System Requirements Specification

Index

For

Health Care System

Version 1.0

TABLE OF CONTENTS

HEALTH CARE SYSTEM

System Requirements Specification

BACKEND-SPRING BOOT RESTFUL APPLICATION

1 PROJECT ABSTRACT

The **Health Care System Application** is implemented using Spring Boot with a MySQL database, designed using state-of-the-art technology frameworks, aimed at enhancing the healthcare management experience. This app ensures comprehensive management of user profiles, doctor engagements, and patient records, thereby streamlining the interactions between patients, doctors, and administrative staff.

You are tasked with developing a medical platform where users can seamlessly register, access, and update personal and medical details. The system should support functionalities such as managing doctor profiles, creating and maintaining patient records, scheduling and managing appointments, and providing the capability to search and filter through records and appointments efficiently. Ensure that all operations within the system are secure, maintain accuracy and consistency, and offer real-time updates to enhance user engagement and operational efficiency.

Following is the requirement specifications:

	Health Care System
Modules	
1	User
2	Doctor
3	Patient Record
4	Appointment
User Module Functionalities	
1	Register a user
2	Get user details by id
3	Update an user by id
4	Delete an user by id
5	Get all users
6	Search users by their username

Doctor Module Functionalities	
1	Create doctor
2	Get doctor by id
3	Get all doctors

Patient Record Module Functionalities	
1	Create patient record
2	Get patient records for specific user by user id
3	Update patient record
4	Delete patient record
5	Get patient record details
6	Get all patient records
7	Search patient records by diagnosis
8	Get patient records by doctor
9	Flag patient record for review

Appointment Module Functionalities	
1	Create appointment
2	Get user appointments
3	Update appointment
4	Cancel appointment
5	Get appointment details
6	Reschedule appointment
7	Check appointment status
8	Get appointments by date
9	Get appointments by doctor
10	Mark appointment as completed
11	Get appointment history for user

2 Assumptions, Dependencies, Risks / Constraints

2.1 User Constraints

- When updating a user's profile, if the user ID does not exist, the service method should throw a NotFoundException with the message "User not found with id [userId]".
- When fetching user details by ID, if the user ID does not exist, the service method should throw a NotFoundException with the message "User not found with id [userId]".

2.2 Doctor Constraints

- When fetching a doctor by ID, if the doctor ID does not exist, the service method should throw a NotFoundException with the message "Doctor not found with id [doctorId]".

2.3 PatientRecord Constraints

- When updating a patient record, if the record ID does not exist, the service method should throw a NotFoundException with the message "Patient record not found with id [recordId]".
- When fetching patient record details by ID, if the patient record ID does not exist, the service method should throw a NotFoundException with the message "Patient record not found with id [recordId]".
- When flagging a patient record for review, if the record ID does not exist, the service method should throw a NotFoundException with the message "Patient record not found with id [recordId]".

2.4 Appointment Constraints

- When updating an appointment, if the appointment ID does not exist, the service method should throw a NotFoundException with the message "Appointment not found with id [appointmentId]".
- When fetching appointment details by ID, if the appointment ID does not exist, the service method should throw a NotFoundException with the message "Appointment not found with id [appointmentId]".
- When rescheduling an appointment, if the appointment ID does not exist, the service method should throw a NotFoundException with the message "Appointment not found with id [appointmentId]".
- When checking the status of an appointment, if the appointment ID does not exist, the service method should throw a NotFoundException with the message "Appointment not found with id [appointmentId]".
- When marking an appointment as completed, if the appointment ID does not exist, the service method should throw a NotFoundException with the message "Appointment not found with id [appointmentId]".

Common Constraints

- For all rest endpoints receiving @RequestBody, validation check must be done and must throw custom exceptions 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 the 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

3.1 User

- Id must be of type id.
- Username should not be blank, min 4 and max 20 characters and unique in the system.
- Password should not be blank, min 6 and max 100 characters.
- Email should not be blank and must be of type email.
- First name should not be blank.
- Last name should not be blank.

3.2 Doctor

- Id must be of type id.
- Name should not be blank.
- Speciality should not be blank.

3.3 PatientRecord

- Id must be of type id.
- User ID should not be null.
- Doctor ID should not be null.
- Date should not be null.
- Diagnosis should not be blank.
- Treatment should not be blank.
- flaggedForReview should not be null.

3.4 Appointment

- Id must be of type id.
- User ID should not be null.
- Doctor ID should not be null.
- appointmentTime should not be null.
- Status should not be blank.

4 REST ENDPOINTS

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

4.1 USERCONTROLLER

URL Exposed		Purpose
1. /api/users/{userId}		Retrieves details of a user by their ID
Http Method	GET	
Parameter 1	Long (userId)	
Return	UserDTO	
2. /api/users		Register a new user
Http Method	POST **The user data to be created must be received in the controller using @RequestBody.**	
Parameter	-	
Return	UserDTO	
3. /api/users/{userId}		Updates the profile details of an existing user
Http Method	PUT **The user data to be updated must be received in the controller using @RequestBody.**	
Parameter 1	Long (userId)	
Return	UserDTO	
4. /api/users/{userId}		Deletes a user from the system by their ID
Http Method	DELETE	
Parameter 1	Long (userId)	
Return	-	
5. /api/users		Retrieves a list of all registered users
Http Method	GET	
Parameter 1	-	
Return	List<UserDTO>	

URL Exposed	Purpose
6. /api/users/search Http Method: GET Request Parameter: String (query) Return: List<UserDTO>	Searches for users based on username

4.2 DoctorController

URL Exposed	Purpose
1. /api/doctors Http Method: POST **The doctor data to be created must be received in the controller using @RequestBody.** Parameter 1: - Return: DoctorDTO	Adds a new doctor to the system
2. /api/doctors/{doctorId} Http Method: GET Parameter 1: Long (doctorId) Return: DoctorDTO	Retrieves details of a specific doctor by their ID
3. /api/doctors Http Method: GET Parameter 1: - Return: List<DoctorDTO>	Retrieves a list of all doctors

4.3 PatientRecordController

URL Exposed	Purpose
1. /api/patient-records/user/{userId} Http Method: GET Parameter 1: Long (userId) Return: List<PatientRecordDTO>	Retrieves all patient records for a specific user
2. /api/patient-records Http Method: POST **The patient-record data to be created must be received in the**	Adds a new patient record to the system

	controller using @RequestBody.	
Parameter 1	-	
Return	PatientRecordDTO	

3. /api/patient-records/{recordId}		
Http Method	GET	Retrieves detailed information about a specific patient record
Parameter 1	Long (recordId)	
Return	PatientRecordDTO	

4. /api/patient-records/{recordId}		
Http Method	PUT **The patient-record data to be updated must be received in the controller using @RequestBody.**	Updates an existing patient record
Parameter 1	Long (recordId)	
Return	PatientRecordDTO	

5. /api/patient-records/{recordId}		
Http Method	DELETE	Deletes a specific patient record
Parameter 1	Long (recordId)	
Return	-	

6. /api/patient-records		
Http Method	GET	Retrieves a list of all patient records in the system
Parameter 1	-	
Return	List<PatientRecordDTO>	

7. /api/patient-records/search		
Http Method	GET	Searches patient records based on a diagnosis
Request Parameter	String (query)	
Return	List<PatientRecordDTO>	

8. /api/patient-records/doctor/{doctorId}		
Http Method	GET	Retrieves all patient records associated with a specific doctor
Parameter 1	Long (doctorId)	
Return	List<PatientRecordDTO>	

9. /api/patient-records/flag/{recordId}		
Http Method	PUT	Flags a patient record for further review
Parameter 1	Long (recordId)	

Return	PatientRecordDTO	

4.4 APPOINTMENTCONTROLLER

URL Exposed		Purpose
1. /api/appointments/user/{userId}		Retrieves all appointments for a specific user
Http Method	GET	
Parameter 1	Long (userId)	
Return	List<AppointmentDTO>	
2. /api/appointments		Schedules a new appointment
Http Method	POST **The appointment data to be created must be received in the controller using @RequestBody.**	
Parameter 1	-	
Return	AppointmentDTO	
3. /api/appointments/{appointmentId}		Updates details of an existing appointment
Http Method	PUT **The appointment data to be updated must be received in the controller using @RequestBody.**	
Parameter 1	Long (appointmentId)	
Return	AppointmentDTO	
4. /api/appointments/{appointmentId}		Cancels a specific appointment
Http Method	DELETE	
Parameter 1	Long (appointmentId)	
Return	-	
5. /api/appointments/{appointmentId}		Retrieves details of a specific appointment
Http Method	GET	
Parameter 1	Long (appointmentId)	
Return	AppointmentDTO	

6. /api/appointments/reschedule/{appointmentId}		Reschedules an existing appointment
Http Method	PUT **The appointment data to be updated must be received in the controller using @RequestBody.**	
Parameter 1	Long (appointmentId)	
Return	AppointmentDTO	

7. /api/appointments/status/{appointmentId}		Checks the status of a specific appointment
Http Method	GET	
Parameter 1	Long (appointmentId)	
Return	String	

8. /api/appointments/date		Retrieves appointments scheduled for a specific date
Http Method	GET	
Request Parameter	LocalDate (date)	
Return	List<AppointmentDTO>	

9. /api/appointments/doctor/{doctorId}		Retrieves all appointments associated with a specific doctor
Http Method	GET	
Parameter 1	Long (doctorId)	
Return	List<AppointmentDTO>	

10. /api/appointments/complete/{appointmentId}		Marks an appointment as completed
Http Method	PUT	
Parameter 1	Long (appointmentId)	
Return	AppointmentDTO	

11. /api/appointments/history/user/{userId}		Retrieves historical appointment data for a user
Http Method	GET	
Parameter 1	Long (userId)	
Return	List<AppointmentDTO>	

5 Template Code Structure

5.1 Package: com.healthcare

Resources

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

5.2 Package: com.healthcare.repository

Resources

Class/Interface	Description	Status

UserRepository (interface)	• Repository interface exposing CRUD functionality for User Entity.
	• It must contain the methods for:
	  ○ Finding all users by their username.
	  ○ Finding all users by email.
	• You can go ahead and add any custom methods as per requirements.
		Partially implemented.

DoctorRepository (interface)	• Repository interface exposing CRUD functionality for Doctor Entity.
	• It must contain the methods for:
	  ○ Finding all doctors by speciality.
	  ○ Finding all doctors by name.
	• You can go ahead and add any custom methods as per requirements.
		Partially implemented.

PatientRecordRepository (interface)	• Repository interface exposing CRUD functionality for PatientRecord Entity.
	• It must contain the methods for:
	  ○ Finding all patient records by flagged records by user.
	  ○ Finding all patient records by diagnosis.
	• You can go ahead and add any custom methods as per requirements.
		Partially implemented.

AppointmentRepository (interface)	• Repository interface exposing CRUD functionality for Appointment Entity.
	• It must contain the methods for:
	  ○ Finding all appointments by doctor and date in range.
	  ○ Finding all appointments by date range.
	  ○ Finding all appointments by user id.
	• You can go ahead and add any custom methods as per requirements.
		Partially implemented.

5.3 PACKAGE: COM.HEALTHCARE.SERVICE

Resources

Class/Interface	Description	Status
UserService (interface)	• Interface to expose method signatures for user related functionality. • Do not modify, add or delete any method.	Already implemented.
DoctorService (interface)	• Interface to expose method signatures for doctor related functionality. • Do not modify, add or delete any method.	Already implemented.
PatientRecordService (interface)	• Interface to expose method signatures for patient-record related functionality. • Do not modify, add or delete any method.	Already implemented.
AppointmentService (interface)	• Interface to expose method signatures for appointment related functionality. • Do not modify, add or delete any method.	Already implemented.

5.4 PACKAGE: COM.HEALTHCARE.SERVICE.IMPL

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

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.
PatientRecordServiceImpl (class)	• Implements PatientRecordService. • Contains template method implementation. • Need to provide implementation for patient-record related functionalities. • Do not modify, add or delete any method signature	To be implemented.
AppointmentServiceImpl (class)	• Implements AppointmentService. • Contains template method implementation. • Need to provide implementation for appointment related functionalities. • Do not modify, add or delete any method signature	To be implemented.

5.5 PACKAGE: COM.HEALTHCARE.CONTROLLER

Resources

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

5.6 PACKAGE: COM.HEALTHCARE.DTO

Resources

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

5.7 PACKAGE: COM.HEALTHCARE.ENTITY

Resources

Class/Interface	Description	Status
User (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 **users table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.

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 **doctors table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.
PatientRecord (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 **patient_records table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.
Appointment (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 an **appointments table**. • Generate the **id** using the IDENTITY strategy.	Partially implemented.

5.8 PACKAGE: COM.HEALTHCARE.EXCEPTION

Resources

Class/Interface	Description	Status
NotFoundException (Class)	● Custom Exception to be thrown when trying to fetch or delete the user/ doctor/patient-record/appointment info which does not exist. ● Need to create Exception Handler for the 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 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. **If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) then you need to use CTRL+Shift+B-command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.**
8. **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.**
9. **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.**
10. **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.**
11. **Default credentials for MySQL:**
    a. **Username: root**
    b. **Password: pass@word1**

12. 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'

13. Mandatory: Before final submission run the following command:

    mvn test

14. You need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.