# Props, State & Component Composition

React with TypeScript using Vite

# Learning Outcomes

By the end of this module, you will be able to:

- Explain Props, State, and Component Composition in React with TypeScript
- Apply these concepts to real-world UI problems
- Prepare reusable components for a capstone project

# Why Props, State & Composition Matter

- React applications are built from small, reusable components
- Data flow must be predictable and type-safe
- Poor component design leads to:
  - Tight coupling
  - Difficult debugging
  - Poor scalability

👉 Props, State & Composition are the foundation of clean React architecture

# What Are Props?

Props (Properties) are read-only inputs passed from parent to child components.

Key Characteristics:

- Immutable (cannot be modified by child)
- Enable component reuse
- Support one-way data flow

📌 Think of props as function parameters for components.

# Props with TypeScript

TypeScript ensures type safety for props.

Benefits:

- Prevents invalid data passing
- Improves IDE autocomplete
- Makes components self-documenting

Best Practice:

- Always define a Props interface or type
- Avoid using any

# Props Example (Conceptual)

- Parent component passes data
- Child component receives data via props
- Child cannot modify the data

Example Use Cases:

- Passing labels, titles, IDs
- Passing callback functions
- Passing configuration flags

# Common Props Pitfalls

- ❌ Mutating props inside a component
- ❌ Passing too many unrelated props
- ❌ Using any instead of typed interfaces
- ❌ Deeply nested prop drilling

- ✅ Prefer clear prop contracts
- ✅ Use composition to reduce prop depth

# What Is State?

State represents mutable data managed inside a component.

Key Characteristics:

- Managed using React Hooks (useState)
- Changes cause re-render
- Local to the component by default

📌 State controls dynamic UI behavior

# State with TypeScript

Why TypeScript with State?

- Prevents invalid state values
- Makes state transitions predictable
- Reduces runtime bugs

Best Practices:

- Explicitly type complex state
- Keep state minimal and meaningful

# State Example Scenarios

Typical state use cases:

- Form input values
- Toggle (open / close)
- Counters and selections
- API loading states

Rule of Thumb:

- If data changes over time and affects UI → it belongs in state

# Props vs State

| Props | State |
| --- | --- |
| Passed from parent | Managed inside component |
| Read-only | Mutable |
| External data | Internal data |
| Improves reuse | Controls behavior |

# Lifting State Up

Problem:

- Multiple components need the same data

Solution:

- Move state to the nearest common parent
- Pass data & handlers via props

Benefits:

- Single source of truth
- Predictable data flow

# What Is Component Composition?

Component Composition means:

- Building complex UIs using small, reusable components
- Passing components as children or props

React favors composition over inheritance

# Composition Patterns

Common patterns:

- Children Props
- Layout Components
- Reusable UI Containers
- Controlled Components

Benefits:

- Cleaner code
- Easier testing
- Better scalability

# Composition with Children

Instead of hardcoding UI:

- Parent defines layout
- Children inject content

Use Cases:

- Cards
- Modals
- Page layouts
- Tabs

📌 Promotes flexibility without increasing props

# Real-World Mapping

| Concept | Real-World Example |
| --- | --- |
| Props | User profile data |
| State | Login status |
| Composition | Dashboard layout |
| Lifted State | Shared filters |
| Children | Page content |

# Anti-Patterns to Avoid

❌ Large “God components”

❌ Mixing UI logic with business logic

❌ Excessive prop drilling

❌ Storing derived data in state

✅ Break components early

✅ Keep logic reusable

✅ Use composition wisely

# Design Decisions & Dependencies

Things to consider:

- Component reusability
- Prop responsibility boundaries
- State ownership
- TypeScript strictness

Dependencies Impact:

- Vite → fast dev rebuilds
- TypeScript → stricter design choices
- React Hooks → functional composition

# Preparing for the Capstone

Assets you should prepare:

- Reusable typed components
- Shared layout components
- Clean prop contracts
- Well-managed local state

These will directly feed into:

👉 Capstone Project Architecture

# Module Summary

- Props define what a component receives
- State defines how a component behaves
- Composition defines how components collaborate
- TypeScript enforces correctness and clarity

This module builds the core mental model for scalable React apps.