React Best Practices

Gain mastery of the world's most popular and powerful front-end library

Shaun Wassell

Software Developer, Educator

Pearson

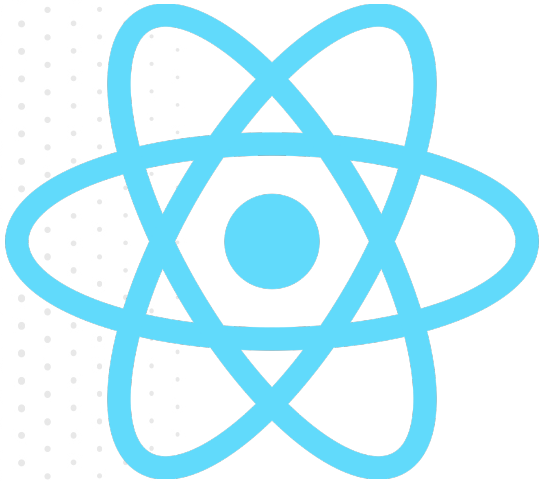
What We'll Be Covering

- Component Best Practices
- Project Structure and Organization Best Practices
- Routing and Navigation Best Practices
- Performance Optimization Best Practices
- Security Best Practices

Q & A

Please submit your questions during the lecture via the Q&A widget

Questions will be answered after the next break



Section 1: Component Best Practices

- Smart vs. Dumb Components
- Functional components with hooks vs. class components
- Composition over inheritance
- Prop types and default props
- Pure components

Smart vs. Dumb Components

- "Smart" Components:
  - Manage state, logic, and data sources
  - Less reusable, handles business logic and state
- "Dumb" Components
  - Focus on UI, receive data via props
  - Highly reusable, ideal for UI elements

When to Use Smart vs. Dumb Components

- "Smart" Components - Maximize Control:
  - State Management (a ListFilter component)
  - Data Fetching (a Page component)
  - Data Distribution (a Container component)
- "Dumb" Components - Maximize Reusability:
  - Presentational Purposes (a UserListItem component)
  - Reusable UI Elements (Buttons, Text Inputs, etc.)

Pure/Memoized Components

- Implements shallow prop/state comparison automatically
- Best for simple props/state, improves performance
- Minimizes re-renders, suitable for static data
- Not ideal for deep nested object changes.
- Use by extending React.PureComponent or React.memo

Function vs. Class Components

- Class Components:
  - Use ES6 classes, built-in state and lifecycle methods
  - Verbose but clear structure
  - Easier to onboard OOP devs
- Function Components:
  - Plain JS functions, use hooks for state & lifecycle
  - Simpler, more readable
  - Recommended for most cases
- The verdict: use Classes for complex lifecycle needs, functions for everything else

Use Composition, Not Inheritance

- Inheritance - A "is a" B
- Composition - A "has a" B
- Composition:
  - simplifies maintenance vs. complex inheritance hierarchies
  - Supports customization without altering component internals
  - Aligns with React’s design philosophy for component reuse.

Prop Types and Default Props

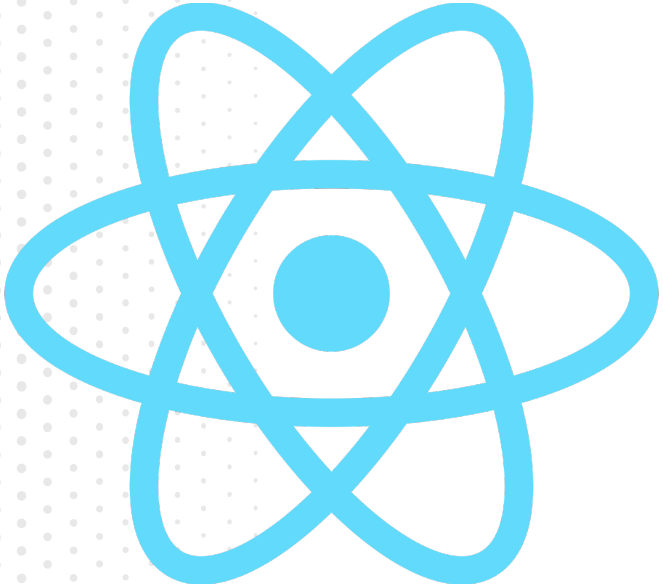
- PropTypes:
  - Enforce type safety, catching bugs early.
  - Enhances documentation/readability, clarifies component interfaces
  - Development tool only, no production performance penalty
- DefaultProps:
  - Ensure consistent behavior and prevent undefined errors

Break Time!

Q & A

Please submit your questions via the Q&A widget

Questions submitted now will be answered after the next break



Section 2: Project Structure and Organization Best Practices

- Component folder structure - Feature- vs. Function-based
- Keeping component files lean
- Helper/util functions and where to put them

Feature- vs. Function-based Folder Structure

- Function-Based Structure:
  - Organizes by file type, i.e. "server endpoints", "database schemas", "pages", "lists"
  - Pros: Easier file location, intuitive for newcomers
  - Cons: Can lead to coupling, complicates scaling

Feature- vs. Function-based Folder Structure

- Feature-Based Structure:
  - Groups by feature/domain, enhancing modularity
  - I.e. "Products", "Users", "Articles"
  - Pros: Improves cohesion, isolates development/testing, facilitates reuse
  - Cons: May result in deep nesting, complex inter-feature communication

Keep Component Files Lean

- Obey the Single Responsibility Principle
- What code is reused? Extract it
- Use custom hooks for logic/state, keeping files clean
- Avoid inline styles/logic, use separate files instead
- Separate utilities/data fetching, focus on component role

Where Should I Put Helpers/Util?

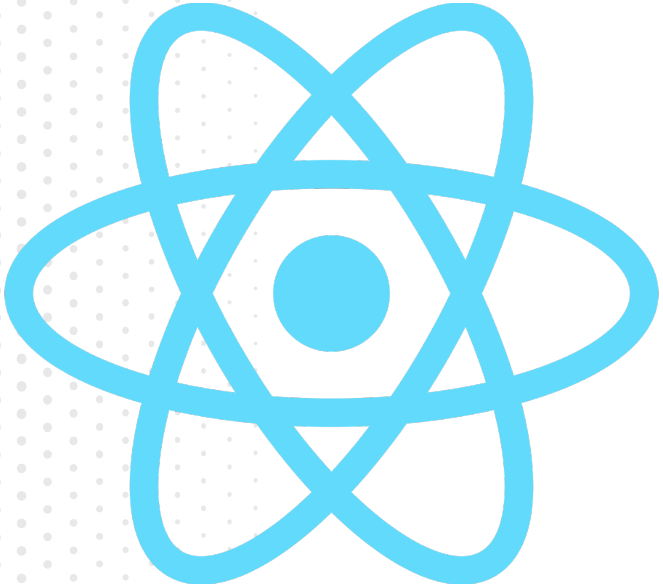
- Centralize in utils/helpers directory for easy access
- Categorize by functionality, easing maintenance and discovery
- Emphasize pure functions for reliability and testability
- Document functions clearly for long-term maintainability
- Ensure reliability with unit tests for utility functions

Break Time!

Q & A

Please submit your questions via the Q&A widget

Questions submitted now will be answered after the next break



Section 3: Routing and Navigation Best Practices

- React Router best practices
- Dynamic routing and lazy-loaded routes
- Route guards and authentication flow

React Router Best Practices

- Keep Your Route Definitions Centralized
- Use Human-Readable, Semantic URLs
- Use Nested Routes for Hierarchical Pages
- Protect Private Routes

Lazy-Loading Routes

- Why load the entire site before the user can see anything?
- Reduces initial load time and uses resources more efficiently
- Consider pre-loading components when a user looks like they're going to need them
- Use Error handling to handle loading errors gracefully

Route Guards and Authentication Flow

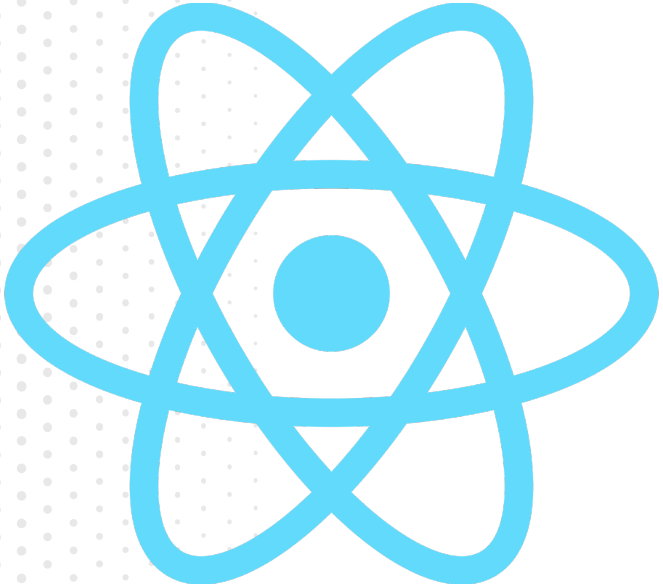
- Control access to routes based on auth status
- This doesn't *really* stop users, you need to do that on the back-end
- Use HOCs/hooks for component protection
- Store auth status securely in client-side storage (i.e. HTTP-only cookies)
- Use Context for global auth state management

Break Time!

Q & A

Please submit your questions via the Q&A widget

Questions submitted now will be answered after the next break



Section 4: Performance Optimization Best Practices

- Using React.memo, useMemo, and useCallback
- Virtualized lists for performance
- Profiling and debugging with React DevTools
- Avoiding common pitfalls (e.g., unnecessary renders)

Use React.memo, useMemo, and useCallback

- React.memo prevents re-renders with unchanged props
- useMemo avoids recalculating data on every render
- useCallback memoizes function props to prevent unnecessary re-renders
- Apply selectively to address proven performance bottlenecks
- Test impact for real performance benefits, avoiding overuse

Use Virtualized Lists

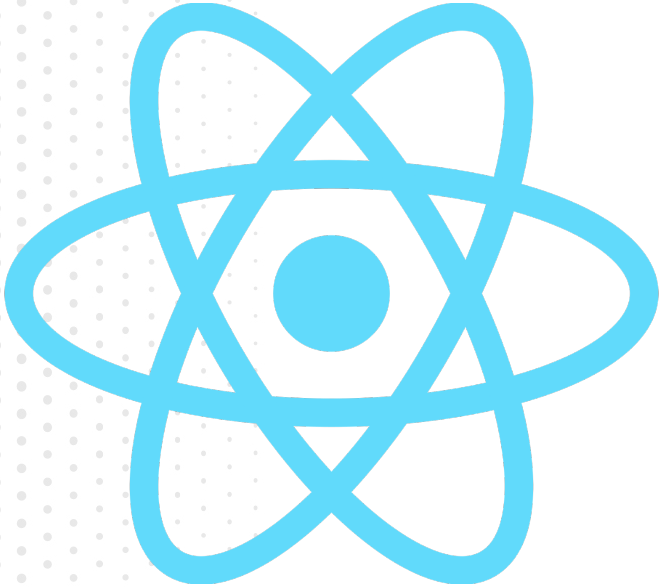
- Why render all the items in a huge list when the user is only going to see a small part of it?
- Only render the items that fit in the user's viewport
- Some libraries recycle DOM nodes
- Most popular libraries include react-virtualized, react-window, and react-infinite-loader

Use React DevTools for Profiling & Debugging

- Inspect component hierarchy, state, and props for debugging
- Profile to identify and analyze render performance bottlenecks
- Inspect custom hooks to understand and debug their logic

Avoid Common Pitfalls

- Forgetting to Clean Up in useEffect
- Misusing Refs
- Over-reliance on Third-Party Libraries
- Not Planning for State Management Strategy
- Skipping tests



Section 5: Security Best Practices

- Protecting against cross-site scripting (XSS)
- Safe data handling and avoiding exposing sensitive data

Protect Against XSS

- XSS allows hackers to run arbitrary code on your site.

  This can lead to:

  - Stealing user credentials
  - Stealing personal information
  - Run crypto miners in users' browsers
- Never trust user-defined values
- Watch out for third-party libraries

Handle Sensitive Data Safely

- Use HTTPS for API Calls
- Avoid Storing Sensitive Data in State
- Don't Put API Keys, etc. in Front-end Code
- Sanitize User Input on the Back-end - don't trust the front-end
- Security audits aren't a bad idea

Q & A

This is the last Q&A session