

# COMP3322 Modern Technologies on World Wide Web

## Workshop 5: ReactJS

### Introduction

In this workshop, we will use **React** to implement a simple web page, as shown below. The web page allows retrieving, displaying and adding commodities from/to a MongoDB database through the web service that we built using the [node.js/express.js](https://nodejs.org/en/) environment in [workshop 4](#).

Upon initial load, you will see a page as shown in Fig. 1. A number of existing commodities from the MongoDB database will be loaded and displayed in a table (you can add more data through database operations directly).

Name	Category	Status
iPhone	Phone	in stock
iPad	Tablet	out of stock

Fig. 1

Name	Category	Status
iPhone	Phone	in stock

Fig. 2

If you enter some string in the “Search...” box, the displayed commodity table changes and only the commodities whose name starts with the entered string will be displayed (Fig. 2).

If you click the “Hide Out-of-Stock Commodity” button, the commodities whose status is “out of stock” will no longer be displayed in the table, and the text on the button changes to “Show Out-of-Stock Commodity” (Fig. 3).

Name	Category	Status
iPhone	Phone	in stock

Fig. 3

If you click the “Show Out-of-Stock Commodity” button, the page view goes back to Fig. 1. If you enter a new commodity’s name and category in the respective input text boxes (at

the lower part of the page), select the status in the select dropdown list (e.g., Fig. 4), and then click the “Submit” button, the new commodity’s information will be sent to the server side to store in the MongoDB database after issuing an alert, and the page view becomes Fig. 5, i.e., the newly added commodity is displayed in the commodity table.

Name	Category	Status
iPhone	Phone	in stock
iPad	Tablet	out of stock

Hide Out-of-Stock Commodity

Name:

Category:

Status:

Submit

Fig. 4

Name	Category	Status
iPhone	Phone	in stock
iPad	Tablet	out of stock
lenovo	Computer	in stock

Hide Out-of-Stock Commodity

Name:

Category:

Status:

Submit

Fig. 5

## Prepare the Web Service

Create a folder “workshop5”. Inside the “workshop5” folder, make a copy of your workshop4 project folder, and **rename the folder** name to “**serverapp**”. In this workshop, we are going to run the app you built in workshop 4 as the **server side** and allow our React app (**client side**) to make use of the web services it provides. We are going to run this **server app** on your localhost with the port **3001** (instead of 3000), since we are going to run our React app on the port of 3000.

To change the port number uses by the server app, open **serverapp/bin/www** and change this line

```
var port = normalizePort(process.env.PORT || '3000');
```

to

```
var port = normalizePort(process.env.PORT || '3001');
```

Next we have to explicitly set the serverapp.

Open **./serverapp/routes/user.js**, and change the middleware handling HTTP GET requests for “/commodities” by adding the “Access-Control-Allow-Origin” line as follows:

```
router.get('/commodities', function(req, res) {
  res.set({"Access-Control-Allow-Origin": "http://localhost:3000"});
  //Get the data
  req.commodity.find(function(err, docs) {
    if (err === null)
      res.json(docs);
    else
      res.send({msg: err });
  });
});
```

Do the same change to the middleware handling HTTP POST requests for “/addcommodity” as follows:

```
router.post('/addcommodity', function (req, res) {
  res.set({"Access-Control-Allow-Origin": "http://localhost:3000"});
  var addRecord = new req.commodity({
    category: req.body.category,
    name: req.body.name,
    status: req.body.status
  });

  //add new commodity document
  addRecord.save(function (err, result) {
    res.send((err === null) ? { msg: ' ' } : { msg: err });
  });
});
```

Especially, in the above two middlewares, we set [Access-Control-Allow-Origin](#) header into the response message, in order to allow our react app which 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; see more at <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> ).

Note that 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 [workshop4](#), and you can leave all other files as they are in the “**serverapp**” folder.

Launch the server app as follows (as what you did in workshop 4):

**Step 1:** Launch a terminal and start MongoDB server using the “**data**” directory in the “**serverapp**” folder as the database location, as follows: (replace “**YourPath**” by the actual path on your computer that leads to “workshop5” directory)

```
mongod --dbpath YourPath/workshop5/serverapp/data
```

In this way, you can reuse the “workshop4” database you used in workshop 4. [Leave this terminal open and do not close it during your entire workshop practice session](#), in order to allow connections to the database from your server app.

**Step 2:** Launch another terminal and switch to the “**serverapp**” directory, and run the “**npm start**” command to start the serverapp server. [Leave this terminal open and do not close it during your entire workshop practice session](#), in order to allow connections to the server app from your React app.

## Create a New React App

Launch a terminal. Go to your “**workshop5**” directory and create a React app named “**myreactapp**” using the following commands:

```
cd YourPath/workshop5
npm init react-app myreactapp
```

Go inside the “**myreactapp**” folder just created. Since we are going to use the jQuery library when implementing our own React 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.  
  
Local: http://localhost:3000/  
On Your Network: ***  
  
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. 6

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).

Note that with the above steps, we have created a new React app. If you wish to add React into an existing app, refer to steps here: <https://reactjs.org/docs/add-react-to-an-existing-app.html>.

## Exercise 1: Understand the Project File Structure

**Step 1:** The HTML file loaded after you launched the app using “**npm start**” is **index.html** under **myreactapp/public/**, together with image and configuration files. 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.register()` 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 — see <https://create-react-app.dev/docs/making-a-progressive-web-app/>).

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

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

```
function App() {
  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 of `class` "App-intro", and a `<a>` element. All these elements render the page view in Fig. 6.

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

```
export default App;
```

## Exercise 2: Create our Web Page Using React

We are going to modify **index.js** and **App.js** to create the page as shown in Figures 1-5.

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

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

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

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

**CommodityPage** 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 **CommodityPage** (instead of **App** as in the default React app you studied in Exercise 1); and hence we use **import CommodityPage from './App'**; at the beginning of **index.js**.

**Step 2:** In **App.js**, replace the content by the following code, which creates the **CommodityPage** component:

```
import React, { Component } from 'react';
import $ from 'jquery';

class CommodityPage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      commodities: [],
      filterText: '',
      showOutOfStockCommodity: true
    };

    this.handleFilterTextChange = this.handleFilterTextChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }

  handleFilterTextChange(filterText) {
    this.setState({
      filterText: filterText
    });
  }

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

```

    })
  }

  componentDidMount() {
    this.loadCommodities();
  }

  loadCommodities() {
    $.ajax({
      url: "http://localhost:3001/users/commodities",
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({ commodities: data });
      }.bind(this),
      error: function (xhr, ajaxOptions, thrownError) {
        alert(xhr.status);
        alert(thrownError);
      }
    });
  }

  render() {
    return (
      <div>
        <SearchBar
          filterText={this.state.filterText}
          onFilterTextChange={this.handleFilterTextChange}
        />
        <CommodityTable
          commodities={this.state.commodities}
          filterText={this.state.filterText}
          showOutOfStockCommodity={this.state.showOutOfStockCommodity}
        />
        <ShowHideButton
          showOutOfStockCommodity={this.state.showOutOfStockCommodity}
          onClick={this.handleClick}/>
      </div>
    );
  }
}

export default CommodityPage;

```

The component returns a `<div>` element as the container, containing a [SearchBar](#) component, a [CommodityTable](#) component, and a [ShowHideButton](#) component, corresponding to the search bar, the commodity table and the “Show/Hide Out-of-Stock Commodity” button in the view respectively (see Fig. 7). We will implement the component [AddCommodityForm](#) in Exercise 3.

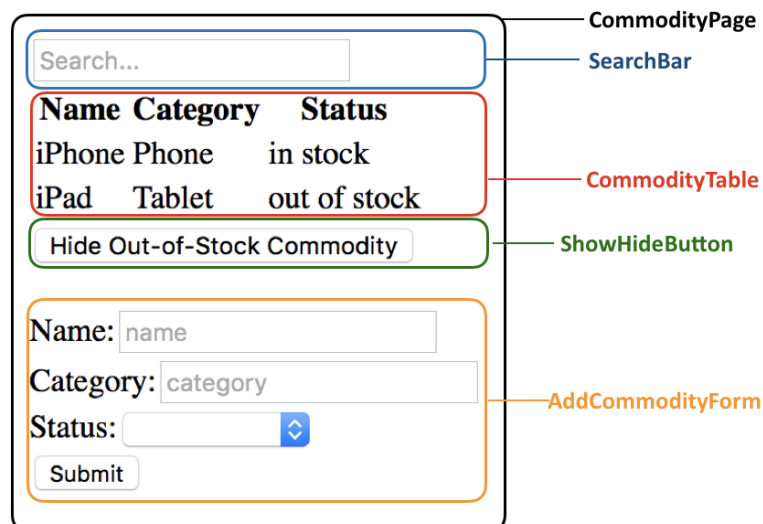


Fig. 7

There are currently three states `commodities`, `filterText` and `showOutOfStockCommodity` in the **CommodityPage** component, maintaining the commodities to be displayed in the table, text that user enters in the search box and the status of the "Show/Hide Out-of-Stock Commodity" button. Value of the `commodities` state is used in the **CommodityTable** component (and the **AddCommodityForm** component to be implemented); value of the `filterText` state is used in both the **SearchBar** component and the **CommodityTable** component; and value of the `showOutOfStockCommodity` is used in both the **CommodityTable** component and the **ShowHideButton** component. That's why we maintain them in the parent component **CommodityPage**. The two event handler functions `handleFilterTextChange` and `handleButtonClick` set the values of `filterText` and `showOutOfStockCommodity` states upon change event on the search box and click event on the button, respectively.

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 commodities from the web service, which we have launched in "[Prepare the 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 callback function, we bind "this" to the function using `.bind(this)` in the code.

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

```
class SearchBar extends Component {
  constructor(props) {
    super(props);
    this.handleFilterTextChange = this.handleFilterTextChange.bind(this);
  }

  handleFilterTextChange(e) {
    this.props.onFilterTextChange(e.target.value);
  }
}
```



```

render() {
  return (
    <form>
      <input
        type="text"
        placeholder="Search..."
        value={this.props.filterText}
        onChange={this.handleFilterTextChange}
      />
    </form>
  );
}
}

```

The component returns a `<form>` element, containing a search input text box. The value displayed in the search box is decided by `filterText` contained in the **props** passed into the component, which is value of the state `filterText` in the `CommodityPage` component. The event handler `handleFilterTextChange` in the `SearchBar` component passes user input value in the search box to the event handler `handleFilterTextChange` in the `CommodityPage` component, through its `props.onFilterTextChange`.

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

```

class CommodityTable extends Component {
  render() {
    const filterText = this.props.filterText;
    const showOutOfStockCommodity = this.props.showOutOfStockCommodity;

    var rows = this.props.commodities.map((commodity) => {
      if (commodity.name.indexOf(filterText) === -1) {
        return null;
      }

      if (showOutOfStockCommodity || commodity.status === "in stock") {
        return (
          <CommodityRow
            commodity={commodity}
            key={commodity.name}
          />
        );
      }
      return null;
    });

    return (
      <table>
        <thead>
          <tr>
            <th>Name</th>
            <th>Category</th>
            <th>Status</th>
          </tr>
        </thead>
        <tbody>{rows}</tbody>
      </table>
    );
  }
}

```

```
}
```

The component displays commodities in a table as follows:

- (1) The commodities displayed should match the search string, if a search string has been entered in the “Search...” box;
- (2) Out-of-stock commodities are displayed, if `props.showOutOfStockCommodity` received from the parent `CommodityPage` component is true (when the button element we are going to implement in **Step 6** displays “Hide Out-of-Stock Commodity”).
- (3) A `CommodityRow` component is used to return the table row showing each commodity.

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

```
class CommodityRow extends Component {
  render() {
    const commodity = this.props.commodity;

    return (
      <tr>
        <td>{commodity.name}</td>
        <td>{commodity.category}</td>
        <td>{commodity.status}</td>
      </tr>
    );
  }
}
```

**Step 6:** In `App.js`, add the following code to create the `ShowHideButton` component:

```
class ShowHideButton extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

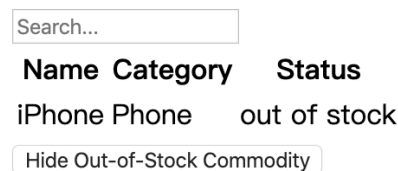
  handleClick() {
    this.props.onButtonClick();
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.props.showOutOfStockCommodity ? 'Hide Out-of-Stock
Commodity' : 'Show Out-of-Stock Commodity'}
      </button>
    );
  }
}
```

The component returns a `<button>` element. The text displayed on the button is decided by `showOutOfStockCommodity` contained in the **props** passed into the component, which is value of the state `showOutOfStockCommodity` in the `CommodityPage` component. The event handler `handleButtonClick` in the `ShowHideButton` component invokes the event

handler `handleButtonClick` in the `CommodityPage` component, through its `props.onButtonClick`.

**Step 7:** Now launch the app using “npm start” and browse the web page at <http://localhost:3000/>. You should see a page as shown below. You can test typing search string and clicking the button to see the effectiveness.



The screenshot shows a web application interface. At the top, there is a search bar with the placeholder text "Search...". Below the search bar, there is a table with three columns: "Name", "Category", and "Status". The table contains one row of data: "iPhone", "Phone", and "out of stock". Below the table, there is a button labeled "Hide Out-of-Stock Commodity".

Fig. 8

### Exercise 3: Adding a Component for Adding New Commodity

Next, we will add code in `App.js` to implement the following functionality: a form is displayed underneath the `ShowHideButton` component, which includes two text input boxes for entering name and category, a select element for selecting status, and a “Submit” button (see Fig. 7). After you have typed name and category, selected status, and clicked the “Submit” button, an HTTP POST AJAX request is sent to the web service, which stores the new commodity into the MongoDB database, and the newly added commodity should be added into the commodity table (see Figures 4-5).

To implement the above functionality, we need to add a component `AddCommodityForm`, and render it in the `CommodityPage` component.

**Step 1:** In the `CommodityPage` component, add three additional states as follows:

```
constructor(props) {
  super(props);
  this.state = {
    commodities: [],
    filterText: '',
    showOutOfStockCommodity: true,
    newCommodityName: '',
    newCommodityCategory: '',
    newCommodityStatus: ''
  };

  this.handleFilterTextChange =
this.handleFilterTextChange.bind(this);
  this.handleButtonClick = this.handleButtonClick.bind(this);
}
```

Then add three event handlers as follows to handle the change events on the name input text box, category input text box and status select element (we are going to implement these components in the `AddCommodityForm` in **Step 2**), respectively. These event handlers change the respective state values in the `CommodityPage` component according to user input values in the `AddCommodityForm`. In addition, you should add code to bind “this” to the three functions.

```

handleNameChange(name) {
  this.setState({
    newCommodityName: name
  })
}

handleCategoryChange(category) {
  this.setState({
    newCommodityCategory: category
  })
}

handleStatusChange(status) {
  this.setState({
    newCommodityStatus: status
  })
}

```

Add another event handler as follows to handle the submit event on the form in the [AddCommodityForm](#) component. Again, you should add code to bind “this” to this function in the [CommodityPage](#) component

```

handleAddFormSubmit(e) {
  alert("Add (" + this.state.newCommodityName + ", " +
this.state.newCommodityCategory + ", " + this.state.newCommodityStatus +
") to the form");
  $.post("http://localhost:3001/users/addcommodity",
  {
    "category" : this.state.newCommodityCategory,
    "name" : this.state.newCommodityName,
    "status" : this.state.newCommodityStatus
  },
  function(data, status){
    if (data.msg === ''){
      let newcommodities=this.state.commodities;
      newcommodities.push({
        "category" : this.state.newCommodityCategory,
        "name" : this.state.newCommodityName,
        "status" : this.state.newCommodityStatus
      });
      this.setState({
        commodities: newcommodities,
        newCommodityName: '',
        newCommodityCategory: '',
        newCommodityStatus: ''
      });
    } else
      alert(data.msg);
  }).bind(this)
);
e.preventDefault();
}

```

In this event handler, we produce an AJAX POST request using jQuery’s [\\$.post](#) API. When a success response is received from the server side, the new commodity is added into the state [commodities](#) array. The value of [commodities](#) is used by the [CommodityTable](#)

component to decide the table rows to display, and hence the new commodity is to be displayed in the [CommodityTable](#).

Next, you are going to develop the [AddCommodityForm](#) component, which is **the only component you have to implement in this Workshop**. In the [CommodityPage](#) component, you should add code for rendering the [AddCommodityForm](#) component after the [ShowHideButton](#) component, passing in the following **props**:

- the value of `newCommodityName` state
- the value of `newCommodityCategory` state
- the value of `newCommodityStatus` state
- the event handler to handle the change event on the name input text box
- the event handler to handle the change event on the category input text box
- the event handler to handle the change event on the select element
- the event handler to handle the submit event on the form

```
<p></p>
<AddCommodityForm name={this.state.newCommodityName}
  category={this.state.newCommodityCategory}
  status={this.state.newCommodityStatus}
  onChange={this.handleChange}
  onCategoryChange={this.handleCategoryChange}
  onStatusChange={this.handleStatusChange}
  onFormSubmit={this.handleAddFormSubmit}
/>
```

**Step 2:** Implement the [AddCommodityForm](#) component returning a `<form>` element, which includes a name input text box, a category input text box, a status select element, and a submit button. The view should be like Fig. 1. The select element can have the following options:

```
<option value="0"></option>
<option value="in stock">in stock</option>
<option value="out of stock">out of stock</option>
```

You should associate values of the name, category and status input elements with respective states in the [CommodityPage](#) component (i.e., `newCommodityName`, `newCommodityCategory`, and `newCommodityStatus`) through the **props** that the [AddCommodityForm](#) component receives (similar to how the value of the search box in the [SearchBar](#) component is associated with the state `filterText` in the [CommodityPage](#) component).

In addition, in the [AddCommodityForm](#) component, you should implement the code for passing the handling of the change events on the input elements and the submit event on the form to event handlers `handleChange`, `handleCategoryChange`, `handleStatusChange`, and `handleAddFormSubmit` in the [CommodityPage](#) component (similar to how handling of change event on the search box in the [SearchBar](#) component is passed to `handleFilterTextChange` in the [CommodityPage](#) component).

```
this.handleChange = this.handleChange.bind(this);
// omit handleCategoryChange, handleStatusChange and handleAddFormSubmit
```

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 adding a new commodity to test your code.

When you complete the project, your workshop5 directory should be in the following file hierarchy structure:

```
.
├── myreactapp
│   ├── README.md
│   ├── node_modules
│   ├── package-lock.json
│   ├── package.json
│   ├── public
│   │   ├── favicon.ico
│   │   ├── index.html
│   │   └── manifest.json
│   └── src
│       ├── App.css
│       ├── App.js
│       ├── App.test.js
│       ├── index.css
│       ├── index.js
│       ├── logo.svg
│       └── serviceWorker.js
└── serverapp
    ├── app.js
    ├── bin
    │   └── www
    ├── data
    ├── node_modules
    ├── package-lock.json
    ├── package.json
    ├── public
    │   ├── images
    │   ├── javascripts
    │   └── stylesheets
    ├── routes
    │   ├── index.js
    │   └── users.js
    └── views
        ├── error.pug
        ├── index.pug
        └── layout.pug
```