React 102

12 April 2022
Robert Schaap

React 102

- State Management – Local versus global
- Making API Calls in React
- Styling
- Testing

State Management

- State can be described as everything needed to keep an application running
- In React, state can be seen as a snapshot of what your application or specific components look like.
- Much like a light button may have an ON and an OFF state, a menu on a webpage may have an OPEN and a CLOSE state.
- In the context of React “something” manages that state and the developer writes code for that management to be done, and the application to respond to it via React’s APIs

State Management - Local

```
import ReactDOM from 'react-dom';
import React from 'react’;

const Page = () => (
  <main>
    <h1>My application</h1>
    <Component />
  </main>
);

ReactDOM.render(<Page />, document.getElementById("root"));
```

State Management - Local

```
import React, { useState } from 'react';

const Component = () => {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      {count}
    </button>
  );
};
```

State Management - Local

```
const Component = () => {
  const [count, setCount] = useState(0);

  return (
    <section>
      <AnotherComponent1 count={count} />
      <button onClick={() => setCount(count + 1)}>{count}</button>
    </section>
  );
};
```

State Management - Local

```
const Component = () => {
  const [count, setCount] = useState(0);

  return (
    <section>
      <AnotherComponent1 count={count} />
      <AnotherComponent2 increment={() => setCount(count + 1)} />
    </section>
  );
};
```

State Management - Context

- Context is React’s native way to deal with global state
- It works based on the provider-consumer principle
- Create a context provider, pass it an initial state and consume it with the useContext hook

State Management - Context

```
import { createContext } from "react";

const GlobalCount = createContext();

const Page = () => {
  const [count, setCount] = useState(0);
  const increment = () => setCount(count + 1);

  return (
    <GlobalCount.Provider value={[count, increment]}>
      <Component />
    </GlobalCount.Provider>
  );
};
```

State Management - Context

```
const Component = () => {
  const [count, increment] = useContext(GlobalCount);

  return (
    <button onClick={increment}>{count}</button>;
  );
};
```

State Management - Redux

- Redux or rather React-Redux is one of the earlier ways to deal with global state in React
- These days Context should be able to handle a lot of basic scenarios and there are other options such as Zustand, however Redux is still one of the more popular and robust libraries
- It used to be quite explicit and boilerplate heave, but newer iterations like Redux Toolkit reduce a lot of this and can be used to quickly scaffold global state

State Management - Redux

- It works based on the selector/dispatcher principle

- Best thought of as a separate datastore that sits outside of your application, with a few custom hooks allowing you to select from and dispatch to this store

State Management - Redux

```
const store = { count: 0 };

const reducer = (state, action) => {
  if (action.type === 'INCREMENT') return state + 1;
  return state;
}

const Component = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();
  const incremement = () => dispatch({ type: 'INCREMENT' });

  return <button onClick={increment}>Count: {count}</button>;
};
```

State Management - Redux

```
const slice = createSlice({
  initialState: { count: 0 },
  reducers: {
    increment: (state) => { state.count += 1 },
  },
});

const store = configureStore({ reducer: slice.reducer });

const increment = slice.actions.increment;
```

State Management - Redux

```
const Page = () => (
  <Provider store={store}>
    <Component />
  <Provider>
);

const Component = () => {
  const count = useSelector(state => state.count);
  const dispatch = useDispatch();

  return (
    <button onClick={() => dispatch(increment())}>{count}</button>
  );
};
```

API Calls

- One of the first things you’ll generally need when building an application is the ability to make API calls to fetch some sort of data
- React is focused on UI, as such it does not come with anything included.
- The native Fetch API available in browsers is a common choice
- If you’re looking for something prebuilt, Axios is a popular choice and if you desire data caching, React Query and React SWR are newer options

API Calls – Fetch

```
const Component = () => {
  useEffect(() => {
    fetch("users", {
      method: "POST",
      body: JSON.stringify({ name: "Robert" }),
    })
      .then((res) => res.json())
      .then(console.log);
  }, []);

  return <div />;
};
```

API Calls – Axios

```
const Component = () => {
  useEffect(() => {
    axios
      .post("users", { name: "Robert" })
      .then(res => console.log(res.data));
  }, []);

  return <div />;
};
```

Styling

- React has no opinions on styling so the choice is up to you. Tons of options available that are used in other frameworks as well.

For something opinionated:

- Bootstrap
- Material UI
- Ant Design

For something more freehand

- Tailwind
- Styled components
- (S)CSS (Modules)

React Testing Library

- React Testing Library is the React implementation of DOM testing library; a utility library for querying DOM nodes.
- Its goal is to allow us to query the DOM similarly to how a user would.
- Gained a lot of traction in the React community.
- Recommended together with Jest but other testing frameworks can be used.

github.com/testing-library

testing-library.com

Rendering

- Render a component by using the render function and passing the component as well as its props.
- Nothing more needed, its now it’s own DOM/ReactDOM

```
import { render } from "@testing-library/react";

render(<MyComponent />);
```

Basic Query Types

- “get” functions find either a single or an array of DOM nodes, or throw

```
getByX();
getAllByX();
```

- “find” functions return a Promise
- Wait/retry for 1000ms

```
findByX();
findAllByX();
```

- “query” functions find either a single or an array of DOM nodes
- Returns **null** or an empty array if none are found

```
queryByX();
queryAllByX();
```

getByText

- Finds a DOM node by its text content
- Takes a **string**, regular expression or **function**

```
const { getByText } = render(<MyComponent />);

getByText("Some text");
getByText(/some more text/i);
getByText((n) => n.contains("Some text"));

expect(getByText("Some text")).toBeInTheDocument();
```

getByTestId

- Finds a DOM node by a **data-testid** attribute set on the DOM node

```
<div data-testid="my-component">Some text</div>
```

```
const { getByTestId } = render(<MyComponent />);

expect(getByTestId("my-component")).toBeInTheDocument();
```

Dealing with events

```
import { fireEvent, render } from "@testing-library/react";

const { getByTestId } = render(<MyComponent />);

fireEvent.click(getByTestId("my-component-increment-button"));

expect(getByTestId("my-component-counter")).toHaveTextContent(2);
```

Dealing with events

```
const { getByTestId } = render(<MyComponent />);

fireEvent.change(
  getByTestId("my-component-input-field"),
  { target: { value: "new_value" } },
);

expect(getByDisplayValue("new_value")).toBeInTheDocument();
```

Dealing with time

```
import {
 render, waitForElementToBeRemoved
} from "@testing-library/react";

const { getByTestId } = render(<MyComponent />);

await waitForElementToBeRemoved(() => getByTestId("loader"));

expect(getByTestId("my-component")).toBeInTheDocument();
```

Dealing with time

```
import {
 render, waitFor
} from "@testing-library/react";

const { getByTestId } = render(<MyComponent />);

await waitFor(() => {
  expect(getByTestId("my-component")).toBeInTheDocument();
});
```

Putting it all together

```
it("should just work", async () => {
  render(<MyComponent />);

  await waitForElementToBeRemoved(
    () => screen.getByTestId("loader"),
  );

  expect(
    screen.queryByTestId("hello-world"),
  ).not.toBeInTheDocument();

  fireEvent.click(screen.getByTestId("button"));

  expect(screen.getByTestId("hello-world")).toBeInTheDocument();
});
```

Thank You!