# Better React Components

# What are we going to cover

The component tree

Different options for authoring components

- Extending React.Component
- Stateless Functional Components

State and props

- PropType validation

Composition versus Mixins

- PureComponent

Context

# Build a component tree

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

There is always one **root** component

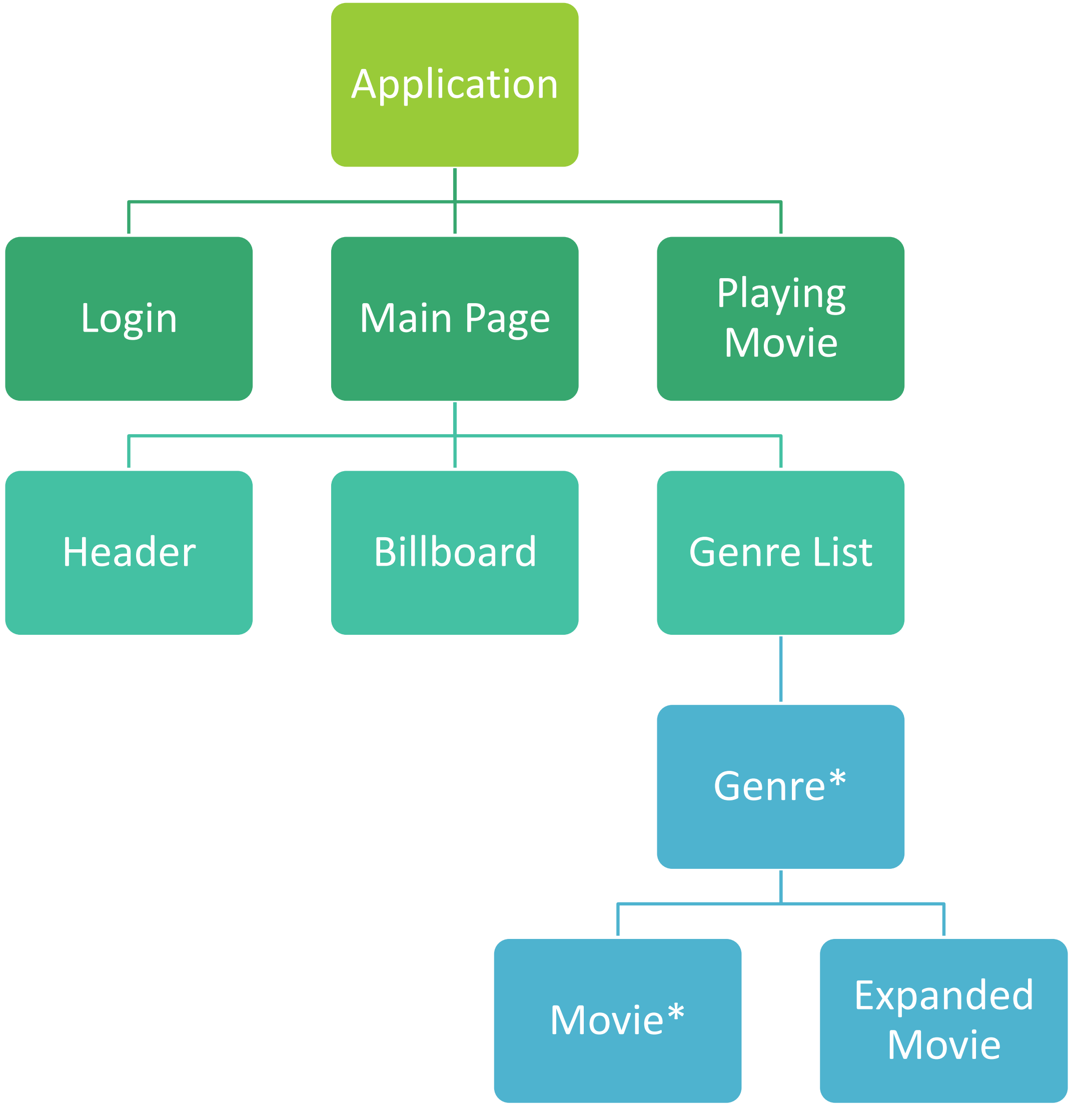
- Each component can have zero or more children

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



# Options for authoring components

ECMAScript 2015 classes extending React.Component

- The new generic way to create components

Stateless 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

# ES6 Classes extending React.Component

The recommended way to write most complex components

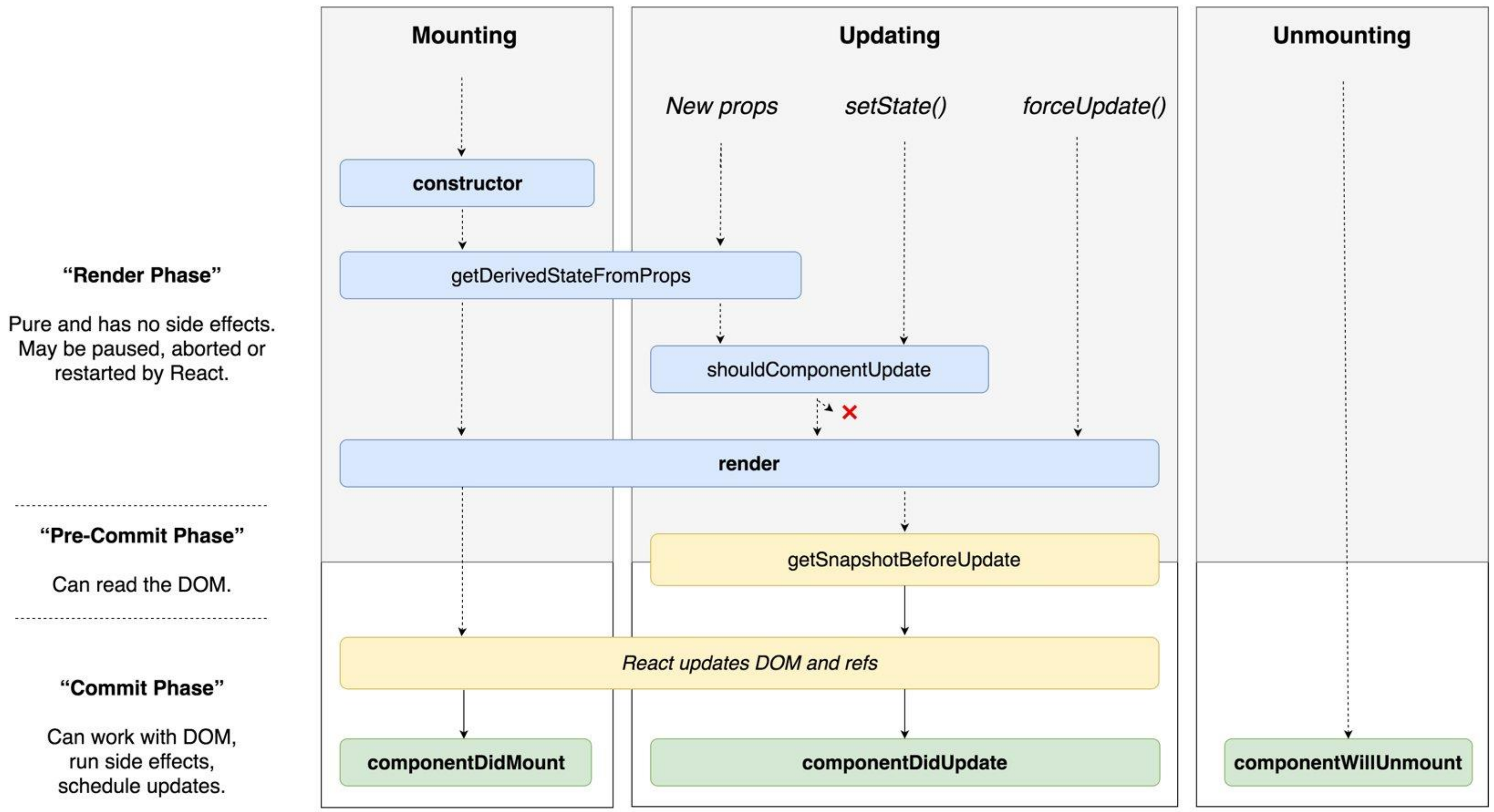
Set the **state** object on the component

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

No Autobinding

- Use properties pointing to ES6 fat arrow function
  - The recommended approach
- Use the bind function in the component constructor
  - Also a good approach
- Use ES6 fat arrow functions in the render function
  - Be careful with repeatedly creating a new function with each render

# Component lifecycle methods



Mounting

Updating

Unmounting

New props

setState()

forceUpdate()

constructor

“Render Phase”

Pure and has no side effects.
May be paused, aborted or
restarted by React.

getDerivedStateFromProps

shouldComponentUpdate

render

“Pre-Commit Phase”

Can read the DOM.

getSnapshotBeforeUpdate

“Commit Phase”

Can work with DOM,
run side effects,
schedule updates.

*React updates DOM and refs*

componentDidMount

componentDidUpdate

componentWillUnmount

# 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

# 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() {
    const { firstName } = this.props;

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

# Container Component

```
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} />;
  }
}
```

# Stateless Functional Components

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

- Useful for simple components
- They should be pure function and only depend on the properties passed

No **state** managements or **lifecycle** functions

- Extend React.Component when you need these

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

- Set them on the component function object

Will possibly contain **performance** enhancements in the future

- With React version 15 just a simple wrapper around extending React.Component

**“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;
```

# 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 {
    componentDidCatch(error, info) {
      console.warn('Oops', error, info)
    }

    render() {
      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>
```

# 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

# 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

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 ways 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
   }));
 };
```

# 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

# Repeating elements

**Repeating** elements are very common

- UL => LI
- TBODY => TR
- etc

Repeated elements should always have a unique **key** property

- React will warn if duplicate keys are used
- With duplicate keys only the first element is shown

Always use **unique** object properties to determine the key value

- Do not use an array index
- React uses the key to associate a DOM element with its data

# A list of movies

```
const Movie = ({ movie }) =>
  <li>{movie.title}</li>;

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

# 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 PureComponent {
  // Other code

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

# React Context (before React 16.3)

The **Context** provides a way of passing objects transparently to client components

- Unlike props that need to be explicitly passed at each level

This context is available in each **child** component

- Makes it a bit magical

Use the static **contextTypes** property to declare the required context items

- Just like a components propTypes

Use the static **childContextTypes** to declare provided context items

- And the getChildContext() function to provide the actual context object

**“If you have to use context, use it sparingly”**

- Recommendation from the Facebook team

# React Context (React 16.3)

React 16.3 will receive a completely **new Context API**

- The previous API will be deprecated

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

- 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

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

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

export default Clock;
```

# 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 **Presentational** and **Container** components

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

# Best practices - Performance

Use **immutable** objects and **PureComponent** in strategic locations

Don’t define **fat arrow functions** in the render

# 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 on components

- Store state at the appropriate level

Extending React.Component is great for general purpose components

- But consider Stateless Functional Components whenever possible

Declare the expected properties for each component

- React will validate them and warn you about mismatched

Prefer component composition over mixins or inheritance

Try to avoid React context before React 16.3

- It’s powerful but can obfuscate things