Hooks, Forms & Side Effects

React + TypeScript (Vite)

Why Hooks & Side Effects Matter

Modern React applications must:

- Manage component logic cleanly
- Handle async operations (API calls)
- Respond to user input efficiently

Hooks enable:

- Logic reuse
- Cleaner components
- Predictable lifecycle behavior

What Are Hooks?

Hooks are functions that let you “hook into” React features from function components.

Key Characteristics:

- Only work in function components
- Replace class-based lifecycle logic
- Encourage composition over inheritance

📌 Hooks simplify state, lifecycle, and logic reuse

Rules of Hooks

To ensure predictable behavior:

- Call hooks only at the top level
- Call hooks only inside React functions
- Never call hooks inside loops or conditions

Why:

- React relies on hook call order

useState Hook (Quick Recap)

Purpose:

- Manage local component state

Common Use Cases:

- Form values
- Toggle states
- Counters
- UI flags (loading, error)

TypeScript Advantage:

- Enforces valid state values
- Prevents accidental misuse

useEffect Hook

useEffect is used to handle side effects.

Examples of Side Effects:

- API calls
- Subscriptions
- Timers
- DOM interactions

📌 Effects run after render

useEffect Dependencies

Dependency Array Controls:

- When the effect runs
- Prevents unnecessary executions

Common Patterns:

- Empty array → run once
- Specific variables → run on change
- No array → run every render (usually a mistake)

Side Effects with TypeScript

Benefits of TypeScript:

- Typed API responses
- Safer async operations
- Clear cleanup logic

Best Practices:

- Always handle cleanup
- Avoid side effects in render logic
- Keep effects focused and small

Common useEffect Pitfalls

❌ Missing dependency array
❌ Infinite re-render loops
❌ Mixing multiple responsibilities in one effect
❌ Ignoring cleanup functions

✅ One effect → one responsibility
✅ Depend on stable references

Forms in React

Forms are controlled using:

- State
- Event handlers

Types of Forms:

- Controlled forms (recommended)
- Uncontrolled forms

📌 Controlled forms provide better validation and predictability

Controlled Forms with TypeScript

Key Concepts:

- Form values stored in state
- Inputs derive value from state
- Changes handled via events

TypeScript Benefits:

- Typed form data
- Safer validations
- Reduced runtime errors

Handling Form Events

Common events:

- onChange
- onSubmit
- onBlur

Best Practices:

- Prevent default submission
- Centralize form state
- Validate before submission

Form Validation Strategies

Validation Approaches:

- Inline validation
- Schema-based validation
- Custom validation logic

Considerations:

- User experience
- Error feedback
- Accessibility

Custom Hooks

Custom Hooks:

- Extract reusable logic
- Share behavior across components
- Improve code readability

Naming Convention:

- Must start with use

📌 Custom hooks are key to scalable architecture

Anti-Patterns to Avoid

❌ Too many states instead of a single object
❌ Side effects inside event handlers only
❌ Overusing effects for simple logic
❌ Creating hooks inside components

✅ Keep hooks reusable
✅ Keep logic isolated

Integration Points & Dependencies

Important Integration Areas:

- API services
- Environment configs (Vite)
- Form libraries (optional)
- Backend validation

Design Impact:

- Hook reusability
- Error handling strategy
- Performance considerations

Preparing Assets for Capstone

You should prepare:

- Custom hooks (API, auth, forms)
- Typed form models
- Effect-driven data loaders
- Reusable validation logic

These assets will be reused in:

👉 Final Capstone Application

Summary

- Hooks manage logic and lifecycle
- Forms capture and validate user input
- Side effects handle async & external operations
- TypeScript enforces correctness

This module enables real-world React behavior management