System Requirements Specification

Index

For

Appointment Scheduler Application Beginner

Version 1.0

TABLE OF CONTENTS

APPOINTMENT SCHEDULER APPLICATION

System Requirements Specification

BACKEND-SPRING DATA RESTFUL APPLICATION

1 PROJECT ABSTRACT

The **Appointment Scheduler Application** is implemented using Spring Data with a MySQL database. This application is engineered to optimize the management of medical appointments, facilitating seamless interaction between patients and healthcare providers.

You are tasked with building a system that allows users to easily book and manage patients with healthcare providers. The application offers functionalities to create, update, and delete doctor profiles.

Following is the requirement specifications:

	Appointment Scheduler Application
Modules	
1	Doctor
Doctor Module Functionalities	
1	List all doctors **(must return doctors by name in ascending order and that also in pages)**
2	Get doctor by id
3	Create doctor **(must be transactional)**
4	Update doctor by id **(must be transactional)**
5	Delete doctor by id **(must be transactional)**
6	Get doctor by speciality **(must use dynamic method)**

2 Assumptions, Dependencies, Risks / Constraints

2.1 Doctor Constraints

- When fetching a doctor by ID, if the doctor ID does not exist, the service method should throw a NotFoundException with “Doctor not found” message.
- When updating a doctor, if the doctor ID does not exist, the service method should throw a NotFoundException with “Doctor not found” message.
- When removing a doctor, if the doctor ID does not exist, the service method should throw a NotFoundException with “Doctor not found” message.

Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exception if data is invalid
- All the business validations must be implemented in dto classes only.
- All the database operations must be implemented on entity object only
- Do not change, add, remove any existing methods in service layer
- In Repository interfaces, custom methods can be added as per requirements.
- All RestEndpoint methods and Exception Handlers must return data wrapped in **ResponseEntity.**

3 Business Validations

Doctor:

- Name should not be blank.
- Hospital name should not be blank.
- Speciality should not be blank.
- DailyTime should not be null.
- Similarly all validations must be applied in the constructor also.

4 Database Operations

Doctor:

- Class must be treated as an entity.
- The table must have a name as “doctors” in the database.
- Id must be of type id and generated by IDENTITY technique.
- Name must not be null.

- hospitalName should have a column name as “hospital_name” and must not be null.
- Speciality must not be null.
- dailyTime should have a column name as “daily_time” and must not be null.

5 REST ENDPOINTS

Rest End-points to be exposed in the controller along with method details for the same to be created.

5.1 DOCTORCONTROLLER

URL Exposed		Purpose
1. /api/doctors		Fetches all the doctors
Http Method	GET	
Parameter	-	
Return	Page<DoctorDTO>	
2. /api/doctors/{id}		Get a doctor by id
Http Method	GET	
Parameter 1	Long (id)	
Return	DoctorDTO	
3. /api/doctors		Create a new doctor
Http Method	POST **The doctor data to be created must be received in the controller using @RequestBody.**	
Parameter	-	
Return	DoctorDTO	
4. /api/doctors/{id}		Updates existing doctor by id
Http Method	PUT **The doctor data to be updated must be received in the controller using @RequestBody.**	
Parameter 1	Long (id)	
Return	DoctorDTO	
5. /api/doctors/{id}		
Http Method	DELETE	

Parameter 1	Long (id)
Return	-

Deletes a doctor by id

6. /api/doctors/specialty/{specialty}	
Http Method	GET
Parameter 1	String (specialty)
Return	List<DoctorDTO>

Fetches all doctor with given specialty

6 TEMPLATE CODE STRUCTURE

6.1 PACKAGE: COM.APPOINTMENT

Resources

AppointmentSchedulerApplication (Class)	This is the Spring Boot starter class of the application.	Already Implemented

6.2 PACKAGE: COM.APPOINTMENT.REPOSITORY

Resources

Class/Interface	Description	Status
DoctorRepository (interface)	● Repository interface exposing CRUD functionality for Doctor entity. ● It must contain the methods for: ○ Finding a list of doctors by their speciality and ordered by name in ascending order. ○ Finding all doctors ordered by name in pages. ● You can go ahead and add any custom methods as per requirements.	Partially implemented.

6.3 PACKAGE: COM.APPOINTMENT.SERVICE

Resources

Class/Interface	Description	Status
DoctorService (interface)	• Interface to expose method signatures for doctor related functionality. • Do not modify, add or delete any method.	Already implemented.

6.4 PACKAGE: COM.APPOINTMENT.SERVICE.IMPL

Class/Interface	Description	Status
DoctorServiceImpl (class)	• Implements DoctorService. • Contains template method implementation. • Need to provide implementation for doctor related functionalities. • Do not modify, add or delete any method signature.	To be implemented.

6.5 PACKAGE: COM.APPOINTMENT.CONTROLLER

Resources

Class/Interface	Description	Status
DoctorController (Class)	• Controller class to expose all rest-endpoints for doctor related activities. • May also contain local exception handler methods.	To be implemented

6.6 PACKAGE: COM.APPOINTMENT.DTO

Resources

Class/Interface	Description	Status
DoctorDTO (Class)	Use appropriate annotations for validating attributes of this class.	Partially implemented.

6.7 PACKAGE: COM.APPOINTMENT.ENTITY

Resources

Class/Interface	Description	Status
Doctor (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **id** as primary key. • Map this class with a **doctor table**. • Generate the **id** using the IDENTITY strategy	Partially implemented.

6.8 PACKAGE: COM.APPOINTMENT.EXCEPTION

Resources

Class/Interface	Description	Status
NotFoundException (Class)	• Custom Exception to be thrown when trying to fetch, update or delete the doctor or schedule info which does not exist. • Need to create Exception Handler for same wherever needed (local or global)	Already implemented.

ErrorResponse (Class)	• RestControllerAdvice Class for defining global exception handlers. • Contains Exception Handler for **InvalidDataException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.
RestExceptionHandler (Class)	• RestControllerAdvice Class for defining rest exception handlers. • Contains Exception Handler for **NotFoundException** class. • Use this as a reference for creating exception handler for other custom exception classes	Already implemented.

6 METHOD DESCRIPTIONS

1. ServiceImpl Class - Method Descriptions

a. DoctorServiceImpl – Method Descriptions

- Declare a private final variable with name `doctorRepository` of type `DoctorRepository` interface.

Method	Task	Implementation Details
`@Autowired public DoctorServiceImpl(DoctorRepository`	Constructor-based dependency injection	- Annotated with `@Autowired`. - Injects the repository dependency through constructor.

`doctorRepository)`		- Assigns to the `doctorRepository` field.

Method	Task	Implementation Details
`createDoctor`	To implement logic for saving a new doctor	- Convert the incoming DoctorDTO to a Doctor entity using convertToEntity(). - Call doctorRepository.save(doctor) to store the doctor. - Convert the saved entity back to DoctorDTO using convertToDTO() and return it.
`updateDoctor`	To implement logic for updating doctor details by ID	- Call doctorRepository.findById(doctorId) to find the doctor. - If not found, throw NotFoundException with message "Doctor not found". - Update fields: name, hospital name, specialty, and daily time using setters. - Save updated doctor using doctorRepository.save(doctor). - Return updated DoctorDTO using convertToDTO().
`getDoctorById`	To implement logic for retrieving a doctor by ID	- Call doctorRepository.findById(doctorId). - If not found, throw NotFoundException with message "Doctor not found". - If found, convert to DoctorDTO using convertToDTO() and return it.
`deleteDoctor`	To implement logic to delete a doctor by ID	- Call doctorRepository.findById(doctorId). - If not found, throw NotFoundException with message "Doctor not found". - If found, delete using doctorRepository.deleteById(doctorId). - Return true after successful deletion.
`getAllDoctors`	To implement logic to retrieve all doctors with pagination	- Accept a Pageable object as input. - Call doctorRepository.findAllByOrderByNameAsc(pageable). - Map each Doctor to DoctorDTO using map(this::convertToDTO). - Return a Page<DoctorDTO> object.
`findDoctorsBySpecialty`	To implement logic to get doctors based on specialty	- Accept a specialty string as input. - Call doctorRepository.findBySpecialtyOrderByNameAsc(specialty). - Convert each Doctor to DoctorDTO using stream().map(this::convertToDTO). - Return the list of DoctorDTO objects.

2. Controller Class - Method Descriptions

a. DoctorController – Method Descriptions

- Declare a private final variable with name `doctorService` of type `DoctorService` interface.

Method	Task	Implementation Details
`@Autowired public DoctorController (DoctorService doctorService)`	Constructor-based dependency injection	- Annotated with `@Autowired`. - Injects the repository dependency through constructor. - Assigns to the `doctorService` field.

Method	Task	Implementation Details
`getAllDoctors`	To implement logic to fetch all doctors with pagination	- Request type: GET, URL: /api/doctors - Method name: getAllDoctors, returns ResponseEntity<Page<DoctorDTO>> - Accept @RequestParam for 'page' and 'size' - Create Pageable object using PageRequest.of(page, size) - Call doctorService.getAllDoctors(pageable) - Return the result with HttpStatus.OK
`getDoctorById`	To implement logic to fetch a doctor by ID	- Request type: GET, URL: /api/doctors/{id} - Method name: getDoctorById, returns ResponseEntity<DoctorDTO> - Use @PathVariable for doctor ID - Call doctorService.getDoctorById(id) - Return the doctor with HttpStatus.OK - If NotFoundException is thrown, return HttpStatus.NO_CONTENT
`createDoctor`	To implement logic to create a new doctor	- Request type: POST, URL: /api/doctors - Method name: createDoctor, returns ResponseEntity<DoctorDTO> - Use @Validated @RequestBody to accept DoctorDTO - Call doctorService.createDoctor(doctorDTO) - Return the created doctor with HttpStatus.CREATED

`updateDoctor`	To implement logic to update a doctor by ID	- Request type: PUT, URL: /api/doctors/{id} - Method name: updateDoctor, returns ResponseEntity<DoctorDTO> - Use @PathVariable for ID and @Validated @RequestBody for DoctorDTO - Call doctorService.updateDoctor(id, doctorDTO) - Return updated doctor with HttpStatus.OK - If NotFoundException is thrown, return HttpStatus.NO_CONTENT
`deleteDoctor`	To implement logic to delete a doctor by ID	- Request type: DELETE, URL: /api/doctors/{id} - Method name: deleteDoctor, returns ResponseEntity<Void> - Use @PathVariable to accept doctor ID - Call doctorService.deleteDoctor(id) - Return response with HttpStatus.NO_CONTENT
`getDoctorsBySpecialty`	To implement logic to fetch doctors based on specialty	- Request type: GET, URL: /api/doctors/specialty/{specialty} - Method name: getDoctorsBySpecialty, returns ResponseEntity<List<DoctorDTO>> - Use @PathVariable to accept the specialty value - Call doctorService.findDoctorsBySpecialty(specialty) - Return the filtered list with HttpStatus.OK

7 EXECUTION STEPS TO FOLLOW FOR BACKEND

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers need to go to the Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal.**
3. **cd into your backend project folder**
4. **To build your project use command:**

   mvn clean package -Dmaven.test.skip
5. **To launch your application, move into the target folder (cd target). Run the following command to run the application:**

   java -jar <your application jar file name>
6. **This editor Auto Saves the code.**
7. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was**

stopped from the previous logout.

8. To test any Restful application, the last option on the left panel of IDE, you can find ThunderClient, which is the lightweight equivalent of POSTMAN. Please use 127.0.0.1 instead of localhost to test rest endpoints.
9. To test any UI based application the second last option on the left panel of IDE, you can find Browser Preview, where you can launch the application.
10. Default credentials for MySQL:
    a. Username: root
    b. Password: pass@word1

11. To login to mysql instance: Open new terminal and use following command:
    a. sudo systemctl enable mysql
    b. sudo systemctl start mysql

    NOTE: After typing any of the above commands you might encounter any warnings.

    >> Please note that this warning is expected and can be disregarded. Proceed to the next step.

    c. mysql -u root -p

    The last command will ask for password which is 'pass@word1'
12. Mandatory: Before final submission run the following command:

    mvn test