The Complete Beginner's Guide to React

By Kristen Dyrr

Software Engineer and Web Developer

Table of Contents

Chapter 1: Beginner's Guide to React.js, With Examples

React.js is a JavaScript library that was created by Facebook. It is often thought of as the "view" in a model-view-controller (MVC) user interface. This makes sense when you consider the fact that the only function that must be implemented in React is the "render" function. The render function provides the output that the user sees (the "view").

Let's take a look at why you may want to use React and how to set up a basic interface.

Download the source code

You can download all of the files associated with this tutorial from here.

Learn React online

If you are keen to learn React from the ground-up feel free to check Learn and Understand React JS on Zenva Academy which covers all the basics + lots of bonus topics like React Router and Flux.

Tutorial requirements

- This tutorial assumes that you have at least a beginner's grasp of HTML and JavaScript.
- You will need to download the React library if you want to go beyond the testing phase. We show you how to get around this during testing.
- You will need a text editor of some sort. Notepad++ is popular for those on Windows machines, and TextMate is popular on a Mac. Editors that highlight code are preferable.
- Normally, you would incorporate React into a larger application. If you want to test the basic code with external data files at the end of the tutorial, you will need to use a local or remote web server to get the page to work. MAMP is popular on Mac, and WAMP is most common on Windows machines. You can also use a lightweight Mongoose web server, or Python's HTTP server. Many people use React with Node.js, so you can also use a Node.js server. The React library download page (above) also includes a server and other options.

Downloading React and getting started

There are many options for downloading and installing React and its various add-ons, but the fastest way to get started and begin playing around with React is to simply serve the JavaScript files directly from the CDN (as described on the React GitHub page… the most common CDN options are listed there):

```
<!-- The core React library -->
<script src="https://fb.me/react-0.14.1.js"></script>
<!-- The ReactDOM Library -->
<script src="https://fb.me/react-dom-0.14.1.js"></script>
```

Both the download pages go into detail on the various ways to download, install, and serve React in various formats, but we're going to stick with this most basic option so we can focus on learning how to code with the React library itself. It's a good idea to have the React API open while you work for reference.

From there, we create an index.html file, and a main.js file. I've also included a css file for basic styling:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Game Development at ZENVA.com</title>
    <!-- Just for basic styling. -->
    <link rel="stylesheet" href="assets/css/base.css" />
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="main.js"></script>
  </body>
</html>
```

In order to get around using a server while testing, I'm calling the React.js, react-dom.js, and the browser.min.js babel-core files from the CDN. You wouldn't want to do this in production. The babel-core file allows us to use JSX, and the script type must be "text/babel" in order for it to work properly. Our main JavaScript code goes in main.js, and that's where it all starts.

Why React is better with JSX

Most React implementations make use of JSX, which allows you to put XML-like syntax right within JavaScript. Since React displays output as it's main function, we will be using HTML in just about every component. JSX simplifies the code that would normally have to be written in JavaScript, which makes your code much more readable and simplified.

JSX is not required, but consider the difference between two very simple statements. The following statement is created without JSX:

```
var element = React.createElement('div', { className: 'whatever' }, 'Some text');
```

The following is with JSX:

```
var element = <div className="whatever">
    Some text
</div>
```

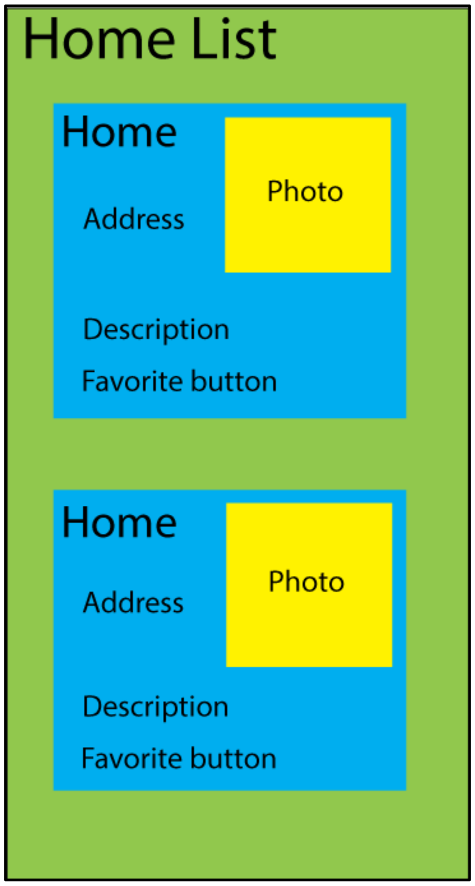
As you can see, the JSX code is much easier to read. Now that we have a basic understanding of what our output syntax will look like, let's dig in and learn the building blocks of React.

Understanding React components

React is based on components and states. This is what makes React such a popular library. When you want to create an application, you usually break it into simpler parts. When programming with React, you will want to break your interface into its most basic parts, and those will be your React components.

Components are wonderful because they are modular and reusable. You can take a basic component used in one area of an application and reuse it in others without having to duplicate code. This helps to speed up development.

Components can be nested, so that the most basic components can be grouped into a parent component. For example, if you were to create a home listing interface with React, the top level component would be the home list itself. Within the list, you would have a description of a single home. Within the home component, you would have the address of the home, as well as other small components such as a photo, perhaps a favorite or save button, and a link to view details and a map.



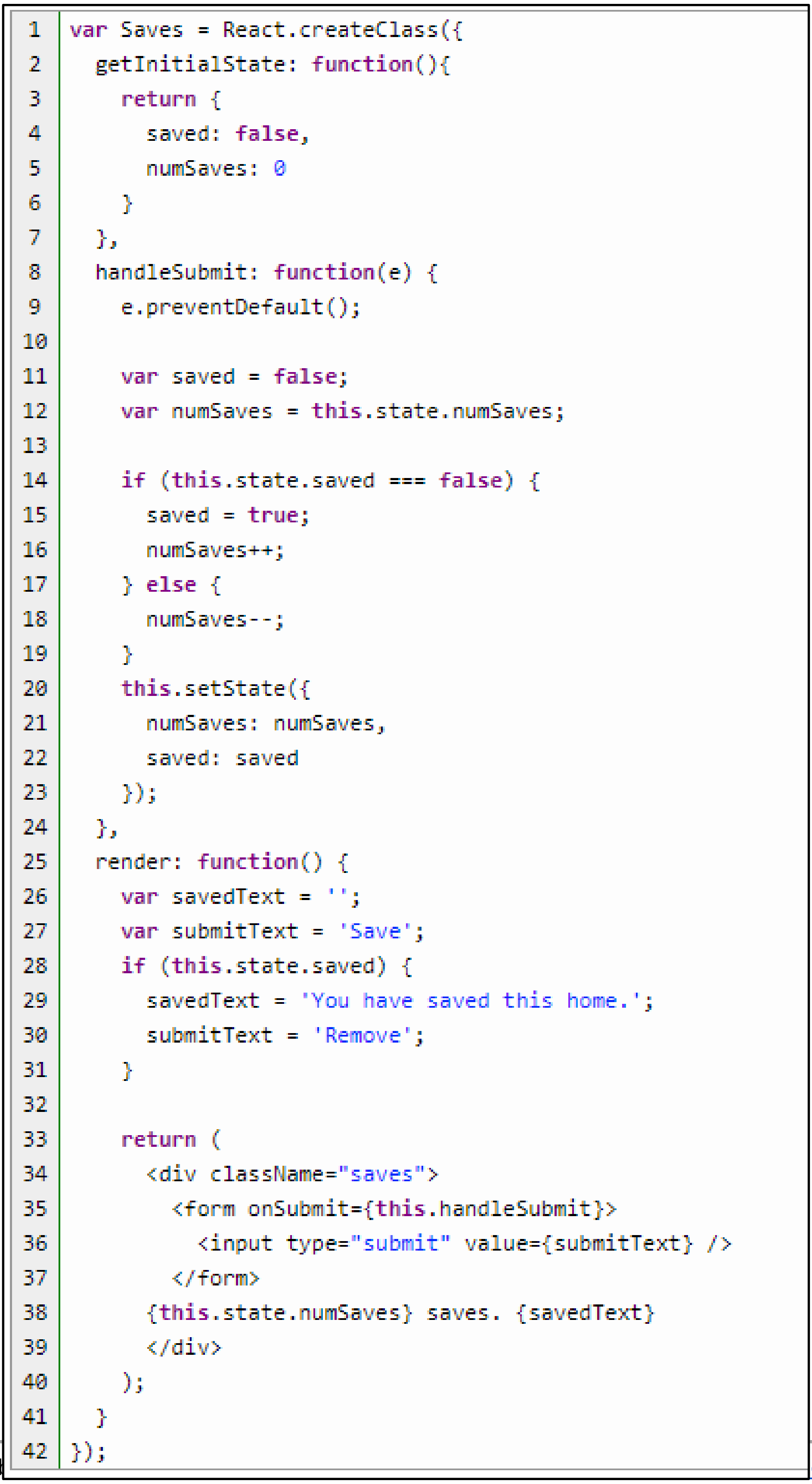
Now let's see how this information can be updated when it changes.

React component states

Component states are what give React its name. Any time a component's state changes, its "render" function is called, updating the output. In essence, each component "reacts" to changes, which is handy for any user interface. Data stored in a state should be information that will be updated by the component's event handlers (changes that should update in real time as the interface is being used).

If we were to have a home listing, any number of things may change in the database. New photos could be added, and a home can be bought and sold. You wouldn't necessarily want any of these things to update in real time in the interface.

One thing that may change very quickly, however, would be the number of times a home is saved (or favorited or liked) by a user. The little component that displays the number of saves could update as soon as the person clicks the Save button by updating the state the moment the button is clicked. Let’s build the Saves component to show what this would look like:

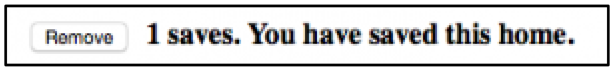


```
var Saves = React.createClass({
  getInitialState: function(){
    return {
      saved: false,
      numSaves: 0
    }
  },
  handleSubmit: function(e) {
    e.preventDefault();

    var saved = false;
    var numSaves = this.state.numSaves;

    if (this.state.saved === false) {
      saved = true;
      numSaves++;
    } else {
      numSaves--;
    }
    this.setState({
      numSaves: numSaves,
      saved: saved
    });
  },
  render: function() {
    var savedText = '';
    var submitText = 'Save';
    if (this.state.saved) {
      savedText = 'You have saved this home.';
      submitText = 'Remove';
    }

    return (
      <div className="saves">
        <form onSubmit={this.handleSubmit}>
          <input type="submit" value={submitText} />
        </form>
      {this.state.numSaves} saves. {savedText}
      </div>
    );
  }
});
```

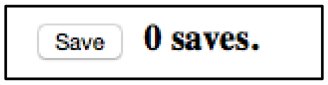
This won't actually do anything yet, because we don't have any statement to call on Saves. Still, this component describes what we do with components and states. You also won't find this code anywhere in the files, because we will have to move things around a bit to make it work the way we want, but this is fine for now.

You may notice a naming convention right away. Most element names start with a lowercase letter, followed by capitalization. The names of React classes, however, begin with an uppercase letter. We then have the React.createClass() function, which creates our component.

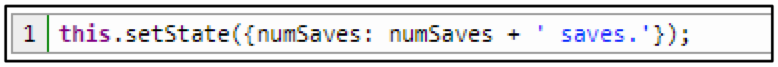
The render method is that most-important method that produces the actual output, and is normally placed last in the component. As you can see, the output depends on the value of two states: saved and numSaves. Here's what the output will look like when the user has saved the home:



And when they have not saved the home:



Just don't include computed data in a state. What that means in this case is that, while you want to update the number of saves in the state when the Save button is clicked, you don't want to save the following in the state:



```
this.setState({numSaves: numSaves + ' saves.'});
```

Save the addition of the string for the render function. All we want to save in the state is the data that gets updated by the component's event handler, and that is simply the number of favorites.

You can also see JSX at work in the render method. The return value (just don't forget the parentheses around the return output!) appears to be just regular HTML right within your JavaScript code! Well, not exactly. JSX is JavaScript, so the "class" attribute is discouraged. Instead, use "className." In addition, JavaScript expressions used as attribute values must be enclosed in curly braces rather than quotes.

Because we are dealing with states, we need to set an initial state so that the state variables will always be available in the render method without errors. This is why we use the getInitialState() method at the top of the component.

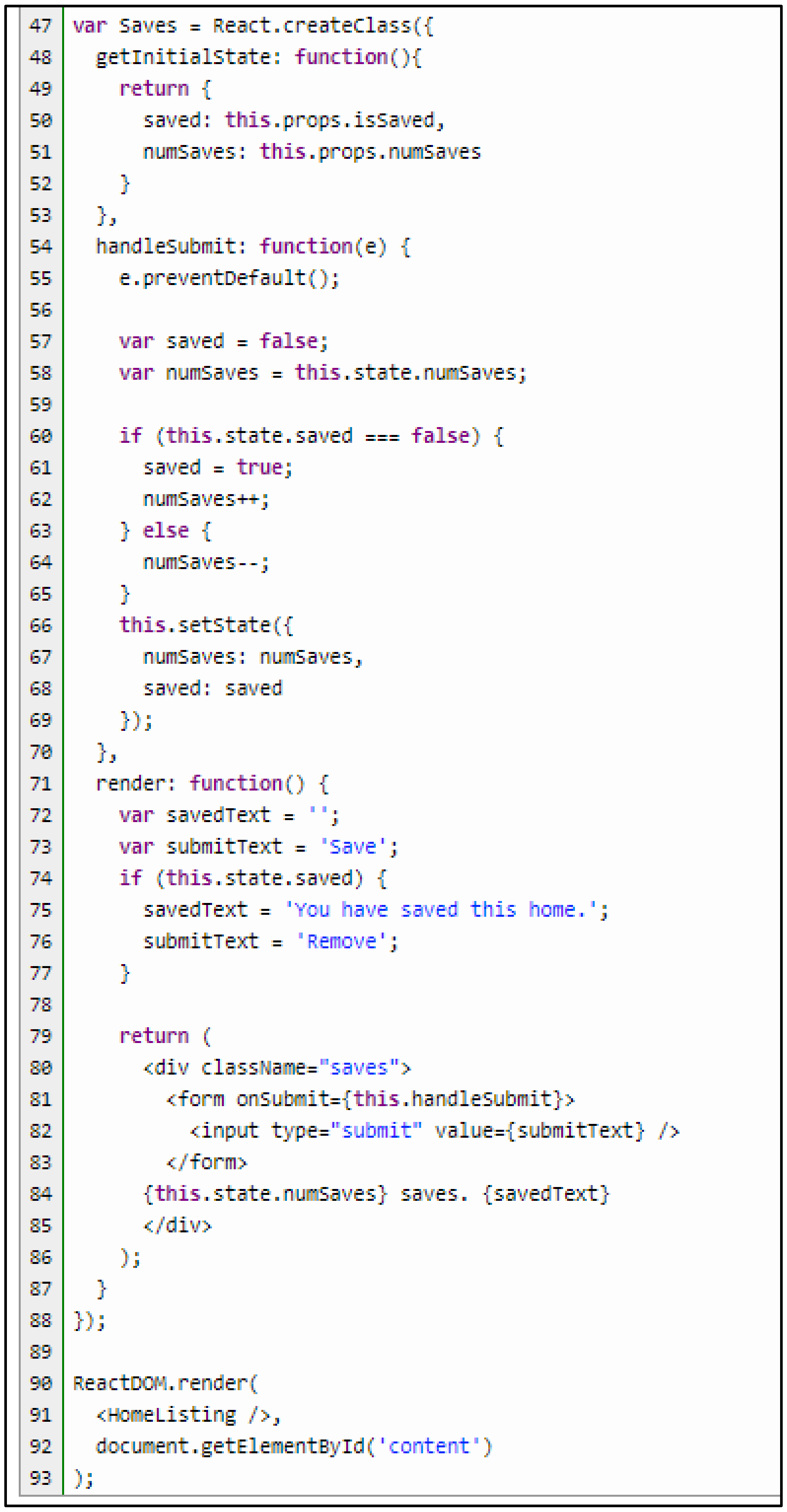
The handleSubmit function is where we handle the pressing of the Save (or Remove) button. We first have to call preventDefault() in order to prevent the form from being submitted the normal way. We then process the data, so that the user will save the home if it's not already saved, or remove it from their saves if it is saved. As soon as we save the new state at the end of the function, the render method will be called automatically, which will update the display. Now, let's look at the rest of the example in more detail.

How to use props

Components pass properties to their children components through the use of props. One thing to keep in mind is that props should never be used to determine the state of a component. A property would be something such as the address of a home, passed from the listing component to the individual home component. The state, on the other hand, should depend on data that may be updated, such as the number of times people have saved the home.

Let's look at the rest of this basic example. Taking into account the proper use of props and states, you may notice that there are some problems with this code:

```
var HomeListing = React.createClass({
  render: function() {
    return (
      <div className="homeList">
        <Home
          key={0}
          id={0}
          isSaved={false}
          photo="assets/images/home.jpg"
          address="12345 Beverly Dr"
          numSaves={52}
        >
          This is a home in the city
        </Home>
      </div>
    );
  }
});

var Home = React.createClass({
  render: function() {
    return (
      <div className="home">
        <span className="homeAddress">
          {this.props.address}
        </span>
        <Photo src={this.props.photo}></Photo>
        <span className="homeDescription">
          {this.props.children}
        </span>
        <Saves
          id={this.props.id}
          isSaved={this.props.isSaved}
          numSaves={this.props.numSaves}
        ></Saves>
      </div>
    );
  }
});

var Photo = (props) => {
  return (<div className="homePhoto">
    <img src={props.src} />
  </div>);
};
```



```
var Saves = React.createClass({
  getInitialState: function(){
    return {
      saved: this.props.isSaved,
      numSaves: this.props.numSaves
    }
  },
  handleSubmit: function(e) {
    e.preventDefault();

    var saved = false;
    var numSaves = this.state.numSaves;

    if (this.state.saved === false) {
      saved = true;
      numSaves++;
    } else {
      numSaves--;
    }
    this.setState({
      numSaves: numSaves,
      saved: saved
    });
  },
  render: function() {
    var savedText = '';
    var submitText = 'Save';
    if (this.state.saved) {
      savedText = 'You have saved this home.';
      submitText = 'Remove';
    }

    return (
      <div className="saves">
        <form onSubmit={this.handleSubmit}>
          <input type="submit" value={submitText} />
        </form>
        {this.state.numSaves} saves. {savedText}
      </div>
    );
  }
});

ReactDOM.render(
  <HomeListing />,
  document.getElementById('content')
);
```

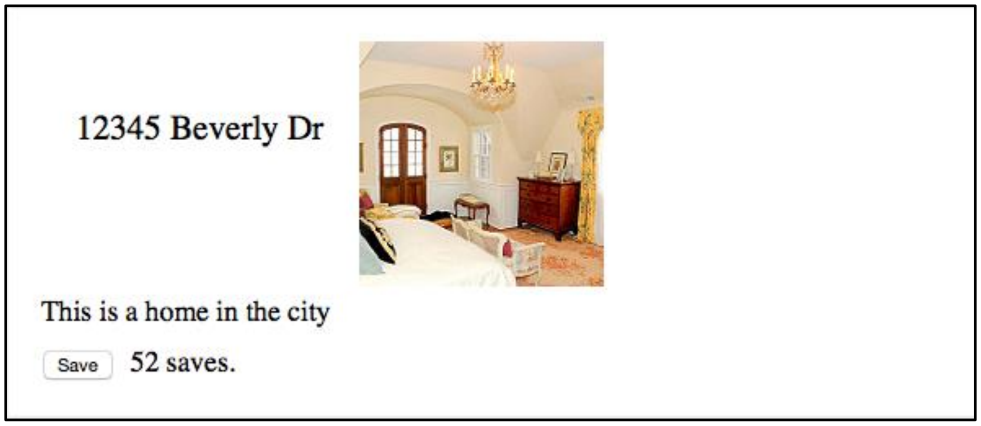
For this basic example, we are just going to hard-code a home listing. It all starts at the bottom, where we have the ReactDOM.render() call. Version 0.14 of React separated React into two modules: React and ReactDOM. ReactDOM exposes DOM-specific methods, and React includes the core tools shared by React on different platforms. <HomeListing /> is basically a call to a component, in XML-style code. The naming convention states that regular HTML code is lowercase, while component names are capitalized. Everything will be placed in the "content" tag, which is included in the index.html file.

Normally, the HomeListing tag would include attributes that tell the component where to grab data, such as the location of a file URL. These attributes would be called using props. For now, we will focus on other props within our nested components. The HomeListing component in this example consists only of a render function, which calls on a nested component called "Home." The call to Home contains all the attributes needed to describe the home listing, with JavaScript attribute values (including raw numbers, as well as true and false values) in curly braces.

The Home component is where we see our first use of props. Basically, any custom attribute sent through when calling a child component may be accessed through this.props. The most unique of these is this.props.children. The "children" property accesses whatever was included within the opening and closing tags of the call to the component (not including the attributes).

Notice in the call to the Home component that we include a "key" attribute, as well as an "id" attribute with the same value. The key is used internally by React, and is not available as a prop. It allows React to keep track of components that may be shuffled or removed. Since we also need some sort of id for our application, we pass in a second "id" attribute with the same value, which may be used as a prop.

Most of our output is displayed directly at the Home level, but we also call on two more children components: Photo and Saves. Photo doesn't really need to be a child component in this instance, but it's common for photos to have special features, which could be included in the future. Because it's so small, it uses special syntax available to stateless components that have only a render method and nothing else. As you can see, the syntax allows the component to be extremely small and simple. The most important component in this case, however, is the Saves component. First, let's take a look at our output from this code:


12345 Beverly Dr

This is a home in the city

52 saves.

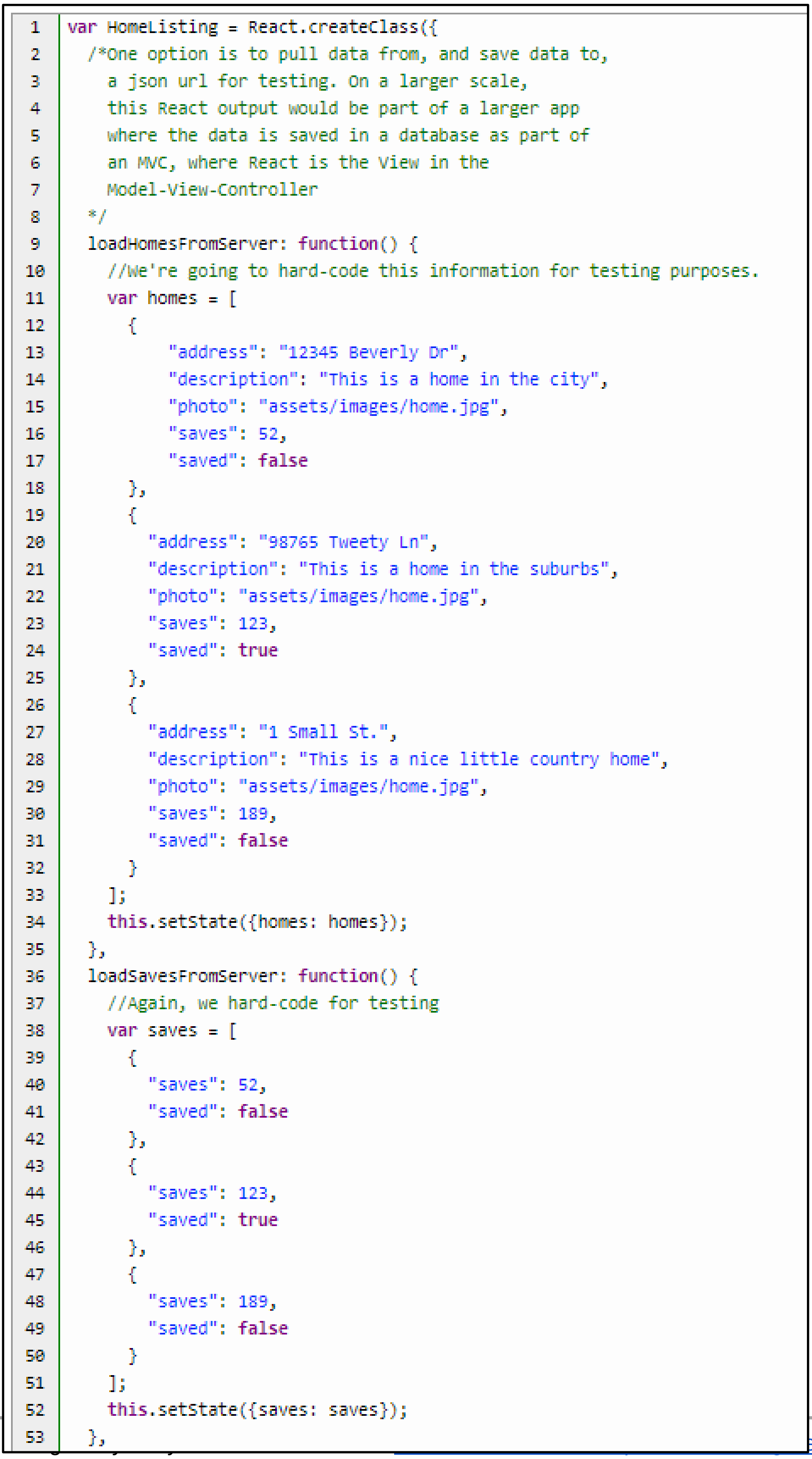
It pretty much looks like what we want, doesn't it? Well, not so fast. As you can see from the Saves component (which we discussed above), all the action is happening right here. The output shows that 52 people have saved the home before you. The problem is, how does the application save any new information. How will it "remember" that you saved the home too?

This is where things get just a little more complicated. First of all, you would normally include React as one piece of a larger app that will do all the processing of the data, where data is pulled from a database or file, then saved back to the database or file as the interface is being used. For now, we can hard-code this data as JSON, and show you at what point the data would be saved at the other end.

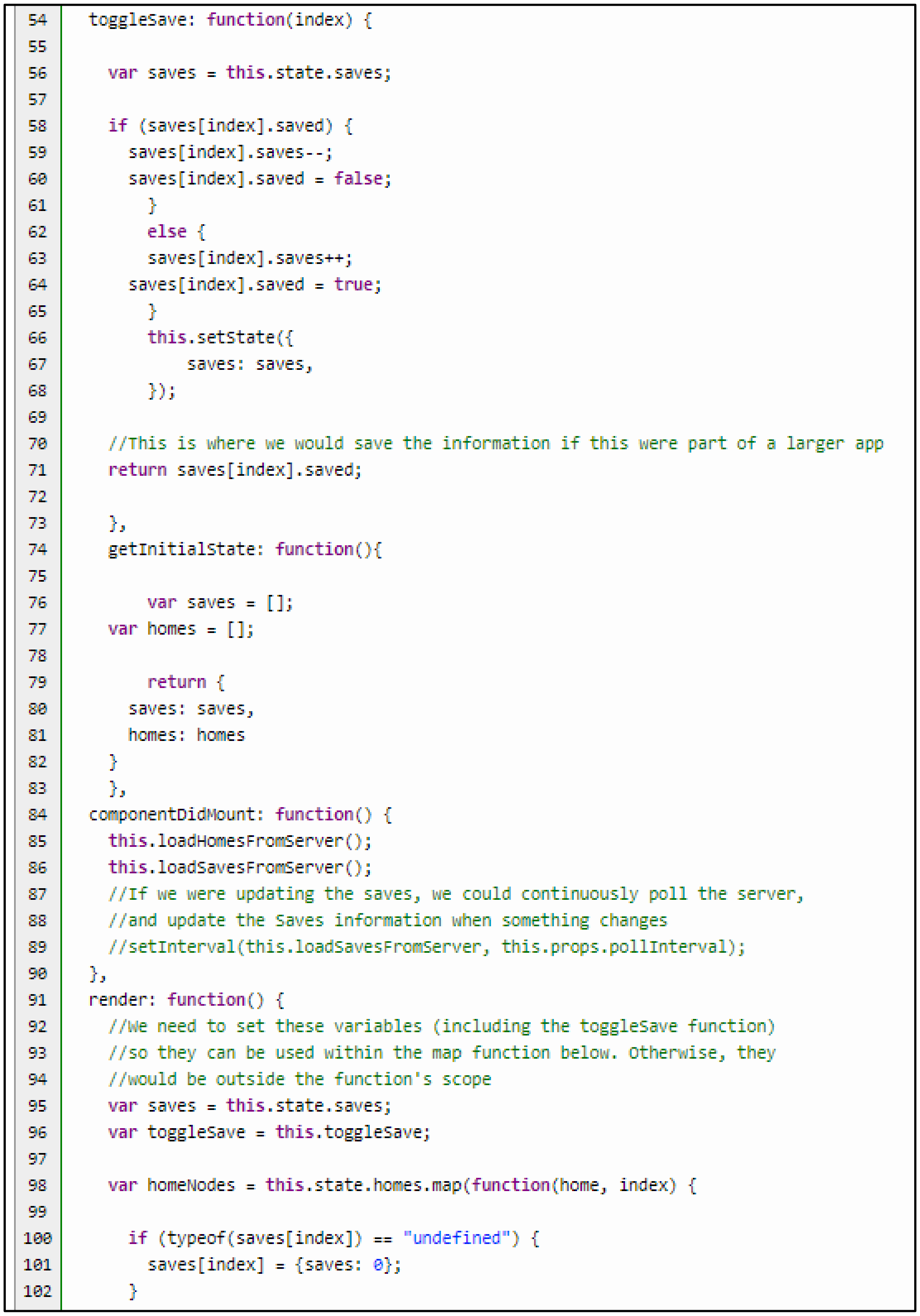
Organizing your interface

What we need to do is pull data from the top level component, and save data at the top level as well. This is because the total number of saves is being shared with all users. If our application was sharing information that is only relevant to each individual user, we could pull and save data in a child component, because what other people see wouldn't matter.

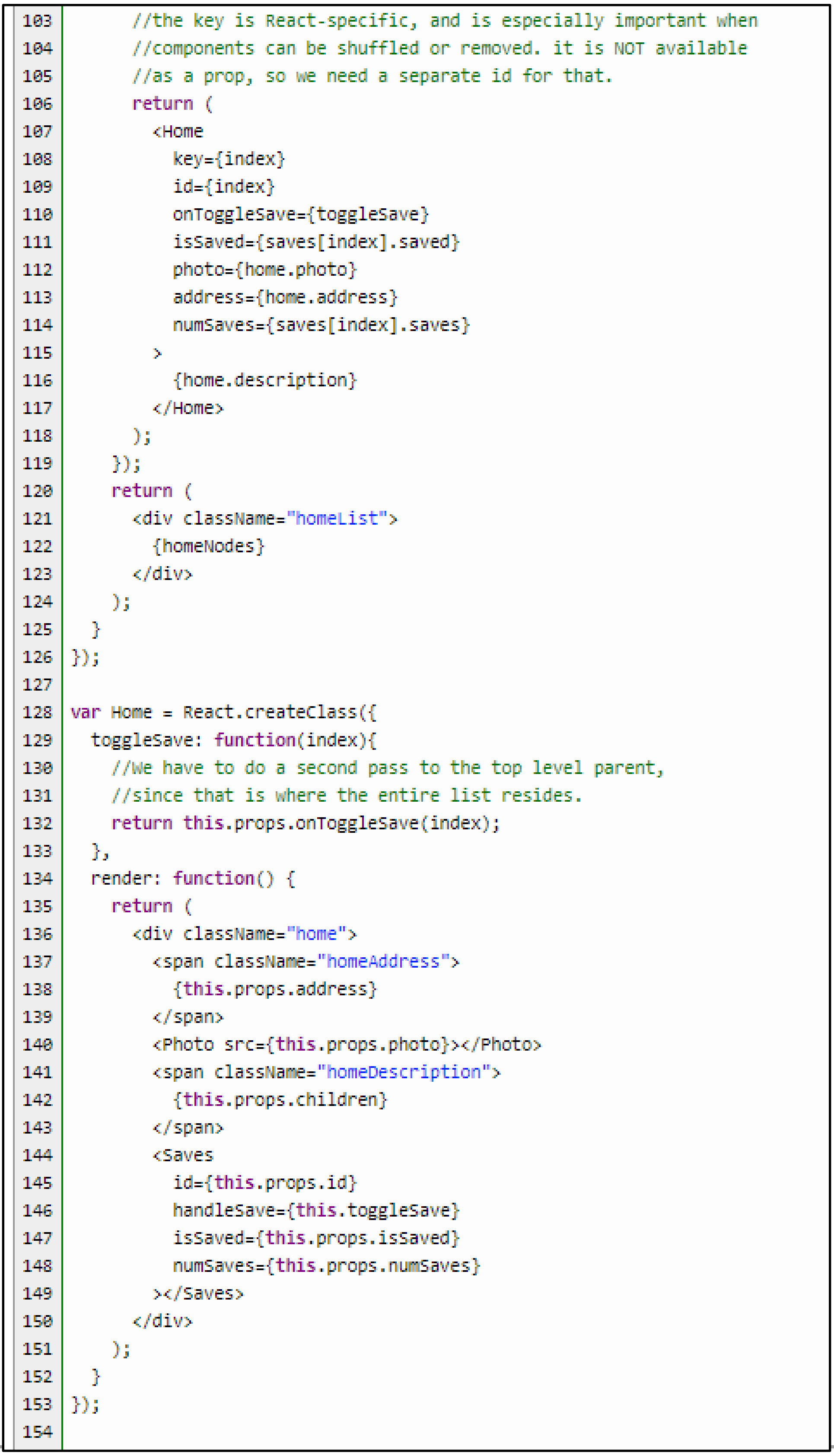
Here is our updated code:



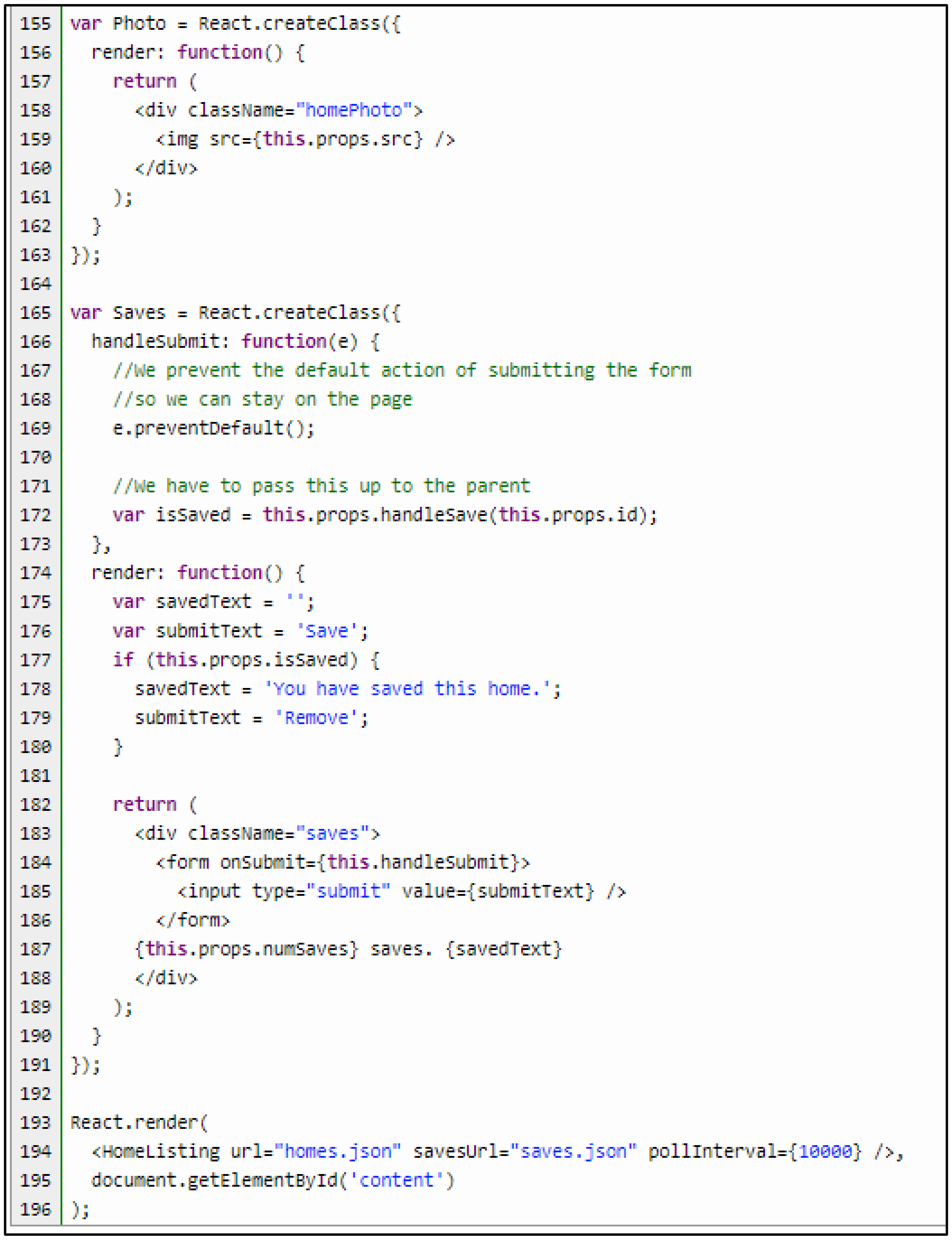
```
var HomeListing = React.createClass({
  /*One option is to pull data from, and save data to,
    a json url for testing. On a larger scale,
    this React output would be part of a larger app
    where the data is saved in a database as part of
    an MVC, where React is the View in the
    Model-View-Controller
  */
  loadHomesFromServer: function() {
    //We're going to hard-code this information for testing purposes.
    var homes = [
      {
          "address": "12345 Beverly Dr",
          "description": "This is a home in the city",
          "photo": "assets/images/home.jpg",
          "saves": 52,
          "saved": false
      },
      {
        "address": "98765 Tweety Ln",
        "description": "This is a home in the suburbs",
        "photo": "assets/images/home.jpg",
        "saves": 123,
        "saved": true
      },
      {
        "address": "1 Small St.",
        "description": "This is a nice little country home",
        "photo": "assets/images/home.jpg",
        "saves": 189,
        "saved": false
      }
    ];
    this.setState({homes: homes});
  },
  loadSavesFromServer: function() {
    //Again, we hard-code for testing
    var saves = [
      {
        "saves": 52,
        "saved": false
      },
      {
        "saves": 123,
        "saved": true
      },
      {
        "saves": 189,
        "saved": false
      }
    ];
    this.setState({saves: saves});
  },
```



```
toggleSave: function(index) {

  var saves = this.state.saves;

  if (saves[index].saved) {
    saves[index].saves--;
    saves[index].saved = false;
      }
      else {
      saves[index].saves++;
    saves[index].saved = true;
      }
      this.setState({
          saves: saves,
      });

  //This is where we would save the information if this were part of a larger app
  return saves[index].saved;

  },
  getInitialState: function(){

      var saves = [];
  var homes = [];

      return {
    saves: saves,
    homes: homes
  }
  },
componentDidMount: function() {
  this.loadHomesFromServer();
  this.loadSavesFromServer();
  //If we were updating the saves, we could continuously poll the server,
  //and update the Saves information when something changes
  //setInterval(this.loadSavesFromServer, this.props.pollInterval);
},
render: function() {
  //We need to set these variables (including the toggleSave function)
  //so they can be used within the map function below. Otherwise, they
  //would be outside the function's scope
  var saves = this.state.saves;
  var toggleSave = this.toggleSave;

  var homeNodes = this.state.homes.map(function(home, index) {

    if (typeof(saves[index]) == "undefined") {
      saves[index] = {saves: 0};
    }
```



```
      //the key is React-specific, and is especially important when
      //components can be shuffled or removed. it is NOT available
      //as a prop, so we need a separate id for that.
      return (
        <Home
          key={index}
          id={index}
          onToggleSave={toggleSave}
          isSaved={saves[index].saved}
          photo={home.photo}
          address={home.address}
          numSaves={saves[index].saves}
        >
          {home.description}
        </Home>
      );
    });
    return (
      <div className="homeList">
        {homeNodes}
      </div>
    );
  }
});

var Home = React.createClass({
  toggleSave: function(index){
    //We have to do a second pass to the top level parent,
    //since that is where the entire list resides.
    return this.props.onToggleSave(index);
  },
  render: function() {
    return (
      <div className="home">
        <span className="homeAddress">
          {this.props.address}
        </span>
        <Photo src={this.props.photo}></Photo>
        <span className="homeDescription">
          {this.props.children}
        </span>
        <Saves
          id={this.props.id}
          handleSave={this.toggleSave}
          isSaved={this.props.isSaved}
          numSaves={this.props.numSaves}
        ></Saves>
      </div>
    );
  }
});

```



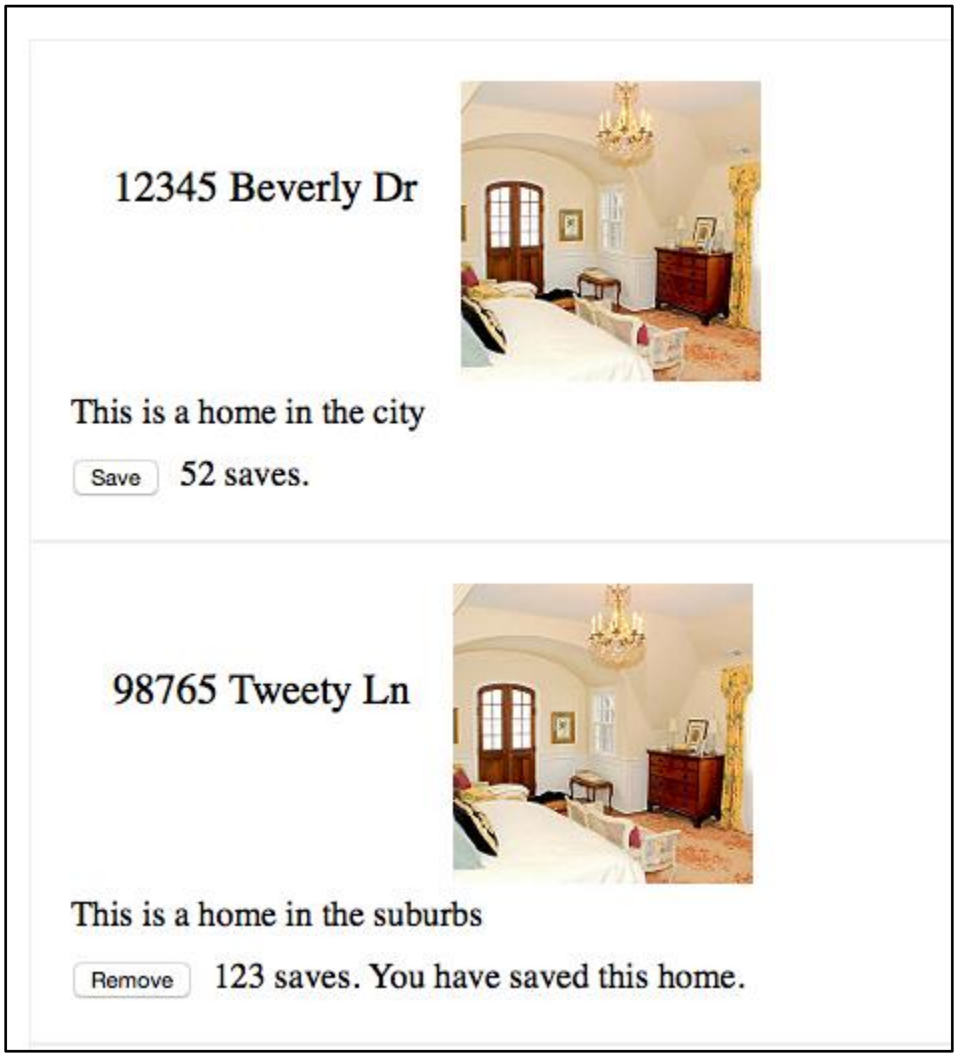
```
var Photo = React.createClass({
  render: function() {
    return (
      <div className="homePhoto">
        <img src={this.props.src} />
      </div>
    );
  }
});

var Saves = React.createClass({
  handleSubmit: function(e) {
    //We prevent the default action of submitting the form
    //so we can stay on the page
    e.preventDefault();

    //We have to pass this up to the parent
    var isSaved = this.props.handleSave(this.props.id);
  },
  render: function() {
    var savedText = '';
    var submitText = 'Save';
    if (this.props.isSaved) {
      savedText = 'You have saved this home.';
      submitText = 'Remove';
    }

    return (
      <div className="saves">
        <form onSubmit={this.handleSubmit}>
          <input type="submit" value={submitText} />
        </form>
      {this.props.numSaves} saves. {savedText}
      </div>
    );
  }
});

React.render(
  <HomeListing url="homes.json" savesUrl="saves.json" pollInterval={10000} />,
  document.getElementById('content')
);
```

The first difference here is that React.render() calls on the HomeListing component using a few attributes. These attributes are not being used at the moment, because that would require the use of a server of some sort (see the requirements section above for links). You can easily move the homes data to a homes.json file, and the saves data to a saves.json file. Another option is to create some database calls to pull the data, and save the updated data.

You'll now see that the HomeListing component does all the work. The reason we have separated the saves information from the homes information is that we don't want to update changes to the homes in real time. Changes to home information just don't happen often enough to make it worth the cost in resources to update every single thing on the page.

Another change you would probably make when pulling "real" data from a database would be to include whatever key is used to save a home in the database. That key would be used when referencing the homes, as well as the save data for each home. For now, we're just counting on the JavaScript map method to give us array indexes that line up between the two lists of homes and saves data, simply because the arrays are in the correct order and the same size.

Here is the output of the new JavaScript file:



12345 Beverly Dr

This is a home in the city

52 saves.

98765 Tweety Ln

This is a home in the suburbs

123 saves. You have saved this home.

It’s pretty much the same as the old output, except that the interface now works properly! We still aren’t saving the data, but you’ll see that there is a comment in the toggleSave() function (which is a custom function), showing where you would save the new data to a database, or to a json file. We also make use of the React method componentDidMount(). This method is automatically called once when the component has finished rendering. This is the perfect place to load our information. It’s also the best place to put any polling function, such as the setInterval function, for updating any changes to the saves in real time.

You’ll also see that the custom load functions, which include JSON-style data, have a setState() call at the end. Even though we won’t be updating the home data in real time, setting it in a state makes it easier to populate the output with the correct data after the component is loaded. The reason we separate the saves into another state is so that you can comment out the setInterval() call to update only the save data in real time, and nothing else. The interval time can be sent through props, as it is in the call to HomeListing.

```
setInterval(this.loadSavesFromServer, this.props.pollInterval);
```

Another thing that can be confusing to people is that attributes are not automatically passed down to all the nested children when they are nested more than one level. You’ll see that the function attribute passed to Home must also be passed down to the Saves component, and it can have a completely different name! The same goes for all the other props. Here, we pass the props sent to the Home component down the next level to the Saves component:

```
<Saves
  id={this.props.id}
  handleSave={this.toggleSave}
  isSaved={this.props.isSaved}
  numSaves={this.props.numSaves}
></Saves>
```

Note that the toggleSave() function call could have been passed directly through as another props value, but we can also do as we did here and call a local function first, which then calls the parent function through props. That gives you the ability to make any additional changes.

So now that we have all that sorted out, you should know that each component should be separated out into separate JavaScript files, and organized into a “components” folder. This is for when you have a more complex design, and have a local or remote server running. Then, each component opens its children components using require().

Chapter 2: Form Validation Tutorial with React.JS

React.js is a fantastic user interface primarily library because the user's view updates automatically when a state changes. This ability to show changes to the user quickly also makes it a good fit for user-facing **form errors**. React allows you to easily display errors as the form is being filled so the user doesn't have to fill in the entire form, press the submit button, then try to figure out what they got wrong.

This does not mean React can be used to make a form secure. Hackers can bypass your React form altogether, passing variables directly to the server. You will still have to provide security and additional error-handling on the server side. Using React for error-checking is mainly for making a form easier to use.

We are going to create a simple donation form in our example. The code shows a few places where additional functionality can be added, but we are only going to focus on error-checking in this tutorial.

Download the source code

You may download all the files associated with this tutorial from here.

Tutorial requirements

- You must have beginner's knowledge of React.js, and at least a beginner's knowledge of JavaScript. If you need to know the basics, check out the previous chapter.
- You will need to download the React library in order to use React on a server and go beyond the testing phase. We show you how to get around this during testing.
- You will need to download a text editor of some sort. Notepad++ is popular on Windows, and TextMate is popular on Mac machines. An editor that highlights code is preferable.
- In order to use React as part of a full website, it will need to be incorporated into a larger application. When testing the basic code with external data files, you will need to use a local or remote web server in order to get the page to work. MAMP is a good one for the Mac, and WAMP is most the most common server used on Windows machines. For a complete solution, a Node.js server is one of the most popular servers to use with React. The React library download page (above) also includes a server and other options.

Getting started with the tutorial

There are all sorts of download options for React, but we're just going to use React from a CDN. React is available from more than one CDN, but here are the most popular links:

```
<!-- The core React library -->
<script src="https://fb.me/react-0.14.1.js"></script>
<!-- The ReactDOM Library -->
<script src="https://fb.me/react-dom-0.14.1.js"></script>
```

We are going to add a couple more JavaScript files to our index.html file:

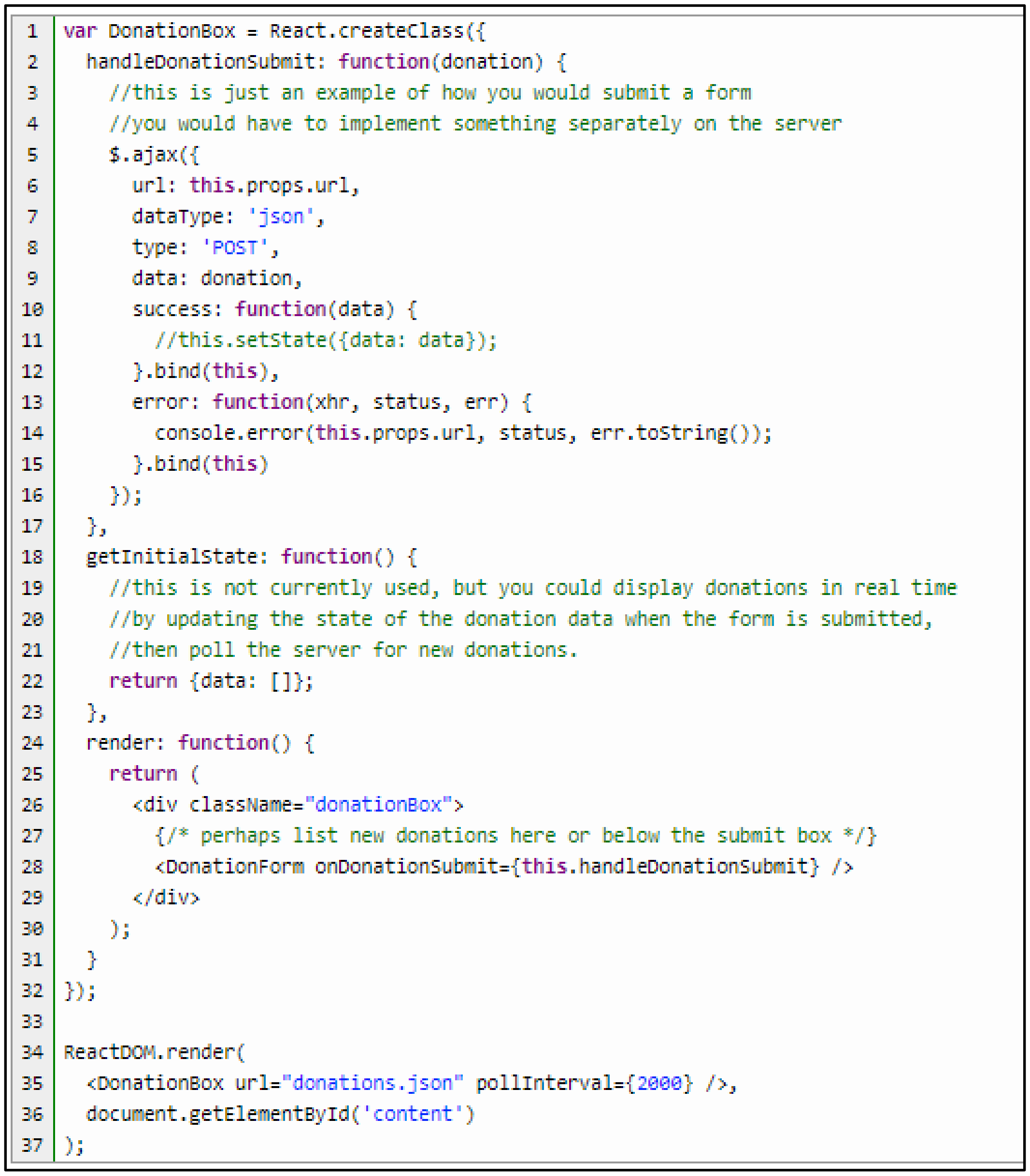
```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Game Development at ZENVA.com</title>
    <!-- Not present in the tutorial. Just for basic styling. -->
    <link rel="stylesheet" href="css/base.css" />
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/classnames/2.1.5/index.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel;harmony=true" src="scripts/example.js"></script>
  </body>
</html>
```

The babel-core browser.min.js file allows us to use JSX, which will greatly simplify our code. Please see the previous chapter for more on why React is better with JSX. Since this is a form, we are also including jquery, which will make form submission much easier. We don't actually make much use of it in the code, other than an example submission function. Lastly, we include the Classnames library, which is a teeny file that makes combining class names easier.

Finally, we call our example.js script, which is where all the action takes place. Just be sure to put "text/babel" in your script type so you can make use of JSX in your code.

Now let's dig into the code:

Setting up for form submission



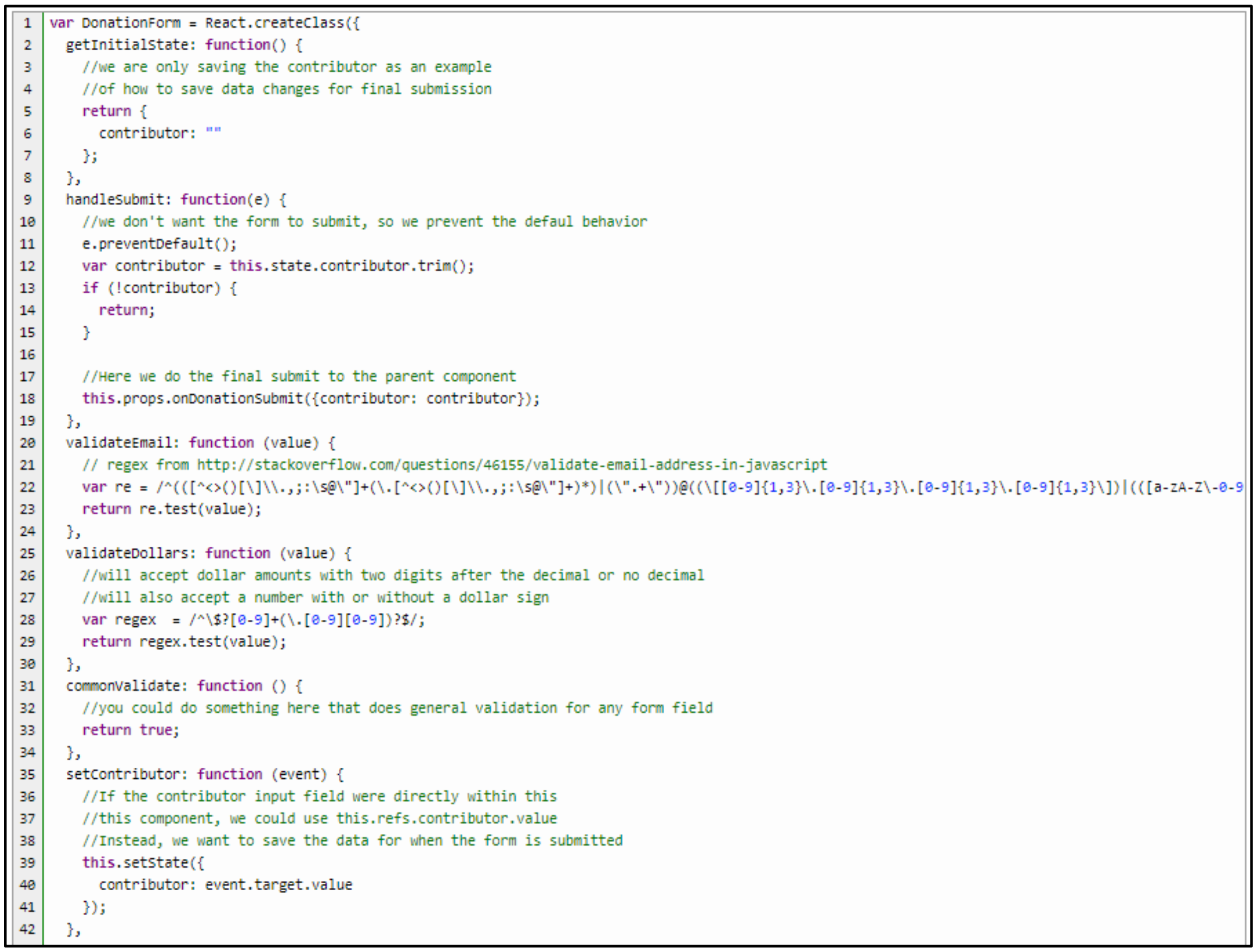
```
var DonationBox = React.createClass({
  handleDonationSubmit: function(donation) {
    //this is just an example of how you would submit a form
    //you would have to implement something separately on the server
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: donation,
      success: function(data) {
        //this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    //this is not currently used, but you could display donations in real time
    //by updating the state of the donation data when the form is submitted,
    //then poll the server for new donations.
    return {data: []};
  },
  render: function() {
    return (
      <div className="donationBox">
        {/* perhaps list new donations here or below the submit box */}
        <DonationForm onDonationSubmit={this.handleDonationSubmit} />
      </div>
    );
  }
});

ReactDOM.render(
  <DonationBox url="donations.json" pollInterval={2000} />,
  document.getElementById('content')
);
```

The ReactDOM.render call at the bottom of the code is the first call that starts the app. We use XML-style JSX to call the DonationBox component. This component doesn't do much at the moment, but provides a few examples of some of the additional things you can do with React besides error-checking. For example, you could show new donations as they are made, polling the server for new ones. I've also included some jquery ajax for an example of how the final form submission can be handled.

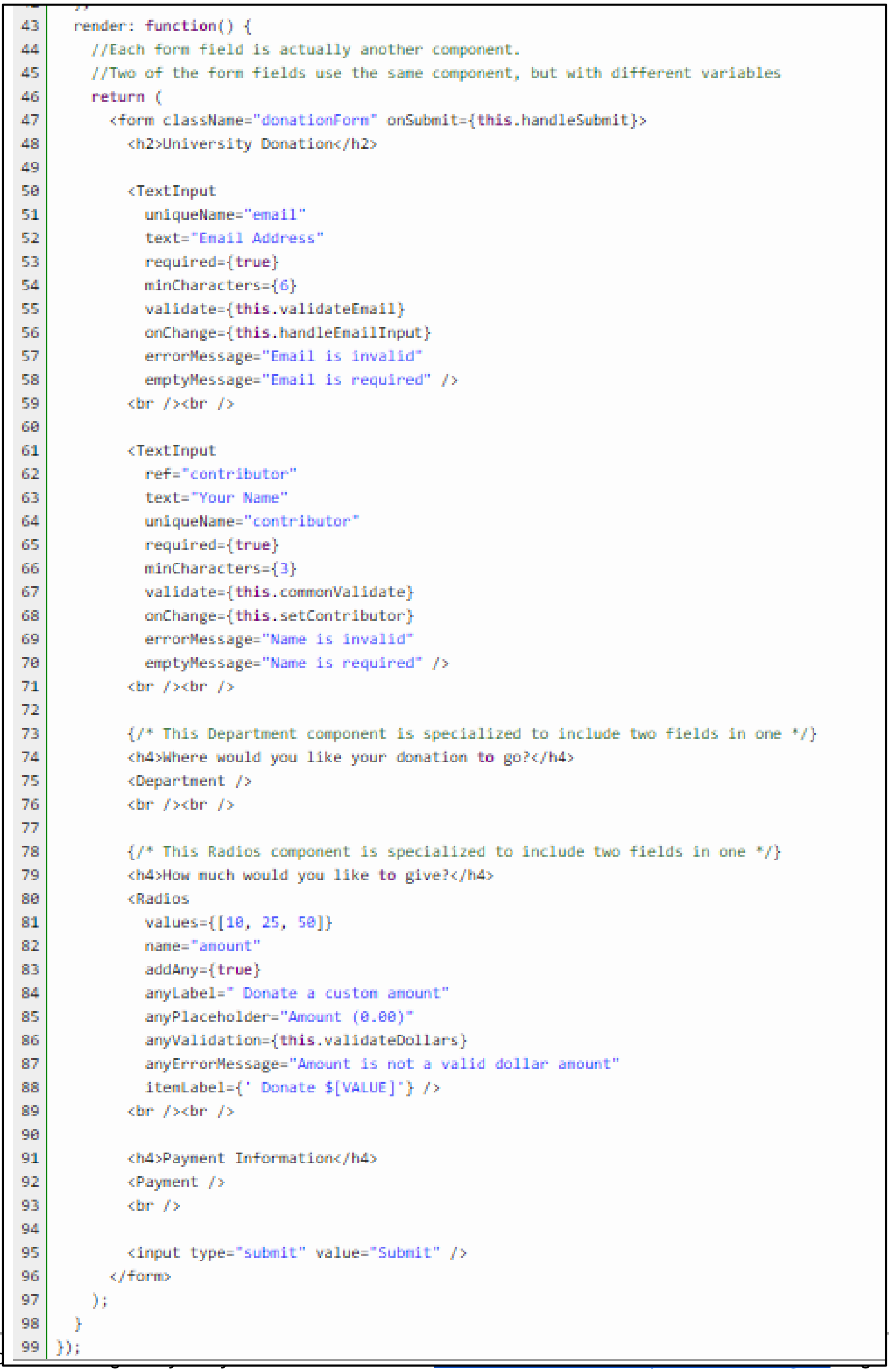
The real action starts in the DonationForm component:

Creating abstract form elements

The DonationForm component is our largest component, because this is where all the other form components are included, and where we do most of our form validation. Let’s take a look:



```
var DonationForm = React.createClass({
  getInitialState: function() {
    //we are only saving the contributor as an example
    //of how to save data changes for final submission
    return {
      contributor: ""
    };
  },
  handleSubmit: function(e) {
    //we don't want the form to submit, so we prevent the defaul behavior
    e.preventDefault();
    var contributor = this.state.contributor.trim();
    if (!contributor) {
      return;
    }

    //Here we do the final submit to the parent component
    this.props.onDonationSubmit({contributor: contributor});
  },
  validateEmail: function (value) {
    // regex from http://stackoverflow.com/questions/46155/validate-email-address-in-javascript
    var re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9
    return re.test(value);
  },
  validateDollars: function (value) {
    //will accept dollar amounts with two digits after the decimal or no decimal
    //will also accept a number with or without a dollar sign
    var regex  = /^\$?[0-9]+(\.[0-9][0-9])?$/;
    return regex.test(value);
  },
  commonValidate: function () {
    //you could do something here that does general validation for any form field
    return true;
  },
  setContributor: function (event) {
    //If the contributor input field were directly within this
    //this component, we could use this.refs.contributor.value
    //Instead, we want to save the data for when the form is submitted
    this.setState({
      contributor: event.target.value
    });
  },
```



```
  render: function() {
    //Each form field is actually another component.
    //Two of the form fields use the same component, but with different variables
    return (
      <form className="donationForm" onSubmit={this.handleSubmit}>
        <h2>University Donation</h2>

        <TextInput
          uniqueName="email"
          text="Email Address"
          required={true}
          minCharacters={6}
          validate={this.validateEmail}
          onChange={this.handleEmailInput}
          errorMessage="Email is invalid"
          emptyMessage="Email is required" />
        <br /><br />

        <TextInput
          ref="contributor"
          text="Your Name"
          uniqueName="contributor"
          required={true}
          minCharacters={3}
          validate={this.commonValidate}
          onChange={this.setContributor}
          errorMessage="Name is invalid"
          emptyMessage="Name is required" />
        <br /><br />

        {/* This Department component is specialized to include two fields in one */}
        <h4>Where would you like your donation to go?</h4>
        <Department />
        <br /><br />

        {/* This Radios component is specialized to include two fields in one */}
        <h4>How much would you like to give?</h4>
        <Radios
          values={[10, 25, 50]}
          name="amount"
          addAny={true}
          anyLabel=" Donate a custom amount"
          anyPlaceholder="Amount (0.00)"
          anyValidation={this.validateDollars}
          anyErrorMessage="Amount is not a valid dollar amount"
          itemLabel={' Donate $[VALUE]'} />
        <br /><br />

        <h4>Payment Information</h4>
        <Payment />
        <br />

        <input type="submit" value="Submit" />
      </form>
    );
  }
});
```

This b from
zero to Full-Stack engineer.

We save one piece of information (the contributor) for passing to the parent and saving on the server, just as an example. The handleSubmit method shows how you would pass the variable to the parent element when the form is submitted. What we want to focus on, though, is the validation functions and component calls. I decided to include form elements in a rather interesting way to show the power of React. The best examples can be seen in the calls to the TextInput and Radios components.

As you can see, there are two calls to the TextInput component, but with different variables and validation functions included in the attributes (available as props in the child components). We do this because the text input is a reusable component. All you have to do is enter different attributes depending on the results you would like to see. You could even add an attribute that gives a different error message depending on whether the field contains a number or dollar amount (we show an example of this on a different field).

Each TextInput component gets its own validation function, which can be accessed from the component using this.props.validate(value). The component itself doesn't care what type of validation is going on. It simply calls validate, and the parent component takes care of which validation function is being called.

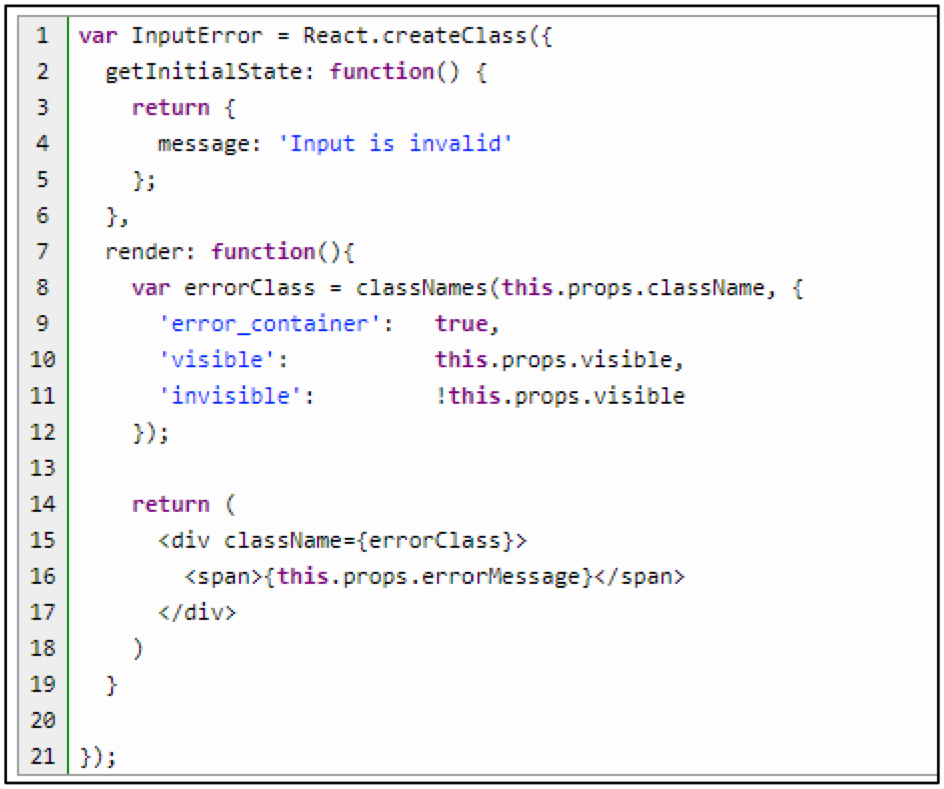
I've also included a commonValidate function as an example of how you could do some basic validation on all form fields. In this case, we use it with the second TextInput component and return true, because we need the validate prop function to exist, but we don't actually want to validate the second field.

The Radios component is interesting because we are actually passing all of the possible values through in a simple array. We also have an optional text field for adding user-generated text, which has its own validation.

The rest of the components are specific to this donation form, but are separated into new components in order to simplify the DownationForm component code. Let's take a deeper look at those now.

Creating input fields

One component that will be reused in every form field component is an error message. If we want to validate fields as the user enters them, we need to be able to bring up an error message as they are typing. So, let's start with that:

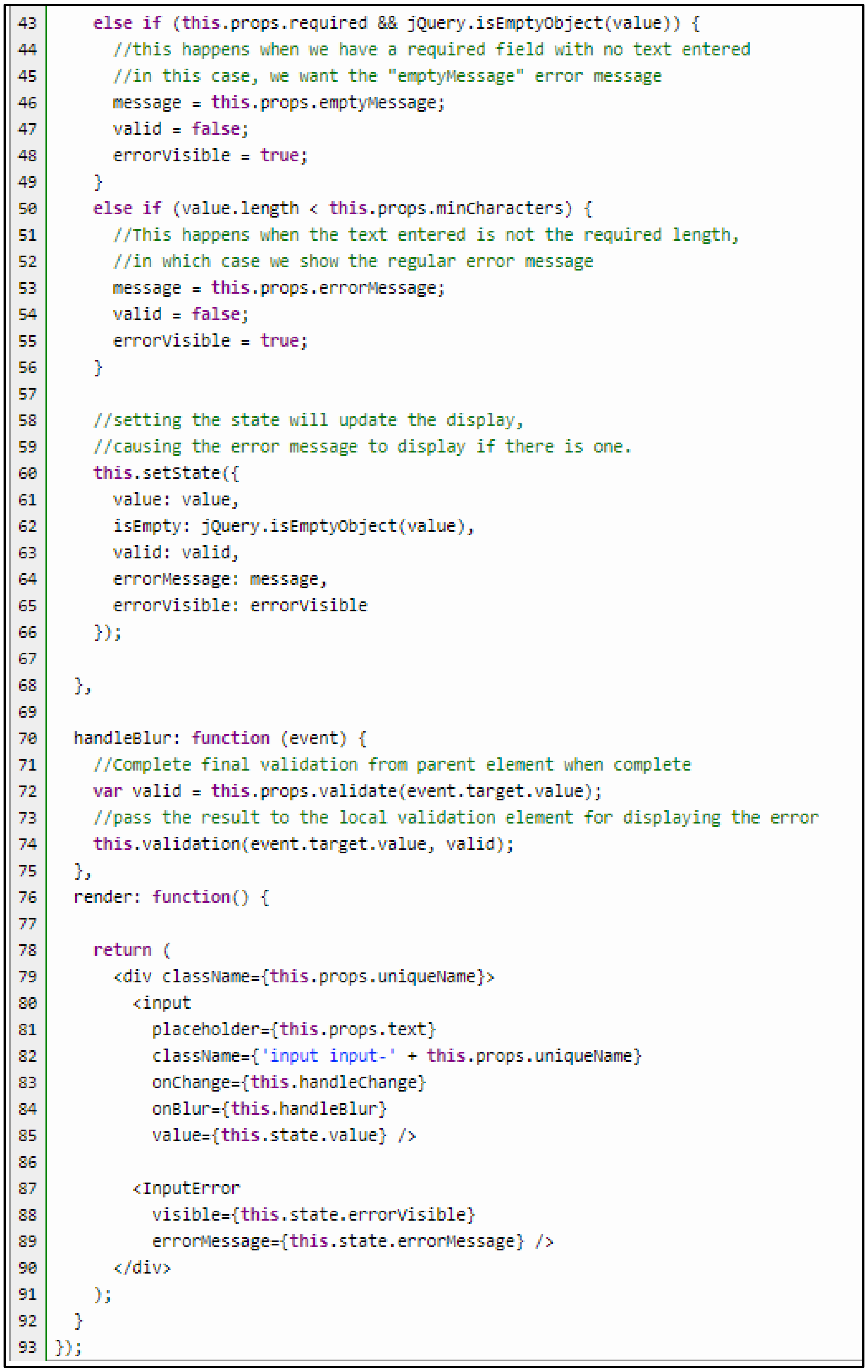


```
var InputError = React.createClass({
  getInitialState: function() {
    return {
      message: 'Input is invalid'
    };
  },
  render: function(){
    var errorClass = classNames(this.props.className, {
      'error_container':   true,
      'visible':           this.props.visible,
      'invisible':         !this.props.visible
    });

    return (
      <div className={errorClass}>
        <span>{this.props.errorMessage}</span>
      </div>
    )
  }

});
```

This component is very small, but powerful. All we have to do when calling this component is include an error message, and a boolean value that tells the component whether it should be displayed or not. The css file (included in the download) will do the rest.

Now let's take a look at all of our form field components:

```javascript
var TextInput = React.createClass({
  getInitialState: function(){
    //most of these variables have to do with handling errors
    return {
      isEmpty: true,
      value: null,
      valid: false,
      errorMessage: "Input is invalid",
      errorVisible: false
    };
  },

  handleChange: function(event){
    //validate the field locally
    this.validation(event.target.value);

    //Call onChange method on the parent component for updating it's state
    //If saving this field for final form submission, it gets passed
    // up to the top component for sending to the server
    if(this.props.onChange) {
      this.props.onChange(event);
    }
  },

  validation: function (value, valid) {
    //The valid variable is optional, and true if not passed in:
    if (typeof valid === 'undefined') {
      valid = true;
    }

    var message = "";
    var errorVisible = false;

    //we know how to validate text fields based on information passed through props
    if (!valid) {
      //This happens when the user leaves the field, but it is not valid
      //(we do final validation in the parent component, then pass the result
      //here for display)
      message = this.props.errorMessage;
      valid = false;
      errorVisible = true;
    }
```

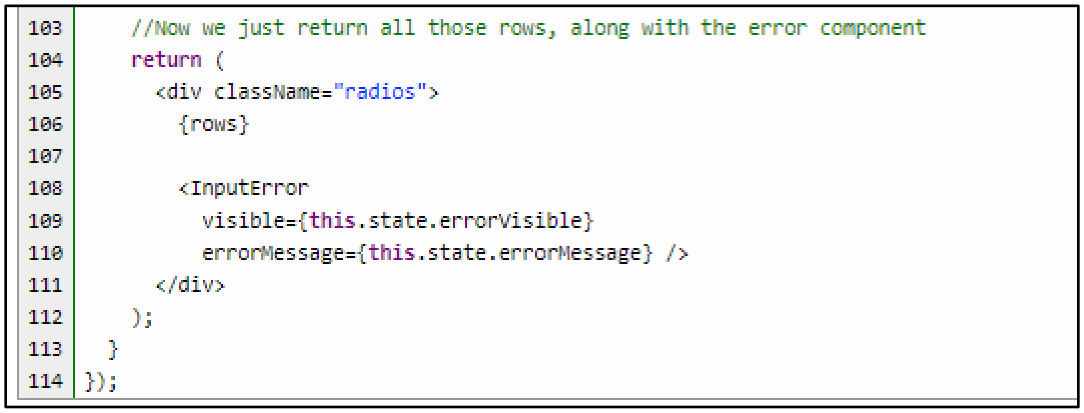


```
    else if (this.props.required && jQuery.isEmptyObject(value)) {
      //this happens when we have a required field with no text entered
      //in this case, we want the "emptyMessage" error message
      message = this.props.emptyMessage;
      valid = false;
      errorVisible = true;
    }
    else if (value.length < this.props.minCharacters) {
      //This happens when the text entered is not the required length,
      //in which case we show the regular error message
      message = this.props.errorMessage;
      valid = false;
      errorVisible = true;
    }

    //setting the state will update the display,
    //causing the error message to display if there is one.
    this.setState({
      value: value,
      isEmpty: jQuery.isEmptyObject(value),
      valid: valid,
      errorMessage: message,
      errorVisible: errorVisible
    });

  },

  handleBlur: function (event) {
    //Complete final validation from parent element when complete
    var valid = this.props.validate(event.target.value);
    //pass the result to the local validation element for displaying the error
    this.validation(event.target.value, valid);
  },
  render: function() {

    return (
      <div className={this.props.uniqueName}>
        <input
          placeholder={this.props.text}
          className={'input input-' + this.props.uniqueName}
          onChange={this.handleChange}
          onBlur={this.handleBlur}
          value={this.state.value} />

        <InputError
          visible={this.state.errorVisible}
          errorMessage={this.state.errorMessage} />
      </div>
    );
  }
});
```

The first one is our InputText component. This component includes some validation based on the props sent from the parent. In this case, we show an error message when there are not enough characters, but we show a different message when the field is empty. Both messages were sent as props. All of this validation occurs while the user is typing. This could be annoying for some fields, but it’s a great example of what is possible.

In addition to the local validation, we also call the parent validation function when the user leaves the field (indicating they are finished with it). You could also do all validation in the parent, or just do local validation. There are many options.

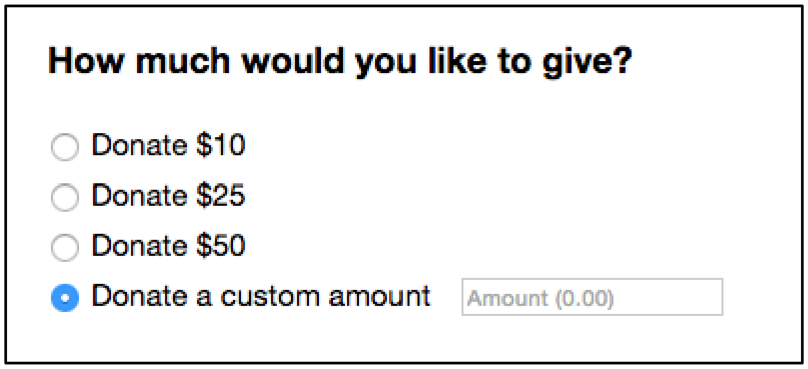
```
var Radios = React.createClass({
  getInitialState: function() {
    //displayClass is the class we use for displaying or hiding
    //the optional "any value" text field
    return {
      displayClass: 'invisible',
      valid: false,
      errorMessage: "Input is invalid",
      errorVisible: false
    };
  },
  handleClick: function(displayClass, e) {
    //if we click any option other than the "any value" option,
    //we hide the "any value" text field. Otherwise, show it
    if (displayClass == 'invisible') {
      this.setState(
        {
          displayClass: displayClass,
          errorVisible: false
        }
      );
    }
    else {
      this.setState({displayClass: displayClass});
    }
  },
  handleAnyChange: function(e) {
    //this validation is specifically for the optional "any value" text field
    //Since we have no idea what the requirements are locally, we call the parent
    //validation function, then set the error states accordingly
    if (this.props.anyValidation(e.target.value)) {
      this.setState(
        {
          valid: true,
          errorMessage: "Input is invalid",
          errorVisible: false
        }
      );
    }
    else {
      this.setState(
        {
          valid: false,
          errorMessage: this.props.anyErrorMessage,
          errorVisible: true
        }
      );
    }
  },
  render: function() {
    var rows = [];
    var label = "";

    //we have passed in all the options for the radios, so we traverse the array
    for (var i = 0; i < this.props.values.length; i++) {
      //We do this little replace for when we want to display the value as part of
      //additional text. Otherwise, we would just put '[VALUE]' when passing
      //the itemLabel prop from the parent component, or leave out '[VALUE]' entirely
      label = this.props.itemLabel.replace('[VALUE]', this.props.values[i]);

      //You'll see that even the <br /> field has a key. React will give you errors
      //if you don't do this. This is just an axample of what's possible, but
      //you would normally add extra spacing with css
      rows.push(<input
        key={this.props.name + '-' + i}
        type="radio"
        ref={this.props.name + '-' + this.props.values[i]}
        name={this.props.name}
        value={this.props.values[i]}
        onClick={this.handleClick.bind(this, 'invisible')} />,

        <label key={this.props.name + '-label-' + i} htmlFor={this.props.values[i]}>{label}</label>,

        <br key={this.props.name + '-br-' + i} />);
    }

    //The "any value" field complicates things a bit
    if (this.props.addAny) {
      //We passed in a separate label just for the option that
      //activates the "any value" text field
      label = this.props.anyLabel;
      rows.push(<input
        key={this.props.name + '-' + i}
        type="radio"
        ref={this.props.name + '-any'}
        name={this.props.name} value="any"
        onClick={this.handleClick.bind(this, 'visible')} />,

        <label key={this.props.name + '-label-' + i} htmlFor={this.props.values[i]}>{label}</label>);

      //and now we add the "any value text field, with all its special variables"
      rows.push(<div key={this.props.name + '-div-' + (i+2)} className={this.state.displayClass}>
        <input
          className="anyValue"
          key={this.props.name + '-' + (i+1)}
          type="text"
          placeholder={this.props.anyPlaceholder}
          onChange={this.handleAnyChange}
          ref={this.props.name} />
      </div>);
    }
```



```
    //Now we just return all those rows, along with the error component
    return (
      <div className="radios">
        {rows}

        <InputError
          visible={this.state.errorVisible}
          errorMessage={this.state.errorMessage} />
      </div>
    );
  }
});
```

Next up is our Radios component. This component actually doesn't have any validation unless the optional "addAny" prop is set to true. In that case, an extra radio button is added which will display an "anyValue" text field when selected. This text field gets its own validation function, called through the props sent from the parent.

We also have to handle the appearing and disappearing act of the text field. When the "addAny" radio button is clicked, the text field is displayed. When any other option is selected, it's hidden. We do this with an onClick attribute, but we have to use "bind" in order to send a variable to our handler function. The variable tells the function whether to show or hide the text field.



The validation handler for the text field simply calls the parent validation field, and uses that result to determine whether to show the InputError component.

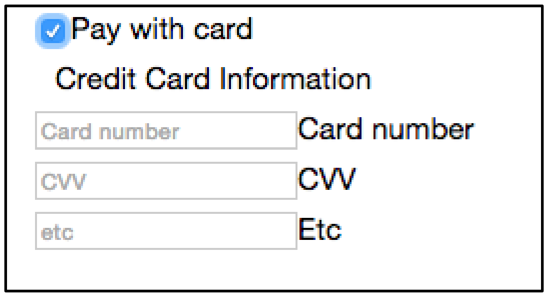
You'll notice that some of the code looks a bit wonky because there are keys for even the <label> and <br /> tags. This is to prevent errors, because React requires a key for all fields. Since we are sending all the values to the Radios component, we have to traverse a for loop of values. JSX can't just be placed raw into the for loop, so we push each row of JSX to an array. When we do it this way, the keys aren't automatically included, so we have to include them in every single tag.

Now let’s take a look at a couple of custom form field components.

```
var Payment = React.createClass({
  //we have no error checking for this one, so there are no error states
  getInitialState: function() {
    return {
      displayClass: 'invisible'
    };
  },
  handleClick: function(displayClass, e) {
    //we simply set the state in order to update the display when
    //we want to show the extra options
    this.setState({displayClass: displayClass});
  },
  render: function() {
    //we take full control over the checkbox that allows us to show additional options
    //this will ensure that we truly toggle the options, and don't wind up with a case
    //where the checkbox is not checked but the extra options show and vice versa
    var optionsClass = "invisible";
    var isChecked = false;
    if (this.state.displayClass == 'invisible') {
      optionsClass = "visible";
    }
    else {
      isChecked = true;
    }

    //We could have extra checkboxes, but this is just to show how to properly show other options
    //when a checkbox is checked. We won't do error checking on the payment info here.
    return (
      <div className="payment">
        <a href="#">PayPal button goes here</a>
        <br />
        <input type="checkbox" checked={isChecked} onChange={this.handleClick.bind(this, optionsClass)} name="card" />Pay with card<br />
        <div id="Choices" className={this.state.displayClass}>Credit Card Information<br />
            <input type="text" placeholder="Card number" ref="card" />Card number<br />
            <input type="text" placeholder="CVV" ref="cvv" />CVV<br />
            <input type="text" placeholder="etc" ref="whatever" />Etc<br />
        </div>
        <InputError
          visible={this.state.errorVisible}
          errorMessage={this.state.errorMessage} />
      </div>
    );
  }
});
```

```
var Department = React.createClass({
  getInitialState: function() {
    return {
      displayClass: 'invisible'
    };
  },
  handleClick: function(e) {
    //We're doing another one of these "any value" fields, only shown when
    //a specific "other" option is chosen
    var displayClass = 'invisible';
    if (e.target.value == 'other') {
      displayClass = 'visible';
    }
    this.setState({displayClass: displayClass});
  },
  render: function() {
    //This is a select field with options and sub-options, plus an "any value" field
    return (
      <div className="department">
        <select onChange={this.handleClick} multiple={false} ref="department">
          <option value="none"></option>
          <optgroup label="College">
            <option value="muir">Muir</option>
            <option value="revelle">Revelle</option>
            <option value="sixth">Sixth</option>
          </optgroup>
          <optgroup label="School">
            <option value="jacobs">Jacobs School of Engineering</option>
            <option value="global">School of Global Policy and Strategy</option>
            <option value="medicine">School of Medicine</option>
          </optgroup>
          <option value="scholarships">Scholarships</option>
          <option value="other">Other</option>
        </select>
        <div className={this.state.displayClass}>
          <input className="anyValue" type="text" placeholder="Department" ref="any-department" />
        </div>

        <InputError
          visible={this.state.errorVisible}
          errorMessage={this.state.errorMessage} />
      </div>
    );
  }
});
```

Payment validation is beyond the scope of this tutorial, but we've included a payment section since donations need a way to donate money! Our Payment component also shows a common example of how payments are accepted. There may be some sort of PayPal button, then an option to fill in payment information directly. This is just a simple example of showing a new section of information when a checkbox is clicked.



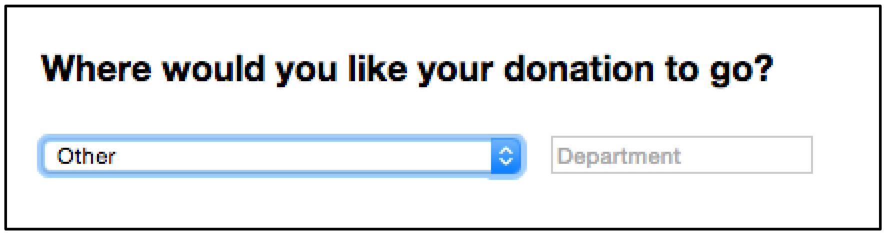
✓ Pay with card

Credit Card Information

Card number	Card number
CVV	CVV
etc	Etc

One thing that can be annoying about checkbox toggling is that it can get out of sync. In order to prevent that, we take full control over it. When the credit card fields are toggled off, we ensure that the checkbox is not checked. When they are on, the checkbox is checked.

In the Department component, we have included select options with sub-options, plus an option to put any value. This is somewhat similar to what we've done in the previous examples of showing and hiding other fields. The main reason it's included here is to show how we handle this action in every type of field that gives multiple options. It's also a good example of how to do sub-options.



Where would you like your donation to go?

Other | Department

One other thing to note about the select field is the multiple={false} attribute. That's not required, but it's included here in order to show that it's possible to have a select field that takes multiple values. When the attribute is set to true, all of the options will be shown in a box rather than appearing as a drop-down. The returning value is then an array.

Chapter 3: How to Submit Forms and Save Data with React.js and Node.js

In the previous two chapters, I covered the basics of React.js and form error handling in React.js. In each case, I used hard-coded data. React is a front-end library, providing no easy way to save to, or read information from, a database or file. That is the part we will cover in this chapter.

React is a user interface library, so it is common to use a different library or framework to do all the back-end work. One of the most common pairings is React.js with Node.js. Lucky for us, we already have a tutorial on the basics of Node easily accessible on HTML5 Hive!

The main React Facebook tutorial does something similar to what we will be doing with this tutorial, and they also provide a Node server, as well as many other server examples within their download files. Although our Node server will be similar, our front-end React interface will have a few extra complications. In fact, I'll be using the same files from Chapter 2, with a few updates that will allow the saving and reading of information from a file or database.

Download the tutorial files

You can download all the files used in this tutorial from here.

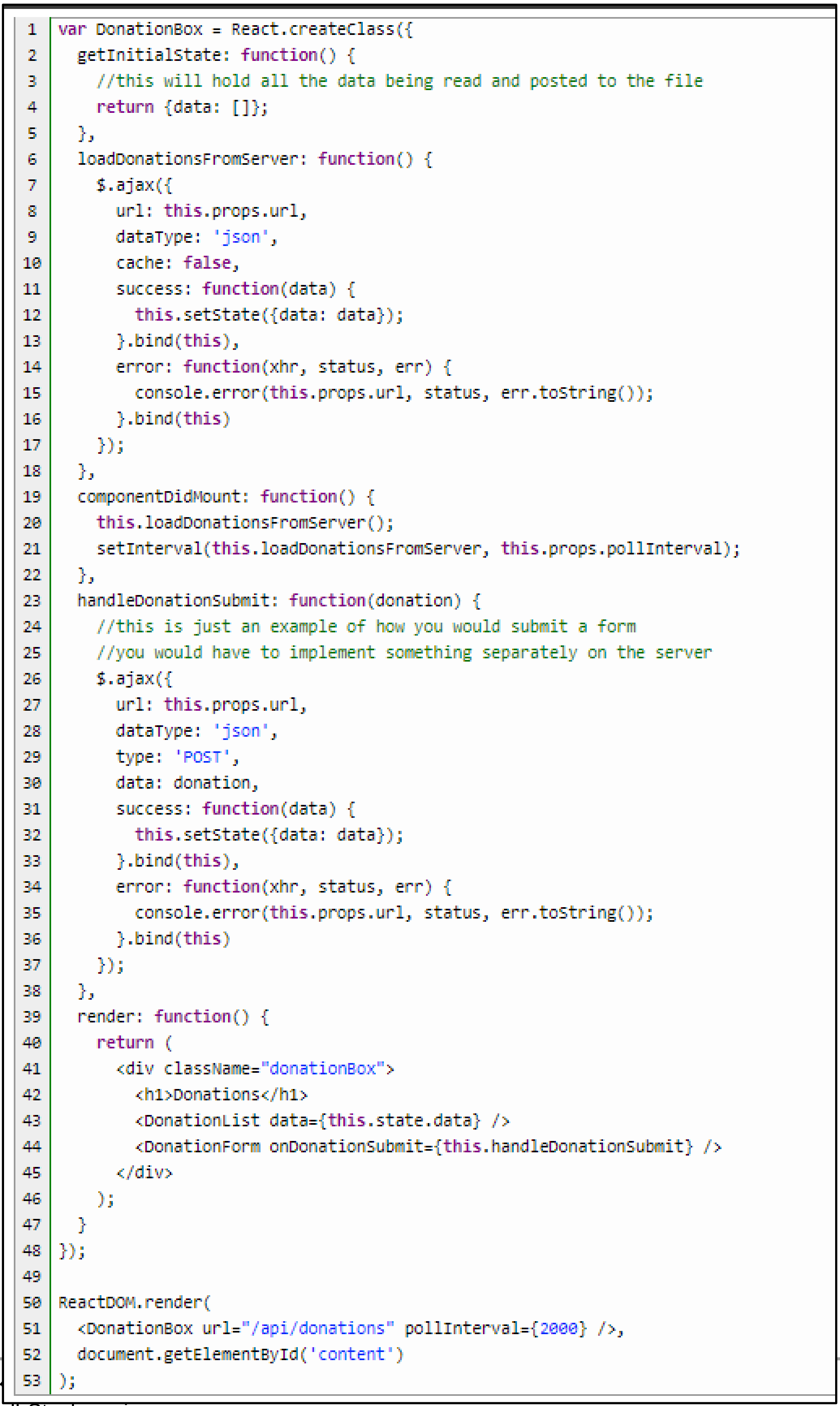
Tutorial requirements

- You must have basic knowledge of React.js and JavaScript. For a beginner's guide to React, please see Chapter 1. In addition, I will be using many of the same functions from Chapter 2, so you may want to read that one as well.
- You will need to use the React library, although we will be using CDN links to get around that during testing.
- You will need to download a text editor of some sort. You can use just a plain text editor, although Notepad++ is popular on Windows, and TextMate is popular on Mac machines. An editor with code highlighting capability is preferable.
- You will need to read and follow the directions in most of Chapter 1 in order to create the server that will be used later in this tutorial.

Making revisions to a React user interface

I will be using the same files I created for Chapter 2, with some simple revisions that will make it possible to read and save data. The files with all the revisions are provided above. The index file from the previous tutorial remains the same, but now we are going to be pulling data from a file, then posting new form entries to the file.

The original file was set up with some of the code needed to post and read data, but some parts were commented out, since we had no server to do the work. All we have to do is uncomment those lines, and we have this:



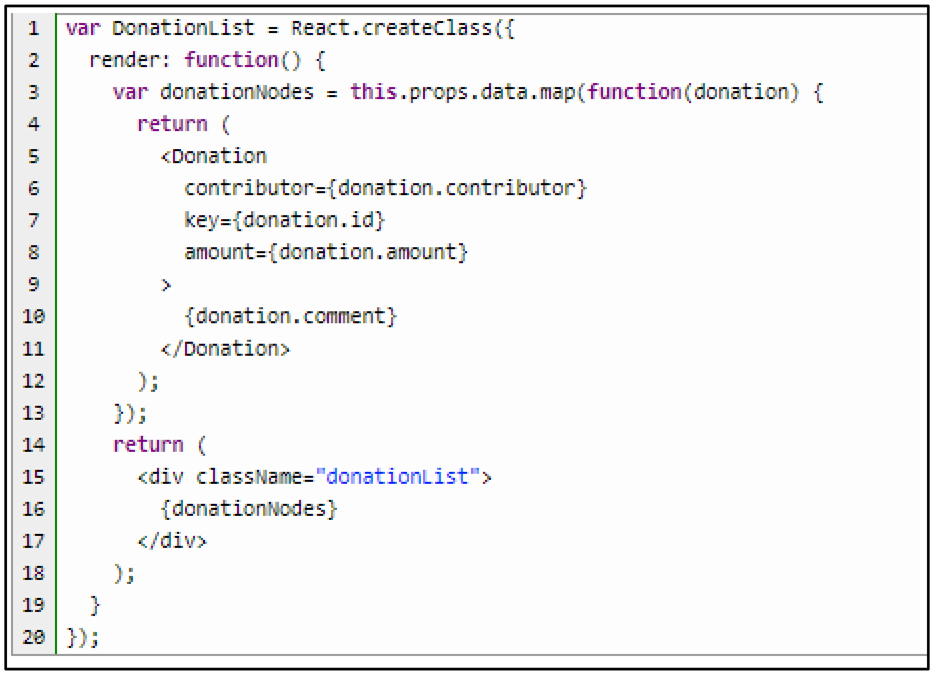
```
var DonationBox = React.createClass({
  getInitialState: function() {
    //this will hold all the data being read and posted to the file
    return {data: []};
  },
  loadDonationsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  componentDidMount: function() {
    this.loadDonationsFromServer();
    setInterval(this.loadDonationsFromServer, this.props.pollInterval);
  },
  handleDonationSubmit: function(donation) {
    //this is just an example of how you would submit a form
    //you would have to implement something separately on the server
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: donation,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  render: function() {
    return (
      <div className="donationBox">
        <h1>Donations</h1>
        <DonationList data={this.state.data} />
        <DonationForm onDonationSubmit={this.handleDonationSubmit} />
      </div>
    );
  }
});

ReactDOM.render(
  <DonationBox url="/api/donations" pollInterval={2000} />,
  document.getElementById('content')
);
```

So far, everything we have here is similar to Facebook's Commenting tutorial. We save all the data from the file in a DonationBox component state, then set an interval to pull new donations from the server so the user can see new donations as they come in, in close to real time. On each form submission, the data is pushed to the database (or in this case, the file). We include JQuery in the index file in order to make the loading and submitting of data easier.

Let's continue with the changes I've made to the form before discussing the server side.

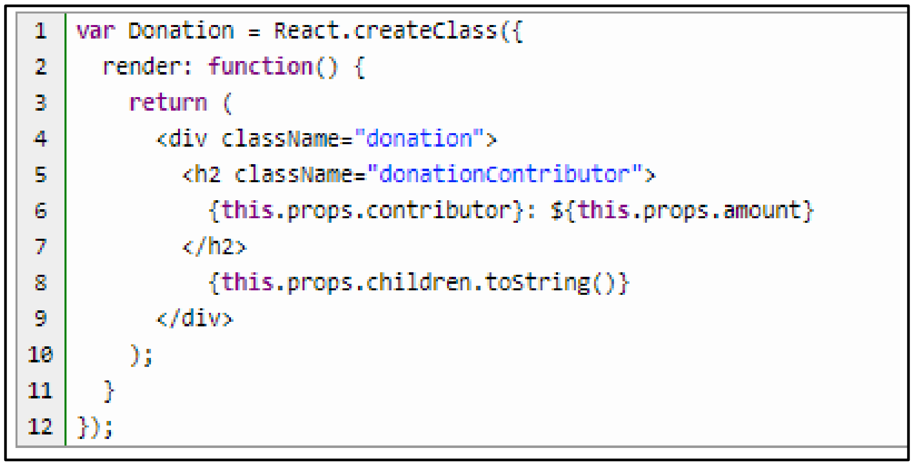
Displaying new data from everyone

We now have a new listing that will display all new donations coming from all users.



```
var DonationList = React.createClass({
  render: function() {
    var donationNodes = this.props.data.map(function(donation) {
      return (
        <Donation
          contributor={donation.contributor}
          key={donation.id}
          amount={donation.amount}
        >
          {donation.comment}
        </Donation>
      );
    });
    return (
      <div className="donationList">
        {donationNodes}
      </div>
    );
  }
});
```

Similar to the Facebook tutorial, we are mapping out all the data from the JSON file into a new Donation component for each entry. We then display the entire donationNodes list. Let's take a look at the new Donation component:



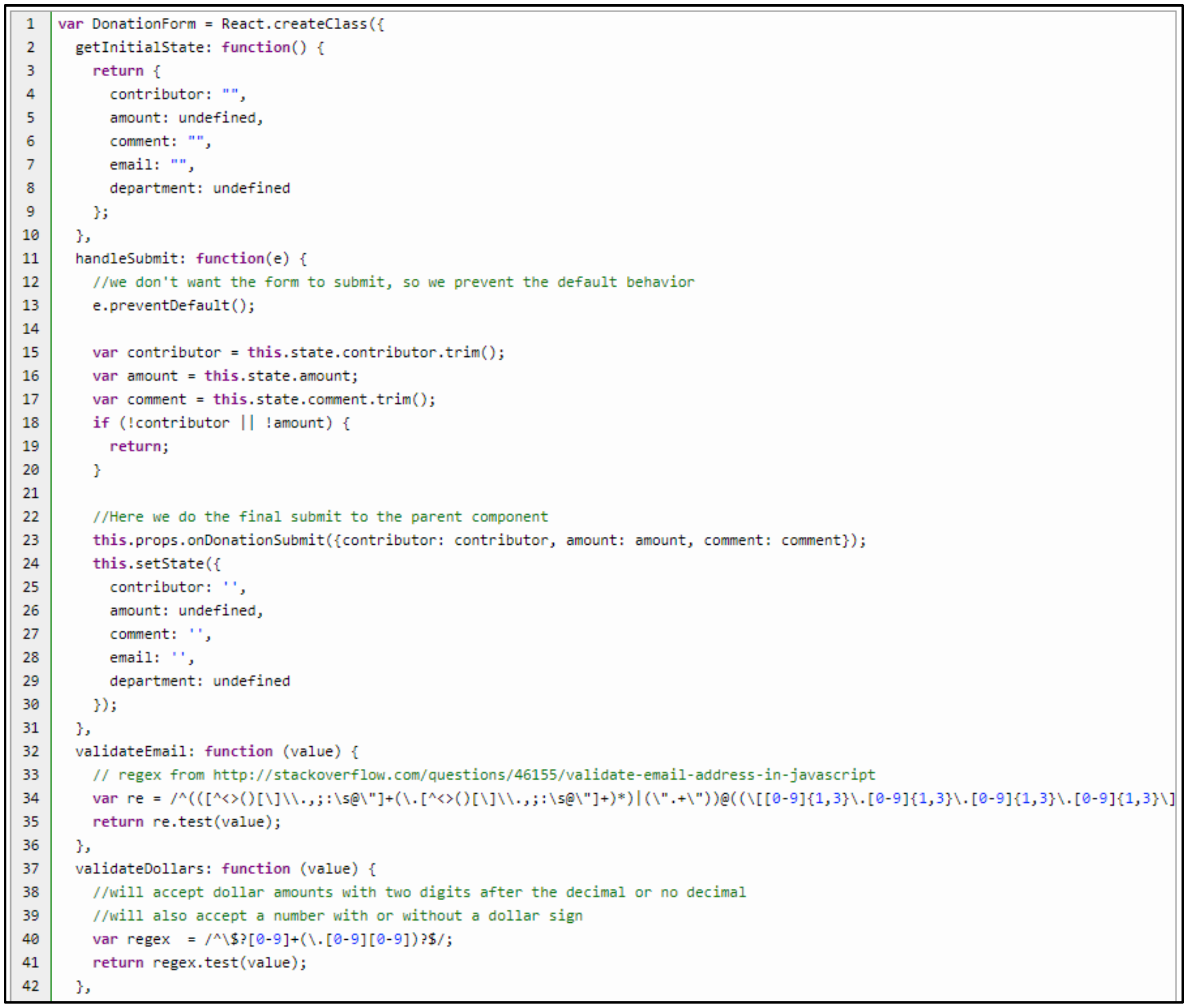
```
var Donation = React.createClass({
  render: function() {
    return (
      <div className="donation">
        <h2 className="donationContributor">
          {this.props.contributor}: ${this.props.amount}
        </h2>
          {this.props.children.toString()}
      </div>
    );
  }
});
```

This is actually simpler than what we see in the Facebook tutorial, because we are expecting comments by contributors to be in plain text. We are simply displaying the main donation information, leaving out anything private. In fact, we are only saving the public information for the purposes of this tutorial, but it's easy enough to store all of the data in a database, then only display the information that is public.

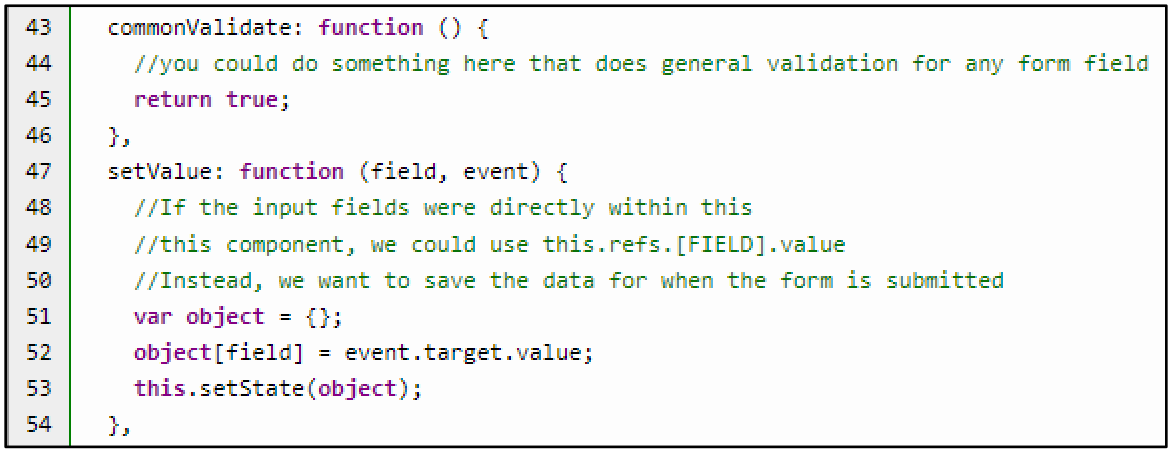
So far, our additions are very similar to what you can find in the Facebook tutorial, but our original Donation app is nothing like that tutorial. This is where the changes become a little more complicated.

Submitting form data

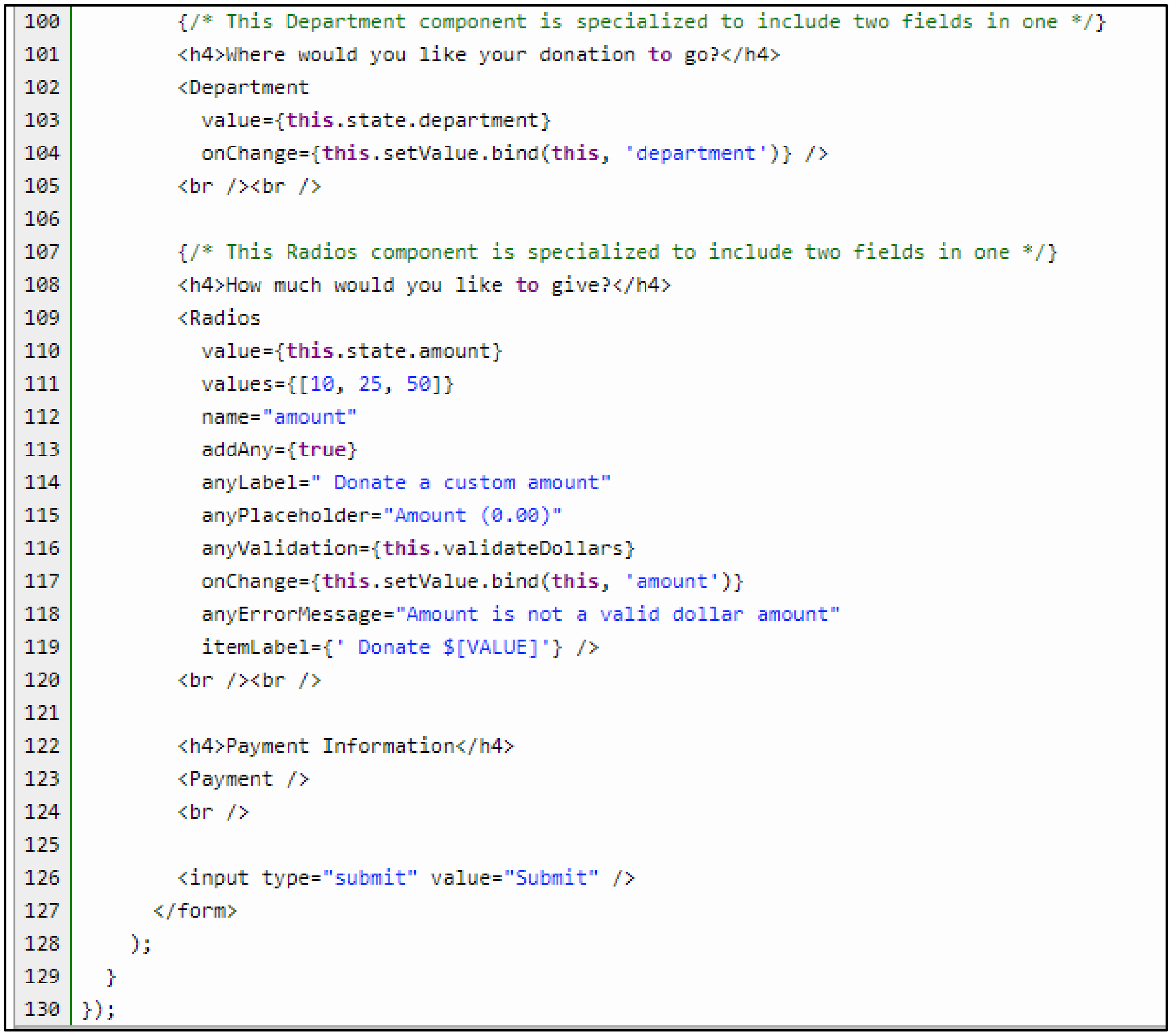
In the original donation app, we had some fairly complicated calls to other components that allowed us to serve up multiple fields from a single component. Now that we have to think about actually saving and displaying that data, we have to add a value attribute to the components that will allow us to empty all the fields after submission, and we also have to save all the data in a state.



```
var DonationForm = React.createClass({
  getInitialState: function() {
    return {
      contributor: "",
      amount: undefined,
      comment: "",
      email: "",
      department: undefined
    };
  },
  handleSubmit: function(e) {
    //we don't want the form to submit, so we prevent the default behavior
    e.preventDefault();

    var contributor = this.state.contributor.trim();
    var amount = this.state.amount;
    var comment = this.state.comment.trim();
    if (!contributor || !amount) {
      return;
    }

    //Here we do the final submit to the parent component
    this.props.onDonationSubmit({contributor: contributor, amount: amount, comment: comment});
    this.setState({
      contributor: '',
      amount: undefined,
      comment: '',
      email: '',
      department: undefined
    });
  },
  validateEmail: function (value) {
    // regex from http://stackoverflow.com/questions/46155/validate-email-address-in-javascript
    var re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\]
    return re.test(value);
  },
  validateDollars: function (value) {
    //will accept dollar amounts with two digits after the decimal or no decimal
    //will also accept a number with or without a dollar sign
    var regex  = /^\$?[0-9]+(\.[0-9][0-9])?$/;
    return regex.test(value);
  },
```



```
  commonValidate: function () {
    //you could do something here that does general validation for any form field
    return true;
  },
  setValue: function (field, event) {
    //If the input fields were directly within this
    //this component, we could use this.refs.[FIELD].value
    //Instead, we want to save the data for when the form is submitted
    var object = {};
    object[field] = event.target.value;
    this.setState(object);
  },
  render: function() {
    //Each form field is actually another component.
    //Two of the form fields use the same component, but with different variables
    return (
      <form className="donationForm" onSubmit={this.handleSubmit}>
        <h2>University Donation</h2>

        <TextInput
          value={this.state.email}
          uniqueName="email"
          text="Email Address"
          textArea={false}
          required={true}
          minCharacters={6}
          validate={this.validateEmail}
          onChange={this.setValue.bind(this, 'email')}
          errorMessage="Email is invalid"
          emptyMessage="Email is required" />
        <br /><br />

        <TextInput
          value={this.state.contributor}
          uniqueName="contributor"
          text="Your Name"
          textArea={false}
          required={true}
          minCharacters={3}
          validate={this.commonValidate}
          onChange={this.setValue.bind(this, 'contributor')}
          errorMessage="Name is invalid"
          emptyMessage="Name is required" />
        <br /><br />

        <TextInput
          value={this.state.comment}
          uniqueName="comment"
          text="Is there anything you'd like to say?"
          textArea={true}
          required={false}
          validate={this.commonValidate}
          onChange={this.setValue.bind(this, 'comment')}
          errorMessage=""
          emptyMessage="" />
        <br /><br />
```



```
        {/* This Department component is specialized to include two fields in one */}
        <h4>Where would you like your donation to go?</h4>
        <Department
          value={this.state.department}
          onChange={this.setValue.bind(this, 'department')} />
        <br /><br />

        {/* This Radios component is specialized to include two fields in one */}
        <h4>How much would you like to give?</h4>
        <Radios
          value={this.state.amount}
          values={[10, 25, 50]}
          name="amount"
          addAny={true}
          anyLabel=" Donate a custom amount"
          anyPlaceholder="Amount (0.00)"
          anyValidation={this.validateDollars}
          onChange={this.setValue.bind(this, 'amount')}
          anyErrorMessage="Amount is not a valid dollar amount"
          itemLabel={' Donate $[VALUE]'} />
        <br /><br />

        <h4>Payment Information</h4>
        <Payment />
        <br />

        <input type="submit" value="Submit" />
      </form>
    );
  }
});
```

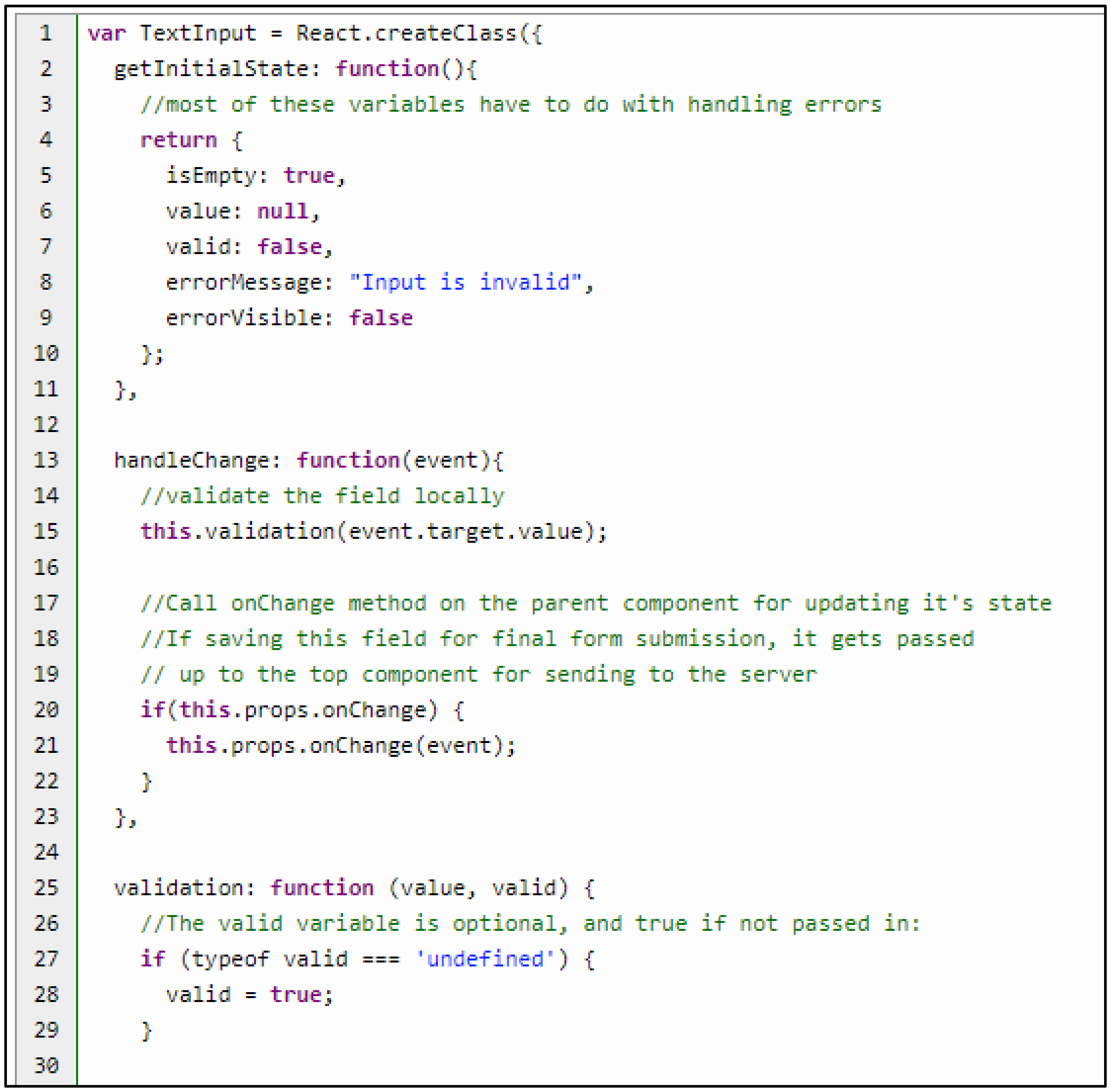
As you can see, we've added new states for each piece of data we plan on saving. The other bits of data can always be added later, but we are just going to focus on the data that we want to display. On submit, we send all the data to the parent component to do the actual saving to the server. We then reset all the fields so they will appear empty after submission.

We also have a new method called setValue. Because we do not have form fields directly within this component, we have to have a way to set the state as we go (using the onChange attribute in each component call). Rather than create a new function for every form field, we use this single function by taking the field as a variable, and using it to create a new object that can be used to set the state for that field. The field name is included in the call to the component.

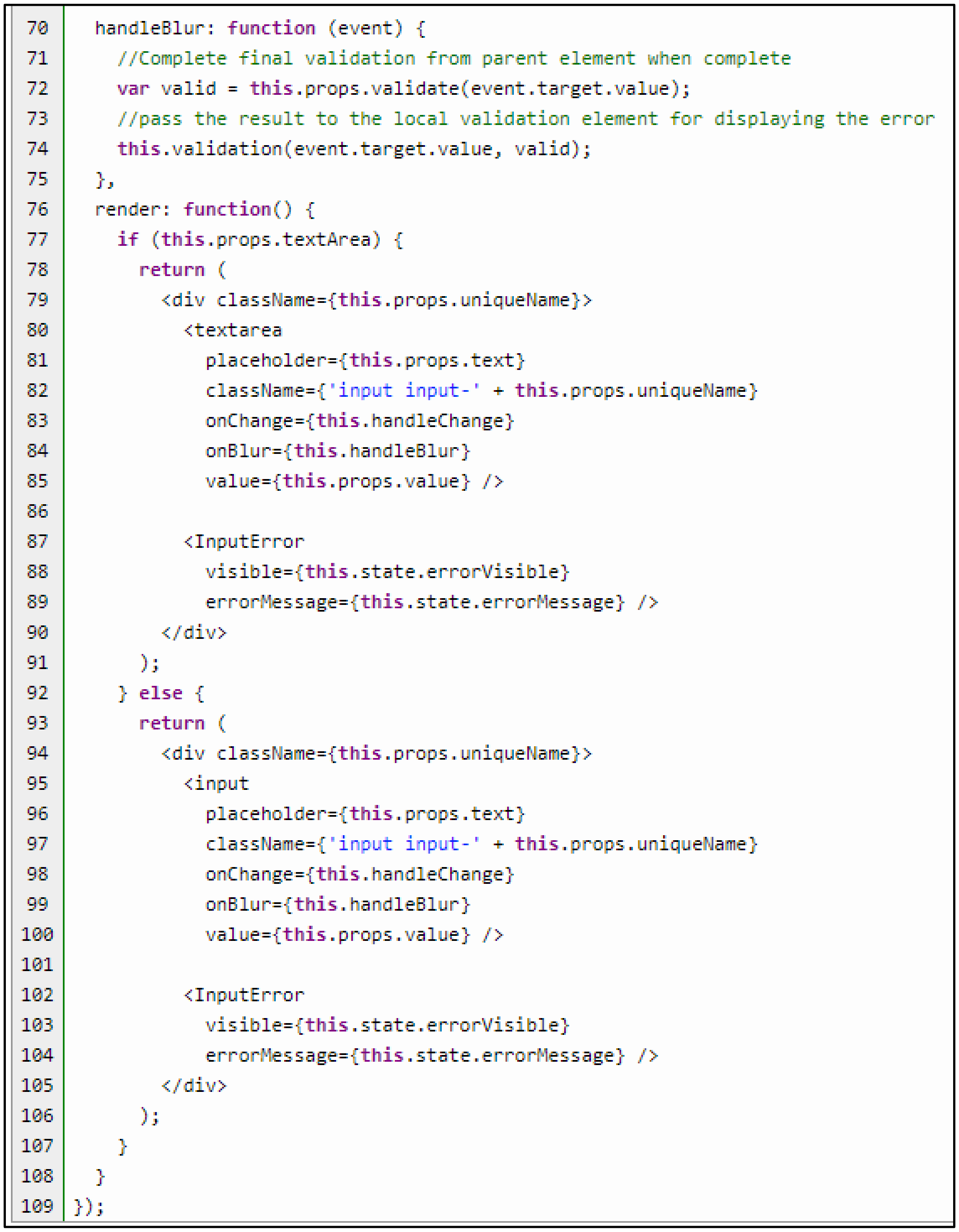
Each component call also has a new value attribute. This is what allows us to empty all the fields when the form is submitted. I've have also added a new text field, with an option for producing a textarea rather than a single line text field. This allows us to add a new comment field for contributors who want to say something about why they're donating. Now let's take a look at the few changes made to the form element components.

Emptying fields on form submission

There are no changes to the InputError component, but we do have a few small changes to the other components which will allow us to empty all the fields after submission.



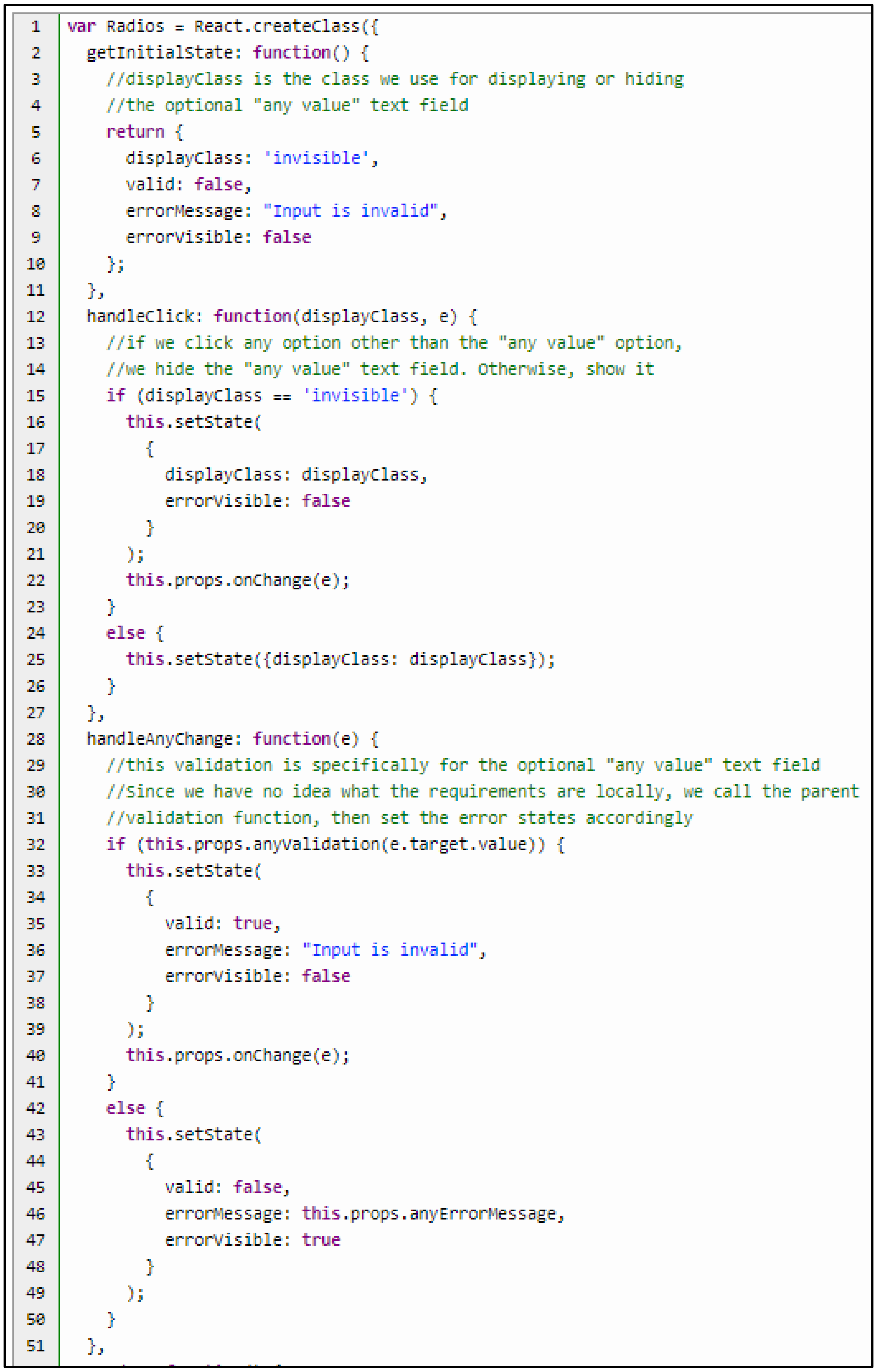
```
var TextInput = React.createClass({
  getInitialState: function(){
    //most of these variables have to do with handling errors
    return {
      isEmpty: true,
      value: null,
      valid: false,
      errorMessage: "Input is invalid",
      errorVisible: false
    };
  },

  handleChange: function(event){
    //validate the field locally
    this.validation(event.target.value);

    //Call onChange method on the parent component for updating it's state
    //If saving this field for final form submission, it gets passed
    // up to the top component for sending to the server
    if(this.props.onChange) {
      this.props.onChange(event);
    }
  },

  validation: function (value, valid) {
    //The valid variable is optional, and true if not passed in:
    if (typeof valid === 'undefined') {
      valid = true;
    }

    var message = "";
    var errorVisible = false;

    //we know how to validate text fields based on information passed through props
    if (!valid) {
      //This happens when the user leaves the field, but it is not valid
      //(we do final validation in the parent component, then pass the result
      //here for display)
      message = this.props.errorMessage;
      valid = false;
      errorVisible = true;
    }
    else if (this.props.required && jQuery.isEmptyObject(value)) {
      //this happens when we have a required field with no text entered
      //in this case, we want the "emptyMessage" error message
      message = this.props.emptyMessage;
      valid = false;
      errorVisible = true;
    }
    else if (value.length < this.props.minCharacters) {
      //This happens when the text entered is not the required length,
      //in which case we show the regular error message
      message = this.props.errorMessage;
      valid = false;
      errorVisible = true;
    }

    //setting the state will update the display,
    //causing the error message to display if there is one.
    this.setState({
      value: value,
      isEmpty: jQuery.isEmptyObject(value),
      valid: valid,
      errorMessage: message,
      errorVisible: errorVisible
    });

  },

```



```
  handleBlur: function (event) {
    //Complete final validation from parent element when complete
    var valid = this.props.validate(event.target.value);
    //pass the result to the local validation element for displaying the error
    this.validation(event.target.value, valid);
  },
  render: function() {
    if (this.props.textArea) {
      return (
        <div className={this.props.uniqueName}>
          <textarea
            placeholder={this.props.text}
            className={'input input-' + this.props.uniqueName}
            onChange={this.handleChange}
            onBlur={this.handleBlur}
            value={this.props.value} />

          <InputError
            visible={this.state.errorVisible}
            errorMessage={this.state.errorMessage} />
        </div>
      );
    } else {
      return (
        <div className={this.props.uniqueName}>
          <input
            placeholder={this.props.text}
            className={'input input-' + this.props.uniqueName}
            onChange={this.handleChange}
            onBlur={this.handleBlur}
            value={this.props.value} />

          <InputError
            visible={this.state.errorVisible}
            errorMessage={this.state.errorMessage} />
        </div>
      );
    }
  }
});
```

The TextInput component has the simplest change because all we have to do is set the value to the props value sent to the component. We were already calling this.props.onChange from the handleChange function for real-time error processing, and simply rearranged the error processing a bit in the parent component so we can save the data. We still call the parent validation method when the user leaves the field.

We do add one additional input field to handle the textarea option. When the text field is a textarea, we have to use the textarea field. Otherwise, it’s a normal text field. There are no other changes.



```
var Radios = React.createClass({
  getInitialState: function() {
    //displayClass is the class we use for displaying or hiding
    //the optional "any value" text field
    return {
      displayClass: 'invisible',
      valid: false,
      errorMessage: "Input is invalid",
      errorVisible: false
    };
  },
  handleClick: function(displayClass, e) {
    //if we click any option other than the "any value" option,
    //we hide the "any value" text field. Otherwise, show it
    if (displayClass == 'invisible') {
      this.setState(
        {
          displayClass: displayClass,
          errorVisible: false
        }
      );
      this.props.onChange(e);
    }
    else {
      this.setState({displayClass: displayClass});
    }
  },
  handleAnyChange: function(e) {
    //this validation is specifically for the optional "any value" text field
    //Since we have no idea what the requirements are locally, we call the parent
    //validation function, then set the error states accordingly
    if (this.props.anyValidation(e.target.value)) {
      this.setState(
        {
          valid: true,
          errorMessage: "Input is invalid",
          errorVisible: false
        }
      );
      this.props.onChange(e);
    }
    else {
      this.setState(
        {
          valid: false,
          errorMessage: this.props.anyErrorMessage,
          errorVisible: true
        }
      );
    }
  },
```

```
  render: function() {
    var rows = [];
    var label = "";

    //we have passed in all the options for the radios, so we traverse the array
    for (var i = 0; i < this.props.values.length; i++) {
      //We do this little replace for when we want to display the value as part of
      //additional text. Otherwise, we would just put '[VALUE]' when passing
      //the itemLabel prop from the parent component, or leave out '[VALUE]' entirely
      label = this.props.itemLabel.replace('[VALUE]', this.props.values[i]);

      //You'll see that even the <br /> field has a key. React will give you errors
      //if you don't do this. This is just an axample of what's possible, but
      //you would normally add extra spacing with css
      rows.push(<input
        key={this.props.name + '-' + i}
        type="radio"
        ref={this.props.name + '-' + this.props.values[i]}
        name={this.props.name}
        value={this.props.values[i]}
        selected={this.props.value==this.props.values[i]?true:false}
        onClick={this.handleClick.bind(this, 'invisible')} />,

        <label key={this.props.name + '-label-' + i} htmlFor={this.props.values[i]}>{label}</label>,

        <br key={this.props.name + '-br-' + i} />);
    }

    //The "any value" field complicates things a bit
    if (this.props.addAny) {
      //we passed in a separate label just for the option that
      //activates the "any value" text field
      var selected = false;
      label = this.props.anyLabel;
      if (this.props.value != undefined && this.props.values.indexOf(this.props.value) == -1) {
        selected = true;
      }
      rows.push(<input
        key={this.props.name + '-' + i}
        type="radio"
        ref={this.props.name + '-any'}
        name={this.props.name} value="any"
        selected={selected}
        onClick={this.handleClick.bind(this, 'visible')} />,

        <label key={this.props.name + '-label-' + i} htmlFor={this.props.values[i]}>{label}</label>);

      //and now we add the "any value text field, with all its special variables"
      var value = "";
      if (selected) {
        value = this.props.value;
      }

      rows.push(<div key={this.props.name + '-div-' + (i+2)} className={this.state.displayClass}>
        <input
          className="anyValue"
          key={this.props.name + '-' + (i+1)}
          type="text"
          value={value}
          placeholder={this.props.anyPlaceholder}
          onChange={this.handleAnyChange}
          ref={this.props.name} />
      </div>);
    }

    //Now we just return all those rows, along with the error component
    return (
      <div className="radios">
        {rows}

        <InputError
          visible={this.state.errorVisible}
          errorMessage={this.state.errorMessage} />
      </div>
    );
  }
});
```

The only changes made to the Radios component have to do with the addition of the value attribute. You may have noticed that the default and reset values sent to the Radios and Department components are "undefined." That's because there's always a possibility that one of the radio or select options could be 0 or an empty string, but both components also contain an "any value" text field. The only way to truly unset these radio and select fields, therefore, is to set the value to undefined.

Properly setting the regular radio buttons is easy. We just select the radio button using the selected attribute if it matches the value, as we iterate through each radio button:

```
selected={this.props.value==this.props.values[i]?true:false}
```

Just because we are using the value property to empty the fields doesn't mean that we don't have to set the current value. Remember that we are saving the state as the user makes changes, but that state change causes the display to update, which means that we have to set the value correctly even when the field is not empty. Otherwise, the fields would be set to empty as soon as a user makes a change!

We then put the “undefined” setting to use when deciding whether to select the “other” option:

```
var selected = false;
if (this.props.value != undefined && this.props.values.indexOf(this.props.value) == -1) {
  selected = true;
}
```

If the value is undefined, then we know not to select anything, because we are emptying all the fields. If it’s not undefined, however, and the value does not match any of the other radio buttons, then we know we have a custom value.

For the optional text field, we just need to set the value if it exists:

```
var value = "";
if (selected) {
  value = this.props.value;
}
```

There are no changes to the Payment component, but we do have a small change to the Department component (even though we aren’t actually saving the value):

```
var Department = React.createClass({
  getInitialState: function() {
    return {
      displayClass: 'invisible'
    };
  },
  handleClick: function(e) {
    //We're doing another one of these "any value" fields, only shown when
    //a specific "other" option is chosen
    this.props.onChange(e);
    var displayClass = 'invisible';
    if (e.target.value == 'other') {
      displayClass = 'visible';
    }
    this.setState({displayClass: displayClass});
  },
  render: function() {
    //This is a select field with options and sub-options, plus an "any value" field
    var value = this.props.value;
    if (this.props.value != undefined && ['none', 'muir', 'revelle', 'sixth', 'jacobs', 'global', 'medicine', 'scholarships'].indexOf(this.props.value) == -1) {
      value = 'other';
    }
    else if (this.props.value == undefined) {
      value = 'none';
    }

    return (
      <div className="department">
        <select value={value} onChange={this.handleClick} multiple={false} ref="department">
          <option value="none"></option>
          <optgroup label="College">
            <option value="muir">Muir</option>
            <option value="revelle">Revelle</option>
            <option value="sixth">Sixth</option>
          </optgroup>
          <optgroup label="School">
            <option value="jacobs">Jacobs School of Engineering</option>
            <option value="global">School of Global Policy and Strategy</option>
            <option value="medicine">School of Medicine</option>
          </optgroup>
          <option value="scholarships">Scholarships</option>
          <option value="other">Other</option>
        </select>
        <div className={this.state.displayClass}>
          <input className="anyValue" value={this.props.value=='other'?this.props.value:''} type="text" onChange={this.props.onChange} placeholder="Department" ref="
        </div>

        <InputError
          visible={this.state.errorVisible}
          errorMessage={this.state.errorMessage} />
      </div>
    );
  }
});
```

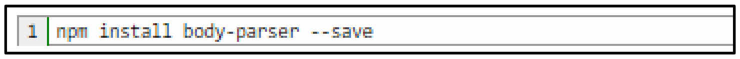
Since this is a select field with an “any value” text field, we have to check for “undefined” again:

```
    var value = this.props.value;
    if (this.props.value != undefined && ['none', 'muir', 'revelle', 'sixth', 'jacobs', 'global', 'medicine', 'scholarships'].indexOf(this.props.value) == -1) {
      value = 'other';
    }
    else if (this.props.value == undefined) {
      value = 'none';
    }
```

The Department component is a little bit different from the others, in that the options are simply hard-coded. The only reason it’s a separate component is to simplify the code in the DonationForm component. As such, the easiest way to set the correct value is to simply check against the hard-coded values. This is similar to what we did in the Radios component, but with hard-coded values. We then set the text field based on whether they’ve selected “other.”

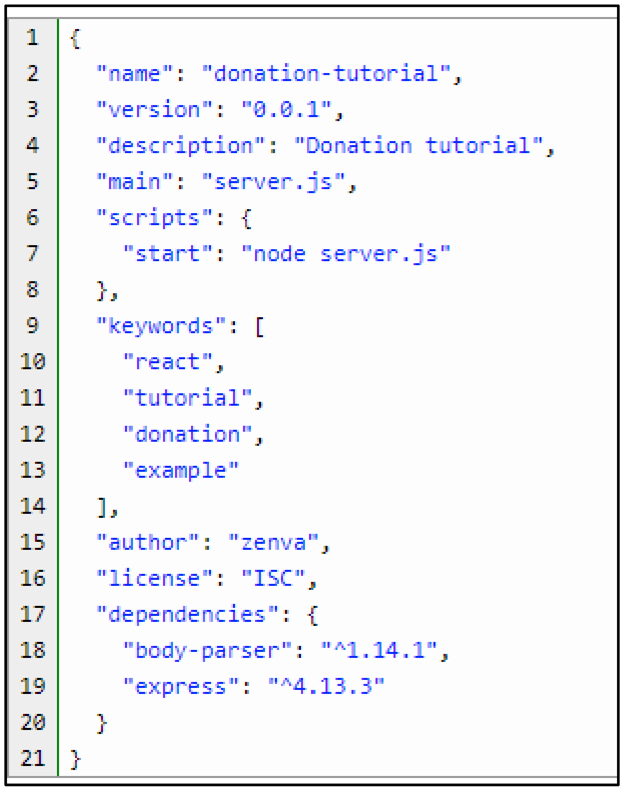
Saving data to the server

Now we get to save the data and watch the changes happen! Our example hinges on the installation of Node.js. Most of what needs to be done is covered in the beginner's Node.js tutorial. If you follow all the directions in that tutorial, within the folder for your project, you'll wind up with a package.json file. The only addition to that tutorial is to install body-parser along with express. After installing express, you will need to run the same command, but with body-parser:



```
npm install body-parser --save
```

This will give you a package.json file similar to the one included with the files for this tutorial:



```
{
  "name": "donation-tutorial",
  "version": "0.0.1",
  "description": "Donation tutorial",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "keywords": [
    "react",
    "tutorial",
    "donation",
    "example"
  ],
  "author": "zenva",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.14.1",
    "express": "^4.13.3"
  }
}
```

We use server.js for our example, since we are creating a server, although the file in the Node.js tutorial is called script.js. You can use either name as long as the name in your package file matches the name of your Node file.

From there, you would create your Node server in whatever way you see fit. The server can do additional processing and error checking, and should include security measures as well. It can also save information to a database, and read from a database. In our example file, we simply save to, and read from, a file called donations.json. In fact, our server is so simple that it's very similar to Facebook's server.js file, included within their tutorial files. You can't really make it more simple than that!

If using the files with this tutorial, all you have to do is install Node.js (which includes npm), then go to the same folder as the package.json file and run the following command on the command line:

```
npm install
```

Once it's installed, just run the following command:

```
node server.js
```

But look at the following line in package.json:

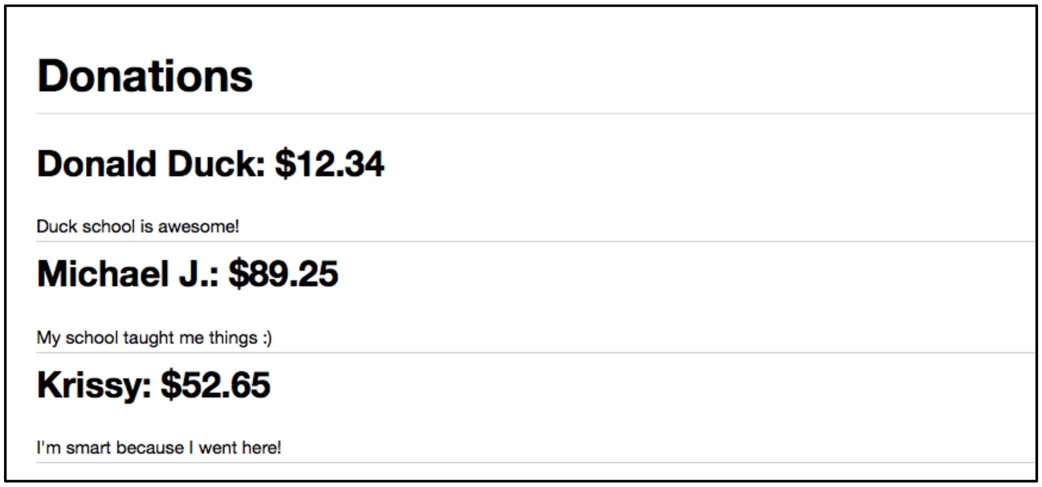
```
"start": "node server.js"
```

This means that you can also start the server by running the following command:

```
npm start
```

The server will then start, and tell you right where you need to go in your browser: http://localhost:3000/

Once you open that URL, you should see a list of donations, with the ability to add new donations. In addition, if you add a field directly to the donations.json file, it will display on the screen almost immediately!



Donations

Donald Duck: $12.34

Duck school is awesome!

Michael J.: $89.25

My school taught me things :)

Krissy: $52.65

I'm smart because I went here!

Chapter 4 Creating a Crossword Puzzle game with React.JS

In this tutorial, I will show you how to create a crossword puzzle app, using React.js.

React is a high-performance, open-source, reactive Javascript UI library for client-side web applications. At first, you may find working with React confusing, but I will try to make this tutorial as simple as possible by taking it step by step.

Download the source code

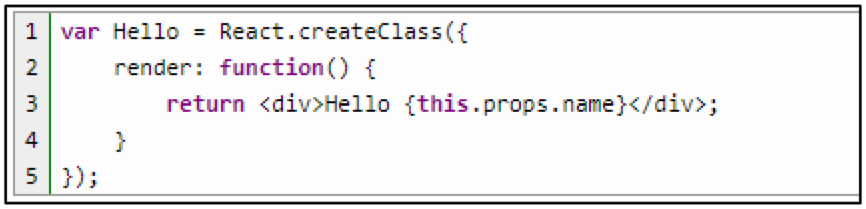
You can download all the files associated with this tutorial here and here.

Tutorial requirements

- A good understanding of HTML, CSS and most importantly JavaScript.
- Download the React library or you can use the CDN.
- A text editor, Notepad++ or Sublime.

Intro to JSFiddle

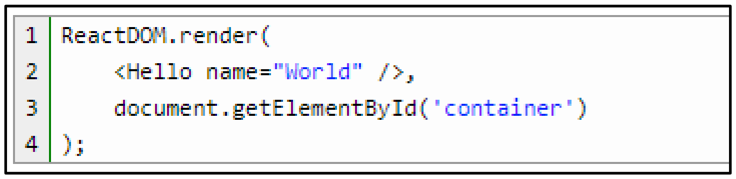
JSFiddle allows us to test our code right inside the browser. Start by going to facebook.github.io/react/ and click on the "Get Started" link, then click on "React JSFiddle" link there. Click here to take you to the Hello World example done with React.



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

Looking at this simple Hello World example, you will notice two major pieces of the React library. The first being the **`React.createClass`** function that is used to define components. The component being defined here, is "Hello World" and is assigned a variable called Hello. To define the Hello World component, we provide a function called render, and within this render function we return something that looks like HTML code with some kind of variable interpolation.

Further down, we see a call to another React function, **`ReactDOM.render`**, and within this call, there is also HTML looking snippet with an element name that refers to the component that is defined in the above **`React.createClass`** function, along with an attribute that matches the interpolation within the render function.



```
ReactDOM.render(
    <Hello name="World" />,
    document.getElementById('container')
);
```

The second argument of the **`ReactDOM.render`** function, `document.body`, tells React where to render the **`Hello`** component.

Downloading React

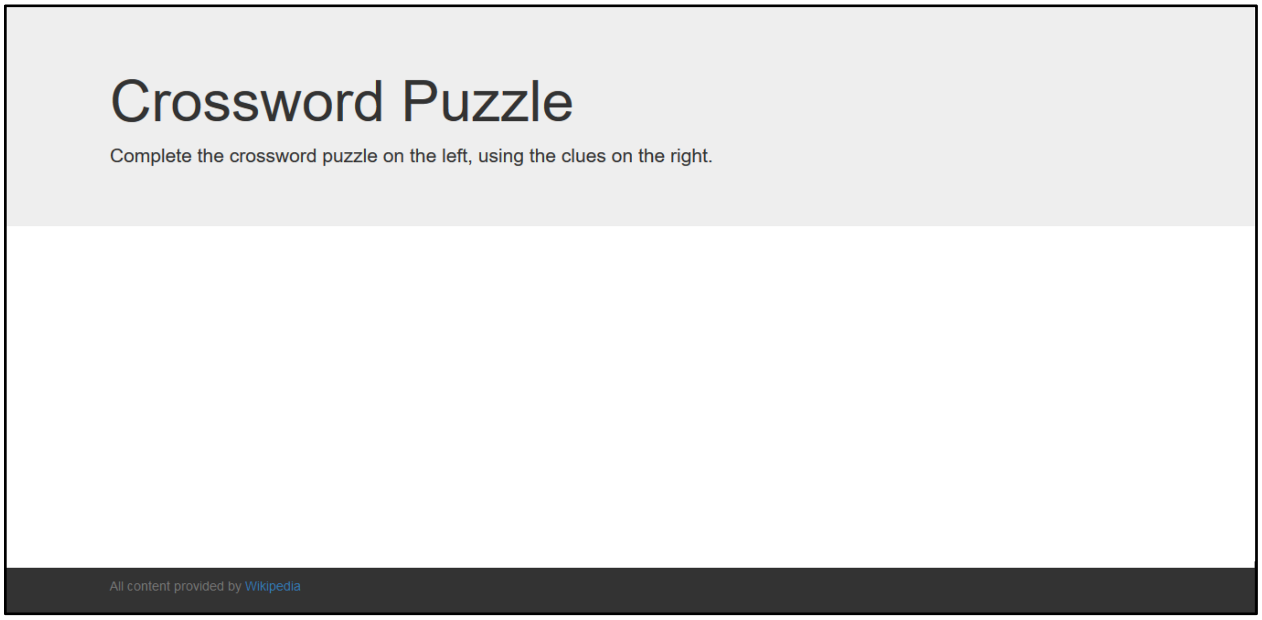
Back on the React website, this time click on “Download React *version number*”, then the “Download Starter Kit” link. This download link includes the React and some simple example apps. The current version I am using is React 0.14.5.

In your text editor, we will create a skeleton of our Crossword Puzzle game, by starting with an empty HTML page with some boilerplate. We also reference the React script, the first one is the React library itself, including the ReactDOM and the second is the Babel CDN. You will notice there is a div with an id of game, this is to be the container for our application. We also add the Bootstrap CDN. Should you wish to download Bootstrap for yourself, have a look at my tutorial on Bootstrap.

```html
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv='Content-type' content='text/html; charset=utf-8' />
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css" />
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css" />
    <title>Basic Example</title>
  </head>
  <body>
    <div class="jumbotron">
      <div class="container">
        <h1>Crossword Puzzle</h1>
        <p>Complete the crossword puzzle on the right, using the clues on the left.</p>
      </div>
    </div>
    <div class="container">
      <div id="game" />
    </div>
    <div id="footer">
      <div class="container">
        <p class="credit text-muted">
          All content provided by <a href="http://en.wikipedia.org/">Wikipedia</a>
        </p>
      </div>
    </div>
    <script src="react-0.14.3/build/react.js"></script>
    <script src="react-0.14.3/build/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
    <script src="puzzle.js" type="text/babel"></script>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"></script>
  </body>
</html>
```

We have just added the non-interactive part of the game.

Within the **`game div`**, we will define our very first React application. We'll start with a new ScriptBlock, with the type of **`text/babel`** named **`puzzle.js`**. This is where all our React code for the crossword puzzle will reside.


Crossword Puzzle

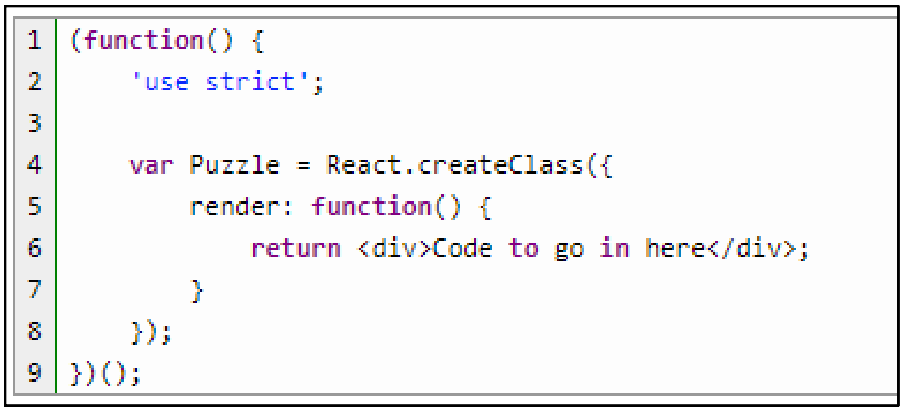
Complete the crossword puzzle on the left, using the clues on the right.

All content provided by Wikipedia

Why Babel? Babel replaced JSX Transformer with the release of React 0.14.0. As the JavaScript language evolved, the React creators believed that it was time to deprecate JSX and start implementing the new features of Babel. So, what is Babel? It is a next generation JavaScript compiler.

Defining Components

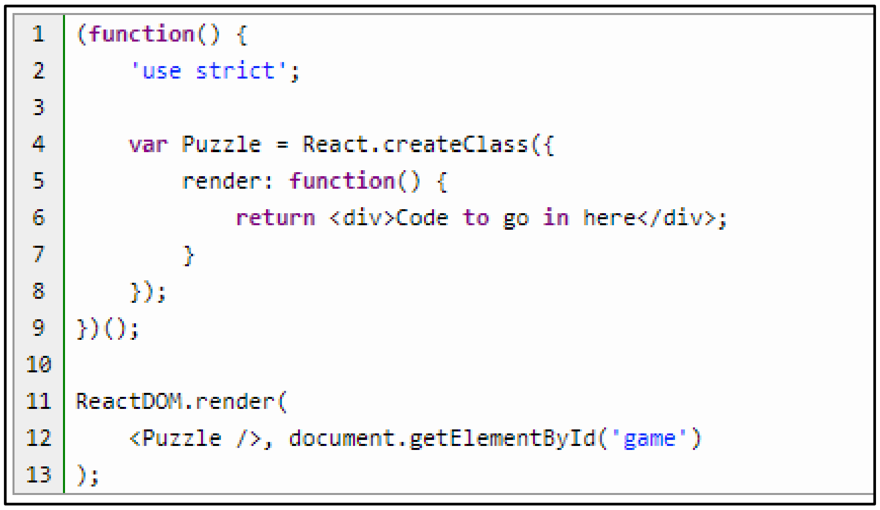
The top-level namespace for React is **`React`** with the uppercase R. New components are defined by calling the **`createClass`** function. **`createClass`** function has one argument, which is an **`object`**, and defines the new components and the minimum requirement for that object is to define a render function. Within that render function, we return a single React component. Here, we return some HTML, this is the JSX. In this case, we returning a div with some text. Currently this div is where we will fill it with more content. To transform this JSX to JavaScript, we use the Babel compiler to transform the HTML syntax into plain JavaScript.



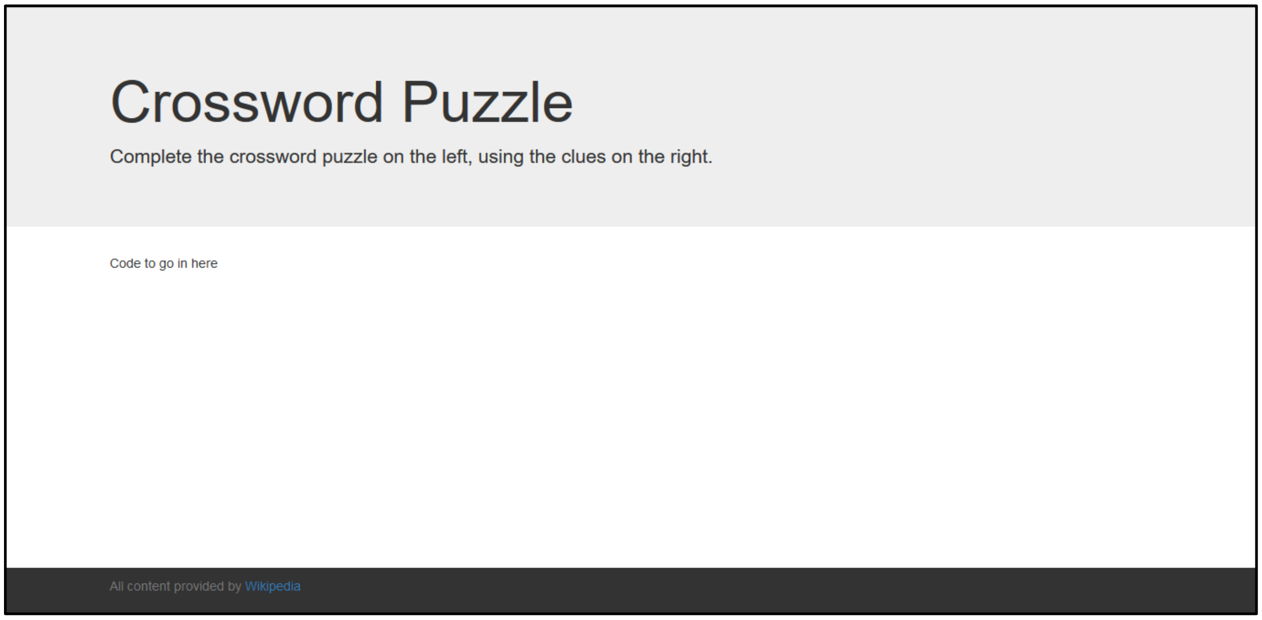
```
(function() {
    'use strict';

    var Puzzle = React.createClass({
        render: function() {
            return <div>Code to go in here</div>;
        }
    });
})();
```

Rendering Components

The above is the definition of the React component, now to render that component. The rendering process is triggered to link the top-level component to an existing DOM node, and populate that DOM node. Rendering is triggered via the **`ReactDOM.render`** function. Render requires two arguments, the first being the JSX expression that defines the react component to render, and the second argument is the existing DOM element to bond the component to.



```
(function() {
    'use strict';

    var Puzzle = React.createClass({
        render: function() {
            return <div>Code to go in here</div>;
        }
    });
})();

ReactDOM.render(
    <Puzzle />, document.getElementById('game')
);
```



Crossword Puzzle

Complete the crossword puzzle on the left, using the clues on the right.

Code to go in here

All content provided by Wikipedia

So far, we have created and rendered our **`Puzzle`** component. The “Code to go in here” should appear in your browser along with the Jumbotron header and credits.

Populating Props

Props, or properties, are the constant immutable model values that React component uses to render. They are supplied as XML attributes within the JSX syntax.

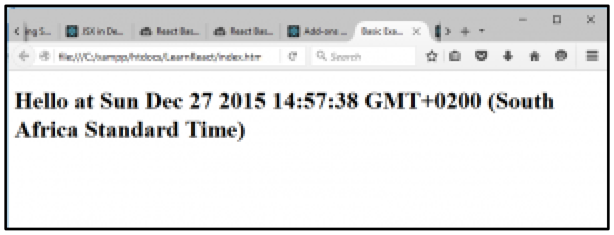
Looking at that Hello World program in JSFiddle, let’s change it to the following:

```
var Hello = React.createClass({
    render: function() {
        return <div><h1>Today is the {this.props.today}</h1>{this.props.num}</div>
    }
});

ReactDOM.render(
    <Hello today={new Date().toDateString()} num="24" />,
    document.getElementById('container')
);
```

Here, we have passed a string representing the current date to the **`Hello`** component as a property called **`today`**. The today attribute is a JavaScript expression, therefore it is enclosed in curly braces. The value assigned to num attribute is a string literal, so it is enclosed in double quotes.

`this.props.today` and **`this.props.num`** are used to access the **`today`** and the **`num`** value as a property on the **`props`** object.

When you click on Update on JSFiddle to save the changes you made, then click Run to run the code you edited, the output should be something like this:



Hello at Sun Dec 27 2015 14:57:38 GMT+0200 (South Africa Standard Time)

One more thing to remember, JSX (prior to version 0.14.0) did not own a white space, so to get a space between the **`today`** prop and the **`num`** prop, another JSX expression containing a string literal with a space was added, like this.

```
var Hello = React.createClass({
    render: function() {
        return <div><h1>Today is the {this.props.today}{" "}{this.props.num}</h1></div>
    }
});
```

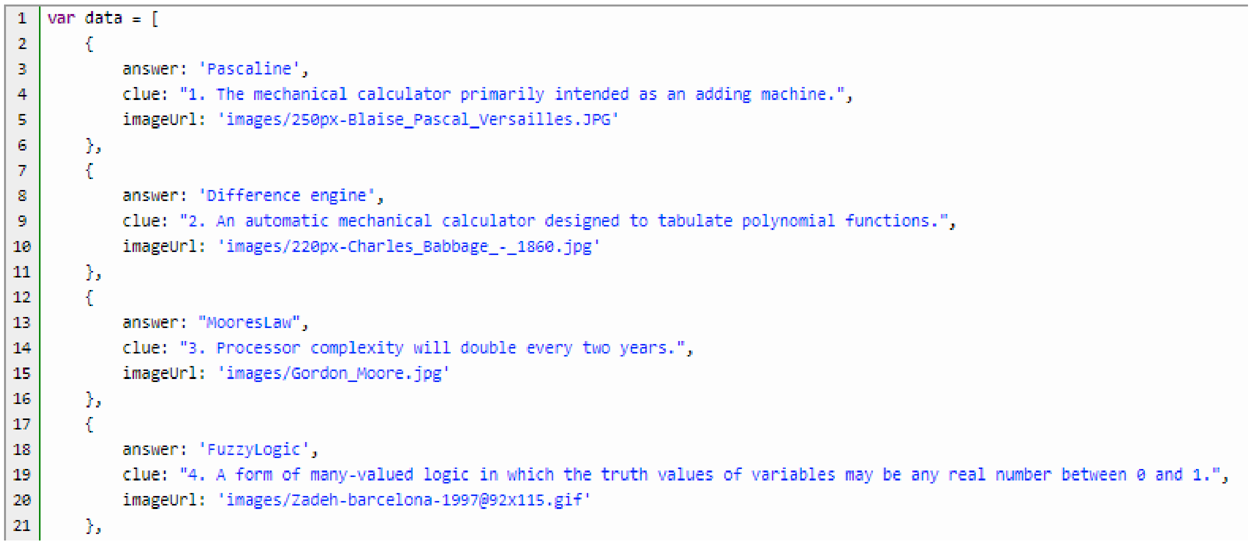
Because **`<h1>`** is a block-level element, you would not have noticed the white space. The example in Figure 9 shows the **`num`** attribute moved inside the **`<h1>`** element to demonstrate the purpose of the empty string literal.

However, since we are using Babel and not JSX Transformer, this white space is not a problem. You can easily implement that line of code like so:

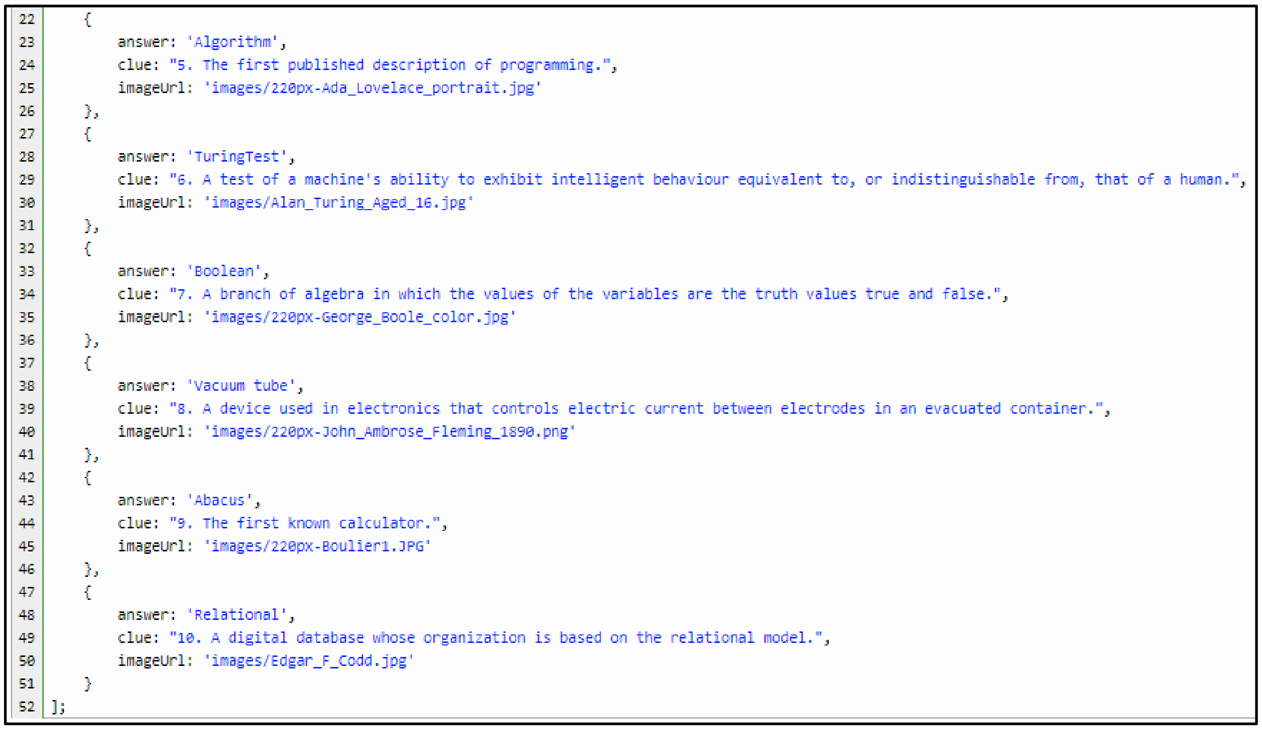
```
var Hello = React.createClass({
    render: function() {
        return <div><h1>Today is the {this.props.today} {this.props.num}</h1></div>
    }
});
```

So far, we have understood how to define components, render the components to be viewed on the browser, and populating properties.

Before we continue, let's add some data as JSON. I'll create a JavaScript object called data and this is to be an array.



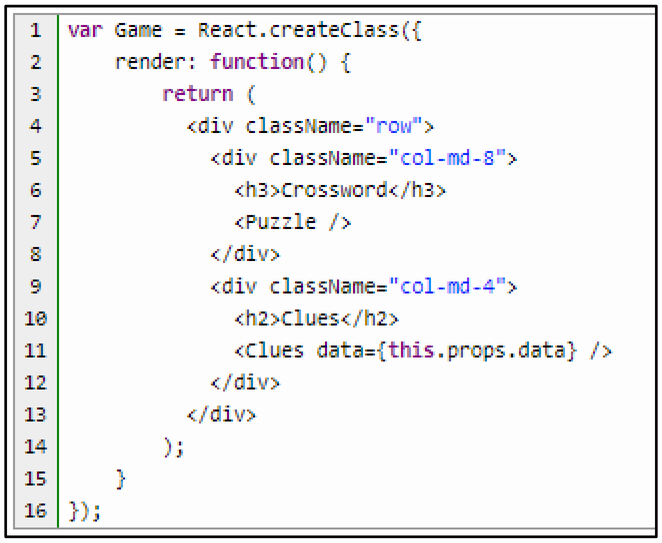
```
var data = [
    {
        answer: 'Pascaline',
        clue: "1. The mechanical calculator primarily intended as an adding machine.",
        imageUrl: 'images/250px-Blaise_Pascal_Versailles.JPG'
    },
    {
        answer: 'Difference engine',
        clue: "2. An automatic mechanical calculator designed to tabulate polynomial functions.",
        imageUrl: 'images/220px-Charles_Babbage_-_1860.jpg'
    },
    {
        answer: "MooresLaw",
        clue: "3. Processor complexity will double every two years.",
        imageUrl: 'images/Gordon_Moore.jpg'
    },
    {
        answer: 'FuzzyLogic',
        clue: "4. A form of many-valued logic in which the truth values of variables may be any real number between 0 and 1.",
        imageUrl: 'images/Zadeh-barcelona-1997@92x115.gif'
    },
```



```
    {
        answer: 'Algorithm',
        clue: "5. The first published description of programming.",
        imageUrl: 'images/220px-Ada_Lovelace_portrait.jpg'
    },
    {
        answer: 'TuringTest',
        clue: "6. A test of a machine's ability to exhibit intelligent behaviour equivalent to, or indistinguishable from, that of a human.",
        imageUrl: 'images/Alan_Turing_Aged_16.jpg'
    },
    {
        answer: 'Boolean',
        clue: "7. A branch of algebra in which the values of the variables are the truth values true and false.",
        imageUrl: 'images/220px-George_Boole_color.jpg'
    },
    {
        answer: 'Vacuum tube',
        clue: "8. A device used in electronics that controls electric current between electrodes in an evacuated container.",
        imageUrl: 'images/220px-John_Ambrose_Fleming_1890.png'
    },
    {
        answer: 'Abacus',
        clue: "9. The first known calculator.",
        imageUrl: 'images/220px-Boulier1.JPG'
    },
    {
        answer: 'Relational',
        clue: "10. A digital database whose organization is based on the relational model.",
        imageUrl: 'images/Edgar_F_Codd.jpg'
    }
];
```

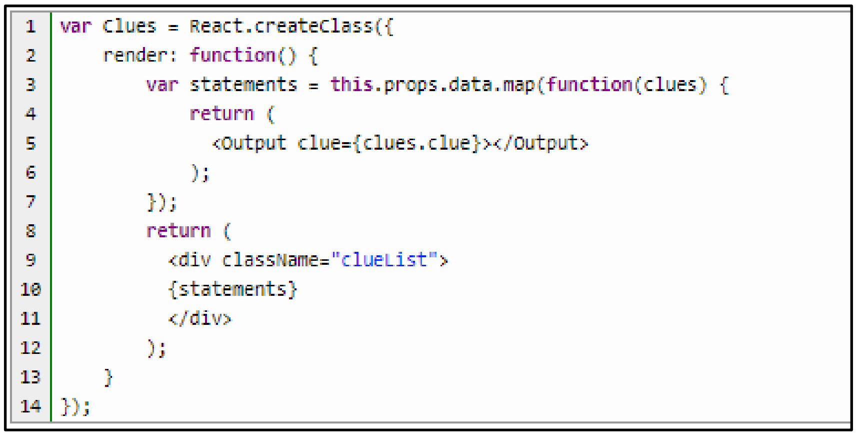
Populating Properties in the Game

We will change our code to make Game the top-level component. This component will also render the JSON data. Also, added Bootstrap for mobile view.



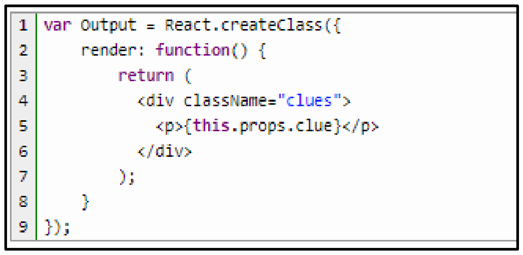
```
var Game = React.createClass({
    render: function() {
        return (
          <div className="row">
            <div className="col-md-8">
              <h3>Crossword</h3>
              <Puzzle />
            </div>
            <div className="col-md-4">
              <h2>Clues</h2>
              <Clues data={this.props.data} />
            </div>
          </div>
        );
    }
});
```

You will notice there is what looks like an HTML element named Clues, but it is just a React component we will be creating next.



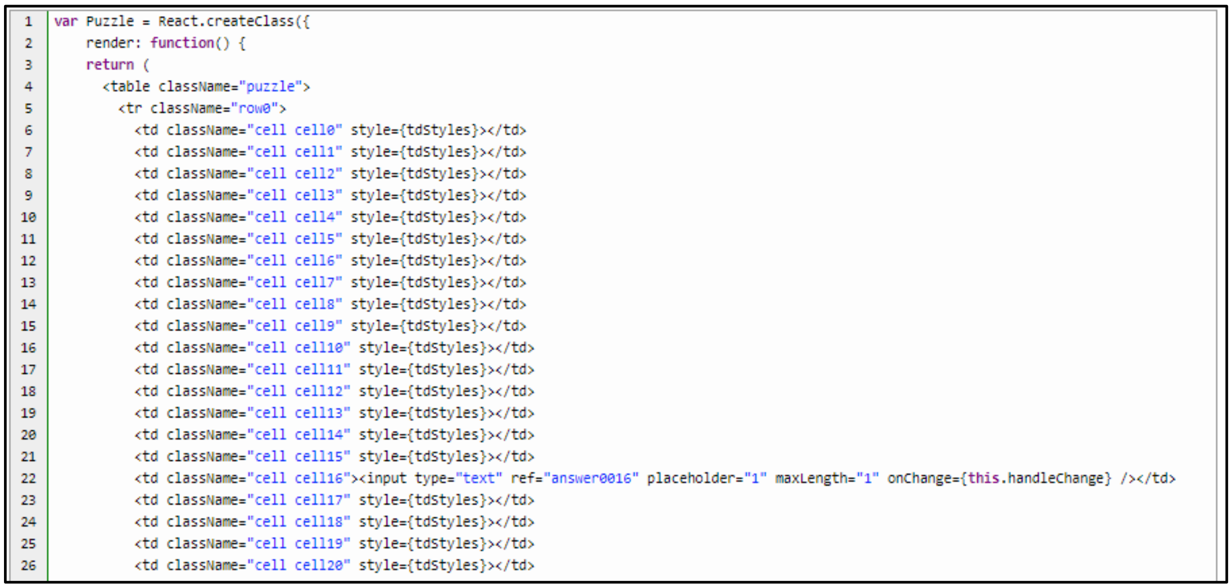
```
var Clues = React.createClass({
    render: function() {
        var statements = this.props.data.map(function(clues) {
            return (
              <Output clue={clues.clue}></Output>
            );
        });
        return (
          <div className="clueList">
          {statements}
          </div>
        );
    }
});
```

In the statements variable, you will notice **`map`**, which takes each item from the data array, performs transformation and returns a new array containing the transformed elements. Here, we rendered the statements dynamically and returned it. Here, we finally render the population of the props.

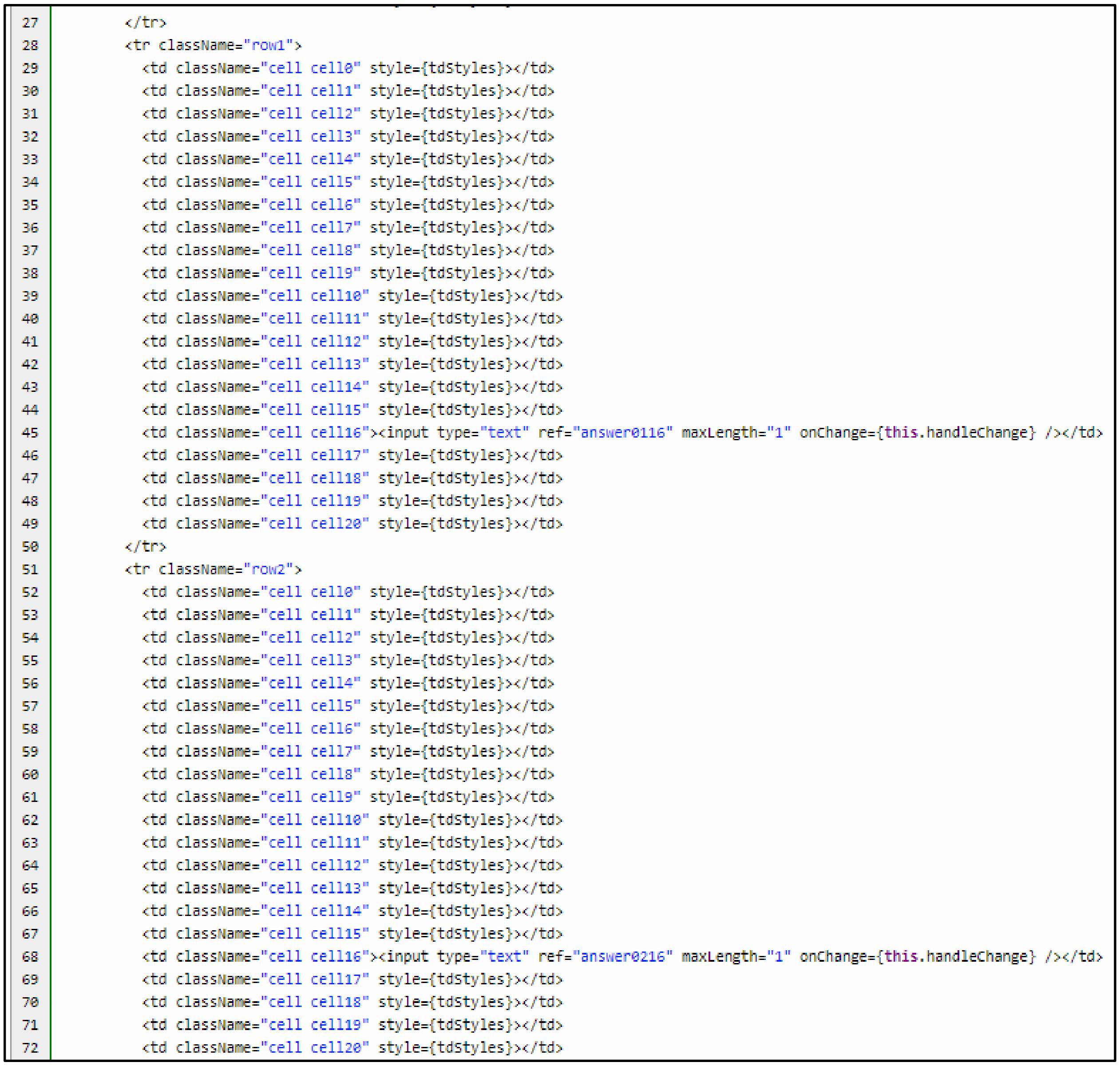


```
var Output = React.createClass({
    render: function() {
        return (
          <div className="clues">
            <p>{this.props.clue}</p>
          </div>
        );
    }
});
```

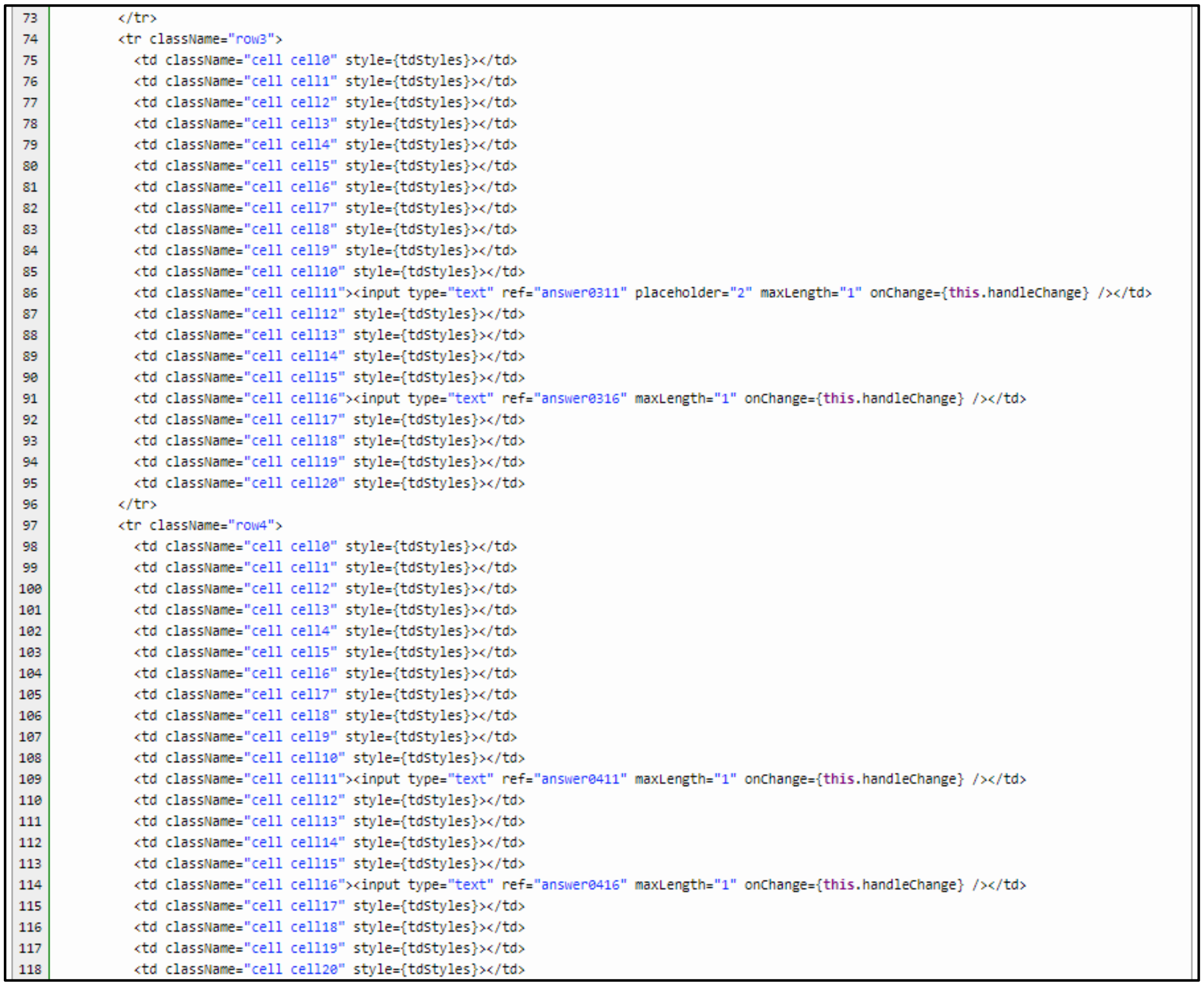
Once we have this props, we are ready to complete the layout of the crossword. This is a very basic layout in table form.



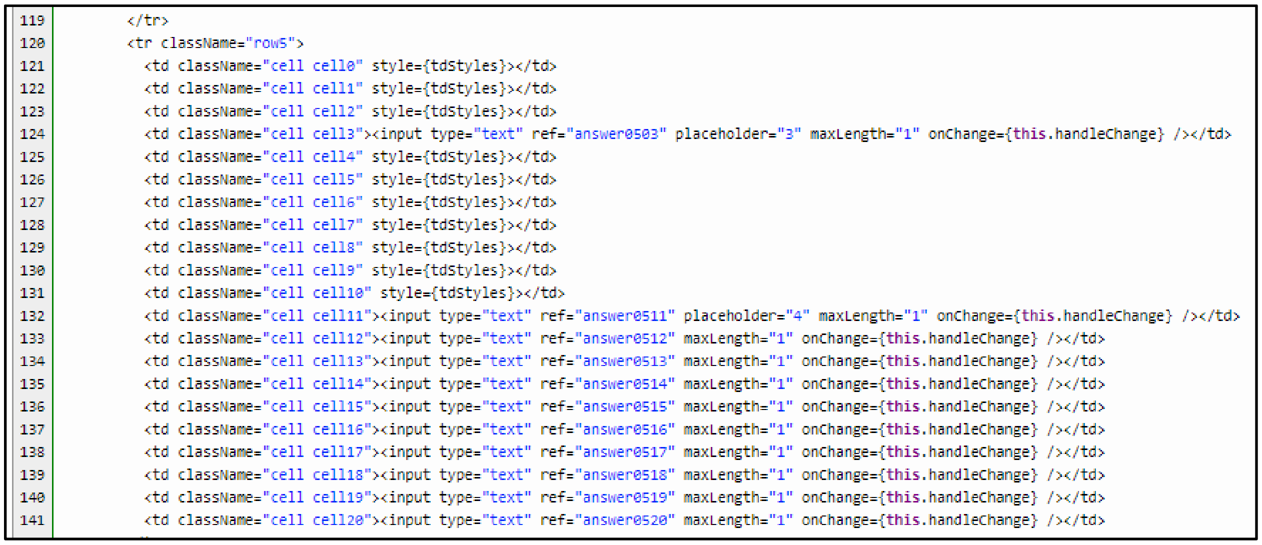
```
var Puzzle = React.createClass({
    render: function() {
    return (
      <table className="puzzle">
        <tr className="row0">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11" style={tdStyles}></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0016" placeholder="1" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



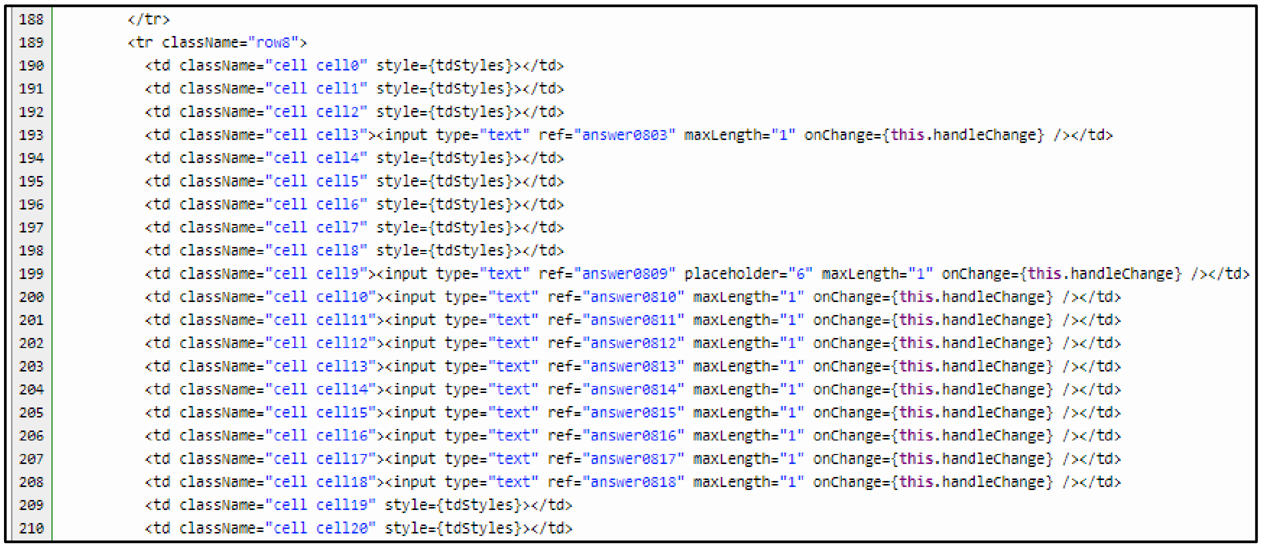
```
        </tr>
        <tr className="row1">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11" style={tdStyles}></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0116" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
        <tr className="row2">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11" style={tdStyles}></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0216" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



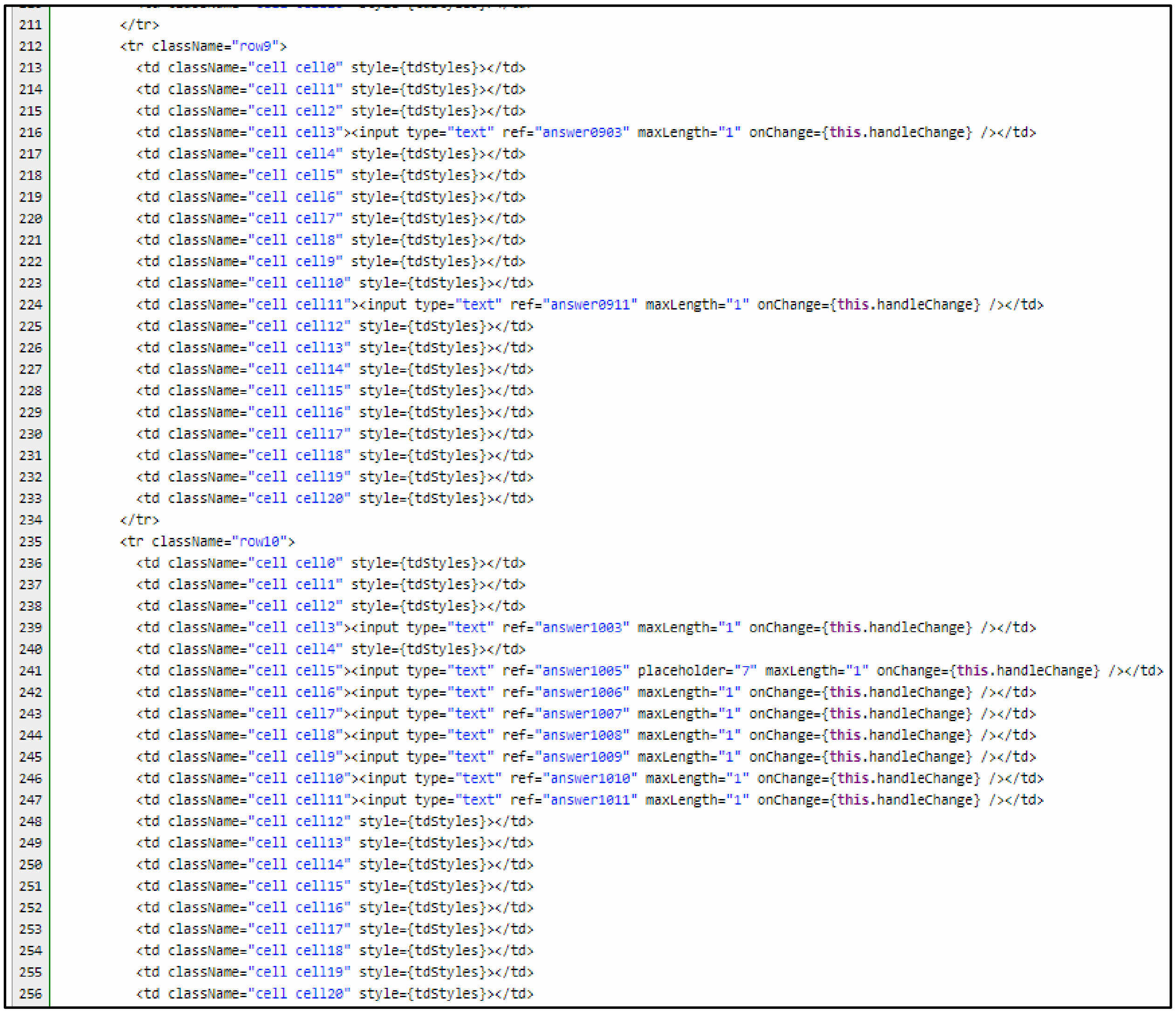
```
        </tr>
        <tr className="row3">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer0311" placeholder="2" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0316" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
        <tr className="row4">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer0411" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0416" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



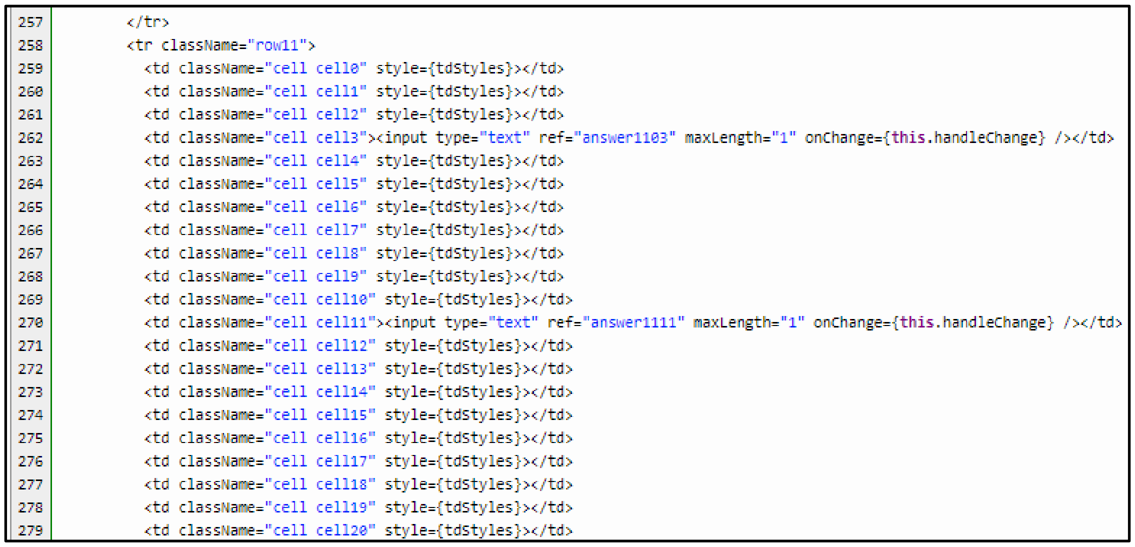
```
        </tr>
        <tr className="row5">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer0503" placeholder="3" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer0511" placeholder="4" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12"><input type="text" ref="answer0512" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell13"><input type="text" ref="answer0513" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell14"><input type="text" ref="answer0514" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell15"><input type="text" ref="answer0515" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell16"><input type="text" ref="answer0516" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17"><input type="text" ref="answer0517" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell18"><input type="text" ref="answer0518" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell19"><input type="text" ref="answer0519" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell20"><input type="text" ref="answer0520" maxLength="1" onChange={this.handleChange} /></td>
        </tr>
        <tr className="row6">
          <td className="cell cell0"><input type="text" ref="answer0600" placeholder="5" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell1"><input type="text" ref="answer0601" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell2"><input type="text" ref="answer0602" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell3"><input type="text" ref="answer0603" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4"><input type="text" ref="answer0604" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell5"><input type="text" ref="answer0605" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell6"><input type="text" ref="answer0606" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell7"><input type="text" ref="answer0607" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell8"><input type="text" ref="answer0608" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer0611" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0616" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
        <tr className="row7">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer0703" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer0711" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16"><input type="text" ref="answer0716" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



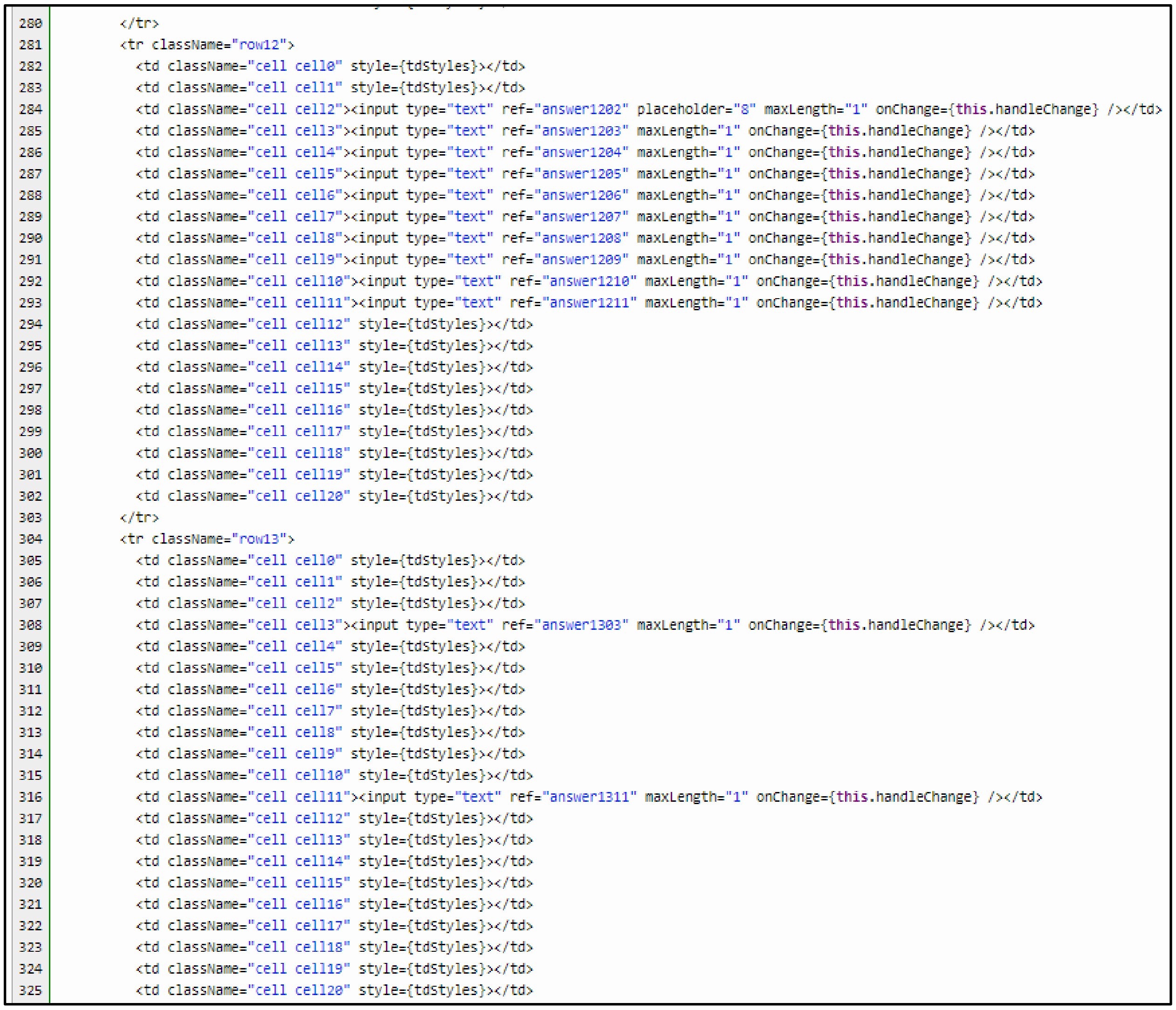
```
        </tr>
        <tr className="row8">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer0803" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9"><input type="text" ref="answer0809" placeholder="6" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell10"><input type="text" ref="answer0810" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell11"><input type="text" ref="answer0811" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12"><input type="text" ref="answer0812" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell13"><input type="text" ref="answer0813" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell14"><input type="text" ref="answer0814" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell15"><input type="text" ref="answer0815" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell16"><input type="text" ref="answer0816" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell17"><input type="text" ref="answer0817" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell18"><input type="text" ref="answer0818" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



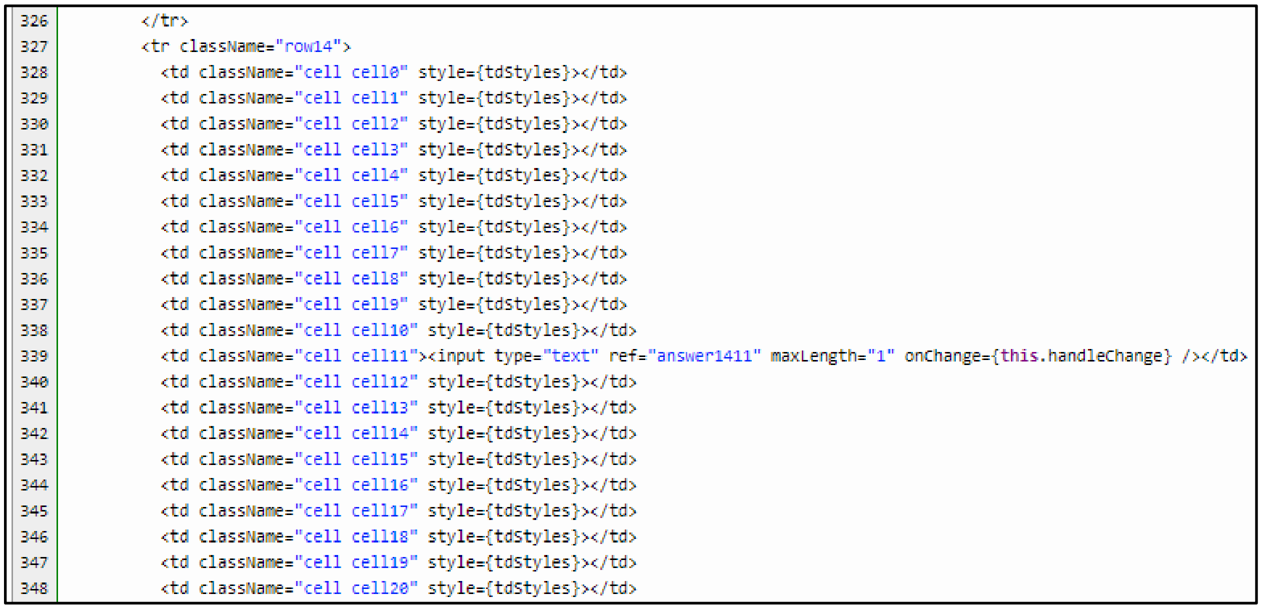
```
        </tr>
        <tr className="row9">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer0903" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer0911" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
        <tr className="row10">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer1003" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5"><input type="text" ref="answer1005" placeholder="7" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell6"><input type="text" ref="answer1006" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell7"><input type="text" ref="answer1007" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell8"><input type="text" ref="answer1008" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell9"><input type="text" ref="answer1009" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell10"><input type="text" ref="answer1010" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell11"><input type="text" ref="answer1011" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



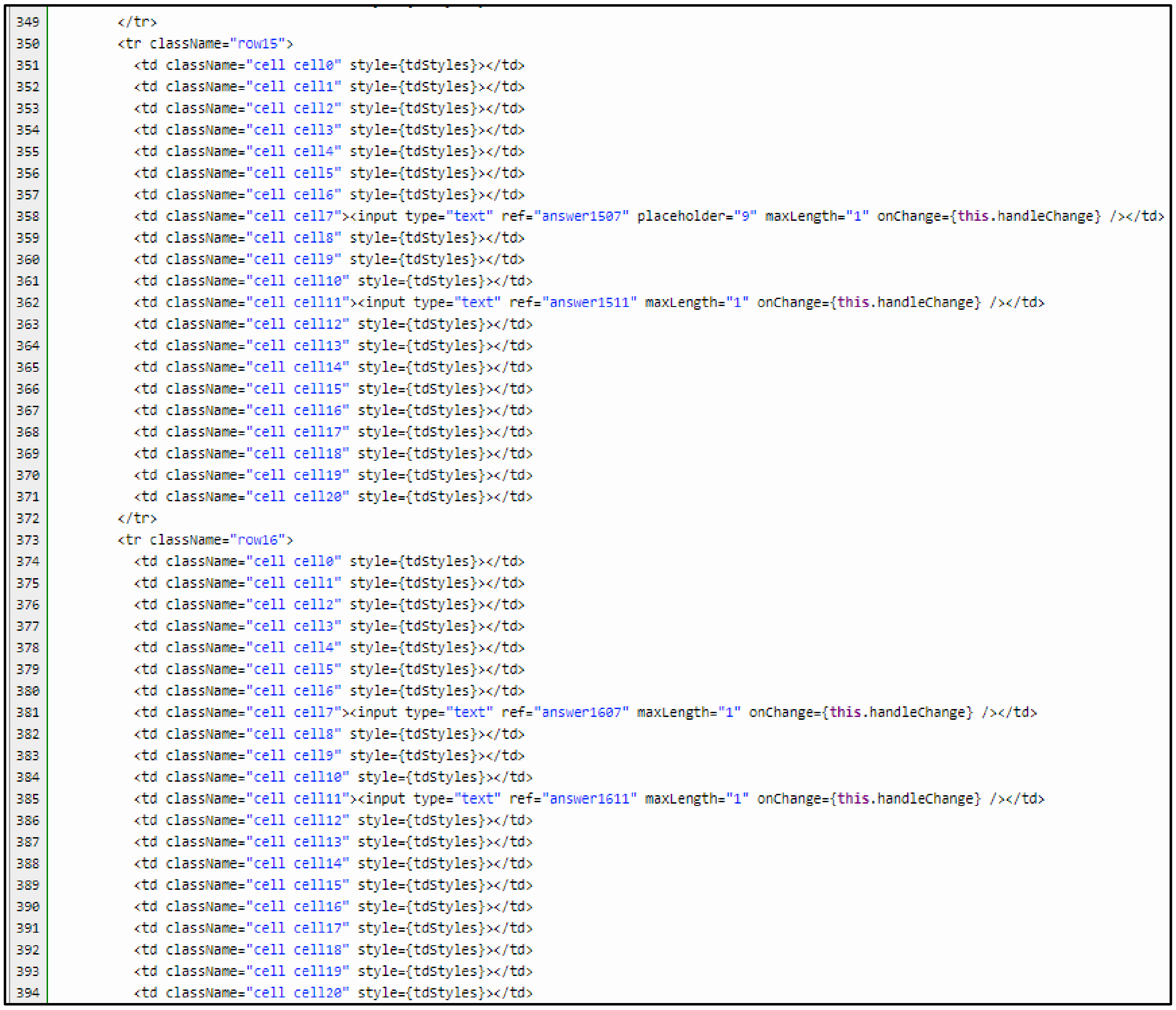
```
        </tr>
        <tr className="row11">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer1103" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer1111" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



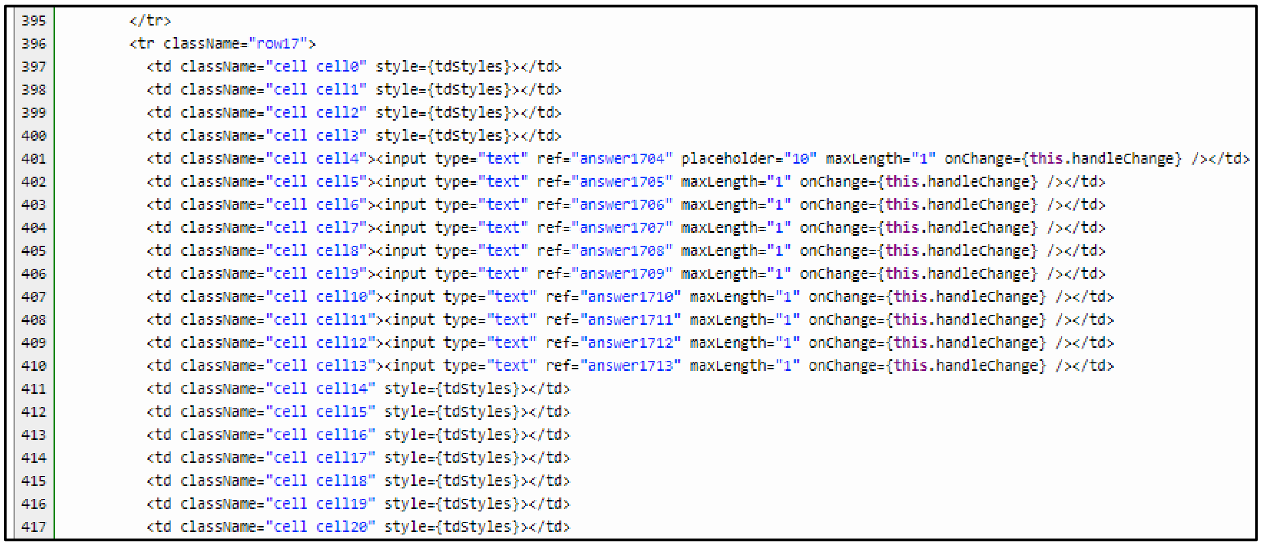
```
        </tr>
        <tr className="row12">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2"><input type="text" ref="answer1202" placeholder="8" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell3"><input type="text" ref="answer1203" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4"><input type="text" ref="answer1204" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell5"><input type="text" ref="answer1205" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell6"><input type="text" ref="answer1206" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell7"><input type="text" ref="answer1207" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell8"><input type="text" ref="answer1208" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell9"><input type="text" ref="answer1209" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell10"><input type="text" ref="answer1210" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell11"><input type="text" ref="answer1211" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
        <tr className="row13">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3"><input type="text" ref="answer1303" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer1311" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



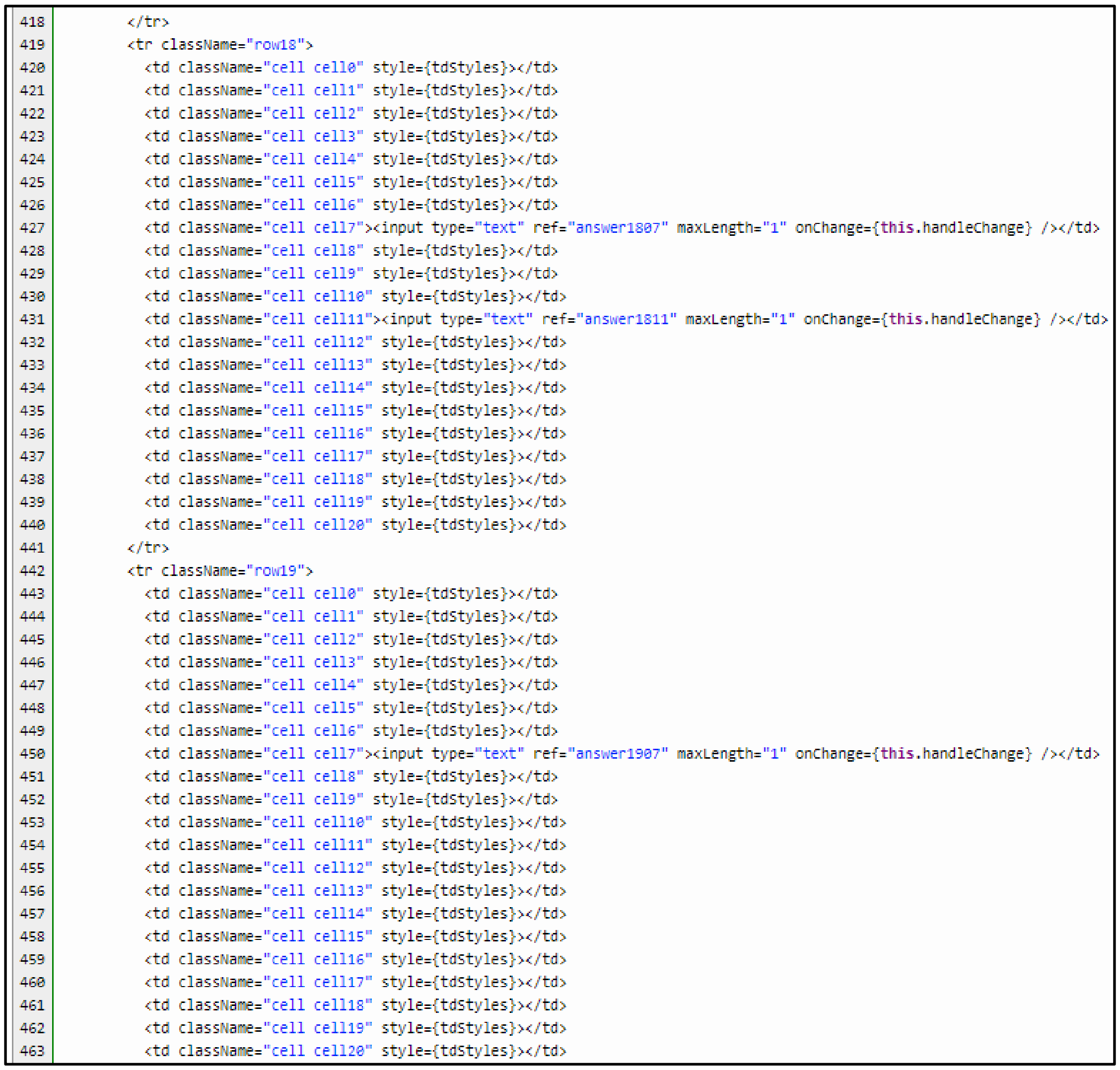
```
        </tr>
        <tr className="row14">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7" style={tdStyles}></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer1411" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



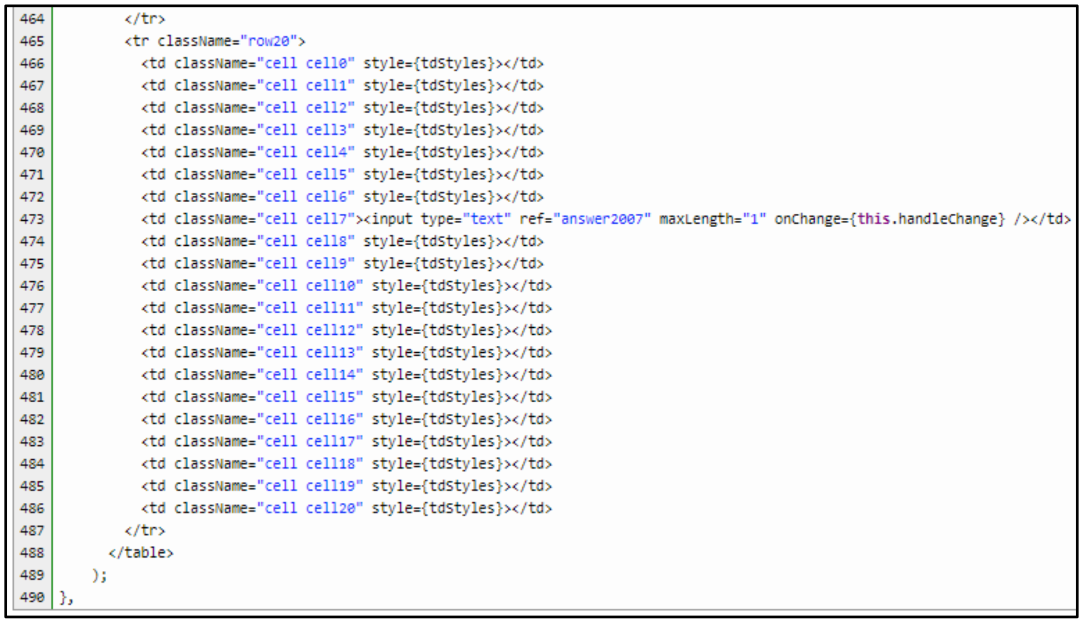
```
      </tr>
      <tr className="row15">
        <td className="cell cell0" style={tdStyles}></td>
        <td className="cell cell1" style={tdStyles}></td>
        <td className="cell cell2" style={tdStyles}></td>
        <td className="cell cell3" style={tdStyles}></td>
        <td className="cell cell4" style={tdStyles}></td>
        <td className="cell cell5" style={tdStyles}></td>
        <td className="cell cell6" style={tdStyles}></td>
        <td className="cell cell7"><input type="text" ref="answer1507" placeholder="9" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell8" style={tdStyles}></td>
        <td className="cell cell9" style={tdStyles}></td>
        <td className="cell cell10" style={tdStyles}></td>
        <td className="cell cell11"><input type="text" ref="answer1511" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell12" style={tdStyles}></td>
        <td className="cell cell13" style={tdStyles}></td>
        <td className="cell cell14" style={tdStyles}></td>
        <td className="cell cell15" style={tdStyles}></td>
        <td className="cell cell16" style={tdStyles}></td>
        <td className="cell cell17" style={tdStyles}></td>
        <td className="cell cell18" style={tdStyles}></td>
        <td className="cell cell19" style={tdStyles}></td>
        <td className="cell cell20" style={tdStyles}></td>
      </tr>
      <tr className="row16">
        <td className="cell cell0" style={tdStyles}></td>
        <td className="cell cell1" style={tdStyles}></td>
        <td className="cell cell2" style={tdStyles}></td>
        <td className="cell cell3" style={tdStyles}></td>
        <td className="cell cell4" style={tdStyles}></td>
        <td className="cell cell5" style={tdStyles}></td>
        <td className="cell cell6" style={tdStyles}></td>
        <td className="cell cell7"><input type="text" ref="answer1607" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell8" style={tdStyles}></td>
        <td className="cell cell9" style={tdStyles}></td>
        <td className="cell cell10" style={tdStyles}></td>
        <td className="cell cell11"><input type="text" ref="answer1611" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell12" style={tdStyles}></td>
        <td className="cell cell13" style={tdStyles}></td>
        <td className="cell cell14" style={tdStyles}></td>
        <td className="cell cell15" style={tdStyles}></td>
        <td className="cell cell16" style={tdStyles}></td>
        <td className="cell cell17" style={tdStyles}></td>
        <td className="cell cell18" style={tdStyles}></td>
        <td className="cell cell19" style={tdStyles}></td>
        <td className="cell cell20" style={tdStyles}></td>
```



```
      </tr>
      <tr className="row17">
        <td className="cell cell0" style={tdStyles}></td>
        <td className="cell cell1" style={tdStyles}></td>
        <td className="cell cell2" style={tdStyles}></td>
        <td className="cell cell3" style={tdStyles}></td>
        <td className="cell cell4"><input type="text" ref="answer1704" placeholder="10" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell5"><input type="text" ref="answer1705" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell6"><input type="text" ref="answer1706" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell7"><input type="text" ref="answer1707" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell8"><input type="text" ref="answer1708" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell9"><input type="text" ref="answer1709" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell10"><input type="text" ref="answer1710" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell11"><input type="text" ref="answer1711" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell12"><input type="text" ref="answer1712" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell13"><input type="text" ref="answer1713" maxLength="1" onChange={this.handleChange} /></td>
        <td className="cell cell14" style={tdStyles}></td>
        <td className="cell cell15" style={tdStyles}></td>
        <td className="cell cell16" style={tdStyles}></td>
        <td className="cell cell17" style={tdStyles}></td>
        <td className="cell cell18" style={tdStyles}></td>
        <td className="cell cell19" style={tdStyles}></td>
        <td className="cell cell20" style={tdStyles}></td>
```

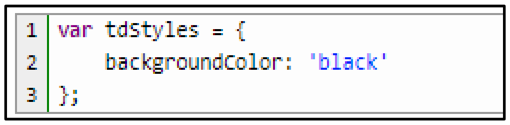


```
        </tr>
        <tr className="row18">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7"><input type="text" ref="answer1807" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11"><input type="text" ref="answer1811" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
        <tr className="row19">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7"><input type="text" ref="answer1907" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11" style={tdStyles}></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
```



```
        </tr>
        <tr className="row20">
          <td className="cell cell0" style={tdStyles}></td>
          <td className="cell cell1" style={tdStyles}></td>
          <td className="cell cell2" style={tdStyles}></td>
          <td className="cell cell3" style={tdStyles}></td>
          <td className="cell cell4" style={tdStyles}></td>
          <td className="cell cell5" style={tdStyles}></td>
          <td className="cell cell6" style={tdStyles}></td>
          <td className="cell cell7"><input type="text" ref="answer2007" maxLength="1" onChange={this.handleChange} /></td>
          <td className="cell cell8" style={tdStyles}></td>
          <td className="cell cell9" style={tdStyles}></td>
          <td className="cell cell10" style={tdStyles}></td>
          <td className="cell cell11" style={tdStyles}></td>
          <td className="cell cell12" style={tdStyles}></td>
          <td className="cell cell13" style={tdStyles}></td>
          <td className="cell cell14" style={tdStyles}></td>
          <td className="cell cell15" style={tdStyles}></td>
          <td className="cell cell16" style={tdStyles}></td>
          <td className="cell cell17" style={tdStyles}></td>
          <td className="cell cell18" style={tdStyles}></td>
          <td className="cell cell19" style={tdStyles}></td>
          <td className="cell cell20" style={tdStyles}></td>
        </tr>
      </table>
    );
},
```

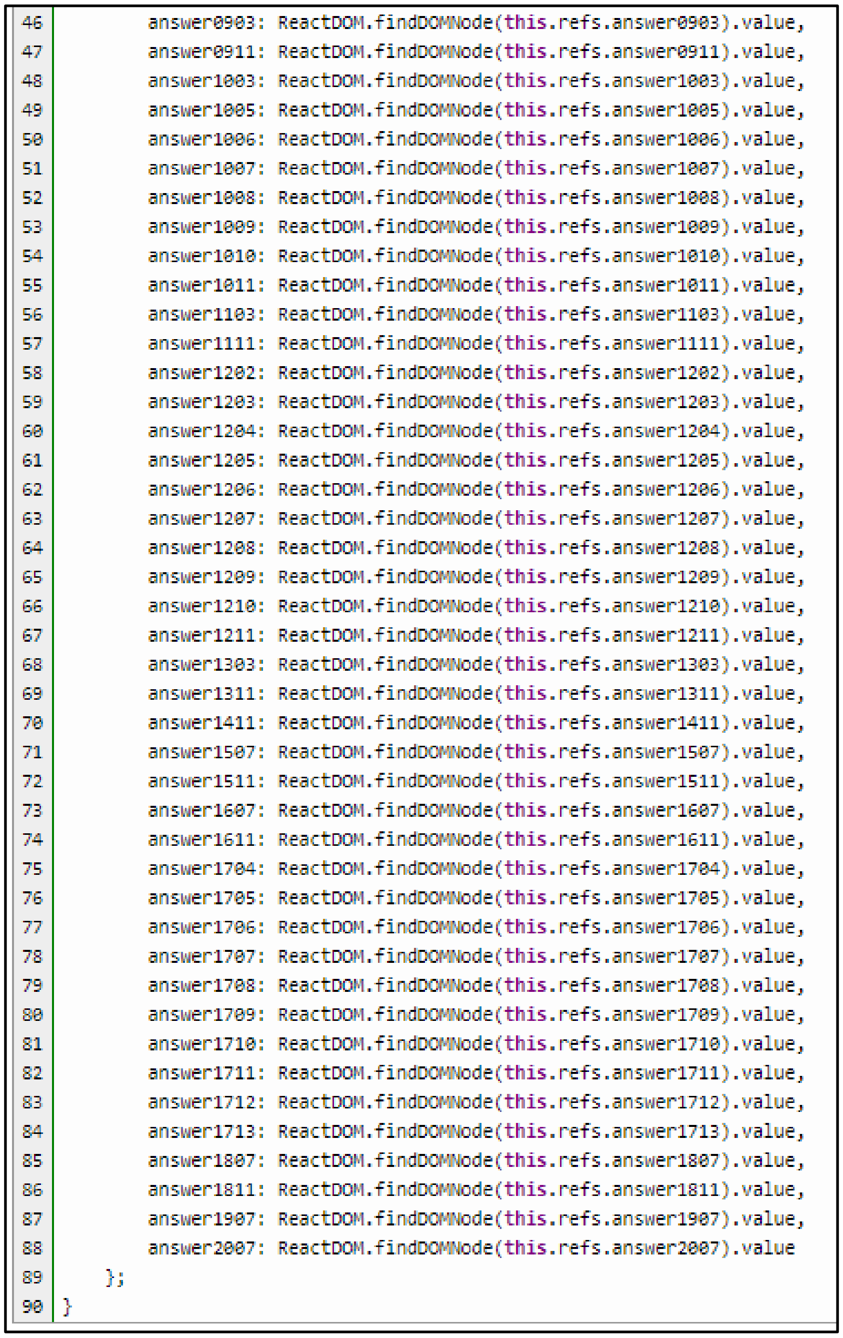
I used input boxes for the letters. These input boxes has a maximum length of **1**, to allow only one letter. For the other cells, I added a **`style`** to change the background color black.



```
var tdStyles = {
    backgroundColor: 'black'
};
```

Finally, those input textboxes will not work unless you handle those changes. You will notice that in each input box, there is a ref attribute. Refs provide a way to get reference to owned components. In the above Puzzle class, we added refs to each input element, so we can access that component instantly via **`this.refs`** in the following code.

```
handleChange: function(e) {
    var data = {
        answer0016: ReactDOM.findDOMNode(this.refs.answer0016).value,
        answer0116: ReactDOM.findDOMNode(this.refs.answer0116).value,
        answer0216: ReactDOM.findDOMNode(this.refs.answer0216).value,
        answer0311: ReactDOM.findDOMNode(this.refs.answer0311).value,
        answer0316: ReactDOM.findDOMNode(this.refs.answer0316).value,
        answer0411: ReactDOM.findDOMNode(this.refs.answer0411).value,
        answer0416: ReactDOM.findDOMNode(this.refs.answer0416).value,
        answer0503: ReactDOM.findDOMNode(this.refs.answer0503).value,
        answer0511: ReactDOM.findDOMNode(this.refs.answer0511).value,
        answer0512: ReactDOM.findDOMNode(this.refs.answer0512).value,
        answer0513: ReactDOM.findDOMNode(this.refs.answer0513).value,
        answer0514: ReactDOM.findDOMNode(this.refs.answer0514).value,
        answer0515: ReactDOM.findDOMNode(this.refs.answer0515).value,
        answer0516: ReactDOM.findDOMNode(this.refs.answer0516).value,
        answer0517: ReactDOM.findDOMNode(this.refs.answer0517).value,
        answer0518: ReactDOM.findDOMNode(this.refs.answer0518).value,
        answer0519: ReactDOM.findDOMNode(this.refs.answer0519).value,
        answer0520: ReactDOM.findDOMNode(this.refs.answer0520).value,
        answer0600: ReactDOM.findDOMNode(this.refs.answer0600).value,
        answer0601: ReactDOM.findDOMNode(this.refs.answer0601).value,
        answer0602: ReactDOM.findDOMNode(this.refs.answer0602).value,
        answer0603: ReactDOM.findDOMNode(this.refs.answer0603).value,
        answer0604: ReactDOM.findDOMNode(this.refs.answer0604).value,
        answer0605: ReactDOM.findDOMNode(this.refs.answer0605).value,
        answer0606: ReactDOM.findDOMNode(this.refs.answer0606).value,
        answer0607: ReactDOM.findDOMNode(this.refs.answer0607).value,
        answer0608: ReactDOM.findDOMNode(this.refs.answer0608).value,
        answer0611: ReactDOM.findDOMNode(this.refs.answer0611).value,
        answer0616: ReactDOM.findDOMNode(this.refs.answer0616).value,
        answer0703: ReactDOM.findDOMNode(this.refs.answer0703).value,
        answer0711: ReactDOM.findDOMNode(this.refs.answer0711).value,
        answer0716: ReactDOM.findDOMNode(this.refs.answer0716).value,
        answer0803: ReactDOM.findDOMNode(this.refs.answer0803).value,
        answer0809: ReactDOM.findDOMNode(this.refs.answer0809).value,
        answer0810: ReactDOM.findDOMNode(this.refs.answer0810).value,
        answer0811: ReactDOM.findDOMNode(this.refs.answer0811).value,
        answer0812: ReactDOM.findDOMNode(this.refs.answer0812).value,
        answer0813: ReactDOM.findDOMNode(this.refs.answer0813).value,
        answer0814: ReactDOM.findDOMNode(this.refs.answer0814).value,
        answer0815: ReactDOM.findDOMNode(this.refs.answer0815).value,
        answer0816: ReactDOM.findDOMNode(this.refs.answer0816).value,
        answer0817: ReactDOM.findDOMNode(this.refs.answer0817).value,
        answer0818: ReactDOM.findDOMNode(this.refs.answer0818).value,
```



```
        answer0903: ReactDOM.findDOMNode(this.refs.answer0903).value,
        answer0911: ReactDOM.findDOMNode(this.refs.answer0911).value,
        answer1003: ReactDOM.findDOMNode(this.refs.answer1003).value,
        answer1005: ReactDOM.findDOMNode(this.refs.answer1005).value,
        answer1006: ReactDOM.findDOMNode(this.refs.answer1006).value,
        answer1007: ReactDOM.findDOMNode(this.refs.answer1007).value,
        answer1008: ReactDOM.findDOMNode(this.refs.answer1008).value,
        answer1009: ReactDOM.findDOMNode(this.refs.answer1009).value,
        answer1010: ReactDOM.findDOMNode(this.refs.answer1010).value,
        answer1011: ReactDOM.findDOMNode(this.refs.answer1011).value,
        answer1103: ReactDOM.findDOMNode(this.refs.answer1103).value,
        answer1111: ReactDOM.findDOMNode(this.refs.answer1111).value,
        answer1202: ReactDOM.findDOMNode(this.refs.answer1202).value,
        answer1203: ReactDOM.findDOMNode(this.refs.answer1203).value,
        answer1204: ReactDOM.findDOMNode(this.refs.answer1204).value,
        answer1205: ReactDOM.findDOMNode(this.refs.answer1205).value,
        answer1206: ReactDOM.findDOMNode(this.refs.answer1206).value,
        answer1207: ReactDOM.findDOMNode(this.refs.answer1207).value,
        answer1208: ReactDOM.findDOMNode(this.refs.answer1208).value,
        answer1209: ReactDOM.findDOMNode(this.refs.answer1209).value,
        answer1210: ReactDOM.findDOMNode(this.refs.answer1210).value,
        answer1211: ReactDOM.findDOMNode(this.refs.answer1211).value,
        answer1303: ReactDOM.findDOMNode(this.refs.answer1303).value,
        answer1311: ReactDOM.findDOMNode(this.refs.answer1311).value,
        answer1411: ReactDOM.findDOMNode(this.refs.answer1411).value,
        answer1507: ReactDOM.findDOMNode(this.refs.answer1507).value,
        answer1511: ReactDOM.findDOMNode(this.refs.answer1511).value,
        answer1607: ReactDOM.findDOMNode(this.refs.answer1607).value,
        answer1611: ReactDOM.findDOMNode(this.refs.answer1611).value,
        answer1704: ReactDOM.findDOMNode(this.refs.answer1704).value,
        answer1705: ReactDOM.findDOMNode(this.refs.answer1705).value,
        answer1706: ReactDOM.findDOMNode(this.refs.answer1706).value,
        answer1707: ReactDOM.findDOMNode(this.refs.answer1707).value,
        answer1708: ReactDOM.findDOMNode(this.refs.answer1708).value,
        answer1709: ReactDOM.findDOMNode(this.refs.answer1709).value,
        answer1710: ReactDOM.findDOMNode(this.refs.answer1710).value,
        answer1711: ReactDOM.findDOMNode(this.refs.answer1711).value,
        answer1712: ReactDOM.findDOMNode(this.refs.answer1712).value,
        answer1713: ReactDOM.findDOMNode(this.refs.answer1713).value,
        answer1807: ReactDOM.findDOMNode(this.refs.answer1807).value,
        answer1811: ReactDOM.findDOMNode(this.refs.answer1811).value,
        answer1907: ReactDOM.findDOMNode(this.refs.answer1907).value,
        answer2007: ReactDOM.findDOMNode(this.refs.answer2007).value
    };
}
```

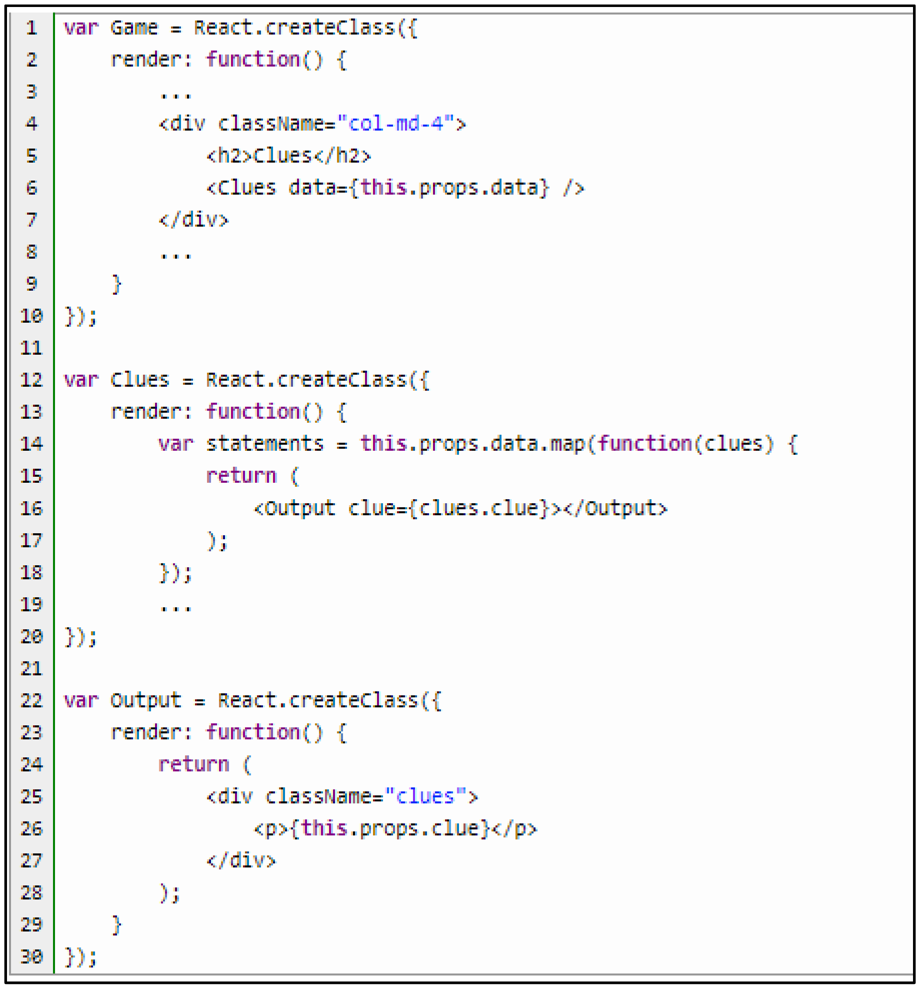
The **`findDOMNode`** is an escape hatch used to access the underlying DOM node from the React component instance. Here, each input component is accessed via its **`ref`** and then the **`findDOMNode`** method is used to get the underlying DOM node, which is a text input. This allows us to type in the textbox, causing a change in its value.

So, we have added components, populated props, add refs to our input components, and accessed those refs with findDOMNode method. We have completed a visual of the crossword puzzle.

Composing Components

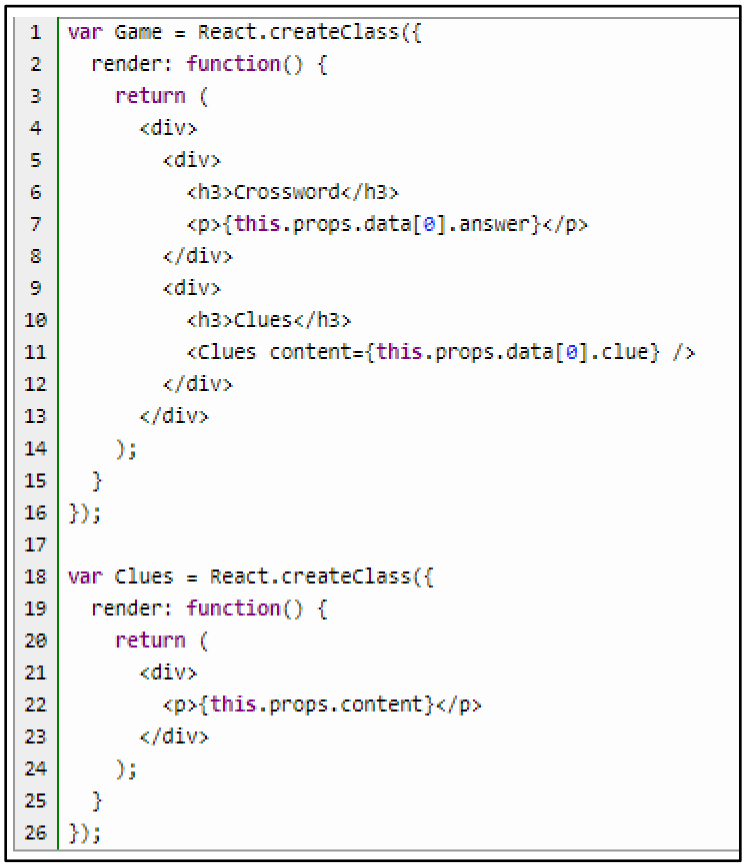
The ability to nest components within other components is a very important feature of React. Because components should be completely self-contained, any component can be nested inside any other component.

In our game, we have composed quite a few components. In Game class, we composed the Clues component and in Clues class, we composed the Output component.



```
var Game = React.createClass({
    render: function() {
        ...
        <div className="col-md-4">
            <h2>Clues</h2>
            <Clues data={this.props.data} />
        </div>
        ...
    }
});

var Clues = React.createClass({
    render: function() {
        var statements = this.props.data.map(function(clues) {
            return (
                <Output clue={clues.clue}></Output>
            );
        });
        ...
});

var Output = React.createClass({
    render: function() {
        return (
            <div className="clues">
                <p>{this.props.clue}</p>
            </div>
        );
    }
});
```

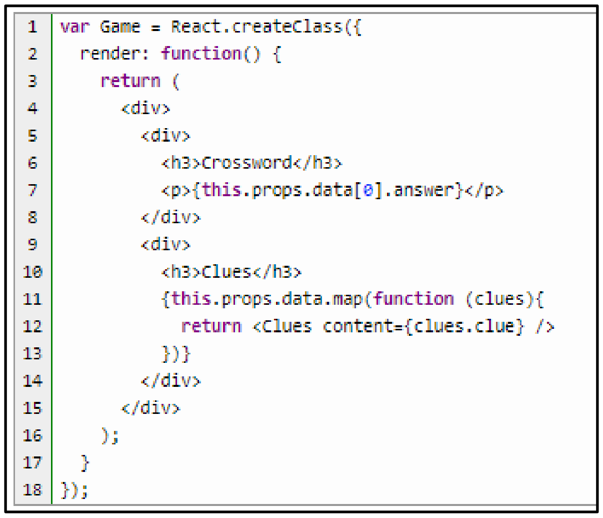
We used map, which takes each item of an array, performs some transformation on it, and returns a new array containing the transformed elements. Testing on JSFiddle, we see that we can get a particular value from an array using the props command.



```
var Game = React.createClass({
  render: function() {
    return (
      <div>
        <div>
          <h3>Crossword</h3>
          <p>{this.props.data[0].answer}</p>
        </div>
        <div>
          <h3>Clues</h3>
          <Clues content={this.props.data[0].clue} />
        </div>
      </div>
    );
  }
});

var Clues = React.createClass({
  render: function() {
    return (
      <div>
        <p>{this.props.content}</p>
      </div>
    );
  }
});
```

While this works for single array value, we are hard-coding the Game component. To handle more than just a set number of clues, we need to change that by mapping each clue to a new Clues component with its content attribute correctly set to the clue value.

So, our code can be one component shorter.



```
var Game = React.createClass({
  render: function() {
    return (
      <div>
        <div>
          <h3>Crossword</h3>
          <p>{this.props.data[0].answer}</p>
        </div>
        <div>
          <h3>Clues</h3>
          {this.props.data.map(function (clues){
            return <Clues content={clues.clue} />
          })}
        </div>
      </div>
    );
  }
});
```

Events

React has a simple, consistent, normalized event system. Just as React has its own DOM abstract, it also has its own event abstraction. This is used to normalize the event behavior, as React makes events behave the same across all supported browsers. React also creates a uniform event system for DOM event objects, replacing them with a wrapper called SyntheticEvent.

DOM events are the events generated by the browser, things like button click events, changing of text inputs, and form submission events. DOM events are the events that receive a React SyntheticEvent object, which is a normalized event object across all browsers.

Let’s try to handle an input change event in JSFiddle. This example, has an input text box and what the user types will be echoed into a label. Much like the behavior in AngularJS.

```
var Echo = React.createClass({
  getInitialState: function() {
    return {value: ''};
  },
  handleChange: function(e) {
    this.setState({value: e.target.value});
  },
  render: function() {
    return (
      <div>
        <input type="text" placeholder="Enter your name" onChange={this.handleChange} />
        <br />
        <label>Hello {this.state.value}!</label>
      </div>
    );
  }
});

ReactDOM.render(
  <Echo />,
  document.getElementById('container')
);
```

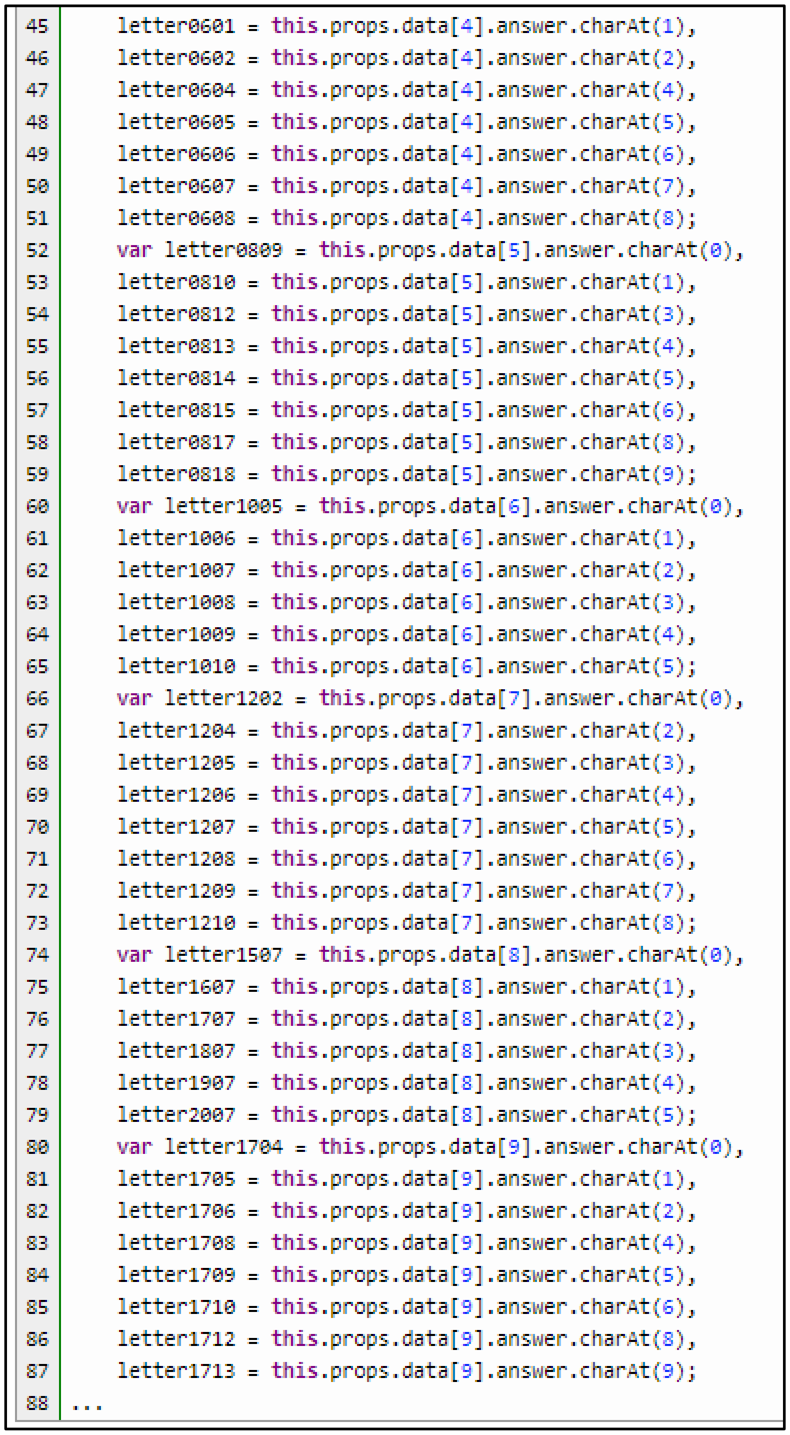
We initialized the variable value to an empty string. We included an input textbox which has an onChange attribute as well as a label component that outputs what you typed in the textbox as you type it in. The onChange event handler, named handleChange, merely sets the state of the value object.

In the puzzle game, I've removed the handleChange event handler I've had in the input in the first part of this tutorial, and instead, added a Button which has a clickHandler event.

```
<button className="btn btn-lg btn-primary" style={toRight} onClick={this.clickHandler}>Check answers</button>
```

In the clickHandler function, I created some variables to extract the letters from the answer in data.

```
clickHandler: function(e) {
    var letter0016 = this.props.data[0].answer.charAt(0),
    letter0116 = this.props.data[0].answer.charAt(1),
    letter0216 = this.props.data[0].answer.charAt(2),
    letter0316 = this.props.data[0].answer.charAt(3),
    letter0416 = this.props.data[0].answer.charAt(4),
    letter0516 = this.props.data[0].answer.charAt(5),
    letter0616 = this.props.data[0].answer.charAt(6),
    letter0716 = this.props.data[0].answer.charAt(7),
    letter0816 = this.props.data[0].answer.charAt(8);
    var letter0311 = this.props.data[1].answer.charAt(0),
    letter0411 = this.props.data[1].answer.charAt(1),
    letter0511 = this.props.data[1].answer.charAt(2),
    letter0611 = this.props.data[1].answer.charAt(3),
    letter0711 = this.props.data[1].answer.charAt(4),
    letter0811 = this.props.data[1].answer.charAt(5),
    letter0911 = this.props.data[1].answer.charAt(6),
    letter1011 = this.props.data[1].answer.charAt(7),
    letter1111 = this.props.data[1].answer.charAt(8),
    letter1211 = this.props.data[1].answer.charAt(9),
    letter1311 = this.props.data[1].answer.charAt(10),
    letter1411 = this.props.data[1].answer.charAt(11),
    letter1511 = this.props.data[1].answer.charAt(12),
    letter1611 = this.props.data[1].answer.charAt(13),
    letter1711 = this.props.data[1].answer.charAt(14),
    letter1811 = this.props.data[1].answer.charAt(15);
    var letter0503 = this.props.data[2].answer.charAt(0),
    letter0603 = this.props.data[2].answer.charAt(1),
    letter0703 = this.props.data[2].answer.charAt(2),
    letter0803 = this.props.data[2].answer.charAt(3),
    letter0903 = this.props.data[2].answer.charAt(4),
    letter1003 = this.props.data[2].answer.charAt(5),
    letter1103 = this.props.data[2].answer.charAt(6),
    letter1203 = this.props.data[2].answer.charAt(7),
    letter1303 = this.props.data[2].answer.charAt(8);
    var letter0512 = this.props.data[3].answer.charAt(1),
    letter0513 = this.props.data[3].answer.charAt(2),
    letter0514 = this.props.data[3].answer.charAt(3),
    letter0515 = this.props.data[3].answer.charAt(4),
    letter0517 = this.props.data[3].answer.charAt(6),
    letter0518 = this.props.data[3].answer.charAt(7),
    letter0519 = this.props.data[3].answer.charAt(8),
    letter0520 = this.props.data[3].answer.charAt(9);
    var letter0600 = this.props.data[4].answer.charAt(0),
```



```
    letter0601 = this.props.data[4].answer.charAt(1),
    letter0602 = this.props.data[4].answer.charAt(2),
    letter0604 = this.props.data[4].answer.charAt(4),
    letter0605 = this.props.data[4].answer.charAt(5),
    letter0606 = this.props.data[4].answer.charAt(6),
    letter0607 = this.props.data[4].answer.charAt(7),
    letter0608 = this.props.data[4].answer.charAt(8);
    var letter0809 = this.props.data[5].answer.charAt(0),
    letter0810 = this.props.data[5].answer.charAt(1),
    letter0812 = this.props.data[5].answer.charAt(3),
    letter0813 = this.props.data[5].answer.charAt(4),
    letter0814 = this.props.data[5].answer.charAt(5),
    letter0815 = this.props.data[5].answer.charAt(6),
    letter0817 = this.props.data[5].answer.charAt(8),
    letter0818 = this.props.data[5].answer.charAt(9);
    var letter1005 = this.props.data[6].answer.charAt(0),
    letter1006 = this.props.data[6].answer.charAt(1),
    letter1007 = this.props.data[6].answer.charAt(2),
    letter1008 = this.props.data[6].answer.charAt(3),
    letter1009 = this.props.data[6].answer.charAt(4),
    letter1010 = this.props.data[6].answer.charAt(5);
    var letter1202 = this.props.data[7].answer.charAt(0),
    letter1204 = this.props.data[7].answer.charAt(2),
    letter1205 = this.props.data[7].answer.charAt(3),
    letter1206 = this.props.data[7].answer.charAt(4),
    letter1207 = this.props.data[7].answer.charAt(5),
    letter1208 = this.props.data[7].answer.charAt(6),
    letter1209 = this.props.data[7].answer.charAt(7),
    letter1210 = this.props.data[7].answer.charAt(8);
    var letter1507 = this.props.data[8].answer.charAt(0),
    letter1607 = this.props.data[8].answer.charAt(1),
    letter1707 = this.props.data[8].answer.charAt(2),
    letter1807 = this.props.data[8].answer.charAt(3),
    letter1907 = this.props.data[8].answer.charAt(4),
    letter2007 = this.props.data[8].answer.charAt(5);
    var letter1704 = this.props.data[9].answer.charAt(0),
    letter1705 = this.props.data[9].answer.charAt(1),
    letter1706 = this.props.data[9].answer.charAt(2),
    letter1708 = this.props.data[9].answer.charAt(4),
    letter1709 = this.props.data[9].answer.charAt(5),
    letter1710 = this.props.data[9].answer.charAt(6),
    letter1712 = this.props.data[9].answer.charAt(8),
    letter1713 = this.props.data[9].answer.charAt(9);
...
```

Forms

React has some very basic form support built-in, which includes binding values to the form, binding values back out of the form and accessing individual form elements. In this Puzzle game, each white box has an input element within them. The input boxes are there for the input of a character for the crossword.

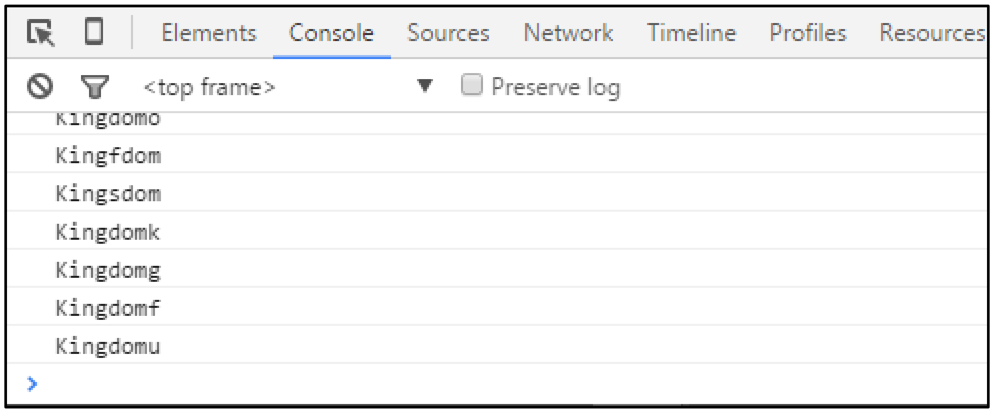
```
<td className="cell cell16">
  <input type="text"
      ref="answer0016"
      placeholder="1"
      maxLength="1" />
</td>
```

You will notice there is no value attribute for each <input>. If the <input> element has value, then it would have been a *controlled* component. We are not required to have a controlled component, in this instance.

Let’s take a simple forms input example to test on JSFiddle.

```
var Puzzle = React.createClass({
  render: function() {
    return <input type="text" value="Kingdom" />;
  }
});
```

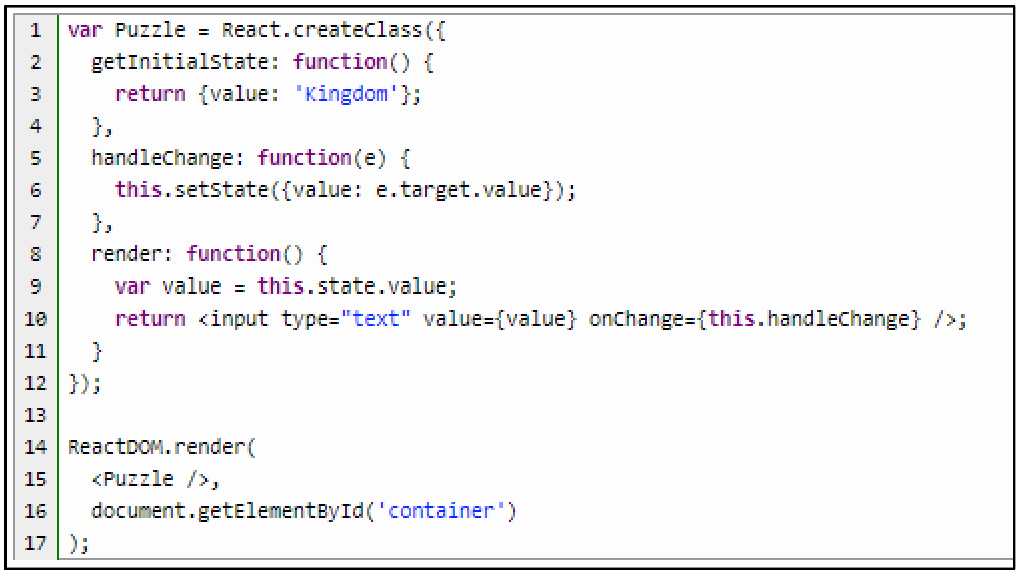
As you type a character, anywhere in the input box, you will notice that the changed value is being written to the console. To view the changes in the console, open your Developer Tools in Chrome or Web Console in Firefox.



```
Kingdomo
Kingfdom
Kingsdom
Kingdomk
Kingdomg
Kingdomf
Kingdomu
```

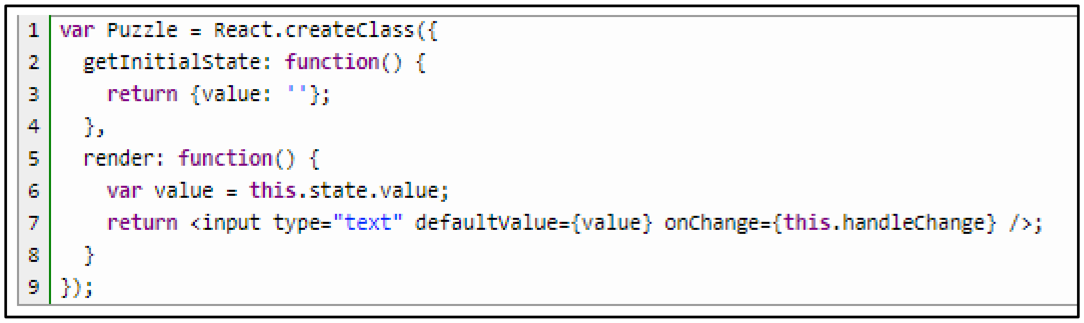
However, this immediately triggers a re-render for the component which resets the value. From the user's point of view, they are typing, but nothing is happening. The secret to making this work is to populate the input boxes value from the owning component's state. This is consistent with the React component design guidance because the value of the input box can change, therefore, it is state.

To allow the user to change the content of the input box we need to update the onChange handle to modify the state. We also initialize the value in a getInitialState function.



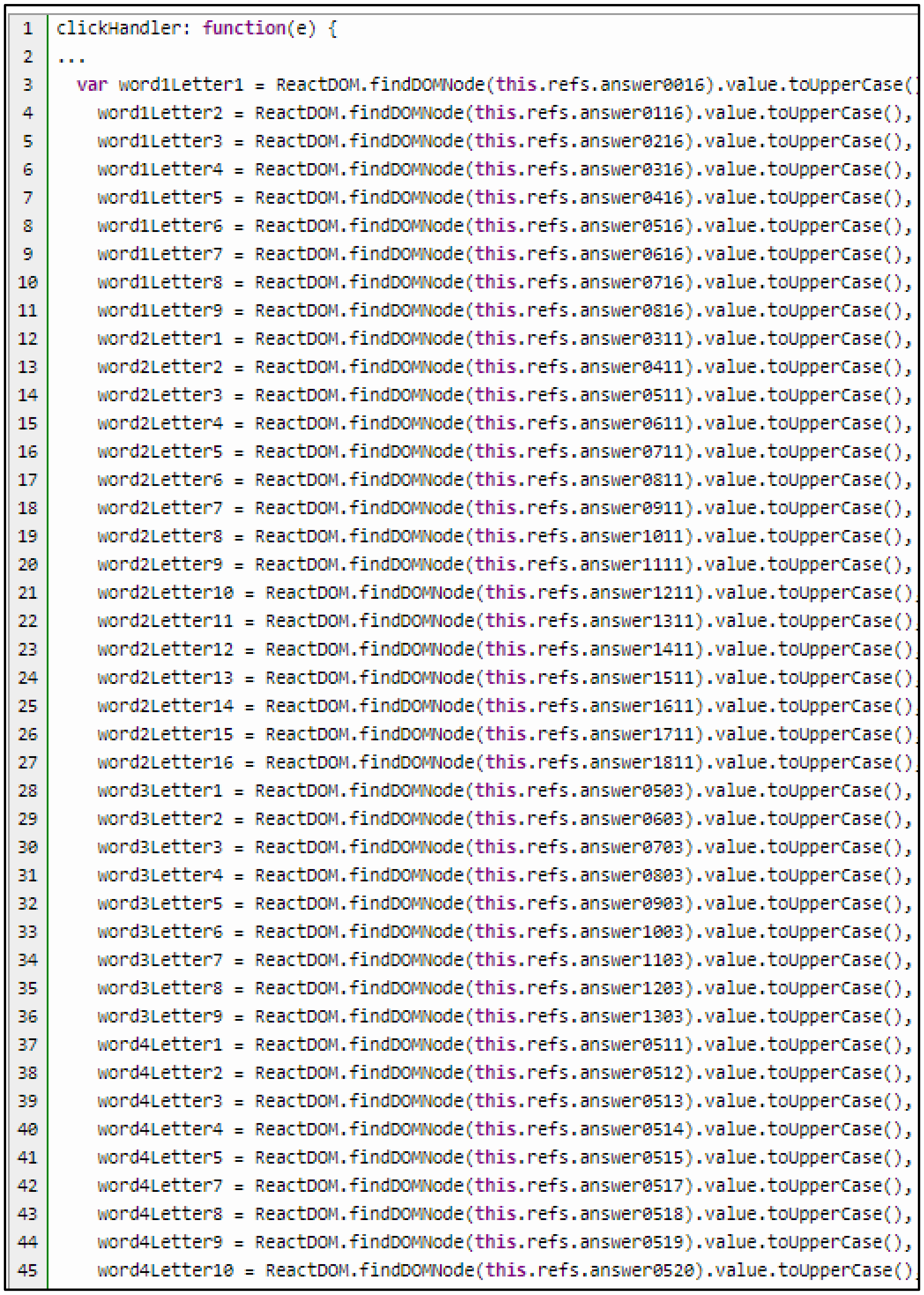
```
var Puzzle = React.createClass({
  getInitialState: function() {
    return {value: 'Kingdom'};
  },
  handleChange: function(e) {
    this.setState({value: e.target.value});
  },
  render: function() {
    var value = this.state.value;
    return <input type="text" value={value} onChange={this.handleChange} />;
  }
});

ReactDOM.render(
  <Puzzle />,
  document.getElementById('container')
);
```

Since we will be rendering our components without a value, this is referred to as *uncontrolled* components. Using uncontrolled components tells React to exempt the value of the control from the usual rendering process. We can still handle the onChange event to detect changes to the input, however. If you still want to initialize the component with a non-empty value, you can supply a defaultValue prop, which gives the input box a starting value.

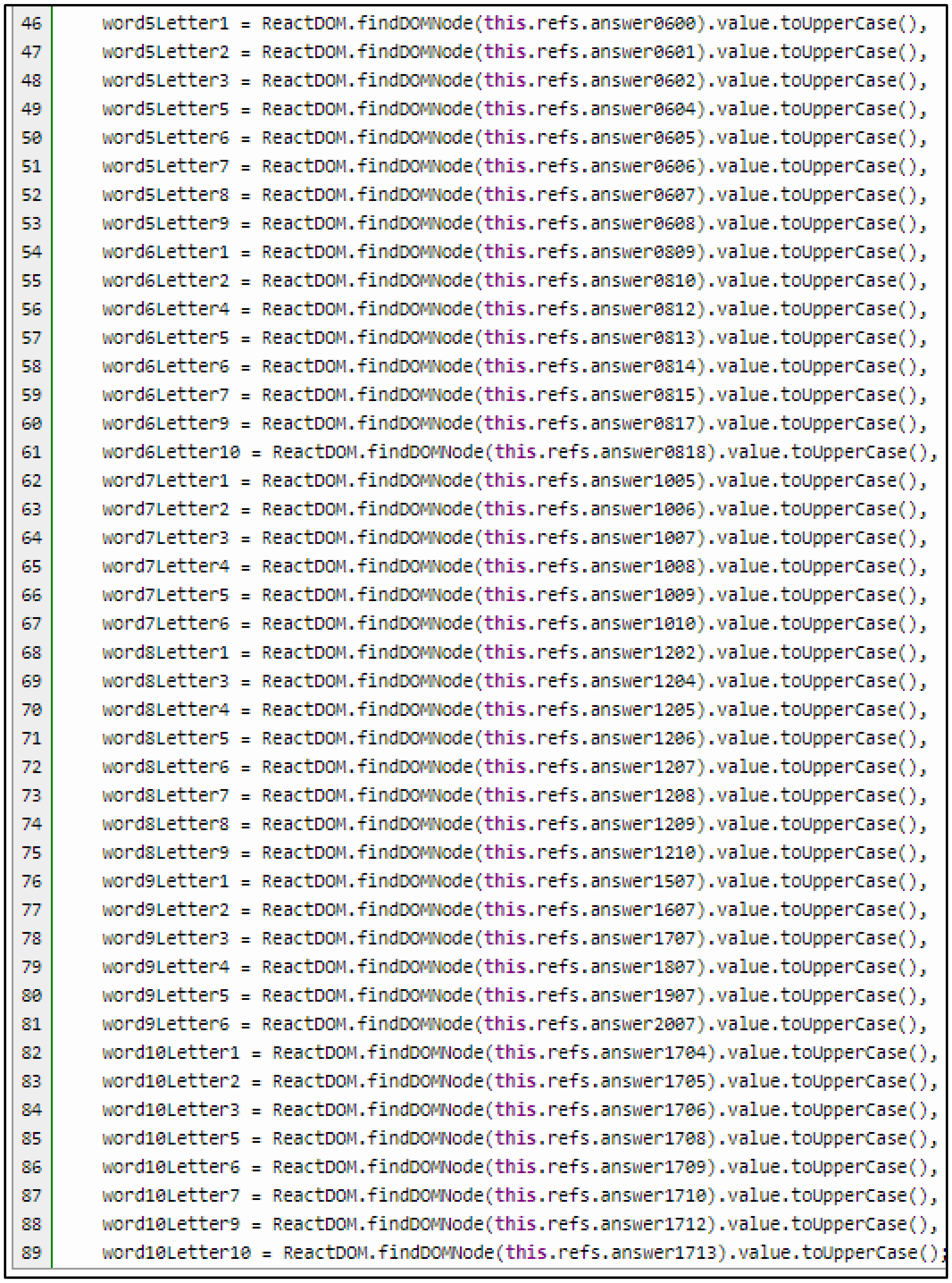


```
var Puzzle = React.createClass({
  getInitialState: function() {
    return {value: ''};
  },
  render: function() {
    var value = this.state.value;
    return <input type="text" defaultValue={value} onChange={this.handleChange} />;
  }
});
```

Now to complete the onClick event handler.

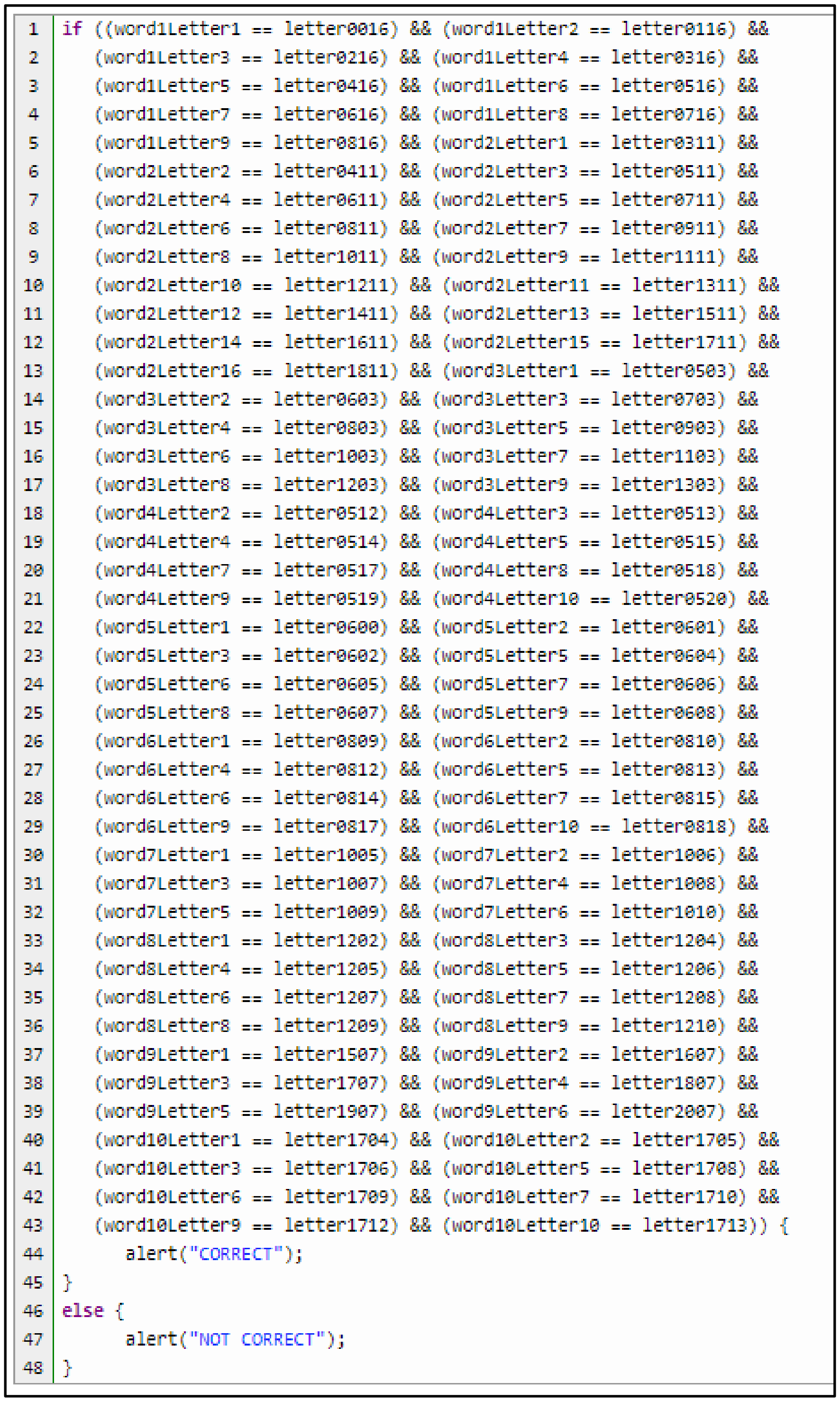


```
clickHandler: function(e) {
...
  var word1Letter1 = ReactDOM.findDOMNode(this.refs.answer0016).value.toUpperCase(
    word1Letter2 = ReactDOM.findDOMNode(this.refs.answer0116).value.toUpperCase(),
    word1Letter3 = ReactDOM.findDOMNode(this.refs.answer0216).value.toUpperCase(),
    word1Letter4 = ReactDOM.findDOMNode(this.refs.answer0316).value.toUpperCase(),
    word1Letter5 = ReactDOM.findDOMNode(this.refs.answer0416).value.toUpperCase(),
    word1Letter6 = ReactDOM.findDOMNode(this.refs.answer0516).value.toUpperCase(),
    word1Letter7 = ReactDOM.findDOMNode(this.refs.answer0616).value.toUpperCase(),
    word1Letter8 = ReactDOM.findDOMNode(this.refs.answer0716).value.toUpperCase(),
    word1Letter9 = ReactDOM.findDOMNode(this.refs.answer0816).value.toUpperCase(),
    word2Letter1 = ReactDOM.findDOMNode(this.refs.answer0311).value.toUpperCase(),
    word2Letter2 = ReactDOM.findDOMNode(this.refs.answer0411).value.toUpperCase(),
    word2Letter3 = ReactDOM.findDOMNode(this.refs.answer0511).value.toUpperCase(),
    word2Letter4 = ReactDOM.findDOMNode(this.refs.answer0611).value.toUpperCase(),
    word2Letter5 = ReactDOM.findDOMNode(this.refs.answer0711).value.toUpperCase(),
    word2Letter6 = ReactDOM.findDOMNode(this.refs.answer0811).value.toUpperCase(),
    word2Letter7 = ReactDOM.findDOMNode(this.refs.answer0911).value.toUpperCase(),
    word2Letter8 = ReactDOM.findDOMNode(this.refs.answer1011).value.toUpperCase(),
    word2Letter9 = ReactDOM.findDOMNode(this.refs.answer1111).value.toUpperCase(),
    word2Letter10 = ReactDOM.findDOMNode(this.refs.answer1211).value.toUpperCase()
    word2Letter11 = ReactDOM.findDOMNode(this.refs.answer1311).value.toUpperCase()
    word2Letter12 = ReactDOM.findDOMNode(this.refs.answer1411).value.toUpperCase()
    word2Letter13 = ReactDOM.findDOMNode(this.refs.answer1511).value.toUpperCase()
    word2Letter14 = ReactDOM.findDOMNode(this.refs.answer1611).value.toUpperCase()
    word2Letter15 = ReactDOM.findDOMNode(this.refs.answer1711).value.toUpperCase()
    word2Letter16 = ReactDOM.findDOMNode(this.refs.answer1811).value.toUpperCase()
    word3Letter1 = ReactDOM.findDOMNode(this.refs.answer0503).value.toUpperCase(),
    word3Letter2 = ReactDOM.findDOMNode(this.refs.answer0603).value.toUpperCase(),
    word3Letter3 = ReactDOM.findDOMNode(this.refs.answer0703).value.toUpperCase(),
    word3Letter4 = ReactDOM.findDOMNode(this.refs.answer0803).value.toUpperCase(),
    word3Letter5 = ReactDOM.findDOMNode(this.refs.answer0903).value.toUpperCase(),
    word3Letter6 = ReactDOM.findDOMNode(this.refs.answer1003).value.toUpperCase(),
    word3Letter7 = ReactDOM.findDOMNode(this.refs.answer1103).value.toUpperCase(),
    word3Letter8 = ReactDOM.findDOMNode(this.refs.answer1203).value.toUpperCase(),
    word3Letter9 = ReactDOM.findDOMNode(this.refs.answer1303).value.toUpperCase(),
    word4Letter1 = ReactDOM.findDOMNode(this.refs.answer0511).value.toUpperCase(),
    word4Letter2 = ReactDOM.findDOMNode(this.refs.answer0512).value.toUpperCase(),
    word4Letter3 = ReactDOM.findDOMNode(this.refs.answer0513).value.toUpperCase(),
    word4Letter4 = ReactDOM.findDOMNode(this.refs.answer0514).value.toUpperCase(),
    word4Letter5 = ReactDOM.findDOMNode(this.refs.answer0515).value.toUpperCase(),
    word4Letter7 = ReactDOM.findDOMNode(this.refs.answer0517).value.toUpperCase(),
    word4Letter8 = ReactDOM.findDOMNode(this.refs.answer0518).value.toUpperCase(),
    word4Letter9 = ReactDOM.findDOMNode(this.refs.answer0519).value.toUpperCase(),
    word4Letter10 = ReactDOM.findDOMNode(this.refs.answer0520).value.toUpperCase()
```



```
    word5Letter1 = ReactDOM.findDOMNode(this.refs.answer0600).value.toUpperCase(),
    word5Letter2 = ReactDOM.findDOMNode(this.refs.answer0601).value.toUpperCase(),
    word5Letter3 = ReactDOM.findDOMNode(this.refs.answer0602).value.toUpperCase(),
    word5Letter5 = ReactDOM.findDOMNode(this.refs.answer0604).value.toUpperCase(),
    word5Letter6 = ReactDOM.findDOMNode(this.refs.answer0605).value.toUpperCase(),
    word5Letter7 = ReactDOM.findDOMNode(this.refs.answer0606).value.toUpperCase(),
    word5Letter8 = ReactDOM.findDOMNode(this.refs.answer0607).value.toUpperCase(),
    word5Letter9 = ReactDOM.findDOMNode(this.refs.answer0608).value.toUpperCase(),
    word6Letter1 = ReactDOM.findDOMNode(this.refs.answer0809).value.toUpperCase(),
    word6Letter2 = ReactDOM.findDOMNode(this.refs.answer0810).value.toUpperCase(),
    word6Letter4 = ReactDOM.findDOMNode(this.refs.answer0812).value.toUpperCase(),
    word6Letter5 = ReactDOM.findDOMNode(this.refs.answer0813).value.toUpperCase(),
    word6Letter6 = ReactDOM.findDOMNode(this.refs.answer0814).value.toUpperCase(),
    word6Letter7 = ReactDOM.findDOMNode(this.refs.answer0815).value.toUpperCase(),
    word6Letter9 = ReactDOM.findDOMNode(this.refs.answer0817).value.toUpperCase(),
    word6Letter10 = ReactDOM.findDOMNode(this.refs.answer0818).value.toUpperCase(),
    word7Letter1 = ReactDOM.findDOMNode(this.refs.answer1005).value.toUpperCase(),
    word7Letter2 = ReactDOM.findDOMNode(this.refs.answer1006).value.toUpperCase(),
    word7Letter3 = ReactDOM.findDOMNode(this.refs.answer1007).value.toUpperCase(),
    word7Letter4 = ReactDOM.findDOMNode(this.refs.answer1008).value.toUpperCase(),
    word7Letter5 = ReactDOM.findDOMNode(this.refs.answer1009).value.toUpperCase(),
    word7Letter6 = ReactDOM.findDOMNode(this.refs.answer1010).value.toUpperCase(),
    word8Letter1 = ReactDOM.findDOMNode(this.refs.answer1202).value.toUpperCase(),
    word8Letter3 = ReactDOM.findDOMNode(this.refs.answer1204).value.toUpperCase(),
    word8Letter4 = ReactDOM.findDOMNode(this.refs.answer1205).value.toUpperCase(),
    word8Letter5 = ReactDOM.findDOMNode(this.refs.answer1206).value.toUpperCase(),
    word8Letter6 = ReactDOM.findDOMNode(this.refs.answer1207).value.toUpperCase(),
    word8Letter7 = ReactDOM.findDOMNode(this.refs.answer1208).value.toUpperCase(),
    word8Letter8 = ReactDOM.findDOMNode(this.refs.answer1209).value.toUpperCase(),
    word8Letter9 = ReactDOM.findDOMNode(this.refs.answer1210).value.toUpperCase(),
    word9Letter1 = ReactDOM.findDOMNode(this.refs.answer1507).value.toUpperCase(),
    word9Letter2 = ReactDOM.findDOMNode(this.refs.answer1607).value.toUpperCase(),
    word9Letter3 = ReactDOM.findDOMNode(this.refs.answer1707).value.toUpperCase(),
    word9Letter4 = ReactDOM.findDOMNode(this.refs.answer1807).value.toUpperCase(),
    word9Letter5 = ReactDOM.findDOMNode(this.refs.answer1907).value.toUpperCase(),
    word9Letter6 = ReactDOM.findDOMNode(this.refs.answer2007).value.toUpperCase(),
    word10Letter1 = ReactDOM.findDOMNode(this.refs.answer1704).value.toUpperCase(),
    word10Letter2 = ReactDOM.findDOMNode(this.refs.answer1705).value.toUpperCase(),
    word10Letter3 = ReactDOM.findDOMNode(this.refs.answer1706).value.toUpperCase(),
    word10Letter5 = ReactDOM.findDOMNode(this.refs.answer1708).value.toUpperCase(),
    word10Letter6 = ReactDOM.findDOMNode(this.refs.answer1709).value.toUpperCase(),
    word10Letter7 = ReactDOM.findDOMNode(this.refs.answer1710).value.toUpperCase(),
    word10Letter9 = ReactDOM.findDOMNode(this.refs.answer1712).value.toUpperCase(),
    word10Letter10 = ReactDOM.findDOMNode(this.refs.answer1713).value.toUpperCase();
```

Finally, we check that the letter the user types in is the correct answer to the clues.



```
if ((word1Letter1 == letter0016) && (word1Letter2 == letter0116) &&
   (word1Letter3 == letter0216) && (word1Letter4 == letter0316) &&
   (word1Letter5 == letter0416) && (word1Letter6 == letter0516) &&
   (word1Letter7 == letter0616) && (word1Letter8 == letter0716) &&
   (word1Letter9 == letter0816) && (word2Letter1 == letter0311) &&
   (word2Letter2 == letter0411) && (word2Letter3 == letter0511) &&
   (word2Letter4 == letter0611) && (word2Letter5 == letter0711) &&
   (word2Letter6 == letter0811) && (word2Letter7 == letter0911) &&
   (word2Letter8 == letter1011) && (word2Letter9 == letter1111) &&
   (word2Letter10 == letter1211) && (word2Letter11 == letter1311) &&
   (word2Letter12 == letter1411) && (word2Letter13 == letter1511) &&
   (word2Letter14 == letter1611) && (word2Letter15 == letter1711) &&
   (word2Letter16 == letter1811) && (word3Letter1 == letter0503) &&
   (word3Letter2 == letter0603) && (word3Letter3 == letter0703) &&
   (word3Letter4 == letter0803) && (word3Letter5 == letter0903) &&
   (word3Letter6 == letter1003) && (word3Letter7 == letter1103) &&
   (word3Letter8 == letter1203) && (word3Letter9 == letter1303) &&
   (word4Letter2 == letter0512) && (word4Letter3 == letter0513) &&
   (word4Letter4 == letter0514) && (word4Letter5 == letter0515) &&
   (word4Letter7 == letter0517) && (word4Letter8 == letter0518) &&
   (word4Letter9 == letter0519) && (word4Letter10 == letter0520) &&
   (word5Letter1 == letter0600) && (word5Letter2 == letter0601) &&
   (word5Letter3 == letter0602) && (word5Letter5 == letter0604) &&
   (word5Letter6 == letter0605) && (word5Letter7 == letter0606) &&
   (word5Letter8 == letter0607) && (word5Letter9 == letter0608) &&
   (word6Letter1 == letter0809) && (word6Letter2 == letter0810) &&
   (word6Letter4 == letter0812) && (word6Letter5 == letter0813) &&
   (word6Letter6 == letter0814) && (word6Letter7 == letter0815) &&
   (word6Letter9 == letter0817) && (word6Letter10 == letter0818) &&
   (word7Letter1 == letter1005) && (word7Letter2 == letter1006) &&
   (word7Letter3 == letter1007) && (word7Letter4 == letter1008) &&
   (word7Letter5 == letter1009) && (word7Letter6 == letter1010) &&
   (word8Letter1 == letter1202) && (word8Letter3 == letter1204) &&
   (word8Letter4 == letter1205) && (word8Letter5 == letter1206) &&
   (word8Letter6 == letter1207) && (word8Letter7 == letter1208) &&
   (word8Letter8 == letter1209) && (word8Letter9 == letter1210) &&
   (word9Letter1 == letter1507) && (word9Letter2 == letter1607) &&
   (word9Letter3 == letter1707) && (word9Letter4 == letter1807) &&
   (word9Letter5 == letter1907) && (word9Letter6 == letter2007) &&
   (word10Letter1 == letter1704) && (word10Letter2 == letter1705) &&
   (word10Letter3 == letter1706) && (word10Letter5 == letter1708) &&
   (word10Letter6 == letter1709) && (word10Letter7 == letter1710) &&
   (word10Letter9 == letter1712) && (word10Letter10 == letter1713)) {
      alert("CORRECT");
}
else {
      alert("NOT CORRECT");
}
```

I just demonstrated a basic way of creating a crossword puzzle with React. There is more that can be done here, like adding dynamic styles if a word is wrong. What else can you do to improve on this crossword puzzle game?