1. Problem Statement

Case Study: DesignHub – Real-Time Collaborative Design

DesignHub is a Figma-like platform where:

- Users can create, edit, and comment on design files in real time.
- Each feature (files, users, comments, notifications, preferences) is managed by a different team.
- State must be modular: easy to test, maintain, and scale as features grow.
- Performance is critical—only components using changed state should re-render.
- Teams want to use middleware (devtools, persistence, logging) on specific state slices.

CASE STUDY:
ARCHITECTING A MODULAR GLOBAL STATE SYSTEM FOR A COLLABORATIVE DESIGN PLATFORM
DESIGNHUB

The challenge:
How do you architect a global state system that is **modular, type-safe, and scalable**—so each feature team can own their slice, and the app remains fast and maintainable?

2. Learning Objectives

By the end of this tutorial, you will:

- Understand what Zustand slices are and why they matter.
- Architect modular state using slices for different features.
- Combine slices into a single store with full type safety.

- Apply middleware to specific slices or the whole store.
- Test and maintain slices independently.
- Avoid pitfalls like tight coupling and unnecessary re-renders.

---

3. Concept Introduction with Analogy

Analogy: The DesignHub Control Tower

- **Slices** are like specialized teams in a control tower: one team manages flights, one manages weather, one manages communications.
- Each team (slice) has its own dashboard, rules, and logs—but they all work together in the same tower (store).
- If the weather team updates a forecast, only the weather dashboard changes—not the flight or comms dashboards.

---

4. Technical Deep Dive

A. What Are Zustand Slices?

- A **slice** is a function that returns a piece of state and its actions, with its own types.
- Slices are composed together to create the full store.
- Each slice can have its own middleware, selectors, and tests.

Why slices?

- Modularity: Each feature owns its state logic.
- Scalability: Add or remove features without touching unrelated code.
- Testability: Test slices in isolation.
- Performance: Components subscribe only to the state they use.

B. Defining Slices: Example Types

```
// store/slices/userSlice.ts
export interface UserSlice {
  user: { id: string; name: string } | null;
  setUser: (user: { id: string; name: string }) => void;
  clearUser: () => void;
}

export const createUserSlice = (set) => ({
  user: null,
  setUser: (user) => set({ user }),
  clearUser: () => set({ user: null }),
});

// store/slices/fileSlice.ts
export interface File {
  id: string;
  name: string;
  content: string;
}
export interface FileSlice {
  files: File[];
  addFile: (file: File) => void;
  updateFile: (id: string, content: string) => void;
}

export const createFileSlice = (set, get) => ({
  files: [],
  addFile: (file) => set((state) => ({ files: [...state.files, file] })),
  updateFile: (id, content) =>
    set((state) => ({
      files: state.files.map((f) =>
        f.id === id ? { ...f, content } : f
      ),
    })),
});
```

C. Combining Slices into a Single Store

```
import { create } from 'zustand';
import { devtools, persist } from 'zustand/middleware';
import { createUserSlice, UserSlice } from './slices/userSlice';
import { createFileSlice, FileSlice } from './slices/fileSlice';

type DesignHubStore = UserSlice & FileSlice;

export const useDesignHubStore = create<DesignHubStore>()(
  devtools(
    persist(
      (set, get) => ({
        ...createUserSlice(set, get),
        ...createFileSlice(set, get),
        // Add more slices here
      }),
      { name: 'designhub-store' }
    )
  )
);
```

- **Order matters:** Middleware like `devtools` and `persist` can wrap the whole store or individual slices.

D. Using Slices in Components

```
import { useDesignHubStore } from './store';

function UserProfile() {
  const user = useDesignHubStore((s) => s.user);
  const setUser = useDesignHubStore((s) => s.setUser);

  if (!user) return <button onClick={() => setUser({ id: 'u1', name: 'Alex' })}>Login</button>;
  return <div>Welcome, {user.name}</div>;
}

function FileList() {
  const files = useDesignHubStore((s) => s.files);
  const addFile = useDesignHubStore((s) => s.addFile);

  return (
    <div>
      <button onClick={() => addFile({ id: Date.now().toString(), name: 'New', content: '' })}>
        Add File
      </button>
      <ul>
```

```
        {files.map((f) => (
          <li key={f.id}>{f.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

E. Testing and Maintaining Slices

- Slices can be tested independently by calling their factory functions with mock `set` and `get`.
- Example (Jest)

```
import { createUserSlice } from './userSlice';

test('setUser sets user', () => {
  let state = { user: null };
  const set = (fn) => { state = { ...state, ...fn(state) }; };
  const slice = createUserSlice(set);
  slice.setUser({ id: 'u2', name: 'Sam' });
  expect(state.user.name).toBe('Sam');
});
```

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

A. Create Feature Slices

```
// store/slices/commentSlice.ts
export interface Comment {
  id: string;
  fileId: string;
  author: string;
  text: string;
}
export interface CommentSlice {
  comments: Comment[];
  addComment: (comment: Comment) => void;
  getCommentsByFile: (fileId: string) => Comment[];
}
export const createCommentSlice = (set, get) => ({
  comments: [],
  addComment: (comment) => set((state) => ({ comments: [...state.comments, comment] })),
  getCommentsByFile: (fileId) => get().comments.filter((c) => c.fileId === fileId),
});
```

B. Combine All Slices in the Store

```
import { create } from 'zustand';
import { devtools, persist } from 'zustand/middleware';
import { createUserSlice, UserSlice } from './slices/userSlice';
import { createFileSlice, FileSlice } from './slices/fileSlice';
import { createCommentSlice, CommentSlice } from './slices/commentSlice';

type DesignHubStore = UserSlice & FileSlice & CommentSlice;

export const useDesignHubStore = create<DesignHubStore>()(
  devtools(
    persist(
      (set, get) => ({
        ...createUserSlice(set, get),
        ...createFileSlice(set, get),
        ...createCommentSlice(set, get),
      }),
      { name: 'designhub-store' }
    )
  )
);
```

C. Using Slices in the App

```
function CommentsPanel({ fileId }) {
  const comments = useDesignHubStore((s) => s.getCommentsByFile(fileId));
  const addComment = useDesignHubStore((s) => s.addComment);

  return (
    <div>
      <ul>
        {comments.map((c) => (
          <li key={c.id}>{c.author}: {c.text}</li>
        ))}
      </ul>
      <button onClick={() => addComment({ id: Date.now().toString(), fileId, author: 'Alex', text: 'Hello!'
        Add Comment
      </button>
    </div>
  );
}
```

6. Interactive Challenge / Mini-Project

Your Turn!

1. Create a `notificationsSlice`:
   - Fields: `notifications: { id: string; message: string; read: boolean }[]`
   - Actions: `addNotification`, `markAsRead`, `clearNotifications`
2. Add the slice to the main store.
3. Build a `NotificationsPanel` component that displays unread notifications and lets users mark them as read.

7. Common Pitfalls & Best Practices

Common Pitfalls & Best Practices (Zustand Slices)

Pitfall	Best Practice
Mixing unrelated state in one slice	Keep slices focused on a single feature
Tight coupling between slices	Use actions/selectors, not direct state access
Not typing slices	Always define interfaces for each slice
Middleware order mistakes	Apply `devtools` / `persist` after combining slices
Not testing slices independently	Test each slice with mock `set` / `get`

8. Optional: Programmer's Workflow Checklist

- Define an interface and factory for each slice.
- Combine slices in the main store with middleware.
- Use selectors to subscribe only to needed state.
- Test slices in isolation with mock set/get.
- Document slice boundaries and responsibilities.