

# Templates and Databinding

SWE 432, Fall 2016

Design and Implementation of Software for the Web

# Today

- What are templates?
- What are frontend components?
- How can I use these with React?

For further reading:

<https://facebook.github.io/react/docs/tutorial.html>

React Guide: <https://facebook.github.io/react/docs/why-react.html>

Starter code snippets: <https://facebook.github.io/react/downloads/react-15.3.2.zip>

# What's wrong with this code?

```
function createItem(value, key)
{
    $('#todoItems').append(
        '<div class="todoItem" data-index=' + key
        + '"><input type="text" onchange="itemChanged(this)" value='
        + value + '"><button onclick="deleteItem(this.parentElement)">x</button></div>'
    );
}
```

# What's wrong with this code?

```
function createItem(value, key)
{
    $('#todoItems').append(
        '<div class="todoItem" data-index=' + key
        + '><input type="text" onchange="itemChanged(this)" value="'
        + value + '><button onclick="deleteItem(this.parentElement)">✖</button></div>'
    );
}
```

No syntax checking

# Anatomy of a Non-Trivial Web App

The screenshot shows the Twitter homepage with various widgets highlighted:

- User profile widget:** Points to the sidebar on the left showing Thomas LaToza's profile with 61 tweets, 229 following, and 163 followers.
- Who to follow widget:** Points to the "Who to follow" section on the right, which lists users like Grace Lewis, Gregorio Robles, and Loren Terveen with "Follow" buttons.
- Follow widget:** Points to the "Find friends" button in the "Who to follow" section.
- Feed widget:** Points to the main timeline feed on the right, which displays tweets from "Talks at Google", "Toyota USA", and "davidcshepherd".
- Feed item widget:** Points to the individual tweet items in the feed, such as the one from "davidcshepherd" about Software Engineering research.

# Typical properties of web app UIs

- Each widget has both visual presentation & logic
  - e.g., clicking on follow button executes some logic related to the containing widget
  - Logic and presentation of individual widget strongly related, loosely related to other widgets
- Some widgets occur more than once
  - e.g., Follow widget occurs multiple times in Who to Follow Widget
    - Need to generate a copy of widget based on data
  - Changes to data should cause changes to widget
    - e.g., following person should update UI to show that the person is followed. Should work even if person becomes followed through other UI
- Widgets are hierarchical, with parent and child
  - Seen this already with container elements in HTML...

# Idea 1: Templates

```
$('#todoItems').append(  
  '<div class="todoItem" data-index="' + key  
  + '"><input type="text" onchange="itemChanged(this)" value="'  
+ value + '"><button onclick="deleteItem(this.parentElement)">&#x2716;</button></div>'  
);
```

- Templates describe repeated HTML through a single common representation
  - May have variables that describe variations in the template
  - May have logic that describes what values are used or when to instantiate template
  - Template may be instantiated by binding variables to values, creating HTML that can be used to update DOM

# Server side vs. client side

- Where should template be instantiated?
- *Server-side* frameworks: Template instantiated on server
  - Examples: JSP, ColdFusion, PHP, ASP.NET
  - Logic executes on server, generating HTML that is served to browser
- *Front-end* framework: Template runs in web browser
  - Examples: React, Angular, Meteor, Ember, Aurelia, ...
  - Server passes template to browser, browser generates HTML on demand

```
<html>
<head><title>First JSP</title></head>
<body>
<%
    double num = Math.random();
    if (num > 0.95) {
%
        <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
<%
    } else {
%
        <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
<%
    }
%
>
```

```
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

# Server side vs. client side

- Server side
  - Oldest solution.
  - True when “real” code ran on server, Javascript
- Client side
  - Enables presentation logic to exist entirely in browser
  - e.g., can make call to remote web service, no need for server to be involved
  - (What we’ve looked at in this course).

# Logic

- Templates require combining logic with HTML
  - Conditionals - only display presentation if some expression is true
  - Loops - repeat this template once for every item in collection
- How should this be expressed?
  - Embed code in HTML (ColdFusion, JSP, Angular)
  - Embed HTML in code (React)

# Embed code in HTML

```
<cfcomponent name = "PersonalChef">
  <cffunction name = "makeToast" returnType = "component">
    <cfargument name = "color" required="yes">

    <cfset this.makeToast = "Making your toast #arguments.color#" />
    <cfreturn this />
  </cffunction>
</cfcomponent>
```

```
<html>
<head><title>First JSP</title></head>
<body>
  <%
    double num = Math.random();
    if (num > 0.95) {
  %>
    <h2>You'll have a luck day!</h2><p>(<%= num %>)</p>
  <%
    } else {
  %>
    <h2>Well, life goes on ... </h2><p>(<%= num %>)</p>
  <%
    }
  %>
```

- Template takes the form of an HTML file, with extensions
  - Custom tags (e.g., `<% %>`) enable logic to be embedded in HTML
  - Uses another language (e.g., Java, C) or custom language to express logic
  - Next lecture: **Angular**

# Embed HTML in code

```
function createItem(value, key)
{
  $('#todoItems').append(
    '<div class="todoItem" data-index=' + key
    + '"><input type="text" onchange="itemChanged(this)" value='
    + value + '"><button onclick="deleteItem(this.parentElement)">✖</button></div>'
  );
}
```

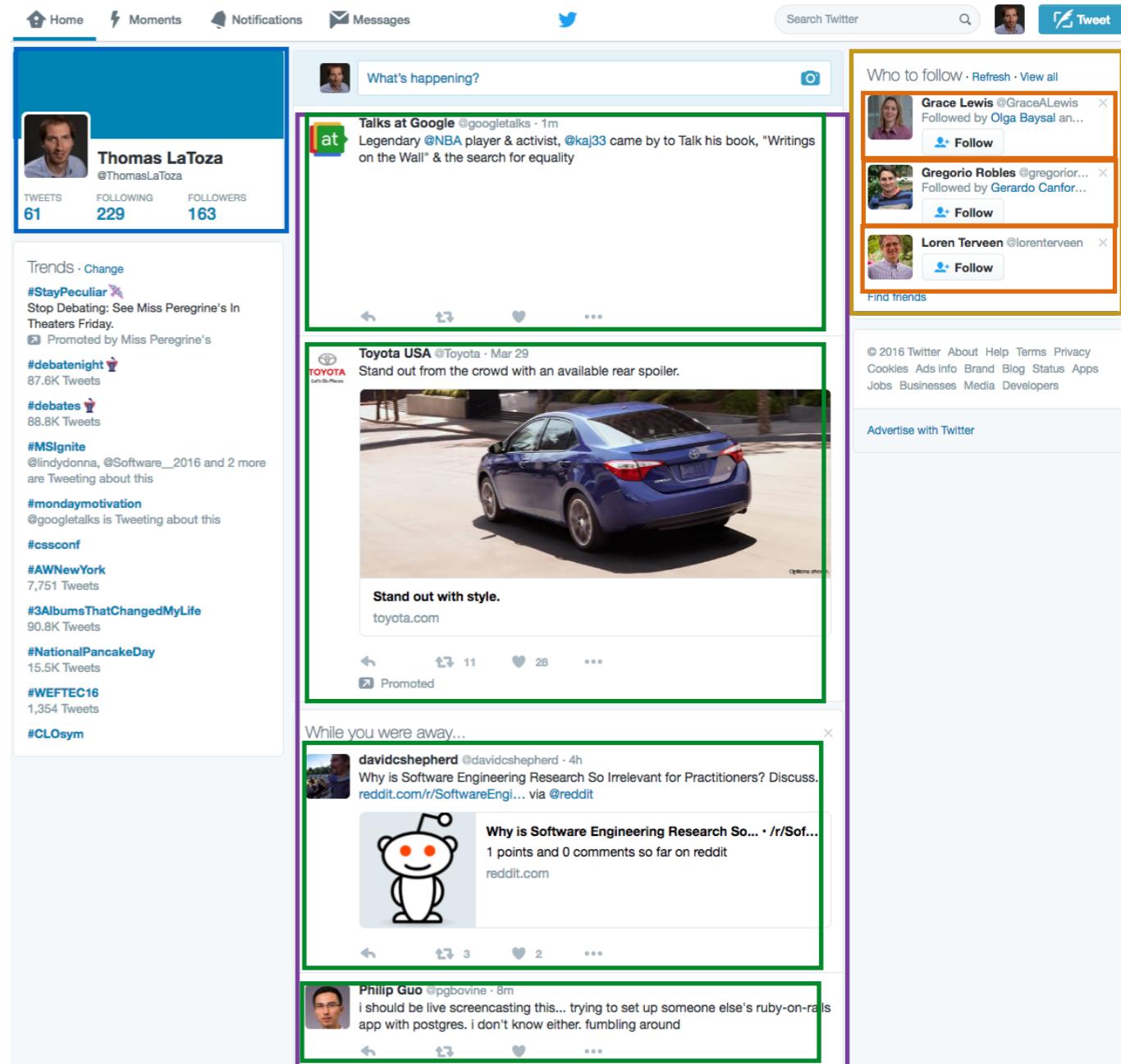
- Template takes the form of an HTML fragment, embedded in a code file
  - HTML instantiated as part of an expression, becomes a value that can be stored to variables
  - Uses another language (e.g., Javascript) to express logic
  - This lecture: **React**

# Templates enable HTML to be rendered multiple times

- Rendering takes a template, instantiates the template, outputs HTML
- Logic determines which part(s) of templates are rendered
- Expressions are evaluated to instantiate values
  - e.g., { this.props.name }
  - Different variable values ==> different HTML output

# Idea 2: Components

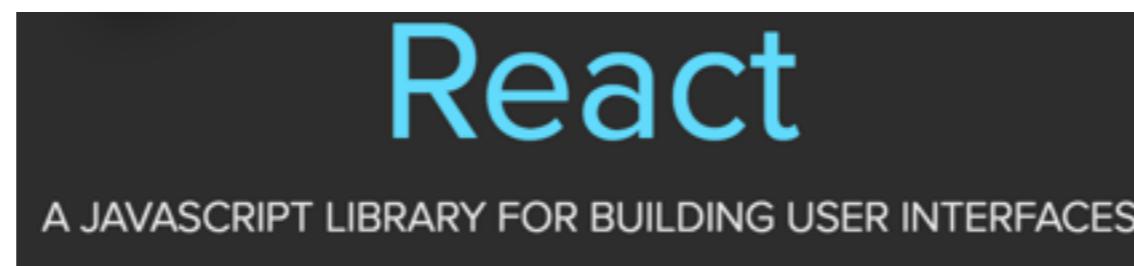
- Web pages are complex, with lots of logic and presentation
- How can we organize web page to maximize modularity?
- Solution: Components
  - Templates that correspond to a specific widget
  - Encapsulates related logic & presentation using language construct (e.g., class)



# Components

- Organize related logic and presentation into a single unit
  - Includes necessary *state* and the logic for updating this state
  - Includes presentation for *rendering* this state into HTML
  - Outside world *must* interact with state through accessors, enabling access to be controlled
- Synchronizes state and visual presentation
  - Whenever state changes, HTML should be rendered again
- Components instantiated through custom HTML tag

# React: Front End Framework for Components



- Originally build by Facebook
- Opensource frontend framework
- React has a TON of features...
  - We're only going to be scratching the surface
- Official documentation & tutorials
  - <https://facebook.github.io/react/index.html>

# Example

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

“Create a HelloMessage component”

Creates a new component with the provided functions.

```
var HelloMessage = React.createClass({  
  render: function() {  
    return <div>Hello {this.props.name}</div>;  
  }  
});  
  
ReactDOM.render(<HelloMessage name="John" />,  
  document.getElementById('mountNode'));
```

“Return a div with a name as the presentation”

Render generates the HTML for the component. The HTML is dynamically generated by the library.

“Replace mountNode with rendered HelloMessage”

Instantiates component, replaces mountNode innerHTML with rendered HTML. Second parameter should always be a DOM element.

# Example - Properties

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

```
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

ReactDOM.render(<HelloMessage name="John" />,
  document.getElementById('mountNode'));
```

“Create a HelloMessage component”

Corresponding custom tag for component.

“Set the name property of HelloMessage to John”

Components have a this.props collection that contains a set of properties instantiated for each component.

# Embedding HTML in Javascript

```
return <div>Hello {this.props.name}</div>;
```

- HTML embedded in JavaScript
  - HTML can be used as an expression
  - HTML is checked for correct syntax
- Can use { expr } to evaluate an expression and return a value
  - e.g., { 5 + 2 }, { foo() }
- Output of expression is HTML

# JSX

- How do you embed HTML in JavaScript and get syntax checking??
- Idea: extend the language: JSX
  - Javascript language, with additional feature that expressions may be HTML
  - Can be used with ES6 or traditional JS (ES5)
- It's a new language
  - Browsers *do not* natively run JSX
  - If you include a JSX file as source, you will get an error

# Transpiling

- Need to take JSX code and trans-compile (“transpile”) to Javascript code
  - Take code in one language, output *source* code in a second language
- Where to transpile?
  - Serverside, as part of build process
    - Fastest, least work for client. Only have to execute transpiling once.
  - Clientside, through library.
    - Include library that takes JSX, outputs JS.
    - Easy. Just need to include a script.

# Babel

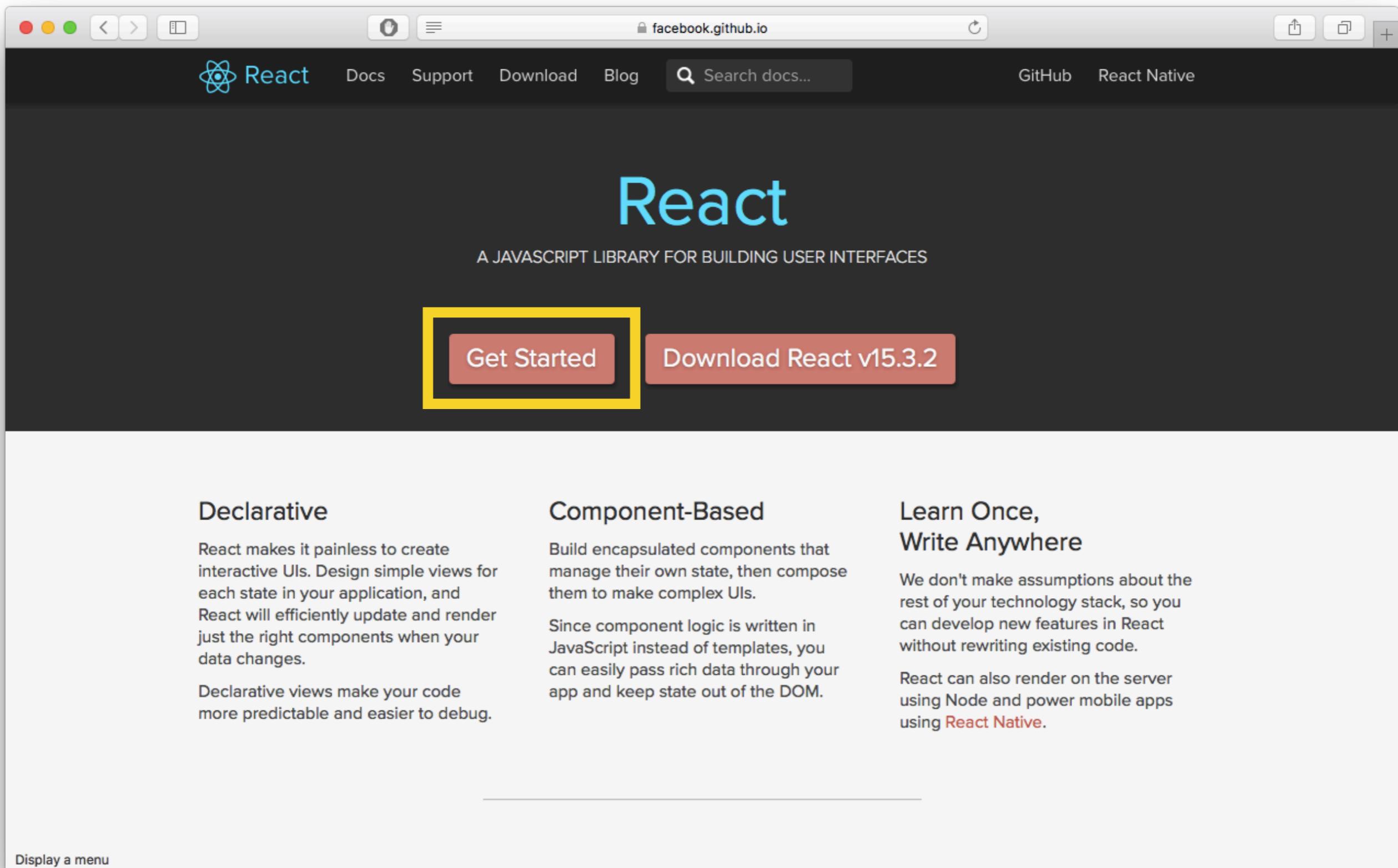
```
<script src="https://cdnjs.com/libraries/babel-core/  
5.8.34"> </script>
```

```
<script type="text/babel">  
//JSX here  
</script>
```

## Babel client side

- Transpiler for Javascript
- Takes JSX (or ES6) and outputs traditional Javascript (a.k.a ES5)
- Can use server side or client side
- Using Babel serverside: <https://facebook.github.io/react/docs/language-tooling.html>

<https://babeljs.io/>

A screenshot of a Mac OS X desktop showing a web browser window. The window title is 'facebook.github.io'. The page content is the official React website. The header includes the React logo, navigation links for 'Docs', 'Support', 'Download', and 'Blog', a search bar, and links to 'GitHub' and 'React Native'. The main title 'React' is in large blue text, followed by the subtitle 'A JAVASCRIPT LIBRARY FOR BUILDING USER INTERFACES'. Below the subtitle are two buttons: 'Get Started' (highlighted with a yellow border) and 'Download React v15.3.2'. The main content area is divided into three columns: 'Declarative', 'Component-Based', and 'Learn Once, Write Anywhere'.

**Declarative**

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

**Component-Based**

Build encapsulated components that manage their own state, then compose them to make complex UIs.

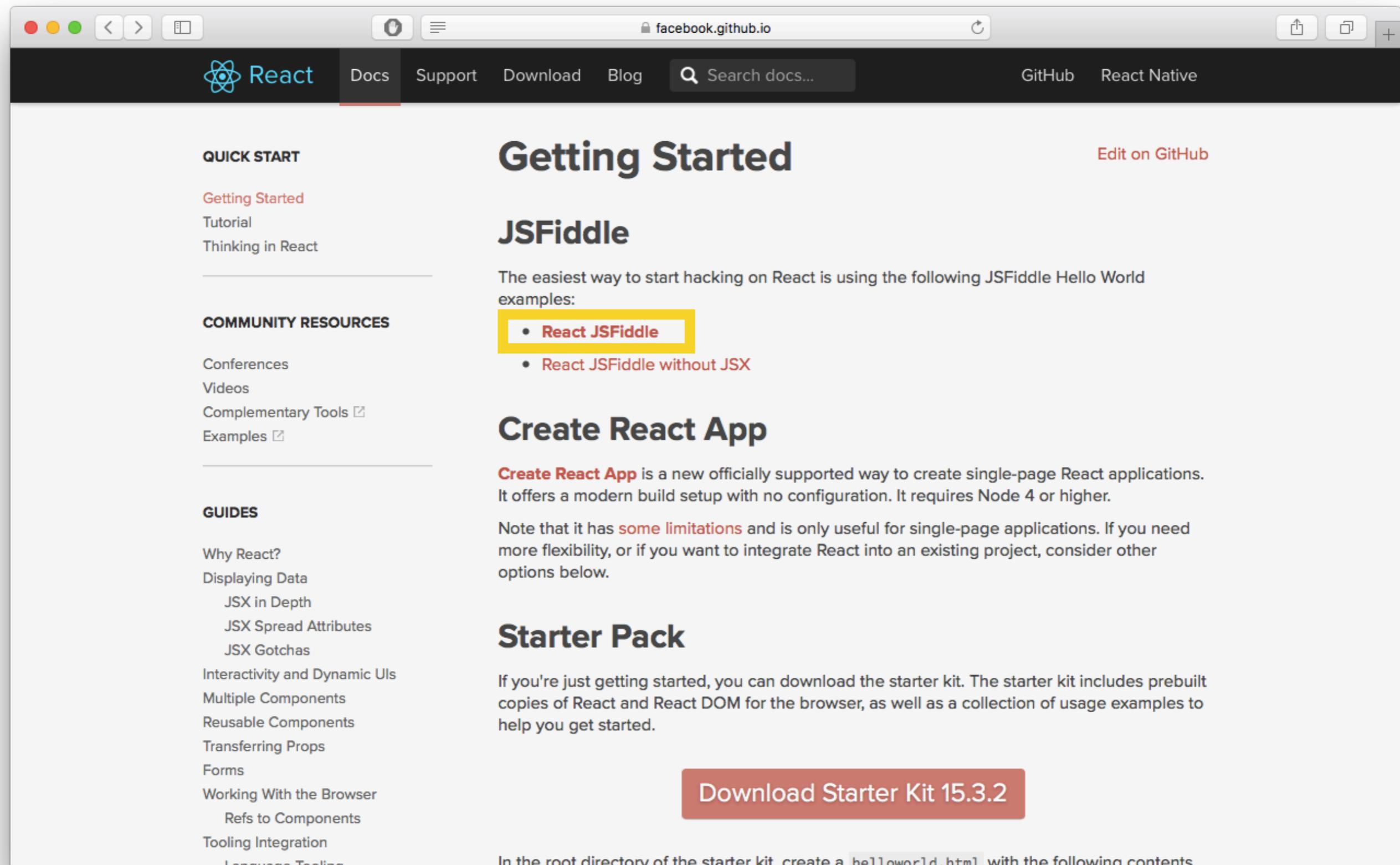
Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

**Learn Once, Write Anywhere**

We don't make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using [React Native](#).

Display a menu



A screenshot of a Mac OS X desktop showing a web browser window. The window title is 'facebook.github.io'. The page content is the 'Getting Started' documentation for React.js. The left sidebar has sections for 'QUICK START' (Getting Started, Tutorial, Thinking in React), 'COMMUNITY RESOURCES' (Conferences, Videos, Complementary Tools, Examples), and 'GUIDES' (Why React?, Displaying Data, JSX in Depth, JSX Spread Attributes, JSX Gotchas, Interactivity and Dynamic UIs, Multiple Components, Reusable Components, Transferring Props, Forms, Working With the Browser, Refs to Components, Tooling Integration, Language Tooling). The main content area features a large 'Getting Started' heading, a 'JSFiddle' section with examples, a 'Create React App' section with details and a 'Starter Pack' download button, and a note about creating a 'helloworld.html' file.

# Getting Started

[Edit on GitHub](#)

## JSFiddle

The easiest way to start hacking on React is using the following JSFiddle Hello World examples:

- [React JSFiddle](#)
- [React JSFiddle without JSX](#)

## Create React App

**Create React App** is a new officially supported way to create single-page React applications. It offers a modern build setup with no configuration. It requires Node 4 or higher.

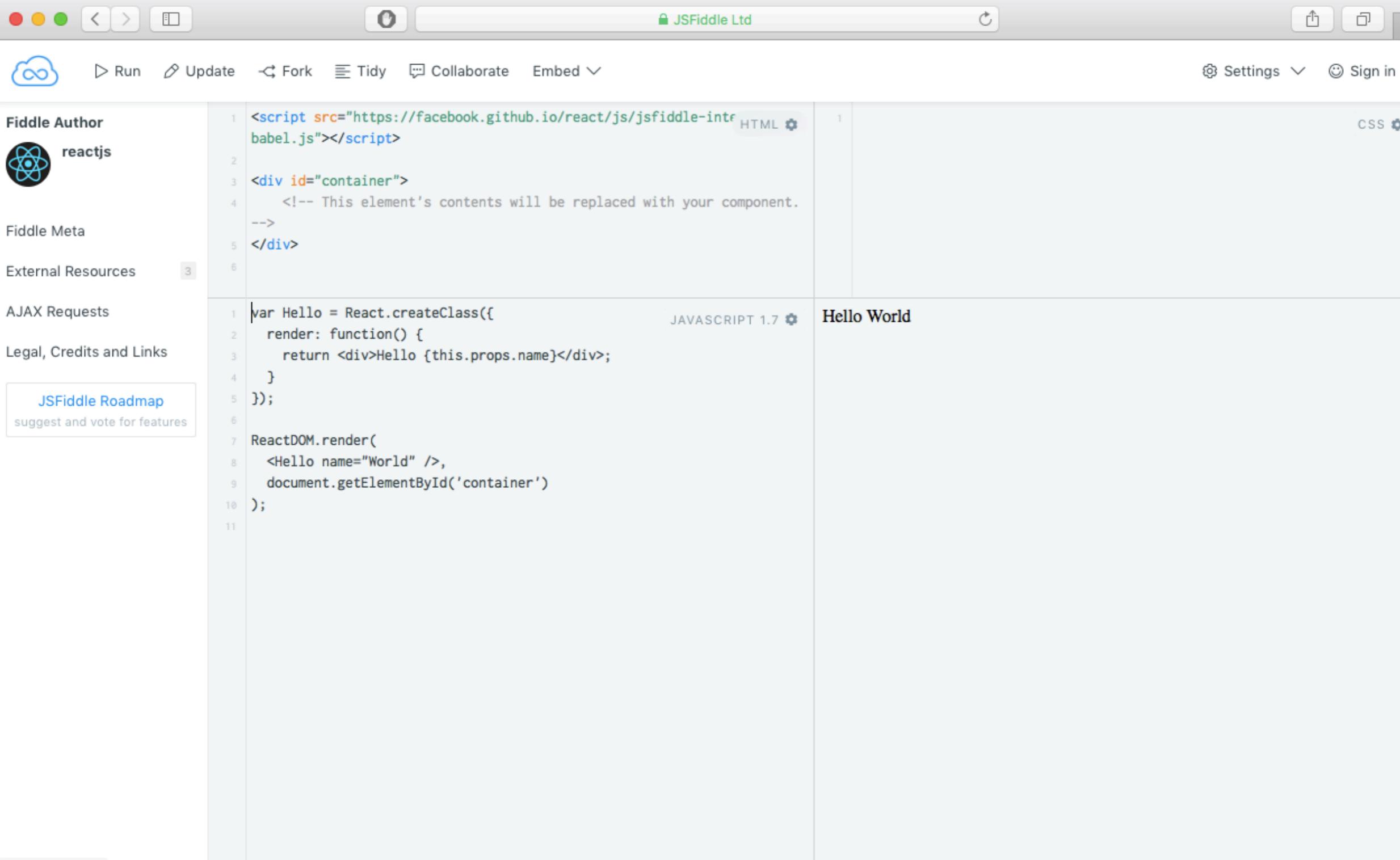
Note that it has [some limitations](#) and is only useful for single-page applications. If you need more flexibility, or if you want to integrate React into an existing project, consider other options below.

## Starter Pack

If you're just getting started, you can download the starter kit. The starter kit includes prebuilt copies of React and React DOM for the browser, as well as a collection of usage examples to help you get started.

[Download Starter Kit 15.3.2](#)

In the root directory of the starter kit, create a `helloworld.html` with the following contents.



The screenshot shows the JSFiddle interface with a React application. The left sidebar includes 'Fiddle Author' (reactjs), 'Fiddle Meta', 'External Resources' (3), 'AJAX Requests', 'Legal, Credits and Links', and a 'JSFiddle Roadmap' section. The main area has an 'HTML' tab with the following code:

```
1 <script src="https://facebook.github.io/react/js/jsfiddle-intro-babel.js"></script>
2
3 <div id="container">
4   <!-- This element's contents will be replaced with your component.
5   --&gt;
6 &lt;/div&gt;</pre>

The 'JAVASCRIPT 1.7' tab contains the following code:



```
1 var Hello = React.createClass({
2   render: function() {
3     return <div>Hello {this.props.name}</div>;
4   }
5 });
6
7 ReactDOM.render(
8   <Hello name="World" />,
9   document.getElementById('container')
10);
```



The output on the right shows the text "Hello World".


```

# Defining Components

- Two syntax choices:
  - Anonymous object module pattern
  - Class

```
var HelloMessage = React.createClass({  
  render: function() {  
    return <div>Hello  
      {this.props.name}</div>;  
  }  
});
```

**Module pattern**

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

**Class**

# Module Pattern vs. Classes

```
var HelloMessage = React.createClass({  
  render: function() {  
    return <div>Hello  
    {this.props.name}</div>;  
  }  
});
```

## Module pattern

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

## Class

- Classes are equivalent... except
  - Use constructor to set up state
  - Must manually *bind this* instance (these are not the classes you are looking for...)
- Classes offers shorter and cleaner syntax.
- But.... most documentation still using module pattern
- In this course, you can use whichever you'd like.

<https://facebook.github.io/react/docs/reusable-components.html>

# Demo: Greeting App

<https://jsfiddle.net/69z2wepo/57792/>

# Reacting to change

- What happens when state of component changes?
  - e.g., user adds a new item to list
- Idea
  1. Your code updates `this.state` of component when event(s) occur (e.g., user enters data, get data from network) using `this.setState(newState)`
  2. Calls to `this.setState` *automatically* cause `render` to be invoked by framework
  3. Reconciliation: Framework *diffs* output of `render` with *previous* call to `render`, updating only part of DOM that *changed*

```
class LikeButton extends React.Component {
  constructor() {
    super();
    this.state = { liked: false };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({liked: !this.state.liked});
  }

  render() {
    const text = this.state.liked ? 'liked' : 'haven\'t liked';
    return (
      <div onClick={this.handleClick}>
        You {text} this. Click to toggle.
      </div>
    );
  }
}

ReactDOM.render(<LikeButton />, document.getElementById('example'));
```

<https://jsfiddle.net/foqdzyLf/1/>

# What is state?

- All internal component data that, when changed, should trigger UI update
  - Stored as single JSON object `this.state`
- What isn't state?
  - Anything that could be computed from state (redundant)
  - Other components - should build them in render
  - Data duplicated from properties.

# Properties vs. State

- Properties should be **immutable**.
  - Created through attributes when component is instantiated.
  - Should *never* update
- State **changes** to reflect the current state of the component.
  - Can (and should) change based on the current internal data of your application.

```
var HelloMessage = React.createClass({  
  render: function() {  
    return <div>Hello {this.props.name}</div>;  
  }  
});  
ReactDOM.render(<HelloMessage name="John" />,  
  document.getElementById('mountNode'));
```

Property:

State:

```
handleClick() {  
  this.setState({liked: !this.state.liked});  
}
```

# Nesting components

- UI is often composed of nested components
  - Like containers in HTML, corresponds to hierarchy of HTML elements
  - But...now each element is a React component that is generated
- Parent *owns* instance of child
  - Occurs whenever component instantiates other component in render function
  - Parent configures child by passing in properties through attributes

# Nesting components

```
render: function() {  
  return (  
    <div>  
      <PagePic pagename={this.props.pagename} />  
      <PageLink pagename={this.props.pagename} />  
    </div>  
  );  
}
```

Establishes ownership by creating in render function.

Sets pagename property of child to value of pagename property of parent

# Reconciliation

```
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
```

```
<Card>
  <p>Paragraph 2</p>
</Card>
```

- Process by which React updates the DOM with each new render pass
- Occurs based on order of components
  - Second child of Card is destroyed.
  - First child of Card has text mutated.

<https://facebook.github.io/react/docs/multiple-components.html>

# Reconciliation with Keys

```
render: function() {
  var results = this.props.results;
  return (
    <ol>
      {results.map(function(result) {
        return <li key={result.id}>{result.text}</li>;
      })}
    </ol>
  );
}
```

- Problem: what if children are dynamically generated and have their own state that must be persisted across render passes?
  - Don't want children to be randomly transformed into other child with different state
- Solution: give children identity using keys
  - Children with keys will always keep identity, as updates will reorder them or destroy them if gone

# Demo: Todo in React

<https://jsfiddle.net/69z2wepo/57794/>