MediWay

Smart Health Management System

2) High-level technical architecture (Spring Boot stack)

- **Frontend:** React (web) — consumes REST APIs, mirrors wireframes/storyboards from report.
- **Backend:** Java Spring Boot (REST API)
    - Spring Boot (2.7+ / 3.x)
    - Spring Web (REST controllers)
    - Spring Data JPA (Hibernate)
    - PostgreSQL (primary relational DB)
    - Spring Security (JWT) — *authentication is supporting infra but not a primary graded use case.*
    - Swagger / OpenAPI for API docs
    - JUnit + Mockito + Spring MockMvc for unit & integration tests
    - Flyway or Liquibase for DB migrations
- **External integrations:** QR generator library (zxing) or service, Payment gateway sandbox (Stripe or PayHere), optional email service (SendGrid/Mailgun).
- **Hosting/CI:** GitHub Actions for CI (tests + build), deploy backend to Render/Heroku/AWS Elastic Beanstalk.

---

3) Database design (ERD summary)

Primary tables and important fields (PK = primary key, FK = foreign key):

1. **patients**
    - patient_id (UUID, PK)
    - full_name
    - email (unique)
    - phone
    - dob (date)
    - gender
    - height_cm
    - weight_kg
    - address
    - password_hash
    - qr_code_data (string or link)

    - created_at, updated_at
2. **doctors**
    - doctor_id (UUID, PK)
    - full_name
    - specialization
    - contact
    - availability (JSON / separate schedule table)
3. **doctor_schedules** (recommended separate)
    - schedule_id (PK)
    - doctor_id (FK → doctors)
    - date (date)
    - time_slot (e.g., "09:00-09:30")
    - slot_status (available/booked)
4. **appointments**
    - appointment_id (UUID, PK)
    - patient_id (FK → patients)
    - doctor_id (FK → doctors)
    - schedule_id (FK → doctor_schedules)
    - status (BOOKED / CANCELLED / COMPLETED)
    - reason
    - created_at
5. **payments**
    - payment_id (UUID, PK)
    - appointment_id (FK → appointments)
    - amount (decimal)
    - method (CARD / INSURANCE / CASH)
    - transaction_id
    - status (PENDING / SUCCESS / FAILED)
    - paid_at
6. **reports** (metadata)
    - report_id (UUID, PK)
    - generated_by (admin id)
    - report_type
    - filters (JSON)
    - generated_at
    - file_path (optional)
7. **admins**
    - admin_id, name, email, password_hash, role

Relationships

- patients 1..* → appointments
- doctors 1..* → doctor_schedules
- appointments → payments (0..1)
- reports are generated from aggregated patient/appointment/payment data

Indexes

- index on patients.email
- index on doctor_schedules(doctor_id, date)
- index on appointments(patient_id, doctor_id, status)

---

4) REST API endpoints (Spring Boot controllers)

Patients / Profile / QR

- `POST /api/patients/register`
  - Request: patient data (name, email, phone, dob, gender, password, etc.)
  - Behavior: create patient, create QR code payload (qr_code_data), store hashed password.
  - Response: patient_id, qr_code_data or QR URL.
- `GET /api/patients/{patientId}` — get profile
- `PUT /api/patients/{patientId}` — update profile

*(Auth endpoints like login are required but **do not** count as graded use cases — still implement minimal endpoints with JWT for API security.)*

Doctors & Schedules

- `GET /api/doctors?specialization={spec}` — list doctors (filter by specialization)
- `GET /api/doctors/{doctorId}/schedules?date=YYYY-MM-DD` — available slots

Appointments (Use case A: Scheduling)

- `POST /api/appointments`
  - Request: patientId, doctorId, scheduleId, reason.
  - Behavior: check slot availability (transactional), create appointment, mark schedule booked, return confirmation id.
- `PUT /api/appointments/{appointmentId}` — update or cancel (status change)
- `GET /api/appointments/patient/{patientId}` — list patient appointments

Payments (Use case B: Payment Processing)

- `GET /api/payments/patient/{patientId}` — list bills / unpaid items
- `POST /api/payments`
  - Request: appointmentId, method, payment details (tokenized card or insurance id)
  - Behavior: call payment gateway, create payment record, mark appointment as paid if success.
- `GET /api/payments/{paymentId}/receipt` — return receipt/pdf metadata

Reports (Use case C: Generate Statistical Reports)

- `POST /api/reports/generate`
  - Request: reportType, dateRange, filters
  - Behavior: aggregate DB data, generate CSV/PDF, store metadata row in `reports`.
- `GET /api/reports?type=&from=&to=` — list reports
- `GET /api/reports/{id}/download` — download file

QR / Utilities (Use case D: Registration & QR)

- `GET /api/patients/{patientId}/qrcode` — generate/serve QR image (on-demand)
- `POST /api/scan` — (kiosk) Provide QR data; server returns patient info and status

---

5) Suggested improvements to original design (brief — for the group report)

(You'll include this in the group critique; these are concrete and easy to justify.)

1. **Separate doctor_schedules table** instead of storing availability as blob — improves atomic booking and concurrency control (prevents double-booking).
2. **Make QR data stateless & verifiable** (include patient_id + issued_at + HMAC) rather than storing raw images only — improves offline kiosk verification.
3. **Add payments table normalization** and transaction handling with idempotency key — required for safe retries and correct grading of payment use case.
4. **Add report filters as JSON** and store generated parameters to make reports reproducible and auditable (aligns with requirement for health administrators).

These changes are small but meaningful and directly map to the requested use-cases.

---

6) Division of work — each member implements one substantial use case

Each member should implement **end-to-end** functionality for their use case, include tests (≥80% of functionality coverage for their code), UI consistency with wireframes, and document the API + DB migrations they produce.

Shalon — Use Case: Appointment Scheduling (Make / Update / Cancel Appointment)

Why: central business use case; coordinates locking and transactions.
Responsibilities & steps

1. Create `Appointment` entity, `DoctorSchedule` entity, JPA repositories.
2. Implement transactional booking logic in `AppointmentService`:
    - Check schedule slot availability (pessimistic lock or optimistic version).
    - Create appointment, mark schedule booked atomically.
    - Emit event / send notification (async optional).
3. Implement controllers:
    - `POST /api/appointments`
    - `PUT /api/appointments/{id}` for update/cancel
    - `GET /api/appointments/patient/{id}`
4. Write unit tests:
    - success booking, double-book prevention, cancel flow, invalid slot.
    - Use `@DataJpaTest` and MockMvc for controller tests.
5. Integration test covering a full booking -> payment-ready appointment.
6. Deliverables: API docs for these endpoints, DB migration scripts for doctor_schedules & appointments, sample Postman collection.

Shirantha — Use Case: Reports Generation (Admin dashboards + Export)

Why: one of the four substantial business use cases (analytics & decision-making).
Responsibilities & steps

1. Design `Report` entity and repository; implement storage of filter JSON and generated file path.
2. Implement aggregation queries (JPQL / native SQL) for:
    - daily/weekly/monthly patient visits,
    - department-wise statistics,
    - appointment waiting times,
    - resource utilization (beds/staff) — scaffold fake data where needed.
3. Implement `POST /api/reports/generate`:
    - Accept filters → run aggregator → generate CSV/PDF (use Apache POI / iText or simpler CSV lib).
    - Store report metadata and return report_id.
4. Implement `GET /api/reports/{id}/download`.
5. Frontend: minimal admin dashboard page with filters, charts (Chart.js).
6. Tests:
    - Unit tests for aggregators, edge cases for empty data, and download endpoints.
7. Deliverables: sample exported report, OpenAPI examples, example charts.

Navodya — Use Case: Patient Registration & QR Code Generation

Why: foundational use case (patient onboarding + QR identity).
Responsibilities & steps

1. Implement `Patient` entity + `PatientRepository`.
2. Implement `POST /api/patients/register`:
    - Validate input, hash password (BCrypt), create patient row.
    - Generate QR payload (JSON with patientId + issuedAt + signature) and store `qr_code_data` (or store QR image path).
    - Return QR content and patient id.
3. Implement `GET /api/patients/{id}/qrcode` to generate/serve PNG (use ZXing).
4. Implement minimal `GET /api/patients/{id}` and `PUT /api/patients/{id}` (profile update).
5. Tests:
    - registration success/failure, duplicate email, QR generation content verification.
6. Deliverables: DB migration, sample QR PNG, Swagger docs for endpoints.

Nipuni — Use Case: Payment Processing (Pay hospital bill / Insurance flows)

Why: complex business workflow, must handle success/failure and reconciliation.
Responsibilities & steps

1. Implement `Payment` entity and `PaymentRepository`.
2. Implement `GET /api/payments/patient/{id}` to list unpaid items (derived from appointments).
3. Implement `POST /api/payments`:
    - Accept payment request (appointmentId, method, payment token/insurance info), call payment gateway sandbox, update `payments` table atomically.
    - Support partial insurance coverage flow (simulate insurance approvals in sandbox/mock).
4. Implement `GET /api/payments/{id}/receipt` — produce receipt metadata or PDF.
5. Ensure idempotency for retries (idempotency key handling).
6. Tests:
    - success, failure, partial insurance, retry/idempotency cases.
7. Deliverables: sample payment receipt, test vectors for gateway mocking.

---

7) Implementation & testing guidelines (applies to all members)

- **Project structure (suggested Maven layout)**
- `src/main/java/com/mediw ay`
- `/config`

- `/controller`
- `/service`
- `/repository`
- `/model (entities, dtos)`
- `/exception`
- `src/test/java/...`
- **DB migrations:** use Flyway. Each member provides SQL migration for their entities/tables.
- **DTOs & Validation:** use DTOs for request/response; validate with `@Valid` and `javax.validation` annotations.
- **Transactions:** annotate service methods with `@Transactional` for atomic operations.
- **Testing targets:** aim for unit tests + slice tests. For controllers use MockMvc; for services use Mockito and H2 in-memory DB for data-layer tests.
- **Coverage goal:** individually aim for ≥80% coverage of the functionality they implemented (not necessarily whole project). Use Jacoco in CI to enforce.
- **API docs:** annotate controllers with Swagger / Springdoc OpenAPI and include example request/response.
- **Logging & error handling:** centralized `@ControllerAdvice` and consistent error DTO.
- **Security:** minimal JWT auth to protect endpoints (but assignment says avoid simple functions as graded use-case — auth ok as infra).

---

8) Commit, PR & collaboration workflow (quick checklist)

- Each feature on separate branch `feature/<member>-<usecase>`.
- One PR per substantial use case, include:
  - Description & endpoints implemented
  - DB migration file(s)
  - Test summary and how coverage was measured
  - Postman collection / curl examples
- Peer review: at least one other member reviews PR before merge.
- Group report: coordinate critique/improvements and attach API and UI screenshots.

---

9) What to submit (per assignment rubric)

- **Group report**: critique + suggested improvements with updated UML and UI changes (align changes I listed earlier). Each change must be justified and referenced.
- **Individual code**: one use case per member (full implementation, tests ≥80% of their functionality).
- **Unit & integration tests**: include coverage reports (Jacoco HTML).
- **Demo**: short video or screenshots showing the implemented use case working.
- **API docs & Postman**: included in repo.