Surugiu Ioana-Monica
Teodorescu Ana-Maria 1241EB

Milestone 3 - SDT Project
Digital Library Management System

1) Introduction

Digital Library Management System is a useful digital platform designed to help users search for books, borrow and return items as well as access digital materials such as e-books and research papers. The platform integrates multiple functionalities, such as: catalog browsing, borrowing and reservation workflows, user notifications, recommendation algorithms and access to digital resources.

This milestone investigates and evaluates three distinct architectural styles that could be used to implement the system:

1. **Monolithic Architecture**
2. **Microservices Architecture**
3. **Event-Driven Architecture**

For each architectural style, we analyze how the system would be structured, how its components interact and how data flows between modules. We also provide component and deployment diagrams to illustrate each approach.

In the end, we will compare the three alternatives and determine which architectural style is best suited for our project, providing a clear and well-supported justification for the chosen design direction.

2) Architecture 1 – Monolithic

2.1) How the Digital Library Management System is structured (components & data flow)

In the monolithic style, all backend logic is one deployable application (a single Java/Spring Boot app):

Main internal modules (logical components inside the monolith):

- **Presentation Layer**
  - REST controllers / HTTP endpoints for:
    - Authentication & registration
    - Searching the catalog
    - Borrow/return operations
    - Managing digital resources
    - Admin management pages

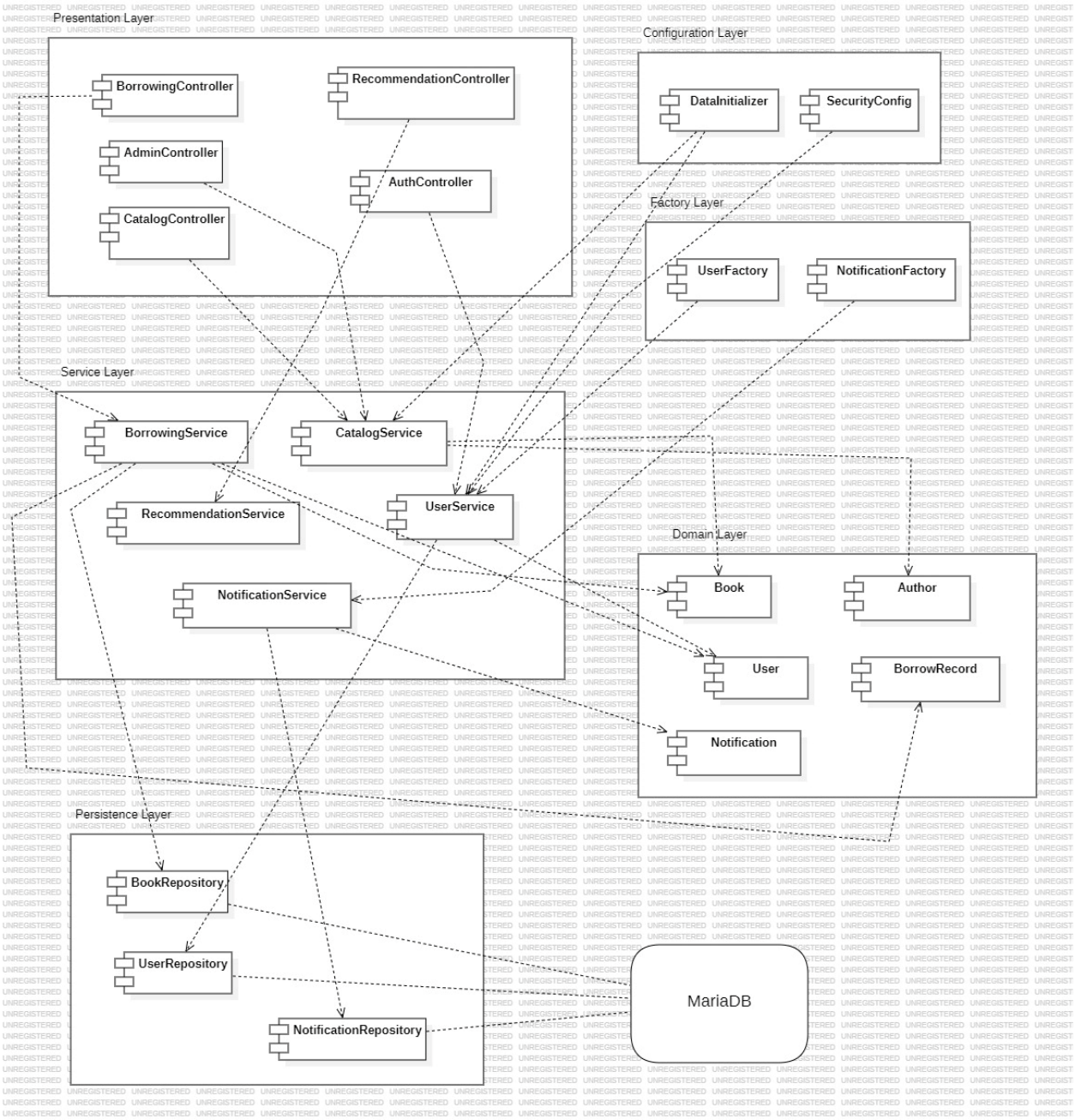
- **Application / Service Layer**
  - *UserService* - register users, login, manage profiles and roles (User / Admin).
  - *CatalogService* - manage books, authors, categories, search, filter.
  - *BorrowingService* - borrow, return, due dates, overdue detection.
  - *NotificationService* - sends notifications (email / in-app) to users (Observer pattern fits here).
  - *RecommendationService* - uses different algorithms (Strategy) to suggest books.
  - *DigitalResourceService* - managing access to e-books, PDFs, research papers.
- **Domain Layer**
  - Entities: *User, Admin, Book, BorrowRecord, Notification, Recommendation*
  - Patterns from milestone 2 are used here: *Factory (users/notifications), Builder (Book), Strategy, Observer, Singleton*.
- **Persistence Layer**
  - Repositories / DAOs for all entities.
  - Single relational database schema (e.g. dlms_db) with tables:
    - users, books, borrow_records, notifications, recommendations, etc.

Data flow example - "borrow a book" use case:

1. User sends HTTP request POST /borrow/{bookId} from UI.
2. Controller calls BorrowingService.borrowBook(userId, bookId).
3. BorrowingService checks availability via CatalogService, creates BorrowRecord, saves via repository.
4. BorrowingService notifies NotificationService which creates a "borrow confirmation" notification.
5. Data is stored in the same DB and the response is returned to the UI.

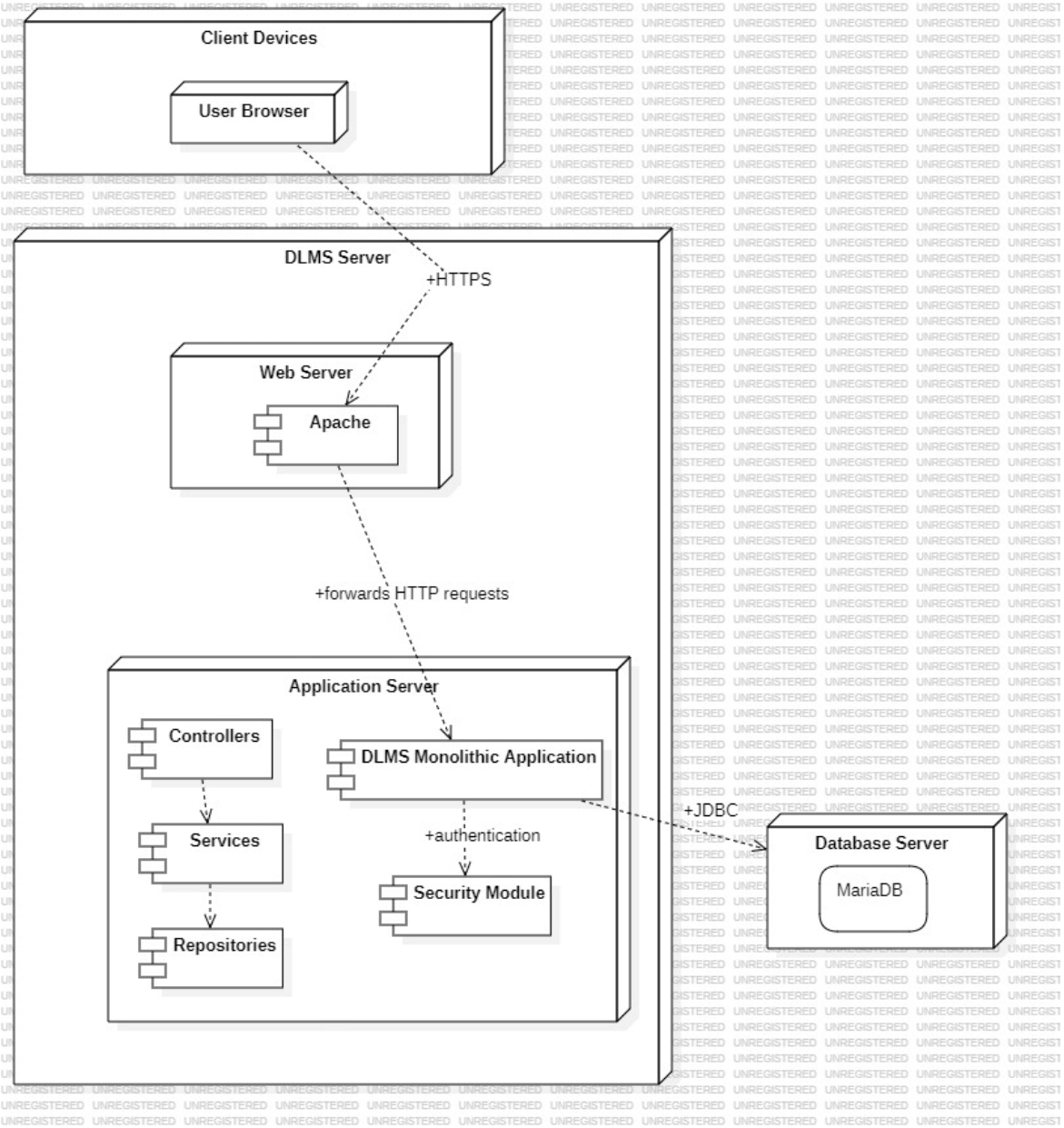
Surugiu Ioana-Monica
Teodorescu Ana-Maria 1241EB

2.2) Monolithic component diagram



Surugiu Ioana-Monica
Teodorescu Ana-Maria 1241EB

2.3) Monolithic deployment diagram



2.4) Pros & Cons of monolithic Digital Library Management System

Advantages:

- Simple to develop at the beginning - one codebase, one build, one deployment.
- Easier to reuse the existing implementation from Milestone 2 (we already have a monolithic POC).
- Easier debugging: you can step through the whole request in one IDE.
- Central database ensures strong data consistency

Disadvantages:

- Limited scalability: you can't scale catalog or notification logic separately - you scale the whole app.
- Tight coupling between modules; the project becomes harder to maintain as features grow (recommendations, analytics, reports).
- Technology lock-in: everything must be Java (for example); cannot use other tech for recommendation engine.
- Deployment is risky: a small change in notifications requires re-deploying the whole system.

3) Architecture 2 – Microservices

2.1) Main services and responsibilities

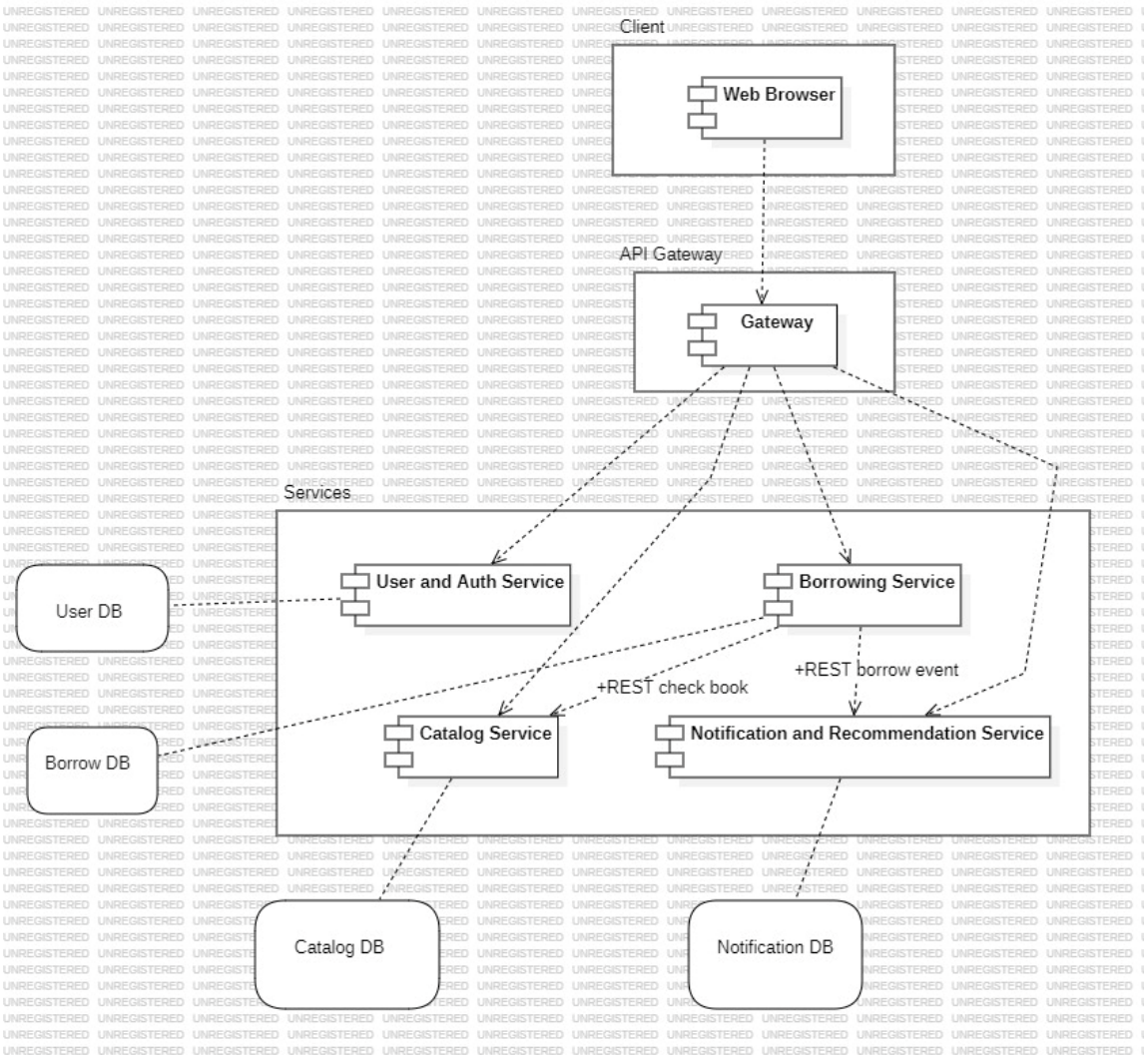
We split DLMS into **independent services**, each with its own domain & data.

1. **API Gateway / Edge Service**
    - Single entry point for clients.
    - Routes requests to internal services via HTTP.
    - Handles auth / token validation.
2. **User & Auth Service**
    - Manages users, roles (User, Admin).
    - Registration, login, JWT generation.
    - Owns users table.
3. **Catalog Service**
    - Manages books, categories, authors.
    - Search and filter.
    - Owns books, authors, categories.
4. **Borrowing Service**
    - Manages borrow/return operations, due dates, overdue status.
    - Owns borrow_records.
5. **Notification & Recommendation Service**
    - Sends notifications (email / in-app) about:
        - borrowed books
        - due dates
    - Generates recommendations based on user history and preferences.
    - Owns notifications, recommendation_profiles.

Data Flow example – borrow a book (microservices version):

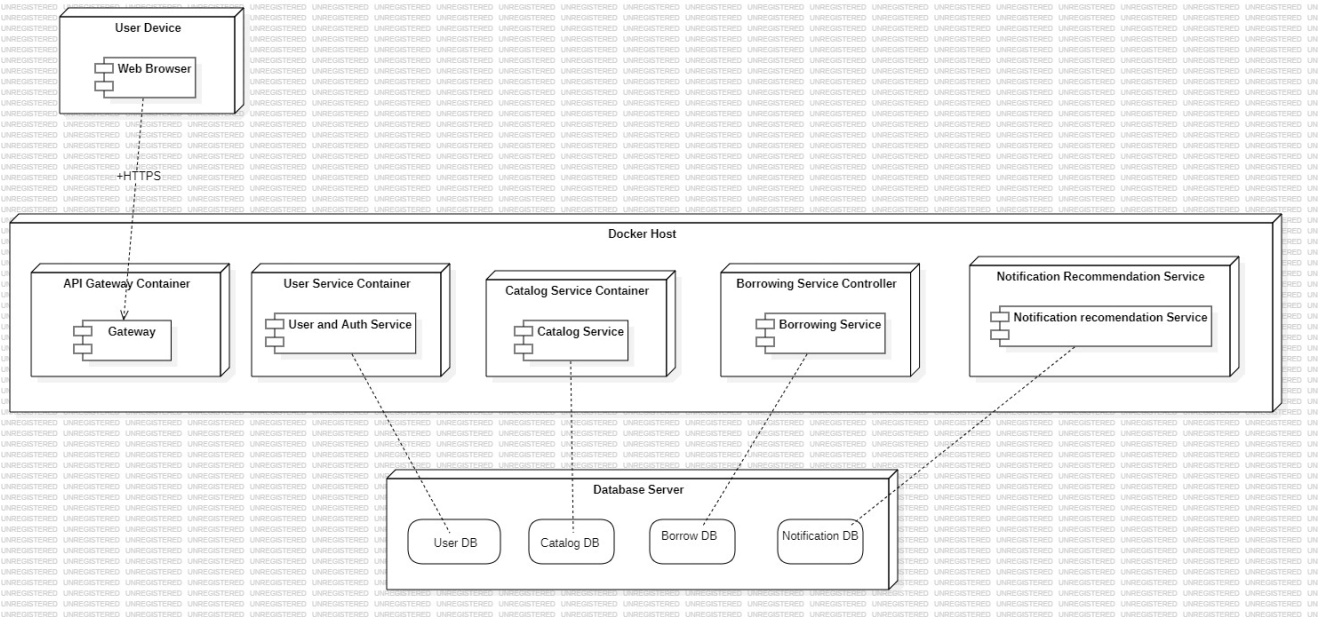
1. Browser calls POST /borrow/{bookId} → **API Gateway**.
2. Gateway forwards to **Borrowing Service**.
3. Borrowing Service calls **Catalog Service** (REST) to check availability.
4. If available:
    - Borrowing Service creates BorrowRecord in its own DB.
    - Borrowing Service calls **Notification Service** to send confirmation (synchronous REST) – in Milestone 5 this will become an async message via queue.
5. Response goes back to the client through the gateway.

2.2) Microservices component diagram



Surugiu Ioana-Monica
Teodorescu Ana-Maria 1241EB

2.3) Microservices deployment diagram



2.4) Pros & Cons of microservices Digital Library Management System

Advantages:

- Scalability: heavy read traffic (book search) can be scaled separately from Borrowing or Notifications.
- Independent development & deployment: you can update Recommendation Service without touching User or Catalog.
- Technology flexibility: can keep Java/Spring for main services and, in future, maybe use another tech for recommendation algorithms.
- It is good for Multiple services, REST communication for Postman tests, Docker containers

Disadvantages:

- Increased complexity:
    - You have to handle service discovery, API gateways, configuration, logging across services.
- Distributed data management:
    - No simple ACID across services; need patterns like sagas, eventual consistency.
- More operational overhead:
    - More containers, more monitoring, more failure modes.

4) Architecture 3 - Event-Driven distributed architecture

4.1) Components & responsibilities

We reuse the same domains, but change how they interact:

- **API Gateway / Backend-for-Frontend**
  - Exposes REST endpoints to the UI (login, search, borrow, etc.).
  - For commands (borrow book, return book), it calls the respective service.
  - For queries (catalog search), it might call Catalog directly.
- **User & Auth Service**
  - Registers users, manages roles, authentication.
  - Publishes events:
    - UserRegistered
    - UserUpdated
- **Catalog Service**
  - Manages books, authors, categories, availability.
  - Publishes events:
    - BookCreated
    - BookUpdated
    - BookMarkedUnavailable / BookMarkedAvailable
  - Consumes events: could consume UserRegistered if we ever want personalized catalog.
- **Borrowing Service**
  - Handles borrowing and returning books.
  - Publishes events:
    - BookBorrowed (contains userId, bookId, dueDate)
    - BookReturned
    - BookOverdue
  - Consumes events:
    - BookMarkedUnavailable (to prevent conflicts, if needed)
- **Notification Service**
  - Focuses ONLY on sending emails/in-app messages.
  - Consumes events:
    - UserRegistered → send welcome email.
    - BookBorrowed → send borrow confirmation.
    - BookOverdue → send overdue reminder.

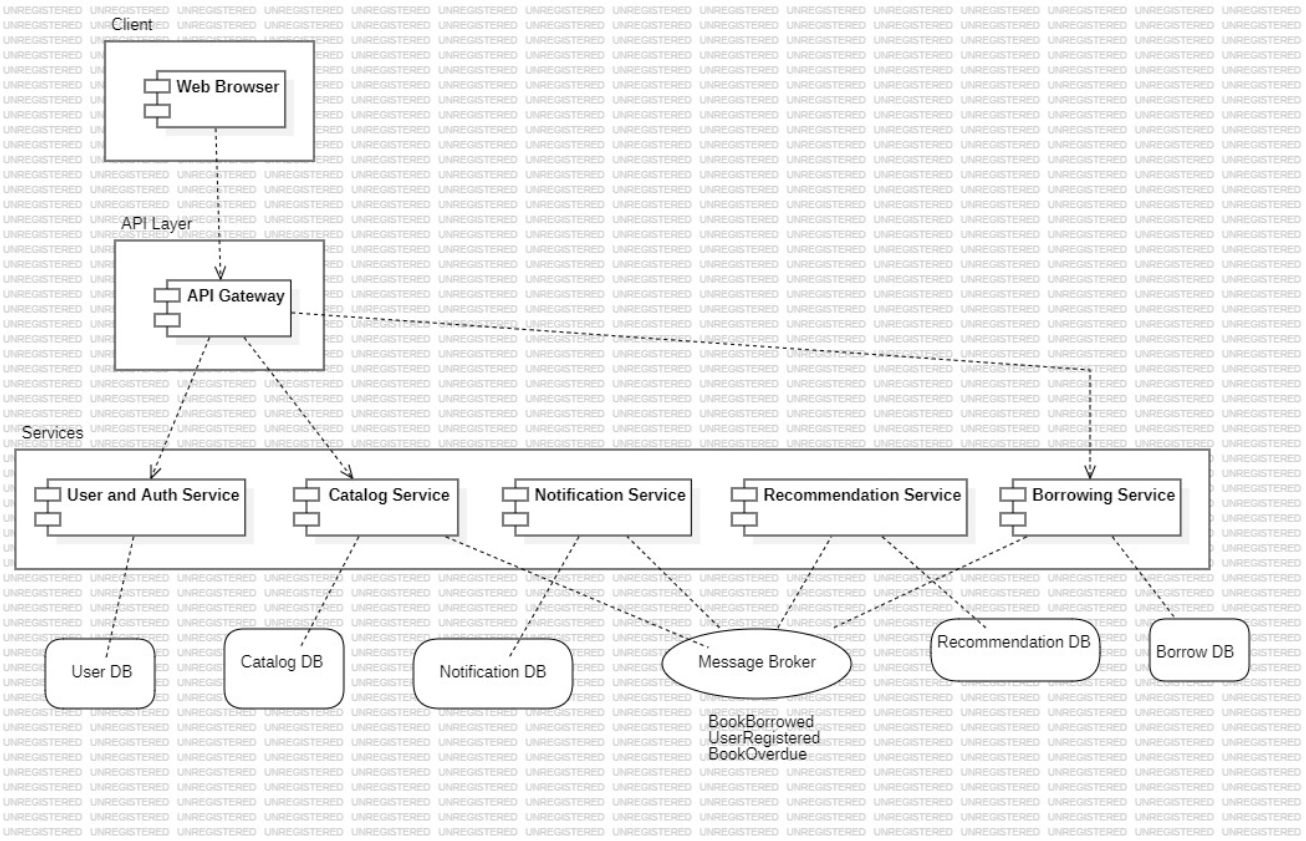
- **Recommendation / Analytics Service**
  - Builds recommendation models or simple statistics (most borrowed books, user history).
  - Consumes events:
    - BookBorrowed, BookReturned, maybe UserRegistered.
  - Updates its own data store (recommendation profiles or analytics DB).
- **Message Broker**
  - Central infrastructure component (RabbitMQ/Kafka).
  - Has topics/queues per event type (book-borrowed, user-registered).
  - Producers (Borrowing Service, User Service, Catalog Service) publish events.
  - Consumers (Notification, Recommendation) subscribe.

Data flow example - "borrow a book"

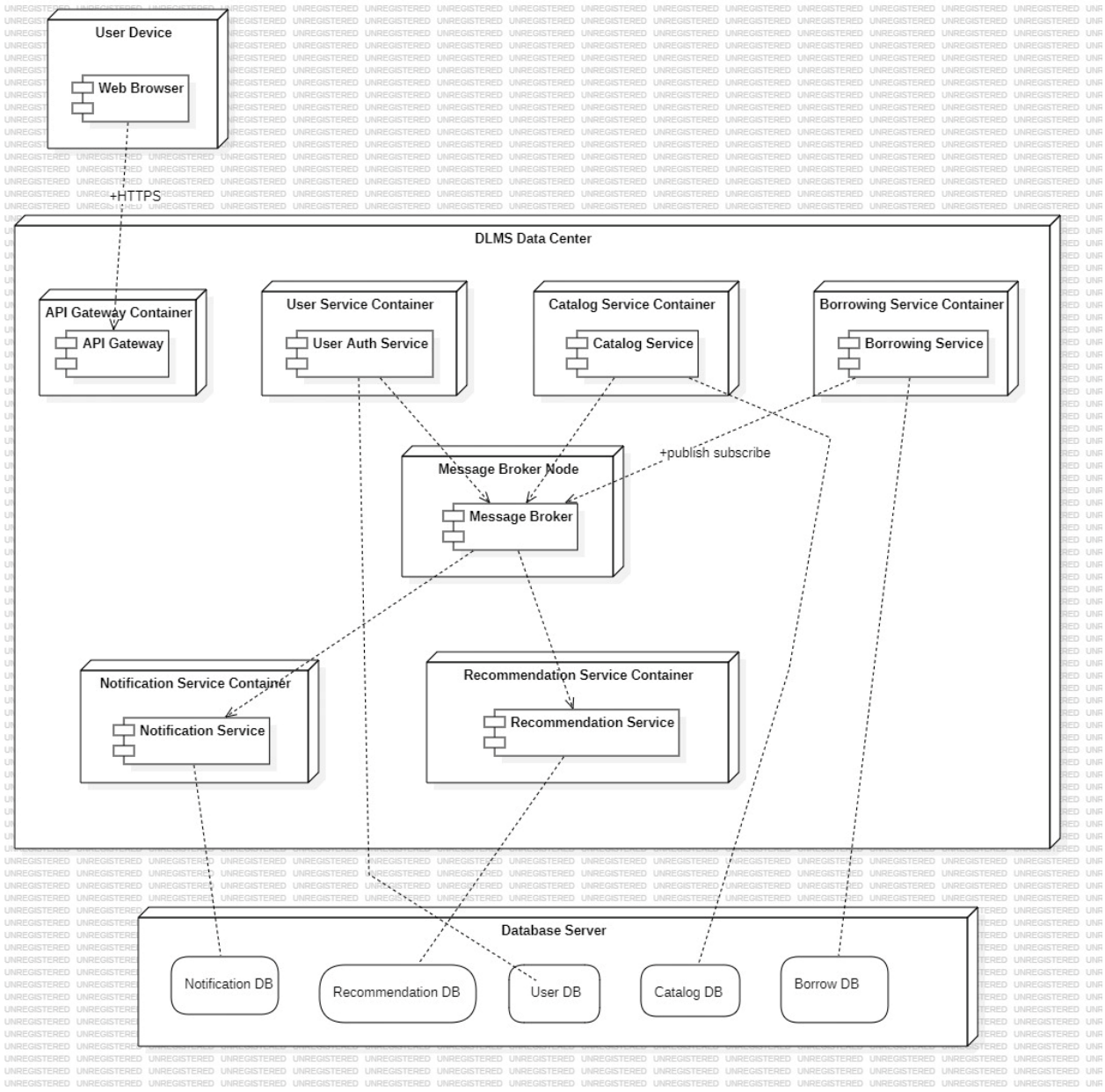
1. User clicks "Borrow" in the UI => **API Gateway** calls Borrowing Service via REST: *POST /borrowings?userId=...&bookId=...*
2. **Borrowing Service**:
   - Checks availability by calling **Catalog Service** (REST or cached copy).
   - Creates a BorrowRecord in its DB.
   - Publishes **BookBorrowed event** to the Message Broker.
3. **Notification Service** receives BookBorrowed:
   - Sends a confirmation email or in-app notification.
4. **Recommendation Service** receives BookBorrowed:
   - Updates user reading history and recommendation model.
5. UI gets immediate **HTTP response** from Borrowing Service (via Gateway). Notifications and recommendations happen **asynchronously**, which improves responsiveness.

Surugiu Ioana-Monica
Teodorescu Ana-Maria 1241EB

4.2) Event-Driven component diagram



4.3) Event-Driven deployment diagram



4.4) Pros & Cons of Event-Driven Digital Library Management System

Advantages:

- Loose coupling: producers don't need to know who consumes the events; you can add new consumers ("Reporting Service") without changing existing services.
- Good performance & user experience:
  - User gets quick HTTP response.
  - Heavy work (emails, analytics) happens in background.

- Scalable: You can independently scale consumers that handle popular events (BookBorrowed).
- Great for features like notifications, analytics, recommendation - exactly what a digital library benefits from.

Disadvantages:

- Complexity of design:
  - You must design event schemas carefully and ensure consumers handle failures.
- Eventual consistency:
  - Notification or recommendation data may be slightly behind the “source of truth”.
- Harder debugging:
  - A single user action may produce a chain of events across multiple services.

5) Comparison of the three Architectures

5.1) Development complexity

From a development perspective, the **monolithic architecture** is the simplest to implement for the Digital Library Management System because all functionalities (authentication, catalog browsing, borrowing workflows, notifications, and recommendations) are contained within a single Spring Boot application. This eliminates the need for distributed communication, multiple deployment environments, or complex configuration. However, this simplicity comes at the cost of flexibility.

In contrast, the **microservices architecture** introduces a moderate increase in complexity, as the system is divided into domain-specific services such as User & Auth, Catalog, Borrowing, and Notification/Recommendation. Each service requires its own database and exposes REST endpoints, which also means developers must manage inter-service communication, Docker containers, and basic networking.

The **event-driven architecture** is the most complex, because it builds on top of microservices while adding asynchronous communication through a message broker. Designing event schemas, ensuring idempotency, handling duplicate messages, and coordinating eventual consistency significantly raises the implementation effort, making full adoption challenging within the timeframe of an academic project.

5.2) Scalability and maintainability

A **monolithic application** offers very limited scalability for DLMS because all features run within the same process, meaning high-demand operations like catalog search or recommendation retrieval cannot be scaled independently. As the project grows, adding new modules tends to increase code coupling, making long-term maintenance more difficult.

Microservices address these issues by isolating core functional areas into independently deployable units. For example, the Catalog Service can scale horizontally to accommodate increased search traffic, while the Borrowing Service can scale separately during peak borrowing periods. This separation also enhances maintainability, as each service evolves within its own codebase.

Event-driven architecture provides the highest scalability and extensibility for DLMS: services communicate through events rather than direct calls, making it easy to integrate new components like analytics dashboards, recommendation engines, or reporting modules without modifying existing services. This model is ideal for background-heavy features such as overdue reminders or reading-history-based recommendations, which can be processed asynchronously and scaled independently.

5.3) Alignment with course & next milestones

Although the **monolithic architecture** is straightforward and aligns well with early project stages, it does not adequately support the distributed systems concepts required in later milestones. As the course progresses into containerization, service decomposition, and inter-service communication, the monolithic model becomes increasingly restrictive.

The microservices architecture aligns perfectly with Milestone 4, which requires multiple Spring Boot services, database-per-service patterns, RESTful interactions tested through Postman, and Dockerized deployment. It also provides a clean foundation for demonstrating domain separation and proper architectural layering.

Finally, the **event-driven** style is highly relevant to Milestone 5, which introduces asynchronous messaging and message brokers. Borrowing, Notification, and Recommendation services in DLMS naturally benefit from an event-driven approach, making it an excellent evolutionary step after the microservices foundation is in place.

Surugiu Ioana-Monica
Teodorescu Ana-Maria 1241EB

6) Conclusion + Which architecture is most suitable?

After evaluating the monolithic, microservices, and event-driven architectures, the **Microservices Architecture emerges as the most suitable choice** for the Digital Library Management System at this stage of the project.

While the monolithic approach provides simplicity, strong consistency, and rapid initial development, it becomes restrictive as the system grows in functionality - particularly in areas such as personalized recommendations, user notifications, and complex catalog interactions. In contrast, the microservices architecture offers a balanced combination of modularity, scalability, and maintainability, enabling the core functionalities of the DLMS - User & Auth, Catalog, Borrowing, and Notification/Recommendation - to be cleanly separated into independently deployable services. This separation not only aligns with the system's natural domain boundaries but also allows each service to scale based on its own traffic patterns, improving performance and resource efficiency.

From an educational and project-planning standpoint, **microservices** also **align directly with the upcoming course milestones**, which require containerized deployment using Docker, REST-based communication, distributed design principles, and testing through Postman.

At the same time, choosing microservices now creates a smooth evolutionary path toward an event-driven architecture later, should the system require asynchronous processing for notifications, analytics, or recommendation updates.

Therefore, the microservices architecture provides the **best balance between technical suitability, course alignment, and long-term extensibility**, making it the most appropriate architectural foundation for the next stages of the DLMS project.