State Management in React: Context Providers & Zustand (with TypeScript)

1. Problem Statement

Case Study: TaskFlow Project Management

TaskFlow is a project management app:

- Users can add, edit, and complete tasks from anywhere in the app.
- The theme (light/dark) and user profile must be available globally.
- As the app grows, prop drilling and context performance become issues.
- The team wants a state management solution that is **type-safe**, easy to test, and scales from small to large features.

The challenge:
How do you architect state management so that global data (user, theme, tasks) is accessible, type-safe, and performant—without unnecessary re-renders or boilerplate?

---

2. Learning Objectives

By the end of this tutorial, you will:

- Understand when to use Context Providers vs. Zustand.
- Create a type-safe React Context Provider.
- Build and use a Zustand store with TypeScript.
- Combine Context and Zustand for scalable, maintainable state.
- Avoid common pitfalls (re-renders, type errors, boilerplate).

---

3. Concept Introduction with Analogy

Analogy: The TaskFlow Control Center

- **Context Providers** are like a central PA system: announcements (state) are broadcast to all rooms (components) that listen, but every room hears every change.
- **Zustand** is like a smart intercom: each room subscribes only to the messages it cares about, and only those rooms react when something changes—saving energy and avoiding noise.

---

4. Technical Deep Dive

A. Context Providers with TypeScript

When to Use Context

- Best for **global, rarely-changing state** (theme, auth, locale).
- Avoid for large, frequently-changing data (e.g., task lists).

Type-Safe Context Example: Theme

```tsx
import React, { useState, useContext } from 'react';

// 1. Define the context type
interface ThemeContextType {
 theme: 'light' | 'dark';
 toggleTheme: () => void;
}

// 2. Create the context
const ThemeContext = React.createContext<ThemeContextType | undefined>(undefined);

// 3. Provider implementation
export const ThemeProvider = ({ children }) => {
 const [theme, setTheme] = useState<'light' | 'dark'>('light');
 const toggleTheme = () => setTheme(t => (t === 'light' ? 'dark' : 'light'));

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

// 4. Custom hook for safe consumption
export const useTheme = () => {
 const context = useContext(ThemeContext);
 if (!context) throw new Error('useTheme must be used within ThemeProvider');
 return context;
};
```

Usage:

```tsx
import { useTheme } from './ThemeProvider';

function ThemeSwitcher() {
 const { theme, toggleTheme } = useTheme();
 return (
   <button onClick={toggleTheme}>
     Switch to {theme === 'light' ? 'dark' : 'light'}
   </button>
 );
}
```

Key Points:

- Types are enforced at every level.
- If a component is outside the provider, you get a clear error.
- No prop drilling—any component can access the theme.

B. Zustand: Modern, Type-Safe Global State

Why Zustand?

- Minimal API, no boilerplate, no reducers or providers needed.
- **Selective subscription:** Components only re-render for the state they use.
- Great TypeScript support out of the box.
- Handles async actions, middleware, and even persistence.

Install Zustand

```
`npm install zustand`
```

1. Define the State Interface

```
// store/userStore.ts
import { create } from 'zustand';

interface User {
 id: string;
 name: string;
 email: string;
}

interface UserStore {
 user: User | null;
 setUser: (user: User) => void;
 clearUser: () => void;
}

const useUserStore = create<UserStore>((set) => ({
 user: null,
 setUser: (user) => set({ user }),
 clearUser: () => set({ user: null }),
}));

export default useUserStore;
```

2. Using the Store in Components

```
import useUserStore from './store/userStore';

function Profile() {
 const user = useUserStore((state) => state.user);
 const clearUser = useUserStore((state) => state.clearUser);

 if (!user) return <div>Not logged in</div>;
 return (
   <div>
     <h2>{user.name} ({user.email})</h2>
```

```
      <button onClick={clearUser}>Logout</button>
    </div>
  );
}
```

- **Selector pattern:** Components only re-render when the selected state changes.
- **No context provider needed:** Just import and use the store anywhere.

3. Async Actions and Middleware

```
interface Task {
  id: string;
  title: string;
  completed: boolean;
}

interface TaskStore {
  tasks: Task[];
  fetchTasks: () => Promise<void>;
  addTask: (title: string) => void;
}

const useTaskStore = create<TaskStore>((set) => ({
  tasks: [],
  fetchTasks: async () => {
    const response = await fetch('/api/tasks');
    const tasks = await response.json();
    set({ tasks });
  },
  addTask: (title) =>
    set((state) => ({
      tasks: [...state.tasks, { id: Date.now().toString(), title, completed: false }],
    })),
}));
```

C. Combining Context and Zustand

- Use **Context** for global app settings (theme, locale, auth).
- Use **Zustand** for business/domain state (tasks, projects, notifications).
- You can wrap Zustand stores in context if you want to provide custom hooks or middleware, but it's often not necessary.

D. Comparison and Best Practices

State Management: Context Provider vs Zustand

Feature	Context Provider	Zustand
Boilerplate	Medium (provider, hooks)	Minimal (just a hook)
Type Safety	Manual (define types)	Built-in via generics
Performance	Risk of over-render	Fine-grained subscriptions
Async Actions	Manual (custom logic)	Native support
Persistence	Manual	Built-in (middleware)

Feature	Context Provider	Zustand
Best For	Theme, locale, auth	Tasks, user, large state

5. Step-by-Step Data Modeling & Code Walkthrough

A. User Context Provider (Theme Example)

```tsx
// context/ThemeContext.tsx
import React, { useState, useContext } from 'react';

interface ThemeContextType {
 theme: 'light' | 'dark';
 toggleTheme: () => void;
}

const ThemeContext = React.createContext<ThemeContextType | undefined>(undefined);

export const ThemeProvider: React.FC<{ children: React.ReactNode }> = ({ children }) => {
 const [theme, setTheme] = useState<'light' | 'dark'>('light');
 const toggleTheme = () => setTheme(t => (t === 'light' ? 'dark' : 'light'));
 return (
   <ThemeContext.Provider value={{ theme, toggleTheme }}>
     {children}
   </ThemeContext.Provider>
 );
};

export const useTheme = () => {
 const ctx = useContext(ThemeContext);
 if (!ctx) throw new Error('useTheme must be used within ThemeProvider');
 return ctx;
};
```

B. Zustand Store for Tasks

```tsx
// store/taskStore.ts
import { create } from 'zustand';

interface Task {
id: string;
title: string;
completed: boolean;
}

interface TaskStore {
tasks: Task[];
addTask: (title: string) => void;
toggleTask: (id: string) => void;
}

const useTaskStore = create<TaskStore>((set) => ({
tasks: [],
addTask: (title) =>
  set((state) => ({
    tasks: [...state.tasks, { id: Date.now().toString(), title, completed: false }],
  })),
toggleTask: (id) =>
  set((state) => ({
    tasks: state.tasks.map((task) =>
      task.id === id ? { ...task, completed: !task.completed } : task
    ),
  })),
}));
```

```
export default useTaskStore;
```

C. Using Zustand Store in Components

```
import useTaskStore from './store/taskStore';

function TaskList() {
const tasks = useTaskStore((state) => state.tasks);
const toggleTask = useTaskStore((state) => state.toggleTask);

return (
 <ul>
   {tasks.map((task) => (
     <li key={task.id}>
       <label>
         <input
           type="checkbox"
           checked={task.completed}
           onChange={() => toggleTask(task.id)}
         />
         {task.title}
       </label>
     </li>
   ))}
 </ul>
);
}
```

6. Interactive Challenge / Mini-Project

Your Turn!

1. Create a Zustand store for notifications:
    - Each notification has `id`, `message`, `type` (`'info' | 'error' | 'success'`), and `read: boolean`.
    - Add actions: `addNotification`, `markAsRead`, and `clearNotifications`.
2. Use the store in a NotificationList component to display unread notifications and mark them as read.

7. Common Pitfalls & Best Practices

Common Pitfalls & Best Practices (Context vs Zustand)

Pitfall	Best Practice
Using Context for large, changing state	Use Zustand for business/domain state
Not typing store/actions	Always type state and actions for safety
Unnecessary re-renders in Context	Use Zustand’s selectors for performance
Mixing concerns in one store	Split stores by domain (user, tasks, etc.)
Not using custom hooks for Context	Always wrap context in a custom hook

8. Optional: Programmer’s Workflow Checklist

- Use Context for global, rarely-changing settings.
- Use Zustand for business/domain state (tasks, notifications, etc.).
- Define TypeScript interfaces for all state and actions.
- Use selectors in Zustand to avoid unnecessary re-renders.
- Test stores independently from UI components.