Movie Store Spring Application with Microservices

Movie Service

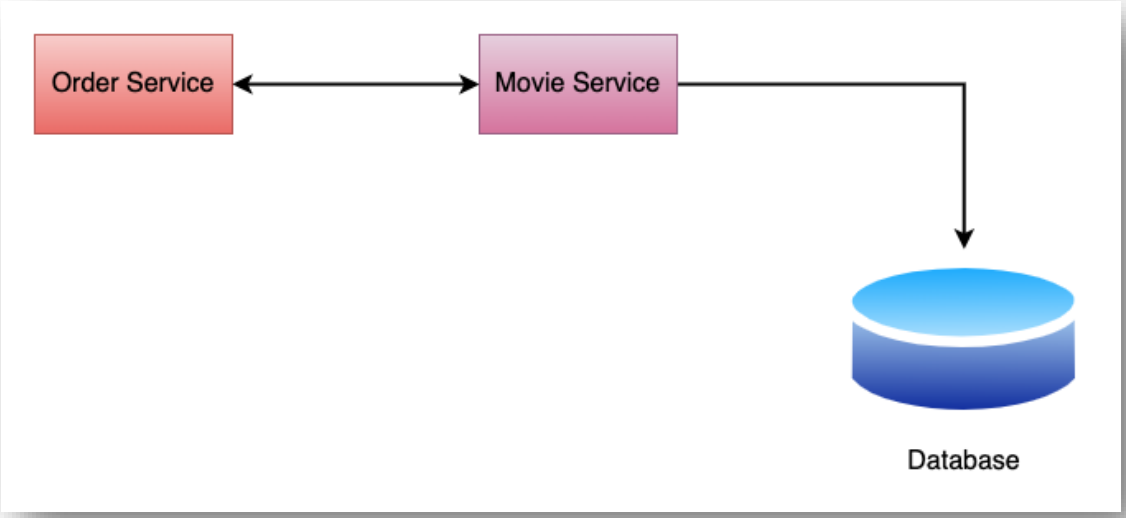

Figure 1: Service Flow

1. Navigate to https://start.spring.io/
2. Select Java as the Language and Select Maven from the Project Menu
3. Add Project Metadata
    a. Group – com.adeesha
    b. Artifact – movie-service
    c. Name – movie-service
    d. Description – Movie Service for Movie Store
    e. Package Name – com.adeesha.movie-service
4. Add Required Dependencies.
    a. Spring Web – For rest APIs
    b. Spring Data JPA – For Database Communication.
    c. MySQL Driver – To establish the connection with SQL DB.

    d. Lombok – To reduce boilerplate codes.

5. Generate the File and Open the project via IntelliJ.
6. Create the project structure.
7. **Configure the database.**
8. Go to mySQL workbench and crate a database named movies.
9. Navigate to resources -> application.properties and configure the DB connection with below details.
    a. Spring.datasource.url=jdbc:mysql://localhost:3306/movies
    b. Spring.datasource.username = <Username>
    c. Spring,datasource.password = <Password>
10. Configure the Hibernate Properties.
    a. Spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQLDialect
        i. Hibernate will create the queries that are compatible with MySQL
    b. Spring.jpa.hibernate.ddl-auto=update.
        i. If the tables are not created it automatically tells the hibernate to create the tables and update them accordingly.
11. Create the required packages.
    a. Model – to keep all JPA entities.
    b. Controller – MVC controller, Define the endpoint URLs.
    c. Repository – To keep our spring data JPA repositories.
    d. Exception – To handle custom exceptions.
    e. Entity – For DB Table Structure, to keep JPA Entities.
    f. Service – To keep all the interfaces and classes related to service layer.
    g. DTO – to keep all the DTO classes.
    h. Mapper – To keep all the mapping classes.
12. **Go to Entity Package and create a class named Movie.**
13. Add the relevant instance variables.
    a. Movie ID, Movie Name, Genre
14. Add relevant annotations to reduce boilerplate codes.
    a. @Getter
    b. @Setter

c. @NoArgsConstructor
d. @AllArgsConstructor

15. Make the Movie class as a JPA entity by using JPA annotations.

a. @Entity – To Specify the class as a JPA entity
b. @Table – Specify the table details.
    i. Specify the name attribute to give the table name.
    ii. @Table (name = “movies”)

16. Configure the Primary Key and Columns.

a. Add @Id annotation
b. Add @GeneratedValue (strategy = GenerationType.Identity) to auto increment the id.
c. Add @Column annotation to map the columns
    i. Ex - @Column (name= movie_name)

17. Create MovieRepository interface inside the repository package.

18. Extends the MovieRepositoryInterface from JpaRepository.

a. Jpa repository enables CRUD operations.

19. Create Movie DTO

20. Create Movie DTO inside the DTO Package.

a. DTO class is used to transfer the data between client and the server
b. When building the REST services we use this Movie DTO as a response for REST API’s
c. Add the required instance variables.
d. Movie ID, Movie Name, Genre

21. Add relevant annotations to reduce boilerplate codes.

a. @Getter
b. @Setter
c. @NoArgsConstructor
d. @AllArgsConstructor

22. Create Mapper classes to Map Movie Entity into MovieDto and Vise versa

23. Create a class Named Movie Mapper.

a. Create a Static method mapToMovieDto with the return type movieDTO and pass the Movie entity as a parameter.
b. Create a static method mapToMovie with the return type Movie and pass the MovieDTO as a parameter.

24. Building REST API's

25. Building Add Movie REST API

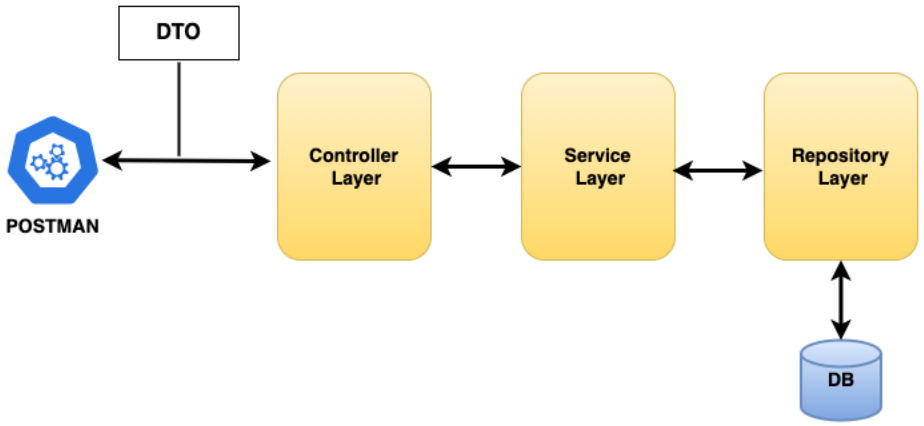

Figure 2: Development Steps

1. As shown in the above figure Controller layer depends on Service Layer. And Service Layer depends on Repository layer. We have already created the Repository layer. Since controller layer depends on service layer, we need to implement service layer 1st.
    a. Create the MovieService Interface
    b. Create the MovieServiceImpl class that implents the MovieService Interface.
    c. Build AddMovie REST API
    d. Test AddMovie REST API Using Postman.
2. Create a package called impl and create a class called MovieServiceImpl and implents it with the MovieService interface.
    a. Override the addMovie method. (You can also use the shortcut/Suggestion given in IntelliJ)
    b. Annotate the **MovieServiceImpl** class with @Service annotation. @Service tells the spring container to create the spring bean for this class.
    c. Add the dependency Injection inside the class.

i. Private MovieRepository movieRepository;
ii. Add @AllArgsConstructor to the Class to use constructor-based dependency injection.
iii. Implement the addMovie method.
iv. Now we need to add a movie to the database. But the DB can save only JPA entity. Postman passes DTO objects. So we need to convert the DTO into JPA entity using the mapper class we implemented earlier.
v. Save the created Movie JPA entity to DB and assign it to a local variable.
vi. Since DB returns JPA entity but postman consumes DTO objects we need to convert it to DTO. For that return the savedMovie as a MovieDto using the mapper class.

3. Navigate to the controller package and create a class called MovieController.
    a. Annotate this class with @RestController annotation. This annotation signifies that this class is capable to handle http requests and responses.
    b. Annotate the class with @RequestMapping annotation to declare the base URL for API.
    c. Add the dependency injection.
    d. Annotate with @AllArgsConstructor.
4. Implement the logic of addMovie method. This method returns ResponseEntity<MovieDto> and accepts a parameter of type MovieDTO.
    a. Annotate with @PostMapping annotation to make this method a rest API.
    b. Add @RequestBody annotation inside the parameter braces to extract the JSON from the HTTP request and it will convert that JSON into MovieDto java object.
5. Run the Spring Boot Application and test it with Postman.
6. Verify the results by checking the DB.

26. Implement the ResourceNotFoundException class in the Exception Package.

a. This exception is created to throw an exception when a resource is not found.
b. Ex – When a user tries to retrieve a movie with an Id that is not in the DB, this exception should be thrown.
c. Extend this class from RunTimeException class.

d. Create a Constructor for the class with a parameter called message of the type String.
e. Pass the message to the super class constructor.
f. Annotate the class with @ResponseStatus and pass the value as HttpStatus.NOT_FOUND.

27. Build Get Movie Rest API

a. Build Flow ->
    i. First Change the Service Layer.
    ii. Then Change the Controller Layer.
    iii. Why> ->. Because the Controller layer depends on Service Layer.
b. In the Service layer define GetMovieByIdMethod.
    i. The Return type of the method should be Movie DTO.
    ii. Method takes a Parameter of type int as the id.
c. Implement the above created method in MovieServiceImpl class.
d. Implement the logic for GetMovieById method.
e. Navigate to the Controller and Build the getMovieById rest API.
f. Annotate the method with @GetMapping annotation to map the incoming Http request to this GetMovieById method.
g. Pass the "{id}" to @GetMapping annotation.
h. Add the Parameter int movieId by @PathVariable annotation to bind the value "{id}" to method argument.

28. Build getAllMovies Rest API

a. Create a method that return a list of MovieDTO's with the name getAllMovies().
b. Implement the above created method in MovieServiceImpl class.
    i. Hint: Use Java Streams for mapping.
c. Navigate to the Controller package and create the getAllMovies Rest API.
d. Write the logic of the method and annotate with it @GetMapping annotation to map the incoming http requests to getAllMovies() method.

29. Build Update Movie Rest API

a. Use the same build flow. First Implement the Service layer and then the Controller Layer.

b. Create updateMovie method with the parameters of int Id and MovieDto updatedMovie.
c. Implement the above created method in MovieServiceImpl class.
    i. Find the Movie by Id, if not found throw ResourceNotFoundException.
    ii. Set the relevant properties.
    iii. Save the above set properties to a new object called UpdatedMovieObject using the save method in movieRepository class.
    iv. Return the updated movie object by converting it to a DTO.
d. Navigate to the Controller Package in implement the logic of updateMovie method.
    i. Write the necessary logic of the method and annotate the method with @PutMapping annotation to map incoming Http put requests to updateMovieMethod.
    ii. Pass the Id to PutMapping annotation. This is a URL Template variable.
    iii. Use the @PathVariable(“id) annotation to bind the above URL template variable to method argument.
    iv. Use @RequestBody annotation to the int movieId argument to extract the updated JSON from the request and it will convert into MovieDTO JPA entity.

30. Build Delete Movie Rest API

a. Create a void method called deleteMovie with a int id parameter.
b. Implements the above created method in MovieServiceImpl class.
c. Navigate to controller and implement the necessary logic to deleteMovie method.
d. Use @DeleteMapping and @PathVariable annotations.

```
@DeleteMapping("{id}")
public ResponseEntity<String> deleteMovie(@PathVariable("id") int
movieId){
    movieService.deleteMovie(movieId);
    return ResponseEntity.ok("Movie Deleted with the Id: " + movieId);
}
```

e. Notice here the ResponseEntity is String because when deleting we don’t need a MovieDTO. Instead we simply need a message to display.

Order Service

1. Navigate to https://start.spring.io/
2. Select Java as the Language and Select Maven from the Project Menu
3. Add Project Metadata
    a. Group – com.adeesha
    b. Artifact – movie-service
    c. Name – movie-service
    d. Description – Movie Service for Movie Store
    e. Package Name – com.adeesha.movie-service
4. Add Required Dependencies.
    a. Spring Web – For rest APIs
    b. Spring-Reactive-Web – For spring Web-Flux
    c. Lombok – To reduce boilerplate codes.
5. Generate the File and Open the project via IntelliJ.
6. Navigate to resources and then edit the application properties.
    a. spring.application.name=shopping-service
    b. server.port = 8091
7. Create a new package called WebClient.
    a. Create a class called to WebClient and write the configurations.
8. Create a package called Service.
    a. Create an interface called OrderService
    b. Create a package called Impl.
        i. Create a Class called OrderServiceImpl inside the Impl Package.