**IP Lab**

**CA Assignment**

Name: Shashwat Tripathi

Class: D15A

Roll no: 64

**1. Sample program to demonstrate the use of Promise and Callback functions in asynchronous programming in JavaScript.**

// Function to simulate an asynchronous task

function performAsyncTask(data, callback) {

setTimeout(() => {

if (data) {

callback(null, `Task completed with data: ${data}`);

} else {

callback("Error: Data is missing", null);

}

}, 2000);

}

// Using Callbacks

function usingCallbacks() {

console.log("Using Callbacks:");

performAsyncTask("Sample Data", (err, result) => {

if (err) {

console.error(err);

} else {

console.log(result);

}

});

}

usingCallbacks();

// Using Promises

function usingPromises() {

console.log("\nUsing Promises:");

const promiseTask = new Promise((resolve, reject) => {

performAsyncTask("Promise Data", (err, result) => {

if (err) {

reject(err);

} else {

resolve(result);

}

});

});

promiseTask

.then((result) => {

console.log(result);

})

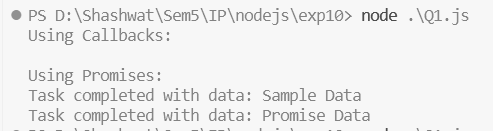
.catch((error) => {

console.error(error);

});

}

usingPromises();



**2. Sample Program to demonstrate the use of generator Iterator**

// Define a generator function

function\* numberGenerator(start, end, step) {

for (let i = start; i <= end; i += step) {

yield i;

}

}

// Create an iterator using the generator

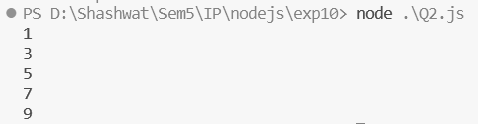
const iterator = numberGenerator(1, 10, 2);

// Iterate through the generator using a for...of loop

for (const number of iterator) {

console.log(number);

}



**3. Sample Program on Routing using REACT, NODE.js, Express.js**

**React**

**App.js**

import "./App.css";

import Home from "./components/Home.jsx";

import About from "./components/About.jsx";

import Contact from "./components/Contact.jsx";

import { Routes, Route, BrowserRouter } from "react-router-dom";

function App() {

return (

<>

<BrowserRouter>

<Routes>

<Route path="/" element={<Home />} />

<Route path="/about" element={<About />} />

<Route path="/contact" element={<Contact />} />

</Routes>

</BrowserRouter>

</>

);

}

export default App;

**/components/Home.js**

import React from 'react'

const Home = () => {

return (

<div>

Home component

</div>

)

}

export default Home

**/components/About.js**

import React from 'react'

const About = () => {

return (

<div>

About component

</div>

)

}

export default About

**/components/Contact.js**

import React from 'react'

const Contact = () => {

return (

<div>

Contact component

</div>

)

}

export default Contact

**Express**

const express = require('express');

const app = express();

const port = 3000;

app.listen(port, () => console.log(`Server challa hai at port ${port}!`));

app.get('/', (req, res) => {

res.send('Home');

});

app.get('/about', (req, res) => {

res.send('About');

})

app.get('/contact', (req, res) => {

res.send('Contact');

})

**Node**

const http = require('http');

// Create a server object

http.createServer(function (req, res) {

// http header

res.writeHead(200, {'Content-Type': 'text/html'});

const url = req.url;

if(url ==='/about') {

res.write(' Welcome to about us page');

res.end();

}

else if(url ==='/contact') {

res.write(' Welcome to contact us page');

res.end();

}

else {

res.write('Hello World!');

res.end();

}

}).listen(3000, function() {

// The server object listens on port 3000

console.log("server start at port 3000");

});

**4. Explain Where REST APIs are used explain the working of REST API with example.**

REST (Representational State Transfer) APIs are a set of architectural constraints and principles for designing networked applications. They are used to build web services that are simple, scalable, and stateless. REST is an architectural style that makes use of the HTTP protocol and adheres to a set of constraints for designing networked application**s.**

**Key principles of REST:**

1. Stateless: Each request from a client to a server must contain all the information needed to understand and fulfill that request. The server should not store any information about the client's state between requests.

2. Client-Server: The client and server should be separate, independent components that interact with each other. This separation allows for better scalability and flexibility.

3. Uniform Interface: A uniform and consistent way of interacting with resources (usually via HTTP methods like GET, POST, PUT, DELETE) makes the system easy to understand and use.

4. Resource-Based: In a REST API, everything is treated as a resource, which can be any object or data. Resources are identified by URIs, and each resource can have multiple representations (e.g., JSON, XML, HTML).

5. Stateless Communication: Each request to the server must be self-contained, meaning the server doesn't rely on the client's previous requests. Each request should contain all the information required.

6. Layered System: The architecture can be composed of multiple layers, with each layer providing specific functionality. This enables load balancing, caching, and more.

**Where REST APIs are used:**

REST APIs are commonly used in web development and distributed systems to provide a standard way to access resources and services. They are often used in scenarios such as:

1. Web Services: Providing a way for different software applications to communicate over the internet.

2. Mobile Applications: Backend APIs for mobile apps to interact with servers.

3. IoT (Internet of Things): Controlling and monitoring IoT devices remotely.

4. Microservices: Building distributed systems using a collection of small, independently deployable services.

**Example:**

const express = require('express');

const app = express();

const books = [

{ id: 1, title: 'Book 1', author: 'Author 1', genre: 'Fiction' },

{ id: 2, title: 'Book 2', author: 'Author 2', genre: 'Non-Fiction' },

];

// Get all books

app.get('/books', (req, res) => {

res.json(books);

});

// Get a specific book by ID

app.get('/books/:id', (req, res) => {

const book = books.find(b => b.id === parseInt(req.params.id));

if (!book) return res.status(404).json({ message: 'Book not found' });

res.json(book);

});

// Add a new book

app.post('/books', (req, res) => {

const newBook = req.body;

books.push(newBook);

res.status(201).json(newBook);

});

// Update a book by ID

app.put('/books/:id', (req, res) => {

const book = books.find(b => b.id === parseInt(req.params.id));

if (!book) return res.status(404).json({ message: 'Book not found' });

// Update book properties

book.title = req.body.title;

book.author = req.body.author;

book.genre = req.body.genre;

res.json(book);

});

// Delete a book by ID

app.delete('/books/:id', (req, res) => {

const index = books.findIndex(b => b.id === parseInt(req.params.id));

if (index === -1) return res.status(404).json({ message: 'Book not found' });

books.splice(index, 1);

res.json({ message: 'Book deleted' });

});

app.listen(3000, () => {

console.log('Server is running on port 3000');

});

In this example, we have created a simple REST API for managing books. You can send HTTP requests to the specified endpoints to interact with the bookstore data. For instance:

GET /books retrieves a list of all books.

GET /books/1 retrieves the book with ID 1.

POST /books adds a new book.

PUT /books/1 updates the book with ID 1.

DELETE /books/1 deletes the book with ID 1.

**5. Explain the use of WebPack in React.**

Webpack is a powerful and widely-used build tool in the React ecosystem. It is primarily used for bundling and optimizing JavaScript, CSS, and other assets in your React applications. Webpack simplifies the process of managing dependencies, code splitting, and optimizing assets for production. Here's an explanation of the key uses of Webpack in React development:

1. Module Bundling: Webpack's primary purpose is to bundle various modules and dependencies into a single or multiple JavaScript files. In React applications, you typically write modular code using ES6 modules (import/export statements). Webpack analyzes your application's dependencies and creates a bundle that includes all required modules. This reduces the number of HTTP requests and improves performance.

2. Loaders: Webpack allows you to use loaders to preprocess files other than JavaScript, such as CSS, images, and fonts. In React, you often use CSS preprocessors like SCSS or LESS. Loaders can compile these files into a format that browsers can understand.

3. Code Splitting: React applications can become large, making it essential to split your code into smaller, more manageable chunks. Webpack supports code splitting, which means you can create separate bundles for different parts of your application. This can significantly reduce the initial load time and improve user experience.

4. Asset Optimization: Webpack provides plugins for optimizing assets. For instance, you can use the "Terser" plugin for JavaScript minification, the "OptimizeCSSAssetsPlugin" for CSS optimization, and the "image-webpack-loader" to optimize images.

5. Development Server: Webpack comes with a built-in development server that provides live reloading and other development features. This simplifies the development workflow, as you can instantly see changes without manually refreshing the browser.

6. Environment Configuration: Webpack allows you to set up different configurations for development and production environments. This is helpful for managing various build options, such as enabling debugging tools in development and minimizing code in production.

7. Integration with Babel: React applications commonly use Babel for transpiling JSX and ES6/ES7 code into plain JavaScript that browsers can understand. Webpack can be configured to work seamlessly with Babel, ensuring that your React code is transpiled correctly.

8. Hot Module Replacement (HMR): HMR is a feature provided by Webpack that enables developers to see changes in the application without the need for a full page refresh. This speeds up development and allows for a more interactive development experience.