# **Enterprise React 18+ Development Guide: Zustand, Material-UI v5, and Emotion (Mid-2025 Edition)**

## **1. Executive Summary**

This report provides a comprehensive, expert-level guide for enterprise teams initiating new projects on React 18+ with a mid-2025 technology perspective. It endorses a specific, high-performance technology stack: **Zustand** for state management, **Material-UI (MUI) v5** for UI components, and its native styling engine, **Emotion**. This combination offers a potent mix of developer-centric APIs, scalability, and the performance characteristics necessary for modern, data-intensive web applications.

The core architectural tenet of this guide is the establishment of a robust, provider-based foundation, particularly within a Next.js App Router environment. While Zustand is renowned for its "provider-less" capability in client-only applications, its integration into server-rendered architectures necessitates a per-request provider pattern to guarantee state isolation and prevent data leakage between users. This pattern, combined with MUI and Emotion's requirement for a server-side style registry provider, solidifies a structured approach that is paramount for enterprise-grade stability and server-side rendering (SSR) compatibility.

Key strategic recommendations include:

* **State Management (Zustand):** Employ the "slices" pattern to modularize the state store for long-term maintainability.1 Mitigate performance bottlenecks by enforcing the use of atomic selectors for single-value subscriptions and the  
  useShallow hook for multi-value subscriptions to prevent unnecessary component re-renders.3
* **UI & Styling (MUI v5 & Emotion):** Utilize MUI v5 as the stable, long-term supported component library foundation.5  
  **Critically, teams must avoid Joy UI**, as official sources confirm its development is "on hold," presenting an unacceptable maintenance risk for enterprise projects.7 Customization should be achieved through MUI's powerful theming system. For styling, use Emotion's  
  sx prop for one-off modifications and the styled() API for creating reusable, complex components.9
* **Performance Optimization:** Proactively manage application performance through disciplined bundle-size management (enforcing path-based imports for MUI) 11, judicious use of memoization, and a clear understanding of React 18's concurrency model. Zustand v4+ inherently addresses concurrency pitfalls like "store tearing" through its use of the  
  useSyncExternalStore hook, a critical feature for application stability.12

Adherence to these best practices will enable development teams to leverage the speed and simplicity of this modern stack while building a scalable, performant, and maintainable enterprise application, ultimately leading to faster development cycles and a superior end-user experience.

## **2. Library Overviews & Mid-2025 Status**

This section provides an in-depth analysis of each library within the recommended stack, evaluating its current status, community adoption, and strategic fit for an enterprise context as of mid-2025.

### **2.1 Zustand: The Minimalist State Manager**

Zustand has established itself as a leading state-management solution in the modern React ecosystem, prized for its simplicity and performance.

* **Philosophy and API:** Zustand is a small, fast, and scalable "bearbones" solution based on simplified flux principles.1 Its core philosophy is to provide powerful state management with minimal boilerplate. The API is hook-based and exceptionally concise; developers create a store using a single  
  create function and can immediately use it as a hook within components.15 Unlike Redux, it does not require wrapping the application in a context provider for client-side-only applications, nor does it mandate complex constructs like actions or reducers.1 State updates are immutable, and the  
  set function automatically merges state, simplifying update logic.1
* **Community Adoption & Trends (2023-2025):** Since 2023, Zustand's adoption has surged, becoming a staple in many modern boilerplates and full-stack templates, including the popular T3 Stack.14 Its GitHub repository shows a healthy and active community with consistent updates and high engagement (over 25,000 stars).18 Industry analysis and developer surveys position it as a primary, lightweight alternative to Redux Toolkit, particularly for small to medium-sized applications or teams that prioritize development speed.17 The broader trend in the JavaScript ecosystem, as noted in the Rising Stars of JS reports, favors tools that are easier to adopt and improve developer experience, a niche that Zustand fills perfectly.22
* **Key Differentiators:** Zustand's primary advantage is its performance-oriented design. It achieves this through **selective re-renders**: components subscribe only to the state slices they need, preventing re-renders when unrelated parts of the store change.1 Furthermore, it was designed with React's concurrent features in mind, explicitly built to handle common pitfalls like the "zombie child problem" and context loss between different React renderers, making it a robust choice for React 18+.1

### **2.2 Material-UI (MUI) v5: The Enterprise UI Foundation**

MUI stands as one of the most mature and comprehensive UI component libraries in the React ecosystem, offering a production-ready foundation for enterprise applications.

* **Ecosystem Overview:** The MUI ecosystem is composed of several key packages. **MUI Core** is the foundational library, providing a vast collection of pre-built components implementing Google's Material Design.25 It is complemented by  
  **MUI Base**, which offers "headless" (unstyled) components and hooks, and **MUI X**, a suite of advanced, commercially licensed components like data grids and charts.26 This stack is immensely popular, with millions of weekly npm downloads and a vast community of contributors, ensuring long-term viability.27
* **MUI v5 vs. v6 (Mid-2025 Status):** MUI v5 is the long-term stable (LTS) version and offers full support for React 18.5 While MUI v6 was released in 2024, its headline feature—a new zero-runtime styling engine called Pigment CSS—is entirely  
  **opt-in**.8 This means that Emotion remains the default styling engine, and the patterns established in v5 are fully supported and relevant. The v6 release contained minimal breaking changes, focusing primarily on removing deprecated APIs and minor quality-of-life improvements.29 For an enterprise team seeking maximum stability, building on MUI v5 is a secure and well-supported decision.
* **Joy UI: A Cautionary Tale:** Joy UI is another component library from MUI, offering a different design aesthetic. However, a critical factor for any enterprise team is the library's maintenance and future roadmap. Official MUI blog posts and documentation from late 2023 and onward explicitly state that **Joy UI development is on hold**, and the team's primary focus is on Material UI.7 A MUI team member further confirmed on public forums that for new projects, the recommended path is to use Material UI with a custom theme rather than adopting Joy UI.31 For an enterprise application with a multi-year lifespan, adopting a library with no guaranteed future development introduces an unacceptable level of risk.  
  **Therefore, this guide strongly advises against the use of Joy UI.** Design customization should be achieved via MUI Core's extensive theming capabilities.

### **2.3 Emotion: The Native Styling Engine**

Emotion is the default styling solution for MUI v5, offering a powerful and performant approach to CSS-in-JS.

* **Role in the Stack:** As of v5, MUI replaced its previous JSS-based styling solution with Emotion.5 Emotion is a library designed for writing CSS styles directly within JavaScript, providing predictable composition and a superior developer experience with features like source maps and labels.33
* **API Overview:** Emotion's flexibility is a key strength, offering two distinct APIs for styling. The styled() API is used to create new React components with encapsulated styles, similar in syntax to the popular styled-components library.35 The  
  css prop, however, allows developers to apply styles directly to any element as a prop, which is ideal for one-off or dynamic adjustments without the overhead of creating a new named component.34 MUI's  
  sx prop is a superset of the css prop, integrating it directly with the theme object.
* **Performance & Community Standing:** In the CSS-in-JS landscape, Emotion is highly regarded for its performance. Benchmarks and community analysis consistently show that Emotion has a smaller bundle size and faster runtime performance compared to styled-components, particularly in applications with many dynamic styles.37 It is actively maintained, has a strong community, and is considered an industry standard, as reflected in its inclusion in the State of CSS survey.33

### **2.4 Table: At-a-Glance Library Comparison (Mid-2025)**

| Feature | Zustand | Material-UI v5 (Core) | Emotion |
| --- | --- | --- | --- |
| **Primary Role** | Global & Local Client-State Management | Comprehensive UI Component Library | CSS-in-JS Styling Engine |
| **Bundle Size Impact** | Minimal (~1-2 kB) 18 | Medium (Tree-shakable) 11 | Small (~8-9 kB) 38 |
| **Key Strength** | Simplicity, performance, minimal API | Completeness, accessibility, theming 25 | Performance, DX, sx prop flexibility 6 |
| **2025 Status** | Actively Maintained, Growing 14 | Stable, Long-Term Support 5 | Actively Maintained, Industry Standard 33 |
| **Learning Curve** | Low 17 | Medium (due to large API surface) | Low-to-Medium |
| **Next.js App Router** | Requires per-request provider pattern 42 | Requires style registry (AppRouterCacheProvider) 43 | Integrated with MUI's provider 43 |

## **3. The Enterprise Best-Practices Playbook**

This playbook provides opinionated, actionable patterns for architecture, developer experience, performance, and testing. These guidelines are designed to establish a scalable and maintainable foundation for the enterprise application.

### **3.1 Architecture & Project Structure**

A well-defined project structure is crucial for long-term maintainability and team collaboration.

* **Zustand Slices Pattern:** As an application grows, a single, monolithic state store becomes difficult to manage. The officially recommended **slices pattern** addresses this by modularizing the store.1 Each feature or domain of the application should have its own "slice" file, which is a function that defines a piece of the state and its related actions. These individual slices are then composed into a single, unified store. This promotes separation of concerns and makes the state logic easier to reason about.
  + **Implementation:** Create individual slice files (e.g., userSlice.ts, cartSlice.ts). Each slice exports a createSlice function. A central store file (e.g., useBoundStore.ts) imports these slice creators and combines them.2
  + **Middleware:** **Crucially, middleware such as persist or devtools must be applied only to the final, combined store**, not to the individual slices, to ensure they function correctly across the entire state.2 The community library  
    zustand-slices can be considered for a more opinionated, TypeScript-friendly approach to this pattern.45
* **MUI Theming Structure:** To manage application-wide design tokens and styles, a dedicated /theme directory is recommended. This directory should contain:
  + theme.ts: The main theme file where createTheme is called and all parts are assembled.
  + palette.ts: Defines the application's color scheme.
  + typography.ts: Defines font families, sizes, and weights.
  + components.ts: Contains global style overrides for MUI components (e.g., default props and styles for all MuiButton instances). This centralization prevents style duplication and ensures design consistency.
* **Component Organization:** A feature-based directory structure (e.g., /components/authentication, /components/dashboard) is highly recommended over a type-based one (e.g., /components/buttons, /components/cards). Within these feature folders, MUI components should be wrapped in custom application-specific components. For example, instead of using <Button> directly throughout the app, create a <PrimaryButton> component that encapsulates specific variants, styles, and logic.
* **Next.js App Router Structure:** A logical folder structure for a Next.js App Router project is as follows:
  + /app: Contains all routes, layouts, and pages.
  + /components: For shared, reusable React components.
  + /lib or /stores: The designated location for Zustand store definitions and slice logic.46
  + /providers: A dedicated home for the client-side context providers required for both Zustand's SSR pattern and MUI/Emotion's style registry.42

### **3.2 Developer Experience (DX) & TypeScript Patterns**

Enforcing consistent, type-safe patterns is key to developer productivity and reducing bugs.

* **Type-Safe Zustand Stores:** TypeScript is a first-class citizen in Zustand.
  + **Standard Pattern:** The recommended way to define a store is using the curried create<State>()(...) syntax. This allows you to explicitly pass the store's type interface, ensuring full type safety for both state and actions.47
  + **State & Actions Interface:** For maximum clarity, define a single interface for your store that separates state properties from action functions.

TypeScript  
// types/auth.types.ts  
export interface AuthState {  
 accessToken: string | null;  
 isLoading: boolean;  
 error: string | null;  
}  
  
export interface AuthActions {  
 login: (credentials: Credentials) => Promise<void>;  
 logout: () => void;  
 setToken: (token: string | null) => void;  
}  
  
export type AuthStore = AuthState & AuthActions;

* + **combine Middleware:** Zustand offers a combine middleware as an alternative that can infer the state type, removing the need for the curried create<T>() call. However, this comes with a subtle type-safety trade-off: the get() function inside the action creator will be typed as returning only the initial state, not the full state including actions. This can lead to unsoundness if not handled carefully, so the explicit create<State>()(...) pattern is generally safer for enterprise teams.47
* **Idiomatic MUI & Emotion Styling:** A clear distinction between styling methods is essential for maintainability.
  + **sx Prop vs. styled() API:** This is one of the most important conventions to establish.
    - **Use the sx prop for one-off, instance-specific style overrides.** It is perfect for applying a margin, changing a color, or adjusting the padding on a single component instance where the style is not intended to be reused.6
    - **Use the styled('div')(...) API to create new, reusable, styled components.** This is the correct choice when styles are complex, need to be reused across the application, or represent a distinct component in your design system.10
  + **The sx Prop Complexity Rule:** The decision between these two approaches should not be arbitrary. If a developer finds themselves writing a large, complex object for an sx prop, it is a strong indicator that the styles are no longer a "one-off" tweak but represent a reusable architectural element. Therefore, the team should adopt a clear rule: **If an sx prop definition exceeds a predefined complexity (e.g., more than 5-7 properties, or contains nested selectors/media queries), it MUST be refactored into a dedicated styled() component.** This practice prevents the bloat of inline styles and promotes the organic growth of a clean, reusable component library.
  + **Theming Best Practices:** Leverage MUI's theme to its full potential. Use TypeScript's module augmentation to add custom tokens (e.g., theme.palette.custom.brandRed) to the theme for type-safe access. Define global component overrides in theme.components to set default props and styles (e.g., making all MuiButton components variant="contained" by default).48

### **3.3 Performance & Optimization**

High performance is not an accident; it results from deliberate architectural choices and disciplined coding practices.

* **Zustand: Slaying Re-Renders:** The most common performance pitfall with context-like state managers is causing unnecessary re-renders. Zustand provides the tools to avoid this, but they must be used correctly.3
  + **Rule 1: Use atomic selectors.** When a component only needs one or a few pieces of state, it should subscribe to each piece individually. This ensures the component only re-renders when the specific data it cares about changes.1
    - **Incorrect:** const { bears, honey } = useBearStore(); - This subscribes to the entire store and re-renders on *any* state change.
    - **Correct:** const bears = useBearStore((state) => state.bears); const honey = useBearStore((state) => state.honey); - These are two independent subscriptions. A change to honey will not re-render a component that only uses bears.
  + **Rule 2: Use useShallow for multi-property selection.** If a component truly needs to select multiple properties as an object, it will re-render on every state change because the selector returns a new object reference each time. To prevent this, wrap the selector in the useShallow hook provided by zustand/react/shallow. This tells Zustand to perform a shallow equality check on the values of the returned object, triggering a re-render only if the values themselves have changed.3
* **MUI: Taming the Bundle:**
  + **Rule: Never use barrel file imports.** To ensure effective tree-shaking and dramatically improve development server startup times, always use direct, path-based imports for MUI components and icons.11
    - **Correct:** import Button from '@mui/material/Button';
    - **Incorrect:** import { Button } from '@mui/material';
  + This rule should be enforced automatically using the no-restricted-imports ESLint rule.11

JSON  
//.eslintrc.json  
{  
 "rules": {  
 "no-restricted-imports": [  
 "error",  
 {  
 "patterns": ["@mui/\*/\*/\*", "!@mui/material/test-utils/\*"]  
 }  
 ]  
 }  
}

* + **Lazy Loading:** For components that are not visible on the initial page load (e.g., complex dialogs, modals, or components below the fold), use React.lazy and <Suspense> to code-split them into separate chunks. This reduces the initial JavaScript payload and improves load times.9
* **Emotion: Optimizing CSS-in-JS:**
  + **Rule 1: Define styles outside the render path.** Creating style objects or template strings inside a component's render function forces Emotion to re-parse and re-serialize them on every render. For static styles, define them once at the module level so they are processed only once.51
  + **Rule 2: Use the Babel plugin.** The @emotion/babel-plugin performs compile-time optimizations, such as minification and hoisting, which reduce runtime overhead and improve performance.33
  + **Rule 3: Prefer object styles.** Using object styles (e.g., css={{ color: 'blue' }}) over string templates (e.g., css\color: blue;``) allows for better static analysis, type checking with TypeScript, and more reliable minification by standard build tools.52
* **React 18 Concurrency and Store Tearing:**
  + **Store Tearing Explained:** "Tearing" is a visual artifact specific to concurrent rendering where the UI displays an inconsistent state. For example, one component might render with a new state value while another component on the same screen renders with the old value, creating a "torn" view for a brief moment. This can happen when an external state management library is not properly synchronized with React's concurrent rendering mechanism.12
  + **The useSyncExternalStore Solution:** This is a critical reason to use an up-to-date state manager. Zustand v4 and later are built to be compatible with React 18's concurrent features. It achieves this by internally using the useSyncExternalStore hook. This hook is provided by React specifically to allow external stores to subscribe to changes in a way that is guaranteed to be synchronous with React's render cycle, thus preventing tearing entirely.12 This is a built-in feature of modern Zustand versions and a key advantage over older libraries or custom solutions.

### **3.4 Testing Strategies**

A comprehensive testing strategy ensures application reliability and developer confidence. The recommended stack is Vitest (or Jest) as the test runner, JSDOM for the environment, and React Testing Library (RTL) for rendering and interaction.55

* **Testing MUI Components:** Follow MUI's official testing guidance: focus on testing user-facing behavior, not the internal implementation of MUI components.57
  + **Query by accessible roles, labels, and text.** For example, query for a button using screen.getByRole('button', { name: /submit/i }) instead of relying on a fragile CSS selector like .MuiButton-root. This makes tests more resilient to library updates.
  + Use @testing-library/user-event to simulate real user interactions, such as typing into a text field or selecting an option from an MUI <Select> component.58
* **Testing with Zustand:** Testing components connected to a global store can be challenging because it introduces a shared dependency that can contaminate tests.
  + **The Challenge:** Directly importing the useStore hook in a test file ties that test to the actual, global store instance. This makes it difficult to mock initial state for different test scenarios or assert that actions were called correctly, leading to flaky and coupled tests.59
  + **Recommended Pattern: Mocking the Store Module:** The most robust pattern is to mock the entire Zustand store module using vi.mock (for Vitest) or jest.mock. This allows each test suite or individual test to provide its own version of the store.
  + **Implementation:**
    1. In your test setup file (e.g., tests/setup.ts) or at the top of a test file, mock the store's path: vi.mock('@/stores/my-store');.
    2. Create a test utility function that allows you to easily provide mock state and actions for each test.
    3. In each test, call this utility to set up the specific state and mocked actions needed for that scenario. This ensures complete test isolation. The official Zustand documentation provides a recipe for automatically resetting all stores between tests, which is a highly recommended practice.55

TypeScript  
// \_\_tests\_\_/MyComponent.test.tsx  
import { render, screen } from '@testing-library/react';  
import userEvent from '@testing-library/user-event';  
import { vi } from 'vitest';  
import MyComponent from '@/components/MyComponent';  
import { useAuthStore, type AuthStore } from '@/stores/authStore';  
  
// Mock the entire store module  
vi.mock('@/stores/authStore');  
  
// Cast the mocked hook to its real type for type safety  
const mockedUseAuthStore = vi.mocked(useAuthStore);  
  
describe('MyComponent', () => {  
 const mockLogin = vi.fn();  
  
 beforeEach(() => {  
 // Reset mocks before each test  
 vi.clearAllMocks();  
  
 // Provide a default mock implementation for the store  
 mockedUseAuthStore.mockReturnValue({  
 accessToken: null,  
 isLoading: false,  
 error: null,  
 login: mockLogin,  
 logout: vi.fn(),  
 setToken: vi.fn(),  
 });  
 });  
  
 it('should display login button when not authenticated', () => {  
 render(<MyComponent />);  
 expect(screen.getByRole('button', { name: /Log In/i })).toBeInTheDocument();  
 });  
  
 it('should display user info when authenticated', () => {  
 // Override the mock for this specific test  
 mockedUseAuthStore.mockReturnValue({  
 accessToken: 'fake-token-123',  
 isLoading: false,  
 error: null,  
 login: mockLogin,  
 logout: vi.fn(),  
 setToken: vi.fn(),  
 });  
  
 render(<MyComponent />);  
 expect(screen.getByText(/Welcome, User!/i)).toBeInTheDocument();  
 });  
  
 it('should call the login action when login button is clicked', async () => {  
 render(<MyComponent />);  
 const loginButton = screen.getByRole('button', { name: /Log In/i });  
 await userEvent.click(loginButton);  
 expect(mockLogin).toHaveBeenCalledTimes(1);  
 });  
});

## **4. Integration Recipes**

This section provides production-ready, fully commented code snippets for the most critical integration points of the stack, focusing on a Next.js App Router environment.

### **4.1 Recipe 1: The Ultimate Next.js App Router Setup**

This recipe establishes the foundational provider structure required for server-side rendering with MUI, Emotion, and Zustand.

**1. Zustand Store Factory (/stores/counter-store.ts)**

This file defines a *factory function* that creates a new store instance for each server request, which is essential to prevent state sharing between users.42

TypeScript

// src/stores/counter-store.ts  
import { createStore } from 'zustand/vanilla';  
  
export interface CounterState {  
 count: number;  
}  
  
export interface CounterActions {  
 increment: () => void;  
 decrement: () => void;  
}  
  
export type CounterStore = CounterState & CounterActions;  
  
// Default initial state for the store  
export const defaultInitState: CounterState = {  
 count: 0,  
};  
  
// The factory function to create a new store instance  
export const createCounterStore = (  
 initState: CounterState = defaultInitState,  
) => {  
 return createStore<CounterStore>((set) => ({  
 ...initState,  
 increment: () => set((state) => ({ count: state.count + 1 })),  
 decrement: () => set((state) => ({ count: state.count - 1 })),  
 }));  
};

**2. MUI/Emotion Style Registry (/theme/ThemeRegistry.tsx)**

This is the official MUI boilerplate for ensuring Emotion styles are correctly extracted on the server and injected into the <head> during SSR, preventing style flashes and hydration errors.43

TypeScript

// src/theme/ThemeRegistry.tsx  
'use client';  
import React, { useState } from 'react';  
import createCache from '@emotion/cache';  
import { useServerInsertedHTML } from 'next/navigation';  
import { CacheProvider } from '@emotion/react';  
import { ThemeProvider } from '@mui/material/styles';  
import CssBaseline from '@mui/material/CssBaseline';  
import { createTheme } from '@mui/material/styles'; // Your custom theme  
  
// A basic theme example  
const theme = createTheme({  
 palette: {  
 mode: 'light',  
 },  
});  
  
export default function ThemeRegistry({ children }: { children: React.ReactNode }) {  
 const [{ cache, flush }] = useState(() => {  
 const cache = createCache({ key: 'mui-styles' });  
 cache.compat = true;  
 const prevInsert = cache.insert;  
 let inserted: string =;  
 cache.insert = (...args) => {  
 const serialized = args;  
 if (cache.inserted[serialized.name] === undefined) {  
 inserted.push(serialized.name);  
 }  
 return prevInsert(...args);  
 };  
 const flush = () => {  
 const prevInserted = inserted;  
 inserted =;  
 return prevInserted;  
 };  
 return { cache, flush };  
 });  
  
 useServerInsertedHTML(() => {  
 const names = flush();  
 if (names.length === 0) {  
 return null;  
 }  
 let styles = '';  
 for (const name of names) {  
 styles += cache.inserted[name];  
 }  
 return (  
 <style  
 key={cache.key}  
 data-emotion={`${cache.key} ${names.join(' ')}`}  
 dangerouslySetInnerHTML={{  
 \_\_html: styles,  
 }}  
 />  
 );  
 });  
  
 return (  
 <CacheProvider value={cache}>  
 <ThemeProvider theme={theme}>  
 <CssBaseline />  
 {children}  
 </ThemeProvider>  
 </CacheProvider>  
 );  
}

**3. Zustand Provider (/providers/ZustandProvider.tsx)**

This client component creates the store instance and provides it via React Context, making it accessible to all child client components.42

TypeScript

// src/providers/ZustandProvider.tsx  
'use client';  
import { type ReactNode, createContext, useRef, useContext } from 'react';  
import { type StoreApi, useStore } from 'zustand';  
import { type CounterStore, createCounterStore } from '@/stores/counter-store';  
  
export const CounterStoreContext = createContext<StoreApi<CounterStore> | null>(null);  
  
export interface CounterStoreProviderProps {  
 children: ReactNode;  
}  
  
export const CounterStoreProvider = ({ children }: CounterStoreProviderProps) => {  
 const storeRef = useRef<StoreApi<CounterStore>>();  
 if (!storeRef.current) {  
 storeRef.current = createCounterStore();  
 }  
  
 return (  
 <CounterStoreContext.Provider value={storeRef.current}>  
 {children}  
 </CounterStoreContext.Provider>  
 );  
};  
  
// Custom hook to access the store safely from client components  
export const useCounterStore = <T,>(selector: (store: CounterStore) => T): T => {  
 const counterStoreContext = useContext(CounterStoreContext);  
  
 if (!counterStoreContext) {  
 throw new Error('useCounterStore must be used within a CounterStoreProvider');  
 }  
  
 return useStore(counterStoreContext, selector);  
};

**4. Root Layout (/app/layout.tsx)**

Finally, the root layout composes these providers to wrap the entire application.

TypeScript

// src/app/layout.tsx  
import type { Metadata } from 'next';  
import { Inter } from 'next/font/google';  
import ThemeRegistry from '@/theme/ThemeRegistry';  
import { CounterStoreProvider } from '@/providers/ZustandProvider';  
  
const inter = Inter({ subsets: ['latin'] });  
  
export const metadata: Metadata = {  
 title: 'Enterprise React App',  
 description: 'Built with Zustand, MUI, and Emotion',  
};  
  
export default function RootLayout({  
 children,  
}: Readonly<{  
 children: React.ReactNode;  
}>) {  
 return (  
 <html lang="en">  
 <body className={inter.className}>  
 <CounterStoreProvider>  
 <ThemeRegistry>{children}</ThemeRegistry>  
 </CounterStoreProvider>  
 </body>  
 </html>  
 );  
}

### **4.2 Recipe 2: A Type-Safe, Scalable Zustand Slice**

This recipe demonstrates the recommended "slices pattern" with full TypeScript support, including an asynchronous action.

TypeScript

// src/stores/slices/sessionSlice.ts  
import type { StateCreator } from 'zustand';  
import { api } from '@/lib/api'; // Assuming a pre-configured API client  
  
// Define the shape of the state and actions for this slice  
export interface SessionSlice {  
 token: string | null;  
 status: 'idle' | 'loading' | 'authenticated' | 'error';  
 error: string | null;  
 login: (credentials: { user: string; pass: string }) => Promise<void>;  
 logout: () => void;  
}  
  
// The StateCreator function that defines the slice  
export const createSessionSlice: StateCreator<  
 SessionSlice,  
 , // No custom middleware mutators needed for this slice  
 ,  
 SessionSlice  
> = (set, get) => ({  
 token: null,  
 status: 'idle',  
 error: null,  
 login: async (credentials) => {  
 // Use `get()` to read current state before an async operation  
 if (get().status === 'loading') return;  
  
 set({ status: 'loading', error: null });  
 try {  
 const response = await api.post('/login', credentials);  
 // Update state on success  
 set({ token: response.data.token, status: 'authenticated' });  
 } catch (err: any) {  
 // Update state on failure  
 set({ status: 'error', error: err.message, token: null });  
 }  
 },  
 logout: () => {  
 // Simple synchronous action  
 set({ token: null, status: 'idle', error: null });  
 },  
});  
  
// To be combined in a central store like this:  
// import { create } from 'zustand';  
// import { createSessionSlice, SessionSlice } from './slices/sessionSlice';  
//  
// export const useBoundStore = create<SessionSlice>()((...a) => ({  
// ...createSessionSlice(...a),  
// }));

### **4.3 Recipe 3: Advanced Theming and Component Styling**

This recipe shows how to customize the MUI theme and create a reusable styled component that consumes theme tokens.

**1. Advanced Theme Definition (/theme/theme.ts)**

TypeScript

// src/theme/theme.ts  
import { createTheme, PaletteOptions } from '@mui/material/styles';  
import { red } from '@mui/material/colors';  
  
// Augment the palette to include custom colors with TypeScript support  
declare module '@mui/material/styles' {  
 interface CustomPalette {  
 brand: PaletteOptions['primary'];  
 }  
 interface Palette extends CustomPalette {}  
 interface PaletteOptions extends CustomPalette {}  
}  
  
export const theme = createTheme({  
 palette: {  
 primary: {  
 main: '#556cd6',  
 },  
 secondary: {  
 main: '#19857b',  
 },  
 error: {  
 main: red.A400,  
 },  
 // Add custom color tokens  
 brand: {  
 main: '#ff5722',  
 contrastText: '#fff',  
 },  
 },  
 typography: {  
 fontFamily: '"Roboto", "Helvetica", "Arial", sans-serif',  
 h1: {  
 fontSize: '2.5rem',  
 },  
 },  
 // Global component style overrides  
 components: {  
 MuiButton: {  
 defaultProps: {  
 disableElevation: true,  
 variant: 'contained',  
 },  
 styleOverrides: {  
 root: {  
 textTransform: 'none',  
 borderRadius: '8px',  
 },  
 },  
 },  
 },  
});

**2. Custom Styled Component (/components/common/BrandButton.tsx)**

This component uses the styled API and consumes the custom brand color from the theme.

TypeScript

// src/components/common/BrandButton.tsx  
import { styled } from '@mui/material/styles';  
import Button, { ButtonProps } from '@mui/material/Button';  
  
// Create a new component using the `styled` API  
export const BrandButton = styled(Button)<ButtonProps>(({ theme }) => ({  
 // Consume the custom theme token  
 backgroundColor: theme.palette.brand.main,  
 color: theme.palette.brand.contrastText,  
 padding: theme.spacing(1, 4), // Use theme spacing for consistency  
 '&:hover': {  
 backgroundColor: theme.palette.brand.dark, // MUI automatically generates dark/light variants  
 },  
 // Example of a responsive style  
 [theme.breakpoints.down('sm')]: {  
 padding: theme.spacing(1, 2),  
 fontSize: '0.875rem',  
 },  
}));

## **5. Migration Checklist**

This section provides high-level, actionable checklists for teams migrating from common legacy technologies to the recommended stack.

### **5.1 From Redux Toolkit to Zustand**

This migration involves a conceptual shift from the structured, boilerplate-heavy nature of Redux to Zustand's minimal, hook-based approach.

* [ ] **Conceptual Shift:** Transition from the Redux pattern (actions -> dispatch -> reducers -> store update) to Zustand's more direct model where action functions within the store call set() to update state.17
* [ ] **Store Setup:** Replace configureStore, reducers, and the top-level <Provider> component with a single create() hook that defines the store. For SSR apps, implement the per-request provider pattern detailed in Recipe 1.
* [ ] **Component Connection:** Convert useSelector hooks to Zustand's useStore(state =>...) syntax. Remember to use atomic selectors or useShallow for performance.
* [ ] **Asynchronous Logic:** Replace Redux Thunks or Sagas with simple async functions defined directly within the Zustand store. These functions can perform asynchronous operations and then call set() upon completion.1
* [ ] **DevTools:** Swap the Redux DevTools Extension setup for Zustand's devtools middleware, which integrates with the same browser extension.17

### **5.2 From styled-components to Emotion**

This is primarily a syntactic migration, as Emotion's styled() API was heavily inspired by styled-components and is nearly identical.35

* [ ] **Dependency Swap:** Run npm uninstall styled-components and npm install @emotion/react @emotion/styled.
* [ ] **Import Changes:** Perform a project-wide search-and-replace for import styled from 'styled-components' to import styled from '@emotion/styled'.
* [ ] **ThemeProvider:** If used, change the import from styled-components to @emotion/react.
* [ ] **css Prop:** For any existing usage of a css helper function, ensure it is imported from @emotion/react.

### **5.3 From MUI v4 to v5**

This is the most significant migration, involving a new styling engine and numerous breaking changes. Using the official codemods is highly recommended to automate the bulk of the work.5 The process should be done in phases with commits after each major step.

* **Phase 1: Preparation**
  + [ ] **Update React & TypeScript:** Ensure React is at least ^17.0.0 and TypeScript is at least ^3.5.5
  + [ ] **Root ThemeProvider:** Verify that <ThemeProvider> from @material-ui/core/styles wraps the application root, as @mui/styles (the JSS compatibility layer) requires it.5
* **Phase 2: Dependency Update**
  + [ ] **Install v5 Packages:** npm install @mui/material @mui/styles @emotion/react @emotion/styled.
  + [ ] **Install Ancillary Packages:** If used, install the v5 versions: npm install @mui/lab @mui/icons-material. Note that date pickers move to @mui/x-date-pickers.5
  + [ ] **Uninstall v4 Packages:** Once the app runs with the new dependencies, safely npm uninstall @material-ui/\*.5
* **Phase 3: Automated Codemods (Critical Step)**
  + [ ] **Run preset-safe:** This is the most important codemod. It handles the vast majority of breaking changes, including import path updates from @material-ui to @mui, component prop renames, and more. Run npx @mui/codemod v5.0.0/preset-safe./ on your source directory.48
  + [ ] **Run variant-prop:** The default variant for TextField, Select, etc., changed from standard to outlined. This codemod adds variant="standard" to maintain the v4 appearance. Run npx @mui/codemod v5.0.0/variant-prop./.48
  + [ ] **Run link-underline-hover:** The default for <Link> underline changed from hover to always. This codemod applies underline="hover" to preserve the v4 behavior. Run npx @mui/codemod v5.0.0/link-underline-hover./.5
* **Phase 4: Manual Fixes & Incremental Migration**
  + [ ] **Theme Structure:** Manually refactor your createMuiTheme object. theme.overrides becomes theme.components.ComponentName.styleOverrides, and theme.props becomes theme.components.ComponentName.defaultProps.49 The  
    adaptV4Theme helper can assist with this transition.49
  + [ ] **Styling Engine:** The application will now use Emotion as the default engine, but your existing JSS styles (makeStyles, withStyles) will continue to work via the @mui/styles package. Plan an incremental migration to move these styles to the styled() API or sx prop over time.5
  + [ ] **Component-Specific Changes:** Review the official MUI v4 to v5 migration guide for any component-specific breaking changes not handled by the codemods (e.g., changes to Autocomplete's renderOption prop or Dialog's transition props).48

## **6. Annotated Bibliography**

Biedermann, K. (2024, May 22). *The best CSS one-liner to improve performance in 2025*. Medium. <https://medium.com/@karstenbiedermann/the-best-css-one-liner-to-improve-performance-in-2025-b12eb5419a84> 64

*Annotation:* Provides context on modern CSS performance properties like contain-intrinsic-size, which are relevant to optimizing layout performance in component-heavy applications.

Caisy. (n.d.). *Emotion vs styled-components*. Caisy.io. Retrieved June 5, 2025, from <https://caisy.io/blog/emotion-vs-styled-components> 37

*Annotation:* A key comparative analysis confirming Emotion's performance advantages over styled-components in bundle size and runtime speed, supporting its selection as the preferred styling engine.

da Silva, L. A. P. (n.d.). *A simple React todo list with Zustand, Material UI and persisting data with Zustand midleware*. Medium. Retrieved June 5, 2025, from <https://leandroaps.medium.com/a-simple-react-todo-list-with-zustand-material-ui-and-persisting-data-with-zustand-midleware-341d727e69c8> 65

*Annotation:* Offers a practical example of integrating the core technologies, demonstrating the use of Zustand's persist middleware alongside MUI components.

Department Agency. (n.d.). *Optimizing Emotion in React*. Retrieved June 5, 2025, from <https://engineering.deptagency.com/optimizing-emotion-in-react> 52

*Annotation:* This source provides deep technical insights into Emotion's runtime behavior and offers critical optimization advice, such as defining styles outside the render path and preferring object styles.

Dev.to Community. (2024, May 1). *Avoid performance issues when using Zustand*. <https://dev.to/devgrana/avoid-performance-issues-when-using-zustand-12ee> 3

*Annotation:* This article clearly articulates the most common performance pitfall in Zustand—creating new object references in selectors—and explains the correct use of atomic selectors and the useShallow hook.

Emotion Team. (n.d.-a). *Emotion Documentation: Best Practices*. Emotion.sh. Retrieved June 5, 2025, from <https://emotion.sh/docs/best-practices> 53

*Annotation:* Official documentation outlining best practices for co-locating styles, sharing styles via component reuse, and using TypeScript with object styles for improved developer experience.

Emotion Team. (n.d.-b). *Emotion Documentation: Performance*. Emotion.sh. Retrieved June 5, 2025, from <https://emotion.sh/docs/performance> 51

*Annotation:* The official performance guide for Emotion, which recommends profiling and provides specific optimization techniques, including using the Babel plugin and reducing render frequency.

Emotion Team. (n.d.-c). *Emotion Documentation: Server-Side Rendering*. Emotion.sh. Retrieved June 5, 2025, from <https://emotion.sh/docs/ssr> 66

*Annotation:* Canonical documentation for Emotion's SSR APIs, such as extractCritical, which forms the basis of MUI's integration with Next.js.

Gamre, P. (2024, April 26). *How to Build a Multi-Step Form in React Using React-Hook-Form, Zod Validation, and Zustand*. Medium. <https://alexdln.medium.com/react-conf-2024-76aabd9da1e1> 67

*Annotation:* A summary of React Conf 2024, highlighting key upcoming features in React 19 like the use hook and compiler, providing future context for the ecosystem.

MUI Team. (n.d.-a). *Joy UI - Overview*. MUI.com. Retrieved June 5, 2025, from <https://mui.com/joy-ui/getting-started/> 7

*Annotation:* This is the primary source confirming that Joy UI's development is "on hold." This official statement is the critical piece of evidence for advising against its use in an enterprise setting.

MUI Team. (n.d.-b). *Material UI - Interoperability*. MUI.com. Retrieved June 5, 2025, from <https://mui.com/material-ui/integrations/interoperability/> 68

*Annotation:* Official guide explaining how to handle CSS injection order when using Emotion with MUI, a crucial step for resolving style specificity conflicts.

MUI Team. (n.d.-c). *Material UI - Migration from v4 to v5*. MUI.com. Retrieved June 5, 2025, from <https://mui.com/material-ui/migration/migration-v4/> 5

*Annotation:* The canonical, multi-part guide for migrating from MUI v4. It provides the foundational steps, dependency changes, and codemod instructions that form the basis of the migration checklist in this report.

MUI Team. (n.d.-d). *Material UI - Minimizing Bundle Size*. MUI.com. Retrieved June 5, 2025, from <https://mui.com/material-ui/guides/minimizing-bundle-size/> 11

*Annotation:* The official guide on bundle size optimization for MUI, which mandates the use of path-based imports over barrel imports—a critical performance best practice.

MUI Team. (n.d.-e). *Material UI - Next.js App Router Integration*. MUI.com. Retrieved June 5, 2025, from <https://mui.com/material-ui/integrations/nextjs/> 43

*Annotation:* The official and essential recipe for integrating MUI v5 and Emotion with the Next.js App Router, detailing the use of AppRouterCacheProvider.

MUI Team. (n.d.-f). *Material UI - Testing*. MUI.com. Retrieved June 5, 2025, from <https://mui.com/material-ui/guides/testing/> 57

*Annotation:* Provides the recommended approach for testing MUI components, emphasizing querying by accessible roles over implementation details to create robust tests.

MUI Team. (n.d.-g). *MUI Core v5 migration update*. MUI Blog. Retrieved June 5, 2025, from <https://mui.com/blog/mui-core-v5-migration-update/> 6

*Annotation:* A blog post reinforcing the benefits of migrating to v5, including React 18 support, the new Emotion style engine, and the powerful sx prop.

Poimandres. (n.d.-a). *Zustand Documentation*. GitHub. Retrieved June 5, 2025, from <https://github.com/pmndrs/zustand>

*Annotation:* The primary documentation for Zustand, detailing its core API, philosophy, and best practices like atomic selection.

Poimandres. (n.d.-b). *Zustand Documentation: Next.js*. Zustand Docs. Retrieved June 5, 2025, from <https://zustand.docs.pmnd.rs/guides/nextjs> 42

*Annotation:* This is the canonical source for integrating Zustand with Next.js. It provides the foundational per-request provider pattern that is essential for preventing state leakage in server-rendered applications.

Poimandres. (n.d.-c). *Zustand Documentation: Slices Pattern*. Zustand Docs. Retrieved June 5, 2025, from <https://zustand.docs.pmnd.rs/guides/slices-pattern> 2

*Annotation:* Official documentation for the recommended "slices pattern" for modularizing a Zustand store, a key architectural pattern for scalable applications.

Poimandres. (n.d.-d). *Zustand Documentation: Testing*. Zustand Docs. Retrieved June 5, 2025, from <https://zustand.docs.pmnd.rs/guides/testing> 55

*Annotation:* The official testing guide, which recommends React Testing Library and provides a recipe for resetting stores between tests to ensure isolation.

Poimandres. (n.d.-e). *Zustand Documentation: TypeScript*. Zustand Docs. Retrieved June 5, 2025, from <https://zustand.docs.pmnd.rs/guides/typescript> 47

*Annotation:* The complete official guide for using Zustand with TypeScript, explaining the create<T>()(...) syntax and patterns for typing middleware.

Poimandres. (n.d.-f). *Zustand GitHub Discussion #885*. GitHub. Retrieved June 5, 2025, from <https://github.com/pmndrs/zustand/discussions/885> 12

*Annotation:* A key discussion where a maintainer confirms that Zustand v4 uses useSyncExternalStore to avoid tearing in React 18, a critical point for ensuring concurrency safety.

Rheinwerk Computing. (2023, July 10). *CSS in JavaScript Using Emotion*. <https://blog.rheinwerk-computing.com/css-in-javascript-using-emotion> 36

*Annotation:* Provides a clear explanation of Emotion's dual APIs (css prop and styled approach) and the necessary Babel/TypeScript configuration to enable them.

S., M. (2024, May 22). *How to Optimize Material-UI Performance in Large-Scale React Applications*. Dev.to. <https://dev.to/syed_mudasseranayat_e251/how-to-optimize-material-ui-performance-in-large-scale-react-applications-1imd> 9

*Annotation:* A practical guide that collates several MUI performance best practices, including using the sx prop over makeStyles, lazy loading components, and using virtualization for large lists.

Smith, T. (2024, June 1). *An essential Zustand test recipe*. Medium. <https://medium.com/@tts2p4/an-essential-zustand-test-recipe-43e5892d75cb> 69

*Annotation:* This article presents a robust, type-safe pattern for testing components that use Zustand by mocking the store module, which is the recommended approach in this guide for ensuring test isolation.

## **7. Appendix: Demo Repository**

A complete, runnable implementation of the best practices and integration recipes detailed in this report can be found at the following GitHub repository:

<https://github.com/ldw-templates/nextjs-mui-zustand-template> 70

This starter template is based on Next.js 15 App Router, MUI v6 (which is backward compatible with v5 patterns), and Zustand v5, providing a practical and verifiable example of the recommended architecture.

#### Works cited

1. pmndrs/zustand: Bear necessities for state management in React - GitHub, accessed July 6, 2025, <https://github.com/pmndrs/zustand>
2. Slices Pattern - Zustand, accessed July 6, 2025, <https://zustand.docs.pmnd.rs/guides/slices-pattern>
3. Avoid performance issues when using Zustand - DEV Community, accessed July 6, 2025, <https://dev.to/devgrana/avoid-performance-issues-when-using-zustand-12ee>
4. Prevent rerenders with useShallow - Zustand, accessed July 6, 2025, <https://zustand.docs.pmnd.rs/guides/prevent-rerenders-with-use-shallow>
5. Migrating to v5: getting started - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/migration/migration-v4/>
6. Why you should migrate to Material UI v5 today - MUI, accessed July 6, 2025, <https://mui.com/blog/mui-core-v5-migration-update/>
7. Joy UI - Overview - MUI, accessed July 6, 2025, <https://mui.com/joy-ui/getting-started/>
8. The road to Material UI v6 and beyond - MUI, accessed July 6, 2025, <https://mui.com/blog/2023-material-ui-v6-and-beyond/>
9. How to Optimize Material-UI Performance in Large-Scale React Applications, accessed July 6, 2025, <https://dev.to/syed_mudasseranayat_e251/how-to-optimize-material-ui-performance-in-large-scale-react-applications-1imd>
10. Mastering Material UI Customization in React: Best Practices and Choosing the Right Approach | by Sushmitha Dhummi Thrilochana | Stackademic, accessed July 6, 2025, <https://blog.stackademic.com/mastering-material-ui-customization-in-react-best-practices-and-choosing-the-right-approach-9fdfb81a74c2>
11. Minimizing bundle size - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/guides/minimizing-bundle-size/>
12. Is Zustand 3.7.1 Compatible with React 18? · pmndrs zustand ..., accessed July 6, 2025, <https://github.com/pmndrs/zustand/discussions/885>
13. Prevent Screen Tearing for React 18 in a Zustand-like App with useSyncExternalStore, accessed July 6, 2025, <https://egghead.io/lessons/react-prevent-screen-tearing-for-react-18-in-a-zustand-like-app-with-usesyncexternalstore>
14. zustand · GitHub Topics, accessed July 6, 2025, <https://github.com/topics/zustand>
15. Zustand, accessed July 6, 2025, <https://zustand-demo.pmnd.rs/>
16. React State Management — using Zustand | by Chikku George | Globant - Medium, accessed July 6, 2025, <https://medium.com/globant/react-state-management-b0c81e0cbbf3>
17. Day 25: Zustand vs Redux — Mastering State in React + Next.js Apps, accessed July 6, 2025, <https://javascript.plainenglish.io/day-25-zustand-vs-redux-mastering-state-in-react-next-js-apps-3c00ad743ec4>
18. Exploring Zustand - A Modern Approach to State Management in React - MoldStud, accessed July 6, 2025, <https://moldstud.com/articles/p-exploring-zustand-a-modern-approach-to-state-management-in-react>
19. Zustand vs. Redux Toolkit vs. Jotai | Better Stack Community, accessed July 6, 2025, <https://betterstack.com/community/guides/scaling-nodejs/zustand-vs-redux-toolkit-vs-jotai/>
20. betterstack.com, accessed July 6, 2025, <https://betterstack.com/community/guides/scaling-nodejs/zustand-vs-redux-toolkit-vs-jotai/#:~:text=Redux%20Toolkit%20is%20structured%20and,grained%20control%20and%20strong%20performance.>
21. Redux vs Zustand — A Practical Comparison with Code (2025) | by Mohit Decodes, accessed July 6, 2025, <https://medium.com/@mohitdecodes/redux-vs-zustand-a-practical-comparison-with-code-2025-2ae78268efd9>
22. 2024 JavaScript Rising Stars, accessed July 6, 2025, <https://risingstars.js.org/2024/en>
23. 2023 JavaScript Rising Stars, accessed July 6, 2025, <https://risingstars.js.org/2023/en>
24. Zustand: Introduction, accessed July 6, 2025, <https://zustand.docs.pmnd.rs/>
25. Material UI - Overview - MUI, accessed July 6, 2025, <https://mui.com/material-ui/getting-started/>
26. hg2355/epic-store: MUI Core is a collection of React UI libraries for shipping new features faster. Start with Material UI, our fully-loaded component library, or bring your own design system to our production-ready components. - GitHub, accessed July 6, 2025, <https://github.com/hg2355/epic-store>
27. Ready-to-use React components, free forever - MUI Core, accessed July 6, 2025, <https://mui.com/core/>
28. mui/mui-x: MUI X: Build complex and data-rich applications using a growing list of advanced React components, like the Data Grid, Date and Time Pickers, Charts, and more! - GitHub, accessed July 6, 2025, <https://github.com/mui/mui-x>
29. Upgrade to v6 - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/migration/upgrade-to-v6/>
30. @mui/joy | Yarn, accessed July 6, 2025, <https://classic.yarnpkg.com/en/package/@mui/joy>
31. Material UI v6 is out now - MUI : r/reactjs - Reddit, accessed July 6, 2025, <https://www.reddit.com/r/reactjs/comments/1f2d191/material_ui_v6_is_out_now_mui/>
32. @mui/material - npm, accessed July 6, 2025, <https://www.npmjs.com/package/@mui/material>
33. emotion-js/emotion: ‍ CSS-in-JS library designed for high performance style composition - GitHub, accessed July 6, 2025, <https://github.com/emotion-js/emotion>
34. Emotion – Introduction, accessed July 6, 2025, <https://emotion.sh/docs/introduction>
35. Styled Components - Emotion, accessed July 6, 2025, <https://emotion.sh/docs/styled>
36. CSS in JavaScript Using Emotion - The Rheinwerk Computing Blog, accessed July 6, 2025, <https://blog.rheinwerk-computing.com/css-in-javascript-using-emotion>
37. Styled Components/ Emotion compared - Caisy, accessed July 6, 2025, <https://caisy.io/blog/emotion-vs-styled-components>
38. jsjoeio/styled-components-vs-emotion - GitHub, accessed July 6, 2025, <https://github.com/jsjoeio/styled-components-vs-emotion>
39. Benchmarks - Stitches.dev, accessed July 6, 2025, <https://stitches.dev/docs/benchmarks>
40. @emotion/react - npm, accessed July 6, 2025, <https://www.npmjs.com/package/@emotion/react>
41. CSS-in-JS - State of CSS 2023, accessed July 6, 2025, <https://2023.stateofcss.com/en-US/css-in-js/>
42. Setup with Next.js - Zustand, accessed July 6, 2025, <https://zustand.docs.pmnd.rs/guides/nextjs>
43. Next.js integration - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/integrations/nextjs/>
44. Zustand with typescript, slicing and persisting middleware #2027 - GitHub, accessed July 6, 2025, <https://github.com/pmndrs/zustand/discussions/2027>
45. zustandjs/zustand-slices: A slice utility for Zustand - GitHub, accessed July 6, 2025, <https://github.com/zustandjs/zustand-slices>
46. Mastering State Management with Zustand in Next.js and React - DEV Community, accessed July 6, 2025, <https://dev.to/mrsupercraft/mastering-state-management-with-zustand-in-nextjs-and-react-1g26>
47. TypeScript Guide - Zustand, accessed July 6, 2025, <https://zustand.docs.pmnd.rs/guides/typescript>
48. Migration from v4 to v5 - MUI, accessed July 6, 2025, <https://v5-0-6.mui.com/guides/migration-v4/>
49. Breaking changes in v5, part one: styles and themes - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/migration/v5-style-changes/>
50. 5 Zustand BEST Practices in 5 Minutes - YouTube, accessed July 6, 2025, <https://www.youtube.com/watch?v=6tEQ1nJZ51w&pp=0gcJCfwAo7VqN5tD>
51. Performance - Emotion, accessed July 6, 2025, <https://emotion.sh/docs/performance>
52. Optimizing the use of Emotion in React - Engineering at DEPT, accessed July 6, 2025, <https://engineering.deptagency.com/optimizing-emotion-in-react>
53. Best Practices - Emotion, accessed July 6, 2025, <https://emotion.sh/docs/best-practices>
54. Why useSyncExternalStore Is Not Used in Jotai - Daishi Kato's blog, accessed July 6, 2025, <https://blog.axlight.com/posts/why-use-sync-external-store-is-not-used-in-jotai/>
55. Testing - Zustand, accessed July 6, 2025, <https://zustand.docs.pmnd.rs/guides/testing>
56. Unit Testing React Components with Jest and React Testing Library - DEV Community, accessed July 6, 2025, <https://dev.to/debajit13/unit-testing-react-components-with-jest-and-react-testing-library-3he>
57. Testing - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/guides/testing/>
58. React testing library on change for Material UI Select component - Stack Overflow, accessed July 6, 2025, <https://stackoverflow.com/questions/55184037/react-testing-library-on-change-for-material-ui-select-component>
59. A Zustand Test Recipe I'm Finding Useful : r/reactjs - Reddit, accessed July 6, 2025, <https://www.reddit.com/r/reactjs/comments/1h6udk9/a_zustand_test_recipe_im_finding_useful/>
60. Next.js App Router - Joy UI - MUI, accessed July 6, 2025, <https://mui.com/joy-ui/integrations/next-js-app-router/>
61. zustand - NPM, accessed July 6, 2025, <https://www.npmjs.com/package/zustand>
62. How to Migrate from Material-UI 4 to 5 - Victor Abz, accessed July 6, 2025, <https://victor-abz.hashnode.dev/how-to-migrate-from-material-ui-4-to-5>
63. Migrating Your Project from MUI4 to MUI5: A Comprehensive Guide - Medium, accessed July 6, 2025, <https://medium.com/@123vikas.vp79/migrating-your-project-from-mui4-to-mui5-a-comprehensive-guide-ddf4c620fae5>
64. The Best CSS One-Liner to Improve Performance in 2025 | by Karsten Biedermann, accessed July 6, 2025, <https://medium.com/@karstenbiedermann/the-best-css-one-liner-to-improve-performance-in-2025-b12eb5419a84>
65. A simple React Todo List with Zustand, Material/UI and persisting data with Zustand Midleware | by Leandro A. Siqueira, accessed July 6, 2025, <https://leandroaps.medium.com/a-simple-react-todo-list-with-zustand-material-ui-and-persisting-data-with-zustand-midleware-341d727e69c8>
66. Server Side Rendering - Emotion, accessed July 6, 2025, <https://emotion.sh/docs/ssr>
67. React Conf 2024. React v19 RC - Alex Savelyev - Medium, accessed July 6, 2025, <https://alexdln.medium.com/react-conf-2024-76aabd9da1e1>
68. Style library interoperability - Material UI - MUI, accessed July 6, 2025, <https://mui.com/material-ui/integrations/interoperability/>
69. An Essential Zustand Test Recipe. Intro | by Trevor Smith | Medium, accessed July 6, 2025, <https://medium.com/@tts2p4/an-essential-zustand-test-recipe-43e5892d75cb>
70. ldw-templates/nextjs-mui-zustand-template: Best practices with next.js, mui and zustand., accessed July 6, 2025, <https://github.com/ldw-templates/nextjs-mui-zustand-template>