

Hands-on Lab - Async Callback (30 mins)



Objective for Exercise:

After completing this lab, you will be able to:

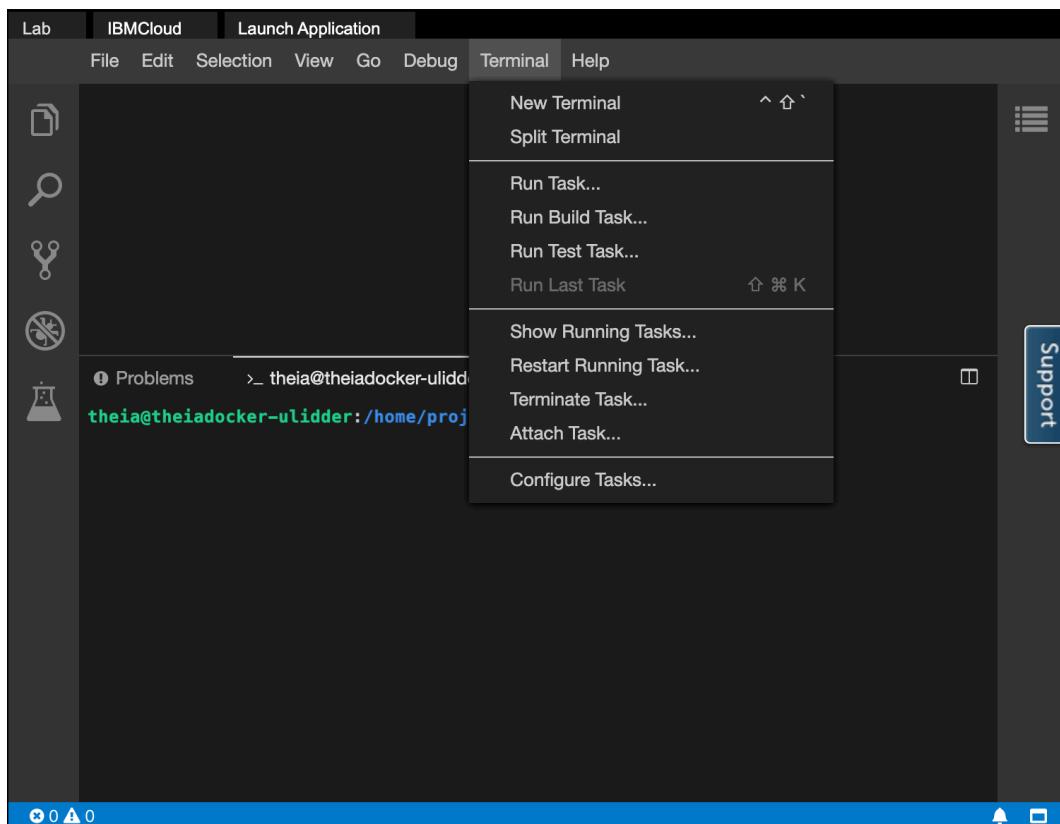
- Describe asynchronous callbacks
- Create a Node.js application

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/
```

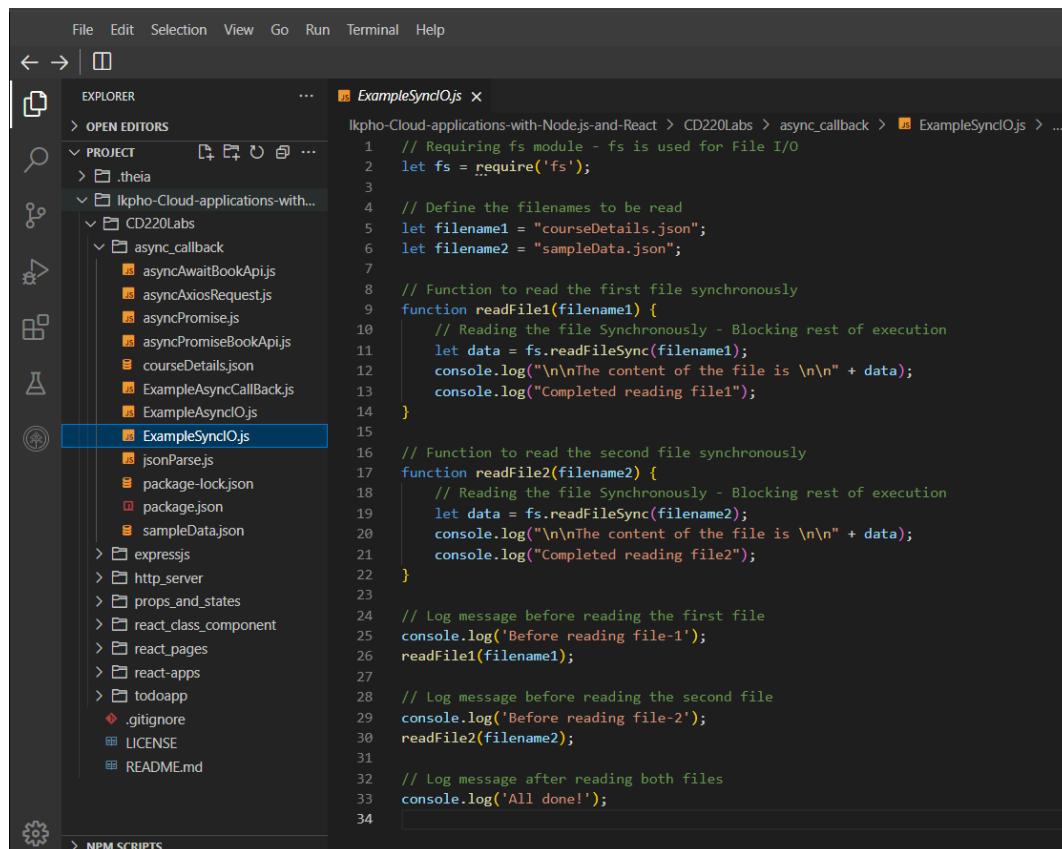
6. List the contents of this directory to see the artifacts for this lab.

```
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.



```
File Edit Selection View Go Run Terminal Help
← → | □
EXPLORER
OPEN EDITORS
PROJECT .theia
lkpho-Cloud-applications-with-Node.js-and-React
  CD220Labs
    async_callback
      asyncAwaitBookApi.js
      asyncAxiosRequest.js
      asyncPromise.js
      asyncPromiseBookApi.js
      courseDetails.json
      ExampleAsyncCallBack.js
      ExampleAsyncIO.js
      ExampleSyncIO.js
      jsonParse.js
      package-lock.json
      package.json
      sampleData.json
    expressjs
    http_server
    props_and_states
    react_class_component
    react_pages
    react_apps
    todoapp
    .gitignore
    LICENSE
    README.md
NPM SCRIPTS

ExampleSyncIO.js ×
lkpho-Cloud-applications-with-Node.js-and-React > CD220Labs > async_callback > ExampleSyncIO.js > ...
1 // Requiring fs module - fs is used for File I/O
2 let fs = require('fs');
3
4 // Define the filenames to be read
5 let filename1 = "courseDetails.json";
6 let filename2 = "sampleData.json";
7
8 // Function to read the first file synchronously
9 function readFile1(filename1) {
10   // Reading the file Synchronously - Blocking rest of execution
11   let data = fs.readFileSync(filename1);
12   console.log("\n\nThe content of the file is \n\n" + data);
13   console.log("Completed reading file1");
14 }
15
16 // Function to read the second file synchronously
17 function readFile2(filename2) {
18   // Reading the file Synchronously - Blocking rest of execution
19   let data = fs.readFileSync(filename2);
20   console.log("\n\nThe content of the file is \n\n" + data);
21   console.log("Completed reading file2");
22 }
23
24 // Log message before reading the first file
25 console.log('Before reading file-1');
26 readFile1(filename1);
27
28 // Log message before reading the second file
29 console.log('Before reading file-2');
30 readFile2(filename2);
31
32 // Log message after reading both files
33 console.log('All done!');
```

▼ You can also click here to view the code

```
// Requiring fs module - fs is used for File I/O
let fs = require('fs');
```

```

// Define the filenames to be read
let filename1 = "courseDetails.json";
let filename2 = "sampleData.json";

// Function to read the first file synchronously
function readFile1(filename1) {
    // Reading the file Synchronously - Blocking rest of execution
    let data = fs.readFileSync(filename1);
    console.log("\n\nThe content of the file is \n\n" + data);
    console.log("Completed reading file1");
}

// Function to read the second file synchronously
function readFile2(filename2) {
    // Reading the file Synchronously - Blocking rest of execution
    let data = fs.readFileSync(filename2);
    console.log("\n\nThe content of the file is \n\n" + data);
    console.log("Completed reading file2");
}

// Log message before reading the first file
console.log('Before reading file-1');
readFile1(filename1);

// Log message before reading the second file
console.log('Before reading file-2');
readFile2(filename2);

// Log message after reading both files
console.log('All done!');

```

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.

```

ExampleAsyncIO.js
1 // Requiring fs module - fs is used for File I/O
2 let fs = require('fs');
3
4 let filename1 = "courseDetails.json";
5 let filename2 = "sampleData.json";
6
7 // Reading the file Asynchronously - Not blocking rest of execution
8 function readFile1(filename1) {
9     // Using fs.readFile to read the file asynchronously
10    fs.readFile(filename1, (err, data) => {
11        if (err) {
12            // Logging the error if any occurs
13            console.log(err);
14        } else {
15            // Logging the content of the file if read successfully
16            console.log("\n\nThe content of the file is \n\n" + data);
17            console.log("Completed reading file1");
18        }
19    });
20}
21
22 function readFile2(filename2) {
23     // Using fs.readFile to read the file asynchronously
24    fs.readFile(filename2, (err, data) => {
25        if (err) {
26            // Logging the error if any occurs
27            console.log(err);
28        } else {
29            // Logging the content of the file if read successfully
30            console.log("\n\nThe content of the file is \n\n" + data);
31            console.log("Completed reading file2");
32        }
33    });
34}

```

▼ You can also click here to view the code

```

// Requiring fs module - fs is used for File I/O
let fs = require('fs');
let filename1 = "courseDetails.json";
let filename2 = "sampleData.json";
// Reading the file Asynchronously - Not blocking rest of execution
function readFile1(filename1) {
    // Using fs.readFile to read the file asynchronously
    fs.readFile(filename1, (err, data) => {
        if (err) {
            // Logging the error if any occurs
            console.log(err);
        } else {
            // Logging the content of the file if read successfully
            console.log("\n\nThe content of the file is \n\n" + data);
            console.log("Completed reading file1");
        }
    });
}
function readFile2(filename2) {
    // Using fs.readFile to read the file asynchronously
    fs.readFile(filename2, (err, data) => {
        if (err) {
            // Logging the error if any occurs
            console.log(err);
        } else {
            // Logging the content of the file if read successfully
            console.log("\n\nThe content of the file is \n\n" + data);
            console.log("Completed reading file2");
        }
    });
}
// Log message before reading the first file
console.log('Before reading the file-1');
readFile1(filename1);
// Log message before reading the second file
console.log('Before reading the file-2');
readFile2(filename2);
// Log message after initiating the reading of both files
console.log('All done!');

```

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.

The screenshot shows a code editor interface with a dark theme. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, and Help. Below the menu is a toolbar with icons for back, forward, and other navigation. The left sidebar is the Explorer view, showing a project structure under 'PROJECT'. The 'async_callback' folder contains several files: asyncAwaitBookApi.js, asyncAxiosRequest.js, asyncPromise.js, asyncPromiseBookApi.js, courseDetails.json, and ExampleAsyncCallBack.js (which is selected). Other folders and files visible include .theia, lkpho-Cloud-applications-with..., CD220Labs, expressjs, http_server, props_and_states, react_class_component, react_pages, react_apps, todoapp, .gitignore, LICENSE, and README.md. The right pane displays the code for `ExampleAsyncCallBack.js`:

```
// This method will be provided as a parameter
function firstCallBackMethod() {
    // Logging a message inside the callback method
    console.log("Inside the first call back method");
}

// Log message before calling setTimeout
console.log("Going to call setTimeout with a delay of 5 seconds");
// Call the function firstCallBackMethod after a delay using setTimeout
setTimeout(firstCallBackMethod, 5000);
```

▼ You can also click here to view the code

```
// This method will be provided as a parameter
function firstCallBackMethod() {
    // Logging a message inside the callback method
    console.log("Inside the first call back method");
}

// Log message before calling setTimeout
console.log("Going to call setTimeout with a delay of 5 seconds");
// Call the function firstCallBackMethod after a delay using setTimeout
setTimeout(firstCallBackMethod, 5000);
```

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 a parameter. 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.

The screenshot shows the VS Code interface with the following details:

- File Explorer:** On the left, it shows a tree view of the project structure. The root folder is `Ikpho-Cloud-applications-with-Node.js-and-React`, which contains `.theia`, `CD220Labs`, and `async_callback`. Inside `async_callback`, there are files: `asyncAwaitBookApi.js`, `asyncAxiosRequest.js`, and `asyncPromise.js` (which is selected). Other files in `async_callback` include `asyncPromiseBookApi.js`, `courseDetails.json`, `ExampleAsyncCallBack.js`, `ExampleAsyncOjs`, `ExampleSyncOjs`, `jsonParse.js`, `package-lock.json`, `package.json`, and `sampleData.json`. The `CD220Labs` folder also contains `expressjs`, `http_server`, `props_and_states`, `react_class_component`, `react_pages`, `react-apps`, and `todoapp`. Additionally, there are `.gitignore`, `LICENSE`, and `README.md`.
- Code Editor:** The main area displays the content of `asyncPromise.js`. The code uses `prompt-sync` and `fs` modules to handle user input and file reading via promises.
- Terminal:** At the bottom, there is a terminal tab labeled `NPM SCRIPTS`.

▼ You can also click here to view the code

```
// Requiring prompt-sync module to enable synchronous user input
let prompt = require('prompt-sync')();
// Requiring fs module - fs is used for File I/O
let fs = require('fs');
// Creating a new Promise to handle file reading
const methCall = new Promise((resolve, reject) => {
    // Prompting the user to input the filename
    let filename = prompt('What is the name of the file?');
    try {
        // Reading the file synchronously
        const data = fs.readFileSync(filename, { encoding: 'utf8', flag: 'r' });
        // Resolving the promise with the file data if read successfully
        resolve(data);
    } catch (err) {
        // Rejecting the promise if an error occurs
        reject(err);
    }
});
```

```

        // Rejecting the promise if an error occurs
        reject(err);
    }
});

// Logging the promise object
console.log(methCall);

// Handling the resolved and rejected states of the promise
methCall.then(
    // Logging the file data if the promise is resolved
    (data) => console.log(data),
    // Logging an error message if the promise is rejected
    (err) => console.log("Error reading file")
);

```

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.

```

asyncAxiosRequest.js
1 // Requiring axios module for making HTTP requests
2 const axios = require('axios').default;
3
4 // Function to connect to a URL and handle the response
5 const connectToURL = (url) => {
6     // Sending a GET request to the specified URL using axios
7     const req = axios.get(url);
8     // Logging the initial promise object
9     console.log(req);
10    // Handling the promise resolution
11    req.then(resp => {
12        // Logging the fulfillment message
13        console.log("Fulfilled");
14        // Logging the response data
15        console.log(resp.data);
16    })
17    // Handling the promise rejection
18    .catch(err => {
19        // Logging the rejection message with the URL
20        console.log("Rejected for url " + url);
21        // Logging the error message
22        console.log(err.toString());
23    });
24
25    // Valid URL
26    connectToURL('https://raw.githubusercontent.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-Node.js-and-React/master/CD220Labs/as');
27    // Invalid URL
28    connectToURL('https://raw.githubusercontent.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-Node.js-and-React/master/CD220Labs/as');

```

▼ You can also click here to view the code

```

// Requiring axios module for making HTTP requests
const axios = require('axios').default;
// Function to connect to a URL and handle the response
const connectToURL = (url) => {
    // Sending a GET request to the specified URL using axios
    const req = axios.get(url);
    // Logging the initial promise object
    console.log(req);
    // Handling the promise resolution
    req.then(resp => {
        // Logging the fulfillment message
        console.log("Fulfilled");
        // Logging the response data
        console.log(resp.data);
    })
    // Handling the promise rejection
    .catch(err => {
        // Logging the rejection message with the URL
        console.log("Rejected for url " + url);
        // Logging the error message
        console.log(err.toString());
    });
}
// Valid URL
connectToURL('https://raw.githubusercontent.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-Node.js-and-React/master/CD220Labs/as');
// Invalid URL
connectToURL('https://raw.githubusercontent.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-Node.js-and-React/master/CD220Labs/as');

```

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.

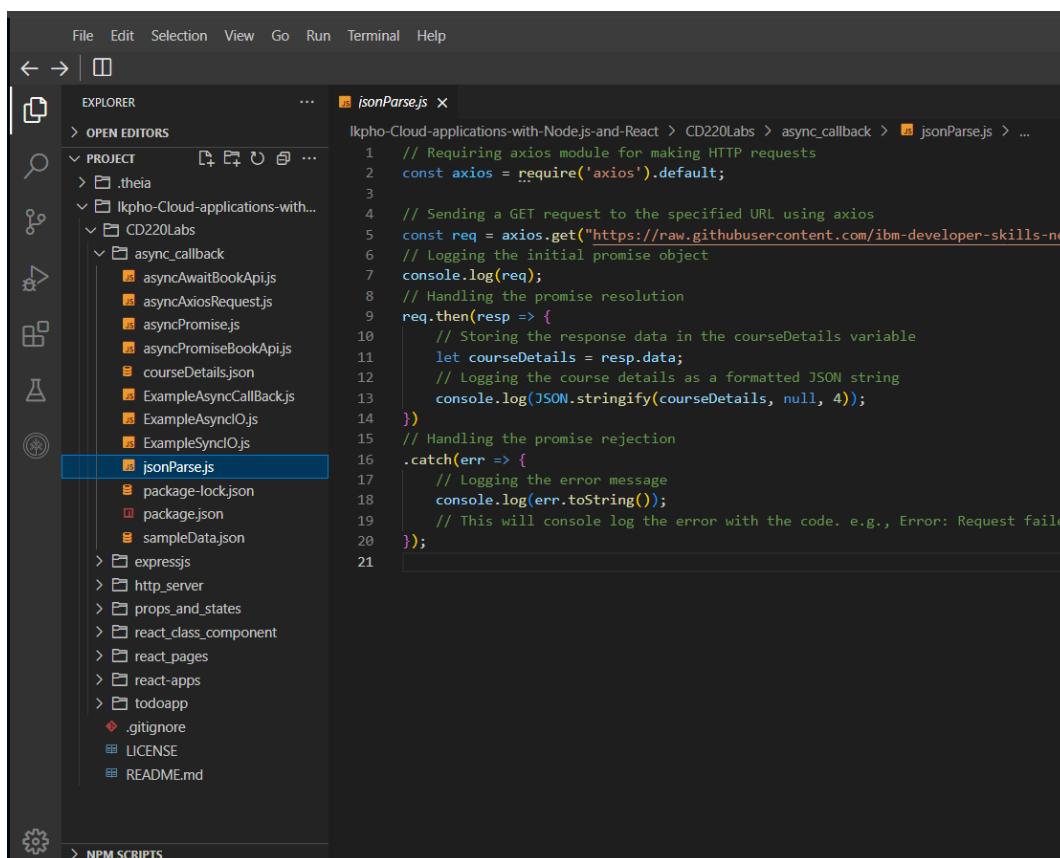
```
node asyncAxiosRequest.js
```

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-Node.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.



The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows the project structure under "PROJECT". The "async_callback" folder is expanded, showing files like "asyncAwaitBookApi.js", "asyncAxiosRequest.js", "asyncPromise.js", "asyncPromiseBookApi.js", "courseDetails.json", "ExampleAsyncCallBack.js", "ExampleAsyncIO.js", and "ExampleSyncIO.js". The file "jsonParse.js" is selected and highlighted in blue.
- Code Editor:** Displays the content of "jsonParse.js". The code uses the axios module to make a GET request to a valid URL, logs the initial promise object, handles the response, and logs the course details as a formatted JSON string. It also includes a catch block for errors.
- Terminal:** Not visible in the screenshot.
- Status Bar:** Shows "NPM SCRIPTS" and other standard status bar icons.

▼ You can also click here to view the code

```
// Requiring axios module for making HTTP requests
const axios = require('axios').default;
// Sending a GET request to the specified URL using axios
const req = axios.get("https://raw.githubusercontent.com/ibm-developer-skills-network/lkpho-Cloud-applications-with-Node.js-and-React/master/CD220Labs/async_callback/sampleData.json")
// Logging the initial promise object
console.log(req);
// Handling the promise resolution
req.then(resp => {
  // Storing the response data in the courseDetails variable
  let courseDetails = resp.data;
  // Logging the course details as a formatted JSON string
  console.log(JSON.stringify(courseDetails, null, 4));
})
// Handling the promise rejection
```

```
.catch(err => {
  // Logging the error message
  console.log(err.toString());
  // This will console log the error with the code. e.g., Error: Request failed with status code 404
});
```

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 readable 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 I/O 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 learned how to use Async Callback programming we will go further and extend the capabilities of our server side.

Author(s)

[Lavanya](#)

© IBM Corporation. All rights reserved.