Exercise: React Components

React is “a JavaScript library for building user interfaces,” that gives you tools for interacting with the DOM that are cleaner and more performant than with raw JavaScript or jQuery.

React is **declarative**, rather than **imperative**. Imperative code, like raw JavaScript or jQuery, says *how* something ought to occur. Declarative code (HTML being an example) states *what* something ought to be, and hides the *how* away from the developer. Therefore, in React, you write **components** that looks a lot like HTML, and the React library, hidden from you, handles the details of how these components are actually rendered to the DOM.

Think of React components as superpowered HTML tags that:

- Can be given information for how they ought to behave
- Can communicate with each other
- Can be composed, reused and nested together to build small to very large applications

React is used on some of the largest websites in the world, including Instagram and Facebook, where React was developed. It is fast, does an excellent job supporting code modularity and reuse, and developers tend to find it clean and fun to work with.

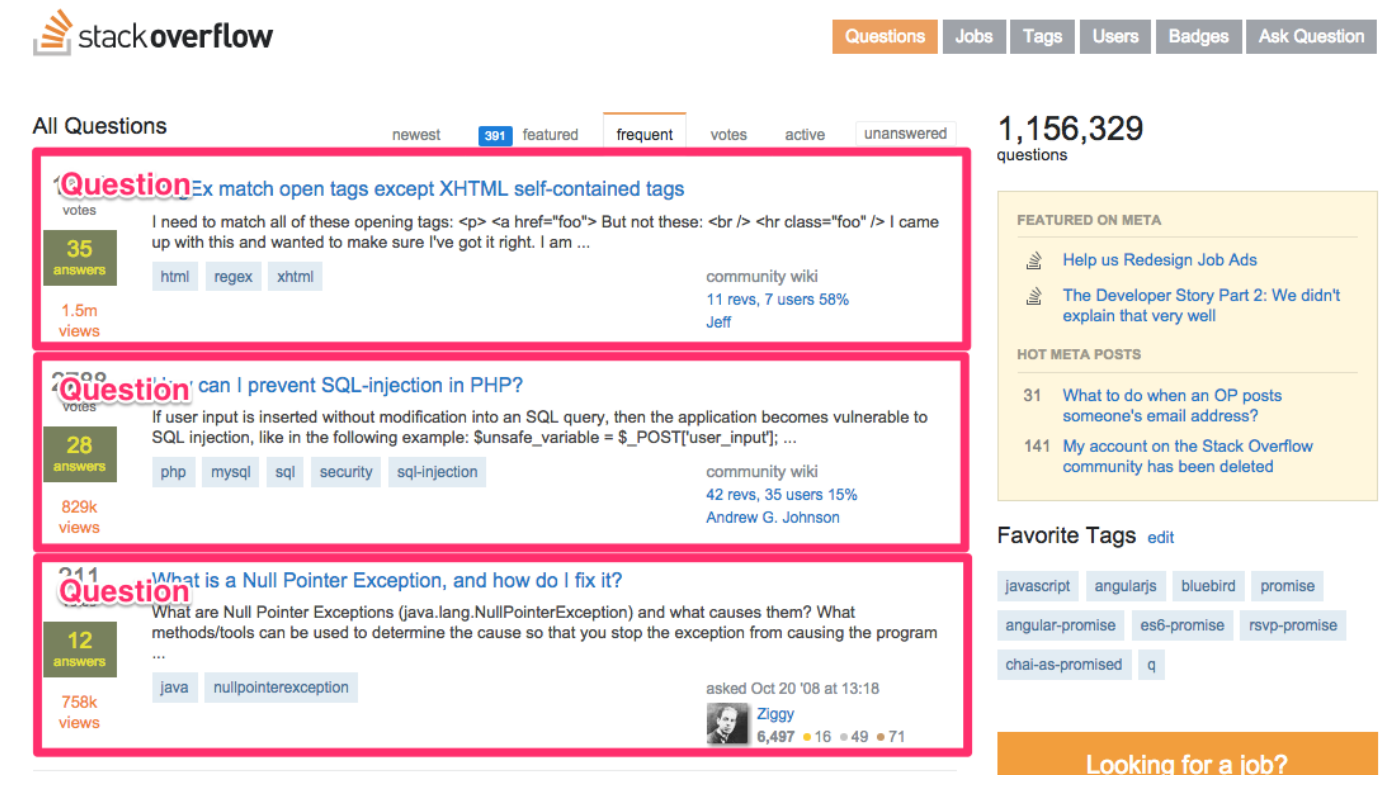
During this exercise you will:

- Learn how to create React components
- Learn how to render React components
- Learn how to use JSX and ES6 in React code
- Learn how to utilize `props`
- Learn how React components handle user events
- Learn the difference between stateless functional components and class components
- Learn why there is `state` and how to update it

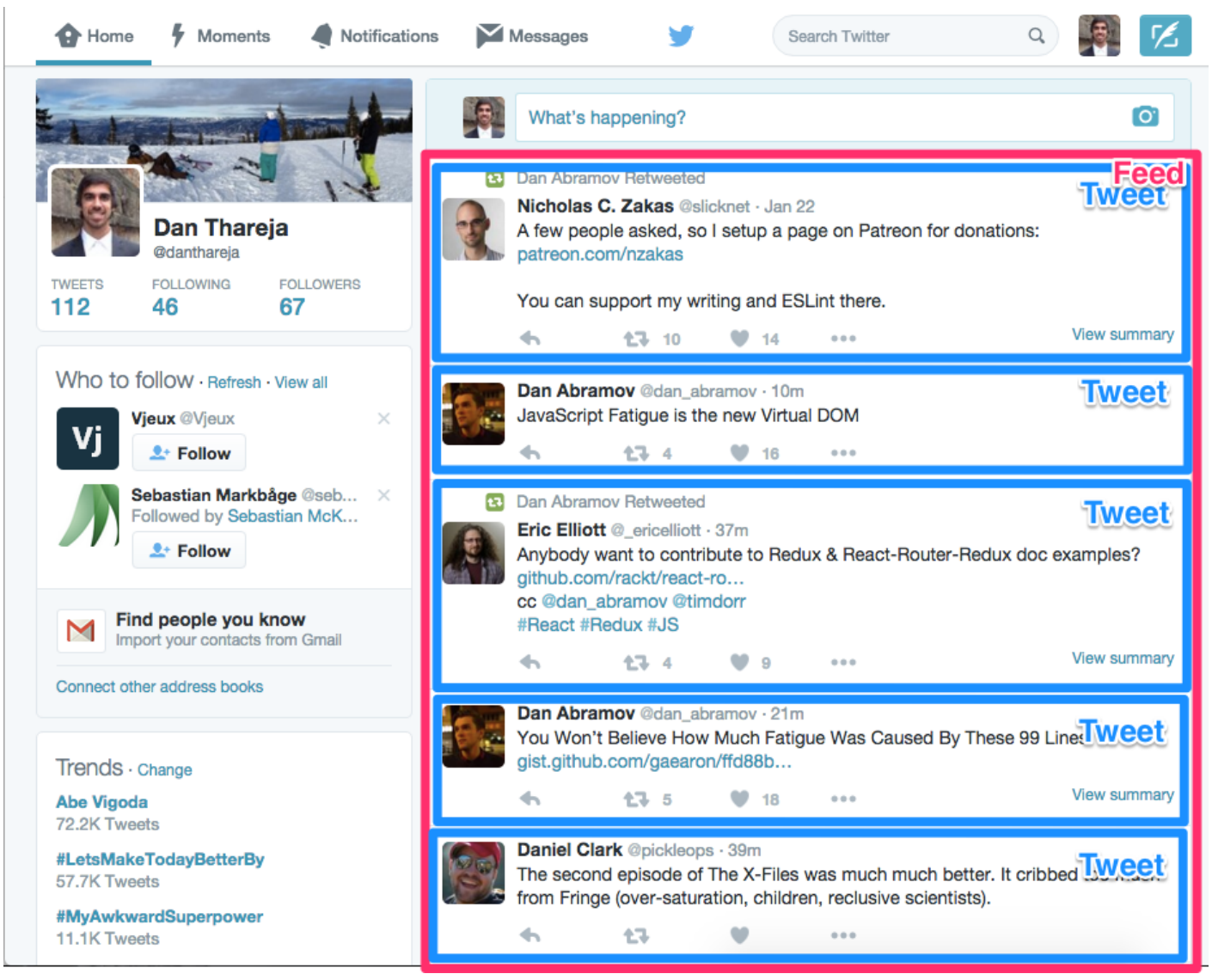
An example React App

React apps contain components, with names, that are potentially nested. The following represents possible component structures for some popular websites:

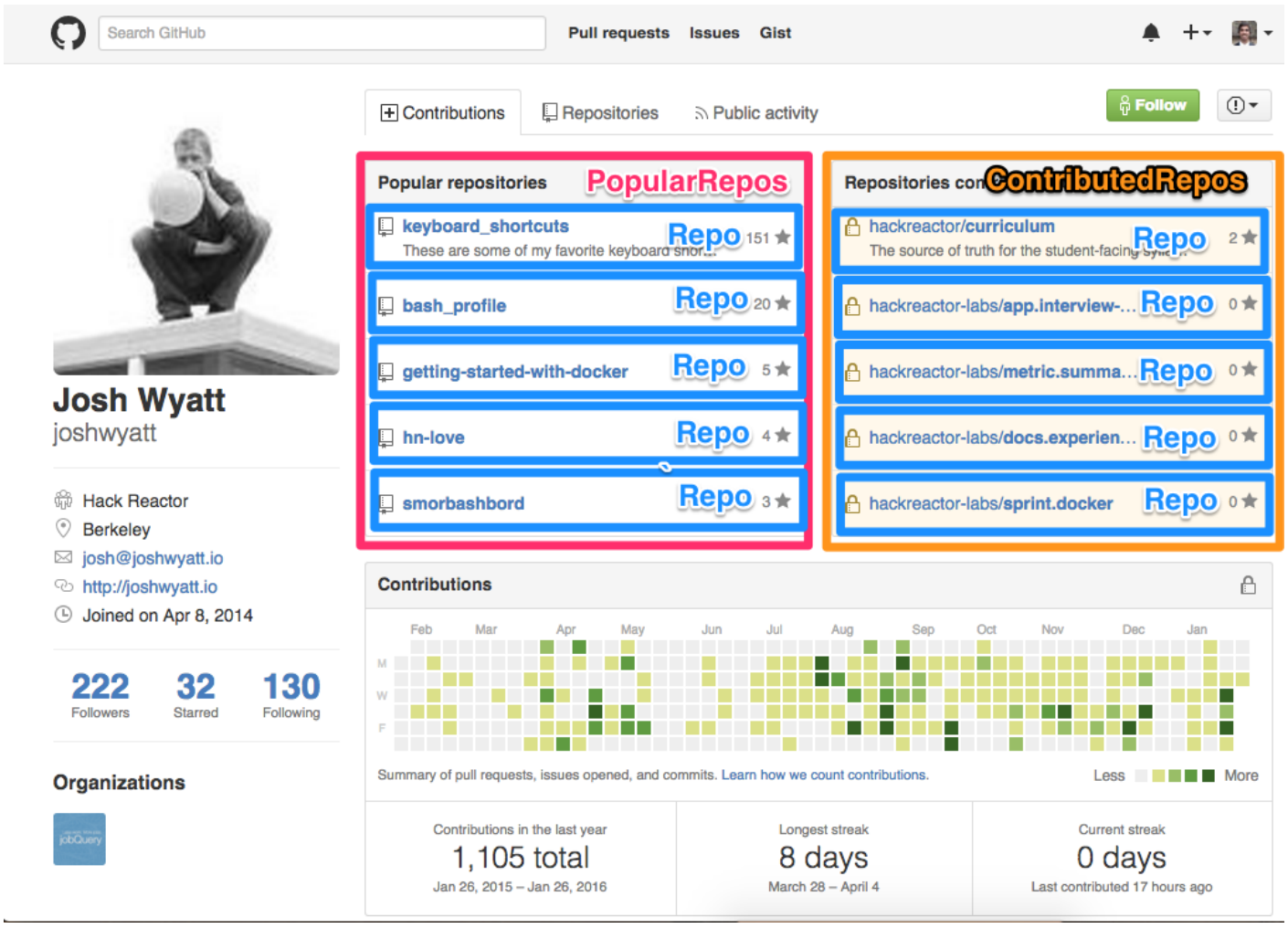
The Stack Overflow homepage might be composed of many `<Question>` components:



The Twitter homepage might be composed of a `<Feed>` component containing many nested `<Tweet>` components:



A GitHub profile page might be composed of a `<PopularRepos>` component, and a `<ContributedRepos>` component, both containing nested `<Repo>` components, demonstrating component reuse:



Setup

- [ ] `npm install -g live-server`
- [ ] `live-server`

Bare Minimum Requirements

There are two steps to making components in a React application:

1. Create a component
2. Render it to the page

Creating and rendering a React component

The following is a very basic React component, written in ES6, and **JSX**. JSX is a JavaScript syntax extension allowing you to write declaritive HTML-like code:

```
var App = () => (
  <div>Some cliche salutation</div>
);
```

In its most basic form, to *create* a React component you just write a function (with a capitalized name by convention) that returns JSX. The JSX should represent the intended HTML of the rendered component.

To *render* a React component you use `ReactDOM.render(componentInstance, DOMElement)`. JSX returns a component instance when you wrap a component in an HTML tag. Thus, to render the component `App` created above:

```
ReactDOM.render(<App />, document.getElementById("actual-dom-element-where-I-want-to-render-my-component"));
```

- [ ] Inside `app.js`, create a `GroceryList` component that contains an unordered list of 2 grocery items. Render this component to the `div` tag in `index.html` with an `id` of `app`

- [ ] To appreciate what ES6 and JSX offers when writing React code, copy the snippets above into the Babel REPL and see what they look like in ES5

Creating and rendering nested React components

When creating React components, you can return JSX representations of React component instances in addition to JSX representations of HTML. For example, assuming you have created a `TodoList` component:

```
var App = () => (
  <div>
    <h2>My Todo List</h2>
    <TodoList />
  </div>
);
```

- [ ] Create React components for the 2 items in your grocery list. For example, if your grocery list contains "cucumbers" and "kale", create a `Cucumbers` component and a `Kale` component
- [ ] Use these two new components inside your `GroceryList` component instead of the hardcoded `<li>`s

Component Properties aka "props"

React gives you a way to pass data into components, making them more dynamic and reusable. When creating a component, you can expect an argument to be passed in, typically called `props`. When creating an instance of this component, you can pass in these `props` using a syntax similar to passing properties into HTML elements.

Note that inside of JSX, you can write JavaScript expressions inside of curly braces:

```
// Here we create a `TodoList` component
var TodoList = (props) => (
  <ul>
    <li>{props.todos[0]}</li>
    <li>{props.todos[1]}</li>
    <li>{props.todos[2]}</li>
  </ul>
);

var App = () => (
  <div>
    <h2>My Todo List</h2>
    <TodoList todos={['Learn React', 'Crush Recast.ly', 'Maybe sleep']}/> //
Here we are creating an instance of the `TodoList` component
  </div>
);
```

- [ ] Create a reusable **`GroceryListItem`** component that dynamically renders a given grocery item
- [ ] Refactor **`GroceryList`** to dynamically render an array of **`groceryItems`**, utilizing your new **`GroceryListItem`** component

Handling User Events

React components come with built-in support for many user events.

```
var TodoList = (props) => {

  // This function will be called when the first `<li>` below is clicked on
  // Notice that event handling functions receive an `event` object
  // We want to define it where it has access to `props`

  var onListItemClick = (event) => {
    console.log('I got clicked');
  };

  // Because we used curly braces with this arrow function
  // we have to write an explicit `return` statement
  return (
    <ul>
      <li onClick={onListItemClick}>{props.todos[0]}</li>
      <li>{props.todos[1]}</li>
      <li>{props.todos[2]}</li>
    </ul>
  );
}

var App = () => (
  <div>
    <h2>My Todo List</h2>
    <TodoList todos={['Learn React', 'Crush Recast.ly', 'Maybe sleep']}/> //
Here we are creating an instance of the `TodoList` component
  </div>
);
```

Making applications interactive with state

Class components

Up until now we have been working with *stateless functional components* which are great when all you need to do is receive `props` and render JSX. Recall that `props` are immutable and cannot be changed once passed in from a parent. If an entire React application is created with only functional stateless components, it would not be very different from a static HTML page.

To make applications interactive, our components need to do more than simply receive `props`. Sometimes components need to store data that cannot be explicitly passed in as `props` and re-render this data changes.

React makes this possible with class components. To demonstrate this, we'll refactor each `<li>` of our `TodoList` into a `TodoListItem` class component:

```
// A class component can be defined as an ES6 class
// that extends the base Component class included in the React library
class TodoListItem extends React.Component {

  // A `constructor` method is expected on all ES6 classes
  // When React instantiates the component,
  // it will pass `props` to the constructor
  constructor(props) {
    // Equivalent to ES5's React.Component.call(this, props)
    super(props);
  }

  // Every class component must have a `render` method
  // Stateless functional components are pretty much just this method
  render() {

    // `props` is no longer passed as an argument,
    // but instead accessed with `this.props`
    return (
      <li>{this.props.todo}</li>
    );

  }

}

// Update our `TodoList` to use the new `TodoListItem` component
// This can still be a stateless function component!
var TodoList = (props) => (
  <ul>
    {props.todos.map(todo =>
      <TodoListItem todo={todo} />
    )}
  </ul>
);
```

- [ ] Refactor `GroceryListItem` to be a class component

State

We're going to add a feature to our `TodoListItem` that toggles a crossed-out style when its `<li>` is clicked. For this to be possible, each `TodoListItem` must know whether or not it should currently display as crossed out. Additionally, this information

changes over time as users click the item, so there is no way to model this data as an immutable `prop`. Instead, this data will live on the `state` object of the `TodoListItem` component.

`state` is only available on class components. We can initialize a class component's `state` in its constructor. To update the state, invoke `this.setState`. Whenever `this.setState` is called, the component re-renders.

```
class TodoListItem extends React.Component {
  constructor(props) {
    super(props);

    // `state` is just an object literal
    this.state = {
      done: false
    };
  }

  // When a list item is clicked, we will toggle the `done`
  // boolean, and our component's `render` method will run again
  onListItemClick() {
    this.setState({
      done: !this.state.done
    });
  }

  render() {
    // Making the style conditional on our `state` lets us
    // update it based on user interactions.
    var style = {
      textDecoration: this.state.done ? 'line-through' : 'none'
    };

    // You can pass inline styles using React's `style` attribute to any
component
    // snake-cased css properties become camelCased this this object
    return (
      <li style={style} onClick={this.onListItemClick.bind(this)}>
{this.props.todo}</li>
    );
  }
}
```

- [ ] Make it so that when your mouse hovers over a `<li>` of a `GroceryListItem` that it turns bold

1 **React Components**
Submitted on 07/30/2019

Submit the URL to your completed project below.

https://github.com/KeitelDOG/hrr40-react-components.git

RESET INPUT SUBMIT

✓ Submitted

NEXT: