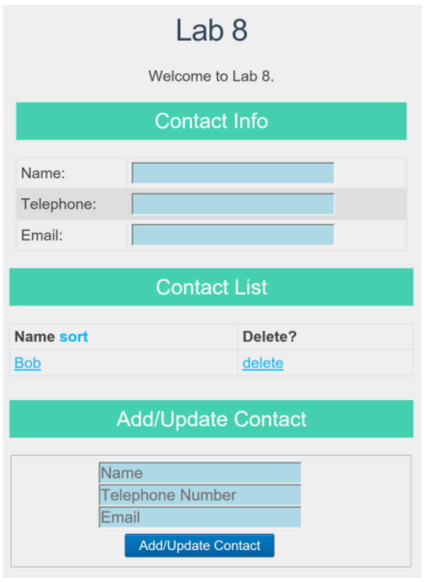
COMP3322A Modern Technologies on World Wide Web

Lab 8: React

Introduction

In this lab exercise, we will use React to re-implement the web page that we have worked on in the previous two labs, as shown below. The web page allows retrieving, displaying, adding, updating, sorting and deleting contacts from/to a MongoDB database through the web service that we built using the node.js/express.js environment in lab 6 (and used in both lab 6 and lab 7). When using React to implement the Web front-end and Node.js/Express.js/MongoDB to implement the back-end Web service, we are implementing the Web application using the MERN stack.



Lab 8

Welcome to Lab 8.

Contact Info

Name:	
Telephone:	
Email:	

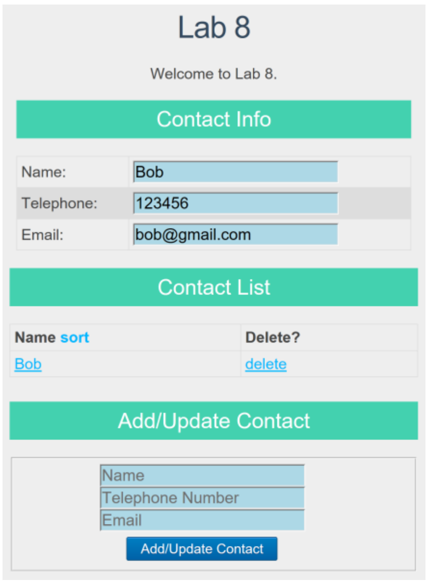
Contact List

Name sort	Delete?
Bob	delete

Add/Update Contact

Name
Telephone Number
Email
Add/Update Contact

Fig. 1 Upon initial page load (suppose Bob is already in the database)



Lab 8

Welcome to Lab 8.

Contact Info

Name:	Bob
Telephone:	123456
Email:	bob@gmail.com

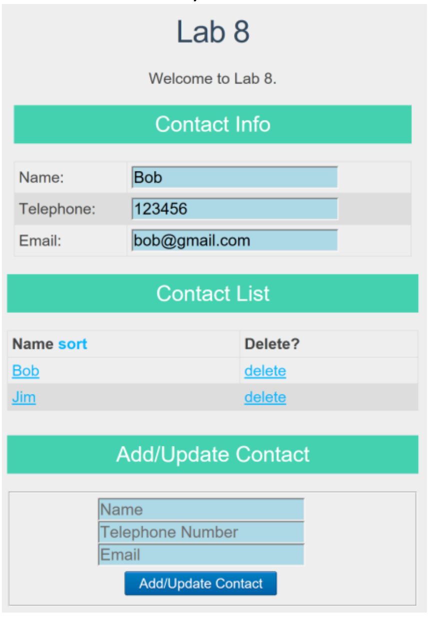
Contact List

Name sort	Delete?
Bob	delete

Add/Update Contact

Name
Telephone Number
Email
Add/Update Contact

Fig. 2 After clicking "Bob" under Contact List



Lab 8

Welcome to Lab 8.

Contact Info

Name:	Bob
Telephone:	123456
Email:	bob@gmail.com

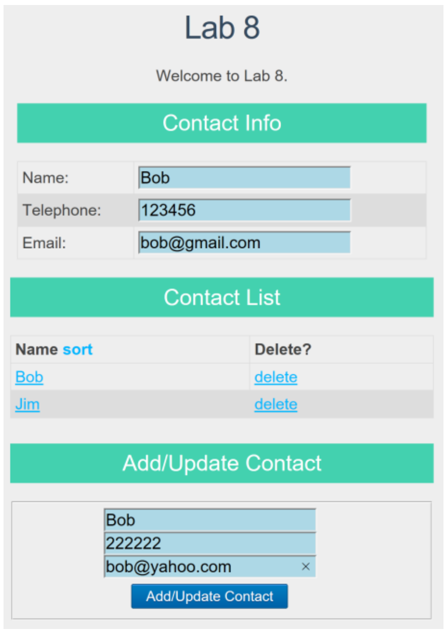
Contact List

Name sort	Delete?
Bob	delete
Jim	delete

Add/Update Contact

Name
Telephone Number
Email
Add/Update Contact

Fig. 3 After adding a new contact "Jim"


Lab 8

Welcome to Lab 8.

Contact Info

Name:	Bob
Telephone:	123456
Email:	bob@gmail.com

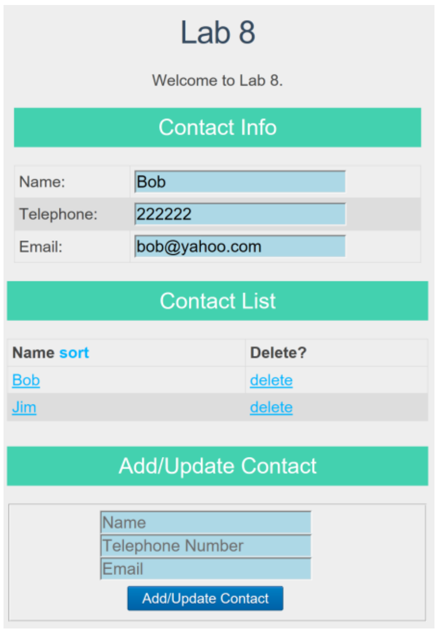
Contact List

Name sort	Delete?
Bob	delete
Jim	delete

Add/Update Contact

Bob
222222
bob@yahoo.com

Fig. 4 Enter Bob's new information


Lab 8

Welcome to Lab 8.

Contact Info

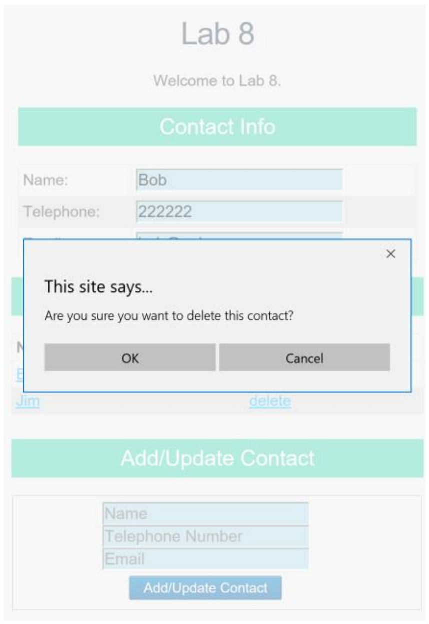
Name:	Bob
Telephone:	222222
Email:	bob@yahoo.com

Contact List

Name sort	Delete?
Bob	delete
Jim	delete

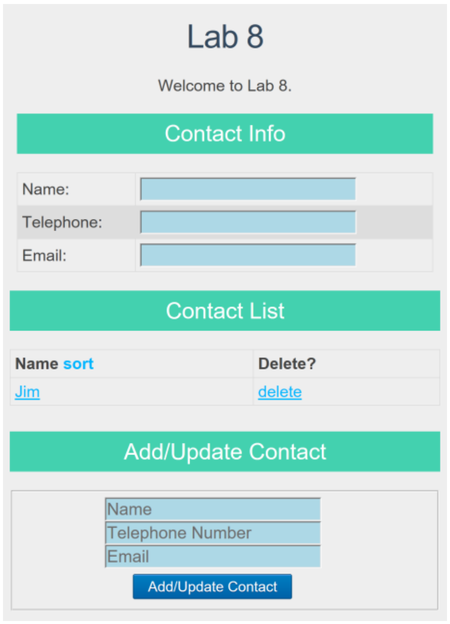
Add/Update Contact

Fig. 5 After clicking "Add/Update Contact" button


This site says...

Are you sure you want to delete this contact?

Fig. 6 After clicking "delete" of "Bob"


Lab 8

Welcome to Lab 8.

Contact Info

Name:	
Telephone:	
Email:	

Contact List

Name sort	Delete?
Jim	delete

Add/Update Contact

Fig. 7 After clicking "OK" to confirm deletion

Set up the Back-end Web Service

Create a folder "**lab8**". Inside the "**lab8**" folder, make a copy of your **lab6** project folder, and rename the folder to "**webservice**". In this lab, we are going to run the web service you built in lab 6 as the back-end web service side, and allow our React app (front-end) to access the web service it provides. We are going to run this web service on your localhost on the port of 3001 (instead of 3000), since we are going to run our React app on the port of 3000.

Launch a terminal and switch to the "**webservice**" directory, and run the following command to install CORS package:

```
npm install cors
```

We will need this CORS package for providing a middleware used to enable CORS (Cross-Origin Resource Sharing) with various options (see https://www.npmjs.com/package/cors).

Open **app.js** in the "**webservice**" folder. At the end of app.js, replace "module.exports = app;" by the following code:

```
//module.exports = app;
var server = app.listen(3001, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log("Example app listening at http://%s:%s", host, port);
})
```

Open **users.js** in "**webservice/routes/**" directory, add the following code at the beginning:

```
var cors = require('cors');
```

Change the middleware handling HTTP GET requests for "/contactList" as follows:

```
/*
* GET contactList.
*/
router.get('/contactList', cors(), function(req, res) {
    var db = req.db;
    var collection = db.get('contactList');
    collection.find({},{},function(err,docs){
            if (err === null)
                    res.json(docs);
            else res.send({msg: err});
    });
});
```

Change the middleware handling HTTP POST requests for "/addContact" as follows:

```
/*
 * POST to addContact.
 */
router.post('/addContact', cors(), function(req, res) {
    var db = req.db;
    var collection = db.get('contactList');
    collection.insert(req.body, function(err, result){
        res.send(
            (err === null) ? { msg: '' } : { msg: err }
        );
    });
});
```

Change the middleware handling HTTP PUT requests for "/updateContact" as follows:

```
/*
* PUT to updateContact
*/
router.put('/updateContact/:id', cors(), function (req, res) {
    var db = req.db;
    var collection = db.get('contactList');
    var contactToUpdate = req.params.id;
    var filter = { "_id": contactToUpdate};
```

```
   collection.update(filter,  {  $set:  {"name":  req.body.name,  "tel":  req.body.tel,  "email":
req.body.email}}, function (err, result) {
       res.send(
          (err === null) ? { msg: '' } : { msg: err }
       );
   })
});
```

And change the middleware handling HTTP DELETE requests for “/deleteContact” as follows:

```
/*
 * DELETE to deleteContact.
 */
router.delete('/deleteContact/:id', cors(), function(req, res) {
   var db = req.db;
   var contactID = req.params.id;
   var collection = db.get('contactList');
   collection.remove({'_id':contactID}, function(err, result){
            res.send((err === null)?{msg:''}:{msg:err});
   });
});
```

In the above middlewares, we use the cors middleware to allow client-side code received from our React app (that will be running at http://localhost:3000/) to access this Web service running at http://localhost:3001/ (i.e., resolve the cross-domain reference issue; learn more of CORS at https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS).

Further, we add the following code for handling preflight requests:

```
/*
 * Handle preflighted request
 */
router.options("/*", cors());
```

A CORS preflight request is an HTTP OPTIONS request, which is sent to check if the CORS protocol is understood. In this lab exercise, when your browser is about to send an HTTP PUT or DELETE request to a server running in another domain, it first automatically sends an OPTIONS request to check whether the actual request is safe to send; if so, the browser will follow up sending the actual PUT or DELETE request. (For “simple requests” such as GET and POST, such a preflight request will not be sent by your browser.) The front-end code does not need to deal with sending a preflight request (as browser automatically sends it). But at the back-end, we need to handle/respond to such OPTIONS requests; that’s why we add the above middleware into **users.js**. See more at https://developer.mozilla.org/en-US/docs/Glossary/Preflight_request.

You can remove the pug template engine-related code and module, indexRouter and index.js from the project as what we did in Lab 7. Our React app will only make use of the Web service implemented by app.js and user.js, but not any other modules in the app you built in Lab 6.

Launch the web service app as follows (as what you did in Lab 6):

Step 1: Launch a terminal and switch to the directory where MongoDB is installed. Start MongoDB server using the “**data**” directory in the “**webservice**” folder as the database location, as follows: (replace “**YourPath**” by the actual path on your computer that leads to “lab8” directory)

```
./bin/mongod  --dbpath YourPath/lab8/webservice/data
```

In this way, you can reuse the “lab6” database you used in Lab 6.

Leave this terminal open and do not close it during your entire lab practice session, in order to allow connections to the database from your server app.

Launch another terminal and switch to the directory where MongoDB is installed. Then you can use the following commands to add contacts into the database, if you need them for testing purpose. (If there are still contacts in your “lab6” database as you inserted before, you do not have to do this.)

```
./bin/mongo
use lab6
db.contactList.insert({'name':'Bob', 'tel':'1234567', 'email':'bob@gmail.com'})
```

Step 2: Launch another terminal and switch to the “**webservice**” directory, and run the following command to launch your server app (**NOT** “npm start”!):

```
node app.js
```

In this way, your web service will be running on the port of 3001, as specified by code at the end of **app.js**. **Leave this terminal open and do not close it during your entire lab practice session,** in order to allow connections to the web service from your React app.

Create a New React App

Launch a terminal. Go to your “**lab8**” directory and create a React app named “myreactapp” using the following commands:

```
cd YourPath/lab8
npx create-react-app myreactapp
```

Go inside the “**myreactapp**” folder just created. Since we are going to use the jQuery library for the React app to communicate with the webservice app, install the jQuery module in the React app as follows:

```
cd myreactapp
npm install jquery
```

Then launch the React App as follows:

```
npm start
```

After successfully launching the app, you should see prompts like the following in your terminal:

```
Compiled successfully!

You can now view myreactapp in the browser.

  http://localhost:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

And a web page should be loaded automatically in your browser, as follows:

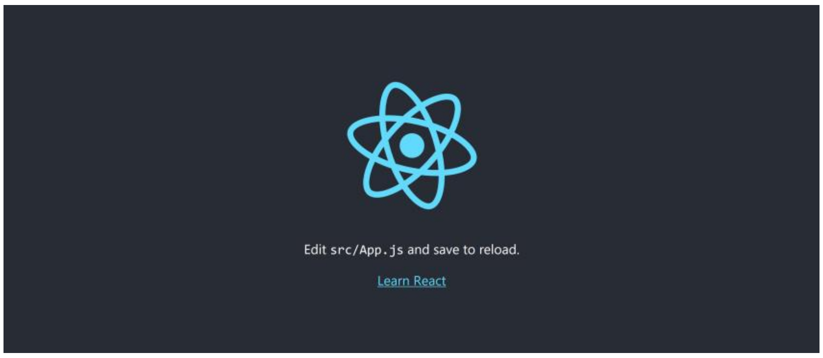


Fig. 8

Using the command "npm start", we are running a development build (not optimized), rather than a production build (optimized build which can be created using "npm run build" instead).

Lab Exercise 1: Understand the Project File Structure

Step 1: The HTML file loaded after you launched the React app using "npm start" is **index.html** under myreactapp/public/, together with an image file and a configuration file. In **index.html**, a <div> element with id "root" is created as follows, in which React elements will be rendered:

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

Step 2: The JavaScript files to render React elements are located under myreactapp/src/. Find **index.js** and **App.js** in this directory and open them in a text editor.

Step 3: In **index.js**, it first loads **React** and **ReactDOM** modules:

```
import React from 'react';
import ReactDOM from 'react-dom';
```

and css file:

```
import './index.css';
```

and exported components from other JavaScript files (**App.js** and **serviceWorker.js**):

```
import App from './App';
import * as serviceWorker from './serviceWorker';
```

index.js mainly renders the **App** component in the 'root' <div> element (in **index.html**). serviceWorker is used in production environment to register a service worker to serve assets from local cache (i.e., to allow the app to load faster on subsequent visits in production environment); serviceWorker.unregister(); is used to disable this functionality.

```
ReactDOM.render(<App />, document.getElementById('root'));
serviceWorker.unregister();
```

Step 4: In **App.js**, it creates a class component **App**:

```
class App extends Component {
 render() {
   return (
    <div className="App">
      <header className="App-header">
       <img src={logo} className="App-logo" alt="logo" />
       <p>
         Edit <code>src/App.js</code> and save to reload.
       </p>
       <a
         className="App-link"
         href="https://reactjs.org"
         target="_blank"
         rel="noopener noreferrer"
       >
         Learn React
       </a>
      </header>
    </div>
   );
 }
```

The component returns a <div> element of class “App”, and the styling rules on this class (given in **App.css**) are applied to this <div> element. Note that the class attribute becomes className in React. The <div> element contains a <header> element of class "App-header". Within the header, there is an <img> element, a <p> element and a <a> element. All these elements render the page view in Fig. 8.

At last, **App.js** exposes the App component to other modules using the following statement:

```
export default App;
```

Lab Exercise 2: Create Front-end Web Page Using React

We are going to modify **index.js** and **App.js** to create the page as shown in Figures 1-7.

Step 1: In **index.js**, replace the content by the following code:

```
import React from 'react';
import ReactDOM from 'react-dom';
import ContactPage from './App';

ReactDOM.render(
  <ContactPage/>,
  document.getElementById('root')
);
```

With the above code, we render the element returned by the ContactPage component (to be implemented in **App.js**) in the “root” <div> (in myreactapp/public/index.html).

ContactPage is the component to render the entire view in Fig. 1, enclosing other components to implement different parts in the view. Note that the component exported from **App.js** will be ContactPage (instead of App as in the default React app you studied in Lab Exercise 1); hence we use import ContactPage from './App'; at the beginning of **index.js.**

Step 2: In **App.js** , replace the content by the following code, which creates the ContactPage component. Copy **App.css** that we have provided to replace the default **App.css** in myreactapp/src/, in order to use the provided styling rules to style the page.

```
import React from 'react';
import ReactDOM from 'react-dom';
import './App.css';
import $ from 'jquery';

class ContactPage extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      contacts: [],
      displayContact:{'name':'','tel':'','email':''},
    };
   this.handleDisplayInfo = this.handleDisplayInfo.bind(this);
  }

  handleDisplayInfo(contact){
    this.setState({
      displayContact:{'name':contact.name,'tel':contact.tel,'email':contact.email}
    });
  }

  componentDidMount() {
    this.loadContacts();
  }

  loadContacts() {
    $.ajax({
        url: "http://localhost:3001/users/contactList",
        dataType: 'json',
        success: function(data) {
          this.setState({
```

```
      contacts: data
     });
    }.bind(this),
    error: function (xhr, ajaxOptions, thrownError) {
      alert(xhr.status);
      alert(thrownError);
    }.bind(this)
  });
 }

 render() {
  return (
   <div id="wrapper">
    <h1> Lab 8</h1>
    <p>Welcome to Lab 8.</p>
    <ContactInfo
     displayContact={this.state.displayContact}
    />
    <ContactList
     contacts={this.state.contacts}
     handleDisplayInfo={this.handleDisplayInfo}
    />
   </div>
  );
 }
}

export default ContactPage;
```

The component returns a <div> element as the container, containing a few HTML elements and a ContactInfo component and a ContactList component, corresponding to the displayed contact information and the contact list in the page view, respectively. We will implement another component AddOrUpdateContactForm in the <div> in Lab Exercise 3. The component structure in this React app is illustrated in the following figure.

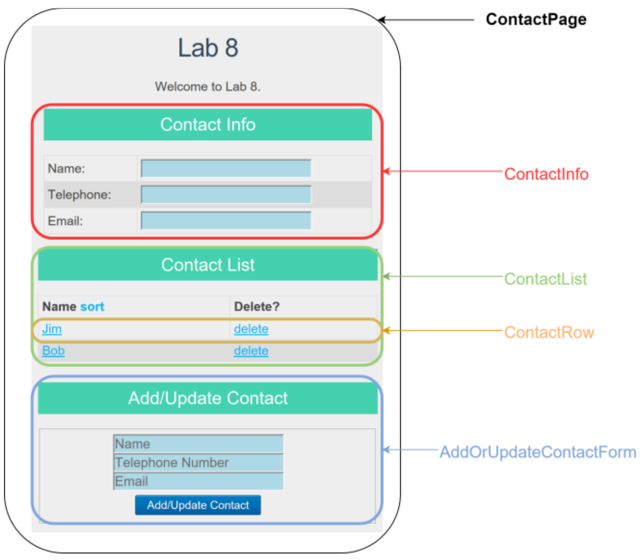


Fig. 8 An illustration of component structure

There are currently two states contacts and displayContact in the ContactPage component, maintaining the contacts to be displayed under “Contact List”, and contact information that is displayed under “Contact Info”, respectively. Value of the contacts state is used in the ContactList component (and the AddOrUpdateContactForm component to be implemented later); value of the displayContact state is used in the ContactInfo component and changed by events captured in the ContactList and AddOrUpdateContactForm components. We therefore maintain them in the parent component ContactPage (as their values are used/set by different child components of ContactPage). The event handler function handleDisplayInfo sets the values of displayContact based on values of the input contact.

The componentDidMount() function is a function defined in the React.Component abstract class, and is invoked immediately after the component is mounted (refer to https://reactjs.org/docs/react-component.html#componentdidmount). Inside this function, we call a jQuery AJAX API to create an HTTP GET AJAX request, for retrieving the contacts from the Web service, which we have launched in “Set up the Back-end Web Service”. Especially, in order to use jQuery APIs in React, we have imported the jQuery module as “import $ from 'jquery';" at the beginning of **App.js**. Refer to http://api.jquery.com/jquery.ajax/ to learn about our settings in the $.ajax function call. To make “this” accessible inside the success and error callback functions, we bind “this” to both functions using .bind(this) in the code.

Step 3: In **App.js** , add the following code to create the ContactInfo component:

```
class ContactInfo extends React.Component{
 render() {
   const contact = this.props.displayContact;
   return (
    <div id="wrapper">
    <div id="contactInfo">
      <h2>Contact Info</h2>
      <table>
       <thead>
         <tr>
          <td>Name:</td>
          <td><input id='contactInfoName'  readonly value={contact.name}></input></td>
         </tr>
         <tr>
          <td>Telephone:</td>
          <td><input id='contactInfoTel'  readonly value={contact.tel}></input></td>
         </tr>
         <tr>
          <td>Email:</td>
          <td><input id='contactInfoEmail'  readonly value={contact.email}></input></td>
         </tr>
       </thead>
      </table>
    </div>
    </div>
   );
 }
}
```

The component returns a <div> element. The value displayed under the “Contact Info” is decided by displayContact contained in the **props** passed into the component, which is value of the state displayContact in the ContactPage component.

Step 4: In **App.js**, add the following code to create the ContactList component:

```
class ContactList extends React.Component {
 constructor(props) {
  super(props);
  this.handleDisplayInfo = this.handleDisplayInfo.bind(this);
 }

 handleDisplayInfo(contact){
  this.props.handleDisplayInfo(contact);
 }

 render() {
  let rows = [];
  this.props.contacts.map((contact) => {
    rows.push(
     <ContactRow
      contact={contact}
      handleDisplayInfo={this.handleDisplayInfo}
     />
    );
  });

  return (
   <div id="contactList">
   <h2> Contact List </h2>
   <table>
    <thead>
     <tr>
      <th>Name <span > <a id='sort' onClick="">sort</a> </span></th>
      <th>Delete?</th>
     </tr>
    </thead>
    <tbody>{rows}</tbody>
   </table>
   </div>
  );
 }
}
```

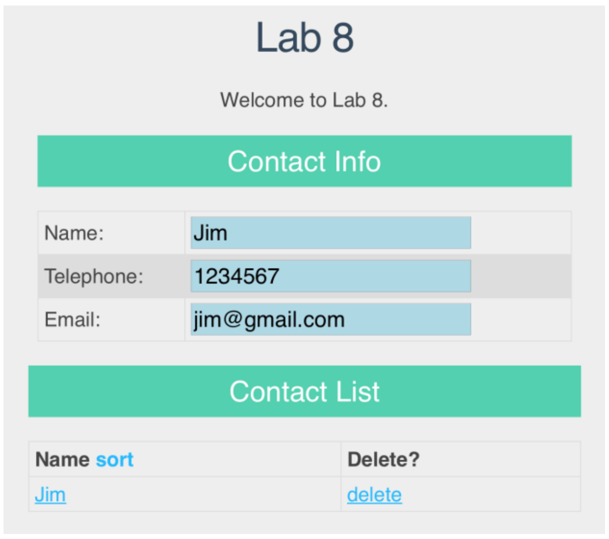
The component displays contacts in a table by using a ContactRow component to return the table row showing each contact.

Step 5: In **App.js**, add the following code to create the ContactRow component:

```
class ContactRow extends React.Component {
  constructor(props) {
    super(props);
    this.handleDisplayInfo = this.handleDisplayInfo.bind(this);

  }

  handleDisplayInfo(e) {
    e.preventDefault(e);
    this.props.handleDisplayInfo(this.props.contact);
  }

 render() {
    const contact = this.props.contact;

    return (
      <tr>
        <td><a href="" onClick={this.handleDisplayInfo}
rel={contact.Name}>{contact.name}</a></td>
        <td><a href="" onClick="" rel={contact._id}>delete</a></td>
      </tr>
    );
  }
}
```

We capture onClick events on the link showing a contact's name and add event handler to handle them.

Step 6: Now launch the app using "npm start" and browse the web page at http://localhost:3000/. You should see a page like the following (except that the information under "Contact Info" is still empty). You can test clicking the contact name for display to see the effectiveness.



Lab 8

Welcome to Lab 8.

Contact Info

Name:	Jim
Telephone:	1234567
Email:	jim@gmail.com

Contact List

Name sort	Delete?
Jim	delete

Fig. 9 After clicking "Jim"

Lab Exercise 3: Add the Component for Adding or Updating Contact

Next, we will add an AddOrUpdateContactForm component in **App.js** to implement the Add/Update Contact form on the web page, which is also rendered in the ContactPage component.

Step 1: In the ContactPage component, add the following in the <div> that it returns:

```
  <AddOrUpdateContactForm
      newContactName={this.state.newContactName}
      newContactTel={this.state.newContactTel}
      newContactEmail={this.state.newContactEmail}
      onNameChange={this.handleNameChange}
      onTelChange={this.handleTelChange}
      onEmailChange={this.handleEmailChange}
      handleAddOrUpdateSubmit={this.handleAddOrUpdateSubmit}
      displayContact={this.state.displayContact}
   />
```

Then add three additional states in the ContactPage component as follows:

```
    newContactName: '',
    newContactTel: '',
    newContactEmail: '',
```

And add three event handlers in the ContactPage component as follows to handle the change events on the name input textbox, telephone number input textbox and email input textbox (in the AddOrUpdateContactForm component which we are going to implement in **Step 2**), respectively. These event handlers change the respective state values in the ContactPage component according to user input values in AddOrUpdateContactForm. In addition, you should add code to bind “this” to the three functions.

```
handleNameChange(name) {
  this.setState({
    newContactName: name
  })
}

handleTelChange(tel) {
  this.setState({
    newContactTel: tel
  })
}

handleEmailChange(email) {
  this.setState({
    newContactEmail: email
  })
}
```

Add another event handler as follows to handle the click event on the button in the AddOrUpdateContactForm component. Again, you should add code to bind “this” to this function in the ContactPage component.

```
handleAddOrUpdateSubmit(e) {
    e.preventDefault();

   if (this.state.newContactName === '' || this.state.newContactTel === ''||
this.state.newContactEmail === '') {
       alert('Please fill in all fields');
   }
  else{
    var existingIndex = -1;

    for(var i=0; i < this.state.contacts.length; i++){
      if(this.state.newContactName === this.state.contacts[i].name){
        existingIndex = i;
        break;
      }
    }

    if(existingIndex >= 0){
      var existingContact = {
        "_id" : this.state.contacts[existingIndex]._id,
        "name" : this.state.newContactName,
        "tel" : this.state.newContactTel,
        "email" : this.state.newContactEmail
      }
      this.handleUpdate(existingIndex, existingContact);
    }else{
      $.post("http://localhost:3001/users/addContact",
      {
        "name" : this.state.newContactName,
        "tel" : this.state.newContactTel,
        "email" : this.state.newContactEmail
      },
      function(data, status){
        if (data.msg ===''){
          this.loadContacts();
          this.setState({
            newContactName: '',
            newContactTel: '',
            newContactEmail: ''
          });
        } else
          alert(data.msg);
      }.bind(this)
      );
    }
  }
 }
```

In this event handler, we check if the added contact exists in contact list. If this is a new contact, we produce an AJAX POST request using jQuery's $.post API. When a success response is received from the server side, the new contact is added into the state contacts array. The value of contacts is used by the ContactList component to decide the table rows to display, and hence the new contact is to be displayed in the ContactList. If this added contact exists in the contact list, the event handler will call another function handleUpdate to handle updating the existing contact as follows:

```
handleUpdate(existingIndex, existingContact){
  $.ajax({
    url: ?,
    type: ?,
    data: ?,
    dataType: ?,
    success: function(data) {
          ?
    }.bind(this),
      error: function (xhr, ajaxOptions, thrownError) {
          alert(xhr.status);
          alert(thrownError);
      }.bind(this)
  });
}
```

You should replace "?" in the above function to implement the update function. Especially, when a success response is received, you should update the respective contact's information in the contacts state, and clear all the input textboxes; if the contact updated is displayed under "Contact Info", you should update the displayed information as well.
 (You may consider implementing this handleUpdate function after you have completed **Step 2** below, when you have a better idea of the workflow of AddOrUpdateContactForm component).

Step 2: Implement the AddOrUpdateContactForm component following the sketch below, which returns a <div> element that includes a name input textbox, a telephone number input textbox, an email input textbox, and a button. The view should be like that in Fig. 1.

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

     //bind this to functions
     ???
  }

  handleNameChange(e) {
    ?
  }

  handleTelChange(e) {
    ?
  }

  handleEmailChange(e) {
    ?
  }

  handleAddOrUpdateSubmit(e) {
    ?
  }

  render() {
    return (
      <div id="addContact">
        <h2> Add/Update Contact </h2>
        <fieldset>
          <input className="input_text"
            type="text"
            placeholder = "Name"
            value={this.props.newContactName}
            onChange={this.handleNameChange}
          />
          <br/>

          //Render the telephone number input textbox
          ??

          // Render the email input textbox
          ??

          <button className="myButton" onClick={this.handleAddOrUpdate}>Add/Update 
Contact</button>
        </fieldset>
      </div>
    );
  }
}
```

Especially, you should associate values of the name, telephone number and email input elements with respective states in the ContactPage component (i.e., newContactName, newContactTel, and newContactEmail) through the **props** that the AddOrUpdateContactForm component receives.

In addition, you should implement the code for passing the handling of the change events on the input textboxes and the click event on the button to event handlers handleNameChange, handleTelChange, handleEmailChange, and handleAddOrUpdateSubmit in the ContactPage component.

Associate the input textboxes with the class of "input_text", such that they can be styled using respective rules in App.css.

Launch the React app using “npm start” and browse the web page at http://localhost:3000/. You should see a page as that in Fig. 1. Try adding a new contact or updating an existing contact to test your code.

Lab Exercise 4: Deleting an Existing Contact

Next, we will add code in App.js to implement the deletion of an existing contact.

Step 1: In the ContactPage component, add the following attribute when returning the ContactList component:

```
handleDelete={this.handleDelete}
```

Implement the handleDelete function in the ContactPage component by following the code sketch below. Also, remember to bind "this" with the handleDelete function.

```
handleDelete(e){
   e.preventDefault(e);

   var confirmation = window.confirm('Are you sure you want to delete this contact?');
   if(confirmation === true){
     var id = e.target.rel;
     $.ajax({
       type: ?,
       url: ?,
       dataType: ?,
       success: function(data) {
              //implement the deletion of the deleted contact from the contacts state and the
display under "Contact Info"
              ??
       }.bind(this),
       error: function (xhr, ajaxOptions, thrownError) {
              alert(xhr.status);
              alert(thrownError);
       }.bind(this)
     });
   }
 }
```

You should replace "?" with correct code to implement delete function. Especially, you should send an HTTP DELETE request to the back-end Web service to delete the contact in database. If a success response is received, delete the contact from the contacts state; if the contact deleted is displayed under "Contact Info", you should clear the displayed information there as well. (You may check out **Step 2** and **Step 3** below, such that you can have a better idea of the entire workflow for deletion handling).

Step 2: In the ContactList component, add the following attribute when returning the ContactRow component (**Hint**: in rows.push()):

```
handleDelete={this.handleDelete}
```

Then add the event handler function handleDelete in the ContactList component, which passes the click event on the "delete" link of a ContactRow component further to event handler handleDelete in the ContactPage component.

Step 3: In the ContactRow component, add the event handler function handleDelete, which passes the click event on the "delete" link to event handler handleDelete in the ContactList component. Then use this handleDelete function as event handler to onClick event on the "delete" link (you can replace onClick="" by onClick={this.handleDelete}).

Launch the React app using "npm start" and browse the web page at http://localhost:3000/. You should see the complete page as shown in Fig. 1. Try deleting an existing contact to test your code.

Lab Exercise 5: Sorting the Contact List

Next, we will add code in App.js to implement sorting the contacts in the contact list, when "sort" in the Contact List is clicked. When the "sort" link is clicked for the first time, the contacts should be sorted according to alphabetic order of their names; when "sort" is clicked again, the contacts should be listed in reverse alphabetical order of their names; and the list shuffles between alphabetic order and reverse alphabetical order when "sort" is clicked again and again.

Step 1: In the ContactPage component, add the following attribute when returning the ContactList component:

```
handleSortList={this.handleSortList}
```

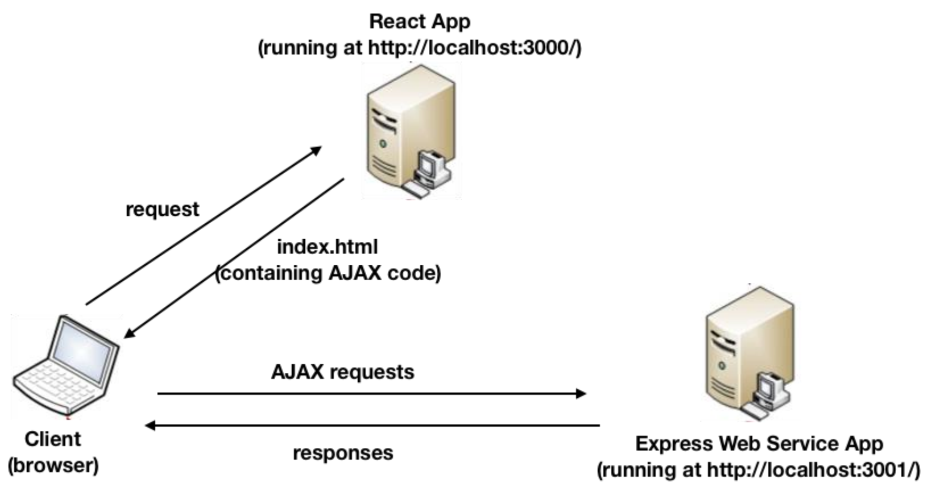
Implement the handleSortList function in the ContactPage component as follows. Also, remember to bind "this" with the handleSortList function. You can add a new order state into ContactPage component, and initialize its value to 1.

```
handleSortList() {
   // to complete
}
```

Step 2: In the ContactList component, add the event handler function handleSortList, which passes the click event on the "sort" link to event handler handleSortList in the ContactPage component. Then use this handleSortList function as event handler to onClick event on the "sort" link (you can replace onClick="" by onClick={this. handleSortList }).

Relaunch the React app using "npm start" and browse the web page at http://localhost:3000/. You should see the complete page as shown in Fig. 1. Try sorting the contact list to test your code.

Finally, for better understanding of interactions among client, React App and Express App in the Web system you have just build, please refer to the following figure:



Since the code running at the client was received from the React App instead of from the Express App, the AJAX HTTP requests the client sends to the Express App are considered to be from a different origin — hence the need for handling CORS in the Express App.

Submission:

You should submit the following files and folders only:
(1) ./webservice (app.js and user.js)
(2) ./myreactapp (index.js and App.js)

Please compress the above folder/files in a .zip file and submit it on Moodle before 23:59 Wednesday Dec. 4, 2019:

(1) Login Moodle.
(2) Find “Labs” in the left column and click “Lab 8”.
(3) Click “Add submission”, browse your .zip file and save it. Done.
(4) You will receive an automatic confirmation email, if the submission was successful.
(5) You can “Edit submission” to your already submitted file, but ONLY before the deadline.