

WA3488 ADP Phase 1 React Project - Overview & Work Steps

Overview

The project for phase01 involves using React to build the Customer management application.

Here are some high-level points to remember about the project:

- You are required to build a web app using React.
- The app should be functionally similar to the solution demo app that appears later in these instructions.
- You can start from scratch using create-react-app or by unzipping and running the starter application.
- A set of suggested steps to follow is available later in this document.

Development Environment

Project development environment setup includes:

- Windows, Mac or Linux PC
- Chrome web browser
- Visual Studio Code Editor/IDE
- NodeJS

You can use the supplied lab machines or your own computer if you already have a development environment.

The Project Zip

The supplied project zip is named: **WA3488-project-student.zip** should be copied to your development machine and unzipped in a convenient location. Note the location as you will be needing it later.

The Project Zip contains these files and directories:

Filename	Description
Project-Overview.pdf	This file
Application Requirements.pdf	Features and functionality the application must have

WA3488 ADP Phase 1 React Project - Overview & Work Steps

Application Testing.pdf	Manual testing scripts
/ProjectAssets	Directory containing demo and solution files

The Project Demo

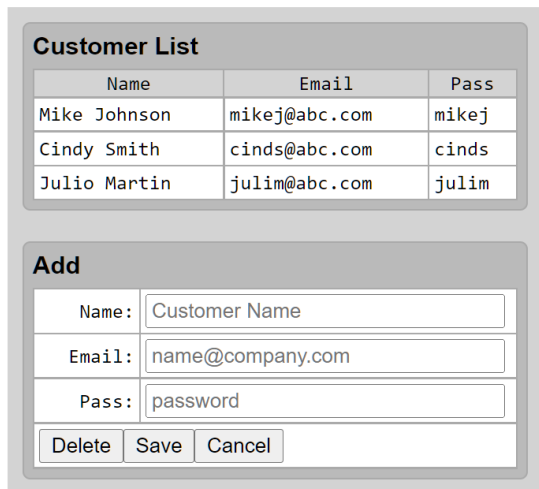
Before you start coding you should run and inspect the Project Demo. It is an example of what you will be trying to accomplish when creating your own application.

The project demo can be found here: **ProjectAssets\front-end-app-demo.zip**

Running the Demo:

1. Unzip the demo into your ProjectWork directory
2. Open a terminal
3. Navigate to the **demo** directory
4. Execute the **rundemo.bat** file
5. Wait for the server to start and the browser to pop-up - showing the application.

The demo application looks like this:



The screenshot displays a web application interface. At the top, there is a section titled "Customer List" containing a table with three columns: Name, Email, and Pass. The table lists three customers: Mike Johnson, Cindy Smith, and Julio Martin. Below the table is an "Add" section with three input fields for Name, Email, and Pass, each with a label and a text box. At the bottom of the "Add" section are three buttons: Delete, Save, and Cancel.

Name	Email	Pass
Mike Johnson	mikej@abc.com	mikej
Cindy Smith	cinds@abc.com	cinds
Julio Martin	julim@abc.com	julim

Add

Name:

Email:

Pass:

Whenever you have a question about how your application should work you can refer back to this demo. Your application does not have to look exactly like the demo but it should function in a similar fashion.

The demo uses an in-memory data structure that resets when you refresh the browser.

The Project Solution

The available project solutions have a number of uses:

- Solutions can be run and tested to see how the app is supposed to function
- Solution code can be inspected for ideas on how to proceed with your own coding
- If you run into trouble with your own implementation you can build on top of a given solution

The solution code comes in the form of a zipped up repository that is available here:

ProjectAssets\front-end-app-solutions.zip – solutions for stages 01-05

The solution for each stage of development is tagged in the repository and can be checked out as needed.

Unzip and setup the solution repository:

1. Unzip the contents of the **front-end-app-solutions.zip** file into your C:\ProjectWork directory. You should end up with the following: **C:\ProjectWork\front-end-app-solutions** directory.
2. Open a terminal & navigate to the project directory
3. Run "npm install" to download dependencies

Execute the following in the project directory to see a list of the available tags:

```
git tag
```

You should see:

```
01-initial-react-app
02-front-end-app-static
03-front-end-app-memdb
04-separate-components
05-uses-rest-server
```

The first tag corresponds to an empty react project, ready for development.

Each of the other tags corresponds to a specific project stage solution.

By default the repository's working directory contents correspond to the final solution (05...)

Checking out solutions:

Solutions can be checked out one at a time.

The command syntax used to check out a solution looks like this:

```
git checkout {tag-name}
```

example:

```
git checkout 02-front-end-app-static
```

Always shut down the React development server before checking out a new solution. You can restart it (npm run start) as soon as the solution has been checked out.

Note about project solutions:

When you have a question about how your application should be coded you can refer back to the project solution. Although your application should work like the solution does, your application code may not exactly match that of the solution. Where differences occur you should be able to explain how why you coded the way you did.

Application Requirements & Testing

Before coding starts, a list of requirements for the application to fulfill is created. A requirements list for the application you'll be coding - "**Application Requirements.pdf**" can be found in the student project zip file.

Use the requirements list as a guide to the features you need to add as you construct your application.

While features are being added to the application the developer performs manual testing based on test scripts to verify the application's functionality. A set of manual testing scripts has been provided in the "**Application Testing.pdf**" file that can be found in the project zip file.

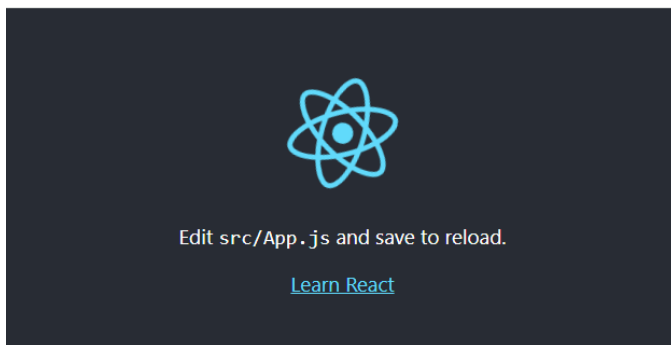
You should use the testing scripts to make sure your application's functionality fulfills the stated application requirements.

Basic Project Work Steps

Stage01 – Create an initial React app project

1. Create a new React project and name it "**front-end-app**"
2. Download dependencies
3. Start the application (`npm run start`)
4. Open the app in a browser: `http://localhost:3000/`

you should see this:



WA3488 ADP Phase 1 React Project - Overview & Work Steps

The state of the application at this point corresponds to the solution tag: "01-initial-react-app"

Remember to commit and tag your code before moving on.

Stage02 – Create a static version of the application

In this stage you will create a static version of the application that has the following features:

- A title
- A list of customers displaying "name", "email" and "password" for each
- A customer form with:
 - A title that displays the form's state: "Add" or "Update"
 - Labelled Input fields for: "name", "email" and "password"
 - "delete", "save" and "cancel" buttons

At this stage the application:

- Should not yet be styled with CSS
- Clicking Buttons (or clicking items in the list) should only output simple messages.
- Data should come from a static array that you set up in the App.js file
- All work should be done inside the App component.

The idea is to create a static HTML template for the application that you can built-out into a fully dynamic application using React. At this point JavaScript should be limited to simple function stubs.

When you are done with this stage the application should look what you see here below but fields and buttons should be inactive.

Customer List

Name	Email	Pass
Jack Jackson	jackj@abc.com	jackj
Katie Kates	katiek@abc.com	katiek
Glen Glenns	gleng@abc.com	gleng

Update

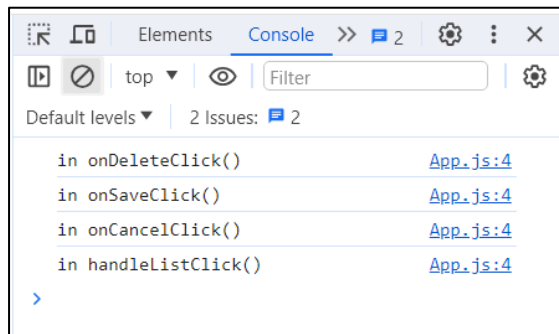
Name:

Email:

Pass:

When you click the various buttons in the app, "delete", "save", "cancel" your minimal JavaScript code should output messages to the console like this:

WA3488 ADP Phase 1 React Project - Overview & Work Steps



The same type of console logging should be done when you click on an item in the customer list.

The state of the application at this point corresponds to the solution tag: "02-front-end-app-static"

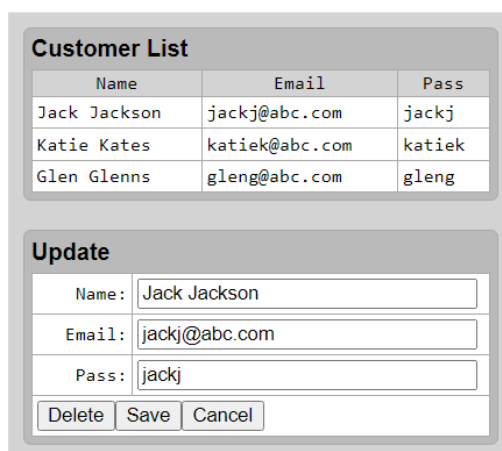
Remember to commit and tag your code before moving on.

Stage03 – Create a dynamic version of the application that uses in-memory data

In this stage you add styling and dynamic capabilities to the application.

Add Styling:

- CSS Styling should be added so that your app resembles the demo app
- Some styles already appear in the App.css file. You just need to make sure they are applied to the App component.



Name	Email	Pass
Jack Jackson	jackj@abc.com	jackj
Katie Kates	katiek@abc.com	katiek
Glen Glenns	gleng@abc.com	gleng

Update
Name:
Email:
Pass:

Add List-Item Selection:

Earlier, when you clicked on an item in the list, a message was output to the JavaScript console. But, the item was not selected. That is, the clicked-on item is not listed in bold and is not displayed in the form in the bottom half of the screen. The form in the bottom half of the screen currently shows a single, hard coded customer: Jack Jackson.

In this part of the project you should implement these requirements and make sure that the application's behavior when clicking on an item in the list matches what is written in the "Application Requirements.pdf" document.

- Items in the list are selected by clicking on them
- Only one item can be selected at a time
- The selected list item should appear in **bold** lettering
- The item selected in the list should appear in the input form
- Clicking on an already selected item should cause it to lose its selection

Hints:

- Manage the currently selected item with a React `useState()` function.
- Manage the object shown in the form with `useState()` as well.

Before moving on, think about how you would approach getting a selected list item to be rendered in bold. Then check the application to see which parts of your solution idea may already be in place.

Keeping track of requirements

Your app currently implements some of the requirements outlined in the "Application Requirements.pdf" file.

Go to the list and take note of which requirements are already implemented and which still need to be implemented. You might want to copy the requirements list into a text file or word document where you can mark items DONE as you complete them.

Add List-Item De-Selection:

Sometimes after selecting an item in the list you may want to de-select the item. You can do this if needed by selecting another item in the list. But what if you want to return the list to a state where no items are selected?

For this part of the project you should update the application code so that clicking on an already selected list item deselects the item and clears the form below the list.

Hints:

- If no items in the list are selected the `formObject` should be set to the "blankCustomer" variable.
- Customer items will always have an `id` ≥ 0 . The "blankCustomer" always has `id = -1`

WA3488 ADP Phase 1 React Project - Overview & Work Steps

- It might be a good idea to commit all existing changes to git before you start working on this. That way you can always revert the app back if you run into any issues while experimenting with the code.

Implement Cancel Function

When the app first opens, no customer is listed in the form at the bottom of the screen. Only after you click on a customer in the list is it shown in the form. After that you can change selected customers but you can't get back to the original state where no customer is selected.

In this part of the project you will implement the cancel button so that it takes the user back to the state where no customer is selected.

Before getting started think about how you might do this.

Hint: No items in the list will be highlighted if the formObject's value is taken from the "blankCustomer" variable.

After you've coded your implementation try testing it out:

- Refresh the app
- Click on an item to select it
- Click on the "Cancel" button.
- The customer should be de-selected and the form input fields are cleared.

This kind of manual testing can be done by following the test scripts that are located in the "application testing.pdf" file. Make it a habit to verify new feature functionality by running through the test scripts after each implementation.

Upgrade memdb.js

Depending on how you've proceeded up until now the data you see in the app has been provided by a hard coded array of customers:

- Inside the App.js file (if building on your own code) or
- Inside a minimal memdb.js file (if building on solution code)

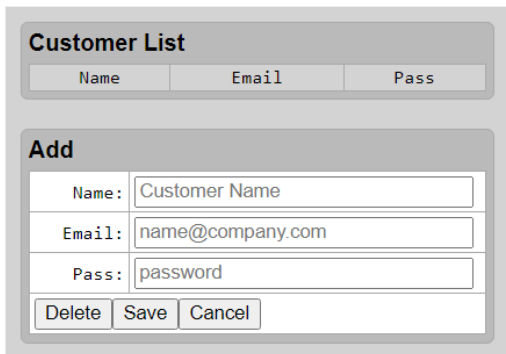
In this part of the project you are going to replace this bare-bones data array with a fully implemented memdb.js in-memory data store.

The new implementation includes not only an array but several functions that can be used to interact with the array. The functions will allow you to add, update and delete items from the array thus simulating a more realistic data store.

1. Copy the **ProjectAssets\memdb.js** file into the **\src** directory of your project. (if you already have a memdb.js file then overwrite it)
1. Import all the functions from memdb.js into your App.js file. (excluding non-exported functions)

WA3488 ADP Phase 1 React Project - Overview & Work Steps

2. Create a "customers" variable in the App component that is managed by React's useState() function. (pass an empty array to useState([]))
3. Save App.js
4. Go back to the browser and refresh the application. The compile errors should have gone away but no customers are showing in the application.



The screenshot shows a web application interface. At the top, there is a table titled "Customer List" with three columns: "Name", "Email", and "Pass". Below the table is a form titled "Add". The form has three input fields: "Name" with the placeholder text "Customer Name", "Email" with the placeholder text "name@company.com", and "Pass" with the placeholder text "password". At the bottom of the form are three buttons: "Delete", "Save", and "Cancel".

This should make sense. Remember we created the "customers" variable by passing an empty array to useState([]). To update the "customers" variable we'll need to use another React hook, this time it will be the useEffect hook.

Update Customers with the useEffect Hook

React's useEffect hook allows you to perform side effects like data fetching, direct DOM updates, and timer functions in React components.

In this part of the project you will add a useEffect hook to your app that retrieves a list of customers from the updated memdb.js code. After retrieving the list useEffect re-renders the application so that customer data appears in the list on-screen.

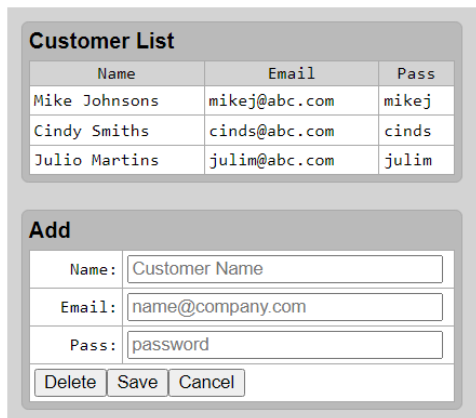
1. Create the following getCustomers function in your App component:

```
const getCustomers = function() {  
  log("in getCustomers()");  
}
```

2. Add an import for useEffect to App.js
3. Add a useEffect statement to the App component that calls the getCustomers() method
4. Update the "getCustomers" so that it calls "setCustomers" with the output of calling the getAll() function that was imported from the updated memdb.js file.
5. Save the App.js file

WA3488 ADP Phase 1 React Project - Overview & Work Steps

6. Go to the browser and refresh the application. A different set of data(from the updated memdb.js file) should be shown:



The screenshot shows a web application interface. At the top, there is a section titled "Customer List" containing a table with three columns: "Name", "Email", and "Pass". The table lists three customers: Mike Johnsons, Cindy Smiths, and Julio Martins. Below the table is an "Add" section with three input fields labeled "Name:", "Email:", and "Pass:". The "Name:" field contains the placeholder text "Customer Name". Below the input fields are three buttons: "Delete", "Save", and "Cancel".

Name	Email	Pass
Mike Johnsons	mikej@abc.com	mikej
Cindy Smiths	cinds@abc.com	cinds
Julio Martins	julim@abc.com	julim

Add

Name:

Email:

Pass:

Check Requirements and Re-Test the App

Updating the memdb.js file was a relatively fundamental and important change to the app. It is possible for recent code changes to have broken some of the previously completed features. In this part of the project you will go back and confirm which features/requirements the application fulfills in its current state.

1. If you are using Git, now would be a good time to commit changes. In general after adding features or fixing bugs is usually a good time to commit changes. That's because the application is functioning as expected.
2. Take a look at the "Application Requirements" list (the one in ProjectAssets or a writable copy you are working with).
3. Verify and mark down which requirements the current application fulfills. You may want to test the application to make sure previously implemented features are still functioning. You should use the test scripts in the "Application Testing" document (located in ProjectAssets). The numbers of the test scripts match the numbers of the requirements that they test.
4. Sometimes a requirement that you thought you had implemented already will fail during testing. For example, clicking cancel no longer clears the selected item or the form input fields. In such a case you should go back to the code and try to determine what is responsible for the failure. What you will find fits into one of these categories:
 - The existing implementation is incomplete. Only part of the feature was successfully implemented.
 - Changes made after you completed the implementation have caused the implementation to fail. This is often referred to as a 'regression'.
5. Fix any broken or incomplete implementations.
6. Make a note of all the successfully completed requirements.
7. Commit the fixes to Git.

Implement the Delete Button

The purpose of the delete button is to remove the currently selected record. In this part of the project you should make the delete button work as intended.

Make the following changes to your app component:

- Update the onDeleteClick function to delete the currently selected customer using the memdb "deleteById" function.
 - If no customer is currently selected, exit the function without calling deleteById
 - After the deletion, clear the form by calling setFormObject with blankCustomer
1. Save your changes, refresh the app and test it.
 2. After deleting a few customers, set the app's data to its original state (three customers) by refreshing the browser again

Implementing Form Modes

Your application needs to add new records and to update existing records. The forms for these two types of functions are almost the same. In this case you are going to reuse the same form. To do that you will need to know which "mode" the form is in at any point in time.

Do the following to implement form modes in your application:

- Create a string variable named "mode" that will hold either "Add" or "Update"
- Set the mode variable's value based on the current formObject. The form should be in "Add" mode when it is displaying a new empty record (the "blankCustomer" with id=-1). The form should be in "Update" mode when it is displaying one of the customers from the customer list. In this case the id is always ≥ 0
- Display the mode in the header above the form
 1. Save your changes and update the browser
 2. Because no customer is selected, the "mode" initially displayed in the form header should be "Add"
 3. Click to select an item in the list.
 4. The displayed mode should change to "Update"
 5. Click once more on the same item to de-select it
 6. The displayed mode should change back to "Add"
 7. If your app does not work like this then make whatever changes are needed to get it working.

Hint: Do NOT manage the "mode" variable with React useState()

Activate the Input Fields

Because of the way React works, input fields are not active by default, meaning that if you try to enter or modify text in the input field nothing will happen. The default value of the input field will remain the same no matter what keystrokes you enter into the field.

In order to have the fields update you need to implement an onChange handler that saves the data for each keystroke and then calls React to refresh the screen.

1. Open the App.js file
3. Find the code for the "handleInputChange" method

```
const handleInputChange = function (event) {  
  log("in handleInputChange()");  
}
```

This code is currently called whenever you enter a keystroke into any of the input fields.

4. Verify the above by checking the JSX code that displays the "name" input field.

```
<tr>  
  <td className={'label'} >Name:</td>  
  <td><input  
    type="text"  
    name="name"  
    onChange={ (e) => handleInputChange(e) }  
    value={formObject.name}  
    placeholder="Customer Name"  
    required /></td>  
</tr>
```

5. Any keystrokes entered into the input field trigger the handleInputChange(e) method.
6. Go to the browser and refresh the app.
7. Make sure that the Browser's JavaScript console is open and cleared.
8. Place the cursor in the "Name" input field and type some text.
9. You should see this line appear in the console:

```
in handleInputChange() App.js:5
```

10. Note that when the exact same message is sent to the console multiple times it is indicated by a number to the left of the message like this:

```
7 in handleInputChange() App.js:5
```

11. Now we know that handleInputChange is called for each keystroke, but the keystrokes are still not showing up in the input fields.
12. Go back to the "handleInputChange" method and add the code shown in bold below:

```
const handleInputChange = function (event) {
```

WA3488 ADP Phase 1 React Project - Overview & Work Steps

```
log("in handleChange()");
const name = event.target.name;
const value = event.target.value;
let newFormObject = {...formObject}
newFormObject[name] = value;
setFormObject(newFormObject);
}
```

Read through the code, it should do the following:

- Get the name attribute value from the input field
- Get the input field's value (which includes the newly entered text)
- Clones the current formObject
- Updates the property of the field that was changed
- Calls setFormObject which updates the formObject and re-renders the page

The code here has been implemented in multiple lines to make what its doing clear. A much shorter working implementation is also possible. (want to give that a try? get the one above working first before you do.)

13. Save the App.js file.
14. Go to the browser and refresh the app.
15. Place the cursor in the "Name" input field and type some text. This time the text will show up in the input field.
16. Click on one of the items in the customer list. This sets the initial values of the input fields to match the selected customer record.
17. Place the cursor in the "name" field and try editing the value. All of the characters or delete or backspace keystrokes will show up as you edit the field.

Check Requirements and Re-Test the App

Its possible that the additional changes you've made to the application may have broken some of the previously completed features. In this part of the project you will go back and confirm which features/requirements the application fulfills in its current state.

1. Go back to the previous "Check Requirements and Re-Test the App" section of this document and follow the instructions listed there. When you are done, come back here.
2. Take a look at your requirements list and note which requirements are still left to implement.
3. If there are requirements that are not on the list but that you've implemented in your code, add them to the list!

Implement the Save Button

Although you can now type text into the input fields the data is not yet saved. In this part of the project you should update the "onSaveClick" method so that it adds new records and updates existing records when the "save" button is clicked.

WA3488 ADP Phase 1 React Project - Overview & Work Steps

1. Open the App.js file and find the code for that handles the "save" button's click event.

```
let onSaveClick = function () {  
  log("in onSaveClick()");  
}
```

This code is not currently doing much.

2. Update the onSaveClick method as shown in bold below so that:
 - It checks if the current form mode is "Add"
 - When mode is "add" it calls the "post()" method in memdb.js to insert the new record
 - It checks if the current form mode is "Update"
 - When mode is "update" it calls the "put()" method in memdb.js to update an existing record
 - After add or update the code clears the current formObject by setting it to blankCustomer.
This has the additional effect of de-selecting any item currently selected in the customer list.

3. Test your implementation

- Save the App.js file and refresh the browser.
- Check that the mode shows up in the form header as "Add"
- Type some values like these into the input fields:

name: John Doe
email: johnd@abc.com
pass: pass

- Click on the Save button. The newly added customer should show up in the list.
- Click on the newly added record in the list to select it. The form title should change to "Update"
- Edit the "Pass" input field and change the password to "johnd"
- Click on the Save button. The updated password should show in the list.

The state of the application at this point should corresponds to the solution tag: "03-front-end-app-memdb"

Remember to commit and tag your code before moving on.

Stage04 - Refactor the App into Separate Components

The application is currently implemented as a single component (App). In this part of the project you will separate the application into three components.

- App
- CustomerList
- CustomerAddUpdateForm

This change will, among other things, increase modularity, support reusability and simplify testing.

For this stage, it's up to you to formulate and implement the requirements as you see fit.

Hint: Although the ultimate aim is to have each of the separate components in their own files, you can do it in stages. In the first stage you would still keep all the components in the same file. When you have that working you would do whatever extra work is needed to separate them into individual files.

Hint: Its a good idea to commit all existing changes to Git before you start working on this. That way you can revert back if you run into any issues while refactoring the code.

- 1 As you work, save your files frequently and update the browser page to check your work.
- 2 Try separating out a the components one at a time doing the simplest one (CustomerList) first.

The state of the application after separating out components corresponds to the solution tag: "04-separate-components"

Remember to commit and tag your code before moving on.

Stage05 - Refactor the App to use REST Server

The application is currently setup to use an in-memory array inside of memdb.js as a data store. Real world applications are more likely to use a REST API as their data source. In this part of the project you will update the application to use a typical REST API.

This is an extra-credit(optional) part of the project. Your options are:

- do this part
- skip this part

For this stage, it's up to you to formulate and implement the requirements as you see fit.

Hint: Its a good idea to commit all existing changes to Git before you start working on this. That way you can revert back if you run into any issues while refactoring the code.

Loose Work Steps:

1. Start up the provided REST server:
 - Open a terminal
 - Unzip REST server from **ProjectAssets\back-end-rest-server.zip** into your ProjectWork directory
 - Install dependencies: **npm install**
 - Start the server: **npm run start**
 - Open a browser to: **http://localhost:4000/customers**
2. In your front-end project, Copy **src\memdb.js** to **src\restdb.js**

WA3488 ADP Phase 1 React Project - Overview & Work Steps

3. Create fetch statements in **restdb.js** that:

- in `getAll()`, retrieves the whole list of customers
- in `post()`, adds a new customer
- in `put()`, updates an existing customer
- in `deleteById()`, deletes an existing customer

4. Here's a sample fetch statement to get you started. It calls the REST service requesting a list of all customers and then updates the "customers" variable with the data it receives:

```
const baseUrl = 'http://localhost:4000/customers';

export async function getAll(setCustomers) {
  const myInit = {
    method: 'GET',
    mode: 'cors' };
  const fetchData = async (url) => {
    try {
      const response = await fetch(url, myInit);
      if (!response.ok) {
        throw new Error(`Error fetching data: ${response.status}`);
      }
      const data = await response.json();
      setCustomers(data);
    } catch (error) {
      alert(error);
    }
  }
  fetchData(baseUrl);
}
```

Note how "setCustomers" is being passed to `getAll`. This is needed so that `setCustomers` can be called after the data is returned.

Old Call to `getAll()`

```
// App.js
const getCustomers = function(){
  log("in getCustomers()");
  setCustomers(getAll());
}
```

New call to `getAll()`

```
// App.js
const getCustomers = function () {
  log("in getCustomers()");
  getAll(setCustomers);
}
```


WA3488 ADP Phase 1 React Project - Overview & Work Steps

You will need to create fetch statements similar to the one above for the other operations: POST, PUT and DELETE.

For these other operations you will need to pass a method that resets the formObject after the REST operation completes.

Here is an example of the old and the new call to "deleteById()":

Old Call to deleteById()

```
// App.js
let onDeleteClick = function () {
  if(formObject.id >= 0){
    deleteById(formObject.id);
  }
  setFormObject(blankCustomer);
}
```

New call to deleteById()

```
// App.js
let onDeleteClick = function () {
  let postOpCallback = () => { setFormObject(blankCustomer); }
  if (formObject.id >= 0) {
    deleteById(formObject.id, postOpCallback);
  } else {
    setFormObject(blankCustomer);
  }
}
```

Hint: If the customer list is not refreshing after operations like add, update or delete try altering the useEffect statement to add formObject as a dependency, like this:

```
useEffect(() => { getCustomers() }, [formObject]);
```

5. Update the operations one at a time in this order. Get each one working before moving on to the next:
 getAll(), post(), put(), delete()
6. As you proceed, verify that your changes are having the desired effect by testing the application.

The state of the application after separating out components corresponds to the solution tag: "05-uses-rest-server"

Remember to commit and tag your code before moving on.

Where to go from here

After completing the application through stage05, if you still have some time you can try implementing some of the requirements listed below. There are no solution files for these requirements. You will verify that you have succeeded by passing the manual tests that you come up with before you start coding.

Choose one of the following topics to work on.

Topic01 - Update the application to work with larger data sets

Up until now you have been working with data sets that include just a few customer records. But what about if there were 100 customers, or more. What kind of issues might you expect when working with that amount of data in the current application?

Try enhancing the application based on the requirements set out below. Feel free to adjust the requirements or add to them as you get to know more about the issues.

Requirements for larger data sets

- The customers list should be able to function with up to 100 records.
- The list should display a number indicating how many records it is holding
- The list should have scroll bars for easier navigation
- Entering a search term should narrow down the list
- The list should include pageup and pagedown controls

High Level Work Steps

- Create a data set with at least 100 customers.
- Substitute the new data set by replacing the data.json file in the rest server directory and restarting the server
- To enable scrolling, wrap the current HTML list with a div that has a fixed height. And add CSS styles to enable the vertical scroll bar.
- Add an input field where the user can enter some search text along with a button that applies the search term and another that cancels the search term.
- Use the JavaScript Array.filter() method to apply the search term(text) to restrict the list.
- As an alternative to scrolling, set a pagesize and only show that number of customers on each page. Maintain the currentPage value with a React useState variable. Then add pageUp and pageDown buttons that set the currentPage value. deactivate the buttons as needed to avoid having the user go past the end (or beginning) of the data while paging.

Hints:

- You don't need to implement both paging and scrolling. Just one or the other.
- Use an online data generator like the one at <https://www.mockaroo.com/> to create the data set.

Topic02 – Use Separate pages for the customer list and the add/update form

The current application all fits on one page. But what if you want more room for the list or you just don't want to see the add/update form unless you need it? In that case you might want to update the application to use more than one page. For example, when the app starts it will display the customer list. The user would then click an "Add" or "Update" button to navigate to a separate add/update page. To do this you would most likely need the React router.

Try enhancing the application based on the requirements set out below. Feel free to adjust the requirements or add to them as you get to know more about the issues.

Requirements for separate list and edit pages

- The initial page should hold the customers list
- The initial page should have an "add" and an "update" button
- The add and update buttons should navigate to the second page
- The second page should show in add/update form as well as "save", "cancel" and "delete" buttons
- "cancel" navigates the user back to the initial list
- "save" button adds or updates a customer record depending on the form's current mode
- The "delete" button deletes the currently selected customer when the form is in update mode

High Level Work Steps

- Add the React router to the project (install and import it)
- Create an EditPage component to hold the addUpdate component
- Add routing code to the App component
- Pass the currently selected customer when navigating to the EditPage
- If no customer is passed then the AddUpdate form should come up in "add" mode
- Disable the "Update" button when no customer is selected

Hints:

- Use the App component as the initial page that shows the list
- Use the EditPage component as the second page that comes up and shows the AddEditForm