BACK-OFFICE SYSTEM FOR PUBLISHING MANAGEMENT

MULTIPLATFORM DEVELOPMENT WITH A MODERN, MAINTAINABLE AND SCALABLE ARCHITECTURE

Author: Francesc Ferrer Rubio
Version: 1.0.1
Date: January 20th, 2026

ABSTRACT

This project describes the design and implementation of a Back Office System (BOS) to manage a book publishing catalog. Applying a data-driven methodology, we defined a success strategy with measurable metrics. The technical solution is composed of an API specification contract, a relational Database (for books, authors and collections), a RESTful API server, and a native Android APP as frontend. The development was guided by agile principles throughout planning and execution.

INTRODUCTION

The project consists of the development of a complete and modern solution for the internal management of a publishing house. The system allows administrator users to perform CRUD operations on the core business entities: books, authors, and collections. Non-administrator users are limited to read-only operations.

The architecture is based on four main pillars:

1. **RESTful API specification contract**: A contract is established between the server and the client through the definition of an API specification using the OpenAPI syntax. A library containing the controller interfaces and DTOs is generated, packaged, and published to be consumed by the backend.
2. **Android Mobile Application**: The frontend of the project consists of a native Android application that serves as the user interface. It implements an authentication screen to secure access to the system. The interface is dynamic and adapts according to the user's role and the operations they are allowed to perform.
3. **RESTful API**: A backend that centralizes the business logic and acts as a bridge between the application and the database. It implements robust role-based authorization, and JWTs are generated by an AuthRepository that is decoupled from the rest of the ecosystem.
4. **Relational Database**: Where the information is structured and persisted within the system. Liquibase is used to automatically manage versioning and evolution of the database schema.

STATE OF THE ART

The state of the art in multiplatform application development can be analyzed from several perspectives:

From the **mobile frontend perspective**, there is a clear preference for native architectures in order to achieve better performance and user experience. For this reason, Kotlin with Jetpack Compose has been chosen for the frontend.

Regarding the **backend**, the standard approach is to use frameworks that improve development speed, security, and scalability. Kotlin, Gradle, and Spring Boot have been selected due to their balance between ecosystem maturity and language modernity.

Current trends in **software design** favor clean and decoupled architectures. For this reason, the Hexagonal Architecture, also known as Ports and Adapters, has been adopted. Establishing a contract is a robust approach that facilitates development and enables parallel work between frontend and backend teams. In REST API-based solutions, the strictest way to define such a contract is through an API-First approach.

Nowadays, development teams rely on **DevOps** practices such as continuous integration and continuous delivery (CI/CD), as well as environment virtualization, to ensure robust and consistent deployments. Docker has been chosen for containerization as it is a de facto standard, and GitHub Actions[1][2] has been selected for being modern, lightweight, easy to configure, and offering a user interface integrated directly into GitHub.

An effort has also been made to emulate a real-world working environment in the **organization** of the project. Although Jira was not used due to its cost and excessive complexity for a small project, agile methodologies were applied through the use of Scrumban, work tickets, and GitHub Projects.

[1] JetBrains. *How to Choose a CI/CD Tool: A Framework*. Bedrina, Olga. 2025-05-27. https://blog.jetbrains.com/teamcity/2023/08/how-to-choose-cicd-tool/

[2] NorthFlank. *GitHub Actions vs Jenkins (2025). Which CI/CD is right for you?* Emeni, Deborah. 2025-04-24. https://northflank.com/blog/github-actions-vs-jenkins

FEASIBILITY STUDY

The development of any software product requires a prior analysis to assess its chances of success. In the business world, traditionally linked to profit-driven objectives, a project must ensure a positive economic return, as well as technical and resource feasibility, and must also align with the company's overall strategy.
This feasibility study follows a structured and sequential reasoning process:

- **First**, the **Market Study** provides a macro-level overview, reviewing the most relevant economic, technical, and time-related aspects at a general level. This represents an initial approach to identifying potential risks and costs associated with the development.
- **Next**, the **SWOT Analysis** is applied to deepen the micro-level perspective, analyzing internal and external factors in order to define a concrete Strategic Plan aimed at increasing the chances of success.
- **Subsequently**, the plan is materialized through **Methodology and Success Metrics**, where a real working environment is simulated by establishing OKRs and KPIs, allowing the definition of measurable and objective milestones.
- **Finally**, based on all the previous information, the **Timeline Planning** is created. This realistic work schedule allocates time and effort to a series of specific milestones that must be executed within defined deadlines using the available resources.

Motivation

The present project arises from specific motivations and needs of an already existing company, **Editorial Denes**[3]. As a company with a low level of technological adoption and increasingly obsolete internal tools, it was necessary to modernize its systems.

The **single source of truth** for its catalog currently resides in a billing software from the 1990s, which presents a high usability barrier and is installed on legacy hardware running Windows XP as its operating system. There is no direct access to the database of this software; information can only be consulted through its user interface or by generating plain text (TXT) listings, which often contain errors and are difficult to process.

Although the company has had a website for approximately 20 years (which has evolved over time), it has always been based on very basic and limited templates. The website has suffered from several performance, security, and user experience issues. Due to the

[3] Editorial Denes. https://www.editorialdenes.com/

absence of a database, scaling and improving the current web solution (a Ruby and Jekyll-based template) is particularly challenging.
This situation imposes several limitations on users:

- **Lack of autonomy**: they depend on obsolete hardware to consult catalog information and rely on a developer to update website content.
- **Inability to scale**: inventory management and web presence are implemented as isolated systems that are neither integrated nor interconnected, limiting both improvement and growth.

The client expressed the need for a new solution to manage the data of their catalog that meets the following requirements:

- **Provide a single source of truth**: enabling full control over book, author, and collection data.
- **Guarantee self-sufficiency**: eliminating dependency on external software tied to physical hardware or recurring payments.
- **Ensure a low barrier to use**: allowing employees to use the system as easily as possible, with minimal (ideally zero) training.
- **Serve as a scalable foundation for future iterations**: enabling improvements such as inventory management, royalties administration, the development of a new website sharing the same data, or integration with distributors.
- **Be durable over time and comply with industry standards**: using a modern and future-proof programming language, a clean and decoupled architecture, and optimal performance.
- Be **ready for a production** environment **at zero cost**.

Market Study

The project is framed within the sector of content management systems (CMS) and enterprise back-office tools, a highly competitive market saturated with both generic and industry-specific solutions.

The opportunity does not lie in direct competition, but rather in demonstrating the ability to develop a tailored solution for a specific sector—the publishing industry—with full control over architecture, functionality, and security. This type of custom development is in demand by businesses with specific needs that standard solutions cannot optimally address, whether due to complexity (a small business does not require excessively powerful and comprehensive tools), suitability (standard solutions do not adapt to particular requirements), or economic factors (a simple, custom-built solution may be more cost-effective than a complex system with recurring payments or third-party dependencies).

This project arises precisely to address this opportunity: the development of a custom-built tool over which full control is maintained, and to which future complementary modules can be added.

This application has a real client: **Editorial Denes**, which currently requires a new database and a new content management system (CMS), and is seeking an agile and user-friendly solution that allows non-technical profiles to manage the catalog independently.

All of this makes the project a true **Minimum Viable Product (MVP)**, validated by a genuine business need and ensuring its use and maintenance beyond the completion of the academic program.

Technical and Economic Feasibility

The economic feasibility of the project is total, as it is based on the use of free or marginal-cost resources, typical of a modern development environment.

Hardware Resources

- **Development**: A personal computer with sufficient computing capacity is required to run a local environment (IDE, Android emulator, and Docker). This resource is already available to the author.
- **Production / Deployment**: Environments will be deployed on cloud services with free-tier plans such as Google Cloud Platform. The PostgreSQL[4] database will also be deployed on services offering free plans, such as NeonTech.

Software Resources

All software infrastructure is based on open-source technologies.

- **Development Tools**: Android Studio, IntelliJ IDEA Community
- **Languages and Frameworks**: Kotlin, Spring Boot, Jetpack Compose
- **DevOps Tools**: Docker, GitHub Actions, GitHub Packages
- **Database**: PostgreSQL, Liquibase
- **Project Management**: GitHub Projects

Human Resources

The project will be carried out by a single developer (the author), who will fully assume the following roles:

- **Analyst**: requirements analysis and architecture design
- **Designer**: UX and UI
- **Full-Stack Developer**: frontend and backend development
- **Systems Administrator (DevOps)**: container configuration, CI/CD pipeline setup, and deployment
- **Product Owner**

Time Feasibility

The project is ambitious but feasible within the established academic deadlines. Its iterative, milestone-based planning enables the achievement of incremental objectives and continuous feedback, thereby minimizing the risk of not reaching a functional version.

[4] Koyeb. Top PostgreSQL Database free tiers in 2025. Broshar, Alisdair. 2025, 8 gener. https://www.koyeb.com/blog/top-postgresql-database-free-tiers-in-2025

SWOT Method

SWOT	Weaknesses	Strengths
Internal Analysis	- One-man project - Academic project without multidisciplinary profiles - Limited to Android devices - No monetization or economic profitability plan	- Modern and in-demand technology stack - Decoupled and scalable architecture - Process automation - Deep and real knowledge of the publishing sector
	Threats	**Opportunities**
External Analysis	- Risk of becoming obsolete due to new standards - Highly competitive sector	- Excellent technical portfolio - Easily iterable to introduce improvements - Reusable for other projects

As a strategic tool, the following SWOT analysis has been carried out:

Weaknesses

- The project is developed by a single person: development blockers may take longer to resolve, and feature delivery speed may be affected.
- It is an academic project with the inherent limitations of not being a professional team: absence of roles such as architect, senior developer, designer, etc.
- Mobile application limited to Android devices.
- No solution has been implemented to economically monetize the investment.

Threats

- The emergence of new technological standards that could render some aspects of the project obsolete.
- Very high competition in the content management systems (CMS) sector, with numerous mature and consolidated solutions.

Strengths

- Use of modern technologies with strong demand in the labor market.
- Architecture designed to be decoupled, easy to maintain, and scalable.
- Process automation to save time on repetitive tasks and ensure quality standards.
- The student has several years of experience in the publishing sector, providing real and in-depth knowledge of the domain and product needs.

Opportunities

- The project represents an excellent and comprehensive technical portfolio to demonstrate skills in potential recruitment processes.
- The project foundation can be extended with new features such as order management, shopping cart functionality, user management, etc.

- The RESTful API architecture can be reused for external projects, for example by adding a web client to the ecosystem that consumes the API.

Strategic Plan (Cross SWOT Analysis)

Cross SWOT	Weaknesses	Strengths
Threats	**Reorientation Strategies (W–T)** - CI/CD with GitHub Actions - Strict API-First approach - Organized planning - Functional and technical refinements	**Defensive Strategies (S-T)** - Project visibility - High-quality documentation
Opportunities	**Survival Strategies (W–O)** - Focus on a solid MVP - Quality over quantity	**Offensive Strategies (S–O)** - Modern, future-proob tech stack - Clean and decoupled architecturea

Based on the analysis of internal and external factors, an action plan is defined to maximize the project's potential and mitigate risks. By cross-referencing the elements of the SWOT analysis, the following strategies are proposed, each associated with concrete, measurable, and executable initiatives.

Reorientation Strategies (W–T)

Compensating for the limitation of being a single developer by leveraging automation and ensuring quality.

- Implement the following GitHub Actions across all repositories to reduce potential issues and improve development speed:
  - **Static code checks**: pre-runtime validation ensuring code follows conventions.
  - **Build checks**: validation that branches compile correctly and that all tests pass.
- Avoid technical debt and unnecessary development:
  - Apply a strict API-First approach.
  - Apply SOLID, KISS, and YAGNI design principles.
- Organized and meticulous planning to compensate for the absence of a team:
  - Use GitHub Projects to plan work using a Kanban board.
  - Use GitHub Projects to create work tickets that enforce functional and technical refinement instead of coding directly.

Survival Strategies (W–O)

Ensuring that the project fulfills its primary purpose: effectively demonstrating technical competencies.

- Focus development on a solid MVP rather than secondary features.
  - Use GitHub Projects to define milestones that establish iterative objectives, guiding the project from a minimal MVP to increasingly complete and functional versions.

- Prioritize quality over quantity: a well-designed backend and application with partial functionality is more valuable than a complete but poorly structured system.
    - Ensure unit and integration test coverage across the entire codebase.
- Emulate a production environment using Docker and CI/CD:
    - The entire system can be executed in a remote environment.

Offensive Strategies (S–O)

Positioning the project as a high-level technical portfolio.

- Increase project visibility:
    - Pin the project and its repositories on the GitHub profile.
- Highlight technical decisions:
    - All repositories will include high-quality README files explaining architectural decisions and deployment or usage instructions.

Defensive Strategies (S–T)

Ensuring project maintainability in the face of technological evolution.

- Select technologies with broad support and future projection.
    - **Backend**: Kotlin 2 with Spring Boot 3.5.
    - **Frontend**: Kotlin 2 with Jetpack Compose.
    - **Documentation**: OpenAPI 3.1
- Implement a clean and decoupled architecture that allows changes in specific libraries or frameworks with minimal impact.
    - Apply Hexagonal Architecture.

Conclusions of the SWOT Analysis

The application of the SWOT method is not merely an academic requirement, but a fundamental and widely used tool in the planning phase of any project. Its usefulness has gone beyond simple factor identification, providing a structured framework for decision-making even before writing the first line of code.

This analysis has made it possible to:

1. **Anticipate risks**: identifying threats has enabled the design of proactive strategies to mitigate them from the very system design stage.
2. **Optimize resources**: recognizing inherent weaknesses allows prioritization of what truly generates impact, maximizing the efficiency of the available time.
3. **Maximize value**: the offensive strategy (S–O) guides the project toward a clear dual objective—meeting functional requirements while creating a high-level technical portfolio, thereby increasing tangible value.
4. **Define direction**: the strategy matrix is not mere speculation, but a concrete action plan that guides development, ensuring that every technical decision aligns with the goal of strengthening the project against its weaknesses and potential threats, while leveraging its main strengths and opportunities.

In conclusion, the SWOT analysis stands out as a key tool for building a development strategy that guarantees a final result that is viable, robust, maintainable, and valuable.

Methodology and Success Metrics (OKRs / KPIs)

To ensure the success of the project and the effective execution of the strategies defined in the SWOT analysis, the following **Objectives and Key Results (OKRs)** have been established. These objective metrics allow the evaluation of progress and the achievement of quality, planning, and technical value goals.

OKR 0: Client Value and Usability

Objective: Deliver a tool that is genuinely useful and adopted by the client.

- **Key Result 1 (KR1)**: The client can perform all CRUD operations on Authors, Collections, and Books without requiring technical assistance or consulting documentation.
    - **KPI**: Qualitative client feedback
- **KR2**: UI/UX designed to minimize clicks and confusing actions
    - **KPI**: Number of screens required to complete a key action ≤ 3

OKR 1: Technical Quality and Robustness

Objective: Develop production-ready, clean, and maintainable code.

- **KR1**: Achieve and maintain 80% unit and integration test coverage across all critical repositories (backend and frontend).
    - **KPI**: Code coverage percentage reported by Gradle.
- **KR2**: Implement a minimum of two automated GitHub Actions workflows in each repository.
    - **KPI**: Number of active GitHub Actions workflows passing successfully.

OKR 2: Efficient Planning and Execution

Objective: Manage the project using agile methodologies, compensating for limited resources.

- **KR1**: Refine 100% of tasks before moving them to *In Progress* on the Kanban board.
    - **KPI**: Percentage of issues with a clear description and acceptance criteria.
- **KR2**: Complete 100% of tasks defined within each milestone.
    - **KPI**: Milestone completion rate.

OKR 3: Value as a Technical Portfolio

Objective: Turn the project into a tangible demonstration of technical competencies.

- **KR1**: Ensure that 100% of repositories include a professional README with badges, project description, tech stack, and deployment guide.
    - **KPI**: README elements checklist completion.
- **KR2**: Deploy and keep the production environment active.
    - **KPI**: Health check endpoint returns HTTP 200 OK.

OKR 4: Maintainability and Future Readiness

Objective: Ensure that the project remains relevant and easy to improve in the long term.

- **KR1**: Define the complete API contract in the api-spec repository before implementing any backend endpoints.
    - **KPI**: Controller interfaces are auto-generated from the api-spec.

- **KR2**: Achieve a pure domain layer, 100% independent of frameworks and infrastructure libraries.
    - **KPI**: Zero external framework imports in the domain module files.

Timeline Planning / Work Schedule

Project planning has been structured around major milestones representing the achievement of significant functional objectives. These milestones will be recorded as **GitHub Milestones** and broken down into smaller tasks, known as **issues** in GitHub.

Each task will be associated with one or more Git branches, which will be merged into the main branch through Pull Requests. All project activity will be accessible and traceable through a Kanban board in **GitHub Projects**.

This phased approach is intended to apply best practices from **Agile methodologies** commonly used in industry, while also ensuring continuous value delivery through iterative and incremental functional releases.

Milestones Table

Phase	Milestone	Description	Delivery (Estimate)	Deliverables
🔧 **Phase 0**	M0: Setup and definition	Initial GitHub Projects setup and complete definition of the API contract	September 22 (2 weeks)	- api-spec repository created - Complete and stable OpenAPI specification - Generated DTO code and Controller interfaces
⚙️ **Phase 1**	M1: Backend MVP (Authors)	Implementation of basic use cases (CRUD) for the Author entity with full security	October 13 (3 weeks)	- Functional Authors RESTful API - Authentication and role-based authorization operational - Postman configured - Basic CI/CD & dockerization
📱 **Phase 2**	M2: Frontend MVP (Authors)	UI development for author management and API integration	October 27 (2 weeks)	- Android APP with Authors CRUD - Functional Login - Token management using DataStore
🚀 **Phase 3**	M3: DevOps & Deployment	Deployment automation through CI/CD	November 3 (1 week)	- Deployed environment - CI/CD pipeline improvements - Dockerization improvements
📚 **Phase 4**	M4: Full Functionality	Complete relational database design and implementation of Book and Collection entities and their relationships	December 1 (4 weeks)	- Full CRUD for Books and Collections - Complete Android application
✅ **Phase 5**	M5: Polishing & Delivery	Final refinements, report writing, and defense preparation	December 15 (2 weeks)	- Polished READMEs - Minor bug fixes - Final report completed

Detailed Description of the Milestones

M0: Setup and Definition (Specification)

Objective: Establish the immutable contract that will serve as the foundation between frontend and backend.

Key Tasks:

- Write a complete OpenAPI specification (schemas, endpoints, and DTOs).
- Configure the OpenAPI Generator plugin to achieve the desired behavior.
- Publish the auto-generated code so it can be consumed as an external library.

M1: Backend MVP (Authors)

Objective: Implement a functional and secure backend core.

Key Tasks:

- JWT authentication, role management, and Spring Security configuration.
- Author CRUD implementation using Hexagonal Architecture.
- Repository layer implementation with Spring Data JPA.
- Unit and integration tests.
- Global exception handler.
- Basic CI/CD and basic Dockerization.
- Postman configuration.

M2: Frontend MVP (Authors)

Objective: Develop a functional mobile app covering the complete Author entity workflow.

Key Tasks:

- Login screen.
- Author management screens.
- Implement AuthRepository using DataStore.
- Configure Retrofit for API consumption.
- Handle loading states and error management.

M3: DevOps & Deployment

Objective: Automate deployment to ensure continuous delivery.

Key Tasks:

- Configure GitHub Actions for automatic builds and test execution.
- Deploy the application and database to a cloud service.
- Configure the domain and SSL certificate.

M4: Full Functionality

Objective: Complete the system's core functionality.

Key Tasks:

- Replicate the design pattern from M1 and M2 for the Book and Collection entities.
- Implement relationships between entities.
- Implement the corresponding application screens.

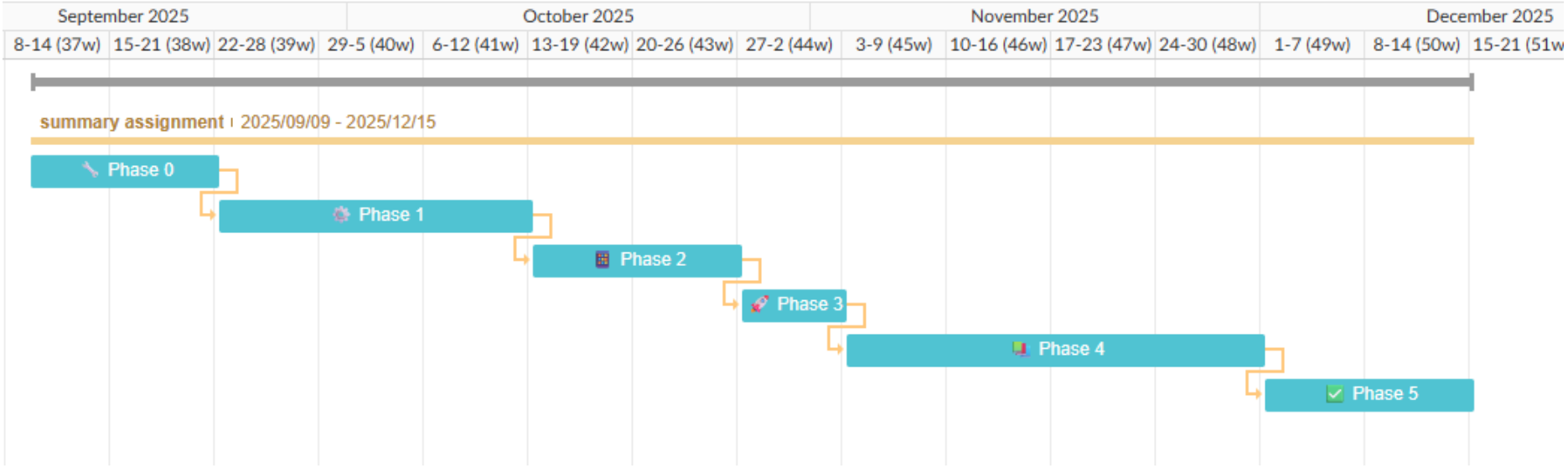
M5: Polishing and Delivery

Objective: Prepare everything for final delivery and defense.

Key Tasks:

- Final usability testing.
- Minor bug fixes.
- Polish README files.
- Complete the final report.
- Prepare the presentation.

Gantt Diagram[5]



[5] GanttPRO. https://app.ganttpro.com/

GitHub Projects: Kanban Board and Milestones

GitHub is primarily designed as a Git version control platform rather than a specialized project management tool. However, it increasingly offers features oriented toward planning and project organization. Although it is far less versatile and powerful than other planning solutions such as Jira, it works very well for small teams or independent projects that do not rely on corporate infrastructures providing dedicated project management tools.

With the timeline planning, milestone definition, and work schedule already established, it is straightforward to create a Kanban board, replicate the defined milestones, add iterations (or sprints), and create issues (equivalent to Jira tickets) that break the work down into manageable tasks.

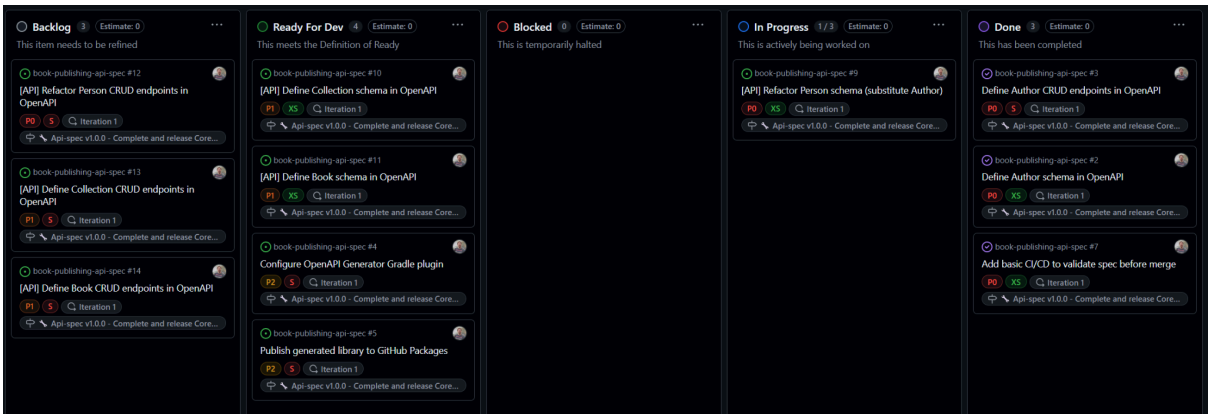
As the project progresses or reaches completion, the resulting information and metrics will be highly valuable in a real professional environment.

One limitation encountered when using this tool is that milestones are linked to individual repositories rather than to a project as a whole. This means that repositories must be created before milestones can be defined, and that during the second half of the project, milestones may need to be duplicated or triplicated depending on the number of repositories involved in achieving the same goal.

Work Implementation Cycle

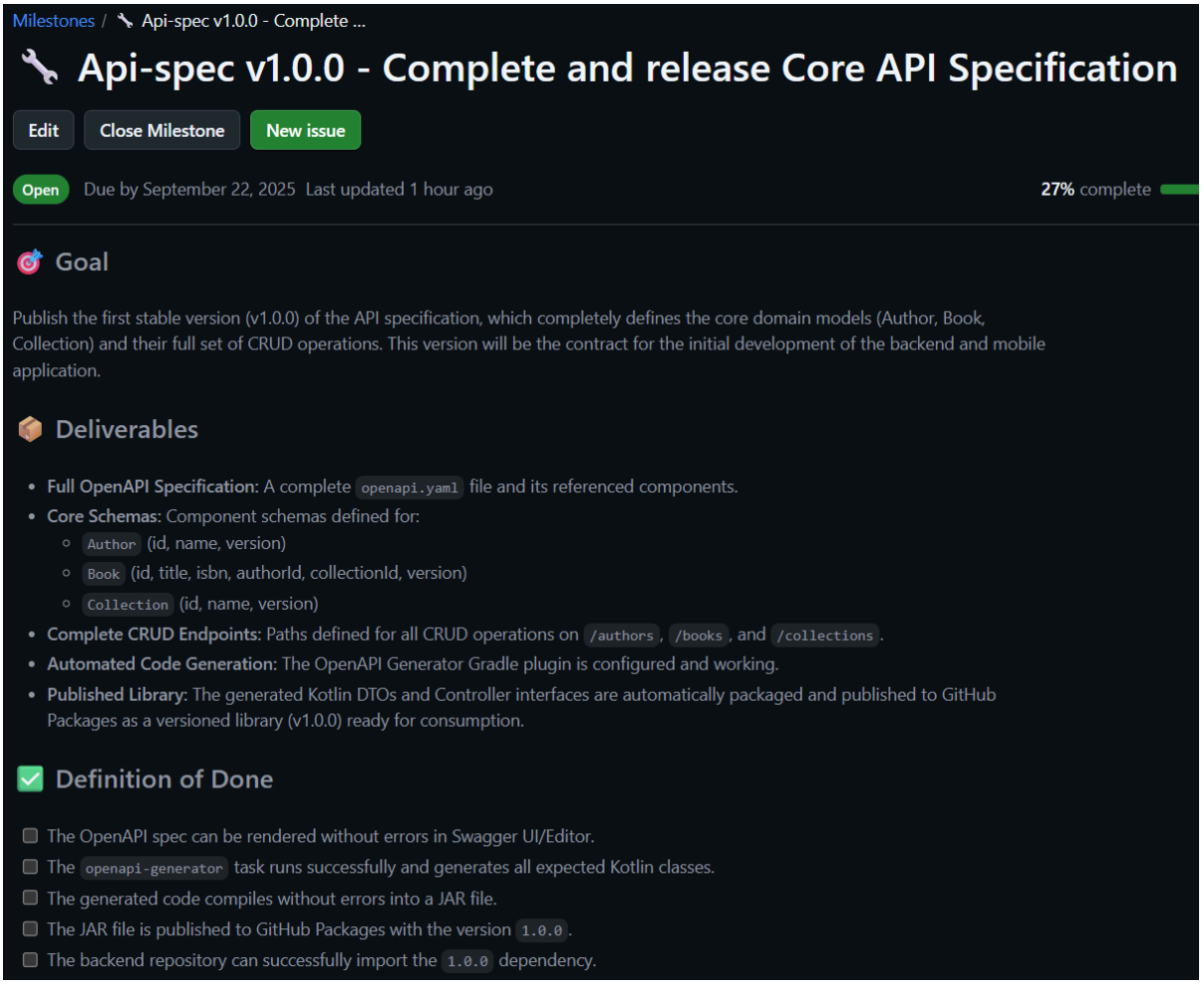
With a board to manage tasks and milestones to organize them, the foundations of the project's implementation cycle have been established:

1. **Milestones**: Create a milestone that defines an objective, its deliverables, and the scope of what must be completed.
2. **Task creation**: Create work items (GitHub Issues) required to achieve the milestone.
3. **Refinement and planning**: Perform functional and technical refinement of the tasks, prioritize them, estimate their effort, and assign them to an iteration (sprint).
4. **Implementation**: Complete all tasks until the milestone is achieved.
5. **Repeat the cycle**.

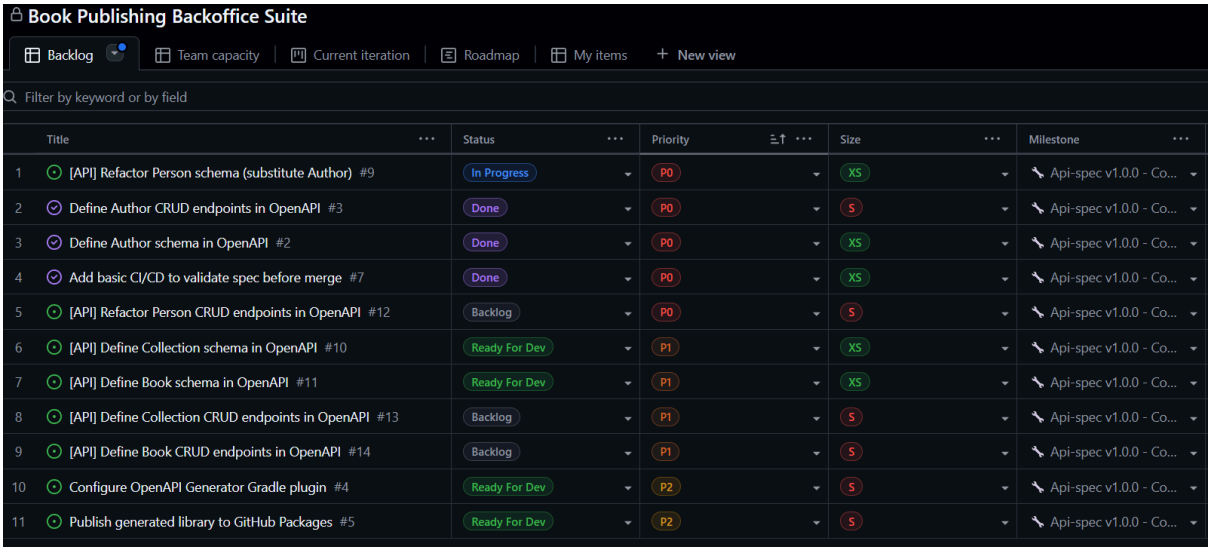


Kanban board view[6]

[6] Book Publishing Back Office Suite, GitHub Project (board view) https://github.com/orgs/CescFe/projects/9/views/1?layout=board

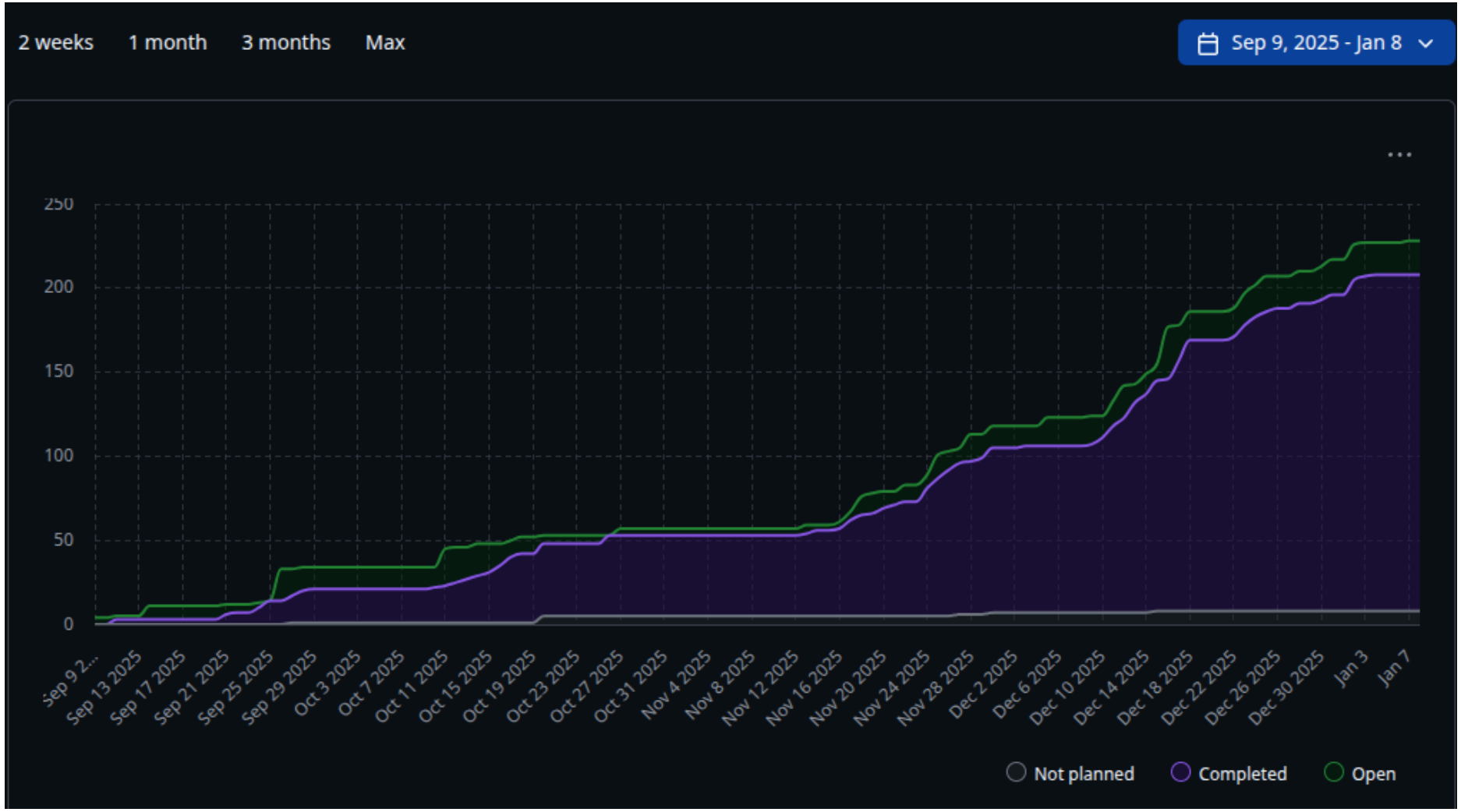


Initial Milestone[7]



Project Backlog view with issues from the first milestone[8]

[7] Project's initial Milestone: https://github.com/CescFe/book-publishing-api-spec/milestone/1
[8] Book Publishing Back Office Suite: https://github.com/orgs/CescFe/projects/9/views/1



The Burn up chart shows the progress of the project items over time, showing how much work has been completed and how much is left to do

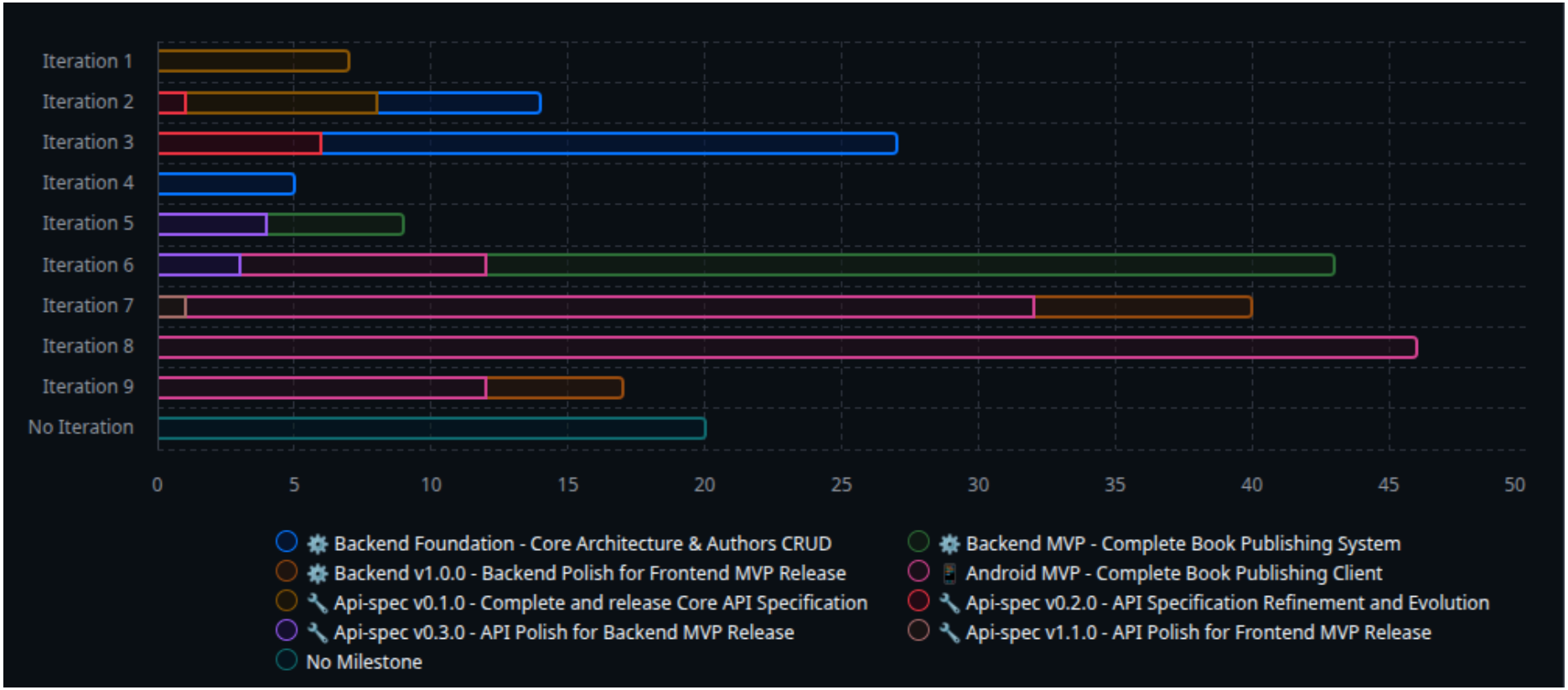


Chart of completed issues grouped by iteration and milestone

REQUIREMENTS ANALYSIS

Requirements Description

Functional Requirements (FR)

The main objective of the system is to provide a comprehensive tool for managing the digital catalog of a publishing house. The functionalities are focused on basic CRUD operations for maintaining the data of the main entities, which can be managed by an administrator user:

- **FR-01: Author management**. The administrator shall be able to create, view, update, and delete authors in the system. Each author will be defined using basic and essential attributes.
- **FR-02: Collection management**.The administrator shall be able to create, view, update, and delete collections in the system.
- **FR-03: Book management**. The administrator shall be able to create, view, update, and delete books. Each book will be defined by attributes such as title, ISBN, number of pages, cover image, among others, and will be linked to an existing author and an existing collection.
- **FR-04: User authentication**. The system shall restrict access through a secure login mechanism. Only authenticated users will be able to access the system functionalities. Administrator users will have access to full functionality, while non-administrator users will be limited to read-only access to entities.

Non-Functional Requirements (NFR)

These requirements define the system's constraints and quality attributes, ensuring that it is robust, secure, and maintainable:

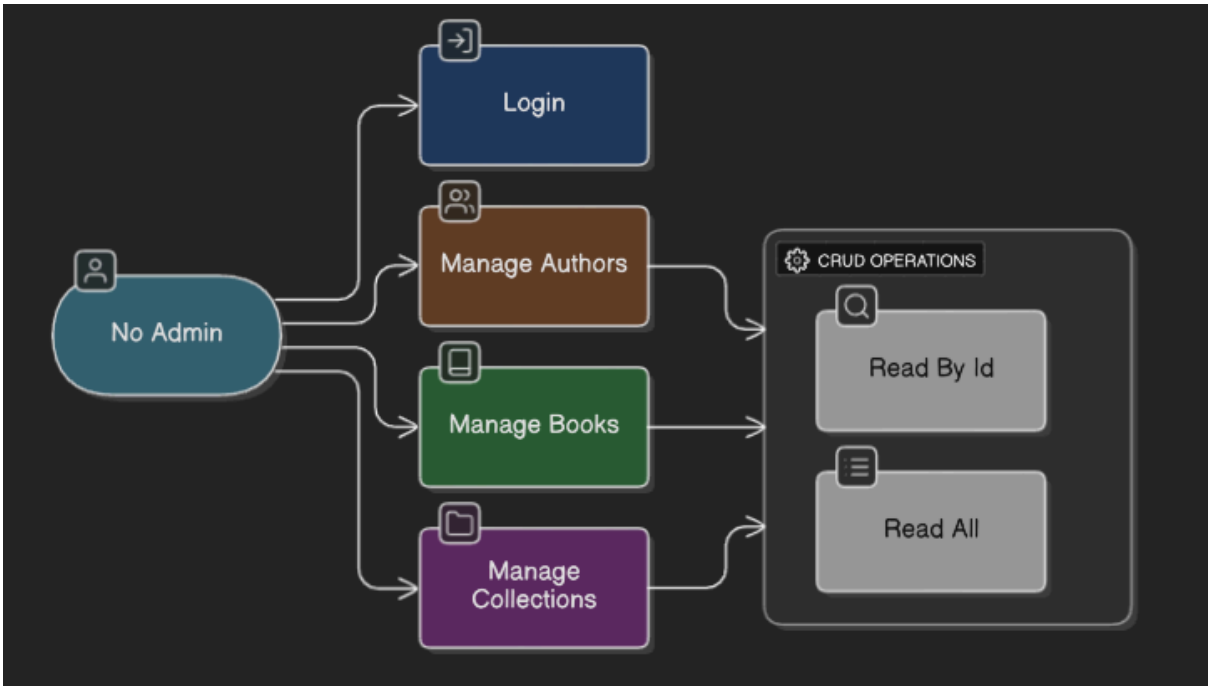
- **NFR-01: Clean architecture**. The backend shall be implemented following the principles of Hexagonal Architecture (also known as Ports and Adapters), decoupling business logic from infrastructure concerns.
- **NFR-02: API-First design**. The API shall be defined first using OpenAPI 3.0, acting as a single and immutable contract between the frontend and the backend.
- **NFR-03: Security**. Access to the API shall require authentication using JWT (JSON Web Tokens), ensuring that all communications are properly authorized.
- **NFR-04: Usability**. The mobile application must be intuitive and easy to use for all types of users, with a minimal learning curve.

- **NFR-05: Compatibility**. The mobile application shall be compatible with recent versions of the Android operating system.
- **NFR-06: Performance**. Read (query) operations must have a response time of less than 2 seconds under normal load conditions.
- **NFR-07: Maintainability**. The codebase shall be well documented, follow Kotlin coding conventions, and use static code analysis tools to ensure quality.
- **NFR-08: Deployment**. The system must support automated deployment through CI/CD pipelines using GitHub Actions and be packaged in Docker containers to ensure consistency across environments.

Main Use Case Diagrams

Use Case Diagram: Non-Administrator User

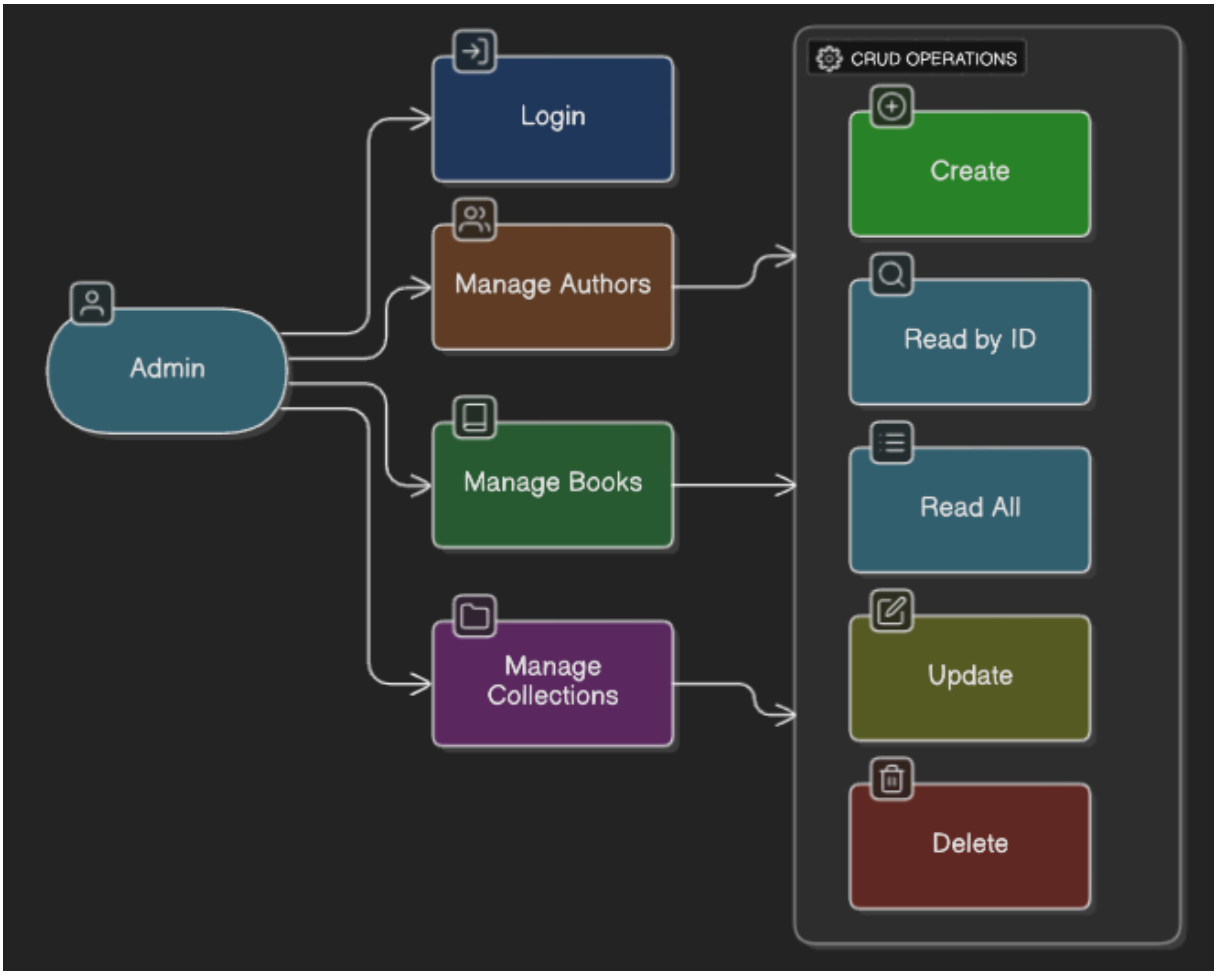
The following diagram shows that the actor (a standard non-administrator user) can log in and perform read-only operations on the three entities: **Author**, **Book**, and **Collection**.



Use case diagram for a user with read-only permissions

Use Case Diagram: Administrator User

The following diagram provides an overview showing that the actor (administrator) can log in and perform full CRUD operations on each of the entities: **Author**, **Book**, and **Collection**.



Use case diagram for an administrator user

Business Rules and Ubiquitous Language[9]

To establish solid foundations for building a maintainable and scalable project, it is essential to define business rules[10] and to establish a **Ubiquitous Language**: a shared business language understood and used by all stakeholders, both technical and non-technical. This language is also reflected in the business rules and will be particularly useful when applying Domain-Driven Design (DDD)[11].

Main Business Rules

Author context:

- An author must have a name.
- If an author has an email address, it must be unique.

---

[9] *Ubiquitous Language*. Fowler, Martin. 2006-10-31. https://martinfowler.com/bliki/UbiquitousLanguage.html

[10] Uncovering Hidden Business Rules with DDD Aggregates. Tune, Nick. 2019-11-30, https://medium.com/nick-tune-tech-strategy-blog/uncovering-hidden-business-rules-with-ddd-aggregates-67fb02abc4b

[11] *Domain Driven Design*. Fowler, Martin. 2020-04-22. https://martinfowler.com/bliki/DomainDrivenDesign.html

Collection context:

- A collection must have a name.
- The collection name must be unique.
- “Out of collection” is considered a valid collection.

Book context:

- A book must have a title.
- A book must have an author.
- A book must belong to a collection.
- A book must have a base price.
- If a book has an ISBN, it must be unique.
- A book must have a tax rate (VAT).
- By default, the tax rate is 4%.
- A book must have a retail price (RRP – Recommended Retail Price).
- A book may have more than one language, but no more than four.
- A book may have between zero and three subgenres.
- A book must have a status.

Ubiquitous Language

A book can have four possible states:

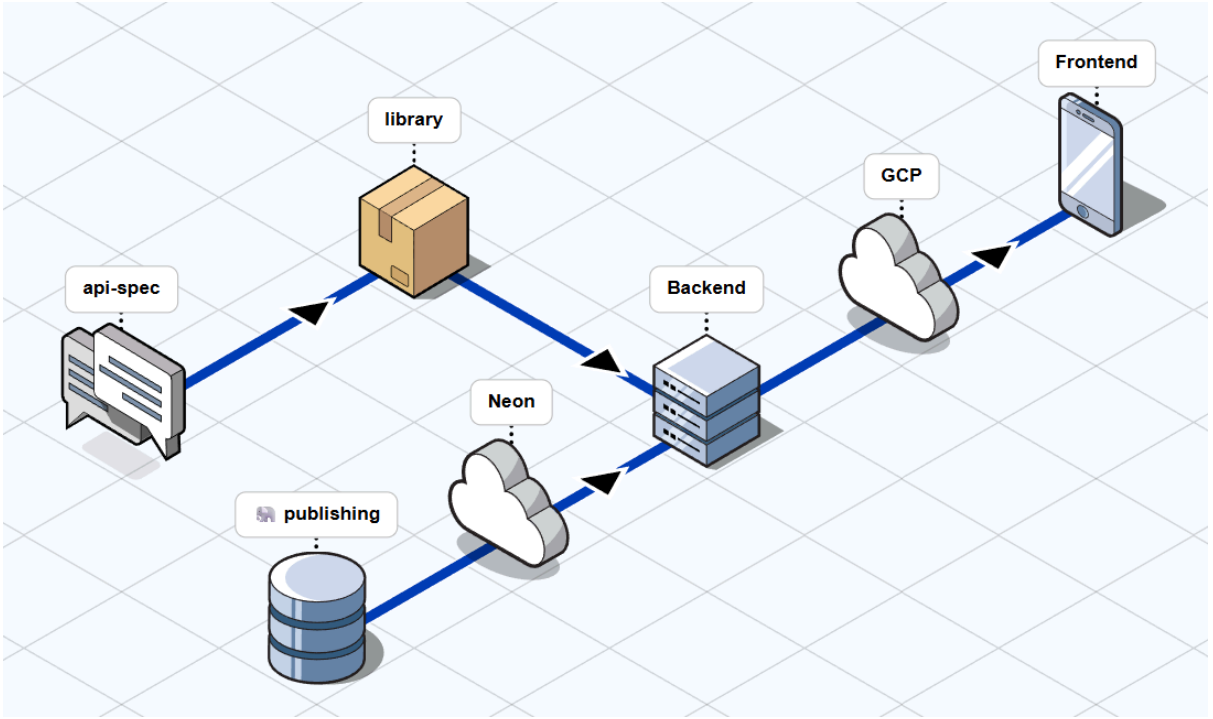
- **Draft**: The book has not yet been released for sale.
- **Published**: The book has been released and copies are available.
- **Out of stock**: The book has been released but no copies are currently available.
- **Discontinued**: The book has been released but has been permanently withdrawn from the market due to lack of stock or any other reason.

A collection or a book can be categorized according to the target reader’s age group:

- **Children** (0 to 11 years)
- **Young Adult** (12 to 17 years)
- **Adult** (18+).

ARCHITECTURE

The project architecture has been designed following a modular and decoupled approach, with the goal of meeting client requirements while ensuring maintainability, scalability, and consistency across the different system components.



Architectural solution diagram, created using FossFlow[12]

The core of the architectural design is the public repository **book-publishing-api-spec**, which formally defines the RESTful API contract using the OpenAPI 3.1 standard. This repository describes all available endpoints, data models, requests, and responses, establishing a single, versioned contract that acts as the source of truth for the rest of the system.

Based on this specification, the **OpenAPI Generator** plugin is used to automatically generate an external Kotlin library, which is published via **GitHub Packages**. This library contains the controller interfaces and the Data Transfer Objects (DTOs) for requests and responses. This approach ensures that the backend implements exactly the defined contract, preventing inconsistencies, reducing errors, and guaranteeing that the API documentation is always up to date.

[12] FossFlow. Stan Smith. https://github.com/stan-smith/FossFLOW

The public repository **book-publishing-backend**, developed using Kotlin, Spring Boot, and Gradle, consumes this generated library and uses it within its REST infrastructure layer, applying the **Ports and Adapters Architecture** (also known as **Hexagonal Architecture**).

Regarding data persistence, the backend uses **Liquibase** to define and version the schema of a relational **PostgreSQL** database. Database migrations are executed automatically using a Docker image configured through a docker-compose.yml file. The database deployment and update process is managed through a **GitHub Action**, enabling controlled and reproducible application of changes to a database hosted on **NeonTech**.

The backend packages the Java Archive (JAR) using a **Dockerfile**, which is deployed to **Google Cloud Platform** via a cloudbuild.yaml file on every commit to the `main` branch. This provides a scalable and reliable infrastructure for exposing the RESTful API.

Finally, the system is consumed by the public repository **book-publishing-app**, a frontend developed as a native Android application using **Kotlin** and **Jetpack Compose**. The app communicates with the backend through the REST API, clearly separating presentation logic from business logic and following best practices for modern mobile application development.

Overall, this architecture enables a clear separation of responsibilities, facilitates parallel development, and ensures a high level of consistency between the API contract, the backend, and the mobile client.

API CONTRACT

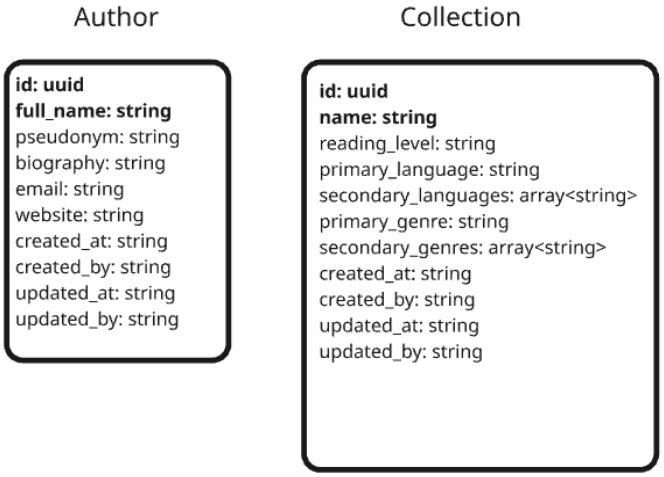
Once the planning and milestones have been defined, along with the requirements, use cases, business rules, and an architectural proposal for the system, we can proceed to define the REST API contract.

Starting with this step is essential in order to apply the **API-First methodology**[13] adopted in this project. The contract is not part of either the server environment or the client environment, as it does not directly participate in the execution of the system; instead, it represents a formal agreement between the client and the server.

The contract is contained in a repository named **book-publishing-api-spec**[14], hosted on GitHub. **Gradle**[15] is used as the build automation and dependency management tool, and **OpenAPI**[16] is used as the descriptor for the API format.

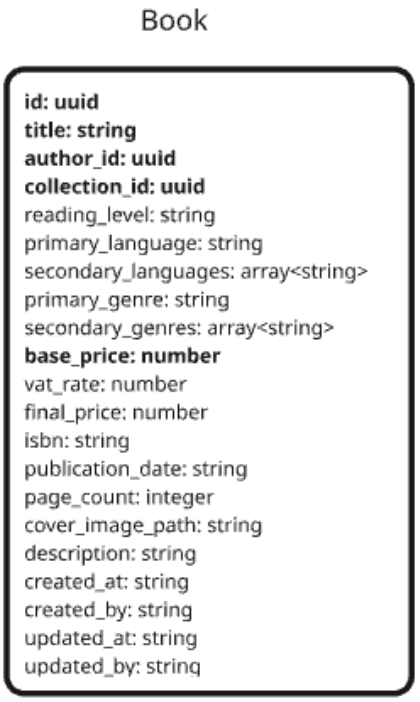
Conceptual Design of OpenAPI Components

The first step is to analyse the schemas that will make up the API specification. Accordingly, the following structure has been defined for the **Book**, **Author**, and **Collection** objects, specifying not only the attribute names but also whether they are required (marked in **bold**) and their data types. These schemas represent the **Data Transfer Objects (DTOs)** that the REST API will send (responses) or receive (requests) through the different HTTP methods.



[13] Postman. *Guide to API-first*. https://www.postman.com/api-first/
[14] Contract's Repository. https://github.com/CescFe/book-publishing-api-spec
[15] Gradle Build Tool. https://gradle.org/
[16] OpenAPI Specification. https://spec.openapis.org/oas/v3.1.2.html

Design of the OpenAPI schemas for Author and Collection (top image) and Book (right image), showing attributes and data types (required fields in bold)



SERVER ENVIRONMENT

On the other hand, once the contract has been established, work can begin on the server environment, which consists of two main components: the **backend** in the form of a **RESTful API**, where requests are processed, business logic is executed, and interactions with the **database** take place; and the database itself, where the system information is stored.

RESTful API

The backend is hosted in the **book-publishing-backend**[17] repository. The chosen programming language is **Kotlin**, the framework is **Spring Boot**, and **Gradle** is used as the dependency and build management tool.

- **Kotlin**: a modern, statically typed language (variable types are known at compile time) that runs on the Java Virtual Machine (JVM). Fully interoperable with Java, it provides a more concise and safer alternative. It combines object-oriented and functional programming paradigms, reduces verbosity, and helps prevent Null Pointer Exceptions.
- **Spring Boot**: a Java framework that simplifies application development by automating configuration, promoting the *Convention over Configuration* principle[18], and providing dependency starter packages.
- **Gradle Build Tool**: the dependency management and build automation tool responsible for compilation, packaging, test execution, deployment, and artifact publishing.

Main Dependencies

- **book-publishing-api-spec**: the API contract described in the specification repository.
- **Spring Data JPA**[19]: the most widely used implementation of the Java Persistence API (JPA), facilitating access to and interaction with the database.
- **Spring Security**[20]: the de facto standard for managing authentication and authorization.
- **JUnit 5**: the most popular unit testing framework in the Java ecosystem.
- **MockMvc**: a testing framework used to simulate HTTP operations.

---

[17] Backend repository: https://github.com/CescFe/book-publishing-backend
[18] Wikipedia. *Convention over Configuration*. https://en.wikipedia.org/wiki/Convention_over_configuration
[19] Spring Data JPA. https://spring.io/projects/spring-data-jpa
[20] Spring Security. https://spring.io/projects/spring-security

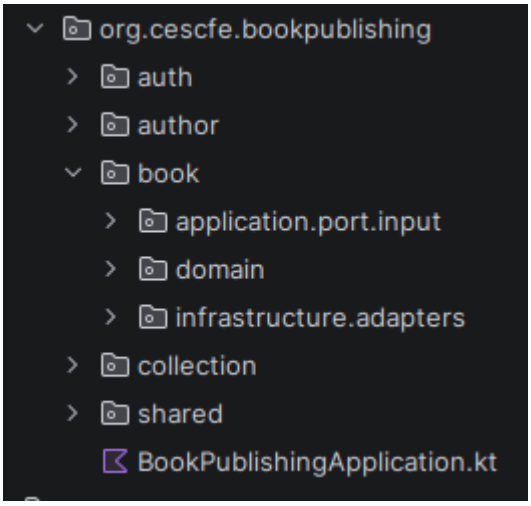
- **Testcontainers**: a testing framework based on Docker images, used to verify interactions with the database.
- **PostgreSQL**: an object-relational database management system (ORDBMS[21]) used as the project's database.
- **Liquibase**[22]: a tool for managing and versioning database schemas.
- **Docker**: enables separation of the application from the infrastructure. It is used both to manage production deployment and to run a local image that simulates the real database environment.
- **Spotless**: a static analysis and code formatting tool that ensures consistent code style and adherence to best practices.
- **GitHub Actions**: used to manage continuous integration and continuous delivery (CI/CD) workflows.

Methodologies Applied

- **API-First**[23]: an approach to API development from a product-oriented perspective, aiming to produce modular and interoperable APIs. In this project, this methodology has clearly been applied by generating a real API contract in the **book-publishing-api-spec** repository, which is consumed by the backend.
- **Arquitectura Hexagonal**[24] **(Ports and Adapters)**: proposed by Alistair Cockburn in 2005, its goal is to create loosely coupled architectures by isolating business logic from external concerns (user interfaces, databases, APIs), thereby improving maintainability, adaptability, and testability. In this project, the codebase is organised into the three classic layers of hexagonal architecture: **domain**, **application**, and **infrastructure**. Additionally, **ports** (interfaces that isolate layers and act as contracts) and **adapters** (implementations of those contracts) are applied.
- **Domain-Driven Design**[25]: a methodology that prioritises understanding and modelling the specific problems of the domain in which the system operates. In software design, it captures and represents domain concepts. In the backend, DDD principles are followed by starting the implementation from the **Domain layer**, defining Value Objects[26], their structure, and the associated business rules.
- **Vertical Slice Architecture**[27]: an approach that organises code by features or use cases, grouping related components together. In this project, the vertical slice approach is applied on top of the hexagonal architecture by first dividing the codebase into **contexts** (auth, author, book, collection, and shared) and then into **layers** (domain, application, and infrastructure).

---

[21] Geeks for Geeks. *Difference between RDBMS and ORDBMS*. Kawalpreet, 2025-07-15
https://www.geeksforgeeks.org/dbms/difference-between-rdbms-and-ordbms/
[22] Liquibase. https://docs.liquibase.com/oss/user-guide-4-33/intro-to-liquibase
[23] Medium. Three Principles of API First Design. Trieloff, Lars. 2017-06-02.
https://medium.com/adobetech/three-principles-of-api-first-design-fa6666d9f694
[24] Geeks for Geeks. *Hexagonal Architecture - System Design*. Navlani, Nehal. 2025-07-23.
https://www.geeksforgeeks.org/system-design/hexagonal-architecture-system-design/
[25] Geeks for Geeks. *Domain-Driven Design (DDD)*. Ali, Asad. 2025-07-12
https://www.geeksforgeeks.org/system-design/domain-driven-design-ddd/
[26] Medium. Value Objects. Sánchez, Miguel Ángel. 2019-01-03
https://medium.com/all-you-need-is-clean-code/value-objects-d4c24115fa69
[27] Baeldung. Vertical Slice Architecture. Trandafair, Emanuel. 2024-06-26
https://www.baeldung.com/java-vertical-slice-architecture



Backend Folder structure,,
applying Hexagonal Architecture and Vertical Slice

Applied Design Patterns

Several design patterns have been applied throughout the codebase in order to prevent tight coupling, with the aim of producing software that is easier to understand, maintain, and test. The objective is that anyone reviewing the system's behaviour or introducing changes can understand the code with the lowest possible entry barrier and learning curve.

- **Single Responsibility Principle (SRP)**: A class or module should have only one reason to change, meaning it should have a single, well-defined responsibility. Some examples found in the codebase include:
    - **Controller**: in the REST infrastructure layer, its sole responsibility is handling HTTP requests. It delegates the execution of the use case to the application layer.
    - **Use case or interactor**: in the application layer, its only responsibility is to execute a specific business use case (such as *"create a book"*).
    - **Repository**: in the persistence infrastructure layer, its only responsibility is data access.
- **Open/Closed Principle (OCP)**: Classes, modules, or functions should be open for extension but closed for modification. Examples include:
    - **PasswordEncoder**: its configuration is open for extension through the PasswordConfig class, while its usage inside the UserService class is closed to modification.
    - **Repositoris**: the interface defining BookRepository can be extended without modifying existing code, while the code that uses this repository (for example, the GetBookInteractor class) remains closed to modification.
- **Liskov Substitution Principle (LSP)**: Subtypes must be substitutable for their base types without altering the correctness of the program.
    - **Value Classes**: for example, BookId (a value object in the domain layer) is represented at runtime as a UUID. Therefore, it can be used wherever a UUID is expected, without losing compile-time type safety.
- **Interface Segregation Principle (ISP)**: Clients should not be forced to depend on interfaces they do not use; it is preferable to split interfaces into smaller, more specific ones.
    - **Use cases**: they are segregated, each with its own interface. For example, GetBookUseCase and ListBookUseCase.
    - **Validations**: specific validations depend on their own interfaces. For instance, BookDomainService is responsible only for ISBN uniqueness validations.

    - **Repository usage**: each client uses only what it needs. For example, the BookRepository interface exposes multiple methods, but GetBookInteractor only uses findById(), while ListBooksInteractor uses findAllSummary() and countAll().
- **Dependency Inversion Principle (DIP)**: High-level modules should not depend on low-level modules; both should depend on abstractions, and abstractions should not depend on details. This means prioritising abstraction over implementation.
    - **Application layer**: the application layer is abstracted from infrastructure through the use of ports and adapters. The domain is accessed via the BookRepository interface, so the GetBookInteractor service (application layer) depends on a domain interface, not on an infrastructure implementation.
    - **Infrastructure layer**: the GetBookController depends on the GetBookUseCase interface and is abstracted from the concrete GetBookInteractor implementation.
- **Factory Pattern**: A creational design pattern that provides an interface for creating objects while delegating the responsibility of deciding which concrete object to instantiate.
    - **Object Mother**:Applied in tests (for example, AuthorObjectMother) to reduce test verbosity and abstract object creation logic.
    - **Exceptions Handling**: Applied to the management of controlled exceptions, simplifying their creation (for example, BookDomainException).
- **Convention over Configuration** (also known as *Coding By Convention*): The framework establishes default rules and conventions, reducing repetitive configuration and allowing developers to focus on the specific aspects of app logic.
    - **Minimal configuration**: as much configuration as possible is delegated to the framework, avoiding custom configuration classes. Examples include using Jackson for JSON serialization or not defining manual beans for application-layer services.
    - **Spring Data JPA**: query methods are delegated to JPA whenever possible, avoiding manual query implementations for methods such as *findById()* or *existsByIsbn()*.
    - **Dependency Injection**: Unnecessary or optional uses of *@Autowired* have been avoided, for example in JpaBookRepository.
- Other very popular principles that have been actively applied are **KISS** (*Keep It Simple, Stupid*), **YAGNI** (*You Ain't Gonna Need It*), **DRY** (*Don't Repeat Yourself*), or **Least Astonishment**, by trying not to include unnecessary codification and complexities, maintaining a clear nomenclature in the methods, and reusing code through function extraction (in coherence with the *Vertical Slice*, within the same context or, when transversal, locating them in the *Shared* context).
- Other less popular principles that have been applied are **DbC** (Design By Contract) with the use of preconditional validations in the creation of domain Value Objects (example: BookTitle and the use of *init* and *require*); **Fail Fast** with immediate validations upon object construction (ex.: BookId) or use case execution (ex.: GetBookInteractor), **CQS** (Command Query Separation) differentiating use cases into queries or commands, and the *readOnly* property has been applied to query transactions.

Configured Environments

Through different application files, the following environments have been configured:

- **local**: intended for local development and debugging. It does not raise the database, as this is raised separately with docker-compose, creating a PostgreSQL container that emulates the production one.
- **test**: intended for persistence integration tests. It does execute Liquibase, raising a database image.
- **development**: ghost, born and dies in GitHub CI/CD.
- **production**: connected to the real database.

Database

A database is a digital collection of structured and organized information that can be easily accessed, managed, and updated. It is generally controlled by a Database Management System or DBMS, which is software that acts as an interface to efficiently and securely manage said information.

The first thing we needed to know before implementing our database was whether we needed a relational database (SQL) or a non-relational database (NoSQL). As a general rule, SQL databases are suitable for structured and consistent data, and systems with complex relationships that require integrity; while NoSQL is suitable for unstructured and changing data, systems with massive scalability, flexibility, and speed. There are solutions that combine the best of both, implementing the Command Query Responsibility Segregation (CQRS[28]) pattern, reserving write operations to persist data in an SQL DB and delegating read operations to snapshots stored in a NoSQL DB.

Our case clearly fits the general case of relational databases. Next, with the ER diagram, we will model the domain, with the logical design, we will transform it into relational tables, and with the physical design, we will implement it in a DBMS.

Conceptual Design of Entity Relationship

The first step in designing the database structure was to design an Entity-Relationship Diagram, where the entities Book, Author, and Collection have been visually represented with rectangles, the relationships between entities and their cardinality have been defined with diamonds, and the attributes of each entity have been assigned with ellipses. Multivalue attributes have been represented with double ellipses.

Entities and attributes

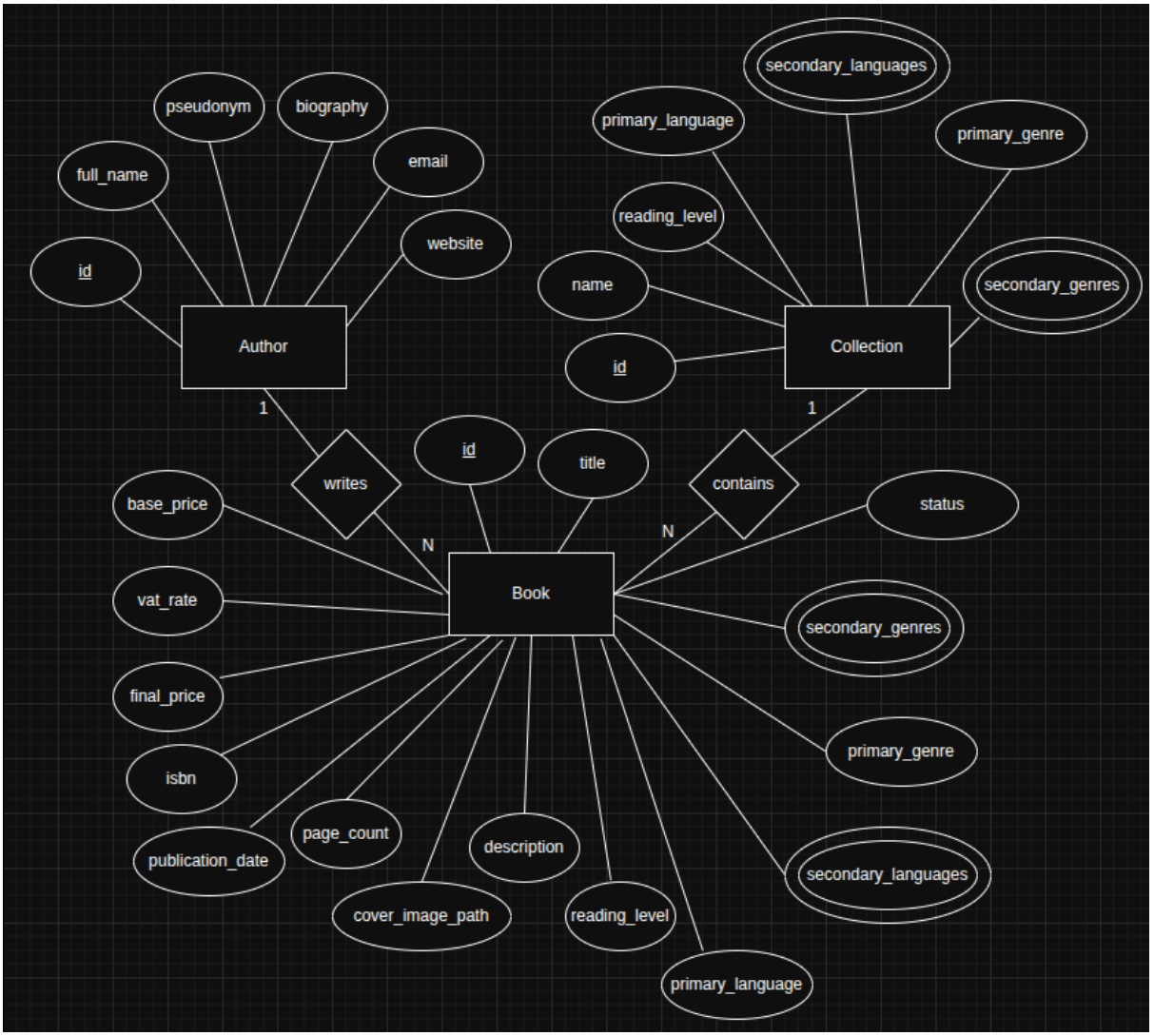
- **Book**: id (PK), title, base_price, vat_percentage, final_price, isbn, publication_date, number_of_pages, cover_image_path, description, reading_level, main_language, other_languages, main_genre, secondary_genres.
- **Author**: id (PK), full_name, pseudonym, biography, email, website.
- **Collection**: id (PK), name, reading_level, main_language, other_languages, main_genre, secondary_genres.

[28] *CQRS*. Fowler, Martin. https://martinfowler.com/bliki/CQRS.html

Relationships and cardinalities

- 1 Author **writes** N books: An author can write many books, a book only belongs to one author.
- 1 Collection **contains** N books: A collection can contain many books, a book only belongs to one collection.

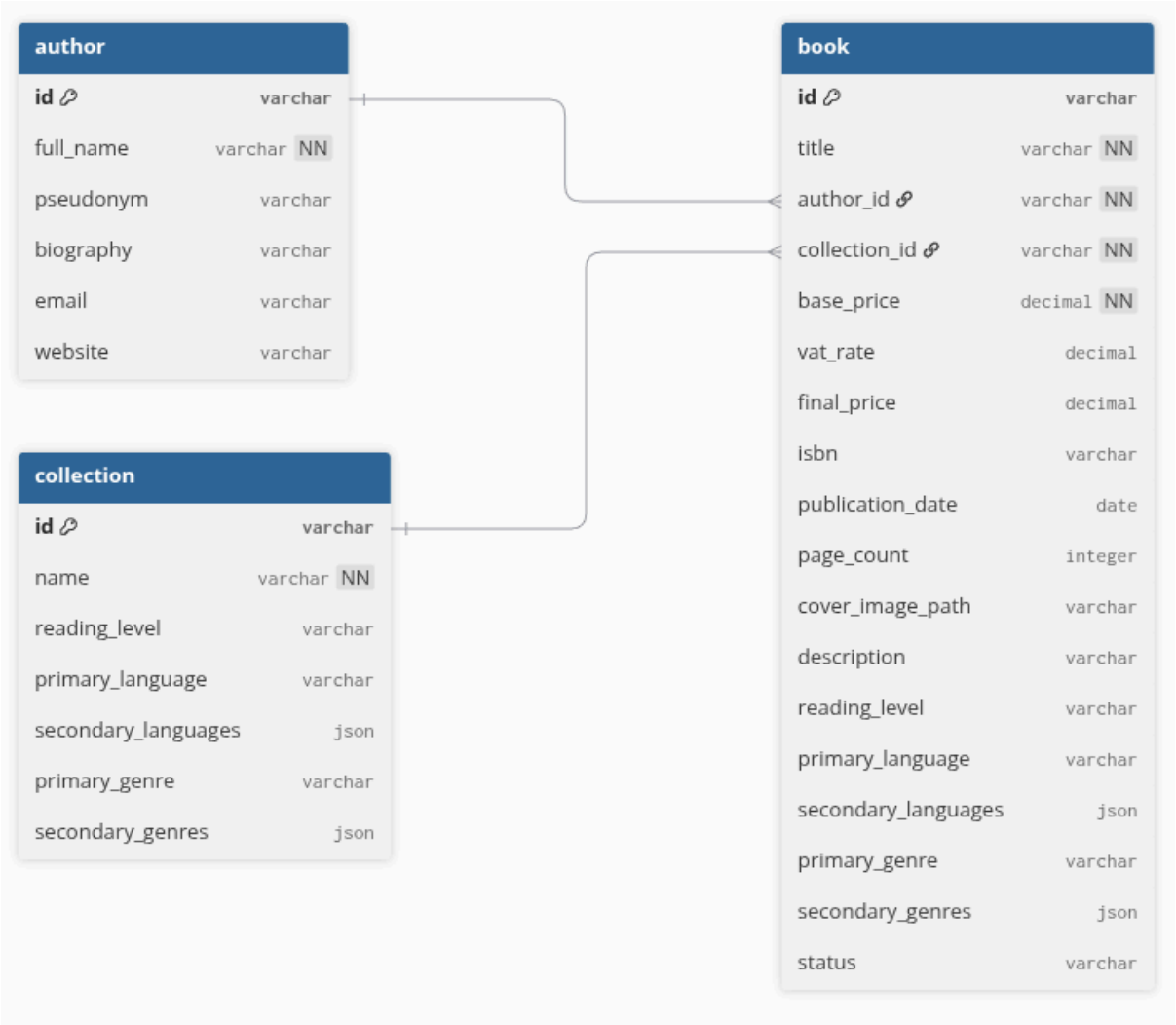
Taking into account the definition of the client's requirements, the use cases, the business rules, and the API contract that has already been designed, the simplest possible schema has been defined, which nonetheless fulfills all the technical and functional requirements that must be achieved in this project.



Project Entity-Relationship Diagram

Relational Logical Design

The next step is to perform a relational logical design that defines the data structure without depending on a specific DBMS. The following DBML code has been written, which is easily convertible into a graphic representation. Fields that are primary or foreign key, non-nullable fields, and types have been indicated. The code and graph are attached below.



Tables of the relational logical design

Physical Design with Liquibase

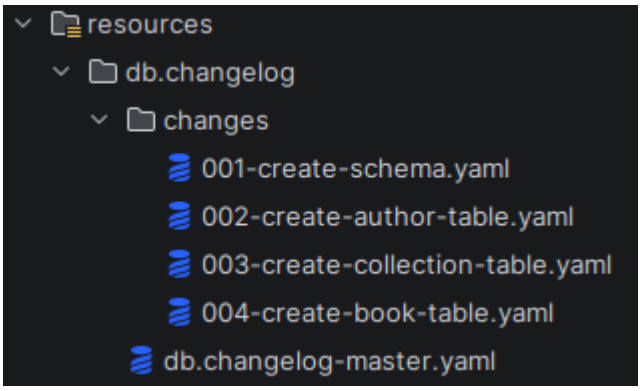
The physical design adapts the logical model to a specific DBMS, defining data types, constraints, and indexes. We have chosen PostgreSQL as the BD manager, which is technically an ORDBMS, as it is a widely used option in the industry (58% use among professionals according to the StackOverFlow survey in 2025[29]) and for its native UUID and JSONB types, which we will use as the types for PKs and multivalue attributes.

Liquibase has been chosen to manage and version the DB and its changes from the backend repository. This facilitates DB control, application testing (especially its persistence layer), and centralizes the server environment infrastructure.

The Liquibase files can be found in the backend repository, within the *resources* path. An attempt has been made to atomize the schema creation to make it more readable and maintainable. The schema is not managed with SQL files, thus applying the Dependency Inversion Principle, prioritizing abstraction over implementation: we do not know which DBMS is implemented (we configure this to be managed from docker-compose), but we use the native Liquibase syntax, which manages the implementation transparently for us.

[29] StackOverFlow. *2025 Developer Survey*. https://survey.stackoverflow.co/2025/technology#most-popular-technologies-database-prof

Liquibase files in the backend



During the physical implementation, the following improvements have been added:

- **Addition of uniqueness constraints**: thanks to the contract, we know which fields are unique.
- **Addition of indexes**: to improve some queries, indexes have been added to unique fields and JSONB fields.
- **Addition of audit fields**: the standards *created_by*, *created_at*, *updated_by*, and *updated_at* have been added.

Some code screenshots are attached:

Liquibase migration history

```yaml
databaseChangeLog:
  - include:
      file: changes/001-create-schema.yaml
      relativeToChangelogFile: true
  - include:
      file: changes/002-create-author-table.yaml
      relativeToChangelogFile: true
  - include:
      file: changes/003-create-collection-table.yaml
      relativeToChangelogFile: true
  - include:
      file: changes/004-create-book-table.yaml
      relativeToChangelogFile: true
```

Schema creation

```yaml
databaseChangeLog:
  - changeSet:
      id: create-publishing-schema
      author: francesc
      changes:
        - sql:
            sql: "CREATE SCHEMA IF NOT EXISTS publishing"
```

```yaml
databaseChangeLog:
  - changeSet:
      id: create-book-table
      author: francesc
      changes:
        - createTable:
            tableName: book
            schemaName: publishing
            columns:
              - column:
                  name: id
                  type: UUID
                  constraints:
                    primaryKey: true
                    nullable: false
              - column:
                  name: title
                  type: VARCHAR(200)
                  constraints:
                    nullable: false
              - column:
                  name: author_id
                  type: UUID
                  constraints:
                    nullable: false
              - column:
                  name: collection_id
                  type: UUID
                  constraints:
                    nullable: false
              - column:
                  name: base_price
                  type: NUMERIC(10, 2)
                  constraints:
                    nullable: false
              - column:
                  name: vat_rate
                  type: NUMERIC(3, 2)
              - column:
                  name: final_price
                  type: NUMERIC(10,2)
```

Start of the Book table creation file with the declaration of the first of its attributes, the type and some constraints

End of the same file, where the indexes are defined

```
- addForeignKeyConstraint:
    referencedTableSchemaName: publishing
    referencedColumnNames: id
    constraintName: fk_book_collection
- createIndex:
    tableName: book
    schemaName: publishing
    indexName: idx_book_title
    columns:
      - column:
          name: title
- createIndex:
    tableName: book
    schemaName: publishing
    indexName: idx_book_author_id
    columns:
      - column:
          name: author_id
- createIndex:
    tableName: book
    schemaName: publishing
    indexName: idx_book_collection_id
    columns:
      - column:
          name: collection_id
- createIndex:
    tableName: book
    schemaName: publishing
    indexName: idx_book_isbn
    columns:
      - column:
          name: isbn
    unique: true
- sql:
    sql: CREATE INDEX idx_book_secondary_languages
      ON publishing.book USING GIN (secondary_languages)
- sql:
    sql: CREATE INDEX idx_book_secondary_genres
      ON publishing.book USING GIN (secondary_genres)
```

Description of Tables and Fields

Author Table

This is the table where the information of the Author entity lives. Its fields are:

- **id (PK)**: it is the author's UUID. It is the table's **primary key**.
- **full_name**: It is the author's name and surname(s). It is a **required field**.
- **pseudonym**: It is the author's pseudonym.
- **biography**: Author's biography.
- **email**: Author's email. It is a **unique field**.
- **website**: Author's website.

Collection Table

This is the table where the information of the Collection entity lives. Its fields are:

- **id (PK)**: it is the collection's UUID. It is the table's **primary key**.
- **name**: It is the name of the collection. It is a **required and unique field**.
- **reading_level**: It is the reading level of the collection's target reader.
- **main_language**: Main language in the collection's books.
- **other_languages**: Other languages present in the collection.
- **main_genre**: Main genre transversal to the collection.
- **secondary_genres**: Secondary genres or subgenres of the collection.

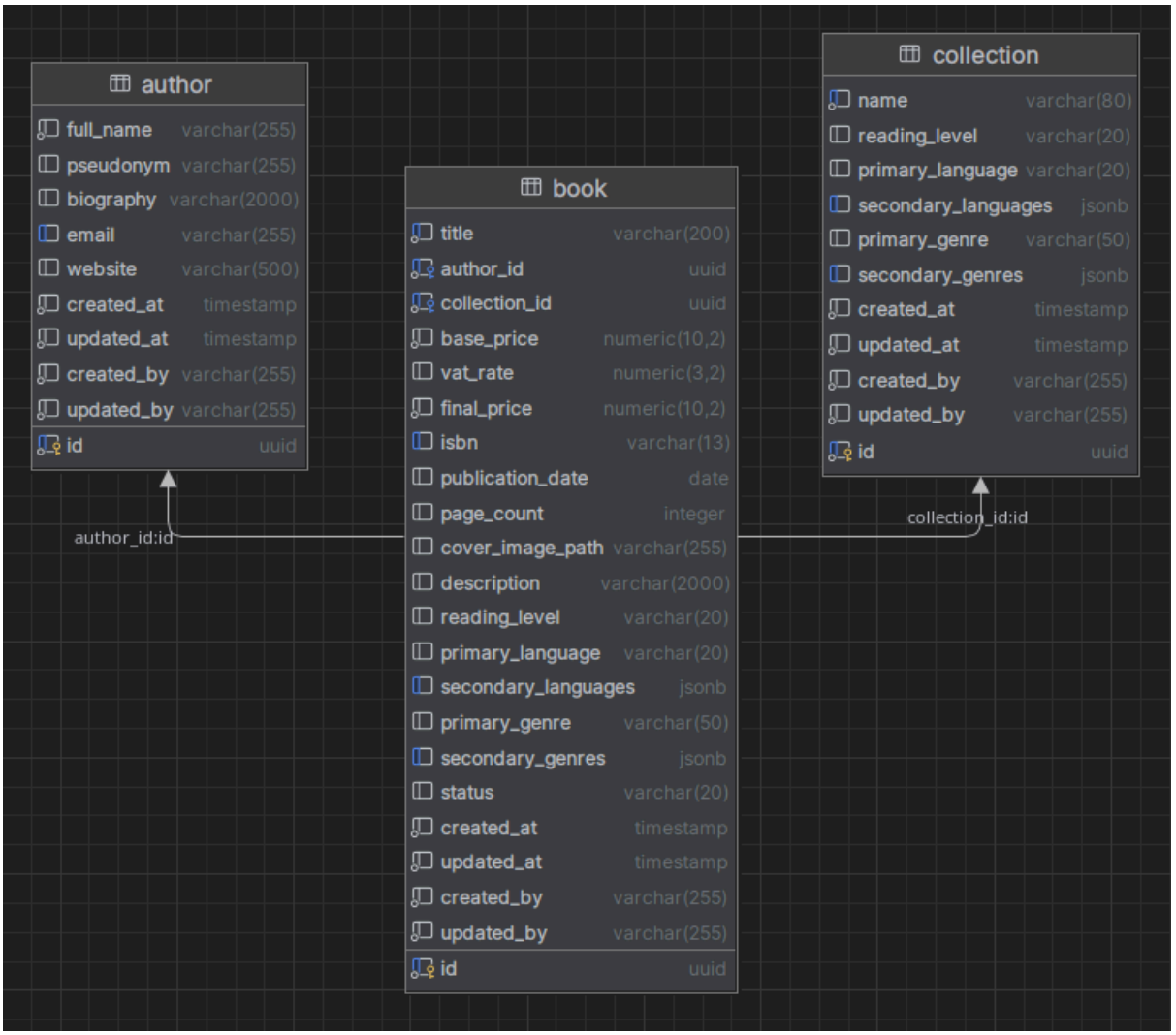
Book Table

This is the table where the information of the Book entity lives. Its fields are:

- **id (PK)**: it is the book's UUID. It is the table's **primary key**.
- **title**: It is the title of the book. It is a **required field**.
- **author_id**: it is the UUID of the book's author. It is the **foreign key** that references the related element of the author table.
- **collection_id**: it is the UUID of the collection the book belongs to. It is the foreign key that references the related element of the collection table.
- **base_price**: It is the price of the book before taxes. It is a **required field**.
- **vat_percentage**: It is the VAT percentage.
- **final_price**: It is the book's base price plus taxes, i.e., the Public Sale Price.
- **isbn**: It is the International Standard Book Number, a 13-digit code that identifies the book's edition. It is a **unique field**.
- **publication_date**: It is the publication date of the book's edition.
- **number_of_pages**: It is the total number of pages of the book's edition.
- **cover_image_path**: It is the internal reference to the location of the book cover image resource.
- **description**: book synopsis.
- **reading_level**: It is the reading level of the book's target reader.
- **main_language**: Main language in which the book is written.
- **other_languages**: Other languages contained in the book.
- **main_genre**: Main genre of the book.
- **secondary_genres**: Secondary genres or subgenres of the book.

PostgreSQL Diagram

Finally, this is the PostgreSQL schema of our database.

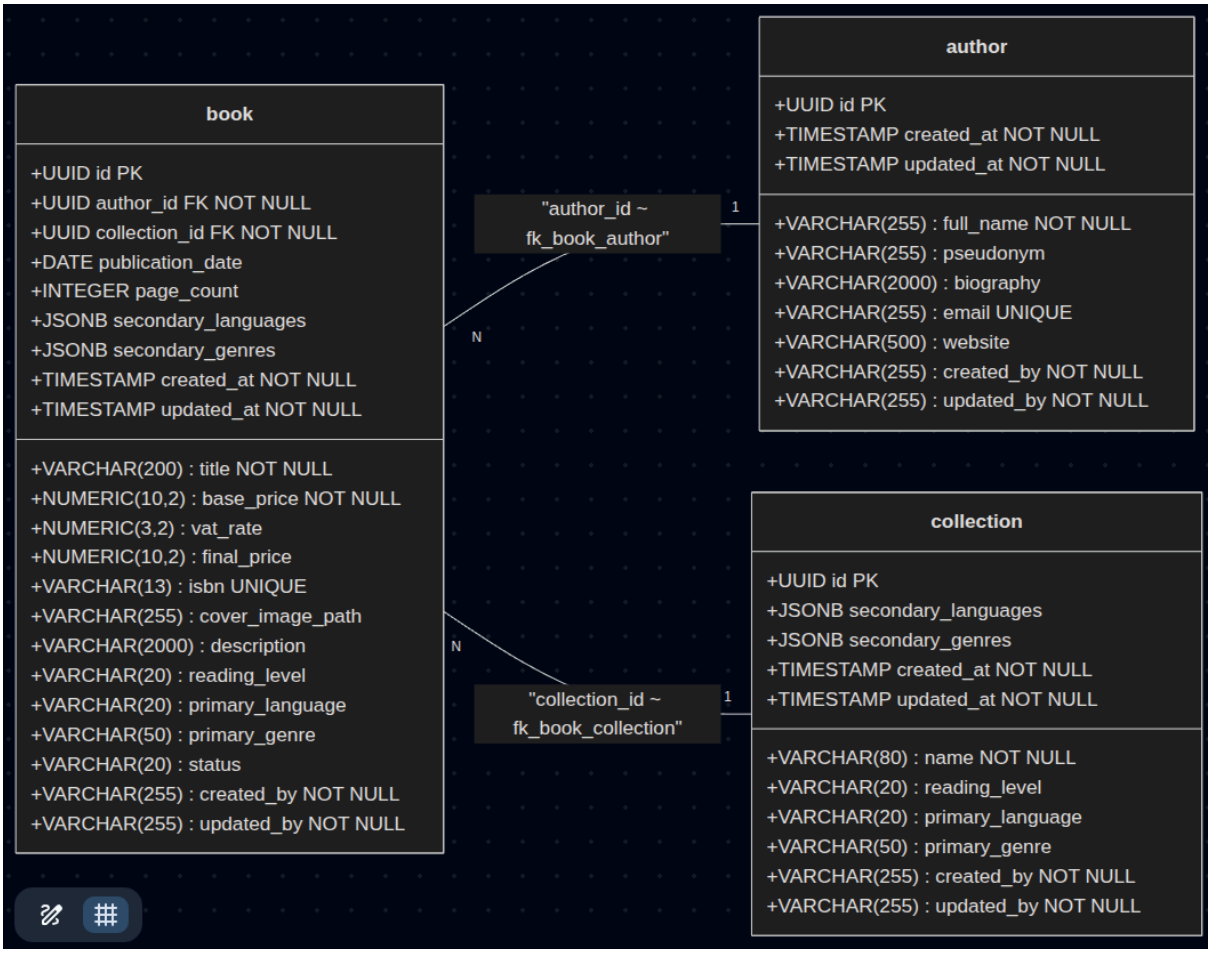


Object-Oriented

The following section describes how the tables and relationships have been transferred to classes, objects, and execution flows.

Class Diagrams

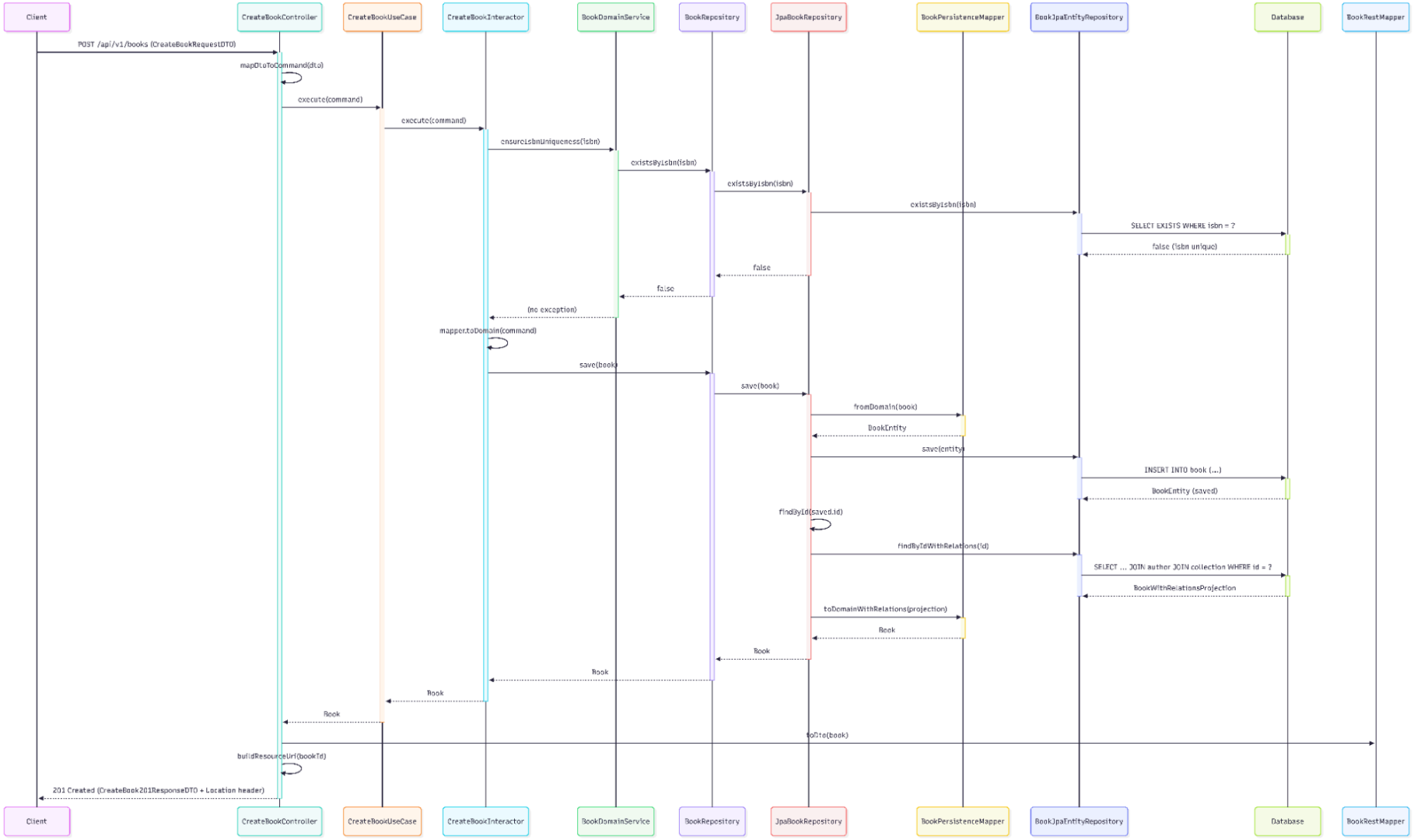
The class diagram represents the transition from the relational model to the object-oriented model, defining the domain classes, their attributes, and the relationships between them.

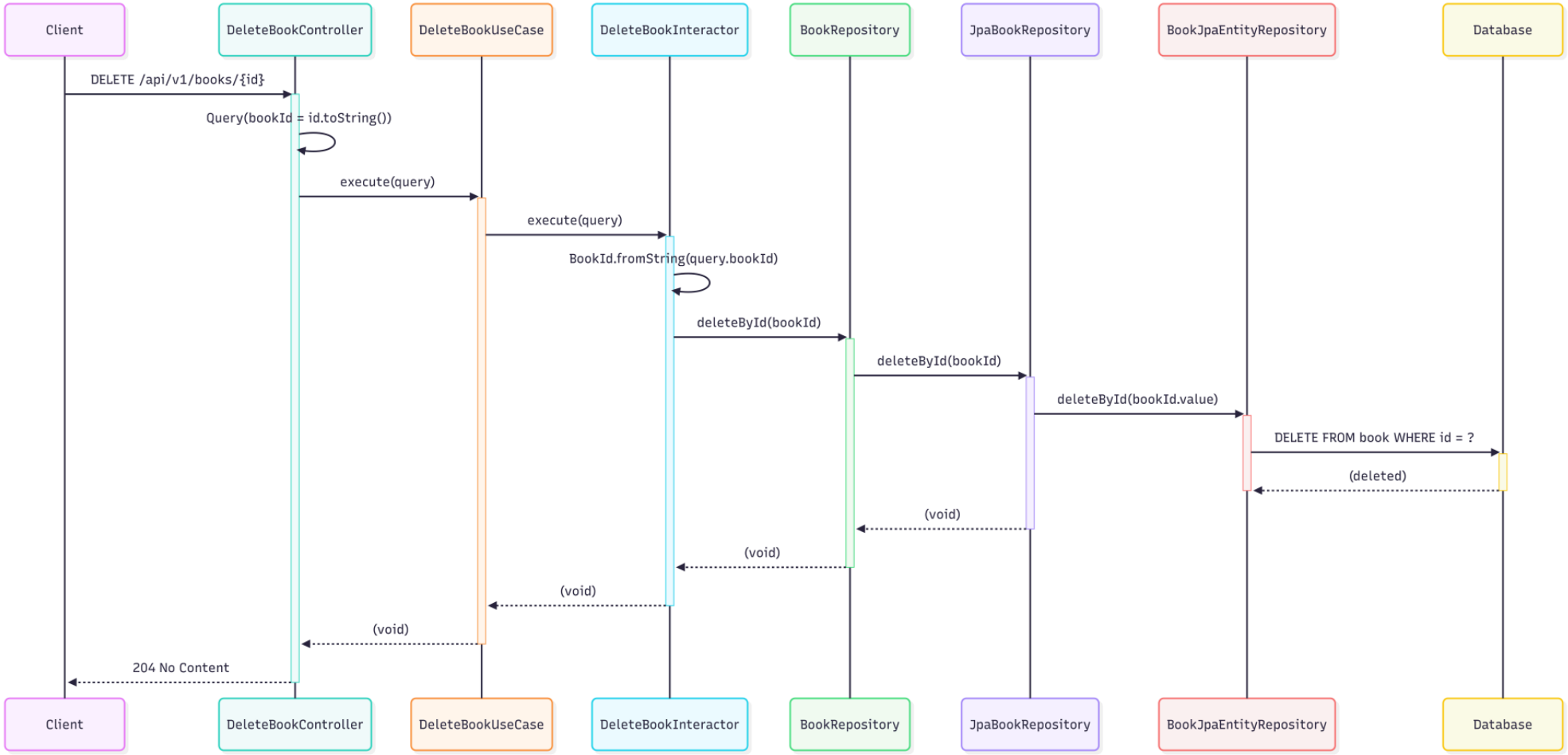


Project Class Diagram

Sequence Diagram

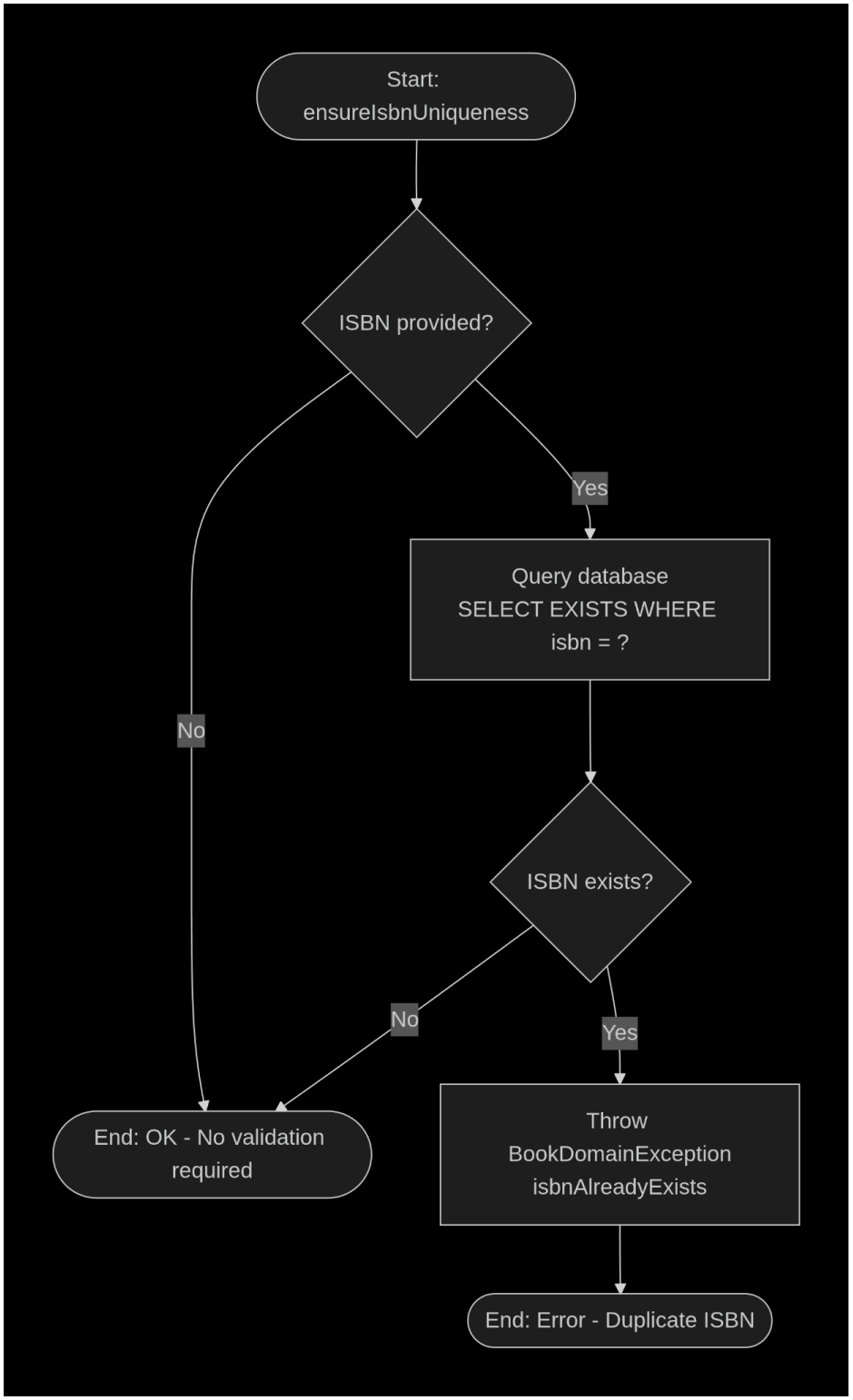
Sequence diagrams describe the temporal interaction between the different objects of the system for the execution of the defined use cases. Hereunder, we attached the sequence diagrams for the Use Case of Creating and Deleting a Book.

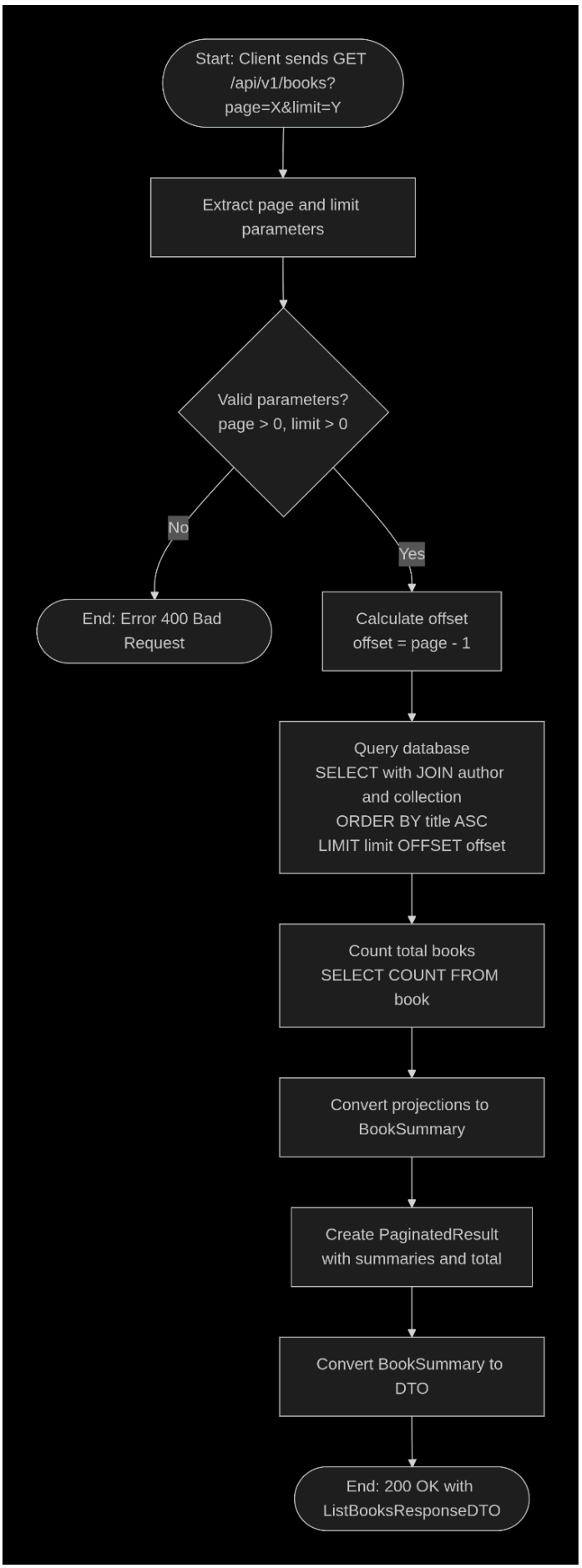

Client
CreateBookController
CreateBookUseCase
CreateBookInteractor
BookDomainService
BookRepository
JpaBookRepository
BookPersistenceMapper
BookJpaEntityRepository
Database
BookRestMapper
POST /api/v1/books (CreateBookRequestDTO)
mapDtoToCommand(dto)
execute(command)
execute(command)
ensureIsbnUniqueness(isbn)
existsByIsbn(isbn)
existsByIsbn(isbn)
existsByIsbn(isbn)
SELECT EXISTS WHERE isbn = ?
false (isbn unique)
false
false
(no exception)
mapper.toDomain(command)
save(book)
save(book)
fromDomain(book)
BookEntity
save(entity)
INSERT INTO book (...)
BookEntity (saved)
findById(savedId)
findByIdWithRelations(id)
SELECT ... JOIN author JOIN collection WHERE id = ?
BookWithRelationsProjection
toDomainWithRelations(projection)
Book
Book
Book
Book
Book
toDto(book)
buildResourceUri(bookId)
201 Created (CreateBook201ResponseDTO + Location header)


Client
DeleteBookController
DeleteBookUseCase
DeleteBookInteractor
BookRepository
JpaBookRepository
BookJpaEntityRepository
Database
DELETE /api/v1/books/{id}
Query(bookId = id.toString())
execute(query)
execute(query)
BookId.fromString(query.bookId)
deleteById(bookId)
deleteById(bookId)
deleteById(bookId.value)
DELETE FROM book WHERE id = ?
(deleted)
(void)
(void)
(void)
(void)
204 No Content
Client
DeleteBookController
DeleteBookUseCase
DeleteBookInteractor
BookRepository
JpaBookRepository
BookJpaEntityRepository
Database

Activity Diagram

Activity diagrams allow us to model the execution flow of the system processes, including decisions and validations. The activity diagram for the use case of validating a book's ISBN uniqueness and the book listing diagram are attached.




Start: Client sends GET /api/v1/books? page=X&limit=Y
Extract page and limit parameters
Valid parameters? page > 0, limit > 0
No
Yes
End: Error 400 Bad Request
Calculate offset offset = page - 1
Query database SELECT with JOIN author and collection ORDER BY title ASC LIMIT limit OFFSET offset
Count total books SELECT COUNT FROM book
Convert projections to BookSummary
Create PaginatedResult with summaries and total
Convert BookSummary to DTO
End: 200 OK with ListBooksResponseDTO

CLIENT ENVIRONMENT

On the other hand, the client environment is a piece of hardware or software that requests and uses services or data provided by a server. In our case, the client is an Android APP.

Android APP

The frontend is hosted in the **book-publishing-app**[30] repository. It is a native Android application that consumes the backend's RESTful API to manage books, authors, and collections. The chosen language is **Kotlin**, the framework is **Android SDK** with **Jetpack Compose** for the user interface, and **Gradle** as the dependency manager.

- **Kotlin**: a modern, statically typed language that runs on the JVM. Completely interoperable with Java, it offers a more concise and secure alternative. It mixes object-oriented programming with functional programming, reduces verbosity, and helps prevent Null Pointer Exceptions.
- **Android SDK**: Google's mobile platform that offers the necessary tools and APIs to develop native applications for Android devices.
- **Jetpack Compose**: modern and declarative toolkit for building native Android user interfaces. It allows creating the UI declaratively, simplifying interface development and maintenance.
- **Gradle Build Tool**: dependency manager, responsible for compilation, packaging, test execution, and APK generation.

Most Important Dependencies

- **Retrofit**: a typed HTTP client library for Android and Java. It simplifies communication with RESTful APIs through declarative interfaces.
- **OkHttp**: an efficient HTTP client for Android and Java. It is used as the basis for Retrofit and offers interceptors for logging and authentication.
- **Kotlinx Serialization**: Kotlin native serialization library to convert objects to JSON and vice versa, necessary for API communication.
- **Jetpack Navigation**: navigation component that manages navigation between application screens in a typed and secure way.
- **Material Design 3**: Google's design system that provides consistent and modern user interface components, with support for light and dark themes.
- **ViewModel**: component of the Android architecture that manages UI-related data in a lifecycle-aware manner.

[30] Repositor del Frontend. https://github.com/CescFe/book-publishing-app

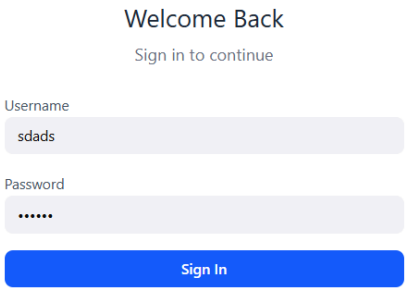
- **Kotlin Coroutines**: library for asynchronous programming that simplifies working with web operations and databases.
- **JUnit 5**: unit testing framework to verify business logic and ViewModels.
- **Compose Testing**: library for declaratively testing Compose components.
- **Spotless**: static analysis and code formatting tool, which will ensure that the code has a consistent format and applies good practices.

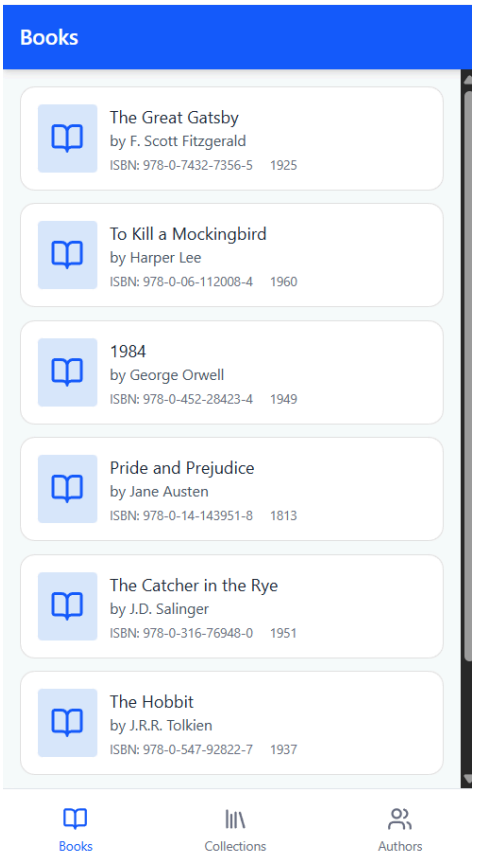
Applied Methodologies

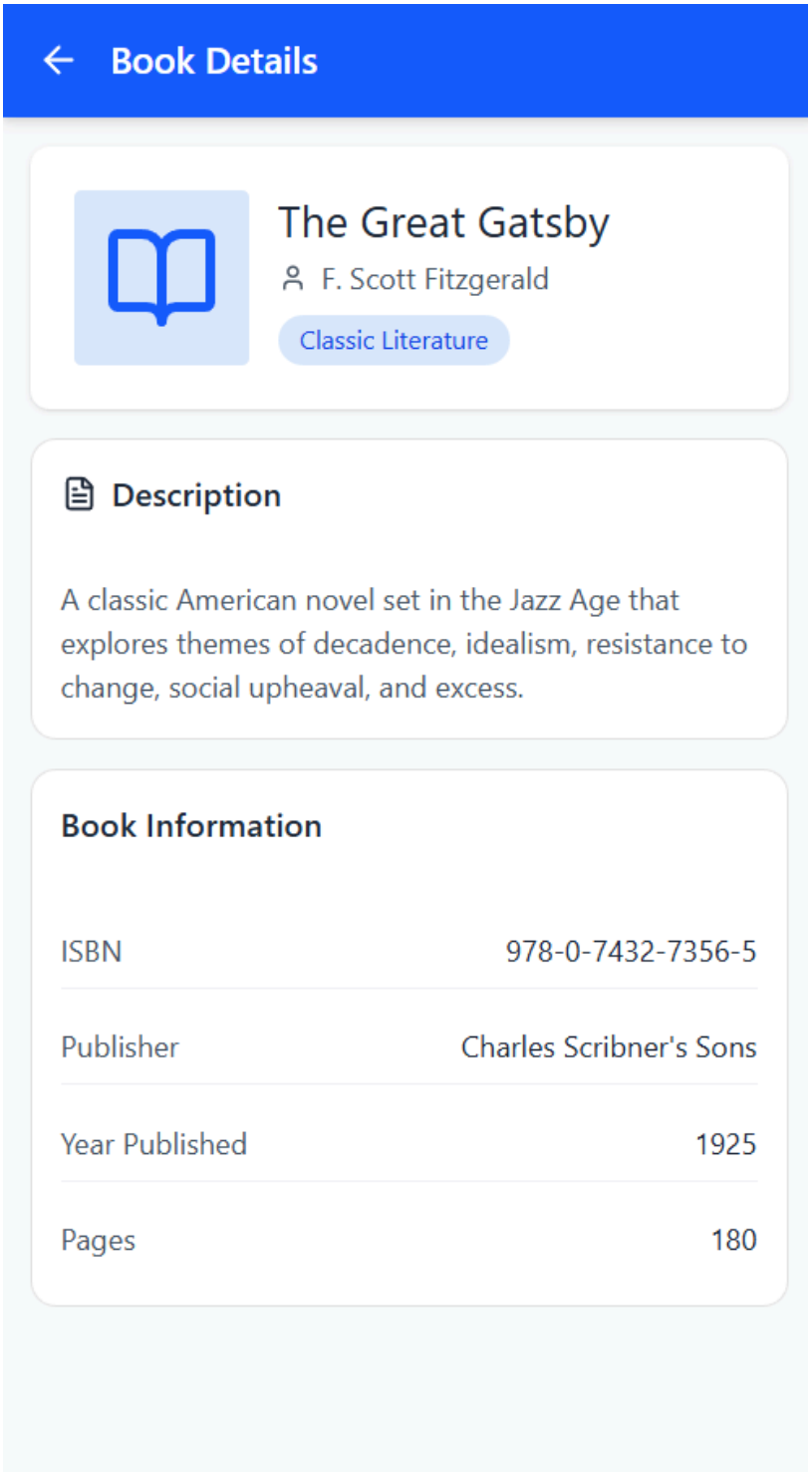
- **Model-View-ViewModel** (MVVM): architectural pattern that separates presentation logic from business logic. ViewModels manage the UI state and communication with the domain layer, while Composables represent the view and react to state changes.
- **Vertical Slice Architecture**: organizes code by feature or use case, grouping related components. In this case, the Vertical Slice has been applied on top of the clean architecture, dividing the classes first by contexts (auth, author, book, collection, and shared) and then by layers (ui, domain, data). This facilitates the localization of all code related to a specific functionality in a single place.

Mockups

The design of some Mockups is attached below:

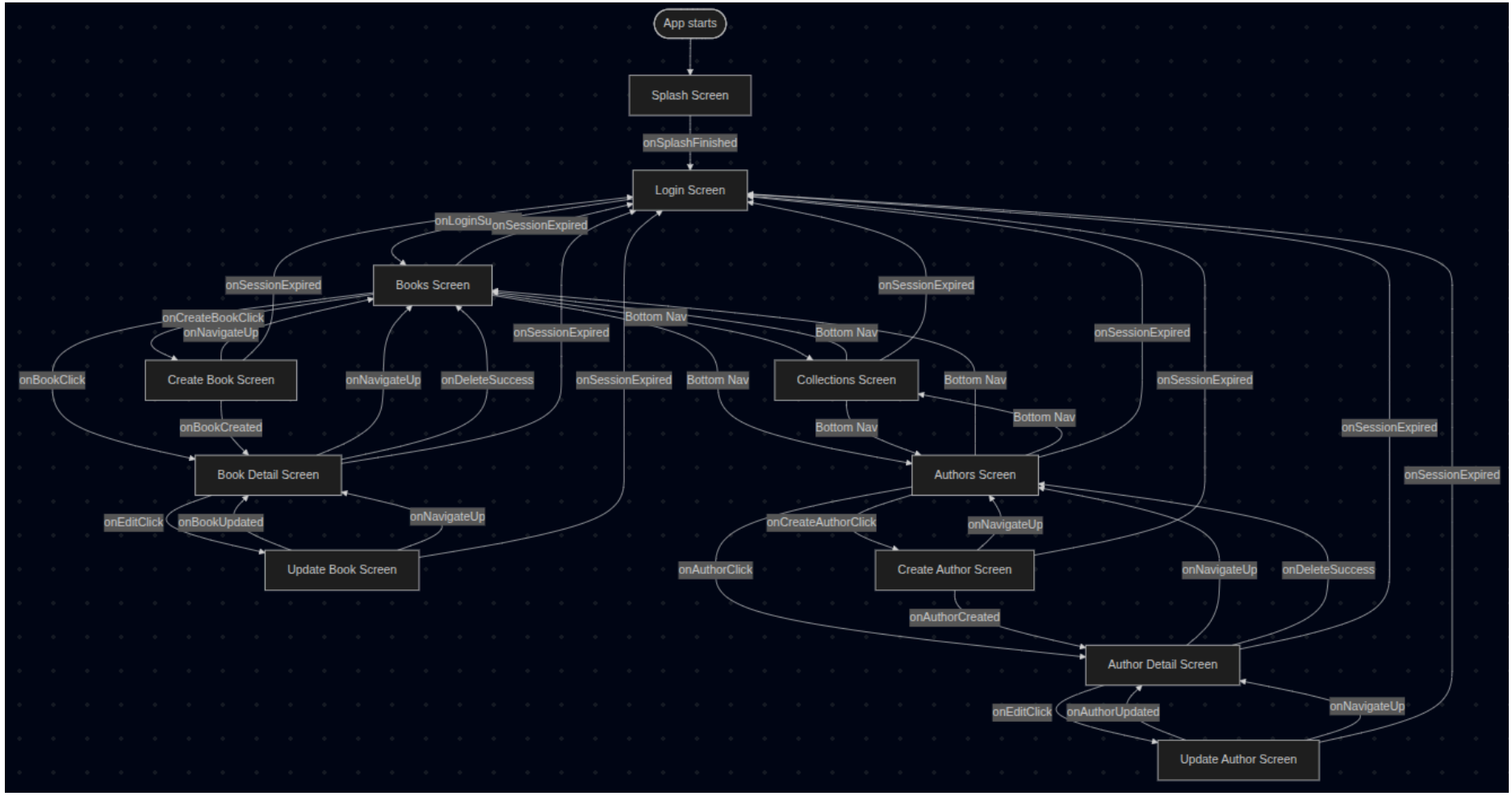






Mockups of the authentication page, the book listing and the book's detail

Mobile APP Map



API CONTRACT IMPLEMENTATION

OpenAPI Endpoint Modelling

The first step is to have the models designed (as already done in previous chapters). The next step, in order to be able to use them, is to add actual content to the repository.

```yaml
openapi: 3.1.2
info:
  title: Book Publishing API
  version: 1.1.0
  description: |-
    API for managing books, authors, and collections in a publishing system.
    This API provides CRUD operations for books, collections and authors secured with jwt.
  contact:
    name: FrancescFe
    url: https://github.com/FrancescFe
```

API specification header

Next, we can start defining the CRUD operations that make up the REST API. A style has been applied that aims to keep the main file (usually named openapi.yaml) as lightweight as possible, extracting shared components and schemas into separate files. This approach avoids code duplication and improves the maintainability and readability of the specification.

```yaml
  /api/v1/authors:
    get:
      tags:
        - GetAllAuthors
      operationId: getAuthors
      summary: Get all authors
      description: Retrieve a paginated list of all authors
      parameters:
        - $ref: './shared/components/parameters/pagination.yaml#/page'
        - $ref: './shared/components/parameters/pagination.yaml#/limit'
        - $ref: './shared/components/parameters/pagination.yaml#/search'
      responses:
        '200':
          $ref: './author/components/responses/authors-list.yaml'
        '400':
          $ref: './shared/components/responses/errors/bad-request.yaml'
```

GET Authors operation retrieving the list of authors

OpenAPI Components Modelling

Throughout this process, we follow the previously defined **Work Implementation Cycle**. This implies creating a work item, performing functional refinement from a product perspective, and carrying out a technical refinement where, ideally, a proposed solution approach is included and documented in the GitHub issue description.

It is true that with this methodology a significant amount of time is invested before starting the “strict” implementation phase; however, this also reduces the time required for coding, helps avoid errors, and minimises the need for later refactoring.

Example of a refined ticket

At this point, we can finally implement the object using OpenAPI syntax.

Afterwards, additional API schemas and the remaining endpoints are implemented.

```yaml
type: object
required:
  - id
  - full_name
properties:
  id:
    $ref: '../../../shared/components/schemas/id.
  full_name:
    type: string
    description: Full name of the author
    minLength: 1
    maxLength: 255
    example: "John Ronald Reuel Tolkien"
  pseudonym:
    type: string
    description: Pseudonym of the author
    minLength: 1
    maxLength: 255
    example: "J.R.R. Tolkien"
  email:
    type: string
    format: email
    description: Author's contact email
    example: "tolkien@example.com"
```

Example of the AuthorSummary structure

OpenAPI Generator Plugin Configuration

The most effective way to establish a real contract for our REST API that aligns with the API-First methodology being applied is through code generation. If the backend must use predefined classes, this guarantees that it complies with the contract and that the API documentation remains the single source of truth. Code generation also significantly reduces the likelihood of errors.

There are currently several tools available for this purpose, with **OpenAPI Generator**[31]being the most widely adopted. This is the tool selected for this project (version 7.15). Integrating and configuring this plugin for the first time, without prior experience, is not particularly intuitive; the most effective approach was to consult the official documentation[32] and review public GitHub repositories that already use it.

To configure the generator, it is necessary to specify the dependencies required by the generated code at compile time. These dependencies are declared as compileOnly because this project only generates code to be consumed by other projects and is not executed directly. The main dependencies include Jackson for JSON serialization, Jakarta Validation for data validation, Spring Boot for REST controllers, and Swagger/SpringDoc for API documentation.

```
dependencies { Add Starters...
    compileOnly("com.fasterxml.jackson.core:jackson-databind:2.20.0")
    compileOnly("jakarta.validation:jakarta.validation-api:3.1.1")
    compileOnly("io.swagger.core.v3:swagger-annotations:2.2.37")
    compileOnly("org.springframework.boot:spring-boot-starter-web:3.5.6")
    compileOnly("org.springframework.boot:spring-boot-starter-validation:3.5.6")
    compileOnly("org.springdoc:springdoc-openapi-starter-webmvc-ui:2.8.13")
    compileOnly("io.swagger.core.v3:swagger-models:2.2.37")
    compileOnly("io.swagger.core.v3:swagger-core:2.2.37")
}
```

Dependencies required to use the OpenAPI Generator plugin

The generator configuration options allow fine-grained control over how the code is generated. For example, interfaceOnly generates only interfaces because the implementation is manually provided in the backend; useBeanValidation adds validation annotations to the models; and modelNameSuffix appends the “DTO” suffix to class names to follow project conventions. The useSpringBoot3 option ensures compatibility with Spring Boot 3.

[31] OpenAPI Generator website. https://openapi-generator.tech/
[32] OpenAPI Generator repository. https://github.com/OpenAPITools/openapi-generator

```
modelNameSuffix.set("DTO")
configOptions.set(
    mapOf(
        "useTags" to "true",
        "useBeanValidation" to "true",
        "delegatePattern" to "false",
        "dateLibrary" to "java8",
        "useSpringBoot3" to "true",
        "enumPropertyNaming" to "UPPERCASE",
        "interfaceOnly" to "true",
        "modelPropertyNaming" to "original",
        "serializationLibrary" to "jackson",
        "skipDefaultInterface" to "true",
```

Plugin Configuration

Once configured, the plugin provides two main Gradle tasks that can be executed from the command line: openApiValidate and openApiGenerate. The first task validates the OpenAPI specification to ensure it is correct and, when the recommend option is enabled, provides suggestions for improvement. The second task generates the Kotlin Spring code according to the specified configuration. In addition, the build task has been configured to depend on code generation, so every time the project is compiled, the generated code is automatically updated if there are changes in the specification.

Publishing the External Library

A further improvement to the API contract consists of packaging the autogenerated classes and publishing them as an external library that will be consumed by the backend. This is the key step to guaranteeing a strict API-First implementation, treating the API as a product in its own right. This approach facilitates scalability, enforces a clear separation of responsibilities by fully decoupling the contract from the backend, and enables reuse across multiple projects while ensuring consistency. It also enforces version control, managing evolution, integration, and backward compatibility with previous versions.

The project has been configured to publish the external library as a Maven package hosted on GitHub Packages. To automate this process, a CI/CD GitHub Action has been created to publish the package. A Personal Access Token (PAT) was generated and stored as a GitHub Secret variable.

BACKEND IMPLEMENTATION

Implementation of a Use Case in the Backend

Now we will discuss the process followed to implement a use case in the backend, using the creation of an author as an example. We start from the assumption that the endpoint has already been defined in the contract and the corresponding classes have been generated thanks to OpenAPI Generator. The first step is modeling the domain layer.

Domain Layer Modeling

In the domain layer, we apply DDD, and we assist ourselves by defining Value Objects and establishing business rules. The Author Value Object is defined as a data class, and each of its attributes are also Value Objects defined as value classes. We apply business rules and contract validations at the moment of constructing each of them.

In the domain, we must also declare the interface of the repository methods we need

```
data class Author(  Cesc Fe +1
    val id: AuthorId,
    val fullName: FullName,
    val pseudonym: Pseudonym? = null,
    val biography: Biography? = null,
    val email: Email? = null,
    val website: Website? = null,
    val audit: Metadata? = null,
)

@JvmInline  Fran Fe +1
value class AuthorId(
    val value: UUID,
) {
    companion object {  Fran Fe +1
        private val UUID_REGEX = Regex( pattern = "^[0-9a-fA-F]{8}-[0-9a-fA-F]{4}

        fun generate(): AuthorId = AuthorId( value = UUID.randomUUID())  Cesc Fe

        fun fromString(value: String): AuthorId {  Fran Fe
            require(value.matches(UUID_REGEX)) {
                throw AuthorDomainException.authorIdInvalidFormat( id = value)
            }
            return AuthorId( value = UUID.fromString( name = value))
        }
    }
}
```

Persistence Modeling in the Infrastructure Layer

In the infrastructure layer, we need to model the author entity that will help us persist information in the database using JPA. Metadata management has been delegated to JPA's AuditingEntityListener.

```
@Entity  12 Usages  Cesc Fe
@Table(name = "author", schema = "publishing")
data class AuthorEntity(
    @Id
    @Column(name = "id", columnDefinition = "UUID")
    val id: UUID,
    @Column(name = "full_name", nullable = false)
    val fullName: String,
    @Column(name = "pseudonym")
    val pseudonym: String?,
    @Column(name = "biography")
    val biography: String?,
    @Column(name = "email")
    val email: String?,
    @Column(name = "website")
    val website: String?,
): AuditableEntity()
```

We must also define the methods we need in AuthorJpaEntityRepository, as well as implement it in JpaAuthorRepository. Next, the persistence mapper must be implemented, which allows us to convert a domain object into a persistence object and vice versa.

```
override fun save(author: Author): Author {  Cesc Fe
    val entity = authorMapper.fromDomain(author)
    return authorMapper.toDomain( entity = authorJpaEntityRepository.save(entity))
}
```

Service Modeling in the Application Layer

The application layer orchestrates the use case. First, we define the use case interface, then the mapper that allows us to convert the input value into a domain object or vice versa.

```
interface CreateAuthorUseCase {  19 Usages  1 i
    fun execute(command: Command): Author

    data class Command(  Cesc Fe +1
        val fullName: String,
        val pseudonym: String? = null,
        val biography: String? = null,
        val email: String? = null,
        val website: String? = null,
    )
```

Now we can implement the service:

```
@Service  1 Usage  Cesc Fe +1
@Transactional
open class CreateAuthorInteractor(
    private val authorRepository: AuthorRepository,
    private val mapper: CreateAuthorUseCaseMapper,
    private val authorDomainService: AuthorDomainService,
) : CreateAuthorUseCase {
    override fun execute(command: CreateAuthorUseCase.Command): Author {
        authorDomainService.ensureEmailUniqueness(command.email)
        val author = mapper.toDomain( input = command)
        return authorRepository.save(author)
    }
}
```

Controller Modeling in the Infrastructure Layer

Finally, we can implement the interface generated by OpenAPI Generator. The controller flow is: mapping the request DTO to Input Value (command in this case), executing the use case, mapping the response to response DTO, creating the URI resource that will be sent as a header in the response, and returning the response in the form of ResponseEntity.

```
@RestController  2 Usages  Cesc Fe +1
@Tag(name = "CreateAuthor")
open class CreateAuthorController(
    private val createAuthorUseCase: CreateAuthorUseCase,
    private val mapper: AuthorRestMapper,
) : CreateAuthorApi {
    override fun createAuthor(  1 Usage  Cesc Fe +1
        createAuthorRequestDTO: CreateAuthorRequestDTO,
    ): ResponseEntity<CreateAuthor201ResponseDTO> {
        val inputValues = mapDtoToInputValues(createAuthorRequestDTO)
        val createdAuthor = createAuthorUseCase.execute( command = inputValues)
        val responseDto = mapper.toDto(createdAuthor)
        val uri = buildResourceUri( authorId = createdAuthor.id.value)
        return ResponseEntity.created( location = uri).body(responseDto)
    }

    open private fun buildResourceUri(authorId: UUID): URI =  1 Usage  Cesc Fe
        ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand( ...uriVariableValues = authorId)
            .toUri()
```

Securing the Application

The application implements JWT (JSON Web Tokens) based authentication and role-based authorization. The system guarantees that only authenticated users access the resources for which they have permissions based on their role.

The implemented system provides secure authentication with stateless JWT, secure password storage with BCrypt, role-based authorization, flexible configuration with environment variables, and separation between authentication and authorization.

This implementation follows security best practices and allows the system to scale and be maintained securely.

JWT Secret Configuration

The JWT Secret is the secret key used to sign and verify tokens. It is configured via environment variables to avoid exposing it. The JwtUtil class validates the secret, thus guaranteeing resistance to brute force attacks and the use of secure cryptographic algorithms.

Token Generation and Validation

The *generateToken()* method creates a JWT with username, issue date, expiration date, and signature (HMAC-SHA256 with the secret). The token is validated with *validateToken()* and compared with the authenticated user and expiration.

User Authentication and Password Management

Users are configured with Spring Boot's AuthProperties, allowing different configurations per profile. In the PasswordConfig class, BCrypt is used for password hashing.

The authentication process goes through UserService, which manages loading users upon application startup, stored encoded in cache for faster loading. Each user has an associated ROLE_ADMIN or ROLE_USER.

When the client sends credentials to the */api/v1/auth/login* endpoint, the AuthenticationManager class validates the username and password. Upon success, it returns a JWT with a 15-minute expiration date, a scope, and a userId.

Role Management and Authorization

The ADMIN role has full access to the application (read, write, and delete), while the USER role has only read access. ScopeService determines permissions according to the role, while SecurityConfig defines authorization rules: write operations are reserved for ADMIN, read operations for USER and ADMIN, and public endpoints (login, health, and swagger UI) are defined.

JWT Filter and Request Validation

The JwtRequestFilter intercepts every HTTP request to validate the JWT with the JwtUtil class.

Error Handling

The application implements a centralized exception management system that guarantees consistent responses and facilitates maintenance. A GlobalExceptionHandler has been implemented that intercepts and processes all exceptions, and domain exceptions are encapsulated in business rules.

GlobalExceptionHandler

The GlobalExceptionHandler is a class that centralizes exception management throughout the application. Spring automatically intercepts exceptions thrown from any REST controller. Different types of exceptions are processed:

- **Domain**: grouped in BookDomainException, AuthorDomainException, and CollectionDomainException.
- **Validation**: MethodArgumentNotValidException, ConstraintViolationException.
- **Infrastructure**: HttpMessageNotReadableException, NoHandlerFoundException, MethodArgumentTypeMismatchException.
- **Security**: BadCredentialsException.
- **Generic**: Exception.

Standardized Response Model

All error responses use the same model defined in ApiError, which provides a consistent structure.


```
{
    "status": 409,
    "error": "Conflict",
    "message": "Author with email 'jkrowling@example.com' already exists",
    "code": "EMAIL_ALREADY_EXISTS",
    "details": {
        "path": "/api/v1/authors",
        "exceptionType": "AuthorDomainException"
    },
    "timestamp": "2026-01-08T01:06:51.628599473Z"
}
```

FRONTEND IMPLEMENTATION

Implementation of a User Story in the Frontend

To implement a user story in the APP, for example, viewing the list of collections, we will follow these steps.

Domain Layer Implementation

In this layer, domain objects are modeled, ensuring that the same validations applied in the backend are propagated, and the ports used by the data layer are implemented.

```
data class CollectionSummary(
    val id: String,
    val name: String,
    val readingLevel: ReadingLevel?,
    val primaryLanguage: Language?,
    val primaryGenre: Genre?
)
```

Domain model and port of the collection list

```
10 Usages  2 Implementations
interface CollectionsRepository {
    14 Usages  2 Implementations
    suspend fun getCollections(): DomainResult<List<CollectionSummary>>
}
```

Data Layer Implementation

In this layer, data is obtained from the backend, modeled as DTOs, mapping methods are created, the API response is defined, and the repository adapter is implemented.

```
7 Usages
@Serializable
data class CollectionSummaryDTO(
    val id: String,
    val name: String,
    @SerialName( value = "reading_level")
    val readingLevel: String? = null,
    @SerialName( value = "primary_language")
    val primaryLanguage: String? = null,
    @SerialName( value = "primary_genre")
    val primaryGenre: String? = null
)
```

DTO and repository implementation

```
8 Usages
class CollectionsRepositoryImpl(private val collectionsApi: CollectionsApi) : CollectionsRepository {
    11 Usages
    override suspend fun getCollections(): DomainResult<List<CollectionSummary>> = try {
        val response = collectionsApi.getCollections()
        val collections = response.data.map { it.toDomain() }
        DomainResult.Success( data = collections)
    } catch (e: Exception) {
        RepositoryErrorHandler.handleException(e)
    }
}
```

ViewModel Implementation in the UI Layer

The ViewModel manages the UI state (loading, error, success) and coordinates with the repository.

```
7 Usages
data class CollectionsUiState(
    val collectionSummaries: List<CollectionSummary> = emptyList(),
    val isLoading: Boolean = false,
    @get:StringRes val errorResId: Int? = null,
    val sessionExpired: Boolean = false
)

3 Usages
class CollectionsViewModel(
    private val collectionsRepository: CollectionsRepository = CollectionsRepositoryImpl(
        RetrofitClient.collectionsApi
    )
) : ViewModel() {

    11 Usages
    private val _uiState = MutableStateFlow( value = CollectionsUiState())
    val uiState: StateFlow<CollectionsUiState> = _uiState.asStateFlow()

    init {
        loadCollections()
    }
```

Screen Implementation in the UI Layer

First, we define the card for each collection list item in the CollectionSummaryCard class. Then, in CollectionsScreen, we define the Composable that shows the UI and observes and reacts to the state defined from the ViewModel.

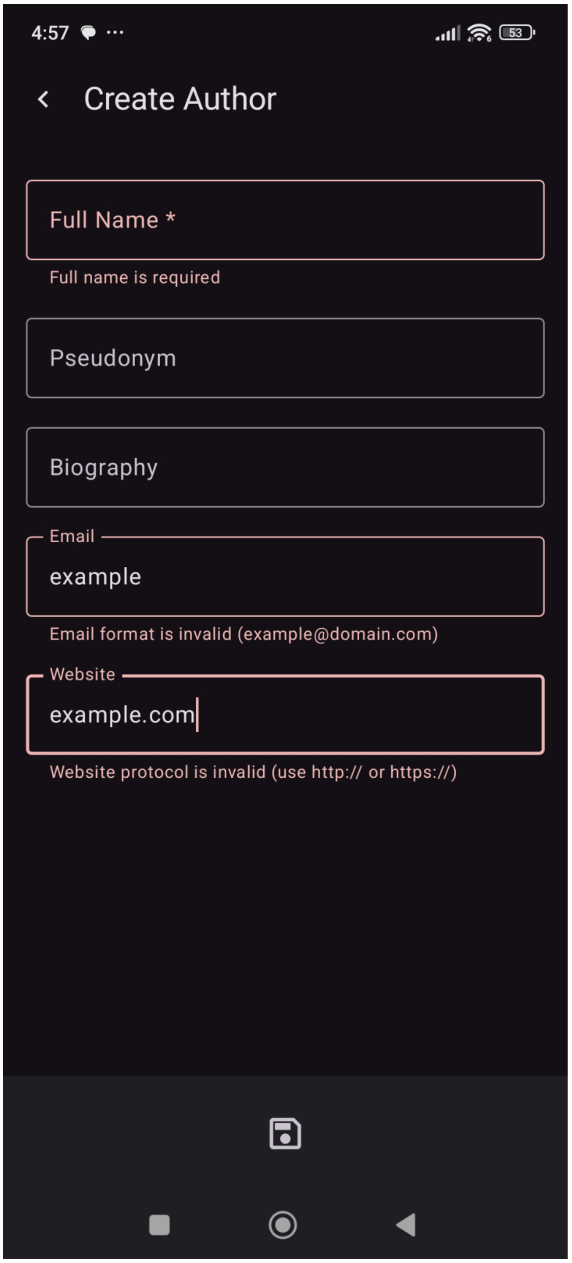
```
@Composable
internal fun CollectionsScreenContent(
    uiState: CollectionsUiState,
    onRetry: () -> Unit,
    onNavigate: (BottomNavItem) -> Unit = {}
) {
    Scaffold(
        modifier = Modifier.testTag( tag = "collections_screen"),
        topBar = {
            TopAppBar(
                title = { Text(text = stringResource("Collections List")) }
            )
        },
        bottomBar = {
            AppBottomBar(
                selectedItem = BottomNavItem.Collections,
                onItemClick = onNavigate
            )
        }
```

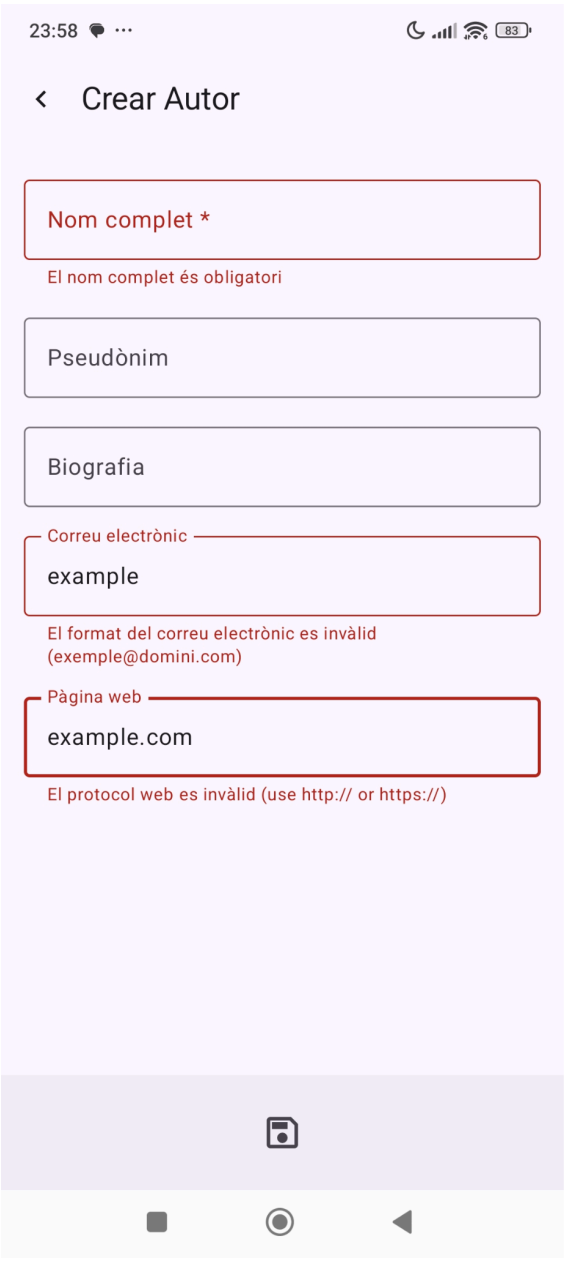
To allow the user to reach this screen, the AppNavigation class must be updated with the new navigation flow.

Preventing Errors with Warnings

The application implements a real-time validation system that shows warnings to the user as they type. This is possible through the updateField() function, which runs every time a field changes. Warning messages are displayed using Material Design 3's OutlinedTextField, thus ensuring adherence to Google's recommendations for native Android development best practices.

In editing or creation forms, the same validations as in the backend have been implemented to minimize errors.





Ensuring Functionality on Most Used Mobile Devices

An attempt has been made to guarantee functionality on the most used mobile devices in the following ways:

- **Compatible SDK configuration:** `build.gradle.kts` is configured with at least Android SDK 7.0 (Nougat), which is from 2016.
- **Responsive layouts in Jetpack Compose:** adaptive modifiers like `fillMaxWidth()`, `fillMaxSize()`, and `weight(1f)` have been used.
- **Support for long forms:** `verticalScroll()` is used with small screens in mind.
- **Convention over Configuration:** adaptive components are delegated to Material Design 3.
- **Previews:** use of previews to validate in different configurations.

Internationalization

The application is duly configured to support locations dynamically based on the language configured on the user's Android device. The application supports the following three locations:

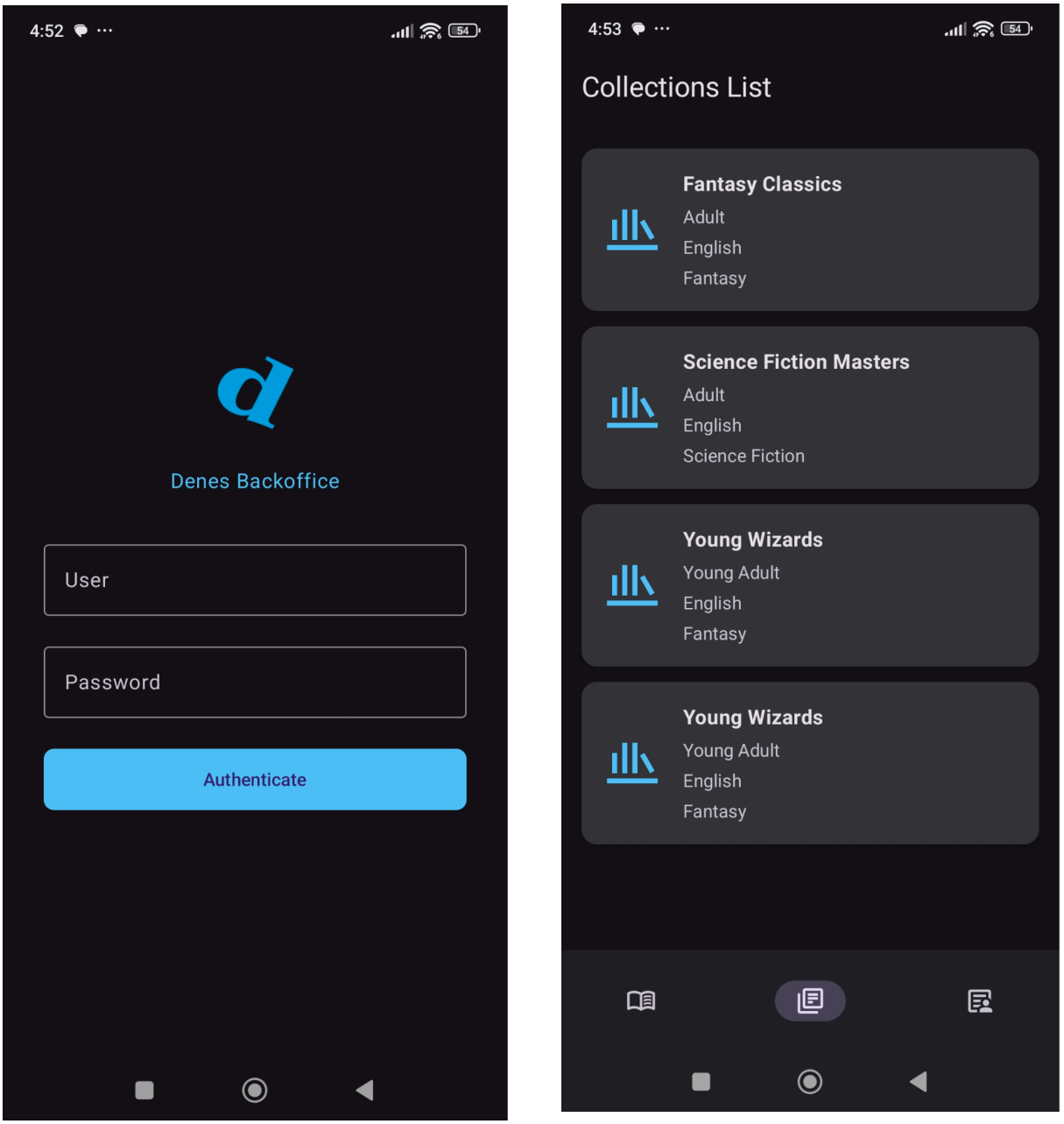
- **English:** language in which the app has been implemented and default location.
- **Catalan:** normative Valencian dialectal variant.
- **Spanish:** normative Castilian dialectal variant.

Theme and Appearance

The application is duly configured to support theme changes dynamically based on the user's Android device configuration.

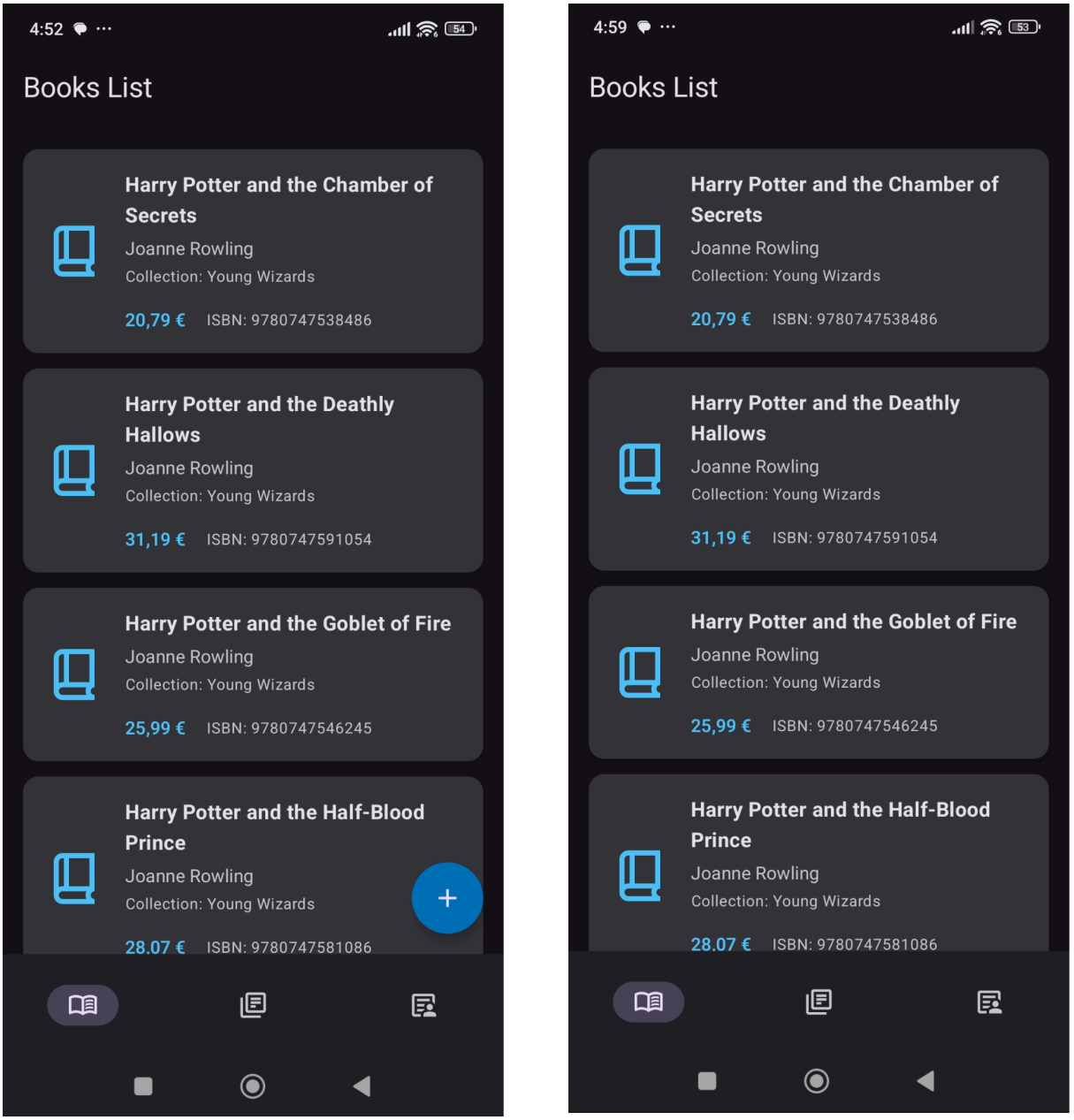
- **Light mode:** applied automatically when the device uses light mode.
- **Dark mode:** applied automatically when the device uses dark mode.
- **Corporate colors:** the blue color scheme uses Editorial Denes' corporate color.
- **Material Design 3:** Uses the Material 3 color scheme for a consistent theme.
- **Material Symbols and Icons:** Uses Google Fonts Symbols & Icons to adopt Android standards.
- **Active actions according to user role:** automatically validates the user's role and adjusts the interface accordingly, allowing only those actions for which the user has permissions.

MVP Images



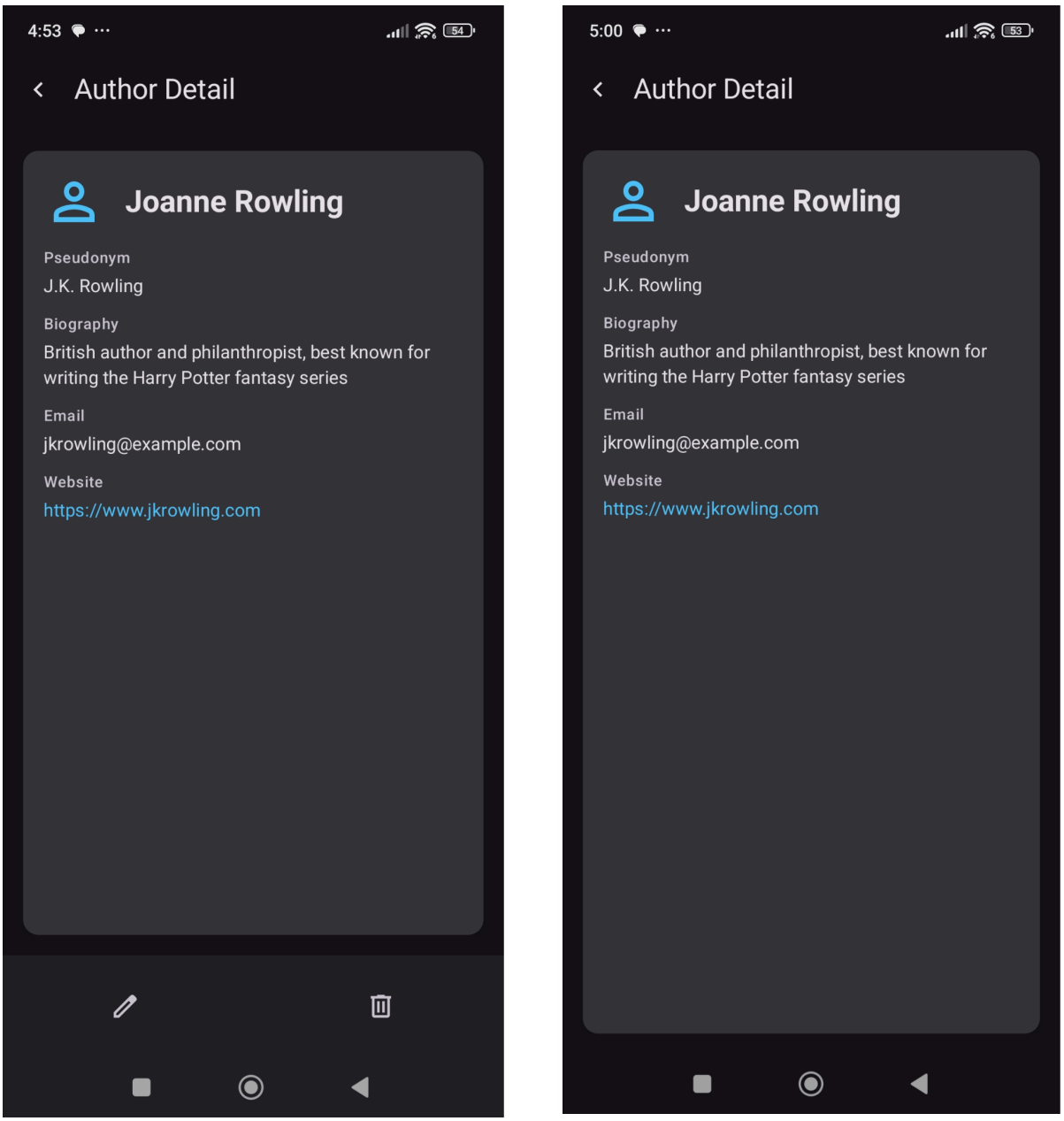
On the left, the authentication screen. On the right, the collection list screen.

Book List (Admin Session vs. Base User Session)



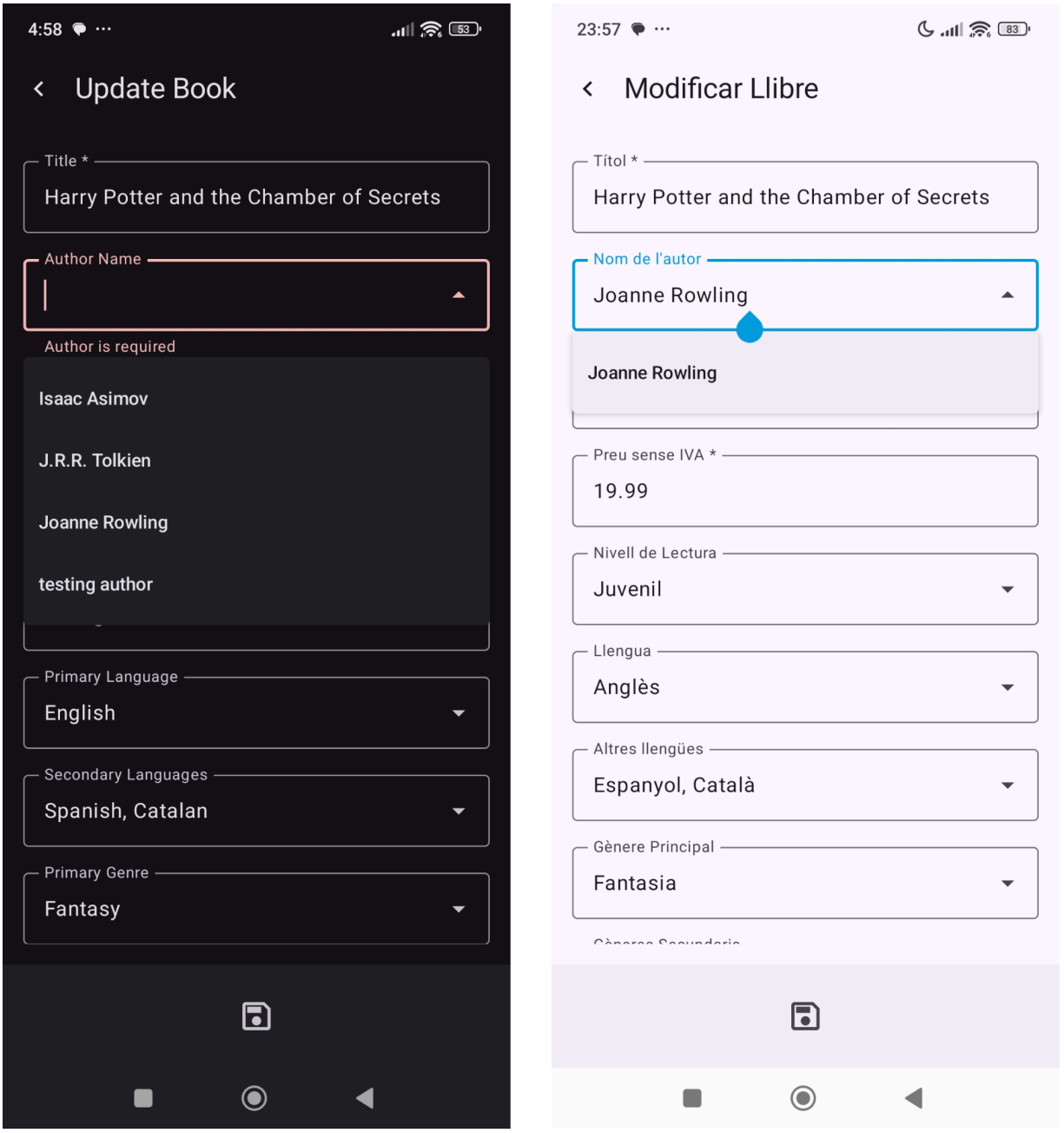
Comparison of the administrator session against the session of a base user from the book list screen. The difference is that the Floating Action Button at the bottom right, which allows navigating to the author creation view, is only present in the administrator session

Author Detail (Admin Session vs. Base User Session)



Comparison of the administrator session against the session of a base user from the author detail screen. The difference is that the Bottom Navigation Bar, which allows editing or deleting an author, is only present in the administrator session

Book Update (English Dark Mode vs. Valencian Light Mode)



Comparison of an administrator session against a base user session from the author detail screen. The difference is that the Bottom Navigation Bar allowing to edit or delete an author is only present in the administrator session

TESTING PHILOSOPHY

We followed a testing strategy aligned with the architectures: Hexagonal + DDD (backend) and MVVM (frontend). The goal is to ensure code quality and maintainability through tests that validate business logic and integration between layers.

The testing philosophy has two main concerns: to guarantee the correct behavior of the implemented functionality and to establish that behavior, creating a contract with future implementations.

This strategy ensures code quality, facilitates maintenance, and allows for early detection of regressions in the development cycle.

Fundamental Principles

To achieve this goal, the fundamental principle followed is **FIRST**:

- **Fast**: prioritizing fast unit tests without frameworks or infrastructure. Optimizing integration tests (slower) with mocks, testcontainers, and a limited context to ensure appropriate speed for their purpose.
- **Independent**: no dependencies between tests or shared states. BeforeEach only configures mocks, not states.
- **Repeatable**: idempotent, guaranteeing consistent results. Data is generated with ObjectMothers and specific, reproducible datasets.
- **Self-validating**: binary result (pass/fail) thanks to clear assertions and verifications.
- **Thorough**: covers both happy paths and bad paths.

Coverage and Strategy

- **Separation by layers**: both in the backend and frontend, each layer has its own tests.
- **Realistic tests**: in the backend, integration tests use Docker containers (Testcontainers) with a real PostgreSQL instance.
- **Maintainability**: use of the Object Mother pattern to create test data consistently and reusably.
- **Pyramid distribution**: more unit tests (fast and cheap), fewer integration or instrumentation tests (slower but necessary).
- **Avoid duplication**: overlapping coverage between tests has been avoided.
- **CI/CD integration**: they are part of GitHub Actions.
- **Edge cases**: validation of business rules and exception handling.
- **Framework functionalities are not tested**: reliance on Spring Boot and JPA to avoid duplicating framework tests.

Backend

We implemented 254 tests, covering 88% of code lines and 77% of branches.

Unit Tests

Validate isolated components without external dependencies or Spring.

- **Domain Tests**: validate entities and value objects, business rules, and calculations.
- **Use Case Tests**: validate application logic with mocked repositories and domain services. Verify successful flows and exception handling.
- **Mapper Tests**: validate transformation between layers.
- **Controller Tests**: validate exception handling and delegation to use cases, without loading the Spring context.

Integration Tests

Validate integration between components and with real infrastructure.

- **Repository Tests**: use Testcontainers with a real PostgreSQL instance. Validate persistence, JPA mappings, audit fields, JSONB handling, and pagination.
- **Controller Tests**: use MockMvc with a partial Spring context. Validate JSON serialization, HTTP validations, security (JWT), and HTTP responses.
- **Dataset-based Tests**: load SQL scripts to validate JPA mappings and audit fields with realistic data.

Tools and Technologies

- **JUnit 5**: testing framework.
- **Mockito Kotlin**: mocking for unit tests.
- **Testcontainers**: Docker containers for integration tests.
- **Spring Boot Test**: utilities for tests with Spring.
- **MockMvc**: REST controller testing.
- **Spring Security Test**: security and JWT authentication testing.
- **Object Mothers**: pattern for creating test objects consistently.

Frontend

We implemented 390 tests: 320 unit tests and 70 instrumented tests.

Unit Tests

Execute on the JVM without an Android device.

- **ViewModels**: validate presentation logic, state management, and data transformation.
- **Repositories**: validate data transformation and error handling.
- **Models and DTOs**: validate serialization/deserialization and transformation between layers.
- **Domain Logic**: business validations and rules.
- **Utilities**: shared components like error mapping and token managers.

Instrumented Tests (UI)

Execute on a device or emulator. Validate the user interface with Jetpack Compose Testing.

- **Screens**: rendering, states, and navigation.
- **Shared Components**: dialogs, buttons, and reusable elements.
- **User Flows**: creation, editing, and deletion of resources.
- **Permissions and Roles**: visibility based on permissions.

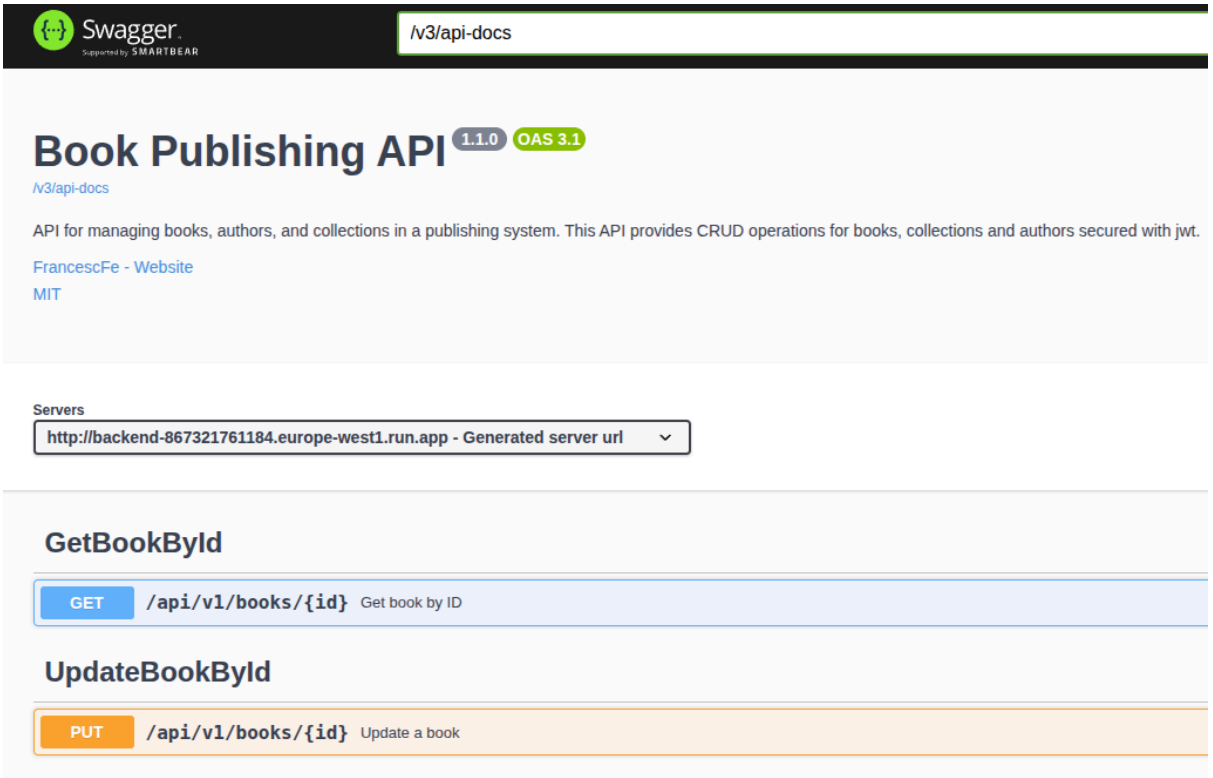
Tools and Technologies

- **JUnit 4**:primary testing framework for the Android environment.
- **Kotlin Coroutines Test**: management and verification of asynchronous code in ViewModels and repositories.
- **Compose UI Test**: official library for verifying the behavior and state of the user interface built with Jetpack Compose.
- **Manual Mocks for Dependencies**: lightweight, controlled implementations to simulate repositories, services, and external clients, ensuring test isolation.

DOCUMENTATION

External Code Documentation

The OpenAPI specification of this project is excellent API documentation, which should be the only channel of communication with other services or clients. The OpenAPI Generator plugin already generates the documentation and it is accessible from the backend at the following route:



/swagger-ui/index.html

No further documentation has been generated as it is not necessary at the current system stage nor a best practice recommendation.

A future improvement could be integrating Redocly (economical), Stoplight (premium), or using a static web template (and hosting it on GitHub Pages) to improve the current presentation of SwaggerUI documentation.

Internal Code Documentation

The OpenAPI Generator plugin already adds internal documentation to the classes it creates. No further documentation has been generated as it is not necessary nor a best practice recommendation.

```kotlin
/**
 *
 * @param fullName Full name of the author
 * @param id Unique identifier
 * @param pseudonym Pseudonym of the author
 * @param biography Brief biography of the author
 * @param email Author's contact email
 * @param website Author's personal or professional website
 * @param createdAt Timestamp when the record was created
 * @param createdBy Identifier of the user/system who created the record
 * @param updatedAt Timestamp when the record was last updated
 * @param updatedBy Identifier of the user/system who last updated the record
 */
data class CreateAuthor201ResponseDTO(  9 Usages

    @get:Size(min=1,max=255)
    @Schema(example = "John Ronald Reuel Tolkien", required = true, description = "
    @get:JsonProperty( value = "full_name", required = true) val fullName: kotlin.St

    @Schema(example = "da420b0a-64aa-470d-991e-7fcb7a936229", readOnly = true, desc
    @get:JsonProperty( value = "id") val id: java.util.UUID? = null,

    @get:Size(min=1,max=255)
    @Schema(example = "J.R.R. Tolkien", description = "Pseudonym of the author")
    @get:JsonProperty( value = "pseudonym") val pseudonym: kotlin.String? = null,
```

An attempt has been made to apply the best possible practices and design patterns to ensure self-explanatory, well-structured, and simple-to-understand code. Furthermore, strict version control has been carried out applying Conventional Commits[33]. Adding to this that all work and its refinement has been documented with work parts available on GitHub three clicks away, it has been decided that adding extra in-code comments does not add value; on the contrary: it worsens code readability and is a risk as they can easily become obsolete.

User Manual

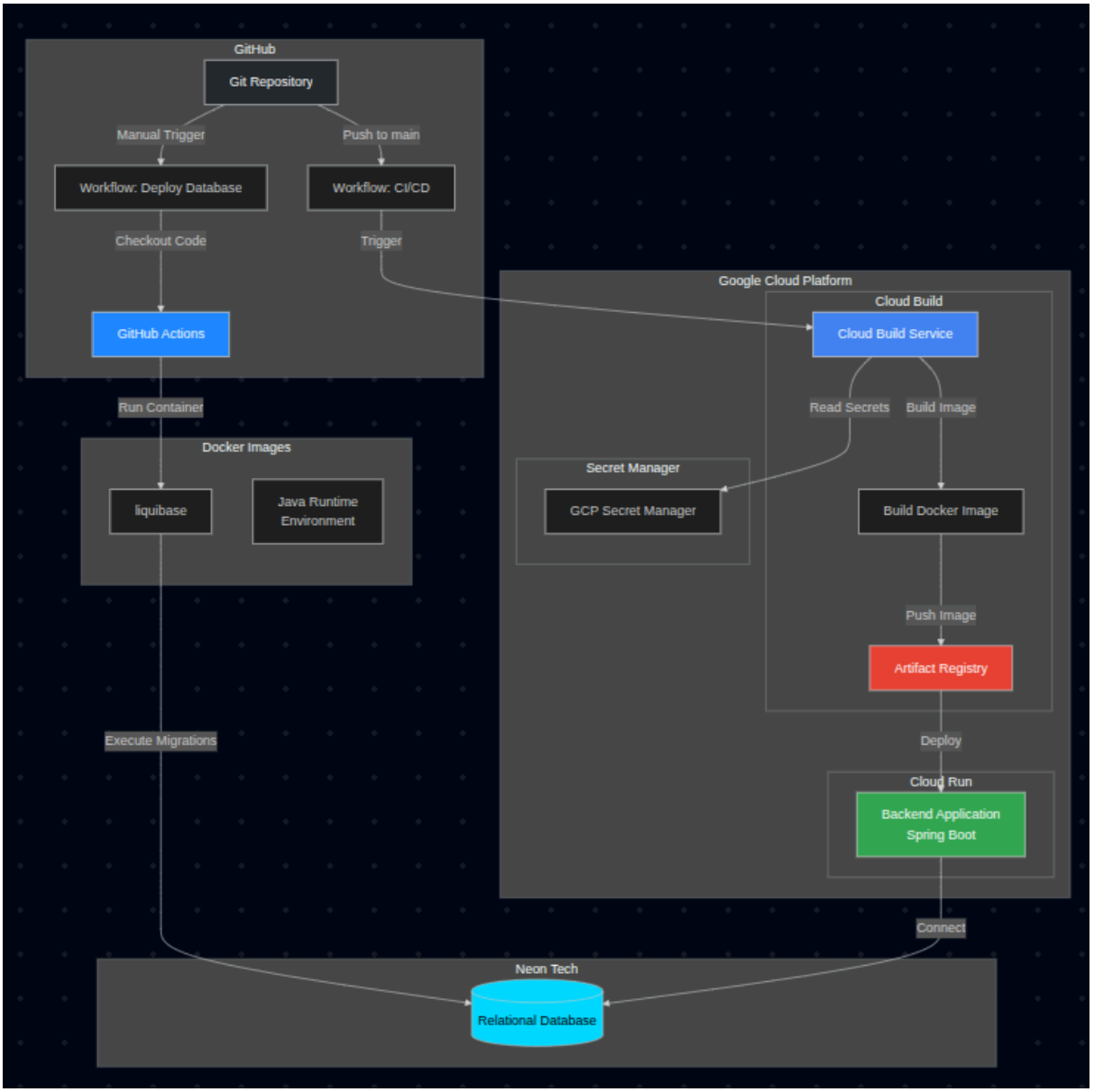
A great effort has been made so that the APP does not need a user manual and contains an invisible tutorial that guides the user through its functionalities and navigation. That is why Google's app design recommendations have been applied; forms guide the user with error messages, and the app's minimalist design does not leave much room for mistakes.

A user manual requires constant maintenance to not become outdated and useless quickly, while also being a sign of an application with design problems.

[33] Conventional Commits. https://www.conventionalcommits.org/en/v1.0.0/

DEPLOYMENT

Deployment Diagram



Deployment diagram of the server environment

Server Deployment Description

The system uses an automated deployment with Docker and Google Cloud Platform, divided into two main processes.

Database Deployment

A docker-compose has been configured to execute the Liquibase migration in a controlled manner. This is useful not only for database deployment but also for virtualization in testing and local development environments.

```yaml
services:
  postgres:
    container_name: postgres
    image: ${POSTGRES_IMAGE_VERSION}
    environment:
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    ports:
      - ${POSTGRES_PORT}:5432
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d ${POSTGRES_DB}"]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - book-publishing-network

  liquibase:
    container_name: liquibase
    image: ${LIQUIBASE_IMAGE_VERSION}
    platform: linux/amd64
    depends_on:
      postgres:
        condition: service_healthy
    restart: on-failure
    volumes:
      - ./src/main/resources/db/changelog/db.changelog-master.yaml:/liquibase/changelog.yaml:ro
      - ./src/main/resources/db/changelog/changes:/liquibase/changes
    command: [
      "--changeLogFile=changelog.yaml",
      "--url=jdbc:postgresql://postgres:5432/${POSTGRES_DB}",
      "--username=${POSTGRES_USER}",
      "--password=${POSTGRES_PASSWORD}",
      "update"
    ]
    networks:
      - book-publishing-network
```

Configured docker-compose

The production environment database lives on NeonTech cloud servers, because it offers a free plan that suited our needs. It offers 0.5GB storage capacity (more than enough for our use). A GitHub Action has been configured to automate the Liquibase migration; this allows not depending on entering the Neon UI to make schema changes: everything is controlled from the backend and GitHub.

```yaml
name: Deploy Database to pro

on:
  workflow_dispatch:

jobs:
  deploy-liquibase:
    runs-on: ubuntu-latest
    env:
      LIQUIBASE_IMAGE_VERSION: liquibase/liquibase:4.19.0

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Run Liquibase update
        run: |
          docker run --rm \
            -v ${{ github.workspace }}/src/main/resources/db/changelog:/liquibase/changelog:ro \
            ${{ env.LIQUIBASE_IMAGE_VERSION }} \
            --changeLogFile=changelog/db.changelog-master.yaml \
            --url="jdbc:postgresql://${{ secrets.NEON_DB_HOST }}/${{ secrets.NEON_DB_NAME }}?sslmode=require" \
            --username="${{ secrets.NEON_DB_USER }}" \
            --password="${{ secrets.NEON_DB_PASSWORD }}" \
            --driver=org.postgresql.Driver \
            update

      - name: Verify deployment
        run: |
          echo "✅ Liquibase migration completed successfully"
```

GitHub Action to execute Liquibase migration

Backend Deployment

The backend deployment is completely automated:

1. **Trigger:** When a push is made to the `main` branch, Google Cloud Build is automatically activated.
2. **Image Build:** Cloud Build executes the `cloudbuild.yaml` file which:
    - Read secrets from Google Secret Manager (`GITHUB_USERNAME` and `TOKEN_PAT`).
    - Builds the Docker image using the multi-stage Dockerfile.
    - Tags the image with the Artifact Registry repository name.
3. **Publication:** The Docker image is published to Google Artifact Registry.
4. **Deployment:** The image is automatically deployed to Google Cloud Run.
1. **Configuration:** The application starts with the `pro` profile that uses the environment variables configured in Cloud Run.

Prerequisites

For deployment, the following must be configured:

- **Google Cloud Platform:**
  - Project created with billing enabled.
  - Cloud Build API enabled.
  - Artifact Registry API enabled.
  - Cloud Run API enabled.
  - Secret Manager configured with GitHub credentials.
- **GitHub:**
  - Secrets configured for the database.
  - Secrets configured for authentication.
- **Neon Database:**
  - PostgreSQL database created.
  - Access credentials available

To carry out a successful deployment, it is necessary to configure several environment variables responsible for:

- Consuming the external library of the API specification contract.
- Authentication and authorization with JWT.
- Database connection.
- Dockerizing the environment.

Server Description

Throughout the app development, a server hosted on the Render cloud was used, but due to it being intended more for proofs of concept and having a policy of putting the server to sleep after 15 minutes of inactivity (needing about 3 minutes to wake up again), it was decided to migrate to Google Cloud Platform.

The Render server will remain as a staging or development environment but is currently deactivated, until we configure a dev or stg database.

Google Cloud Platform

The application is deployed to Google Cloud Platform using the following services:

- **Google Cloud Run**
  - Service: Cloud Run (serverless container platform)
  - Region: `europe-west1`
  - Features:
    - Container execution without server management
    - Automatic scaling according to load
    - Included in free plan
    - Billing per use (only execution time is paid)
    - Automatic HTTP/2 and HTTPS support
    - Integration with other GCP services
- **Google Artifact Registry**
  - Service: Artifact Registry
  - Region

    - Repository
    - Function: Storage of generated Docker images
    - URL
- **Google Cloud Build**
    - Service: Cloud Build (CI/CD)
    - Function: Automated build and deployment
    - Trigger: Push to GitHub repository `main` branch
    - Configuration: File `cloudbuild.yaml`
- **Google Secret Manager**
    - Service: Secret Manager
    - Function: Secure credential storage
    - Stored secrets:
        - GitHub user for private package access
        - GitHub Personal Access Token

Neon Database

- **Service:** Neon (PostgreSQL Serverless)
- **Type:** Managed PostgreSQL Database
- **Features:**
    - PostgreSQL 16
    - Mandatory SSL connections
    - Serverless plans with pay-per-use

GitHub Actions

- **Service:** GitHub Actions (CI/CD)
- **Function:** Execution of workflows for validation and deployment
- **Main Workflows:**
    - Pull Request Validation: Executes tests and code validations
    - Deploy Database: Manual deployment of database migrations
    - Integration Validation: Integration validation with E2E tests

Deployment Architecture

The deployment architecture follows a serverless model:

- **Development:** Code is developed locally and hosted on GitHub.
- **CI/CD:** GitHub Actions validates code and Cloud Build builds the image.
- **Storage:** Image is saved in Artifact Registry.
- **Execution:** Cloud Run executes the image with automatic scaling.
- **Database**: Application connects to Neon PostgreSQL via SSL.

This architecture offers:

- **High availability:** Automatic scaling and redundancy.
- **Security:** Centralized secret management and SSL connections.
- **Cost-efficiency:** Billing only for real use.
- **Maintainability**: Automated deployment without manual intervention

Testing the Server

The recommended way to test the server is using Postman.

1. Copy the Postman collection `book-publishing-api-collection.json`[34] (folder: `./postman` in backend).
2. Point to the base URL
3. To authenticate as administrator use the admin credentials
4. To authenticate as base user use the base user credentials
5. The token has a duration of 15 minutes.

Example of successful authentication cURL:

```
postman request POST '{{my_base_url}}/api/v1/auth/login' \
  --header 'Content-Type: application/json' \
  --body '{
    "username": "my_username",
    "password": "my_password"
}'
```

[34] Backend Postman Collection: https://github.com/CescFe/book-publishing-backend/blob/main/postman/book-publishing-api-collection.json

EVALUATION OF OBJECTIVES AND RESULTS

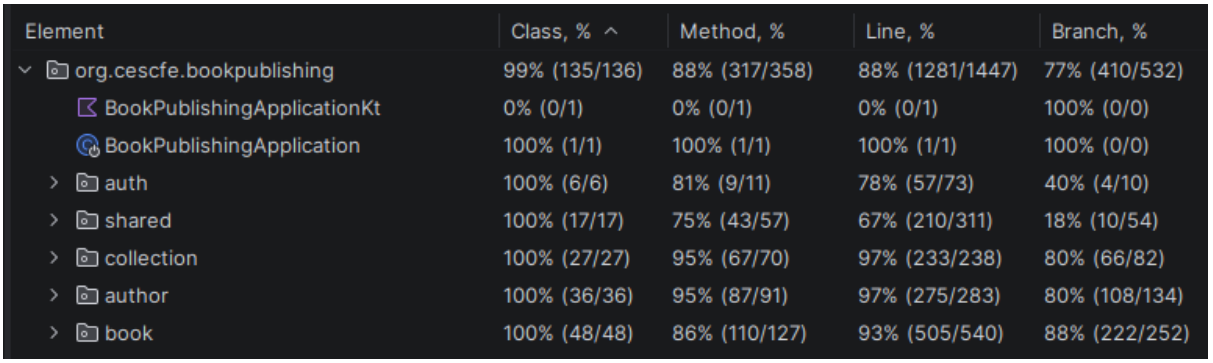
This section aims to critically analyze the level of achievement of the Objectives and Key Results (OKRs) defined at the beginning of the project. The evaluation is based on objective evidence (technical metrics, project artifacts, and observable results), as well as qualitative assessments where appropriate.

OKR 0: Client Value and Usability (86.67%)

Objective: Develop production-ready, clean, and maintainable code.
Conclusion: L'eina compleix l'objectiu de ser usable i funcional sense dependència tècnica externa. El disseny de la interfície prioritza la simplicitat i redueix la càrrega cognitiva de l'usuari. Es va retallar l'alcanç del CRUD de col·leccions i a l'app mòbil no es pot crear, esborrar o actualitzar una col·lecció, només es poden visualitzar.

OKR 1: Technical Quality and Robustness (88.5%)

Objective: Develop production-ready, clean, and maintainable code.
Conclusion: A high level of quality has been ensured through automated testing and continuous integration, reducing the risk of regressions and facilitating code evolution. Gradle reports a backend test coverage of 88% of code lines and 77% of branches. It was not possible to use Gradle to evaluate test coverage for the mobile application, as it cannot measure the coverage of instrumentation tests.



Element	Class, %	Method, %	Line, %	Branch, %
org.cescfe.bookpublishing	99% (135/136)	88% (317/358)	88% (1281/1447)	77% (410/532)
BookPublishingApplicationKt	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)
BookPublishingApplication	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
auth	100% (6/6)	81% (9/11)	78% (57/73)	40% (4/10)
shared	100% (17/17)	75% (43/57)	67% (210/311)	18% (10/54)
collection	100% (27/27)	95% (67/70)	97% (233/238)	80% (66/82)
author	100% (36/36)	95% (87/91)	97% (275/283)	80% (108/134)
book	100% (48/48)	86% (110/127)	93% (505/540)	88% (222/252)

OKR 2: Efficient Planning and Execution (100%)

Objective: Manage the project using agile methodologies despite limited resources.
Conclusion: Discipline in backlog management and prior refinement enabled an orderly and predictable execution of the project.

Key Result	KPI	Result	Evaluation
OKR 0: Client Value and Usability (86.67%)			
KR1: The client can perform all CRUD operations	Qualitative customer feedback	The client can manage Authors, Collections, and Books autonomously	⚠️ Partially (73.33%)
KR2: Optimized UI/UX	Number of screens per key action ≤ 3	Main actions are completed in ≤ 3 screens	✅ Achieved (100%)
OKR 1: Technical Quality and Robustness (88.5%)			
KR1: ≥ 80% test coverage	% coverage reported by Gradle	Coverage above 80% in critical repositories	⚠️ Partially (77%)
KR2: CI automation	Number of active GitHub Actions	≥ 2 active and green workflows per repository	✅ Achieved (100%)
OKR 2: Efficient Planning and Execution (100%)			
KR1: Fully refined tasks	% of issues with defined AC	100% of issues refined before moving to In Progress	✅ Achieved (100%)
KR2: Milestone completion	% of completed tasks	100% of tasks completed within the milestone	✅ Achieved (100%)
OKR 3: Value as a Technical Portfolio (100%)			
KR1: Professional README	README checklist	All repositories meet the criteria	✅ Achieved (100%)
KR2: Active production environment	Health check	Endpoint returns 200 OK	✅ Achieved (100%)

OKR 4: Maintainability and Future Readiness (100%)			
KR1: API-first approach	Controllers generated from api-spec	Contract defined prior to implementation	✅ Achieved (100%)
KR2: Pure domain layer	Framework imports in domain	0 external framework imports in the domain module	✅ Achieved (100%)

OKR 3: Value as a Technical Portfolio (100%)

Objective: Turn the project into a tangible demonstration of technical skills.
Conclusion: The project is suitable for presentation as a professional portfolio, both in terms of documentation quality and real operational readiness in production.

OKR 4: Maintainability and Future Readiness (100%)

Objective: Ensure long-term sustainability and evolution of the project.
Conclusion: The adopted architecture (API-first and pure domain) ensures high maintainability, facilitates the integration of new channels (web frontend, new clients), and reduces the cost of future evolution.

Overall Compliance Rate

After reviewing the achievement percentages shown in the table in this chapter, it can be concluded that 95.03% of the Key Results have been fully achieved (or exceeded) or partially achieved with a high degree of compliance.

The two Key Results that did not reach 100% correspond to qualitative objectives or technical threshold targets (customer-perceived usability and test coverage). Although they did not fully reach the initially defined target values, they fully meet the expected functional requirements and do not compromise the quality or viability of the project.

Overall, the results obtained confirm the success of the defined strategies, the robustness of the technical solution, and the validity of the project both as a functional product and as an exercise in sustainable architectural design.

Global Conclusion

The evaluation of the OKRs and KPIs demonstrates that the project has achieved the objectives initially defined, both from a functional and a technical perspective. The architectural decisions, planning discipline, and quality-focused approach made it possible to build a robust, extensible solution aligned with professional best practices.

The applied methodology has proven to be an effective tool for:

- **Maintaining a focus on value**: OKRs linked to customer experience ensured that technical decisions always served an end-user purpose.
- **Quantifying technical progress**: Objective metrics such as test coverage and automation enabled the evaluation of real improvements rather than perceptions.
- **Balancing ambition with reality**: Partial achievement of OKR 0 and OKR 1 reflects a healthy adaptation to real-world constraints, prioritizing quality over absolute completeness.
- **Building for the future**: Success in OKRs 3 and 4 ensures that the project not only works today but is also structured to evolve and serve as a technical reference.

CONCLUSIONS

This Book Publishing Management Back Office System project has been a valuable opportunity to put into practice the knowledge acquired during my academic studies as well as through my professional experience over the past three and a half years. The development of the project allowed me to work with full autonomy within an agile planning framework applied to a realistic use case, addressing the solution from multiple perspectives: business vision, product design, user experience, and technical architecture.

Throughout the project, I was able to integrate and consolidate concepts that I had previously understood only partially or in isolation. Designing and implementing the solution from scratch, and progressively addressing the technical and decision-making challenges that arose, helped me gain a deeper understanding of the full lifecycle of a software development project, from initial definition to deployment and final evaluation.

As detailed in the OKR and KPI evaluation chapter, the project achieved a high level of compliance with the initially defined objectives, with an overall compliance rate of 95.03%. Although some Key Results were only partially achieved these did not compromise the core functionality or the technical robustness of the solution. Instead, they provided clear indicators for potential areas of future improvement.

Beyond strictly technical knowledge, the project also allowed me to reflect on priority management, decision-making under time and resource constraints, and on which aspects truly deliver value to a client when delivering a functional software solution.

This project does not conclude with the submission of this report. As outlined in previous sections, the architectural foundation and the decisions made enable the continued evolution of the solution, the completion of pending backlog items, and the progressive and sustainable incorporation of new functionalities.

In conclusion, this work has been a highly enriching experience, not only due to its scope and complexity, but also because of the self-imposed quality standards, the knowledge gained, the confidence built, and the consolidation of my growth both as a student and as a software development professional.

BIBLIOGRAPHY

- *Software Design Principles*. Bastos Pérez-Cuadrado, Carlos. https://leanpub.com/softwaredesignprinciples
- *OpenAPI: Beginner to Guru*. Thompson, John. udemy.com/course/openapi-beginner-to-guru/
- *API First Engineering with Spring Boot*. Thompson, John. https://www.udemy.com/course/api-first-engineering-with-spring-boot
- Develop for Android. https://developer.android.com/develop
- OpenAPI Generator. *Workflow Integrations*.https://openapi-generator.tech/docs/integrations/
- Liquibase. *Getting Started with Liquibase and Gradle*. https://contribute.liquibase.com/extensions-integrations/directory/integration-docs/gradle/
- Gradle. *Enabling and Configuring the Configuration Cache*. https://docs.gradle.org/9.2.1/userguide/configuration_cache_enabling.html
- Swagger. *What Is OpenAPI?* https://swagger.io/docs/specification/v3_0/about/
- Medium. *Configuring Liquibase in a Spring Boot project with Gradle*. Heaver, Edward. https://edward-heaver.medium.com/configuring-liquibase-in-a-spring-boot-project-with-gradle-909b6b370970
- Spring. *OAuth 2.0 Resource Server JWT*. https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html
- Geeks for Geeks. Spring Boot 3.0 - JWT Authentication with Spring Security using MySQL Database. Rout, Amiya. https://www.geeksforgeeks.org/springboot/spring-boot-3-0-jwt-authentication-with-spring-security-using-mysql-database/
- Medium. *Implement JWT authentication in a Spring Boot 3 application*. Cabrel Tiogo, Eric. https://medium.com/@tericcabrel/implement-jwt-authentication-in-a-spring-boot-3-application-5839e4fd8fac
- Baelding. *Creating a Spring Security Key for Signing a JWT Token*. Michael, Olayemi. https://www.baeldung.com/spring-security-sign-jwt-token
- Medium. *EncryptedSharedPreferences is Deprecated - What Should Android Developers Use Now?* https://medium.com/@n20/encryptedsharedpreferences-is-deprecated-what-should-android-developers-use-now-7476140e8347
- Free Code Camp. Open-Closed Principle - SOLID Architecture Concept Explained. Chris, Kolade. https://www.freecodecamp.org/news/open-closed-principle-solid-architecture-concept-explained/
- GitHub Docs. *Managing caches*. https://docs.github.com/en/actions/how-tos/manage-workflow-runs/manage-caches
- Github Community. *How to Create Cleanup Cache Action.* https://github.com/orgs/community/discussions/158792
- Learning Legendario. *Análisis DAFO y creación de estrategias*. https://learninglegendario.com/analisis-dafo-creacion-estrategias-came-dafo-cruzado/