
Embrace Opportunity

Software Engineering Day 5

React Lifecycle, Hooks, and Routes

Today's Overview

- React Lifecycle
- React Hooks
- Routing with React Router

- Cheat Sheet:
  - iX Cheat Sheet
  - iX Cheat Sheet - Day 5

ReactJS

Lifecycle - old (deprecated)

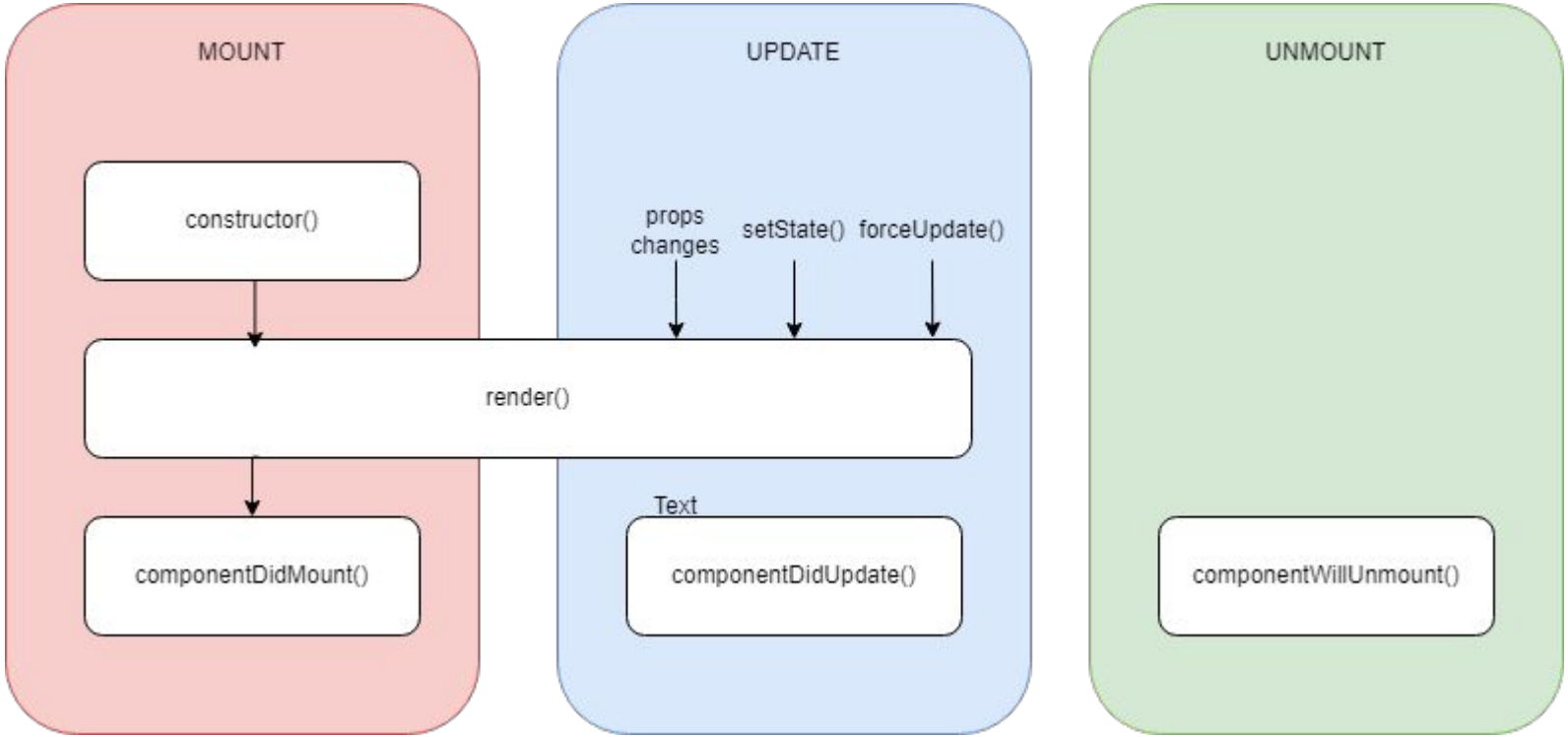
React Lifecycle - Introduction

- These functions allow you to interact with different stages of a components lifecycle.
- These methods can perform actions during different points in the lifecycle:
  - Initiate states
  - Make API requests
  - Updating UI
  - Cleaning up resources

Lifecycle - Phases

- A component’s lifecycle has three main phases:
  - Mounting phase
    - Begins when the component is first created.
  - Updating phase
    - Occurs when a component’s state or props change.
  - Unmounting phase
    - Occurs when a component is removed.

Lifecycle - Phases



Lifecycle - React - The Switch

- React lifecycle methods such as *componetDidMount()*, *componentDidUpdate()* and *componentWillUnmount()* were utilized in class based components.
- However React has moved away from class components with lifecycle and moved to the recommended function component with hooks.
    - This is where the next section comes into play.
    - Hooks are created in order to “hook-in” to function components and have replaced the older lifecycle methods of the class components.

ReactJS

Hooks

Hooks - Introduction

- Introduction:
  - **Hooks** let you use different React features from your functional components.
    - They are exclusively available in function components and other custom hooks.
    - Hooks allow you to use state and other React features without writing a class.
    - Hooks don’t work inside classes — they let you use React without classes. Brought on by React moving away from class components in favour of the recommended function component.
  - Hook Types:
    - State Hooks - Managing states.
    - Effect Hooks - Managing side effects.
    - Custom Hooks (hooks created from other React hooks)

Hooks - Rules

- There are two rules to follow when using hooks in React:
  - Only call hooks at the top level:
    - Hooks should not be called inside loops, conditions, or nested functions.
    - Hooks should always be called at the top level of a React function, before any early returns.
  - Only call hooks from React functions:
    - Hooks should not be called from JavaScript functions.
    - Hooks should only be called from:
      - React function components.
      - Custom hooks (hooks made from other React hooks).

Hooks - Best Practices

- Encapsulation:
  - Encapsulate related logic into custom hooks to make your component code cleaner and more reusable.
- Separation of Concerns:
  - Keep UI logic separate from state management when possible. This separation makes the code easier to maintain and test.
- Use Libraries When Needed:
  - Don’t shy away from using state management libraries if your application’s state management requirements become too complex for plain React hooks. Sometimes the additional abstraction and capabilities provided by these libraries can significantly simplify your application logic.
    - For example on Day 13 we will introduce Redux, a state management library that helps manage “global” states.

Hooks - State Hooks

- Introduction:
  - State Hooks:
    - Called inside a function component to add some local state to it.
    - React will preserve this state between re-renders.
    - e.g. *useState*
- *useState* Hook:
  - Used to create a state variable for a component and give a function to be able to update the state of the variable.
    - Never update state directly.
  - Setup:
    - State - Variable name.
    - Function - Called to update state.
    - Default value - Initial state assigned to the variable.

Hooks - useState - Example

- Syntax:

```
const ["Variable Name / State Name", "Function Name"] = useState("Default Value");
```

Hooks - Effect Hooks

- Introduction:
  - Side effect:
    - Occurs when a component needs to be updated after rendered. However this update is controlled by an external source, e.g. when a component updates based on a connection to a network / fetching from an api, etc…
    - These are operations outside of the scope of React.
    - Common side effects:
      - Request to an API for data.
      - Interacting with the browser APIs (*document*, *window*)
      - Using unpredictable timing functions (*setTimeout*, *setInterval*)

Hooks - Effect Hooks

- When should an effect hook be used?
  - Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's render phase), these operations should be handled by the effect hook.
- Effect Hooks:
  - Adds the ability to perform side effects from a function component.
  - It serves the same purpose as the lifecycle methods: *componentDidMount*, *componentDidUpdate*, and *componentWillUnmount*
  - e.g. *useEffect*

Hooks - useEffect

- *useEffect* lets a component connect to and synchronize with external systems:
  - Includes fetching data from an API, updating the DOM, or subscribing to an event, etc…
- Setup:
  - Setup code:
    - Function with the Effect's logic.
      - Establishing connections, fetching data, etc…
  - Dependencies (optional):
    - Include props and all variables and functions declared directly inside of your component.
  - Clean-up code:
    - Often, effects create resources that need to be cleaned up before the component leaves the screen, such as a subscription or timer.

Hooks - useEffect

- Syntax:

```
useEffect(() => {
    //Establishing a connection, fetching data, etc...
    console.log("Setup Code");
    return () => {
      // Disconnecting from the system
      console.log("Cleanup Code");
    };
  }, ["Dependencies"]);
```

Hooks - useEffect

- Differences in passing a dependency array, an empty array, and no dependencies:
  - Defined dependency array:
    - Effect runs after the initial render and after re-renders with changed dependencies.
      - Subscribes to changes in state.
  - Empty array:
    - Effect only runs after the initial render.
      - Executes the callback function every time any state is changed.
  - No dependencies:
    - Effect runs after every single render (and re-render) of your component.

Hooks - useEffect - Example - Defined

- Now lets look at an example implemented with our Blog App.
- We'll utilize a defined dependency array for this example.
- Inside our *BlogsPage* component from *frontend/src/pages/Blogs/index.jsx*

```
useEffect(() => {
    // Filtering blogPosts based on the categoryId.
    const blogs = blogPosts.filter((x) =>
      categoryId !== undefined
        ? x.categories.find((y) => y.id.toString() === categoryId.toString())
        : true
    );
    setBlogs(() => blogs);
}, [categoryId]);
```

- Please refer to the vue resource for full code snippet:

Hooks - useEffect - Example - Defined

- Example explanation:
  - The *useEffect* is used to update the *blogs* state whenever *categoryId* changes. This ensures that whenever a user selects a different category, the list of blog posts updates accordingly.
  - Inside the effect, the *blogPosts* are filtered based on the *categoryId*. If a *categoryId* is defined, it filters *blogPosts* to include only those that belong to the selected category. If *categoryId* is undefined, it shows all blog posts.
  - *setBlogs* is called within the effect to update the component's state with the filtered list of blog posts.

Hooks - useContext

- *useContext* is helpful for passing props down multiple levels of child components.
  - Alternative to prop drilling, without having to pass a prop through the whole tree.
- *useContext* - Allows you to access a context object in a functional component.

Hooks - useRef

- *useRef* is mainly used to access a child component imperatively.
  - Reference a value that is not required for rendering.
- Refs are a special attribute that are available on all React components.
- *useRef* - Create a mutable ref object that persists for the lifetime of the component.
- They are useful when you need to work with non-React systems, such as the built in browser APIs.

```
import { useRef } from "react";
    const inputRef = useRef(null);
```

Hooks - Performance Hooks

- These hooks are used to optimize re-rendering performance by skipping unnecessary work.
- *useMemo* - Returns a memorized value that helps in performance optimizations.
- *useCallback* - Returns a memorized version of a callback to help a child component not re-render unnecessarily

```
import { useMemo } from "react";
    const cachedValue = useMemo(calculatedValue, dependencies)
```

```
import { useCallBack } from "react";
    const cachedFn = userCallback(fn, dependencies);
```

ReactJS

Routing with React Router

Routing with React Router

- React is a single page application (SPA), which means a single HTML is loaded with the content being updated dynamically while interacting with the page.
- This allows content to be rendered faster without the full page being refreshed.
- React Router allows the requests to be intercepted and inject content dynamically from components created.
- React Router helps you to route/navigate to and render a new component.

- Official Documentation

Routing with React Router

- Key functionality of routing and React Router:
  - A route makes the connection between a URL(address) and a component.
    - When the user visits, or interacts with a certain address React Router will render its corresponding component.
  - React Router includes the infrastructure methods:
    - *BrowserRouter* - Web applications.
    - *HashRouter* - Static sites.
  - React Router includes the component:
    - Link - A navigation component that maintains the app's state.
      - Works similarly to the anchor <a> tag.
      - Will be used to replace anchor tags within an application.

Routing with React Router

- React Router even includes hooks for routers.
  - Some hooks will only work with certain routers (Aka hooks with routers that us Data APIs):
    - *useNavigate*
    - *useParams*
- Nested Routes (Added with React Router v6):
  - Adds the ability to have multiple components render on the same page with route parity.
    - A route is wrapped with another router, and so on.
  - Routes can have children which define segments in the URL.

React Router - Installation

- Firstly we will have to install react router:

```sh
npm install react-router-dom
```

- Utilizing React Router:
  - Imported inside our *.jsx* file.

```
import { createBrowserRouter } from "react-router-dom";
import { Link, useNavigate, useParams } from "react-router-dom";
```

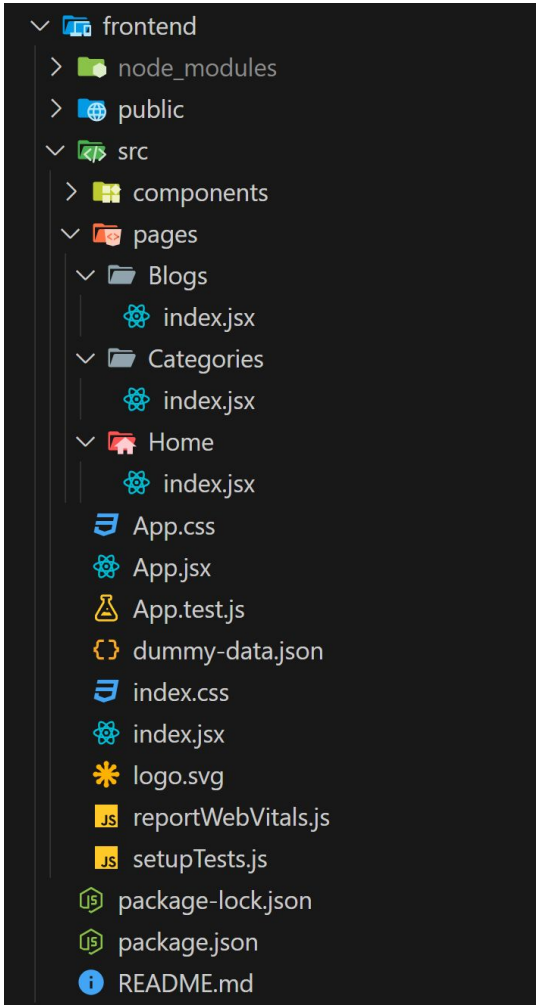
React Router - Routers

- Router is a stateful, top-level component that makes all the other components and hooks work.
  - Router Documentation
- React Router introduces several routers for different app environments:
  - Data supported routers:
    - *createBrowserRouter*
    - *createMemoryRouter*
    - *createHashRouter*
    - *createStaticRouter*
- *createBrowserRouter* - is the recommended router for all web applications.
  - Uses the DOM History API:
    - Allows manipulation of the browser session history, that is the pages visited in the tab or frame that the current page is loaded in.

React Router - Routing

- From Day 3 we should have the component file structure and files.
- We will setup the pages directory and files.
    - Creating these pages in frontend/src/pages/… :
        - Home
        - Categories
        - Blogs

**Note - Move the Home dir from components to pages.*



React Router - createBrowserRouter

- Open *frontend/src/App.jsx*
  - We originally had the *HomePage* component render in the main *App* component, however now that we have created the *HomePage* component in pages.
  - We will update *App.jsx* with React router using the router:
    - *createBrowserRouter*

```jsx
import React from "react";

import { createBrowserRouter, RouterProvider } from "react-router-dom";

import HomePage from "./pages/Home";
import BlogsPage from "./pages/Blogs";
import CategoriesPage from "./pages/Categories";

const router = createBrowserRouter([
  {
    path: "/",
    element: <HomePage />,
  },
  {
    path: "/home",
    element: <HomePage />,
  },
  {
    path: "/categories",
    element: <CategoriesPage />,
  },
  {
    path: "/blogs/:categoryId?",
    element: <BlogsPage />,
  },
]);
function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

React Router - Component - Link

- A helpful component of React Router is the *Link* component to navigate to other pages.
- *Link* is similar to the anchor element *<a>* in HTML
  - Let us go and replace the anchor tags previously used with *Link*.
- We will first open *frontend/src/pages/Home/index.jsx*
  - Importing the NavBar component
    - Then in the NavBar component we will add the use of *Link*.

```
import React from "react";

import Navbar from "../../components/Navbar";

export default function HomePage() {

  return (
    <>
      <Navbar />
    </>
  );
}
```

React Router - Component - Link

- We will then open *frontend/src/component/NavBar/index.jsx*
- Inside we will be using the Bootstrap *NavBar*, and using React Router *Link* in order to navigate back to home assigned to our title.
- As you will see the *Link* is using: to=”/home”, which you would notice is the path used in the *createBrowserRouter*

```
import React from "react";

import { Link } from "react-router-dom";

export default function Navbar() {
  return (
    <nav className="navbar navbar-expand-lg">
      <div style={{ margin: "0px 5%" }} className="container-fluid">
        <Link className="navbar-brand" to="/home">
          iX Software Engineering Blog
        </Link>
      </div>
    </nav>
  );
}
```

React Router - Hook

- A hook we will utilize in the project is - *useNavigate* Hook:
  - This hook will only work when using a data router:
    - Such as using the *createBrowserRouter* data router previously set up.
  - Used to trigger a navigation event from within a component.
  - *navigate* function has two signatures:
    - A “To” value same as Link, with an optional “options” argument
      - e.g. *navigate(“route path”)*
    - Passing a delta to navigate in the history stack
      - e.g. *navigate(-1)*

React Router - Hook - useNavigate - Example

- Using *CategoriesPage* component and *CategoryList* subcomponent created on Day 3.
    - We will update the *CategoryList* subcomponent with *useNavigate* to be able to navigate to the respective blogs.
- Inside the *CategoriesPage* component *frontend/src/component/Category/index.jsx we utilize the subcomponent CategoryList* passing in the prop *categories*

```
import React from "react";

import CategoryList from "../../components/CategoryList";

const data = require("../../dummy-data.json");
const categories = data.categories;

export default function CategoriesPage() {
  return (
    <>
      <CategoryList
          categories={categories}
      ></CategoryList>
    </>
  );
}
```

React Router - Hook - useNavigate - Example

- Now we can update the *CategoryList* subcomponent to navigate based on an event with the *category.id*
- Firstly import *useNavigate*.
- Setting the navigate function with the path: *“/blogs/” + category.id*.
    - Notice the path that we are navigating too, it will be utilized in the next hook.
    - This is also the path we set when utilizing *createBrowserRouter*.

```
import React from "react";
import PropTypes from "prop-types";
import { useNavigate } from "react-router-dom";

export default function CategoryList({ categories }) {
  const navigate = useNavigate();

  return (
    <div className="category-list">
      {categories.map((category, index) => {
        return (
          <div
            key={index}
            className="card"
            style={{ borderRadius: "0px", border: "none" }}
            onClick={() => {
              navigate("/blogs/" + category.id);
            }}
          >
          </div>
        );
      })}
    </div>
  );
}

CategoryList.propTypes = {
  categories: PropTypes.array.isRequired,
};
```

React Router - Hook - useParams

- *useParams* Hook:
  - Returns the key/value pairs of the dynamic params from the current URL that were matched by the <Router path>.
  - Access the parameters of the current route to manage the dynamic routes in the URL.
- We will use this hook to filter our blogs on the *BlogsPage*.

React Router - Hook - useParams - Example

- Opening *BlogsPage* from *frontend/src/pages/Blogs/index.jsx*
- Firstly importing *useParams*.
- We utilize the hook to set *categoriesId* by the *Id* passed through the URL.
- Thanks to the hook in conjunction to *createBrowserRouter*

```
{
  path: "/blogs/:categoryId?",
  element: <BlogsPage />,
},
```

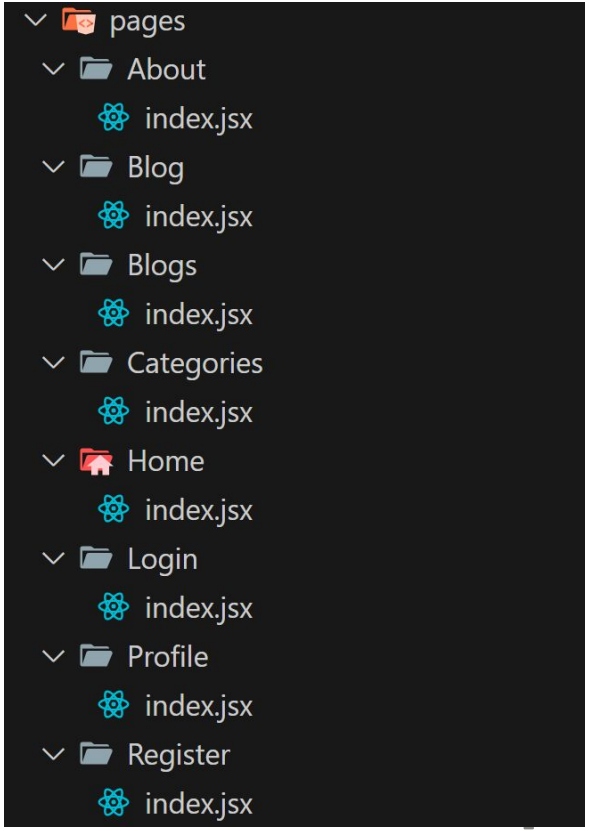
```
import React from "react";

import { useParams } from "react-router-dom";

const data = require("../../dummy-data.json");
let blogPosts = data.blogPosts;
const categories = data.categories;

export default function BlogsPage() {

  const { categoryId } = useParams();

  //Filtering blogs by the ctergoryId passed in the URL
   blogPosts = blogPosts.filter((x) =>
    categoryId !== undefined
      ? x.categories.find((y) => y.id.toString() === categoryId)
      : true
  );

}
```

Exercise

Creating the Capstone Routes

Exercise:

- Create the file structure for all the necessary pages of the Capstone project (which will be utilized later).
  - File directory:
    - *frontend/src/pages/…*
  - Pages necessary:
    - HomePage, BlogsPage, BlogPage, LoginPage, RegisterPage, ProfilePage, CategoriesPage, AboutPage, LoginPage
- Update the *createBrowserRouter* from *frontend/src/App.jsx* to include the new pages.



Homework

Apply what we have learned

Homework

- Start building the front end application for the Capstone project
- Create Capstone repository
- Push changes to GitHub & submit link to homework repository

Next Class

Asynchronous JS and APIs