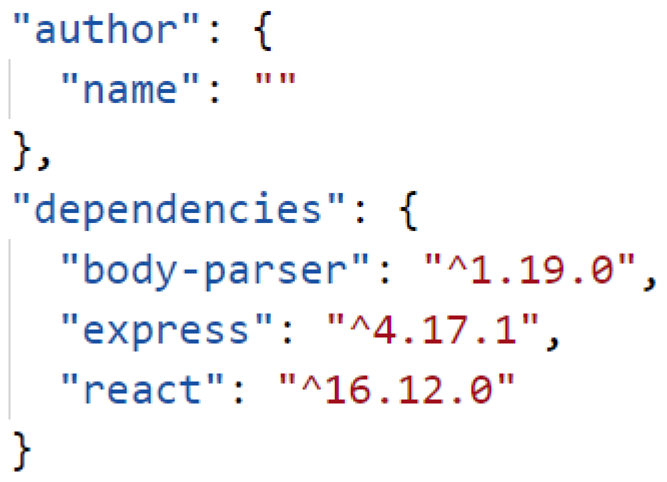
Supplemental Lab 3

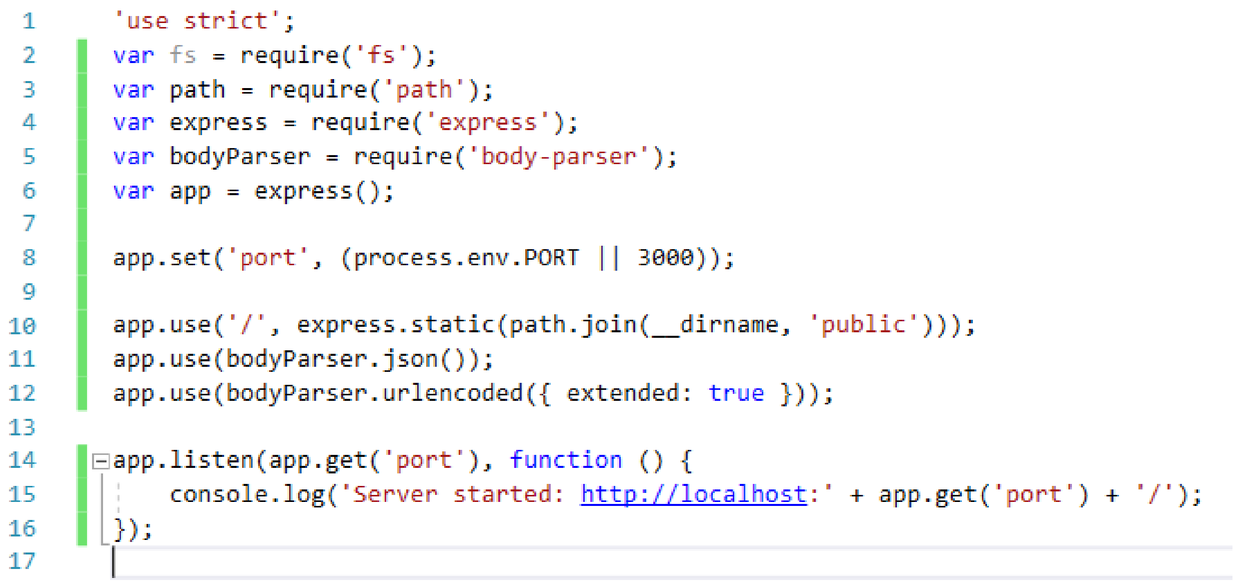
For Lab 3 we will be focusing on input validation. You will create a folder named lastname_lab3. We will be starting this project in class, so you may have some of the first steps completed. If you started this project in class, skip down to Part 2 below, otherwise start here at Part 1:

Part 1: For this project we will be creating a form for inserting customers. Once the project has been created, make sure to create a folder under solution explorer for public, then a folder under that for scripts. Create the insertcustomer.html and insertemployee.html pages in the public folder. Then create the insertemployee.js and insertcustomer.js files inside the scripts folder. From here, open up a terminal and run npm init. Change the file name of the server file from index.js to server.js. Once that is complete go into the package.json file and add the dependencies for react and express and body-parser. The code for what the dependencies are should look like is below on the left. After the package.json file is saved, run the command npm install in the terminal. Once that is complete, the basic setup of the project should be done.



```
"author": {
  "name": ""
},
"dependencies": {
  "body-parser": "^1.19.0",
  "express": "^4.17.1",
  "react": "^16.12.0"
}
```

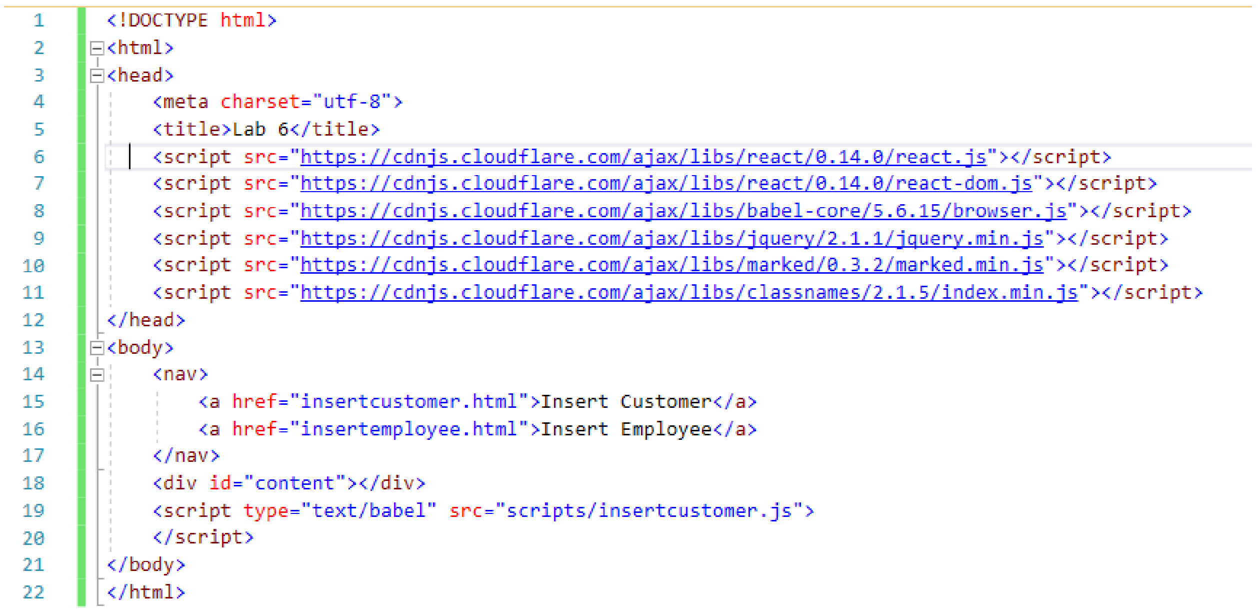
Begin by opening the server.js file, this is where the js code that will initialize the project will be created. The code that will need to be input into the file is below. Line 1 just tells the program to be strict in how it deals with variables and to now allow implicit conversion between data types. Lines 2-6 create the variables that will be used by the program to access the specific libraries. Not all of these will be used in this lab, but some, like line 4, will be used in the next lab. Line 8 will set the processing port for the project so when it run it will know which port to run on. Line 10 will allow the project to access the public directory so all the html files we placed in there will function. Lines 11 and 12 will assist in future labs with accessing and utilizing json files for data storage. Line 14 is what starts the app and gets the node server listening on the correct port. Line 15 outputs a console log showing the name of the site and the port that is being utilized. This is all the code that is required on the server.js file



```
'use strict';
var fs = require('fs');
var path = require('path');
var express = require('express');
var bodyParser = require('body-parser');
var app = express();

app.set('port', (process.env.PORT || 3000));

app.use('/', express.static(path.join(__dirname, 'public')));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: true }));

app.listen(app.get('port'), function () {
    console.log('Server started: http://localhost:' + app.get('port') + '/');
});

```

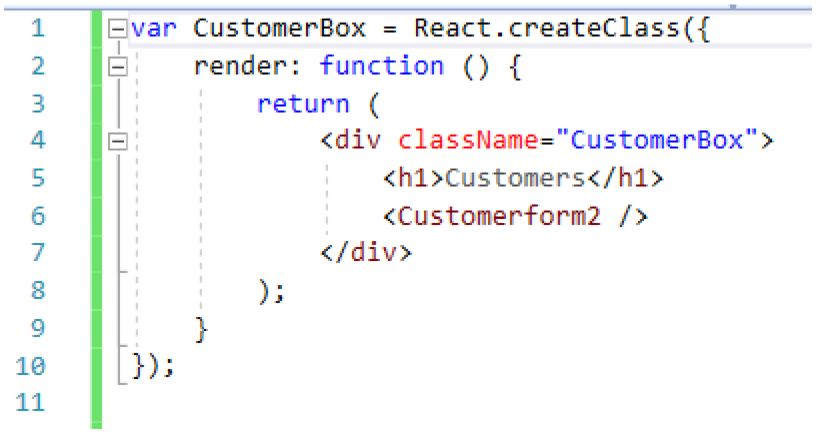
PART 2

Now that the basic format of the project has been created, we can start to adapt and modify the html and js documents in order to get the code that will show up on our pages. The first page to be modified is the insertcustomers.html page, open that up and enter the code below. This will create the basic html form, as well as links to the scripts being used by this program, which are from lines 6-11. These are all external libraries that perform specific functions. There is also a link on line 19 to the script that we will create.



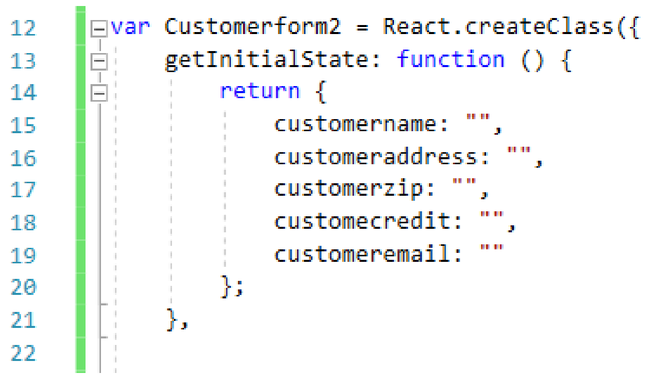
```
1   <!DOCTYPE html>
2   <html>
3   <head>
4       <meta charset="utf-8">
5       <title>Lab 6</title>
6       <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"></script>
7       <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.js"></script>
8       <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.6.15/browser.js"></script>
9       <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
10      <script src="https://cdnjs.cloudflare.com/ajax/libs/marked/0.3.2/marked.min.js"></script>
11      <script src="https://cdnjs.cloudflare.com/ajax/libs/classnames/2.1.5/index.min.js"></script>
12  </head>
13  <body>
14      <nav>
15          <a href="insertcustomer.html">Insert Customer</a>
16          <a href="insertemployee.html">Insert Employee</a>
17      </nav>
18      <div id="content"></div>
19      <script type="text/babel" src="scripts/insertcustomer.js">
20      </script>
21  </body>
22  </html>
```

Once the changes have been made to the html file, open the insertcustomer.js file. We will be creating several classes that will build the form that we will use in the program, as well as perform the input validation. The first class we will create is the CustomerBox class, which will return the basic div that will hold our form. The code below creates the class and assigns it to a variable in line 1. The only method within this class is one to render the information within the function we create to the screen. The use of the return statement on line 3 will take all the information within the parentheses of the return statement and return them as html code to whenever the CustomerBox class is instantiated. The code to be returned is the creation of a div on line 4, and h1 on line 5 and the calling of another class on line 6.



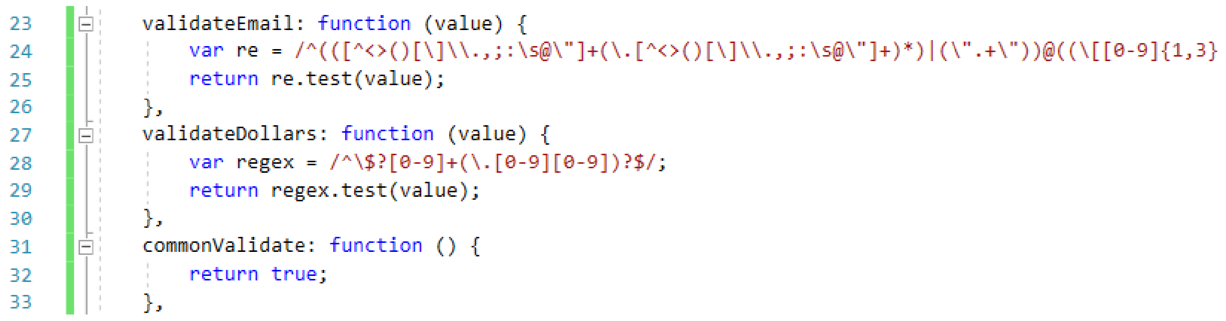
```
1   var CustomerBox = React.createClass({
2       render: function () {
3           return (
4               <div className="CustomerBox">
5                   <h1>Customers</h1>
6                   <Customerform2 />
7               </div>
8           );
9       }
10  });
11
```

The next class to be created is the one that was called on line 6 above. The code below starts the creation process for this class on line 12, and assigns the class to a variable name Customerform2. This variable must start with an uppercase letter, or this class may not run. The getInitialState method on line 13 will set the initial state of all the fields we will use in this form to whatever is returned from the function called after the semicolon. In this case lines 14-19 return the values for all the variables that are set to blank spaces.



```
12  var Customerform2 = React.createClass({
13      getInitialState: function () {
14          return {
15              customername: "",
16              customeraddress: "",
17              customerzip: "",
18              customecredit: "",
19              customeremail: ""
20          };
21      },
22
```

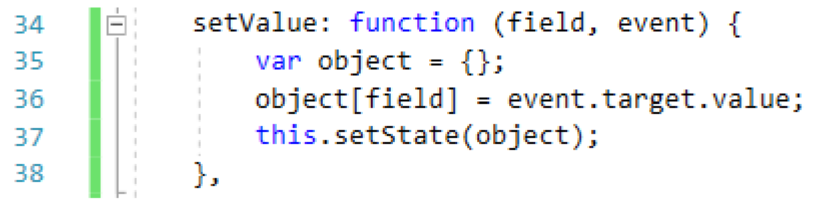
Note the comma at the end of line 21, that lets the system know this class is not done being created and there are more methods that have to be set up. The image below shows several of the validation methods being used. The validateEmail method take a variable name value as a parameter, as shown in line 23. The variable on line 24 gives a format string to compare that will recognize all required attributes of an email address. This string has been cut off in the image due to length, but the string is pasted as text below the image. Line 25 then compares the variable string to the value given the method and returns a Boolean value of true or false. This lets the system know if validation has succeeded or failed. The same process works for validate dollars, it compares the value and returns true or false. The commonValidate method does not test anything, it is a blank method we can add items to later.



```
23      validateEmail: function (value) {
24          var re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-9]{1,3}
25          return re.test(value);
26      },
27      validateDollars: function (value) {
28          var regex = /^\$?[0-9]+(\.[0-9][0-9])?$/;
29          return regex.test(value);
30      },
31      commonValidate: function () {
32          return true;
33      },
```

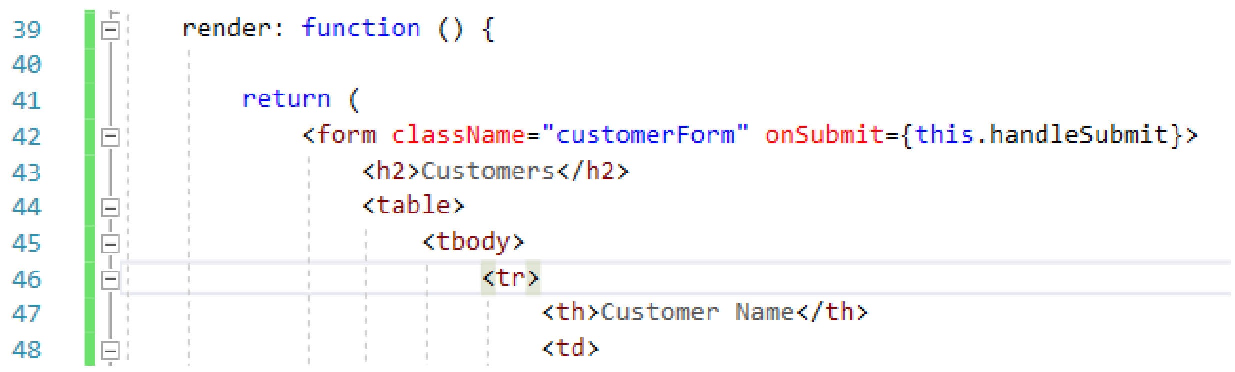
```
var re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"))@((\[[0-
9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))$/;
```

The next method to be created in this class is setValue. This function takes a field and an event as parameters on line 34. Line 35 creates a holder array named object. Then on line 36, this array is populated using the field name as an index value. The value of the item is the value of the target field. Line 37 sets the state of the object to this, which is the referring method.



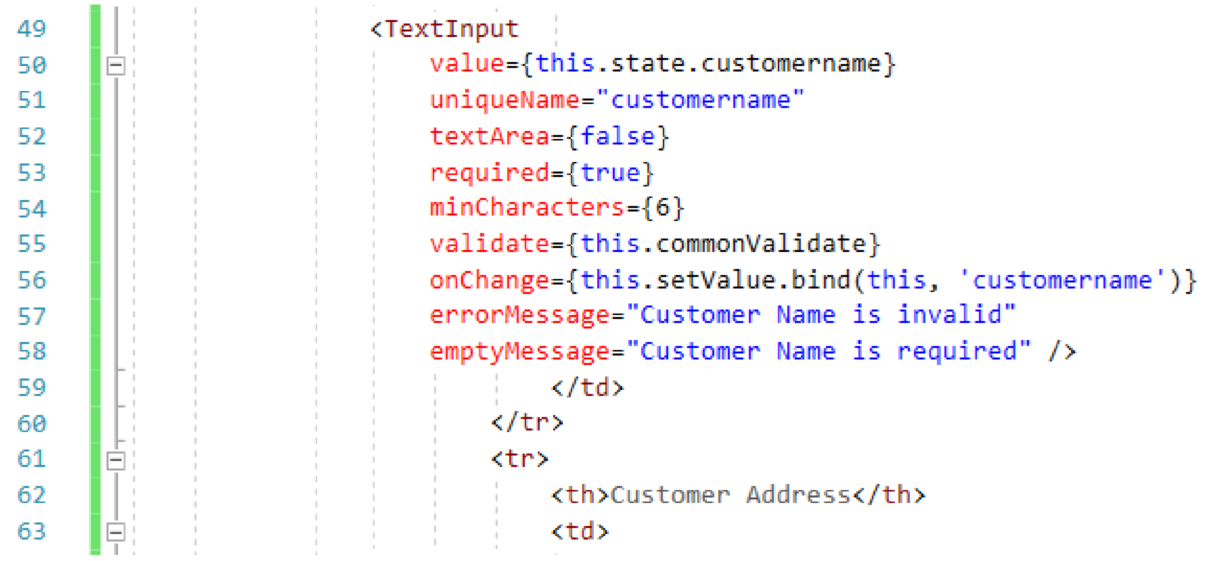
```
34      setValue: function (field, event) {
35          var object = {};
36          object[field] = event.target.value;
37          this.setState(object);
38      },
```

It is not time to render the information for this class. We use the render method again, and create a function the will return the information back to the object call used back in line 6. Lines 42-48 are just basic html that will be rendered to the screen. Line 42 does have a link to a method named handleSubmit which will only occur onSubmit of the form. This will be used in later labs.



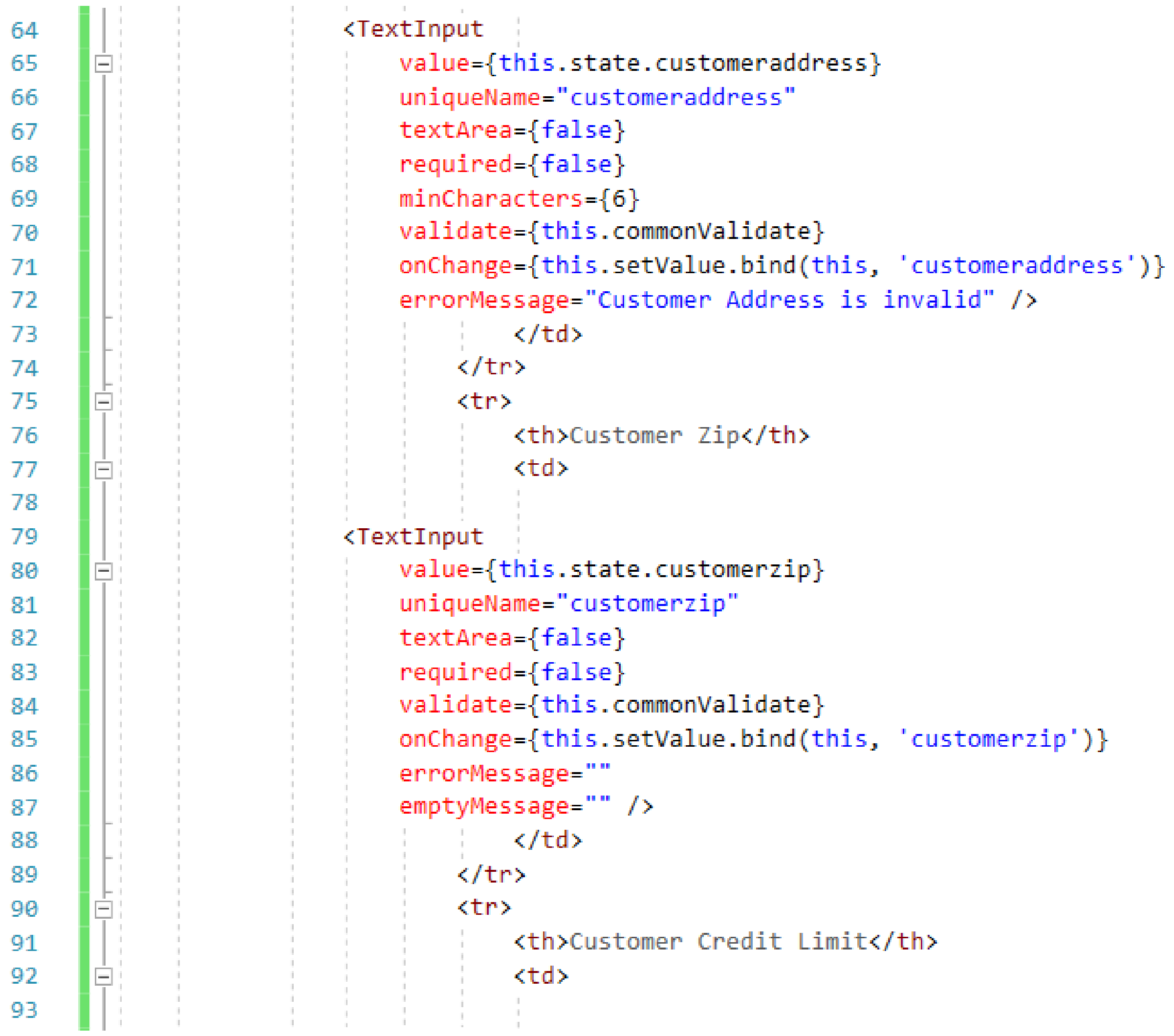
```
39      render: function () {
40
41          return (
42              <form className="customerForm" onSubmit={this.handleSubmit}>
43                  <h2>Customers</h2>
44                  <table>
45                      <tbody>
46                          <tr>
47                              <th>Customer Name</th>
48                              <td>
```

The code below creates and instantiates an object of the type TextInput. This variable type we will be creating further along during this lab. This object type takes several arguments, which are given in the code below. The method is instantiated in line 49, note we do not put the closing to the tag on this line. Line 50 sets the value of the TextInput object to the state of the referring object which is named customername. Line 51 gives this object the name of customername so it can be referred back to by the program. Line 52 lets the system know it is not a textArea, line 53 makes it required and line 54 makes the minimum characters 6. Line 55 tells the validation function to use the commonValidate method we created up above. Line 56 sets an onChange option, so that when a change is made to this object, it sets the value and binds it to customername. Lines 57 and 58 just tell what the error messages should be if the field is shown to be invalid or not entered. Notice the closing tag at the end of line 58, that closes the starting TextInput tag from line 49. This lets this object be fully instantiated with all the options given in lines 50 to 58. Lines 59 through 63 are there to help with formatting the table the form is created in.



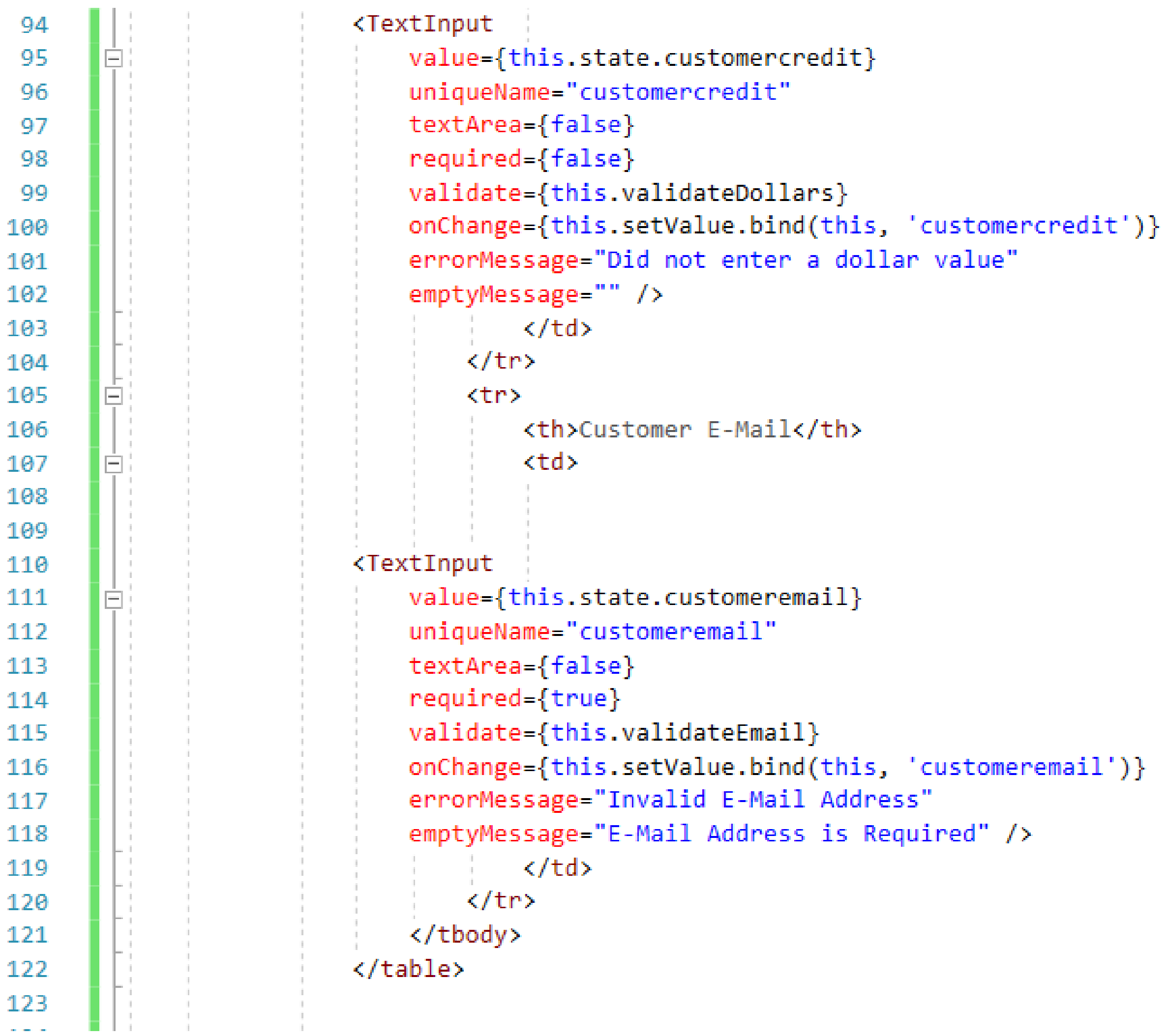
```
49  <TextInput
50      value={this.state.customername}
51      uniqueName="customername"
52      textArea={false}
53      required={true}
54      minCharacters={6}
55      validate={this.commonValidate}
56      onChange={this.setValue.bind(this, 'customername')}
57      errorMessage="Customer Name is invalid"
58      emptyMessage="Customer Name is required" />
59          </td>
60      </tr>
61      <tr>
62          <th>Customer Address</th>
63          <td>
```

The next code creates two more TextInput fields, one for customer address and one for customer zip. There are not many differences between the name field above and these two fields. The main difference is that neither of these fields are set to required. Therefore, the emptyMessage line of code from the customer name is not used within these two fields. Notice both are again created with the TextInput tag, but the tag is not closed until all the options have been entered.

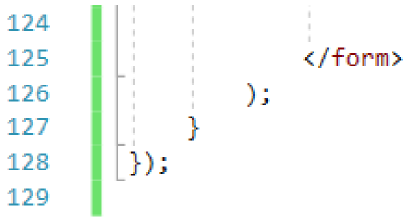


```
<TextInput
    value={this.state.customeraddress}
    uniqueName="customeraddress"
    textArea={false}
    required={false}
    minCharacters={6}
    validate={this.commonValidate}
    onChange={this.setValue.bind(this, 'customeraddress')}
    errorMessage="Customer Address is invalid" />
            </td>
        </tr>
        <tr>
            <th>Customer Zip</th>
            <td>

<TextInput
    value={this.state.customerzip}
    uniqueName="customerzip"
    textArea={false}
    required={false}
    validate={this.commonValidate}
    onChange={this.setValue.bind(this, 'customerzip')}
    errorMessage=""
    emptyMessage="" />
            </td>
        </tr>
        <tr>
            <th>Customer Credit Limit</th>
            <td>

```

The last two TextInput objects to be created are below. Note that for customercredit, it uses the validateDollars method from above to ensure a numerical or dollar value has been entered. The customeremail field users the validateEmail method to ensure the correct format of the email address. Lines 119 to 122 end the table which holds the five fields we are using on this form. Line 124 ends the form that we created above, while line 126 closes the method for return. Line 127 closes the render function, while line 128 closes the Customerform2 class created way back on line 12

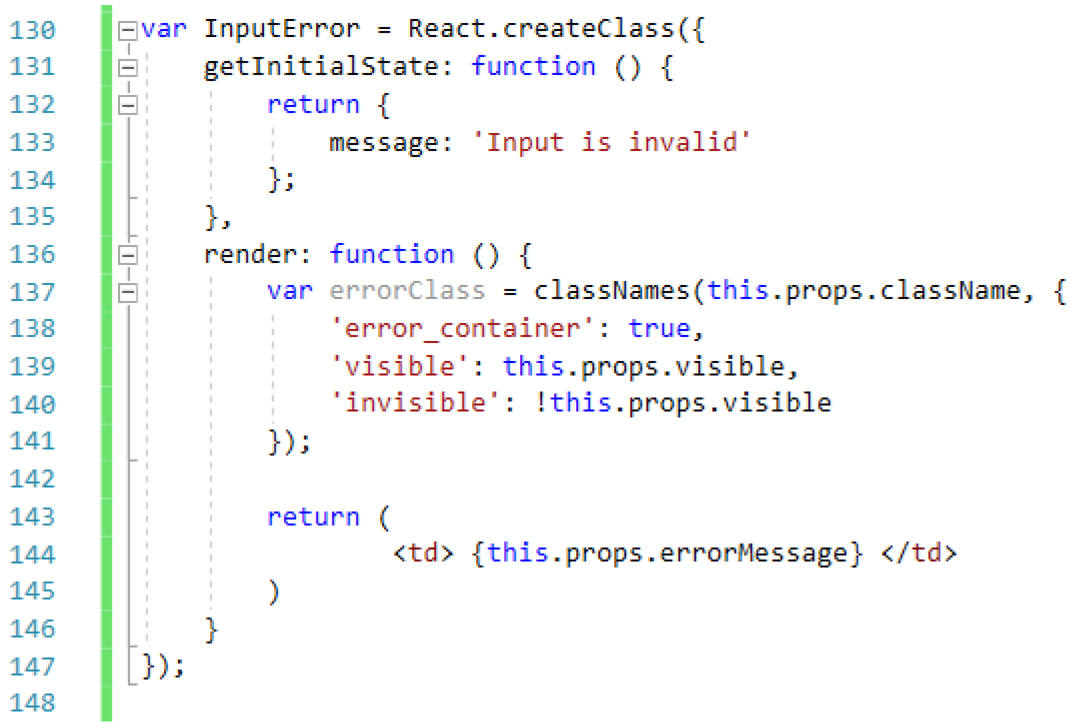


```
94                  <TextInput
95                      value={this.state.customercredit}
96                      uniqueName="customercredit"
97                      textArea={false}
98                      required={false}
99                      validate={this.validateDollars}
100                     onChange={this.setValue.bind(this, 'customercredit')}
101                     errorMessage="Did not enter a dollar value"
102                     emptyMessage="" />
103                             </td>
104                         </tr>
105                         <tr>
106                             <th>Customer E-Mail</th>
107                             <td>
108
109
110                 <TextInput
111                     value={this.state.customeremail}
112                     uniqueName="customeremail"
113                     textArea={false}
114                     required={true}
115                     validate={this.validateEmail}
116                     onChange={this.setValue.bind(this, 'customeremail')}
117                     errorMessage="Invalid E-Mail Address"
118                     emptyMessage="E-Mail Address is Required" />
119                             </td>
120                         </tr>
121                     </tbody>
122                 </table>
123
```



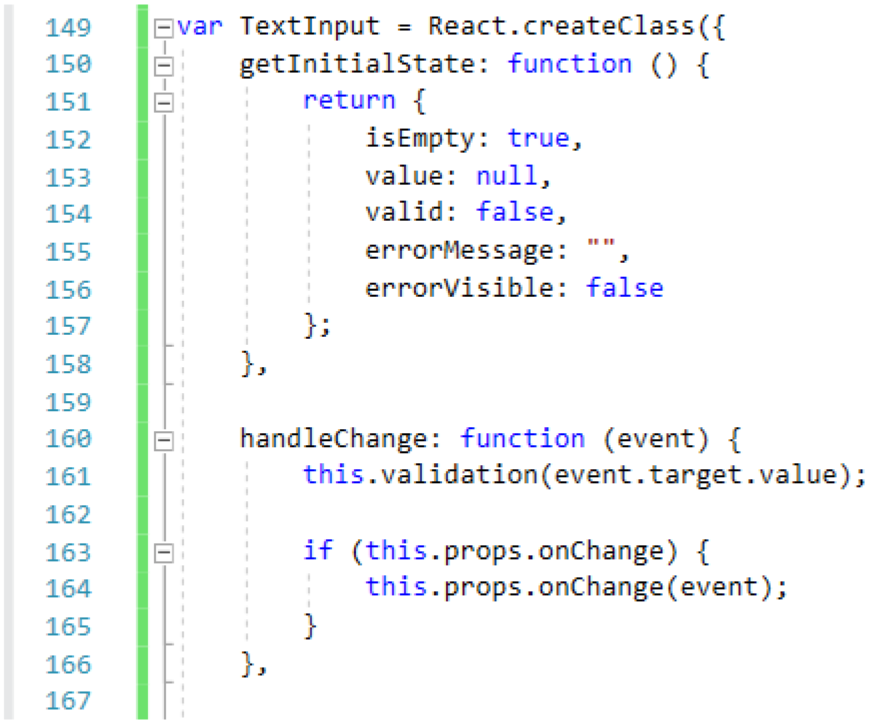
```
124
125             </form>
126         );
127     }
128 });
129
```

Now that the form class is created, we will start working on our additional helper classes. One class is to assist with input errors. This class is below, line 130 names and creates the class. Line 131 gets the initial start for the referring objects and will return the message on line 133 if that input is not valid. Line 136 creates the render function, which will send this information back to the referring call to InputError, which will happen later on in the code. Line 137 creates the variable to hold the errors and sets the class names for whether the error container exists, and whether or not it is visible or invisible. Line 143 then returns the value on line 144, which creates a starting and ending td tag so the error message will fit into our table. The command this.props.errorMessage starts at this, which is the referring object, then gets the property named errorMessage to be output.



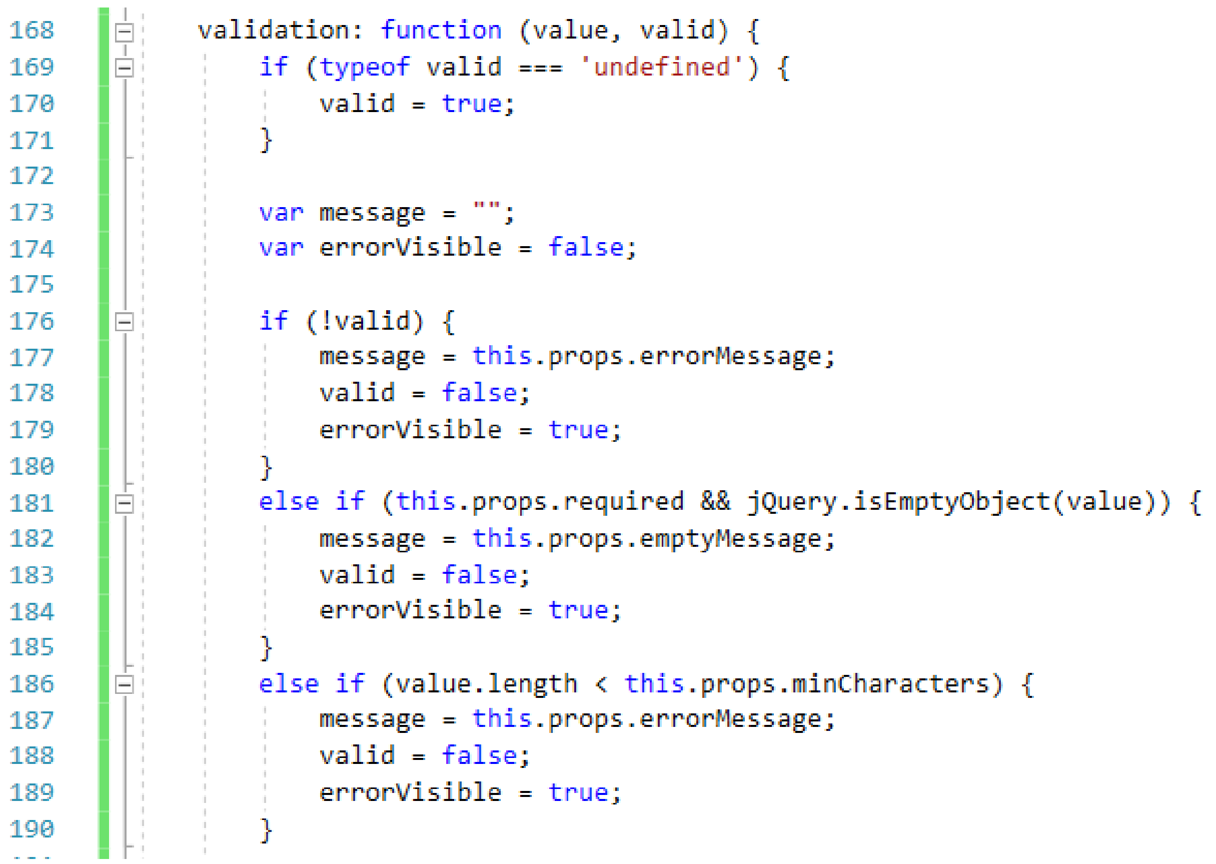
```
130  var InputError = React.createClass({
131      getInitialState: function () {
132          return {
133              message: 'Input is invalid'
134          };
135      },
136      render: function () {
137          var errorClass = classNames(this.props.className, {
138              'error_container': true,
139              'visible': this.props.visible,
140              'invisible': !this.props.visible
141          });
142
143          return (
144                  <td> {this.props.errorMessage} </td>
145          )
146      }
147  });
148
```

We will now create the TextInput class that we used to create objects earlier. We name the class on line 149, while 150 creates a method to set the initial state of several of the objects for this class. These objects are listed and returned on lines 152 to 156. Line 160 creates a method to handle when a change occurs, it will run the function, tagging the event that caused the change (Such as an onclick or tab). Line 161 runs validation on the referring object (What causes the event), and from that event looks at the value of the target of that event. The method that will provide this validation will be further along in this lab. Lines 162 and 164 just check to see if there has been a change to the property, and if so grabs the event that causes that change.



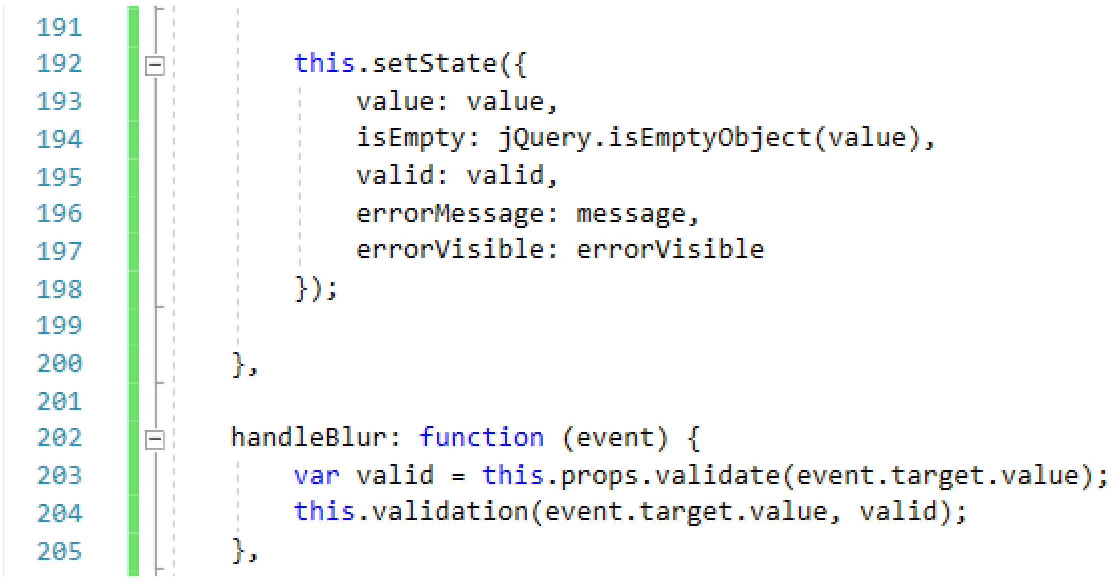
```
149  var TextInput = React.createClass({
150      getInitialState: function () {
151          return {
152              isEmpty: true,
153              value: null,
154              valid: false,
155              errorMessage: "",
156              errorVisible: false
157          };
158      },
159
160      handleChange: function (event) {
161          this.validation(event.target.value);
162
163          if (this.props.onChange) {
164              this.props.onChange(event);
165          }
166      },
167
```

The validation method discussed in the previous segment of code is now going to be created. This has 2 parameters, the value and whether or not that value is valid. Line 169 checks to see if the valid variable is not defined, and if so sets the value to true. Line 173 instantiates the message variable to blank to any errors can be added to it, while line 174 sets the visibility of the message to false at the start. We do not want an error message to show until we are sure there is an error. Line 176 start and if statement to trigger if the item is not valid. If so, the message is set to the property of the errorMessage option which was declared in the initial state area above. It also sets valid to false and makes the errorMessage visible. The next else if statement on line 181 checks to see if the property of the TextField object of required is there and also if the form object is empty. If so, it sets the message to the emptyMessage option on line 182, sets valid to false and the error message to visible. The last else if on line 186 checks to see if the length of the value is less than the minimum characters required and if so sets the correct error message, valid option and visibility of the error message.



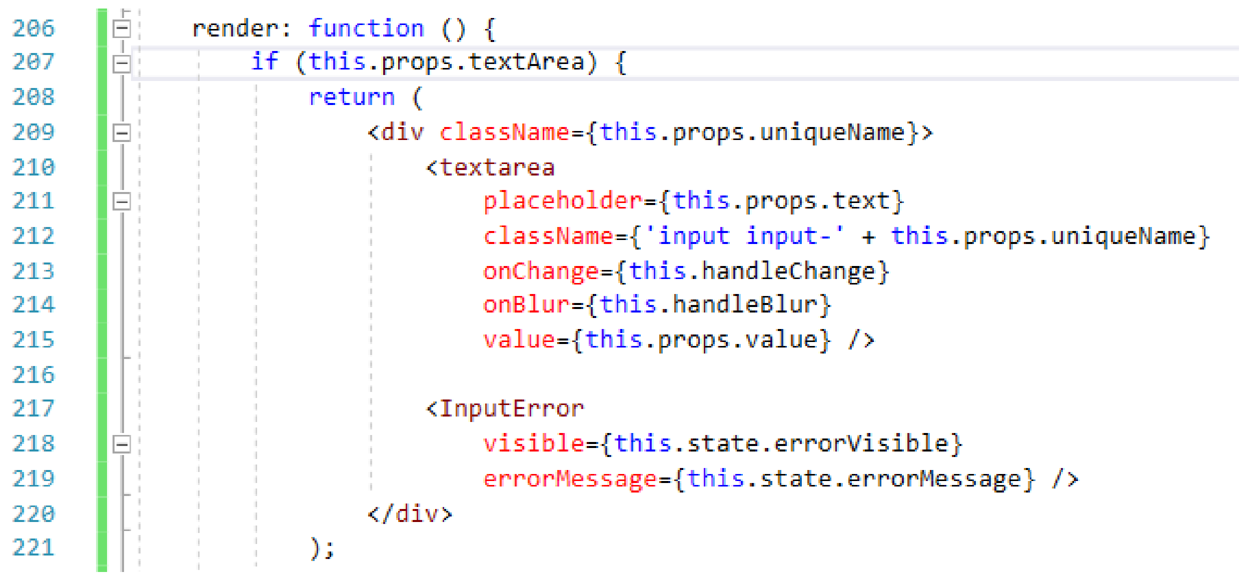
```
168     validation: function (value, valid) {
169         if (typeof valid === 'undefined') {
170             valid = true;
171         }
172 
173         var message = "";
174         var errorVisible = false;
175 
176         if (!valid) {
177             message = this.props.errorMessage;
178             valid = false;
179             errorVisible = true;
180         }
181         else if (this.props.required && jQuery.isEmptyObject(value)) {
182             message = this.props.emptyMessage;
183             valid = false;
184             errorVisible = true;
185         }
186         else if (value.length < this.props.minCharacters) {
187             message = this.props.errorMessage;
188             valid = false;
189             errorVisible = true;
190         }
```

The next function is one to set the states of all the options for the TextField objects. It sets the value to the value given on line 193. Line 194 sets the isEmpty variable to check to see if whatever value is given is an empty object. Lines 195 to 197 set each of the variables to their respective values. Down on line 202, a method called handleBlur is created, which is triggered when the event blur happens. A blur is when the focus on a field is moved to another location. Whatever object is the target of the blur is validated on line 203 and that validation value of true or false is placed into a variable called valid. Line 204 sends the value of the target as well as the true or false value to the validation function to be fully processed.



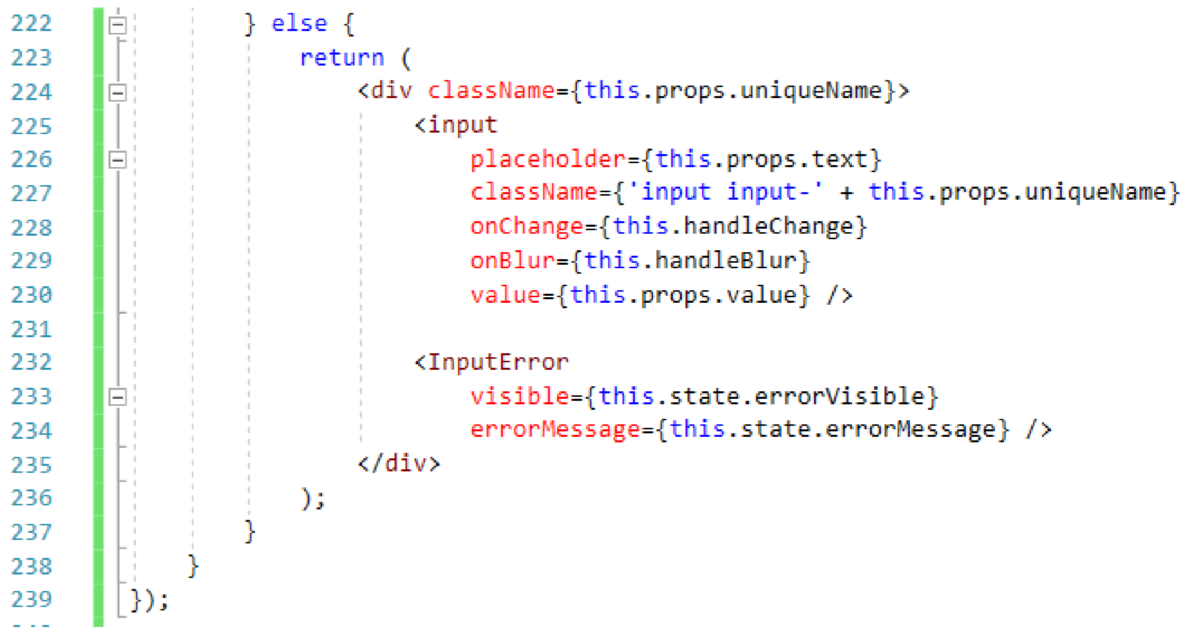
```
191
192         this.setState({
193             value: value,
194             isEmpty: jQuery.isEmptyObject(value),
195             valid: valid,
196             errorMessage: message,
197             errorVisible: errorVisible
198         });
199
200     },
201
202     handleBlur: function (event) {
203         var valid = this.props.validate(event.target.value);
204         this.validation(event.target.value, valid);
205     },
```

We will now render the items to be sent back to the instantiating object. Line 207 checks to see if the tag for textarea is set to true. If so a div will be created on line 209, with a textarea being created on line 210. This textarea received all the base options for the textarea on the next five lines. Line 211 puts placeholder text if needed, line 212 sets the className option. Line 213 lets the system know what to do if an onChange event occurs. In this case, the program will run the handleChange method from above. If a blue event occurs, line 214 tells the program to run the handleBlue event from above. Finally, line 215 will set the value of the object to whatever property value is given. Line 217 instantiates and object that will run the InputError class. These items will be set to visible and have the errorMessage variable set on lines 218 and 219. Whenever the TextField class is instantiated into an object that is a textarea, this is the code that will be created and shown on the form.



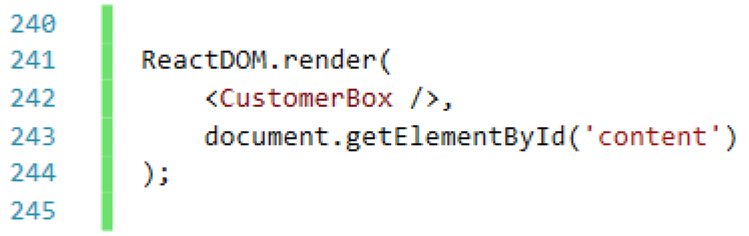
```
206    render: function () {
207        if (this.props.textArea) {
208            return (
209                <div className={this.props.uniqueName}>
210                    <textarea
211                        placeholder={this.props.text}
212                        className={'input input-' + this.props.uniqueName}
213                        onChange={this.handleChange}
214                        onBlur={this.handleBlur}
215                        value={this.props.value} />
216
217                    <InputError
218                        visible={this.state.errorVisible}
219                        errorMessage={this.state.errorMessage} />
220                </div>
221            );
```

The code below will perform if textarea is not selected, it will just create a standard input field with options. Line 224 creates the div, while line 225 creates the input tag. The values are set on lines 226 to 230 the same as were with the textarea option up above. The InputError object is also create on line 232 in the same manner as the textarea as well. Lines 236 to 239 close the method for the return statement, the else statement, the render function and finally the TextField class is closed on line 239.



```
222             } else {
223                 return (
224                     <div className={this.props.uniqueName}>
225                         <input
226                             placeholder={this.props.text}
227                             className={'input input-' + this.props.uniqueName}
228                             onChange={this.handleChange}
229                             onBlur={this.handleBlur}
230                             value={this.props.value} />
231 
232                         <InputError
233                             visible={this.state.errorVisible}
234                             errorMessage={this.state.errorMessage} />
235                     </div>
236                 );
237             }
238         }
239 });
```

The final code to be added is the code to tell the program to render on the ReactDOM, which is the document object model. We create the code to access the react dom on line 241. Line 242 instantiated the CustomerBox object, while line 243 tells the CustomerBox object where it will instantiate on the html page. This completes one of the 2 pages for the lab.



```
240 
241 ReactDOM.render(
242     <CustomerBox />,
243     document.getElementById('content')
244 );
245 
```

STEP 3

For this lab, perform a similar process for the insetemployees page. This will be created in a similar manner. The main difference is that the employees use the following fields in this order:

employeeid

employeename

employeeemail

employeephone

employeesalary

The name customer should not show anywhere on any of the pages for the employees.

Once the customer and employee pages are complete, zip the entire lastname_lab3 project and upload into the dropbox in D2L.