
Embrace Opportunity

Software Engineering Day 4

Understanding State Management and Props in React

Today’s Overview

- React State Management
- React Props

- Cheat Sheet:
  - iX Cheat Sheet
  - iX Cheat Sheet - Day 4

React

State Management

State Management - Introduction

- What is State Management in React
  - This refers to the methods and techniques to handle, organise, and share data within an application.
- State Management involves:
  - Systematic management.
  - Manipulation of data.
  - Ensuring seamless integration and synchronization across components.
- State Types:
  - Local
  - Global (“shared” state)

State Management - Introduction

- Types of states in a project:
  - Local:
    - State that is defined, accessed and updated with the component without affecting other components.
      - If defined in a parent component, it can be accessed and updated in a child component using “prop drilling”.
        - However, this can cause issues when handling many components in a tree, which would be better utilized as global states.
    - Example:
      - Using local state to track the values of a form component for form submission.

State Management - Introduction

- Global (“shared” state):
  - State that is accessible across multiple components.
    - This is necessary when retrieving and updating data across an application
    - Improves the communication between components across the application.
    - It alleviates the issues of prop drilling.
  - Example:
    - Using global state to store our authentication, blogs, and categories across our application.
      - However, these would be replaced by Redux “store” and “slice” that we will learn on Day 13.

State Management - Introduction

- Choosing between the two:
  - State management complexity between components.
    - How far down a tree of components state is needed to be accessed and updated.
  - How many components rely on a single state to be updated.
- Benefits:
  - Developing dynamic and interactive applications.
  - Handles evolving data:
    - User interaction.
    - Triggered events.
  - Maintains data integrity.
  - Enhance performance.

State Management - Best Practices

- Best Practices for State Management:
  - Keep states minimal:
    - Local where possible.
    - Only storing necessary data for a component to render.
  - Only update state indirectly:
    - The state update function declared from *useState*.
  - Use a state management library:
    - When an application's states becomes complex, its best to use a library such as Redux.
      - We will learn about Redux in Day 13 of the course.

State Management - Brief Recap

- State:
  - Encapsulated data that is persistent between component renderings.
  - Is mutable - can be changed over time.
  - It represents the current state of a component.
- React moving to function components introduced hooks, in order to “hook into” components allowing you to update components after initial render.
  - Hooks will be taught more in depth tomorrow.

State Management - Local State

- We will be focusing on local state for today, as global state and further state management using Redux will be taught on Day 12.
- React has a built-in state management hook which we utilize:
  - *useState*:
    - Syntax:

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

State Implementation - Initialize State

- To initialize a state:

```
import { useState } from 'react';
const [categoryId, setCategoryId] = useState();
```

State Implementation - Update State - Example

- Update state:
  - Never update a state directly, only using the declared function.
- We will look at an example of implementing state inside our *BlogsPage* component created on Day 3.
  - Please refer to the vue resource for full code snippet.
- From Day 3 homework we have created our blogs page component and subcomponents. We have already imported dummy data for *blogPosts* and *categories* which we will utilize for our components in this example.

State Implementation - Update State - Example

- Opening *BlogsPage* component (*frontend/src/components/Blogs/index.jsx)*
  - We'll declare our states inside our component:

```
export default function BlogsPage() {
  //Initializing our states:
  const [categoryId, setCategoryId] = useState();
  const [blogs, setBlogs] = useState([]);
}
```

State Implementation - Update State - Example

- Now that we have opened our *BlogsPage* component and declared our states, we will creating a component that incorporates the use of buttons in order to set a *categoryId* with an *onClick* event. This can be utilized later in order to filter the blog posts on *BlogsPage* by *categoryId*.
- Let us first create our *CategoriesList* component as follows, to be embed into *BlogsPage* component after.

```
const CategoriesList = () => {
  return categories.map((category, index) => {
    return categoryId === category.id.toString() ? (
      <button
        key={index}
        onClick={() => setCategoryId(category.id)}
        style={{ color: "blue" }}
      >
        <p key={index}>{category.title}</p>
      </button>
    ) : (
      <button
        key={index}
        onClick={() => setCategoryId(category.id)}
        style={{ color: "black" }}
      >
        <p key={index}>{category.title}</p>
      </button>
    );
  });
}
```

State Implementation - Update State - Example

- Now that our *CategoriesList* component has been created, let us embed it into our *BlogsPage* component.
- From this example, it can be seen that when a user interacts with the button in the *CategoriesList* component, the *categoryId* (‘state name’) will be updated from the function *setCategoryId* (‘declared update state function’) with the value *category.id* (dummy data).

```
return (
  <>
    <Navbar />
    <div className="container">
      <Heading />
      <div className="scroll-menu">
        <CategoriesList />
      </div>
      <div style={{ display: "flex", justifyContent: "space-between" }}>
        <p className="page-subtitle">Blog Posts</p>
      </div>
      <BlogList blogPosts={blogPosts} />
    </div>
    <Footer />
  </>
);
```

React

Understanding Props

Understanding Props - Introduction

- React components use props to communicate with one another.
- Parent components can pass information to a child component through using props.
- Props (properties)
    - Unlike HTML attributes, you can pass:
        - Any JavaScript value:
            - Objects
            - Arrays
            - Functions
            - ...etc
- Props are the only argument to components.

Props - Passing Between Components

- React uses a one-way data flow:
  - Data is only transferred from parent component to child component
- Syntax:
  - Multiple props or singular prop can be passed into a component.
  - Props can be even “deconstructed” and passed as individual variables.

```
export default function ParentComponent(prop) {
  return (
    <p>{prop}</p>
  )
}
```

```
export default function ParentComponent({propA, propB, propC}) {
  return (
    <p>{propA}</p>
    <p>{propB}</p>
    <p>{propC}</p>
  )
}
```

Props - PropTypes

- To assist with validation props in an application, which helps greatly with components receiving correct information:
  - We have PropTypes:
    - This mechanism ensures that the passed value is of the correct data type
- Installation:

```sh
npm install prop-types --save
```

- Imports:

```
import PropTypes from 'prop-types';
```

Props - PropTypes

- Implementation:

```
BlogItem.propTypes = {
  index: PropTypes.number.isRequired,
  blogPost: PropTypes.object.isRequired,
  imageOrientation: PropTypes.string,
};
```

- Explanation:
  - *BlogItem* component should have three props passed to it: *index*, *blogPost*, and *imageOrientation* with their corresponding data types.

Props - PropTypes - Types and Syntax

- Library Documentation

Type	Class	Example
String	PropType.string	“helllo”
Object	PropType.object	{name: “Rohit”}
Number	PropType.number	10
Boolean	PropType.bool	true/false
Function	PropType.func	const say = {console.log(“hello”)}
Symbol	PropType.symbol	Symbol(“m”)

Props - Example

- Let us look at example that incorporates the aspects we've learnt about Props, which we will pass props from a parent component to a child component and utilizing propTypes.
- For full code snippets of the example please refer to the vue resource.
- In this case we will handle an example in which the BlogsPage incorporates its subcomponents and how each component passes props between it.
- We will update our apps components with the structure:
    - BlogsPage
        - BlogList
            - BlogItem
                - BlogItemText (We will handle this component for an exercise)

Props - Example - BlogsPage

- First let us call our *BlogList* component from our *BlogsPage*, opening from *frontend/src/pages/Blogs/index.jsx*:

```
return (
  <>
    <Navbar />
    <div className="container">
      <Heading />
      <div className="scroll-menu">
        <CategoriesList />
      </div>
      <div style={{ display: "flex", justifyContent: "space-between" }}>
        <p className="page-subtitle">Blog Posts</p>
      </div>
      <BlogList blogPosts={blogPosts} />
    </div>
    <Footer />
);
```

Props - Example - BlogList

- Next step is to update our *BlogList* (*frontend/src/components/BlogList/index.jsx*) component with the prop passed to it, and utilizing the prop inside the component.
- We can see that *BlogList* component has the prop *blogPosts*. Which you will notice the prop passed by *BlogsPage*.
- We also utilize propTypes at the bottom, validating the prop *blogPosts*, that it has the data type of array.
- Then we are importing our *BlogItem* component and again passing down three props: *index*, *blogPost* and *imageOrientation*.

```
import React from "react";
import PropTypes from "prop-types";
import "./index.css";
import BlogItem from "../BlogItem";

export default function BlogList({ blogPosts }) {
  return (
    <div className="blog-list">
      {blogPosts.map((blogPost, index) => {
        return (
          <div
            key={index}
            style={{
              width: "100%",
            }}
          >
            <BlogItem
              index={index}
              blogPost={blogPost}
              imageOrientation={"top"}
            />
          </div>
        );
      })}
    </div>
  );
}

BlogList.propTypes = {
  blogPosts: PropTypes.array.isRequired,
};
```

Props - Example - BlogItem

- Next we will be utilizing the props: *index*, *blogPost*, *imageOrientation*, with our *BlogItem* (*frontend/src/components/BlogItem/index.jsx*) component passed down from *BlogList*.
- In our vue resource we also have propTypes for our *BlogItem* with their relevant props and data types.
- And lastly in our exercise we will update the subcomponent at the tree: *BlogItemText*, passing in two props: *blogPost* and *headerFontSize*.
- We can see from this example how important props are with passing information between components and making the app dynamic.

```
export default function BlogItem({
  index,
  blogPost,
  imageOrientation,
}) {
if (imageOrientation === "top") {
    return (
      <div key={index} className="card-1">
        <img src={blogPost.image} className="card-img-top" alt="..." />
        <div className="card-text-bottom">
          <BlogItemText
            blogPost={blogPost}
            headerFontSize="20px"
          ></BlogItemText>
        </div>
      </div>
    );
  } else {
    return (
      <div key={index} className="card-2">
        <img src={blogPost.image} className="card-img-left" alt="..." />
        <div className="card-text-right">
          <BlogItemText
            blogPost={blogPost}
            headerFontSize="20px"
          ></BlogItemText>
        </div>
      </div>
    );
  }
}
```

Props - Example - Function

- Functions can also be passed as props in components.
  - Such as callback functions, or even state setter functions.
  - propTypes even validates functions with data type *func*.
- For full code snippets of the example please refer to the vue resource.
- Now that we have learned about props, and the fact that a function can be passed as a prop as well. From our earlier state example where we created our *CategoriesList* component, we will now update with passing props.
  - For this example, we will update the component with a callback function passed as a prop.
  - This will now build onto our state example, and filter the blogs by the *categoryId* set with our *CategoriesList* component.

Props - Example - Function

- From *BlogsPage* component (*frontend/src/components/Blogs/index.jsx*) we'll update our embed *CategoriesList* component to pass props: *categories*, *categoryId* and *setCategoryId*.

```
return (
  <>
    <Navbar />
    <div className="container">
      <Heading />
      <div className="scroll-menu">
        <CategoriesList
            categories={categories}
            categoryId={categoryId}
            setCategoryId={setCategoryId}>
        </CategoriesList>
      </div>
      <div style={{ display: "flex", justifyContent: "space-between" }}>
        <p className="page-subtitle">Blog Posts</p>
      </div>
      <BlogList blogPosts={blogs} />
    </div>
    <Footer />
  </>
);
```

Props - Example - Function

- Let us now move our *CategoriesList* component out of *BlogsPage* component to *frontend/src/components/CategoriesList/index.js*
- Now we must update the component to accept the three props: *categories*, *categoryId*, and *setCategoryId* we have passed from *BlogsPage*.
- Setting up the propTypes as well for the required data types.
- And building out the component to utilize the props including the function.

```
import React from "react";
import PropTypes from "prop-types";

export default function CategoriesList({
  categories,
  categoryId,
  setCategoryId,
}) {
  return categories.map((category, index) => {
    return categoryId === category.id.toString() ? (
      <button
        key={index}
        onClick={() => setCategoryId(category.id)}
        style={{ color: "blue" }}
      >
        <p key={index}>{category.title}</p>
      </button>
    ) : (
      <button
        key={index}
        onClick={() => setCategoryId(category.id)}
        style={{ color: "black" }}
      >
        <p key={index}>{category.title}</p>
      </button>
    );
  });
}

CategoriesList.propTypes = {
  categories: PropTypes.array.isRequired,
  categoryId: PropTypes.string.isRequired,
  setCategoryId: PropTypes.func.isRequired,
};
```

Exercise

Passing Props With BlogItem

Exercise: Passing Props With BlogItem

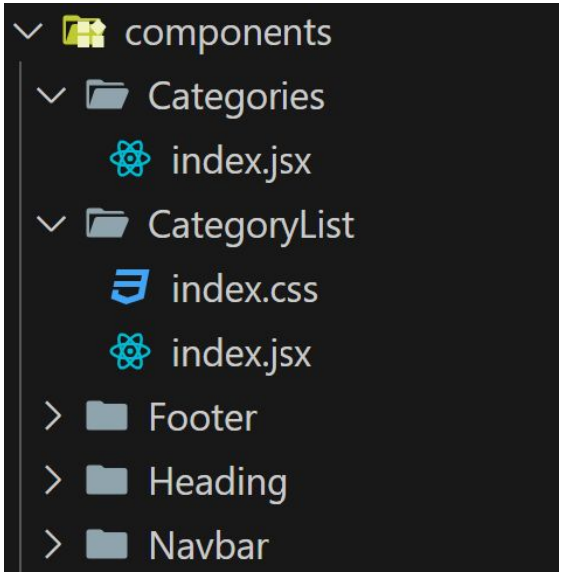
- As was shown earlier we had the tree of components in which we passed props between *BlogsPage*, *BlogList* and *BlogItems*
- Now update the *BlogItemsText (frontend/src/components/BlogItemText/index.jsx)* component that we utilized in the *BlogItem* component.
- The criteria for the exercise:
  - Create the propType for *BlogItemText*.
    - The props are:
      - *blogPost*
  - Update the component to utilize the information passed in the props.

Homework

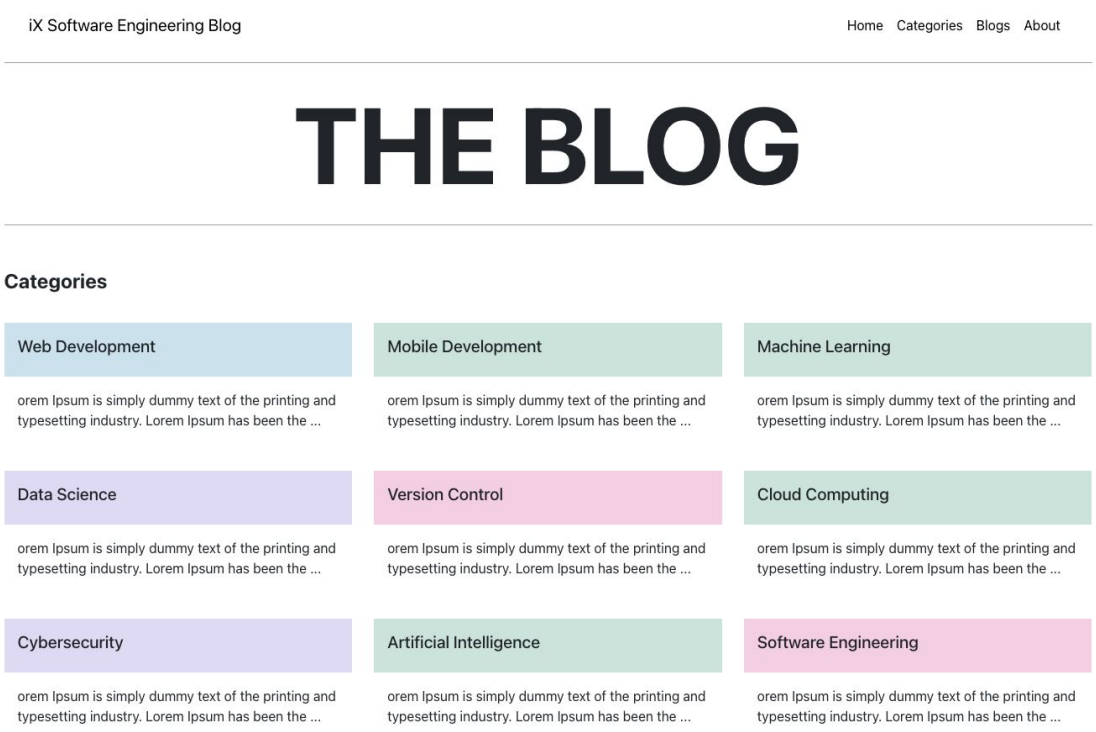
Apply what we have learned

Homework

- Create and Updating our CategoriesPage component with the structure:
  - Categories
    - *frontend/src/components/Categories/index.jsx*
      - CategoryList
        - *frontend/src/components/CategoryList/index.jsx*



Homework - Expected Result



Homework - Requirements

- Requirement:
  - CategoriesPage Component:
    - Components Embedded:
      - NavBar
      - Heading
      - CategoryList
        - Passing *categories* as the prop
      - Footer
  - CategoryList Subcomponent:
    - Should return a card for each category:
      - Title : category.title
      - Description: category.description
      - Background Color: category.color

Next Class

React Lifecycle, Hooks, and Routes