Day 23

React Basics with Vite (JSX, Components, Props) - Brief with Examples

1. Setting Up React with Vite

First, let's create a new React project using Vite:

```
npm create vite@latest my-react-app --template react
cd my-react-app
npm install
npm run dev
This starts a development server at http://localhost:5173.
```

2. JSX (JavaScript XML)

JSX allows writing HTML-like syntax inside JavaScript.

Example: Rendering Dynamic Content

```
// App.jsx
function App() {
  const name = "John Doe";
  return (
    <div>
      <h1>Hello, {name}!</h1>  {/* JSX with dynamic value */}
      <p>Today is {new Date().toLocaleDateString()}</p>
    </div>
  );
}
```

Output:

Hello, John Doe!

Today is 6/18/2025

3. Components

Components are reusable UI blocks.

Example: Button Component

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

// App.jsx
import Button from "./Button";

function App() {
  return (
    <div>
      <Button label="Click Me" />
      <Button label="Submit" />
    </div>
  );
}
```

Output:

Two buttons: Click Me and Submit.

4. Props (Properties)

Props pass data from parent to child components.

Example: User Profile Component

```
// UserProfile.jsx
function UserProfile({ name, age, email }) {
  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
```

```
    <p>Email: {email}</p>
   </div>
  );
}

// App.jsx
import UserProfile from "./UserProfile";

function App() {
  return (
   <div>
    <UserProfile name="Alice" age={25} email="alice@example.com" />
    <UserProfile name="Bob" age={30} email="bob@example.com" />
   </div>
  );
}
```

Output:

Two user profiles with different data.

5. Real-World Example: Todo List

Let's build a simple **Todo List** using components and props.

TodoItem Component

```
// TodoItem.jsx
function TodoItem({ task, completed }) {
  return (
   <li style={{ textDecoration: completed ? "line-through" : "none" }}>
    {task}
   </li>
  );
}
```

TodoList Component

```
// TodoList.jsx
import TodoItem from "./TodoItem";

function TodoList() {
  const todos = [
    { task: "Learn React", completed: false },
    { task: "Build a project", completed: true },
    { task: "Deploy to Vercel", completed: false },
  ];

  return (
    <ul>
      {todos.map((todo, index) => (
        <TodoItem key={index} task={todo.task} completed={todo.completed} />
      ))}
    </ul>
  );
}
```

App.jsx

```
import TodoList from "./TodoList";

function App() {
  return (
    <div>
      <h1>My Todo List</h1>
      <TodoList />
    </div>
  );
}
```

Output:

- ✅ Learn React
- ~~Build a project~~ (completed)
- ❌ Deploy to Vercel

Key Takeaways

✓ JSX = HTML-like syntax in JavaScript.

✓ Components = Reusable UI blocks (like Button, UserProfile).

✓ Props = Pass data from parent to child (name, age, task).

✓ Vite = Faster React setup than create-react-app.

Day 24

State & Hooks (useState, useEffect) - React Fundamentals

1. What is State?

State is data that changes over time in a component.

Example: A counter, form input, API data.

2. useState Hook

useState lets you add state to functional components.

Basic Syntax

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

- state → Current value
- setState → Function to update state
- initialValue → Starting value

Example 1: Counter App

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

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

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

Output:

- A counter that increases/decreases when clicked.

Example 2: Toggle Button (Boolean State)

```
function ToggleButton() {
 const [isOn, setIsOn] = useState(false);

 return (
  <button onClick={() => setIsOn(!isOn)}>
   {isOn ? "ON" : "OFF"}
  </button>
 );
}
```

Output:

- A button that toggles between **ON** and **OFF**.

3. useEffect Hook

useEffect lets you perform **side effects** (e.g., API calls, timers, DOM updates).

Basic Syntax

```
useEffect(() => {
 // Side effect code
 return () => {
  // Cleanup (optional)
```

```
  };
}, [dependencies]);
```

- Runs after component renders.
- Dependencies control when it re-runs.

Example 1: Fetching Data (API Call)

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

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

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/users")
      .then((res) => res.json())
      .then((data) => setUsers(data));
  }, []); // Empty dependency = runs once on mount

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

Output:

- Fetches and displays a list of users.

Example 2: Updating Document Title

```
function DocumentTitleUpdater() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `Count: ${count}`;
```

```
  }, [count]); // Runs when `count` changes

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

Output:

- The browser tab title updates with the current count.

Example 3: Cleanup (Timer)

```
function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const interval = setInterval(() => {
      setSeconds((prev) => prev + 1);
    }, 1000);

    return () => clearInterval(interval); // Cleanup on unmount
  }, []);

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

Output:

- A timer that increments every second.
- Stops when the component unmounts.

4. Real-World Example: Todo List with State

Let's enhance our Todo List from Day 23 with state management.

Step 1: Add State for Todos

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

function TodoList() {
  const [todos, setTodos] = useState([
    { id: 1, task: "Learn React", completed: false },
    { id: 2, task: "Build a project", completed: true },
  ]);

  const toggleComplete = (id) => {
    setTodos(
      todos.map((todo) =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  };

  return (
    <ul>
      {todos.map((todo) => (
        <li
          key={todo.id}
          style={{ textDecoration: todo.completed ? "line-through" : "none" }}
          onClick={() => toggleComplete(todo.id)}
        >
          {todo.task}
        </li>
      ))}
    </ul>
  );
}
```

Output:

- Clicking a todo **toggles** its completion status.

Step 2: Add New Todos (Form Input)

```
function TodoList() {
  const [todos, setTodos] = useState([]);
  const [input, setInput] = useState("");

  const addTodo = () => {
    if (input.trim()) {
      setTodos([...todos, { id: Date.now(), task: input, completed: false }]);
      setInput("");
    }
  };

  return (
    <div>
      <input
        type="text"
        value={input}
        onChange={(e) => setInput(e.target.value)}
        placeholder="Add a new task"
      />
      <button onClick={addTodo}>Add</button>
      <ul>
        {todos.map((todo) => (
          <li key={todo.id}>{todo.task}</li>
        ))}
      </ul>
    </div>
  );
}
```

Output:

- Type a task and click Add to include it in the list.

Key Takeaways

✓ useState → Manages dynamic data in components.

✓ useEffect → Handles side effects (API calls, timers, etc.).

✓ Dependencies → Controls when useEffect runs.

✓ Cleanup → Prevents memory leaks (e.g., clearInterval).

Day 25

Conditional Rendering & Lists – Professional Edition

1. Conditional Rendering (When & How)

Real-World Use Cases:

- Auth Walls: Show admin buttons only if user.role === 'admin'
- Empty States: Display "No search results" vs data table
- Feature Flags: Hide unfinished features in production

Pro Patterns:

```
// 1. Component-level gates (cleaner than nested ternaries)
function AdminPanel() {
  if (!user.isAdmin) return <UpgradePrompt />;
  return <AdminTools />;
}

// 2. Fragment shorthand for multiple elements
{hasItems && (
  <>
    <SearchResults />
    <Pagination />
  </>
```

```
)}
```

Live Example:

```
// Netflix-style "New Episode" badge
function MediaCard({ show }) {
  return (
    <div>
      <Thumbnail src={show.image} />
      {show.isNew && <div className="badge">NEW</div>}
    </div>
  );
}
```

2. Lists & Keys (Production-Grade)

Critical Rules:

- Never use index as key (causes bugs with CRUD operations)
- Extract list items to separate components for performance

Enterprise Example:

```
// ✅ Correct: Stable ID from database
function UserList({ users }) {
  return (
    <ul>
      {users.map(user => (
        <UserItem key={user.id} {...user} />
      ))}
    </ul>
  );
}
// ❌ Dangerous: Index keys break on re-sorts
{users.map((user, index) => <Item key={index} />}
```

Pro Tip:

Use uuid for client-side generated keys:

```
npm install uuid
```

```
import { v4 as uuidv4 } from 'uuid';

key={uuidv4()}
```

3. Real-World Project (E-Commerce Cart)

Task: Build a cart that:

1. Shows "Empty cart" message when no items
2. Displays items with quantity controls
3. Highlights low-stock items (< 5 remaining)

```
function Cart() {
 const [items, setItems] = useState([]);
 return (
  <div>
   <h2>Your Cart</h2>
   {items.length === 0 ? (
    <EmptyCartCTA />
   ) : (
    <ul>
     {items.map(item => (
      <li
       key={item.id}
       className={item.stock < 5 ? 'low-stock' : ''}
      >
       {item.name} - ${item.price}
       <button onClick={() => removeItem(item.id)}>🗑</button>
      </li>
     ))}
    </ul>
   )}
```

```
  </div>
 );
}
```

4. Interview-Ready Challenge

Problem: Render a dashboard with:

- Loading spinner while fetching data
- Error message if API fails
- Data table with alternating row colors

Solution Framework:

```
function Dashboard() {
 const { data, error, loading } = useFetch('/api/metrics');
 if (loading) return <Spinner />;
 if (error) return <ErrorAlert message={error.message} />;
 return (
  <table>
   {data.map((row, index) => (
    <tr
     key={row.id}
     className={index % 2 === 0 ? 'even' : 'odd'}
    >
     <td>{row.metric}</td>
     <td>{row.value}</td>
    </tr>
   ))}
  </table>
 );
}
```

Day 26

Forms in React (Controlled Components)

1. Controlled vs Uncontrolled Components

Feature	Controlled Components	Uncontrolled Components
State	Managed by React (useState)	Managed by the DOM
Updates	Via onChange + setState	Via ref or DOM APIs
Best For	Forms with validation, dynamic updates	Simple forms, file inputs

2. Basic Controlled Form Example

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

function LoginForm() {
  const [formData, setFormData] = useState({
    email: "",
    password: "",
  });

  const handleChange = (e) => {
    const { name, value } = e.target;
    setFormData({ ...formData, [name]: value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("Submitted:", formData);
    // Add API call here
  };
```

```
 return (
  <form onSubmit={handleSubmit}>
   <input
    type="email"
    name="email"
    value={formData.email}
    onChange={handleChange}
    placeholder="Email"
   />
   <input
    type="password"
    name="password"
    value={formData.password}
    onChange={handleChange}
    placeholder="Password"
   />
   <button type="submit">Login</button>
  </form>
 );
}
```

Key Points:

- Each input is bound to state (value={formData.field}).
- Changes update state via onChange.
- Form submission accesses the latest state.

3. Handling Different Input Types

Text Inputs, Textarea, Select

```
function UserProfileForm() {
 const [formData, setFormData] = useState({
  username: "",
  bio: "",
```

```
  country: "usa",
  subscribe: false,
});

const handleChange = (e) => {
  const { name, value, type, checked } = e.target;
  setFormData({
    ...formData,
    [name]: type === "checkbox" ? checked : value,
  });
};

return (
  <form>
    <input
      type="text"
      name="username"
      value={formData.username}
      onChange={handleChange}
    />
    <textarea
      name="bio"
      value={formData.bio}
      onChange={handleChange}
    />
    <select
      name="country"
      value={formData.country}
      onChange={handleChange}
    >
      <option value="usa">USA</option>
      <option value="uk">UK</option>
```

```
    </select>
    <label>
      <input
        type="checkbox"
        name="subscribe"
        checked={formData.subscribe}
        onChange={handleChange}
      />
      Subscribe to newsletter
    </label>
  </form>
 );
}
```

File Input (Uncontrolled)

```
function FileUpload() {
 const fileInput = useRef(null);

 const handleSubmit = (e) => {
  e.preventDefault();
  console.log("Selected file:", fileInput.current.files[0]);
 };

 return (
  <form onSubmit={handleSubmit}>
   <input type="file" ref={fileInput} />
   <button type="submit">Upload</button>
  </form>
 );
}
```

4. Form Validation

```
function SignupForm() {
```

```
const [formData, setFormData] = useState({
  email: "",
  password: "",
});
const [errors, setErrors] = useState({});

const validate = () => {
  const newErrors = {};
  if (!formData.email.includes("@")) {
    newErrors.email = "Invalid email";
  }
  if (formData.password.length < 6) {
    newErrors.password = "Password too short";
  }
  setErrors(newErrors);
  return Object.keys(newErrors).length === 0;
};

const handleSubmit = (e) => {
  e.preventDefault();
  if (validate()) {
    console.log("Valid form:", formData);
  }
};

return (
  <form onSubmit={handleSubmit}>
    <input
      type="email"
      name="email"
      value={formData.email}
      onChange={handleChange}
```

```
    />
    {errors.email && <p className="error">{errors.email}</p>}
    <input
      type="password"
      name="password"
      value={formData.password}
      onChange={handleChange}
    />
    {errors.password && <p className="error">{errors.password}</p>}
    <button type="submit">Sign Up</button>
  </form>
 );
}
```

5. Real-World Example: Multi-Step Form

```
function MultiStepForm() {
 const [step, setStep] = useState(1);
 const [formData, setFormData] = useState({
  name: "",
  email: "",
  address: "",
 });

 const nextStep = () => setStep(step + 1);
 const prevStep = () => setStep(step - 1);

 return (
  <form>
   {step === 1 && (
    <div>
     <input
      name="name"
```

```
        value={formData.name}
        onChange={handleChange}
        placeholder="Name"
      />
      <button type="button" onClick={nextStep}>
        Next
      </button>
    </div>
  )}
  {step === 2 && (
    <div>
      <input
        name="email"
        value={formData.email}
        onChange={handleChange}
        placeholder="Email"
      />
      <button type="button" onClick={prevStep}>
        Back
      </button>
      <button type="button" onClick={nextStep}>
        Next
      </button>
    </div>
  )}
  {step === 3 && (
    <div>
      <input
        name="address"
        value={formData.address}
        onChange={handleChange}
        placeholder="Address"
```

```
        />
        <button type="button" onClick={prevStep}>
          Back
        </button>
        <button type="submit">Submit</button>
      </div>
    )}
  </form>
 );
}
```

Key Takeaways

✓ Controlled Components = Form inputs bound to React state.

✓ Uncontrolled Components = Use ref for file inputs or simple cases.

✓ Validation = Check inputs before submission.

✓ Multi-Step Forms = Manage steps with state.

Day 27

React Router (Basic Routing, NavLink)

1. Setting Up React Router

First, install React Router in your project:

```
npm install react-router-dom
```

2. Basic Routing Setup

Wrap your app with <BrowserRouter> and define routes
with <Routes> and <Route>.

Example: Basic Routes

```
// main.jsx / App.jsx
import { BrowserRouter, Routes, Route } from "react-router-dom";
```

```
import Home from "./Home";
import About from "./About";
import Contact from "./Contact";

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </BrowserRouter>
  );
}
```

3. Navigation with <Link> and <NavLink>

- <Link> → For basic navigation.
- <NavLink> → Adds active class styling when the link is active.

Example: Navigation Menu

```
import { Link, NavLink } from "react-router-dom";

function Navbar() {
  return (
    <nav>
      {/* Regular Link */}
      <Link to="/">Home</Link>

      {/* NavLink (adds 'active' class automatically) */}
      <NavLink
        to="/about"
        className={({ isActive }) => isActive ? "active-link" : ""}
```

```
    >
      About
    </NavLink>

    <NavLink to="/contact">Contact</NavLink>
   </nav>
  );
}
```

CSS for Active Link:

```
.active-link {
  color: red;
  font-weight: bold;
}
```

4. Dynamic Routes & URL Parameters

Extract parameters from URLs using useParams().

Example: User Profile Page

```
// App.jsx (Route Definition)
<Route path="/user/:id" element={<UserProfile />} />

// UserProfile.jsx
import { useParams } from "react-router-dom";

function UserProfile() {
  const { id } = useParams();
  return <h1>User ID: {id}</h1>;
}
```

URL Example:

/user/123 → Displays **"User ID: 123"**

5. Programmatic Navigation (useNavigate)

Redirect users programmatically (e.g., after login).

Example: Login Redirect

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

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

  const handleLogin = () => {
    // ... authentication logic
    navigate("/dashboard"); // Redirects to dashboard
  };

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

6. Nested Routes

Organize routes hierarchically.

Example: Dashboard Layout

```
// App.jsx
<Route path="/dashboard" element={<Dashboard />}>
  <Route path="stats" element={<Stats />} />
  <Route path="settings" element={<Settings />} />
</Route>

// Dashboard.jsx (Parent Layout)
import { Outlet } from "react-router-dom";

function Dashboard() {
  return (
    <div>
```

```
    <h1>Dashboard</h1>
    <nav>
      <Link to="stats">Stats</Link>
      <Link to="settings">Settings</Link>
    </nav>
    <Outlet /> {/* Renders child routes here */}
   </div>
  );
}
```

URL Structure:

- /dashboard/stats → Shows <Stats> inside <Dashboard>
- /dashboard/settings → Shows <Settings> inside <Dashboard>

7. 404 Not Found Route

Catch unmatched routes with *.

```
<Route path="*" element={<NotFound />} />
```

Key Takeaways

✓ <BrowserRouter> → Required at the root.

✓ <Route> → Defines URL paths and components.

✓ <Link> / <NavLink> → For navigation (NavLink adds active styles).

✓ useParams() → Access URL parameters.

✓ useNavigate() → Programmatic redirects.

✓ Nested Routes → Organize complex UIs.

Next Steps

- Try building a multi-page blog with dynamic post IDs.
- Explore route guards (protected routes for auth).

Day 28

Context API (State Management)

1. What is Context API?

- Solves prop drilling (passing props through multiple layers).
- Provides a way to share state globally.
- Works well for theme settings, user auth, cart data, etc.

2. Basic Context API Setup

Step 1: Create a Context

```
// ThemeContext.js
import { createContext } from "react";

const ThemeContext = createContext();
export default ThemeContext;
```

Step 2: Provide the Context (Provider)

Wrap components that need access to the context with Provider.

```
// App.jsx
import ThemeContext from "./ThemeContext";

function App() {
  const [theme, setTheme] = useState("light");

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

Step 3: Consume the Context (useContext)

Access context values in child components.

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

function Header() {
  const { theme, setTheme } = useContext(ThemeContext);

  return (
    <header className={theme}>
      <button onClick={() => setTheme(theme === "light" ? "dark" : "light")}>
        Toggle Theme
      </button>
    </header>
  );
}
```

3. Real-World Example: User Authentication

Step 1: Create Auth Context

```
// AuthContext.js
import { createContext, useState } from "react";

const AuthContext = createContext();

export function AuthProvider({ children }) {
  const [user, setUser] = useState(null);

  const login = (userData) => setUser(userData);
  const logout = () => setUser(null);

  return (
```

```
   <AuthContext.Provider value={{ user, login, logout }}>
     {children}
   </AuthContext.Provider>
  );
}
export default AuthContext;
```

Step 2: Wrap App with AuthProvider

```
// main.jsx
import { AuthProvider } from "./AuthContext";

ReactDOM.createRoot(document.getElementById("root")).render(
 <AuthProvider>
   <App />
 </AuthProvider>
);
```

Step 3: Use Auth in Components

```
// LoginButton.jsx
import { useContext } from "react";
import AuthContext from "./AuthContext";
function LoginButton() {
 const { user, login, logout } = useContext(AuthContext);
 return (
   <div>
     {user ? (
       <button onClick={logout}>Logout</button>
     ) : (
       <button onClick={() => login({ name: "John Doe" })}>Login</button>
     )}
   </div>
 );
}
```

4. Optimizing Context Performance

Problem: Context re-renders all consumers when value changes.

Solution: Memoize the context value or split contexts.

Example: Splitting Contexts

```
// Separate Theme and ThemeUpdate contexts
const ThemeContext = createContext("light");
const ThemeUpdateContext = createContext();

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

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

// Now components that only need setTheme won't re-render when theme changes
```

5. When to Use Context vs Redux/Zustand

Context API	Redux/Zustand
Best for small-medium apps	Better for large/complex apps
Built into React	Requires extra library
Can cause re-renders	Optimized for performance

Key Takeaways

✓ createContext → Creates a new context.

✓ Provider → Supplies context to child components.

✓ useContext → Accesses context values.

✓ Best for theme, auth, language settings.

✓ Optimize by splitting contexts or memoization.

Next Steps

- Try building a shopping cart with Context.
- Explore useReducer + Context for complex state logic.

Day 29

Custom Hooks & API Calls

1. What are Custom Hooks?

- Functions that encapsulate reusable logic.
- Follow naming convention: useSomething (e.g., useFetch).
- Can call other hooks (unlike regular functions).

2. Basic Custom Hook Example (useToggle)

A hook to manage boolean state (e.g., toggle buttons, modals).

```
// useToggle.js
import { useState } from "react";
export default function useToggle(initialValue = false) {
  const [value, setValue] = useState(initialValue);
  const toggle = () => setValue(!value);

  return [value, toggle];
}
```

Usage: Toggle Button

```
import useToggle from "./hooks/useToggle";
```

```
function ToggleComponent() {
  const [isOn, toggleIsOn] = useToggle(false);
  return (
    <button onClick={toggleIsOn}>
      {isOn ? "ON" : "OFF"}
    </button>
  );
}
```

3. API Fetching with Custom Hooks (useFetch)

Step 1: Create useFetch Hook

```
// useFetch.js
import { useState, useEffect } from "react";
export default function useFetch(url) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch(url);
        if (!response.ok) throw new Error("Network error");
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };
    fetchData();
  }, [url]); // Re-fetch if URL changes
```

```
  return { data, loading, error };
}
```

Step 2: Use the Hook in a Component

```
function UserList() {
  const { data: users, loading, error } = useFetch(
    "https://jsonplaceholder.typicode.com/users"
  );
  if (loading) return <p>Loading...</p>;
  if (error) return <p>Error: {error}</p>;
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>{user.name}</li>
      ))}
    </ul>
  );
}
```

4. Advanced: useFetch with POST Requests

Extend useFetch to handle POST/PUT/DELETE.

```
// useFetch.js (updated)
export function useFetch(url, options = {}) {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);
  const fetchData = async (overrideOptions = {}) => {
    setLoading(true);
    try {
      const mergedOptions = { ...options, ...overrideOptions };
      const response = await fetch(url, mergedOptions);
      const result = await response.json();
      setData(result);
```

```
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };
  useEffect(() => {
    if (options.method === undefined) {
      fetchData(); // Auto-fetch for GET requests
    }
  }, [url]);
  return { data, loading, error, fetchData }; // Expose fetchData for manual calls
}
```

Usage: POST Request

```
function AddPost() {
  const [title, setTitle] = useState("");
  const { fetchData } = useFetch(
    "https://jsonplaceholder.typicode.com/posts",
    { method: "POST" }
  );
  const handleSubmit = () => {
    fetchData({
      body: JSON.stringify({ title }),
      headers: { "Content-Type": "application/json" },
    });
  };
  return (
    <div>
      <input
        value={title}
        onChange={(e) => setTitle(e.target.value)}
      />
```

```
    <button onClick={handleSubmit}>Submit</button>
   </div>
  );
}
```

5. Real-World Example: useLocalStorage Hook

Persist state in localStorage.

```
// useLocalStorage.js
import { useState, useEffect } from "react";
export default function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const storedValue = localStorage.getItem(key);
    return storedValue ? JSON.parse(storedValue) : initialValue;
  });
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);
  return [value, setValue];
}
```

Usage: Theme Persistence

```
function ThemeToggle() {
  const [theme, setTheme] = useLocalStorage("theme", "light");
  const toggleTheme = () => {
    setTheme(theme === "light" ? "dark" : "light");
  };
  return (
    <button onClick={toggleTheme}>
      Current Theme: {theme}
    </button>
  );
}
```

Key Takeaways

✓ Custom Hooks = Reusable logic containers (useX naming).

✓ useFetch = Abstract API calls into a single hook.

✓ useLocalStorage = Sync state with localStorage.

✓ Dependency Arrays = Control when effects run.

Next Steps

- Build a useDebounce hook for search inputs.
- Explore libraries like React Query for advanced data fetching.

Day 30

Comprehensive React Review & Project

1. Project Structure

```
src/
 ├── components/
|    ├── Task.jsx         # Individual task component
|    ├── TaskList.jsx     # Displays all tasks
|    ├── AddTask.jsx      # Form to add new tasks
|    └── Navbar.jsx       # Navigation
 ├── context/
|    └── TaskContext.js   # Global state management
 ├── hooks/
|    └── useLocalStorage.js # Custom hook
 ├── pages/
|    ├── Home.jsx         # Main task view
|    └── About.jsx        # Static page
└── App.jsx              # Main app with routes
```

2. Key Concepts Implemented

1. JSX & Components (Day 23)

```
// Task.jsx
function Task({ task, onDelete }) {
  return (
    <div className="task">
      <h3>{task.text}</h3>
      <button onClick={() => onDelete(task.id)}>Delete</button>
    </div>
  );
}
```

2. State & Hooks (Day 24)

```
// AddTask.jsx
function AddTask({ onAdd }) {
  const [text, setText] = useState("");
  const handleSubmit = (e) => {
    e.preventDefault();
    if (!text.trim()) return;
    onAdd(text);
    setText("");
  };
  return (
    <form onSubmit={handleSubmit}>
      <input
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Add new task"
      />
    </form>
  );
}
```

3. Lists & Conditional Rendering (Day 25)

```
// TaskList.jsx
function TaskList({ tasks, onDelete }) {
  return (
    <div>
      {tasks.length === 0 ? (
        <p>No tasks yet. Add one!</p>
      ) : (
        tasks.map((task) => (
          <Task key={task.id} task={task} onDelete={onDelete} />
        ))
      )}
    </div>
  );
}
```

4. Forms (Day 26)

(See AddTask.jsx above - fully controlled form)

5. React Router (Day 27)

```
// App.jsx
function App() {
  return (
    <BrowserRouter>
      <Navbar />
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
}
// Navbar.jsx
```

```
function Navbar() {
  return (
    <nav>
      <NavLink to="/" end>Home</NavLink>
      <NavLink to="/about">About</NavLink>
    </nav>
  );
}
```

6. Context API (Day 28)

```
// TaskContext.js
const TaskContext = createContext();
export function TaskProvider({ children }) {
  const [tasks, setTasks] = useState([]);
  const addTask = (text) => {
    setTasks([...tasks, { id: Date.now(), text }]);
  };
  const deleteTask = (id) => {
    setTasks(tasks.filter(task => task.id !== id));
  };
  return (
    <TaskContext.Provider value={{ tasks, addTask, deleteTask }}>
      {children}
    </TaskContext.Provider>
  );
}
export const useTasks = () => useContext(TaskContext);
```

7. Custom Hooks (Day 29)

```
// useLocalStorage.js
function useLocalStorage(key, initialValue) {
  const [value, setValue] = useState(() => {
    const stored = localStorage.getItem(key);
```

```
    return stored ? JSON.parse(stored) : initialValue;
  });
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);
  return [value, setValue];
}
// Updated TaskContext.js to persist tasks
const [tasks, setTasks] = useLocalStorage("tasks", []);
```

3. Putting It All Together

Final Home.jsx

```
function Home() {
  const { tasks, addTask, deleteTask } = useTasks();
  return (
    <div>
      <h1>Task Manager</h1>
      <AddTask onAdd={addTask} />
      <TaskList tasks={tasks} onDelete={deleteTask} />
    </div>
  );
}
```

Wrap App with Providers

```
// main.jsx
ReactDOM.createRoot(document.getElementById('root')).render(
  <TaskProvider>
    <App />
  </TaskProvider>
);
```

4. Key Features Implemented

1. Component Architecture (Task, TaskList, AddTask)

2. State Management (useState + Context API)
3. Persistence (Custom useLocalStorage hook)
4. Routing (Home & About pages)
5. Form Handling (Controlled components)
6. Conditional Rendering (Empty task list message)

5. How to Extend Further

1. Add task editing functionality
2. Implement due dates and filtering
3. Add user authentication
4. Connect to a backend API (replace useLocalStorage)

Final Challenge

Convert this app to use dark/light mode by:

1. Creating a ThemeContext
2. Adding a toggle button in Navbar
3. Storing preference in localStorage