profile or tasks).

1. What is JWT?

Concepts:

• JWT is a secure token that the server gives the client after login.

• The client (React) stores the token and sends it with requests to prove identity.

• The server (NestJS) verifies the token before responding.

2. Install JWT Packages in NestJS

npm install @nestjs/jwt passport-jwt @nestjs/passport passport

3. Setup JWT Auth Module in NestJS

1. Create an auth module, controller, and service:

nest generate module auth

nest generate service auth

nest generate controller auth

2. Configure JWT Strategy (auth.module.ts):

import { JwtModule } from '@nestjs/jwt';

@Module({

imports: [

JwtModule.register({

secret: 'your-secret-key', // Use env variable in real apps

signOptions: { expiresIn: '1h' },

}),

],

})

export class AuthModule {}

3. Create a Login Endpoint (auth.controller.ts):

@Controller('auth')

export class AuthController {

constructor(private jwtService: JwtService) {}

@Post('login')

login(@Body() body: { username: string, password: string }) {

// Fake user validation (replace with real database check)

if (body.username === 'admin' && body.password === 'password') {

const payload = { username: body.username };

const token = this.jwtService.sign(payload);

return { token };

}

throw new UnauthorizedException();

}

}

4. Protect Routes with JWT in NestJS

Concepts:

• Use Guards to protect routes.

• Validate JWT token in requests.

Example:

import { JwtAuthGuard } from './jwt-auth.guard';

@Controller('tasks')

export class TasksController {

@UseGuards(JwtAuthGuard)

@Get()

getTasks() {

return this.tasksService.getTasks();

}

}

• JWT Guard validates the token before allowing access.

5. Store & Use JWT Token in React

Concepts:

• Store the token in localStorage.

• Send the token in the Authorization header with requests.

Example:

// Login function in React

async function login(username, password) {

const res = await fetch('http://localhost:3001/auth/login', {

method: 'POST',

headers: { 'Content-Type': 'application/json' },

body: JSON.stringify({ username, password }),

});

const data = await res.json();

localStorage.setItem('token', data.token); // Store token

}

// Fetch protected tasks

async function fetchTasks() {

const token = localStorage.getItem('token');

const res = await fetch('http://localhost:3001/tasks', {

headers: { Authorization: `Bearer ${token}` },

});

const data = await res.json();

console.log(data);

}

Hands-On Practice:

1. Login with JWT

• Question: How do you generate a JWT token in NestJS after a successful login?

Answer:

const payload = { username: user.username };

const token = this.jwtService.sign(payload);

return { token };

2. Store JWT in React

• Question: Where do you store the JWT token after login in React?

Answer:

localStorage.setItem('token', token);

3. Send JWT with Requests

• Question: How do you include the token in the Authorization header?

Answer:

fetch('http://localhost:3001/tasks', {

headers: { Authorization: `Bearer ${token}` },

});

Mini Project: Add Login to Full Stack To-Do App

1. Backend (NestJS):

• Create an auth module.

• Implement a login route that returns a JWT token.

• Protect /tasks route with JWT.

2. Frontend (React):

• Add a login form.

• Store JWT token in localStorage.

• Send the token when fetching tasks.

Deployment (Vercel + Heroku)

Overview:

Why It’s Important:

Deployment lets you host your app online so anyone can use it.

• Vercel is perfect for deploying React frontends.

• Heroku (or Render) is great for deploying NestJS backends.

Real-World Example:

Deploy your full-stack to-do app so users can access it from anywhere.

1. Deploy React App with Vercel

Steps:

1. Push your React app to GitHub.

2. Sign up for Vercel:

https://vercel.com

3. Connect your GitHub repo and click Deploy.

4. Configure environment variables (if needed):

• For example: REACT\_APP\_API\_URL=https://your-backend-url.com

2. Deploy NestJS Backend with Heroku

Steps:

1. Push your NestJS app to GitHub.

2. Create a Procfile in the root of your backend:

web: npm run start:prod

3. Setup production build:

npm install --save @nestjs/serve-static

npm run build

4. Sign up for Heroku:

https://heroku.com

5. Install Heroku CLI:

npm install -g heroku

6. Create a Heroku app:

heroku login

heroku create your-app-name

git push heroku main

7. Configure MongoDB Atlas on Heroku:

• Add environment variable:

MONGODB\_URI=your-mongo-connection-string

3. Update React App to Use Production Backend

Concepts:

• Use environment variables for API URLs in React.

Example (.env in React):

REACT\_APP\_API\_URL=https://your-heroku-app.herokuapp.com

Fetch Example:

const API\_URL = process.env.REACT\_APP\_API\_URL;

fetch(`${API\_URL}/tasks`);

4. Test the Live App

• Open your React frontend on Vercel.

• Try logging in and adding tasks (talking to the NestJS backend on Heroku).

Hands-On Practice:

1. Vercel Deployment

• Question: What file do you configure in React to store the backend URL?

Answer:

.env

2. Heroku Deployment

• Question: What command pushes your NestJS app to Heroku?

Answer:

git push heroku main

3. Configure MongoDB on Heroku

• Question: How do you set the MongoDB URI on Heroku?

Answer:

heroku config:set MONGODB\_URI=your-mongo-connection-string

Mini Project: Deploy Full Stack To-Do App

1. Backend (NestJS):

• Deploy to Heroku.

• Set MongoDB Atlas connection.

2. Frontend (React):

• Deploy to Vercel.

• Use production API URL from Heroku.

3. Test the full flow:

• Visit your React app.

• Login, fetch, and add tasks.