Hooks in Focus

LEARNING OBJECTIVES

- Hooks API and the motivation behind them.
- Learn to incorporate stateful properties in a function component using the useState() hook
- Side effects using the useEffect hook
- Learn to ingest data from a Context provider using hooks
- Learn to implement the useReducer() hook
- Learn to write your own hook

Hooks in Focus

Understanding Hooks

“In the initial version of React, there were only Class Components!”

Class Components

They are sub-classed from the Component class in the React library and offer built-in support for state and lifecycle methods that allow developers to customize behaviour of the component easily.

Function Components

- Function Components are simple **JavaScript functions** that return a React element and can also take props.
- Simple to understand, readable and easier to debug than Class Components

Function Components: Scope of work

When a function component’s **scope of work expands** to necessitate the use of state and lifecycle like behaviour, the traditional approach has always been to **rewrite the component** as a class component.

How do you decide what type of component to build?

How do you decide what type of component to build?

A feature rich class component

A much lighter but feature deficient Function component

What if you build a function component and at a later stage it requires the use of state and lifecycle functions?

The Amazing Hooks API
Making React better since version 16.8

Motivation behind the Hooks API

1. Class Components Can Get Complex Quickly

- ➢ May lead to poor developer experience
- ➢ Makes debugging difficult
- ➢ Difficult to maintain

Motivation behind the Hooks API

2. Class Components are difficult to read and process

- ➢ Difficult to optimize
- ➢ Difficult to minify

Function Components are simple, faster, easier to read and optimize!

Motivation behind the Hooks API

3. Hooks allow you to write reusable & stateful logic easily!

Two patterns for authoring reusable logic

First pattern for authoring reusable logic

The first is **Render props** which enable you to hive off logic and state while enabling you to implement custom render logic.

Render props enabled components simply return a function that needs to return React components.

Second pattern for authoring reusable logic

The second technique is **higher order components** which are components that accept other components and return a new component that can be augmented with new features.

This allows you to **write components** that can power up existing components with new features and is a commonly used pattern in products such as **Redux.**

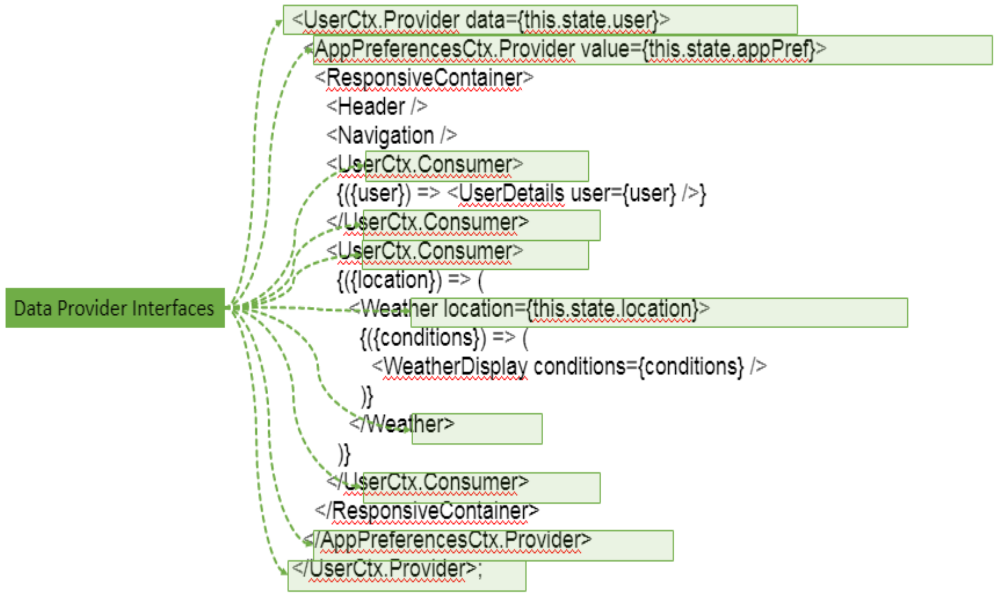
It can also be done using the Context API which involves the use of the Provider and Consumer components.

Wrapper Hell

Wrapper Hell

There are two popular types of sharing logic in components which are **Higher-Order Components** and **Rendering Props**.
When your applications have a massive quantity of nested Components, they will cause a “Wrapper Hell.”

Wrapper Hell: Example



```
<UserCtx.Provider data={this.state.user}>
 <AppPreferencesCtx.Provider value={this.state.appPref}>
  <ResponsiveContainer>
   <Header />
   <Navigation />
   <UserCtx.Consumer>
    {({user}) => <UserDetails user={user} />}
   </UserCtx.Consumer>
   <UserCtx.Consumer>
    {({location}) => (
     <Weather location={this.state.location}>
      {({conditions}) => (
       <WeatherDisplay conditions={conditions} />
      )}
     </Weather>
    )}
   </UserCtx.Consumer>
  </ResponsiveContainer>
 </AppPreferencesCtx.Provider>
</UserCtx.Provider>;
```

There is a need to untangle such complicated hierarchies and simplify components while still being able to author reusable and stateful logic.

HOOKS

Hooks

- ➢ Hooks let you take a simple function component and incrementally add state, logic and behavioural features such as side effects and more to Function components.

- ➢ These are core React features that were out of bounds for Function components.

- ➢ Hooks are optional which means you don’t have to use Hooks if you don’t want to and it doesn’t break any existing conventions or APIs.

- ➢ Hooks as a feature can be gradually brought into an evolving product without the need to rewrite components or parts of your application.

- ➢ One of the core benefits of React, as a library is that it can be gradually adopted. This makes it straightforward to build your product, knowing you can incrementally add features without breaking or completely rewriting your code.

STATE

SIDE EFFECTS

REDUCER

REF

HOOKS

MEMOISATION

CONTEXT

SYNCHRONOUS DOM OPS

Hooks

useState()	Create and update state properties
useEffect()	Run side effects. Can be used to simulate *componentDidMount(), componentDidUpdate()* & *componentWillUnmount()* like behavior
useContext()	Consume data from Context providers, without writing Provider and Consumer components
useReducer()	Creation and management of complex state logic

Hooks

- Hooks offer an opt-in way to create & share reusable logic
- They’re just functions that return data & methods

Hooks in Focus

The useState hook

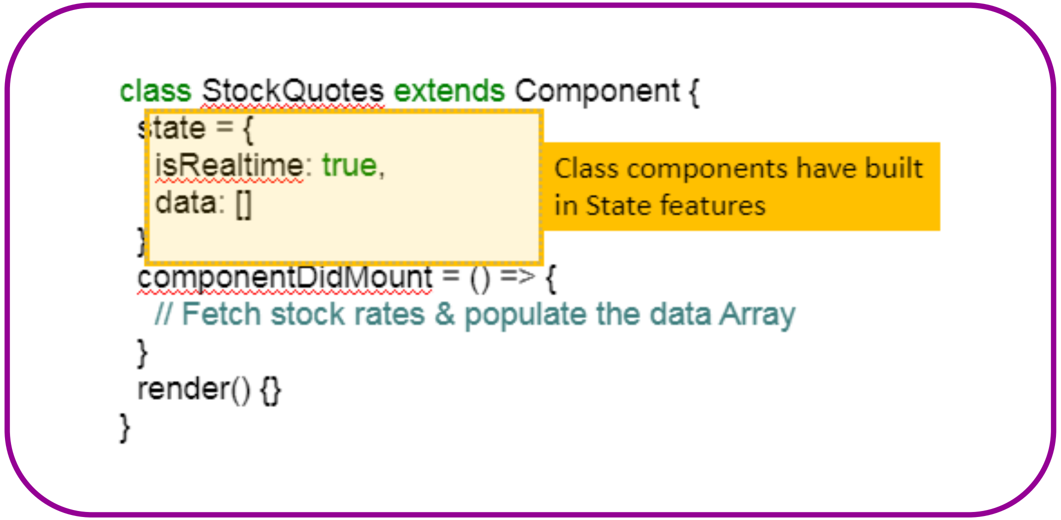
Function Components

- Function components when compared to class components offer a cleaner syntax that is easier to read, understand and debug.

- Function Components are just simple functions that accept arguments and return React elements. But function components do not offer standard React features like state, side effects and context.

Function Components: Example

For instance, state is a fundamental React feature and while class components offer built-in state features, a function component lacks the ability to offer stateful properties.



```
class StockQuotes extends Component {
  state = {
   isRealtime: true,
   data: []
  }
  componentDidMount = () => {
   // Fetch stock rates & populate the data Array
  }
  render() {}
}
```

With the **Hooks API**, we can easily plug in state to a function component by using the useState hook.

Hands On

- The useState hook function is amazingly simple to use and offers a straightforward way to add stateful properties to a function component.
- You can also set initial values to state properties by passing a value as an argument to the useState function.
- This can also be a function which can be used to compute a value first
- you can create as many stateful properties by simply using the useState function as many times you need.

useState hook: What should not be done?

When using the setter function, the state property is set as a whole and if you use this convention, as is the case in class components, you’ll simply end up overwriting the state object structure.

Hooks in Focus

Side effects using the useEffect hook

Class components let you run side effects in Lifecycle methods

```
class GetStockQuotes extends Component {
  state = {
    delay: 8000,
    stockRate: 0
  };
  fetchRates = async () => {
    if (this.props.symbol) {
      const {rate} = await getStockRates(this.props.symbol);
      this.setState({
        stockRate: rate
      });
    }
  };
  componentDidMount = () => {
    this.fetchRates(this.props.symbol);
    this.timer = setInterval(this.fetchRates, this.state.delay);
  };
  componentDidUpdate = (prevProps) => {
    if (prevProps.symbol !== this.props.symbol) {
             this.fetchRates(this.props.symbol);
    }
  }
  componentWillUnmount = () => {
    clearInterval(this.timer);
  };
  render() {
    return (
      <h1>
        {this.props.symbol} : {this.state.stockRate}
      </h1>
    );
  }
```

Fetching initial data using componentDidMount()

```
class GetStockQuotes extends Component {
 state = {
  delay: 8000,
  stockRate: 0
 };
 fetchRates = async () => {
  if (this.props.symbol) {
   const {rate} = await getStockRates(this.props.symbol);
   this.setState({
    stockRate: rate
   });
  }
 };
 componentDidMount = () => {
  this.fetchRates(this.props.symbol);
  this.timer = setInterval(this.fetchRates, this.state.delay);
 };
 componentDidUpdate = (prevProps) => {
  if (prevProps.symbol !== this.props.symbol) {
       this.fetchRates(this.props.symbol);
  }
 }
 componentWillUnmount = () => {
  clearInterval(this.timer);
 };
 render() {
  return (
   <h1>
    {this.props.symbol} : {this.state.stockRate}
   </h1>
  );
 }
```

Runs whenever the component updates

```
class GetStockQuotes extends Component {
  state = {
    delay: 8000,
    stockRate: 0
  };
  fetchRates = async () => {
    if (this.props.symbol) {
      const {rate} = await getStockRates(this.props.symbol);
      this.setState({
        stockRate: rate
      });
    }
  };
  componentDidMount = () => {
    this.fetchRates(this.props.symbol);
    this.timer = setInterval(this.fetchRates, this.state.delay);
  };
  componentDidUpdate = (prevProps) => {
    if (prevProps.symbol !== this.props.symbol) {
             this.fetchRates(this.props.symbol);
    }
  }
  componentWillUnmount = () => {
    clearInterval(this.timer);
  };
  render() {
    return (
      <h1>
        {this.props.symbol} : {this.state.stockRate}
      </h1>
    );
  }
```

Runs whenever the component is unmounted

```
function StockQuotes({symbol, rate})
=> {
  return <div>{symbol} : {rate}</div>
}
```

```
class GetStockQuotes extends Component {
  state = {
    delay: 8000,
    stockRate: 0
  };
  fetchRates = async () => {
    if (this.props.symbol) {
      const {rate} = await getStockRates(this.props.symbol);
      this.setState({
        stockRate: rate
      });
    }
  };
  componentDidMount = () => {
    this.fetchRates(this.props.symbol);
    this.timer = setInterval(this.fetchRates, this.state.delay);
  };
  componentDidUpdate = (prevProps) => {
    if (prevProps.symbol !== this.props.symbol) {
             this.fetchRates(this.props.symbol);
    }
  }
  componentWillUnmount = () => {
    clearInterval(this.timer);
  };
  render() {
    return (
      <h1>
        {this.props.symbol} : {this.state.stockRate}

    );
```

Side effects using the **useEffect()** hook

```
function GetStocks({symbol}) {
  const [rate, setRate] = useState(0);

  useEffect(() => {
    getRates(symbol).then(quote =>
setRate(quote));
  });


  return <div>{symbol} : {rate}</div>
}
```

The useEffect hook runs after every render when the DOM is updated.

```
function GetStocks({symbol}) {
  const [rate, setRate] = useState(0);

  useEffect(() => {
    getRates(symbol).then(quote =>
setRate(quote));
  });


  return <div>{symbol} : {rate}</div>
}
```

Download and Setup

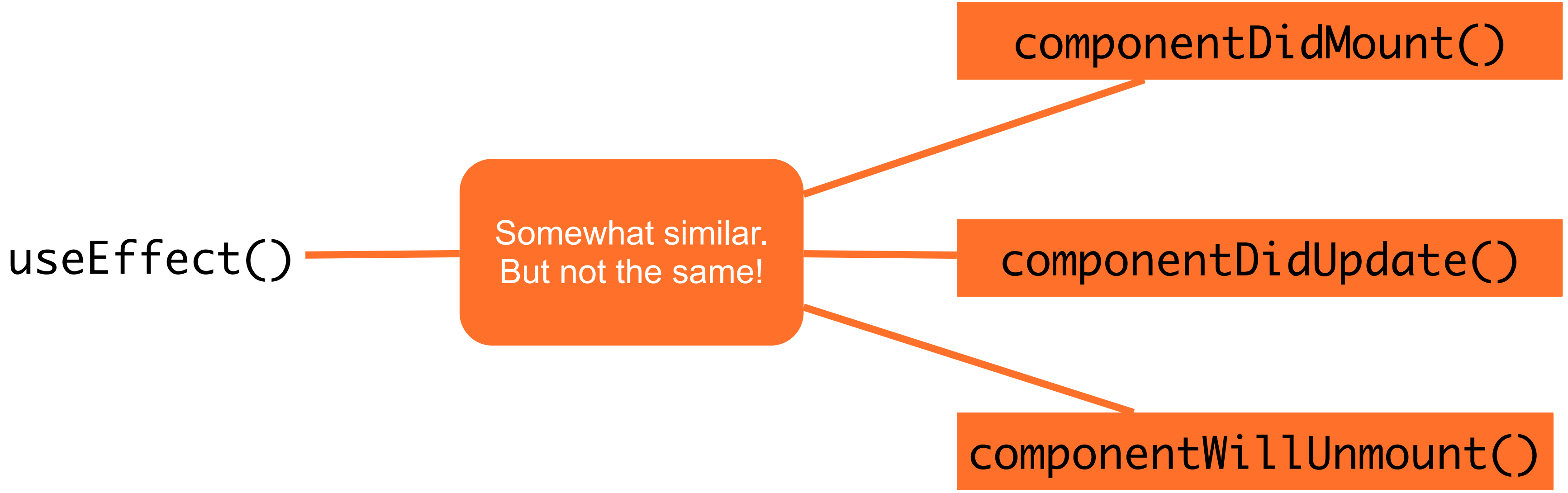
https://my.pcloud.com/publink/show?
code=XZ7XkQkZnJWXWbnhH7zqQujOKjLWAQPoiDG
V

HOOKS ARE VERY DIFFERENT FROM LIFECYCLE METHODS

```
useEffect(() => {
    let timer = setInterval(() => {
      getQuote(selected).then(q => setQuote(q));
    }, 5000);


    return () => clearInterval(timer);
  }, [selected]);
```

AN AMALGAMATION OF FUNCTIONALITY



It is best **NOT** to compare the useEffect hook with Lifecycle functions

useEffect HOOK RUNS EVERY TIME A COMPONENT RENDERS OR UPDATES

Lifecycle methods run at specific milestones during the life of a class component

useEffect() runs every time a function component renders.

The DOM has updated by the time useEffect runs

useEffect() synchronizes a side effect and consequential update with one or more dependencies

RUNS EVERY SINGLE TIME A COMPONENT RENDERS

```
function GetStocks({symbol}) {
  const [rate, setRate] = useState(0);
  useEffect(() => {
    getRates(symbol).then(quote =>
setRate(quote));
  });


  return <div>{symbol} : {rate}</div>
}
```

A runaway side-effect

```
useEffect(() => {
   getCategories().then(categs => {
     if (categs.length > 0) {
       setCategories(categs);
       setSelected(categs[0]);
     }
   });
 }, []);
```

This will ensure that the side effect only runs once when the component renders for the first time

```
useEffect(() => {
   getCategories().then(categs => {
     if (categs.length > 0) {
       setCategories(categs);
       setSelected(categs[0]);
     }
   });
 }, []);
```

This will ensure that the side effect only runs once when the component renders for the first time

Loosely similar to componentDidMount() (Though not the same!)

ALL INSTANCES OF useEffect WILL RUN IN THE FIRST RENDER

```
useEffect(() => {
    selected && getQuote(selected).then(q => setQuote(q));
  }, [selected]);
```

- **This effect will run on the first render**
- **Thereafter this effect will only run when ‘selected’ updates and has a value different from the one in the previous render**
- **Loosely similar to componentDidUpdate() (But not the same!)**

IMPLEMENT CLEANUP

```
useEffect(() => {
    let timer = setInterval(() => {
      getQuote(selected).then(q => setQuote(q));
    }, 5000);


    return () => clearInterval(timer);
  }, [selec
```

- **This effect will run on the first render**
- **Thereafter this effect will only run when ‘selected’ updates and has a value different from the one in the previous render**
- **At every render, the returned function will provide cleanup from the previous render, allowing you to unsubscribe/cancel timers etc.**
- **Not exactly the same as componentWillUnmount()**

INCLUDE ALL PROPS STATE OR OTHER VARIABLES HERE THAT ARE NEEDED BY THE SIDE EF

```
const GetStocks = ({symbol}) => {
  const [rate, setRate] = useState(0);
  useEffect(() => {
    getRates(symbol).then(quote => setRate(quote));
  }, [symbol]);


  return <div>{symbol} : {rate}</div>
}
```

- **Dependencies must always be in the scope of the function component**
- **If you do not include dependencies, then values from the previous render may be used leading to inconsistencies/bugs**

```
if (condition) {
  useEffect(() => {}, [variable]);
}
```

✗

```
function Component () {
  function something() {
    useEffect(() => {}, [variable])
  }
}
```

✗

```
for(let x = 0; x <= 10; x ++) {
  useEffect(() => {}, [variable]);
}
```

✗

Do NOT invoke useEffect() within conditionals, functions or loops

SEPARATE CONCERNS AND IMPLEMENT MULTIPLE INSTANCES OF THE us

```
const App = () => {

  useEffect(() => {
    getCategories().then(categs => {
      if (categs.length > 0) {
        setCategories(categs);
        setSelected(categs[0]);
      }
    });
  }, []);

  useEffect(() => {
    selected && getQuote(selected).then(q => setQuote(q));
  }, [selected]);

   useEffect(() => {
    let timer = setInterval(() => {
      getQuote(selected).then(q => setQuote(q));
    }, 5000);


    return () => clearInterval(timer);
  }, [selected]);

  return ( );
```

You can use multiple instances of useEffect to run multiple side effects

EASY FOR YOU TO VISUALIZE WHICH SIDE EFFECT RUNS WHEN

```
const App = () => {

  useEffect(() => {
    getCategories().then(categs => {
      if (categs.length > 0) {
        setCategories(categs);
        setSelected(categs[0]);
      }
    });
  }, []);

  useEffect(() => {
    selected && getQuote(selected).then(q => setQuote(q));
  }, [selected]);

   useEffect(() => {
    let timer = setInterval(() => {
      getQuote(selected).then(q => setQuote(q));
    }, 5000);


    return () => clearInterval(timer);
  }, [selected]);

  return ( );
```

1. This will be applied first

2. This will be applied second

3. This will be applied at the end

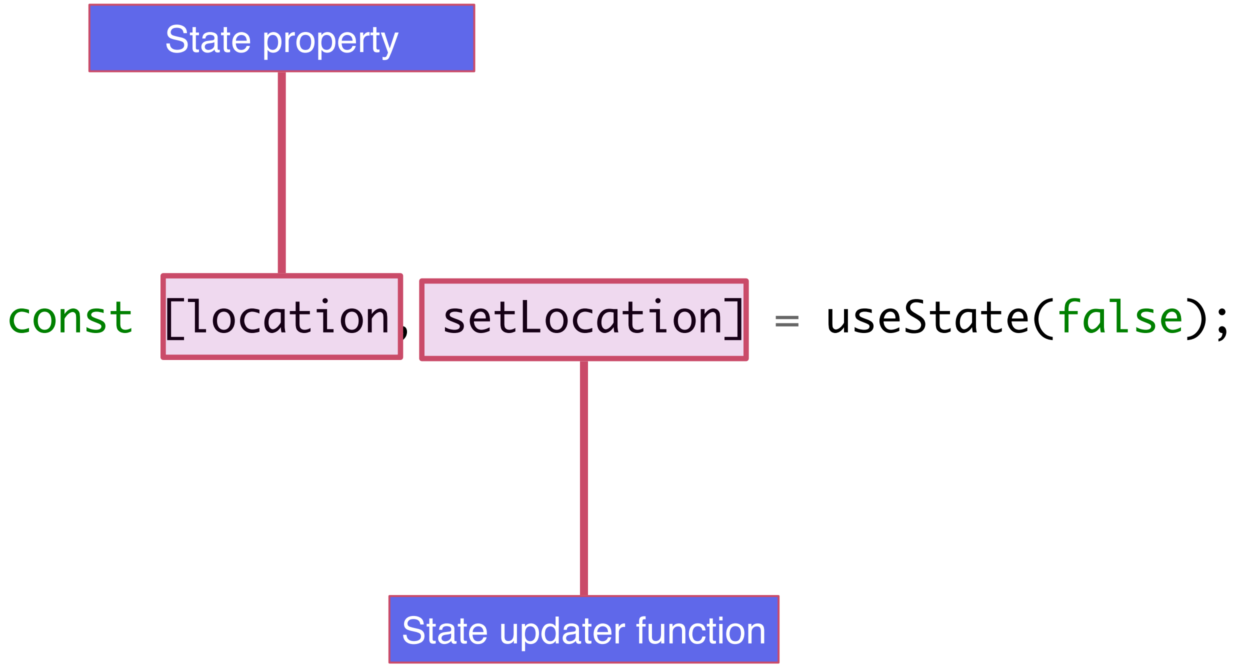
Hooks in Focus

The useReducer() hook

The useReducer() hook

```
const [location, setLocation] = useState(false);
```

The useReducer() hook



```
const [location, setLocation] = useState(false);
```

The useReducer() hook

useState() works great when you have individual and isolated stateful properties

```
const [name, setName] = useState("");
const [age, setAge] = useState(18);
const [showPanel, setShowPanel] = useState(false);
```

The useReducer() hook

```
{
    courseId: "crs-01",
    courseTitle: "Introduction to React.js",
    category: "JavaScript",
    lessonCount: 2,
    lessons: [
      {
        lessonId: "lss-01",
        title: "Pre-requisites",
        type: "text"
      },
      {
        lessonId: "lss-02",
        title: "Welcome to the course",
        type: "video"
      }
     ]
```

The useReducer() hook

```
const [course, setCourse] = useState({
    courseId: "crs-01",
    courseTitle: "Introduction to React.js",
    category: "JavaScript",
    lessonCount: 2,
    lessons: [
      {
        lessonId: "lss-01",
        title: "Pre-requisites",
        type: "text"
      },
      {
        lessonId: "lss-02",
        title: "Welcome to the course",
        type: "video"
      }
    ]
```

The useReducer() hook

Updating a state object using useState() would require you to manually merge all properties with updates

```
<button onClick={() => setCourse( Current Object + Updates )}>Update</button>
```

The useReducer() hook

```
const [courseId, setCourseId] = useState("");
const [courseTitle, setCourseTitle] = useState("");
const [category, setCategory] = useState("");
const [lessonCount, setLessonCount] = useState(0);
const [lessons, setLessons] = useState([]);
```

The useReducer() hook

```
const [courseId, [illegible]Cour[illegible]] = useState("");
const [courseTitle, [illegible]rseTitle] = useState("");
const [category, set[illegible]ory] = useState("");
const [lessonCount[illegible]onCount] = useState(0);
const [lessons, [illegible]Lesson[illegible] useState([]);
```

The useReducer() hook

```
{
    courseId: "crs-01",
    courseTitle: "Introduction to React.js",
    category: "JavaScript",
    lessonCount: 2,
    lessons: [
      {
        lessonId: "lss-01",
        title: "Pre-requisites",
        type: "text"
      },
      {
        lessonId: "lss-02",
        title: "Welcome to the course",
        type: "video"
      }
    ]
```

What is **lessonCount** depends on updates to the **lessons** array??

useState() doesn’t allow multiple dependent state transitions

The useReducer() hook

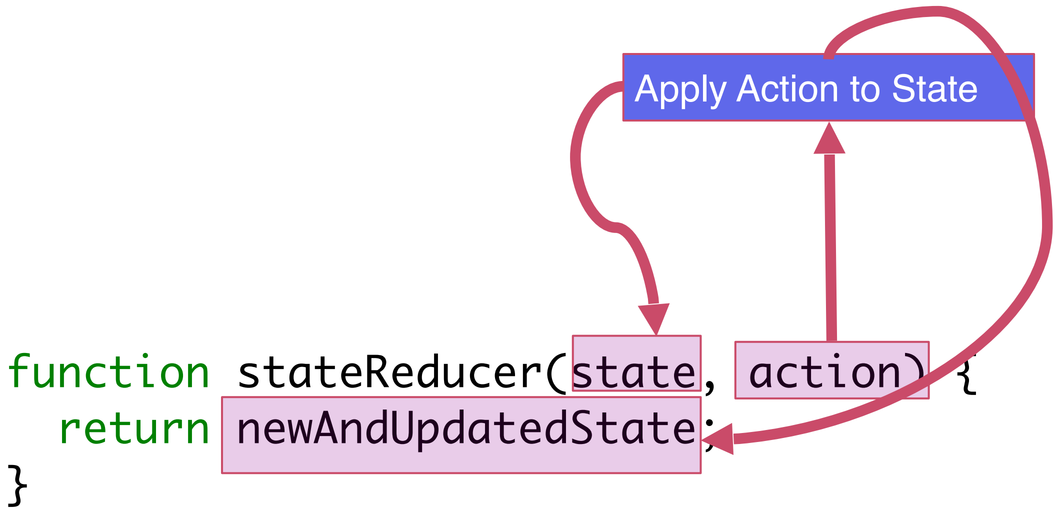
useReducer() hook offers a much better solution as compared to the useState hook.

A **reducer** is a pure function, which returns the same result for a given set of arguments in functional programming, making the results predictable.

The useReducer() hook

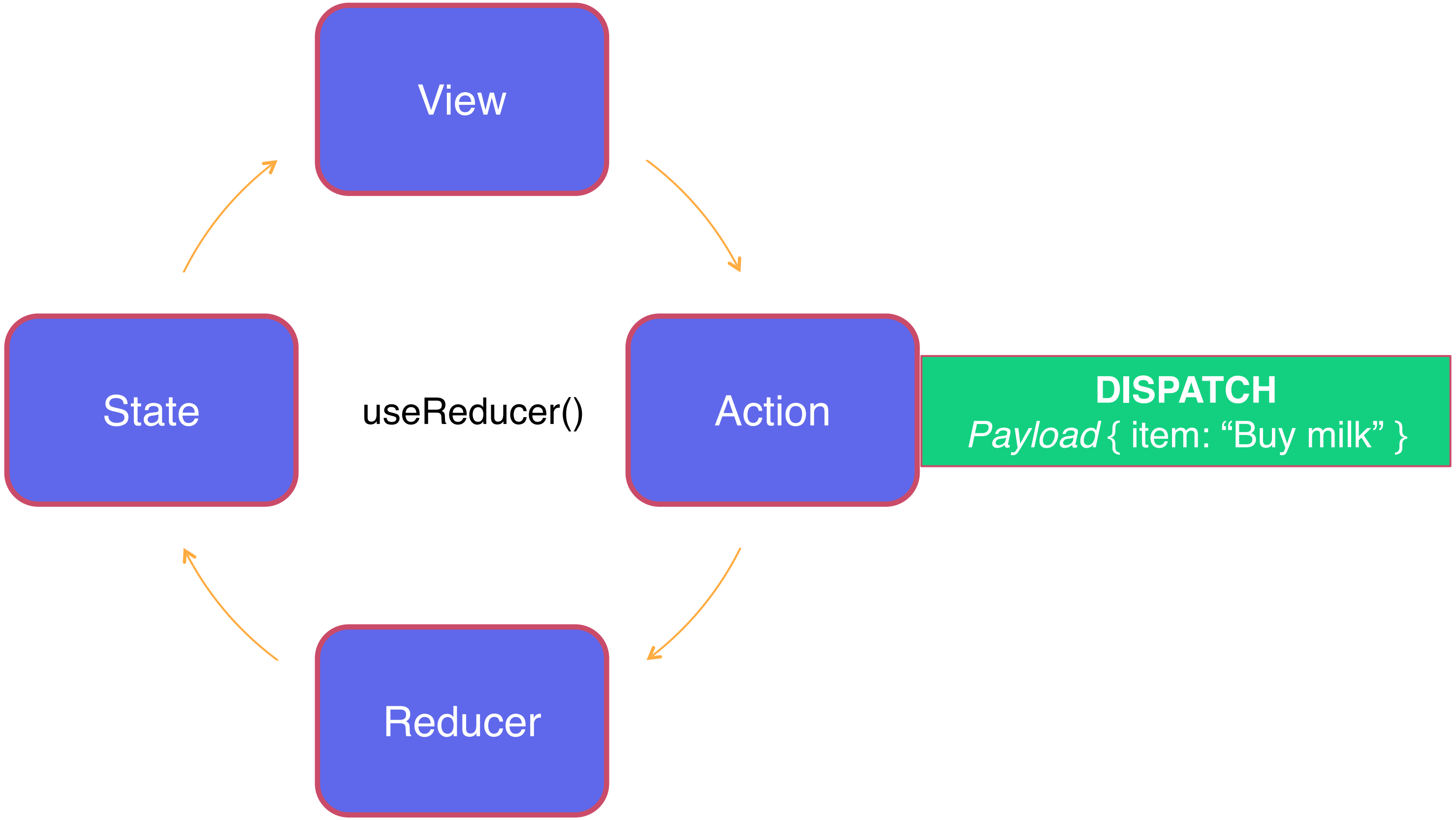
The **role of a Reducer** in React:

Reducer offers predictable state transitions based on the existing state and an action.
The reducer’s job is to apply the said action to the provided state and return a new and updated version of the state.

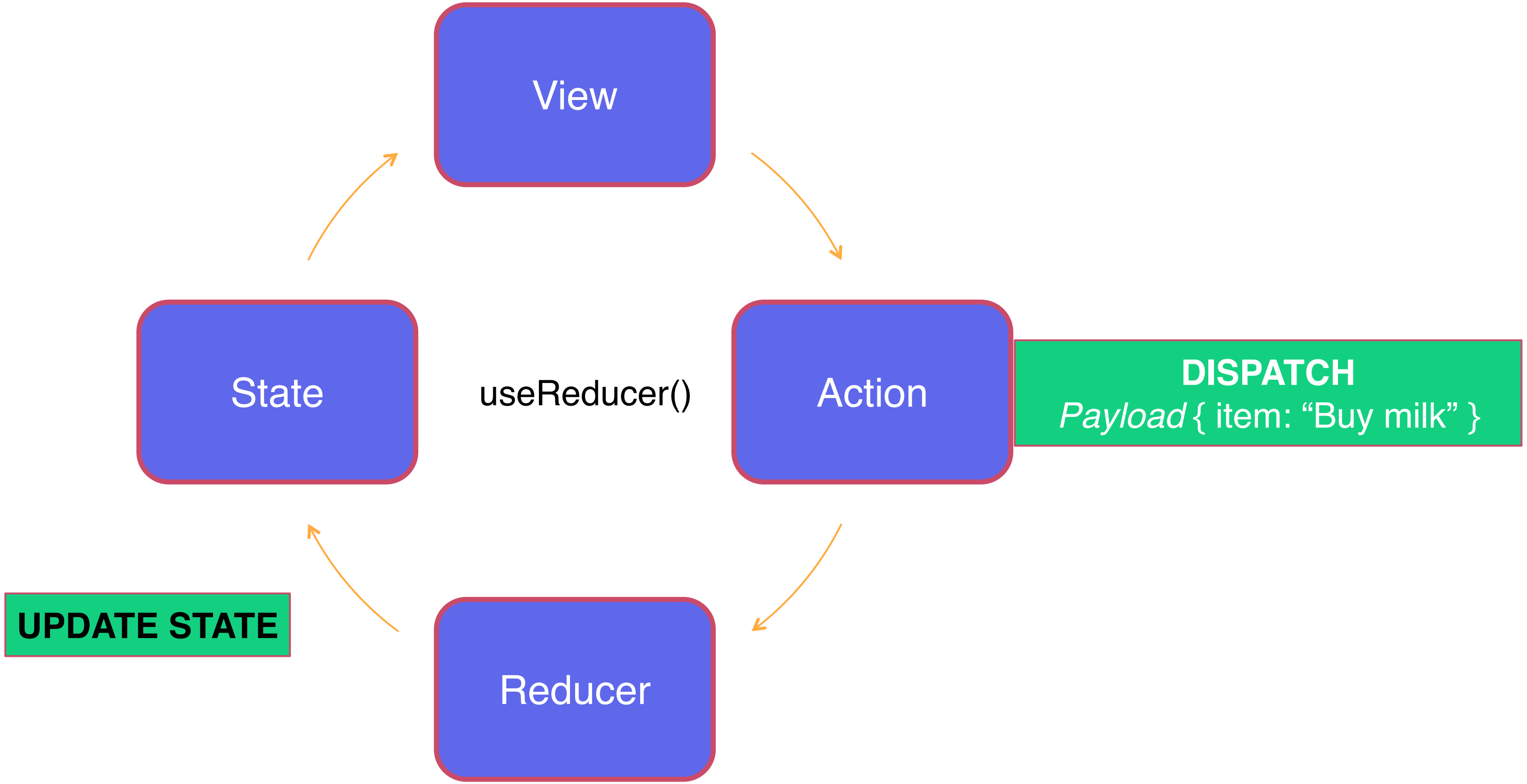


```
function stateReducer(state, action) {
  return newAndUpdatedState;
}
```

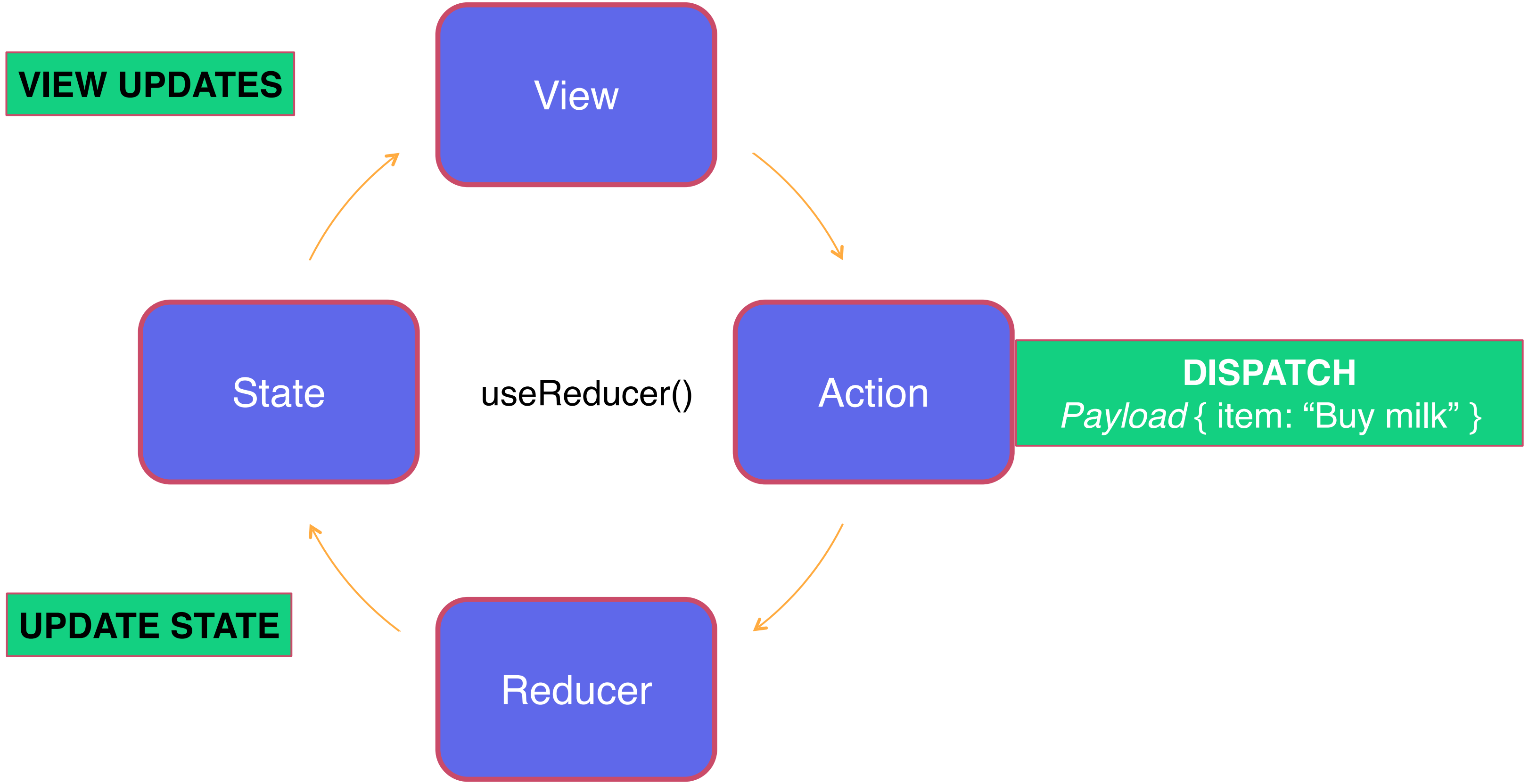
The useReducer() hook



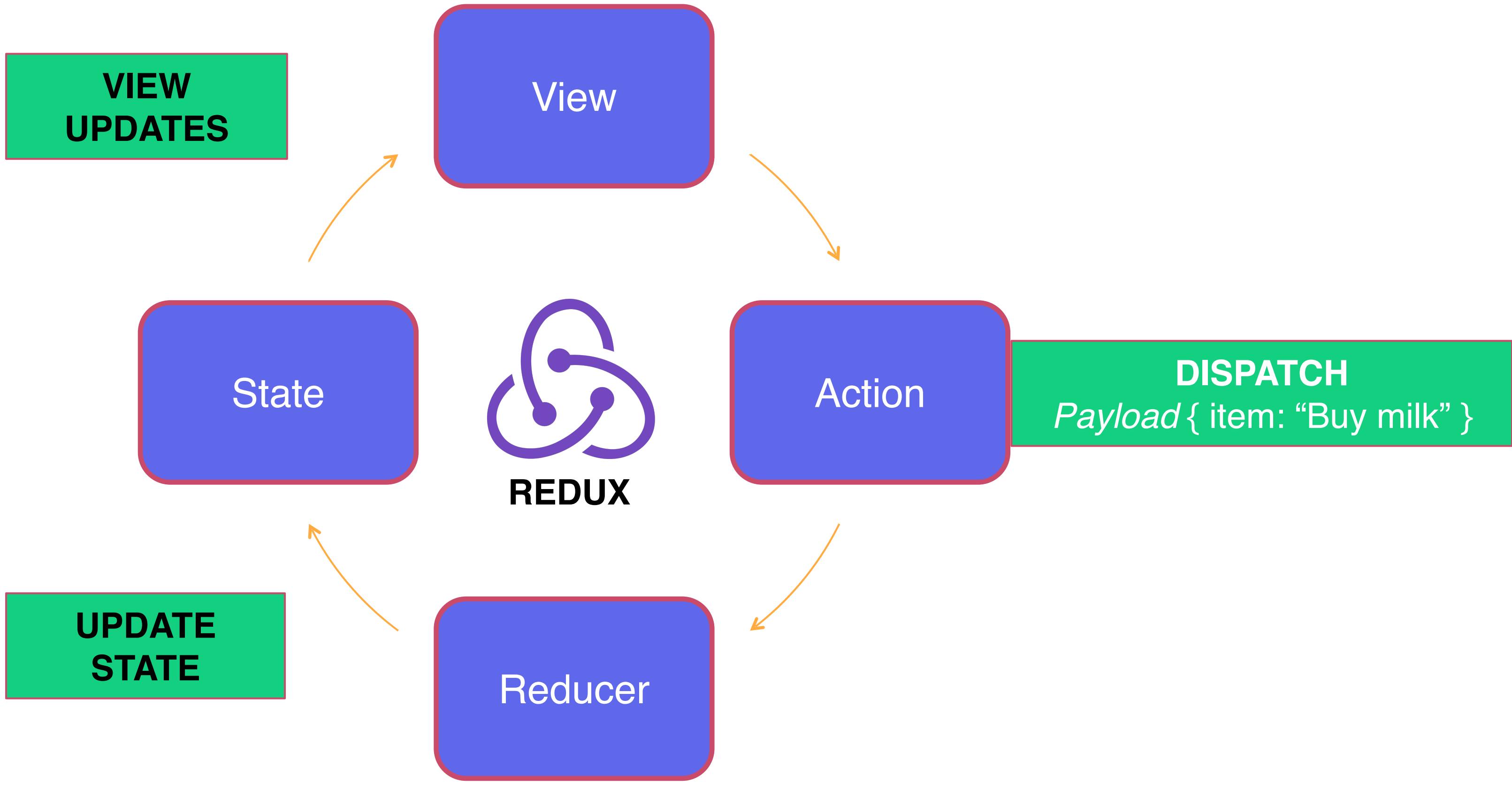
The useReducer() hook



The useReducer() hook



he useReducer() hook



The useReducer() hook

```
const [state, dispatch] = useReducer(courseReducer, course, initState);
```

- ✓ Predictable state container
- ✓ Predefined actions to update the state
- ✓ Testable

The useReducer() hook

Data Persistence Made Easy!

```
const [state, dispatch] = useReducer(reducer, init,
initStateFn);

useEffect(() => {
  // fetch initial state from storage
  // dispatch({type: "INIT_STATE", fetchedState})
},[]);

useEffect(() => {
  // persist to database
}, [state]);
```

Initialize application state by fetching data from storage on first render

Persist application state to storage whenever it updates

The useReducer() hook

useReducer()

✓ Complex State Logic & Transitions
✓ Multiple & Inter-dependent state transitions

Hooks in Focus

Build Your Own Hook!

Building Your Own Hook

A custom hook allows us to incorporate resolution dependent, responsive features in a function component.

They have incredible power and bring simplicity to a React application.

Building Your Own Hook

A Function Component

```
const App = () => {
  return (
    <div className="container">
      <div className="cover" />
      <FormPanel />
    </div>
  );
};
```

A Function Component using Hooks

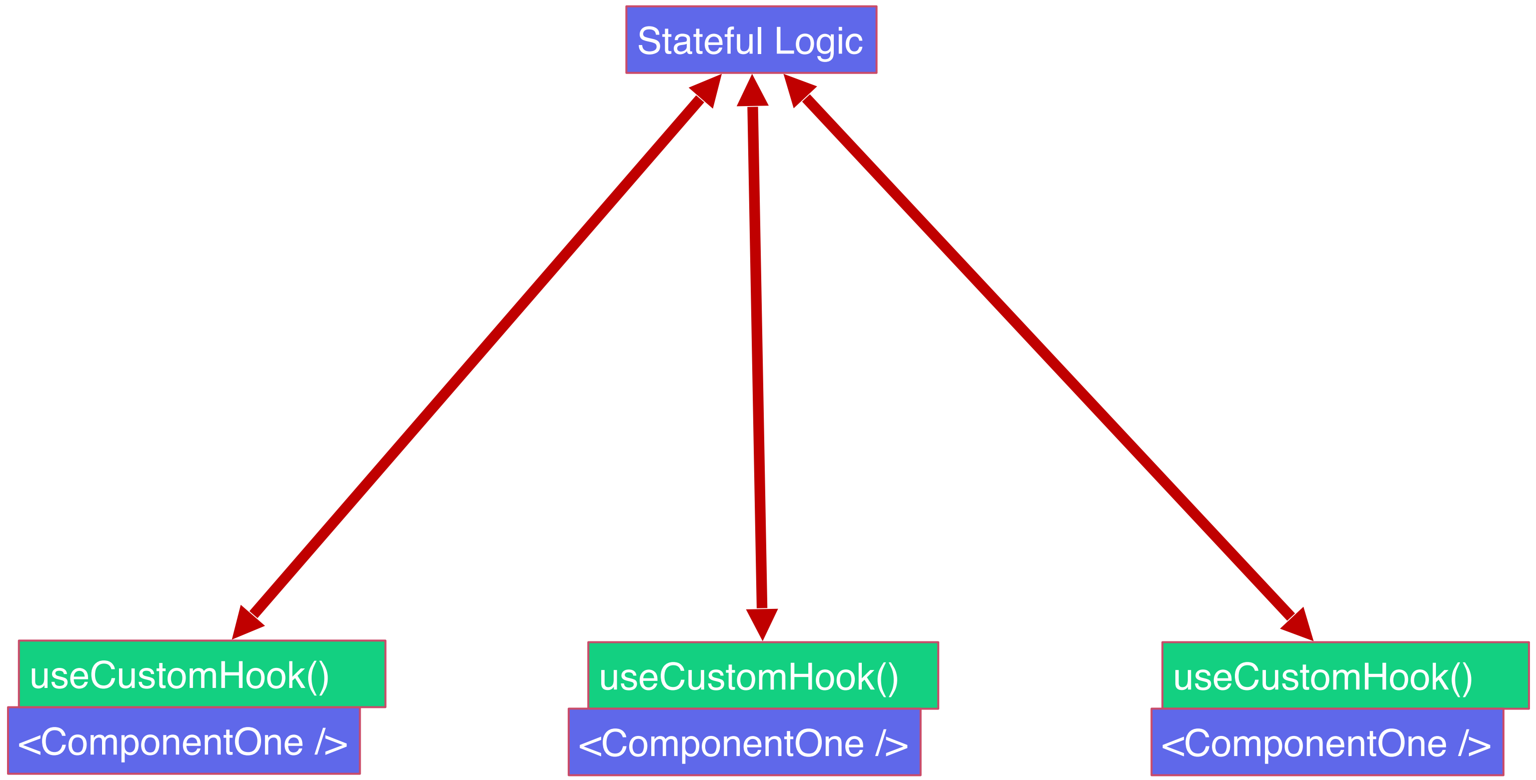
```
const App = () => {
  const {power} =
useSuperPowers();
  return (
    <div className="container">
      <div className="cover" />
      <FormPanel />
    </div>
  );
};
```

Building Your Own Hook

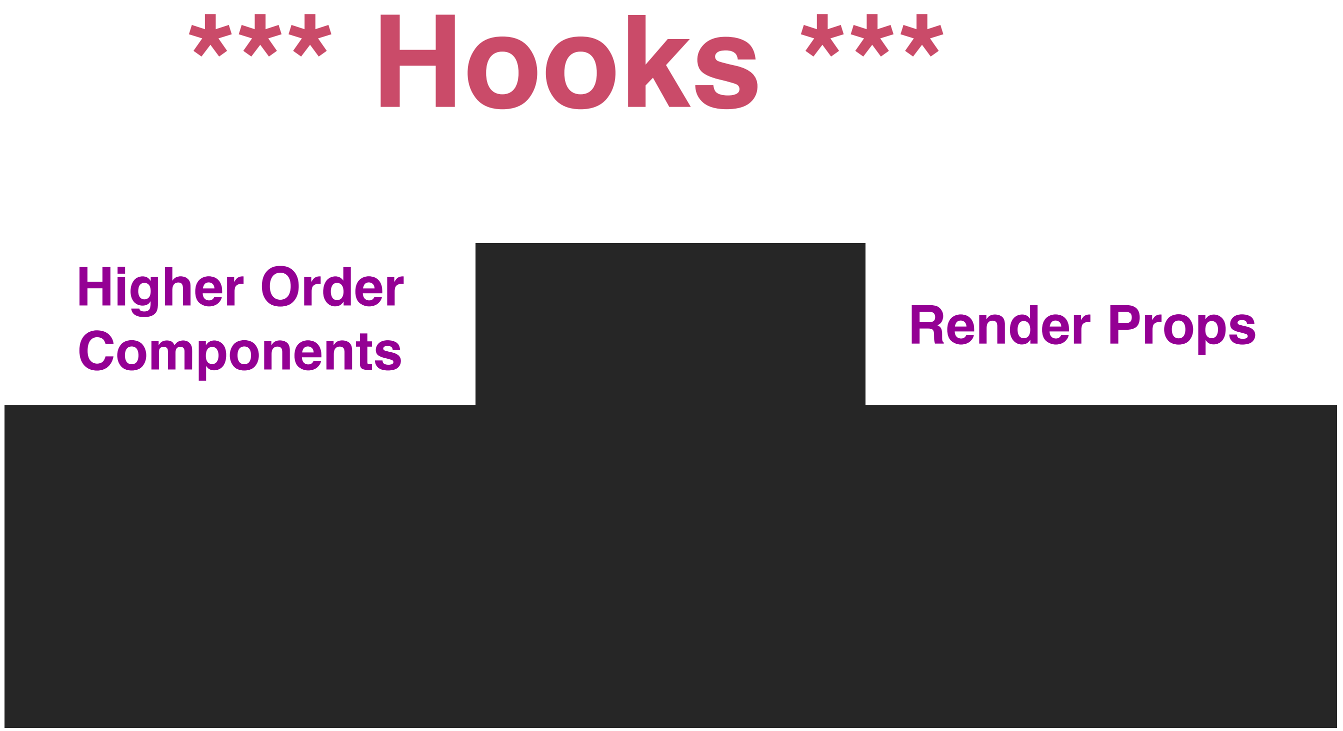
One can write and share their own hook!

Your own hook could be a custom feature or service that your application uses, or it could be a feature that the entire React community might find useful.

Building Your Own Hook



Building Your Own Hook



Building Your Own Hook

What is a custom hook?

A custom hook is a simple JavaScript function that begins with the word ‘use’. This is how React and the toolchain recognize the function as a hook.

```
function useSuperPowers(opts) {

}
```

Building Your Own Hook

```
function useSuperPowers(opts) {
        // State
        // Side Effects
        // Logic

        return {
                // Properties
                // Methods
        }
}
```

Building Your Own Hook

Building hooks is very simple.

They’re functions that begin with the word ‘use’ and can incorporate their own **state**, **logic** and **side effects** which they’re able to share with function components that use them.

thank you!