Hands-on Lab – Async Callback (30 mins)

Skills Network

Objective for Exercise:

After completing this lab, you will be able to:

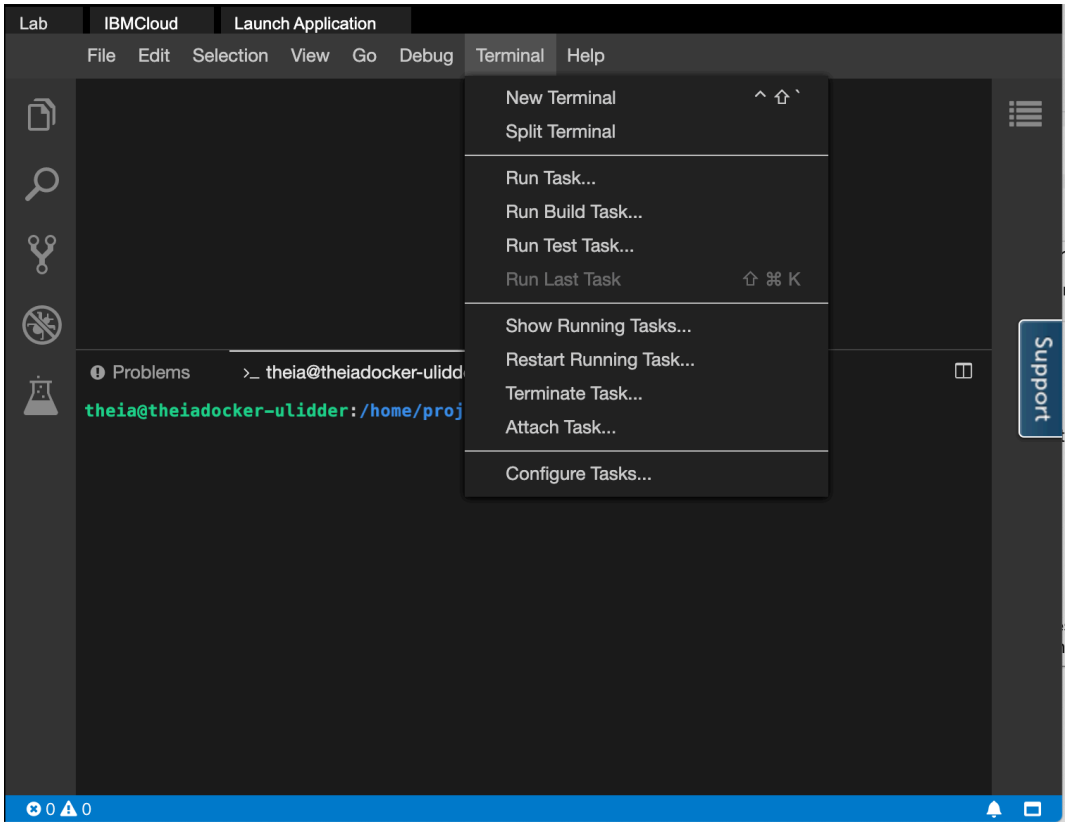
- Describe asynchronous callbacks
- Create a Node.js application

Prerequisites

- Basic knowledge of JavaScript

Task 1 : Set-up – Check for the cloned files

1. Open a terminal window by using the menu in the editor: Terminal > New Terminal.



2. Change to your project folder.

```
cd /home/project
```

3. Check if you have the folder **lkpho-Cloud-applications-with-Node.js-and-React**

```
ls /home/project
```

If you do, you can skip to step 5.

4. Clone the git repository that contains the artifacts needed for this lab, if it doesn't already exist.

```
git clone https://github.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-Node.js-and-React.git
```

5. Change to the directory for this lab.

   ```
   cd lkpho-Cloud-applications-with-Node.js-and-React/CD220Labs/async_callback/
   ```

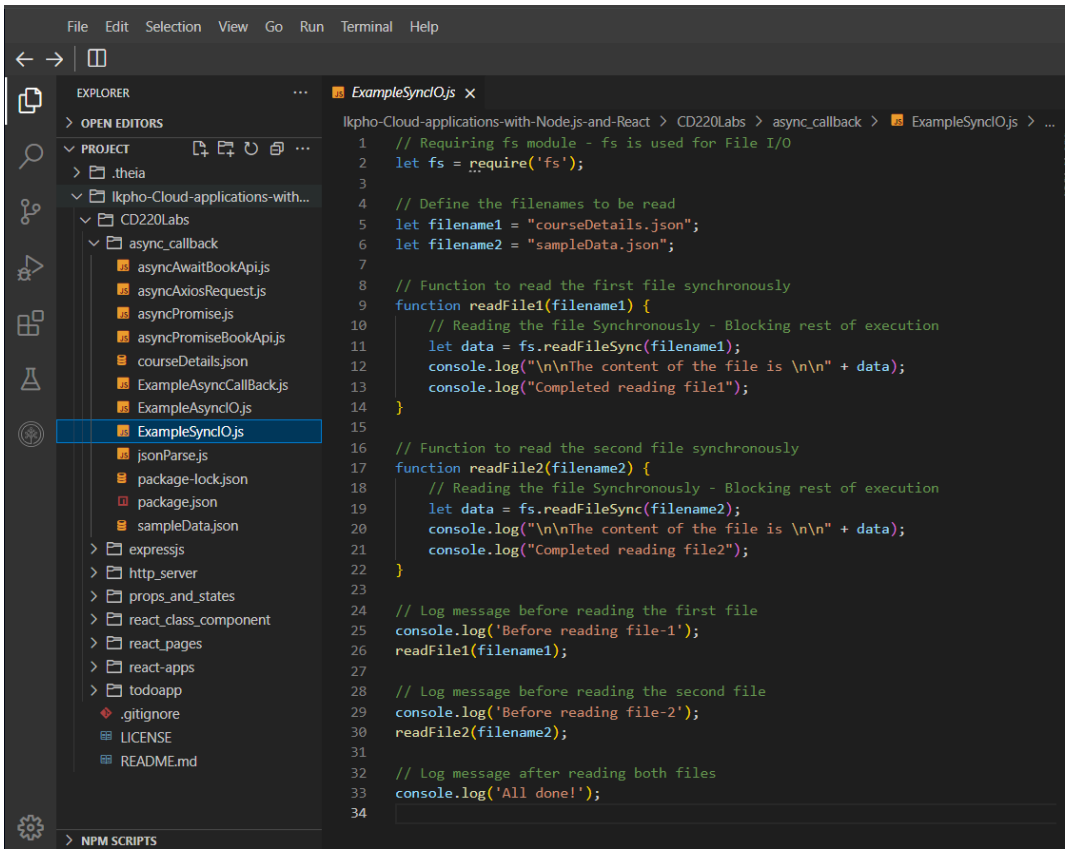
6. List the contents of this directory to see the artifacts for this lab.

   ```
   ls
   ```

   You must have a few exercise files that you will be running in the exercises.

Task 2: Synchronous I/O

In the files explorer view ExampleSyncIO.js. It would appear like this.



▶ You can also click here to view the code

Here is an explanation of the code in it:

- `require('fs')` imports the Node.js built-in fs module, which provides file system-related functionality.
- `readFile1(filename1)` and `readFile2(filename2)` define functions to read `courseDetails.json` and `sampleData.json` respectively synchronously using `fs.readFileSync`, which blocks further execution until each file is read.
- Before calling `readFile1(filename1)` and `readFile2(filename2)` functions, log messages are printed to indicate the start of the reading process for each file.
- After initiating the reading of both files, a final log message "All done!" is printed to indicate that the code has executed beyond the file reading initiation.

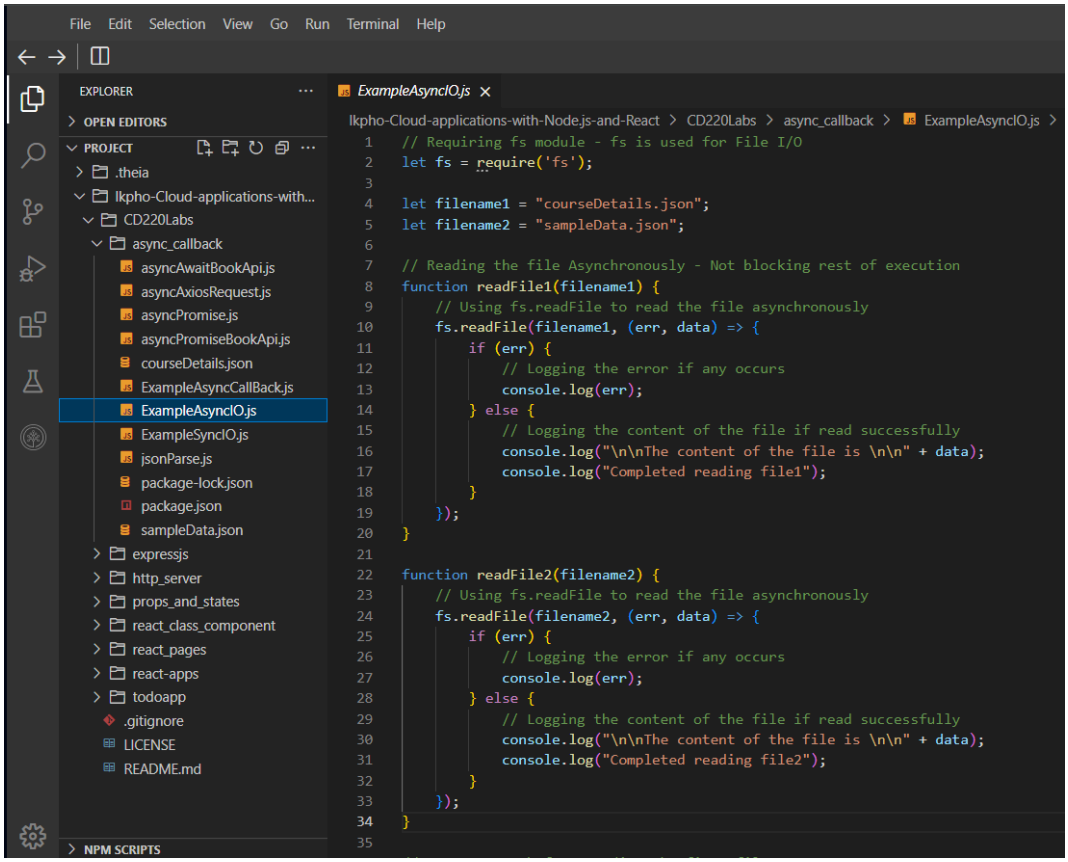
2. In the terminal window run the server with the following command.

   ```
   node ExampleSyncIO.js
   ```

Observe that the two files are being read synchronously, one after the other. This will be evident from the order in which the console log appears.

Task 3: Asynchronous IO

1. In the files explorer view ExampleAsyncIO.js. It would appear like this.



▶ You can also click here to view the code

Explanation of the Code

- `require('fs')` imports the Node.js built-in fs module, which provides file system-related functionality.
- `readFile1(filename1)` and `readFile2(filename2)` define functions to read `courseDetails.json` and `sampleData.json` respectively using `fs.readFile`, which reads the files asynchronously, allowing the rest of the code to execute without waiting for the file reading to complete.
- In each function, `fs.readFile(filename, callback)` reads the specified file. If an error occurs during reading, it logs the error; otherwise, it logs the content of the file and a completion message.
- Before calling `readFile1(filename1)` and `readFile2(filename2)` functions, log messages are printed to indicate the start of the reading process for each file.
- After initiating the reading of both files, a final log message "All done!" is printed to indicate that the code has executed beyond the file reading initiation.

2. In the terminal window run the server with the following command.

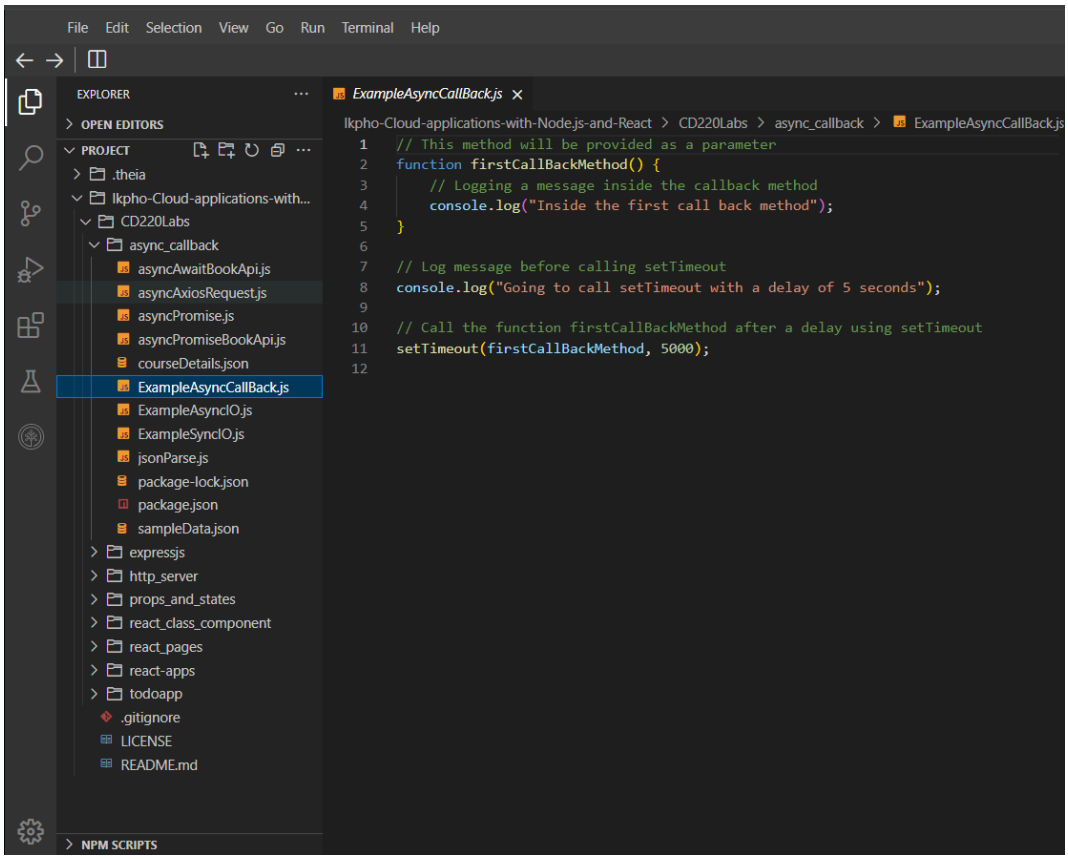
```
node ExampleAsyncIO.js
```

Observe that the two files are being read asynchronously. This will be evident from the order in which the console log appears. The following three console log appears before the file content is printed though the logs are called in the code in different order.

```
Before the reading the file-1
Before the reading the file-2
All done!
```

Task 4: Creating Callback Functions

1. In the files explorer view ExampleAsyncCallBack.js. It would appear like this.



▶ You can also click here to view the code

Explanation of the Code

- `firstCallBackMethod()` is a function that logs a message "Inside the first call back method". This function will be used as a callback.
- Before calling `setTimeout`, a log message "Going to call setTimeout with a delay of 5 seconds" is printed to indicate the impending timer setup.
- `setTimeout(firstCallBackMethod, 5000)` schedules the execution of `firstCallBackMethod()` after a delay of 5000 milliseconds (5 seconds), allowing the rest of the code to continue executing immediately while the timer runs in the background.

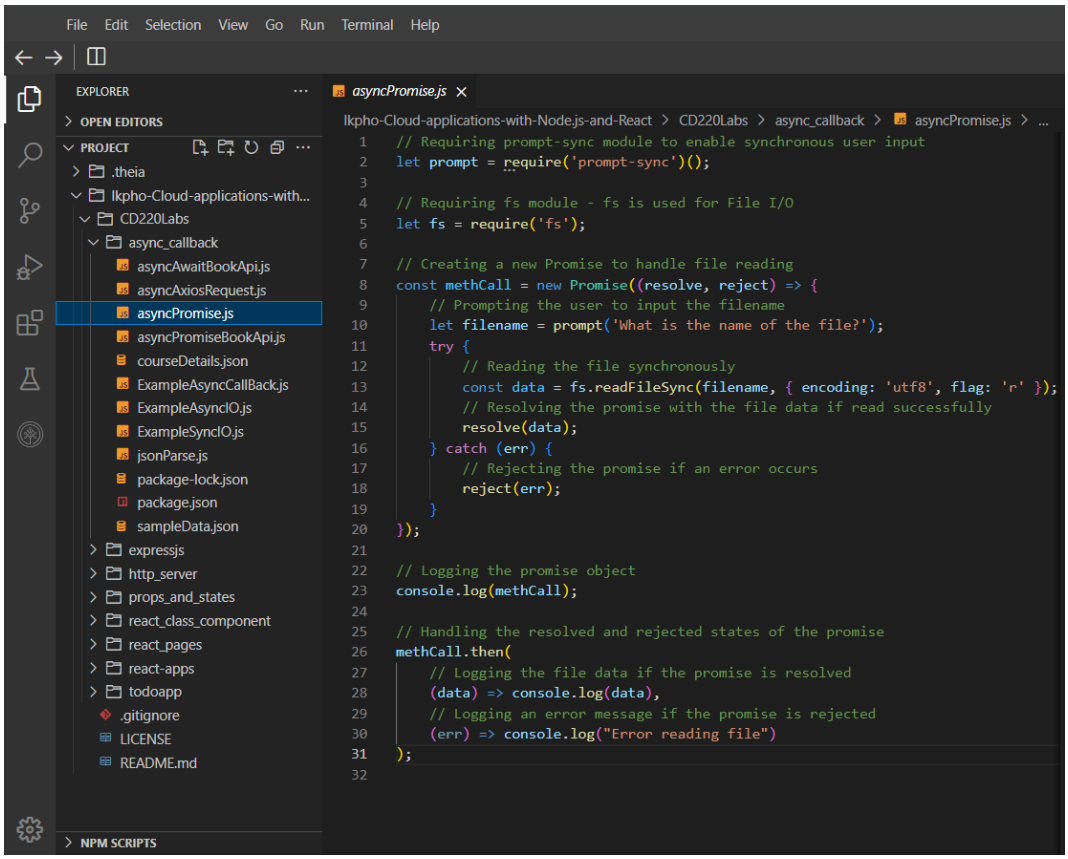
2. In the terminal window run the server with the following command.

   ```
   node ExampleAsyncCallBack.js
   ```

setTimeout is a built-in library method which allows you to pass a method which needs to be called on timeout, as aparameter. Here firstCallBackMethod is defined and then passed as a parameter to setTimeOut. As you may have observed, the method will be called after 5 seconds. This is called `callback`

Task 5: Promises

1. In the files explorer view asyncPromise.js. It would appear like this.



▶ You can also click here to view the code

Explanation of the Code

- `require('prompt-sync')()` imports the `prompt-sync` module to enable synchronous user input. `require('fs')` imports the Node.js built-in `fs` module for file system-related functionality.
- `const methCall = new Promise((resolve, reject) => { ... })` creates a new promise to handle the file reading process. The promise takes a callback function with `resolve` and `reject` parameters.
- Inside the promise:
  - `let filename = prompt('What is the name of the file?')` prompts the user to input the filename.
  - `fs.readFileSync(filename, { encoding: 'utf8', flag: 'r' })` attempts to read the specified file synchronously with UTF-8 encoding.
  - If the file is read successfully, `resolve(data)` is called with the file data, resolving the promise.
  - If an error occurs, `reject(err)` is called with the error, rejecting the promise.
- `console.log(methCall)` logs the promise object.
- `methCall.then(...).catch(...)` handles the promise:
  - `then((data) => console.log(data))` logs the file data if the promise is resolved.
  - `catch((err) => console.log("Error reading file"))` logs an error message if the promise is rejected.

2. In the terminal window run the following command to install prompt-sync. Using `--save` ensures that the package.json file is updated for dependencies.

   ```
   npm install --save prompt-sync
   ```

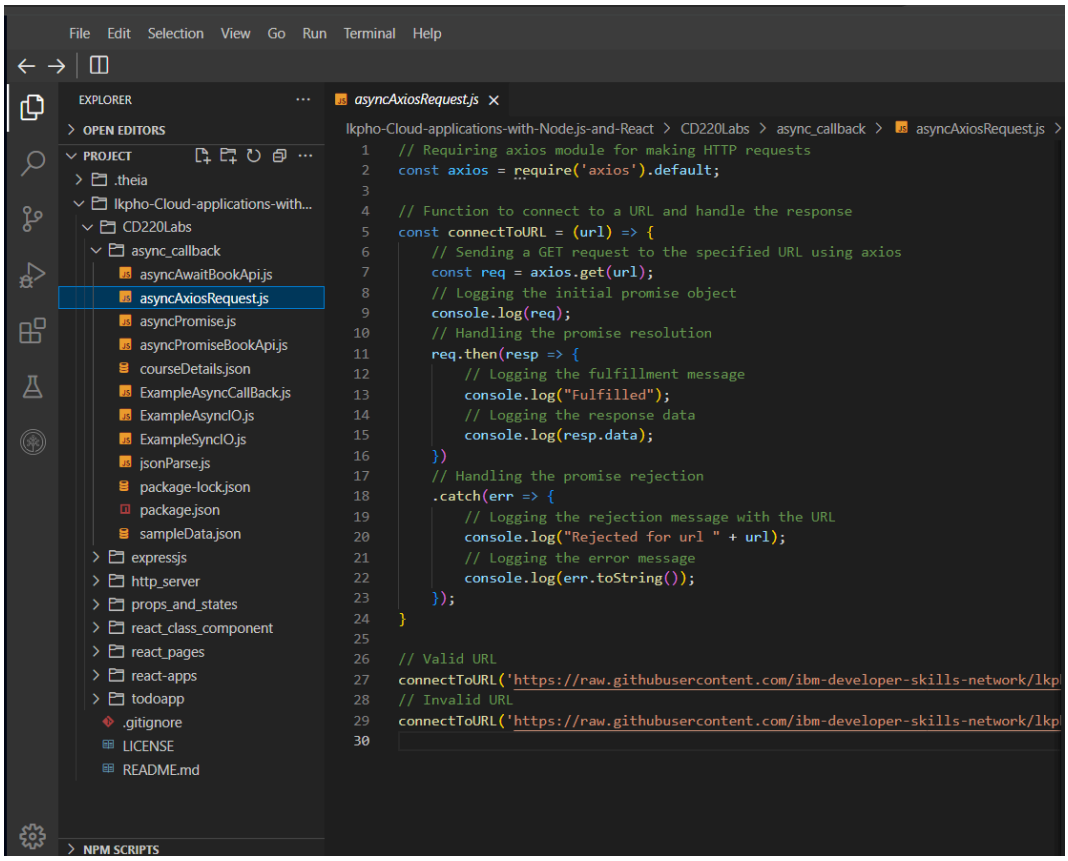
3. In the terminal window run the server with the following command. It will ask you for a filename. Enter a valid filename from the current directory.

   ```
   node asyncPromise.js
   ```

methCall here is a promise object. When the promise is full-filled, the console log will be printed. Run the above command again and try to pass an invalid filename. See the console log printed out as the promise is being rejected.

Task 6: AsyncAxiosRequest

1. In the files explorer view asyncAxiosRequest.js. It would appear like this.



▶ You can also click here to view the code

Explanation of the Code

- `require('axios').default` imports the `axios` module, which is used for making HTTP requests.
- `connectToURL(url)` defines a function that takes a URL as a parameter and sends a GET request to that URL using `axios.get(url)`. It logs the initial promise object returned by `axios.get`.
- Inside the `connectToURL` function:
  - `req.then(resp => { ... })` handles the promise resolution by logging "Fulfilled" and the response data (`resp.data`) if the request is successful.
  - `req.catch(err => { ... })` handles the promise rejection by logging "Rejected for url" followed by the URL and the error message if the request fails.
- The `connectToURL()` function is used to call a URL. Providing a valid URL should successfully fetch the data, while an invalid URL should result in an error and activate the rejection handler.

2. To run this code, we need to install axios package. Run the following command to install axios.

```
npm install --save axios
```

3. In the terminal window run the code with the following command.

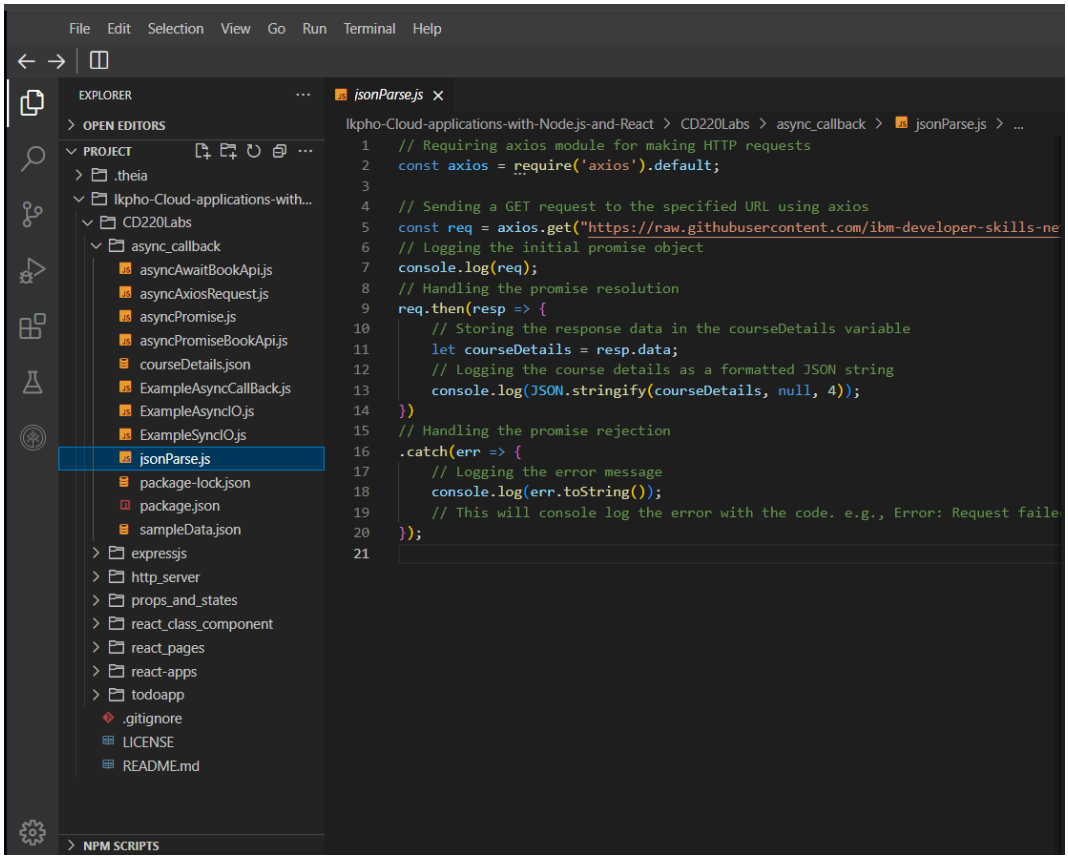
```
node asyncAxiosRequest.js
```

When you run the code, the first connectToURL is an axios request to a valid URL which will return JSON object. The second connectToURL is an axiosRequest to an invalid URL. This will return appropriate error message. The output will be as below.

```
Promise { <pending> }
Promise { <pending> }
Rejected for url https://raw.githubusercontent.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-No
de.js-and-React/master/CD220Labs/async_callback/sampleDate.json
Error: Request failed with status code 404
Fulfilled
{ name: 'Jason', age: '25', department: 'IT' }
```

Task 7: Working with JSON

1. In the files explorer view jsonParse.js. It would appear like this.



▶ You can also click here to view the code

Explanation of the Code

- `require('axios').default` imports the `axios` module, which is used for making HTTP requests.
- `axios.get()` sends a GET request to the specified URL to fetch the `courseDetails.json` file.
- The `req` constant stores the promise returned by `axios.get`.
- `console.log(req)` logs the initial promise object to the console.
- `req.then(resp => { ... })` handles the promise resolution:
  - `let courseDetails = resp.data` stores the response data in the `courseDetails` variable.
  - `console.log(JSON.stringify(courseDetails, null, 4))` logs the course details as a formatted JSON string with indentation.
- `req.catch(err => { ... })` handles the promise rejection:
  - `console.log(err.toString())` logs the error message, which includes the error code and description if the request fails (e.g., `Error: Request failed with status code 404`).

2. To run this code, we need to install axios package. You will be having the axios module that you installed in the previous exercise. If not, run the following command to install axios.

```
npm install --save axios
```

3. In the terminal window run the code with the following command.

```
node jsonParse.js
```

When you run the code, an axios request is made to a remote URL which returns a JSON object. This JSON object is stringified(or formatted in a readble form) and logged on the console. The output will be as below.

```
Promise { <pending> }
{
    "course": {
        "name": "Cloud Application Development",
        "modules": {
            "module1": {
                "name": "Introduction to Server-Side JavaScript",
                "topics": [
                    "Course Intro",
                    "Module Introduction",
                    "Introduction to Node.js",
                    "Introduction to Server-Side Javascript",
                    "Creating a Web Server with Node.js",
                    "Working with Node.js Modules",
                    "Lab - Introduction to Server-side JavaScript",
                    "Module Summary",
                    "Practice Quiz",
                    "Graded Quiz"
                ]
            },
            "module2": {
                "name": "Asynchronous IO with Callback Programming",
                "topics": [
                    "Module Introduction",
                    "Asynchronous I/O with Callback Programming",
                    "Creating Callback Functions",
                    "Promises",
                    "Working with JSON",
                    "Lab",
                    "Module Summary",
                    "Practice Quiz",
                    "Graded Quiz"
                ]
            },
            "module3": {
                "name": "Express Web Application Framework",
                "topics": [
                    "Module Introduction",
                    "Extending Node.js",
                    "Express Web Application Framework",
                    "Your first Express Web Application",
                    "Routing, Middleware, and Templating",
                    "Lab",
                    "Module Summary",
                    "Practice Quiz",
                    "Graded Quiz"
                ]
            },
            "module4": {
                "name": "Building a Rich Front-End Application using REACT & ES6",
                "topics": [
                    "Module Introduction",
                    "Introduction to ES6",
                    "Introduction to Front End Frameworks and React.JS",
                    "Working with React Components",
                    "Passing Data and States between Components",
                    "Connecting React App to External Services (WIP)",
                    "Lab",
                    "Module Summary",
                    "Practice Quiz",
                    "Graded Quiz"
                ]
            }
        }
    }
}
```

Congratulations! You have completed the lab for the second module of this course.

Summary

Now that you have have learned how to use Async Callback programming we will go further and extend the capabilities of our server side.

Author(s)

Lavanya