Date: 15 June 2024 Day: Saturday

Overview:

Day 9 of the internship focused on delving into asynchronous programming concepts in Node.js, essential for handling operations that do not block the execution thread and optimizing application performance through non-blocking I/O operations.

Learning Objectives:

- Asynchronous Programming Concepts:
- Studied the fundamentals of asynchronous programming in Node.js, contrasting it with synchronous execution for handling tasks like file I/O, network requests, and database operations efficiently.
- Explored the event-driven architecture of Node.js, where asynchronous operations leverage event loops to manage I/O operations asynchronously.

Understanding Callbacks:

- Investigated callbacks as a foundational asynchronous pattern in Node.js:
- Callback Functions: Defined callback functions as functions passed as arguments to be executed once an asynchronous operation completes.
- Error-First Callbacks: Implemented error-first callbacks, where the first argument of the callback indicates an error if present, promoting robust error handling in asynchronous code.

```
AsynchronousProgramming > JS index.js > ...
1   const _ = require('lodash')
2   const moment =  require('moment')
3   const numbers = [3,2,5,9,7,6,1,8]
4
5   const sortnumbers = _.sortBy(numbers);
6   const now = moment().format('YYYY-MM-DD')
7   console.log('Date :',now);
8   console.log(sortnumbers)
9   //callback
10  function fetchData(url,callback){
11      // simulation asynchronous request
12      setTimeout(()=>{
13          const data = {id:1,name:'KG'}
14          callback(data)
15      },2000)    // waiting for two seconds
16  }
17  fetchData('https://example/api',(data)=>{
18      console.log(data)
19  })
20  // Promises
21  function fetchData1(url){
22      return new Promise((resolve,reject) => {
23          setTimeout(()=>{
24              const data = {id:1,name:'KG'}
25          //  if(true) reject('error');  // it prints error
26              resolve(data); // if error use reject message
27          },2000)
28      });
29
30  }
```

```
31  fetchData1('https://example/api').then((data)=>{
32      console.log(data)
33  }).catch((err)=>console.log(err));
34
```

Working with Promises:

- Discussed the advantages of promises for enhancing readability and manageability of asynchronous code:

- Promise Chaining: Implemented promise chaining using .then() and .catch() to sequence asynchronous operations and handle errors more gracefully.
- Error Handling with Promises: Explored techniques for error propagation and handling using promises, ensuring consistent error management across asynchronous workflows.
- Promise APIs: Utilized built-in promise APIs such as Promise.all() for parallel execution of asynchronous tasks and Promise.race() for racing asynchronous operations.