Slip1

Write an AngularJS script for addition of two numbers using ng-init, ng-model &

ng-bind. And also demonstrate ng-show, ng-disabled, ng-click directives on button

component.

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>AngularJS Addition Example</title>

<script src="angular.min.js"></script>

</head>

<body ng-app="myApp" ng-controller="myController">

<!-- Input fields for two numbers with ng-model -->

<div ng-init="num1=0; num2=0; result=0">

<label>Number 1:</label>

<input type="number" ng-model="num1" placeholder="Enter first number" />
```

```
<br>

<label>Number 2:</label>

<input type="number" ng-model="num2" placeholder="Enter second number" />

</div>

<!-- Button for calculating the result -->

<button

ng-click="addNumbers()"

ng-disabled="num1 === null || num2 === null"

ng-show="num1 !== null && num2 !== null">

Add Numbers

</button>

<!-- Display the result with ng-bind -->

<div>

<h3>Result: <span ng-bind="result"></span></h3>

</div>

<!-- Show message if either of the inputs is empty -->
```

```
<p ng-show="num1 === null || num2 === null" style="color: red;">

Please enter both numbers to enable addition.

</p>

<script>

// Define the AngularJS application

angular.module('myApp', [])

.controller('myController', function($scope) {

// Function to add numbers

$scope.addNumbers = function() {

$scope.result = $scope.num1 + $scope.num2;
};

});

</script>

</body>

</html>
```

Q.2) Create a Node.js application that reads data from multiple files asynchronously

using promises and async/await.

To create a Node.js application that reads data from multiple files asynchronously using Promises and async/await, you can follow the steps outlined below. This will demonstrate how to use fs.promises API to read files asynchronously.

Step 1: Install Node.js (if not installed already)

Make sure you have Node.js installed on your system. You can check if it's installed by running the following command in the terminal:

```
node -v
```

If it's not installed, download and install Node.js from the official website.

Step 2: Create the Project Directory

Create a new directory for your project:

```
mkdir file-reader
cd file-reader
```

Step 3: Initialize a Node.js Project

Run the following command to initialize a new Node.js project:

```
npm init -y
```

This creates a package.json file with default values.

Step 4: Create Sample Files

Create a few sample text files in the project directory. For example:

file1.txt

file2.txt

file3.txt

You can manually create these files or use the following commands to create them with some content:

```
echo "This is file 1" > file1.txt
echo "This is file 2" > file2.txt
echo "This is file 3" > file3.txt
```

Step 5: Write the Node.js Application

Create a file named index.js in the project directory with the following content:

```
const fs = require('fs').promises; // Using fs.promises API for async file operations
```

```
// Function to read a file
const readFile = async (filePath) => {
  try {
    const data = await fs.readFile(filePath, 'utf-8');
    return data;
  } catch (err) {
    console.error(`Error reading ${filePath}:`, err);
    throw err; // Re-throw the error
  }
};

// Main function to read multiple files asynchronously
const readFiles = async () => {
  try {
    // Use Promise.all to read all files concurrently
    const file1Data = readFile('file1.txt');
    const file2Data = readFile('file2.txt');
    const file3Data = readFile('file3.txt');

    const [file1, file2, file3] = await Promise.all([file1Data, file2Data, file3Data]);

    // Print the contents of each file
    console.log('File 1 content:', file1);
    console.log('File 2 content:', file2);
    console.log('File 3 content:', file3);
```

```
  } catch (err) {
    console.error('Error reading files:', err);
  }
};

// Run the program
readFiles();
```

Step 6: Run the Application

Run the Node.js application using the following command:

```
node index.js
```

Explanation:

1. fs.promises.readFile(): The fs.promises API provides promises for file system operations. It allows us to read files asynchronously using await in an async function.

2. async/await: In the readFile function, we use await to read each file. The await keyword pauses the execution until the promise is resolved.

3. Promise.all(): We use Promise.all() to read all the files concurrently. This helps in parallelizing the file reading, making it more efficient than reading files sequentially.

4. Error Handling: Errors are caught using try/catch blocks in both the readFile and readFiles functions.

Step 7: Output

When you run the app, the contents of the files should be printed in the terminal like this:

File 1 content: This is file 1

File 2 content: This is file 2

File 3 content: This is file 3

Conclusion:

This Node.js application demonstrates how to read multiple files asynchronously using Promises and async/await. Using Promise.all ensures that all files are read concurrently, making the application more efficient. You can expand this to read a larger number of files or use more complex file manipulation tasks as needed.

Slip 2

Slip:2

Write an AngularJS script to print details of bank (bank name, MICR code, IFC

code, address etc.) in tabular form using ng-repeat.

```
<!DOCTYPE html>

<html lang="en" ng-app="bankApp">

<head>

 <meta charset="UTF-8">

 <title>Bank Details</title>

 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

</head>

<body ng-controller="BankController">

 <h1>Bank Details</h1>

 <table border="1">

 <tr>

 <th>Bank Name</th>

 <th>MICR Code</th>

 <th>IFSC Code</th>
```

```
<th>Address</th>

</tr>

<tr ng-repeat="bank in banks">

<td>{{ bank.name }}</td>

<td>{{ bank.micr }}</td>

<td>{{ bank.ifsc }}</td>

<td>{{ bank.address }}</td>

</tr>

</table>

<script>

angular.module('bankApp', [])

.controller('BankContr
{ name: 'SBI', micr: '123456789', ifsc: 'BOA0001234', address: '123 Main St, New York, NY' },

{ name: 'TJSB', micr: '987654321', ifsc: 'CHAS0009876', address: '456 Elm St, Los Angeles, CA' },

{ name: 'HDFC', micr: '456123789', ifsc: 'WF0004567', address: '789 Pine St, Chicago, IL' }

];
```

```
});

</script>

</body>

</html>
```

Q.2) Create a simple Angular application that fetches data from an API using HttpClient.

Implement an Observable to fetch data from an API endpoint.

Here's a basic Angular application that fetches data from an API using HttpClient and implements an Observable to handle data fetching.

Step 1: Set Up Angular Project

First, create an Angular project and navigate into the project directory:

```
ng new angular-http-observable

cd angular-http-observable
```

Then, generate a new service to handle API calls and a component to display the data.

```
ng generate service data

ng generate component data-display
```

Step 2: Install HttpClient Module

Install the HttpClientModule in your app.module.ts file. This module is essential for making HTTP requests in Angular.

```
// src/app/app.module.ts

import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';

import { HttpClientModule } from '@angular/common/http'; // Import HttpClientModule


import { AppComponent } from './app.component';

import { DataDisplayComponent } from './data-display/data-display.component';


@NgModule({
  declarations: [
    AppComponent,
    DataDisplayComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule  // Add HttpClientModule here
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 3: Create the Data Service

Now, configure the DataService to fetch data from an API using an Observable. For this example, we'll use a placeholder API like https://jsonplaceholder.typicode.com/posts.

```
// src/app/data.service.ts

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';


@Injectable({

  providedIn: 'root'

})

export class DataService {


  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';


  constructor(private http: HttpClient) { }


  // Fetch data as an Observable

  getData(): Observable<any> {

    return this.http.get<any>(this.apiUrl);

  }

}
```

Step 4: Display Data in the Component

In the DataDisplayComponent, inject the DataService to fetch and display data. Subscribe to the Observable to receive data and handle any error.

```
// src/app/data-display/data-display.component.ts

import { Component, OnInit } from '@angular/core';

import { DataService } from '../data.service';


@Component({

 selector: 'app-data-display',

 templateUrl: './data-display.component.html',

 styleUrls: ['./data-display.component.css']

})

export class DataDisplayComponent implements OnInit {


 posts: any[] = [];


 constructor(private dataService: DataService) {}


 ngOnInit(): void {

  this.dataService.getData().subscribe(

   (data) => {

    this.posts = data;  // Assign fetched data to posts

   },

   (error) => {

    console.error("Error fetching data:", error); // Handle error
```

```
    }
   );
 }
}
```

Step 5: Display Data in the Template

Edit the data-display.component.html file to display the data.

```
<!-- src/app/data-display/data-display.component.html -->
<div *ngIf="posts.length">
  <h2>Posts</h2>
  <ul>
    <li *ngFor="let post of posts">
      <h3>{{ post.title }}</h3>
      <p>{{ post.body }}</p>
    </li>
  </ul>
</div>
<div *ngIf="!posts.length">
  Loading...
</div>
```

Step 6: Use the Component in App Component Template

Lastly, add the DataDisplayComponent selector in the main app.component.html to display it.

<!-- src/app/app.component.html -->

<app-data-display></app-data-display>

Final Output

Run the application:

ng serve

This will display data fetched from the API on the main page. You now have a simple Angular application that fetches data from an API using HttpClient and Observable.

Slip 3

Slip-3

Write an AngularJS script to display list of games stored in an array on click of

button using ng-click and also demonstrate ng-init, ng-bind directive of AngularJS.

<!DOCTYPE html>

```
<html lang="en" ng-app="gameApp">

<head>

 <meta charset="UTF-8">

 <meta name="viewport" content="width=device-width, initial-scale=1.0">

 <title>Game List with AngularJS</title>

 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

</head>

<body ng-controller="GameController" ng-init="initializeGames()">

 <h1>Game List</h1>

 <!-- ng-bind directive to display the title -->

 <p ng-bind="gameListTitle"></p>

 <!-- ng-click directive to load and display the list of games -->

 <button ng-click="showGames()">Show Games</button>

 <!-- ng-repeat to display the list of games -->

 <ul>

 <li ng-repeat="game in games">{{ game }}</li>
```

```
</ul>

<script>

// Define the AngularJS module

var app = angular.module('gameApp', []);

// Define the GameController

app.controller('GameController', function($scope) {

// ng-init to initialize data (used here to set the title)

$scope.initializeGames = function() {

$scope.gameListTitle = "Click the button to see the list of games:";

$scope.games = []; // Empty list to start with

};

// ng-click function to show the games

$scope.showGames = function() {

// Example array of games

$scope.games = [

"Minecraft",
```

```
"The Witcher 3",

"Cyberpunk 2077",

"Among Us",

"Fortnite"

];

};

});

</script>

</body>

</html>
```

Q2

Find a company with a workforce greater than 30 in the array (use find by

id method)

```
<!DOCTYPE html>

<html lang="en">

<head>
```

```
 <meta charset="UTF-8">

 <meta name="viewport" content="width=device-width, initial-scale=1.0">

 <title>Find Company by Workforce</title>

</head>

<body>

 <h1>Find Company with Workforce Greater Than 30</h1>



 <button onclick="findCompany()">Find Company</button>

 <p id="company-info"></p>

 <script>

 // Array of company objects

 const companies = [

 { id: 1, name: "Company A", workforce: 25 },

 { id: 2, name: "Company B", workforce: 26},

 { id: 3, name: "Company C", workforce: 10 },

 { id: 4, name: "Company D", workforce: 35 },
```

```
{ id: 5, name: "Company E", workforce: 15 }

];

// Function to find a company with workforce greater than 30

function findCompany() {

// Use the find() method to search by id and workforce condition

const company = companies.find(c => c.workforce > 30);

// Display the result in the paragraph with id 'company-info'

if (company) {

document.getElementById("company-info").textContent =

`Found company: ${company.name} with a workforce of ${company.workforce}`;

} else {

document.getElementById("company-info").textContent = "No company found with a

workforce greater than 30.";

}

}

</script>
```

```
</body>

</html>
```

Slip 4

Fetch the details using ng-repeat in AngularJS.

```
<!DOCTYPE html>

<html lang="en" ng-app="studentApp">

<head>

 <meta charset="UTF-8">

 <title>Student Details</title>

 <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>

</head>

<body ng-controller="StudentController">

 <h1>Student Details</h1>
```

```
<table border="1">

<tr>

<th>Name</th>

<th>Roll Number</th>
<th>Class</th>

<th>Grade</th>

</tr>

<tr ng-repeat="student in students">

<td>{{ student.name }}</td>

<td>{{ student.rollNumber }}</td>

<td>{{ student.class }}</td>

<td>{{ student.grade }}</td>

</tr>

</table>

<script>

angular.module('studentApp', [])
```

```
.controller('StudentController', function($scope) {

// Defining student data directly in the controller

$scope.students = [

{

name: 'Ram Kale',

rollNumber: '101',

class: '10th Grade',

grade: 'A'

},

{

name: 'Tushar Maske',

rollNumber: '102',

class: '10th Grade',

grade: 'B+'

},

{
```

```
name: 'Rohit mane',

rollNumber: '103',

class: '10th Grade',

grade: 'A-'

},

{

name: 'Rahul kale',

rollNumber: '104',

class: '10th Grade',

grade: 'B'

}

];

});

</script>

</body>

</html>
```

Q.2) Express.js application to include middleware for parsing request bodies (e.g.,

JSON, form data) and validating input data.

To set up an Express.js application with middleware for parsing request bodies (like JSON and form data) and validating input data, follow these steps:

Step 1: Set up the Express Application

First, ensure that you have Node.js and Express installed. If not, install Express in your project folder:

```
npm init -y
npm install express
```

Step 2: Add Middleware for Parsing Request Bodies

Express provides built-in middleware for parsing JSON and URL-encoded form data.

```
// app.js
const express = require('express');
const app = express();

// Middleware for parsing JSON
app.use(express.json());

// Middleware for parsing URL-encoded form data
```

```
app.use(express.urlencoded({ extended: true }));
```

Step 3: Add Middleware for Input Validation

To validate input data, you can use a validation library like express-validator. Install it by running:

```
npm install express-validator
```

Then, use it to validate the data in your request. Here’s an example where we validate name and email fields in a POST request.

```
// app.js
const { body, validationResult } = require('express-validator');

app.post(
  '/submit',
  [
    // Validation checks
    body('name').isLength({ min: 3 }).withMessage('Name must be at least 3 characters long'),
    body('email').isEmail().withMessage('Invalid email format'),
  ],
  (req, res) => {
    // Check for validation errors
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() });
```

```
  }

  // Process the request if data is valid
  res.send('Data is valid and processed successfully!');
 }
);

app.listen(3000, () => console.log('Server running on port 3000'));
```

Explanation

Body Parsing Middleware: app.use(express.json()) parses incoming JSON requests, and app.use(express.urlencoded({ extended: true })) parses URL-encoded form data.

Validation Middleware: We use express-validator to define validation checks (body('name') and body('email')) and display errors if any validations fail.

Running the Application

Start the server:

```
node app.js
```

The application will listen on port 3000, and you can test your validation by sending requests to http://localhost:3000/submit. This setup ensures that only valid input data is processed.

Slip 5

Q.1) Create a simple Angular component that takes input data and displays it.

Slip5a)

Create a simple Angular component that takes input data and displays it.

ng new slip5a

cd slip5a

Step 1: Generate the Student Component

ng generate component student

step2:

```
// src/app/student/student.component.ts

import { Component, Input } from '@angular/core';

@Component({

 selector: 'app-student',

 templateUrl: './student.component.html',

 styleUrls: ['./student.component.css']

})
```

```
export class StudentComponent {

 @Input() student: { name: string; age: number; grade: string } | undefined;

}
```

Step 3: Display the Input Data in the Template

```
<p>student works!</p>
```

Slip5B)

Implement a simple server using Node.js.

```
// server.js

const http = require('http');

// Define the hostname and port

const hostname = 'localhost';

const port = 3000;

// Create the server
const server = http.createServer((req, res) => {

// Set the response HTTP header with status and content type

res.statusCode = 200;
```

```
res.setHeader('Content-Type', 'text/plain');

// Handle different routes

if (req.url === '/') {

res.end('Welcome to the Home Page!');

} else if (req.url === '/about') {

res.end('This is the About Page.');

} else if (req.url === '/contact') {

res.end('Contact us at: contact@example.com');

} else {

res.statusCode = 404;

res.end('404 - Page Not Found');

}

});

// Start the server

server.listen(port, hostname, () => {

console.log('server contected.....');
```

```
});
```

Slip 6

Q.1) Develop an Express.js application that defines routes for Create and Read operations

on a resource (products).

```
const express = require('express');
const app = express();

app.use(express.json());

let products = [];

app.post('/products', (req, res) => {
   const product = { id: products.length + 1, ...req.body };
   products.push(product);
   res.status(201).json({ message: 'Product created', product });
});

app.get('/products', (req, res) => {
   res.json(products);
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Q2)

Q.2) Find a company with a workforce greater than 30 in the array. (Using find by

id method)

```
const companies = [
   { id: 1, name: 'Company A', workforce: 25 },
   { id: 2, name: 'Company B', workforce: 40 }
];


const largeCompany = companies.find(company => company.workforce > 30);
console.log("Company with workforce > 30:", largeCompany);
```

Slip 7

Q.1) Create a Node.js application that reads data from multiple files asynchronously

using promises and async/await.

To create a Node.js application that reads data from multiple files asynchronously using promises and async/await, we can use the fs.promises API, which allows us to work with file reading operations using promises. Here's a step-by-step guide to set up the application:

Step 1: Set Up the Project

1. Create a new folder for your project, e.g., async-file-reader.

2. Open a terminal, navigate to the folder, and run:

```
npm init -y
```

This will create a package.json file.

3. Create the files you want to read from, e.g., file1.txt, file2.txt, and file3.txt. Add some sample content in each file.

Step 2: Write the Asynchronous File Reading Code

1. Inside the project folder, create a new JavaScript file, e.g., readFiles.js.

2. Import the fs.promises module and use async/await to read multiple files concurrently.

Here's the complete code for readFiles.js:

```
const fs = require('fs').promises;

// Function to read files asynchronously using async/await
async function readFiles(filePaths) {
  try {
    // Use Promise.all to read all files concurrently
```

```
    const fileContents = await Promise.all(
      filePaths.map((path) => fs.readFile(path, 'utf-8'))
    );

    // Log the content of each file
    fileContents.forEach((content, index) => {
      console.log(`Content of file${index + 1}:`);
      console.log(content);
    });
  } catch (error) {
    console.error('Error reading files:', error);
  }
}

// List of file paths to read
const files = ['file1.txt', 'file2.txt', 'file3.txt'];
readFiles(files);
```

Explanation

1. fs.promises: We use fs.promises instead of the regular fs module to work with promises directly.

2. async function readFiles(filePaths): This function takes an array of file paths.

3. Promise.all: We use Promise.all() to read all files concurrently. Each file read returns a promise, and Promise.all() waits for all promises to resolve.

4. Error Handling: We wrap the code in a try-catch block to handle any potential errors while reading files.

Step 3: Run the Application

In the terminal, run:

node readFiles.js

The application will read and log the content of each file. Each file's content will be printed in the console with a corresponding label.

Q.2) Develop an Express.js application that defines routes for Create and Read operations

on a resource (User).

To develop an Express.js application that defines routes for Create and Read operations on a User resource, follow these steps:

Step 1: Set Up the Project

1. Create a new folder for your project, e.g., express-user-api.

2. Open a terminal, navigate to the folder, and run:

npm init -y

This will create a package.json file.

3. Install express and body-parser:

npm install express body-parser

Step 2: Define the Express Application

1. In the project folder, create a new JavaScript file, e.g., app.js.

2. Set up a basic Express server with routes for creating and reading User data.

Here's the code for app.js:

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const PORT = 3000;

// Use body-parser middleware to parse JSON requests
app.use(bodyParser.json());

// Temporary data storage (in-memory)
const users = [];

// Route to Create a new User
app.post('/users', (req, res) => {
  const { name, email } = req.body;

  if (!name || !email) {
    return res.status(400).json({ error: 'Name and email are required.' });
  }

  const newUser = { id: users.length + 1, name, email };
  users.push(newUser);
```

```
  res.status(201).json({ message: 'User created successfully', user: newUser });
});

// Route to Read all Users
app.get('/users', (req, res) => {
  res.status(200).json(users);
});

// Route to Read a single User by ID
app.get('/users/:id', (req, res) => {
  const userId = parseInt(req.params.id, 10);
  const user = users.find((u) => u.id === userId);

  if (!user) {
    return res.status(404).json({ error: 'User not found' });
  }

  res.status(200).json(user);
});

// Start the server
app.listen(PORT, () => {
  console.log(`Server is running on http://localhost:${PORT}`);
});
```

Explanation

1. In-Memory Data Storage: We use an array users to store user data temporarily. In a real application, you would connect to a database.

2. Create User (POST /users):

We use app.post() to handle POST requests at /users.

The req.body object contains JSON data from the request.

If name or email is missing, we return a 400 status with an error message.

We create a new user, assign it an ID, push it to the users array, and return a 201 status with the user data.

3. Read All Users (GET /users):

We use app.get() to handle GET requests at /users.

The route responds with the entire users array.

4. Read a Single User by ID (GET /users/:id):

We use app.get() with a URL parameter to handle GET requests at /users/:id.

We find the user by ID, and if not found, return a 404 status with an error message.

If the user exists, we return the user data.

Step 3: Run the Application

1. Start the server by running:

node app.js

2. The server will be running on http://localhost:3000. You can test the endpoints using a tool like Postman or curl.

Example Requests

Create a User:

URL: http://localhost:3000/users

Method: POST

Body (JSON):

```
{
  "name": "John Doe",
  "email": "john@example.com"
}
```

Response:

```
{
  "message": "User created successfully",
  "user": {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com"
  }
}
```

Read All Users:

URL: http://localhost:3000/users

Method: GET

Response:


```
[
  {
    "id": 1,
    "name": "John Doe",
    "email": "john@example.com"
  }
]
```


Read a Single User by ID:

URL: http://localhost:3000/users/1

Method: GET

Response:

{
"id": 1,
"name": "John Doe",
"email": "john@example.com"
}

This application provides basic Create and Read operations for the User resource.

Slip 8

Q.1) Create a simple Angular application that fetches data from an API using HttpClient.

Implement an Observable to fetch data from an API endpoint.

To create a simple Angular application that fetches data from an API using HttpClient and implements an Observable, follow these steps:

Step 1: Set Up the Angular Project

1. First, ensure you have Angular CLI installed:

npm install -g @angular/cli

2. Create a new Angular project:

```
ng new angular-http-client-demo
```

3. Navigate to the project directory:

```
cd angular-http-client-demo
```

4. Generate a new component to display the data:

```
ng generate component data-view
```

5. Install HttpClientModule in the Angular app. Open app.module.ts and import HttpClientModule.

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent,
    DataViewComponent
  ],
  imports: [
```

```
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 2: Create a Service to Fetch Data from the API

1. Generate a service:

```
ng generate service data
```

2. In data.service.ts, use the HttpClient to fetch data from an API endpoint. Here's an example using the jsonplaceholder.typicode.com API:

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

// Define an interface for the data (optional but useful)
export interface Post {
```

```
  userId: number;
  id: number;
  title: string;
  body: string;
}

@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) { }

  // Method to fetch data as an Observable
  getPosts(): Observable<Post[]> {
    return this.http.get<Post[]>(this.apiUrl);
  }
}
```

Step 3: Use the Service in a Component

1. Open data-view.component.ts and inject DataService to fetch data.

2. Subscribe to the Observable to receive the data and display it.

```
import { Component, OnInit } from '@angular/core';
import { DataService, Post } from '../data.service';

@Component({
  selector: 'app-data-view',
  templateUrl: './data-view.component.html',
  styleUrls: ['./data-view.component.css']
})
export class DataViewComponent implements OnInit {
  posts: Post[] = [];

  constructor(private dataService: DataService) { }

  ngOnInit(): void {
    // Fetch data and subscribe to the Observable
    this.dataService.getPosts().subscribe({
      next: (data) => this.posts = data,
      error: (error) => console.error('Error fetching data:', error),
    });
  }
}
```

Step 4: Display the Data in the Template

Open data-view.component.html and add a template to display the data fetched from the API:

```
<div *ngIf="posts.length > 0; else loading">
  <h2>Posts</h2>
  <ul>
    <li *ngFor="let post of posts">
      <h3>{{ post.title }}</h3>
      <p>{{ post.body }}</p>
    </li>
  </ul>
</div>

<ng-template #loading>
  <p>Loading data...</p>
</ng-template>
```

Step 5: Update app.component.html to Display the DataView Component

Open app.component.html and add the <app-data-view> selector to display the data:

<app-data-view></app-data-view>

Step 6: Run the Angular Application

In the terminal, start the application by running:

ng serve

Then open a web browser and go to http://localhost:4200 to see the application.

This Angular application fetches data from an API using HttpClient and displays it on the page using an Observable. The DataService fetches the data, and DataViewComponent subscribes to the Observable to retrieve and display it.

Q.2) Develop an Express.js application that defines routes for Create, Update operations

on a resource (Employee).

To develop an Express.js application that defines routes for creating and updating an Employee resource, you can follow the steps below:

Step 1: Set up a basic Express.js application

First, make sure you have Node.js and npm installed. Then, create a new directory for your project and install Express.

1. Initialize a new project and install dependencies:

```
mkdir employee-app

cd employee-app

npm init -y

npm install express body-parser
```

2. Create a server.js file for your application.

Step 2: Create the Express application

Inside server.js, define the routes for creating and updating employees.

```
const express = require('express');

const bodyParser = require('body-parser');


const app = express();

const port = 3000;


// Middleware to parse JSON request bodies

app.use(bodyParser.json());


// In-memory "database" for storing employees (in a real app, you'd use a database)
```

```
let employees = [];

// Create Employee Route
app.post('/employees', (req, res) => {
  const { id, name, position, department } = req.body;

  // Simple validation
  if (!id || !name || !position || !department) {
    return res.status(400).json({ message: 'Missing required fields' });
  }

  const newEmployee = { id, name, position, department };
  employees.push(newEmployee);

  res.status(201).json({ message: 'Employee created', employee: newEmployee });
});

// Update Employee Route
app.put('/employees/:id', (req, res) => {
  const { id } = req.params;
  const { name, position, department } = req.body;

  // Find employee by ID
  const employeeIndex = employees.findIndex(emp => emp.id === id);
```

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

  // Update the employee's information
  employees[employeeIndex] = { id, name, position, department };

  res.status(200).json({ message: 'Employee updated', employee: employees[employeeIndex] });
});

// Start the server
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});
```

Step 3: Run the application

Now, you can run the Express server with the following command:

```
node server.js
```

Step 4: Test the routes

1. Create Employee: Send a POST request to http://localhost:3000/employees with the following JSON body:

```
{
  "id": "1",
  "name": "John Doe",
  "position": "Developer",
  "department": "Engineering"
}
```

2. Update Employee: Send a PUT request to http://localhost:3000/employees/1 with the following JSON body:

```
{
  "name": "John Doe",
  "position": "Senior Developer",
  "department": "Engineering"
}
```

Step 5: Expand as needed

You can further expand this application by adding validation, error handling, and persistent storage (e.g., using MongoDB or a SQL database).

Slip 9

Q.1) Find a company with a workforce greater than 30 in the array. (Using find by

id method).

To find a company with a workforce greater than 30 from an array of companies using the find() method by company ID, you can follow the example below.

Assuming we have an array of company objects with the following structure:

```
const companies = [
  { id: 1, name: 'Tech Solutions', workforce: 25 },
  { id: 2, name: 'Innovative Designs', workforce: 40 },
  { id: 3, name: 'Global Enterprises', workforce: 15 },
  { id: 4, name: 'Creative Minds', workforce: 50 },
];
```

To find a company with a workforce greater than 30, using the find() method by company id, you would do the following:

Example:

```
const companies = [
  { id: 1, name: 'Tech Solutions', workforce: 25 },
  { id: 2, name: 'Innovative Designs', workforce: 40 },
  { id: 3, name: 'Global Enterprises', workforce: 15 },
  { id: 4, name: 'Creative Minds', workforce: 50 },
];
```

```
// Function to find a company with workforce > 30
const findCompanyById = (id) => {
  return companies.find(company => company.id === id && company.workforce > 30);
};

// Example: Find company with ID 2
const company = findCompanyById(2);

if (company) {
  console.log(`Company found: ${company.name} with workforce of ${company.workforce}`);
} else {
  console.log('No company found with a workforce greater than 30.');
}
```

Explanation:

The findCompanyById function uses the find() method to search for a company where the id matches the provided id and the workforce is greater than 30.

If a matching company is found, it will be returned. Otherwise, it will return undefined, and the message "No company found with a workforce greater than 30" will be logged.

Example output:

Company found: Innovative Designs with workforce of 40

You can call findCompanyById with different IDs to search for other companies as needed.

Q.2) Create Express.js application to include middleware for parsing request bodies (e.g.,

JSON, form data) and validating input data. Send appropriate JSON responses for

success and error cases.

To create an Express.js application that includes middleware for parsing request bodies (e.g., JSON, form data) and validates input data, follow these steps:

Step 1: Set up the project

First, create a new directory for your project and install the required dependencies.

```
mkdir express-validation-app

cd express-validation-app

npm init -y

npm install express body-parser
```

Step 2: Create the Express application

Now, create a server.js file and add middleware for parsing request bodies (JSON and URL-encoded form data). You'll also validate the input data and send appropriate JSON responses for success and error cases.

```
const express = require('express');
```

```
const bodyParser = require('body-parser');


const app = express();
const port = 3000;


// Middleware to parse JSON and form data
app.use(bodyParser.json()); // for parsing application/json
app.use(bodyParser.urlencoded({ extended: true })); // for parsing application/x-www-form-urlencoded


// Middleware to validate input data
const validateEmployeeData = (req, res, next) => {
  const { name, position, department } = req.body;


  // Simple validation checks
  if (!name || !position || !department) {
    return res.status(400).json({
      success: false,
      message: 'Missing required fields: name, position, or department',
    });
  }


  next();
};


// Route to create a new employee
```

```
app.post('/employees', validateEmployeeData, (req, res) => {
  const { name, position, department } = req.body;

  // Simulate storing employee data (in reality, you'd save this to a database)
  const newEmployee = { id: Date.now(), name, position, department };

  res.status(201).json({
    success: true,
    message: 'Employee created successfully',
    employee: newEmployee,
  });
});

// Route to update employee information
app.put('/employees/:id', validateEmployeeData, (req, res) => {
  const { id } = req.params;
  const { name, position, department } = req.body;

  // In a real-world scenario, you would fetch and update the employee from a database
  // For simplicity, we simulate this using the provided data

  const updatedEmployee = { id, name, position, department };

  res.status(200).json({
    success: true,
```

```
    message: 'Employee updated successfully',
    employee: updatedEmployee,
  });
});

// Error handling middleware
app.use((err, req, res, next) => {
  console.error(err.stack);
  res.status(500).json({
    success: false,
    message: 'Internal server error',
  });
});

// Start the server
app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`);
});
```

Step 3: How it works

Body Parsing Middleware:

body-parser.json() is used to parse incoming requests with a JSON payload.

body-parser.urlencoded({ extended: true }) is used to parse URL-encoded form data.

Input Validation Middleware (validateEmployeeData):

The validateEmployeeData middleware checks that the required fields (name, position, and department) are provided in the request body.

If any of the fields are missing, it sends a 400 Bad Request response with a detailed error message in JSON format.

Employee Routes:

POST /employees: Creates a new employee (with validation).

PUT /employees/:id: Updates an existing employee (with validation).

Error Handling Middleware:

If there is any error in the application, the error-handling middleware catches it and sends a 500 Internal Server Error response with a generic message.

Step 4: Run the application

Start the server with the following command:

node server.js

Step 5: Testing the application

You can use Postman or cURL to test the routes.

1. Create Employee (POST request):

URL: http://localhost:3000/employees

Body (JSON format):

```
{
  "name": "John Doe",
  "position": "Developer",
  "department": "Engineering"
}
```

Response:

```
{
```

```
 "success": true,
 "message": "Employee created successfully",
 "employee": {
  "id": 1682467854123,
  "name": "John Doe",
  "position": "Developer",
  "department": "Engineering"
 }
}
```


2. Update Employee (PUT request):

URL: http://localhost:3000/employees/1682467854123

Body (JSON format):


```
{
 "name": "John Doe",
 "position": "Senior Developer",
 "department": "Engineering"
}
```


Response:

```
{
  "success": true,
  "message": "Employee updated successfully",
  "employee": {
    "id": "1682467854123",
    "name": "John Doe",
    "position": "Senior Developer",
    "department": "Engineering"
  }
}
```


3. Invalid Create Request (Missing fields):

URL: http://localhost:3000/employees

Body (JSON format):


```
{
  "name": "Jane Doe",
  "position": "Manager"
}
```


Response:

```
{
  "success": false,
  "message": "Missing required fields: name, position, or department"
}
```

Step 6: Expand as needed

You can expand this application by adding more complex validation, error handling, and persistence with a database like MongoDB or SQL.

You can also include additional middleware like authentication, logging, etc.

Slip 10

Q.1) Implement a simple server using Node.js.

To implement a simple server using Node.js, you can use the built-in http module. Here is a basic example of how to do this:

1. Create a file named server.js.

2. Write the following code:

```
// Import the http module
```

```
const http = require('http');

// Define the port for the server
const port = 3000;

// Create the server
const server = http.createServer((req, res) => {
  // Set the response header
  res.writeHead(200, { 'Content-Type': 'text/plain' });

  // Send the response body
  res.end('Hello, world!\n');
});

// Make the server listen on the specified port
server.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Explanation:

http.createServer() is used to create the server, where you pass a callback function to handle requests (req) and responses (res).

res.writeHead(200, { 'Content-Type': 'text/plain' }) sets the status code (200 OK) and the response type (plain text).

res.end() sends the response body to the client.

server.listen(port) makes the server listen for incoming requests on the specified port.

To run the server:

1. Open a terminal and navigate to the folder containing server.js.

2. Run the following command:

node server.js

3. Open your browser and visit http://localhost:3000/. You should see "Hello, world!" displayed in your browser.

Q.2) Extend the previous Express.js application to include middleware for parsing

request bodies (e.g., JSON, form data) and validating input data. Send appropriate JSON

responses for success and error cases.

To extend the previous Node.js application with Express.js to handle request body parsing, validation, and send appropriate JSON responses, follow these steps:

1. Install Express and Middleware

First, you need to install express and body-parser (though Express now has its own built-in middleware for parsing JSON and URL-encoded data).

Run this command to install Express:

npm install express

2. Create the Express Application

Update your server.js to use Express, and add middleware to parse request bodies (for JSON and form data) and validate inputs.

Here’s an extended example with these features:

```
// Import the express module
const express = require('express');
const app = express();

// Use built-in middleware for parsing JSON and URL-encoded form data
app.use(express.json()); // Parses JSON body
app.use(express.urlencoded({ extended: true })); // Parses URL-encoded form data
```

```
// Middleware to validate input
function validateInput(req, res, next) {
  const { name, email } = req.body;
  if (!name || !email) {
    return res.status(400).json({
      success: false,
      message: 'Name and email are required.'
    });
  }

  // Simple email validation
  const emailRegex = /^[\w-]+(\.[\w-]+)*@([\w-]+\.)+[a-zA-Z]{2,7}$/;
  if (!emailRegex.test(email)) {
    return res.status(400).json({
      success: false,
      message: 'Invalid email format.'
    });
  }

  next(); // Proceed to the next middleware or route handler
}

// Define a route that uses the validateInput middleware
app.post('/submit', validateInput, (req, res) => {
```

```
  const { name, email } = req.body;

  // If input is valid, send success response
  res.status(200).json({
    success: true,
    message: 'Data successfully received!',
    data: { name, email }
  });
});

// Handle unknown routes (404 error)
app.use((req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found.'
  });
});

// Define the port for the server
const port = 3000;

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Key Parts:

1. Request Body Parsing:

express.json() middleware parses JSON formatted data in the request body.

express.urlencoded({ extended: true }) middleware parses URL-encoded data (like form submissions).

2. Input Validation:

validateInput is a custom middleware that checks if the name and email fields are present in the request body. If they are not, it sends a 400 error response with a message.

It also validates the email format using a regular expression. If the email is invalid, it sends an error response.

3. JSON Responses:

In the POST /submit route, if validation passes, the server responds with a success message in JSON format.

In case of a 404 (route not found), a JSON response is returned with an appropriate message.

To Run the Server:

1. Create a new server.js file with the above code.

2. Install dependencies by running:

npm install express

3. Start the server:

node server.js

4. Test with a tool like Postman or curl by making a POST request to http://localhost:3000/submit with a JSON body like:

```
{
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

If the data is valid, you will get a response like:

```
{
  "success": true,
  "message": "Data successfully received!",
  "data": {
    "name": "John Doe",
    "email": "john.doe@example.com"
  }
}
```

If the input is missing or invalid (e.g., no email or an incorrectly formatted email), the response will indicate an error:

```
{
  "success": false,
  "message": "Invalid email format."
}
```

This will create a fully functional Express server with body parsing, input validation, and JSON responses.

Slip 11

Q.1) Develop an Express.js application that defines routes for Create operations

on a resource (Movie).

To develop an Express.js application that defines routes for the Create operations on a Movie resource, follow these steps:

Steps to Create the Application:

1. Install Dependencies: First, you need to install Express.js if you haven't already.

Run the following command in your project folder to install Express:

npm install express

2. Create server.js File: This file will contain your Express application, including the routes and basic logic for creating movies.

3. Define the Routes for Create Operations: The "Create" operation in REST refers to a POST request to create a new resource. For the Movie resource, we will define a POST /movies route.

server.js File Code:

// Import the express module

```
const express = require('express');
const app = express();

// Use middleware to parse JSON request bodies
app.use(express.json());

// In-memory storage for movies (for simplicity)
let movies = [];

// POST route to create a new movie
app.post('/movies', (req, res) => {
  // Extract movie data from the request body
  const { title, director, year, genre } = req.body;

  // Simple validation to check that the required fields are present
  if (!title || !director || !year || !genre) {
    return res.status(400).json({
      success: false,
      message: 'All fields (title, director, year, genre) are required.'
    });
  }

  // Create a new movie object
  const newMovie = { id: movies.length + 1, title, director, year, genre };
```

```
  // Add the new movie to the in-memory storage
  movies.push(newMovie);

  // Send success response
  res.status(201).json({
    success: true,
    message: 'Movie created successfully!',
    data: newMovie
  });
});

// Handle unknown routes (404 error)
app.use((req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found.'
  });
});

// Define the port for the server
const port = 3000;

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
```

});

Explanation of the Code:

1. Body Parsing:

The line app.use(express.json()); uses Express's built-in middleware to parse incoming JSON request bodies. This allows us to access the data sent in the request body via req.body.

2. In-Memory Storage:

We use a simple array (movies) to store the movie data. In a real-world application, this would be replaced with a database.

3. POST /movies Route:

This route handles the Create operation for the Movie resource.

It expects a JSON body with title, director, year, and genre fields.

It performs a basic validation to ensure that all required fields are provided.

If the validation passes, a new movie object is created, assigned an id, and added to the movies array.

A success response with a 201 status code is sent, containing the created movie.

4. 404 Handler:

If the user attempts to access an undefined route, a 404 error is sent with a message.

To Run the Server:

1. Create a new folder for your project, and inside that folder, create the server.js file with the above code.

2. Initialize your project (if not already done) and install Express:

npm init -y

npm install express

3. Start the server:

node server.js

4. You should see a message indicating that the server is running at http://localhost:3000/.

Testing the API:

You can use Postman or curl to test the Create operation.

Example Request (using curl):

```
curl -X POST http://localhost:3000/movies \
-H "Content-Type: application/json" \
-d '{"title": "Inception", "director": "Christopher Nolan", "year": 2010, "genre": "Sci-Fi"}'
```

Example Response (on success):

```
{
  "success": true,
  "message": "Movie created successfully!",
  "data": {
    "id": 1,
```

```
    "title": "Inception",
    "director": "Christopher Nolan",
    "year": 2010,
    "genre": "Sci-Fi"
  }
}
```


Error Case (missing fields):


```
curl -X POST http://localhost:3000/movies \
-H "Content-Type: application/json" \
-d '{"title": "Inception", "director": "Christopher Nolan"}'
```


Response:


```
{
  "success": false,
  "message": "All fields (title, director, year, genre) are required."
}
```


This implementation handles the creation of movie resources and returns appropriate success or error responses in JSON format.

Q.2) Create Angular application that print the name of students who play basketball

using filter and map method.

To create an Angular application that prints the names of students who play basketball using the filter and map methods, follow these steps:

Step 1: Set up the Angular Application

1. Install Angular CLI (if you haven't already): If you don’t have Angular CLI installed, you can install it globally by running:

npm install -g @angular/cli

2. Create a New Angular Project: Create a new Angular project by running:

ng new student-app

3. Navigate to the Project Directory:

cd student-app

4. Run the Application: Start the development server to make sure everything is set up correctly:

ng serve

The app should be running at http://localhost:4200.

Step 2: Modify the Application

1. Create the Component: We will modify the default AppComponent to display the names of students who play basketball using filter and map methods.

2. Update app.component.ts:

Open src/app/app.component.ts and replace its contents with the following code:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  // List of students
  students = [
    { name: 'Alice', playsBasketball: true },
```

```
    { name: 'Bob', playsBasketball: false },
    { name: 'Charlie', playsBasketball: true },
    { name: 'David', playsBasketball: false },
    { name: 'Eva', playsBasketball: true }
  ];

  // Filter and map the list of students who play basketball
  basketballPlayers = this.students
    .filter(student => student.playsBasketball)  // Filter those who play basketball
    .map(student => student.name);  // Extract only their names
}
```

Explanation:

The students array contains objects with student names and a playsBasketball boolean indicating if they play basketball.

We use the filter() method to select only the students who play basketball (playsBasketball: true).

Then, we use the map() method to extract only the name property of those students.

3. Update app.component.html:

Now, update src/app/app.component.html to display the filtered list of basketball players:

```
<div style="text-align:center">
  <h1>
    Students Who Play Basketball
  </h1>
  <ul>
    <li *ngFor="let player of basketballPlayers">
      {{ player }}
    </li>
  </ul>
</div>
```

Explanation:

The <ul> element uses Angular's *ngFor directive to loop through the basketballPlayers array and display each name in a <li> element.

Step 3: Run the Application

Now, run the Angular application:

```
ng serve
```

Open your browser and navigate to http://localhost:4200. You should see the list of students who play basketball.

Final Output:

The page will display:

Students Who Play Basketball

- Alice

- Charlie

- Eva

Recap:

filter(): This method is used to create a new array containing only the students who play basketball.

map(): This method is then used to extract the name of each student who plays basketball from the filtered list.

This approach demonstrates how to use filter and map in an Angular component to manipulate and display data.

Slip 12

Q.1) Write an AngularJS script to print details of Employee (employee name, employee

Id,Pin code, address etc.) in tabular form using ng-repeat.

To create an AngularJS script that prints the details of employees (such as employee name, employee ID, pin code, address, etc.) in a tabular form using ng-repeat, follow these steps.

Step 1: Set up the HTML and AngularJS Script

1. Include AngularJS Library: You can either download AngularJS and include it locally or link to a CDN.

For simplicity, we'll use the AngularJS CDN.

2. Create an HTML file (e.g., index.html):

Here's an example of how to implement this using ng-repeat to display employee details in a table:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Employee Details</title>
  <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"></script>
</head>
<body ng-app="employeeApp" ng-controller="employeeController">
```

```
<h2>Employee Details</h2>

<table border="1" cellpadding="10">
  <thead>
    <tr>
      <th>Employee ID</th>
      <th>Name</th>
      <th>Pin Code</th>
      <th>Address</th>
    </tr>
  </thead>
  <tbody>
    <tr ng-repeat="employee in employees">
      <td>{{ employee.employeeId }}</td>
      <td>{{ employee.name }}</td>
      <td>{{ employee.pinCode }}</td>
      <td>{{ employee.address }}</td>
    </tr>
  </tbody>
</table>

<script>
  // Create AngularJS module and controller
  var app = angular.module('employeeApp', []);
```

```
  app.controller('employeeController', function($scope) {
    // Define an array of employees
    $scope.employees = [
      { employeeId: 101, name: 'John Doe', pinCode: '12345', address: '123 Elm St, Springfield' },
      { employeeId: 102, name: 'Jane Smith', pinCode: '67890', address: '456 Oak St, Shelbyville' },
      { employeeId: 103, name: 'Sam Johnson', pinCode: '11223', address: '789 Pine St, Capital City' },
      { employeeId: 104, name: 'Emily Davis', pinCode: '33445', address: '101 Maple St, Riverton' }
    ];
  });
 </script>

</body>
</html>
```

Explanation:

AngularJS Module and Controller:

We create an AngularJS module called employeeApp and a controller employeeController. Inside the controller, we define an array employees with objects that contain details for each employee.

ng-repeat:

The ng-repeat="employee in employees" directive is used to loop through the employees array and display each employee's details in a table row (<tr>).

For each employee, we bind the values of employeeId, name, pinCode, and address to the respective table cells using the double curly braces {{ }}.

Table Structure:

We use a <table> with column headers (<th>) for each property (ID, Name, Pin Code, Address).

The employee data is displayed dynamically in the rows (<tr>), with each row corresponding to an employee.

Step 2: Running the Application

1. Save the code in a file called index.html.

2. Open index.html in your browser.

You should see a table displaying employee details, like this:

Summary:

This AngularJS example demonstrates how to use ng-repeat to dynamically generate table rows for employee details.

The ng-repeat directive loops over the array of employees, and the data for each employee is bound to the respective table columns using AngularJS expressions.

Q.2) Develop an Express.js application that defines routes for Create operations

on a resource (User).

To develop an Express.js application that defines routes for Create operations on a User resource, follow these steps:

Step 1: Set up the Express.js Application

1. Install Express: First, you need to initialize a Node.js project and install Express.

Open a terminal and run the following commands:

```
mkdir user-app
cd user-app
npm init -y  # Initializes a new Node.js project
npm install express  # Installs Express.js
```

2. Create the Server File: Create a file called server.js to define the Express application.

Step 2: Define the Express Routes

We will define a POST route to create a user. The user will have a name, email, and password, and the data will be sent in the request body.

Step 3: Write the Code

Here is an example implementation of the Express.js application with a Create route for the User resource.

server.js:

```
// Import the express module
const express = require('express');
const app = express();

// Middleware to parse JSON bodies
app.use(express.json());

// In-memory storage for users (for simplicity)
let users = [];
```

```
// POST route to create a new user
app.post('/users', (req, res) => {
  const { name, email, password } = req.body;

  // Simple validation to check that the required fields are present
  if (!name || !email || !password) {
    return res.status(400).json({
      success: false,
      message: 'Name, email, and password are required.'
    });
  }

  // Create a new user object
  const newUser = { id: users.length + 1, name, email, password };

  // Add the new user to the in-memory storage
  users.push(newUser);

  // Send success response
  res.status(201).json({
    success: true,
    message: 'User created successfully!',
    data: newUser
  });
});
```

```
// Handle unknown routes (404 error)
app.use((req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found.'
  });
});

// Define the port for the server
const port = 3000;

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Explanation of the Code:

1. Express Setup:

We use express.json() middleware to parse incoming JSON data in the request body.

The users array is used to simulate in-memory storage for users. In a real application, this data would typically be stored in a database.

2. POST /users Route:

The POST /users route is used to create a new user.

We extract the name, email, and password from the request body using req.body.

If any of these fields are missing, a 400 error response is sent with a message.

A new user object is created with an id, name, email, and password.

The new user is added to the users array, and a success response with a 201 status code is returned.

3. 404 Handler:

If an undefined route is accessed, a 404 error with a message is sent.

Step 4: Run the Application

1. After creating the server.js file, open a terminal and run:

node server.js

2. The server will start running on http://localhost:3000/.

Step 5: Test the API

You can test the Create operation using Postman or curl.

Example Request (using curl):

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name": "John Doe", "email": "john.doe@example.com", "password": "password123"}'
```

Example Response (on success):

```
{
  "success": true,
  "message": "User created successfully!",
  "data": {
```

```
    "id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com",
    "password": "password123"
  }
}
```


Error Case (missing fields):


```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name": "John Doe", "email": "john.doe@example.com"}'
```


Example Response (error):


```
{
  "success": false,
  "message": "Name, email, and password are required."
}
```


Recap:

POST /users Route: Accepts data in the request body, validates it, and creates a new user.

Validation: Checks that all required fields (name, email, and password) are provided before creating a user.

Success Response: Returns a JSON object with a success message and the created user.

Error Handling: If required fields are missing, a 400 error with a message is returned.

This Express.js application provides a simple Create API for the User resource.

Slip 13

Q.1) Extend the previous Express.js application to include middleware for parsing

request bodies (e.g., JSON, form data) and validating input data. Send appropriate JSON

responses for success and error cases

To extend the previous Express.js application with middleware for parsing request bodies (such as JSON and form data) and validating input data, we can follow these steps:

Step 1: Set up Middleware for Parsing Request Bodies

1. Body Parsing Middleware: Express provides built-in middleware like express.json() to parse JSON data and express.urlencoded() for form data. We'll add both to handle different types of input.

2. Validation Middleware: We'll create a validation middleware to ensure the required fields are present in the request body. If any field is missing, the middleware will send an error response.

Step 2: Modify the Application

Here's the updated version of the server.js file that includes middleware for parsing request bodies and validating input data.

server.js:

```
// Import the express module
const express = require('express');
const app = express();

// Middleware to parse JSON bodies and URL-encoded form data
app.use(express.json());  // For parsing application/json
app.use(express.urlencoded({ extended: true }));  // For parsing application/x-www-form-urlencoded

// In-memory storage for users (for simplicity)
let users = [];

// Middleware to validate user input
function validateUserInput(req, res, next) {
  const { name, email, password } = req.body;

  // Check if the required fields are provided
  if (!name || !email || !password) {
```

```
    return res.status(400).json({
      success: false,
      message: 'Name, email, and password are required.'
    });
  }

  // Proceed to the next middleware if validation passes
  next();
}

// POST route to create a new user
app.post('/users', validateUserInput, (req, res) => {
  const { name, email, password } = req.body;

  // Create a new user object
  const newUser = { id: users.length + 1, name, email, password };

  // Add the new user to the in-memory storage
  users.push(newUser);

  // Send success response
  res.status(201).json({
    success: true,
    message: 'User created successfully!',
    data: newUser
```

```
  });
});

// Handle unknown routes (404 error)
app.use((req, res) => {
  res.status(404).json({
    success: false,
    message: 'Route not found.'
  });
});

// Define the port for the server
const port = 3000;

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Key Changes:

1. Body Parsing Middleware:

express.json() is used to parse incoming requests with JSON payloads.

express.urlencoded({ extended: true }) is used to parse URL-encoded form data (i.e., when form data is sent with Content-Type: application/x-www-form-urlencoded).

2. Validation Middleware:

The validateUserInput middleware checks if the name, email, and password fields are present in the request body. If any of these fields are missing, it sends a 400 Bad Request error with a message. If validation passes, the middleware calls next() to proceed to the route handler.

3. Error Handling for Invalid Routes:

A catch-all middleware at the end of the file handles 404 errors for unknown routes.

Step 3: Running the Application

1. Start the server by running the following command:

node server.js

2. The server will run at http://localhost:3000/.

Step 4: Test the API

You can now test the API with both JSON and form data payloads.

Test Case 1: Sending JSON Data

Use Postman or curl to test the Create operation with a JSON payload:

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name": "John Doe", "email": "john.doe@example.com", "password": "password123"}'
```

Expected Response (Success):

```
{
  "success": true,
  "message": "User created successfully!",
  "data": {
    "id": 1,
    "name": "John Doe",
    "email": "john.doe@example.com",
```

```
    "password": "password123"
  }
}
```


Test Case 2: Sending Form Data

You can also send form data using Postman or curl with the application/x-www-form-urlencoded content type:


```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/x-www-form-urlencoded" \
-d "name=Jane Smith&email=jane.smith@example.com&password=password456"
```


Expected Response (Success):


```
{
  "success": true,
  "message": "User created successfully!",
  "data": {
    "id": 2,
    "name": "Jane Smith",
    "email": "jane.smith@example.com",
    "password": "password456"
  }
}
```

Test Case 3: Missing Required Fields

If you send a request without some required fields (e.g., missing email), you should get a validation error:

```
curl -X POST http://localhost:3000/users \
-H "Content-Type: application/json" \
-d '{"name": "John Doe", "password": "password123"}'
```

Expected Response (Error):

```
{
  "success": false,
  "message": "Name, email, and password are required."
}
```

Recap:

Body Parsing Middleware: We use express.json() and express.urlencoded() to handle JSON and form data.

Validation Middleware: A custom middleware is added to check if required fields (name, email, and password) are present in the request body.

Error Handling: Appropriate error messages and status codes are sent if validation fails, and success responses are sent on successful user creation.

This setup makes the application more robust by handling different content types and ensuring that the input is properly validated before creating a new user.

Q.2) Create a simple Angular component that takes input data and displays it

To create a simple Angular component that takes input data and displays it, follow these steps:

Step 1: Set up the Angular Application

If you haven't already created an Angular application, follow these steps:

1. Install Angular CLI (if you haven't already):

npm install -g @angular/cli

2. Create a new Angular project:

ng new input-display-app

3. Navigate to the project folder:

cd input-display-app

4. Run the Angular application:

ng serve

Your application will be available at http://localhost:4200.

Step 2: Create a Simple Input Display Component

1. Generate a new component: Use Angular CLI to generate a new component called input-display.

ng generate component input-display

This will create a new folder src/app/input-display with the following files:

input-display.component.ts

input-display.component.html

input-display.component.css

input-display.component.spec.ts (for testing, which we'll ignore for now)

2. Modify the component to take input and display it:

input-display.component.ts:

In the input-display.component.ts, define a property to bind the input and a method to handle input changes.

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-input-display',
  templateUrl: './input-display.component.html',
  styleUrls: ['./input-display.component.css']
})
export class InputDisplayComponent {
  inputData: string = '';  // Property to store the input data

  // Optional method to handle changes (if needed)
  onInputChange(value: string): void {
    this.inputData = value;
  }
}
```

inputData: This is a property that will hold the value entered by the user.

onInputChange(value): This method is optional and demonstrates how you can handle input changes manually if needed.

input-display.component.html:

In the input-display.component.html, use two-way data binding to capture the input and display it.

```
<div style="text-align:center">
  <h2>Input Data Display</h2>

  <!-- Input field to capture user data -->
  <input type="text" [(ngModel)]="inputData" placeholder="Enter something" />

  <!-- Display the input data -->
  <p>You entered: {{ inputData }}</p>
</div>
```

The [(ngModel)] directive binds the inputData property to the input field. This is two-way binding, so any change in the input field will automatically update the inputData property and vice versa.

{{ inputData }} displays the value of the inputData property.

Important: Add FormsModule to your app module

To use ngModel for two-way binding, you need to import the FormsModule in your app.module.ts.

Open src/app/app.module.ts and import FormsModule:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { InputDisplayComponent } from './input-display/input-display.component';
import { FormsModule } from '@angular/forms';  // <-- Import FormsModule

@NgModule({
 declarations: [
  AppComponent,
  InputDisplayComponent
 ],
 imports: [
  BrowserModule,
  FormsModule  // <-- Add FormsModule to imports
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 3: Use the InputDisplay Component

In src/app/app.component.html, include the input-display component:

<app-input-display></app-input-display>

Step 4: Run the Application

1. Start the development server if it's not already running:

ng serve

2. Open your browser and navigate to http://localhost:4200. You should see an input field where you can enter data, and the text you type will be displayed below the input.

Recap:

InputDisplayComponent captures user input using [(ngModel)], which creates a two-way binding between the input field and the component's inputData property.

The entered data is displayed dynamically as it changes using interpolation ({{ inputData }}).

The FormsModule is required for using ngModel.

This simple Angular component demonstrates how to take input from the user and display it in real-time.

Slip 14

Q.1) Create Angular application that print the name of students who got 85% using filter

and map method.

To create an Angular application that prints the names of students who scored 85% or more using the filter and map methods, follow these steps:

Step 1: Set Up Angular Application

1. Install Angular CLI (if you haven't already):

npm install -g @angular/cli

2. Create a new Angular project:

ng new student-grades-app

3. Navigate into the project folder:

cd student-grades-app

4. Run the Angular application:

ng serve

Your application will be available at http://localhost:4200.

Step 2: Generate a New Component

You need to generate a new component that will display the list of students who scored 85% or more.

Run the following command to generate a new component called student-list:

ng generate component student-list

This will generate the necessary files for your component:

student-list.component.ts

student-list.component.html

student-list.component.css

student-list.component.spec.ts (for testing, which we'll ignore for now)

Step 3: Modify the student-list Component

1. Open student-list.component.ts and add the logic to filter and map the student data:

student-list.component.ts:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-student-list',
  templateUrl: './student-list.component.html',
  styleUrls: ['./student-list.component.css']
})
export class StudentListComponent {
  // Array of students with their names and grades
  students = [
    { name: 'Alice', grade: 90 },
```

```
    { name: 'Bob', grade: 75 },
    { name: 'Charlie', grade: 85 },
    { name: 'David', grade: 92 },
    { name: 'Eve', grade: 88 }
  ];

  // Filter and map students who got 85% or more
  filteredStudents = this.students
    .filter(student => student.grade >= 85)  // Filter students with grade >= 85
    .map(student => student.name);  // Map to only return the names of the students
}
```

students: An array of student objects, each containing a name and grade.

filteredStudents: This array is generated by:

First using filter() to only include students with a grade of 85% or more.

Then using map() to extract only the name property from the filtered students.

2. Now, modify the student-list.component.html file to display the names of the students who passed the filter.

student-list.component.html:

```
<div style="text-align: center;">
  <h2>Students Who Got 85% or More</h2>

  <!-- Display the list of student names -->
  <ul>
    <li *ngFor="let student of filteredStudents">{{ student }}</li>
  </ul>
</div>
```

The *ngFor directive is used to loop over the filteredStudents array and display each student's name.

Step 4: Use the student-list Component

To display this component, open src/app/app.component.html and include the app-student-list component:

app.component.html:

```
<app-student-list></app-student-list>
```

Step 5: Run the Application

1. If the Angular development server isn't already running, start it with:

ng serve

2. Open your browser and go to http://localhost:4200. You should see a list of students who scored 85% or higher.

Expected Output:

Students Who Got 85% or More

- Alice
- Charlie
- David
- Eve

Recap:

The application creates an array of students with names and grades.

The filter() method is used to filter out students who scored less than 85%.

The map() method is then used to extract the name property from the filtered students.

The result is displayed in an unordered list (<ul>) using Angular's *ngFor directive.

Q.2) Develop an Express.js application that defines routes for Create, Update operations

on a resource (Employee).

To develop an Express.js application that defines routes for Create and Update operations on an Employee resource, follow these steps:

Step 1: Initialize the Project

1. Create a new directory for your project and navigate to it:

mkdir employee-api

cd employee-api

2. Initialize the project:

npm init -y

3. Install Express:

npm install express

4. Create the server.js file where the Express application will be set up.

Step 2: Create the Express Application

1. Create the server.js file:

Inside the employee-api folder, create a file called server.js:

```
const express = require('express');
const app = express();
const port = 3000;

// Middleware to parse JSON request bodies
app.use(express.json());

// In-memory storage for employees (for simplicity)
let employees = [];

// Route to Create a new Employee
app.post('/employees', (req, res) => {
  const { name, position, salary } = req.body;
```

```
// Validate the input data
if (!name || !position || !salary) {
  return res.status(400).json({
    success: false,
    message: 'All fields (name, position, salary) are required.'
  });
}

// Create a new employee object
const newEmployee = {
  id: employees.length + 1,
  name,
  position,
  salary
};

// Add the new employee to the in-memory database
employees.push(newEmployee);

// Send success response
res.status(201).json({
  success: true,
  message: 'Employee created successfully!',
  data: newEmployee
});
```

```
});

// Route to Update an Employee
app.put('/employees/:id', (req, res) => {
  const { id } = req.params;
  const { name, position, salary } = req.body;

  // Find the employee by ID
  const employee = employees.find(emp => emp.id === parseInt(id));

  // If the employee doesn't exist, return 404 error
  if (!employee) {
    return res.status(404).json({
      success: false,
      message: 'Employee not found.'
    });
  }

  // Update the employee's information
  employee.name = name || employee.name;
  employee.position = position || employee.position;
  employee.salary = salary || employee.salary;

  // Send success response
  res.status(200).json({
```

```
    success: true,
    message: 'Employee updated successfully!',
    data: employee
  });
});

// Start the server
app.listen(port, () => {
  console.log(`Server running at http://localhost:${port}/`);
});
```

Step 3: Explanation of the Routes

1. Create Route (POST /employees):

The POST route creates a new employee by accepting a JSON body with the fields name, position, and salary.

It checks if all required fields are provided, and if not, returns a 400 Bad Request error.

If the fields are valid, it creates a new employee, adds it to the employees array, and returns the created employee in the response.

2. Update Route (PUT /employees/:id):

The PUT route updates an existing employee. The id of the employee to be updated is passed as a route parameter (:id).

It looks up the employee by id, and if found, it updates the employee's details with the new name, position, and salary (only the fields provided in the request body).

If the employee is not found, it returns a 404 Not Found error.

Otherwise, it returns the updated employee data.

3. Middleware:

The middleware express.json() is used to parse incoming JSON request bodies.

Step 4: Test the Application

1. Start the server:

Run the following command to start the Express server:

```
node server.js
```

The server will be running at http://localhost:3000.

2. Test the Create Route (POST /employees):

Using Postman or curl, test creating an employee.

```
curl -X POST http://localhost:3000/employees \
-H "Content-Type: application/json" \
-d '{"name": "John Doe", "position": "Software Engineer", "salary": 75000}'
```

Expected Response:

```
{
  "success": true,
  "message": "Employee created successfully!",
  "data": {
    "id": 1,
    "name": "John Doe",
    "position": "Software Engineer",
    "salary": 75000
```

```
  }
}
```

3. Test the Update Route (PUT /employees/:id):

To update the employee's details, use the PUT method with the employee ID in the URL. For example:

```
curl -X PUT http://localhost:3000/employees/1 \
-H "Content-Type: application/json" \
-d '{"position": "Senior Software Engineer", "salary": 80000}'
```

Expected Response:

```
{
  "success": true,
  "message": "Employee updated successfully!",
  "data": {
    "id": 1,
    "name": "John Doe",
    "position": "Senior Software Engineer",
    "salary": 80000
  }
}
```

Step 5: Error Handling

If you try to update an employee that doesn't exist, you will get a 404 error:

```
curl -X PUT http://localhost:3000/employees/99 \
-H "Content-Type: application/json" \
-d '{"position": "Manager", "salary": 90000}'
```

Expected Response:

```
{
  "success": false,
  "message": "Employee not found."
}
```

If you attempt to create an employee with missing fields, you will get a 400 error:

```
curl -X POST http://localhost:3000/employees \
-H "Content-Type: application/json" \
-d '{"name": "Jane Smith", "position": "Developer"}'
```

Expected Response:

{

"success": false,

"message": "All fields (name, position, salary) are required."

}

Recap

Create Route: POST /employees to create a new employee.

Update Route: PUT /employees/:id to update an existing employee.

Error Handling: Includes checks for missing fields or non-existent employees.

Slip 15

Q.1) Find an emp with a Salary greater than 25000 in the array. (Using find by id method)

To find an employee with a salary greater than 25,000 using the find method by the employee's id, you can follow these steps in an Express.js application or in a simple JavaScript code snippet.

Example Code:

Here’s how you would find an employee in an array of employee objects based on the id and salary condition:

```
// Sample array of employees
const employees = [
  { id: 1, name: 'John Doe', position: 'Software Engineer', salary: 30000 },
  { id: 2, name: 'Jane Smith', position: 'Manager', salary: 25000 },
  { id: 3, name: 'Sam Brown', position: 'Developer', salary: 28000 },
  { id: 4, name: 'Lisa White', position: 'HR', salary: 22000 }
];

// Function to find employee by id with salary > 25000
function findEmployeeByIdAndSalary(id) {
  return employees.find(emp => emp.id === id && emp.salary > 25000);
}

// Example usage: Find employee with id 1
const employee = findEmployeeByIdAndSalary(1);

if (employee) {
  console.log('Employee found:', employee);
} else {
  console.log('No employee found with the given criteria');
}
```

Explanation:

employees.find(): The find() method is used to search the employees array and return the first employee that satisfies the given condition.

Condition: emp.id === id && emp.salary > 25000 checks if the employee's id matches the specified id and if their salary is greater than 25,000.

Example Output:

If you search for employee with id = 1, the output would be:

Employee found: { id: 1, name: 'John Doe', position: 'Software Engineer', salary: 30000 }

If no employee matches the criteria (e.g., searching for id = 4), the output will be:

No employee found with the given criteria

Recap:

You can use the find() method to locate an employee in an array by id and ensure their salary exceeds 25,000 using a logical && condition.

Q.2) Create Angular application that print the name of students who got 85% using filter

and map method.

To create an Angular application that prints the names of students who got 85% using the filter and map methods, follow these steps:

Step 1: Set Up the Angular Application

1. Install Angular CLI (if you haven't already):

npm install -g @angular/cli

2. Create a new Angular project:

ng new student-filter-app

3. Navigate to the project folder:

cd student-filter-app

4. Run the Angular application:

ng serve

The application will be available at http://localhost:4200.

Step 2: Generate a New Component

1. Generate a new component:

ng generate component student-list

This will create the following files:

student-list.component.ts

student-list.component.html

student-list.component.css

Step 3: Modify the student-list Component

1. Open the student-list.component.ts file and add the student data and logic for filtering and mapping the students who scored 85% or more.

student-list.component.ts:

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-student-list',
  templateUrl: './student-list.component.html',
  styleUrls: ['./student-list.component.css']
})
export class StudentListComponent {
  // Array of students with their names and grades
  students = [
    { name: 'Alice', grade: 90 },
    { name: 'Bob', grade: 75 },
    { name: 'Charlie', grade: 85 },
    { name: 'David', grade: 92 },
    { name: 'Eve', grade: 88 }
  ];

  // Filter students who got 85% or more and map to return only their names
  filteredStudentNames = this.students
    .filter(student => student.grade >= 85)  // Filter students with grade >= 85
    .map(student => student.name);  // Map to return only the names of the students
}
```

students: An array of student objects, each containing a name and a grade.

filteredStudentNames: This is created by using the filter() method to select students with grades >= 85 and then using the map() method to return only the name property of the filtered students.

2. Open the student-list.component.html file and display the filtered student names.

student-list.component.html:

```
<div style="text-align: center;">
  <h2>Students Who Got 85% or More</h2>

  <!-- Display the list of student names who scored >= 85% -->
  <ul>
    <li *ngFor="let student of filteredStudentNames">{{ student }}</li>
  </ul>
</div>
```

*ngFor="let student of filteredStudentNames": This Angular directive is used to loop through the filtered list of student names and display each student's name in an unordered list.

Step 4: Use the student-list Component

1. Open src/app/app.component.html and include the student-list component:

app.component.html:

<app-student-list></app-student-list>

Step 5: Run the Application

1. Start the development server (if it's not already running):

ng serve

2. Open your browser and navigate to http://localhost:4200. You should see the list of student names who scored 85% or higher.

Expected Output:

Students Who Got 85% or More

- Alice
- Charlie
- David

- Eve

Recap:

The application contains an array of students with their names and grades.

The filter() method is used to select students who scored 85% or more.

The map() method extracts the name property of the filtered students.

The result is displayed in an unordered list (<ul>) using Angular's *ngFor directive.

This Angular application demonstrates how to filter and map data to display the names of students who achieved a specific grade.