Better React Components

What are we going to cover

Component Types

- Class based versus functional components
- Higher order components

Component Properties

- Validation

Component State

- Classes versus functions

Lifecycle Functions

- Classes versus functions

Context API

Best Practices

Component Types

Options for authoring components

ECMAScript 2015 classes extending React.Component

- The generic way to create components

Functional Components

- The **preferred** way to create simple components

React.createClass()

- The original way to create components
- Has been deprecated and is removed from React 16
- Available as a separate NPM package create-react-class

ES6 Classes extending React.Component

A common way to write most complex components

- Functional components and hooks are the future

Set the **state** object on the component

The **propTypes** and **defaultProp** are defined as static properties on the component type

No Autobinding

- Use properties pointing to ES6 fat arrow function
- Or the bind function in the component constructor
- Use ES6 fat arrow functions in the render function

Class Component

```
class Person extends React.Component {
  state = { firstName: 'Maurice' };

  setFirstName = e => this.setState({
     firstName: e.target.value
    });

  render() {
    return (
      <input
        value={this.state.firstName}
        onChange={this.setFirstName}
      />
    );
  }
}
```

PureComponent

A **pure component** will always render the same markup with the same props and state

- No side effects

Uses the **shouldComponentUpdate** lifecycle function to prevent rendering the same result

- Can lead to a big performance improvement
- Easy to implement yourself

Does a **shallow** comparison on props and state

- Deep comparison would be costly

Use **immutable** principle's and never change a property on an object

- Including adding/deleting from an array
- Always create a new object instead
- Either use ES6 syntax or a library like Immutable.js

PureComponent

```
class MoviesPresentation extends React.PureComponent {
  // Other code

  render() {
    const { movies } = this.props;
    return (
      <ol>
        {movies.map(movie =>
          <Movie key={movie.id} movie={movie} />)}
      </ol>
    );
  }
}
```

Functional Components

A React **component** as a simple JavaScript function

- They should be pure function and only depend on the properties passed

State managements and **lifecycle** functions using **hooks**

- New in React 16.8

Using **propTypes** and **defaultProps** works as before

- Set them on the component function object

Better runtime **performance** then classes

“This is the recommended pattern, when possible”

- Recommendation from the Facebook team

Pure Function Component

```
const Person = ({firstName, setFirstName}) => (
  <input type="text"
         value={firstName}
         onChange={e =>
           setFirstName(e.target.value)} />);

Person.propTypes = {
  firstName: PropTypes.string.isRequired,
  setFirstName: PropTypes.func.isRequired
};

export default Person;
```

Build focused components

Components should do **one** thing

- Split the UI into many small components
- Use composition to create the complete functionality

Presentational components

- Are concerned with UI
- Only work with props

Container components

- Concerned with state and optional props
- Renders presentational components
  - And optionally other container components

Higher-Order Components

- Add more generic container components to other components

Container versus presentational components

The UI should be build using **presentational** components

- They receive the data to render as props
- Usually no state

Container components contain state management logic

- Do AJAX requests etc
- Render no DOM elements themselves, only presentational components
- Hooks can be used to replace most container components

Presentation Component

```
import React, { Component } from "react";
import PropTypes from "prop-types";

export default class PersonPresentation extends Component {
  static propTypes = {
    firstName: PropTypes.string.isRequired,
    setFirstName: PropTypes.func.isRequired
  };
  setFirstName = e => this.props.setFirstName(e.target.value);

  render() {
    return <input type="text"
                  value={this.props.firstName}
                  onChange={this.setFirstName} />;
  }
}
```

Container Component

```
export default class PersonContainer extends Component {
  state = {
    firstName: "Maurice"
  };

  setFirstName = e => this.setState({ firstName: e });

  render() {
    const { firstName } = this.state;

    return <PersonPresentation firstName={firstName}
                   setFirstName={this.setFirstName} />;
  }
}
```

Higher-Order Components

Higher-order components are **functions** that takes a component as argument and return a new component

- Redux connect is a well known example

Use to handle cross cutting concerns

- Data management
- Error handling
- Logging
- …

Error Boundary as HOC

```
function withErrorBoundary(WrappedComponent) {
  return class extends React.Component {
    state = { error: null };

    static getDerivedStateFromError(error) {
      return { error };
    }
    componentDidCatch(error, info) {
      console.warn('Oops', error, info)
    }

    render() {
      const { error } = this.state;
      if (error) return <div>Error: {error.message}</div>
      return <WrappedComponent { ...this.props} />;
    }
  };
}
```

Render props

Render props is an alternative to higher order components

- Higher order components have some caveats that can be addressed using render props

The **children** property passed in is not a React component but a **function**

- This function is called from the render to create the desired DOM

You can add **multiple** render properties for different use cases

- Render and loading indicator when the AJAX request is busy and real component when the data is loaded

Render prop example

```
class Clock extends React.Component {
  static propTypes = {
    children: PropTypes.func.isRequired
  };
  state = {
    now: new Date().toLocaleTimeString()
  };
  componentDidMount() {
    setInterval(() =>
      this.setState({now: new Date().toLocaleTimeString()}),
      1000);
  }
  render() {
    return <div>{this.props.children(this.state)}</div>;
  }
}
```

Render prop usage

```
<Clock>
  {({ now }) => <div>Time: {now}</div>}
</Clock>
```

Component Properties

Component Properties

The component **props** are passed by the parent component

- Just like parameters in a function

Props passed to a child component can be **any kind** of variables

- Props received from a parent component
- State managed by the component itself
- Constant values
- Some computed value

Props should be considered **immutable**

- Never change props or a child object on them

Components and PropType

Properties are the **input parameters** to React components

- They determine what a component will render

Always **declare** properties that are used in a component

- This can prevent hard to detect error

React can **validate** the proper usage of properties

- This is only done with a development build of React
- Error messages will de shown in browser console window

ESLint with the **react** plugin will detect missing propType declarations

Use **defaultProps** to provide a meaningful default value if not specified

Validating props

```
class Person extends Component {
 static propTypes = {
    person: PropTypes.object.isRequired
  };

  render() {
    // ToDo
  }
}
```

Default props

```
class Person extends Component {
  static defaultProps = {
    person: {
      firstName: "(Unknown)"
    }
  };

  render() {
    // ToDo
  }
}
```

Custom PropType validation

The **PropTypes.object** validation is not all that useful

- Passes when any object is provided, even a completely different shape

The **PropTypes.shape** is better

- Specify the expected properties on an object
- For an array use PropTypes.arrayOf(…)

Use **PropTypes.oneOf()** for enumerations

- Or PropTypes.oneOfType() for unions

The properties on PropTypes are **functions**

- Called to validate if a passed property is valid or not

Create your own **custom validators** to do specific validations

- The default validations are very generic

Validating props

```
class Person extends Component {
  static propTypes = {
    person: PropTypes.shape({
      firstName: PropTypes.string.isRequired,
      lastName: PropTypes.string
    }).isRequired
  };

  render() {
    // ToDo
  }
}
```

Custom Validator

```
function personShape(props, propName, componentName) {
  const person = props[propName];
  if (!person || !person.firstName) {
    return new Error(
      `The prop ${propName} on component
${componentName} is missing a firstName property.`
    );
  }
}

class Person extends Component {
  static propTypes = {
    person: personShape
  };
}
```

Component State

Component State

State is **data** in a component that can change

- Just like local variables in a function

Always use **setState()** to mutate the components state

- Never mutate state directly
- Recommended to use immutable principles and use a new object

Calling setState(), replaceState() or forceUpdate() will force the component to **re-render**

- This is an asynchronous action

The setState() function

Calling setState() is **asynchronous**

- The state is not mutated directly

There are **two overloads** to use setState()

- One takes an object with the new state
- The second takes a function that is passed the current state and returns the new state

The **function** version of setState() is more reliable

- When multiple changes are made and they depend on the current state

Calling setState() **merges** the current state with the passed state

- Calling replaceState() deletes the old state first and then set the new state

Using setState

```
class PersonState extends Component {
  state = {
    firstName: "Maurice"
  };

  setFirstName = e => {
    this.setState({ firstName: e });
  };

  render() {
    const { firstName } = this.state;

    return <Person firstName={firstName}
                   setFirstName={this.setFirstName} />;
  }
}
```

A better setState

A functional approach

```
setFirstName = e => {
   this.setState((oldState, props) => ({
     firstName: e,
     version: oldState.version + 1
   }));
 };
```

State in functional components

Hooks allow for stateful functional components

- Available since React 16.8

The **useState()** hook is very simple and easy to use

- Warning: The setter function replaces the complete state

The **useReducer()** hook is more powerful

- Uses a Redux like reducer to update state based on actions being dispatched

The useState() hook

```
const Counter = () => {
  const [count, setCounter] = React.useState(0);

  return (
    <div>
      <div>Count: {count}</div>
      <div>
        <button onClick={() => setCounter(count + 1)}>
          Increment
        </button>
      </div>
    </div>
  );
};
```

What not to store in state!

Values passed into the component as **props**

Values that are **derived** from input props

Values not used in the **render** function

- Store them as properties on the component

Lifecycle Functions

Class Components Lifecycle

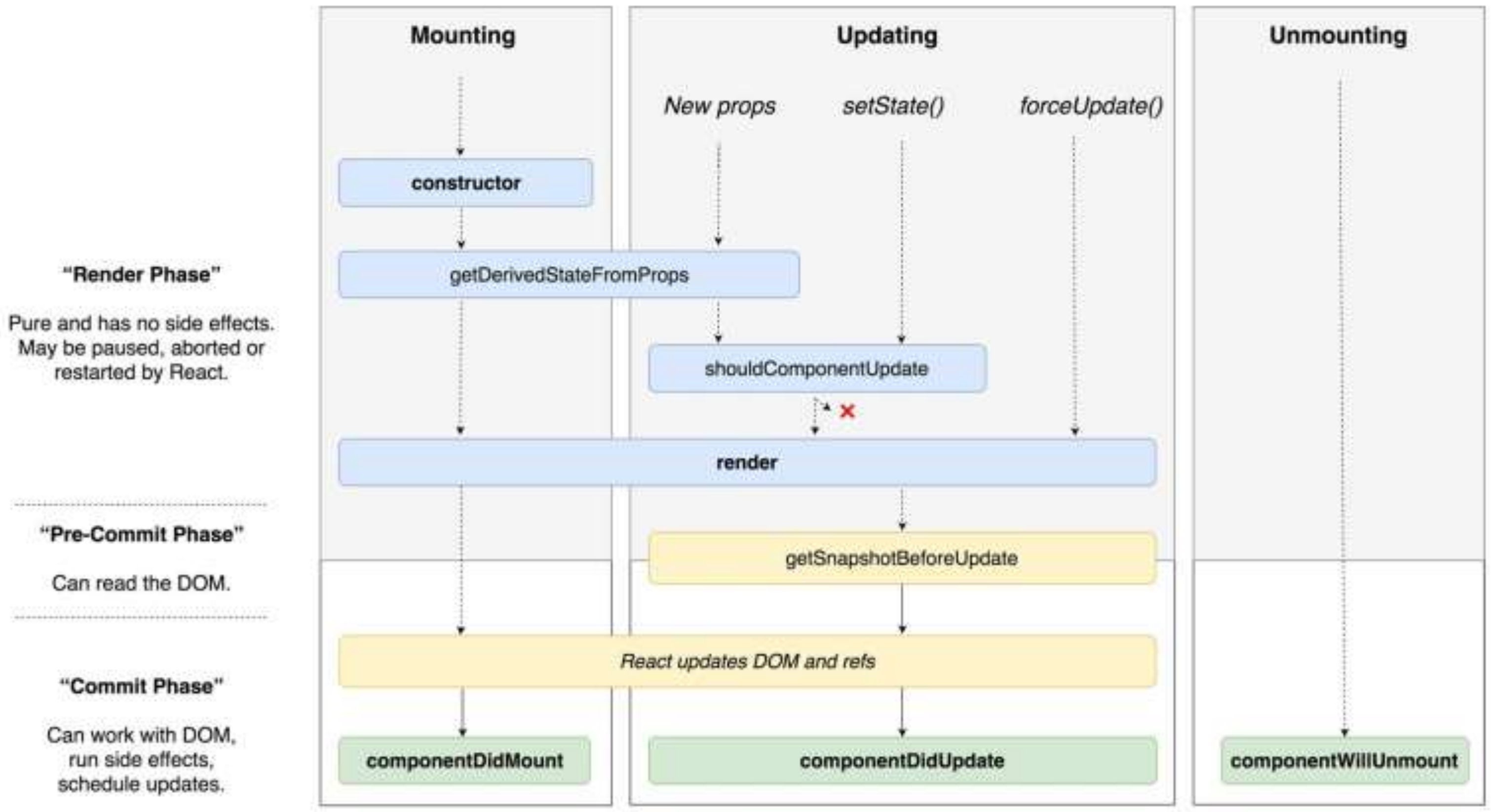
Each React class based component has a number of **lifecycle methods**

- constructor
- getDerivedStateFromProps
- render
- componentDidMount
- componentDidUpdate
- componentDidCatch
- componentWillUnmount
- And more…

The **render()** method is the only one that has to be implemented

- Should return the JSX to be rendered
- Must be a pure function with no side effects

Component lifecycle methods



Functional Component Lifecycle

Functional components use **hooks** for lifecycle management

- useEffect
- useLayoutEffect

As the same suggests it is indented **for side effects**

Optionally return a **cleanup function** from a useEffect hook

The **useEffect** hook sort of combines a number of lifecycle events

- componentDidMount, componentDidUpdate and componentWillUnmount
- How often the effect hook is called depends on the second parameter

A clock with useEffect()

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

const Clock = ({ interval }) => {
  const [now, setNow] = useState(new Date());

  useEffect(() => {
    const handle = setInterval(
        () => setNow(new Date()), interval);
    return () => clearInterval(handle);
  }, [interval]);

  return <div>Time: {now.toLocaleTimeString()}</div>;
};

export default Clock;
```

Context API

React Context

Use **React.createContext()** to create a new context

The **Provider** makes data available

- Can also provide callback functions for updates

The **Consumer** retrieves the data to be used

- Use a child render function

Can replace Redux or similar functionality in order to prevent prop drilling

- Without having to pass props explicitly

The Context

```
import { createContext } from 'react';

const TimeContext = createContext();

export default TimeContext;
```

The Provider

```
import React, { Component } from 'react';
import TimeContext from './TimeContext';

class TimeProvider extends Component {
  state = { now: new Date() };
  componentDidMount() {
    setInterval(() => this.setState({ now: new Date() }), 1000);
  }
  render() {
    const { now } = this.state;
    const { children } = this.props;
    return (<TimeContext.Provider value={now}>
              {children}
            </TimeContext.Provider>);
  }
}

export default TimeProvider;
```

The Consumer

Using render props

```
import React from 'react';
import TimeContext from './TimeContext';

const Clock = () => {
  return (
    <TimeContext.Consumer>
      {now => (<div>
                 {now.toLocaleTimeString()}
               </div>)}
    </TimeContext.Consumer>
  );
};

export default Clock;
```

The Consumer

Using hooks

```
import React, { useContext } from 'react';
import TimeContext from './TimeContext';

const Clock = () => {
  const now = useContext(TimeContext);
  return <div>{now.toLocaleTimeString()}</div>;
};

export default Clock;
```

Best Practices

Build a component tree

The user interface should be constructed using a **tree** of components

There is always one **root** component

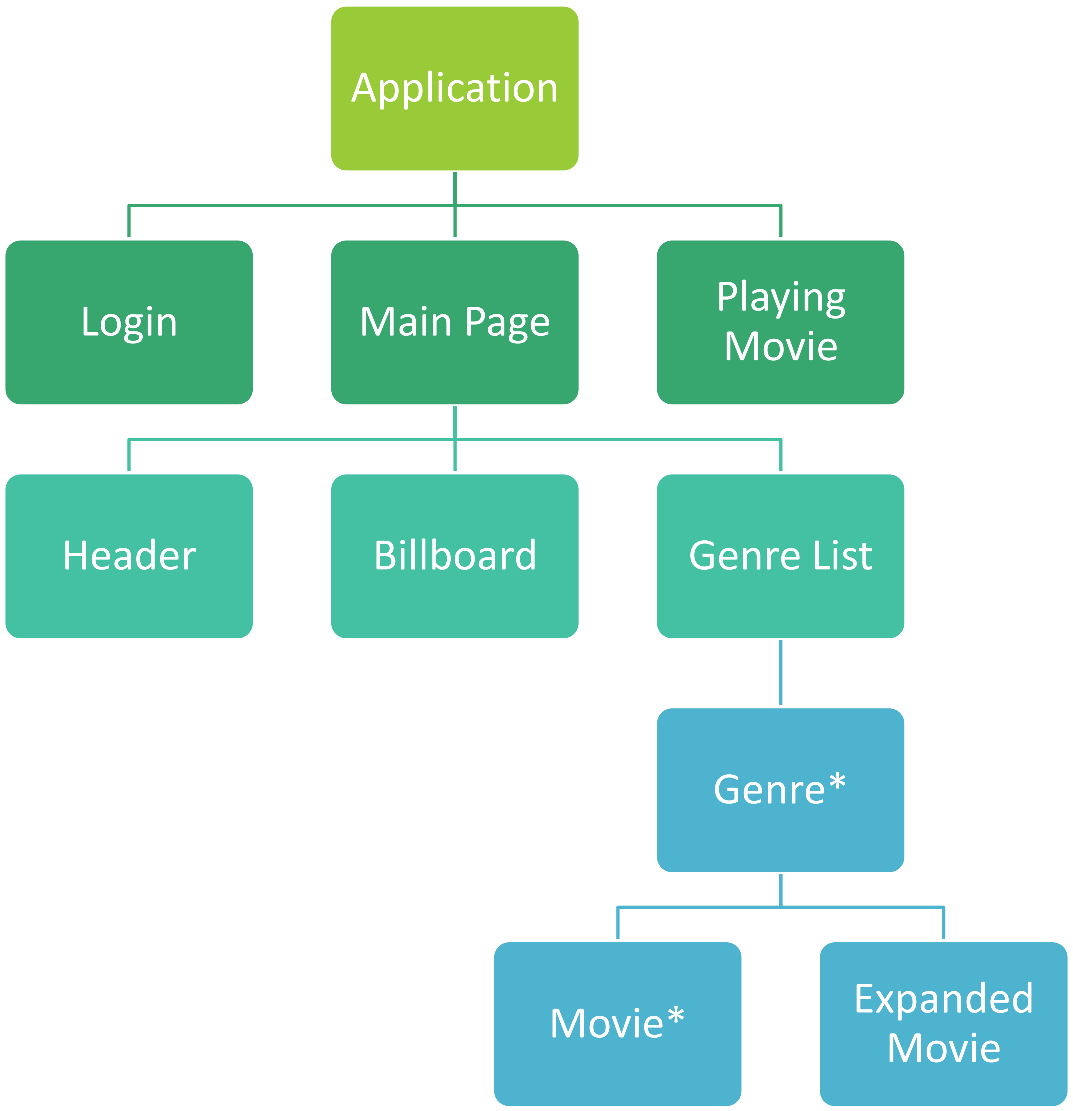
- Each component can have zero or more children

Store **state** as close as possible to the components that need it

- Pass the state as properties to each component

Use **callback functions** as properties to mutate state higher in the tree

Component tree



Best practices - Components

Keep components as **small** as possible

Only use **props** and **state** in the render function

Use **pure functional** components when possible

Use the **Presentational** and **Container** components pattern

Validate props in a component using **prop-types**

- Or use **TypeScript**

Best practices - Performance

Prefer **functional components** over class based components

Use **immutable** objects

- With **React.memo** or **PureComponent** in strategic locations

Best practices - State

Don’t store **props** or **derived data** in component state

Only store data in state that is needed for **rendering**

Use the **functional** version of setState()

Don’t use **Redux** or **MobX** if you don’t need them

Conclusion

A React application is a **tree of components**

- Store state at the appropriate level

Functional components should be the default choice

- Combined with React hooks

Declare the expected properties for each component

- React will validate them and warn you about mismatches

Data that changes and is used when rendering goes into **component state**

Use **lifecycle functions** for side effects

Context can be used to prevent prop drilling