ISEL

Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

SOFTWARE LABORATORY

GROUP 13

Mert CINAR

Yusuf Kenan GOKSU

29.03.2025

INTRODUCTION

The LS project is an information system designed to manage padel courts efficiently. The development is divided into four phases, each with incremental requirements. This document focuses on the first phase, which primarily involves backend service development and API implementation.

The primary goal of this phase was to establish a solid foundation for the system by defining the database schema, implementing the API, and ensuring data consistency and security. The backend was developed using Kotlin and leverages the HTTP4K library to handle HTTP requests. Data is stored in a PostgreSQL database, ensuring persistence and scalability. In the initial development, an in-memory data storage approach was used for testing and rapid prototyping.

The key functionalities implemented in this phase include:

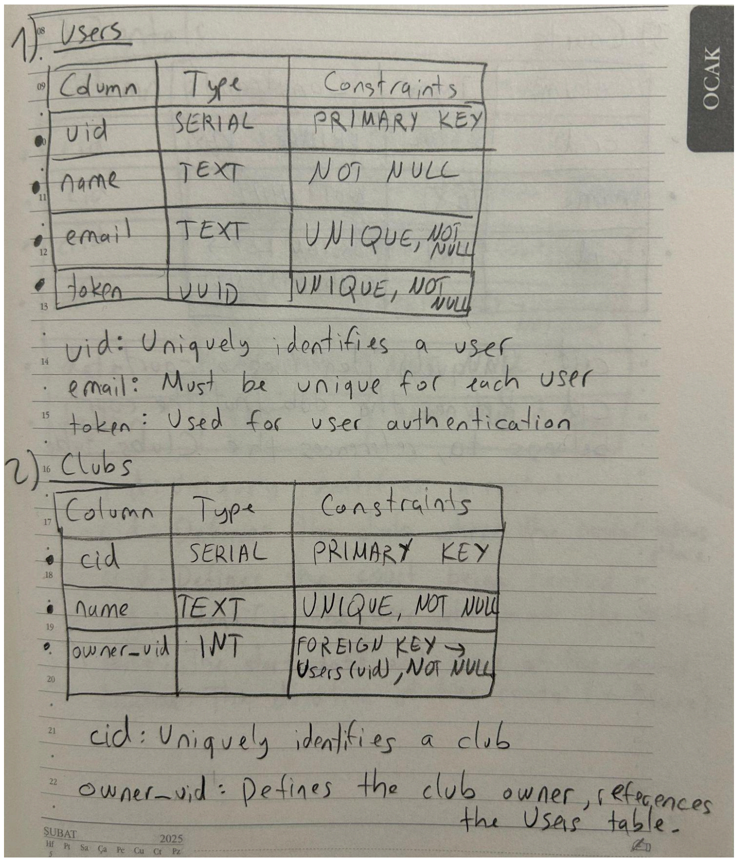
- User Management: Creating users, retrieving user details.
- Club Management: Creating clubs, retrieving club details, listing clubs.
- Court Management: Creating courts, retrieving court details, listing courts per club.
- Rental Management: Creating rentals, retrieving rental details, listing rentals by court, club, or user.
- Authentication & Authorization: Implemented via Bearer Tokens to ensure secure access.
- Pagination: Applied to all listing operations for improved performance.
- Unit Testing: API endpoints and core functionalities are tested for correctness.

This report provides an overview of the conceptual and physical database models, API specifications, request flow, connection management, data access, error handling, and an evaluation of the first phase's outcomes.

Modeling the Database

Conceptual Model

The database structure is designed to support efficient management of padel courts, ensuring data integrity and smooth interactions between different entities.



1). Users

Column	Type	Constraints
uid	SERIAL	PRIMARY KEY
name	TEXT	NOT NULL
email	TEXT	UNIQUE, NOT NULL
token	UUID	UNIQUE, NOT NULL

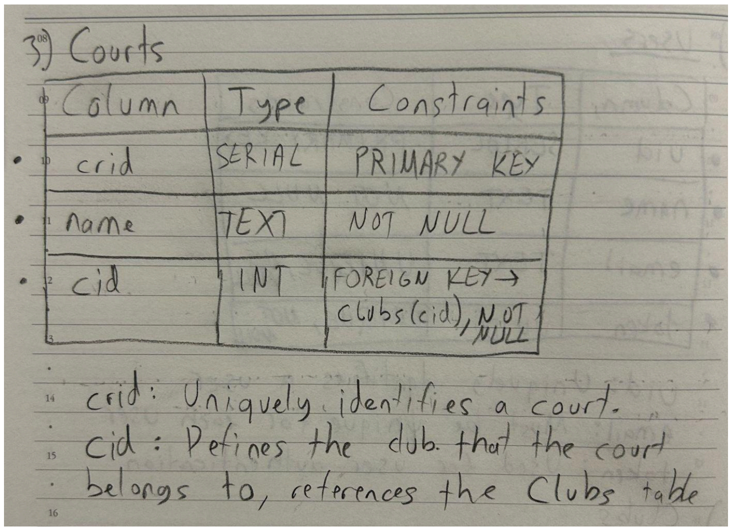
uid: Uniquely identifies a user
email: Must be unique for each user
token: Used for user authentication

2) Clubs

Column	Type	Constraints
cid	SERIAL	PRIMARY KEY
name	TEXT	UNIQUE, NOT NULL
owner_uid	INT	FOREIGN KEY → Users (uid), NOT NULL

cid: Uniquely identifies a club
owner_uid: Defines the club owner, references the Users' table.

Figure 1. Users and Clubs database table

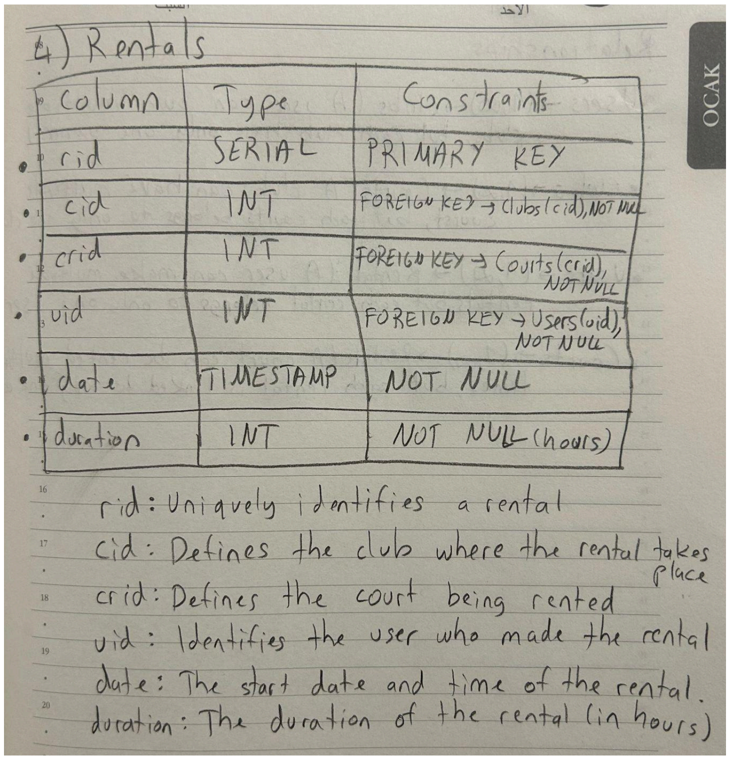


3) Courts

Column	Type	Constraints
crid	SERIAL	PRIMARY KEY
name	TEXT	NOT NULL
cid	INT	FOREIGN KEY → Clubs(cid), NOT NULL

crid: Uniquely identifies a court.
cid: Defines the club that the court belongs to, references the Clubs table

Figure 2. Courts database table



4) Rentals

Column	Type	Constraints
rid	SERIAL	PRIMARY KEY
cid	INT	FOREIGN KEY → Clubs(cid), NOT NULL
crid	INT	FOREIGN KEY → Courts(crid), NOT NULL
uid	INT	FOREIGN KEY → Users(uid), NOT NULL
date	TIMESTAMP	NOT NULL
duration	INT	NOT NULL (hours)

rid: Uniquely identifies a rental
cid: Defines the club where the rental takes place
crid: Defines the court being rented
uid: Identifies the user who made the rental
date: The start date and time of the rental.
duration: The duration of the rental (in hours)

Figure 3. Rentals database table

Database Relationships:

- **Users** → (1,∞**)** → **Clubs :** A user can own multiple clubs, but each club has only one owner.
- **Clubs** → (1,∞**)** → **Courts :** A club can have multiple courts, but each court belongs to only one club.
- **Users** → (1,∞**)** → **Rentals :** A user can make multiple rentals, but each rental belongs to only one user.
- **Courts** → (1,∞**)** → **Rentals :** A court can be rented multiple times, but each rental is linked to only one court.

Design Aspects:

- The database follows a relational model with well-defined entity relationships.
- Each user, club, court, and rental has a unique identifier.
- Users own clubs, and clubs have multiple courts.
- Rentals are associated with users and courts, with defined booking times and durations.
- Indexing is used for efficient query execution.

Restrictions:

- Unique constraints ensure no duplicate entries for users, clubs, or courts.
- Foreign key constraints maintain referential integrity between entities.
- Courts are available for rental 24/7, ensuring unrestricted booking times.

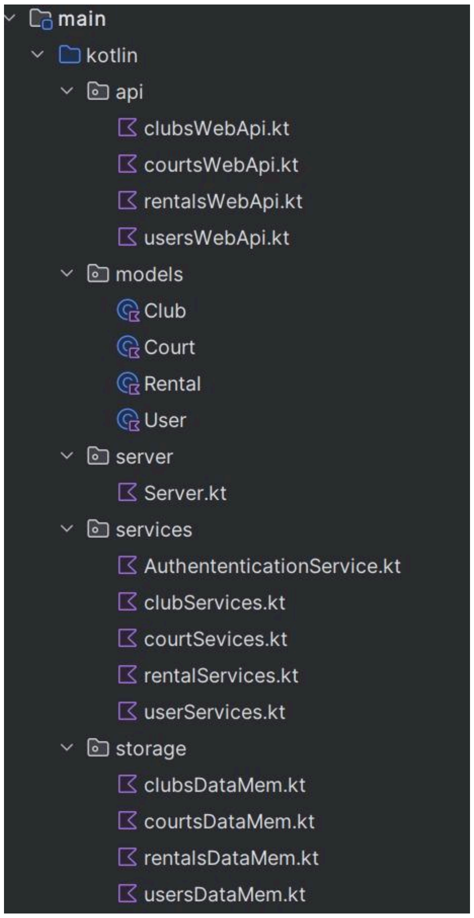
Physical Model

Design Aspects:

- PostgreSQL is used for data persistence.
- Tables are designed for efficient querying and transactional consistency.
- Proper indexing and data normalization techniques are applied.
- Test data is stored in-memory for unit testing purposes.

Software Organization

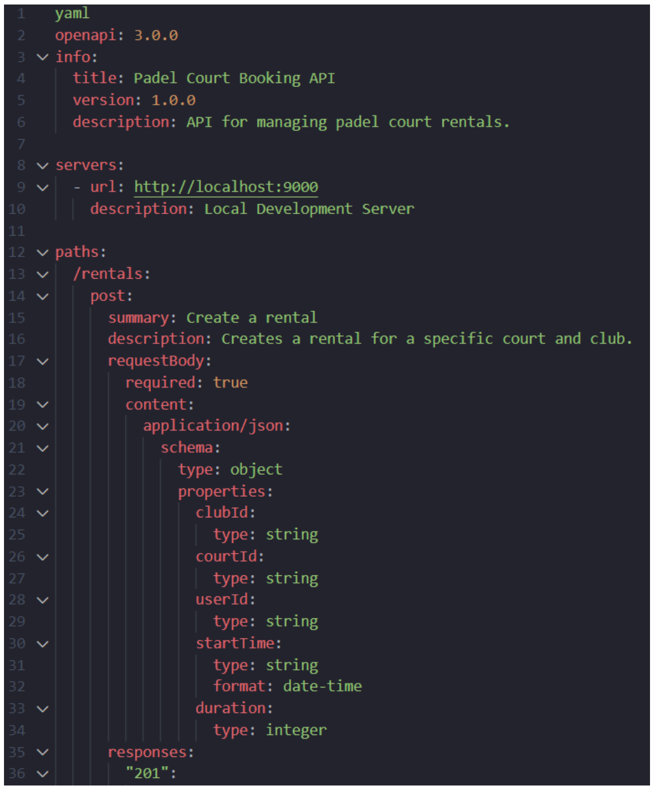
The software is structured to ensure maintainability and modularity. Different components handle various functionalities efficiently.



```
main
  kotlin
    api
      clubsWebApi.kt
      courtsWebApi.kt
      rentalsWebApi.kt
      usersWebApi.kt
    models
      Club
      Court
      Rental
      User
    server
      Server.kt
    services
      AuthententicationService.kt
      clubServices.kt
      courtSevices.kt
      rentalServices.kt
      userServices.kt
    storage
      clubsDataMem.kt
      courtsDataMem.kt
      rentalsDataMem.kt
      usersDataMem.kt
```

Figure 4. Main Folder

Open-API Specification

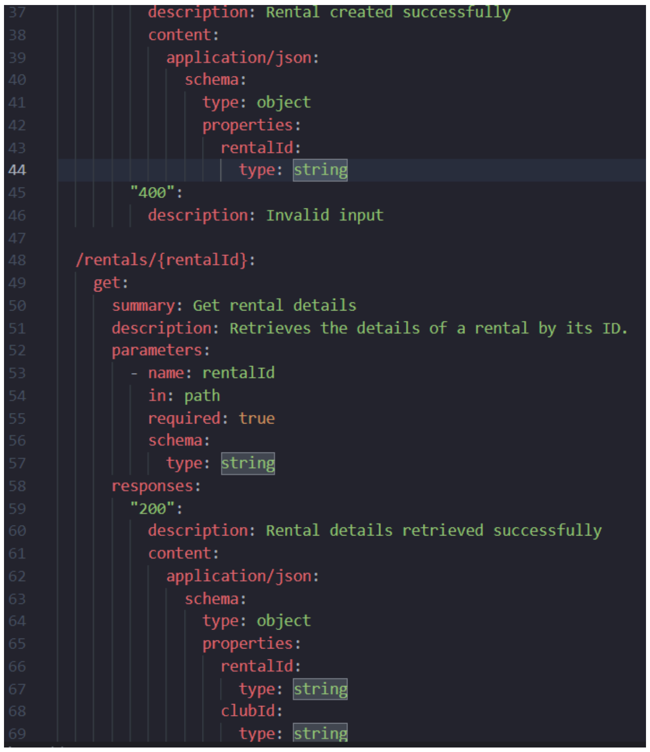


```yaml
yaml
openapi: 3.0.0
info:
  title: Padel Court Booking API
  version: 1.0.0
  description: API for managing padel court rentals.

servers:
  - url: http://localhost:9000
    description: Local Development Server

paths:
  /rentals:
    post:
      summary: Create a rental
      description: Creates a rental for a specific court and club.
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                clubId:
                  type: string
                courtId:
                  type: string
                userId:
                  type: string
                startTime:
                  type: string
                  format: date-time
                duration:
                  type: integer
      responses:
        "201":
```

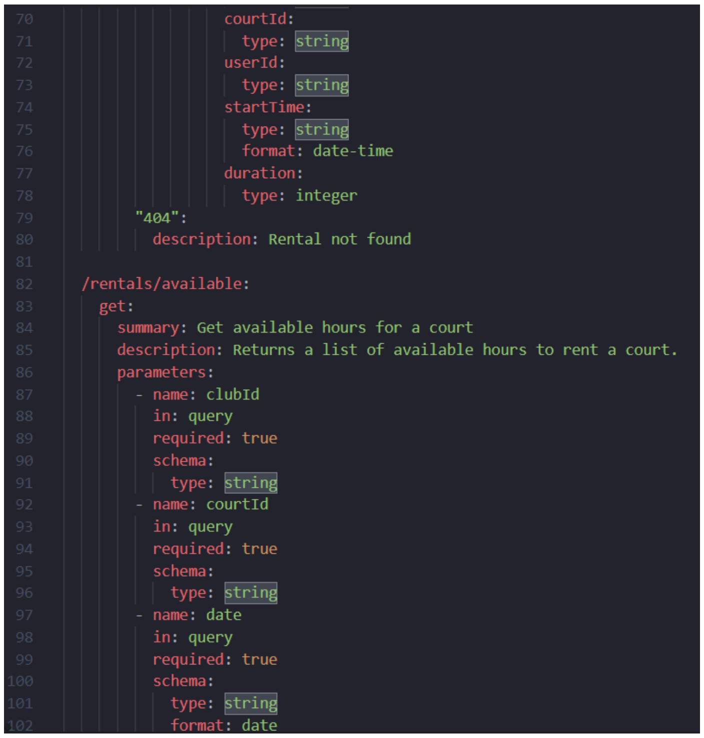
Figure 5. Openapi 1



```yaml
          description: Rental created successfully
          content:
            application/json:
              schema:
                type: object
                properties:
                  rentalId:
                    type: string
        "400":
          description: Invalid input

  /rentals/{rentalId}:
    get:
      summary: Get rental details
      description: Retrieves the details of a rental by its ID.
      parameters:
        - name: rentalId
          in: path
          required: true
          schema:
            type: string
      responses:
        "200":
          description: Rental details retrieved successfully
          content:
            application/json:
              schema:
                type: object
                properties:
                  rentalId:
                    type: string
                  clubId:
                    type: string
```

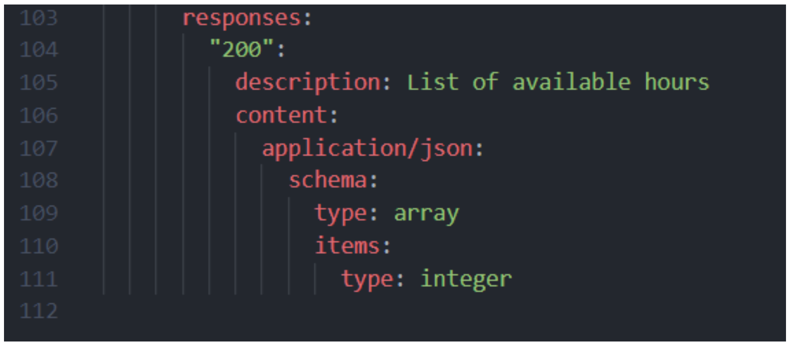
Figure 6. Openapi 2



```yaml
                    courtId:
                      type: string
                    userId:
                      type: string
                    startTime:
                      type: string
                      format: date-time
                    duration:
                      type: integer
          "404":
            description: Rental not found

  /rentals/available:
    get:
      summary: Get available hours for a court
      description: Returns a list of available hours to rent a court.
      parameters:
        - name: clubId
          in: query
          required: true
          schema:
            type: string
        - name: courtId
          in: query
          required: true
          schema:
            type: string
        - name: date
          in: query
          required: true
          schema:
            type: string
            format: date
```

Figure 7. Openapi 3



```yaml
      responses:
        "200":
          description: List of available hours
          content:
            application/json:
              schema:
                type: array
                items:
                  type: integer
```

Figure 8. Openapi 4

Highlights:

- The API follows RESTful principles and supports CRUD operations.
- Endpoints are designed to handle user, club, court, and rental management.
- Authentication is implemented via Bearer Token authentication.
- Pagination is supported for listing operations to enhance performance.
- API routes are implemented in clubsWebApi.kt, courtsWebApi.kt, rentalsWebApi.kt, and usersWebApi.kt.

Request Details

Handling API requests efficiently is crucial for the application's performance and security.

Request Flow:

1. A request is sent from the client application.
2. The API routes the request to the corresponding service.
3. The service processes the request and interacts with the data access layer.
4. If the request involves database operations, transactions are handled accordingly.
5. The processed response is sent back to the client

Relevant Classes and Functions:

- Server.kt: The entry point of the application.
- clubsWebApi.kt, courtsWebApi.kt, rentalsWebApi.kt, usersWebApi.kt: Implements the API routes.
- clubServices.kt, courtSevices.kt, rentalServices.kt, userServices.kt: Handles the business logic of the application.
- clubsDataMem.kt, courtsDataMem.kt, rentalsDataMem.kt, usersDataMem.kt: Manages in-memory data storage.

Parameter Validation:

- User input is validated at the API layer.
- Required parameters must be provided for each request.
- Security checks ensure only authorized users can access specific resources.

Connection Management

Efficient connection management is necessary for system stability and performance.

- Database connections are handled using PostgreSQL connection pools.
- Transactions are used to ensure data integrity in multi-step operations.
- Connections are closed after execution to optimize resource utilization.

Data Access

The data access layer is responsible for interacting with the database and ensuring data retrieval and manipulation are efficient.

- Repository classes provide structured database access.
- Complex SQL queries are optimized for performance and correctness.
- Data is initially stored in memory for testing before full database integration.

Error Handling/Processing

Proper error handling ensures a robust application that provides meaningful feedback to users and developers.

- Errors are handled using structured exception handling.
- Meaningful error messages are returned to the client for debugging.
- Logging mechanisms track errors and system events.

Critical Evaluation

The system has been evaluated based on its implemented functionality and potential areas for improvement.

Completed Functionality:

- All core functionalities outlined in Phase 1 have been implemented.
- Authentication and authorization mechanisms are in place.
- Unit tests have been developed for key API endpoints.

Testing and Results:

To ensure the correctness of the implemented functionalities, a set of unit tests were executed. The test results are as follows:

- Total Tests Executed: 5

- Failures: 0
- Ignored: 0
- Total Duration: 0.042s
- Success Rate: 100%

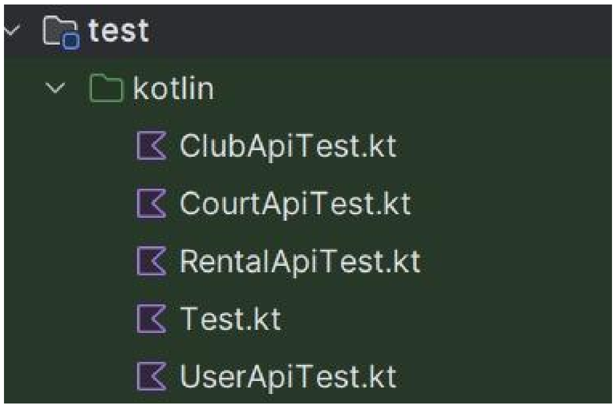
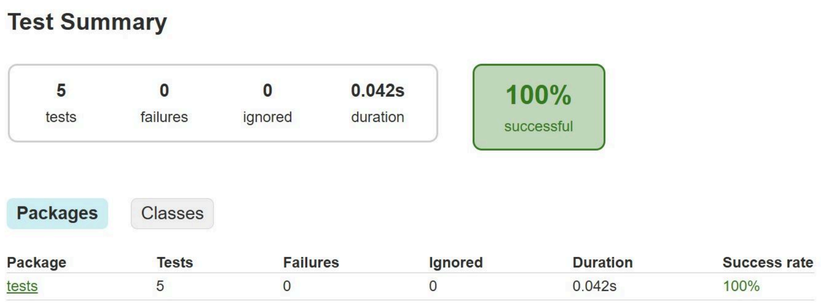


Figure 9. Test Folder


Test Summary

5	0	0	0.042s	100%
tests	failures	ignored	duration	successful

Package	Tests	Failures	Ignored	Duration	Success rate
tests	5	0	0	0.042s	100%

Figure 10. Test results