Sofascore Frontend Academy Lecture 09
May, 2025

SWR. Portals. Refs. React patterns.

Darjan Baričević

Sofascore

TABLE OF CONTENTS

Sofascore

01

SWR

Motivation

- Fetch + useEffect + useState is okayish, but we usually require complex features
  - E.g. polling, local caching, fetching on tab focus, request deduplication
  - We could implement this, but let's not reinvent the wheel 😉
- Solution: SWR (stale-while-revalidate)
  - **Hooks based** HTTP client library

SWR

- yarn add swr
  - Types are added out of the box
- Wrap your app in SWRConfig which accepts value of SWRConfiguration type
  - Fetcher callback needs to be defined
    - **For all intents and purposes of this Academy just c/p my demo**
- Check out App.tsx in our demo project
- e.g. const {data: match, error} = useSWR<MatchDetailsResponse>(matchRoute(matchId), {refreshInterval: 10000})
  - Store result of MatchDetailsResponse type into match variable, error in error variable
  - Poll the server every 10 seconds

SWR example - swr.md

Sofascore

02

React Portal

React Portal

- Official docs
- Render children outside of DOM hierarchy
  - Used with modals, dialogs, notifications, toasts
- Commonly used with Singleton pattern - e.g. Toast notifications
- Also used for expanding menus of elements whose containers have hidden overflows
- ReactDOM specific
- createPortal(child, container)
- Event propagation and Context work as if it is still in the hierarchy

Portal example - portal.md

Sofascore

03

Refs

Refs

- Two distinct usages:
  - Storing data which should not trigger re-renders
  - Manipulating DOM - but, more on that later
- Official docs
- Returns object with current key set to DOM object or null
- useRef hook - const ref = useRef(null)
- createRef method - this.ref = createRef(null)

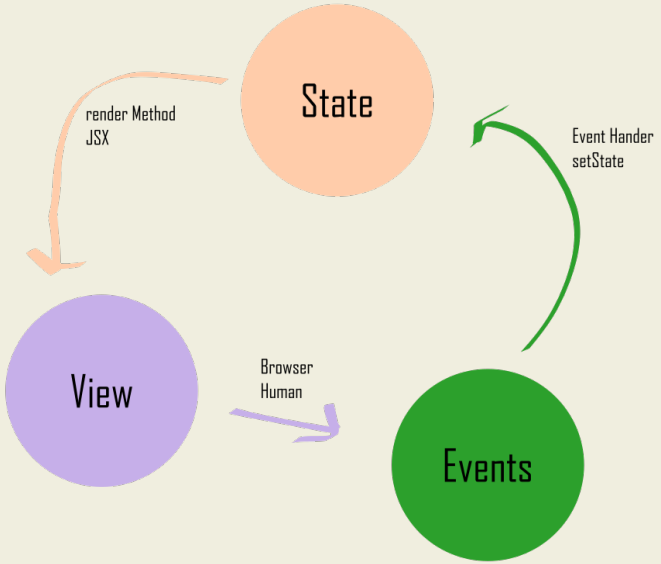
Refs example - refs.md

04

State Management

Introduction

- Simply refers to the way of managing data and application state
- React uses **one-way data flow**, passing data down the component hierarchy from parent to child component -> knowing this helps us identify where our state should live
- See Thinking in React section in the official docs



Types of state

- **Local state** -> refers to the state managed within individual components
- **Global state** -> shared state that needs to be accessed across multiple components or routes in the application
- **Server state** -> data fetched from the external sources like APIs and needs to be synchronised with front-end application
- **UI state** -> visual state of the application

Common approaches

- **Prop drilling** - works for small applications
- **Global state management solutions** - obviously needed for larger applications (Redux, zustand, Mobx etc.)
- **Context API -** allows managing global state without the need of third-party library

05

React Patterns

Overview

- **Design patterns**:
  - HOC pattern
  - Hooks pattern
  - Render Props pattern
  - Container pattern
  - Compound pattern
- **Rendering patterns**:
  - Client-side rendering
  - Server-side rendering
  - Incremental Static Generation etc.

HOC pattern

- stands for **H**igher **O**rder **C**omponent, which is a component that receives another component
- after applying certain logic to received component, it returns a new component with additional logic

```
function withStyles(Component) {
  return props => {
    const style = { padding: '0.2rem', margin: '1rem' }

    return <Component style={style} {...props} />
  }
}

const Button = () = <button>Click me!</button>
const Text = () => <p>Hello World!</p>

const StyledButton = withStyles(Button)
const StyledText = withStyles(Text)
```

HOC pattern example - hoc.md

Hooks patern

- appeared along with Hooks feature in React 16.8 which made possible to use state and lifecycle methods outside of classes
- also called “custom hooks pattern”
- https://usehooks.com/

Hooks pattern example - hooks.md

Render props patern

- allows parent to implement its own rendering logic by calling the child's **render** prop
- it doesn't actually have to be called render, we can name it any way we want
- the child component does not render anything besides the render prop
- very powerful pattern

```
const Title = (props) => props.render();
```

```
<Title render={() => <h1>I am a render prop!</h1>} />
```

Render props pattern example - render-props.md

Container Pattern

- just enforces separation of concerns by separating the view from application logic
- also known as **Presentational pattern**
- originates from old React times when only class components could contain state

Compound Pattern

- pattern of creating UI components which share implicit state by leveraging an explicit parent-child relationship
- state is usually shared with **Context API**
- allows creating really flexible and generic components

```
<Tabs>
  <Tabs.List>
    <Tabs.Tab id="info">Info<Tabs.Tab>
    <Tabs.Tab id="players">Players</Tabs.Tab>
  </Tabs.List>

  <Tabs.Panel tab="info">...<Tabs.Panel>
  <Tabs.Panel tab="players">...</Tabs.Panel>
</Tabs>
```

Thank you for your attention!

Sofascore