ReactJS

A library for creating user interface

Introduction

- React is front end library developed by Facebook
- It's used for handling view layer for web and mobile apps.
- ReactJS allows us to create reusable UI components.
- It is currently one of the most popular JavaScript libraries

Why Should YOU Use React?

- Can be used for parts of your application
- Plays well with other libraries and technologies (meteor, rails, node)
- Components allow you to split work easily

It's fast:

- Because the DOM is slow!
- Computes minimal DOM operations
- Batched reads and writes for optimal DOM performance
- Usually faster than manual DOM operations
- Automatic top-level event delegation (with cross-browser HTML5 events)

React Features:

- **JSX** − JSX is JavaScript syntax extension. It isn't necessary to use JSX in React development, but it is recommended.
- **Components** − React is all about components. You need to think of everything as a component. This will help you to maintain the code when working on larger scale projects.
- **Unidirectional data flow and Flux** − React implements one way data flow which makes it easy to reason about your app. Flux is a pattern that helps keeping your data unidirectional.
- **Free and Open Source** − If a software is free and it is open source, there is needless to say that it is going to be the new favorite of programmers and relevant community.

- The Smallest React example looks like this:

```
ReactDOM.render(
            <h1>Hello, World!</h1>
document.getElementById('root')
 );
```

Output:

Hello, world!

Install and Run the React App Creator

- Install the React app creator (on-time global install):

```
npm -g install create-react-app
```

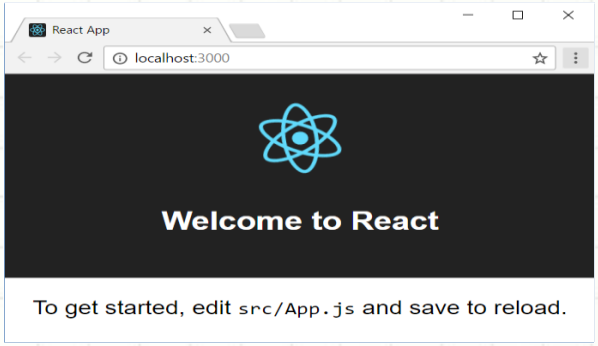
- Run the React app creator (this is very slow: ~5-10 minutes!)

```
create-react-app react-example
```

- Starts your React app from the command line

```
npm start
```

- Browse you app from http://localhost:3000



Introducing JSX:

- Optional HTML-like markup
- Declarative Description od the UI inlined in JS code
- Combines ease-of-use of templates with power of JavaScript
- Preprocessor translates to plain JavaScript:
    - On the fly (suitable during development)
    - Offline using React CLI

Introducing JSX:

- Consider this variable declaration:

  const element = <h1>Hello, world!</h1>;

- This funny tag syntax is neither a string nor HTML.It is called JSX, and it is a syntax extension to JavaScript.

```
var Header = React.createClass({
      render: function () {
         return (
            <h1>Employee Directory</h1>
         );
      }
});
```

Embedding Expressions in JSX

- You can embed any JavaScript extension in JSX by wrapping it in curly braces.
- For example, 2 + 2, user.firstName, and formatName(user) are all valid expressions:

```
function formatName(user) {
        return user.firstName + ' ' + user.lastName;
}
const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};
const element = (
  <h1>
Hello, {formatName(user)}!
</h1>
);
ReactDOM.render(
  element,
        document.getElementById('root')
);
```

Output:
Hello, Harper Perez!

JSX is an Expression Too

- After compilation, JSX expressions become regular JavaScript objects.
- This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Specifying Attributes with JSX

- You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>;
```

- You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

- Don't put quotes around curly braces when embedding a JavaScript expression in an attribute. Otherwise JSX will treat the attribute as a string literal rather than an expression. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Specifying Children with JSX

- If a tag is empty, you may close it immediately with />, like XML:

```
const element = <img src={user.avatarUrl} />;
```

- JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

- Caveat:

Since JSX is closer to JavaScript than HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.
For example, class becomes *className* in JSX, and tabindex becomes *tabIndex.*

JSX Prevents Injection Attacks

- It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

- By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX Represents Objects

- Babel compiles JSX down to React.createElement() calls.
- These two examples are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

- React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object like this:

JSX Represents Objects-2

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world'
  }
};
```

- These objects are called "React elements". You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

Virtual DOM

Re-render everything on every update:

- Create lightweight description of component UI
- Compute minimal set of changes to apply to the DOM
- Difference with the old one
- Batch executes all updates

Rendering an Element into the DOM

- Let's say there is a somewhere in your HTML file

```
<div id="root"></div>
```

- We call this a "root" DOM node because everything inside it will be managed by React DOM.
- Applications built with just React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.
- To render a React element into a root DOM node, pass both to ReactDOM.render():

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(
  element,
  document.getElementById('root')
);};
```

Output:
Hello, world

Updating the Rendered Element

- React elements are immutable,once you create an element, you can't change its children or attributes
- The only way to update the UI is to create a new element, and pass it to **ReactDOM.render().**

Example:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}
setInterval(tick, 1000);
```

Output:
Hello, world!
It is 4:26:46 PM

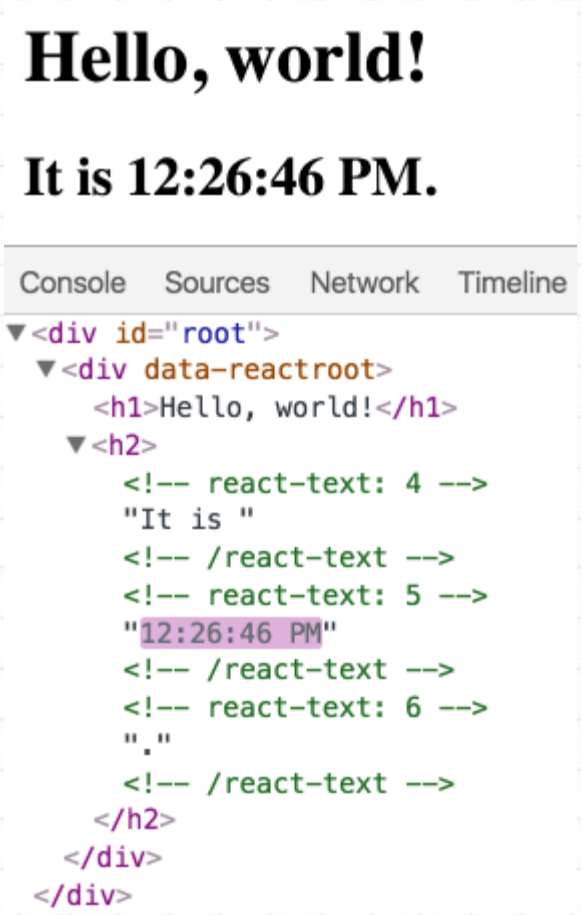
- It calls ReactDOM.render() every second from a setInterval() callback.

React Only Updates What's Necessary

- React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state

Example:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new
Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}
setInterval(tick, 1000);
```



Build components, not templates

- UI Components are the cohesive units.
- UI description and UI logic are tightly coupled and can be collocated.
    - They shouldn't be arbitrarily decoupled
- Full power of JavaScript to express UI.

Creating the UI DescriptionCreating the UI Description:

```
var child1 = React.createElement('li', null, 'First Text Content');
var child2 = React.createElement('li', null, 'Second Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child1,child2);
React.render(root, document.getElementById('example'));
```

Components and Props

- Components let you split the UI into independent, reusable pieces, and think and think about each piece in isolation.
- Components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Functional and Class Components

- The simplest way to define a component is to write a JavaScript function:

```
function Welcome(props) {
return <h1>Hello, {props.name}</h1>;
 }
```

- This function is a valid React component because it accepts a single "props" object argument with data and returns a React element. We call such components "functional" because they are literally JavaScript functions.
- You can also use an ES6 class to define a component:

```
class Welcome extends React.Component {
render() {
return <h1>Hello, {this.props.name}</h1>;
   }
}
```

- The above two components are equivalent from React's point of view.

Rendering a Component

- Previously, we only encountered React elements that represent DOM tags:

```
const element = <div />;
```

- However, elements can also represent user-defined components:

```
const element = <Welcome name="Sara" />;
```

- When React sees an element representing a user-defined component, it passes JSX attributes to this component as a single object. We call this object "props".

- For example, this code renders "Hello, Sara" on the page:

```
function Welcome(props) {
return <h1>Hello, {props.name}</h1>;
 }
const element = <Welcome name="Sara" />;
ReactDOM.render(
element,
document.getElementById('root')
);
```

Output:
Hello, Sara

Composing Components

- Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail.
- A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.
- For example, we can create an App component that renders Welcome component that renders.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div> );}
ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

Output:
Hello, Sara
Hello, Cahal
Hello, Edite

Extracting Components

- Don't be afraid to split components into smaller components.
  For example, consider this comment component:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>  );}
```

Output:

Hello Kitty
I hope you enjoy learning React!
5/19/2017

Props are Read-Only

- Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function:

```
function sum(a,b){
return a+b; }
```

- Such functions are called ”pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.
- In contrast, this function is impure because it changes its own input:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

- React is pretty flexible but it has a single strict rule:
- All React components must act like pure functions with respect to their props.

Of course, application UIs are dynamic and change over time.

Thank you