📘 Capstone Project Documentation

Title: Healthcare & Wellness Management System

Tech Stack:

- **Frontend:** React.js (with Bootstrap)
- **Backend:** Spring Boot (REST APIs)
- **Database:** MySQL
- **Authentication:** Spring Security/JWT (JSON Web Token)
- **Build Tools:** Maven (backend), npm (frontend)
- **Deployment:** Localhost / Docker

---

1. Problem Statement

Healthcare and wellness services often rely on manual record-keeping and fragmented systems, which lead to delays in patient care, lack of proper tracking of medical history, and limited interaction between patients and providers.

The goal is to build a **Healthcare & Wellness Management System** that allows patients to book appointments, track health records, and access wellness services, while administrators and healthcare providers can manage patient records, appointments, and reports.

---

2. Objectives

- Provide a user-friendly web interface for patients and providers.
- Implement secure authentication using JWT.
- Enable appointment scheduling and management.
- Maintain patient health records and visit history.
- Provide wellness service tracking (e.g., fitness programs, therapy sessions).
- Allow Admin users to manage patients, providers, and services.
- Store and manage data using MySQL database.

---

3. Scope of the System

3.1 Roles

1. Patient

- Register/Login
- Browse wellness services and doctors
- Book, reschedule, or cancel appointments
- View health records and appointment history
- Track ongoing wellness programs

2. Doctor / Wellness Provider

- Login with credentials
- View scheduled appointments
- Update patient health records and treatment notes
- Manage availability

3. Admin

- Login with Admin credentials
- Manage patients, providers, and services
- View all appointments
- Generate reports (appointments, patient records, wellness program progress)

---

4. System Architecture

Architecture Style: 3-Tier Architecture

- **Presentation Layer (React.js):** UI for patients, providers, and admin.
- **Business Layer (Spring Boot):** Handles APIs, authentication, and business logic.
- **Data Layer (MySQL):** Stores patients, providers, appointments, wellness programs.

High-Level Architecture Diagram:

```
[React Frontend] → [Spring Boot REST API] → [MySQL Database]
       UI                Business Logic           Persistent Storage
```

---

5. Modules

5.1 Patient Module

- Registration & login (JWT authentication)
- Profile management
- View and update personal health records

5.2 Provider Module

- View assigned appointments
- Update patient health records
- Manage availability schedule

5.3 Appointment Module

- Book, reschedule, or cancel appointments
- View upcoming and past appointments
- Admin: manage all appointments

5.4 Wellness Service Module

- Browse available wellness programs
- Enroll in programs
- Track progress and sessions

5.5 Admin Dashboard

- Manage patients, providers, and wellness services
- View all appointments and program participation
- Generate reports (health records, service utilization)

---

6. Database Design (MySQL)

Tables:

1. `patients` (id, name, email, password, phone, address, dob, gender)
2. `providers` (id, name, email, password, specialization, phone, role)
3. `appointments` (id, patient_id, provider_id, appointment_date, status, notes)
4. `wellness_services` (id, name, description, duration, fee)
5. `enrollments` (id, patient_id, service_id, start_date, end_date, progress)
6. `payments` (id, patient_id, appointment_id, service_id, payment_status, payment_date, transaction_id)

6.1 Relationship Summary (PK–FK)

Relationship	Type
patients → appointments	1:N (`appointments.patient_id → patients.id`)
providers → appointments	1:N (`appointments.provider_id → providers.id`)
patients → enrollments	1:N (`enrollments.patient_id → patients.id`)

Relationship	Type
wellness_services → enrollments	1:N (`enrollments.service_id → wellness_services.id`)
appointments → payments	1:1 (`payments.appointment_id → appointments.id`)
wellness_services → payments	1:N (`payments.service_id → wellness_services.id`)
patients → payments	1:N (`payments.patient_id → patients.id`)

6.2 Table Structures

1. patients

Column	Type	Constraints
id	INT	PRIMARY KEY, AUTO_INCREMENT
name	VARCHAR(100)	NOT NULL
email	VARCHAR(100)	UNIQUE, NOT NULL
password	VARCHAR(255)	NOT NULL
phone	VARCHAR(20)	NULL
address	VARCHAR(255)	NULL
dob	DATE	NULL
gender	ENUM('Male','Female','Other')	NULL

2. providers

Column	Type	Constraints
id	INT	PRIMARY KEY, AUTO_INCREMENT
name	VARCHAR(100)	NOT NULL
email	VARCHAR(100)	UNIQUE, NOT NULL
password	VARCHAR(255)	NOT NULL
specialization	VARCHAR(100)	NULL
phone	VARCHAR(20)	NULL
role	ENUM('DOCTOR','WELLNESS_PROVIDER')	NOT NULL

3. appointments

Column	Type	Constraints
id	INT	PRIMARY KEY, AUTO_INCREMENT
patient_id	INT	FOREIGN KEY → patients(id), ON DELETE CASCADE
provider_id	INT	FOREIGN KEY → providers(id), ON DELETE CASCADE
appointment_date	DATETIME	NOT NULL
status	ENUM('PENDING','CONFIRMED','COMPLETED','CANCELLED')	DEFAULT 'PENDING'
notes	TEXT	NULL

4. wellness_services

Column	Type	Constraints
id	INT	PRIMARY KEY, AUTO_INCREMENT
name	VARCHAR(100)	NOT NULL
description	TEXT	NULL
duration	INT	Duration in days / weeks
fee	DECIMAL(10,2)	NOT NULL

5. enrollments

Column	Type	Constraints
id	INT	PRIMARY KEY, AUTO_INCREMENT
patient_id	INT	FOREIGN KEY → patients(id), ON DELETE CASCADE
service_id	INT	FOREIGN KEY → wellness_services(id), ON DELETE CASCADE
start_date	DATE	NOT NULL
end_date	DATE	NOT NULL
progress	INT	Percentage completed (0–100)

6. payments

Column	Type	Constraints
id	INT	PRIMARY KEY, AUTO_INCREMENT
patient_id	INT	FOREIGN KEY → patients(id), ON DELETE CASCADE
appointment_id	INT	UNIQUE, FOREIGN KEY → appointments(id), ON DELETE CASCADE
service_id	INT	FOREIGN KEY → wellness_services(id), ON DELETE CASCADE
payment_status	ENUM('PENDING','SUCCESS','FAILED')	DEFAULT 'PENDING'
payment_date	DATETIME	DEFAULT CURRENT_TIMESTAMP
transaction_id	VARCHAR(100)	NULL

---

7. Frontend (React.js)

- Routing: React Router (Login, Register, Home, Appointments, Wellness Services, Admin Dashboard)
- State Management: React Hooks
- UI Library: Bootstrap
- API Calls: Axios for HTTP requests to Spring Boot REST APIs

Components:

- Navbar, Footer
- Patient: AppointmentForm, HealthRecord, WellnessEnrollment
- Provider: AppointmentList, PatientRecordUpdate
- Admin: PatientList, ProviderList, ServiceManagement, Reports

8. Backend (Spring Boot)

Dependencies: Spring Web, Spring Data JPA, Spring Security (JWT), MySQL Driver, Validation API, Swagger (OpenAPI Docs)

REST API Endpoints:

Module	Endpoint	Functionality
Authentication	POST /api/auth/register	Register Patient/Provider
Authentication	POST /api/auth/login	Login & get JWT
Patients	GET /api/patients/{id}	Get patient details
Patients	PUT /api/patients/{id}	Update profile
Providers	GET /api/providers/{id}	Get provider details
Appointments	POST /api/appointments	Book appointment
Appointments	GET /api/appointments/{patientId}	View patient appointments
Appointments	GET /api/appointments (Admin)	View all appointments
Wellness Services	GET /api/services	Get all wellness services
Enrollments	POST /api/enrollments	Enroll patient in service
Payments	POST /api/payments	Process payment

9.UX Design Guidelines

General Principles

- **Mobile-first responsive design** using Bootstrap 5 or Material UI.
- Maintain **consistent colors, fonts, and spacing** for visual harmony.
- Use **progressive disclosure** to avoid overwhelming users.
- Provide **inline validation** and friendly error messages.
- Ensure **accessibility compliance**: semantic HTML, ARIA roles, keyboard navigation, contrast ratios.
- Optimize performance with **lazy loading** and skeleton loaders.

User Journey Examples

- **Patient**: Simple registration, quick appointment booking with calendar widget, easy access to health records.
- **Provider**: Dashboard for appointment overview, quick update forms for patient notes.
- **Admin**: Comprehensive management dashboard with filters, search, and reports.

10. UI Component Guidelines

Component	Description	UX Best Practices
Navbar	Main navigation with user info and role-based links	Responsive, clear active state, hamburger menu on mobile
Forms	Registration, login, booking, payment forms	Inline validation, clear placeholders, tooltips
Cards	Show summaries: patient info, appointments, wellness	Clean layout, actionable buttons
Tables	Lists of appointments, payments, patients	Sorting, filtering, pagination, row selection
Modals	Confirmation dialogs (cancel, delete)	Clear message, prominent confirm/cancel buttons
Notifications	Toasts for success/failure	Non-blocking, dismissible, consistent styling
Charts	Progress visualization, stats	Use intuitive chart types, legends, and tooltips

Sample Spring Boot Controller with Swagger Annotations

```java
@RestController
@RequestMapping("/api/auth")
@Tag(name = "Authentication", description = "Authentication APIs")
public class AuthController {

    @Operation(summary = "Register new patient/provider")
    @PostMapping("/register")
    public ResponseEntity<?> register(@RequestBody RegisterRequest request) {
        // registration logic
        return ResponseEntity.ok("User registered");
    }

    @Operation(summary = "Login to get JWT token")
    @PostMapping("/login")
    public ResponseEntity<JwtResponse> login(@RequestBody LoginRequest request) {
        // login logic
        return ResponseEntity.ok(new JwtResponse(token));
    }
}
```

11.Swagger UI Screens for Spring Boot API

1. API Documentation Home / Overview

- Displays the **API title**, version, and description (from your OpenAPI info).
- Shows all **grouped endpoints** organized by **tags** (e.g., Authentication, Patients, Appointments, Wellness Services, Payments).
- Search bar to quickly find any endpoint.

2. Endpoint List and Grouping

- APIs grouped by **tags** on the left sidebar or main content area.
- Expand any tag to see endpoints within the group.
- Example groups:
    - **Authentication**: `/api/auth/register`, `/api/auth/login`

- **Patients**: `/api/patients/{id}`, `/api/patients/{id} (PUT)`
- **Appointments**: `/api/appointments` (GET, POST, etc.)

3. Individual Endpoint Details

- **HTTP method**, **path**, and brief description at the top.
- **Parameters**:
  - Path variables (`{id}`)
  - Query parameters
  - Headers (e.g., Authorization)
  - Request body schema with example JSON payload
- **Responses**:
  - Status codes (200, 400, 401, 404, 500)
  - Response body schema with examples
- **Try it out** button to execute requests live (requires JWT token if secured).

Example:

For `POST /api/auth/login`:

- Request body expects:

```
{
  "email": "user@example.com",
  "password": "password123"
}
```

- Response 200 returns JWT token string.

---

4. Security / Authorization Input

- Top-right **Authorize** button.
- Popup to enter **JWT Bearer token** (e.g., `Bearer eyJhbGciOiJIUzI1NiIs...`).
- Once authorized, all secured endpoints can be tested live with token automatically attached in header.

5. Models / Schemas Section

- Clickable models show data structure of request and response payloads.
- Displays field types, validations, enums, and nested models.
- Useful for developers to understand what data to send and expect.

6. Error Responses

- Common HTTP error codes documented per endpoint.
- Example error responses, e.g.:

```
{
  "timestamp": "2025-08-25T10:30:00Z",
  "status": 401,
  "error": "Unauthorized",
  "message": "JWT token is missing or invalid"
}
```

---

12. Security

- JWT Authentication & Authorization
- Role-based access (Patient / Provider / Admin)
- Passwords hashed using BCrypt

---

13. Testing

- Unit Testing: JUnit & Mockito (service/repository)
- Integration Testing: Spring Boot Test for API endpoints
- Frontend Testing: React Testing Library / Jest

---

14. Deployment

- Backend: Spring Boot app packaged as JAR, deployable on Tomcat
- Frontend: React app build deployed on Netlify
- Database: MySQL hosted locally / RDS

---

15. Future Enhancements

- Telemedicine integration (video consultations)
- Real payment gateway for services
- AI-driven wellness recommendations

- Health tracking dashboard (vitals, activity)
- Mobile app version (React Native)

16. General Guidelines::

1.Project Organization

Ensure a uniform directory layout separating frontend, backend, and documentation clearly.

Distinguish configuration files, source code, and assets into different folders for better clarity and maintenance.

2.Coding Practices

Choose descriptive names for variables, functions, and classes.

Adhere to consistent formatting, including indentation, spacing, and proper commenting throughout the codebase.

3.Git Version Control

Repository Initialization: Start a Git repository at the root of the project using git init.

Branching Model:

Use main (or master) branch exclusively for stable, production-ready code.

Develop new features in separate branches named feature/<feature-name>.

4.Documentation

Regularly update API specifications, UI/UX guidelines, and system architecture diagrams.

Maintain a comprehensive README detailing project setup, dependencies, and usage instructions.

5.Testing and Quality Assurance

Perform thorough testing of features prior to commits.

Keep detailed records of test cases and results for major components.