**Lab 10: Implementing Advanced Hooks**

**Using Hooks to Make a Task List (with localStorage)**

You can use the ToDo App from Lab 3 or create a new project.

**Step 1: Setting Up the Context and Reducer**

**How the Hooks will be used in this application:**

**useReducer Hook**:

Manages the state of the to-do list with actions (ADD\_TODO, TOGGLE\_TODO, REMOVE\_TODO).

Provides a dispatch function to send actions to the reducer.

# useContext Hook:

Accesses the to-do list state and dispatch function from the context in the TodoList component.

# useRef Hook:

Creates a reference to the input element to get its value and reset it after adding a new to-do.

# useEffect Hook:

Logs the updated to-do list to the console whenever the state changes.

This breakdown should help you understand how the different parts of the application work together and how various hooks are used to manage state and effects in a functional component-based approach.

# TodoContext.js

* + First, let's create a context for our to-do application. This context will allow us to share the to-do list state and dispatch function across different components.

import React from 'react';

const TodoContext = React.createContext(); export default TodoContext;

# todoReducer.js

* + Next, we create a reducer function to manage the state of our to-do list. The reducer function will handle adding, toggling, and removing to-dos.

const todoReducer = (state, action) =>

{ switch (action.type) { case 'ADD\_TODO':

return [...state, { id: Date.now(), text:

action.payload, completed: false }]; case 'TOGGLE\_TODO':

return state.map(todo =>

todo.id === action.payload ? { ...todo, completed: !todo.completed } : todo

);

case 'REMOVE\_TODO':

return state.filter(todo => todo.id !== action.payload);

default:

return state;

}

};

export default todoReducer;

**Step 2: Wrap the Context around the main app.**

**App.js or App.jsx**

* + In the main application component, we will use the useReducer hook to manage the to-do list state and provide it to the rest of the application using the TodoContext.Provider.

import React, { useReducer, useRef, useEffect } from 'react';

import TodoContext from './TodoContext'; import todoReducer from './todoReducer'; import TodoList from './TodoList'; import './App.css';

const App = () => {

const [state, dispatch] = useReducer(todoReducer, [], () => {

const localData = localStorage.getItem('todos'); return localData ? JSON.parse(localData) : [];

});

const inputRef = useRef(null); const handleAddTodo = () => {

if (inputRef.current.value)

{ dispatch({ type: 'ADD\_TODO', payload:

inputRef.current.value }); inputRef.current.value = '';

}

};

useEffect(() =>

{ localStorage.setItem('todos', JSON.stringify(state));

}, [state]);

return (

<TodoContext.Provider value={{ state, dispatch }}>

<div className="App">

<h1>Todo List</h1>

<input ref={inputRef} type="text" placeholder="Add a new task" />

<button onClick={handleAddTodo}>Add</button>

<TodoList />

</div>

</TodoContext.Provider>

);

};

export default App;

App.css:

.App {

font-family: 'Arial', sans-serif; background-color: #f7f7f7; padding: 20px;

max-width: 500px; margin: 40px auto; border-radius: 10px;

box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1); text-align: center;

}

/\* Title styling \*/ h1 {

color: #333;

margin-bottom: 20px;

}

/\* Input field styling \*/ input[type="text"] {

width: 70%; padding: 10px; margin-right: 10px;

border: 1px solid #ddd; border-radius: 5px;

box-shadow: inset 0 1px 3px rgba(0, 0, 0, 0.1);

}

/\* Button styling \*/ button {

padding: 10px 20px; border: none; border-radius: 5px;

background-color: #007bff; color: white;

font-size: 14px; cursor: pointer;

transition: background-color 0.3s ease;

}

button:hover {

background-color: #0056b3;

}

/\* List styling \*/ ul {

list-style: none; padding: 0;

}

li {

background-color: white; margin: 10px 0; padding: 15px;

border-radius: 5px;

box-shadow: 0 1px 3px rgba(0, 0, 0, 0.1); display: flex;

justify-content: space-between; align-items: center;

}

/\* Completed task styling \*/ li.completed {

text-decoration: line-through; color: #888;

}

/\* Individual button styling within list items \*/ li button {

background-color: transparent; border: none;

cursor: pointer; color: #007bff; margin-left: 10px;

}

li button:hover

{ color: #0056b3;

}

# Step 3: Creating the TodoList Component TodoList.js

* The TodoList component will consume the context and render the list of to- dos. It will use the useContext and useEffect hooks.

import React, { useContext, useEffect } from 'react'; import TodoContext from './TodoContext';

import TodoItem from './TodoItem';

const TodoList = () => {

// Use context to get state and dispatch

const { state, dispatch } = useContext(TodoContext);

// Log state changes useEffect(() => {

console.log('Todo list updated:', state);

}, [state]);

return (

<ul>

{state.map(todo => (

<TodoItem key={todo.id} todo={todo} dispatch={dispatch} />

))}

</ul>

);

};

export default TodoList;

# Step 4: Creating the TodoItem Component

**TodoItem.js**

* + The TodoItem component represents a single to-do item and provides buttons to toggle its completion status or remove it.

import React from 'react';

const TodoItem = ({ todo, dispatch }) =>

{ return (

<li style={{ textDecoration: todo.completed ? 'line-through' : 'none' }}>

{todo.text}

<button onClick={() => dispatch({ type: 'TOGGLE\_TODO', payload: todo.id })}>

{todo.completed ? 'Undo' : 'Complete'}

</button>

<button onClick={() => dispatch({ type: 'REMOVE\_TODO', payload: todo.id })}>

Remove

</button>

</li>

);

};

export default TodoItem;

# Putting It All Together

Finally, ensure all components and context are imported and used correctly in your project.

**App.js**: The main entry point where the TodoContext.Provider wraps the entire application.

**TodoList.js**: Consumes the context to display the list of to-dos and uses useEffect to log changes.

**TodoItem.js**: Represents individual to-do items and handles toggling and removing to-dos.

**TodoContext.js**: Provides a context for sharing the to-do list state and dispatch function.

**todoReducer.js**: Contains the reducer function to manage the to-do list state.

# Running the Application

To run the application, use the following command in your project directory:

npm run dev

Lab11:CreateCustom **Hooks**

**Step 1: Create Hooks for Popular Functionality**

**useFetch Hook**:

* + - * Fetches data from a given URL.
      * Manages loading and error states.
      * Returns the fetched data, loading state, and any errors encountered.

# useSort Hook:

* + - * Sorts data based on a provided key and direction.
      * Allows toggling between ascending and descending order.
      * Returns the sorted data and functions to request a sort.

# useSearch Hook:

* + - * Filters data based on a search query.
      * Matches the query against all string fields in the data objects.
      * Returns the filtered data.

# Integrate the custom hooks into ProductList Component:

* + - * Fetch products using useFetch.
      * Sort products using useSort.
      * Filter products using useSearch.
      * Provide sorting and search functionalities via buttons and input fields.

1. **useFetch Hook**

Create a custom hook for data fetching:

//src/hooks/useFetch.js

import { useState, useEffect } from 'react';

const useFetch = (url) => {

const [data, setData] = useState([]);

const [loading, setLoading] = useState(true); const [error, setError] = useState(null);

useEffect(() => {

const fetchData = async () => { try {

const response = await fetch(url); const result = await response.json(); setData(result);

} catch (error)

{ setError(error);

} finally

{ setLoading(false);

}

};

fetchData();

}, [url]);

return { data, loading, error };

};

export default useFetch;

# useSort Hook

Create a custom hook for sorting data:

//src/hooks/useSort.js

import { useState, useEffect } from 'react';

const useSort = (data, config = null) => {

const [sortedData, setSortedData] = useState([]);

const [sortConfig, setSortConfig] = useState(config);

useEffect(() => {

let sortableData = [...data];

if (sortConfig !== null) {

sortableData.sort((a, b) => {

if (a[sortConfig.key] < b[sortConfig.key]) {

return sortConfig.direction === 'ascending' ? -1 : 1;

}

if (a[sortConfig.key] > b[sortConfig.key]) {

return sortConfig.direction === 'ascending' ? 1 : -1;

}

return 0;

});

}

setSortedData(sortableData);

}, [data, sortConfig]);

const requestSort = (key) => {

let direction = 'ascending';

if (sortConfig && sortConfig.key === key && sortConfig.direction === 'ascending') {

direction = 'descending';

}

setSortConfig({ key, direction });

};

return { sortedData, requestSort, sortConfig };

};

export default useSort;

# 3. useSearch Hook

Create a custom hook for searching data:

//src/hooks/useSearch.js

import { useState, useEffect } from 'react'; const useSearch = (data, query) => {

const [filteredData, setFilteredData] = useState([]);

useEffect(() => { if (!query) {

setFilteredData(data);

} else {

const lowercasedQuery = query.toLowerCase(); const filtered = data.filter(item =>

Object.values(item).some(value =>

value.toString().toLowerCase().includes(lowercasedQuery

)

)

);

setFilteredData(filtered);

}

}, [data, query]);

return filteredData;

};

export default useSearch;

# Step 2: Integrate Custom Hooks into Components

# 1. ProductList.js

Update ProductList to use the custom hooks:

//src/components/ProductList.js

import React, { useState } from 'react'; import useFetch from '../hooks/useFetch'; import useSort from '../hooks/useSort'; import useSearch from '../hooks/useSearch';

function ProductList({ onSelectProduct })

{ const { data: products, loading, error } = [useFetch('http://localhost:5000/products');](http://localhost:5000/products')%25253B)

const { sortedData, requestSort, sortConfig } =

useSort(products);

const [searchQuery, setSearchQuery] = useState(''); const filteredData = useSearch(sortedData,

searchQuery);

if (loading) return <p>Loading...</p>;

if (error) return <p>Error: {error.message}</p>;

const getClassNamesFor = (name) =>

{ if (!sortConfig) return; return sortConfig.key === name ?

sortConfig.direction : undefined;

};

return (

<div>

<h1>Product List</h1>

<input

type="text" placeholder="Search..." value={searchQuery} onChange={(e) =>

setSearchQuery(e.target.value)}

/>

<button type="button"

onClick={() => requestSort('name')} className={getClassNamesFor('name')}

>

Sort by Name

</button>

<ul>

{filteredData.map((product) => (

<li key={product.id}>

<div className="product-details" onClick={() => onSelectProduct(product)}>

{product.name}

</div>

<div className="product-actions">

<button onClick={() => onSelectProduct(product)}>Edit</button>

<button onClick={() => handleDelete(product.id)}>Delete</button>

</div>

</li>

))}

</ul>

</div>

);

}

export default ProductList;

# Lab 12: Exploring Optimization

# Part 1: Update App Component with Lazy Loading and Suspense

React's lazy and Suspense are used together to dynamically load components when they are needed, reducing the initial load time of the application.

# Update App.js to lazy load Product components

//src/App.js

import React, { useState, Suspense, lazy } from 'react';

import './App.css';

// Lazy load components

const ProductList = lazy(() => import('./components/ ProductList'));

const ProductForm = lazy(() => import('./components/ ProductForm'));

const Product = lazy(() => import('./components/ Product'));

function App() {

const [selectedProduct, setSelectedProduct] = useState(null);

const [isAdding, setIsAdding] = useState(false);

const handleSelectProduct = (product) =>

{ setSelectedProduct(product);

};

const handleBack = () =>

{ setSelectedProduct(null); setIsAdding(false);

};

const handleAddProduct = (product) =>

{ setIsAdding(false);

setSelectedProduct(null); // Clear selected product

};

const handleStartAdding = () =>

{ setIsAdding(true);

};

return (

<div className="App">

<Suspense fallback={<div>Loading...</div>}>

{isAdding ? (

<ProductForm onAddProduct={handleAddProduct} onCancel={handleBack} />

) : selectedProduct ? (

<Product product={selectedProduct} onBack={handleBack} />

) : (

<>

<ProductList onSelectProduct={handleSelectProduct} />

<button onClick={handleStartAdding}>Add Product</button>

</>

)}

</Suspense>

</div>

);

}

export default App;

## Part 2: Explore styled-components

## Benefits of using Styled-Components:

1. **Scoped Styles**
   * **Styled-Components** allow you to scope your styles to individual components, meaning the styles you define are automatically applied only to that specific component, preventing conflicts between styles across the application.

## Dynamic Styling

* + You can pass props to styled components and use those props to dynamically change the styles. This is particularly useful for building reusable components that adapt to different contexts (e.g., color changes based on props).

## Example with props:

const Button = styled.button`

background-color: ${props => props.primary ? 'blue' : 'gray'};

color: white; padding: 10px 20px; border: none; border-radius: 5px;

`;

<Button primary={true}>Primary Button</Button>

<Button primary={false}>Secondary Button</Button>

## No CSS Class Name Collisions

* + Since styles are defined within JavaScript, the class names for elements are automatically generated with unique identifiers to avoid any clashes between global CSS rules. You don’t have to worry about naming collisions or dealing with specific class selectors.

## Easier Maintenance

* + By keeping styles with the components they belong to, Styled-Components makes it easier to maintain and scale your application. The styling for each component is self- contained, and you don’t have to jump between different CSS files.

## Theming Support

* + Styled-Components support *theming*, which allows you to define a set of consistent colors, fonts, and spacing for your entire app. You can easily switch between themes (e.g., light/dark mode) by changing theme values in one place.

## Theming:

// Define theme const theme = {

primaryColor: 'blue', secondaryColor: 'gray',

};

// Apply theme using ThemeProvider

import { ThemeProvider } from 'styled-components';

function App() { return (

<ThemeProvider theme={theme}>

<Button primary={true}>Primary Button</Button>

</ThemeProvider>

);

}

## Performance Optimization

* + Styled-Components optimizes performance by only injecting the styles that are actually used in the page. It also supports server-side rendering, which helps with faster page loads and better SEO.

## No Need for External Stylesheets

* + Since styles are part of the JavaScript file, there’s no need to manage external CSS files. This reduces dependencies and keeps everything in one place, simplifying the overall project structure.

## Powerful Features

* + **Styled-Components** comes with a variety of advanced features, such as:
    - Nesting: You can nest your styles like you would in SCSS.
    - Mixins: Reusable sets of styles.
    - MediaQueries: Dynamically add responsive design within your components.

## Step 1: Install dependencies

If you haven’t already installed Styled-Components, install the library in your project:

npm install styled-components

## Step 2: Create a React component with styled-components The code:

* styled.div creates a styled div element.
* styled.button creates a styled button element.
* styled.h1 creates a styled h1 element.
* We use CSS-in-JS syntax directly inside the styled-components to define the styles for each component.
* In the Button, we use the &:hover selector to define the hover state.

// App.js

import React from 'react';

import styled from 'styled-components';

// Define styled components const Container = styled.div`

display: flex;

justify-content: center; align-items: center; height: 100vh;

background-color: #f0f0f0;

`;

const Button = styled.button` background-color: #3498db; color: white;

padding: 10px 20px; border: none; border-radius: 5px; font-size: 16px; cursor: pointer;

&:hover {

background-color: #2980b9;

}

`;

const Heading = styled.h1` font-size: 36px;

color: #2c3e50;

`;

// App component function App() {

return (

<Container>

<div>

<Heading>Welcome to React with Styled Components</

Heading>

<Button onClick={() => alert('Button clicked!')}

>Click Me</Button>

</div>

</Container>

);

}

export default App;

## Step 3: Run the app

Run your app with npm run dev

Look at your React app styled with Styled-Components!

**Styled-Components** enable **CSS-in-JS**—writing CSS directly inside your JavaScript files. In React, components and styles can be tightly coupled.

# Add a theme:

## Step 1: Define the Theme

First, create a theme object that contains the values you want to apply globally, like colors, fonts, etc.

// src/theme.js

export const theme = { colors: {

primary: '#3498db', secondary: '#2c3e50', background: '#f0f0f0', button: '#2980b9', buttonHover: '#1d6fa5',

},

fonts: {

main: 'Arial, sans-serif', heading: 'Georgia, serif',

},

};

**Step 2: Use ThemeProvider**

Next, import ThemeProvider from **styled-components** and wrap your entire application with it. This makes the theme available to all styled components within your app.

// src/App.jsx

import React from 'react';

import styled, { ThemeProvider } from 'styled-components'; import { theme } from './theme'; // Import the theme

// Define styled components const Container = styled.div`

display: flex;

justify-content: center; align-items: center; height: 100vh;

background-color: ${(props) => props.theme.colors.background};

`;

const Button = styled.button` background-color: ${(props) =>

props.theme.colors.primary}; color: white;

padding: 10px 20px; border: none; border-radius: 5px; font-size: 16px; cursor: pointer;

&:hover {

background-color: ${(props) => props.theme.colors.buttonHover};

}

`;

const Heading = styled.h1` font-size: 36px;

color: ${(props) => props.theme.colors.secondary}; font-family: ${(props) => props.theme.fonts.heading};

`;

function App() { return (

<ThemeProvider theme={theme}>

<Container>

<div>

<Heading>Welcome to Vite + React + Styled Components</Heading>

<Button onClick={() => alert('Button clicked!')}

>Click Me</Button>

</div>

</Container>

</ThemeProvider>

);

}

export default App;

## Step 3: Run Your App

Start the Vite development server again to see the theme applied:

npm run dev

Now, your application is using the theme values globally. For example, you can change the primary button color or the background color of the entire app by modifying the theme.js

file, and the changes will automatically be reflected in all components that use the theme.

**Benefits of Using ThemeProvider:**

1. CentralizedManagement: All styles dependent on the theme (like colors, fonts, spacings, etc.) are managed in one place, making it easy to update.
2. GlobalAccess: With ThemeProvider, all the styled components can access the theme without needing to pass props down manually, which keeps your code cleaner.
3. Flexibility: You can easily switch themes or modify styles dynamically by changing the theme object.

# Next: Switch Themes Dynamically

If you want to switch between themes dynamically (e.g., toggle between light and dark mode), you can modify the theme using state and update it at runtime. Here's a basic way to switch between two themes:

// src/App.jsx

import React, { useState } from 'react';

import styled, { ThemeProvider } from 'styled-components'; import { theme, darkTheme } from './theme'; // Assuming darkTheme is defined

// Define styled components (same as before) const Container = styled.div`

display: flex;

justify-content: center; align-items: center; height: 100vh;

background-color: ${(props) => props.theme.colors.background};

`;

const Button = styled.button` background-color: ${(props) =>

props.theme.colors.primary}; color: white;

padding: 10px 20px; border: none; border-radius: 5px; font-size: 16px; cursor: pointer;

&:hover {

background-color: ${(props) => props.theme.colors.buttonHover};

}

`;

const Heading = styled.h1` font-size: 36px;

color: ${(props) => props.theme.colors.secondary}; font-family: ${(props) => props.theme.fonts.heading};

`;

function App() {

const [isDarkMode, setIsDarkMode] = useState(false);

return (

<ThemeProvider theme={isDarkMode ? darkTheme : theme}>

<Container>

<div>

<Heading>Welcome to Vite + React + Styled Components</Heading>

<Button onClick={() => setIsDarkMode(! isDarkMode)}>

Switch Theme

</Button>

</div>

</Container>

</ThemeProvider>

);

}

export default App;

**Add darkTheme to theme.js:**

//add to src/theme.js export const darkTheme = {

colors: {

primary: '#2c3e50', secondary: '#ecf0f1', background: '#34495e', button: '#2980b9', buttonHover: '#1d6fa5',

},

fonts: {

main: 'Arial, sans-serif', heading: 'Georgia, serif',

},

};

The button toggles between the light and dark theme based on the state.

# Part 3: More Vite Configuration for React

**1. vite.config.js**

When you create a React project with Vite, your vite.config.js (or .ts) typically looks like the following:

* @vitejs/plugin-react enables JSX support and fast refresh.
* defineConfig provides TypeScript support and better IDE suggestions.

import { defineConfig } from 'vite'; import react from '@vitejs/plugin-react';

export default defineConfig({ plugins: [react()],

});

# Configuring Aliases

You can define path aliases to simplify imports:

import { defineConfig } from 'vite'; import react from '@vitejs/plugin-react'; import path from 'path';

export default defineConfig({ plugins: [react()], **resolve: {**

**alias: {**

**'@': path.resolve( dirname, './src'),**

**},**

**},**

});

# Review: Customizing the Development Server

Vite allows you to configure the development server (vite dev) using the server option:

* + port: Sets the port for the local server.
  + open: Automatically opens the browser.

export default defineConfig({

**server: {**

**port: 3000, // Change the dev server port open: true, // Opens browser in dev server**

**},**

});

# Optimizing Build with build Options

To customize the production build:

* + outDir: Specifies the output folder for vite build.
  + sourcemap: Enables source maps for debugging production code.
  + chunkSizeWarningLimit: Adjusts the warning threshold for large chunks.

export default defineConfig({

**build: {**

**outDir: 'prod', // Change output directory sourcemap: true, // Generate source maps chunkSizeWarningLimit: 500, // Increase chunk size**

**limit warning**

**},**

});

# CSS & PostCSS Configuration

Vite allows you to configure CSS pre-processors:

* + This automatically imports a global SCSS file into all components.

export default defineConfig({ css: {

preprocessorOptions: { scss: {

additionalData: `@import "@/styles/global.scss";`,

},

},

},

});

# Configuring Plugins

Vite supports plugins to extend its functionality. Some useful ones for React:

import react from '@vitejs/plugin-react';

**import svgr from 'vite-plugin-svgr';**

export default defineConfig({ plugins: [

react(),

**svgr(), // Enables importing SVGs as React components**

],

});

Now, you can import SVGs like this:

import { ReactComponent as Logo } from '@/assets/logo.svg';

# Using TypeScript in Vite

For TypeScript support, ensure you have vite.config.ts:

* + esbuild.jsxInject automatically imports React in .tsx files, so you don't need to manually import it.

import { defineConfig } from 'vite'; import react from '@vitejs/plugin-react';

export default defineConfig({ plugins: [react()], esbuild: {

jsxInject: `import React from 'react'`,

},

});

# Lab 13: TSX with SCSS

Part 1: Using TypeScript and SASS

* + Move your files into it from the React-REST Axios lab.

# Setting Up TypeScript Types for Objects and Props

* + TypeScript provides statictyping, which prevents errors before running the app.
  + It improves developerexperienceby offering better autocompletion.
  + It makes your codebase easierto **maintain**.

**1: Define a Product Type**

You can add this interface under the imports in Products.tsx, or create a types/ directory for all types:

/src

/types

Product.ts

**Create a TypeScript interface inside src/Product.tsx or src/types/ Product.ts. If you put it in a separate file, you will import it with import**

**{ Product } from '../types/Product';**

export interface Product {

id: string; // Unique identifier name: string; // Product name

description: string; // Short text about the product price: string; // Can store numbers as strings for easy

formatting

stock: number; // Number of items available

}

**Benefits:**

* We define id as a string because it may come from databases like MongoDB (which uses string \_id).
* price is a string to avoid floating point issues (e.g., "10.99" instead of

10.99).

* stock is a number because it’s an integer.

### **2: Define Props for Each Component**

Each component should **only accept properly typed props** to ensure consistency.

**Product.tsx**

import { Product } from '../types/Product'; interface ProductProps {

product: Product; // This component receives a single

product

onBack: () => void; // Function prop with no arguments, used for navigation

}

**ProductForm.tsx**

import { Product } from '../types/Product'; interface ProductFormProps {

onAddProduct: (product: Product) => void; // Function

that receives a new product

onCancel: () => void; // Function to cancel adding a product

}

**ProductList.tsx**

import { Product } from '../types/Product'; interface ProductListProps {

onSelectProduct: (product: Product) => void; // Function

to select a product

}

**Benefits:**

* Helps documenteach component’s expected inputs.
* Prevents passing incorrect types, such as a number when a string is expected.

1. **Converting App.jsx to App.tsx**

### **Benefits:**

* + .tsx files support TypeScript+JSX **syntax**.
  + Helps ensure all React components receive correctly typed props.

**Step 3: Modify App.tsx**

// src/App.tsx

import React, { useState } from 'react';

import ProductList from './components/ProductList'; import Product from './components/Product';

import ProductForm from './components/ProductForm'; import { Product as ProductType } from './types/ Product'; // Rename `Product` to avoid conflicts import './App.scss';

const App: React.FC = () => {

// State for selected product

const [selectedProduct, setSelectedProduct] = useState<ProductType | null>(null);

// State to track if the form is open

const [isAdding, setIsAdding] = useState<boolean>(false);

// Function to handle selecting a product

const handleSelectProduct = (product: ProductType) => { setSelectedProduct(product);

};

// Function to go back to the product list const handleBack = () => {

setSelectedProduct(null); setIsAdding(false);

};

// Function to handle adding a new product

const handleAddProduct = (product: ProductType) => { setIsAdding(false);

setSelectedProduct(null);

};

return (

<div className="App">

{isAdding ? (

<ProductForm onAddProduct={handleAddProduct} onCancel={handleBack} />

) : selectedProduct ? (

<Product product={selectedProduct} onBack={handleBack} />

) : (

<>

<ProductList onSelectProduct={handleSelectProduct} />

<button onClick={() => setIsAdding(true)}>Add Product</button>

</>

)}

</div>

);

};

export default App;

### **We changed:**

**Typed useState<ProductType | null>**

* + Ensures selectedProduct is either null or a Product.
  + Prevents errors like Cannot read properties of null.

**Function Prop Type Consistency**

* + handleSelectProduct accepts a ProductType to match

ProductListProps.

**Type-Safe isAdding State**

* useState<boolean>(false) ensures it always holds a boolean.

1. **Converting Product.jsx to Product.tsx**

### **Benefits:**

* + Ensure the product editing component onlyworkswithvalid **products**.

**Step 4: Modify Product.tsx**

// src/components/Product.tsx

import React, { useState } from 'react'; import axios from 'axios';

import { Product as ProductType } from '../types/Product';

interface ProductProps { product: ProductType; onBack: () => void;

}

const Product: React.FC<ProductProps> = ({ product, onBack }) => {

const [productData, setProductData] = useState<ProductType>(product);

// Handle input changes safely const handleChange = (e:

React.ChangeEvent<HTMLInputElement>) => { const { name, value } = e.target; setProductData((prevState) => ({

...prevState,

[name]: value,

}));

};

// Handle saving to API

const handleSave = async () => { try {

const response = await axios.put(`http:// localhost:5000/products/${productData.id}`, productData);

console.log('Product updated:', response.data); onBack(); // Navigate back after saving

} catch (error) {

console.error('Error updating product:', error);

}

};

return (

<div className="form-container">

<h2>Edit Product</h2>

<form>

<label>Name:</label>

<input type="text" name="name" value={productData.name} onChange={handleChange} />

<label>Description:</label>

<input type="text" name="description" value={productData.description} onChange={handleChange} />

<label>Price:</label>

<input type="text" name="price" value={productData.price} onChange={handleChange} />

<label>Stock:</label>

<input type="number" name="stock" value={productData.stock} onChange={handleChange} />

<button type="button" onClick={handleSave}>Save</

button>

<button type="button" onClick={onBack}>Back to

list</button>

</form>

</div>

);

};

export default Product;

### **We changed:**

**Typed useState<ProductType>**

* + Ensures productData follows the expected structure.
  + Prevents issues like undefined properties.

**Typed Event Handlers (React.ChangeEvent<HTMLInputElement>)**

* + Ensures handleChange only processes valid input elements.

**Error Handling in handleSave**

* + Catches API failures so the app doesn’t break.

# Next:

* + **Convert ProductForm.tsx** to ensure form inputs are type-safe.
  + **Convert ProductList.tsx** to correctly fetch products with TypeScript.

# Converting CSS to SCSS:

**Let’s convert App.css to App.scss**

This file makes use of **SCSS features** like:

* + **Variables ($variable-name)** – Store reusable values.
  + **Nesting (& and children selectors)** – Make styling more readable.
  + **Ampersand (&) for Parent Referencing** – Modify styles within the same block.
  + **Transitions & Pseudo-classes (:hover, :focus, :active)** – Add animations.
  + **Optional: SCSS Mixins & Extends** – Would help reduce repetition.

## Variables in SCSS

$primary-color: #3498db;

$secondary-color: #2ecc71;

$font-family: 'Arial', sans-serif;

$border-radius: 8px;

* + **What SCSS is doing:**
    - It defines **variables** that can be reused throughout the file.
    - Instead of hardcoding values like #3498db everywhere, we store them in variables ($primary-color).
    - This makes it **easy to change styles across the whole app** by modifying a single value.

## Nesting in SCSS (.App example)

.App {

display: flex;

flex-direction: column; justify-content: center; align-items: center; height: 100vh;

background-color: #f8f8f8; font-family: $font-family; color: #333;

transition: background-color 0.3s ease;

&.dark-mode {

background-color: #2c3e50; color: white;

}

}

* **What SCSS is doing:**
  + .App is the **parent selector**.
  + Inside .App, we use &.dark-mode:
    - The & refersto **.App itself**.

This results in:

.App.dark-mode {

background-color: #2c3e50;

color: white;

}

* + This makes it **easier to modify states** like dark mode inside .App without repeating selectors.

1. **Using & for Pseudo-classes (button example)**

button {

padding: 10px 20px; font-size: 16px;

background-color: $primary-color; color: white;

border: none;

border-radius: $border-radius; cursor: pointer;

transition: all 0.3s ease;

&:hover {

background-color: $secondary-color; transform: translateY(-3px);

}

&:active {

transform: translateY(1px);

}

&:focus { outline: none;

}

}

* **What SCSS is doing:**
  + SCSS **nests pseudo-classes (:hover, :active, :focus) inside button**. The &:hover is the same as writing:

button:hover {

background-color: #2ecc71; transform: translateY(-3px);

}

* + The & keeps it **scoped within button**, so we don't need to repeat button

everywhere.

* + The hover and active states **add animations and interactivity**.

1. **Styling .form-container with SCSS Nesting**

.form-container { background-color: white; padding: 20px;

border-radius: $border-radius;

box-shadow: 0 4px 12px rgba(0, 0, 0, 0.1); max-width: 400px;

width: 100%;

transition: transform 0.3s ease;

&:hover {

transform: scale(1.03);

}

label {

display: block; margin: 10px 0 5px; font-weight: bold;

}

input[type='text'], input[type='number'] {

width: 100%; padding: 10px;

border: 1px solid #ddd;

border-radius: $border-radius; margin-bottom: 15px; transition: all 0.3s ease;

&:focus {

border-color: $primary-color;

box-shadow: 0 0 5px rgba(52, 152, 219, 0.5);

}

}

}

* **What SCSS is doing:**
  + .form-container has:
    - **A background color** (white).
    - **A box shadow** (for a 3D effect).
    - **A transform animation** when hovered (scale(1.03)).
  + Inside .form-container, SCSS **nests label and input**:
    - This means the label and input styles **only apply inside .form- container**.
    - The input fields change border color and glow when focused.

1. **Button Styling Inside .form-container**

button { width: 100%;

margin-top: 15px;

background-color: $secondary-color;

}

* + **What SCSS is doing:**
    - Since this button is **inside .form-container**, it **only affects buttons within forms**.
    - It makes the button full-width (width: 100%) and uses the green color ($secondary-color).
    - This prevents accidental styling of other buttons.

## SCSS Advantages Over Regular CSS

### **Better Readability with Nesting**

Instead of writing:

.form-container label { display: block;

}

.form-container input[type="text"] { width: 100%;

}

* + SCSS neststheserules, making it more readable.

### **Easy Theme Changes with Variables**

* If we wanted to change the theme color, we just update:

$primary-color: red;

* + This updates allprimarycolorelementsautomatically.

### **Scoped Selectors Using &**

* Instead of writing:

.App.dark-mode { background-color: black; }

* + SCSS allows us to keep **.dark-mode** inside **.App** using &.

## How SCSS Compiles to CSS

The SCSS:

/\*scss\*/

button {

background-color: $primary-color;

&:hover {

background-color: $secondary-color;

}

}

Compiles into standard CSS:

/\* CSS\*/

button {

background-color: #3498db;

}

button:hover {

background-color: #2ecc71;

}

* The SCSS simplifiesorganizationwhile keeping CSS **lightweight**.

**SCSS Features Used in App.scss**

|  |  |
| --- | --- |
| **Feature** | **What It Does** |
| **Variables ($var)** | Store reusable colors, fonts, and sizes. |
| **Nesting (selector { child {})** | Keep related styles together. |
| **Parent Referencing (&)** | Modify styles based on state changes  (e.g., :hover). |
| **Pseudo-classes (:hover, :focus)** | Add interactivity. |
| **Transitions (transition: all 0.3s ease;)** | Smooth animations. |

# Lab 14: Manage State

# Part 1: Redux

# Project structure:

todo-app/

├── public/

│ └── index.html

├── src/

│ ├── App.jsx

│ ├── App.css

│ ├── main.jsx

│ ├── redux/

│ │ ├── store.js

│ │ └── todosSlice.js

│ └── index.css

├── package.json

└── vite.config.js

Let's convert a todo list to a modern React app with Redux.

**Step 1: Install Redux and React-Redux in the project:**

# npm install @reduxjs/toolkit react-redux

**Step 2: Set Up Redux Store**

1. CreateaReduxslicefor managing the to-do list. In your project, create a new folder src/redux and inside it create a file todosSlice.js:

# A Redux slice:

In Redux, a slice refers to a portion of the Redux state along with the actions that can modify that part of the state. For example, a "todos" slice might handle tasks like adding, removing, or toggling tasks in a to-do list.

**Simplify Redux setup with createSlice**

Instead of manually defining action types, action creators, and reducers, createSlice

bundles them into one step.

**createSlice gives you:**

* Actioncreators: These are functions that automatically generate action objects for you, like addTodo() or removeTodo().
* Actiontypes: The action types are automatically derived from the slice name and the

reducer function names. For example, the action type for addTodo would be todos/ addTodo.

* Reducerfunction: A reducer function is created automatically that handles the state

changes based on the dispatched actions.

// src/redux/todosSlice.js

import { createSlice } from '@reduxjs/toolkit';

const todosSlice = createSlice({ name: 'todos',

initialState: [], reducers: {

addTodo: (state, action) => { state.push({

id: Date.now(),

text: action.payload,

});

},

removeTodo: (state, action) => {

return state.filter((todo) => todo.id !== action.payload);

},

},

});

export const { addTodo, removeTodo } = todosSlice.actions; export default todosSlice.reducer;

1. CreatetheReduxStore: In the src/redux folder, create store.js:

// src/redux/store.js

import { configureStore } from '@reduxjs/toolkit'; import todosReducer from './todosSlice';

export const store = configureStore({ reducer: {

todos: todosReducer,

},

});

**Wrap Your App with Redux Provider**: In src/main.jsx, wrap your App component with

Provider to give Redux access to the entire app.

// src/main.jsx

import React from 'react';

import ReactDOM from 'react-dom/client'; import './index.css';

import App from './App';

import { Provider } from 'react-redux'; import { store } from './redux/store';

ReactDOM.createRoot(document.getElementById('root')).render (

<Provider store={store}>

<App />

</Provider>

);

# Step 3: Set Up the App Component

Now, let's move the logic into the App.jsx file using React hooks and Redux.

1. AppComponent: In src/App.jsx, set up the component with Redux state management and React hooks.

// src/App.jsx

import React, { useState } from 'react';

import { useDispatch, useSelector } from 'react-redux'; import { addTodo, removeTodo } from './redux/todosSlice'; import './App.css';

const App = () => {

const [input, setInput] = useState(''); const dispatch = useDispatch();

const todos = useSelector((state) => state.todos);

const handleAddTodo = () => { if (input.trim()) {

dispatch(addTodo(input.trim())); setInput('');

}

};

const handleRemoveTodo = (id) => { dispatch(removeTodo(id));

};

return (

<div className="container">

<h1>To-Do List</h1>

<div className="input-container">

<input

type="text" id="todo-input" value={input}

onChange={(e) => setInput(e.target.value)} placeholder="Add a new task..."

/>

<button id="add-btn" onClick={handleAddTodo}> Add

</button>

</div>

<ul id="todo-list">

{todos.map((todo) => (

<li key={todo.id} className="todo-item">

<span>{todo.text}</span>

<button onClick={() => handleRemoveTodo(todo.id)}>Delete</button>

</li>

))}

</ul>

</div>

);

};

export default App;

# Step 4: Add the CSS Code

You can use the same css file and add it to the project.

**CSS File**: In the src folder, App.css looks like:

/\* src/App.css \*/ body {

font-family: Arial, sans-serif; display: flex;

justify-content: center; align-items: center; height: 100vh;

margin: 0;

background-color: #f0f0f0;

}

.container {

background-color: white; padding: 20px;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1); width: 300px;

text-align: center;

}

.input-container { display: flex; margin-bottom: 20px;

}

#todo-input { flex: 1;

padding: 10px;

border: 1px solid #ccc; border-radius: 4px 0 0 4px;

}

#add-btn { padding: 10px; border: none;

background-color: #28a745; color: white;

cursor: pointer;

border-radius: 0 4px 4px 0;

}

#add-btn:hover {

background-color: #218838;

}

#todo-list {

list-style: none; padding: 0;

}

.todo-item { display: flex;

justify-content: space-between; align-items: center;

padding: 10px;

border: 1px solid #ccc; border-radius: 4px; margin-bottom: 10px;

}

.todo-item button { background-color: #dc3545; color: white;

border: none; padding: 5px 10px; cursor: pointer; border-radius: 4px;

}

.todo-item button:hover { background-color: #c82333;

}

# Step 5: Create Main App Component

1. **Update App.js**:

Add the following code to define the App component:

import React, { Component } from 'react'; import AddTodo from './components/AddTodo'; import TodoList from './components/TodoList';

class App extends Component

{ render() {

return (

<div className="container">

<h1>To-Do List</h1>

<AddTodo />

<TodoList />

</div>

);

}

}

export default App;

# Step 6: Create the Entry Point

1. **Update index.js**:

Add the following code to render the App component and set up the Redux store:

import React from 'react';

import ReactDOM from 'react-dom'; import { Provider } from 'react-redux'; import store from ‘./redux/store';

import App from './App'; import './styles.css';

ReactDOM.render(

**<Provider store={store}>**

<App />

**</Provider>,** document.getElementById('root')

);

# Step 7: Add CSS Styles

1. **Create styles.css**:

* Inside the src folder, create a file named styles.css.

Add the following CSS code:css

body {

font-family: Arial, sans-serif; display: flex;

justify-content: center; align-items: center; height: 100vh;

margin: 0;

background-color: #f0f0f0;

}

.container {

background-color: white; padding: 20px;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1); width: 300px;

text-align: center;

}

.input-container

{ display: flex; margin-bottom: 20px;

}

input[type="text"] { flex: 1; padding: 10px;

border: 1px solid #ccc; border-radius: 4px 0 0 4px;

}

button {

padding: 10px; border: none;

background-color: #28a745; color: white;

cursor: pointer;

border-radius: 0 4px 4px 0;

}

button:hover {

background-color: #218838;

}

#todo-list {

list-style: none; padding: 0;

}

.todo-item {

display: flex;

justify-content: space-between; align-items: center;

padding: 10px;

border: 1px solid #ccc; border-radius: 4px; margin-bottom: 10px;

}

.todo-item button { background-color: #dc3545; color: white;

border: none;

padding: 5px 10px; cursor: pointer; border-radius: 4px;

}

.todo-item button:hover

{ background-color: #c82333;

}

# Optional: Convert the CSS to Sass (SCSS)

**Step 1: Understanding the Benefits of CSS Preprocessors**

Take the following CSS:

body {

font-family: Arial, sans-serif; display: flex;

justify-content: center; align-items: center; height: 100vh;

margin: 0;

background-color: #f0f0f0;

}

.container {

background-color: white; padding: 20px;

border-radius: 8px;

box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1); width: 300px;

text-align: center;

}

.input-container

{ display: flex; margin-bottom: 20px;

}

input[type="text"] { flex: 1; padding: 10px;

border: 1px solid #ccc; border-radius: 4px 0 0 4px;

}

button {

padding: 10px; border: none;

background-color: #28a745; color: white;

cursor: pointer;

border-radius: 0 4px 4px 0;

}

button:hover {

background-color: #218838;

}

#todo-list {

list-style: none; padding: 0;

}

.todo-item {

display: flex;

justify-content: space-between; align-items: center;

padding: 10px;

border: 1px solid #ccc; border-radius: 4px; margin-bottom: 10px;

}

.todo-item button { background-color: #dc3545; color: white;

border: none; padding: 5px 10px; cursor: pointer; border-radius: 4px;

}

.todo-item button:hover

{ background-color: #c82333;

}

Using SCSS, this can be made more reusable and descriptive with the following. We are using variables, mixins and includes:

// Variables for theming and reusability

$primary-color: #28a745;

$secondary-color: #218838;

$danger-color: #dc3545;

$danger-color-hover: #c82333;

$border-color: #ccc;

$background-color: #f0f0f0;

$box-shadow: 0 2px 10px rgba(0, 0, 0, 0.1);

// Mixins for reusable styles @mixin flex-center {

display: flex;

justify-content: center; align-items: center;

}

@mixin button-style($bg-color, $hover-color)

{ padding: 10px; border: none;

background-color: $bg-color;

color: white; cursor: pointer; border-radius: 4px;

transition: background-color 0.3s;

&:hover {

background-color: $hover-color;

}

}

// Main styles body {

font-family: Arial, sans-serif; @include flex-center;

min-height: 100vh; margin: 0;

background-color: $background-color;

}

.container {

background-color: white; padding: 20px;

border-radius: 8px;

box-shadow: $box-shadow; width: 300px;

text-align: center;

}

.input-container

{ @include flex- center; margin- bottom: 20px;

}

input[type="text"]

{ flex: 1; padding: 10px;

border: 1px solid $border-color;

border-radius: 4px 0 0 4px;

&:focus

{ outline:

none;

box-shadow: 0 0 0 2px rgba($primary-color,

0.3);

}

}

// Using the button mixin button {

@include button-style($primary-color, $secondary- color);

border-radius: 0 4px 4px 0; // Override border- radius for this specific button

}

#todo-list {

list-style: none;

padding: 0;

}

.todo-item {

@include flex-center; padding: 10px;

border: 1px solid $border-color; border-radius: 4px;

margin-bottom: 10px;

button {

@include button-style($danger-color, $danger- color-hover);

padding: 5px 10px; // Override padding for this specific button

}

}

# Part 2: Zustand

**main.tsx — App Bootstrapping + Routing Setup**

// React + ReactDOM are standard for rendering

import React from 'react';

import ReactDOM from 'react-dom/client';

// React Router 7 modern API

import {

createBrowserRouter,

RouterProvider,

redirect,

} from 'react-router-dom';

import './index.css'; // TailwindCSS styling

import App from './App'; // Layout component

import Home from './routes/Home'; // Route page

import About from './routes/About'; // Route page

import Dashboard from './routes/Dashboard'; // Route page

import { requireAuth } from './auth/requireAuth'; // Access control

//Router config using route objects (Router v7+)

const router = createBrowserRouter([

{

path: '/', // Main layout route

element: <App />,

children: [

{ index: true, element: <Home /> }, // '/' → Home page

{ path: 'about', element: <About /> }, // '/about'

{

path: 'dashboard',

// loader runs before route renders

loader: async () => {

await requireAuth(); // Redirects if not logged in

return { message: 'Welcome to the protected dashboard!' };

},

element: <Dashboard />,

},

],

},

]);

// Connect the router to your app

ReactDOM.createRoot(document.getElementById('root')!).render(

<React.StrictMode>

<RouterProvider router={router} />

</React.StrictMode>

);

**App.tsx — Layout + Navigation + Login/Logout**

import { Outlet, Link, useNavigate } from 'react-router-dom';

import { useAuthStore } from './auth/useAuthStore'; // Zustand state

export default function App() {

const { isLoggedIn, login, logout } = useAuthStore();

const navigate = useNavigate(); // Redirect on logout

return (

<div className="p-4 font-sans">

{/\* Navbar \*/}

<nav className="flex gap-4 mb-6 text-blue-600 font-semibold">

<Link to="/">Home</Link>

<Link to="/about">About</Link>

{isLoggedIn && <Link to="/dashboard">Dashboard</Link>}

</nav>

{/\* Auth buttons \*/}

<div className="mb-6">

{isLoggedIn ? (

<button

className="bg-red-500 text-white px-3 py-1 rounded"

onClick={() => {

logout(); // Zustand update + clear localStorage

navigate('/');

}}

>

Logout

</button>

) : (

<button

className="bg-green-500 text-white px-3 py-1 rounded"

onClick={login}

>

Login

</button>

)}

</div>

{/\* Route outlet (Home/About/Dashboard renders here) \*/}

<Outlet />

</div>

);

}

**routes/Home.tsx — Static Route Page**

export default function Home() {

return <h1 className="text-3xl font-bold">Home Page</h1>;

}

Simple content shown when you hit /.

**routes/About.tsx — Static Route Page**

export default function About() {

return <h1 className="text-3xl font-bold">About Us</h1>;

}

Same as above — just showing content for /about.

**routes/Dashboard.tsx — Protected Page with Loader**

import { useLoaderData } from 'react-router-dom';

export default function Dashboard() {

const data = useLoaderData() as { message: string };

return (

<div>

<h1 className="text-3xl font-bold text-green-700">Dashboard</h1>

<p className="mt-4">{data.message}</p>

</div>

);

}

**What**’**s special here:**

* useLoaderData() reads the object returned from the loader() defined in main.tsx
* If the user is logged in, they see the dashboard message
* If not, they're redirected before this even renders

**auth/useAuthStore.ts — Zustand Global Store for Auth**

import { create } from 'zustand';

interface AuthState {

isLoggedIn: boolean;

login: () => void;

logout: () => void;

}

// Zustand store with localStorage persistence

export const useAuthStore = create<AuthState>((set) => ({

isLoggedIn: localStorage.getItem('isLoggedIn') === 'true',

login: () => {

localStorage.setItem('isLoggedIn', 'true');

set({ isLoggedIn: true });

},

logout: () => {

localStorage.removeItem('isLoggedIn');

set({ isLoggedIn: false });

},

}));

This replaces Redux or React Context. You can use useAuthStore() from any component — even deeply nested ones.

**auth/requireAuth.ts — Route Guard Logic**

import { redirect } from 'react-router-dom';

// This is called in the dashboard loader

export function requireAuth() {

const loggedIn = localStorage.getItem('isLoggedIn') === 'true';

if (!loggedIn) {

throw redirect('/'); // React Router redirects before rendering the page

}

}

This is **true route protection** — not just hiding a button, but stopping rendering of the protected page.

**index.css — Tailwind Setup**

@tailwind base;

@tailwind components;

@tailwind utilities;

This enables Tailwind classes like text-3xl, bg-green-500, p-4, etc.

**How It Works Together**

1. **User visits /dashboard**
2. The loader runs requireAuth()
3. If logged in: Dashboard renders and shows the welcome message
4. If not logged in: redirect('/') sends them back to Home
5. Zustand tracks login state across refresh via localStorage
6. Tailwind gives you design with no custom CSS needed

**Next Part: Add a Login. Form:**

**src/routes/Login.tsx**

import { Form, useActionData } from 'react-router-dom';

export default function Login() {

const error = useActionData() as string | undefined;

return (

<div className="max-w-md mx-auto mt-10 p-6 bg-white shadow rounded">

<h1 className="text-2xl font-bold mb-4">Login</h1>

<Form method="post" className="space-y-4">

<div>

<label className="block">Username</label>

<input name="username" className="border px-2 py-1 w-full" />

</div>

<div>

<label className="block">Password</label>

<input type="password" name="password" className="border px-2 py-1 w-full" />

</div>

{error && <p className="text-red-500">{error}</p>}

<button type="submit" className="bg-blue-600 text-white px-4 py-2 rounded">

Log In

</button>

</Form>

</div>

);

}

**src/routes/loginAction.ts**

import { redirect } from 'react-router-dom';

import { useAuthStore } from '../auth/useAuthStore';

export async function loginAction({ request }: { request: Request }) {

const formData = await request.formData();

const username = formData.get('username');

const password = formData.get('password');

if (username === 'admin' && password === '123') {

useAuthStore.getState().login(); // Zustand login

return redirect('/dashboard');

}

return 'Invalid credentials';

}

**How to Test**

1. Visit / — see Home Page
2. Click **Login**
3. Use credentials:
   * Username: admin
   * Password: 123
4. You’ll be redirected to /dashboard
5. Click **Logout** — redirected to /

**We are using action() from Router 7**

Perfect follow-up — let’s walk through **exactly where and how action() is used in your project**, and how it connects everything together.

**What is action()?**

In React Router v6.4+, an action() is a function defined for a route that handles **form submissions** or **mutation logic** (like login, post creation, etc.).

You register the action() function on a specific route when defining your router.

**Where action() is used in your project:**

**1. In main.tsx → You assign it to the /login route:**

{

path: 'login',

element: <Login />, // the login form page

action: loginAction, // the function that handles the form POST

},

So when a user submits the form on /login, React Router automatically calls the loginAction function.

**2. loginAction is your action() function**

This lives in src/routes/loginAction.ts:

import { redirect } from 'react-router-dom';

import { useAuthStore } from '../auth/useAuthStore';

export async function loginAction({ request }: { request: Request }) {

const formData = await request.formData();

const username = formData.get('username');

const password = formData.get('password');

if (username === 'admin' && password === '123') {

useAuthStore.getState().login(); // trigger Zustand login state

return redirect('/dashboard'); // redirect on success

}

return 'Invalid credentials'; // return error for display

}

It receives the FormData from the submission and performs logic (auth in this case).

**3. In your form → <Form method="post">**

In Login.tsx, you use the <Form> component from React Router:

import { Form, useActionData } from 'react-router-dom';

<Form method="post">

<input name="username" />

<input name="password" />

<button type="submit">Log In</button>

</Form>

* When you hit submit, React Router automatically triggers the action() tied to the current route.
* *You* ***don***’***t need an onSubmit handler*** *— React Router wires it all up.*

**The Flow**

|  |  |
| --- | --- |
| **Step** | **What Happens** |
| User goes to /login | Sees <Form method="post"> |
| User submits credentials | React Router triggers loginAction() |
| loginAction() runs | Checks form data, updates Zustand, redirects or returns error |
| Zustand updates state | isLoggedIn = true, UI reacts |
| React Router redirects | To /dashboard or stays on /login |

# Lab 15: TanStack Query

**Why use TanStack Query (formerly React Query)?**

**1. No more manual useEffect + useState soup**

Without TanStack Query:

useEffect(() => {

fetch(...).then(...).catch(...)

}, [])

You have to set up:

* isLoading
* error
* data
* Re-fetch logic
* Caching logic

TanStack Query abstracts all of that into a clean hook interface:

const { data, isLoading, isError } = useQuery({ queryKey, queryFn })

**2. Automatic Caching**

If you re-mount the component or go to another tab and come back:

* TanStack Query uses the cached data immediately
* Then revalidates in the background

This makes apps feel fast and fresh without extra coding.

**No Need to Re-implement Polling, Retry, Pagination**

It has built-in support for:

* Automatic retries
* Background refetching
* Polling intervals
* Pagination/infinite scroll
* Mutation + optimistic updates. In TanStack Query, a mutation is any operation that changes data on the server — usually a:
  + POST (create)
  + PUT or PATCH (update)
  + DELETE (remove)

**4. Consistent Loading/Error States**

Manual logic often results in bugs like:

* Data flickering on refetch
* Stale data showing during transitions TanStack Query handles these gracefully.

**5. Devtools Support**

There’s a built-in **TanStack Query Devtools** for inspecting queries, cache, retries, etc. Great for debugging.

npm install @tanstack/react-query-devtools

**6. It Scales**

In a large app with many API calls:

* You avoid spaghetti code
* Queries are organized and consistent
* It’s easier to reason about side effects

**7. Works Great with SSR**

If you're using Next.js or Remix, TanStack Query integrates beautifully with server-side rendering and hydration.

**Can Use useEffect + fetch and *Don***’***t* Need TanStack Query If:**

* Your app only makes 1–2 API calls
* You don’t need to cache
* You don’t need to poll or do retries
* You can handle everything manually

**Project Setup Instructions**

**1. Install Dependencies**

In the root of your project, run:

npm install @tanstack/react-query @tanstack/react-query-devtools react react-dom

If you're using Vite:

npm install

npm run dev

This is a user directory app that:

Fetches users from a public API  
Loads more users (pagination)  
Adds a new user via a form  
Uses optimistic updates — new users appear instantly  
Uses @tanstack/react-query for all data management

**Code Overview**

**main.tsx**

* Initializes and configures the TanStack Query QueryClient
* Wraps the app in QueryClientProvider
* Adds TanStack Query Devtools for debugging

**App.tsx**

* Renders both the user list and the add-user form

**UserList.tsx**

* Uses useInfiniteQuery to:
  + Fetch users page-by-page
  + Display “Load More” button for pagination
* Shows loading and error states

**AddUserForm.tsx**

* Uses useMutation to POST a new user
* **Optimistically adds** the user to the UI before the server responds
* Rolls back the optimistic change if there's an error
* Avoids cache invalidation to prevent overwriting the UI too soon

**More Detail - See Comments in Code:**

**main.tsx — Application Entry Point**

Sets up the app’s TanStack Query client and renders the app.

* Creates a QueryClient instance with default settings:
  + retry: Will retry failed queries up to 3 times.
  + refetchOnWindowFocus: Automatically refetches data when window regains focus.
  + refetchInterval is commented out (was used for polling every 10s).
* Wraps your app in <QueryClientProvider> so all components can use TanStack Query.
* Includes <ReactQueryDevtools /> for debugging query states in development.

**App.tsx**

Renders main application layout.

* Displays the app title: “User Directory”.
* Includes two child components:
  + <AddUserForm /> – lets users add new entries.
  + <UserList /> – shows a list of users with pagination.

**UserList.tsx — Paginated Data Display**

Fetches and displays a paginated list of users using useInfiniteQuery.

* Calls the public API https://jsonplaceholder.typicode.com/users with pagination (?\_limit=3&\_page=X).
* Uses useInfiniteQuery to:
  + Load 3 users at a time.
  + Track how many pages have been loaded.
* fetchNextPage() loads the next page of users when the button is clicked.
* Shows loading and error messages.
* Renders each page of users inside a separate <ul>.

**AddUserForm.tsx — Form + Optimistic Mutation**

Adds a new user with an optimistic UI update.

* useState() is used to track the form input values.
* useMutation() is used to **send a POST request** to add a user.
* onMutate performs an **optimistic update**:
  + Cancels any running user queries.
  + Gets current cached users.
  + Immediately adds a new user to the beginning of the first page in cache.
  + Stores the previous data for rollback.
* onError restores the previous data if something fails.
* onSettled does **not** invalidate the cache (you commented it out) so the new user doesn't flash and disappear.

# Lab 16: Authentication

We will explore authentication with JWT and then with Okta.

**Part 1: Authentication with JWT**

Instructions for the Express backend (server) and the React frontend (App.jsx):

**Step 1: Backend - Express Server Setup (server.cjs)**

1. **Set Up the Express Server**:
   * Install Express: npm install express.
   * Create a new file called server.cjs to set up the Express server.
   * The server listens on port 5000 (you can change this to another port if needed).
2. **Middleware Setup**:
   * Use express.json() to parse JSON request bodies. This allows the server to read the data from POST requests.
   * **Token Generation**: A simplified function generateToken is used to generate a fake token for login. Replace it with a proper JWT generation mechanism in production.
   * **Token Verification**: The middleware verifyToken is used to check if the incoming request has a valid token. It checks the Authorization header and ensures that it starts with fake\_token\_. In production, replace this with jwt.verify() to validate the token properly.
3. **Login Route (/api/login)**:
   * A POST route is provided for logging in (/api/login).
   * The route checks if the provided username and password match a predefined user ('myuser' and 'mypassword').
   * If the credentials are valid, it generates a token and sends it back in the response.
4. **Protected Resource Route (/api/protected-resource)**:
   * This route is protected by the verifyToken middleware.
   * Only users with a valid token can access this resource. If the token is invalid or missing, the user will receive a 401 Unauthorized response.

**Full server code:**

const express = require('express');

const app = express();

const port = 5000; // Choose your port

// Middleware to parse JSON request bodies

app.use(express.json());

// Simplified token generation (INSECURE - FOR DEMO ONLY)

const generateToken = (username) => {

return `fake\_token\_${username}`; // Replace with proper JWT generation

};

// Middleware to verify the token (INSECURE - FOR DEMO ONLY)

const verifyToken = (req, res, next) => {

const authHeader = req.headers.authorization;

if (authHeader) {

const token = authHeader.split(' ')[1]; // Bearer <token>

// In a real app, verify the token using jwt.verify()

if (token.startsWith('fake\_token\_')) { // Very basic check

next(); // Token is valid (for this demo)

} else {

res.sendStatus(401); // Unauthorized

}

} else {

res.sendStatus(401); // No token provided

}

};

// Login route to authenticate and provide a token

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

const { username, password } = req.body;

if (username === 'myuser' && password === 'mypassword') { // Replace with real authentication

const token = generateToken(username);

res.json({ token });

} else {

res.status(401).json({ message: 'Invalid credentials' });

}

});

// Protected route, only accessible if the token is valid

app.get('/api/protected-resource', verifyToken, (req, res) => {

// Access to this route is protected by the verifyToken middleware

res.json({ message: 'This is protected data!' });

});

// Start the server and listen on the defined port

app.listen(port, () => {

console.log(`Server listening on port ${port}`);

});

1. **Run the Server**:
   * The server starts and listens on port 5000.
   * The server will respond to login requests and protected resource requests, provided the token is valid.

**Step 2: Frontend - React Application (App.jsx)**

* **Install Axios**:
  + Install Axios to make HTTP requests: npm install axios.
* **React App Component**:
  + The component uses state variables: data, loading, and error to manage the API data, loading state, and errors.
  + It retrieves the JWT token stored in localStorage using localStorage.getItem(‘jwtToken').
* **Use Effect Hook**:
  + A useEffect hook is used to fetch data from the protected API route (/api/protected-resource) once the component mounts or when the jwtToken changes.
  + If the token is available, it sends a GET request to the backend with the Authorization header containing the token (Bearer <jwtToken>).
  + If the request is successful, the data is stored in the data state.
  + If an error occurs (e.g., token expired or invalid), it clears the invalid token from localStorage and displays an error message.
* **Login Handling**:
  + The handleLogin function sends a POST request to the /api/login route with predefined credentials ('myuser' and 'mypassword').
  + If the login is successful, the JWT token returned from the server is stored in localStorage.
  + The user can then access protected resources after logging in.
* **Conditional Rendering**:
  + If the data is loading, it displays "Loading...".
  + If there is an error, it displays an error message.
  + If the data is available, it displays the protected data.
  + If no data is available (and the user is not logged in), it displays a message asking the user to log in and a login button.

**Full frontend code:**

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

import axios from 'axios';

const App = () => {

const [data, setData] = useState(null);

const [loading, setLoading] = useState(true);

const [error, setError] = useState(null);

const jwtToken = localStorage.getItem('jwtToken'); // Get token from localStorage

useEffect(() => {

const fetchData = async () => {

try {

const response = await axios.get('/api/protected-resource', { // Your API endpoint

headers: {

Authorization: `Bearer ${jwtToken}`, // Include token in Authorization header

},

});

setData(response.data);

} catch (err) {

setError(err);

if (err.response && err.response.status === 401) {

// Handle 401 Unauthorized (e.g., token expired)

console.error("Token expired or invalid. Redirecting to login.");

localStorage.removeItem('jwtToken'); // Clear invalid token

// Redirect to login page (e.g., using react-router-dom)

// window.location.href = '/login'; // Or use react-router's history.push

} else {

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

}

} finally {

setLoading(false);

}

};

if (jwtToken) { // Only fetch if a token exists

fetchData();

} else {

setLoading(false); // No token, set loading to false

setError("No token available. Please login."); // Optional: Set an error message

// Redirect to login page (e.g. window.location.href = '/login')

}

}, [jwtToken]); // Add jwtToken to dependency array to refetch when token changes

const handleLogin = async () => {

try {

const response = await axios.post('/api/login', {

username: 'myuser', // Replace with your login data

password: 'mypassword'

});

const token = response.data.token; // Assuming your API returns a token

localStorage.setItem('jwtToken', token); // Store token in localStorage

// Optionally: redirect or update state to reflect logged-in status

console.log("Logged in successfully!");

} catch (error) {

console.error("Login failed:", error);

// Handle login errors (display message, etc.)

}

};

// Conditional rendering based on loading state and errors

if (loading) {

return <div>Loading...</div>;

}

if (error) {

return <div>Error: {error.message}</div>;

}

return (

<div>

{data ? (

<div>

<h2>Protected Data:</h2>

<pre>{JSON.stringify(data, null, 2)}</pre> {/\* Display the received data \*/}

</div>

) : (

<div>

<p>No data available. Please login.</p>

<button onClick={handleLogin}>Login</button>

</div>

)}

</div>

);

};

export default App;

**How to Run:**

1. **Ensure the Backend is Set Up**:
   * Make sure server code is saved to server.cjs.
   * Make sure Express is installed with npm install express.
   * Run the server with node server.cjs.
2. **Ensure the Frontend is Set Up**:
   * Make sure the React code is saved to App.jsx.
   * Make sure Axios is installed with npm install axios.
   * Make sure your React app is running and able to make requests to localhost:5000 or the server URL you specify.
3. **Login**:
   * Visit the React app in your browser. If no token is stored in localStorage, it will ask you to log in.
   * Use the username myuser and password mypassword to get a valid token.
   * Once logged in, you can access the protected resource from the /api/protected-resource route.
4. **Security**:
   * The provided code uses a fake token for demonstration purposes. In a production environment, replace the token generation with a proper JWT generation (e.g., using the jsonwebtoken library).
   * Ensure that sensitive data (like passwords and tokens) is handled securely.

**Part 2: Authentication with Okta**

**What is Okta?**

Okta is a leading identity and access management (IAM) platform. In simpler terms, it helps organizations manage and secure user access to their applications and services.

* **Authentication:** Verifying who a user is (e.g., through username and password, multi-factor authentication).
* **Authorization:** Determining what a user is allowed to do once they are authenticated (e.g., accessing specific features or data).
* **Single Sign-On (SSO):** Allowing users to log in once and then access multiple applications without having to re-authenticate.
* **User Management:** Managing user accounts, profiles, and group memberships.

Okta provides a set of APIs and SDKs that developers can integrate into their applications to handle authentication and authorization, offloading the complexity and security risks associated with building these systems from scratch.

**How Okta and React Work Together**

1. **Authentication in React Applications:**
   * React applications, being front-end, rely on a backend service for data and often authentication.
   * Instead of building their own authentication system, developers can integrate Okta into their React application.
   * Okta provides SDKs (like @okta/okta-react) that simplify the process of handling authentication flows within the React application.
   * These SDKs typically handle things like:
     + Redirecting users to the Okta sign-in page.
     + Handling the callback after successful authentication.
     + Storing and managing authentication tokens (like access tokens and ID tokens).
     + Providing components or hooks to check user authentication status.
2. **Securing API Calls:**
   * Once a user is authenticated via Okta in the React application, the application can obtain an access token from Okta.
   * This access token can then be included in the headers of API requests made from the React application to the backend services.
   * The backend services can then verify the access token with Okta to ensure the request is coming from an authenticated user with the necessary permissions.
3. **Single Sign-On (SSO):**
   * If an organization uses Okta for SSO across multiple applications, users who are already logged into another application secured by Okta can seamlessly access the React application without needing to log in again.

**Okta Resources for React:**

* **Okta React SDK (@okta/okta-react):** A library specifically designed to integrate Okta authentication into React applications. It provides components and hooks for handling login, logout, and accessing user information.
* **Okta Developer Documentation:** Comprehensive guides and examples on how to integrate Okta with various front-end frameworks, including React.
* **Okta CLI:** A command-line interface for managing Okta applications and configurations.

**Set up your Okta application to integrate the necessary components in your React application.**

1. **Okta Setup:**
   * **Create an Okta Developer Account:** If you don't already have one, sign up for a free Okta developer account at https://developer.okta.com/.
   * **Create a New Application:**
     + Log in to your Okta developer console.
     + Go to Applications-> **Applications**.
     + Click Add **Application**.
     + Choose Single-PageAppand click **Next**.
     + Give your application a name (e.g., "My React App").
     + Configure the BaseURIsand LoginredirectURIs. Crucially, these must match

the URLs of your React application. For development, this might be http:// localhost:3000 or similar. For production, use your actual domain. The redirect URI is where Okta will send the user after successful authentication. A

common convention is to use /callback or /implicit/callback.

* + - Click **Done**.
  + **Retrieve Client ID and Issuer:** After creating the application, you'll be given a **Client ID** and an **Issuer**. You'll need these for your React application. The Issuer will look something like https://{yourOktaDomain}.okta.com/oauth2/ default.

1. **React Application Setup:**
   * **Install Dependencies:** You'll need the Okta React SDK.

npm install @okta/okta-react @okta/okta-signin-widget

* + **Create an Okta Configuration File (e.g., okta-config.js):**

export const oktaConfig = {

clientId: '{yourClientId}', // From your Okta application issuer: '{yourIssuer}', // From your Okta application redirectUri: window.location.origin + '/callback', //

Must match your Okta config

scopes: ['openid', 'email', 'profile'], // Optional scopes (customize as needed)

};

* + **Wrap the app with SecurityProvider:** make the Okta authentication context available throughout your React application. In your index.js or App.js:

import React from 'react';

import ReactDOM from 'react-dom/client'; import App from './App';

import { SecurityProvider } from '@okta/okta-react'; import { oktaConfig } from './okta-config';

import { BrowserRouter } from 'react-router-dom'; // If using React Router

const root = ReactDOM.createRoot(document.getElementById('root')); root.render(

<BrowserRouter> {/\* If using React Router \*/}

<SecurityProvider {...oktaConfig}>

<App />

</SecurityProvider>

</BrowserRouter>

);

* + **Create a Callback Route:** This route handles the redirect from Okta after authentication.

You'll typically use the useOktaAuth hook here. If you are using React Router, you could have a route like this:

import { useOktaAuth } from '@okta/okta-react';

import { useNavigate } from 'react-router-dom'; // If using React Router

const CallbackPage = () => {

const { oktaAuth } = useOktaAuth();

const navigate = useNavigate(); // If using React Router

useEffect(() => {

const fn = async () => { try {

await oktaAuth.handleLoginRedirect(); navigate('/protected'); // Redirect to a protected

route after login

} catch (err) {

console.error('Error handling login redirect:',

err);

}

};

fn();

}, [oktaAuth, navigate]);

return (

<div>Processing authentication...</div>

);

};

// ... In your App.js route definitions (if using React Router)

<Route path="/callback" element={<CallbackPage />} />

* + **Protect Routes/Components:** Use the useOktaAuth hook to check if the user is authenticated.

import { useOktaAuth } from '@okta/okta-react';

import { Navigate } from 'react-router-dom'; // If using React Router

const ProtectedRoute = ({ children }) => { const { isAuthenticated } = useOktaAuth();

if (!isAuthenticated) {

return <Navigate to="/login" />; // Redirect to login if not authenticated

}

return children;

};

// ... Usage:

<Route path="/protected" element={<ProtectedRoute><MyProtectedComponent /></ ProtectedRoute>} />

* + **Login/Logout:**

import { useOktaAuth } from '@okta/okta-react';

const LoginButton = () => {

const { oktaAuth } = useOktaAuth();

const login = async () => await oktaAuth.signInWithRedirect();

return <button onClick={login}>Log In</button>;

};

const LogoutButton = () => {

const { oktaAuth } = useOktaAuth();

const logout = async () => await oktaAuth.signOut(); return <button onClick={logout}>Log Out</button>;

};

* + **SecurityProvider:** This is essential for providing the Okta authentication context to your application.
  + **oktaConfig:** Centralized configuration makes it easier to manage your Okta settings.
  + **CallbackPage:** Handles the redirect from Okta after authentication, processing the tokens and redirecting the user to the appropriate page.
  + **ProtectedRoute:** A reusable component to protect routes or sections of your application. This is a cleaner approach than embedding authentication checks directly in your components.
  + **useOktaAuth:** Provides access to authentication-related functions and state within your components.
  + **React Router Integration (Optional but Recommended):** The examples show how to integrate Okta authentication with React Router for protected routes and navigation.
  + **Error Handling:** The CallbackPage example includes basic error handling. You should expand on this in a production application.

1. **Authorization (Beyond Authentication):**

Okta primarily handles *authentication* (verifying who the user is). *Authorization* (what the user is allowed to do) is typically handled within your application after the user is authenticated. You can use the user's information (e.g., roles or groups) obtained from Okta to make authorization decisions.

* + **Okta Groups/Roles:** You can configure groups and roles in Okta and assign users to them. These can be included in the ID token returned after authentication.
  + **Custom Authorization Logic:** In your React application, after authenticating with Okta, you can access the user's groups/roles from the ID token and use this information to control access to specific features or resources. This might involve conditional rendering of components, API calls, or other logic.

**Example of authorization logic:**

import { useOktaAuth } from '@okta/okta-react'; const MyComponent = () => {

const { oktaAuth, authState } = useOktaAuth();

if (authState?.isAuthenticated) {

const accessToken = authState.accessToken;

const userGroups = accessToken?.claims?.groups || []; // Assuming 'groups' claim

if (userGroups.includes('admin')) { return (

<div>

{/\* Admin-only content \*/}

<button>Admin Action</button>

</div>

);

} else {

return <div>Regular user content</div>;

}

}

return <div>Please log in.</div>;

};

# Lab 17 (Optional): Next.js

This exercise demonstrates setup for modern Next.js; further exploration will be done as a walk-through. We are using **Next.js 13** with the **app directory** structure. This will set up your layout, pages, and routing.

* Run npx create-next-app@latest my-next-app
* Navigate to the project directory cd my-next-app
* Start the development server with npm run dev

**Project Structure**

The structure of your project will look something like this:

app/

layout.tsx // Global layout file

page.tsx // Home page (maps to /)

about/

page.tsx // About page (maps to /about)

contact/

page.tsx // Contact page (maps to /contact)

globals.css // Global styles (can include fonts, reset styles, etc.)

**1. app/layout.tsx — Global Layout Component**

This file defines the overall layout structure of your application, including the header, footer, and any global content that is shared across different pages.

import { ReactNode } from "react";

import "./globals.css"; // Import global styles

export default function Layout({ children }: { children: ReactNode }) {

return (

<html lang="en"> {/\* Specifies the language of the page \*/}

<body>

{/\* Header section with navigation \*/}

<header>

<nav>

<ul>

<li><a href="/">Home</a></li>

<li><a href="/about">About</a></li>

<li><a href="/contact">Contact</a></li>

</ul>

</nav>

</header>

{/\* Main content area where children (page content) will be rendered \*/}

<main>{children}</main>

{/\* Footer with right \*/}

<footer>

<p>&; 2025 My Next App</p>

</footer>

</body>

</html>

);

}

**2. app/page.tsx — Home Page (Maps to /)**

This file represents your home page and will be rendered when the user navigates to the root URL (/).

export default function HomePage() {

return (

<div>

<h1>Welcome to the Home Page</h1>

<p>Content for the homepage will go here.</p>

</div>

);

}

**3. app/about/page.tsx — About Page (Maps to /about)**

This file represents your About page and will be displayed when the user navigates to /about.

export default function AboutPage() {

return (

<div>

<h1>About Us</h1>

<p>This is the About page.</p>

</div>

);

}

**4. app/contact/page.tsx — Contact Page (Maps to /contact)**

This file represents your Contact page and will be displayed when the user navigates to /contact.

export default function ContactPage() {

return (

<div>

<h1>Contact Us</h1>

<p>This is the Contact page.</p>

</div>

);

}

**5. app/globals.css — Global Styles**

This file contains global CSS that will be applied to all pages in your application. For example:

/\* globals.css \*/

body {

font-family: Arial, sans-serif;

margin: 0;

padding: 0;

}

ul {

list-style-type: none;

}

a {

text-decoration: none;

color: #0070f3;

}

a:hover {

color: #0056b3;

}

**6. Routing in Next.js 13 with app Directory**

With this setup, routing is automatic and is based on the folder and file structure inside the app directory. Here's how it works:

* / corresponds to the app/page.tsx file (Home Page).
* /about corresponds to the app/about/page.tsx file (About Page).
* /contact corresponds to the app/contact/page.tsx file (Contact Page).

**How Routing Works:**

* **app/page.tsx** is your root page (/).
* **app/about/page.tsx** is your About page (/about).
* **app/contact/page.tsx** is your Contact page (/contact).
* The layout component (app/layout.tsx) wraps all of these pages and adds the header, footer, and navigation links.

**More about Next.js:**

Next.js is a framework that supports simple static websites to complex web applications.

* Static Site Generation (SSG)
* Server-Side Rendering (SSR)
* API Routes
* Dynamic Routing
* Image Optimization
* Internationalization (i18n)
* Middleware
* Authentication
* Performance Optimization

**1. Static Site Generation (SSG)**

Next.js makes it easy to generate static HTML at build time, which is great for SEO and performance. You can use **getStaticProps** to fetch data at build time and generate static pages.

**Static generation:**

// app/about/page.tsx

export default function AboutPage({ data }: { data: string }) {

return <div>{data}</div>;

}

export async function getStaticProps() {

const data = 'This is static data fetched at build time.';

return { props: { data } };

}

**2. Server-Side Rendering (SSR)**

Next.js supports server-side rendering (SSR), where pages are generated on the server on each request. You can use **getServerSideProps** to fetch data on the server for each request, making the page always up-to-date with the latest data.

**SSR:**

// app/contact/page.tsx

export default function ContactPage({ contactInfo }: { contactInfo: string }) {

return <div>{contactInfo}</div>;

}

export async function getServerSideProps() {

const contactInfo = 'Server-side rendered content.';

return { props: { contactInfo } };

}

**3. API Routes**

Next.js allows you to create API endpoints easily with its **API Routes** feature. These are useful for handling server-side logic, like fetching data from a database, interacting with external APIs, or performing user authentication.

**An API route:**

// app/api/hello/route.ts

export async function GET() {

return new Response('Hello, world!');

}

This will map to the URL /api/hello.

**4. Image Optimization**

Next.js has built-in image optimization through its <Image> component. This component automatically optimizes and serves images in modern formats (like WebP) for faster load times and better performance.

**Optimized image:**

import Image from 'next/image';

import myImage from '../public/my-image.jpg';

export default function MyComponent() {

return (

<div>

<h1>Optimized Image</h1>

<Image src={myImage} alt="A cool image" width={500} height={300} />

</div>

);

}

**5. Internationalization (i18n)**

Next.js has built-in support for internationalization, allowing you to serve content in multiple languages. You can set up different locales and automatic routing for each language.

**i18n in next.config.js:**

// next.config.js

module.exports = {

i18n: {

locales: ['en', 'fr', 'es'],

defaultLocale: 'en',

},

};

Next.js will automatically route /en, /fr, and /es based on the user's language preferences.

**6. Custom Document (HTML)**

You can customize the HTML template for your application by modifying the default document. For example, you might want to add custom meta tags or a custom <head> tag for SEO or analytics.

**Custom document:**

// app/\_document.tsx

import Document, { Html, Head, Main, NextScript } from 'next/document';

class MyDocument extends Document {

render() {

return (

<Html lang="en">

<Head>

<meta name="description" content="My amazing website" />

<link rel="icon" href="/favicon.ico" />

</Head>

<body>

<Main />

<NextScript />

</body>

</Html>

);

}

}

export default MyDocument;

**7. File-based Routing**

Next.js has **file-based routing**, meaning that your file and folder structure inside the app directory directly maps to URLs. You can easily create dynamic routes and even catch-all routes using this structure.

**Dynamic routes:**

// app/blog/[slug]/page.tsx

export default function BlogPost({ slug }: { slug: string }) {

return <div>Blog Post: {slug}</div>;

}

export async function getStaticProps({ params }: { params: { slug: string } }) {

const slug = params.slug;

return { props: { slug } };

}

This will map to /blog/[slug], where [slug] is a dynamic parameter (e.g., /blog/hello-world).

**8. Middleware**

Next.js 12+ supports **middleware** to run code before a request is completed. Middleware allows you to add logic like authentication checks, redirects, or logging to specific routes.

**Middleware:**

// middleware.ts

export function middleware(req) {

const { pathname } = req.nextUrl;

if (pathname === '/restricted' && !req.cookies.get('auth-token')) {

return new Response('Unauthorized', { status: 401 });

}

return NextResponse.next();

}

**9. Preview Mode**

Next.js has a **Preview Mode** feature, allowing you to preview unpublished content while keeping the website in a live state. It’s useful when you're working with a headless CMS.

**10. Incremental Static Regeneration (ISR)**

With **ISR**, you can update static pages after build time without needing to rebuild the entire site. This is useful when you need fresh content without compromising performance.

**ISR:**

export async function getStaticProps() {

const data = await fetchData();

return {

props: { data },

revalidate: 10, // Rebuild the page every 10 seconds

};

}

**11. Authentication**

You can implement authentication with Next.js using third-party providers (like **NextAuth.js**) or integrate with your own authentication backend (e.g., JWT tokens or OAuth).

**12. Custom Error Pages**

Next.js allows you to define custom error pages for various HTTP status codes like 404 (Page Not Found) or 500 (Server Error). You can add custom error handling in your app by creating files like:

* app/error.tsx for global errors.
* app/404.tsx for 404 errors.

**13. Static File Serving**

You can serve static files such as images, fonts, and documents in the public directory. These files are accessible from the root URL, like /favicon.ico or /robots.txt.

**14. Deploying with Vercel**

Next.js is designed to work seamlessly with **Vercel**, the platform from the creators of Next.js. Vercel automatically handles deployment, scaling, and performance optimization, making it a great choice for hosting your Next.js app. Deploying to Vercel is as simple as pushing your code to GitHub and linking it with Vercel.

**15. Performance Optimization**

Next.js has a number of built-in performance features, including:

* **Automatic Code Splitting**: Only the JavaScript necessary for the current page is loaded.
* **Prefetching**: Next.js automatically prefetches links in the background to make navigation faster.
* **Bundle Analysis**: You can analyze the size of your app’s JavaScript bundles to identify any large dependencies.

**16. Styling Options**

Next.js allows you to style your app using various approaches:

* **CSS Modules**: Scoped styles using .module.css files.
* **SASS/SCSS**: Support for SASS/SCSS for more powerful CSS.
* **Tailwind CSS**: A utility-first CSS framework.
* **CSS-in-JS**: Styled-components, Emotion, etc.

**Comparison: React's Server-Side Rendering (SSR) and Next.js:**

React's **Server-Side Rendering (SSR)** and **Next.js** both provide solutions for rendering JavaScript applications on the server, but they have different approaches, features, and use cases. Here's a detailed comparison between **Next.js** and **React's new SSR** (React 18 SSR with Suspense and Streaming).

**1. Out-of-the-box Features and Setup**

**Next.js:**

* **All-in-one framework**: Next.js is a full-featured framework for React that offers routing, server-side rendering, static site generation (SSG), API routes, file-based routing, image optimization, and more.
* **SSR, SSG, ISR**: Next.js makes it very easy to choose between **Server-Side Rendering (SSR)**, **Static Site Generation (SSG)**, and **Incremental Static Regeneration (ISR)** on a per-page basis.
* **API Routes**: Next.js includes support for building API routes directly within the project.
* **Built-in routing**: With Next.js, routing is done using the file system (e.g., app/page.tsx for /, app/about/page.tsx for /about), which simplifies routing and page management.
* **Automatic code splitting**: Next.js automatically splits the code into smaller bundles, so only the code necessary for the current page is loaded, improving performance.
* **Zero-config**: Next.js works well out-of-the-box with minimal configuration.

**React SSR (with Suspense and Streaming):**

* **React 18 SSR**: React has introduced a new SSR approach using **Suspense** and **Streaming**. React 18 allows you to render pages on the server while streaming the HTML to the client as it's generated.
* **Requires manual setup**: While React SSR is available out-of-the-box starting in React 18, it requires additional setup compared to Next.js. You need to configure SSR using custom server setups (e.g., Express, Fastify, etc.) and create an entry point for your app's server-side rendering process.
* **No routing**: React itself does not provide a built-in routing solution. You have to use a routing library like **React Router** to handle routing in SSR applications, which can be more complex to manage compared to Next.js' file-based routing.
* **Fine-grained control**: React SSR gives you more control over the rendering process. You can choose to stream content on-demand or wait for the entire page to be ready before sending it to the client.

**2. Server-Side Rendering (SSR) Approach**

**Next.js:**

* **Integrated SSR**: Next.js simplifies SSR by providing built-in methods like **getServerSideProps** that automatically handle the fetching and rendering of server-side content for each request.
  + This allows developers to fetch dynamic data before rendering, ensuring that the page is pre-rendered with up-to-date content on the server.
  + It also allows you to combine SSR with static generation and incremental static regeneration (ISR), enabling hybrid models of data fetching and rendering.

export async function getServerSideProps() {

const data = await fetchData();

return { props: { data } };

}

**React SSR:**

* **Manual setup with Suspense & Streaming**: React's SSR is more flexible and powerful but requires a lot more work to set up. You need to create a custom server and manage the streaming of the React app’s components to the client, which requires careful management of when and how data is fetched and rendered.
* **Suspense for Data Fetching**: React SSR can leverage **Suspense** for data fetching, meaning components can be rendered on the server while data is being fetched asynchronously. Once the data is available, it can be streamed and hydrated on the client.  
    
  // A component with Suspense

const Component = React.lazy(() => import('./Component'));

function App() {

return (

<Suspense fallback={<div>Loading...</div>}>

<Component />

</Suspense>

);

}

**3. Static Site Generation (SSG) and Hybrid Models**

**Next.js:**

* **SSG**: Next.js supports static site generation out-of-the-box, allowing you to generate HTML at build time for improved performance and SEO.
* **ISR (Incremental Static Regeneration)**: Next.js allows you to statically generate pages and then update them in the background without needing to rebuild the entire site, making it ideal for large websites that need to be updated frequently.

export async function getStaticProps() {

const data = await fetchData();

return { props: { data }, revalidate: 10 };

}

**React SSR:**

* **No direct SSG support**: React's SSR does not have a built-in method for static site generation like Next.js. You would need to manually implement a build process to generate static HTML or handle static pages in other ways.
* React's SSR with **Suspense** and **Streaming** allows you to do something similar to SSG in terms of static content rendering, but it's more focused on dynamic rendering rather than pre-building static pages.

**4. Performance Optimization**

**Next.js:**

* **Automatic Code Splitting**: Next.js automatically splits the JavaScript bundle into smaller pieces that only load what is necessary for the current page.
* **Image Optimization**: Next.js optimizes images by default using the <Image> component, serving images in modern formats (like WebP) and resizing them automatically.
* **Prefetching**: Next.js pre-fetches pages linked to from the current page, speeding up navigation.

**React SSR:**

* **Streaming and Suspense**: React SSR with Suspense and Streaming allows you to stream HTML to the client as it’s generated. This can improve the time-to-interactive for the user.
* **Manual Optimization**: Performance optimizations such as image optimization and code splitting are not built into React SSR directly. You will need to handle them manually using other tools like **React.lazy**, **Suspense**, and bundlers like Webpack.

**5. Deployment and Hosting**

**Next.js:**

* **Vercel Deployment**: Next.js is tightly integrated with **Vercel**, which offers seamless deployment, serverless functions, and edge caching for Next.js apps.
* **Hybrid Hosting**: Next.js supports hybrid hosting, meaning you can mix static and server-rendered pages in the same application.

**React SSR:**

* **Custom Server**: React's SSR requires you to set up your own server, which could be a custom Express or Fastify server. You are responsible for handling the deployment and scaling of the server.
* **More flexibility**: While React SSR gives you more control over deployment (you can deploy anywhere), it also requires more work to configure and scale the application.

**6. Ecosystem and Community**

**Next.js:**

* **Strong Ecosystem**: Next.js has a large, active community, and its features are well-documented. You can also integrate easily with services like **Vercel**, **Stripe**, and **Auth0** for authentication, payments, and more.
* **Full-stack Capabilities**: Next.js has built-in support for API routes, making it a great choice for full-stack applications.

**React SSR:**

* **Flexibility with Libraries**: React's SSR is more flexible, but it requires integrating third-party libraries for things like routing (e.g., React Router), state management, and more.
* **More Manual Control**: React SSR gives you more control but also requires a deeper understanding of server-side rendering and manual setup of various features.

**Key Differences:**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Next.js** | **React SSR** |
| **SSR Setup** | Built-in with minimal configuration | Requires custom setup with server |
| **Routing** | File-based routing (auto) | Requires React Router or similar |
| **Static Site Generation (SSG)** | Built-in, with ISR support | No built-in support, requires custom setup |
| **Performance Optimizations** | Automatic code splitting, image optimization, prefetching | Requires manual optimization |
| **API Routes** | Built-in API routes | Requires custom API server setup |
| **Deployment** | Best suited for Vercel or static hosting | Requires custom server deployment |
| **Community & Ecosystem** | Strong, integrated tools (Vercel) | More flexibility, but fewer out-of-the-box tools |

**Conclusion:**

* **Next.js** is a **comprehensive, opinionated framework** that provides out-of-the-box features for SSR, SSG, API routes, image optimization, and more. It's great for developers who want to quickly build production-ready apps with minimal setup and without worrying too much about the complexities of manual SSR setup.
* **React SSR** provides **more flexibility** but requires more manual setup. It's perfect for developers who need full control over the server-side rendering process or already have a custom server architecture in place. However, it lacks the built-in tools and ecosystem provided by Next.js.