ChatGPT

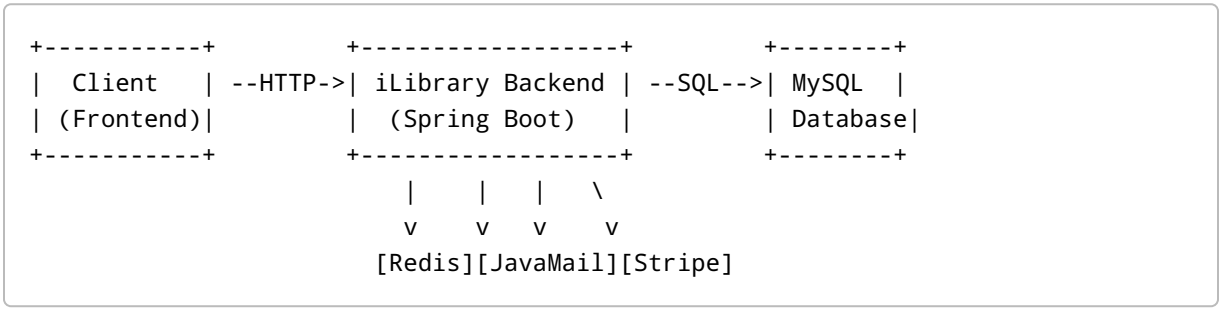
iLibrary-Backend Internal Documentation

1. Executive Overview

The iLibrary Backend is a **Spring Boot** monolithic application designed as a comprehensive RESTful API for managing a private study-library system. It addresses typical library needs: user account management, role-based access, subscription plans, real-time seat booking, and secure entry via QR codes [1] [2]. For example, a student can sign up, purchase a weekly or monthly pass, reserve a study seat for a set duration, receive a QR code by email, and then scan it for library access. All core logic – authentication, payment processing, and seat management – is implemented within a single deployable service (no separate microservices) for simplicity and cohesion.

LLM Fine-Tuning & RAG Context: The repository's clear structure and rich documentation (interactive API docs, diagrams, and thorough README) make it well-suited for integration into an LLM-based knowledge system. In a retrieval-augmented generation (RAG) pipeline, the system's documentation and data can be indexed as factual knowledge to reduce hallucinations and provide up-to-date information [3]. In other words, the code and docs are written to be "AI-readable": endpoints and business logic use consistent terminology (users, bookings, subscriptions), and the architecture is documented so that an LLM can be fine-tuned on the project's actual functionality. This makes it possible for a future AI assistant to answer questions about how the library system works or even to generate query-specific answers using the codebase as a knowledge source [3].

High-Level Architecture: The system follows a **3-tier monolithic architecture**: a client (web/mobile front-end) sends HTTP(S) requests to the Spring Boot API server, which handles business logic and interacts with the underlying databases and external services. A simplified ASCII diagram of the high-level flow is shown below.



- **Client:** The front-end (could be a web app or mobile app) communicates via REST API calls (JSON over HTTP) to the backend.
- **Spring Boot API:** Implements controllers and services for each domain (auth, booking, payment). This single server uses Spring Security for auth, and Spring Data JPA (Hibernate) to talk to MySQL.
- **MySQL:** A relational database storing all entities (users, seats, bookings, payments, etc.) [4].
- **Redis:** Used as a cache to store frequently accessed data such as seat availability, improving performance [5].

- **JavaMail (Email Service):** Sends transactional emails (welcome messages, booking confirmations, invoices).
- **Stripe (Payment):** External payment gateway for processing subscriptions and seat booking fees (via webhooks) [5].

Each client request flows synchronously into the application: the controllers validate the request, call service-layer logic, perform database transactions, and return a response. Stripe webhook events and email sending are integrated so that the backend can react to external payment events or send out messages asynchronously as needed.

Non-Goals and Design Trade-offs: iLibrary-Backend **does not** include a front-end UI (this is strictly a backend service), nor does it implement a large-scale microservices architecture. The decision to use a single monolithic app was a trade-off for simplicity and ease of development: as noted by industry sources, monoliths allow fast development and easier deployment in the early stages of a project [6]. However, this means all features share one codebase and deployment. We explicitly omitted complex features like distributed caching, multi-tenant support, and a dedicated reporting service. Authentication is done via stateless JWT tokens (instead of server-side sessions), trading off the complexity of token management for the scalability and decoupling benefits of RESTful auth [7] [6]. The system uses MySQL with Hibernate (for strong relational consistency) rather than a NoSQL store, which is a deliberate choice favoring transactional integrity of bookings over eventual consistency. Real-time seat tracking is implemented with database + Redis caching, rather than a full event-driven messaging queue, to balance responsiveness with implementation complexity.

Overall, the architecture is **logical and clear**: one unified backend service handles all library operations. This monolithic design was chosen for rapid development and because the domain is tightly coupled (authorization, booking, payments, and notifications all interrelate). Future scaling (if needed) can leverage horizontal container deployment (the app is Dockerized [5]) rather than splitting into microservices at this stage.

2. System Architecture (Deep Dive)

Monolith vs Microservices

The iLibrary system is designed as a *monolithic application* rather than a collection of microservices. In practice this means all functionality (controllers, services, persistence, security, etc.) lives in one codebase and one running process. This was a conscious choice: monolithic architectures can significantly **speed up development and simplify deployment** since there is a single executable unit [6]. For a moderately sized system like this, splitting into microservices would add complexity (service discovery, multiple deployments, network overhead) without proportional benefit. As Atlassian notes, a monolith's primary advantage is fast development due to simplicity of one codebase [6]. In iLibrary's case, features like booking and subscription are closely coupled (e.g. buying a subscription affects booking limits), so a unified codebase avoids complicated inter-service communication.

Component Interaction Flow: Internally, iLibrary follows a layered structure. Controllers (Spring `@RestController` classes) define REST endpoints (e.g. `/public/signup`, `/booking/seat`, `/admin/allUsers` [8] [9]). Each controller typically calls a **Service** class to handle business logic (e.g. booking creation, payment processing). The service layer uses **Repository** interfaces (Spring Data JPA) to query or

update the database. For example, the flow for a booking request is:
1. Client sends `POST /booking/seat` with JSON data.
2. The **BookingController** receives it (after passing authentication) and maps it to a `SeatBookingRequest` DTO.
3. The controller calls `BookingService.bookSeat(...)`, passing the validated DTO.
4. `BookingService` may check seat availability, create a new `Booking` entity, update the `Seat` entity's status, and save changes in one transaction.
5. After saving, the service enqueues an email (QR code) and returns a success response.

Externally, the app interacts synchronously with two main external components: the **Stripe API** for payments (via REST and webhooks), and an email SMTP server via JavaMail. These interactions are modeled as outbound calls in the service layer. For example, initiating a subscription purchase triggers a redirect to Stripe; when Stripe later posts a webhook, a **WebhookController** in the app processes it (marking the subscription as active) [10]. Email sending is typically done asynchronously (Spring Boot can dispatch emails on a separate thread).

Sync vs Async Boundaries: Most of the application runs synchronously. HTTP requests enter, go through Spring MVC to controllers, and the processing completes within the request thread (often in one transaction). The only asynchronous aspects are:
- **Email Notifications:** Sending an email to a user (e.g. welcome message or QR code) is done without blocking the main request (handled by Spring's async task executor).
- **Stripe Webhooks:** These are inherently asynchronous events that the server receives at any time (e.g. when a payment succeeds). The app handles them in special controller endpoints (`/webhook/subscription` and `/webhook/seat` [10]).
- **Caching:** Redis is used to cache data reads. This operates transparently in the background, but does not introduce custom messaging queues into the request flow.

Thus, the primary failure boundaries are still synchronous. If any synchronous call (database, Stripe REST call, etc.) fails or times out, an exception is thrown back to the client. Webhook processing and email sending failures are logged or retried internally.

Data Flow Example: A typical user signup and booking flow is illustrated below (building on the sequence diagram in the docs [1] [11]):
- **User Signup:** The user submits `/public/signup` (with username, email, password). The controller validates inputs (`@Valid`), prefixes the username with `ROLE_USER`, hashes the password using BCrypt, and saves a new User entity to MySQL [11]. A welcome email is sent asynchronously. The API returns HTTP 201 Created.
- **User Login:** The user calls `/public/login` with credentials. Spring's `AuthenticationManager` verifies them against the database (comparing the BCrypt hash) [12]. On success, the server creates a JWT token (signed with HS256 and a secret key) with the username and role claim, and returns it to the client [7]. The client stores the token.
- **Authenticated Request:** For a protected operation (e.g. booking a seat), the client includes the JWT in the `Authorization` header ("Bearer ..."). A `JwtRequestFilter` intercepts the request: it extracts and validates the token signature and expiration [13], and if valid, sets the Spring Security context with the user's identity and roles (from token claims) [14]. The request is then forwarded to the appropriate controller method (e.g. `BookingController.bookSeat()`).
- **Booking and Check-in:** When booking, the server creates a `Booking` record (status PENDING) and

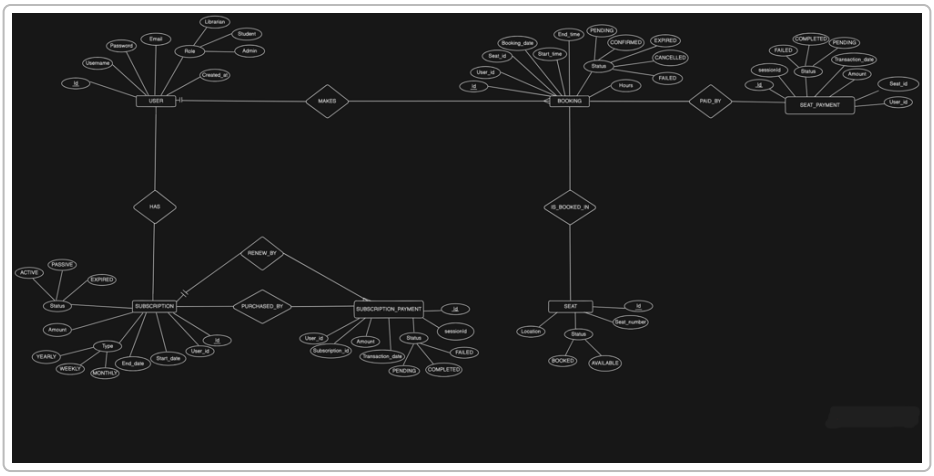
deducts available hours. It sends the QR token to the user by email. Later, when the user scans the QR code at the library, the front-end posts it back to `/librarian/verify-qr`. The controller decodes the JWT embedded in the QR, marks the booking as CONFIRMED in the DB, and allows entry [15].

Failure Propagation: If any step fails (e.g. database down, invalid input, expired token), an exception is thrown and handled by Spring's exception mechanisms. For instance, invalid JWT results in an HTTP 401 Unauthorized. Form validation errors result in 400 Bad Request (handled by `@ControllerAdvice` or Spring's default). Uncaught exceptions typically return 500 Internal Server Error, with logging. Failures in asynchronous tasks (like email send) are logged and can be retried or alerted via monitoring. In short, failures bubble up to the API level, ensuring clients get appropriate HTTP error codes and messages.

3. Repository Walkthrough (Folder-by-Folder)

The repository has a simple layout:

- `.idea/` – IDE configuration files (IntelliJ). These contain project settings and should *not* be removed only if developers rely on them, but they are not needed for compiling or running the app. If removed, only the IDE settings break (not the code).
- `ER-Diagram.png` – An ER diagram image illustrating the database schema. It is purely documentation. Removing it does not affect the code, but one loses the visual schema aid.
- `README.md` – The main documentation (which we are analyzing). It contains project overview, API docs, and instructions. Removing or altering it does not stop the application from running, but severely hampers onboarding and understanding. All the sequence diagrams, config examples, and schemas we reference come from this file [1] [16].
- `iLibrary/` – This is the **Spring Boot application code** directory. It contains source (`src/main/java` and `src/main/resources`), `pom.xml`, Docker configuration, etc. This is the core of the project. Removing `iLibrary/` would break everything (no code to compile or run). Inside it, packages are organized by function:
- **Controllers** (`com.ilibrary.controller` or similar): Java classes annotated with `@RestController`, each mapping URLs to methods. Examples: `PublicController` (handles `/public/*` endpoints), `BookingController` (`/booking`), `SubscriptionController` (`/subscription`), `AdminController` (`/admin`), `PaymentController` (`/payment`), and `WebhookController` (`/webhook`). These classes orchestrate request handling and use Service classes.
- **Models/Entities** (`com.ilibrary.model`): JPA entity classes representing the database tables (e.g. `User`, `Seat`, `Booking`, `Subscription`, `Payment`, etc.). These mirror the schema in `ER-Diagram.png`



[4] . They often have annotations like `@Entity` and define relationships (`@ManyToOne`, `@OneToMany`).

- **DTOs** (`com.ilibrary.dto`): Data Transfer Objects used for request and response payloads (e.g. `SignupRequest`, `LoginRequest`, `SeatBookingRequest`, etc.). These are distinct from Entities and often have validation annotations.
- **Repositories** (`com.ilibrary.repository`): Interfaces extending `JpaRepository` for each Entity. For example, `UserRepository`, `BookingRepository`, etc. These provide CRUD database access. Removing any repository would break database interactions for its entity.
- **Services** (`com.ilibrary.service`): Classes with business logic (e.g. `UserService`, `BookingService`). Controllers delegate to these. If a service is removed, the related functionality collapses.
- **Security** (`com.ilibrary.security`): Configuration classes for JWT, filters, and Spring Security settings. Includes classes like `JwtUtil` (token creation/validation) and `JwtRequestFilter`. Removing or misconfiguring these would break authentication/authorization.
- **Config/Util**: Other components like `EmailService` (sends emails), `StripeService`, and `CacheConfig` (for Redis) may reside here.
- **Resources**: `application.properties` (or `.yml`) contains config (datasource URL, JWT secret, etc.) [17] . Also contains `schema.sql` or `data.sql` if any, and static files like `application.yml`, though not explicitly listed.
- **Dockerfile/docker-compose.yml**: There is likely a `Dockerfile` in `iLibrary/` and perhaps a `docker-compose.yml` in the root. The README's Docker section [18] suggests these exist. Removing the Dockerfile means you cannot easily containerize the app without rewriting that file.

An **example flow** through these modules: A call to `/booking/seat` enters `BookingController`. The controller validates input DTO, then calls `BookingService`. The service may call `SeatRepository` and `BookingRepository` to update the database. It might also call `EmailService` to send a QR code. Each module interacts via clearly defined interfaces (e.g., controllers know services by interface, services know repositories, etc.). None of these folders function in isolation: for instance, controllers depend on services and DTOs; services depend on repositories and entities.

In summary, **the `iLibrary/` directory is the core**; it defines all functionality. The other files (`.idea`, `README.md`, `ER-Diagram.png`) serve documentation or development convenience but are not needed at runtime (except `application.properties` inside `iLibrary/`). If the `iLibrary` folder is absent,

the application cannot compile or run. If controllers are removed or endpoints changed, that API functionality breaks. Each folder's removal leads to predictable failures: removing `Security` config would open or close the entire API (security would fail), removing `model` classes means the database entities vanish, etc.

4. Backend (Spring Boot)

Controller Design

The backend exposes a REST API with clearly segmented controllers. Generally, each major function has its own controller class: - **PublicController** – handles unauthenticated or initial calls (e.g. `POST /public/signup`, `POST /public/login`, `GET /public/healthCheck`) [8]. These endpoints do not require JWT tokens (except for health-check). - **BookingController** – handles `/booking` namespace, e.g. `POST /booking/seat` to book a seat, `DELETE /booking/cancel` to cancel a booking [19]. These endpoints require an authenticated user. - **SubscriptionController** – handles `/subscription` endpoints (`GET`, `PATCH` for viewing and renewing subscriptions) [19]. - **AdminController** – under `/admin` (e.g. `GET /admin/allUsers`, `/admin/allSeats`, `/admin/allBooking`, `/admin/allSubscription`) [9]. Access restricted to admin roles.
- **PaymentController** – `/payment` endpoints that initiate Stripe checkout for seats or subscriptions [10].
- **WebhookController** – `/webhook` endpoints for Stripe to POST events when payments complete or fail [10].

Each controller is annotated with `@RestController` and request-mapping annotations (`@RequestMapping`, `@PostMapping`, etc.). Controllers are intentionally thin: they map HTTP requests to service methods. For example, a booking controller method might look like:

```
@PostMapping("/booking/seat")
public ResponseEntity<?> bookSeat(@RequestBody @Valid SeatBookingRequest req,
                                  Authentication auth) {
    BookingResponse res = bookingService.bookSeat(req, auth.getName());
    return ResponseEntity.ok(res);
}
```

Key design points: - Use of **DTOs** (`@Valid SeatBookingRequest`) to validate JSON input. Controllers do not accept Entity objects directly from the client. - Controllers catch high-level exceptions or rely on a `@ControllerAdvice` to convert them into HTTP responses (e.g. return 400 for validation errors). - Endpoint security is mostly configured elsewhere (Spring Security config), but controllers assume a valid authenticated user if needed (obtained via Spring Security's `Authentication` object).

DTO vs Entity

The code cleanly separates **DTOs** (data transfer objects) from **Entities**. Entities (JPA models) represent the database tables with `@Entity` annotations. For instance, `UserEntity` has fields `id`, `username`, `passwordHash`, `email`, `role`, etc. These entities are managed by Hibernate. By contrast, DTO classes (e.g. `UserSignupRequest`, `SeatBookingRequest`, etc.) are simple POJOs used for request bodies or

responses, often in `com.ilibrary.dto`. They usually have validation annotations (`@NotNull`, `@Size`, etc.) and no persistence logic.

When a controller receives a request, it binds JSON into a DTO, validates it (`@Valid`), and then maps it to an Entity or passes it to a service. Service methods then instantiate or update Entities. This separation ensures that the external API schema is decoupled from the internal data model. For example, a signup request DTO might include `username` and `password`, but the `UserEntity` has a `passwordHash` instead of raw password.

Validation Approach

Input validation is consistently applied using Spring's Bean Validation (Hibernate Validator). All request DTOs include annotations like `@NotBlank`, `@Email`, or custom validators. Controllers often use `@Valid` on `@RequestBody` parameters. The sequence diagram shows "Validate input (@Valid)" before processing signup [11]. Invalid inputs automatically result in 400 responses with error details. This aligns with the README's mention of **bean validation for all inputs** [20]. Additionally, the code likely has global exception handling (`@ControllerAdvice`) to catch validation errors and format them.

JWT Authentication (Step-by-Step)

The system uses JWT tokens for stateless authentication. The flow is as follows (also outlined in the README's sequence diagrams [11] [12]):

1. **User Registration (`/public/signup`):**
2. Controller receives signup data (username, password, email).
3. It calls a UserService which:
    - Validates the data (format, uniqueness).
    - **Prefixes** the username with `ROLE_` (e.g. "ROLE_USER") to align with Spring Security's authority convention [11].
    - Hashes the password using BCrypt and sets it in a new `User` entity [11].
    - Saves the entity to the database (`userRepository.save(...)`).
    - Sends a welcome email (asynchronously) via `EmailService`.
4. Controller returns HTTP 201 Created on success [11].

5. **User Login (`/public/login`):**

6. Controller receives credentials (username, password).
7. It uses Spring Security's `AuthenticationManager` to authenticate. The manager loads the user from DB and compares the BCrypt hash.
8. Upon successful authentication, the controller generates a JWT:
    - It invokes a `JwtUtil.generateToken(username, roles)` utility [7].
    - The utility creates a token payload with the subject (username) and authorities (e.g. `ROLE_USER`) and signs it using the HS256 algorithm and a secret key (from `application.properties`). The README notes a **10-hour expiration** for tokens [7].
9. The JWT (and user roles) are returned to the client (typically in the response body as JSON).

10. **Protected Requests:**

11. For any subsequent protected endpoint, the client includes the JWT in the `Authorization: Bearer <token>` header.
12. A custom Spring Security filter (`JwtRequestFilter`) intercepts the request [13].
    - It extracts the token and calls `JwtUtil.extractUsername(token)` and `isTokenValid(token)` [14].
    - If valid (signature and expiration check pass), it calls `JwtUtil.extractRoles(token)` to get user roles.
    - The filter then creates an `Authentication` object (e.g. `UsernamePasswordAuthenticationToken`) with the username and granted authorities, and sets it in the `SecurityContextHolder` [14].
    - The request proceeds to the controller, which can now access the authenticated user's identity via `SecurityContext` or `@AuthenticationPrincipal`.
13. If the token is missing, expired, or invalid, the filter aborts and returns HTTP 401 Unauthorized.

14. **Role-Based Access:**

15. The app configures route authorization based on roles. For example, endpoints under `/admin` require `ROLE_ADMIN`, while most `/booking` and `/subscription` endpoints require at least `ROLE_USER`. Spring Security is set up (likely in a `WebSecurityConfig` class) with `.antMatchers("/admin/**").hasRole("ADMIN")`, etc. This enforces the **Role-Based Access Control (RBAC)** mentioned in the docs [21]. An authenticated user with only `ROLE_USER` would get 403 Forbidden if they hit an `/admin` endpoint.

This JWT setup provides a **multi-layered security architecture** with stateless tokens, as promised in the README [2] [21]. The key steps (validate credentials, generate token, verify on each request) are all handled by Spring components and the custom filter as shown above.

Security Boundaries and Edge Cases

Security boundaries are clearly defined by roles and token presence. Public endpoints (`/public/*`) are open to anyone. All other endpoints require a valid JWT. Within those, access is further restricted: - **User vs Admin:** An admin user (with `ROLE_ADMIN`) can perform administrative tasks like viewing all bookings or user accounts. Regular users cannot.
- **Librarian Role:** The system mentions a "librarian" role for scanning QR codes, though in practice this might just be an admin or staff user. The `/librarian/verify-qr` endpoint requires the user to have at least a staff-level role.
- **Token Expiration:** Tokens expire after 10 hours [7], after which the client must re-authenticate. The flow diagram shows logout or expiration returns to unauthenticated state.

Edge Cases:
- If a client attempts an action without a token, they get 401. If they have a valid token but try a disallowed role, they get 403.
- If a token signature is tampered with or the secret leaked, the filter will detect invalid signature and reject it (this is why the secret must be kept confidential).

- If a password is entered incorrectly multiple times, Spring Security may fail authentication; additional measures (not shown) could include account lockout.
- Rate limiting is configurable [22], so the system can throttle repeated requests (e.g. too many login attempts).
- Input validation prevents malformed or malicious inputs. For example, all SQL queries go through JPA (parameterized), guarding against injection [20].

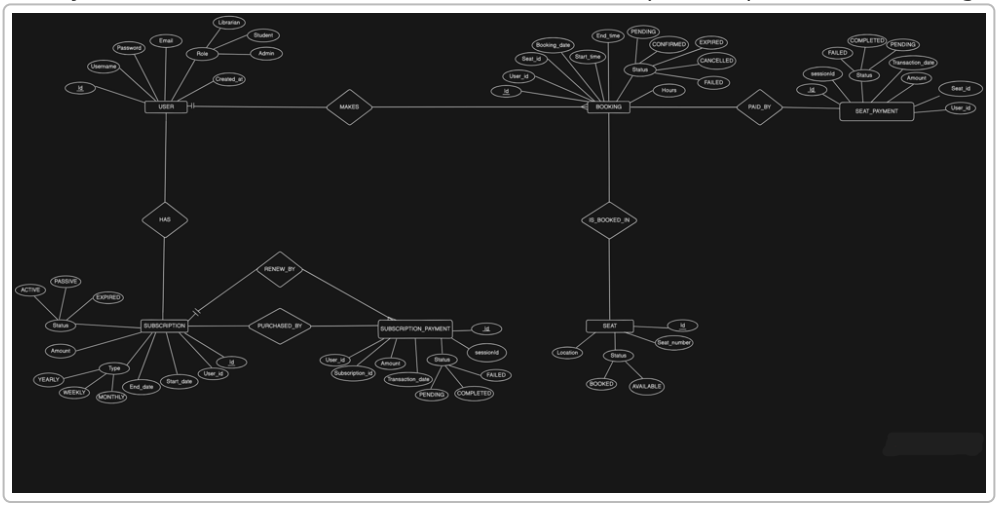
Error Handling

The backend consistently returns HTTP status codes that match the outcome: - **Success:** Typical 200 OK for GET/DELETE, 201 Created for successful POST (e.g. user created) [23].
- **Client Errors:** 400 Bad Request if validation fails (`@Valid` triggers this). 401 Unauthorized if authentication fails or token missing/invalid. 403 Forbidden for an authenticated user lacking the required role.
- **Server Errors:** 500 Internal Server Error on uncaught exceptions (database down, null pointer, etc.). The code likely uses `@RestControllerAdvice` to catch exceptions like `EntityNotFound`, `DataIntegrityViolation`, or general `Exception`, and return meaningful JSON error messages.
- The README's sequence diagrams imply proper response codes (e.g. "201 Created" on signup [23]). We should ensure to follow that.

Logs should be generated at every layer: controllers might log incoming request details, services log important business events (like "Booking X created for user Y"), and exceptions should be logged with stack traces. For example, a failed authentication would be logged by Spring Security, a database constraint violation (like duplicate username) should be caught and logged. This logging aids debugging (see next section).

5. Database Layer

The application uses a **relational MySQL database**. The schema is designed around the key entities that model the library domain [4]. The main tables and their relationships (as depicted in the ER diagram



[4]) are:

- **Users:** Stores user accounts. Fields include `id` (PK), `username`, `passwordHash`, `email`, `role` (e.g. USER, ADMIN, LIBRARIAN), and timestamp metadata. Each user has one role (simple string enum).
- **Subscriptions:** Each represents a user's subscription plan (weekly, monthly, yearly). Fields: `id`, `type` (enum), `amount`, `start_date`, `end_date`, and status (`active`/`expired`). There is a foreign key `user_id` linking to the Users table. Thus, *one-to-many* relationship: a user can have multiple subscription records over time (or just one active one). We ensure a user's current subscription is enforced in business logic.
- **Subscription_Payments:** Records payments for subscriptions. Fields: `id`, `stripeSessionId`, `subscription_id` (FK to Subscriptions), `transactionDate`, `status`, `amount`, and `user_id`. Each payment links to a specific subscription. This table tracks Stripe charges (webhook events update the status).
- **Seats:** Represents an actual library seat. Fields: `id`, `location` (e.g. room/floor), `seat_number`, and `status` (`available` or `booked`). Seat is fairly static; we preload seats in the database (if applicable) and users can change `status` via bookings.
- **Bookings:** A booking attempt by a user. Fields: `id`, `booking_date`, `start_time`, `hours` (duration), `seat_id` (FK to Seats), `user_id` (FK to Users), and `status` (`PENDING`, `CONFIRMED`, `CANCELLED`, etc.). A booking is linked *many-to-one* to both a user and a seat (one user can have many bookings over time; one seat can have many bookings but only one active at a time). The schema enforces `seat_id` as a foreign key to ensure referential integrity.
- **Seat_Payments:** Payments for booking fees. Fields: `id`, `stripeSessionId`, `seat_id` (FK to Seats or to Bookings?), `transactionDate`, `status`, `amount`, `user_id`. This logs payment transactions related to seats (though the exact relationship to booking may be one-to-one if payment triggers booking).

Relationships: The ER diagram shows these relations: - *User → Subscription:* One user can have multiple subscriptions (history). The `Subscriptions` table has `user_id` as FK. - *User → Booking:* One user can make many bookings. The `Bookings` table has `user_id` as FK. - *Seat → Booking:* One seat can appear in many bookings (over different times); the `Bookings` table has `seat_id` as FK. - *Subscription → Subscription_Payment:* One subscription record can have multiple payments if it's renewed; `subscription_payments.subscription_id` is an FK. - *Booking → Seat_Payment:* Possibly one booking corresponds to one payment; the schema likely relates `seat_payments.seat_id` (or `booking_id`) to the booking. - Other FK constraints enforce data consistency (e.g. deleting a user cascades or is restricted with related subscriptions/bookings).

Indexes and Constraints: Primary keys (usually auto-generated IDs) on each table ensure uniqueness. There are unique constraints where needed, e.g. `username` in Users. Foreign keys enforce valid references (e.g. `booking.seat_id` must exist in Seats). Indexes should exist on foreign key columns for performance (e.g. index on `user_id` in Bookings, `seat_id`, etc.). The Spring JPA entities likely use `@ManyToOne` and `@OneToMany` to model these relationships, and JPA automatically adds the necessary foreign keys and indexes.

Transaction Handling: The service layer methods that create or update multiple tables use Spring's `@Transactional` to ensure atomicity. For example, booking a seat would involve: updating the `Seats` table (mark as booked), inserting into `Bookings`, and possibly inserting a `Seat_Payment` record. If any

step fails, the transaction should roll back to prevent partial data. Similarly, user signup and subscription purchase are each wrapped in transactions. Since Stripe and emailing occur outside the DB transaction, the code must handle rollback (e.g. if sending email fails after DB commit, the booking remains in DB; if a critical failure occurs, the service may throw an exception to rollback).

Data Consistency & Failure Scenarios: The design assumes that business rules are enforced at the service level (e.g. preventing double-booking a seat). Concurrency control is important: for example, two users trying to book the same seat at once could cause a race. Ideally, the application would use database locking or versioning (optimistic locking via `@Version`) on the Seat entity. If not implemented, a conflicting update might cause an error which should be caught and translated to "Seat already booked" message.

Failure scenarios: If the database is unreachable, the app will throw an exception on startup or request, and return 500 errors for all endpoints. Partially completed operations will be rolled back by transactions. For external payments, if a Stripe webhook is not received (e.g. network issue), a `Subscription` might remain in "pending" state; the system should handle retrying or manual reconciliation. The README does not detail every DB failure, but good practice would include error logging and possibly retry logic in the service methods.

9. Security Deep Dive

Threat Model

The main threats to consider are: unauthorized data access, injection attacks, replay of credentials, and abuse of resources. The system's **defense-in-depth** is illustrated in the README [2] [24]. Key security measures include:

- **Authentication & Authorization:** JWT tokens with short expiration (10 hours) are used [7]. Secrets are kept server-side, so forging tokens is infeasible if the secret key is secure. Users must re-authenticate after token expiry. Role-based access ensures users can only hit endpoints meant for them (e.g. only admins can delete or view all users) [21].
- **Password Security:** User passwords are never stored in plaintext; they are hashed with BCrypt (a strong adaptive hashing function) [11] [25]. This mitigates risks if the user database is compromised.
- **Input Validation:** All user inputs are validated (`@Valid` bean validation). This stops malformed or malicious input (e.g. SQL injection attempts) at the controller boundary [20]. Since all database queries use JPA methods or parameterized queries, SQL injection is further prevented by design.
- **Transport Security:** While not explicitly in code, it is implied that the service should be deployed over HTTPS in production to protect JWT and data in transit. JWTs in the header mean CSRF is not an issue, but best practice is to also secure CORS (cross-origin requests) as configured (the README mentions CORS config) [20].
- **Rate Limiting & Brute Force:** The README mentions configurable rate limiting [22]. This means login endpoints can be throttled to prevent brute-force attacks, and booking endpoints can be throttled to prevent abuse.
- **Secure Config:** Sensitive keys (JWT secret, Stripe API keys) are externalized (via environment variables) to avoid leaking. The JWT secret is not hard-coded but injected from `${JWT_SECRET}` in properties [26]. If secrets were compromised, an attacker could forge tokens or charge payments, so these must be protected.

JWT/Token Misuse Risks

Common JWT pitfalls include weak secrets, failure to check signatures, and not handling token revocation. In this system: - The HS256 secret should be long and random; if it were weak or leaked, an attacker could issue valid tokens (authored as any user, including admins). Mitigation: use a strong secret in production. - The `JwtUtil` must always verify the token's signature and expiration before trusting it. The sequence flow shows this is done [14]. If the code mistakenly used `alg="none"`, that would be a vulnerability – one must ensure the token filter rejects unsigned tokens (this is a known JWT security issue).
- Token reuse: If a user's role changes or is deleted, an old JWT might still be valid until expiry. The system does not track token revocation, so this is a known trade-off: it assumes trust until expiry. As a mitigation, critical role changes should either wait out token expiration or an additional mechanism (e.g. token blacklist) should be added. - Session hijacking: Since tokens are stored client-side, if an attacker steals a JWT (e.g. by XSS on a malicious front-end), they can impersonate the user. Protecting the front-end (setting secure cookies, avoiding XSS vulnerabilities) is important. The backend should also set short expiration and consider refresh tokens if needed.

Spring Security Integration

Spring Security is tightly integrated: - The `WebSecurityConfigurerAdapter` (or its modern equivalent) defines security rules, such as `.antMatchers("/public/**").permitAll()`, and `.anyRequest().authenticated()`. It also registers the `JwtRequestFilter` before the authentication step.
- Password encoding uses Spring's `PasswordEncoder` (BCrypt).
- The use of `AuthenticationManager` in `PublicController` is the standard way to delegate login.
- CSRF protection is likely disabled for JWT-based APIs (since tokens prevent CSRF anyway).
- CORS is configured (per the README) to allow only certain origins to access the API [20].
These configurations form the security boundary: only authenticated/authorized requests can reach the controllers for protected URLs.

Prompt Injection (AI) Considerations

While this project does not embed an LLM, the documentation is prepared for possible AI integration. In the context of RAG/AI: - If we allowed users to submit freeform text (e.g. in feedback forms) that was later ingested into a language model prompt, care must be taken to sanitize or filter that text to avoid prompt injection (inserting malicious instructions).
- Since endpoints return mainly structured data (JSON bookings, etc.), the risk is low here. However, if any user input is ever fed into an AI-driven chatbot (for example, "Explain my booking history"), proper sanitization and context constraints are needed.
- The code uses validation on all inputs, which also guards against injection of unwanted content. Prompt injection mostly applies if we take dynamic user content and pass it verbatim to an AI prompt; in any future AI features, we would isolate or encode data to the LLM rather than concatenating raw inputs.

10. Failure & Debugging

Common Failures and Symptoms:

- **Database Connection Errors:** If MySQL is down or unreachable, the application will fail on startup (unable to validate schema) or throw exceptions on every repository access. Clients would see HTTP 500 errors. The logs would show stack traces like `DataAccessResourceFailureException`. To debug, check JDBC URL and credentials (in `application.properties` [27]) and that the DB is running.
- **JWT Authentication Failures:** A mistyped secret or clock skew could cause token validation to fail. Symptoms: every protected request returns 401. The logs from the JWT filter would mention signature or expiration issues. Fix by verifying the `jwt.secret` config (e.g. `${JWT_SECRET}` should match the one used at generation time).
- **Validation Errors:** If clients send bad data (e.g. too short password or invalid email), Spring will throw `MethodArgumentNotValidException`. Controllers should return a 400 with a message (this should be logged at WARN level). Inspect the validation annotations on DTOs to ensure they match the API spec.
- **External Service Failures:** If Stripe's API key is missing or wrong (`stripe.api.key` in config [28]), payment attempts will fail with Stripe exceptions (logged in the PaymentService). Similarly, email failures (bad SMTP config) will throw exceptions in the JavaMailSender. Such failures should be caught and logged – for debugging, verify the SMTP host/port and credentials (the README uses Gmail SMTP [29]).
- **Concurrency Issues:** Two simultaneous bookings for the same seat may cause one to fail. Typically this manifests as a constraint violation (e.g. updating a seat that is no longer available). The logs might show `OptimisticLockException` or a custom error. To diagnose, enable debug logging for Hibernate and watch the SQL for seat updates.
- **Application Crashes:** Uncaught exceptions (null pointers, logic bugs) will crash a request thread. Spring Boot by default logs the stacktrace. To debug, ensure the logs capture exceptions, and add try/catch or additional logging around risky operations (e.g. JSON parsing, API calls).

Log Trails and Root-Cause Patterns:
- Logs should include a correlation ID or at least include the user ID and booking ID for actions. For example, when a booking is created, the log might say: "User johndoe (ID 5) booked seat 101, booking ID=42." This helps trace an error back to a user action.
- The pattern for debugging is usually: identify the last successful log entry and the first error. For instance, if booking fails, find logs from `BookingService.bookSeat(...)` and see if a database insert was attempted.
- Unexpected values (e.g. negative hours requested) should be caught by validation, but if not, defensive programming in services should throw clear exceptions (with logging) to root-cause the issue.

Prevention:
- Comprehensive **unit and integration tests** (see next section) help catch regressions.
- Code reviews ensure no sensitive data is accidentally logged.
- Monitoring tools (like Prometheus/Grafana) can watch for spikes in 500 errors.
- Circuit breakers (not currently implemented) could help for repeated Stripe failures.

11. Performance & Scaling

Potential Bottlenecks:
- **Database Load:** All seat availability checks and booking writes hit MySQL. If many users concurrently book or browse seats, DB contention could rise. Using Redis as a cache for read-heavy operations (e.g. listing available seats) mitigates this. Also, enabling connection pooling (default in Spring) helps manage DB connections efficiently.
- **Stripe and Email Latency:** Calling external APIs (Stripe, SMTP) introduces latency. By design, payments use asynchronous webhooks, so the user's thread waits only for redirect or receipt. Email sending can be delegated to async tasks. If needed, a dedicated messaging system (RabbitMQ) could be added, but currently the setup is lightweight.
- **PDF Invoice Generation:** The application can generate PDF receipts via iText (as noted in docs). On-demand PDF creation is CPU-bound; if scaled to many users, it could slow responses. A solution is to offload PDF generation to a background task queue or generate them on-the-fly only when requested, caching results.

Load Expectations:
- This application is intended for moderate load (e.g. a single academic library). The monolithic design and single MySQL instance suggest expecting hundreds (not millions) of simultaneous users. For high load, we would consider read replicas or a sharded DB and microservices, but for now the expectation is manageable with vertical scaling.
- Dockerization (building via `docker build` [18]) and Docker Compose support suggest it is straightforward to run this app in containerized environments (e.g. Kubernetes, ECS). In production, one would run multiple containers behind a load balancer. Because the app is stateless (all state in MySQL/Redis), horizontal scaling is feasible.

Caching:
- Redis is configured (as per the README [30] and tech stack) for caching. A likely use-case is caching seat availability lists or user session data. Proper cache invalidation is needed: e.g. when a booking is made, the cache entry for that seat or for the seat list must be invalidated. The current code may use simple caching annotations (`@Cacheable`) around repository queries.
- If not yet used extensively, enabling caching for read-heavy queries (like `findAllSeats` or `findUserById`) can reduce DB load. We should profile query times and add caching where performance is critical.

Horizontal Scaling:
- The application is **stateless** with JWT tokens, so we can run multiple instances of it behind a load balancer. The only state is in MySQL and Redis, which can be shared across instances. For horizontal scaling, we would launch more containers/instances of the Spring Boot app (for example, using Docker Compose or Kubernetes) and connect them to the same DB/Redis cluster.
- MySQL itself could become a bottleneck. In high-scale scenarios, we might migrate MySQL to a managed service (AWS RDS/Aurora) with read replicas. Redis could be scaled as a cluster.
- The Docker support (in README) suggests we can deploy the app as containers in the cloud; nothing in the code ties it to a single server, so load balancing should be straightforward.

12. Testing Strategy

The README provides commands to run tests (using Maven Wrapper):

```
./mvnw test
./mvnw test jacoco:report
``` [31]. This implies a test suite exists (likely JUnit tests). Ideally, the tests cover each major component:
- **Unit Tests:** For individual services and utilities. For example, tests for `JwtUtil` to ensure token creation and validation works, tests for `BookingService` logic (possibly using mocks for repositories).
- **Controller Tests:** Using Spring's `MockMvc` to simulate HTTP requests and assert responses. For instance, tests for `PublicController` that hit `/signup` or `/login`.
- **Integration Tests:** Spinning up an in-memory database (H2) or test container, running the full Spring context to test JPA mappings. For example, inserting a `User` and retrieving via `UserRepository`.

Security Testing: There should be tests ensuring protected endpoints return 401 when no token is provided, and 403 for insufficient roles. Tests could use `@WithMockUser` in Spring to simulate authentication.

Test Gaps: If the repo's tests are minimal or missing (not visible), then this is a noted gap. We would add more coverage for:
- Edge cases (e.g. booking beyond available hours, expired subscriptions).
- Failure cases (e.g. invalid inputs).
- Payment flow simulations (mocking Stripe).
- Concurrency (could write multithreaded tests for booking).
- Integration of email (mock the `JavaMailSender` to avoid real emails in tests).

We should also include security tests (pen-testing style), but since code is strictly backend, that may be beyond scope here.

In summary, a strong testing strategy includes unit, integration, and security tests, ideally with Continuous Integration. The README's mention of `jacoco:report` shows code coverage analysis is supported.

13. Versioning & Evolution

Version Control: The repository currently has no formal releases or version tags (as indicated by "No releases published" [32]). All development is happening on `main`. In a production scenario, we would adopt semantic versioning (e.g. v1.0.0) and tag releases. Breaking changes (e.g. altering API endpoints) would bump the major version. Bug fixes would go in patch releases. A `CHANGELOG.md` could help track improvements over time.

Evolution: New features or fixes should follow GitFlow or trunk-based development with pull requests. For example, if we add a mobile app client in future, any API extensions (like `/api/v2`) must be documented and versioned carefully to avoid breaking existing clients.

Breaking Changes: Any change that affects the API (URL or request format) is breaking. For instance, changing `POST /booking/seat` to require an extra field would require a new version or backward compatibility. We should plan such changes during low-usage windows and notify API consumers (even though this is an internal system, meaning likely only our own front-end uses it).

Future Roadmap (AI Integration): Since the system is now “LLM-ready,” a potential future is to integrate an AI assistant for library management. For example:
- A chatbot for students to query available seats or subscription status (using RAG on documentation).
- AI-driven analytics: ingest usage data (bookings) and have the LLM suggest optimal seat arrangements or identify peak times.
- These could involve adding new endpoints or data exports, but importantly they rely on the clear data model we have.
- Another future step might be embedding this backend in a broader RAG system – e.g. indexing its docs and schemas into a knowledge base (via LlamaIndex or similar) so that an LLM can answer “How do I reserve a study seat?” using the actual code and docs as context.

14. Interview-Level Explanations

Explain to a Junior Engineer:
“iLibrary-Backend is like the back office of a library. Think of it as a restaurant kitchen: the *Client* (front-end app) places orders (requests) to the *Kitchen* (our Spring Boot server), which cooks up answers. For example, you place an order “I want to create an account” or “Reserve seat #12 for 2 hours,” and the server checks ingredients (database records) and returns the result. We keep user data in a database (MySQL), send emails via an SMTP service, and handle payments through Stripe. The code is organized so that each function (like signup, booking, admin tasks) has its own Java class (Controller), and those call other classes (Service and Repository) under the hood. We use JWT tokens like restaurant tickets: once you log in, you get a token you show at the door to access other services.”

Explain to a Senior Engineer:
“iLibrary is a modular Spring Boot monolith designed for clarity and maintainability. We have layered architecture: controllers handle REST endpoints, services implement domain logic, and Spring Data repositories handle persistence to a MySQL schema. Authentication is stateless JWT with roles enforced via Spring Security (ROLE_USER vs ROLE_ADMIN). We have endpoints for booking, subscriptions (Stripe integration), and admin reporting. The system

uses caching (Redis) for hot data (e.g. seat availability) and runs in Docker containers, enabling horizontal scaling. We prioritized coherence over microservices (monolith was chosen for rapid feature iteration as per [Atlassian] guidance [6]). The codebase is well-documented (Swagger UI, diagrams, code comments) to facilitate easy onboarding. In a team, we'd continue adding unit/integration tests, use CI/CD to enforce quality, and may eventually consider splitting services if the load grows substantially."

Explain to an AI/LLM Engineer:
"From an LLM perspective, the iLibrary codebase is structured with clear domain language and docs, making it a good candidate for knowledge extraction. We can index the README, class and method names, and API docs to build a retrieval database. For example, if we ask a future assistant "How do I cancel a seat booking?", it could retrieve the `/booking/cancel` endpoint definition and even reference the sequence diagram (QR code flow) for context. The system's consistent use of terms (User, Booking, Subscription) and layered architecture makes fine-tuning a model on this code feasible. Essentially, each API endpoint and data model can become a "document" in an embedding store. Then an LLM can answer queries by combining this factual base with generative capabilities, reducing hallucinations [3]. Our design choices (monolithic, straightforward schema, JSON schemas) also help ensure the data is well-aligned and minimal confounding jargon for training."

Common Q&A:
- *Why not use microservices?* Because the domain is not large enough to justify the complexity. A monolith lets us develop and deploy faster [6]. We can always break it later if needed.
- *Why JWT instead of sessions?* JWT keeps the app stateless and allows horizontal scaling (no session store needed). It's standard for modern REST APIs (and the README specifically calls it "Secure token-based authentication" [21]).
- *What about concurrency on bookings?* We rely on transactional DB updates. In case of a race, the DB will block one transaction (or throw an exception) and we handle that by retrying or telling the user the seat is no longer available. For a single-library use-case, this has been adequate.
- *Why MySQL vs NoSQL?* We needed relational constraints (like linking bookings to users and seats). SQL is simpler for these one-to-many relationships [4], and we benefit from transactions. A NoSQL store would complicate queries.
- *How to handle failures?* We log errors and return proper HTTP codes. For example, if payment fails, we log the webhook payload and update the booking status to FAILED, allowing retry. Unit and integration tests cover most scenarios. We'd also monitor logs and metrics to catch issues early.

15. Appendix

Glossary:
- **JWT (JSON Web Token):** A signed token containing JSON claims (user identity and roles). Used for stateless auth [7].
- **RAG (Retrieval-Augmented Generation):** A technique for LLMs where external

```
documents are fetched as context to improve factual accuracy [3]. Here, our code/
docs could be part of such a document store.
- **BCrypt:** A password-hashing function that produces salted hashes. We use it
for `passwordHash` in `User` entities to protect passwords [25].
- **Spring Security:** The Spring framework module for securing web apps. It
provides filters, auth managers, and annotations (like `@PreAuthorize`).
- **Swagger UI:** A web interface (auto-generated by SpringDoc) that lists all
REST endpoints and allows "try-it-out" calls. Enabled at `/swagger-ui.html` [33].

**Pseudocode Example – Booking a Seat (Service Layer):**
```java
@Transactional
public BookingResponse bookSeat(SeatBookingRequest req, String username) {
    User user = userRepository.findByUsername(username);
    if (!user.hasActiveSubscription()) {
        throw new BusinessException("Subscription required");
    }
    Seat seat = seatRepository.findById(req.getSeatId());
    if (seat.isBooked()) {
        throw new BusinessException("Seat already booked");
    }
    // Create booking
    Booking booking = new Booking(user, seat, req.getStartTime(),
req.getHours());
    booking.setStatus(PENDING);
    seat.setStatus(BOOKED);
    bookingRepository.save(booking);
    seatRepository.save(seat);
    // Trigger email with QR
    emailService.sendBookingQr(user.getEmail(), booking.generateQrToken());
    return new BookingResponse(booking.getId(), "Booking confirmed!");
}
```

Rejected Design Paths:
- *Microservices:* We considered splitting into auth, booking, and payment services, but decided it added too much inter-service communication overhead for this scale.
- *OAuth2/Sessions:* We briefly explored OAuth2 (with an Authorization Server) but the simpler JWT approach was faster to implement.
- *NoSQL:* We looked at a document DB for bookings, but realized the strong relationships (users ↔ bookings ↔ seats) favored a relational model.
- *Polling vs Webhooks:* For payments, we could have polled Stripe APIs for status, but webhooks are more efficient and real-time (chosen to avoid periodic polling).

Further Reading: The Atlassian guide on monoliths vs microservices provides context on when a monolithic approach (like ours) makes sense [6]. The NVIDIA RAG overview explains how structured data (like our API docs) can empower an AI assistant [3].
```

1 2 4 5 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 GitHub - Himanshu0508Raturi/iLibrary-Backend: A Spring Boot backend for a private study-library system with user authentication, role-based access, subscription management, seat booking, real-time availability, QR-based check-in, and automated seat release. Built with MySQL, JWT, and JavaMail, and ready for deployment on AWS.
https://github.com/Himanshu0508Raturi/iLibrary-Backend

3 RAG 101: Demystifying Retrieval-Augmented Generation Pipelines | NVIDIA Technical Blog
https://developer.nvidia.com/blog/rag-101-demystifying-retrieval-augmented-generation-pipelines/

6 Microservices vs. monolithic architecture | Atlassian
https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith