System Requirements Specification

Index

For

HealthCare Information System

Version 1.0

TABLE OF CONTENTS

HealthCare Information System

System Requirements Specification

You need to consume APIs exposed by Backend application in Angular to make application work as FULLSTACK

BACKEND-SPRING BOOT RESTFUL APPLICATION

1 PROJECT ABSTRACT

HealthCare Information System is Spring boot RESTful application with MySQL, where it manages hospitals, doctors and patient's profiles.

Following is the requirement specifications:

	HealthCare Information System
Modules	
1	Hospital
2	Doctor
3	Patient
Hospital Module Functionalities	
1	Create a Hospital
2	Update the existing hospital details
3	Get hospital info by hospital id
4	Get all registered hospitals
5	Search hospital Info by hospital name
6	Delete an existing hospital
Doctor Module Functionalities	
1	Create a doctor
2	Update the existing doctor
3	Get a doctor by Id
4	Fetch all registered doctors
5	Delete an existing doctor
6	Search doctor Info by doctor name
7	Get a doctor by hospital Id
8	Search doctor by hospital name

Patient Module Functionalities	
1	Create a patient
2	Update the existing patient
3	Get a patient by Id
4	Get a patient by doctor Id
5	Fetch all registered patients
6	Delete an existing patient
7	Search patientInfo by patient name
8	Search patientInfo by doctor id

2 Assumptions, Dependencies, Risks / Constraints

2.1 HOSPITAL CONSTRAINTS:

- While updating a hospital, if hospitalId does not exist then the operation should throw a ResourceNotFoundException("Hosptal not found with id: " + id).
- While fetching the hospital details by id, if hospitalId does not exist then the operation should throw a HospitalNotFoundException("Hospital with Id - " + id + " not found!").
- While deleting the hospital details by id, if hospitalId does not exist then the operation should throw a HospitalNotFoundException("Hospital with Id - " + id + " not found!").

2.2 DOCTOR CONSTRAINTS

- While updating a doctor, if doctorId does not exist then the operation should throw a DoctorNotFoundException("Doctor with Id - " + id + " not found!").
- While fetching the doctor details by id, if doctorId does not exist then the operation should throw a DoctorNotFoundException("Doctor with Id - " + id + " not found!").

2.3 PATIENT CONSTRAINTS

- While updating a patient, if patientId does not exist then the operation should throw a ResourceNotFoundException("Patient not found with id: " + id).
- While fetching the patient details by id, if patientId does not exist then the operation should throw a ResponseStatusException(HttpStatus.NOT_FOUND, "Patient with Id - " + id + " not found!").

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 onlyDo 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

3.1 Hospital DTO

- Hospital name should not be blank, min 3 and max 100 characters.

3.2 Doctor DTO

- Doctor name should not be empty, min 3 and max 100 characters.
- Hospital id should not be null.

3.3 Patient DTO

- Patient name should not be empty, min 3 and max 100 characters.
- doctor id should not be null.

4 Database Operations

4.1 Doctor Entity

- **Doctor** class should be bound to a table named **`doctors`**.
- `id` should be the **primary key**, generated with **IDENTITY**, mapped to column **`id`**.
- `name` should be mapped to column **`name`**.
- `hospital` should be a **many-to-one** relationship to **Hospital** with **`fetch = FetchType.EAGER`** specified to always load the associated Hospital when fetching a Doctor, and joined via column **`hospital_id`** on the **doctors** table (FK to `hospitals.id`).

4.2 Hospital Entity

- **Hospital** class should be bound to a table named **`hospitals`**.
- `id` should be the **primary key**, generated with **IDENTITY**, mapped to column **`id`**.
- `name` should be mapped to column **`name`**.

4.3 Patient Entity

- **Patient** class should be bound to a table named **`patients`**.
- `id` should be the **primary key**, generated with **IDENTITY**, mapped to column **`id`**.
- `name` should be mapped to column **`name`**.
- `doctor` should be a **many-to-one** relationship to **Doctor** with **`fetch = FetchType.EAGER`** specified to always load the associated Doctor when fetching a Patient, and joined via column **`doctor_id`** on the **patients** table (FK to `doctors.id`).

5 REST ENDPOINTS

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

5.1 HOSPITAL CONTROLLER

URL Exposed		Purpose
1. /hospitals		Create a Hospital
Http Method	POST	
Parameter 1	HospitalDTO	
Return	HospitalDTO	
2. /hospitals/{id}		Update the Hospital
Http Method	PUT	
Parameter 1	Long (id)	
Parameter 2	HospitalDTO	
Return	HospitalDTO	
3. /hospitals/{id}		Fetches the details of Hospital by Id
Http Method	GET	
Parameter 1	Long(id)	
Return	ResponseEntity<HospitalDTO>	
4. /hospitals/{id}		Delete the Hospital
Http Method	DELETE	
Parameter 1	Long (id)	
Return	Boolean	
5. /hospitals/		Fetch all registered Hospitals
Http Method	GET	
Parameter	-	
Return	List<HospitalDTO>	
6. /hospitals/search?name={name}		Fetches the Hospital with the given name
Http Method	GET	

Parameter 1	String (name)	
Return	List<HospitalDTO>	

5.2 DOCTOR CONTROLLER

URL Exposed		Purpose
1. /doctors/		Create a Doctor
Http Method	POST	
Parameter 1	DoctorDTO	
Return	DoctorDTO	
2. /doctors/{id}		Update the Doctor
Http Method	PUT	
Parameter 1	Long (id)	
Parameter 2	DoctorDTO	
Return	DoctorDTO	
3. /doctors/{id}		Fetches the details of Doctor by Id
Http Method	GET	
Parameter 1	Long(id)	
Return	DoctorDTO	
4. /doctors/{id}		Delete the Doctor
Http Method	DELETE	
Parameter 1	Long (id)	
Return	Boolean	
5. /doctors/		Fetch all registered Doctors
Http Method	GET	
Parameter 1	-	
Return	List<DoctorDTO>	
6. /doctors/search?name={name}		Fetches the Doctor with the given name
Http Method	GET	
Parameter 1	String (name)	
Return	List<DoctorDTO>	
7. /doctors/hospital/{hospitalid}		Fetches the Doctor with the given hospital id
Http Method	GET	
Parameter 1	Long (id)	
Return	List<DoctorDTO>	
8. /doctors/searchByHospital?name={name}		Fetches the Doctor with the given hospital name
Http Method	GET	
Parameter 1	Long (id)	
Parameter 2	String (name)	
Return	List<DoctorDTO>	

5.3 PATIENT CONTROLLER

URL Exposed		Purpose
1. /patients/		Create a Patient
Http Method	POST	
Parameter 1	PatientDTO	
Return	PatientDTO	
2. /patients/{id}		Update the Patient
Http Method	PUT	
Parameter 1	Long (id)	
Parameter 2	PatientDTO	
Return	PatientDTO	
3. /patients/{id}		Fetches the details of Patient by Id
Http Method	GET	
Parameter 1	Long(id)	
Return	PatientDTO	
4. /patients/{id}		Delete the Patient
Http Method	DELETE	
Parameter 1	Long (id)	
Return	Boolean	
5. /patients/		Fetch all registered Patients
Http Method	GET	
Parameter 1	-	
Return	List<PatientDTO>	
6. /patients/search?name={name}		Fetches the Patient with the given name
Http Method	GET	
Parameter 1	String (name)	
Return	List<PatientDTO>	
7. /patients/doctor/{doctorid}		Fetches the patient with the given doctor id
Http Method	GET	
Parameter 1	Long (id)	
Return	List<PatientDTO>	
8. /patients/searchByDoctor?id={id}		Search the patient with the given doctor id
Http Method	GET	
Parameter 1	Long (id)	
Return	List<PatientDTO>	

6 TEMPLATE CODE STRUCTURE

6.1 PACKAGE: COM.HEALTHCARE

Resources

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

6.2 PACKAGE: COM.HEALTHCARE.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 **doctorId** as primary key. • Map this class with a **doctors table**. • Generate the **doctorId** using the IDENTITY strategy.	Partially implemented.
Hospital(Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **hospitalId** as primary key. • Map this class with a **hospitals table**. • Generate the **hospitalId** using the IDENTITY strategy.	Partially implemented.

Patient (Class)	• This class is partially implemented. • Annotate this class with proper annotation to declare it as an entity class with **patientId** as primary key. • Map this class with a **patients table**. • Generate the **patientId** using the IDENTITY strategy.	Partially implemented.

6.3 PACKAGE: COM.HEALTHCARE.DTO

Resources

Class/Interface	Description	Status
DoctorDTO (class)	Use appropriate annotations for validating attributes of this class. (Refer **Business Validation** section for validation rules).	Partially implemented.
HospitalDTO (class)	Use appropriate annotations for validating attributes of this class. (Refer **Business Validation** section for validation rules).	Partially implemented.
PatientDTO (class)	Use appropriate annotations for validating attributes of this class. (Refer **Business Validation** section for validation rules).	Partially implemented.

6.4 PACKAGE: COM.HEALTHCARE.REPOSITORY

Resources

Class/Interface	Description	Status
DoctorDAO (interface)	1. Repository interface exposing CRUD functionality for **Doctor** Entity. 2. You can go ahead and add any custom methods as per requirements.	Partially implemented
HospitalDAO (interface)	1. Repository interface exposing CRUD functionality for **Hospital** Entity. 2. You can go ahead and add any custom methods as per requirements.	Partially implemented
PatienDAO (interface)	1. Repository interface exposing CRUD functionality for **Patient** Entity. 2. You can go ahead and add any custom methods as per requirements.	Partially implemented

6.5 PACKAGE: COM.HEALTHCARE.SERVICE.IMPL

Resources

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.
HospitalServiceImpl (class)	• Implements HospitalService. • Contains template method implementation. • Need to provide implementation for hospital related functionalities. • Do not modify, add or delete any method signature	To be implemented.
PatientServiceImpl (class)	• Implements PatientService. • Contains template method implementation. • Need to provide implementation for patient related functionalities. • Do not modify, add or delete any method signature	To be implemented.

6.6 PACKAGE: COM.HEALTHCARE.SERVICE

Resources

Class/Interface	Description	Status
DoctorService(interface)	• Need to provide implementation for Doctor related functionalities. • Add required repository dependency. • Do not modify, add or delete any method signature.	Already implemented.
HospitalService (interface)	• Need to provide implementation for Hospital related functionalities • Add required repository dependency • Do not modify, add or delete any method signature.	Already implemented.
PatientService (class)	• Need to provide implementation for Patient related functionalities • Add required repository dependency • Do not modify, add or delete any method signature	Already implemented.

6.7 PACKAGE: COM.HEALTHCARE.EXCEPTION

Resources

Class/Interface	Description	Status
GlobalExceptionHandler (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.	Partially implemented.
ResourceNotFound Exception (Class)	• Custom Exception to be thrown when trying to fetch or delete the Hospital info which does not exist. • Need to create Exception Handler for same wherever needed (local or global).	Already created.
DoctorNotFound Exception (Class)	• Custom Exception to be thrown when trying to fetch or delete a doctor info which does not exist. • Need to create Exception Handler for same wherever needed (local or global)	Already created.

HospitalNotFoundException (Class)	• Custom Exception to be thrown when trying to fetch or delete a hospital info which does not exist. • Need to create Exception Handler for same wherever needed (local or global)	Already created.
PatientNotFoundException (Class)	• Custom Exception to be thrown when trying to fetch or delete a patient info which does not exist. • Need to create Exception Handler for same wherever needed (local or global)	Already created.
ErrorResponse (Class)	• Object of this class is supposed to be returned in case of exception through exception handlers	Already created.

6.8 PACKAGE: COM.HEALTHCARE.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
HospitalController (Class)	• Controller class to expose all rest-endpoints for Hospital related activities. • May also contain local exception handler methods.	To be implemented
PatientController (Class)	• Controller class to expose all rest-endpoints for Patient related activities. • May also contain local exception handler methods.	To be implemented

7 METHOD DESCRIPTIONS

1. Service Class - Method Descriptions

A. DoctorServiceImpl – Method Descriptions

Method	Task	Implementation Details
`getAllDoctors()`	Fetches all doctor records	- Calls `doctorDAO.findAll()` - Maps the result to a list of DTOs using `modelMapper` - Returns the list of mapped DTOs
`getDoctorById()`	Fetches doctor by ID	- Uses `doctorDAO.findById(id)` - Throws `DoctorNotFoundException` if a doctor is not found with the message: Doctor with Id - {id} not found! - Maps entity to DTO using `modelMapper` - Returns the DTO

`updateDoctor()`	Updates doctor by ID	- Calls `getDoctorById(id)` internally - If doctor found, maps DTO to entity and saves using `doctorDAO.save()` - Returns the updated DTO - Throws `DoctorNotFoundException` if doctor is not found with the message: Doctor with Id - {id} not found!
`getDoctorsByHospitalId()`	Fetches doctors by hospital ID	- Uses `doctorDAO.findByHospitalId(hospitalId)` - Maps list to DTOs using `modelMapper` - Returns the list of DTOs
`saveDoctor()`	Saves a new doctor	- Maps `doctorDTO` to `Doctor` entity using `modelMapper` - Saves using `doctorDAO.save()` - Returns the saved doctor as DTO
`deleteDoctorById()`	Deletes doctor by ID	- Deletes the doctor using `doctorDAO.deleteById(id)` - Returns `true`
`searchDoctorsByName()`	Searches doctors by name	- Uses `doctorDAO.findByNameContainingIgnoreCase(name)` - Maps list to DTOs - Returns the list of DTOs
`searchDoctorsByHospitalIdOrName()`	Searches doctors by hospital ID or name	- Uses `doctorDAO.findByHospitalIdOrNameContainingIgnoreCase(hospitalId, name)` - Maps result to DTOs - Returns the list of DTOs

B. HospitalServiceImpl – Method Descriptions

Method	Task	Implementation Details
`getAllHospitals()`	Fetches all hospital records	- Calls `hospitalDAO.findAll()` - Maps list to DTOs using `modelMapper` - Returns the list of mapped DTOs
`getHospitalById()`	Fetches hospital by ID	- Uses `hospitalDAO.findById(id)` - Throws `HospitalNotFoundException` if a hospital is not found with the message: Hospital with Id - {id} not found! - Maps entity to DTO using `modelMapper` - Returns the DTO
`updateHospital()`	Updates hospital by ID	- Uses `hospitalDAO.findById(id)` - Throws `ResourceNotFoundException` if hospital is not found with the message: Hospital not found with id: {id} - Maps DTO to entity using `modelMapper`

		- Saves using `hospitalDAO.save()` - Returns the updated DTO
`saveHospital()`	Saves a new hospital	- Maps DTO to entity using `modelMapper` - Saves using `hospitalDAO.save()` - Returns the saved hospital as DTO
`deleteHospitalById()`	Deletes hospital by ID	- Checks existence using `hospitalDAO.existsById(id)` - Throws `HospitalNotFoundException` if not found with message: Hospital with Id - {id} not found! - Deletes hospital using `hospitalDAO.deleteById(id)` - Returns `true`
`searchHospitalsByName()`	Searches hospitals by name	- Uses `hospitalDAO.findByNameContainingIgnoreCase(name)` - Maps result to DTOs using `modelMapper` - Returns the list of mapped DTOs

C. PatientServiceImpl – Method Descriptions

Method	Task	Implementation Details
`getAllPatients()`	Fetches all patients	- Uses `patientDAO.findAll()` to retrieve all patients - Maps each entity to DTO using `modelMapper` - Returns the list of mapped DTOs
`getPatientById()`	Fetch a patient by ID	- Uses `patientDAO.findById(id)` - Throws `ResponseStatusException` if patient is not found with message: Patient with Id - {id} not found! - Maps entity to DTO using `modelMapper` and returns it
`getPatientsByDoctorId()`	Get patients under a doctor	- Uses `patientDAO.findByDoctorId(doctorId)` to fetch patients - Maps entities to DTOs using `modelMapper` - Returns the list of mapped DTOs
`updatePatient()`	Update existing patient details	- Uses `patientDAO.findById(id)` to check for existence - Throws `ResourceNotFoundException` if patient not found with message: Patient not found with id: {id} - Updates patient fields and saves - Maps the updated entity to DTO and returns it
`savePatient()`	Save a new patient	- Calls `doctorService.getDoctorById(patientDTO.getId())` to validate doctor - Maps DTO to entity using `modelMapper` - Saves the entity using `patientDAO.save()` - Maps the saved entity to DTO and returns it
`deletePatientById()`	Delete a patient by ID	- Uses `patientDAO.deleteById(id)` to delete the patient - Returns `true` upon successful deletion

`searchPatientsByName()`	Search patients by name	- Uses `patientDAO.findByNameContainingIgnoreCase(name)` to search - Maps results to DTOs using `modelMapper` - Returns the list of mapped DTOs
`searchPatientsByDoctorId()`	Search patients by doctor ID	- Uses `patientDAO.findByDoctorId(doctorId)` to search - Maps results to DTOs using `modelMapper` - Returns the list of mapped DTOs

2. Controller Class - Method Descriptions

A. DoctorController – Method Descriptions

Method	Task	Implementation Details
`getAllDoctors()`	Get all doctors	- Calls `doctorService.getAllDoctors()` - Returns the list of all `DoctorDTO`
`getDoctorById()`	Get doctor by ID	- Uses `PathVariable Long id` - Calls `doctorService.getDoctorById(id)` - Returns `DoctorDTO` for the given ID
`getDoctorsByHospitalId()`	Get doctors by hospital ID	- Uses `PathVariable Long hospitalId` - Calls `doctorService.getDoctorsByHospitalId(hospitalId)` - Returns list of doctors for the given hospital
`saveDoctor()`	Create a doctor	- Uses `RequestBody DoctorDTO` - Validates with `@Valid` - Calls `doctorService.saveDoctor(doctor)` - Returns the saved `DoctorDTO`
`deleteDoctorById()`	Delete doctor by ID	- Uses `PathVariable Long id` - Calls `doctorService.deleteDoctorById(id)` - Returns a Boolean status of deletion
`searchDoctorsByName()`	Search doctors by name	- Uses `RequestParam String name` - Calls `doctorService.searchDoctorsByName(name)` - Returns list of matching `DoctorDTO`
`updateDoctor()`	Update doctor by ID	- Uses `PathVariable Long id` and `RequestBody DoctorDTO` - Validates with `@Valid` - Calls `doctorService.updateDoctor(id, updatedDoctor)` - Returns updated `DoctorDTO`

`searchDoctorsByHospitalIdAndName()`	Search doctors by hospital ID and name	- Uses `RequestParam Long hospitalId` and `RequestParam String name` - Calls `doctorService.searchDoctorsByHospitalIdOrName(hospitalId, name)` - Returns filtered list of `DoctorDTO`

B. HospitalController – Method Descriptions

Method	Task	Implementation Details
`getAllHospitals()`	Fetches all hospitals	- Calls `hospitalService.getAllHospitals()` - Returns list of `HospitalDTO`
`getHospitalById()`	Fetches a hospital by ID	- Accepts `id` as path variable - Calls `hospitalService.getHospitalById(id)` - Returns the hospital DTO wrapped in `ResponseEntity` with status `HttpStatus.OK`
`saveHospital()`	Saves a new hospital	- Accepts `HospitalDTO` in request body (validated with `@Valid`) - Calls `hospitalService.saveHospital(hospital)` - Returns the created `HospitalDTO`
`updateHospital()`	Updates an existing hospital	- Accepts `id` as path variable and `HospitalDTO` as request body (validated with `@Valid`) - Calls `hospitalService.updateHospital(id, updatedHospital)` - Returns the updated `HospitalDTO`
`deleteHospitalById()`	Deletes a hospital by ID	- Accepts `id` as path variable - Calls `hospitalService.deleteHospitalById(id)` - Returns a Boolean `true` upon successful deletion
`searchHospitalsByName()`	Searches hospitals by name	- Accepts `name` as request parameter - Calls `hospitalService.searchHospitalsByName(name)` - Returns a list of matched `HospitalDTO`

C. PatientController – Method Descriptions

Method	Task	Implementation Details
`getAllPatients()`	Get all patients	- Calls `patientService.getAllPatients()` - Returns the list of `PatientDTO`
`getPatientById()`	Get patient by ID	- Accepts path variable `id` - Calls `patientService.getPatientById(id)` - Returns a `PatientDTO`
`updatePatient()`	Update patient	- Accepts path variable `id` and request body `PatientDTO` - Calls `patientService.updatePatient(id, updatedPatient)` - Returns the updated `PatientDTO`
`getPatientsByDoctorId()`	Get patients by doctor ID	- Accepts path variable `doctorId` - Calls `patientService.getPatientsByDoctorId(doctorId)` - Returns list of `PatientDTO`
`savePatient()`	Save new patient	- Accepts request body `PatientDTO` - Calls `patientService.savePatient(patient)` - Returns the saved `PatientDTO`
`deletePatientById()`	Delete patient by ID	- Accepts path variable `id` - Calls `patientService.deletePatientById(id)` - Returns a Boolean `true` on successful deletion
`searchPatientsByName()`	Search patients by name	- Accepts query parameter `name` - Calls `patientService.searchPatientsByName(name)` - Returns list of `PatientDTO`
`searchPatientsByDoctorId()`	Search patients by doctor ID	- Accepts query parameter `doctorId` - Calls `patientService.searchPatientsByDoctorId(doctorId)` - Returns list of `PatientDTO`

8 CONSIDERATIONS

A. There is no roles in this application
B. You can perform the following 3 possible actions

Hospital
Doctor
Patient

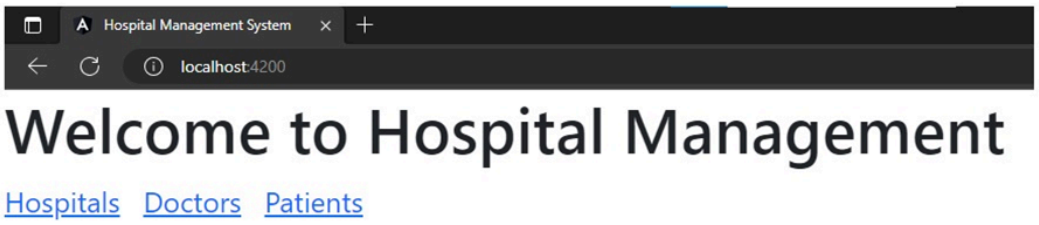
FRONTEND-ANGULAR SPA

1 Problem Statement

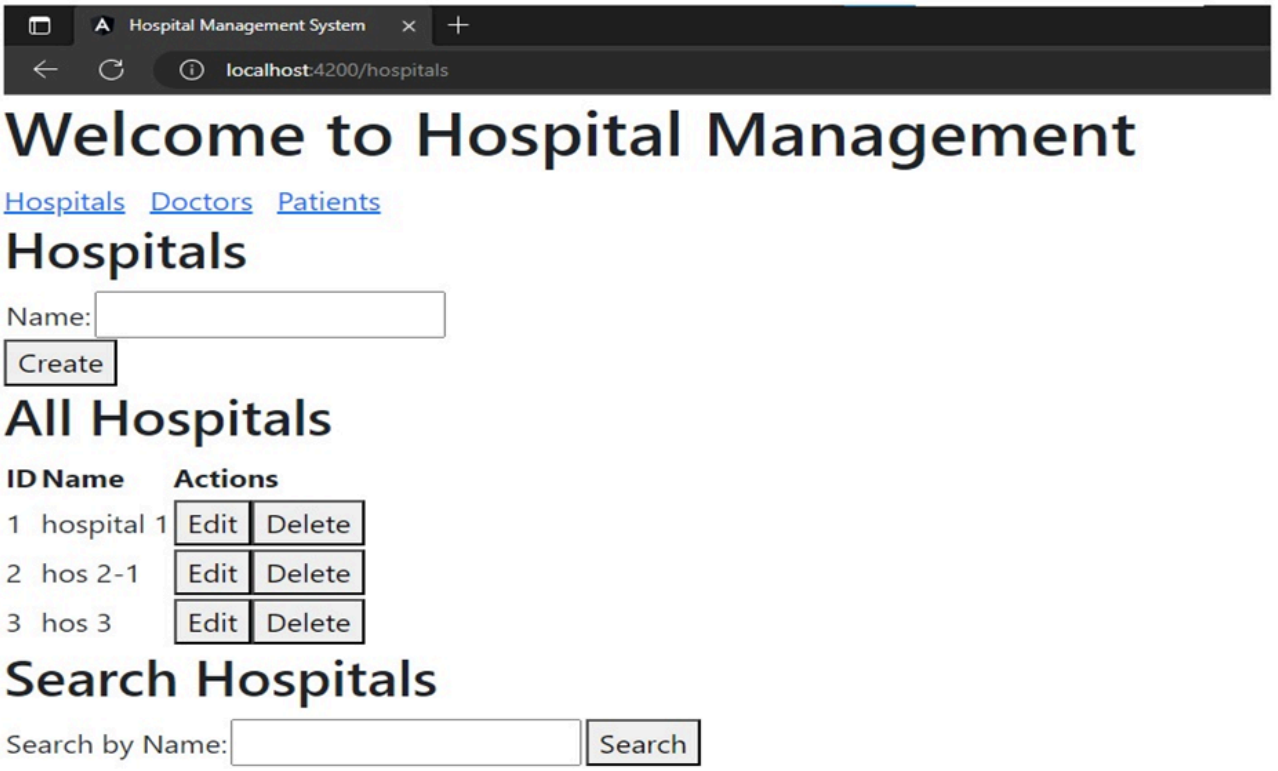
The HealthCare information system is SPA (Single Page Application), it allows the management of hospitals, doctors, and patients' profiles. It helps us to view, update, delete, create and filter all modules along with the functionality to search all modules.

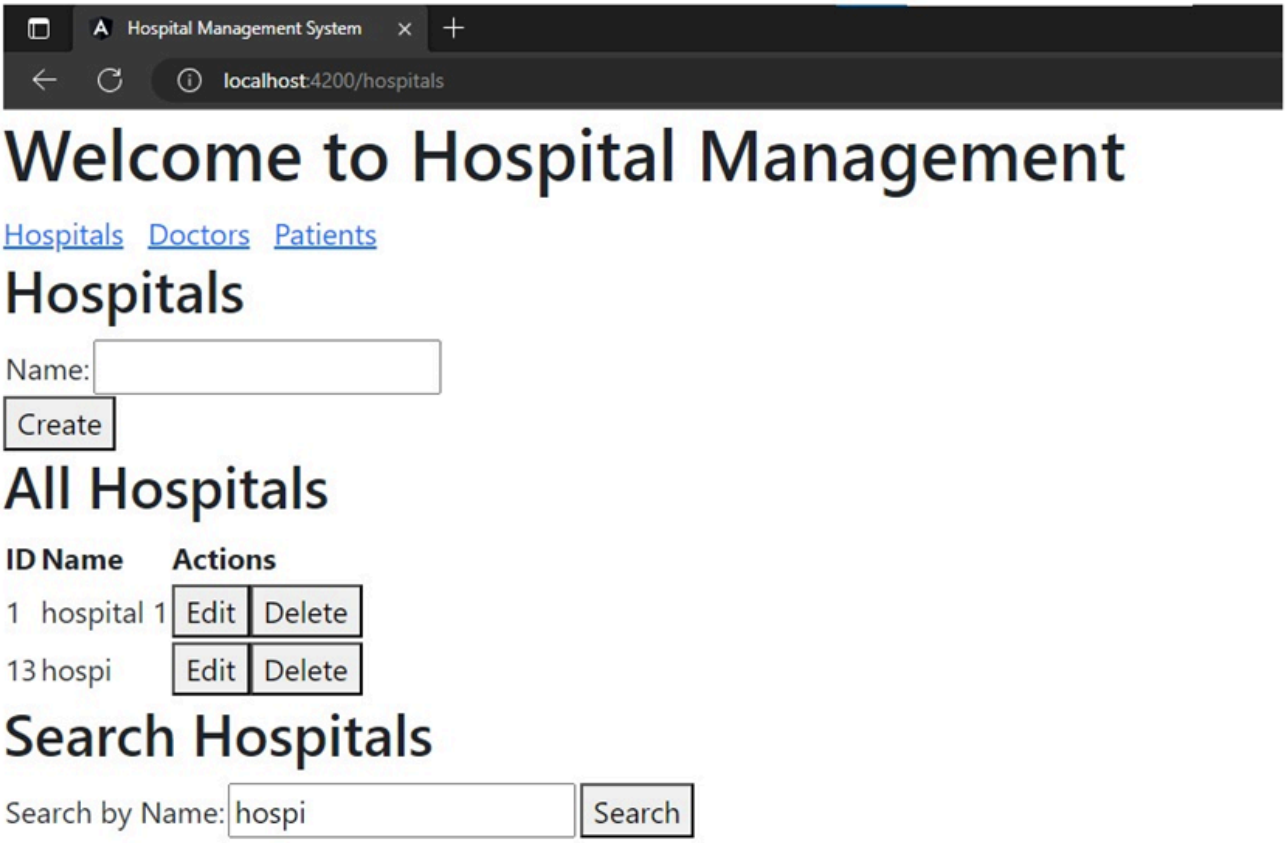
2 Proposed Healthcare Information System Wireframe

2.1 Homepage

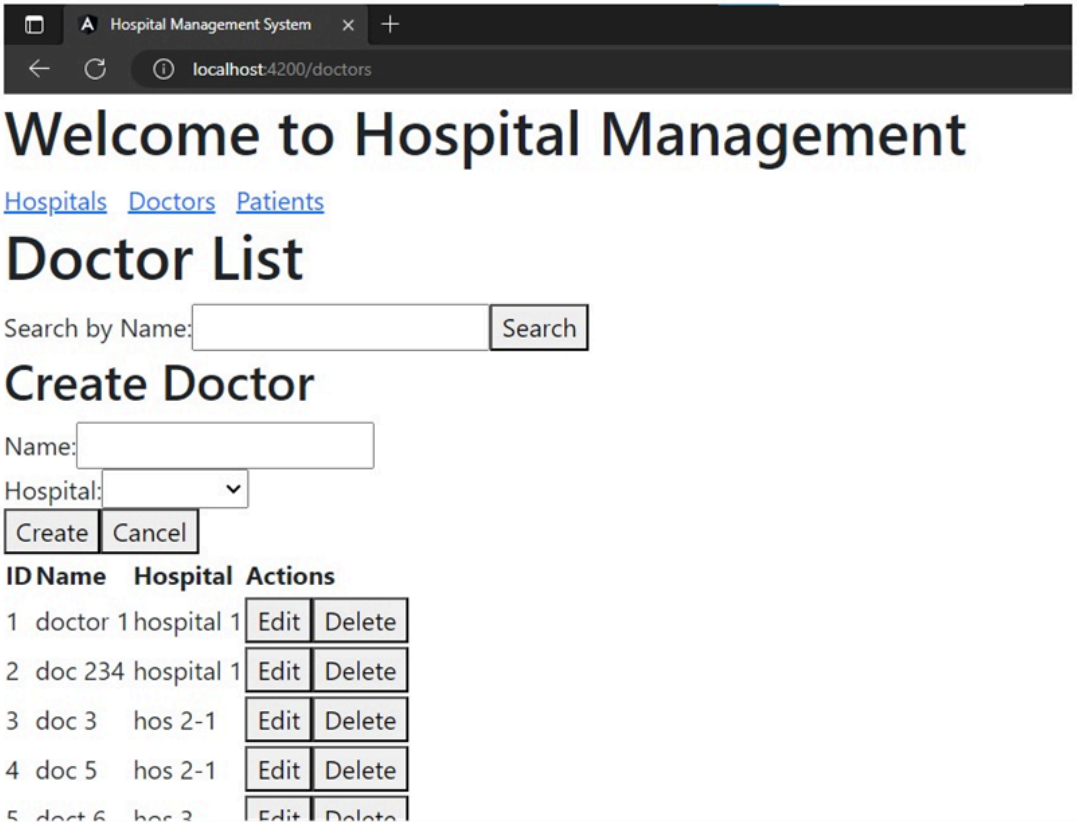


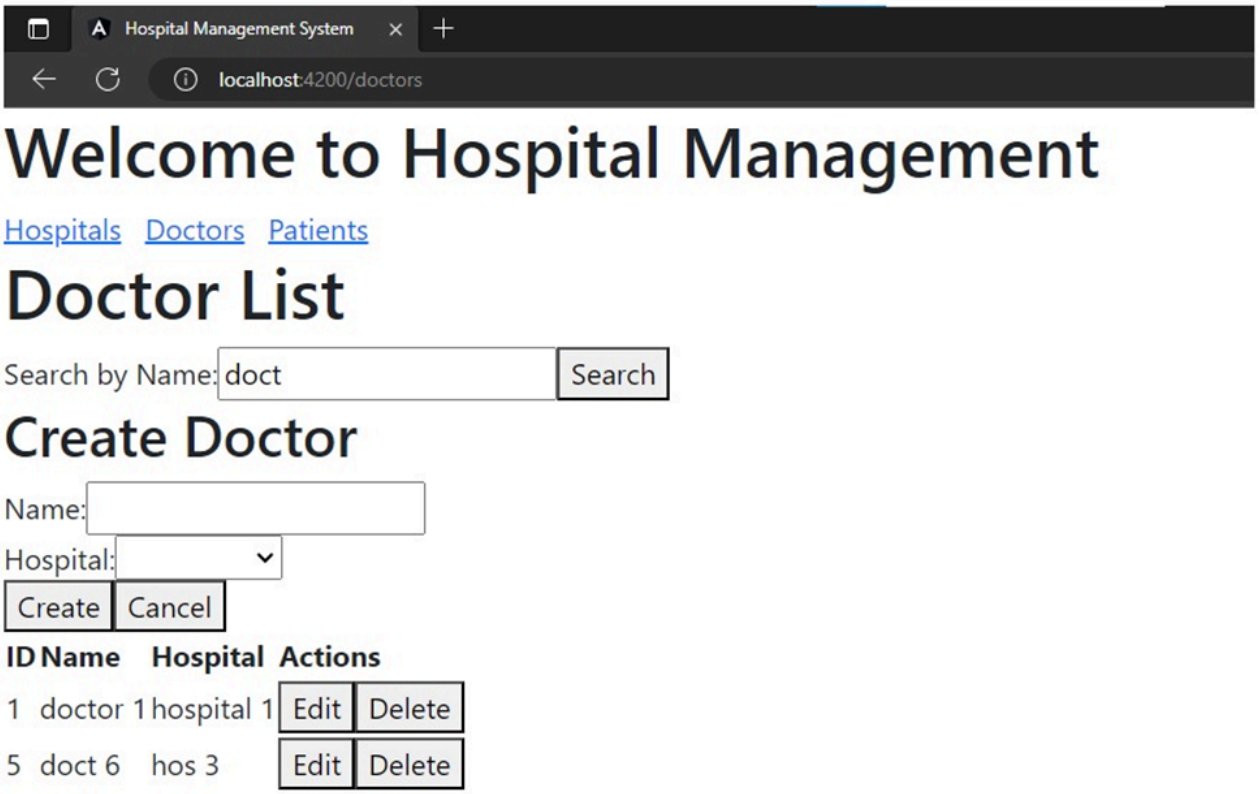
2.2 HOSPITALS PAGE



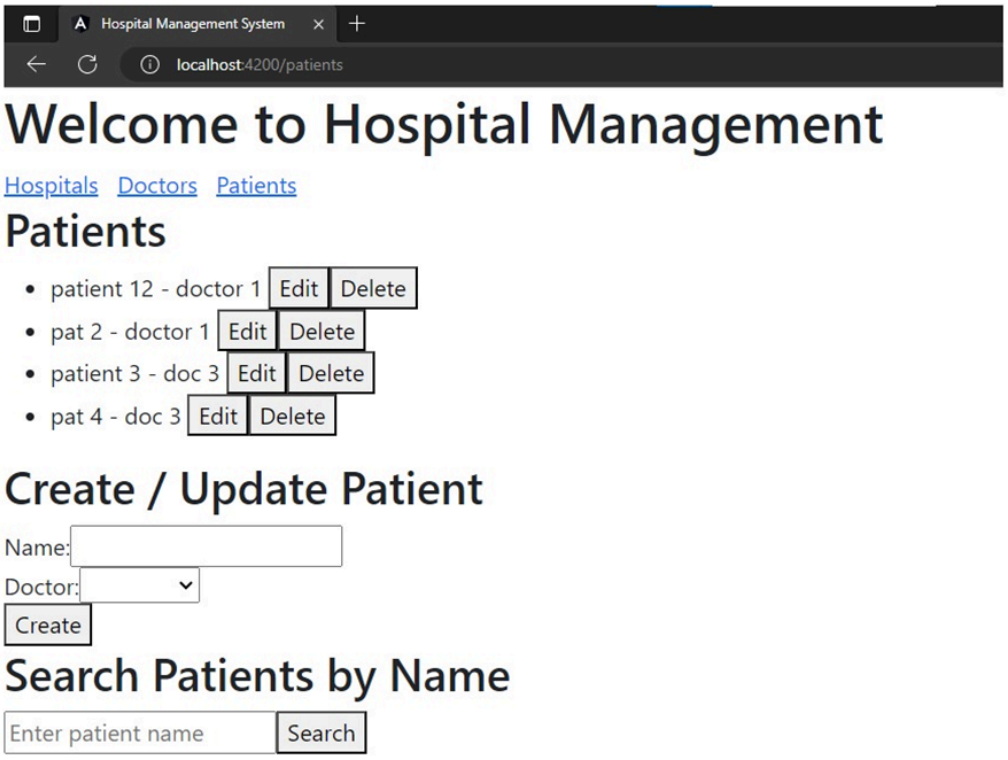


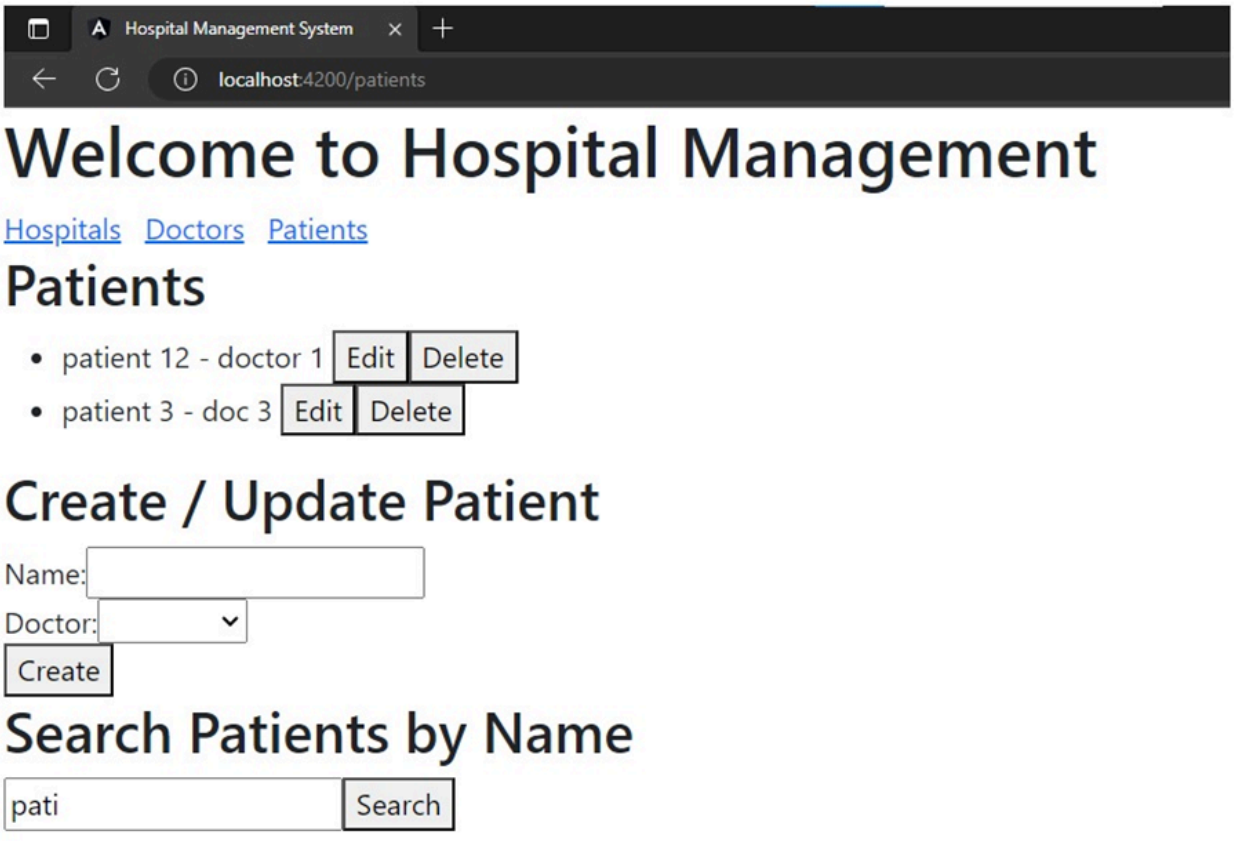
2.3 DOCTORS PAGE





2.4 PATIENTS PAGE





3 BUSINESS-REQUIREMENT:

As an application developer, develop the Healthcare Information System (Single Page App) with below guidelines:

User Story #	User Story Name	User Story
US_01	Home Page	As a user I should be able to visit the Home page as the default page. **Acceptance Criteria:** **App Shell — `app.component.html`** **HTML Structure** ● **Top-level heading:** Use an `h1` with text **"Welcome to Hospital Management."** ● **Navigation bar (`<nav>`):** Provides navigation links using `routerLink` to switch between: ○ **Hospitals** → router link to `/hospitals`, with active state. ○ **Doctors** → router link to `/doctors`, with active state. ○ **Patients** → router link to `/patients`, with active state. ● **Router outlet:** Place a single `<router-outlet>` under the nav to render routed views. **Dynamic Behavior** ● Clicking each nav item loads the respective component into the router outlet. ● Active link styling is applied via routerLinkActive. ● The heading and nav persist while routed content swaps beneath.

		## Routing — `app-routing.module.ts` **Purpose** Define client-side routes and initial redirect behavior. **Routes to Configure** - **Default redirect:** Empty path (`''`) must redirect to `/` with **full** path match (as in your file). - **Hospitals page:** `/hospitals` → **HospitalComponent** - **Doctors page:** `/doctors` → **DoctorComponent** - **Patients page:** `/patients` → **PatientComponent** **Functions & Responsibilities** - Register the route map above with the root router. - Export RouterModule so the router works app-wide. **Dynamic Behavior** - URL changes drive which component renders in the router outlet. - Landing on the site follows the default redirect rule before navigation. ## Root Component — `app.component.ts` **Purpose** Own the application shell: heading + navigation + outlet. **Responsibilities** - Bind the title in the component. - Provide a stable layout for all routed views .

		Root Module — `app.module.ts` **Purpose** Assemble the app: declare components, import Angular modules, provide services, and bootstrap the root. **Dynamic Behavior** ● App boots into **AppComponent**. ● Router resolves the default redirect, then renders the chosen feature component in the outlet. ● Services are injectable across the app (hospitals/doctors/patients features). **** Kindly refer to the screenshots for any clarifications. ****
US_02	Hospitals Page	As a user I should be able to see the hospital page and perform all operations: **Acceptance Criteria:** **Hospital Page — `hospital.component.html`** **HTML Structure** ● **Heading (h2):** Text: **Hospitals**. ● **Create/Update Form (table-based layout):** ○ **Row 1:** ■ Label cell: **Name:** ■ Input cell: single **text** input bound to either: ■ `newHospital.name` when **not** in edit mode, or ■ `selectedHospital.name` when **in** edit mode. ■ The input should: ■ Be **required**. ■ Trigger a key handler to keep `selectedHospital.name` in sync while editing. ○ **Action Buttons (below the form):** ■ **Create** button: ■ Visible **only when not** in edit

		mode. ■ Triggers create-or-update handler. ■ **Update** button: ■ Visible **only when** in edit mode. ■ Triggers create-or-update handler. ■ **Cancel** button: ■ Visible **only when** in edit mode. ■ Resets the form and exits edit mode. ● **Heading (h2):** Text: **All Hospitals**. ● **Hospitals List Table:** ○ **Header row: ID**, **Name**, **Actions**. ○ **Body rows:** one row per hospital: ■ **ID**: hospital id. ■ **Name**: hospital name. ■ **Actions**: ■ **Edit** button → enters edit mode with the selected hospital data prefilled. ■ **Delete** button → removes the hospital and refreshes the list. ● **Heading (h2):** Text: **Search Hospitals**. ● **Search Section (table-based layout):** ○ **Row:** ■ Label cell: **Search by Name:** ■ Input cell: text input bound to **searchName**. ■ Button cell: **Search** button to apply the filter by name. **Dynamic Behavior** ● The **Name** input switches its binding based on mode: ○ **Create mode:** uses **newHospital**. ○ **Edit mode:** uses **selectedHospital** and also listens to **key** events to keep the selected object updated. ● The visible action buttons change based on **isEditMode**. ● Searching replaces the current **hospitals** list with results.

Hospital Page — `hospital.component.ts`

Purpose

Manage hospital records: **list**, **create**, **update**, **delete**, **search**, and **load by id**. Control UI state for **create vs. edit** modes.

State

- **hospitals:** array holding all loaded hospitals.
- **newHospital:** model used for creating a hospital.
- **selectedHospital:** model used when editing a hospital.
- **searchName:** text used to filter hospitals by name.
- **isEditMode:** boolean flag to toggle between form modes.

Lifecycle

- **ngOnInit()**
  - Loads all hospitals into the list when the page initializes.

Functions & Responsibilities

1. **createOrUpdateHospital()**
   - **If not in edit mode:** delegates to **createHospital()**.
   - **If in edit mode:** delegates to **updateHospital()**.

2. **getAllHospitals()**
   - Retrieves the full hospitals list from the service.
   - Updates the **hospitals** array with results.

3. **createHospital()**
   - Sends the **newHospital** to the service for creation.
   - On success:
     - Refreshes the list.
     - Clears the form via **resetForm()**.

4. **updateHospital()**
   - Sends **selectedHospital** to the service for update (using its id).
   - On success:
     - Refreshes the list .
     - Clears the form via **resetForm()**.

5. **deleteHospital(id)**
   - Sends a delete request for the given id.

		○ On success: ■ Refreshes the list. 6. **searchHospitalsByName()** ○ Calls the service with **searchName** to retrieve filtered results. ○ Replaces **hospitals** with the filtered list. 7. **getHospitalById(id)** ○ Fetches a single hospital and sets **selectedHospital** with the result. 8. **editHospital(hospital)** ○ Copies the clicked hospital into **selectedHospital**. ○ Also copies into **newHospital** so the input displays the current name. ○ Sets **isEditMode = true** to switch the form into **Update/Cancel** mode. 9. **cancelEdit()** ○ Resets and exits edit mode by calling **resetForm()**. 10. **resetForm()** ○ Re-initializes **selectedHospital** and **newHospital** to empty models. ○ Sets **isEditMode = false**. 11. **onKeyPress(event)** ○ While **in edit mode**, keeps **selectedHospital.name** updated with the input's current value (helps real-time sync). **Dynamic Behavior** ● **Create vs. Update mode** is controlled via **isEditMode**. ● After **create**, **update**, or **delete**, the component **refreshes the list** and **resets** the form state. ● **Search** filters the current list using the service and updates the displayed rows. **** Kindly refer to the screenshots for any clarifications. ****

US_03	Doctors Page	As a user I should be able to see the doctor page and perform all operations: **Acceptance Criteria:** **Doctors Page — `doctor.component.html`** **HTML Structure** ● **Top heading:** Use an `h1` with the text **"Doctor List."** ● **Search Section:** ○ A **label** reading **"Search by Name:"** associated with a text **input**. ○ The input's value is bound to a field used for searching. ○ A **button** labeled **"Search"** triggers the search action. ● **Create/Update Doctor Form (template-driven):** ○ A form with a **template reference** (e.g., `#doctorForm="ngForm"`) that **submits** via a handler. ○ **Subheading (h2):** Shows **"Create Doctor"** when not editing, **"Update Doctor"** when editing. ○ **Form fields:** ■ **Name:** ■ Text input bound to the selected doctor's name. ■ **Hospital:** ■ A **select** dropdown bound to the selected doctor's hospital id. ■ Populated from a list of hospitals. ○ **Form buttons:** ■ **Primary submit button**: label switches between **"Create"** and **"Update"** depending on editing state. ■ **Cancel** button: resets the form and exits editing mode. ● **Doctors List Table:** ○ **Header row: ID**, **Name**, **Hospital**, **Actions**. ○ **Rows:** one per doctor with: ■ **ID** (doctor id), ■ **Name** (doctor name), ■ **Hospital** (doctor's hospital name), ■ **Actions**:

		■ **Edit** button to load the doctor into the form in edit mode. ■ **Delete** button to remove the doctor and refresh the list. **Dynamic Behavior** ● The **form header** and **submit button label** change based on whether editing is active. ● Selecting a **hospital** in the dropdown should set the **hospital id** on the selected doctor. ● Clicking **Edit** moves the row's data into the form and enables edit mode. ● Clicking **Cancel** clears the form and exits edit mode. ● Clicking **Search** filters the doctors list by **name**. ## Doctors Page — `doctor.component.ts` **Purpose** Manage doctors: **list**, **create**, **update**, **delete**, **search**, and **load hospitals** for assignment. Toggle between **create** and **edit** states. **State** ● **doctors:** array of all loaded doctors for display. ● **selectedDoctor:** the doctor currently being created/edited; includes nested **hospital** field. ● **hospitals:** array of available hospitals for the dropdown. ● **searchName:** search query for filtering by doctor name. ● **editing:** boolean flag to indicate **edit mode** vs **create mode**. **Lifecycle** ● **ngOnInit()** ○ Ensures **selectedDoctor.hospital** exists. ○ Loads the **doctors list**. ○ Loads the **hospitals list** for the dropdown. **Functions & Responsibilities** 1. **getAllDoctors()** ○ Fetches all doctors from the service. ○ Updates the **doctors** array used in the table. 2. **getHospitals()** ○ Fetches all hospitals from the service.

		○ Populates the **hospitals** array for the dropdown selector. 3. **saveDoctor()** ○ Entry point for the form submission. ○ If **editing** is true → calls **updateDoctor()**. ○ If **editing** is false → calls **createDoctor()**. 4. **createDoctor()** ○ Sends the current **selectedDoctor** to the service to create a new record. ○ On success: refreshes the list and **clears the form**. 5. **updateDoctor()** ○ Ensures the **hospital object** on **selectedDoctor** correctly reflects the chosen **hospital id** and name. ○ Sends an update to the service using the doctor's **id** and current values. ○ On success: refreshes the list and **clears the form**. 6. **deleteDoctor(id)** ○ Requests deletion of the doctor with the given **id**. ○ On success: refreshes the list. 7. **editDoctor(doctor)** ○ Copies the selected row into **selectedDoctor** for editing. ○ Sets **editing = true** to switch the form to **Update** mode. 8. **cancelEdit()** ○ Calls **clearForm()** to reset fields and exit edit mode. 9. **clearForm()** ○ Resets **selectedDoctor** to an empty doctor object. ○ Sets **editing = false**. 10. **searchDoctorsByName()** ○ Invokes the service using **searchName** to retrieve matching doctors. ○ Replaces **doctors** with the filtered result set.

		Dynamic Behavior ● **Editing mode** governs: ○ Form title (“Create Doctor” vs “Update Doctor”), ○ Submit button label (“Create” vs “Update”), ○ Whether the form uses **create** or **update** flow. ● **List refresh** occurs after **create**, **update**, or **delete**. ● **Dropdown selection** determines the doctor’s **hospital id** and should map to the correct hospital name before saving updates. **** Kindly refer to the screenshots for any clarifications. ****
US_04	Patients Page	As a user I should be able to see the patient page and perform all operations: **Acceptance Criteria:** **Patient Page — `patient.component.html`** **HTML Structure** ● **Section: Patients list** ○ Use a `div` wrapper. ○ Add an `h2` with text **“Patients.”** ○ Render a `ul` containing `li` items for each patient: ■ Show **patient name** and **assigned doctor’s name**. ■ Include **Edit** and **Delete** buttons per item. ● **Section: Create / Update Patient** ○ Use a `div` wrapper. ○ Add an `h2` with text **“Create / Update Patient.”** ○ Include a `form` with: ■ **Name** input (text), bound to the selected patient’s name, **required**. ■ **Doctor** select dropdown, bound to the selected patient’s doctor id, options populated from doctors list. ■ **Buttons**: ■ **Create** button visible only when **not editing**. ■ **Update** and **Cancel** buttons visible only when **editing**.

- **Section: Search Patients by Name**
  - Use a `div` wrapper.
  - Add an `h2` with text **"Search Patients by Name."**
  - Include:
    - A **text input** bound to the search field (name).
    - A **Search** button that triggers filtering.

Dynamic Behavior

- **Patient list** updates after create, update, or delete.
- **Form mode** toggles:
  - Not editing → **Create** button only.
  - Editing → **Update** and **Cancel** buttons only.
- **Selecting a doctor** sets the **doctor.id** on the selected patient.
- **Search** filters the list by the entered name; empty search reloads all patients.

Patient Page — `patient.component.ts`

Purpose

Handle **listing**, **creating**, **updating**, **deleting**, and **searching** patients, and **loading doctors** for assignment. Manage edit state and reset flows.

State

- **patients:** array of all patients for display.
- **doctors:** array of all doctors for the dropdown.
- **selectedPatient:** the patient record currently being created or edited.
- **editing:** boolean; **true** when editing an existing patient, **false** when creating a new one.
- **searchName:** string used to filter patients by name.

Lifecycle

- **ngOnInit()**
  - Ensures **selectedPatient.doctor** exists.
  - Loads **patients** list.
  - Loads **doctors** list.

Functions & Responsibilities

1. **getAllPatients()**
   - Retrieves all patients from the service.
   - Updates the **patients** array for the UI.

		2. **getAllDoctors()** ○ Retrieves all doctors from the service. ○ Populates the **doctors** array for the select dropdown. 3. **createPatient()** ○ Sends **selectedPatient** to the service to create a new record. ○ On success: reloads the list and **clears the form**. 4. **updatePatient()** ○ Sends current **selectedPatient** to the service to update. ○ On success: reloads the list and **clears the form**. 5. **deletePatient(id)** ○ Requests deletion for the given **patient id**. ○ On success: reloads the list. 6. **editPatient(patient)** ○ Copies the chosen patient into **selectedPatient**. ○ Sets **editing = true** to switch the form into update mode. 7. **clearSelectedPatient()** ○ Resets **selectedPatient** to an empty patient with a nested **doctor** object. ○ Sets **editing = false**. 8. **searchPatientsByName()** ○ When **searchName** is non-empty: fetches filtered patients by name and replaces **patients** with the result. ○ When **searchName** is empty: calls **getAllPatients()** to restore the full list. 9. **cancelEdit()** ○ Resets **selectedPatient** and **doctor** nested object. ○ Sets **editing = false**. **Dynamic Behavior** ● **Form mode** is governed by **editing**: ○ **false** → Create flow. ○ **true** → Update/Cancel flow. ● **List refresh** occurs after **create**, **update**, and **delete** operations. ● **Search** replaces the list with matches; clearing the

		query restores all patients.
	Services	**Doctor Service — `doctor.service.ts`** **Overview** This service handles **HTTP requests** related to doctors in the Hospital Management System. It communicates with the backend using **HttpClient** and provides methods for CRUD operations and search functionalities. • **baseUrl**: Base API endpoint for doctor-related requests. Example: `http://localhost:8081/healthcare/doctors` • **http**: Angular's `HttpClient` injected for making HTTP requests. **Methods & Responsibilities** **1. getAllDoctors()** • **Purpose:** Retrieve a list of all doctors. • **Action:** Sends a **GET** request to the `/doctors` endpoint. • **Usage in Component:** Used to populate the doctor list whenever the doctor page loads. • **Returns:** HTTP response containing an array of doctor objects. **2. getDoctorById(id: number)** • **Purpose:** Fetch details of a single doctor by their unique ID. • **Action:** Sends a **GET** request with the doctor ID appended to the endpoint. • **Usage in Component:** Called when viewing or editing a doctor's details. • **Returns:** HTTP response with the doctor object matching the ID. **3. createDoctor(doctor: Doctor)** • **Purpose:** Add a new doctor record. • **Action:** Sends a **POST** request with the new doctor object in the request body. • **Usage in Component:** Triggered when the **Create Doctor** form is submitted.

		• **Returns:** HTTP response containing the newly created doctor object. **4. updateDoctor(id: number, doctor: Doctor)** • **Purpose:** Update details of an existing doctor. • **Action:** Sends a **PUT** request to the endpoint with doctor ID and the updated doctor object in the request body. • **Usage in Component:** Triggered when updating a doctor's details in edit mode. • **Returns:** HTTP response containing the updated doctor object. **5. deleteDoctor(id: number)** • **Purpose:** Delete a doctor by their ID. • **Action:** Sends a **DELETE** request to the endpoint with doctor ID. • **Usage in Component:** Called when the **Delete** button is clicked in the doctor list. • **Returns:** HTTP response confirming deletion (usually void). **6. searchDoctorsByName(name: string)** • **Purpose:** Search for doctors by their name. • **Action:** Sends a **GET** request to `/doctors/search` with the `name` query parameter. • **Usage in Component:** Triggered when a search term is entered in the search box. • **Returns:** HTTP response containing a filtered list of doctor objects. **7. searchDoctorsByHospitalIdAndName(hospitalId: number, name: string)** • **Purpose:** Search for doctors by hospital and name combined. • **Action:** Sends a **GET** request to `/doctors/searchByHospital` with both `hospitalId` and `name` as query parameters. • **Usage in Component:** Useful when filtering doctors within a specific hospital. • **Returns:** HTTP response containing a filtered list of doctor objects belonging to that hospital. **Dynamic Behavior – DoctorService** • Each method directly communicates with the backend REST API at the `/doctors` endpoint.

- Methods are **consumed by** `DoctorComponent` to display the doctor list, view individual records, handle creation or update operations, search doctors, and perform deletions.

Hospital Service — `hospital.service.ts`

Overview

This service manages all the **HTTP requests related to hospitals** in the Hospital Management System. It interacts with the backend REST API using Angular's **HttpClient** and provides methods for CRUD operations and searching hospitals.

- **baseUrl**: Base API endpoint for doctor-related requests.
  Example:
  `http://localhost:8081/healthcare/hospitals`
- **http**: Angular's `HttpClient` injected for making HTTP requests.

Methods & Responsibilities

1. getAllHospitals()

- **Purpose:** Retrieve a list of all hospitals.
- **Action:** Sends a **GET** request to the `/hospitals` endpoint.
- **Usage in Component:** Used to populate the hospitals list when the hospital page is loaded.
- **Returns:** HTTP response containing an array of hospital objects.

2. getHospitalById(id: number)

- **Purpose:** Fetch details of a specific hospital by its unique ID.
- **Action:** Sends a **GET** request with the hospital ID appended to the URL.
- **Usage in Component:** Called when viewing or editing details of a particular hospital.
- **Returns:** HTTP response with the hospital object matching the ID.

3. createHospital(hospital: Hospital)

- **Purpose:** Add a new hospital record.

		• **Action:** Sends a **POST** request with the hospital object in the request body. • **Usage in Component:** Triggered when submitting the **Create Hospital** form. • **Returns:** HTTP response containing the newly created hospital object. **4. updateHospital(id: number, hospital: Hospital)** • **Purpose:** Update the details of an existing hospital. • **Action:** Sends a **PUT** request with the hospital ID in the URL and the updated hospital object in the request body. • **Usage in Component:** Triggered when submitting the **Update Hospital** form in edit mode. • **Returns:** HTTP response containing the updated hospital object. **5. deleteHospital(id: number)** • **Purpose:** Delete a hospital by its ID. • **Action:** Sends a **DELETE** request to the endpoint with the hospital ID. • **Usage in Component:** Triggered when the **Delete** button is clicked on the hospital list. • **Returns:** HTTP response confirming deletion (usually void). **6. searchHospitalsByName(name: string)** • **Purpose:** Search hospitals by their name. • **Action:** Sends a **GET** request to `/hospitals/search` with the `name` query parameter. • **Usage in Component:** Triggered when a user searches hospitals by name from the search box. • **Returns:** HTTP response containing a filtered list of hospital objects matching the search term. **Dynamic Behavior – HospitalService** • Each method communicates with the backend REST API at the `/hospitals` endpoint. • Methods are **consumed by `HospitalComponent`** to handle the hospital lifecycle — displaying lists, creating, updating, searching, and deleting hospitals. ---

Patient Service — `patient.service.ts`

Overview

This service handles all **patient-related API interactions** in the Hospital Management System. It provides methods for **CRUD operations** (Create, Read, Update, Delete) and supports **searching patients** by their name or doctor's name. All calls are made via Angular's **HttpClient** to the backend REST API.

- **baseUrl**: Base API endpoint for doctor-related requests.
  Example: `http://localhost:8081/healthcare/patients`
- **http**: Angular's `HttpClient` injected for making HTTP requests.

Methods & Responsibilities

1. getAllPatients()

- **Purpose:** Retrieve all patient records.
- **Action:** Sends a **GET** request to the `/patients` endpoint.
- **Usage in Component:** Called when the patient page loads to populate the patient list.
- **Returns:** HTTP response containing an array of patient objects.

2. getPatientById(id: number)

- **Purpose:** Fetch details of a single patient using their unique ID.
- **Action:** Sends a **GET** request with the patient ID in the URL.
- **Usage in Component:** Used when editing or viewing a specific patient's details.
- **Returns:** HTTP response containing the patient object.

3. createPatient(patient: Patient)

- **Purpose:** Add a new patient record.
- **Action:** Sends a **POST** request with the patient data in the request body.
- **Usage in Component:** Triggered when submitting the **Create Patient** form.
- **Returns:** HTTP response containing the newly created

		patient object. **4. updatePatient(id: number, patient: Patient)** • **Purpose:** Update details of an existing patient. • **Action:** Sends a **PUT** request with the patient ID in the URL and updated patient data in the request body. • **Usage in Component:** Triggered when the user edits a patient's details and submits the **Update Patient** form. • **Returns:** HTTP response containing the updated patient object. **5. deletePatient(id: number)** • **Purpose:** Remove a patient record permanently. • **Action:** Sends a **DELETE** request with the patient ID in the URL. • **Usage in Component:** Triggered when the **Delete** button is clicked from the patient list. • **Returns:** HTTP response confirming deletion (usually void). **6. searchPatientsByName(name: string)** • **Purpose:** Find patients based on their name. • **Action:** Sends a **GET** request to `/patients/search` with `name` as a query parameter. • **Usage in Component:** Used in the **Search Patients by Name** input field. • **Returns:** HTTP response containing a filtered list of patients matching the given name. **7. searchPatientsByDoctorName(name: string)** • **Purpose:** Find patients assigned to a doctor by the doctor's name. • **Action:** Sends a **GET** request to `/patients/searchByDoctor` with `name` as a query parameter. • **Usage in Component:** Helpful when filtering patients based on their doctor's name. • **Returns:** HTTP response containing a filtered list of patients whose doctor's name matches the search term.

		Dynamic Behavior – PatientService • Every method in this service interacts directly with the backend API at the `/patients` endpoint. • Methods are consumed by `PatientComponent` to **list, create, update, search, and delete patient records**. **** Kindly refer to the screenshots for any clarifications. ****

4 CONSTRAINTS

1. On the page load, input focus must come to the first name input field.
2. You should be able to press the “TAB” key and “SHIFT + TAB” to navigate from top field to bottom field and vice-versa.

9 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.**
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**
    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

10 EXECUTION STEPS TO FOLLOW FOR FRONTEND

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**

   Application menu (Three horizontal lines at left top) -> Terminal ->New Terminal.
3. **This is a web-based application, to run the application on a browser, use the internal browser in the environment.**
4. **You can follow series of command to setup Angular environment once you are in your project-name folder:**
   a. **npm install -> Will install all dependencies -> takes 10 to 15 min**
   b. **npm run start -> To compile and deploy the project in browser. You can press <Ctrl> key while clicking on localhost:4200 to open project in browser -> takes 2 to 3 min**
   c. **npm run test -> to run all test cases. It is mandatory to run this command before submission of workspace -> takes 5 to 6 min**