Institute of Distance and Open Learning

VidyaNagari,Kalina,SantacruzEast-400098

A Practical Journal Submitted in fulfillment

of the degree of

MASTER OF SCIENCE

IN

COMPUTER SCIENCE

YEAR 2023-2024

Part 1-Semester 2

PSCSP2

"A:Cloud Computing (Concepts and Design of Web services)"

By

Mr. YADAV YOGESH RAMASHREY MEENADEVI

Application ID:

Seat No: 4500070

Under the Guidance of

Prof. Trupti Rongare

Institute of Distance and Open Learning

VidyaNagari, Kalina, Santacruz East–400098

CERTIFICATE

This is to certify that, this practical journal entitled **"A: Cloud Computing (Concepts and Design of Web services)"** is a record of work carried out by **Mr. YADAV YOGESH RAMASHREY MEENADEVI**, student of **Master of Science in Computer Science Part 1** class and is submitted to University of Mumbai, in partial fulfillment of the requirement for the award of the degree of **Master of Science in Computer Science.** The practical journal has been approved.

Guide	External Examiner	Coordinator–M.Sc.CS

Prof. Trupti Rongare

INDEX

Sr.No	Date	Practicals	Signature
1		Develop Time Server service that returns current time in Java and call it from clients developed in Java, PHP, Android and .NET.	
2		Develop Web service in Java that returns complex data types (e.g. as List of friends).	
3		Develop Web service in Java that returns matrix multiplication by Strassen's algorithm. Two matrices will be entered at run time by client. Server does the matrix multiplication and returns answer to client.	
4		Demonstrate CRUD operations with suitable database using SOAP or RESTful Web service.	
5		Develop Micro-blogger application (like Twitter) using RESTful Web services.	
6		Develop application to consume Google's search / Google's Map RESTful Web service.	
8		Develop application to download image/video from server or upload image/video to server using MTOM techniques.	

PRACTICAL 1

Aim :–Develop a Time Server service in Java

Part 1: Server Code (TimeServer.java)

This code sets up a RESTful web service using Java with the help of the javax.ws.rs API to return the current time.

1. Make sure you have the necessary dependencies in your project (e.g., javax.ws.rs, Jersey).

2. Run the server code on a Java server container like Tomcat or use a framework like Jersey's Grizzly HTTP server for a standalone version.

```
// TimeServer.java
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

// Define the URI for the service
@Path("/time")
public class TimeServer {

    // Define the GET method to return current time as plain text
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getCurrentTime() {
LocalDateTime now = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        return "Current Time: " + now.format(formatter);
    }
}
```

This service will provide the current time in the format YYYY-MM-DD HH:MM:SS whenever accessed via HTTP GET.

To deploy the server:

1. Add this service to a Jersey-based server (e.g., in a web.xml or via annotations for configuration).

2. Start the server and access http://localhost:8080/<your_app_context>/time to view the result.

Part 2: Client Code (TimeClient.java)

This Java client calls the Time Server and displays the current time received from the server.

```
// TimeClient.java
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.MediaType;

public class TimeClient {
    public static void main(String[] args) {
        // Define the server URI
        String serverUrl = "http://localhost:8080/<your_app_context>/time";

        // Create a new client
        Client client = ClientBuilder.newClient();
```

```
            // Send a GET request to the server and receive the response
            String response = client.target(serverUrl)
.request(MediaType.TEXT_PLAIN)
.get(String.class);

            // Display the server response
System.out.println("Response from Time Server: " + response);
        }
}
```

Server Output (Console log or web server logs):

- If the server logs requests, you'll see log entries indicating incoming requests to the /time endpoint.

Client Output:

Running TimeClient.java should yield something similar to the following output in the console:

sql

Copy code

```
Response from Time Server: Current Time: 2024-11-05 15:23:45
```

PRACTICAL 2

AIM: Develop a Web Service in Java that Returns Complex Data Types

We'll set up the server code for a REST API that provides a list of friends as a JSON response.
Server Code: FriendService.java

This example uses javax.ws.rs and javax.ws.rs.core APIs along with Jersey to create a RESTful web service.
Step 1: Define the Friend Data Model
java
Copy code

```
// Friend.java
public class Friend {
	private int id;
	private String name;
	private String email;

	public Friend(int id, String name, String email) {
		this.id = id;
		this.name = name;
this.email = email;
	}

	// Getters and Setters
	public int getId() { return id; }
	public void setId(int id) { this.id = id; }

	public String getName() { return name; }
	public void setName(String name) { this.name = name; }

	public String getEmail() { return email; }
	public void setEmail(String email) { this.email = email; }
}
```

Step 2: Define the FriendService Endpoint
This class defines an endpoint to return a list of Friend objects as JSON.
java
Copy code

```
// FriendService.java
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import java.util.ArrayList;
import java.util.List;

@Path("/friends")
public class FriendService {

	@GET
	@Produces(MediaType.APPLICATION_JSON)
	public List<Friend>getFriends() {
		List<Friend> friends = new ArrayList<>();
```

```
friends.add(new Friend(1, "Alice Smith", "alice@example.com"));
friends.add(new Friend(2, "Bob Johnson", "bob@example.com"));
friends.add(new Friend(3, "Charlie Brown", "charlie@example.com"));
        return friends;
    }
}
```

This code defines a RESTful endpoint /friends that, when accessed via an HTTP GET request, returns a list of friends in JSON format.

Step 3: Deploy the Service

1. Package and deploy the application on a Java web server (e.g., Tomcat) or use Jersey's Grizzly HTTP server for standalone testing.

2. Access the endpoint at http://localhost:8080/<your_app_context>/friends to retrieve the list of friends.

Testing the Service

Example Output (JSON)

When you access http://localhost:8080/<your_app_context>/friends, the service should return a JSON response similar to this:

json

Copy code

```
[
    {
        "id": 1,
        "name": "Alice Smith",
        "email": "alice@example.com"
    },
    {
        "id": 2,
        "name": "Bob Johnson",
        "email": "bob@example.com"
    },
    {
        "id": 3,
        "name": "Charlie Brown",
        "email": "charlie@example.com"
    }
]
```

Sample Java Client Code to Call the Service

Here's a simple Java client that consumes the FriendService API and prints the list of friends.

java

Copy code

```
// FriendClient.java
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;
import java.util.List;

public class FriendClient {
    public static void main(String[] args) {
        String serverUrl = "http://localhost:8080/<your_app_context>/friends";

        // Create a new client
        Client client = ClientBuilder.newClient();
```

```
            // Send a GET request to retrieve friends list
            List<Friend> friends = client.target(serverUrl)
.request(MediaType.APPLICATION_JSON)
.get(new GenericType<List<Friend>>() {});

            // Display the list of friends
System.out.println("List of Friends:");
            for (Friend friend : friends) {
System.out.println("ID: " + friend.getId() + ", Name: " + friend.getName() + ", Email: " + friend.getEmail());
            }
        }
}
```

Expected Output

Running FriendClient.java should yield something like the following in the console:

yaml

Copy code

```
List of Friends:
ID: 1, Name: Alice Smith, Email: alice@example.com
ID: 2, Name: Bob Johnson, Email: bob@example.com
ID: 3, Name: Charlie Brown, Email: charlie@example.com
```

PRACTICAL 3

Aim:- Web Service in Java for Matrix Multiplication using Strassen's Algorithm

Server Code

This service will:

1. Accept two matrices as JSON input.
2. Multiply the matrices using Strassen's algorithm.
3. Return the resulting matrix in JSON format.

Step 1: Define Matrix Model

java

Copy code

```
// Matrix.java
import java.util.Arrays;

public class Matrix {
	private int[][] data;

	public Matrix() {}

	public Matrix(int[][] data) {
this.data = data;
	}

	public int[][] getData() {
		return data;
	}

	public void setData(int[][] data) {
this.data = data;
	}

	@Override
	public String toString() {
		return Arrays.deepToString(data);
	}
}
```

This class represents a matrix with a 2D integer array data.

Step 2: Implement Strassen's Algorithm

Here's the implementation of Strassen's algorithm for matrix multiplication:

java

Copy code

```
// StrassenMatrixMultiplier.java
public class StrassenMatrixMultiplier {

	public static int[][] multiply(int[][] A, int[][] B) {
		int n = A.length;
		if (n == 1) {
int[][] C = {{A[0][0] * B[0][0]}};
			return C;
		}
```

```
            // Divide matrices into submatrices
int[][] A11 = new int[n / 2][n / 2];
int[][] A12 = new int[n / 2][n / 2];
int[][] A21 = new int[n / 2][n / 2];
int[][] A22 = new int[n / 2][n / 2];
int[][] B11 = new int[n / 2][n / 2];
int[][] B12 = new int[n / 2][n / 2];
int[][] B21 = new int[n / 2][n / 2];
int[][] B22 = new int[n / 2][n / 2];

            // Split matrix A and B
split(A, A11, 0, 0);
split(A, A12, 0, n / 2);
split(A, A21, n / 2, 0);
split(A, A22, n / 2, n / 2);
split(B, B11, 0, 0);
split(B, B12, 0, n / 2);
split(B, B21, n / 2, 0);
split(B, B22, n / 2, n / 2);

            // Calculate intermediate matrices
int[][] M1 = multiply(add(A11, A22), add(B11, B22));
int[][] M2 = multiply(add(A21, A22), B11);
int[][] M3 = multiply(A11, subtract(B12, B22));
int[][] M4 = multiply(A22, subtract(B21, B11));
int[][] M5 = multiply(add(A11, A12), B22);
int[][] M6 = multiply(subtract(A21, A11), add(B11, B12));
int[][] M7 = multiply(subtract(A12, A22), add(B21, B22));

            // Calculate submatrices of C
int[][] C11 = add(subtract(add(M1, M4), M5), M7);
int[][] C12 = add(M3, M5);
int[][] C21 = add(M2, M4);
int[][] C22 = add(subtract(add(M1, M3), M2), M6);

            // Combine submatrices into final result
int[][] C = new int[n][n];
join(C11, C, 0, 0);
join(C12, C, 0, n / 2);
join(C21, C, n / 2, 0);
join(C22, C, n / 2, n / 2);

            return C;
        }

        // Helper methods for Strassen's algorithm
        private static int[][] add(int[][] A, int[][] B) { /*...*/ }
        private static int[][] subtract(int[][] A, int[][] B) { /*...*/ }
        private static void split(int[][] P, int[][] C, int iB, int jB) { /*...*/ }
        private static void join(int[][] C, int[][] P, int iB, int jB) { /*...*/ }
}
```

Note: To keep the code concise, implement add, subtract, split, and join methods similarly to how matrix

operations are typically handled.

Step 3: Define the Matrix Multiplication Service

java

Copy code

```
// MatrixService.java
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

@Path("/matrix")
public class MatrixService {

	@POST
	@Path("/multiply")
	@Consumes(MediaType.APPLICATION_JSON)
	@Produces(MediaType.APPLICATION_JSON)
	public Response multiplyMatrices(MatrixRequest request) {
int[][] A = request.getMatrixA();
int[][] B = request.getMatrixB();

		// Validate matrices
		if (A.length != B.length || A[0].length != B[0].length || A.length != A[0].length) {
			return Response.status(Response.Status.BAD_REQUEST)
.entity("Both matrices must be square and of the same size.")
.build();
		}

		// Perform Strassen's matrix multiplication
int[][] result = StrassenMatrixMultiplier.multiply(A, B);
		return Response.ok(new Matrix(result)).build();
	}
}

// MatrixRequest.java - Request format class to hold input matrices
public class MatrixRequest {
	private int[][] matrixA;
	private int[][] matrixB;

	// Getters and Setters
	public int[][] getMatrixA() { return matrixA; }
	public void setMatrixA(int[][] matrixA) { this.matrixA = matrixA; }

	public int[][] getMatrixB() { return matrixB; }
	public void setMatrixB(int[][] matrixB) { this.matrixB = matrixB; }
}
```

This service takes a JSON object containing two matrices (matrixA and matrixB), multiplies them using Strassen's algorithm, and returns the result as a JSON response.

Testing the Service

To test the service, you can use a tool like **Postman**. Here's an example of the JSON payload you can send to the endpoint POST http://localhost:8080/<your_app_context>/matrix/multiply:

json

Copy code

```
{
```

```
    "matrixA": [
        [1, 2],
        [3, 4]
    ],
    "matrixB": [
        [5, 6],
        [7, 8]
    ]
}
```

Expected Output

The server will return the result of the matrix multiplication in JSON format:

json

Copy code

```
{
    "data": [
        [19, 22],
        [43, 50]
    ]
}
```

Sample Client Code

Here's a Java client to send the matrices to the service and print the result:

java

Copy code

```
// MatrixClient.java
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;

public class MatrixClient {
    public static void main(String[] args) {
        String serverUrl = "http://localhost:8080/<your_app_context>/matrix/multiply";

MatrixRequest request = new MatrixRequest(
                new int[][] {{1, 2}, {3, 4}},
                new int[][] {{5, 6}, {7, 8}}
        );

        Client client = ClientBuilder.newClient();
        Response response = client.target(serverUrl)
.request(MediaType.APPLICATION_JSON)
.post(Entity.entity(request, MediaType.APPLICATION_JSON));

        if (response.getStatus() == Response.Status.OK.getStatusCode()) {
            Matrix resultMatrix = response.readEntity(Matrix.class);
System.out.println("Resulting Matrix:");
            for (int[] row : resultMatrix.getData()) {
                for (int val : row) {
System.out.print(val + " ");
                }
System.out.println();
            }
```

```
            } else {
System.out.println("Error: " + response.readEntity(String.class));
            }
        }}
```

Expected Client Output:

yaml
Copy code

```
Resulting Matrix:
19 22
43 50
```

PRACTICAL 4

Aim:- Demonstrate CRUD Operations with Database Using SOAP or RESTful Web Service

- **Objective**: Implement basic CRUD (Create, Read, Update, Delete) operations using a database.

Setup and Code

Step 1: Set Up Maven Project and Dependencies

Create a Maven project and add the following dependencies in your pom.xml file:

xml

Copy code

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>EmployeeCRUD</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
<!-- Spring Boot Starter for RESTful services -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- Spring Boot Starter for JPA (Data Access) -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- H2 Database dependency for in-memory database -->
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
</dependency>
</dependencies>

<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
</plugins>
</build>
</project>
```

Step 2: Configure the Application

In src/main/resources/application.properties, configure the H2 database:

properties

Copy code

```
# Enable H2 console for debugging
spring.h2.console.enabled=true
```

```
spring.h2.console.path=/h2-console
# Use H2 in-memory database
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
```

Step 3: Create the Employee Entity

1. In src/main/java/com/example/employeecrud, create the Employee.java entity class:

java
Copy code

```
package com.example.employeecrud;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Employee {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String name;
	private String department;
	private Double salary;

	public Employee() {}

	public Employee(String name, String department, Double salary) {
		this.name = name;
this.department = department;
this.salary = salary;
	}

	public Long getId() { return id; }
	public void setId(Long id) { this.id = id; }
	public String getName() { return name; }
	public void setName(String name) { this.name = name; }
	public String getDepartment() { return department; }
	public void setDepartment(String department) { this.department = department; }
	public Double getSalary() { return salary; }
	public void setSalary(Double salary) { this.salary = salary; }
}
```

Step 4: Create the Employee Repository

Create an interface EmployeeRepository.java to handle database operations using Spring Data JPA.

java
Copy code

```
package com.example.employeecrud;

import org.springframework.data.jpa.repository.JpaRepository;

public interface EmployeeRepository extends JpaRepository<Employee, Long> {}
```

Step 5: Create the Employee Controller

In src/main/java/com/example/employeecrud, create the EmployeeController.java to implement the CRUD endpoints.

java

Copy code

```
package com.example.employeecrud;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/employees")
public class EmployeeController {

	@Autowired
	private EmployeeRepositoryemployeeRepository;

	// Create Employee
	@PostMapping
	public Employee createEmployee(@RequestBody Employee employee) {
		return employeeRepository.save(employee);
	}

	// Read All Employees
	@GetMapping
	public List<Employee>getAllEmployees() {
		return employeeRepository.findAll();
	}

	// Read Single Employee by ID
	@GetMapping("/{id}")
	public ResponseEntity<Employee>getEmployeeById(@PathVariable Long id) {
		Optional<Employee> employee = employeeRepository.findById(id);
		return employee.map(ResponseEntity::ok).orElseGet(() ->ResponseEntity.notFound().build());
	}

	// Update Employee by ID
	@PutMapping("/{id}")
	public ResponseEntity<Employee>updateEmployee(@PathVariable Long id, @RequestBody Employee
employeeDetails) {
		Optional<Employee> employee = employeeRepository.findById(id);

		if (employee.isPresent()) {
			Employee existingEmployee = employee.get();
existingEmployee.setName(employeeDetails.getName());
existingEmployee.setDepartment(employeeDetails.getDepartment());
existingEmployee.setSalary(employeeDetails.getSalary());
			Employee updatedEmployee = employeeRepository.save(existingEmployee);
```

```
            return ResponseEntity.ok(updatedEmployee);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Delete Employee by ID
    @DeleteMapping("/{id}")
    public ResponseEntity<Void>deleteEmployee(@PathVariable Long id) {
        if (employeeRepository.existsById(id)) {
employeeRepository.deleteById(id);
            return ResponseEntity.noContent().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }
}
```

Step 6: Run the Application

In the terminal, run the following command to start the Spring Boot application:

bash
Copy code
mvnspring-boot:run

The application will run on http://localhost:8080. You can test the endpoints using tools like **Postman** or **curl**.

Testing the CRUD Operations

1. **Create an Employee**

   - **Endpoint**: POST http://localhost:8080/api/employees
   - **Request Body**:

json
Copy code
```
{
  "name": "Alice",
  "department": "HR",
  "salary": 50000.0
}
```

   - **Response**:

json
Copy code
```
{
  "id": 1,
  "name": "Alice",
  "department": "HR",
  "salary": 50000.0
}
```

2. **Get All Employees**

   - **Endpoint**: GET http://localhost:8080/api/employees
   - **Response**:

json
Copy code

```
[
   {
      "id": 1,
      "name": "Alice",
      "department": "HR",
      "salary": 50000.0
   }
]
```

3. **Get Employee by ID**

   - **Endpoint**: GET http://localhost:8080/api/employees/1
   - **Response**:

json
Copy code

```
{
   "id": 1,
   "name": "Alice",
   "department": "HR",
   "salary": 50000.0
}
```

4. **Update Employee**

   - **Endpoint**: PUT http://localhost:8080/api/employees/1
   - **Request Body**:

json
Copy code

```
{
   "name": "Alice Smith",
   "department": "Finance",
   "salary": 55000.0
}
```

   - **Response**:

json
Copy code

```
{
   "id": 1,
   "name": "Alice Smith",
   "department": "Finance",
   "salary": 55000.0
}
```

5. **Delete Employee**

   - **Endpoint**: DELETE http://localhost:8080/api/employees/1
   - **Response**: 204 No Content (if successful).

Explanation

- **Entity**: The Employee class represents an employee record in the database.
- **Repository**: EmployeeRepository provides CRUD operations without implementing SQL.

- **Controller**: EmployeeController maps HTTP requests to specific CRUD operations.

PRACTICAL 5

Aim:- Develop Micro-Blogger Application (Like Twitter) Using RESTful Web Services

- **Objective**: Build a simplified Twitter-like microblogging application.

1. **Create a Post**: Users can create posts with text content.
2. **Read Posts**: Users can view all posts or a specific post by ID.
3. **Update a Post**: Users can update their existing posts.
4. **Delete a Post**: Users can delete their posts.
5. **Get Posts by User**: Users can filter posts by the author.

Tech Stack

1. **Spring Boot**: For building REST APIs.
2. **H2 Database**: An in-memory database for simplicity.
3. **Maven**: For dependency management.

Project Setup

1. **Initialize a Spring Boot Project** with **Spring Web** and **Spring Data JPA**.
2. **Add dependencies** in pom.xml.

Step-by-Step Code Implementation

Step 1: Set Up Project Dependencies

Add dependencies to pom.xml:

xml

Copy code

```
<project xmlns="http://maven.apache.org/POM/4.0.0">
<modelVersion>4.0.0</modelVersion>
<groupId>com.example</groupId>
<artifactId>MicroBlogger</artifactId>
<version>1.0-SNAPSHOT</version>

<dependencies>
<!-- Spring Boot Starter Web for REST API -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<!-- Spring Data JPA for database access -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<!-- H2 Database for in-memory persistence -->
<dependency>
<groupId>com.h2database</groupId>
<artifactId>h2</artifactId>
<scope>runtime</scope>
```

</dependency>
</dependencies>
</project>

Step 2: Configure the Application

In src/main/resources/application.properties:

properties

Copy code

```
spring.h2.console.enabled=true
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=update
```

Step 3: Create the Post Entity

In src/main/java/com/example/microblogger, create Post.java to represent a blog post:

java

Copy code

```
package com.example.microblogger;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Post {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private Long id;
	private String author;
	private String content;

	public Post() {}

	public Post(String author, String content) {
this.author = author;
this.content = content;
	}

	public Long getId() { return id; }
	public String getAuthor() { return author; }
	public String getContent() { return content; }

	public void setAuthor(String author) { this.author = author; }
	public void setContent(String content) { this.content = content; }
}
```

Step 4: Create the Repository

Create PostRepository.java interface:

java

Copy code

```
package com.example.microblogger;

import org.springframework.data.jpa.repository.JpaRepository;
```

```
import java.util.List;

public interface PostRepository extends JpaRepository<Post, Long> {
	List<Post>findByAuthor(String author);
}
```

Step 5: Implement the Post Controller

Create PostController.java to handle REST API requests:

java

Copy code

```
package com.example.microblogger;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController
@RequestMapping("/api/posts")
public class PostController {

	@Autowired
	private PostRepositorypostRepository;

	// Create a new post
	@PostMapping
	public Post createPost(@RequestBody Post post) {
		return postRepository.save(post);
	}

	// Get all posts
	@GetMapping
	public List<Post>getAllPosts() {
		return postRepository.findAll();
	}

	// Get a post by ID
	@GetMapping("/{id}")
	public ResponseEntity<Post>getPostById(@PathVariable Long id) {
		Optional<Post> post = postRepository.findById(id);
		return post.map(ResponseEntity::ok).orElseGet(() ->ResponseEntity.notFound().build());
	}

	// Update a post by ID
	@PutMapping("/{id}")
	public ResponseEntity<Post>updatePost(@PathVariable Long id, @RequestBody Post postDetails) {
		Optional<Post> post = postRepository.findById(id);
		if (post.isPresent()) {
			Post existingPost = post.get();
existingPost.setAuthor(postDetails.getAuthor());
existingPost.setContent(postDetails.getContent());
```

```
            return ResponseEntity.ok(postRepository.save(existingPost));
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Delete a post by ID
    @DeleteMapping("/{id}")
    public ResponseEntity<Void>deletePost(@PathVariable Long id) {
        if (postRepository.existsById(id)) {
postRepository.deleteById(id);
            return ResponseEntity.noContent().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // Get posts by author
    @GetMapping("/author/{author}")
    public List<Post>getPostsByAuthor(@PathVariable String author) {
        return postRepository.findByAuthor(author);
    }
}
```

Step 6: Run the Application

Start the application using:

bash

Copy code

mvnspring-boot:run

Testing the Micro-Blogger Application

1. **Create a Post**

   - **Endpoint**: POST http://localhost:8080/api/posts
   - **Request Body**:

json

Copy code

```
{
   "author": "john_doe",
   "content": "Hello, this is my first post!"
}
```

   - **Response**:

json

Copy code

```
{
   "id": 1,
   "author": "john_doe",
   "content": "Hello, this is my first post!"
}
```

2. **Get All Posts**

   - **Endpoint**: GET http://localhost:8080/api/posts
   - **Response**:

json
Copy code

```
[
   {
      "id": 1,
      "author": "john_doe",
      "content": "Hello, this is my first post!"
   }
]
```

3. **Get Post by ID**

   - **Endpoint**: GET http://localhost:8080/api/posts/1
   - **Response**:

json
Copy code

```
{
   "id": 1,
   "author": "john_doe",
   "content": "Hello, this is my first post!"
}
```

4. **Update a Post**

   - **Endpoint**: PUT http://localhost:8080/api/posts/1
   - **Request Body**:

json
Copy code

```
{
   "author": "john_doe",
   "content": "Hello, I've updated my first post!"
}
```

   - **Response**:

json
Copy code

```
{
   "id": 1,
   "author": "john_doe",
   "content": "Hello, I've updated my first post!"
}
```

5. **Delete a Post**

   - **Endpoint**: DELETE http://localhost:8080/api/posts/1
   - **Response**: 204 No Content

6. **Get Posts by Author**

   - **Endpoint**: GET http://localhost:8080/api/posts/author/john_doe
   - **Response**:

json
Copy code

```
[
   {
      "id": 1,
      "author": "john_doe",
      "content": "Hello, I've updated my first post!"
   }
]
```


Explanation

- **Entity**: Post represents each micro-blog post.
- **Repository**: PostRepository provides basic CRUD functionality and a custom method for finding posts by author.
- **Controller**: PostController manages the API endpoints, allowing users to create, read, update, delete, and filter posts.

PRACTICAL 6

Aim:- To develop an application that consumes Google's Search or Google Maps RESTful API, you'll need to set up the Google API and use it to make HTTP requests. Here, I'll demonstrate how to consume Google Maps API in a Java Spring Boot application to search for nearby places.

Prerequisites

1. **Google Cloud API Key**:

   - Go to Google Cloud Console, enable **Google Maps API** (such as **Places API**), and create an API key. Note that this API key should be kept secure.

2. **Dependencies**: We'll use Spring Boot along with **RestTemplate** to make HTTP requests.

Project Setup

1. **Initialize a Spring Boot Project** with **Spring Web** dependency.

2. **Add Google API Key**: Store the Google API key in a properties file for security.

Step-by-Step Implementation

Step 1: Set Up Project Dependencies

Add dependencies in pom.xml for Spring Web:

xml

Copy code

```
<dependencies>
<!-- Spring Boot Starter Web for REST API and HTTP requests -->
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
</dependencies>
```

Step 2: Configure Application Properties

Store your **Google API Key** in src/main/resources/application.properties:

properties

Copy code

```
google.api.key=YOUR_GOOGLE_API_KEY
```

Step 3: Create the GoogleMapsService Class

In src/main/java/com/example/googlemaps, create a GoogleMapsService.java class that will handle the HTTP requests to Google Maps API.

java

Copy code

```
package com.example.googlemaps;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;

import java.util.Map;

@Service
public class GoogleMapsService {
```

```
    @Value("${google.api.key}")
    private String apiKey;

    private final RestTemplaterestTemplate = new RestTemplate();

    public Map<String, Object>searchNearbyPlaces(String location, String radius, String type) {
        String url =
UriComponentsBuilder.fromHttpUrl("https://maps.googleapis.com/maps/api/place/nearbysearch/json")
.queryParam("location", location)
.queryParam("radius", radius)
.queryParam("type", type)
.queryParam("key", apiKey)
.toUriString();

        return restTemplate.getForObject(url, Map.class);
    }
}
```

- **location**: The latitude and longitude of the search area (e.g., 37.7749,-122.4194 for San Francisco).
- **radius**: The search radius in meters (e.g., 500 for 500 meters).
- **type**: Type of place (e.g., restaurant, hospital).

Step 4: Create the GoogleMapsController Class

Create GoogleMapsController.java to handle HTTP requests and call GoogleMapsService.

java

Copy code

```
package com.example.googlemaps;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.Map;

@RestController
@RequestMapping("/api/maps")
public class GoogleMapsController {

    @Autowired
    private GoogleMapsServicegoogleMapsService;

    @GetMapping("/nearby")
    public Map<String, Object>getNearbyPlaces(
            @RequestParam String location,
            @RequestParam String radius,
            @RequestParam String type) {
        return googleMapsService.searchNearbyPlaces(location, radius, type);
    }
}
```

- **Endpoint**: /api/maps/nearby
- **Parameters**:

- location: Latitude and longitude (e.g., 37.7749,-122.4194).
- radius: Search radius in meters (e.g., 500).
- type: Place type (e.g., restaurant).

Step 5: Run the Application

Run the Spring Boot application:

bash

Copy code

mvnspring-boot:run

Testing the Application

You can test the API endpoint using a tool like **Postman** or **curl**:

1. **Search for Nearby Restaurants**:
   - **URL**:

http://localhost:8080/api/maps/nearby?location=37.7749,-122.4194&radius=500&type=restaurant

2. **Expected Response** (Sample):

json

Copy code

```
{
    "html_attributions": [],
    "results": [
        {
            "name": "Restaurant A",
            "geometry": {
                "location": {
                    "lat": 37.7741,
                    "lng": -122.4192
                }
            },
            "vicinity": "123 Main St, San Francisco"
        },
        {
            "name": "Restaurant B",
            "geometry": {
                "location": {
                    "lat": 37.7745,
                    "lng": -122.4195
                }
            },
            "vicinity": "456 Market St, San Francisco"
        }
    ],
    "status": "OK"
}
```

The results array contains nearby places with information like name, location coordinates, and vicinity (address).

Explanation

1. **GoogleMapsService**: This service class handles requests to Google Maps API using RestTemplate.
2. **GoogleMapsController**: Exposes the /api/maps/nearby endpoint to accept search parameters and return nearby places.
3. **Output**: The response includes details about nearby places, including names, coordinates, and addresses.

PRACTICAL 7

Aim. Develop application to download image/video from server or upload image/video to server using MTOM techniques.

Steps

1. **Create a JAX-WS web service with MTOM support** for uploading and downloading files.
2. **Generate a client** to interact with the web service.
3. **Run the client to upload and download** images or videos.

Prerequisites

- Java Development Kit (JDK)
- Apache Tomcat or any compatible Java EE server to host the service
- JAX-WS libraries (if using Java SE)

Step 1: Create the MTOM Web Service

1.1 Create the Service Interface

Define a web service interface that enables file upload and download. Annotate the interface to support MTOM.

java

Copy code

```
package com.example.mtom;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.xml.ws.soap.MTOM;
import javax.activation.DataHandler;

@MTOM
@WebService
public interface FileService {

    @WebMethod
    void uploadFile(String fileName, DataHandlerfileData);

    @WebMethod
DataHandlerdownloadFile(String fileName);
}
```

1.2 Implement the Service

In this implementation, files are stored on the server's local file system.

java

Copy code

```
package com.example.mtom;

import javax.activation.DataHandler;
import javax.jws.WebService;
import java.io.*;

@WebService(endpointInterface = "com.example.mtom.FileService")
public class FileServiceImpl implements FileService {

    private static final String FILE_DIRECTORY = "C:/file-storage/";
```

```
	@Override
	public void uploadFile(String fileName, DataHandlerfileData) {
		try (InputStreaminputStream = fileData.getInputStream();
OutputStreamoutputStream = new FileOutputStream(FILE_DIRECTORY + fileName)) {
byte[] buffer = new byte[4096];
			int bytesRead;
			while ((bytesRead = inputStream.read(buffer)) != -1) {
outputStream.write(buffer, 0, bytesRead);
			}
System.out.println("File " + fileName + " uploaded successfully.");
		} catch (IOException e) {
e.printStackTrace();
		}
	}

	@Override
	public DataHandlerdownloadFile(String fileName) {
		File file = new File(FILE_DIRECTORY + fileName);
		if (file.exists()) {
			return new DataHandler(new FileDataSource(file));
		} else {
System.out.println("File " + fileName + " not found on server.");
			return null;
		}
	}
}
```

In this code:

- The uploadFile method reads an incoming file stream and saves it to the server's local storage.
- The downloadFile method retrieves a file from the server's file system.

1.3 Deploy the Web Service

Configure and deploy the service to your web server (e.g., Apache Tomcat).

1. Add the JAX-WS service endpoint to web.xml:

xml
Copy code

```
<servlet>
<servlet-name>FileService</servlet-name>
<servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>FileService</servlet-name>
<url-pattern>/FileService</url-pattern>
</servlet-mapping>
```

2. Deploy the service on your server and access the WSDL at:

bash
Copy code

```
http://localhost:8080/your-app-name/FileService?wsdl
```

Step 2: Generate the Client

Generate a JAX-WS client using the WSDL URL.

shell
Copy code

wsimport -keep -p com.example.mtom.client http://localhost:8080/your-app-name/FileService?wsdl
This will create client stubs in the com.example.mtom.client package.

Step 3: Write the Client Code

The client will call the web service to upload and download files.

3.1 Client Code for Uploading a File

java
Copy code

```
package com.example.mtom.client;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import com.example.mtom.FileService;

public class FileClient {

	public static void main(String[] args) {
FileService service = new FileService_Service().getFileServicePort();

		// Upload an image or video
FileDataSourcedataSource = new FileDataSource("path/to/local/image_or_video.jpg");
DataHandlerfileData = new DataHandler(dataSource);

service.uploadFile("uploaded_image.jpg", fileData);
System.out.println("File uploaded successfully.");
	}
}
```

3.2 Client Code for Downloading a File

java
Copy code

```
package com.example.mtom.client;

import javax.activation.DataHandler;
import java.io.*;

public class FileClientDownload {

	public static void main(String[] args) {
FileService service = new FileService_Service().getFileServicePort();

		// Download an image or video
DataHandlerfileData = service.downloadFile("uploaded_image.jpg");

		if (fileData != null) {
			try (InputStreaminputStream = fileData.getInputStream();
OutputStreamoutputStream = new FileOutputStream("path/to/local/downloaded_image.jpg")) {
byte[] buffer = new byte[4096];
				int bytesRead;
				while ((bytesRead = inputStream.read(buffer)) != -1) {
outputStream.write(buffer, 0, bytesRead);
				}
System.out.println("File downloaded successfully.");
			} catch (IOException e) {
e.printStackTrace();
```

```
                }
            } else {
System.out.println("File not found on server.");
            }
        }
}
```

Step 4: Run the Application

1. **Start the Server**: Ensure the web service is deployed and running on your web server.
2. **Upload a File**: Run the FileClient to upload a file to the server.
3. **Download the File**: Run FileClientDownload to download the file from the server.

Expected Output

When you upload a file:

- On the server console: "File uploaded_image.jpg uploaded successfully."
- On the client console: "File uploaded successfully."

When you download a file:

- On the server console: "File downloaded_image.jpg downloaded successfully."
- On the client console: "File downloaded successfully."