React Introduction

What are we going to cover

What is React

Getting started with building React applications

Transpiling JSX code to ECMAScript

React components and their lifecycle

Working with properties and state in React components

Working with DOM events

Other things you will need

React

A JavaScript library for building **user interfaces**

- Not a full blown framework like AngularJS

Build at **Facebook** and Instagram

- Open source and maintained on GitHub
- React is MIT licensed

Work against a **virtual DOM**

- No differences between browsers
- Can also be rendered on the server when needed

Uses a **unidirectional data flow**

- No two way data binding

React tradeoffs

ADVANTAGES

Large community

Small runtime library

Fast

Stable

Simple and predictable API

Supports server side rendering

Open source

Dedicated team from Facebook

DISADVANTAGES

JavaScript focused

Markup inside JavaScript

Increases memory pressure

Driven by Facebooks needs

Not very opinionated

A minimal React Application

Using **JSX syntax**

- Is transpiled to ECMAScript 5

Define a new React **Component**

- By extending React.Component

Use by including it as markup

- In the **render()** function

Minimal Example

```
import React, { Component } from "react";
import ReactDOM from "react-dom";

class App extends Component {
  render() {
    return (<div>
      <h2>ReactJS</h2>
      <OtherComponent />
    </div>);
  }
}

ReactDOM.render(<App />,
  document.getElementById("root"));
```

The markup

There is a **JavaScript bundle reference**

- /static/js/bundle.js

Also contains React and React-DOM

- Build using **Webpack**

Minimal Markup Example

```
<!DOCTYPE html>
<html>

<head>
  <title>ReactJS</title>
  <base href="/" />
</head>

<body>
  <div id="root" />
  <script src="./index-bundle.js"></script>
</body>

</html>
```

Transpiling React

React is normally written as **JSX code**

- A JavaScript extension
- Can be written in standard JavaScript if preferred

JSX needs to be **transpiled** to standard JavaScript

- Can be done at runtime but this is not advised

Babel is the preferred way to do this

- Usually done using Babel and Webpack
- Also supports most ECMA Script 2015 and newer features

React can be written in pure JavaScript

Create the React elements directly

Not very commonly done

- Hard with more complex components

A pure JavaScript component

```
var app = React.createElement("div", null,
  React.createElement("h2", null, "ReactJS"),
  React.createElement("span", null,
    "Hello from React without JSX")
);

ReactDOM.render(app,
  document.getElementById("app"));
```

The markup

Needs to load

- React.js
- React-dom.js

The pure JavaScript markup

```
<!DOCTYPE html>
<html>
<head>
    <title>ReactJS</title>
    <link href="/Libs/Bootstrap/dist/css/bootstrap.css" rel="stylesheet" />
    <base href="/" />
</head>
<body>
    <div id="app" class="container" />
    <script src="//unpkg.com/react@latest/dist/react.js"></script>
    <script src="//unpkg.com/react-dom@latest/dist/react-dom.js"></script>
    <script src="./pure.js"></script>
</body>
</html>
```

React Components

A React application consists of one ore more **components**

Components are the **view** with embedded view logic

- Results in HTML elements with associated behavior

Each component can contain other **nested components** if needed

- Split large components into smaller subcomponents
- Use the single responsibility pattern

React Components Lifecycle

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

- componentWillMount
- render
- componentDidMount
- componentDidCatch
- componentWillUnmount
- And more…

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

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

Functional Components

React encourages **small functional functions**

- Many of your components should be this simple

Support for lifecycle functions and state using **hooks**

Use **composition** to build the user interface

The **preferred approach** for the future of React

Functional Component Example

```
var OtherComponent = function() {
  return (<span>Hello from React</span>);
};

var App = function() {
  return (<div>
      <h2>ReactJS</h2>
      <OtherComponent />
    </div>);
};
```

React Components Properties

Properties are used to pass data to client components

- Should be treated as read only

Special case for style and class

Typically used in the **render()** function

- this.props.<name>
- Functional components receive a **props** parameter

Component Properties Example

```
class Greeter extends React.Component {
  render() {
    return (
      <span>
        Hello {this.props.firstName}
        {this.props.lastName}
      </span>
    );
  }
}

ReactDOM.render(
  <Greeter firstName="Maurice" lastName="de Beijer" />,
  document.getElementById("app")
);
```

React Components State

React components often act as **state machines**

- State is internal to a component
- But can be passed on as a property

When **state is updated**

- The component render() is called
- React updates the DOM if different

Functional components can use **hooks**

- The useState() hook is simple
- The useReducer() hook for more complex cases

Component State Example

```
class Clock extends React.Component {
  state = {
    time: ""
  };
  componentDidMount() {
    setInterval(() => this.setState({
        time: new Date().toLocaleTimeString()
    }), 1000);
  }
  render() {
    return (
      <span>Time: {this.state.time}</span>
    );
  }
}
```

React Components Event Handling

React normalizes DOM events

- Easier with cross browser differences

Passed a **SyntheticEvent**

Event Handling Example

```
class Greeter extends React.Component {
  greet = () => {
    alert("Hello " + this.props.firstName);
  }
  render() {
    return (
      <span>
        <button onClick={this.greet}>
          Greet
        </button>
      </span>
    );
  }
};
```

Working with real DOM elements

React uses a **shadow DOM**

- Only the differences are really rendered to the DOM

The shadow DOM exposes a set of normalized events

- Not all browser events are exposed

You need to access the real DOM elements

- To get an input elements value
- To add a native DOM event handler

Use **refs** to reach child components

DOM Elements Example

```
class UserForm extends React.Component {
  firstName = null;
  getValue() {
    return this.firstName.value;
  }
  render() {
    return (
      <form>
        <input type="text"
          ref={el => (this.firstName = el)} />
      </form>
    );
  }
}
```

Repeating React Components

Components can be **repeated**

- But every row requires a **key** property

Repeating Components Example

```
class PeopleTable extends React.Component {
  render() {
    const { people } = this.props;

    return (<table>
        <tbody>
          {people.map((person, index) => (
            <PersonRow key={index} person={person} />
          ))}
        </tbody>
      </table>);
  }
}
```

There is more

React is just a **UI library**

- You will need more in complete applications

Use **Flux** as architectural guidance

- Good implementations like Redux

Create **Single Page Applications**

- React-Router-Dom is the defacto standard library to use

Most React applications will need to do **AJAX** calls

- Use the proposed fetch API
- Or alternatives like SuperAgent, Axios or jQuery

Conclusion

ReactJS is a great framework for **UI components**

- Leaves other application concerns for other libraries

Code is typically **written using JSX**

- Use Babel with WebPack to transpile to ECMAScript

Use the appropriate **API** to work with **data**

- Use component state to hold mutating data
- Pass data as props to child components

Use **other libraries** as appropriate

- React isn't a complete framework
- The Flux architecture works well with React