Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

ADVANCED JAVASCRIPT & FRONTEND FRAMEWORKS (WEEKS 7-12)

ARROW FUNCTIONS, SPREAD & REST OPERATORS

CHAPTER 1: INTRODUCTION TO ARROW FUNCTIONS, SPREAD, AND REST OPERATORS

1.1 What Are Arrow Functions, Spread & Rest Operators?

In JavaScript, **arrow functions, spread operator (...), and rest operator (...)** provide cleaner syntax, improve code readability, and make working with functions and arrays easier.

- **Why Are These Concepts Important?**
  - **Arrow functions** provide a shorter syntax for writing functions.
  - **Spread operator** expands arrays and objects into individual elements.
  - **Rest operator** collects multiple arguments into an array.
- **Example: Using All Three Concepts Together**

```
const add = (a, b) => a + b; // Arrow Function
```

```
const numbers = [1, 2, 3];

const newNumbers = [...numbers, 4, 5]; // Spread Operator

const sumAll = (...nums) => nums.reduce((sum, num) => sum + num, 0); // Rest Operator


console.log(add(5, 3)); // Output: 8

console.log(newNumbers); // Output: [1, 2, 3, 4, 5]

console.log(sumAll(1, 2, 3, 4, 5)); // Output: 15
```

This **simplifies function writing, handles multiple parameters,** and **expands data easily**.

CHAPTER 2: ARROW FUNCTIONS IN JAVASCRIPT

2.1 What Are Arrow Functions?

Arrow functions (=>) are a shorter way to define functions in JavaScript, introduced in **ES6**.

- **Example: Regular Function vs. Arrow Function**

```
// Traditional Function

function greet(name) {

  return "Hello, " + name;

}


// Arrow Function

const greetArrow = name => `Hello, ${name}`;
```

```
console.log(greet("Alice")); // Output: Hello, Alice

console.log(greetArrow("Bob")); // Output: Hello, Bob
```

2.2 Implicit Return in Arrow Functions

Arrow functions automatically return values **without needing return**.

- **Example: Implicit Return**

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

console.log(multiply(4, 5)); // Output: 20
```

2.3 Arrow Functions & this Keyword

Arrow functions **do not bind their own this**, making them useful for callbacks and object methods.

- **Example: Arrow Function in an Object**

```
const person = {
  name: "Alice",
  sayName: function() {
    setTimeout(() => {
      console.log(`My name is ${this.name}`);
    }, 1000);
  }
};

person.sayName(); // Output: My name is Alice
```

In regular functions, this would refer to setTimeout, but **arrow functions inherit this from their enclosing scope**.

CHAPTER 3: SPREAD OPERATOR (...) IN JAVASCRIPT

3.1 What is the Spread Operator?

The spread operator (...) **expands arrays, objects, and function arguments** into individual elements.

- **Use Cases of Spread Operator:**

Use Case	Example
Expanding arrays	[...arr1, ...arr2]
Cloning arrays	const copy = [...arr]
Merging objects	{...obj1, ...obj2}
Passing array elements as arguments	Math.max(...nums)

3.2 Using Spread Operator with Arrays

- **Example: Merging Two Arrays**

```
const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

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


console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]
```

3.3 Using Spread Operator with Objects

- **Example: Cloning & Merging Objects**

```
const user = { name: "Alice", age: 25 };
```

```
const updatedUser = { ...user, city: "New York" };


console.log(updatedUser);
```

// Output: { name: "Alice", age: 25, city: "New York" }

This **adds new properties without modifying the original object**.

CHAPTER 4: REST OPERATOR (...) IN JAVASCRIPT

4.1 What is the Rest Operator?

The rest operator (...) **collects multiple arguments into a single array**. It is **used in function parameters** to handle an unknown number of arguments.

- **Example: Function with Rest Operator**

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


console.log(sumAll(1, 2, 3, 4, 5)); // Output: 15
```

Here, numbers **collects all function arguments into an array**.

4.2 Rest Operator with Destructuring

Rest can **extract remaining elements** from an array or object.

- **Example: Array Destructuring with Rest**

```
const [first, second, ...rest] = [10, 20, 30, 40, 50];


console.log(first);  // Output: 10
```

```
console.log(second); // Output: 20

console.log(rest);   // Output: [30, 40, 50]
```

- **Example: Object Destructuring with Rest**

```
const user = { name: "Alice", age: 25, city: "New York", country: "USA" };

const { name, ...details } = user;


console.log(name); // Output: Alice

console.log(details); // Output: { age: 25, city: "New York", country: "USA" }
```

This **separates name while collecting the rest into details**.

Case Study: How JavaScript Uses Arrow Functions, Spread & Rest Operators in Modern Applications

Challenges Faced

- Writing **shorter, more readable functions**.
- Handling **dynamic arguments in API calls**.
- Merging **complex data structures**.

Solutions Implemented

- Used **arrow functions** to improve readability in callbacks.
- Applied **spread operator** to efficiently manage **arrays and objects**.
- Implemented **rest parameters** to collect function arguments dynamically.

🔹 **Example: Fetching API Data Using Arrow Functions & Spread Operator**

```
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json())
  .then(users => {
    const userNames = users.map(user => user.name);
    console.log([...userNames]);
  });
```

This **fetches user data, extracts names, and logs them using the spread operator**.

📝 **Exercise**

✅ Convert a **regular function into an arrow function**.
✅ Merge two **arrays using the spread operator**.
✅ Use the **rest operator** in a function to handle multiple arguments dynamically.
✅ Destructure an **object using the rest operator** to extract specific properties.

Conclusion

- **Arrow functions simplify function writing** and maintain this binding.
- **Spread operator expands arrays and objects,** making them more manageable.

- **Rest operator collects multiple arguments** into an array, making functions more flexible.
- **Combining these features improves readability and efficiency** in JavaScript applications.

PROMISES, ASYNC/AWAIT

CHAPTER 1: INTRODUCTION TO ASYNCHRONOUS JAVASCRIPT

1.1 What is Asynchronous JavaScript?

Asynchronous JavaScript allows functions to execute **without blocking the main thread**. This is crucial for tasks like **fetching data from APIs, reading files, and handling user input**, ensuring a smooth user experience.

🔹 **Why is Asynchronous Programming Important?**

✓ Prevents the browser from **freezing during long-running tasks**.

✓ Enables **real-time updates** (e.g., fetching new posts without page reloads).

✓ Helps handle **delayed tasks like API calls and database queries**.

🔹 **Example: Synchronous vs. Asynchronous Execution**

📌 **Synchronous Code (Blocks Execution)**

```
console.log("Start");

alert("This alert blocks the execution");

console.log("End");  // Executes only after alert is closed
```

📌 **Asynchronous Code (Non-Blocking Execution)**

```
console.log("Start");


setTimeout(() => {
   console.log("Async task completed");
}, 2000);  // Runs after 2 seconds
```

```
console.log("End");
```

Output:

Start

End

Async task completed (after 2 seconds)

The setTimeout() function executes **after the delay, allowing the next statements to run** immediately.

CHAPTER 2: UNDERSTANDING JAVASCRIPT PROMISES

2.1 What is a Promise?

A **Promise** is a JavaScript object that **represents a task that may complete now, later, or fail**. It helps manage asynchronous operations by handling **success (resolved)** or **failure (rejected)** outcomes.

- **Promise States:**

State	Description
Pending	Initial state (waiting for completion).
Resolved (Fulfilled)	Operation successful.
Rejected	Operation failed.

- **Example: Creating a Basic Promise**

```
let myPromise = new Promise((resolve, reject) => {
  let success = true;  // Change this to false to test rejection
  if (success) {
```

```
    resolve("Promise fulfilled!");
  } else {
    reject("Promise rejected!");
  }
});

console.log(myPromise);
```

2.2 Handling Promises Using .then() and .catch()

Promises are handled using **.then()** (for success) and **.catch()** (for errors).

- **Example: Handling a Promise**

```
myPromise
  .then(response => console.log(response))  // If resolved
  .catch(error => console.log(error));  // If rejected
```

2.3 Chaining Promises for Sequential Execution

Promises allow **chaining** multiple async operations in sequence.

- **Example: Fetching User Data and Processing It**

```
fetch("https://jsonplaceholder.typicode.com/users/1")
  .then(response => response.json())  // Convert response to JSON
  .then(data => console.log("User Name:", data.name))  // Process data
```

```
.catch(error => console.log("Error:", error));  // Handle errors
```

If the fetch is successful, it prints the user's name; otherwise, it logs an error.

CHAPTER 3: ASYNC/AWAIT – A SIMPLER WAY TO HANDLE PROMISES

3.1 What is Async/Await?

Async/Await is a **modern approach** to writing asynchronous code in JavaScript. It allows using **asynchronous functions in a synchronous style**, making code **more readable and easier to debug**.

- **Why Use Async/Await Over Promises?**

✓ Makes code **look synchronous** and easier to follow.
✓ Avoids **callback hell** (nested .then() chaining).
✓ Simplifies **error handling** using try...catch.

3.2 Using Async/Await in JavaScript

An async function **automatically returns a promise**, and the await keyword **pauses execution until the promise resolves**.

- **Example: Fetching Data with Async/Await**

```
async function fetchUser() {
  try {
    let response = await fetch("https://jsonplaceholder.typicode.com/users/1");
    let user = await response.json();
    console.log("User Name:", user.name);
```

```
  } catch (error) {
    console.log("Error:", error);
  }
}
```

fetchUser();

✓ await ensures **data is fully retrieved** before executing the next line.

✓ try...catch **handles errors gracefully**.

3.3 Comparing Promises vs. Async/Await

Feature	Promises .then()	Async/Await
Readability	Harder with chaining	Easier to follow
Error Handling	Uses .catch()	Uses try...catch
Code Complexity	Can become nested	Looks cleaner

🔹 **Example: Same API Call Using Both Methods**

📌 **Using Promises**

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log("Error:", error));
```

📌 **Using Async/Await**

```
async function getPost() {
```

```
  try {
    let response = await
fetch("https://jsonplaceholder.typicode.com/posts/1");
    let data = await response.json();
    console.log(data);
  } catch (error) {
    console.log("Error:", error);
  }
}

getPost();
```

The **Async/Await version is easier to read and manage**.

CHAPTER 4: REAL-WORLD USE CASES OF PROMISES & ASYNC/AWAIT

4.1 Handling Multiple API Calls Simultaneously

The Promise.all() method allows executing **multiple async tasks in parallel**.

- **Example: Fetching Multiple APIs at the Same Time**

```
async function fetchData() {
  try {
    let [user, posts] = await Promise.all([
      fetch("https://jsonplaceholder.typicode.com/users/1").then(res
=> res.json()),
```

```
        fetch("https://jsonplaceholder.typicode.com/posts").then(res => res.json())
    ]);

    console.log("User:", user.name);
    console.log("Posts:", posts.slice(0, 3));  // Display first 3 posts
  } catch (error) {
    console.log("Error:", error);
  }
}

fetchData();
```

✓ Fetches **user details and posts at the same time**, reducing wait time.

4.2 Handling Delayed Operations (Simulating API Calls)

- **Example: Using setTimeout() with Promises**

```
function delayedMessage() {
  return new Promise(resolve => {
    setTimeout(() => resolve("Data Loaded!"), 3000);
  });
}
```

```
async function showMessage() {
  console.log("Loading...");
  let message = await delayedMessage();
  console.log(message);
}

showMessage();
```

✓ Simulates **loading data from a slow API**.

Case Study: How Instagram Uses Promises & Async/Await for Live Updates

Challenges Faced by Instagram

✓ Handling **real-time updates** for notifications and messages.
✓ Fetching **multiple API requests at once** (posts, comments, likes).

Solutions Implemented

✓ Used **Promises to load multiple API calls efficiently**.
✓ Applied **Async/Await for clean and structured code**.
✓ Implemented **real-time updates using WebSockets** for messages.

- **Key Takeaways from Instagram's Approach:**

✓ **Efficient async handling improves page performance.**
✓ **Using Promises reduces wait times for multiple API calls.**

📝 **Exercise**

✅ Write an **async function** that fetches user data and logs it.

✅ Use Promise.all() to **fetch multiple API endpoints at the same time**.

✅ Simulate a **loading delay** using setTimeout() and Promises.

Conclusion

✓ Promises help manage async operations using .then() and .catch().

✓ Async/Await makes code more readable and easier to debug.

✓ Using Promise.all() improves performance by fetching multiple APIs in parallel.

✓ Applying Promises & Async/Await correctly enhances real-world applications.

WORKING WITH APIS (FETCH & AXIOS)

CHAPTER 1: INTRODUCTION TO APIS

1.1 What is an API?

An **API (Application Programming Interface)** allows different software applications to communicate with each other. APIs enable developers to **fetch, send, and manipulate data from external sources**, such as databases, third-party services, and cloud applications.

- **Why Use APIs?**

✓ Retrieve **real-time data** (weather, stock prices, news updates).
✓ Send **data to a server** (user authentication, form submissions).
✓ Enable **third-party integrations** (Google Maps, PayPal, Twitter).

- **Example: API in Action**
  - A **weather app** fetching live weather data.
  - An **e-commerce store** retrieving product listings from a database.
  - A **social media feed** updating in real-time.

CHAPTER 2: FETCH API FOR MAKING HTTP REQUESTS

2.1 What is the Fetch API?

The **Fetch API** is a built-in JavaScript function used to make **HTTP requests** (GET, POST, PUT, DELETE) to retrieve and send data between a web application and a server.

🔹 Why Use Fetch API?

✓ **Simple syntax** for fetching data.

✓ Works **natively in browsers** (no extra libraries needed).

✓ Supports **asynchronous operations** using promises.

2.2 Making a GET Request with Fetch API

📌 Example: Fetching Data from an API

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => response.json())  // Convert response to JSON
  .then(data => console.log(data))   // Display data
  .catch(error => console.error("Error:", error));  // Handle errors
```

Output:

```
{
  "userId": 1,
  "id": 1,
  "title": "Sample Title",
  "body": "This is a sample API response."
}
```

🔹 How it Works:

✓ fetch() makes a **request** to the API.

✓ .then(response.json()) converts the response **to JSON format**.

✓ .then(data => console.log(data)) logs the **received data**.

✓ .catch(error => console.error(error)) **handles errors**.

2.3 Making a POST Request with Fetch API

A **POST request** is used to send data to an API.

📌 Example: Sending Data to an API

```
fetch("https://jsonplaceholder.typicode.com/posts", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({
    title: "New Post",
    body: "This is a new post",
    userId: 1
  })
})
.then(response => response.json())
.then(data => console.log("Post created:", data))
.catch(error => console.error("Error:", error));
```

Output:

```
{
  "title": "New Post",
  "body": "This is a new post",
  "userId": 1,
  "id": 101
}
```

🔹 **Key Elements:**

✓ method: "POST" ⟶ Specifies **request type**.

✓ headers: {"Content-Type": "application/json"} ⟶ Defines **data format**.

✓ body: JSON.stringify({...}) ⟶ Converts **JavaScript object to JSON** format before sending.

2.4 Handling Errors in Fetch API

Sometimes, an API request **fails** due to network issues, incorrect URLs, or missing parameters.

📌 **Example: Error Handling in Fetch API**

```
fetch("https://wrongurl.com")
  .then(response => {
    if (!response.ok) throw new Error("Network response was not ok");
    return response.json();
  })
  .catch(error => console.error("Fetch Error:", error));
```

✓ response.ok checks **if the request was successful**.

✓ throw new Error() **displays an error message**.

CHAPTER 3: USING AXIOS FOR API REQUESTS

3.1 What is Axios?

Axios is a **popular JavaScript library** for making HTTP requests. It is a **wrapper around Fetch** that simplifies API calls with built-in error handling.

🔹 **Why Use Axios Instead of Fetch?**

✓ **Easier syntax** (less code required).

✓ **Automatic JSON conversion** (no need for .json()).

✓ **Better error handling** with built-in .catch().

📌 **Installing Axios in a Project:**

npm install axios

Or include it in HTML:

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

3.2 Making a GET Request with Axios

📌 **Example: Fetching Data Using Axios**

```
axios.get("https://jsonplaceholder.typicode.com/posts/1")
  .then(response => console.log(response.data))
  .catch(error => console.error("Error:", error));
```

🔹 **Why Axios is Easier Than Fetch?**

✓ No need for response.json(), Axios **automatically converts JSON**.

✓ Cleaner syntax **reduces lines of code**.

3.3 Making a POST Request with Axios

📌 **Example: Sending Data Using Axios**

```
axios.post("https://jsonplaceholder.typicode.com/posts", {
  title: "New Post",
  body: "This is a new post",
  userId: 1
})
.then(response => console.log("Post created:", response.data))
.catch(error => console.error("Error:", error));
```

✓ **Shorter syntax** compared to Fetch.
✓ **Handles JSON conversion automatically**.

3.4 Axios vs. Fetch: Key Differences

Feature	Fetch API	Axios
Syntax Complexity	Requires .json() for response conversion	Auto-converts JSON
Error Handling	Needs manual error handling (response.ok)	Built-in .catch()
Browser Support	Native browser API	Requires external library
Automatic Timeout Handling	No	Yes
Supports Request Interception	No	Yes

CHAPTER 4: DISPLAYING API DATA ON A WEB PAGE

4.1 Injecting API Data into HTML

📌 Example: Displaying a List of Users

```
<ul id="userList"></ul>


<script>
fetch("https://jsonplaceholder.typicode.com/users")
  .then(response => response.json())
  .then(data => {
    let list = document.getElementById("userList");
    data.forEach(user => {
      let li = document.createElement("li");
      li.textContent = user.name;
      list.appendChild(li);
    });
  })
  .catch(error => console.error("Error:", error));
</script>
```

✓ Dynamically creates a list of users from an API response.
✓ Improves user experience with real-time content updates.

Case Study: How Twitter Uses APIs for Real-Time Updates

Challenges Faced by Twitter

- Delivering **real-time tweet updates** without refreshing the page.
- Handling **millions of API requests per second**.

Solutions Implemented

- Used **Axios** for making API requests efficiently.
- Implemented **WebSockets for real-time tweet updates**.
- Optimized API responses using **caching and pagination**.

🔹 **Key Takeaways from Twitter's API Strategy:**

✓ Using APIs enables real-time dynamic content.

✓ Axios simplifies API requests and handles errors better.

✓ Optimizing API calls improves performance and user experience.

📝 **Exercise**

✅ Use **Fetch API** to get and display a list of posts from an API.

✅ Make a **POST request with Axios** to send form data to an API.

✅ Create an error-handling function for **failed API calls**.

✅ Compare **Fetch vs. Axios performance** by making the same API call using both.

Conclusion

✓ APIs enable web applications to fetch and send data dynamically.

✓ Fetch API is built into JavaScript but requires extra handling.

✓ Axios simplifies API requests with automatic JSON parsing and better error handling.

✓ Integrating APIs into websites improves interactivity and real-time updates.

COMPONENTS, PROPS, AND STATE MANAGEMENT

CHAPTER 1: INTRODUCTION TO COMPONENTS, PROPS, AND STATE MANAGEMENT

1.1 What Are Components, Props, and State in React?

In **React.js**, components, props, and state management form the foundation for building **dynamic and reusable UI elements**.

- **Why Are These Concepts Important?**
  - **Components** enable modular development, making the UI reusable.
  - **Props** pass data between components, ensuring communication.
  - **State management** allows dynamic updates without reloading the page.
- **Example of a Basic React Component Using Props and State:**

```
import React, { useState } from "react";

function Greeting({ name }) {
  const [message, setMessage] = useState("Hello");

  return <h1>{message}, {name}!</h1>;
}
```

export default Greeting;

This **renders a greeting message** dynamically using **props (name)** and **state (message)**.

CHAPTER 2: UNDERSTANDING REACT COMPONENTS

2.1 What Are Components?

A **component** in React is a **self-contained piece of UI** that can be **reused and composed** into larger applications.

- **Types of Components in React:**

Type	Description	Example
Functional Components	Stateless and use hooks	function MyComponent() {}
Class Components	Use this.state and lifecycle methods	class MyComponent extends React.Component {}

2.2 Creating Functional Components

- **Example: Basic Functional Component**

```
function Welcome() {
  return <h1>Welcome to React!</h1>;
}

export default Welcome;
```

- **Uses JSX** (<h1>) to render HTML inside JavaScript.
- **Exports the component** for use in other files.

2.3 Creating Class Components

- **Example: Basic Class Component**

```
import React, { Component } from "react";


class Welcome extends Component {
  render() {
    return <h1>Welcome to React!</h1>;
  }
}


export default Welcome;
```

- **Uses render() method** to return JSX.
- **Handles state** using this.state (covered in Chapter 4).

CHAPTER 3: UNDERSTANDING PROPS (PROPERTIES) IN REACT

3.1 What Are Props?

Props (short for **properties**) allow **data to be passed from a parent component to a child component**.

- **Why Use Props?**
  - **Make components dynamic** by passing different values.
  - **Encourage reusability**, reducing redundant code.
  - **Allow parent-child communication** in React apps.

3.2 Passing Props to a Functional Component

- **Example: Passing name as a Prop**

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}

export default Greeting;
```

- **Rendering in App.js:**

```
<Greeting name="Alice" />
<Greeting name="Bob" />
```

Each component **displays a different name,** making it reusable.

3.3 Default Props & PropTypes

Props can have **default values** and **type validation**.

- **Example: Using Default Props and PropTypes**

```
import PropTypes from "prop-types";

function Button({ label }) {
  return <button>{label}</button>;
}

Button.defaultProps = {
  label: "Click Me",
};
```

```
Button.propTypes = {
  label: PropTypes.string,
};

export default Button;
```

- **defaultProps** ensures a fallback value.
- **propTypes** validates that label must be a string.

CHAPTER 4: STATE MANAGEMENT IN REACT

4.1 What is State?

State is a **built-in React object** that stores component-specific data and triggers re-renders when updated.

- **Props vs. State:**

Feature	Props	State
Mutability	Immutable (cannot be changed)	Mutable (can be updated)
Use Case	Passed from parent to child	Managed within the component
Update Method	Cannot be modified inside component	Modified using useState or setState

4.2 Using useState() in Functional Components

- **Example: State Management with useState()**

```
import React, { useState } from "react";
```

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={() => setCount(count + 1)}>Increase</button>
    </div>
  );
}

export default Counter;
```

This **updates the counter value dynamically** using setCount().

4.3 Using State in Class Components

- **Example: State Management in a Class Component**

```
import React, { Component } from "react";

class Counter extends Component {
  constructor() {
    super();
    this.state = { count: 0 };
```

```
  }

  increaseCount = () => {
    this.setState({ count: this.state.count + 1 });
  };

  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={this.increaseCount}>Increase</button>
      </div>
    );
  }
}

export default Counter;
```

- **Uses this.state to store data** and setState() to update it.

CHAPTER 5: ADVANCED STATE MANAGEMENT TECHNIQUES

5.1 Managing State Across Multiple Components

For complex applications**, global state management** is required. Options include:

- **React Context API**
- **Redux**

5.2 Using React Context API for State Management

- **Example: Creating a Context Provider**

```
import React, { createContext, useState } from "react";

export const ThemeContext = createContext();

function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

export default ThemeProvider;
```

- **Consuming Context in a Component**

```
import { useContext } from "react";
import { ThemeContext } from "./ThemeProvider";
```

```
function DisplayTheme() {
  const { theme } = useContext(ThemeContext);
  return <h1>Current Theme: {theme}</h1>;
}
```

Case Study: How Netflix Uses React Components & State Management

Challenges Faced

- Managing **large-scale component hierarchies**.
- Efficiently handling **state updates** for seamless UI experience.

Solutions Implemented

- **Used functional components & props** for UI modularity.
- **Implemented global state management** to sync movie data dynamically.

🔹 **Key Takeaways from Netflix's UI Strategy:**

- **Reusable components make large applications scalable.**
- **Efficient state management ensures seamless UI updates.**

📝 **Exercise**

✅ Create a **functional component with props** for a user profile.
✅ Implement a **counter component** using the useState() hook.
✅ Use **React Context API** to create a theme switcher.

Conclusion

- **Components allow code reuse** and modular UI development.
- **Props enable dynamic content** and parent-child communication.
- **State management is crucial** for handling UI changes efficiently.
- **Advanced state solutions like React Context & Redux** improve app scalability.

REACT HOOKS (USESTATE, USEEFFECT)

CHAPTER 1: INTRODUCTION TO REACT HOOKS

1.1 What Are React Hooks?

React Hooks are built-in functions that allow functional components to **use state and lifecycle methods** without writing class components.

- **Why Use Hooks?**

✓ Makes **functional components more powerful**.
✓ Reduces **boilerplate code** compared to class components.
✓ Improves **code readability and maintainability**.

- **Commonly Used Hooks:**

Hook	Purpose	Example Usage
useState	Manages component state	Counter, form inputs
useEffect	Handles side effects	API calls, event listeners
useContext	Shares state globally	Theme switching, authentication
useRef	References DOM elements	Handling focus, animations

CHAPTER 2: MANAGING STATE WITH USESTATE

2.1 What is useState?

The useState hook allows **state management** in functional components.

📌 Syntax:

const [state, setState] = useState(initialValue);

✓ state → Stores the value.
✓ setState → Updates the state.

2.2 Example: Counter Using useState

```
import React, { useState } from "react";

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

  return (
    <div>
      <h2>Count: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}

export default Counter;
```

🔹 **How It Works:**

✓ useState(0) initializes count with 0.

✓ Clicking **increments the count dynamically**.

2.3 Updating Objects & Arrays in useState

📌 **Example: Updating an Object in State**

```
const [user, setUser] = useState({ name: "Alice", age: 25 });


const updateAge = () => {
  setUser({ ...user, age: user.age + 1 });
};
```

✓ setUser({ ...user, age: user.age + 1 }) **updates only the age property**, keeping other properties unchanged.

📌 **Example: Adding Items to an Array**

```
const [tasks, setTasks] = useState(["Task 1", "Task 2"]);


const addTask = () => {
  setTasks([...tasks, "New Task"]);
};
```

✓ The **spread operator (...tasks)** prevents overwriting the existing array.

CHAPTER 3: HANDLING SIDE EFFECTS WITH USEEFFECT

3.1 What is useEffect?

The useEffect hook **performs side effects** in functional components, such as:

✓ Fetching API data.

✓ Listening for events.

✓ Updating the DOM.

📌 **Syntax:**

```
useEffect(() => {
  // Side effect code here
}, [dependencies]);
```

✓ The **dependency array** controls when useEffect runs.

3.2 Example: Fetching Data with useEffect

```
import React, { useState, useEffect } from "react";

function UsersList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(response => response.json())
      .then(data => setUsers(data));
  }, []); // Runs only once when the component mounts
```

```
  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}

export default UsersList;
```

- **How It Works:**

✓ fetch() retrieves **user data from an API**.
✓ setUsers(data) **updates state dynamically**.
✓ [] ensures the effect **runs only once** on mount.

3.3 Running useEffect When State Changes

📌 Example: Updating the Document Title

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

useEffect(() => {
  document.title = `Count: ${count}`;
}, [count]); // Runs whenever `count` changes
```

✓ When count updates, the **page title changes dynamically**.

3.4 Cleaning Up Effects

📌 Example: Removing an Event Listener

```
useEffect(() => {
  const handleResize = () => console.log("Resized!");

  window.addEventListener("resize", handleResize);

  return () => window.removeEventListener("resize", handleResize);
}, []);
```

✓ return () => {} ensures **event listeners are removed when the component unmounts**.

CHAPTER 4: COMBINING USESTATE AND USEEFFECT IN REAL-WORLD APPLICATIONS

4.1 Implementing a Live Search Feature

📌 Example: Filtering a List in Real-Time

```
import React, { useState, useEffect } from "react";

function SearchList() {
  const [query, setQuery] = useState("");
  const [filteredItems, setFilteredItems] = useState([]);
```

```
  const items = ["Apple", "Banana", "Cherry", "Date"];


  useEffect(() => {
    setFilteredItems(items.filter(item =>
item.toLowerCase().includes(query.toLowerCase())));
  }, [query]);


  return (
    <div>
      <input type="text" onChange={(e) => setQuery(e.target.value)}
placeholder="Search..." />
      <ul>
        {filteredItems.map((item, index) => <li
key={index}>{item}</li>)}
      </ul>
    </div>
  );
}


export default SearchList;
```

✓ query updates dynamically, **filtering results in real-time**.

Case Study: How Instagram Uses Hooks for Real-Time Updates

Challenges Faced by Instagram

✓ Handling **real-time updates** for new posts and comments.
✓ Managing **state efficiently** for likes, shares, and user interactions.

Solutions Implemented

✓ Used useState to store **post data dynamically**.
✓ Implemented useEffect for **fetching new posts automatically**.
✓ Used **event listeners** to track user activity.

🔹 **Key Takeaways from Instagram's Use of Hooks:**

✓ useState keeps UI updates smooth for user interactions.
✓ useEffect fetches new data without **reloading the page**.
✓ Hooks enable **optimized, real-time user experiences**.

📝 Exercise

✅ Implement a **counter** using useState.
✅ Fetch **weather data** using useEffect and display it.
✅ Create a **to-do list** where users can **add and remove tasks** dynamically.
✅ Track **window size changes** and update the UI accordingly.

Conclusion

✓ **useState enables dynamic state updates** in functional components.
✓ **useEffect manages side effects** like API calls and event listeners.
✓ **Combining hooks improves app interactivity** without reloading the page.

✓ **Hooks power modern web applications** by handling real-time data and user interactions.

CONDITIONAL RENDERING & FORMS IN REACT

CHAPTER 1: INTRODUCTION TO CONDITIONAL RENDERING & FORMS

1.1 What is Conditional Rendering in React?

Conditional rendering in React allows components to **display different content based on conditions**. It enables dynamic UI updates depending on **state changes, user actions, or API responses**.

- **Why Use Conditional Rendering?**

✓ Improves **user experience** by showing/hiding content dynamically.
✓ Reduces **unnecessary re-renders** and performance overhead.
✓ Enables **better user interaction** (e.g., login/logout states).

- **Example: Basic Conditional Rendering**

```
const isLoggedIn = true;

return isLoggedIn ? <h1>Welcome Back!</h1> : <h1>Please Log In</h1>;
```

✓ Displays different messages **based on login state**.

CHAPTER 2: METHODS OF CONDITIONAL RENDERING

2.1 Using if-else Statements

📌 **Example: Showing a Logout Button Only if Logged In**

```
function UserGreeting({ isLoggedIn }) {
```

```
  if (isLoggedIn) {
    return <button>Logout</button>;
  } else {
    return <button>Login</button>;
  }
}
```

✓ if-else handles **complex conditional UI logic**.

2.2 Using the Ternary Operator (? :)

📌 **Example: Displaying a Welcome Message Based on User Login**

```
const isLoggedIn = true;
return (
  <div>
    {isLoggedIn ? <h1>Welcome, User!</h1> : <h1>Please Sign In</h1>}
  </div>
);
```

✓ **Shorter syntax** than if-else.

2.3 Using Logical && Operator

📌 **Example: Showing a Message Only if Condition is True**

```
const isAdmin = true;
```

```
return (
  <div>
    {isAdmin && <h2>Admin Dashboard</h2>}
  </div>
);
```

✓ && renders content **only when the condition is true**.

2.4 Using switch-case for Multiple Conditions

📌 **Example: Displaying User Roles**

```
function UserRole({ role }) {
  switch (role) {
    case "admin":
      return <h1>Admin Panel</h1>;
    case "editor":
      return <h1>Editor Dashboard</h1>;
    default:
      return <h1>Guest View</h1>;
  }
}
```

✓ Best for handling **multiple conditional cases**.

CHAPTER 3: HANDLING FORMS IN REACT

3.1 Why Use Forms in React?

Forms allow users to **input data**, which React can handle using **state and event handlers**.

🔹 **Common Use Cases for Forms:**

✓ **User authentication (Login/Signup forms).**

✓ **Contact forms (Collect user feedback).**

✓ **Search forms (Filter results dynamically).**

3.2 Controlled vs. Uncontrolled Components

📌 **Controlled Component:**

✓ Uses useState() to track form values.

✓ Updates state **on every user input**.

✓ Recommended for **better control** over form data.

📌 **Uncontrolled Component:**

✓ Uses **direct DOM manipulation** (useRef).

✓ Ideal for **one-time form submissions**.

3.3 Creating a Controlled Form with useState

📌 **Example: Handling Form Inputs in React**

```
import { useState } from "react";


function LoginForm() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
```

```
  const handleSubmit = (event) => {
    event.preventDefault();
    console.log("Email:", email, "Password:", password);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="email" value={email} onChange={(e) =>
setEmail(e.target.value)} placeholder="Email" />
      <input type="password" value={password} onChange={(e) =>
setPassword(e.target.value)} placeholder="Password" />
      <button type="submit">Login</button>
    </form>
  );
}

export default LoginForm;
```

✓ useState() stores **input values dynamically**.
✓ onChange() updates **state on user input**.
✓ onSubmit() prevents **page reload and logs data**.

3.4 Using useRef for Uncontrolled Forms

📌 Example: Submitting Form Without useState

```
import { useRef } from "react";


function ContactForm() {
  const nameRef = useRef();
  const messageRef = useRef();


  const handleSubmit = (event) => {
    event.preventDefault();
    console.log("Name:", nameRef.current.value, "Message:",
messageRef.current.value);
  };


  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={nameRef} placeholder="Your Name" />
      <textarea ref={messageRef} placeholder="Your
Message"></textarea>
      <button type="submit">Send</button>
    </form>
  );
}
```

```
export default ContactForm;
```

✓ useRef() **accesses input values without re-renders**.

CHAPTER 4: FORM VALIDATION IN REACT

4.1 Basic Input Validation

📌 Example: Checking if an Email is Valid

```
const [email, setEmail] = useState("");
const [error, setError] = useState("");

const validateEmail = () => {
  if (!email.includes("@")) {
    setError("Invalid email address");
  } else {
    setError("");
  }
};
```

✓ Ensures **only valid email formats** are accepted.

4.2 Disabling Submit Button Until Inputs Are Filled

📌 Example: Enabling Button When Fields Are Valid

```
const isDisabled = email === "" || password === "";
<button disabled={isDisabled}>Login</button>
```

✓ Improves **user experience and prevents empty submissions**.

CHAPTER 5: SUBMITTING FORM DATA TO AN API

5.1 Using Fetch API to Submit Form Data

📌 **Example: Sending Data to a Server**

```
const handleSubmit = async (event) => {
  event.preventDefault();

  const response = await fetch("https://api.example.com/login", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ email, password })
  });

  const data = await response.json();
  console.log("Response:", data);
};
```

✓ **Submits form data** and handles server responses.

Case Study: How Facebook Uses Conditional Rendering & Forms

Challenges Faced by Facebook

✓ Displaying **different content for logged-in vs. guest users**.
✓ Handling **dynamic form inputs** for login, search, and posts.

Solutions Implemented

✓ Used **conditional rendering** for login/logout states.
✓ Implemented **real-time input handling for search and messages**.
✓ Validated **form fields before submission** to prevent errors.

🔹 **Key Takeaways from Facebook's Strategy:**
✓ **Conditional rendering enhances user experience**.
✓ **Controlled forms ensure smooth data management**.
✓ **Dynamic API requests keep content updated in real-time**.

📝 **Exercise**

✅ Implement a **form with validation** (e.g., require username & email).
✅ Use **conditional rendering** to toggle a dark mode button.
✅ Fetch **user details from an API and display them dynamically**.
✅ Prevent form submission if required fields are empty.

Conclusion

✓ **Conditional rendering improves UI interactivity dynamically**.
✓ **Forms manage user input using useState or useRef**.
✓ **Validation prevents incorrect submissions and enhances user experience**.
✓ **API integration allows real-time form submissions to the server**.

ROUTING WITH REACT ROUTER

CHAPTER 1: INTRODUCTION TO REACT ROUTER

1.1 What is React Router?

React Router is a **powerful routing library** for React that enables navigation between different pages **without reloading the page**. It allows developers to build **single-page applications (SPAs)** with multiple views and dynamic routing.

- **Why Use React Router?**

✓ **Enables seamless navigation** without full page reloads.
✓ **Supports dynamic and nested routes** for complex applications.
✓ **Provides easy URL parameter handling** for passing data.

- **Common Features of React Router:**

Feature	Description
Browser-based Routing	Uses URLs for navigation instead of page reloads
Dynamic Routing	Supports route parameters (/user/:id)
Nested Routes	Allows components inside other routes
Protected Routes	Restricts access to certain pages (e.g., login required)
Redirections	Redirects users to another route based on conditions

CHAPTER 2: SETTING UP REACT ROUTER IN A REACT APP

2.1 Installing React Router

To use React Router, install it using **npm or yarn**:

npm install react-router-dom

or

yarn add react-router-dom

2.2 Setting Up Basic Routing

📌 Example: Basic Routing Setup

```
import React from "react";
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";

const Home = () => <h2>Home Page</h2>;
const About = () => <h2>About Page</h2>;

function App() {
 return (
  <Router>
   <nav>
    <Link to="/">Home</Link>
    <Link to="/about">About</Link>
   </nav>
   <Routes>
    <Route path="/" element={<Home />} />
```

```
      <Route path="/about" element={<About />} />
    </Routes>
  </Router>
 );
}

export default App;
```

🔹 **How It Works:**

✓ <Router> wraps the app to enable routing.

✓ <Link> replaces <a> to **prevent full page reloads**.

✓ <Routes> contains multiple <Route> components defining paths.

CHAPTER 3: DYNAMIC ROUTING WITH URL PARAMETERS

3.1 What are Dynamic Routes?

Dynamic routes allow parameters **inside URLs** (e.g., /user/:id) to display **different content** dynamically.

📌 **Example: Defining a Dynamic Route**

```
import { BrowserRouter as Router, Routes, Route, useParams } from "react-router-dom";

const UserProfile = () => {
 let { username } = useParams();
 return <h2>Welcome, {username}!</h2>;
};
```

```
function App() {
 return (
  <Router>
   <Routes>
    <Route path="/user/:username" element={<UserProfile />} />
   </Routes>
  </Router>
 );
}

export default App;
```

🔹 **How It Works:**

✓ useParams() retrieves the **username** from the URL.

✓ If a user visits /user/John, the app displays **"Welcome, John!"**.

3.2 Using Query Parameters in Routes

Query parameters are used **to pass optional values** in URLs.

📌 **Example: Retrieving Query Parameters**

```
import { useSearchParams } from "react-router-dom";

const ProductPage = () => {
 const [searchParams] = useSearchParams();
```

```
 return <h2>Product ID: {searchParams.get("id")}</h2>;
};
```

- **URL Example:** /product?id=123
- **Output:** Product ID: 123

CHAPTER 4: NESTED ROUTES & PROTECTED ROUTES

4.1 Implementing Nested Routes

Nested routes allow **child components** to be rendered inside **parent components**.

📌 **Example: Dashboard with Nested Pages**

```
import { BrowserRouter as Router, Routes, Route, Outlet, Link } from "react-router-dom";


const Dashboard = () => (
 <div>
  <h2>Dashboard</h2>
  <nav>
   <Link to="stats">Stats</Link>
   <Link to="settings">Settings</Link>
  </nav>
  <Outlet />
 </div>
);
```

```
const Stats = () => <h3>Statistics Page</h3>;

const Settings = () => <h3>Settings Page</h3>;


function App() {
 return (
  <Router>
   <Routes>
    <Route path="dashboard" element={<Dashboard />}>
     <Route path="stats" element={<Stats />} />
     <Route path="settings" element={<Settings />} />
    </Route>
   </Routes>
  </Router>
 );
}

export default App;
```

✓ /dashboard/stats displays **Statistics inside Dashboard**.
✓ /dashboard/settings displays **Settings inside Dashboard**.

4.2 Protecting Routes with Authentication

Protected routes restrict access to **authenticated users only**.

📌 Example: Protecting a Dashboard Route

```
import { Navigate } from "react-router-dom";


const ProtectedRoute = ({ user, children }) => {
 return user ? children : <Navigate to="/login" />;
};


<Routes>
 <Route path="/dashboard" element={<ProtectedRoute user={currentUser}><Dashboard /></ProtectedRoute>} />
</Routes>
```

✓ If user is **logged in**, they see the dashboard.
✓ If not, they are **redirected to the login page**.

CHAPTER 5: REDIRECTS & PROGRAMMATIC NAVIGATION

5.1 Redirecting Users After an Action

📌 Example: Redirecting to Home After Login

```
import { useNavigate } from "react-router-dom";


const Login = () => {
 const navigate = useNavigate();


 const handleLogin = () => {
```

```
  // Perform authentication logic
  navigate("/dashboard"); // Redirect to Dashboard
 };

 return <button onClick={handleLogin}>Login</button>;
};
```

✓ useNavigate() **redirects users programmatically** after login.

Case Study: How Facebook Uses React Router for Navigation

Challenges Faced by Facebook

✓ Handling **millions of users navigating between feeds, messages, and profiles**.
✓ Enabling **smooth transitions between pages without full reloads**.

Solutions Implemented

✓ Used **React Router for seamless navigation**.
✓ Implemented **nested routes for different sections (news feed, notifications, settings, etc.)**.
✓ Protected routes **restrict access to private pages**.

- **Key Takeaways from Facebook's Strategy:**

✓ **Dynamic routing improves user engagement.**
✓ **Protected routes ensure security and privacy.**
✓ **Nested routes help organize complex layouts efficiently.**

📝 Exercise

✅ Create a **React Router project** with Home, About, and Contact pages.
✅ Implement a **dynamic route** for user profiles (/user/:username).
✅ Protect a **dashboard route** using authentication logic.
✅ Use **useNavigate()** to redirect users after login.

Conclusion

✓ React Router enables seamless navigation in SPAs.
✓ Dynamic routes and query parameters make applications interactive.
✓ Nested routes structure content effectively.
✓ Protected routes enhance security in web applications.

REDUX TOOLKIT FOR GLOBAL STATE MANAGEMENT

CHAPTER 1: INTRODUCTION TO REDUX TOOLKIT

1.1 What is Redux Toolkit?

Redux Toolkit (RTK) is a **state management library** for React applications. It simplifies **global state management** by reducing boilerplate code and improving performance. RTK makes handling **complex application states** easier and more efficient.

- **Why Use Redux Toolkit Instead of Traditional Redux?**

✓ **Less boilerplate code** – Simplifies reducers and actions.
✓ **Built-in configureStore()** – No need for manual middleware setup.
✓ **Better performance** – Uses **Immer.js** for immutable state updates.
✓ **Built-in createSlice()** – Reduces complexity in defining actions.

- **How Redux Toolkit Works:**

1. **Stores global state** in a Redux store.
2. **Updates state via reducers** using createSlice().
3. **Connects components** to the store using useSelector() and useDispatch().

CHAPTER 2: SETTING UP REDUX TOOLKIT IN A REACT PROJECT

2.1 Installing Redux Toolkit

📌 **Run the following command to install Redux Toolkit and React-Redux:**

npm install @reduxjs/toolkit react-redux

or

yarn add @reduxjs/toolkit react-redux

2.2 Creating the Redux Store

The Redux store **holds global state** and is configured using configureStore().

📌 **Example: Setting Up store.js**

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";

export const store = configureStore({
  reducer: {
    counter: counterReducer
  }
});
```

✓ configureStore() **sets up the Redux store** automatically.
✓ counterReducer manages the **counter slice of the state**.

2.3 Providing the Store to React Components

To make the store **available to all components,** wrap the app inside <Provider>.

📌 **Example: Connecting Redux Store to App in index.js**

```
import React from "react";
import ReactDOM from "react-dom";
import { Provider } from "react-redux";
import { store } from "./store";
import App from "./App";

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

✓ The <Provider> component makes **Redux state accessible** to all components.

CHAPTER 3: MANAGING STATE WITH REDUX SLICES

3.1 What is a Slice?

A **slice** is a collection of **state, reducers, and actions** related to a specific feature in Redux Toolkit.

📌 **Example: Creating a Counter Slice in counterSlice.js**

```
import { createSlice } from "@reduxjs/toolkit";

const initialState = { value: 0 };

const counterSlice = createSlice({
  name: "counter",
  initialState,
  reducers: {
    increment: (state) => { state.value += 1; },
    decrement: (state) => { state.value -= 1; },
    reset: (state) => { state.value = 0; }
  }
});

export const { increment, decrement, reset } = counterSlice.actions;
export default counterSlice.reducer;
```

✓ createSlice() generates **state, reducers, and actions** in one step.
✓ **No need to write action types manually** (like in traditional Redux).

CHAPTER 4: ACCESSING REDUX STATE IN COMPONENTS

4.1 Using useSelector() to Retrieve State

Components use useSelector() to **access state from Redux store**.

📌 Example: Displaying Counter Value

```
import { useSelector } from "react-redux";

function CounterDisplay() {
  const count = useSelector(state => state.counter.value);
  return <h1>Counter: {count}</h1>;
}

export default CounterDisplay;
```

✓ useSelector() extracts **the counter state** from Redux.

4.2 Updating State with useDispatch()

Components use useDispatch() to **update state via Redux actions**.

📌 Example: Buttons to Modify Counter Value

```
import { useDispatch } from "react-redux";
import { increment, decrement, reset } from "./counterSlice";

function CounterButtons() {
  const dispatch = useDispatch();

  return (
    <div>
```

```
        <button onClick={() =>
dispatch(increment())}>Increment</button>

        <button onClick={() =>
dispatch(decrement())}>Decrement</button>

        <button onClick={() => dispatch(reset())}>Reset</button>

      </div>

   );

}


export default CounterButtons;
```

✓ **Clicking a button triggers an action**, updating the global state.

CHAPTER 5: HANDLING ASYNCHRONOUS STATE WITH CREATEASYNCTHUNK

5.1 Fetching Data Using Redux Toolkit

createAsyncThunk() simplifies handling **asynchronous API requests**.

📌 **Example: Fetching User Data in userSlice.js**

```
import { createSlice, createAsyncThunk } from "@reduxjs/toolkit";


export const fetchUsers = createAsyncThunk("users/fetchUsers",
async () => {

   const response = await
fetch("https://jsonplaceholder.typicode.com/users");
```

```
  return response.json();
});

const usersSlice = createSlice({
  name: "users",
  initialState: { users: [], loading: false },
  reducers: {},
  extraReducers: (builder) => {
    builder
      .addCase(fetchUsers.pending, (state) => { state.loading = true;
})
      .addCase(fetchUsers.fulfilled, (state, action) => {
        state.loading = false;
        state.users = action.payload;
      });
  }
});

export default usersSlice.reducer;
```

✓ fetchUsers.pending sets loading = true.
✓ fetchUsers.fulfilled updates the user list when data arrives.

CHAPTER 6: PERSISTING REDUX STATE

6.1 Storing Redux State in Local Storage

Redux state resets when the page reloads. To **persist state,** store it in **local storage**.

📌 Example: Saving State to Local Storage

```
import { configureStore } from "@reduxjs/toolkit";
import counterReducer from "./counterSlice";

const loadState = () => {
  try {
    const savedState = localStorage.getItem("reduxState");
    return savedState ? JSON.parse(savedState) : undefined;
  } catch (error) {
    return undefined;
  }
};

export const store = configureStore({
  reducer: { counter: counterReducer },
  preloadedState: loadState()
});

store.subscribe(() => {
```

```
  localStorage.setItem("reduxState",
JSON.stringify(store.getState()));

});
```

✓ **Loads previous Redux state** from local storage.
✓ **Automatically saves new state** after every change.

Case Study: How Spotify Uses Redux for State Management

Challenges Faced by Spotify

✓ Managing **playlists, user authentication, and audio playback** globally.
✓ Ensuring **state consistency** between different components.

Solutions Implemented

✓ Used **Redux Toolkit for managing user authentication**.
✓ Implemented **async state handling** for fetching songs and playlists.
✓ Used **persistent storage to remember user preferences**.

- **Key Takeaways from Spotify's Strategy:**

✓ **Global state improves data consistency.**
✓ **Redux simplifies complex app state management.**
✓ **Persisting state enhances user experience.**

📝 **Exercise**

✅ Create a **Redux store** and connect it to a React app.
✅ Implement a **counter slice** with increment, decrement, and reset actions.

✅ Fetch **user data from an API** using createAsyncThunk().

✅ Persist Redux state in **local storage**.

Conclusion

✓ **Redux Toolkit simplifies state management** in React applications.

✓ **Slices reduce boilerplate** and make state updates easier.

✓ **Async state management is handled efficiently** using createAsyncThunk().

✓ **Persisting state improves user experience** across sessions.

FETCHING DATA IN REACT WITH API CALLS

CHAPTER 1: INTRODUCTION TO API CALLS IN REACT

1.1 What is an API Call?

An **API (Application Programming Interface) call** allows a React application to **fetch, send, or update data** from a remote server. APIs are used to retrieve data dynamically from **databases, third-party services, or cloud applications**.

- **Why Use API Calls in React?**

✓ Fetch **real-time data** (news, weather, stock prices).
✓ Send **user inputs** (login details, form submissions).
✓ Enable **dynamic content updates** (social media feeds, chat messages).

- **Types of API Requests:**

Request Type	Purpose	Example
GET	Retrieve data	Fetch user list
POST	Send data to a server	Submit form
PUT	Update data	Edit user profile
DELETE	Remove data	Delete a post

CHAPTER 2: FETCHING API DATA USING FETCH API

2.1 What is the Fetch API?

The **Fetch API** is a built-in JavaScript function that allows React to **make HTTP requests** to retrieve or send data.

🔹 **Why Use Fetch API?**

✓ **Native JavaScript feature** (no need for additional libraries).

✓ Supports **async/await** for cleaner asynchronous code.

✓ Works well with **JSON responses**.

2.2 Fetching Data with useEffect in React

📌 **Example: Fetching User Data from an API**

```
import React, { useState, useEffect } from "react";

function UsersList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then(response => response.json()) // Convert to JSON
      .then(data => setUsers(data)) // Store data in state
      .catch(error => console.error("Error:", error));
  }, []); // Runs once when component mounts

  return (
    <ul>
      {users.map(user => (
        <li key={user.id}>{user.name}</li>
```

```
        ))}
      </ul>
    );
}

export default UsersList;
```

🔹 **How It Works:**

✓ fetch() **retrieves** data from the API.

✓ response.json() **parses** data into JavaScript format.

✓ useEffect() ensures **data fetches only once** when the component loads.

2.3 Handling API Errors Gracefully

API requests **can fail** due to network issues or incorrect URLs.

📌 **Example: Adding Error Handling to API Calls**

```
useEffect(() => {
  fetch("https://wrong-url.com")
    .then(response => {
      if (!response.ok) throw new Error("Failed to fetch data");
      return response.json();
    })
    .then(data => setUsers(data))
    .catch(error => console.error("Fetch Error:", error));
```

}, []);

✓ if (!response.ok) **detects errors**.

✓ catch(error => console.error(error)) **logs errors in the console**.

CHAPTER 3: USING AXIOS FOR API REQUESTS

3.1 What is Axios?

Axios is a **JavaScript library** that simplifies API requests by handling JSON parsing and error responses automatically.

🔹 **Why Use Axios Instead of Fetch?**

✓ **Easier syntax** (less code).

✓ **Automatic JSON parsing** (no need for response.json()).

✓ **Better error handling** with .catch().

📌 **Installing Axios:**

npm install axios

or

yarn add axios

3.2 Fetching Data Using Axios

📌 **Example: Fetching Posts from an API with Axios**

import React, { useState, useEffect } from "react";

import axios from "axios";

function PostsList() {

```
  const [posts, setPosts] = useState([]);


  useEffect(() => {
    axios.get("https://jsonplaceholder.typicode.com/posts")
      .then(response => setPosts(response.data))
      .catch(error => console.error("Error:", error));
  }, []);


  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
}

export default PostsList;
```

✓ axios.get() **fetches data in fewer lines** than fetch().
✓ response.data automatically extracts the **JSON payload**.

3.3 Sending Data to an API with POST Request

📌 Example: Submitting a New Post with Axios

```
import axios from "axios";

function createPost() {
  axios.post("https://jsonplaceholder.typicode.com/posts", {
    title: "New Post",
    body: "This is a new post",
    userId: 1
  })
  .then(response => console.log("Post created:", response.data))
  .catch(error => console.error("Error:", error));
}
```

✓ axios.post() sends **data to the API server**.
✓ catch(error) handles **request failures**.

CHAPTER 4: DISPLAYING API DATA DYNAMICALLY IN REACT

4.1 Updating UI Based on API Data

📌 **Example: Displaying Weather Data Dynamically**

```
import React, { useState, useEffect } from "react";

function WeatherApp() {
  const [weather, setWeather] = useState(null);

  useEffect(() => {
```

```
fetch("https://api.weatherapi.com/v1/current.json?key=YOUR_API_KEY&q=New York")
      .then(response => response.json())
      .then(data => setWeather(data.current));
  }, []);

  return weather ? <h2>Temperature: {weather.temp_c}°C</h2> : <p>Loading...</p>;
}

export default WeatherApp;
```

✓ **Live weather updates** without refreshing the page.

4.2 Using Loading States While Fetching Data

📌 Example: Displaying a Loading Message Until API Data is Ready

```
const [loading, setLoading] = useState(true);
const [data, setData] = useState(null);

useEffect(() => {
  fetch("https://api.example.com/data")
    .then(response => response.json())
    .then(data => {
```

```
        setData(data);
        setLoading(false);
      });
}, []);

return loading ? <p>Loading...</p> : <p>Data Loaded</p>;
```

✓ **Prevents UI from displaying empty data while fetching.**

Case Study: How Airbnb Uses API Calls for Real-Time Listings

Challenges Faced by Airbnb

✓ Fetching **real-time availability of properties**.
✓ Managing **user searches dynamically**.
✓ Ensuring **fast response times**.

Solutions Implemented

✓ Used **Axios for efficient API requests**.
✓ Implemented **caching** to reduce API load times.
✓ Optimized **API responses using pagination**.

🔹 **Key Takeaways from Airbnb's Strategy:**

✓ **Fast, efficient API requests improve user experience.**
✓ **Handling errors and loading states prevents UI glitches.**
✓ **Using useEffect ensures real-time updates for user searches.**

📝 **Exercise**

✅ Create a React component that **fetches and displays user data** using Fetch API.

✅ Use Axios to **send a form submission to an API**.

✅ Implement **loading states** while waiting for API data.

✅ Handle **API errors gracefully** and display an error message.

Conclusion

✓ Fetching API data enhances dynamic web applications.

✓ Fetch API and Axios are essential for making HTTP requests in React.

✓ Handling loading states and errors improves user experience.

✓ Using APIs enables real-time updates for apps like Airbnb, Twitter, and Weather apps.

ASSIGNMENT:

BUILD A WEATHER APP USING REACT AND OPENWEATHER API

Assignment Solution: Build a Weather App Using React and OpenWeather API

Step 1: Setting Up the React Project

1.1 Create a React App

📌 Open the terminal and run:

```
npx create-react-app weather-app

cd weather-app

npm install axios
```

✓ npx create-react-app weather-app ⟶ Creates a React project.
✓ npm install axios ⟶ Installs Axios for API requests.

Step 2: Get an OpenWeather API Key

1. **Sign up** at OpenWeather.
2. Go to **API Keys** ⟶ Generate a new API key.
3. Copy the API key (e.g., "YOUR_API_KEY").

Step 3: Creating the Weather Component

3.1 Create a Component Weather.js

📌 **Inside src/, create Weather.js and add the following:**

```
import React, { useState } from "react";
```

```
import axios from "axios";


const Weather = () => {
  const [city, setCity] = useState("");
  const [weather, setWeather] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  const API_KEY = "YOUR_API_KEY"; // Replace with your
OpenWeather API key

  const fetchWeather = async () => {
    if (!city) {
      setError("Please enter a city name.");
      return;
    }
    setLoading(true);
    setError("");

    try {
      const response = await axios.get(
```

```
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid
=${API_KEY}&units=metric`
        );
        setWeather(response.data);
      } catch (err) {
        setError("City not found. Please enter a valid city.");
      } finally {
        setLoading(false);
      }
    };

    return (
      <div className="weather-container">
        <h2>Weather App</h2>
        <input
          type="text"
          placeholder="Enter city..."
          value={city}
          onChange={(e) => setCity(e.target.value)}
        />
        <button onClick={fetchWeather}>Get Weather</button>
```

```
        {loading && <p>Loading...</p>}
        {error && <p className="error">{error}</p>}

        {weather && (
          <div className="weather-info">
            <h3>{weather.name}, {weather.sys.country}</h3>
            <p>Temperature: {weather.main.temp}°C</p>
            <p>Humidity: {weather.main.humidity}%</p>
            <p>Condition: {weather.weather[0].description}</p>
          </div>
        )}
      </div>
    );
};

export default Weather;
```

- **How It Works:**

✓ useState() manages **city input, weather data, loading state, and errors**.
✓ fetchWeather() fetches **real-time weather data** using Axios.
✓ Displays **loading state, error messages, and weather info**.

Step 4: Integrating Weather Component in App.js

📌 **Replace src/App.js with:**

```
import React from "react";
import Weather from "./Weather";
import "./App.css";

function App() {
  return (
    <div className="app-container">
      <Weather />
    </div>
  );
}

export default App;
```

✓ **Imports and renders Weather component** inside the app.

Step 5: Adding CSS for Styling

📌 **Create src/App.css and add:**

```
.app-container {
  text-align: center;
  font-family: Arial, sans-serif;
  background: linear-gradient(to right, #2193b0, #6dd5ed);
```

```
    height: 100vh;
    display: flex;
    justify-content: center;
    align-items: center;
}

.weather-container {
    background: white;
    padding: 20px;
    border-radius: 10px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    width: 300px;
}

input {
    width: 80%;
    padding: 10px;
    margin: 10px 0;
    border: 1px solid #ccc;
    border-radius: 5px;
}
```

```
button {
  padding: 10px 15px;
  border: none;
  background: #ff5733;
  color: white;
  font-weight: bold;
  cursor: pointer;
  border-radius: 5px;
}

button:hover {
  background: #ff794d;
}

.weather-info {
  margin-top: 10px;
}

.error {
  color: red;
}
```

✓ Styled input, button, and weather display for a clean UI.

Step 6: Running the Weather App

📌 **Start the React app by running:**

npm start

✓ Opens http://localhost:3000/, where users can **search for a city and get real-time weather details**.

Case Study: How Weather Apps Use APIs for Real-Time Data

Challenges in Weather Apps

✓ Fetching **accurate, real-time weather data**.
✓ Handling **multiple cities and different weather conditions**.
✓ Providing a **user-friendly and fast interface**.

Solutions Implemented

✓ Used **OpenWeather API** for **real-time weather updates**.
✓ Implemented **error handling for incorrect city names**.
✓ Styled the app with **a clean and responsive UI**.

🔹 **Key Takeaways:**

✓ **APIs enable dynamic weather updates without page reloads.**
✓ **Proper error handling improves user experience.**
✓ **Fetching data on user input creates an interactive app.**

📝 Exercise

✅ **Improve the UI** by adding weather icons based on conditions.
✅ **Enhance the app** by displaying **5-day weather forecasts**.

✅ **Use geolocation API** to fetch weather **based on the user's location**.

Conclusion

✓ **React can fetch and display real-time weather data using APIs.**
✓ **Axios simplifies API calls and handles errors efficiently.**
✓ **Dynamic UI updates improve user experience.**
✓ **Building projects like this strengthens API and React skills.**