ReactJSConsulting

reactjsconsulting.com

React Consulting and Training by Cory House, tailored for your team.

55 Lessons From 5 Years in React

Getting familiar with a new project

1. Review package.json first to understand deps, scripts, and config.
2. Draw tree on the whiteboard, or use React dev tools. Helps to visualize state.

Dev workflow

3. Quickly nav to component or func: CMD click in JSX (VSCode)
4. Quickly nav to parent: CMD+SHIFT+F to Search for <ComponentName (VSCode)
5. Quickly view list of parents: Use React dev tools
6. Create a component state checklist. Use it for every component. (error, no data, lots of data, long values, full list here)
7. Debugging? type debugger. console.assert also handy.
8. Work against mock data and mock API (I like json-server, json-graphql-server)
9. Centralize mock data for Storybook, tests, and mock API.
10. Pass an object to a function and destructure to create named parameters. Reads more clearly. And can destructure in the function signature too. This keeps the calls in the func short and documents the expected object properties.
11. Storybook driven development – Build and test each component in isolation. Document each state in a separate story. Then use Percy or Chromatic to snapshot.
    a. Knobs
    b. Responsive design

JSX

12. You can only write expressions within a return. This limits what you can do in JSX. Options:
    a. Return early. (good for loaders and errors)
    b. Extract a separate function from render when you want the full power of JavaScript (if/else/switch)

Performance

13. Do the most convenient thing. It'll probably be fast enough. Inline func? Fine. Worried about renders? Don't be. Worried about context performance? Okay, then maybe you're misusing context (should rarely change). ☺ Sure, perf test (set Chrome perf to 6x), but don't speculate. Degrade ergonomics after establishing a perf issue.

14. Remember, a render != DOM change. With virtual DOM, the diff is in-memory. Flow: render -> reconcile -> commit. If the DOM doesn't change, there's likely no perf issue. So stop worrying about needless re-renders. React is smart enough to only change the DOM when needed, so it's typically fast enough.
15. Don't slap useMemo, shouldComponentUpdate, PureComponent everywhere. Only where needed. They have overhead because it's an extra diff. If they were typically faster, they'd be the default!

State management

16. Keep state as low as you can. Lift when needed.
17. Avoid state that can be derived. Calc on the fly. Reference objects by id instead of duplicating.
18. Use _myVar convention to resolve state naming conflicts.
19. Don't sync state, derive it. Example, calculate full name on the fly by concatenating firstName and lastName in render. Don't store fullName separately. Doing so risks out of sync issues and requires extra code to keep it in sync.
20. State that changes together, should live together. Reducers help. So does grouping via useState. Consider state machines – they describe valid states, which makes invalid state impossible (as new customer with 5 previous purchases, or an admin with no rights shouldn't be possible. If separate states, they can get outta sync)
21. Probably don't need Redux. Lifting state scales nicely and is easy to understand. Prop drilling pain is overblown. Keep prop names the same. Spread props. Pass child. Memoize. Use context and useReducer cover the rare global needs well. Show slides of diff data approaches from my updated Redux course.
22. Context isn't just useful for global data. Useful for compound components. Can be useful for performance.
23. setLoading(false) in finally to assure it's called

Props

24. Require all props at first
25. Destructure props in func signature to shorten calls below. Useful on event handler funcs too. But what about props with dashes in name like aria-label? Well, don't destructure that by using spread: ...otherProps
26. Make your props as specific as possible
27. Standardize naming. onX for eventHandler props. handleX for the func.
28. Centralize your propTypes
29. Document propTypes via JSDoc style comments = autocomplete and docs in Storybook. Can even use markdown!
30. Spread props or pass children to reduce the pain of prop drilling
31. Prop existence conveys truth. So <Input required /> is sufficient.
32. Honor the native API in your reusable component designs. Pass the full event to event handlers, not merely the value. Then you can use a centralized change handler. Honor

the native names (onBlur, onChange, etc). Doing so maximizes flexibility and minimizes the learning curve.

Styling

33. Mix styling approaches.
    a. Inline styles for dynamic styles.
    b. Namespace via CSS modules.
    c. Use plain Sass for global styles.
    d. CSS in JS remains a hard sell – too many horses in the race.
34. Use classnames to apply multiple styles
35. Use flexbox and CSS Grid over floating styles
36. Create abstraction over flexbox to abstract breakpoints for consistency (bootstrap gives ya this)

Reusable components

37. 3 keys to easy reuse
38. Consider dedicating a person/team to this. Why? Speed. Less decision fatigue. Smaller bundles. Consistency = better UX. Less code = fewer bugs.
39. Look for repeated code – opportunity for reuse. Every reuse is a perf enhancement.
40. DRY out your forms by combining custom hooks, context, and reusable components to create an opinionated custom approach that encapsulates your app's business rules. These tools are the foundation.
41. Accept both a simple string and an element. Use React.isValidElement to tell which you're getting.
42. Create an "as" prop to specify the top-level element.
43. Create a reusable AppLayout using the slot pattern.
44. Centralize alerts in AppLayout and provide function for showing the alerts via context.
45. Gen custom docs via react-docgen
46. Consider creating separate mobile and desktop components if they differ significantly. Lazy load the relevant size.

Testing

47. Prefer RTL over Enzyme. Simpler API = pit of success. Encourages a11y. Easy to debug. Can use same queries for Cypress.
48. JSDOM doesn't render, so can't test responsive design there. Use Cypress to test responsive design behavior.
49. Avoid Jest snapshot tests. They're brittle, they test implementation details, they're often poorly named, they all fail when a single line changes, and they're hard to fix when they fail. Instead, prefer Percy or Chromatic to test visuals
50. Use the scenario selector pattern to run your app against different data. Automate these tests via Cypress/Selenium
51. Use Cypress testing library so your Cy selectors match your React Testing library selectors = No need to change code to support Cypress tests!

52. Cypress driven development – TDD for integration testing. Use Cypress to navigate to the spot you need to test. Use cy.only to call a single test. It should fail first. Make it pass.

Dev env

53. Consider customizing create-react-app (CRA).
    a. Use react-app-rewired to tweak the config without ejecting
    b. Customize linting rules.
    c. Add webpack-bundle-analyzer. Know what's in your bundle.
    d. Consider forking at least react scripts. Consider forking CRA. Create a company framework that generates a project with a single dependency: Your react-scripts fork that includes your company's components, tweaks, dependencies, linting rules, webpack.config, etc.
54. Use Prettier. Convert in one big commit. You'll look like the hero!
55. Lean on ESLint. Use as a teaching tool. Object shorthand. No var. Disallow certain imports (jquery, lodash, moment). Require triple equals. Don't form a committee. Assign someone you trust and enable a lot of good stuff. Can always back off later. Add plugins like jsx-a11y/recommended.
56. Require strict propTypes (or TS). I don't get many type issues. (see link for list)
57. Use .vsextensions to encourage extensions.
58. Keep client and server separate. If embedding React in a server-side tech, use Storybook to develop components in isolation.

Consider a monorepo

59. Why? Rapid feedback. Continuous integration.
60. Easy reuse
61. CI integration tests projects on every PR
62. Use Lerna, Bolt, Yarn Workspaces, or even simply a relative file reference to your reusable components to manage. I typically use Lerna.

Learning

63. Have a system for organizing your knowledge. Find a new tool? Technique? Document it. I use GitHub issues here on my reactjsconsulting repo.

Would your team benefit from a React workshop?

I provide on-site and online consulting and training on React, tailored to your team.
Learn more at reactjsconsulting.com