COMP3322 Modern Technologies on World Wide Web

Workshop – 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 environment in workshop 5.

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).

Fig. 1   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).

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.

Fig. 4   Fig. 5

Prepare the Web Service

Create a folder "workshop6". Inside the "workshop6" folder, make a copy of your workshop5 project folder, and rename the folder name to "serverapp". In this workshop, we are going to run the app you built in workshop 5 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 to allow CORS access to the server API.

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()
  .then(docs => {
    console.log(docs);
    res.json(docs);
  })
  .catch(err => {
    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()
  .then(() => {res.send({ msg: '' });} )
  .catch(err => res.send({ 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). If your web service does not care about the origin, you can simply set Access-Control-Allow-Origin: * which means that the resource can be accessed by any origin.

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 workshop5, 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 5):

Open a terminal for c3322-node container 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

Open another terminal for c3322-node container and go to your "**workshop6**" directory, and create a React app named "**myreactapp**" using the following commands:

```
cd workshop6
npm create vite@latest myreactapp -- --template react
```

The system may install the "create-vite" package (if you haven't worked on React development before), please type "y" to accept the installation request.

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 in development mode as follows:

```
npm run dev
```

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

```
VITE v5.1.5  ready in 1518 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help

```

Press 'q + enter' to terminate the program.

The default setting of the React app is set to server.host=localhost and server.port=5173. Since our app is running inside the docker and we only expose port numbers 3000, 3001, 8000, and 8080, we have to change the default setting to allow access to the react app from the host environment.

Open this file "vite.config.js" (in the myreactapp folder) with an editor, change the export default setting to:

```
export default defineConfig({
  plugins: [react()],
  server: {
    host: "0.0.0.0",
    port: 3000,
  },
```

```
})
```

Start the program again with the command

```
npm run dev
```

You should see the output changed to

```
VITE v5.1.5  ready in 1834 ms

➜  Local:   http://localhost:3000/
➜  Network: http://172.18.0.3:3000/
➜  press h + enter to show help
```

Now enter the URL http://localhost:3000 on your browser. And a web page after loaded looks as follows:
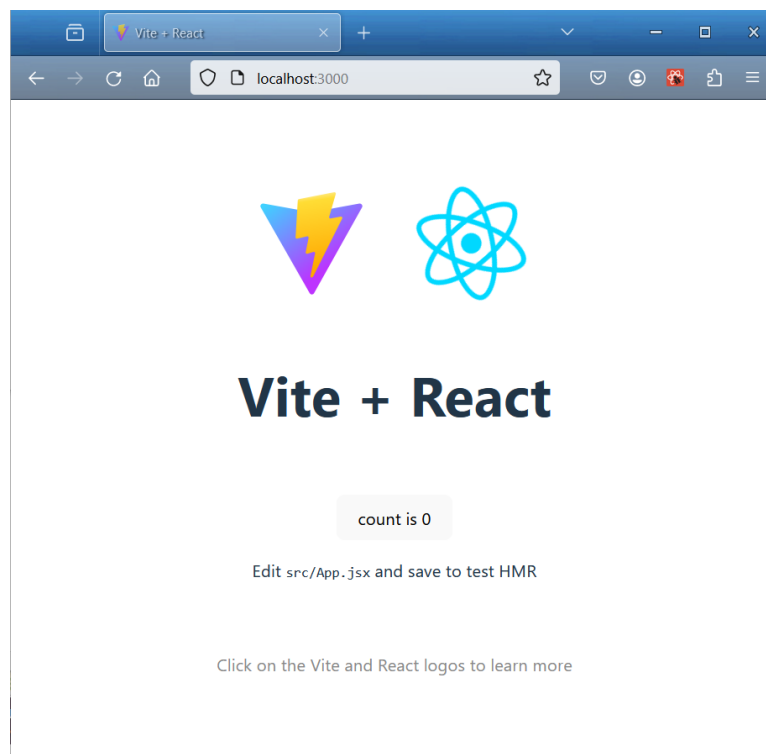

Fig. 6

Vite supports HMR (Hot Module Reload). To test HMR, edit the file App.jsx inside the src folder and change the paragraph to:

```
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
          <br />I add this line!!!
        </p>
```

You should find the program immediately reload the webpage with the new content!!!

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://react.dev/learn/add-react-to-an-existing-project

Exercise 1: Understand the Project File Structure

Step 1: The HTML file loaded after you launched the app using "npm run dev" is the index.html under myreactapp/ folder. 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 main.jsx and App.jsx in this directory and open them in a text editor.

Step 3: In main.jsx, it first loads **React** and **ReactDOM** modules. Please note that the current version of React is 18.2, which has a big change to the rendering APIs.

```
import React from 'react';
import ReactDOM from 'react-dom/client';  //this is for React 18
```

and css file:

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

and exported component from App.jsx file:

```
import App from './App.jsx';
```

main.jsx mainly renders the **App** component in the 'root' <div> element (in **index.html**).

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

For React 18, instead of directly invoking ReactDOM.render() (version before React 18), we call ReactDOM.createRoot() for mounting the root before calling render() to render the React element.

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

```
function App() {
  const [count, setCount] = useState(0)

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
```

```
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}
```

The function returns an "App" component, and the styling rules applied to this component are given in **App.css**. Note that the class attribute becomes className in React.

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

```
export default App;
```

When you have started the Reach app by "npm run dev", when you make changes to the code in main.jsx, App.jsx, and App.css, after you saved the changes, the system would automatically detect the changes and compile the React program again. The browser will automatically load the new React app and refresh the web page.

Exercise 2: Create our Web Page Using React

We are going to modify **main.jsx, App.jsx** and **App.css** to create the page as shown in Figures 1-5.

Step 1: Edit index.html, to replace the title of this app:

```
<title>Workshop 6 - React</title>
```

Step 2: In main.jsx, replace the content of main.jsx by the following code:

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

ReactDOM.createRoot(document.getElementById('root'))
.render(
  <React.StrictMode>
    <CommodityPage />
  </React.StrictMode>,
)
```

With the above code, we render the element returned by the CommodityPage component (to be implemented in App.jsx) in the "root" <div> (in myreactapp/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.jsx 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.jsx'; at the beginning of **main.jsx.**

Step 3: In App.css, replace the content of App.css by the following code:

```css
.App table {
  border-spacing: 10px;
}

.App label {
  display: inline-block;
  width: 80px;
}

.App input, .App select {
  margin: 3px 0;
}
```

With the above code, we add some space to the table, label, input, and select elements. They are the child elements of the CommodityPage component.

Step 4: In App.jsx , replace the content by the following code, which creates the CommodityPage component:

```jsx
import React, { useState, useEffect } from 'react';
import $ from 'jquery';
import './App.css';

function CommodityPage () {
  const [commodities, setCommodities] = useState([]);
  const [filterText, setFilterText] = useState('');
  const [showOutOfStockCommodity, setShowOutOfStockCommodity] = useState(true);

  function handleFilterTextChange(filterText) {
    setFilterText(filterText);
  }

  function handleButtonClick() {
    setShowOutOfStockCommodity(showOutOfStockCommodity
=> !showOutOfStockCommodity);
  }

  useEffect(() => {
    $.get("http://localhost:3001/users/commodities/",
      function(data) {
        setCommodities(data);
      }, 'json'
    );
  }, []);  //only run once after initial render
```

```
  return (
    <div className="App">
      <SearchBar
        filterText={filterText}
        onFilterTextChange={handleFilterTextChange}
      />
      <CommodityTable
        commodities={commodities}
        filterText={filterText}
        showOutOfStockCommodity={showOutOfStockCommodity}
      />
      <ShowHideButton showOutOfStockCommodity={showOutOfStockCommodity}
onButtonClick={handleButtonClick}/>
    </div>
  );
}

export default CommodityPage;
```

The component returns a <div> element with a class attribute 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.
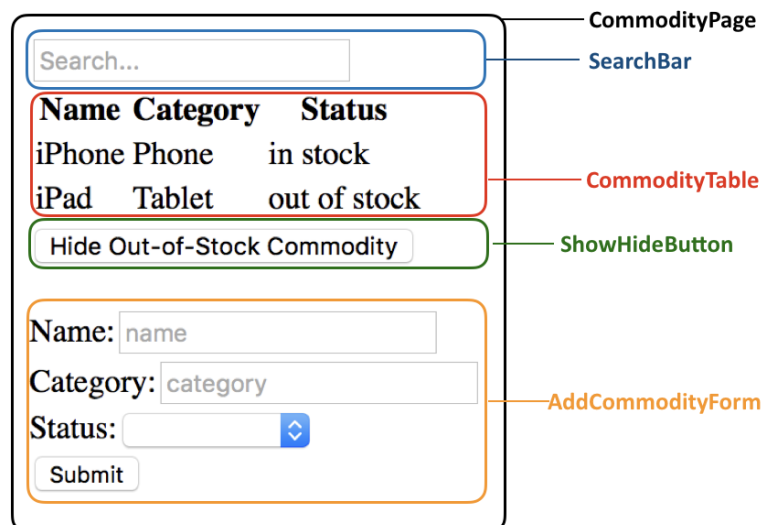

Fig. 7

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.

A setup function is defined in the useEffect Hook, and is invoked immediately after the component is mounted. Inside this function, we call a jQuery GET 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.jsx**. Refer to https://api.jquery.com/jquery.get/ to learn about our settings in the $.get function call.

Step 5: In App.jsx, add the following code (after the CommodityPage component) to create the SearchBar component:

```
function SearchBar ( props ) {

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

  return (
    <form>
      <input
        type="text"
        placeholder="Search..."
        value={props.filterText}
        onChange={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 6: In App.jsx, add the following code to create the CommodityTable component:

```
function CommodityTable ( props ) {
    const filterText = props.filterText;
    const showOutOfStockCommodity = props.showOutOfStockCommodity;

    var rows = 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 7: In App.jsx, add the following code to create the CommodityRow component:

```
function CommodityRow ( props ) {
  const commodity = props.commodity;

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

Step 8: In App.jsx, add the following code to create the ShowHideButton component:

```
function ShowHideButton ( props ) {

  function handleButtonClick() {
    props.onButtonClick();
  }

  return (
    <button onClick={handleButtonClick}>
      {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 9: Now launch the app using "npm run dev" and browse the web page at http://localhost:3000/. You should see the page as shown below. You can test typing search string and clicking the button to see the effectiveness.

Name	Category	Status
iPad	Tablet	out of stock
Mac mini	Computer	in stock
Macbook Air	Laptop	in stock
Dell XPS	Laptop	in stock
Apple Watch	Watch	in stock

Hide Out-of-Stock Commodity

Fig. 8

Exercise 3: Adding a Component for Adding New Commodity

Next, we will add code in App.jsx 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:

```
function CommodityPage () {
  const [commodities, setCommodities] = useState([]);
  const [filterText, setFilterText] = useState('');
  const [showOutOfStockCommodity, setShowOutOfStockCommodity] =
useState(true);
  const [newCommodityName, setNewCommodityName] = useState('');
  const [newCommodityCategory, setNewCommodityCategory] = useState('');
  const [newCommodityStatus, setNewCommodityStatus] = useState('');

```

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.

```
function handleNameChange(name) {
  setNewCommodityName(name);
}

function handleCategoryChange(category) {
  setNewCommodityCategory(category);
}

function handleStatusChange(status) {
  setNewCommodityStatus(status);
}
```

Add another event handler (in the CommodityPage component) as follows to handle the submit event on the form in the AddCommodityForm component.

```
function handleAddFormSubmit(e) {
  alert("Add ("+newCommodityName + ", "+ newCommodityCategory + ", " +
newCommodityStatus + ") to the form");

  let newData = {
      "category" : newCommodityCategory,
      "name" : newCommodityName,
      "status" : newCommodityStatus
    }

  $.post("http://localhost:3001/users/addcommodity",
    newData,
    function(data, status){
      if (data.msg ===''){
        setCommodities([...commodities, newData]);
        setNewCommodityName('');
        setNewCommodityCategory('');
        setNewCommodityStatus('');
      } else
        alert(data.msg);
    }
  );

  e.preventDefault();
}
```

In this event handler, we produce an AJAX POST request using jQuery's $.post API. When a successful 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 <mark>the only component you have to implement in this Workshop</mark>. 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={newCommodityName}
  category={newCommodityCategory}
  status={newCommodityStatus}
  onNameChange={handleNameChange}
  onCategoryChange={handleCategoryChange}
  onStatusChange={handleStatusChange}
  onFormSubmit={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. Please use
s to arrange these four form inputs in separate lines. The view should be like Fig. 1. The select element can have the following options:

```
<option value=""></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 handleNameChange, 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).

Launch the React app using "npm run dev" 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 workshop6 directory might be in the following hierarchy structure:

```
workshop6
├── myreactapp
│   ├── README.md
│   ├── node_modules
│   ├── package-lock.json
```

```
├── package.json
├── public
├── index.html
├── vite.config.js
└── src
    ├── App.css
    ├── App.jsx
    └── main.jsx
serverapp
├── app.js
├── bin
│   └── www
├── data
├── node_modules
├── package-lock.json
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
├── routes
│   ├── main.jsx
│   └── users.js
└── views
    ├── error.pug
    ├── index.pug
    └── layout.pug
```