# React Hooks

Amit Kumar

# React Hooks

React is a library for building user interfaces and one of its perks is that the library itself imposes a strict data flow to the developer.

By enforcing a clear structure (container and presentational components) and a strict data flow (components react to state and props change) its easier than before to create well reasoned UI logic.
The basic theory in React is that a piece of UI can "react" in response to state changes. The basic form for expressing this flow was an ES6 class up until now.
Consider the following example, an ES6 class extending from React.Component, with an internal state:

```
import React, { Component } from "react";
export default class Button extends Component {
  constructor() {
    super();
    this.state = { buttonText: "Click me, please" };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(() => {
      return { buttonText: "Thanks, been clicked!" };
    });
  }
  render() {
    const { buttonText } = this.state;
    return <button onClick={this.handleClick}>{buttonText}</button>;
  }
}
```

As you can see from the code above the component's internal state gets mutated by this.setState when clicking the button. The text's button in turns reacts to this change and receives the updated text.

# React Hooks

With React hooks its possible to express the same logic without an ES6 class.

**Updating the state in React ... without setState**
So what options do we have for managing the internal state in React now that this.setState and classes are not a need anymore?
Enter the first, and most important React hook: useState. It's a function exposed by react itself, you'll import it in your components as:

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

After importing useState you'll destructure two values out of it:

```
const [buttonText, setButtonText] = useState("Click me, please")
```

Confused by this syntax? It's ES 2015 array destructuring.

The names above can be anything you want, it doesn't matter for React. I advise to use descriptive and meaningful variable names depending on the state's purpose.
The argument passed to useState is the actual initial state, the data that will be subject to changes. useState returns for you two bindings:
the actual value for the state
the state updater function for said state
So the previous example, a button component, with hooks becomes:

```
import React, { useState } from "react";
export default function Button() {
  const [buttonText, setButtonText] = useState("Click me, please");
  return (
    <button onClick={() => setButtonText("Thanks, been clicked!")}>
      {buttonText}
    </button>
  );
}
```

Let's now take a look at data fetching with Hooks.

# React Hooks

**In the beginning there was componentDidMount (and render props)**
Data fetching in React! Do you remember the old days of componentDidMount? Here's how to fetch data from an API for rendering out a list:

```
import React, { Component } from "react";
export default class DataLoader extends Component {
  state = { data: [] };
  componentDidMount() {
    fetch("http://localhost:3001/links/")
      .then(response => response.json())
      .then(data =>
        this.setState(() => {
          return { data };
        })
      );
  }
  render() {
    return (
      <div>
        <ul>
          {this.state.data.map(el => (
            <li key={el.id}>{el.title}</li>
          ))}
        </ul>
      </div>
    );
  }
}
```

# React Hooks

There are a couple of shortcomings in the above code, it's not reusable at all. With a render prop we can easily share the data with child components:

```
import React, { Component } from "react";

export default class DataLoader extends Component {
  state = { data: [] };

  componentDidMount() {
    fetch("http://localhost:3001/links/")
      .then(response => response.json())
      .then(data =>
        this.setState(() => {
          return { data };
        })
      );
  }

  render() {
    return this.props.render(this.state.data);
  }
}
```

# React Hooks

Now you would consume the component by providing a render prop from the outside:

```
<DataLoader
  render={data => {
    return (
      <div>
        <ul>
          {data.map(el => (
            <li key={el.id}>{el.title}</li>
          ))}
        </ul>
      </div>
    );
  }}
/>
```

Even this pattern (born for providing a nicer alternative to mixins and HOCs) has its shortcomings.

I guess that's the exact reason which led React engineers to come up with Hooks: provide a better ergonomics for encapsulating and reusing stateful logic in React.

So impatient as I am, one of the first thing I wanted to try with hooks was data fetching. What hook I'm supposed to use for fetching data? Let's see.

# React Hooks

**Fetching data with useEffect**
I thought data fetching with React hooks shouldn't look so different from useState. A quick glance at the documentation gave me a hint: useEffect could be the right tool for the job.

I read: "useEffect serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in React classes, but unified into a single API"

With this knowledge in hand I refactored the first version of Dataloader to use useEffect.
The component becomes a function and fetch gets called inside useEffect.
Moreover, instead of calling this.setState I can use setData (an arbitrary function extracted from useState):

```
import React, { useState, useEffect } from "react";
export default function DataLoader() {
  const [data, setData] = useState([]);
  useEffect(() => {
    fetch("http://localhost:3001/links/")
      .then(response => response.json())
      .then(data => setData(data));
  });
  return (
    <div>
      <ul>
        {data.map(el => (
          <li key={el.id}>{el.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

# React Hooks

At this point I thought "what could be wrong?". I launched the app. This is what I saw in the console:

It was clearly my fault because I've already got a hint of what was going on:

"useEffect serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount"

componentDidUpdate! it's a lifecycle method which runs every time a component gets new props, or a state change happens. That's the trick. If you call useEffect as I did you would see an infinite loop.

For fixing this "bug" you would need to pass an empty array as a second argument to useEffect:

```
//
  useEffect(() => {
    fetch("http://localhost:3001/links/")
      .then(response => response.json())
      .then(data => setData(data));
  }, []); // << super important array
```

//
This array contains so called dependencies for useEffect, that is, variables on which useEffect depends on to re-run.

When the array is empty, the effect runs only once.

# React Hooks

**Cleaning up the effect with useEffect**

Timers, listeners, and persistent connections (WebSocket and friends) are the most common causes of memory leaks in JavaScript.
Consider the following use of useEffect, where we open a connection to a Socket.Io server:

```
useEffect(() => {
  const socket = socketIOClient(ENDPOINT);
  socket.on("FromAPI", data => {
    setResponse(data);
  });
}, []);
```

The problem with this code is that the connection is hold open even after the component unmounts from the DOM (in response to a state change for example).
What not everybody knows about useEffect is that we can return a function to clean up the effect, that is, a function which runs when the component unmounts.
This is the equivalent of componentWillUnmount for classes. Our example becomes:

```
useEffect(() => {
  const socket = socketIOClient(ENDPOINT);
  socket.on("FromAPI", data => {
    setResponse(data);
  });
  return () => socket.disconnect();
}, []);
```

Now the connection closes as expected when the component unmounts.

# React Hooks

**Can I use render props with React hooks?**
There's no point in doing that. Our DataLoader component becomes:

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

export default function DataLoader(props) {
  const [data, setData] = useState([]);

  useEffect(() => {
    fetch("http://localhost:3001/links/")
      .then(response => response.json())
      .then(data => setData(data));
  }, []); // << super important array

  return props.render(data)
}
```

Now you would consume the component by providing a render prop from the outside as we did in the previous example.

Again, there's no point in doing this because Hooks are here to help share logic between components.

Let's see an example in the next section.

# React Hooks

**Your first custom React hook**

Instead of HOCs and render props, we can encapsulate our logic in a React hook and then import that hook whenever we feel the need. In our example we can create a custom hooks for fetching data.

A custom hook is a JavaScript function whose name starts with "use", as a convention. Easier done than said. Let's make a useFetch hook then:

```
// useFetch.js
import { useState, useEffect } from "react";

export default function useFetch(url) {
  const [data, setData] = useState([]);

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

  return data;
}
```

# React Hooks

This is how you would use the custom hook:

```
import React from "react";
import useFetch from "./useFetch";

export default function DataLoader(props) {
  const data = useFetch("http://localhost:3001/links/");
  return (
    <div>
      <ul>
        {data.map(el => (
          <li key={el.id}>{el.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

This is what make hooks so appealing: finally we have a nice, standardized, and clean way for encapsulating and sharing logic.

NOTE: I didn't account for fetch errors in the code above, do your homework!

# React Hooks

**Can I use async/await with useEffect?**

When playing with useEffect I wanted to try async/await inside the hook. Let's see our custom hook for a moment:

```
// useFetch.js
import { useState, useEffect } from "react";
export default function useFetch(url) {
  const [data, setData] = useState([]);
  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => setData(data));
  }, []);
  return data;
}
```

For refactoring to async/await the most natural thing you would do is probably:

```
// useFetch.js
import { useState, useEffect } from "react";
export default function useFetch(url) {
  const [data, setData] = useState([]);
  useEffect(async () => {
    const response = await fetch(url);
    const data = await response.json();
    setData(data);
  }, []);
  return data;
}
```

# React Hooks

"Warning: An Effect function must not return anything besides a function, which is used for clean-up." Followed by a complete explanation of what I was doing wrong. How nice!

Turns out you cannot return a Promise from useEffect.

JavaScript async functions always return a promise and useEffect should exclusively return another function, which is used for cleaning up the effect.

That is, if you were to start setInterval in useEffect you would return a function for cleaning up the interval to avoid memory leaks.

So for making React happy we could rewrite our asynchronous logic like so:

```
// useFetch.js
import { useState, useEffect } from "react";
export default function useFetch(url) {
  const [data, setData] = useState([]);

  async function getData() {
    const response = await fetch(url);
    const data = await response.json();
    setData(data);
  }
  useEffect(() => {
    getData();
  }, []);
  return data;
}
```

Your custom hook will work again.

# React Hooks

**Complex state changes with useReducer**
Similar to useState, useReducer is another hook, convenient for dealing with more complex state changes in React components.
useReducer borrows some theory from Redux, namely the concepts of reducers, action, and dispatch.
To understand how useReducer works take a look at the following custom hook:

```
export function useFetch(endpoint) {
  const [data, dispatch] = useReducer(apiReducer, initialState);

  useEffect(() => {
    dispatch({ type: "DATA_FETCH_START" });

    fetch(endpoint)
      .then(response => {
        if (!response.ok) throw Error(response.statusText);
        return response.json();
      })
      .then(json => {
        dispatch({ type: "DATA_FETCH_SUCCESS", payload: json });
      })
      .catch(error => {
        dispatch({ type: "DATA_FETCH_FAILURE", payload: error.message });
      });
  }, []);

  return data;
}
```

# React Hooks

Here we call the hook by passing in a reducer (you'll see it in a moment), and an initial state:

const [data, dispatch] = useReducer(apiReducer, initialState);
In exchange, we get a state, data and a function for dispatching actions. Then to dispatch actions, which are handled by the reducer to change the state, we call dispatch in our code:

```
 useEffect(() => {
   // dispatch an action
   dispatch({ type: "DATA_FETCH_START" });

   fetch(endpoint)
     .then(response => {
       if (!response.ok) throw Error(response.statusText);
       return response.json();
     })
     .then(json => {
       // dispatch an action on success
       dispatch({ type: "DATA_FETCH_SUCCESS", payload: json });
     })
     .catch(error => {
       // dispatch an action on error
       dispatch({ type: "DATA_FETCH_FAILURE", payload: error.message });
     });
 }, []);
```

These actions end up in a reducer function to calculate the next state:

```
const initialState = {
  loading: "",
  error: "",
  data: []
};
```

# React Hooks

```
function apiReducer(state, action) {
  switch (action.type) {
    case "DATA_FETCH_START":
      return { ...state, loading: "yes" };
    case "DATA_FETCH_FAILURE":
      return { ...state, loading: "", error: action.payload };
    case "DATA_FETCH_SUCCESS":
      return { ...state, loading: "", data: action.payload };
    default:
      return state;
  }
}
```

Here's the complete example:

```
import { useEffect, useReducer } from "react";
const initialState = {
  loading: "",
  error: "",
  data: []
};
function apiReducer(state, action) {
  switch (action.type) {
    case "DATA_FETCH_START":
      return { ...state, loading: "yes" };
    case "DATA_FETCH_FAILURE":
      return { ...state, loading: "", error: action.payload };
    case "DATA_FETCH_SUCCESS":
      return { ...state, loading: "", data: action.payload };
    default:
      return state;
  }
}
```

# React Hooks

```
export function useFetch(endpoint) {
  const [data, dispatch] = useReducer(apiReducer, initialState);

  useEffect(() => {
    dispatch({ type: "DATA_FETCH_START" });

    fetch(endpoint)
      .then(response => {
        if (!response.ok) throw Error(response.statusText);
        return response.json();
      })
      .then(json => {
        dispatch({ type: "DATA_FETCH_SUCCESS", payload: json });
      })
      .catch(error => {
        dispatch({ type: "DATA_FETCH_FAILURE", payload: error.message });
      });
  }, []);

  return data;
}
```

Wrapping up, and resources

React hooks are a nice addition to the library. Born as an RFC in November 2018 they caught up quickly and landed in React 16.8.

React hooks make render props and HOCs almost obsolete and provide a nicer ergonomics for sharing stateful logic.

# React Hooks

**How to Memoize with React.useMemo()**

From time to time React components have to perform expensive calculations. For example, given a big list of employees and a search query, the component should filter the employees' names by the query.

In such cases, with care, you can try to improve the performance of your components using the memorization technique.

***useMemo()* hook**

useMemo() is a built-in React hook that accepts 2 arguments — a function compute that computes a result and the depedencies array:

```
const memoizedResult = useMemo(compute, dependencies);
```

During initial rendering, useMemo(compute, dependencies) invokes compute, memoizes the calculation result, and returns it to the component.
If during next renderings the dependencies don't change, then useMemo() doesn't invoke compute but returns the memoized value.
But if dependencies change during re-rendering, then useMemo() invokes compute, memoizes the new value, and returns it.

That's the essence of useMemo() hook.

If your computation callback uses props or state values, then be sure to indicate these values as dependencies:

```
const memoizedResult = useMemo(() => {
  return expensiveFunction(propA, propB);
}, [propA, propB]);
```

# React Hooks

**Example:**

A component <CalculateFactorial /> calculates the factorial of a number introduced into an input field.
Here's a possible implementation of <CalculateFactorial /> component:

```
import { useState } from 'react';
export function CalculateFactorial() {
  const [number, setNumber] = useState(1);
  const [inc, setInc] = useState(0);
  const factorial = factorialOf(number);
  const onChange = event => {
    setNumber(Number(event.target.value));
  };
  const onClick = () => setInc(i => i + 1);
  return (
    <div>
      Factorial of
      <input type="number" value={number} onChange={onChange} />
      is {factorial}
      <button onClick={onClick}>Re-render</button>
    </div>
  );
}
function factorialOf(n) {
  console.log('factorialOf(n) called!');
  return n <= 0 ? 1 : n * factorialOf(n - 1);
}
```

# React Hooks

**Understanding:**
Every time you change the input value, the factorial is calculated factorialOf(n) and 'factorialOf(n) called!' is logged to console.
On the other side, each time you click Re-render button, inc state value is updated. Updating inc state value triggers <CalculateFactorial /> re-rendering. But, as a secondary effect, during re-rendering the factorial is recalculated again — 'factorialOf(n) called!' is logged to console.
How can you memoize the factorial calculation when the component re-renders? Welcome useMemo() hook!
By using useMemo(() => factorialOf(number), [number]) instead of simple factorialOf(number), React memoizes the factorial calculation.
Let's improve <CalculateFactorial /> and memoize the factorial calculation:

```
import { useState, useMemo } from 'react';
export function CalculateFactorial() {
  const [number, setNumber] = useState(1);
  const [inc, setInc] = useState(0);
  const factorial = useMemo(() => factorialOf(number), [number]);
  const onChange = event => {
    setNumber(Number(event.target.value));
  };
  const onClick = () => setInc(i => i + 1);
  return (
    <div>
      Factorial of
      <input type="number" value={number} onChange={onChange} />
      is {factorial}
      <button onClick={onClick}>Re-render</button>
    </div>
  );
}
```

# React Hooks

**Use memoization with care:**

```
function factorialOf(n) {
  console.log('factorialOf(n) called!');
  return n <= 0 ? 1 : n * factorialOf(n - 1);
}
```

Every time you change the value of the number, 'factorialOf(n) called!' is logged to console. That’s expected.

However, if you click Re-render button, 'factorialOf(n) called!' isn’t logged to console because useMemo(() => factorialOf(number), [number]) returns the memoized factorial calculation. Great!

**Use memoization with care**

While useMemo() can improve the performance of the component, you have to make sure to profile the component with and without the hook. Only after that make the conclusion whether memoization worth it.

When memoization is used inappropriately, it could harm the performance.

**Conclusion**
useMemo(() => computation(a, b), [a, b]) is the hook that lets you memoize expensive computations. Given the same [a, b] dependencies, once memoized, the hook is going to return the memoized value without invoking computation(a, b).

# React Hooks

**useContext Hook:**

Context provides a way to pass data or state through the component tree without having to pass props down manually through each nested component. It is designed to share data that can be considered as global data for a tree of React components, such as the current authenticated user or theme(e.g. color, paddings, margins, font-sizes).

Context API uses Context. Provider and Context. Consumer Components pass down the data but it is very cumbersome to write the long functional code to use this Context API. So useContext hook helps to make the code more readable, less verbose and removes the need to introduce Consumer Component. The useContext hook is the new addition in React 16.8.

Syntax: const authContext = useContext(initialValue);

The useContext accepts the value provided by React.createContext and then re-render the component whenever its value changes but you can still optimize its performance by using memorization.

Example: Program to demonstrate the use of useContext Hook. In this example, we have a button, whenever we click on the button the onClick handler is getting triggered and it changes the authentication status(with a default value to Nopes) with the help of the useContext hook. Let's see the output of the above code:

# React Hooks

The most important are useState and useEffect. useState makes possible to use local state inside React components, without resorting to ES6 classes.

useEffect replaces componentDidMount, componentDidUpdate, and componentWillUnmount with a unified API.

For data fetching I wouldn't jump all in with useEffect ,a lot could still change in the near future with React's concurrent mode.

Similar to useState, useReducer is another hook, convenient for managing complex state changes.

**Five Important Rules for Hooks:**

1. Never call Hooks from inside a loop, condition or nested function
2. Hooks should sit at the top-level of your component
3. Only call Hooks from React functional components
4. Never call a Hook from a regular function
5. Hooks can call other Hooks

It's easy to foresee where React is going: functional components all over the place!

References:
Best Practices: https://www.educative.io/blog/best-practices-react-developer