PRESENTATION TITLE: EVENT MANAGEMENT SYSTEM

Introduction:

Briefly introduce the topic of your presentation, which is the development of an Event Management System.

System Architecture:

Provide an overview of the system architecture, highlighting the various components and their interactions.

Mention the use of Spring Boot as the framework for building the application.

Mention the Spring Boot features used, such as dependency injection, data repositories, and RESTful API development.

Service Layer:

MemberService:

Describe the MemberService interface responsible for handling user-related operations.

Highlight methods like addMember, Authenticate, getMember, updateMember, deleteMember, getAllUsers, and getAllMembers.

Mention how it interacts with the MemberRepo repository.

NotificationService:

Describe the NotificationService interface responsible for generating and managing notifications.

Highlight methods like notificationOnRegistration, notifyOnPayment, notifyOnBooking, notifyOnVenueAdd, deleteNotification, and getNotifications.

Explain how it interacts with the MemberRepo, NotificationRepo, BookingService, and VenueRepo.

VenueService:

Describe the VenueService interface responsible for managing venue-related operations.

Highlight methods like getVenuesByOrganizerId, getAllDistinctPlaces, getVenueOfPlace, addVenue, getVenue, getVenues, updateVenue, and deleteVenue.

Explain how it interacts with the EquipmentRepo, FoodItemRepo, EventRepo, and VenueRepo.

VenueService:

Describe the VenueService interface responsible for managing venue-related operations.

Highlight methods like getVenuesByOrganizerId, getAllDistinctPlaces, getVenueOfPlace, addVenue, getVenue, getVenues, updateVenue, and deleteVenue.

Explain how it interacts with the EquipmentRepo, FoodItemRepo, EventRepo, and VenueRepo.

BookingService:

Describe the BookingService interface responsible for handling booking-related operations.

Highlight methods like addBooking, getUpcomingBookedDates, isAvailable, isAvailableForPayment, getBooking, getBookings, getBookingsByUserId, bookingDetail, payment, and deleteBooking.

Explain how it interacts with the BookingRepo, MemberRepo, and VenueRepo.

EquipmentService:

Describe the EquipmentService interface responsible for managing equipment-related operations.

Highlight methods like getEquipmentsByVenueId, addEquipment, getEquipment, updateEquipment, and deleteEquipment.

Explain how it interacts with the EquipmentRepo.

EventService:

Describe the EventService interface responsible for handling event-related operations.

Highlight methods like getEventsByVenueId, addEvent, getByNameAndVenueId, getEvent, updateEvent, and deleteEvent.

Explain how it interacts with the EventRepo.

FoodItemService:

Describe the FoodItemService interface responsible for managing food item-related operations.

Highlight methods like getFoodItemsByVenueId, addFoodItem, getFoodItem, updateFoodItem, and deleteFoodItem.

Explain how it interacts with the FoodItemRepo.

Code Walkthrough:

Walk through the provided code snippets for each service and their implementation.

Emphasize important logic, such as booking availability checks, notification generation, venue management, etc.

Explain how the services interact with repositories, perform CRUD operations, and ensure data integrity.

Conclusion:

Summarize the key features and functionalities of the Event Management System.

Highlight how the system promotes efficient event planning, booking, and notification management.

Conclude by mentioning the benefits of using Spring Boot for developing such systems.

Certainly! CRUD operations and RESTful API are essential concepts in web development, and they play a significant role in your Event Management System code. Let's dive into these concepts:

CRUD Operations:

CRUD stands for Create, Read, Update, and Delete. These operations are fundamental actions performed on data in a system:

Create (C):

Creating new records or entries in a database.

In your code, this is implemented through methods like addMember, addVenue, addBooking, addEquipment, addEvent, and addFoodItem.

Read (R):

Retrieving existing records or entries from a database.

In your code, this is implemented through methods like getMember, getVenue, getBooking, getEquipment, getEvent, and getFoodItem.

Update (U):

Modifying or updating existing records in a database.

In your code, this is implemented through methods like updateMember, updateVenue, updateBooking, updateEquipment, updateEvent, and updateFoodItem.

Delete (D):

Removing records or entries from a database.

In your code, this is implemented through methods like deleteMember, deleteVenue, deleteBooking, deleteEquipment, deleteEvent, and deleteFoodItem.

RESTful API:

REST (Representational State Transfer) is an architectural style for designing networked applications. A RESTful API adheres to the principles of REST, making it easy for different systems to communicate and interact with each other. Here's how your code implements RESTful APIs:

Resource Identification:

Each entity in your system (e.g., Member, Venue, Booking, etc.) is treated as a resource.

Resources are identified using URIs (Uniform Resource Identifiers).

HTTP Methods:

HTTP methods correspond to CRUD operations:

POST: Used to create a new resource. Examples: addMember, addVenue, etc.

GET: Used to read or retrieve a resource. Examples: getMember, getVenue, etc.

PUT: Used to update an existing resource. Examples: updateMember, updateVenue, etc.

DELETE: Used to delete a resource. Examples: deleteMember, deleteVenue, etc.

HTTP Status Codes:

HTTP status codes indicate the result of an API request. For example:

200 OK: Successful retrieval of a resource.

201 Created: Successful creation of a resource.

204 No Content: Successful deletion of a resource.

400 Bad Request: Invalid request or data.

404 Not Found: Requested resource not found.

500 Internal Server Error: Server error.

Request and Response Formats:

APIs handle data in JSON or XML format.

Clients send requests in the desired format, and the server responds likewise.

Your API endpoints (methods) in the code are designed to handle these formats.

URI Design:

Your API endpoints (URIs) are designed to be logical and descriptive.

For example: /members/{memberId}, /venues/{venueId}, /bookings/{bookingId}, etc.

Overall:

Your code follows RESTful principles by exposing various service methods through APIs, which correspond to CRUD operations. Clients (web or mobile applications) can use these APIs to perform operations on resources like members, venues, bookings, equipment, events, and food items. Each operation is associated with an HTTP method, and the responses follow the appropriate status codes and data formats (JSON/XML).

By adhering to these principles, your code ensures that the system is accessible, maintainable, and follows industry best practices for web development and API design.

Certainly! Your project is an Event Management System that allows users to book venues, equipment, food items, and events. Let's walk through how your code works to provide this functionality:

Member Management:

Users can register and log in as members (users or organizers).

CRUD operations are available for members (users and organizers):

Create: addMember - Register a new member.

Read: getMember - Retrieve member details.

Update: updateMember - Update member information.

Delete: deleteMember - Delete a member account.

Venue Management:

Organizers can add, update, and delete venues.

Users can view venue details and book them.

CRUD operations are available for venues:

Create: addVenue - Add a new venue.

Read: getVenue - Retrieve venue details.

Update: updateVenue - Update venue information.

Delete: deleteVenue - Delete a venue.

Booking Management:

Users can book venues, equipment, and food items for events.

CRUD operations are available for bookings:

Create: addBooking - Book a venue and related items.

Read: getBooking - Retrieve booking details.

Update: Not explicitly implemented (can be added).

Delete: deleteBooking - Delete a booking.

Equipment and Food Item Management:

Organizers can add, update, and delete equipment and food items.

Users can view available equipment and food items during booking.

CRUD operations are available for equipment and food items:

Create: addEquipment and addFoodItem - Add new equipment/food items.

Read: getEquipment and getFoodItem - Retrieve item details.

Update: updateEquipment and updateFoodItem - Update item information.

Delete: deleteEquipment and deleteFoodItem - Delete an item.

Event Management:

Organizers can add, update, and delete events.

Users can view available events during booking.

CRUD operations are available for events:

Create: addEvent - Add a new event.

Read: getEvent - Retrieve event details.

Update: updateEvent - Update event information.

Delete: deleteEvent - Delete an event.

Notification Management:

Notifications are generated for registration, bookings, and payment.

CRUD operations are available for notifications:

Create: Automatically generated on registration, booking, and payment.

Read: getNotifications - Retrieve member-specific notifications.

Update/Delete: Not explicitly implemented (can be added).

Venue and Event Search:

Users can search for venues by place and view events by venue.

The API endpoints allow users to retrieve venues and events based on specific criteria.

Payment and Booking Status:

The payment and booking status is managed through methods like isAvailableForPayment and checkActiveBookings.

Roles and Authentication:

The system distinguishes between user and organizer roles.

Authentication and authorization are implemented for accessing specific functionalities.

RESTful API:

Each entity has a set of API endpoints (URIs) that correspond to CRUD operations.

Requests are made using HTTP methods (POST, GET, PUT, DELETE).

Responses are in JSON format and include appropriate status codes.

Data Flow:

Users interact with the system through a web or mobile app.

The app sends HTTP requests to the server's API endpoints.

The server processes the requests using the implemented service methods.

The server communicates with the database to perform CRUD operations.

The server returns appropriate responses (JSON) to the app.

Overall, your Event Management System code functions as a RESTful API, allowing users to create, read, update, and delete various resources such as members, venues, bookings, equipment, events, and notifications. Users interact with the system through an interface (front-end app), and the server handles the business logic and data storage aspects.

Certainly! Connecting your Spring Boot application to a MySQL database involves several steps. Below, I'll outline the process of setting up a connection between your application and a MySQL database:

Configure Database Properties:

In your Spring Boot application, you need to provide the necessary database connection properties. These properties are typically configured in the application.properties or application.yml file. Here's an example configuration for

MySQL:

properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/your_database_name
spring.datasource.username=your_username
spring.datasource.password=your_password
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
```

Add MySQL Dependency:

Make sure you have the MySQL JDBC driver dependency added to your pom.xml (if using Maven) or build.gradle (if using Gradle) file. Here's an example for Maven:

xml

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.26</version> <!-- Use the appropriate version -->
</dependency>
```

Create Entity Classes:

You need to create entity classes that represent the database tables. These classes should be annotated with @Entity and other relevant annotations, such as @Id, @GeneratedValue, and @Column. For example:

java

```
@Entity
public class Member {
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int memberId;

    @Column(nullable = false)
    private String firstName;

    // Other fields and getters/setters
}
```

Create Repositories:

Create repositories by extending the Spring Data JPA CrudRepository or JpaRepository interfaces. These interfaces provide methods for performing CRUD operations on your entities. For example:

java

```
public interface MemberRepo extends JpaRepository<Member, Integer> {
    // Additional custom query methods can be defined here
}
```

Enable JPA:

Annotate your main application class with @SpringBootApplication and add the @EnableJpaRepositories annotation to enable Spring Data JPA. For example:

java

```
@SpringBootApplication
@EnableJpaRepositories(basePackages = "com.management.Event.repositories")
public class EventManagementApplication {
    public static void main(String[] args) {
        SpringApplication.run(EventManagementApplication.class, args);
    }
}
```

Use Repositories in Services:

Inject the repositories into your service classes to perform database operations. For example:

java

```
@Service
public class MemberServiceImpl implements MemberService {
    @Autowired
    private MemberRepo memberRepo;

    // Implement service methods using memberRepo
}
```

With these steps in place, your Spring Boot application should be able to connect to your MySQL database and perform CRUD operations on your entities. The Spring Data JPA framework handles the underlying database operations, making it easier to work with databases in your application.

To connect your Spring Boot backend to an Angular frontend, you'll need to set up communication between the two using RESTful APIs. Here's a high-level overview of the steps you'll need to take:

Create RESTful APIs in Spring Boot:

Your Spring Boot backend should expose RESTful APIs to provide data to the Angular frontend. These APIs will handle requests for data retrieval, creation, updating, and deletion. You've already implemented these APIs in your Spring Boot code. Make sure they are working correctly and return the required data.

Set Up Angular Project:

Create a new Angular project or use an existing one. You can use the Angular CLI to create a new project:

bash

```
ng new your-project-name
```

Angular Service for API Communication:

Create Angular services to communicate with your Spring Boot APIs. These services will use Angular's built-in HttpClient module to make HTTP requests to your backend. For example:

typescript

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

@Injectable({
```

```
  providedIn: 'root'
})
export class ApiService {
  private baseUrl = 'http://localhost:8080'; // Your Spring Boot backend URL

  constructor(private http: HttpClient) {}
getMembers(): Observable<any> {
    return this.http.get(`${this.baseUrl}/api/members`);
  }
  // Implement methods for other API endpoints (create, update, delete)
}
```

Angular Components:

Create Angular components that will display data to users and handle user interactions. These components will use the services you've created to fetch and manipulate data. For example, you might create a member list component:

```
typescript
import { Component, OnInit } from '@angular/core';
import { ApiService } from './api.service';
@Component({
  selector: 'app-member-list',
  templateUrl: './member-list.component.html',
  styleUrls: ['./member-list.component.css']
})
export class MemberListComponent implements OnInit {
  members: any[] = [];
  constructor(private apiService: ApiService) {}
  ngOnInit() {
    this.apiService.getMembers().subscribe((data) => {
      this.members = data;
    });
  }
}
```

Display Data:

Use Angular's data-binding and component templates to display data from your APIs. For example, you might iterate over the members array to display a list:

```
html

<ul>

 <li *ngFor="let member of members">{{ member.firstName }} {{ member.lastName }}</li>

</ul>
```

Handle User Interactions:

Implement user interactions, such as form submissions, using Angular forms and event bindings. These interactions can trigger API requests to perform CRUD operations on the backend.

Routing:

Set up Angular routing to navigate between different components. Configure routes to map to different URLs and components within your application.

Cross-Origin Resource Sharing (CORS):

Ensure that your Spring Boot backend has CORS configured to allow requests from your Angular frontend. You can configure CORS in your Spring Boot application's configuration.

Run Both Applications:

Start both your Spring Boot backend application and your Angular frontend application. The backend will typically run on a different port (e.g., 8080), and the frontend will run on a separate port (e.g., 4200).

Access the Frontend:

Open a web browser and navigate to http://localhost:4200 to access your Angular application.

By following these steps, you'll be able to create a connection between your Spring Boot backend and your Angular frontend, allowing data to be fetched from and sent to your backend using RESTful APIs.

System Architecture:

Your Event Management System is built using the Spring Boot framework, which provides a foundation for building robust Java applications. The architecture includes several layers:

Service Layer: This layer contains the business logic and interacts with the repositories to handle CRUD operations and other functionalities.

Repository Layer: This layer is responsible for database communication and includes JPA repositories for each entity (Member, Venue, Booking, etc.).

Controller Layer: Spring Boot controllers define RESTful endpoints that receive HTTP requests and delegate to the appropriate services.

Database: MySQL is used as the database to store data such as member details, venues, bookings, equipment, and notifications.

Frontend: Although not shown in the provided code, the frontend (likely built using Angular) would interact with the backend through RESTful APIs to display information and enable user interactions.

Service Layer:

MemberService: Manages member-related operations, such as registration, authentication, updating, and deletion. It interacts with the MemberRepo repository.

NotificationService: Generates and manages notifications for members, including registration, bookings, payments, and venue additions. It interacts with repositories and the BookingService and VenueRepo.

VenueService: Handles venue-related operations, including retrieval, addition, updating, and deletion. It interacts with the EquipmentRepo, FoodItemRepo, EventRepo, and VenueRepo.

BookingService: Manages booking-related operations, such as adding bookings, checking availability, retrieving booking details, and processing payments. It interacts with the BookingRepo, MemberRepo, and VenueRepo.

EquipmentService: Handles equipment-related operations, including retrieving, adding, updating, and deleting equipment. It interacts with the EquipmentRepo.

EventService: Manages event-related operations, such as adding, retrieving, updating, and deleting events. It interacts with the EventRepo.

FoodItemService: Manages food item-related operations, including retrieving, adding, updating, and deleting food items. It interacts with the FoodItemRepo.

Code Walkthrough:

Your code defines interfaces and classes for each service, which encapsulate the business logic and interact with repositories.

The services use Spring's dependency injection to access repositories and other services.

The methods within each service perform various operations, such as creating, retrieving, updating, and deleting data from the database.

Database Interaction:

Each service interacts with corresponding repositories, which extend Spring Data JPA interfaces.

Spring Data JPA provides high-level abstractions for database interactions, allowing you to perform CRUD operations without writing explicit SQL queries.

RESTful API:

Your code defines RESTful APIs using Spring Boot's controller classes.

Controllers handle incoming HTTP requests and delegate work to service methods.

The responses are typically in JSON format and include appropriate HTTP status codes.

Frontend (Not Shown):

Although not provided in the code, the frontend (built using Angular, React, or another framework) would make HTTP requests to the backend's RESTful APIs.

The frontend would display data to users and allow them to interact with the system, such as making bookings, updating profiles, and viewing notifications.

Conclusion:

Your Event Management System provides a comprehensive platform for users to manage events, bookings, and notifications.

It leverages Spring Boot's capabilities to create a robust backend that handles business logic, data storage, and API communication.

The system follows RESTful principles for API design, making it accessible and easy to integrate with different frontends.

By integrating these components, your Event Management System allows users to efficiently manage events, venues, bookings, and notifications through a user-friendly interface.