REACT HOOKS

1. useState Hook

useState to Create State Variables

- The useState hook allows us to create state variables in a React function component.
- State allows us to access and update certain values in our components over time
- When we create a state variable, we must provide it a default value (which can be any data type).
- We get that state variable as the first value in an array, which we can destructure and declare with const.

```
import "./styles.css";
import React from "react";

function App() {
  const [language] = React.useState("React!!!");
  return <h1>I am learning {language}</h1>;
}

export default App;
```

Update State Variables

- useState also gives us a setter function to update the state after it is created.
- To update our state variable, we pass the setter function the new value we want our state to be.
- When you declare your setter function, in most cases you will prefix it with the word "set"

```
import "./styles.css";
import React from "react";

export default function App() {
  const [language, setLanguage] = React.useState("React!!!");

  function changeLanguage() {
    setLanguage("React Hooks");
  }

  return <h1 onClick={changeLanguage}>I am learning {language}</h1>;
}
```

Can Be Used Once Or Many Times

- useState can be used once or multiple times within a single component.
- Sometimes you will want to create multiple state variables and other times you may want to use a single variable with an object (see below).

```
import "./styles.css";
import React from "react";

export default function App() {
  const [language, setLanguage] = React.useState("React!!!");
  const [years] = React.useState(0);

  function changeLanguage() {
    setLanguage("React Hooks");
  }

  return (
    <div>
      <h1 onClick={changeLanguage}>
        I've learned {language} for {years} years
      </h1>
      <button>Add Year</button>
    </div>
  );
}
```

Update State based on Previous Value

- If the new state depends on the previous state, we can take the previous state variable and apply whatever changes we want to make.
- For example, as in the example below, add 1 to the current years value to increment it.
- To guarantee the update is done reliably, we can use a function within the setter function that gives us the correct previous state.

```
import "./styles.css";
import React from "react";

export default function App() {
  const [language, setLanguage] = React.useState("React!!!");
  const [years, setYears] = React.useState(0);

  function changeLanguage() {
    setLanguage("React Hooks");
  }

  function addYear() {
    setYears((prev) => prev + 1);
  }

  return (
    <div>
      <h1 onClick={changeLanguage}>
        I've learned {language} for {years} years
      </h1>
      <button onClick={addYear}>Add Year</button>
    </div>
  );
}
```

Manage State with an Object

- You can use an object with useState, which allows you to manage individual values as key-value pairs.
- As the example below shows, when you are updating state with an object, you need to spread in the previous state.
- Why? Because any properties other than the one you are updating will not be included in the new state.

```
import "./styles.css";
import React from "react";
export default function App() {

  // const [language, setLanguage] = React.useState("React!!!");
  // const [years, setYears] = React.useState(0);

  const [state, setState] = React.useState({
    language: "React",
    years: 0,
  });

  function changeLanguage() {
    setState({ ...state, language: "React Hooks" });
  }

  function addYear() {
    setState((prev) => {
      return {
        ...prev,
        years: prev.years + 1,
      };
    });
  }

  return (
    <div>
      <h1 onClick={changeLanguage}>
        I've learned {state.language} for {state.years} years
      </h1>
      <button onClick={addYear}>Add Year</button>
    </div>
  );
}
```

2. useEffect Hook

useEffect to Perform Side Effects

- useEffect lets us perform side effects in function components.
- Side effects are when we need to reach into the outside world. Such as fetching data from an API or working with the DOM.
- Side effects are actions that can change our component state in an unpredictable fashion (that have caused 'side effects').
- useEffect accepts a callback function (called the 'effect' function), which will by default run every time the component re-renders.
- In the example below, we are interacting with the DOM to change style properties of the document body:

```
import "./styles.css";
import React from "react";

export default function App() {
  React.useEffect(() => {
    document.body.style.background = "navy";
    document.body.style.color = "white";
  });

  return (
    <div>
      <h1>React App</h1>
    </div>
  );
}
```

Run Again when a Value Changes

- useEffect lets us conditionally perform effects with the dependencies array.
- The dependencies array is the second argument passed to useEffect.
- If any one of the values in the array changes, the effect function runs again.
- If no values are included in the dependencies array, useEffect will only run on component mount and unmount.

```
import "./styles.css";
import React from "react";

export default function App() {
  const [color, setColor] = React.useState("navy");

  React.useEffect(() => {
    document.body.style.background = color;
    document.body.style.color = "white";
  }, [color]);

  function changeColor() {
    setColor("gold");
  }

  return (
    <div>
      <h1>React App</h1>
      <button onClick={changeColor}>Change color</button>
    </div>
  );
}
```

Unsubscribe by Returning a Function

- useEffect lets us unsubscribe from listeners that we might have created by returning a function at the end.
- We want to unsubscribe from certain events, such as an event listener, because when the component unmounts (i.e. the user goes to a different page), React may attempt to update state that no longer exists, causing an error.

```
import "./styles.css";
import React from "react";

export default function App() {
  const [color, setColor] = React.useState("navy");

  React.useEffect(() => {
    document.body.style.background = color;
    document.body.style.color = "white";
    window.addEventListener("keydown", handleEnterButton);
    return () => {
      window.removeEventListener("keydown", handleEnterButton);
    };
  }, [color]);

  function changeColor() {
    setColor("gold");
  }

  function handleEnterButton(event) {
    if (event.keyCode === 13) {
      setColor("red");
    }
  }

  return (
    <div>
      <h1>React App</h1>
      <button onClick={changeColor}>Get color</button>
    </div>
  );
}
```

Fetch Data from an API

- useEffect is the hook to use when you want to make an HTTP request (namely, a GET request when the component mounts).
- Note that handling promises with the more concise async/await syntax requires creating a separate function.
- This is because the effect callback function cannot be async.
- In the example below, we resolve our promise (returned from fetch) with a series of .then() callbacks to get our data.

```
import "./styles.css";
import React from "react";
export default function App() {
  const [color, setColor] = React.useState("navy");
  const [user, setUser] = React.useState(null);

  React.useEffect(() => {
    fetch("https://randomuser.me/api/")
      .then((res) => res.json())
      .then((data) => setUser(data.results[0]));
  }, []);

  React.useEffect(() => {
    document.body.style.background = color;
    document.body.style.color = "white";
    window.addEventListener("keydown", handleEnterButton);
    return () => {
      window.removeEventListener("keydown", handleEnterButton);
    };
  }, [color]);

  function changeColor() {
    setColor("gold");
  }
  function handleEnterButton(event) {
    if (event.keyCode === 13) {
      setColor("red");
    }
  }
  return (
    <div>
      <h1>React App</h1>
      <button onClick={changeColor}>Get color</button><br /><br />
      Current user: <pre>{JSON.stringify(user, null, 2)}</pre>
    </div>
  );
}
```

3. useRef Hook

useRef to Reference React Elements

- Refs are a special attribute that are available on all React components. They allow us to create a reference to a given element / component when the component mounts.
- useRef allows us to easily use React refs. They are helpful (as in the example below) when we want to directly interact with an element, such as to clear its value or focus it, as with an input.
- We call useRef (at the top of a component) and attach the returned value to the element's ref attribute to refer to it.

```
import "./styles.css";
import React from "react";

export default function App() {

  const inputRef = React.useRef(null);

  function handleClearInput() {
    inputRef.current.value = "";
    inputRef.current.focus();
  }

  return (
    <form>
      <input type="text" ref={inputRef} />
      <button type="button" onClick={handleClearInput}>
        Clear Input
      </button>
    </form>
  );
}
```

4. useCallback Hook

useCallback Prevents Callbacks from Being Recreated

- useCallback is a hook that is used for improving our component performance.
- Callback functions are the name of functions that are "called back" within a parent component.
- The most common usage is to have a parent component with a state variable, but you want to update that state from a child component.
- What do you do? You pass down a callback function to the child from the parent. That allows us to update state in the parent component.
- useCallback memoizes our callback functions, so they not recreated on every re-render. Using useCallback correctly can improve the performance of our app.

```
import "./styles.css";
import React from "react";

export default function App() {

  const [skill, setSkill] = React.useState("");
  const [skills, setSkills] = React.useState(["HTML", "CSS",
"JavaScript"]);

  function handleChangeInput(event) {
    setSkill(event.target.value);
  }

  function handleAddSkill() {
    setSkills(skills.concat(skill));
  }

  const handleRemoveSkill = React.useCallback(
    (skill) => {
      setSkills(skills.filter((s) => s !== skill));
    },
    [skills]
  );

  return (
    <>
      <input onChange={handleChangeInput} />
      <button onClick={handleAddSkill}>Add Skill</button>
      <SkillList skills={skills}
handleRemoveSkill={handleRemoveSkill} />
    </>
  );
}
```

```jsx
const SkillList = React.memo(({ skills, handleRemoveSkill }) => {
  console.log("re-rendered whenever parent state is updated!");
  return (
    <ul>
      {skills.map((skill) => (
        <li key={skill} onClick={() => handleRemoveSkill(skill)}>
          {skill}
        </li>
      ))}
    </ul>
  );
});
```

5. useMemo Hook

useMemo Can Improve Expensive Operations

- useMemo is very similar to useCallback and helps improve performance. But instead of being for callbacks, it is for storing the results of expensive operations.
- useMemo allows us to memoize, or remember the result of expensive operations when they have already been made for certain inputs.
- Memoization means that if a calculation has been done before with a given input, there's no need to do it again, because we already have the stored result of that operation.
- useMemo returns a value from the computation, which is then stored in a variable.

```
import "./styles.css";
import React from "react";

const skills = ["HTML", "CSS", "JavaScript", "...1000s more"];

export default function App() {

  const [searchTerm, setSearchTerm] = React.useState("");

  const searchResults = React.useMemo(() => {
    return skills.filter((s) => s.includes(searchTerm));
  }, [searchTerm]);

  function handleSearchInput(event) {
    setSearchTerm(event.target.value);
  }

  return (
    <>
      <h3>Search Results</h3>
      <input onChange={handleSearchInput} />
      <ul>
        {searchResults.map((result, i) => (
          <li key={i}>{result}</li>
        ))}
      </ul>
    </>
  );
}
```

6. useContext Hook

useContext Helps Us Avoid Prop Drilling

- In React, we want to avoid the following problem of creating multiple props to pass data down two or more levels from a parent component.
- In some cases, it is fine to pass props through multiple components, but it is redundant to pass props through components which do not need it.
- Context is helpful for passing props down multiple levels of child components from a parent component and sharing state across our app component tree.
- The useContext hook removes the unusual-looking render props pattern that was required in consuming React Context before.
- Instead, useContext gives us a simple function to access the data we provided on the value prop of the Context Provider in any child component.

```
import "./styles.css";
import React from "react";

const UserContext = React.createContext();

export default function App() {
  const [user] = React.useState({ name: "Fred" });

  return (
    <UserContext.Provider value={user}>
      <Main />
    </UserContext.Provider>
  );
}

const Main = () => (
  <>
    <Header />
    <br />
    <div>Main app content</div>
  </>
);

const Header = () => {
  const user = React.useContext(UserContext);
  return <h1>Welcome, {user.name}!</h1>;
};
```

7. useReducer Hook

useReducer is (Another) Powerful State Management Tool

- useReducer is a hook for state management, much like useState, and relies upon a kind of function called a reducer.
- Reducers are simple, predictable (pure) functions that take a previous state object and an action object and return a new state object.
- useReducer can be used in many of the same ways that useState can, but is more helpful for managing state across multiple components that may involve different operations or "actions".
- You will need to reach for useReducer less than useState around your app. But it is very helpful as a powerful means of managing state in smaller applications, rather than having to reach for a third-party state management library like Redux.

```
export default function App() {
  const [state, dispatch] = React.useReducer(reducer, initialState);

  function handleLogin() {
    dispatch({
      type: "LOGIN",
      payload: {
        username: "Reed",
      },
    });
  }

  function handleSignout() {
    dispatch({
      type: "SIGNOUT",
    });
  }

  return (
    <>
      Current user: {state.username}
      <br />
      isAuthenticated: {JSON.stringify(state.isAuth)}
      <br />
      <button onClick={handleLogin}>Login</button>
      <button onClick={handleSignout}>Signout</button>
    </>
  );
}
```