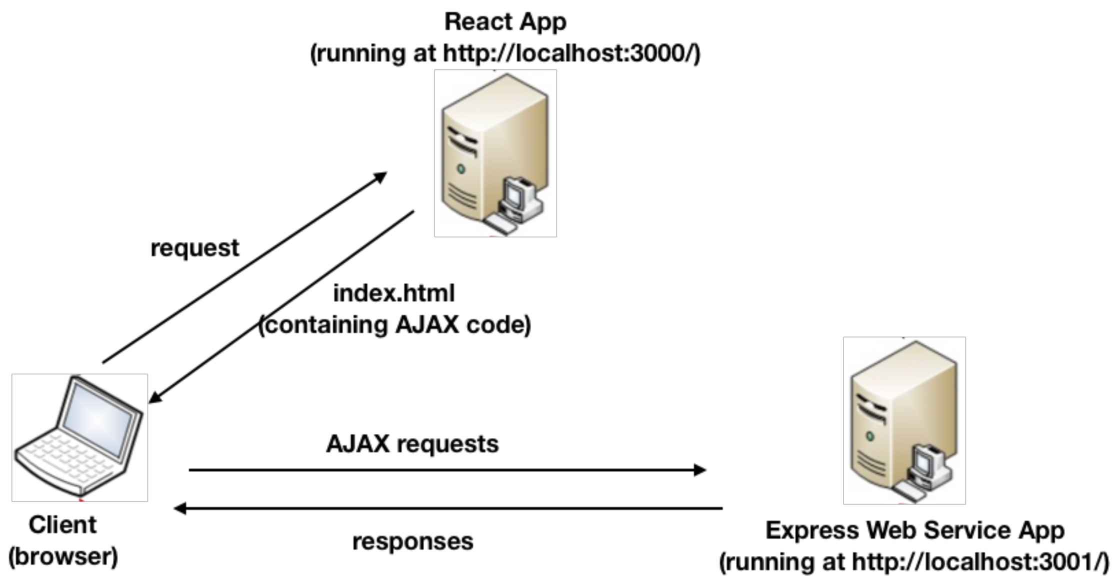
COMP3322A Modern Technologies on World Wide Web

Lab 8: MERN

Overview

In this lab exercise, we will use **React** to re-implement the front-end of the web service that we have built in Lab 6. In this way, we are using MongoDB, Node.js/Express.js and React, i.e., the MERN framework. The resulting web page will look identical as Lab 6 and have exactly the same behaviour.

To re-implement the front end in React, we use the **create-react-app** package to create a development server which hosts the files needed to run the React frontend app. We will reuse the web service that we built using the node.js/express.js environment in lab 6 (including the MongoDB database). The React app will be running on *localhost:3000*, while the Express.js server will be running on *localhost:3001*. The following diagram shows the interaction between your browser, the React App, and the Express.js server:



Lab Exercise

Part 1. Set up the Back-end Web Service

Step 1. Recreate the back-end web service we implemented in **lab6.**

Create a folder "**lab8**". Download lab6_samplesolution.zip from Moodle, extract it and rename the folder to "**webservice**". Copy the folder into the "**lab8**" folder. Follow step 3 in **Lab 6** to set up the MongoDB database (we will still use the database "lab6-db"). Please make sure that you can run the app in "**webservice**" (as did in Lab 6) and see the same web page as in Lab 6 before proceeding to the following steps. This ensures that the back-end service is working correctly.

Step 2. Enable CORS in the Web service

In this lab, we are going to run the web service built in Lab 6 as the back-end service, and allow our React app (front-end) to access the Web service. That is, we are not using .pug templates to generate HTML contents in this lab (though the files exist in your downloaded lab6 sample solution folder), but only the web services provided in users.js (via app.js). We are going to run this web service on your localhost on the port 3001 (instead of 3000), since we are going to run our React app on the port 3000. We are using the front-end code served from localhost:3000 to send AJAX requests to localhost:3001, and localhost:3000 and localhost:3001 with different ports are considered as different domains. Therefore, we need to enable CORS (Cross-Origin Resource Sharing) on the express.js web server. See https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS for a more detailed introduction.

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 with various options (see https://www.npmjs.com/package/cors).

Open **app.js** in the "**webservice**" folder. Add the following code at the beginning of app.js:

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

Then, we make the app use the cors middleware by adding the following line below the line "var app = express();"

```
app.use(cors());
```

Further, we add the following line of code (highlighted in red) for handling pre-flight requests **before** registering the routers:

```
app.options('*', cors());
app.use('/', indexRouter);
app.use('/users', usersRouter);
```

A CORS pre-flight 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 web 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 pre-flight request will not be sent by your browser.) The front-end code does not need to deal with sending a pre-flight request (as browser automatically sends it). But at the back-end server, we need to handle/respond to such OPTIONS requests; that's why we add the above middleware into **app.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, static file serving, indexRouter and index.js from the "webservice" folder. Our React app will only make use of the Web service implemented by app.js and users.js, but not any other modules in the app you built in Lab 6.

At the end of app.js, replace "module.exports = app;" by the following code:

```
//module.exports = app;
app.listen(3001);
```

Step 3. Launch the web service

Make sure you have correctly set up the database, as specified in Step 1. Stop the **express.js server** (NOT the MongoDB server) if you have launched it in Step 1. Now, do not launch the service using "npm start". Instead, we launch the web service using the following command:

```
node app.js
```

This makes the express.js server run at 3001 port, following our specification.

Part 2. Create the React App

Step 4. Create the React App template using "create-react-app" command

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:

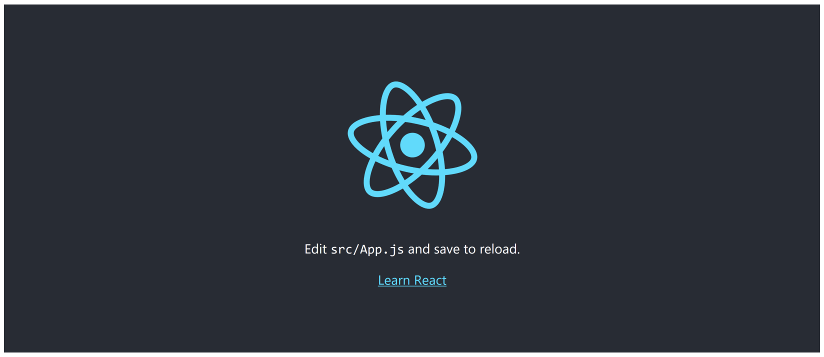
```
Compiled successfully!

You can now view myreactapp in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://xxx.xxx.xx.xx: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:



Edit `src/App.js` and save to reload.

Part 3. Prepare the React App

Step 5. Copy resource files to myreactapp/src folder

Copy the two files, logo.png and style.css from their respective folders of the backend-web service as follows, to myreactapp/src:

```
webservice/public/images/logo.png
webservice/public/stylesheets/style.css
```

We directly reuse logo.png and style.css from Lab 6 for the front-end. We will not need externalJS.js from Lab 6 since we will recreate the logic in React, but you may use it as a reference.

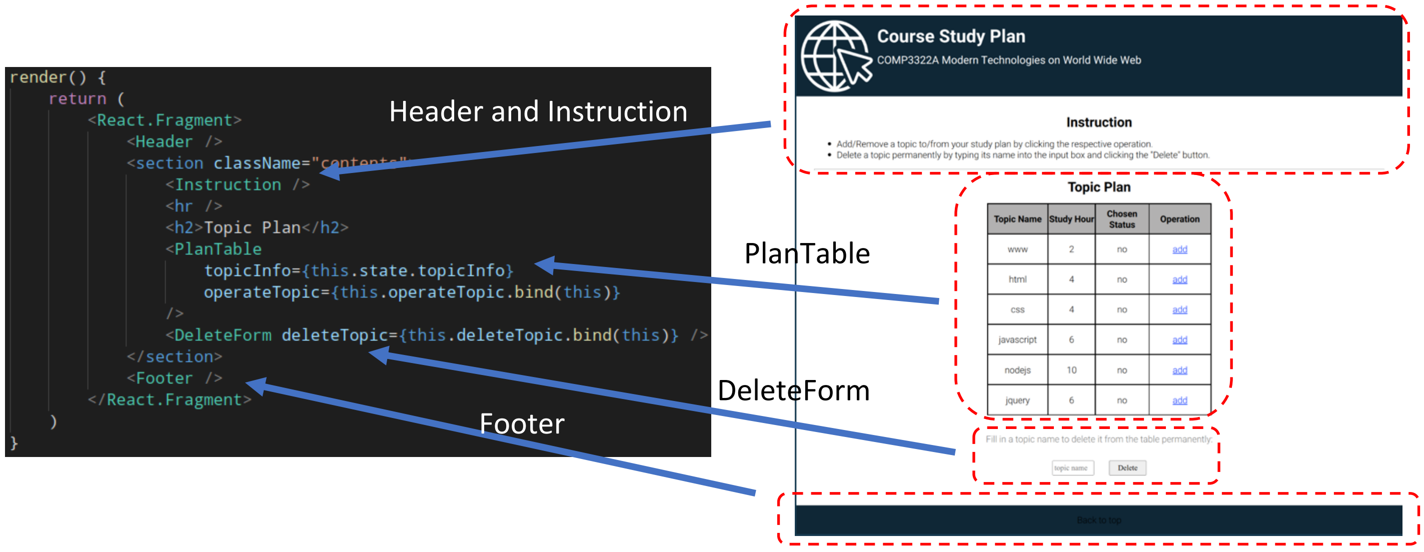
Step 6. Remove the default stylesheet

Edit myreactapp/src/index.js and remove the line that imports index.css. In this lab, we will only use one CSS file, “style.css”, which will be imported in “App.js”.

Step 7. Download code template from Moodle and inspect the provided code

From Moodle, download **lab8_materials.zip**. Extract it and you will find a file named App.js. Replace App.js in myreactapp/src with the downloaded file.

Open App.js and observe the render() function of the class “PlanPage”, which is the React component which renders the main page structure.
Here we have recreated the HTML structure similar to that achieved through index.pug in Lab 6.



```
render() {
    return (
        <React.Fragment>
            <Header />
            <section className="contents">
                <Instruction />
                <hr />
                <h2>Topic Plan</h2>
                <PlanTable
                    topicInfo={this.state.topicInfo}
                    operateTopic={this.operateTopic.bind(this)}
                />
                <DeleteForm deleteTopic={this.deleteTopic.bind(this)} />
            </section>
            <Footer />
        </React.Fragment>
    )
}
```

- Add/Remove a topic to/from your study plan by clicking the respective operation.
- Delete a topic permanently by typing its name into the input box and clicking the "Delete" button.

Topic Name	Study Hour	Chosen Status	Operation
www	2	no	add
html	4	no	add
css	4	no	add
javascript	6	no	add
nodejs	10	no	add
jquery	6	no	add

Fill in a topic name to delete it from the table permanently:

Besides normal HTML elements, the *PlanPage* component has the following components as children:

1. *Header*, *Instruction* and *Footer*: they are defined using functions and return static HTML code;

2. *PlanTable*: This is the component that creates the table containing topic plan.
3. *DeleteForm*: This is the component that creates the form at the bottom where one can delete topics.

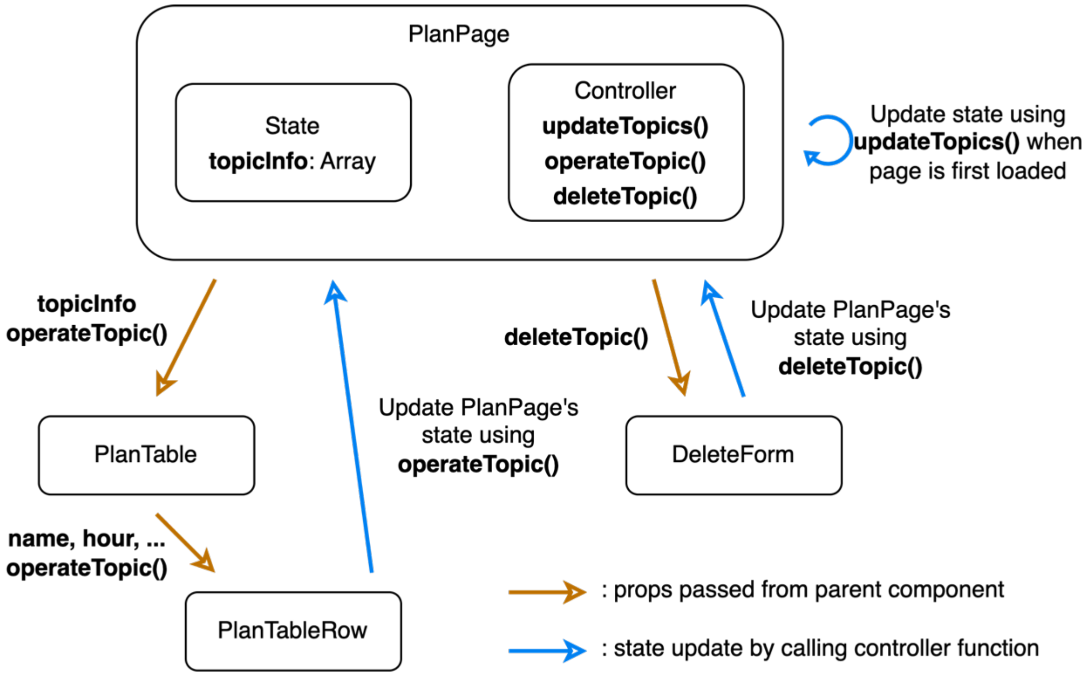
Here <React.Fragment> allows one to group a list of HTML elements/components without adding an extra node to the DOM (https://reactjs.org/docs/fragments.html), which makes it a better wrapping element than <div>.

The above describes what the HTML page is composed of. What is the interaction between the components? The design of this React App is centered around the *PlanPage's* state: the **topicInfo** array. It locally stores all the topics' information. The rendered contents of *PlanTable* and the behavior of *DeleteForm* will change based on the value of **topicInfo**:

1. The *PlanTable* receives information about topics from *PlanPage* through its **topicInfo** props. It will render the topics in **topicInfo** in its table rows through the component *PlanTableRow*. It also receives a function to handle the click event of the links to update topic status through its *operateTopic* props. *operateTopic* will update the internal state of *PlanPage*, which will in turn change the rendered HTML content of *PlanTable*.

2. The *DeleteForm* receives a function, *deleteTopic*, to handle the deletion of topics. The function uses the **topicInfo** state in *PlanPage* to check if the topic is in the table and will update it when executed successfully.

The following diagram summarizes the interaction between the components ("controller" refers to a function that updates a component's state):


PlanPage
State
topicInfo: Array
Controller
updateTopics()
operateTopic()
deleteTopic()
Update state using updateTopics() when page is first loaded
topicInfo
operateTopic()
deleteTopic()
Update PlanPage's state using deleteTopic()
Update PlanPage's state using operateTopic()
PlanTable
DeleteForm
name, hour, ...
operateTopic()
PlanTableRow
: props passed from parent component
: state update by calling controller function

Part 4. Implement PlanPage

Step 8. Implement the constructor of PlanPage

Locate the class "*PlanPage*". We have already provided the render function and empty controller functions "updateTopics", "operateTopic", and "deleteTopic" (which you will implement later). In this step, we implement the constructor of "*PlanPage"*, in which you need to:

1. Call the parent constructor with props as argument
2. Initialize the state of "PlanPage" with the key "**topicInfo**" and an empty array as the value.

Step 9. Implement the controller functions

Similar to the "showAllTopics", "operateTopic", and "deleteTopic" functions we implemented in Lab 6, the controller functions send AJAX requests to the backend web service. But this time instead of directly editing the page's content using DOM APIs, we only need to update the state in *PlanPage* since the page is automatically re-rendered when Plan*Page's* state changes.

Specifically, like "showAllTopics" we implemented in Lab 6, in the "updateTopics" controller function, we need to send an AJAX GET request for http://localhost:3001/users/get_table (note that we need to specify the full URL since the web service is running on another port). Same as in Lab 6, the web service will return a JSON object which is parsed to an array containing topics' information. We directly **set the state "topicInfo" to this array**. (You may reuse some of the code in Lab 6 to send the AJAX request.) Note that we have import $ from 'jquery'; at the beginning of App.js, which allows us to write jQuery code in the functions.

In "operateTopic", we send a PUT request for http://localhost:3001/users/update_status with the _id and op of the operation as we did in Lab 6, where _id and op are the arguments of the function. Same as in Lab 6, we use this _id to identify the topics to operate on and op (which is either "add" or "remove") to indicate the operation. After receiving the server response, call "updateTopics" function to fetch the updated topics information.

In "deleteTopic", we need to send an AJAX DELETE request for http://localhost:3001/users/delete_topic/*topicName*, where *topicName* is the argument of the "deleteTopic" function. Similar to Lab 6, we check if *topicName* exists in the table before sending the request. This time, we use the **topicInfo** state in *PlanPage* to check for the existence of the topic: if there is no such a topic in the table, alert "No such topic in the table"; otherwise, send the DELETE request to the back-end web service. After receiving the server response, call "updateTopics" function to fetch the updated topics information.

Note: you need to use .bind(this) on a callback function of a jQuery AJAX function call, if you want to use "this" in the callback function.

Step 10. Invoke updateTopics() upon page loading

To achieve this, call this.updateTopics() inside *PlanPage's* componentDidMount() function.

Part 5. Implement PlanTable

Step 11. Complete the function PlanTableRow

Locate the *PlanTableRow* function. This function creates a React component which renders a row in the *PlanTable*. *PlanTableRow* receives five attributes as its props:

1. _id: the _id of the topic from MongoDB, which we will use to call "operateTopic".
2. name: the name of the topic.
3. hour: the hours of the topic
4. status: the selection status of the topic
5. operateTopic: a function with two arguments, _id and op, that is to be called when the user clicks the "add" or "remove" action links, depending on the status of this topic.

The format of the table is exactly the same as in Lab 6.

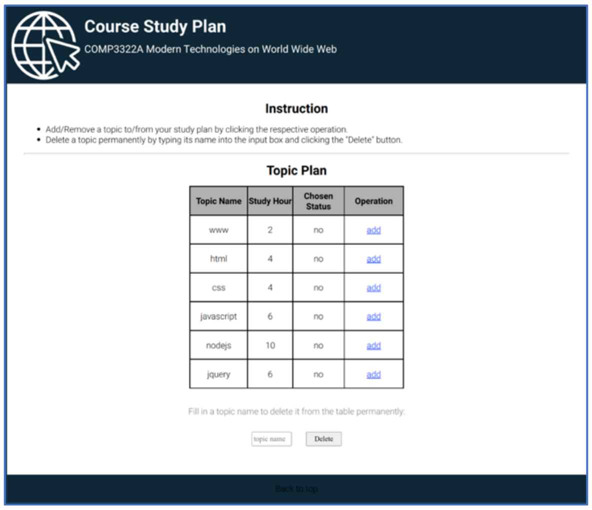
Complete the *PlanTableRow* function to return the HTML elements of a table row (including the add or remove link whose *onClick* is handled by *operateTopic*). Assign the table row to the *row_class* class for CSS styling. Assign the add or remove link to the *operation* class for CSS styling.

Step 12. Complete the implementation of *PlanTable*

Locate the function *PlanTable*. The *PlanTable* component is also defined using a function. It renders a table header as well as the table body containing the *PlanTableRows*. Its props contain the following fields:

1. topicInfo: an array of objects, each representing a topic. This is the **topicInfo** state passed from *PlanPage*.
2. operateTopic: a function. This function is passed in from the *PlanPage* and should be passed onto each *PlanTableRow*.

It uses a map function on the **topicInfo** array to generate a *PlanTableRow* component for each topic. Your task is to fill in the missing attributes (which will become props in *PlanTableRow*) for each table row.
Now when you load the page, you should see exactly the same table as in Lab 6:



Course Study Plan
COMP3322A Modern Technologies on World Wide Web

Instruction

- Add/Remove a topic to/from your study plan by clicking the respective operation.
- Delete a topic permanently by typing its name into the input box and clicking the "Delete" button.

Topic Plan

Topic Name	Study Hour	Chosen Status	Operation
www	2	no	add
html	4	no	add
css	4	no	add
javascript	6	no	add
nodejs	10	no	add
jquery	6	no	add

Fill in a topic name to delete it from the table permanently:

Part 6. Implement DeleteForm

Step 13. Implement the function deleteTopic()

Locate the class *DeleteForm*. Its props contain a function “deleteTopic”, which is passed from the *PlanPage* component. *DeleteForm* also contains a “deleteTopic” function itself, which should call the “deleteTopic” function from the props with the input topic name as the argument.

This component contains an input field and a submit button. In React, we use a technique called “controlled components” to manage the value of input fields. To manage the mutable “value” of an input field, we create a state for it in the containing component. These states correspond to the values of the input fields, and are updated whenever the values of input fields change. Therefore, the value of the state is always in sync with the value of the input field. For more detailed reference, see https://reactjs.org/docs/forms.html

Implement the function deleteTopic() in *DeleteForm*, which calls the “deleteTopic” function in props with the **inputTopicName** state as the argument.

Congratulations! Now you have finished Lab 8. You should see exactly the same page as in Lab 6. Test the behavior of the page to be identical with that implemented in Lab 6 by adding, updating and deleting some topics.

Submission:

Please finish this lab exercise before 23:59 Sunday Dec 4, 2022. You should submit a zip file containing the following files only:

1. In /webservice: app.js
2. In /myreactapp: App.js

(1) Login Moodle.
(2) Find “Labs” section 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.