React Hooks

Topics covered:

- What is the lifecycle of functional components?
- What are React Hooks?
    - useState Hook
    - useEffect Hook
- How to create custom hooks?

1. **Lifecycle of Functional components:**
    - Class-based components in React have access to a wide range of lifecycle functions, whereas functional components do not.
    - However, with the addition of Hooks, we now have one method that enables us to imitate the behavior of some of the class component lifecycle functions.

    We'll discover in-depth information regarding hooks.

All of this is possible because of the *useState* and *useEffect* hooks, two unique React capabilities that enable setting the initial state and utilizing lifecycle events in functional components. By skillfully utilizing these two hooks in your pure JavaScript routines, it is currently possible to simulate the performance of practically every supported lifecycle method.

2. **React Hooks:**

➔ React Hooks are simple JavaScript functions that can be used to separate reusable components from functional components. Hooks can be declarative and manage side effects.

➔ We can "*hook*" into React features like state and lifecycle methods using hooks.

Hooks follow three rules:

- Hooks can only be accessed from within *React function components*.
- Hooks can only be called at the component's top level.
- Hooks are not conditional.

➔ Note: Hooks cannot be used in class-based components.

React includes several standard built-in hooks:

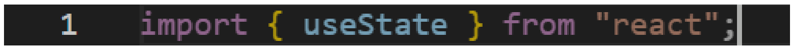
- ***useState****:* To control states. Returns a stateful value as well as an updater function.
- ***useEffect***: To manage side effects such as API calls, subscriptions, timers, mutations, and other things.
- ***useContext****:* To return the context's current value.
- *useReducer:* An alternative to useState for complex state management.

- ***useMemo****:* This method returns a memoized value that can be used to improve performance.
- ***useRef:*** This method returns a ref object with the .current property. The ref object can be changed. It is primarily used to gain immediate access to a child component.

➔ **useState:**

- We can keep track of the state in a function component with the React *useState* Hook.
- The state generally describes data or parameters that applications need to track.

It must first be imported into our component before we can utilize the useState Hook.



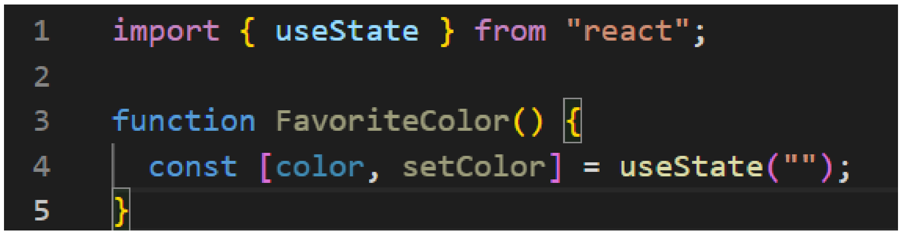
```
import { useState } from "react";
```

Because useState is a named export, we destruct it from reactjs.

useState is called in our function component to initialize our state.
useState takes a starting state and returns two values:

- The present state.
- A function for modifying the state.

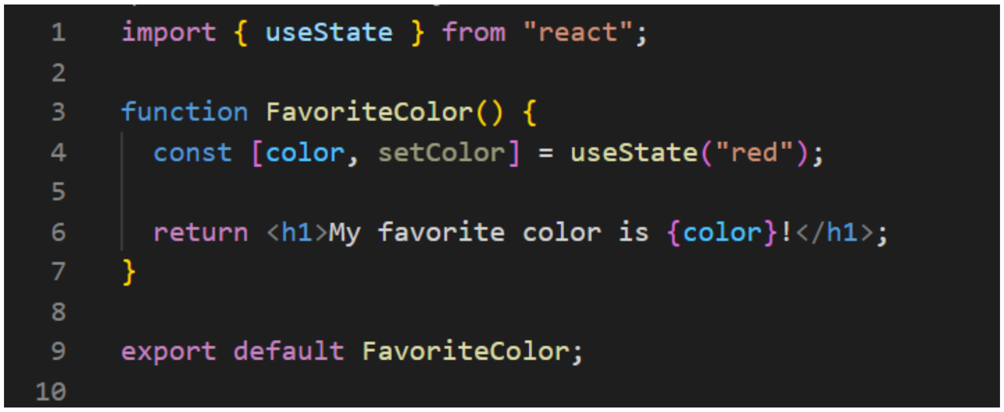
Example:
Initialize state at the function component's top:



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

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

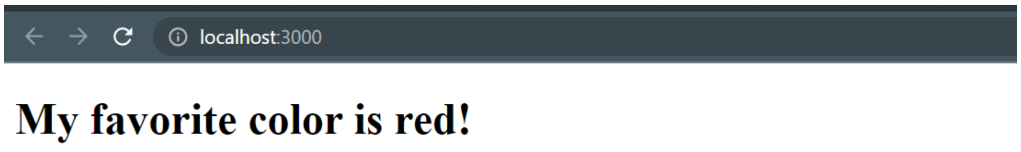
- Again, we are destructuring the values returned by ***useState***.
- The first value, color, represents our current situation.
- The function used to update our state is setColor, the second value.
- These are variables that can be named whatever you choose.
- Finally, we make the initial state an empty string: useState("")

Now, wherever we want in our component, we may include our state.



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

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>;
}

export default FavoriteColor;
```

Output:



localhost:3000

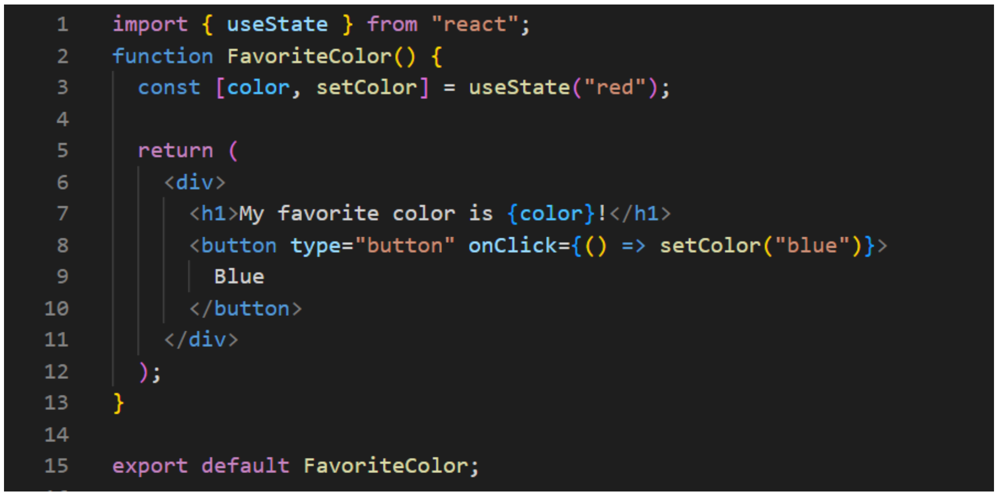
My favorite color is red!

Modify State:

- We utilize our state updater function to update our state.
- Never should we update state directly.

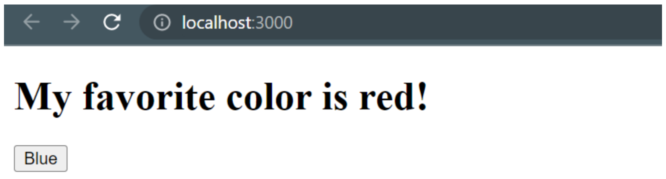
We use our state updater function to update our state.

We should never update state directly.



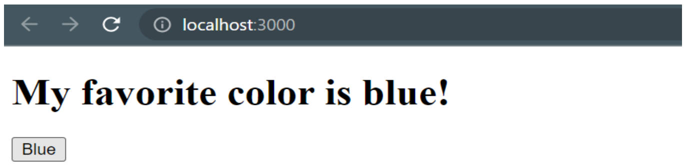
```
import { useState } from "react";
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <div>
      <h1>My favorite color is {color}!</h1>
      <button type="button" onClick={() => setColor("blue")}>
        Blue
      </button>
    </div>
  );
}

export default FavoriteColor;
```

Output:



localhost:3000

My favorite color is red!



localhost:3000

My favorite color is blue!

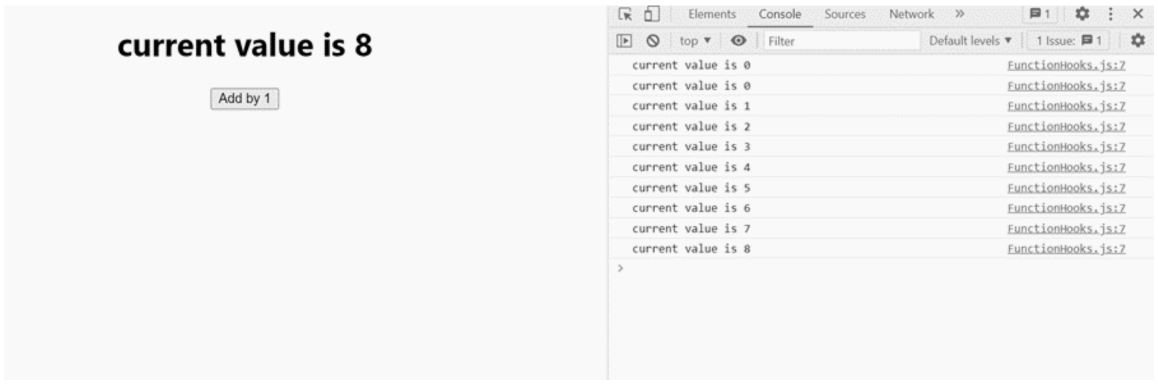
➔ **useEffect:**

- To perform side effects in your components, useEffect Hook is used.
- Fetching data, directly updating the DOM, and timers are examples of side effects.
- useEffect accepts two parameters. The second parameter is not required.

```
useEffect(<function>, <dependency>)
```

Let us see an example to understand what useEffect can do,

```
import React, { useState, useEffect } from "react";
export default function FunctionHooks() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    document.title = `current value is ${value}`;
    console.log(document.title);
  });
  return (
    <div>
      <h1>current value is {value}</h1>
      <button onClick={() => setValue(value + 1)}>Add by
1</button>
    </div>
  );
}
```



```
current value is 0        FunctionHooks.js:7
current value is 0        FunctionHooks.js:7
current value is 1        FunctionHooks.js:7
current value is 2        FunctionHooks.js:7
current value is 3        FunctionHooks.js:7
current value is 4        FunctionHooks.js:7
current value is 5        FunctionHooks.js:7
current value is 6        FunctionHooks.js:7
current value is 7        FunctionHooks.js:7
current value is 8        FunctionHooks.js:7
>
```

- Every render includes useEffect. That is, when the value changes, a render occurs, which then causes another effect to occur.
- This is not what we desire. There are several methods for controlling when side effects occur.
- The second parameter, which accepts an array, should always be included. In this array, we can optionally pass dependencies to useEffect.

When dependency is not passed:

```
useEffect(() => {
  //executes on every render
});
```

When an empty Array is passed:

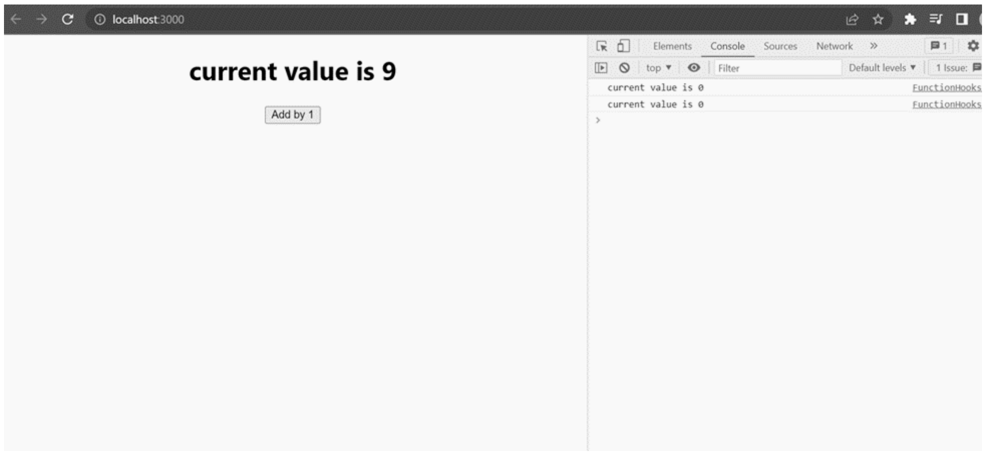
```
useEffect(() => {
  //Executes only on the first render
}, []);
```

When props and states are passed:

```
useEffect(() => {
  //Executes on the first render
  //When any dependency value changes
}, [prop, state]);
```

Let’s see what will happen when we pass an empty are in dependencies:

```
 import React, { useState, useEffect } from "react";

export default function FunctionHooks() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    document.title = `current value is ${value}`;
    console.log(document.title);
  }, []);

  return (
    <div>
      <h1>current value is {value}</h1>
      <button onClick={() => setValue(value + 1)}>Add by
1</button>
    </div>
  );
}
```

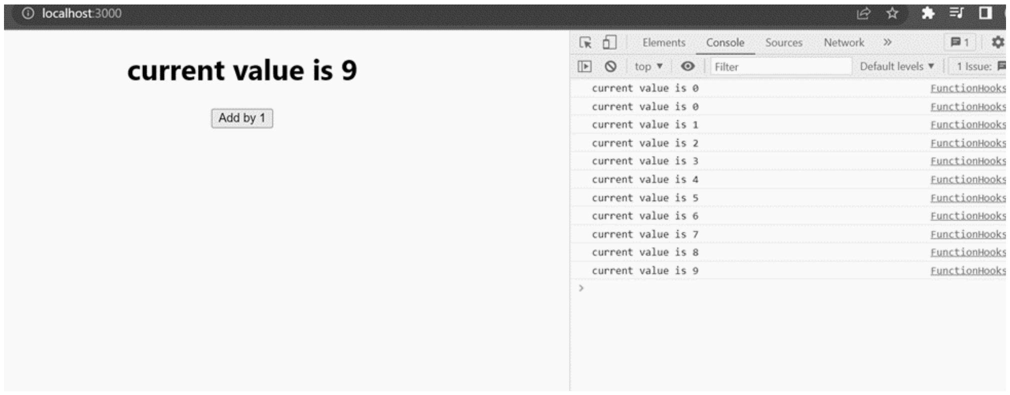


current value is 9

```
current value is 0
current value is 0
```

And what will happen if we pass value as a dependency;

```
import React, { useState, useEffect } from "react";

export default function FunctionHooks() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    document.title = `current value is ${value}`;
    console.log(document.title);
  }, [value]);

  return (
    <div>
      <h1>current value is {value}</h1>
      <button onClick={() => setValue(value + 1)}>Add by
1</button>
    </div>
  );
}
```

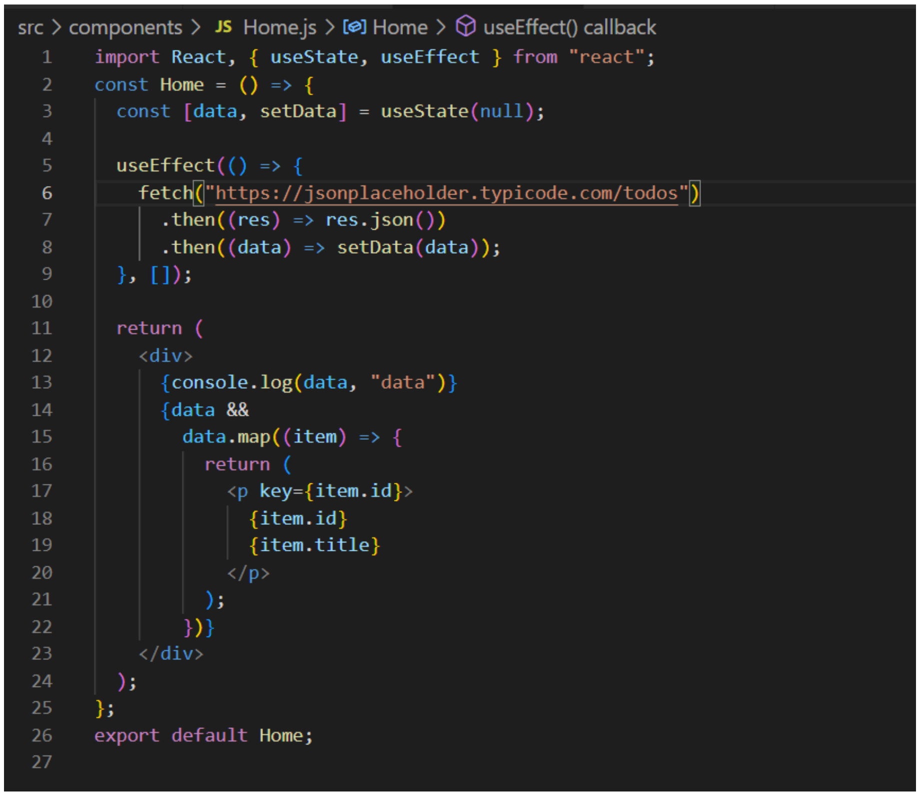


current value is 9

```
current value is 0
current value is 0
current value is 1
current value is 2
current value is 3
current value is 4
current value is 5
current value is 6
current value is 7
current value is 8
current value is 9
```

3. **Custom Hooks:**
    - Hooks are functions that can be reused.
    - When you have component logic that needs to be shared by several components, we can extract it to a custom Hook.

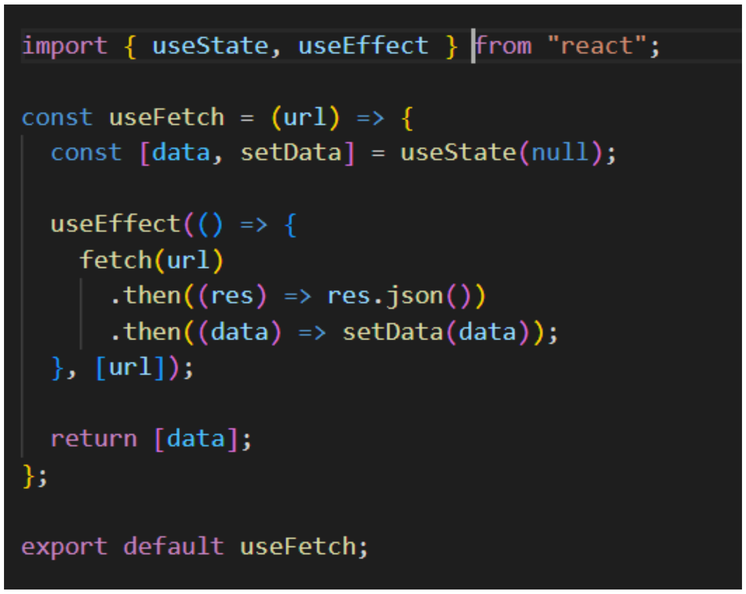
Create a Hook:
- The following code gets and displays data from our Home component.
- To get data, we'll use the JSONPlaceholder service.
- This service is ideal for testing apps when no data is available.
- See the JavaScript Fetch API section for further information.
- Use the JSONPlaceholder service to retrieve fictitious "todo" items and show their titles on the page:



```
import React, { useState, useEffect } from "react";
const Home = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then((res) => res.json())
      .then((data) => setData(data));
  }, []);

  return (
    <div>
      {console.log(data, "data")}
      {data &&
        data.map((item) => {
          return (
            <p key={item.id}>
              {item.id}
              {item.title}
            </p>
          );
        })}
    </div>
  );
};
export default Home;
```

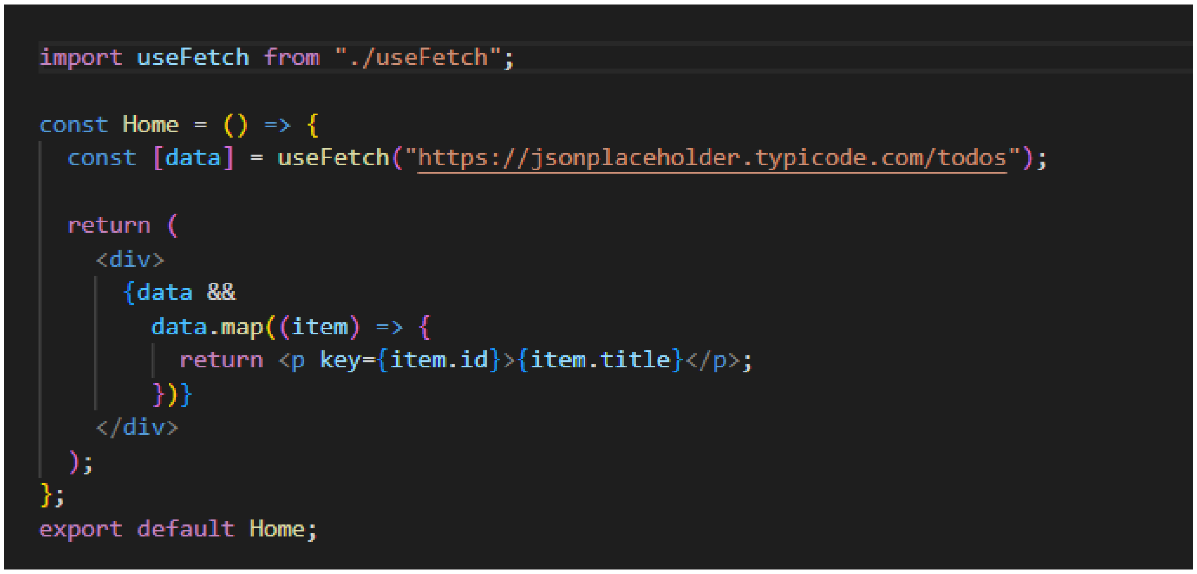
- Because the fetch logic may be used in other components, we will extract it into a new Hook.
- Fetch logic should be transferred to a new file and used as a custom hook:

useFetch.js:



```javascript
import { useState, useEffect } from "react";

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

Home.js



```javascript
import useFetch from "./useFetch";

const Home = () => {
  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <div>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </div>
  );
};
export default Home;
```

- The entire logic required to fetch our data is contained in a function named useFetch that we have built in a new file called useFetch.js.
- The hard-coded URL was deleted, and its place was taken by a url variable that could be provided to the unique Hook.
- Finally, we are sending data back to our Hook.
- We are importing our useFetch Hook into index.js and using it just like any other Hook. Here is where we provide the URL to use to retrieve the data.
- Now, any component can utilize this custom Hook to fetch data from any URL.