Sofascore Frontend Academy Lecture 05,
March, 2024

Context.
More React hooks.
Performance

Darjan Baričević

Sofascore

TABLE OF CONTENTS

Sofascore

01

Context

Context

But first... project structure

- NOTE: This is highly opinionated topic, but can be a useful advice
- NOTE #2: We are migrating from this structure to Feature-sliced design
- pages - file-system based routing provided by Next.js - to be learned later
- model - types of basic entities obtained by backend API
- components - basic building blocks of application and more complex modules
  - e.g. Button, Checkbox, Accordion, Icons, ...
- api - contains route definitions and API methods (e.g. postLogin, getUser, ...)
- utils - common helpers, types, test boilerplate,...
  - e.g. numberSort, getJson, postJson, parseResponse - can be grouped in e.g. utils/api
- hooks - contains custom hooks

Context

But first... project structure

- modules
  - Domain-bound units that serve a specific purpose (e.g. LoginForm, UserProfile, Links, EventCell, ...).
  - Usually responsible for the business logic and presentation part of an application
  - Each module can have separate *components* folder which contains its building blocks
    - Folders in that folder can have module-like organization
    - These components don't have to be domain-agnostic
  - Other previously-mentioned folders (*utils*, *hooks*, *api*...) can also be found in a module, just like *components* folder
- context - various context files - or they can be inside their own modules

Context

Motivation

- We have a value at the top of the app, and need it:
  - A) Pretty much everywhere (e.g. app's theme - light or dark)
  - B) In some component deep in the file tree
- We want to remain flexible where to use some data or logic
- Anti-solution: "prop drilling"
  - Pass everything via props to every component
  - The whole component tree re-renders on change, app is data heavy
- Solution: Context

Context

- Shipped with React
- Wrap your components/App with a ContextProvider component with some value (can be an object)
- Consume your Context via ContextConsumer component or useContext hook
  - Provider has to be rendered above consumer
  - Consumer will receive data from the nearest Provider of the same type (if multiple rendered)
- When Provider value changes, all its Consumers will rerender, to get new value.

Context vs Redux

- Redux is a 3rd party library for state-management
  - Extremely popular with React a few years ago, now it's just popular
  - Propagation of state changes to components
- Context comes out of the box with React
- Context is simpler to use and makes cleaner code in small apps
- Redux is more robust for state management and persisted states
  - Context can but isn't exactly intended to fully replace a state management library
- We at Sofascore use:
  - Context for passing data and avoiding prop drilling
  - Redux for global state management (e.g. reason: redux-saga)

Context example - context.md

Sofascore

02

More hooks

More hooks

useCallback

- Caching a method between re-renders
- Official docs
- If we have a custom onClick method inside our component, each time during a re-render, the method gets a new memory location (reference)
  - useCallback caches that method given dependency array so reference doesn't change
  - React compares object equality with Object.is method, which compares memory references for objects, not property values
- Sometimes is an overkill, more on that later
- Sometimes makes sense in combination with React.memo, which we'll see in Performance part of today's lesson ➡️SOON

More hooks

useMemo

- Caching a **method result** between re-renders given dependency array
- Official docs
- Useful for expensive calculations, not just object memory reference
  - Therefore, more useful than useCallback - useCallback is actually useMemo which necessarily returns a function
- React doesn't guarantee that useMemo results will be always cached, sometimes they might be re-calculated

useMemo example - useMemo.md

Sofascore

More hooks

Custom hooks

- Custom hooks are code extraction - hooks calls extractions, specifically
  - They must still obey **rules of hooks**
  - Of course, not every line has to be a hook call
- Often bundled in unofficial, but widely used libraries
  - Sometimes with small bugs 🙃
  - E.g. https://usehooks-ts.com/
- Common examples:
  - usePersistedState - be careful, not every version from internet works correctly :(
    - useState with state stored in localStorage
  - useMediaQuery - hook which checks if given media query passes

Custom hook example - useResize.md

Sofascore

03

React Performance

Performance basics

- Browser DOM is slow and inconsistent
- Reuse DOM
- SyntheticEvent - All DOM events in React are wrapped in the SyntheticEvent wrapper
  - Browser universal (works for all browsers) event wrapper
  - Has all properties of native browser event (target, currentTarget, stopPropagation, preventDefault, ...)
- Before React 17: Events were pooled and reused, it caused problems for developers

Virtual DOM

- Virtual representation of a UI in the memory
- Abstracts DOM manipulations, event handling
- Synced with real DOM -> reconciliation process
    - That’s how React works

Performance - Developers

- Minimize expensive DOM operations -> follow reconciliation rules
- **Minimize number of unnecessary rerenders**
- When will rerender happen:
  - Prop change - (prop value or reference)
  - State change
  - Context change
  - this.forceUpdate() - class components
  - When parent rerenders

React.memo

- Skip re-rendering a component when its props are unchanged
- Official docs
- const MemoizedComponent = memo(SomeComponent, arePropsEqual?)
- Without arePropsEqual - compares shallow equality, i.e. objects by reference
  - That's why we need useCallback and useMemo
  - Custom deep comparison can terribly hinder performance
- Great when used in right situations
  - When to useMemo and useCallback by Kent C. Dodds
  - It can backfire when we memoize something that constantly changes
  - React is fast by default

Performance example - memoization.md

04

App themes

App themes

- Designers must provide common semantic names for theme variables
- Full support [hopefully] using your selected CSS in JS library
- We will demonstrate something very simple

Themes example - themes.md

Sofascore

01

Panda CSS

Brief history of styling in frontend

- Initially, styles were written in external stylesheets which were later merged in global CSS
- 2014 - rise of Javascript SPA frameworks, first CSS-in-JS solutions appeared (styled-components, Emotion)
- 2024 - introduction of streaming rendering, zero runtime CSS-in-JS solutions are now preferred (pandacss, ant-design, chakra-ui)
- In other words -> we are back at square one

Panda CSS

About Panda

- Official docs
- Zero runtime styling engine
- Combines developer experience of CSS-in-JS and performance of atomic CSS
- Leverages **static analysis** to scan JS/TS files for styles
- Very narrow learning curve

panda

Panda CSS

Main concepts

- Tokens
  - collection of platform-agnostic values for describing visual style
  - for colors, sizing, fonts, shadows, durations etc.
  - they help us make our design consistent
- Patterns
  - set of predefined styles to reduce code repetition, shipped along with library
  - they allow us to write layouts with ease
  - `<Box/>`, `<Flex/>`, `<Container/>`, `<Grid/>`, `<Divider/>` etc.
- Recipes
  - they allow us to write multi-variant styles
  - for dynamic or complex components

Panda CSS

Main concepts

- Responsive design
  - Most important feature
  - They allow us to write responsive styles with much less code
  - Object syntax: `<Box display={{ base: “none”, sm: “block” }} />`
  - Array syntax: `<Box display={[“none”, “block”]} />`

Panda CSS example - panda.md

Sofascore

Thank you for your attention!

Sofascore