Library Management System

Project Documentation

https://github.com/sirdashasan/library-management

Prepared by: Hasan Sırdaş

May 2025

Click here to view the full Screenshots section. To jump to a specific screenshot, select one from the list below.

****Documentation Shortcuts****

Introduction

This project is a Library Management System developed using Java and Spring Boot. It is built using a monolithic architecture with layered design principles. The application allows librarians to manage books, users, borrow histories and overdue situations, while patrons can search, borrow and return books. Key features include role-based access control, JWT-based authentication and exception handling implemented with a global error response structure. The system also provides reactive book availability updates using Spring WebFlux and is containerized via Docker for easy deployment. Additionally, the project is documented using Swagger (OpenAPI) and includes both unit and integration tests to ensure functionality and reliability.

Technologies Used

Languages & Frameworks: Java 21, Spring Boot, Spring MVC, Spring Data JPA, Spring Security, Spring WebFlux
Security & Authentication: JWT, BCrypt, Role-Based Access Control
Database: PostgreSQL, H2 (for testing)
API Documentation: Swagger UI, OpenAPI 3
Testing: JUnit 5, Mockito, Reactor Test, Spring Security Test
Build Tool: Maven
Utilities: Lombok, dotenv-java
Deployment: Docker, Docker Compose
Frontend (Demo): React, JavaScript, Context API, Axios, Tailwind CSS

Installation

- **Clone the project repository**

  git clone https://github.com/sirdashasan/library-management.git

- **Install the required dependencies and build the Project**

  mvn clean package -DskipTests

- **Start the application using Docker Compose**

  docker-compose up –build

  After the containers start successfully, the following resources will be available:

- **Backend API Root:** http://localhost:8080/library/api
- **Swagger UI:** http://localhost:8080/library/api/swagger-ui/index.html
- **Reactive Book Availability Stream (SSE):** http://localhost:8080/library/api/books/availability-stream

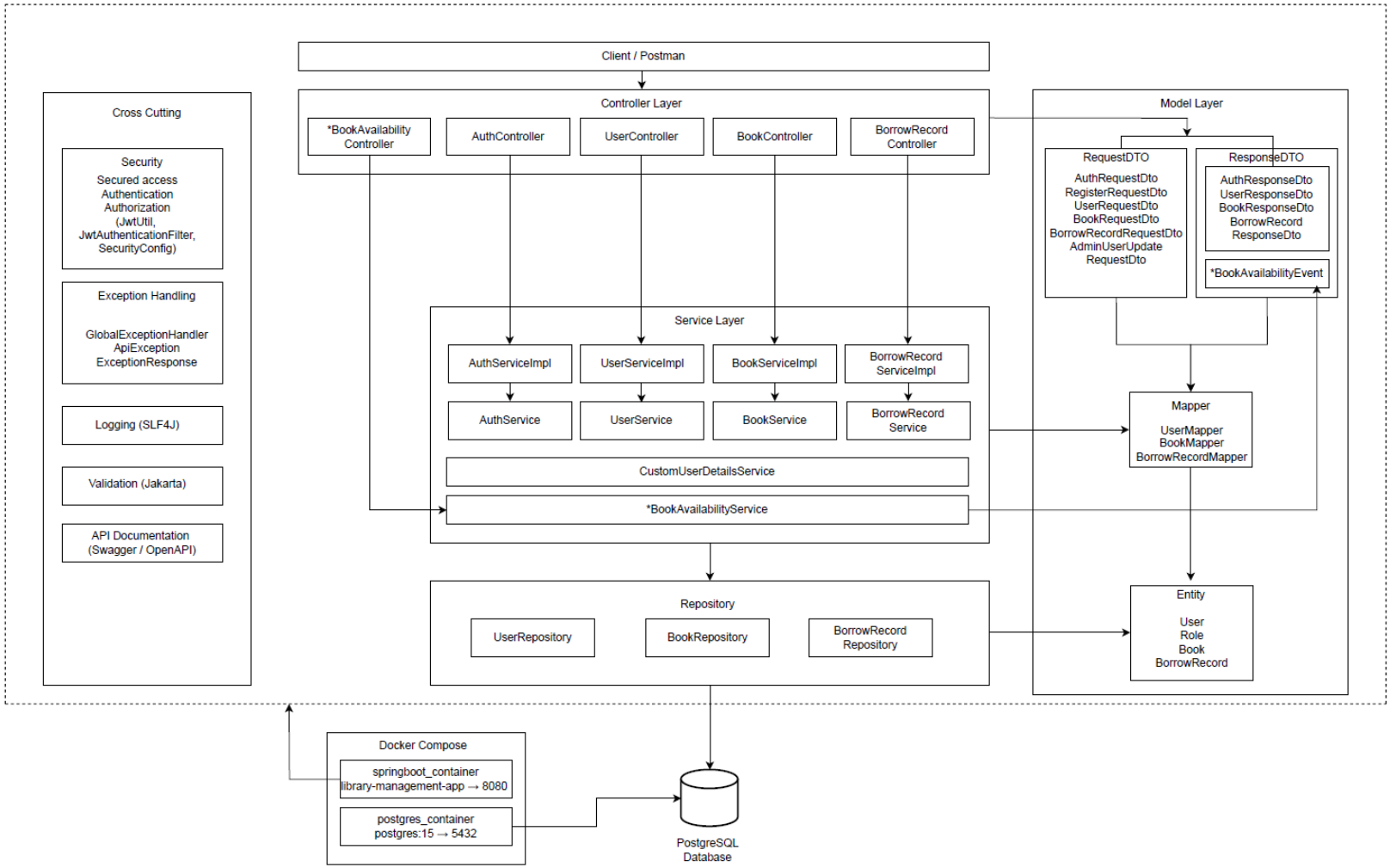
Configuration

- **application.properties:** Used for configuring the server port, context path, PostgreSQL connection, Swagger, JWT settings and other core application properties.

- **application.properties (test):** Contains in-memory H2 database settings and a separate JWT secret used for testing purposes.
- **.env file:** Stores sensitive environment variables such as DB_USERNAME, DB_PASSWORD, and JWT_SECRET. These values are not hardcoded and are injected securely via Docker Compose during runtime. **These values must be provided by the developer or deployment environment.**

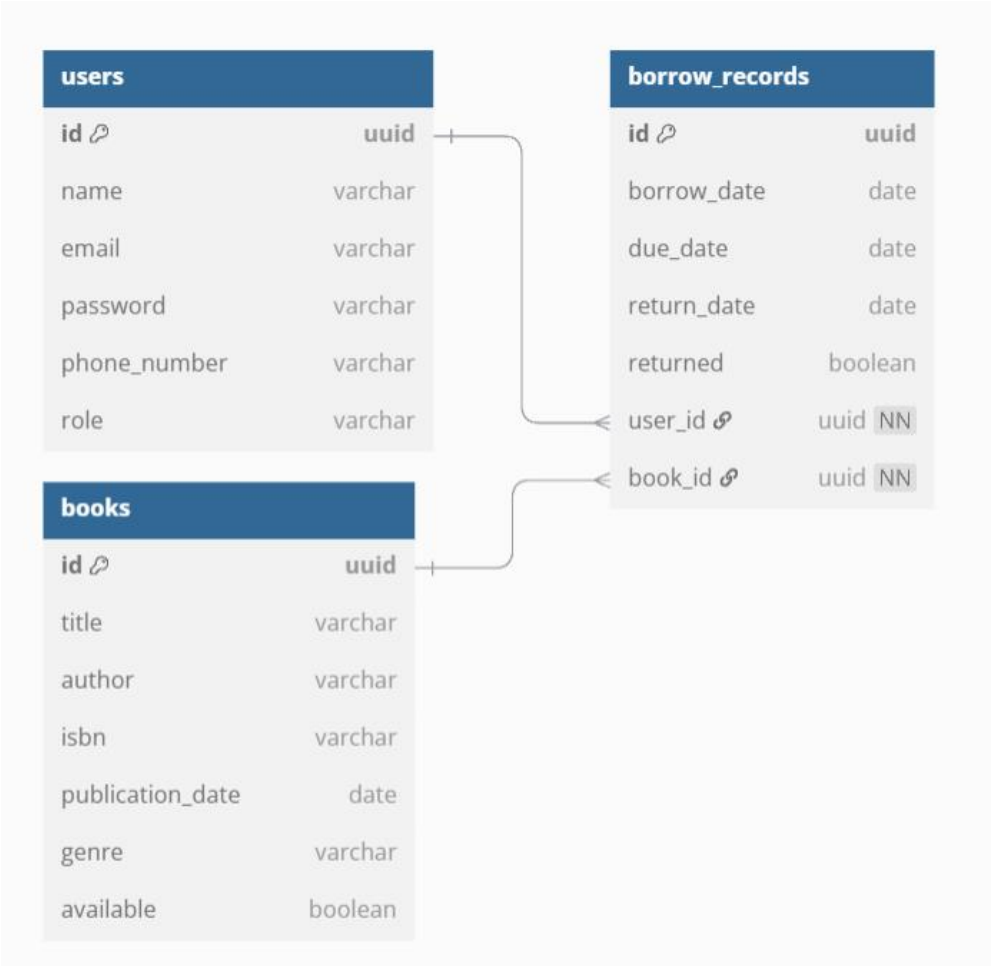
System Architecture

The application follows a layered monolithic architecture, separating concerns across controller, service, repository and model layers. The system also includes cross-cutting components such as security, validation, logging and exception handling, along with a reactive stream for book availability updates.



ER Diagram

The Entity Relationship Diagram (ERD) illustrates the core domain structure of the system, consisting of three main entities: User, Book and BorrowRecord. Each borrow record links a user and a book, forming many-to-one relationships to both User and Book entities.



Package Structure & Main Classes

The project follows a clean, layered package structure based on domain-driven principles.

com.hasan.library_management

```
├──── config
│     ├──── DataInitializer
│     ├──── OpenApiConfig
│     └──── SecurityConfig
├──── controller
│     ├──── AuthController
│     ├──── BookAvailabilityController
│     ├──── BookController
│     ├──── BorrowRecordController
│     └──── UserController
├──── dto
│     ├──── request
│     ├──── ├──── AdminUserUpdateRequestDto
│     ├──── ├──── AuthRequest
│     ├──── ├──── BookRequestDto
│     ├──── ├──── BorrowRecordRequestDto
│     ├──── ├──── RegisterRequest
│     ├──── ├──── UserRequestDto
│     ├──── response
│     ├──── ├──── AuthResponse
│     ├──── ├──── BookAvailabilityEvent
│     ├──── ├──── BookResponseDto
│     ├──── ├──── BorrowRecordResponseDto
│     ├──── ├──── UserResponseDto
├──── entity
│     ├──── Book
```

```
│   ├── BorrowRecord
│   ├── Role
│   ├── User
├── exceptions
│   ├── ApiException
│   ├── ExceptionResponse
│   ├── GlobalExceptionHandler
├── mapper
│   ├── BookMapper
│   ├── BorrowRecordMapper
│   ├── UserMapper
├── repository
│   ├── BookRepository
│   ├── BorrowRecordRepository
│   ├── UserRepository
├── security
│   ├── JwtAuthenticationFilter
│   ├── JwtUtil
├── service
│   ├── impl
│   │   ├── AuthServiceImpl
│   │   ├── BookServiceImpl
│   │   ├── BorrowRecordServiceImpl
│   │   ├── UserServiceImpl
│   ├── AuthService
│   ├── BookAvailabilityService
│   ├── BookService
│   ├── BorrowRecordService
│   ├── CustomUserDetailsService
│   ├── UserService
```

Role-Based Access Control Matrix

The table below summarizes which features are accessible by each user role in the system: Librarian and Patron. It also maps each feature to the corresponding backend API endpoint, helping to understand how role-based authorization is enforced across the application. Green check marks (✅) indicate access is granted, while red crosses (❌) indicate access is restricted.

Feature	Librarian	Patron	Endpoint	Module
View all users	✅	❌	GET /users	User
View user by ID	✅	❌	GET /users/{id}	User
View own user details	✅	✅	GET /users/me	User
Register	✅	✅	POST /auth/register	User
Login	✅	✅	POST /auth/login	User
Update any user	✅	❌	PUT /users/{id}	User
Delete a user	✅	❌	DELETE /users/{id}	User
View book list & details	✅	✅	GET /books	Book
Search books by ID	✅	✅	GET /books/{id}	Book
Search books by title (pagination)	✅	✅	GET /books/search/title	Book
Search books by author (pagination)	✅	✅	GET /books/search/author	Book
Search books by isbn (pagination)	✅	✅	GET /books/search/isbn	Book
Search books by genre (pagination)	✅	✅	GET /books/search/genre	Book
Add a book	✅	❌	POST /books	Book
Update a book	✅	❌	PUT /books/{id}	Book
Delete a book	✅	❌	DELETE /books/{id}	Book
View all borrow histories	✅	❌	GET /borrow-records	Borrow Record
View borrow record by user ID	✅	❌	GET /borrow-records/user/{userId}	Borrow Record
View own borrow records	✅	✅	GET /borrow-records/me	Borrow Record
View overdue records	✅	❌	GET /borrow-records/overdue	Borrow Record
Borrow a book	✅	✅	POST /borrow-records	Borrow Record
Return a book	✅	✅	PUT /borrow-records/return/{id}	Borrow Record
Report overdue	✅	❌	GET /borrow-records/overdue/report	Borrow Record
Book availability stream	✅	✅	GET /books/availability-stream	Book (Reactive)

Note: All endpoints listed above are relative to the base path defined in the configuration: server.servlet.context-path=/library/api

For example, GET /users maps to GET http://localhost:8080/library/api/users.

Class Overview

Entity

- **User:** Represents the users table. Includes fields such as id, name, email, password, phoneNumber and role. It has a one-to-many relationship with BorrowRecord.

- **Book:** Represents the books table. Contains fields like id, title, author, isbn, publicationDate, genre and available. It has a one-to-many relationship with BorrowRecord.

- **BorrowRecord:** Represents the borrow_records table. Includes id, borrowDate, dueDate, returnDate and returned. It has many-to-one relationships with both User and Book.

- **Role (Enum):** Defines application roles as LIBRARIAN and PATRON. Used for role-based access control within the application, particularly in User entity and SecurityConfig.

DTO – Request

- **UserRequestDto:** Used for creating a new user or during user registration. Contains name, email, password, phoneNumber and role fields. Includes validation annotations for input constraints.

- **RegisterRequest:** Specifically designed for the auth/register endpoint. Similar to UserRequestDto, it includes name, email, password, phoneNumber and role, with appropriate validation applied.

- **AdminUserUpdateRequestDto:** Used by librarians (with LIBRARIAN role) to update other users information. Unlike UserRequestDto, it does not include the password field. Input validation is enforced through annotations.

- **AuthRequest:** Used for user login (auth/login endpoint). Contains only email and password. Validation is applied for both fields.

- **BookRequestDto:** Handles both creation and update operations for books. Includes fields like title, author, isbn, publicationDate and genre. Validation constraints are enforced to ensure data integrity.

- **BorrowRecordRequestDto:** Represents the request data for borrowing a book. Includes userId, bookId, borrowDate and dueDate. All fields are validated using standard validation annotations.

DTO – Response

- **UserResponseDto:** Represents the public-facing user data returned in API responses. Does not include sensitive fields such as passwords. Contains id, name, email, phoneNumber and role.
- **AuthResponse:** Returned after a successful registration or login operation. Includes a JWT token used for authenticated requests.
- **BookResponseDto:** Used in book-related API responses. Includes id, title, author, isbn, publicationDate, genre and available status.
- **BorrowRecordResponseDto:** Returned in borrowing history and related operations. Contains detailed borrowing data such as bookTitle, userName, id, userId, bookId, borrowDate, dueDate, returnDate and returned.
- **BookAvailabilityEvent:** Represents real-time book availability updates emitted by BookAvailabilityService. It is used in reactive streams (WebFlux) and excluded from Swagger documentation to avoid conflicts with MVC-based endpoints.

Mapper

- **UserMapper:** Handles the conversion between User entities and their corresponding DTOs. Includes methods like toEntity, toResponseDto and updateEntityFromAdminDto for admin-level user updates.
- **BookMapper:** Responsible for converting between Book entities and DTOs. Provides toEntity, toResponseDto and updateEntity methods to support both creation and update operations.
- **BorrowRecordMapper:** Maps between BorrowRecord entities and DTOs. Includes toEntity and toResponseDto methods to handle conversion during borrowing and return operations.

Repository

- **UserRepository:** Extends JpaRepository<User, UUID> to provide basic CRUD operations. Includes a custom method “findByEmail(String email)” used for authentication and user lookup.
- **BookRepository:** Extends JpaRepository<Book, UUID>, enabling standard CRUD functionality. Contains custom search methods for pagination and filtering:
    - findByTitleContainingIgnoreCase
    - findByAuthorContainingIgnoreCase
    - findByIsbnContainingIgnoreCase
    - findByGenreContainingIgnoreCase
    - Also includes findByIsbn to ensure uniqueness of ISBN during book creation.

- **BorrowRecordRepository:** Extends JpaRepository<BorrowRecord, UUID> and supports advanced queries related to borrowing logic. Custom methods include:

    - findByUserId
    - findByBookId
    - findByReturnedFalse
    - findByReturnedFalseAndDueDateBefore
    - countByUserIdAndReturnedFalse
    - existsByUserIdAndReturnedFalseAndDueDateBefore

    These queries are used to handle borrowing history, overdue checks and borrowing limits.

Service

- **UserService (interface):** Defines the contract for user-related operations such as getAllUsers, getUserById, getOwnUserDetails, updateUser and deleteUser.

- **AuthService (interface):** Abstracts the authentication-related operations, including register() and login() methods.

- **BookService (interface):** Provides book-related business logic such as getAllBooks, getBookById, create, update and delete. Also includes search functionality using methods like searchByTitle, searchByAuthor, searchByIsbn and searchByGenre.

- **BorrowRecordService (interface):** Handles borrowing logic, including getAllBorrowRecords, borrowBook, returnBook, getBorrowRecordsByUserId, getOwnBorrowRecords, getOverdueRecords and generateOverdueReport.

- **BookAvailabilityService:** Enables real-time broadcasting of book availability changes (e.g., when a book is borrowed or returned).
  Utilizes Sinks.Many from Spring WebFlux to implement a publish–subscribe model:
    - sink.tryEmitNext(...) → emits a new BookAvailabilityEvent
    - getStream() → returns a Flux<BookAvailabilityEvent> to stream availability updates

- **CustomUserDetailsService:**
  Implements Spring Security's UserDetailsService interface.
  Responsible for loading user details based on the email found in the JWT token (loadUserByUsername).
  Maps user roles to Spring Security authorities in the ROLE_... format, used during authentication.

Service – Impl

- **UserServiceImpl:** Implements the UserService interface and handles user-related business logic. Uses UserRepository for database access and UserMapper for converting between entities and DTOs. Throws ApiException in exceptional scenarios and uses SLF4J for logging.

- **AuthServiceImpl:**
    - **register:** Checks for existing email, saves the new user, encodes the password, and generates a JWT token.
    - **login:** Authenticates the user via AuthenticationManager and returns a JWT token upon success.

      Throws ApiException on failure and logs non-sensitive information (e.g., email) using SLF4J.

- **BookServiceImpl:** Implements the BookService interface. Checks for ISBN duplication before adding a new book. Uses BookMapper.updateEntity() during book updates. All search methods return pageable results. Combines BookRepository and BookMapper in all operations. Logs operations via SLF4J and throws ApiException in case of errors.
- **BorrowRecordServiceImpl:** Implements the BorrowRecordService interface. Integrates with BookAvailabilityService to emit real-time book availability updates. Contains checkUserEligibility() method to enforce rules such as a maximum of 5 active borrowings and no overdue books. Users violating these conditions cannot borrow new books. The generateOverdueReport() method creates a detailed text report of overdue records, logs it to the console, and writes it to overdue_report.txt. Uses SLF4J for logging and throws ApiException in case of exceptions.

Controller

- **UserController:** Manages user-related endpoints with detailed Swagger/OpenAPI documentation and response codes.
    - GET /users → Returns all users
    - GET /users/{id} → Returns a specific user by ID
    - GET /users/me → Returns the authenticated user's own profile using JWT
    - PUT /users/{id} → Updates a user
    - DELETE /users/{id} → Deletes a user

- **AuthController:** Handles authentication endpoints with proper Swagger/OpenAPI annotations.
    - POST /auth/register → Registers a new user and returns a JWT token
    - POST /auth/login → Authenticates user credentials and returns a JWT token

- **BookController:** Provides endpoints for managing book resources and includes search functionalities.
    - GET /books → Retrieves all books
    - GET /books/{id} → Retrieves a book by ID
    - POST /books → Adds a new book

- PUT /books/{id} → Updates book information
- DELETE /books/{id} → Deletes a book
- GET /books/search/{field} → Supports search by title, author, ISBN or genre with pagination

- **BorrowRecordController:** Manages borrow/return operations with role-based access and Swagger documentation.
  - GET /borrow-records → Returns all borrow records
  - POST /borrow-records → Creates a new borrow record
  - PUT /borrow-records/return/{id} → Marks a book as returned
  - GET /borrow-records/user/{userId} → Returns borrow records of a specific user
  - GET /borrow-records/me → Returns the authenticated user's borrow history
  - GET /borrow-records/overdue → Returns a list of overdue borrow records
  - GET /borrow-records/overdue/report → Generates a text-based overdue report, logs it, and saves it as a file

- **BookAvailabilityController:** Provides a reactive streaming endpoint using Server-Sent Events (SSE).
  - GET /books/availability-stream
    This endpoint streams real-time book availability changes.
    Triggered inside BorrowRecordServiceImpl.borrowBook() and returnBook() after changing book status via book.setAvailable(...).

Security

- **JwtUtil:** Responsible for generating, validating and extracting information from JWT tokens. The token includes the user's email and role as claims. Key methods include:
  - generateToken(...)
  - validateToken(...)
  - getEmailFromToken(...)
  - getRoleFromToken(...)

- **JwtAuthenticationFilter:** Intercepts every HTTP request to check for a valid JWT token in the Authorization header (format: Bearer <token>). If the token is valid, the associated user details are loaded and set into the Spring Security SecurityContext. This filter runs before the UsernamePasswordAuthenticationFilter in the Spring Security filter chain.

Config

- **SecurityConfig:** Contains the full Spring Security configuration for the application. Defines public endpoints, role-based access rules, and integrates the custom JwtAuthenticationFilter into the filter chain. Includes core security settings such as CORS, CSRF, session management policy, and password encryption using BCrypt.

- **OpenApiConfig:** Configures Swagger/OpenAPI documentation for the project. Also sets up a JWT-based security scheme for authorized API testing through the Swagger UI.

- **DataInitializer:** Loads demo data when the application starts. Primarily used to speed up development and testing processes by populating the system with initial users and books.

Exceptions

- **ApiException:** Custom runtime exception that allows throwing application-specific errors with custom HTTP status codes (e.g., NOT_FOUND, BAD_REQUEST). Typically used for domain-level validations.

- **ExceptionResponse:** Data transfer object (DTO) that represents structured error information including message, status code and timestamp. Used internally to build uniform error responses.

- **GlobalExceptionHandler**: Centralized exception handling class annotated with @ControllerAdvice. Handles and formats various exceptions including:
  - ApiException for custom domain errors
  - MethodArgumentNotValidException for validation failures
  - HttpMessageNotReadableException for malformed JSON input
  - BadCredentialsException for invalid login attempts
  - Generic Exception as a fallback for unexpected errors

Testing Overview

CustomUserDetailsServiceTest (Unit Test)

This test class verifies the behavior of the loadUserByUsername(...) method in the CustomUserDetailsService class.

- Uses @ExtendWith(MockitoExtension.class) to enable Mockito-based unit testing.
- The UserRepository dependency is mocked using @Mock.
- The CustomUserDetailsService is injected with its dependencies using @InjectMocks.

Test Methods:

- **loadUserByUsername_shouldReturnUserDetails_whenUserExists()**
  Verifies that the correct UserDetails object is returned when a user is found by email. Also checks that the email, password and ROLE_PATRON authority are included as expected.
- **loadUserByUsername_shouldThrowException_whenUserNotFound()**
  Ensures that a UsernameNotFoundException is thrown when no user is found for the given email. Verifies that the exception message matches the expected output.

BookAvailabilityServiceTest (Unit Test)

This test class verifies the behavior of the publishAvailabilityChange(...) method in the BookAvailabilityService class.

- Uses StepVerifier from the Reactor Test library to test the reactive stream.
- Ensures that emitted events are correctly published and can be consumed through the getStream() method.

Test Method:

- **publishAvailabilityChange_shouldEmitEvent()**
  Verifies that when publishAvailabilityChange(...) is called, a corresponding BookAvailabilityEvent is emitted through the stream.
  Asserts that the emitted event contains the expected bookId and available values.

UserServiceImplTest (Unit Test)

This test class verifies the behavior of user-related methods in the UserServiceImpl class.

- UserRepository and UserMapper are mocked using @Mock.
- Mocks are injected into the service class using @InjectMocks.

Test Methods:

- **getAllUsers_shouldReturnUserList()**
  Ensures the correct number and order of user DTOs are returned when listing all users.

- **getUserById_shouldReturnUser_whenExists()**
  Verifies that a valid user ID returns the correct user DTO.
- **getUserById_shouldThrowException_whenUserNotFound()**
  Asserts that an ApiException is thrown when the user is not found for the given ID.
- **getOwnUserDetails_shouldReturnUser_whenExists()**
  Verifies that a user is correctly retrieved and mapped by email.
- **getOwnUserDetails_shouldThrowException_whenUserNotFound()**
  Ensures that an ApiException is thrown when no user is found for the given email.
- **updateUser_shouldUpdateUser_whenExists()**
  Checks that an existing user is successfully updated and the correct DTO is returned.
- **updateUser_shouldThrowException_whenUserNotFound()**
  Asserts that an ApiException is thrown when attempting to update a non-existent user.
- **deleteUser_shouldDeleteUser_whenExists()**
  Ensures that a valid user is deleted successfully without throwing an exception.
- **deleteUser_shouldThrowException_whenUserNotFound()**
  Verifies that an ApiException with the correct message is thrown when trying to delete a non-existent user.

BookServiceImplTest (Unit Test)

This test class verifies the correct behavior of methods in the BookServiceImpl class.

- BookRepository and BookMapper are mocked using @Mock.
- A real instance of BookServiceImpl is injected with mocked dependencies using @InjectMocks.

Test Methods:

- **getAllBooks_shouldReturnBookList()**
  Ensures that all books in the database are correctly retrieved and returned as a list of DTOs.
- **getBookById_shouldReturnBook_whenExists()**
  Verifies that a valid book ID returns the correct BookResponseDto.
- **getBookById_shouldThrowException_whenNotFound()**
  Asserts that an ApiException is thrown when the book is not found.
- **createBook_shouldCreate_whenIsbnNotExists()**
  Checks that a book is successfully created when the provided ISBN is not already in the database.
- **createBook_shouldThrowException_whenIsbnAlreadyExists()**
  Ensures that an ApiException is thrown if a book with the same ISBN already exists.
- **updateBook_shouldUpdateBook_whenExists()**
  Verifies that an existing book is updated successfully.

- **updateBook_shouldThrowException_whenBookNotFound()**
  Asserts that an ApiException is thrown when trying to update a non-existent book.
- **deleteBook_shouldDeleteBook_whenExists()**
  Ensures that a book is deleted successfully if it exists.
- **deleteBook_shouldThrowException_whenBookNotFound()**
  Verifies that an ApiException is thrown when trying to delete a book that does not exist.
- **searchByTitle_shouldReturnPagedBooks()**
  Ensures that the system returns paginated books filtered by title correctly.
- **searchByAuthor_shouldReturnPagedBooks()**
  Verifies correct pagination and filtering when searching books by author name.
- **searchByIsbn_shouldReturnPagedBooks()**
  Confirms that searching by ISBN returns the appropriate book(s).
- **searchByGenre_shouldReturnPagedBooks()**
  Validates that filtering by genre works correctly and returns a paginated response.

BorrowRecordServiceImplTest (Unit Test)

This test class verifies the correctness of the borrowing and return logic in the BorrowRecordServiceImpl class.

- Dependencies such as BookRepository, UserRepository, BorrowRecordRepository, BookAvailabilityService, and BorrowRecordMapper are mocked using @Mock.
- A real instance of BorrowRecordServiceImpl is injected with these mocks using @InjectMocks.

Test Methods:

- **getAll_shouldReturnBorrowRecords()**
  Ensures that all borrow records are retrieved correctly as a list.
- **borrowBook_shouldCreateBorrowRecord_whenValidRequest()**
  Verifies that a borrow record is created when a valid request is made, and a BookAvailabilityEvent is emitted.
- **borrowBook_shouldThrowException_whenBookNotFound()**
  Asserts that an ApiException is thrown if the requested book does not exist.
- **borrowBook_shouldThrowException_whenUserNotFound()**
  Ensures an ApiException is thrown with the correct message if the user is not found.
- **borrowBook_shouldThrowException_whenBookNotAvailable()**
  Verifies that an error is thrown if the book is already borrowed and unavailable.
- **borrowBook_shouldThrowException_whenUserHasAlreadyBorrowed5Books()**
  Ensures that a user cannot borrow more than 5 books concurrently; an exception is thrown if the limit is exceeded.

- **borrowBook_shouldThrowException_whenUserHasOverdueBooks()**
  Verifies that borrowing is blocked if the user has overdue books.
- **returnBook_shouldMarkAsReturned_whenValidRequest()**
  Ensures that the borrow record is marked as returned and the book is made available again.
- **returnBook_shouldThrowException_whenRecordNotFound()**
  Asserts that an ApiException is thrown if the return request references a non-existent record.
- **returnBook_shouldThrowException_whenAlreadyReturned()**
  Verifies that an exception is thrown when attempting to return a book that has already been returned.
- **getBorrowRecordsByUserId_shouldReturnRecords_whenUserExists()**
  Ensures that a user's borrow records are returned correctly when the user exists.
- **getBorrowRecordsByUserId_shouldThrowException_whenUserNotFound()**
  Asserts that an ApiException is thrown with the correct message if the user does not exist.
- **getOwnBorrowRecords_shouldReturnRecords_whenUserExists()**
  Verifies that borrow records are correctly returned based on the email extracted from the JWT token.
- **getOwnBorrowRecords_shouldThrowException_whenUserNotFound()**
  Ensures that an ApiException is thrown if the user referenced in the token is not found.
- **getOverdueRecords_shouldReturnOverdueRecords()**
  Ensures that overdue records are correctly filtered and returned.
- **generateOverdueReport_shouldReturnFormattedText_whenOverdueExists()**
  Verifies that a properly formatted report of overdue records is generated, logged to the console, and saved as a .txt file.

AuthServiceImplTest (Unit Test)

This test class verifies the correct behavior of register and login operations in the AuthServiceImpl class under both success and failure scenarios.

- Dependencies such as UserRepository, PasswordEncoder, JwtUtil, and AuthenticationManager are mocked using @Mock.
- These mocks are injected into a real instance of AuthServiceImpl using @InjectMocks.

Test Methods:

- **register_shouldReturnToken_whenEmailIsNotRegistered()**
  Verifies that when the provided email is not already registered, the user is successfully created and a JWT token is returned.
- **register_shouldThrowException_whenEmailAlreadyExists()**
  Ensures that an ApiException is thrown if the email is already in use.

- **login_shouldReturnToken_whenCredentialsAreValid()**
  Verifies that a valid login request results in a successful authentication and a JWT token is returned.
- **login_shouldThrowException_whenAuthenticationFails()**
  Asserts that a BadCredentialsException is thrown by Spring Security when authentication fails.
- **login_shouldThrowException_whenUserNotFound()**
  Ensures that an ApiException with an appropriate message is thrown if the user is not found in the database, even after successful authentication.

UserControllerTest (Integration Test)

This test class validates the integration behavior of the /users endpoints, focusing on access control, authorization based on user roles, and expected HTTP responses.

- Uses @SpringBootTest and @AutoConfigureMockMvc to load the full application context and test the API layer in a realistic environment.
- JWT tokens are generated via register/login requests and used to simulate authenticated access.
- Both LIBRARIAN and PATRON roles are tested for their access rights.

Test Methods:

- **getAll_shouldReturnOk**
  Confirms that a user with a LIBRARIAN token can successfully retrieve the list of users.
- **getAll_shouldReturnForbidden_whenNoToken**
  Verifies that access is denied when no token is provided.
- **getAll_shouldReturnForbidden_whenUserIsPatron**
  Ensures that a PATRON role cannot access the /users list.
- **getUserById_shouldReturnUser_whenAuthorizedAsLibrarian**
  Confirms that a LIBRARIAN can retrieve a specific user's information by ID.
- **getUserById_shouldReturnForbidden_whenUserIsPatron**
  Asserts that a PATRON cannot access another user's data.
- **getUserById_shouldReturnNotFound_whenUserDoesNotExist**
  Verifies that a 404 is returned when requesting a non-existent user ID.
- **getOwnDetails_shouldReturnOk_whenAuthorized**
  Confirms that an authenticated user can retrieve their own profile using a valid token.
- **getOwnDetails_shouldReturnForbidden_whenNoToken**
  Verifies that access is denied if no token is provided.
- **getOwnDetails_shouldReturnUnauthorized_whenTokenIsInvalid**
  Asserts that an invalid token results in a 401 Unauthorized response.

- **updateUser_shouldUpdate_whenLibrarianAndValidData**
  Confirms that a LIBRARIAN can successfully update a user's information with valid input.
- **updateUser_shouldReturnForbidden_whenUserIsPatron**
  Ensures that a PATRON cannot update another user's data.
- **updateUser_shouldReturnNotFound_whenUserDoesNotExist**
  Verifies that updating a non-existent user returns a 404 response.
- **updateUser_shouldReturnForbidden_whenNoToken**
  Asserts that an update operation is rejected when no token is provided.
- **deleteUser_shouldDelete_whenLibrarianAndValidId**
  Confirms that a LIBRARIAN can successfully delete another user.
- **deleteUser_shouldReturnForbidden_whenUserIsPatron**
  Ensures that a PATRON cannot delete another user.
- **deleteUser_shouldReturnNotFound_whenUserDoesNotExist**
  Verifies that deleting a non-existent user returns a 404 response.
- **deleteUser_shouldReturnForbidden_whenNoToken**
  Asserts that deletion is denied when no token is provided.

BookControllerTest (Integration Test)

This test class ensures the correctness of all /books endpoints, including CRUD operations and search functionality.
It verifies JWT-based authorization, validation rules, and various error scenarios using full end-to-end tests.

- Built using @SpringBootTest and MockMvc to simulate real HTTP requests within a fully initialized Spring application context.
- Covers all basic CRUD operations and search operations by title, author, isbn, and genre.

Test Methods:

- **getAllBooks_shouldReturnOk_whenAuthorized()**
  Verifies that books are successfully listed when a valid token is provided.
- **getBookById_shouldReturnOk_whenBookExistsAndAuthorized()**
  Ensures that a created book can be fetched correctly by its ID.
- **getBookById_shouldReturnNotFound_whenBookDoesNotExist()**
  Verifies that a 404 status is returned for a non-existent book ID.
- **createBook_shouldReturnOk_whenValidRequestAndAuthorized()**
  Confirms that a book is added successfully when all required fields are provided with a valid token.
- **createBook_shouldReturnForbidden_whenNoTokenProvided()**
  Ensures that a 403 Forbidden response is returned when trying to create a book without authentication.

- **createBook_shouldReturnBadRequest_whenMissingRequiredFields()**
  Verifies that the API returns a 400 Bad Request when required fields are missing.
- **updateBook_shouldUpdate_whenBookExistsAndValidRequest()**
  Confirms that an existing book is successfully updated with valid input.
- **updateBook_shouldReturnNotFound_whenBookDoesNotExist()**
  Ensures a 404 is returned when attempting to update a non-existent book.
- **updateBook_shouldReturnBadRequest_whenValidationFails()**
  Asserts that the API responds with 400 if invalid data (e.g., empty title) is submitted.
- **deleteBook_shouldDelete_whenBookExistsAndAuthorized()**
  Verifies that a book can be deleted and is no longer retrievable afterward.
- **deleteBook_shouldReturnNotFound_whenBookDoesNotExist()**
  Ensures that deleting a non-existent book returns a 404 response.
- **deleteBook_shouldReturnForbidden_whenNoToken()**
  Confirms that deletion is denied when no JWT token is provided.
- **searchByTitle_shouldReturnResults_whenMatchingBooksExist()**
  Validates that books matching the given title are returned.
- **searchByTitle_shouldReturnEmpty_whenNoMatchesFound()**
  Ensures that a search with a non-matching title returns an empty result.
- **searchByAuthor_shouldReturnResults_whenMatchingBooksExist()**
  Verifies that the search by author name returns matching books.
- **searchByAuthor_shouldReturnEmpty_whenNoMatchesFound()**
  Confirms that no results are returned for unmatched authors.
- **searchByIsbn_shouldReturnResults_whenMatchingBookExists()**
  Validates that a matching ISBN returns the correct book.
- **searchByIsbn_shouldReturnEmpty_whenNoMatchFound()**
  Ensures no results are returned when the ISBN does not match any records.
- **searchByGenre_shouldReturnResults_whenMatchingGenreExists()**
  Verifies that books are correctly returned when searched by genre.
- **searchByGenre_shouldReturnEmpty_whenNoMatchFound()**
  Confirms that a search by non-matching genre returns an empty result.

BorrowRecordControllerTest (Integration Test)

This test class verifies the accuracy and access control of all /borrow-records endpoints, including borrowing, returning, and overdue record management.

- Built using @SpringBootTest, MockMvc, and JWT-based authentication to perform full end-to-end integration testing.

Test Methods:

- **getAllBorrowRecords_shouldReturnOk_whenAuthorized()**
  Ensures that an authorized user can retrieve all borrow records successfully.
- **getAllBorrowRecords_shouldReturnForbidden_whenNoToken()**
  Verifies that access is denied when no token is provided.
- **borrowBook_shouldReturnOk_whenValidRequestAndAuthorized()**
  Confirms that a valid borrow request results in successful borrowing.
- **borrowBook_shouldReturnForbidden_whenNoToken()**
  Asserts that borrowing is forbidden without a JWT token.
- **borrowBook_shouldReturnBadRequest_whenDueDateIsNotFuture()**
  Ensures a 400 Bad Request is returned when the due date is not in the future.
- **returnBook_shouldReturnOk_whenValidIdAndAuthorized()**
  Verifies that a book can be returned successfully with a valid ID and token.
- **returnBook_shouldReturnNotFound_whenRecordDoesNotExist()**
  Ensures a 404 Not Found is returned when attempting to return a non-existent record.
- **returnBook_shouldReturnForbidden_whenNoTokenProvided()**
  Asserts that the return operation is blocked without authentication.
- **getRecordsByUserId_shouldReturnList_whenValidUserAndAuthorized()**
  Confirms that a librarian can retrieve borrow records of a specific user.
- **getRecordsByUserId_shouldReturnEmptyList_whenNoRecordsFound()**
  Ensures that an empty list is returned when the user has no borrow records.
- **getRecordsByUserId_shouldReturnForbidden_whenNoToken()**
  Verifies that access to another user's records is denied without a token.
- **getOwnRecords_shouldReturnList_whenUserHasBorrowedBooks()**
  Confirms that an authenticated user can access their own borrow records using /me.
- **getOwnRecords_shouldReturnEmptyList_whenNoRecordsExist()**
  Verifies that /me returns an empty list when the user has no borrow history.
- **getOwnRecords_shouldReturnForbidden_whenNoTokenProvided()**
  Ensures that access to /me is denied without authentication.
- **getOverdueRecords_shouldReturnList_whenOverdueExists()**
  Verifies that overdue records are correctly returned for books past due and not returned.
- **getOverdueRecords_shouldReturnEmptyList_whenNoOverdue()**
  Confirms that an empty list is returned when no overdue records exist.
- **getOverdueRecords_shouldReturnForbidden_whenNoTokenProvided()**
  Ensures that listing overdue records is forbidden without authentication.
- **generateOverdueReport_shouldReturnOk_whenAuthorized()**
  Confirms that a LIBRARIAN user can successfully generate the overdue report.

- **generateOverdueReport_shouldReturnUnauthorized_whenNoToken()**
  Verifies that the report generation endpoint is inaccessible without a token.
- **generateOverdueReport_shouldReturnForbidden_whenNotLibrarian()**
  Ensures that a PATRON user receives a 403 Forbidden when trying to access the report generation endpoint.

AuthControllerTest (Integration Test)

This test class verifies the authentication flow of the application, specifically the behavior of the register and login endpoints.

- Built using @SpringBootTest and MockMvc to perform realistic integration testing for user authentication.

Test Methods:

- **register_shouldReturnOk_whenEmailIsNew()**
  Verifies that a new user can successfully register and receives a 200 OK response.
- **register_shouldReturnBadRequest_whenEmailAlreadyExists()**
  Ensures that a 400 Bad Request is returned when attempting to register with an email that is already in use.
- **login_shouldReturnOk_whenCredentialsAreCorrect()**
  Confirms that a user can log in successfully with valid email and password credentials.
- **login_shouldReturnUnauthorized_whenPasswordIsWrong()**
  Verifies that a 401 Unauthorized response is returned when the password is incorrect.
- **login_shouldReturnUnauthorized_whenEmailDoesNotExist()**
  Ensures that logging in with a non-existent email results in a 401 Unauthorized response.

BookAvailabilityControllerTest (Integration Test)

This test class verifies the real-time event streaming functionality of the /books/availability-stream endpoint, which is powered by the BookAvailabilityService.

- It is a **reactive integration test**, implemented using WebTestClient to test the WebFlux-enabled **Server-Sent Events (SSE)** endpoint.
- The endpoint emits BookAvailabilityEvent objects when books are borrowed or returned.

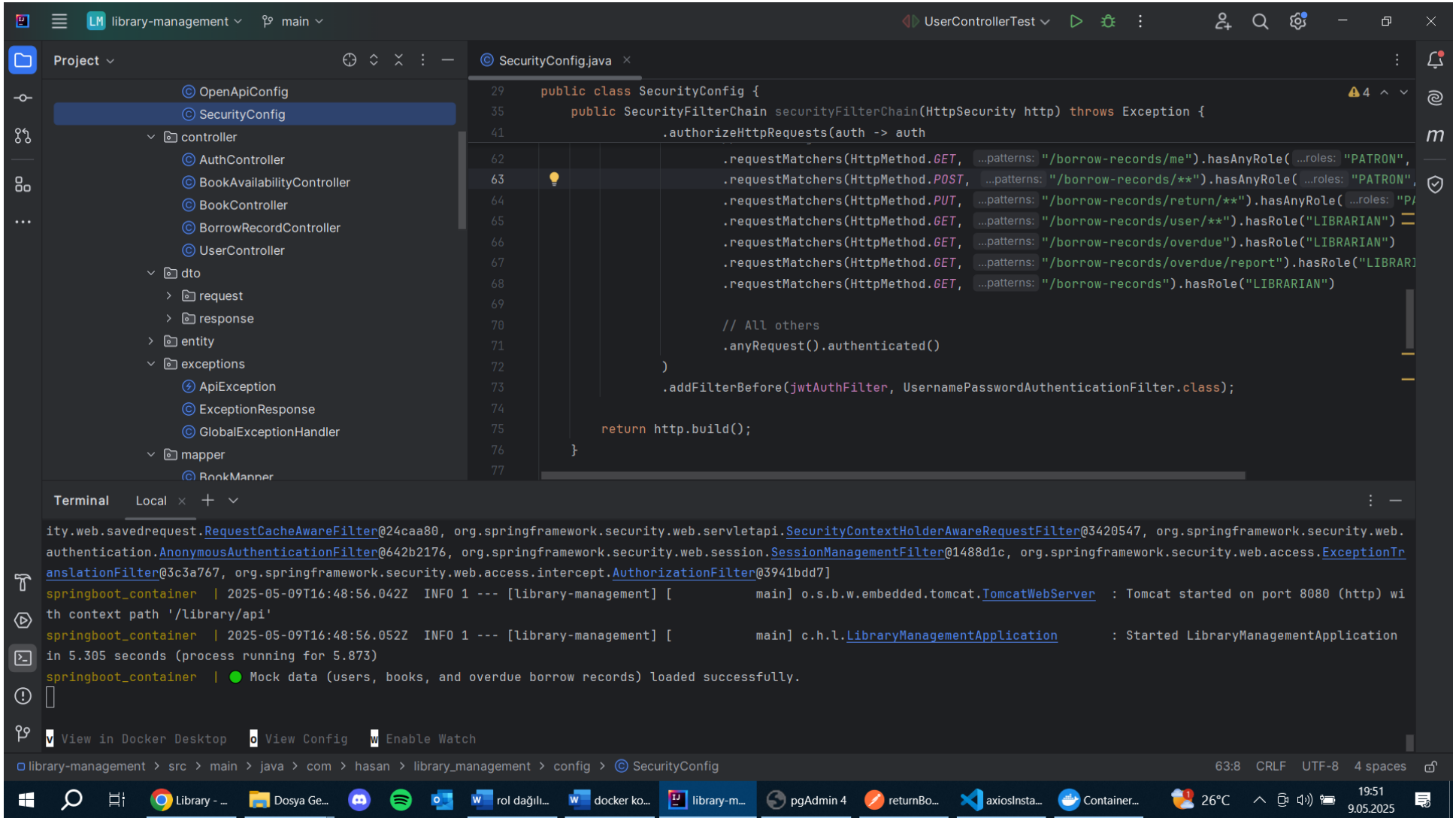
Test Method:

- **streamAvailability_shouldEmitEvents()**
  - Simulates the emission of a new BookAvailabilityEvent via a separate thread.
  - Subscribes to the /books/availability-stream endpoint using WebTestClient.
  - Verifies that:
    - The stream returns 200 OK.

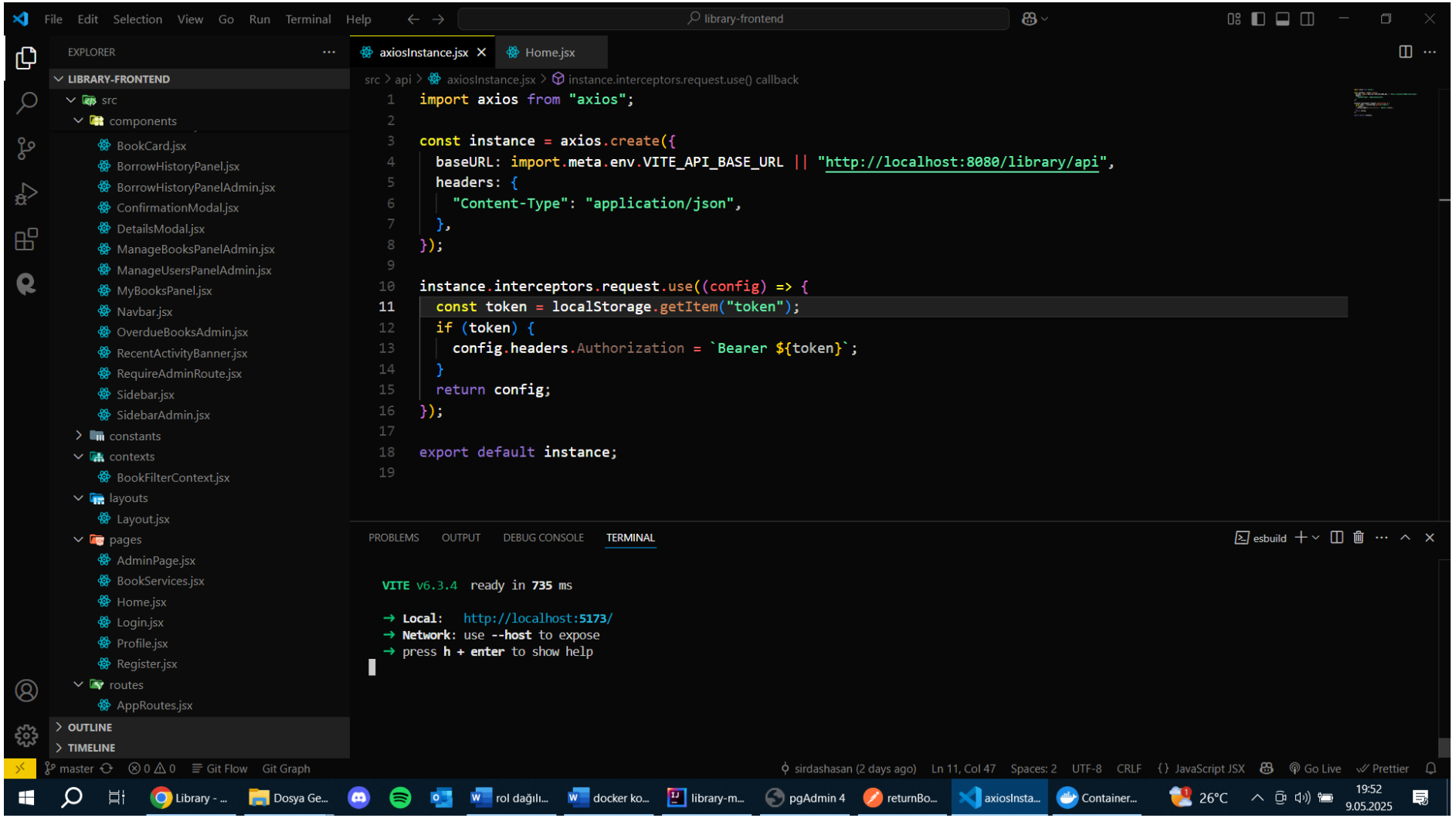
- The emitted event is correctly delivered to the subscriber.
- The event payload includes the expected bookId and available fields with correct values.

Screenshots - Project Startup Screenshots

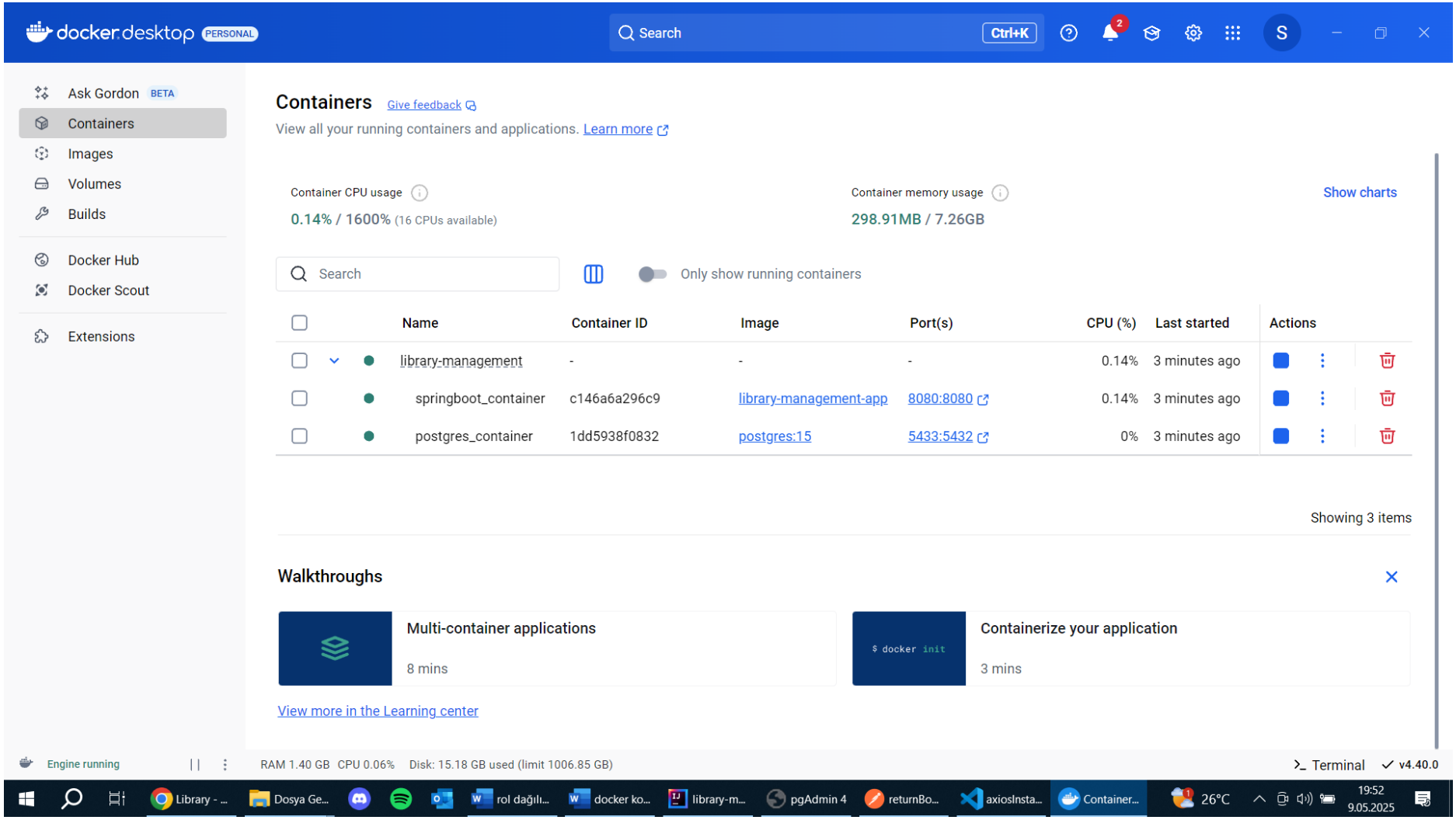
Backend Launch - IntelliJ IDEA



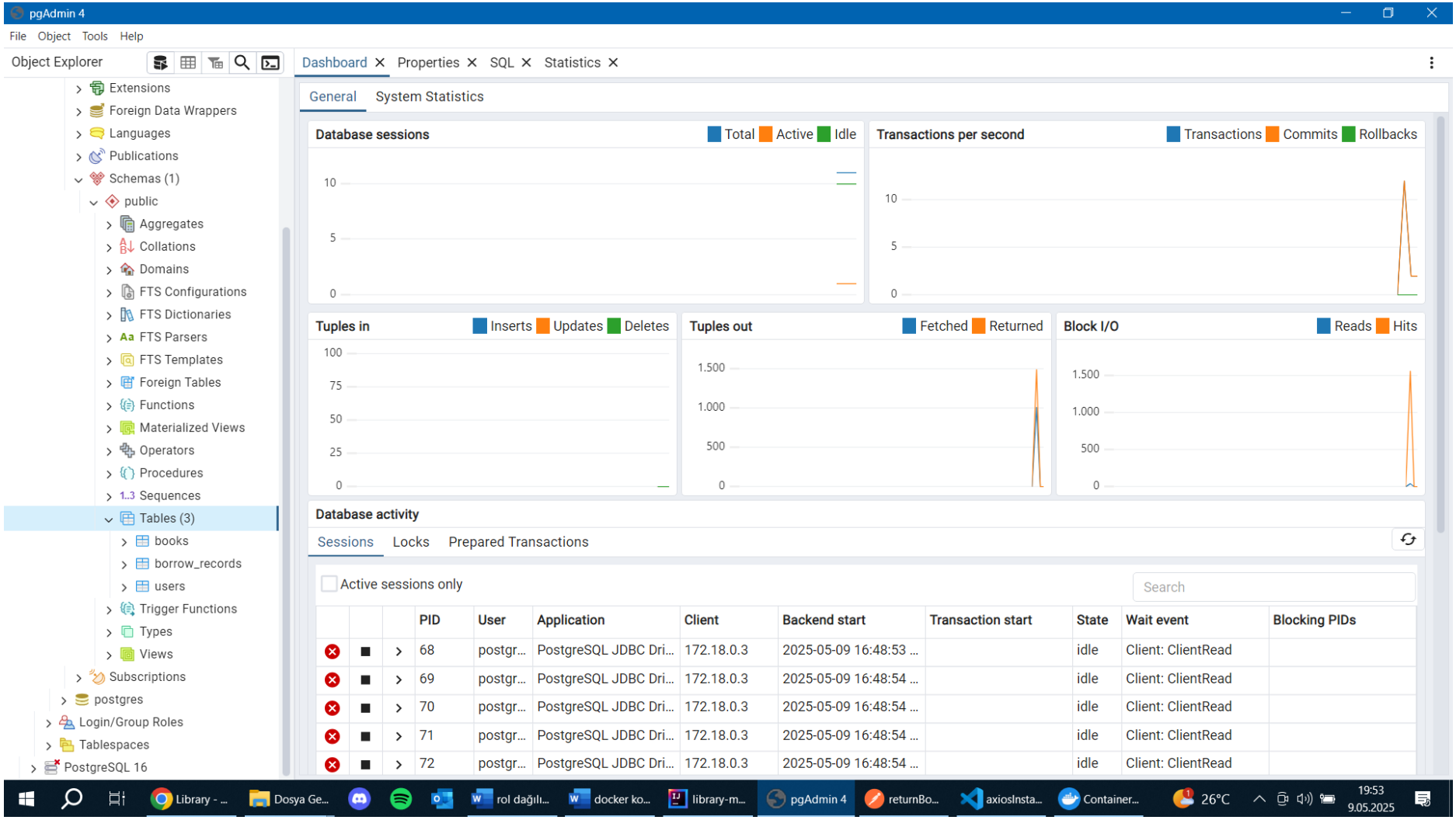
Frontend Launch – VSCode



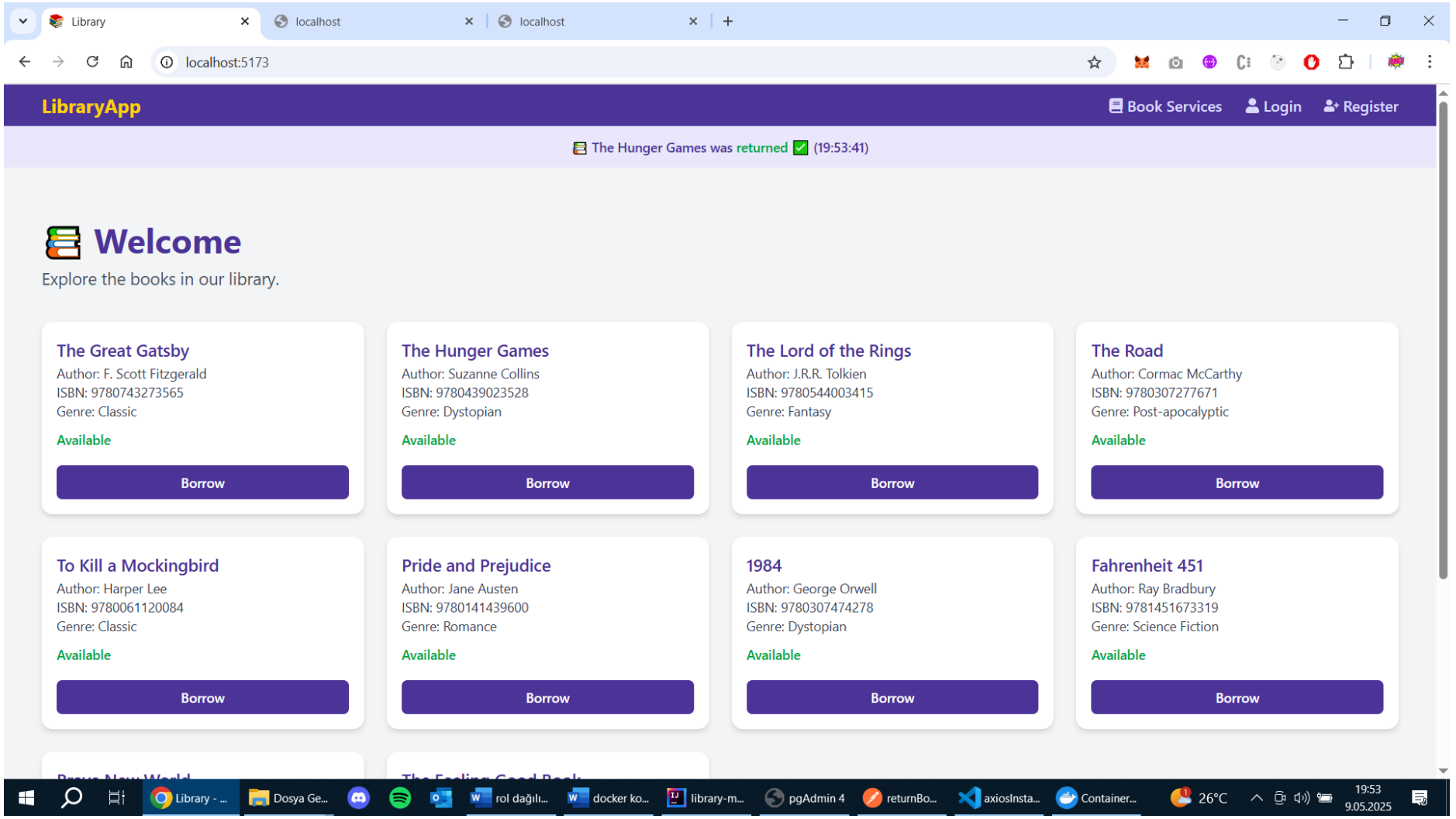
Docker Desktop



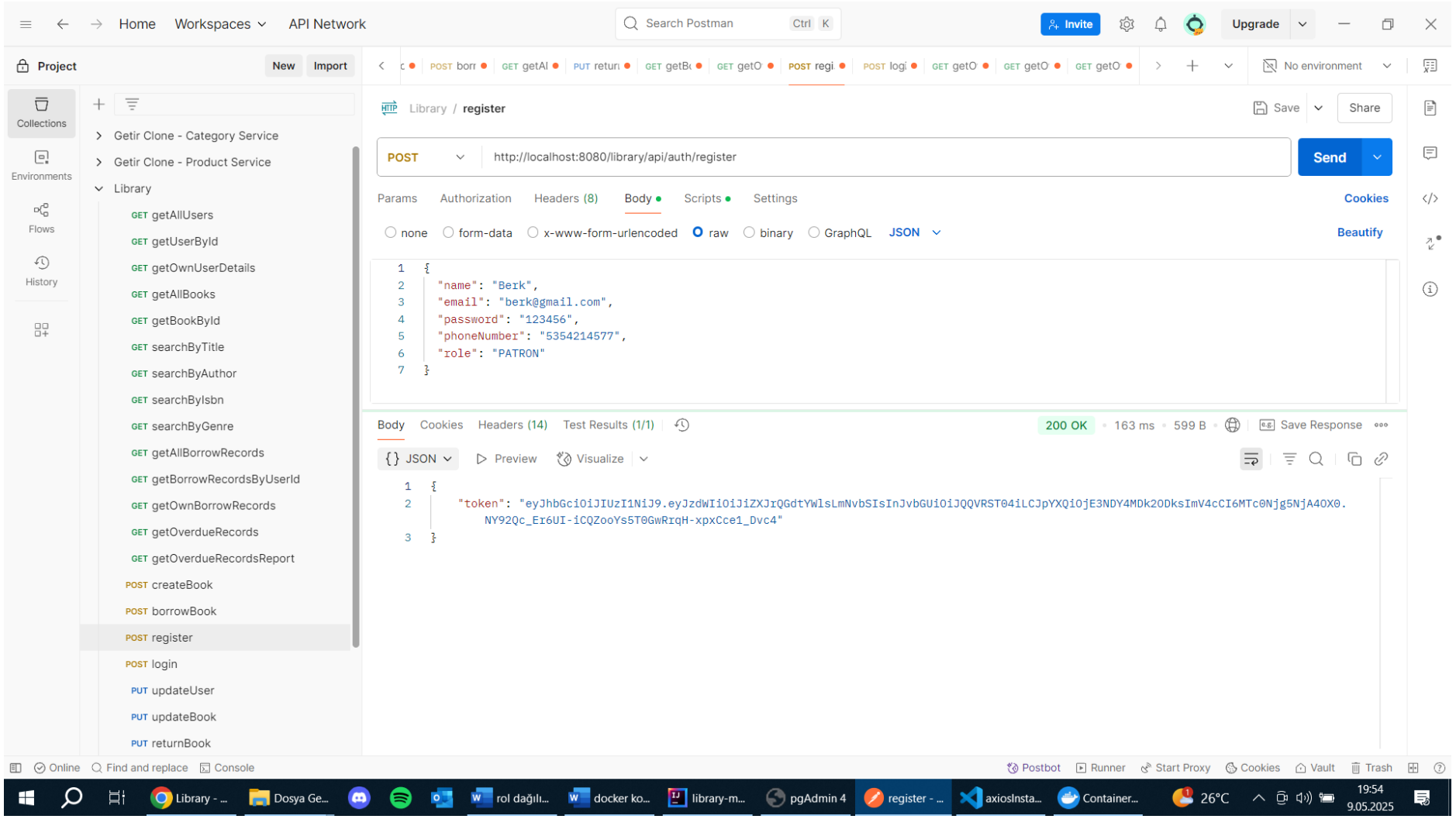
Database Monitoring – pgAdmin4



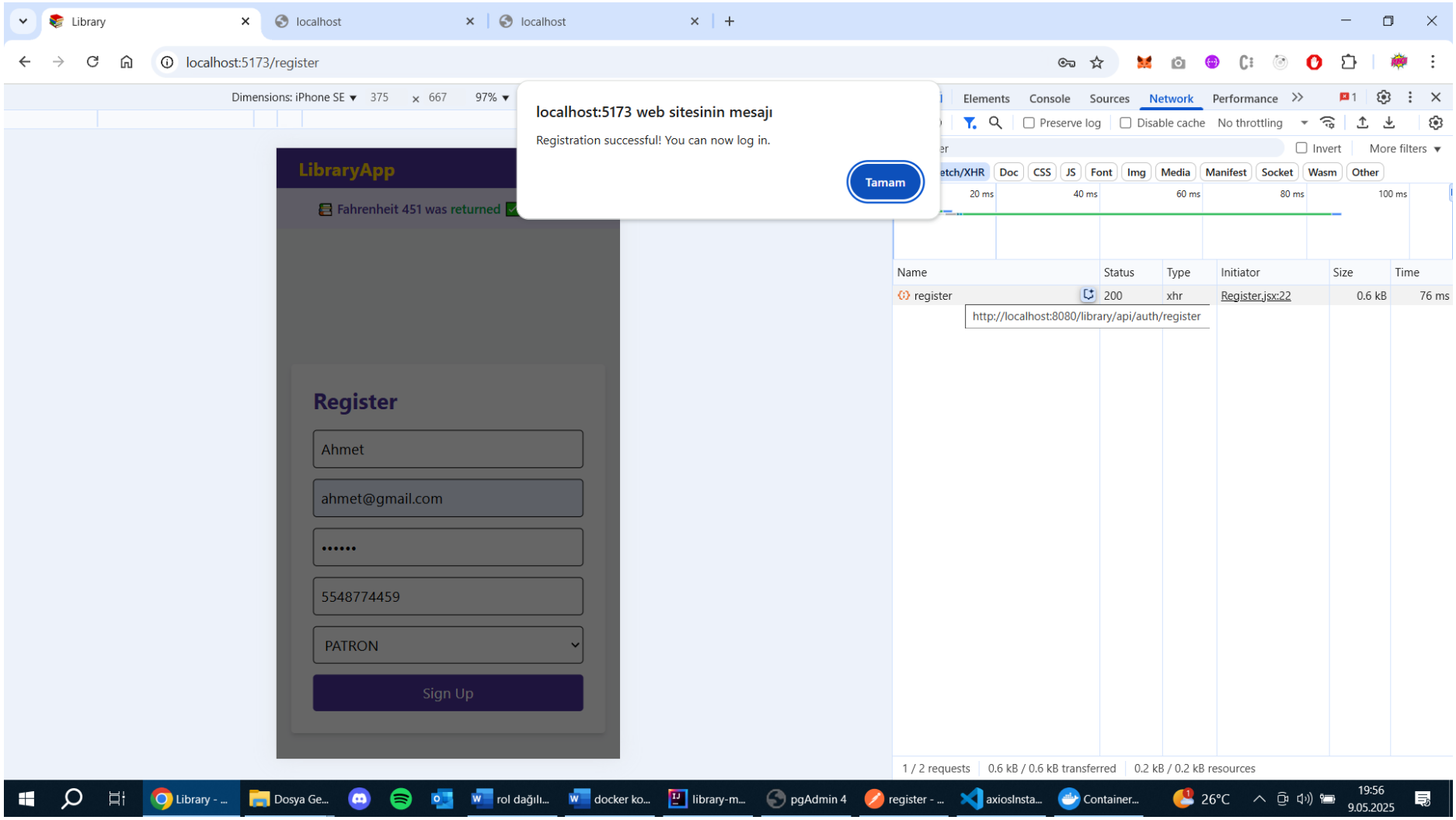
Frontend View – Live Application (React UI)



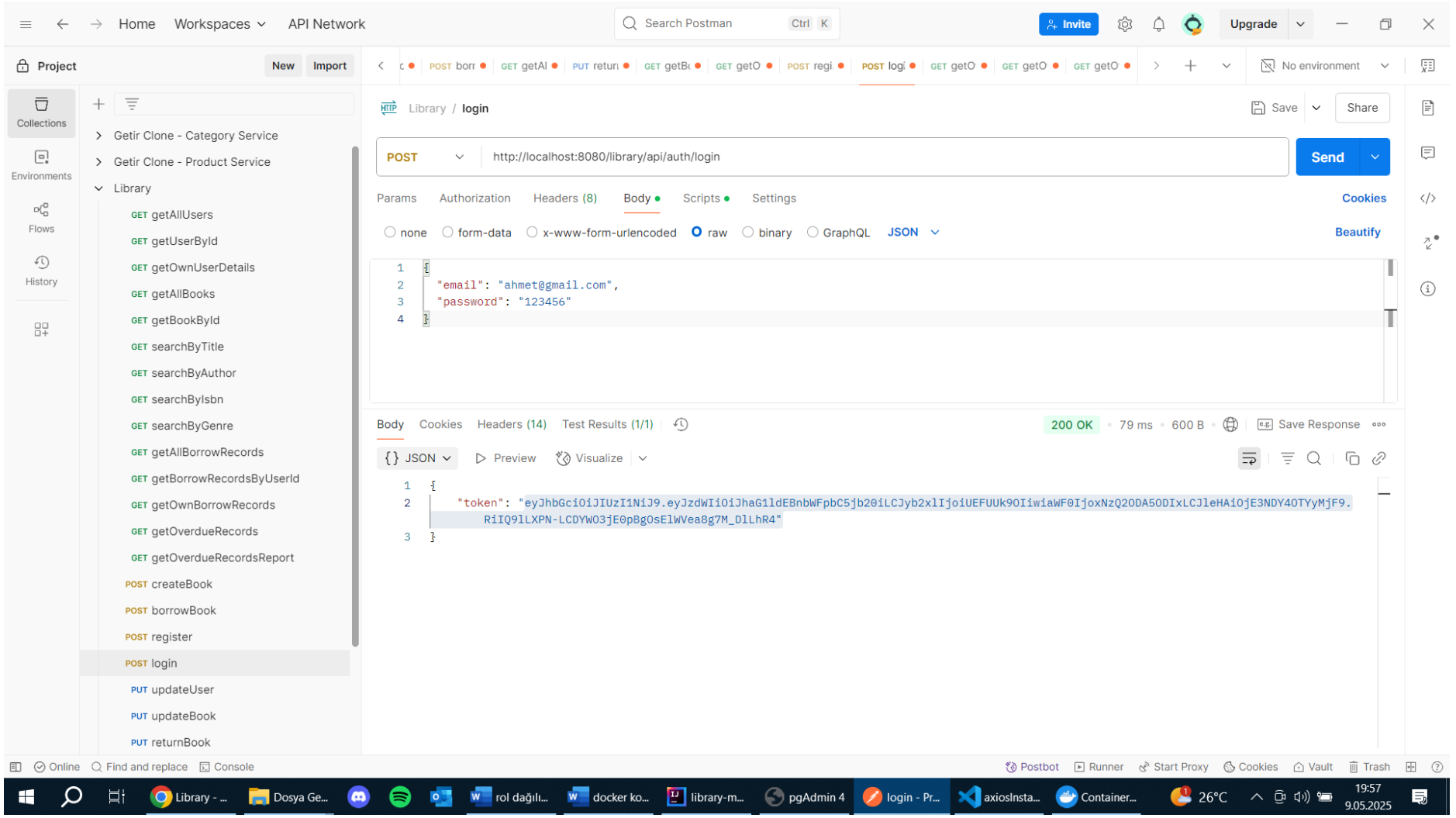
Register (Postman) – with PATRON



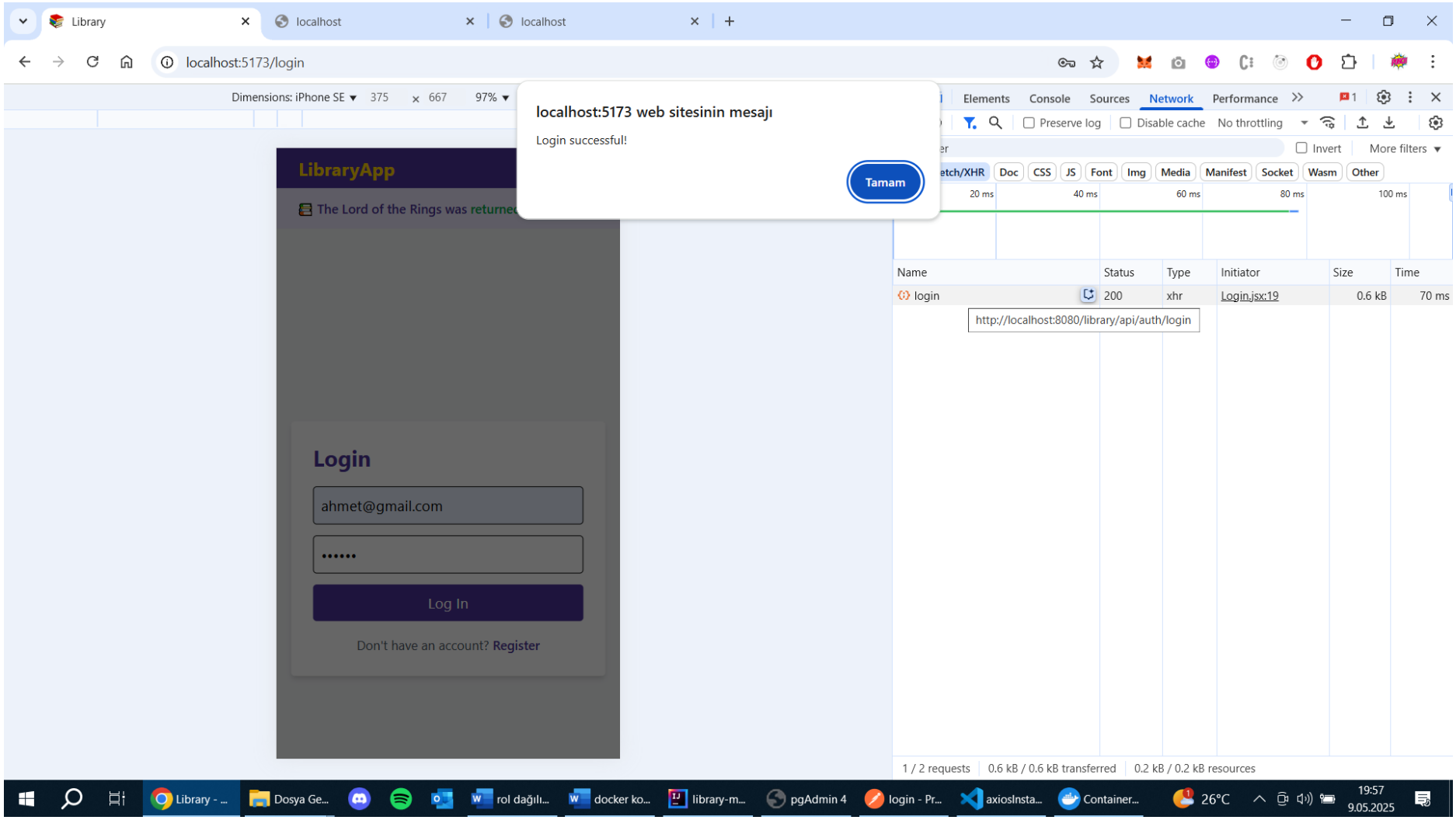
Register (Client View)



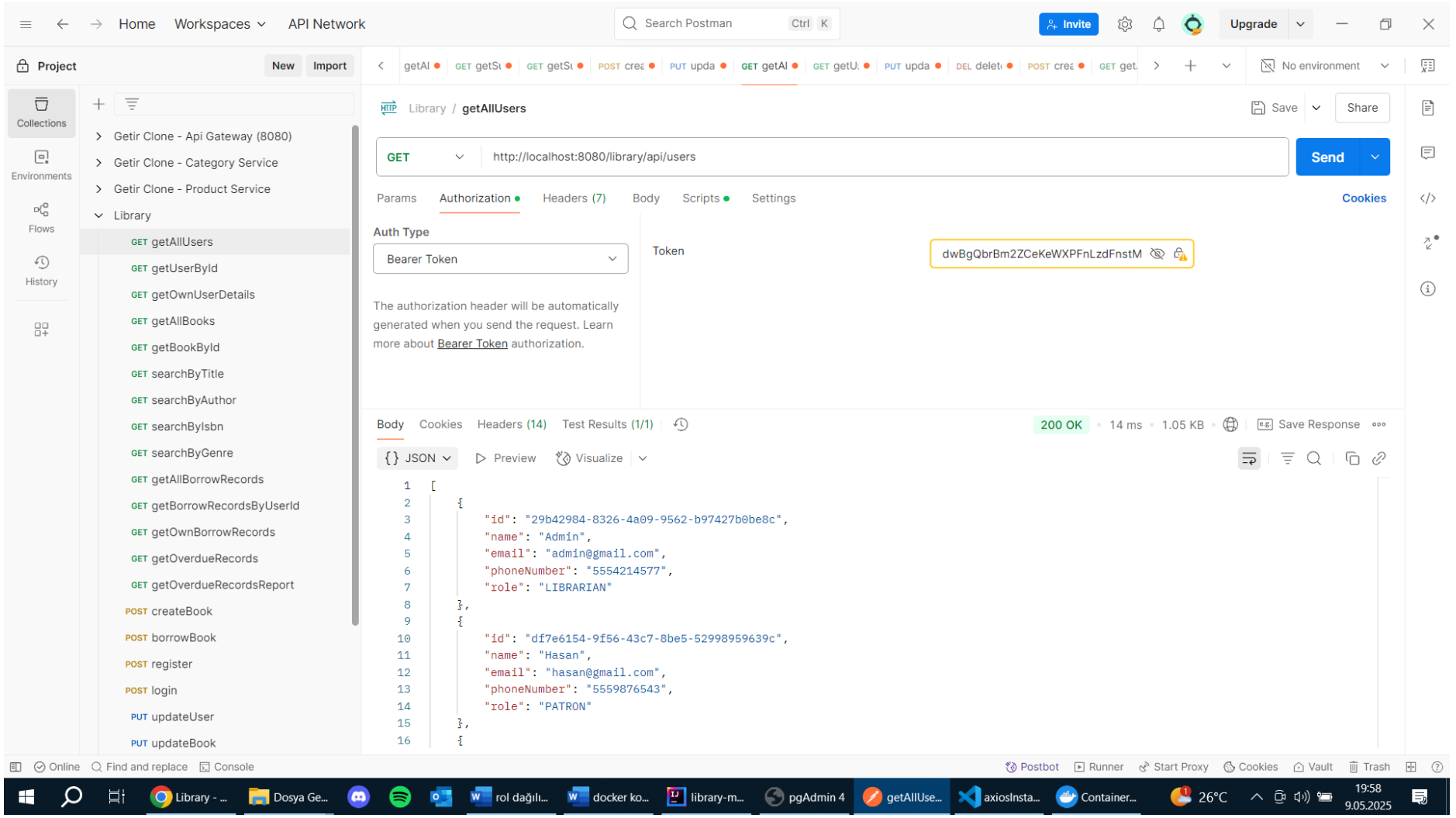
Login (Postman) - with PATRON



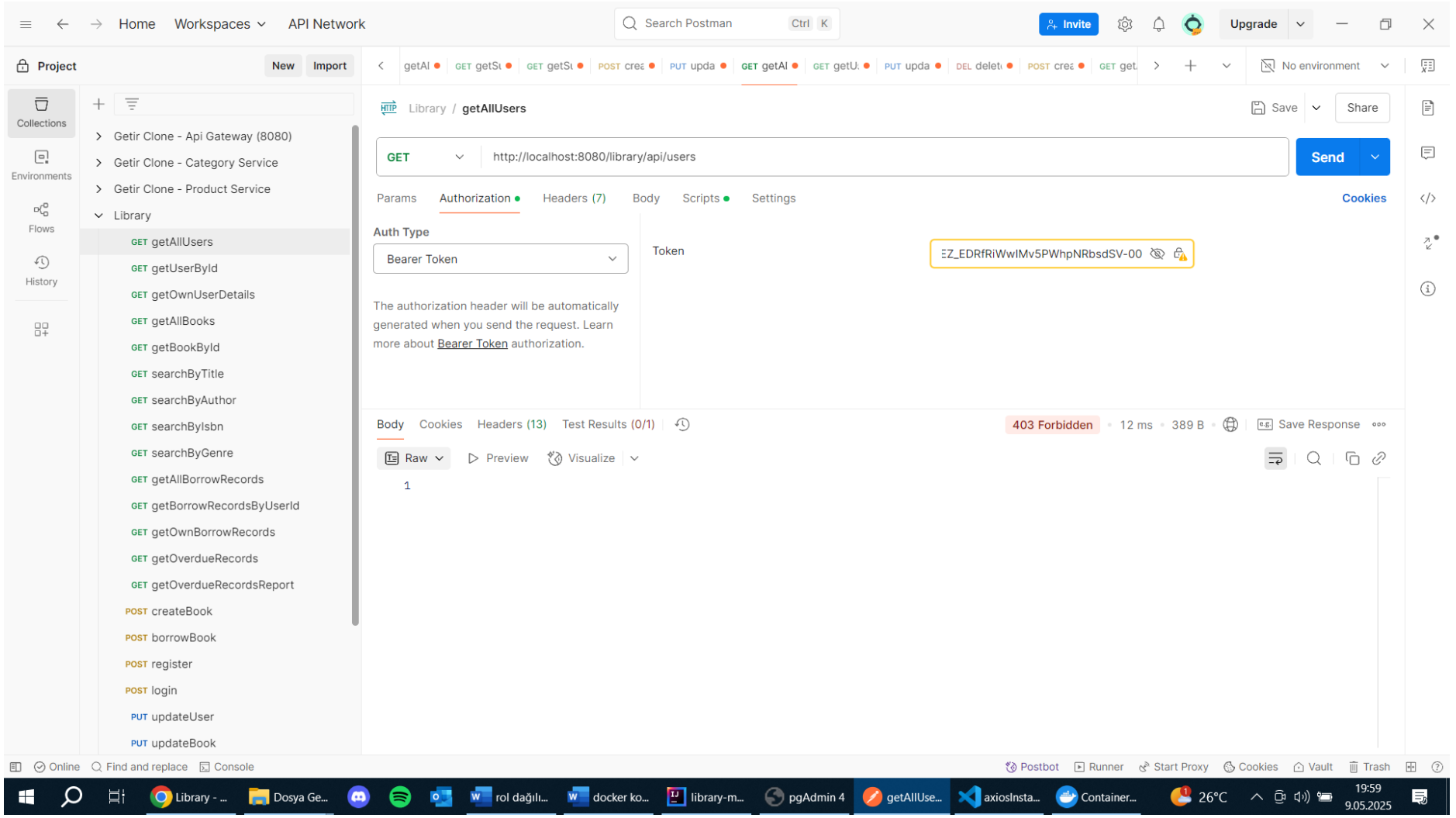
Login (Client View)



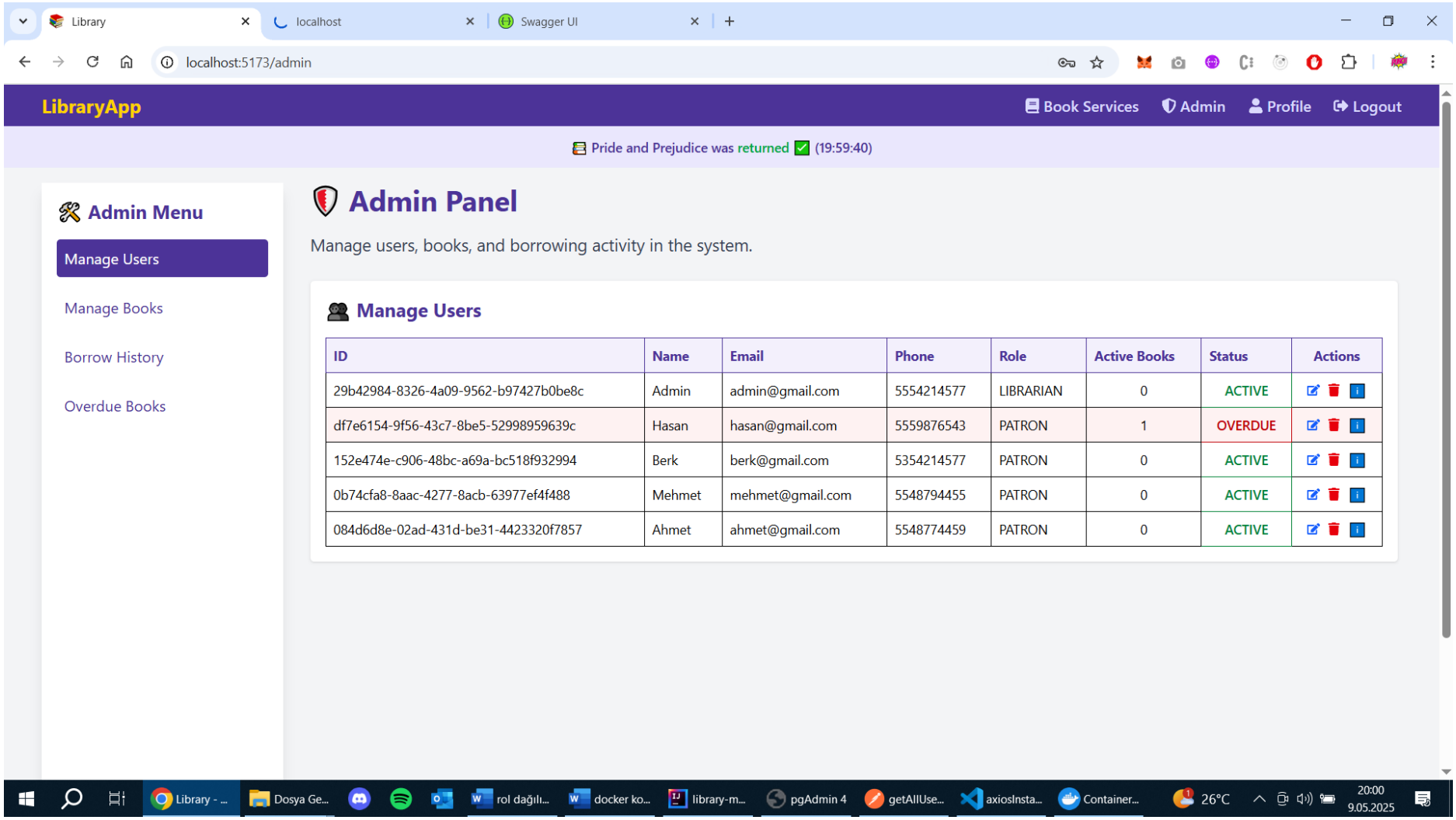
GET /users with LIBRARIAN Token – (200 OK)



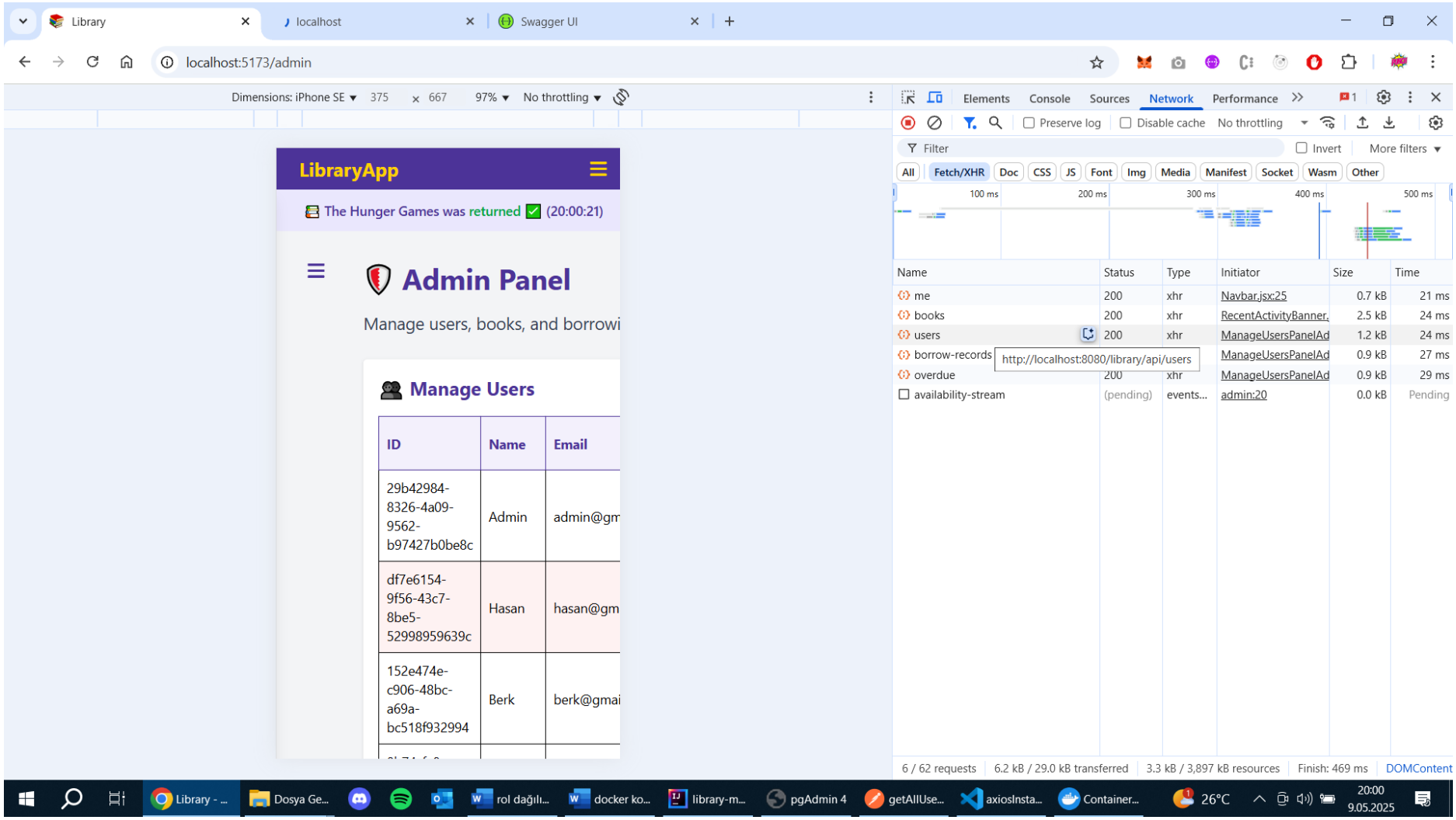
GET /users with PATRON Token – (403 Forbidden)



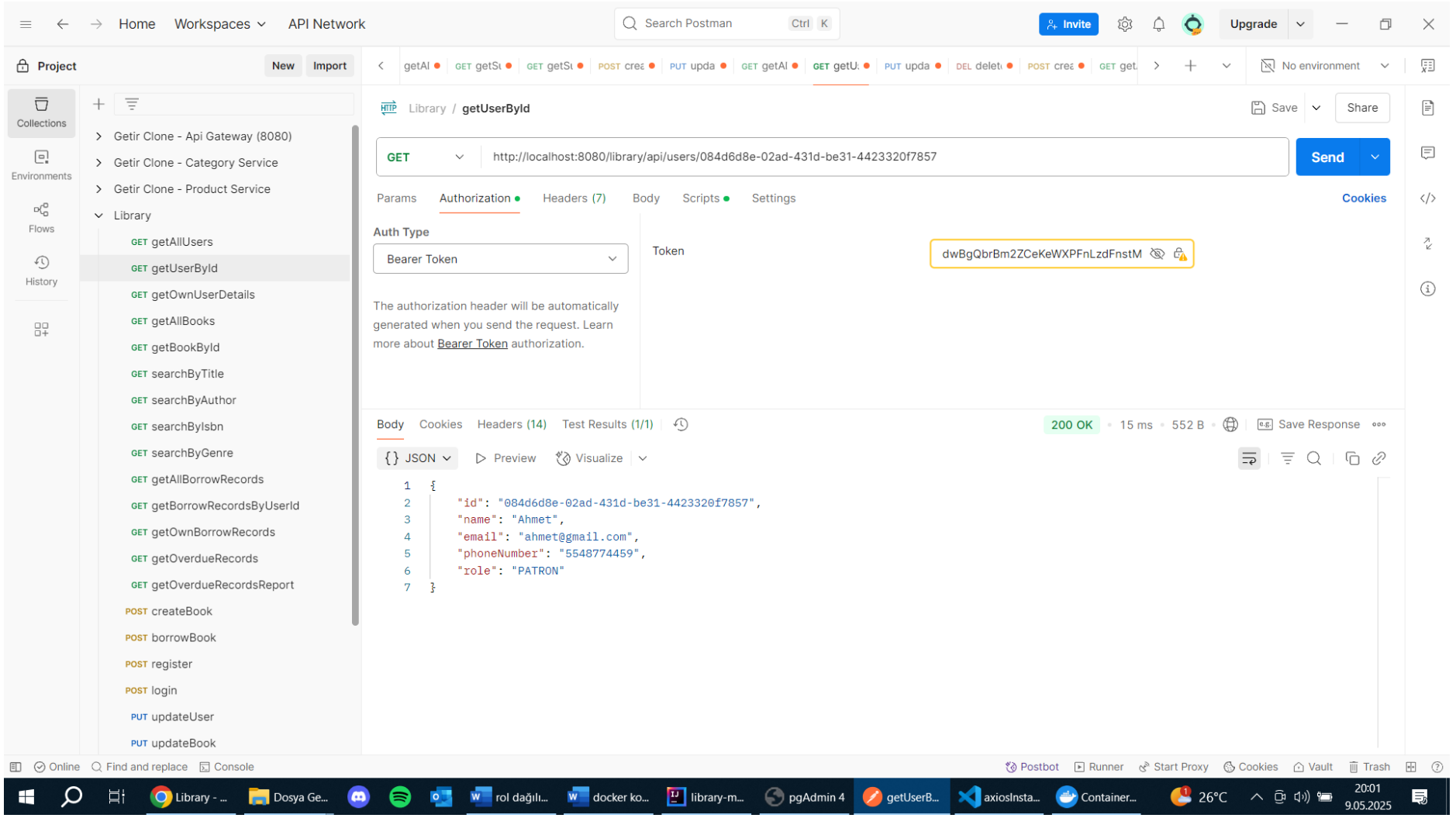
GET /users – Admin Panel View



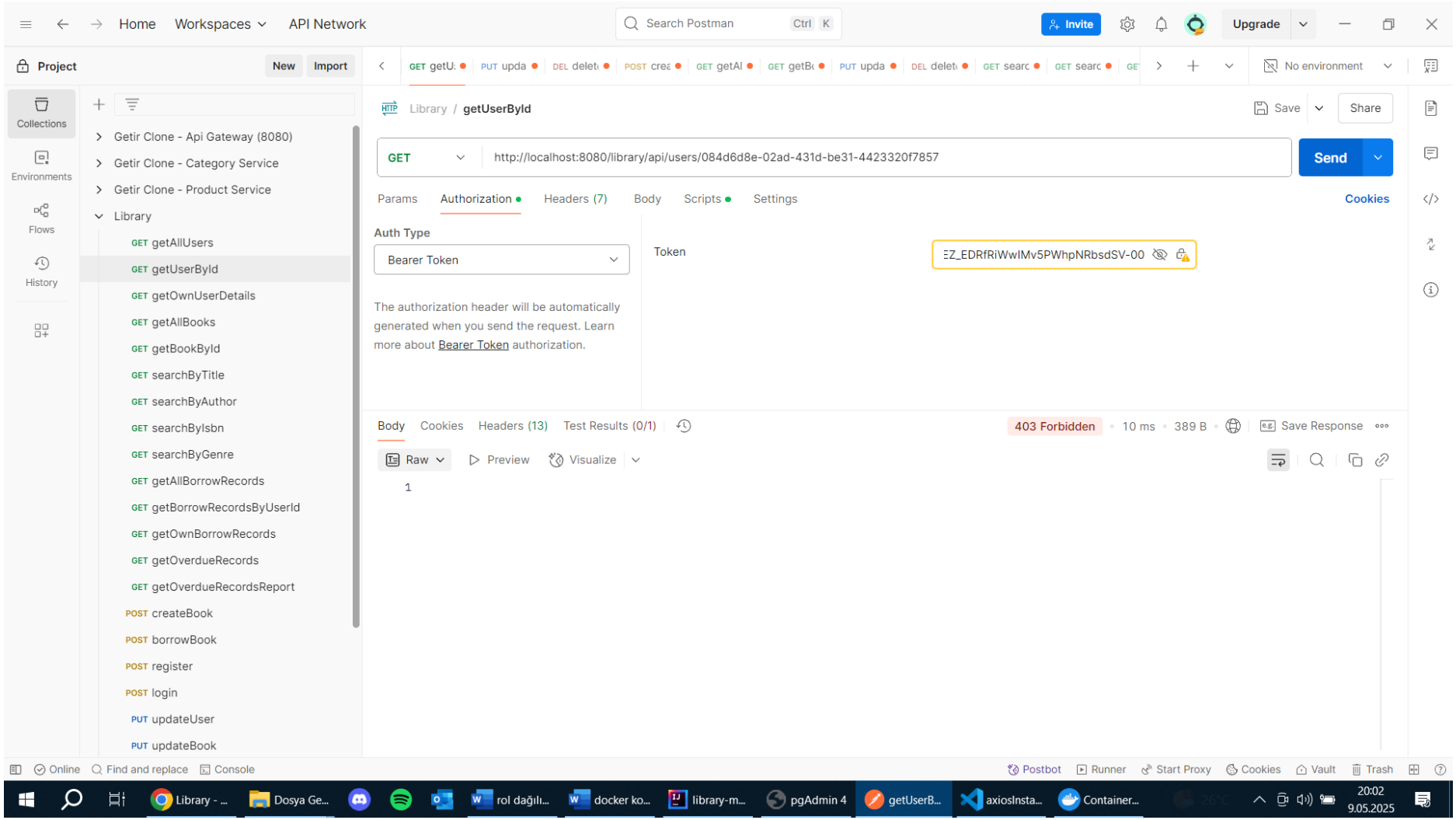
GET /users – Admin Panel View 2



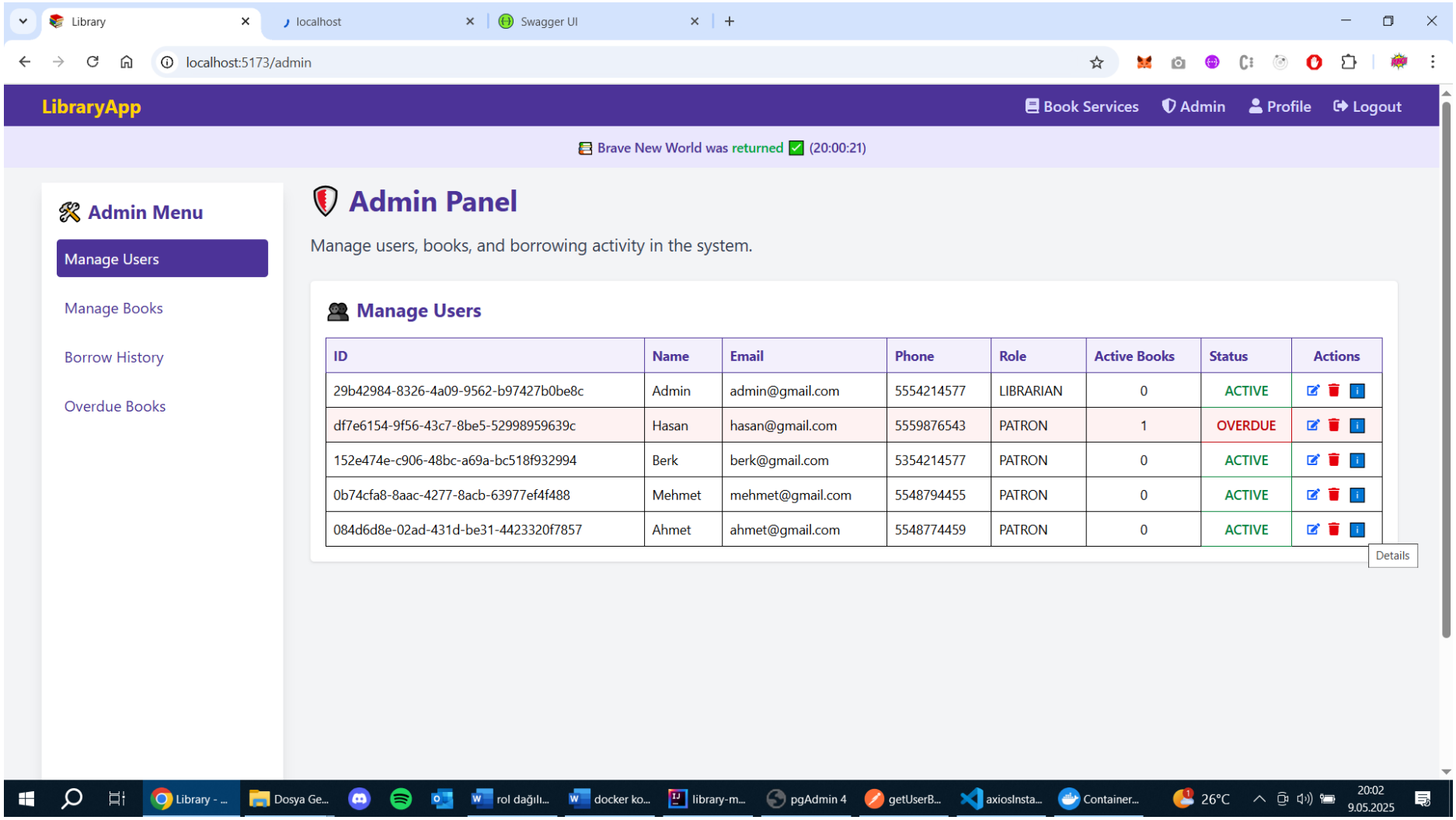
GET /users/{id} with LIBRARIAN Token – (200 OK)



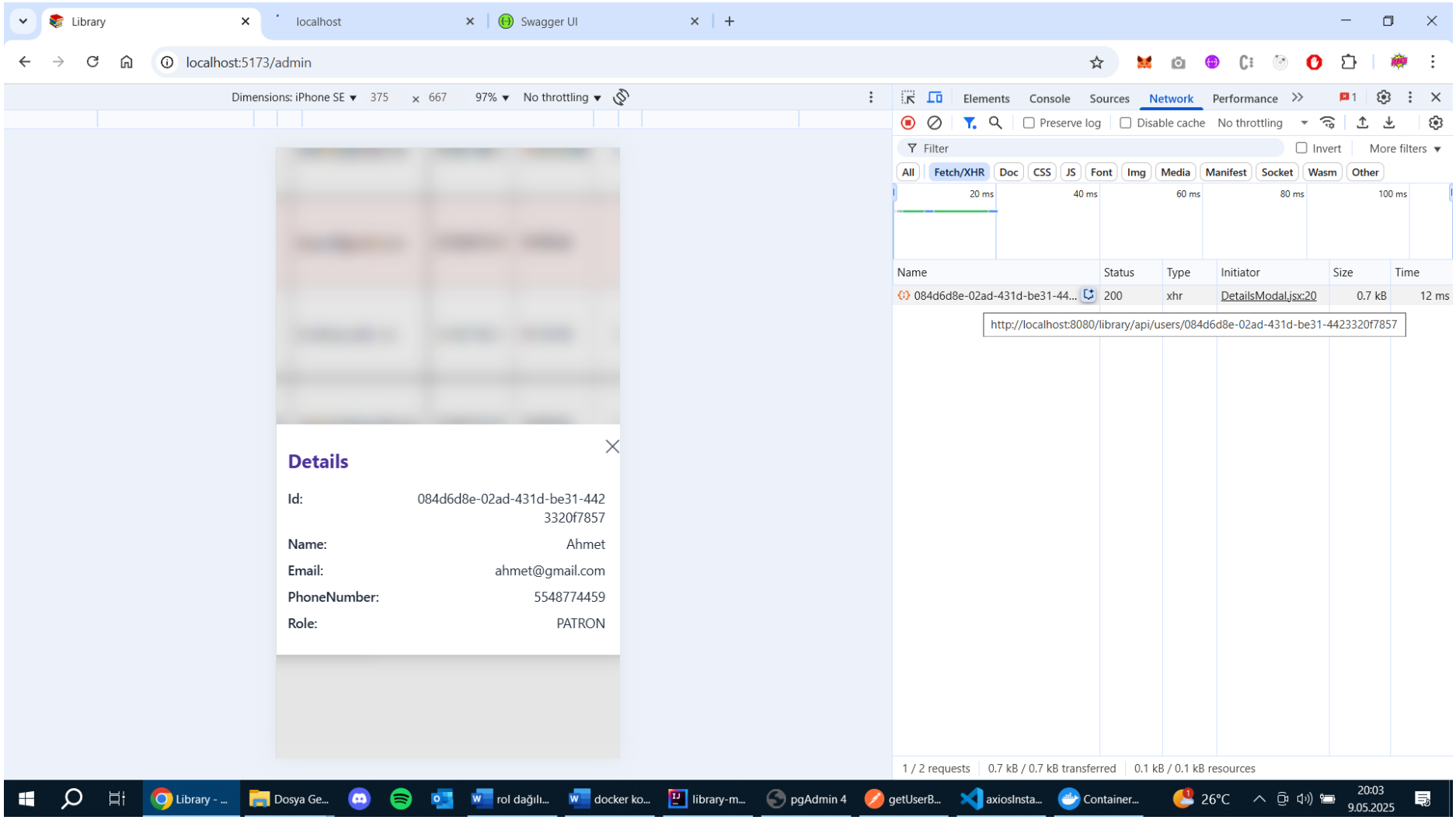
GET /users/{id} with PATRON Token – (403 Forbidden)



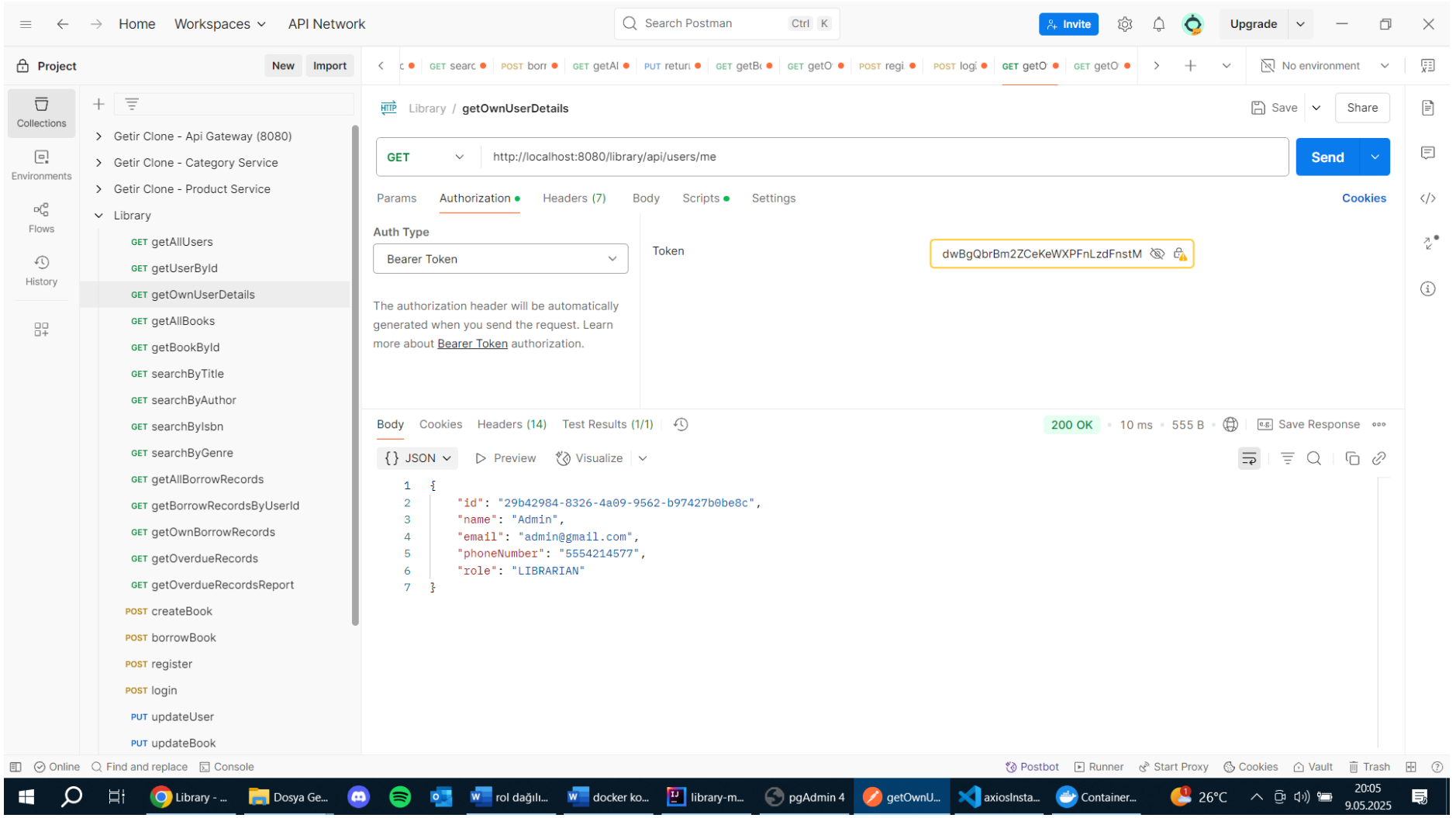
GET /users/{id} – Admin Panel View



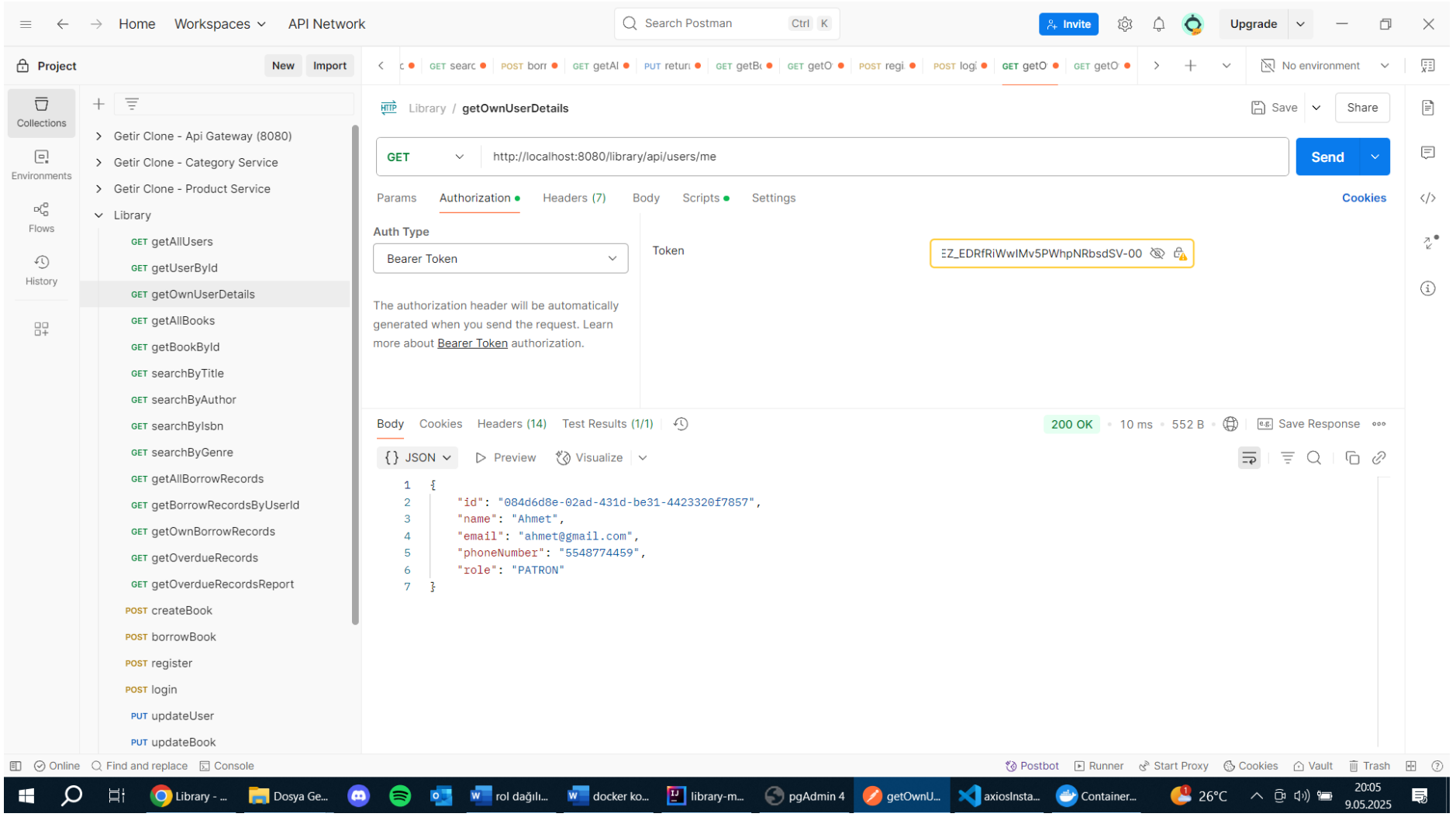
GET /users/{id} – Admin Panel View 2



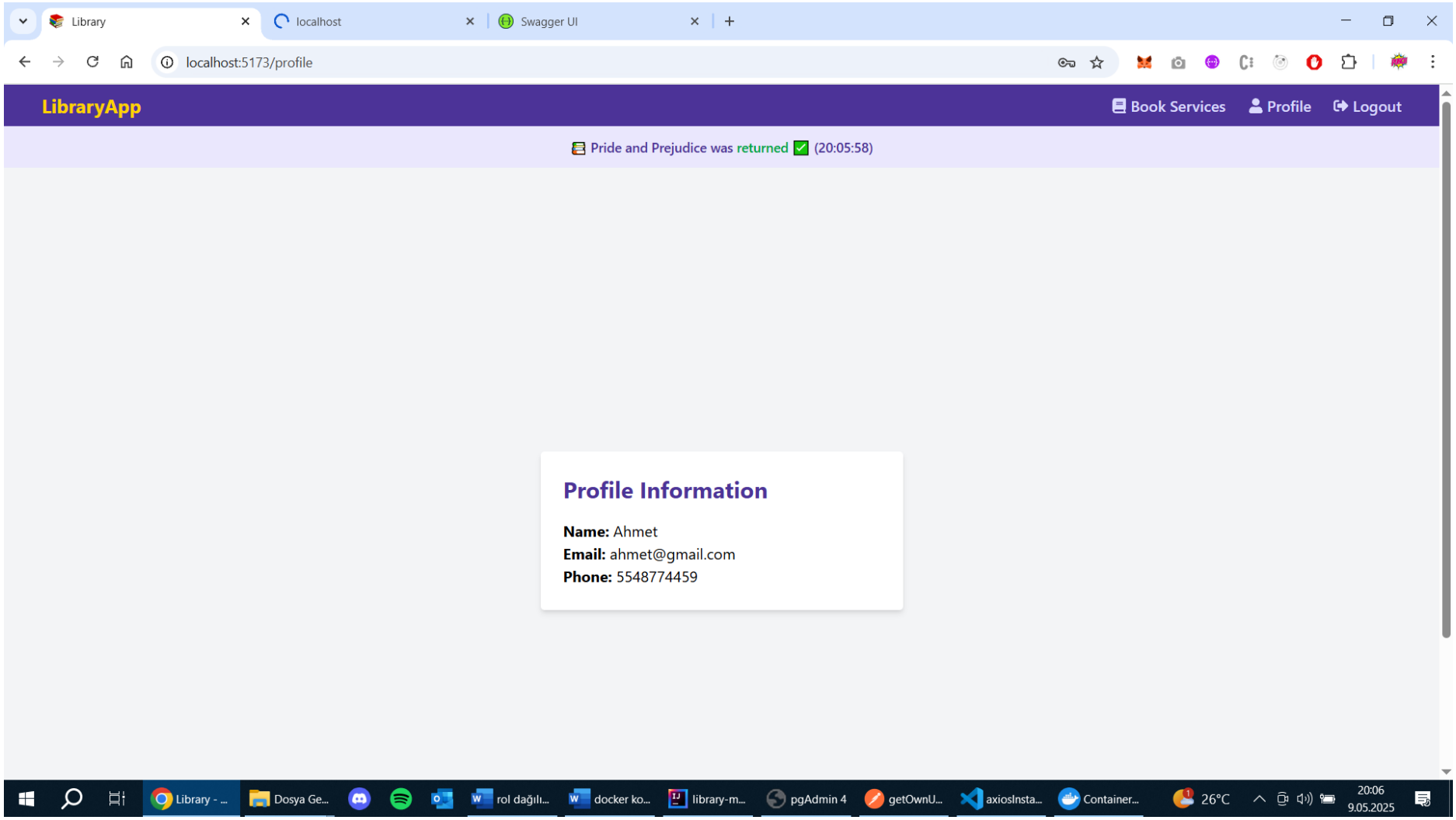
GET /users/me with LIBRARIAN Token – (200 OK)



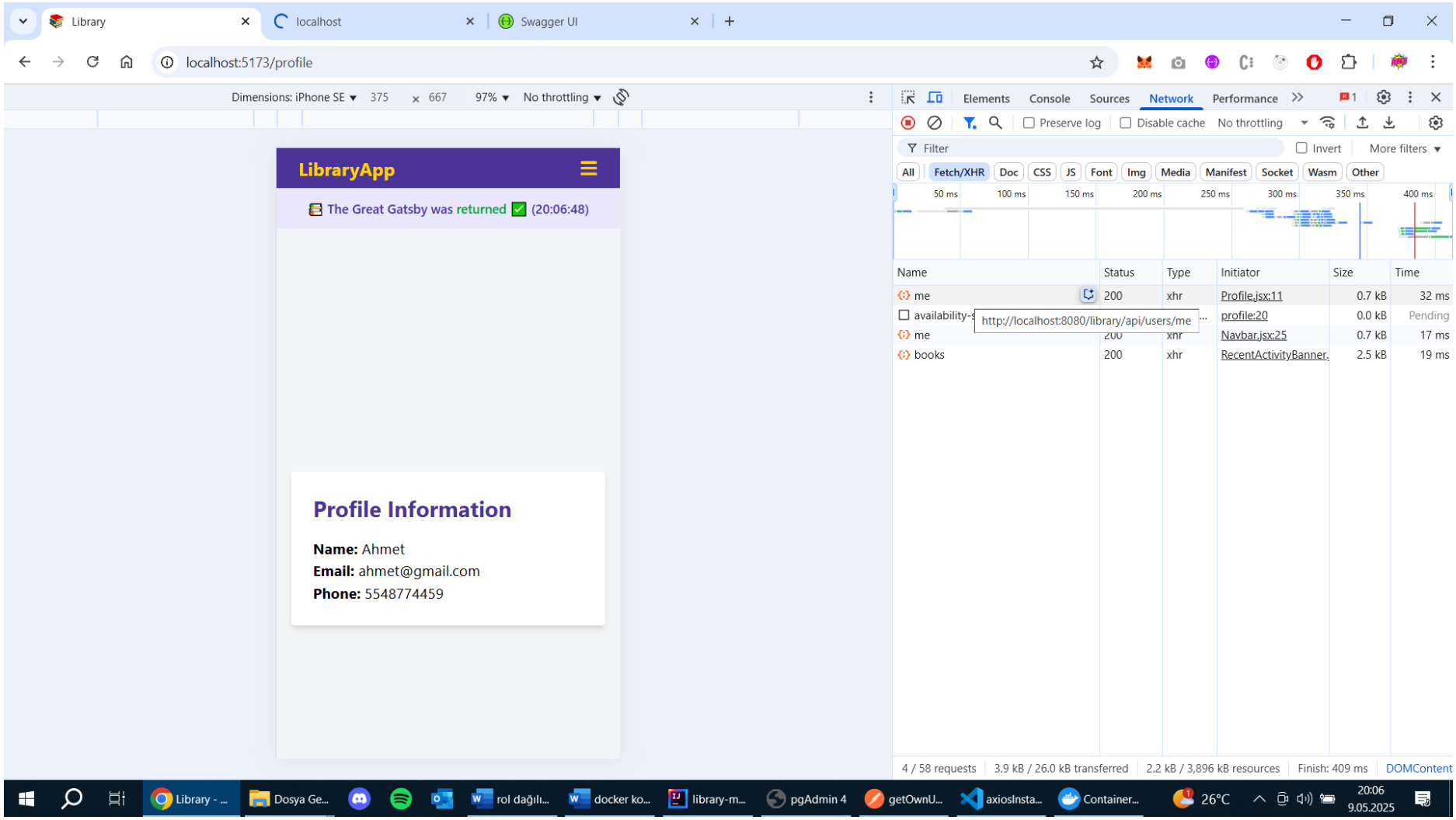
GET /users/me with PATRON Token – (200 OK)



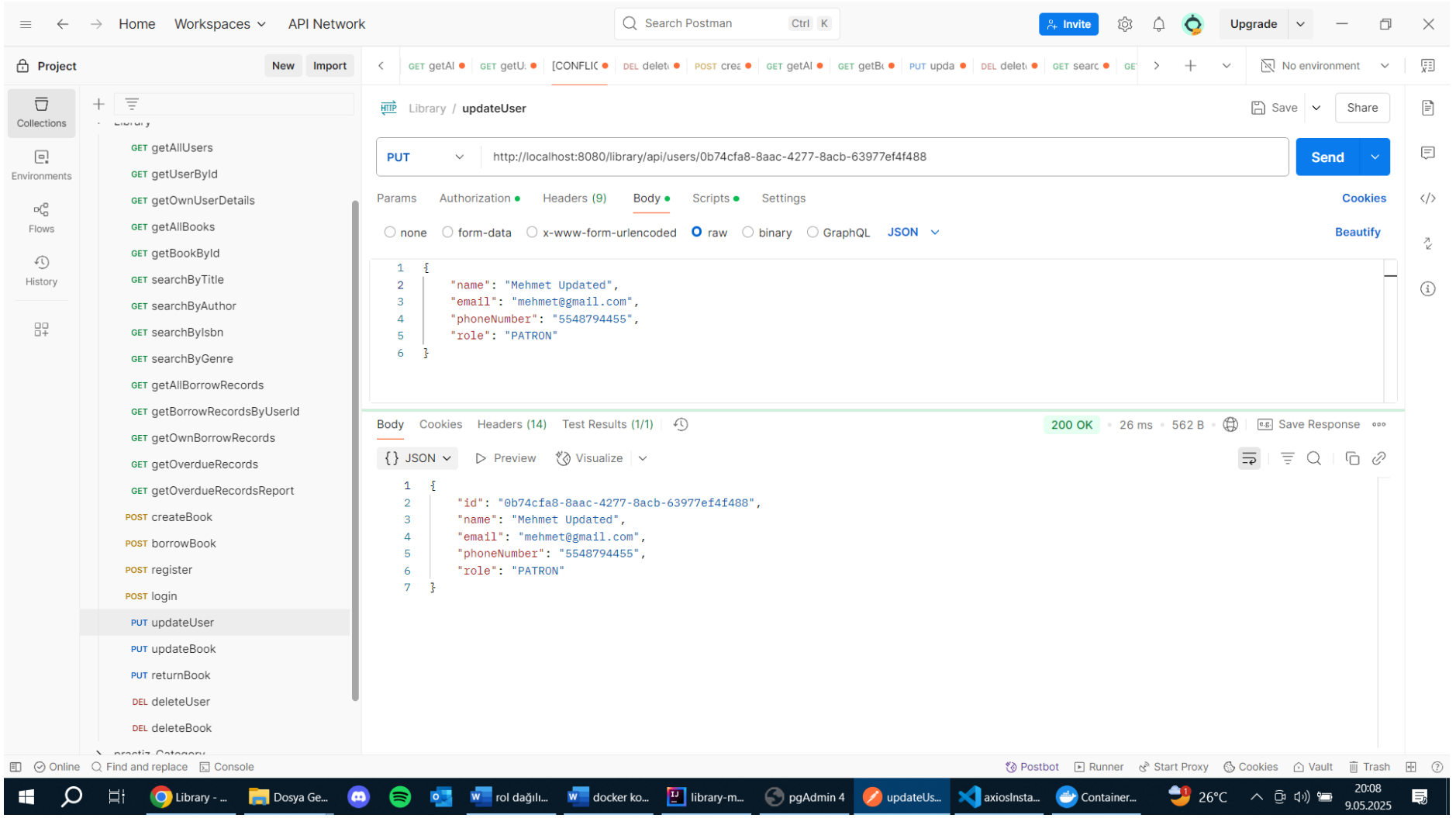
GET /users/me – Profile Page



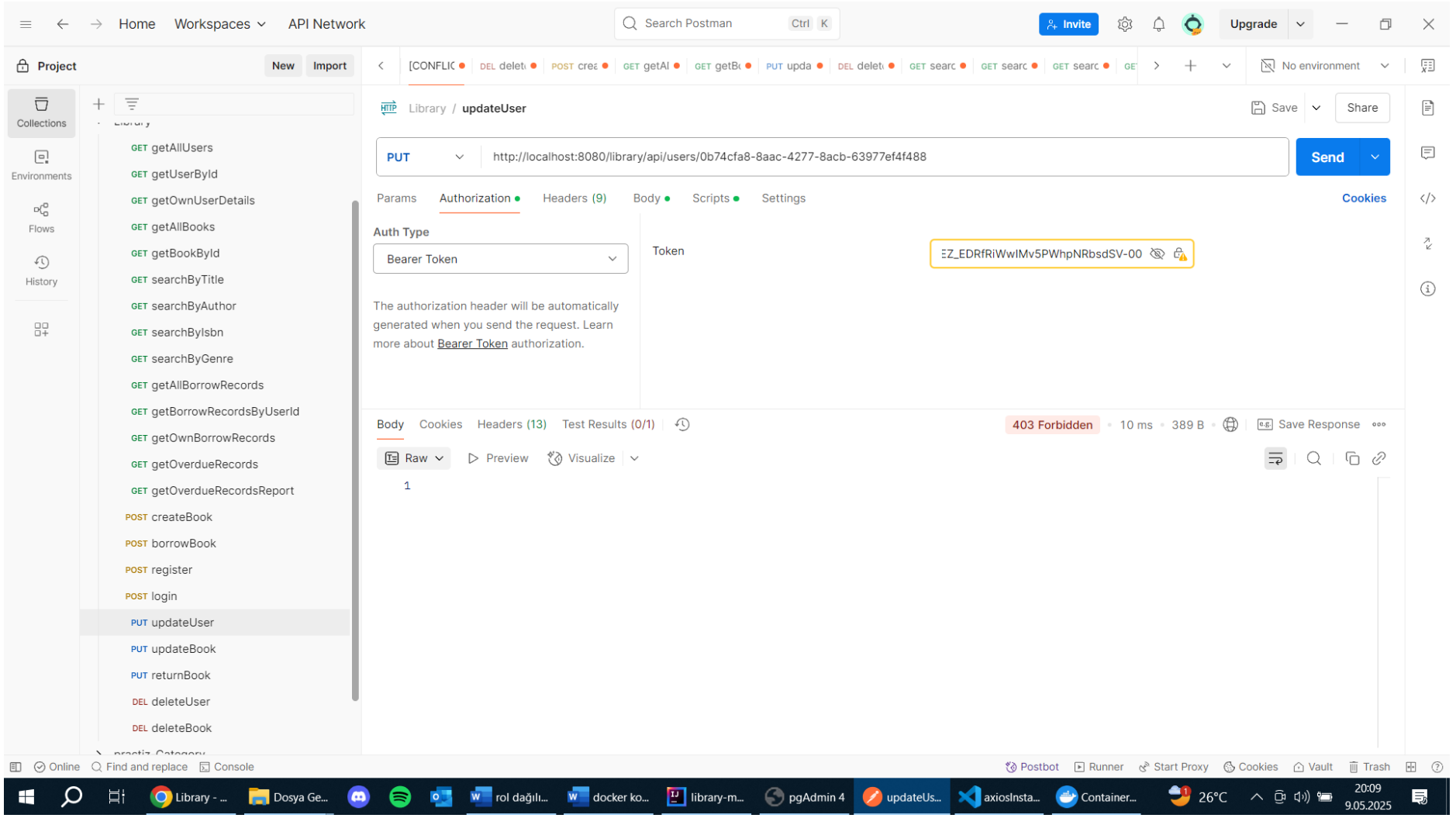
GET /users/me – Profile Page 2



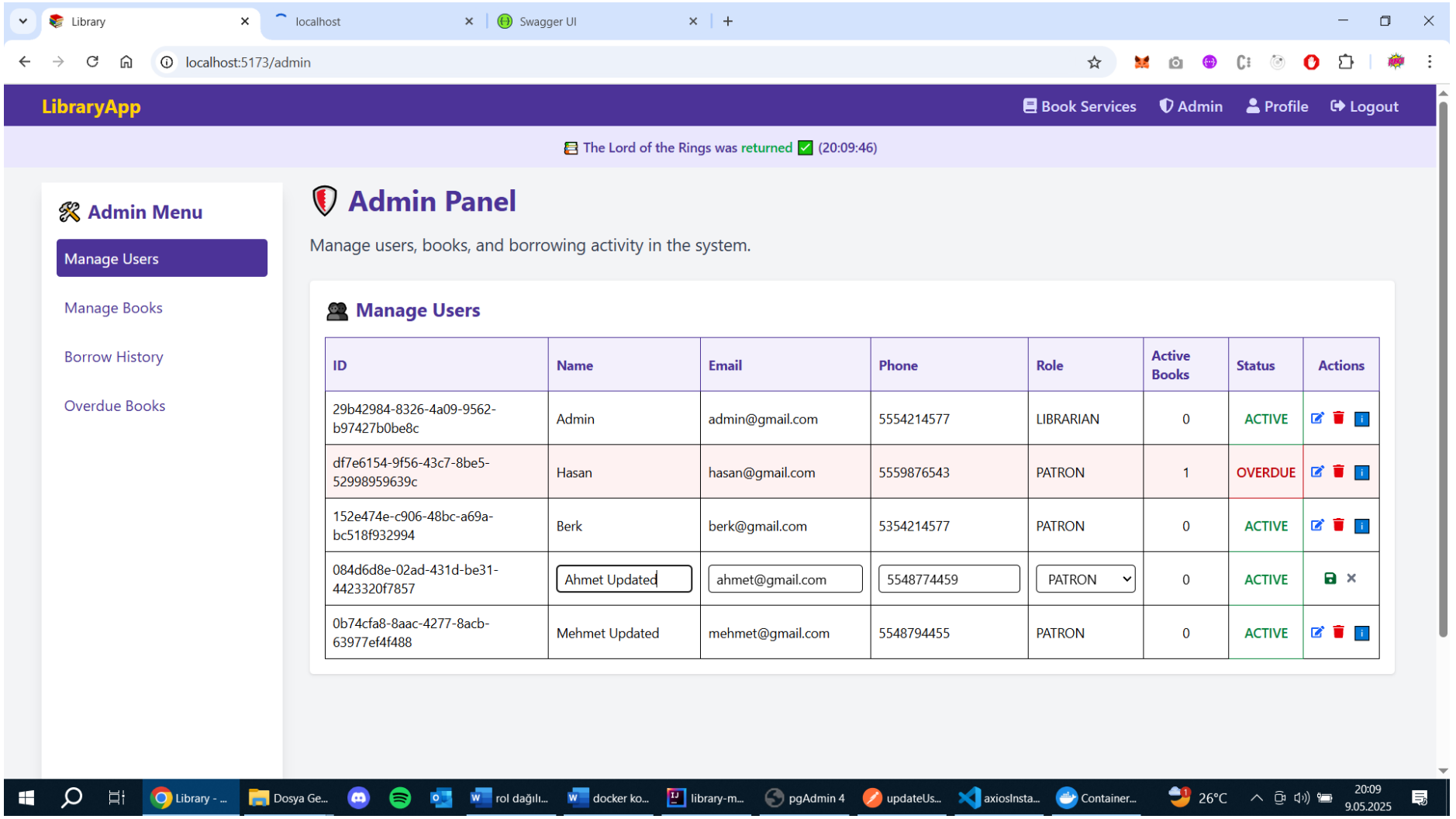
PUT /users/{id} with LIBRARIAN Token – (200 OK)



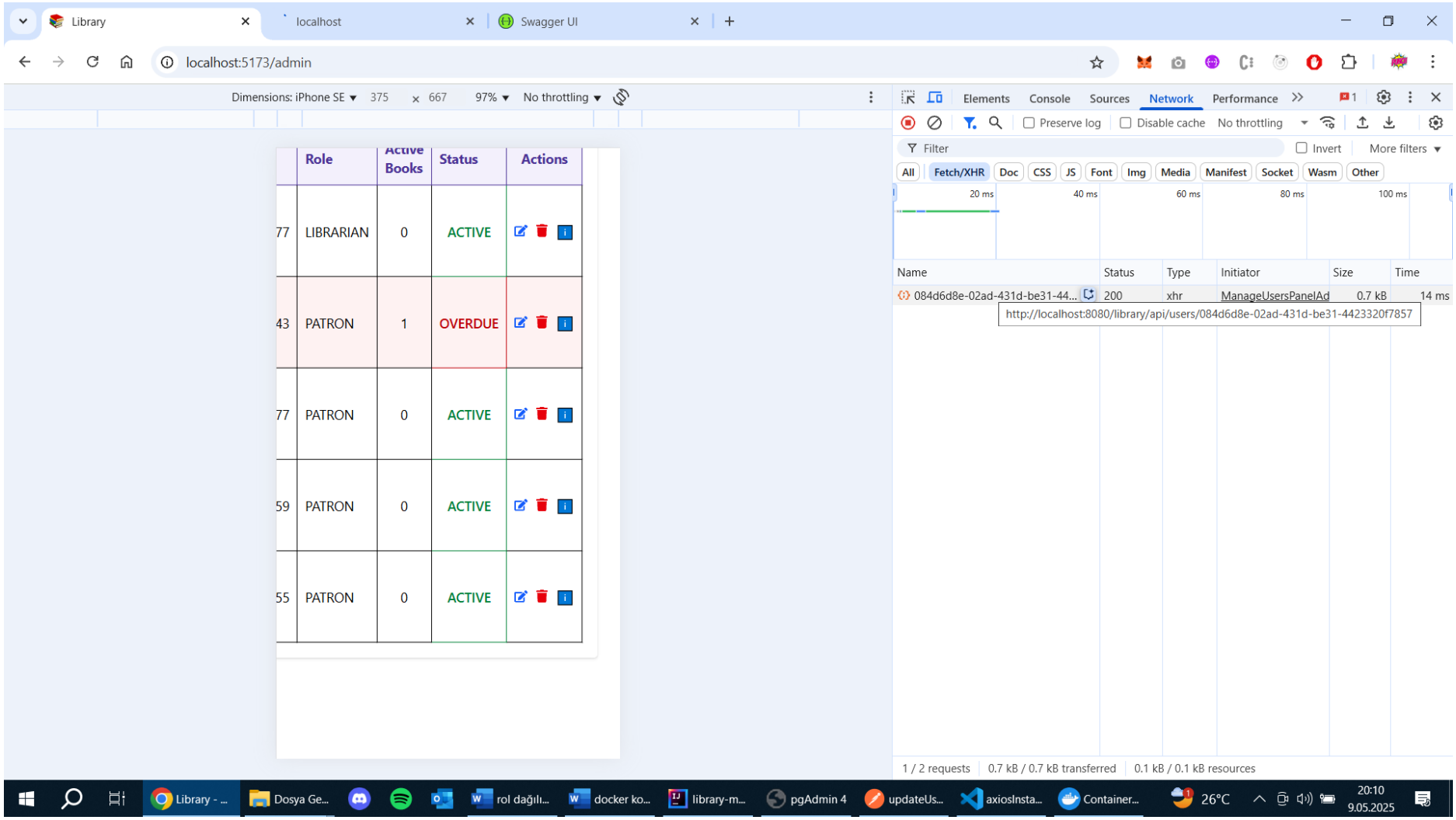
PUT /users/{id} with PATRON Token – (403 Forbidden)



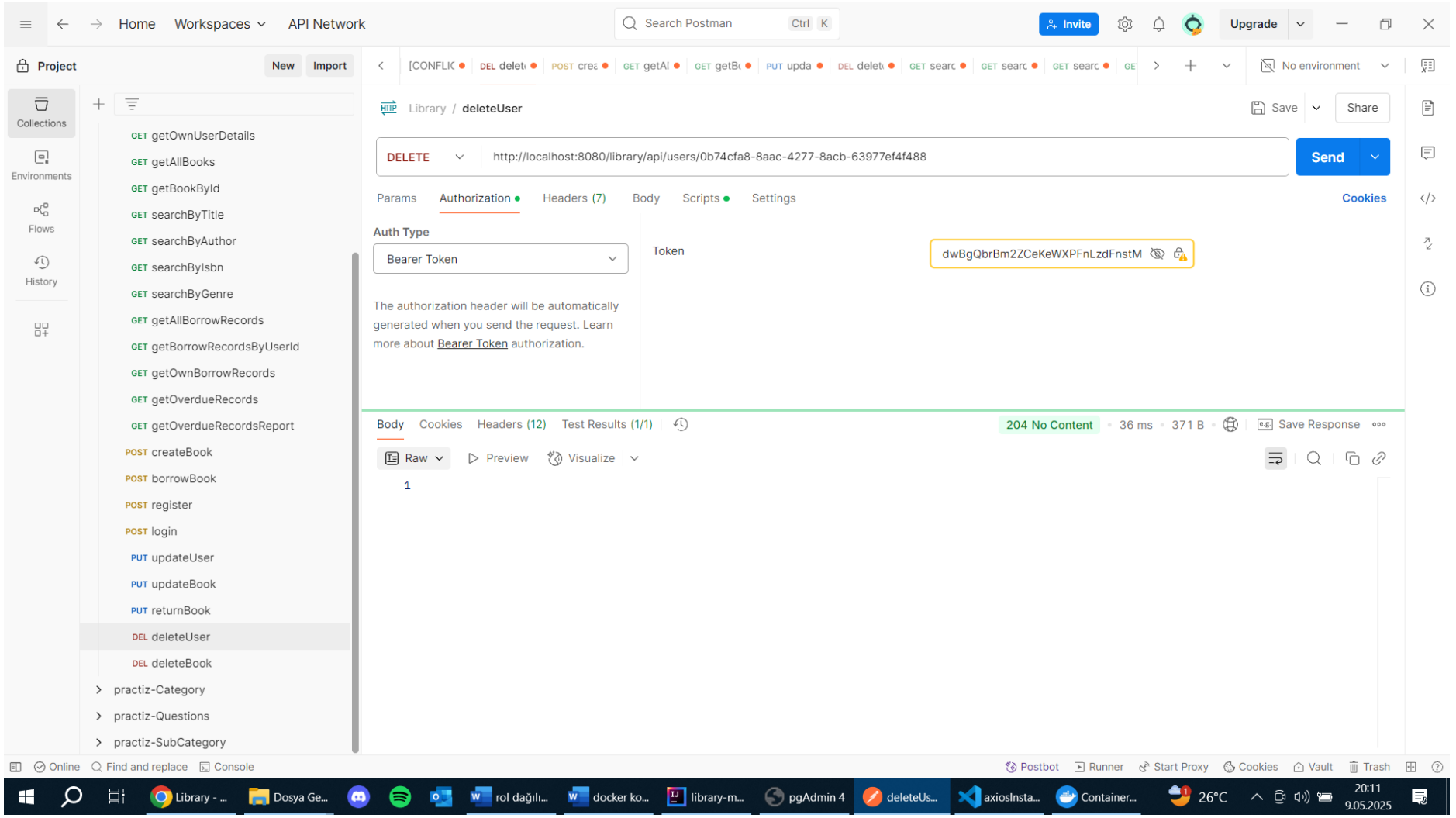
PUT /users/{id} – Admin Panel View



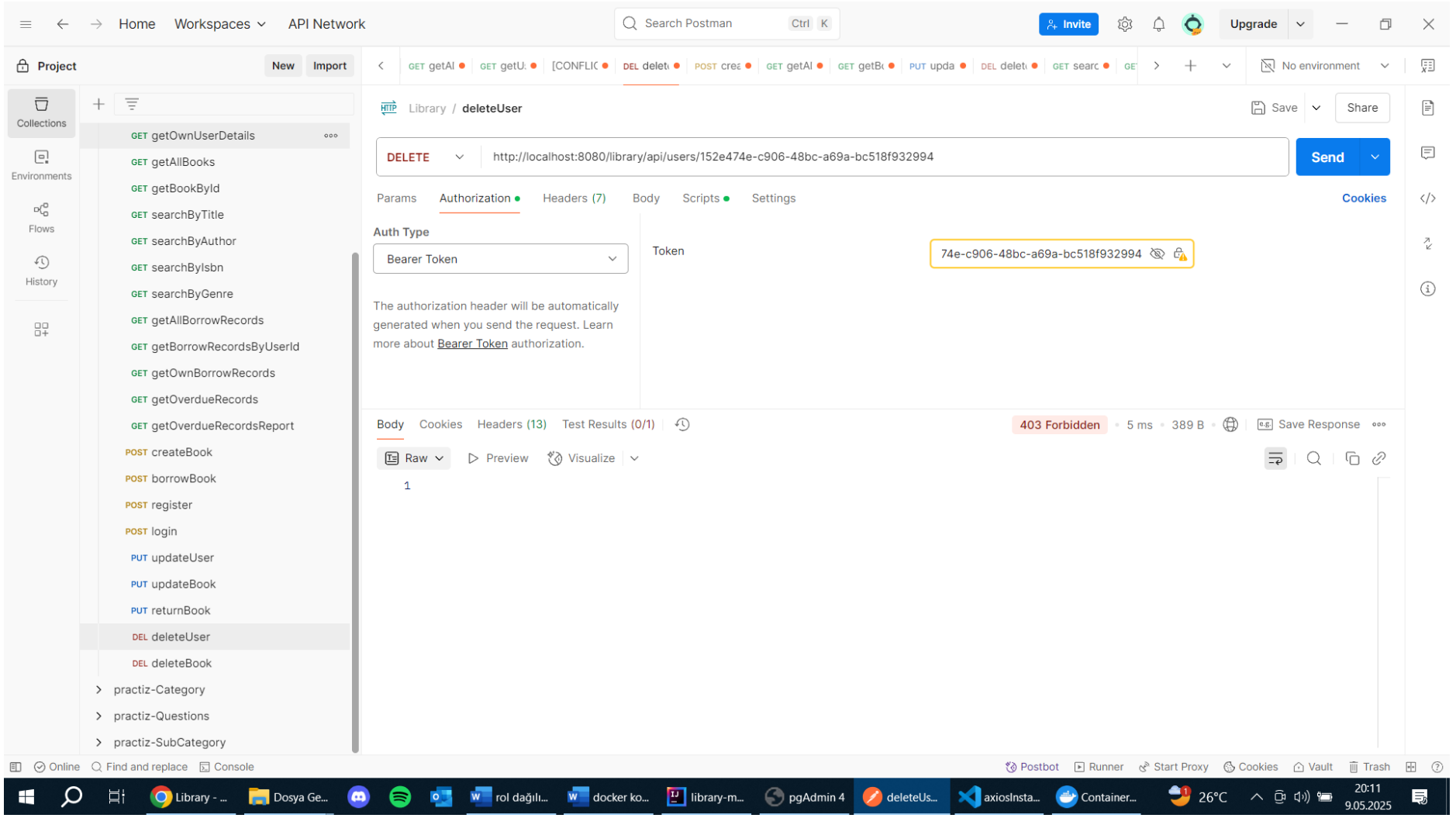
PUT /users/{id} – Admin Panel View 2



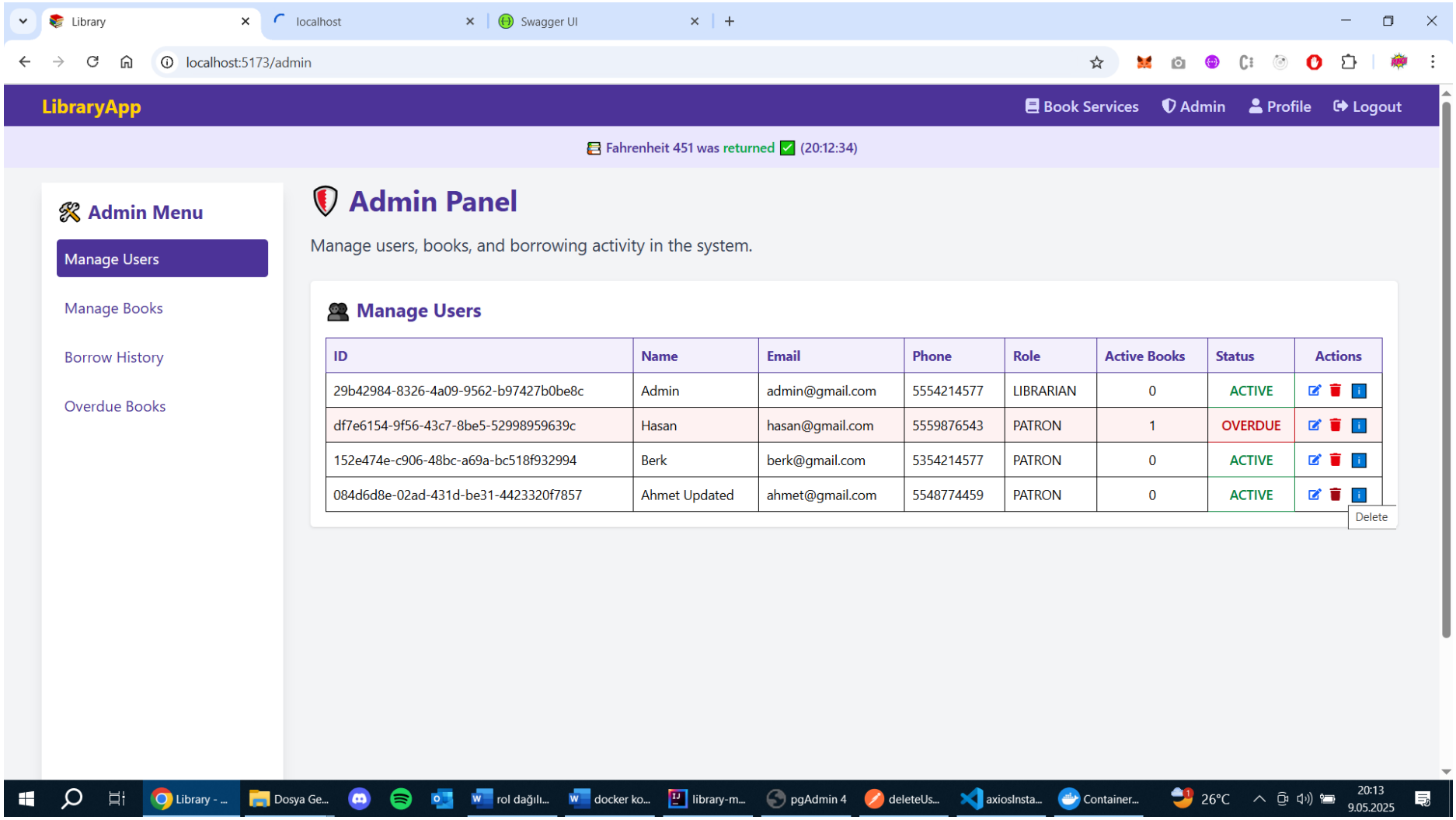
DELETE /users/{id} with LIBRARIAN Token – (200 OK)



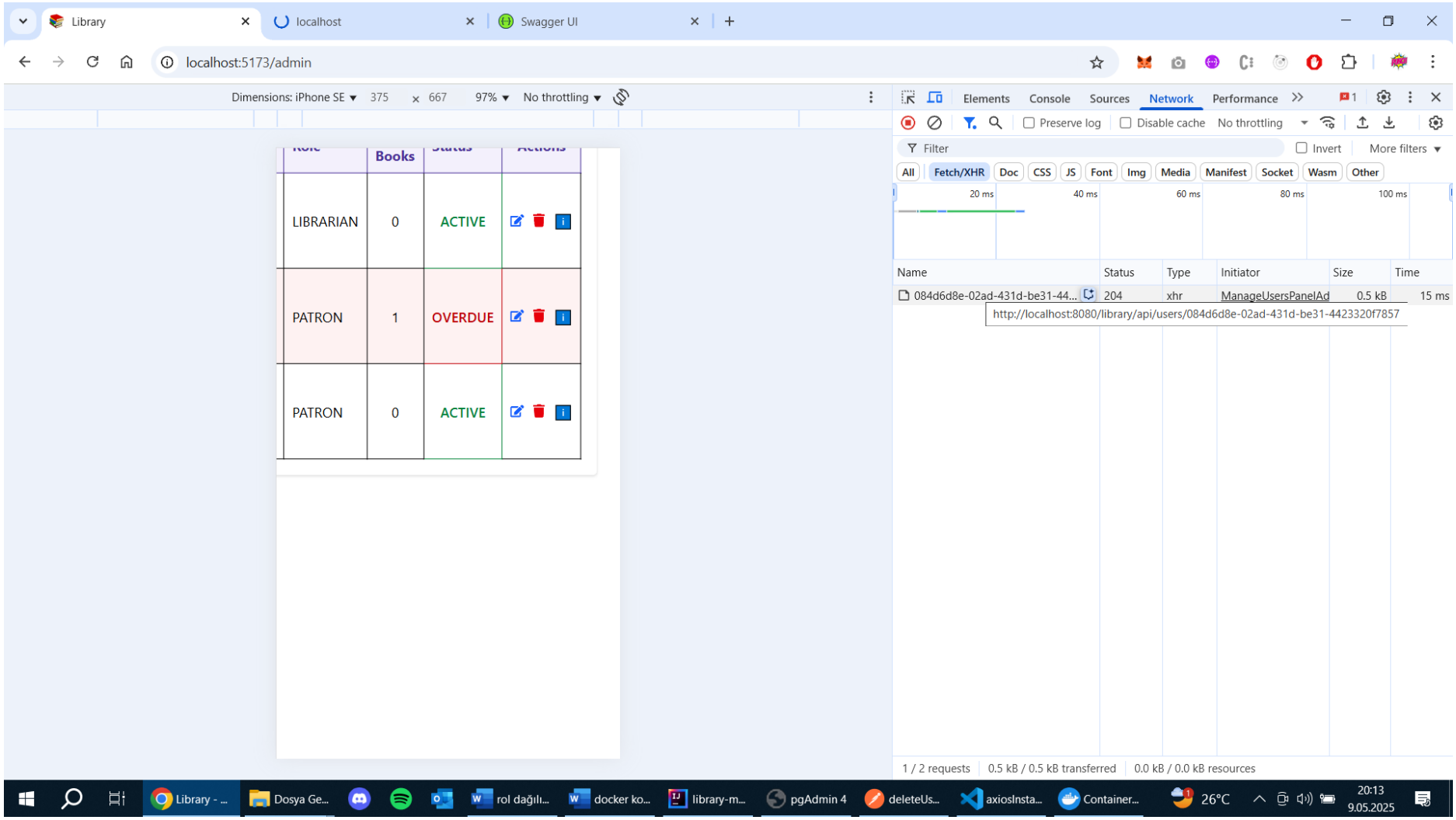
DELETE /users/{id} with PATRON Token – (403 Forbidden)



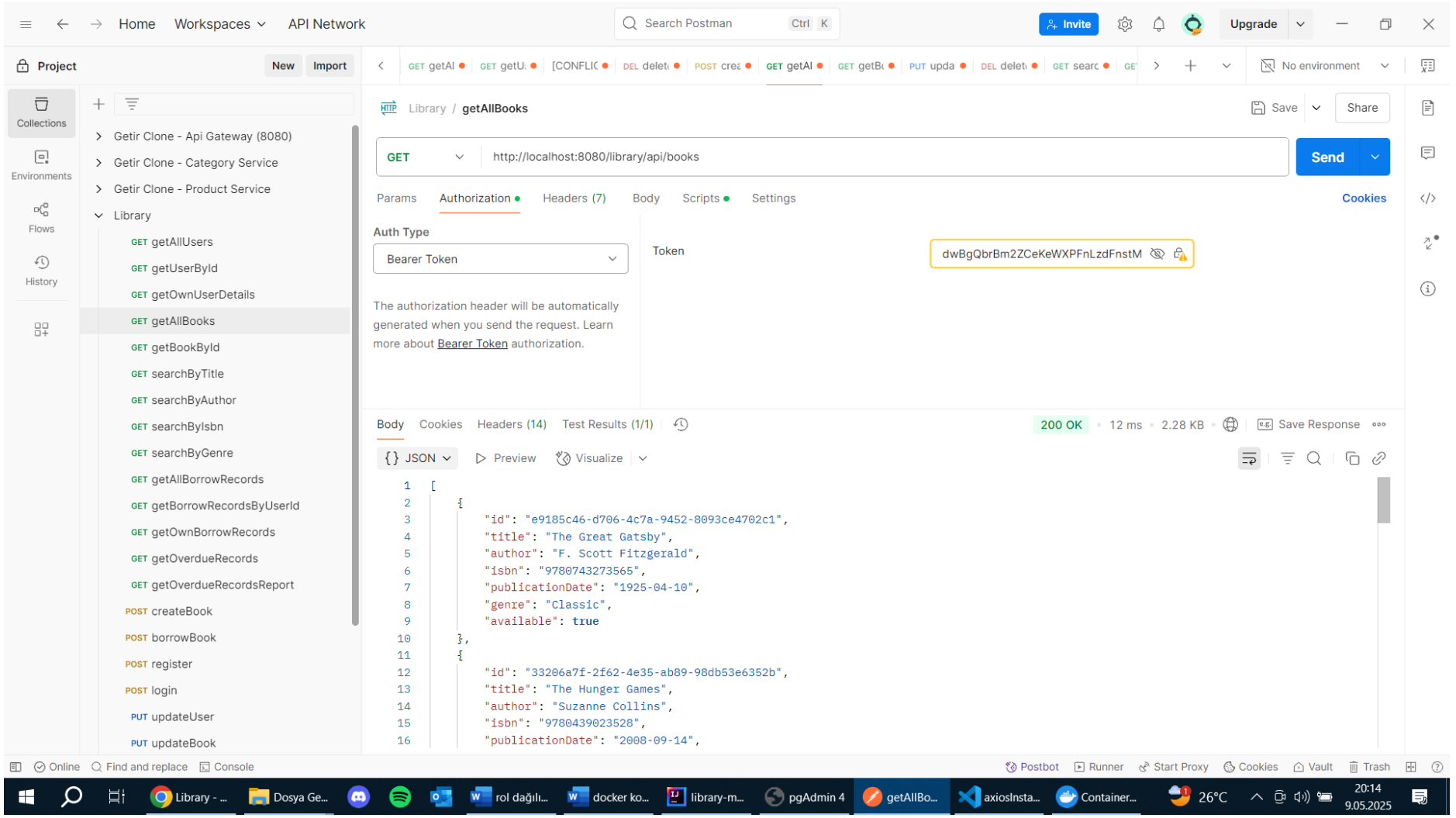
DELETE /users/{id} – Admin Panel View



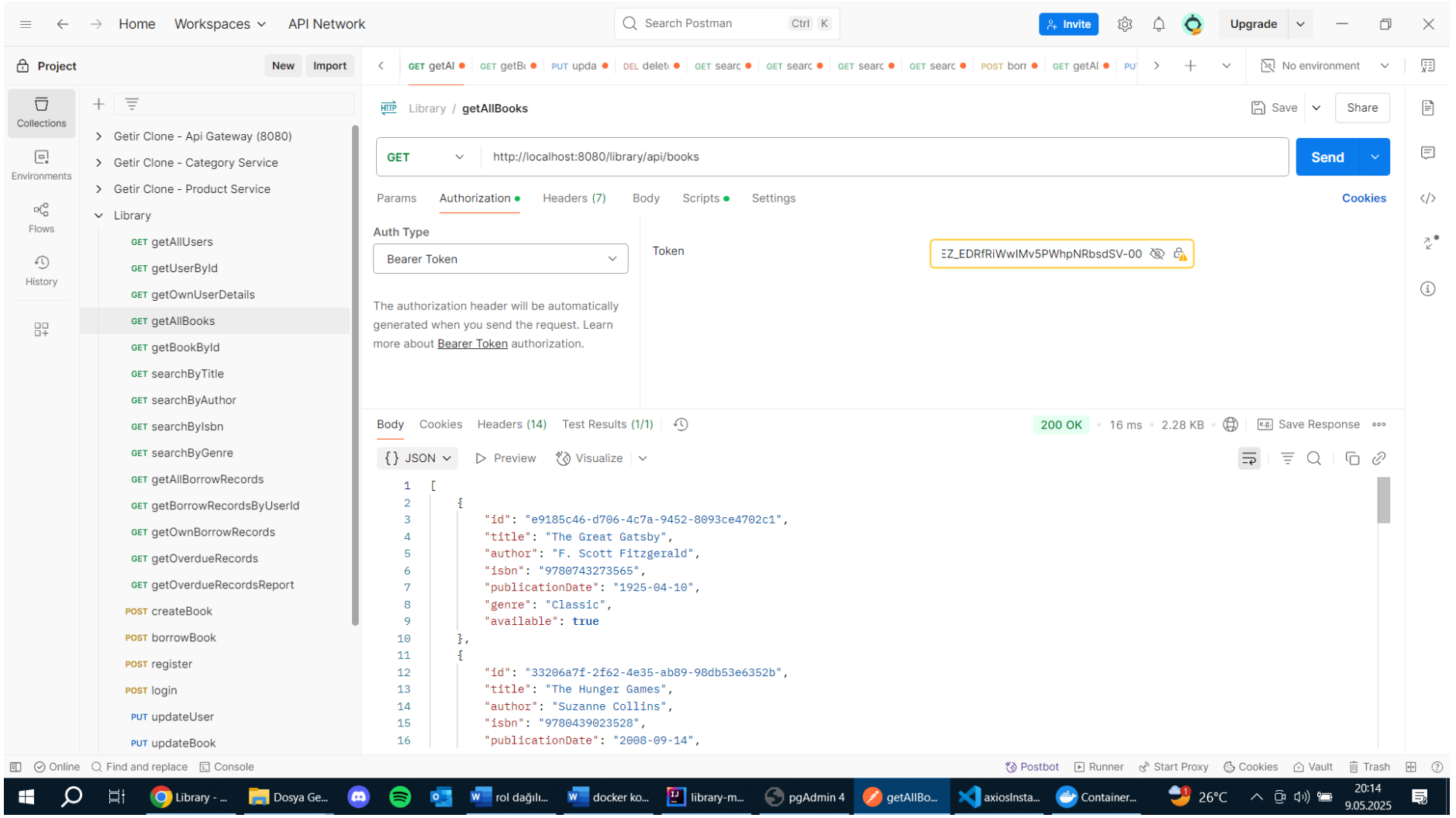
DELETE /users/{id} – Admin Panel View 2



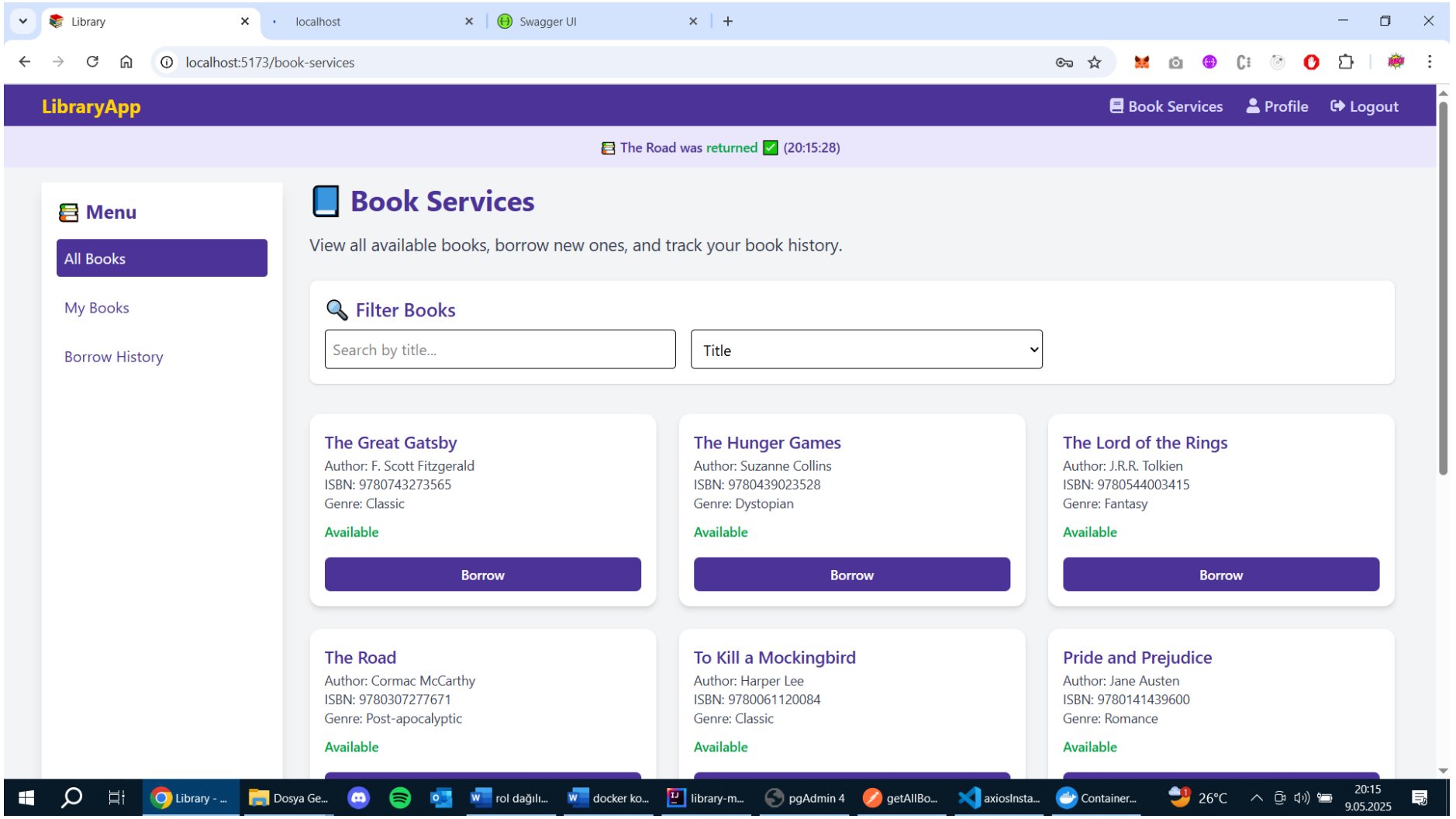
GET /books with LIBRARIAN Token – (200 OK)



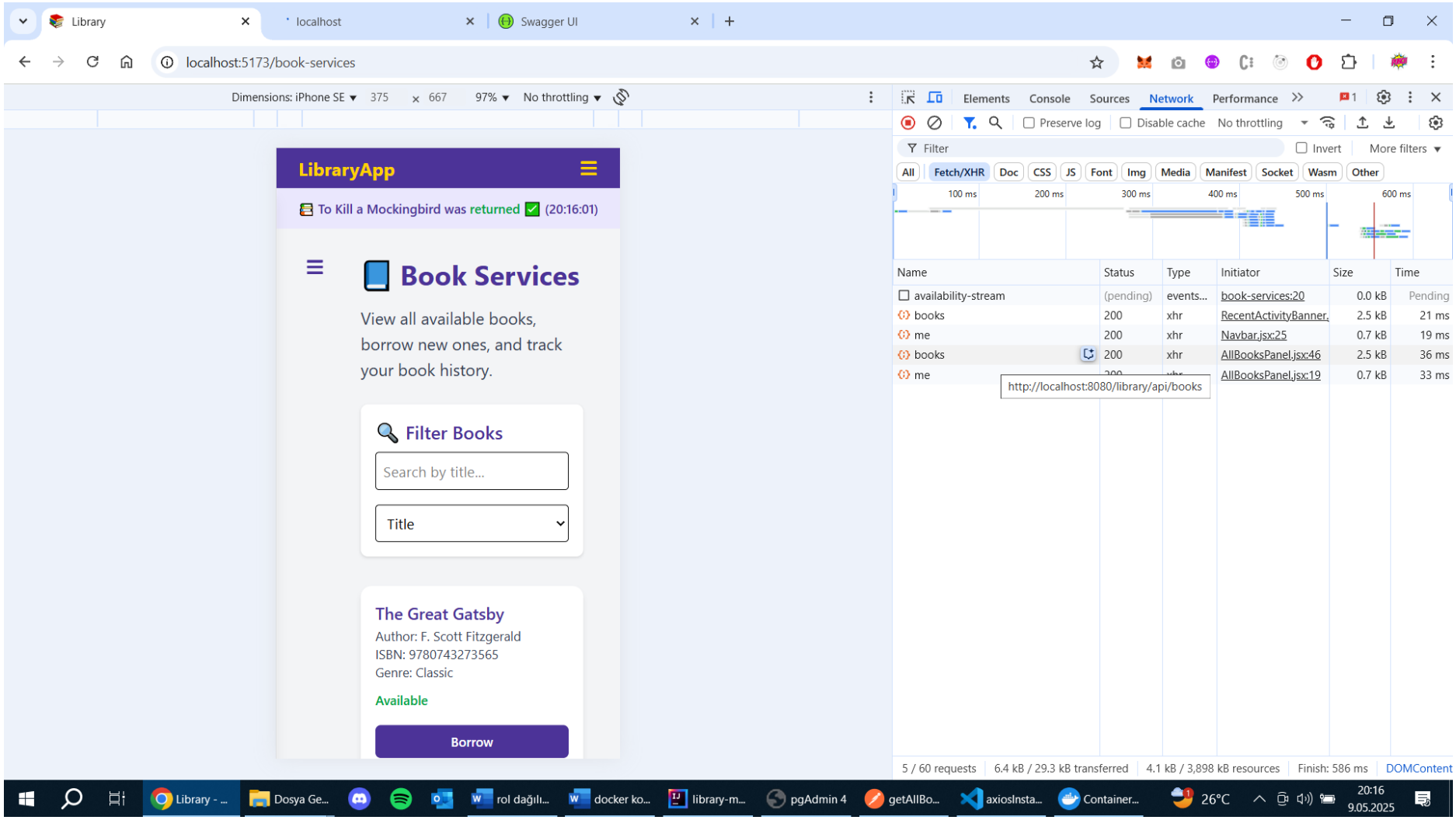
GET /books with PATRON Token – (200 OK)



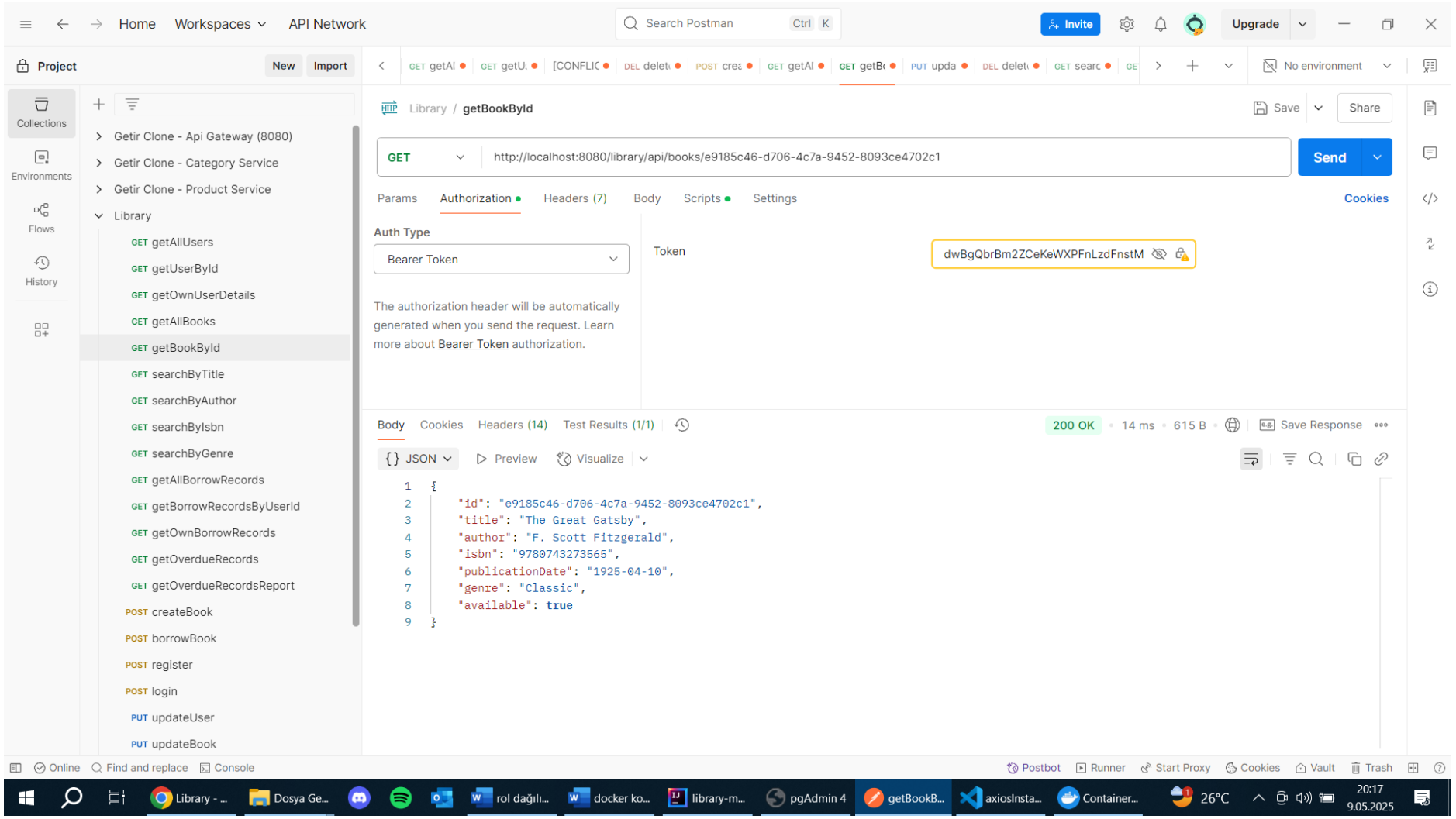
GET /books – Home & All Books



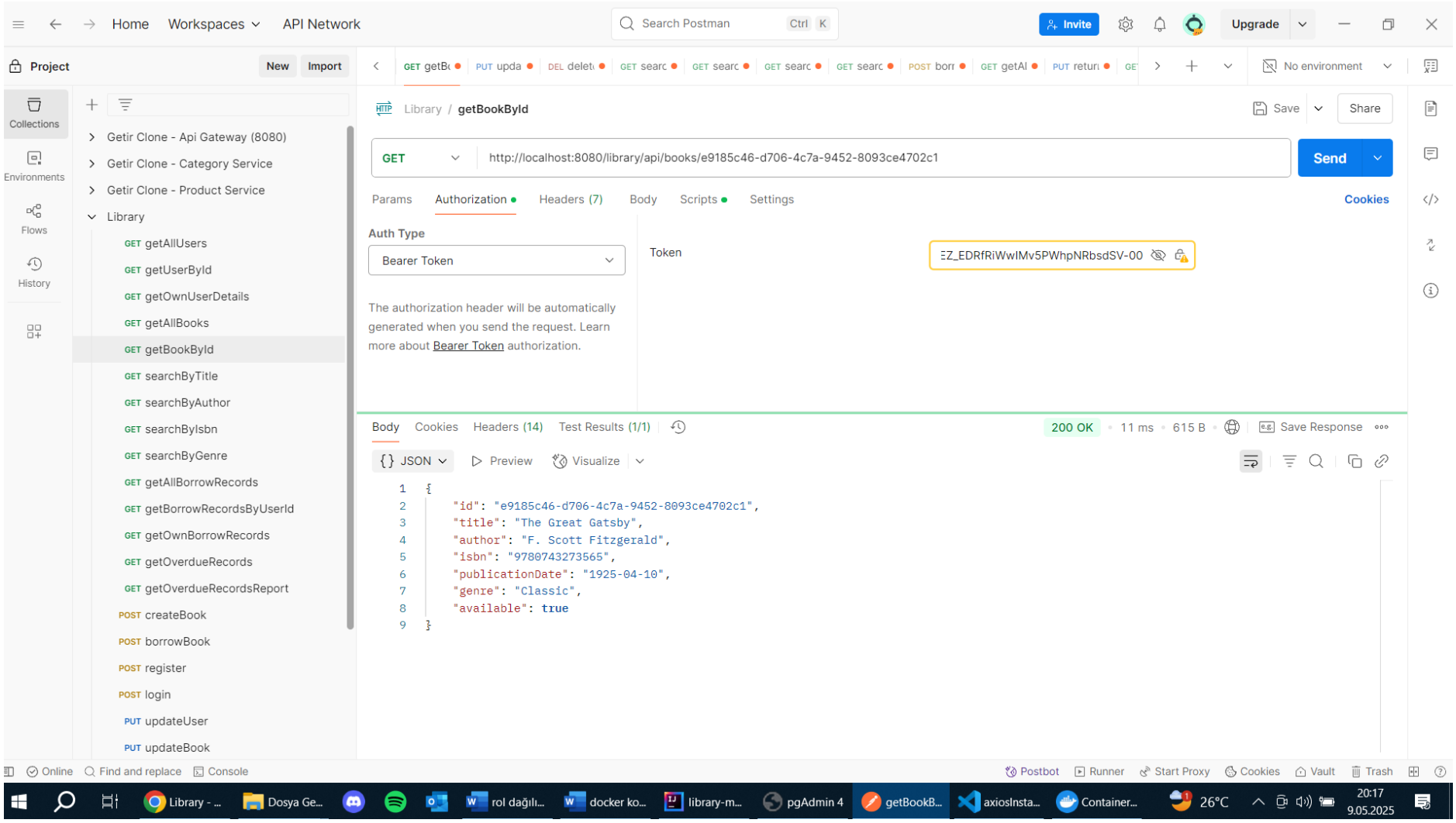
GET /books – Home & All Books 2



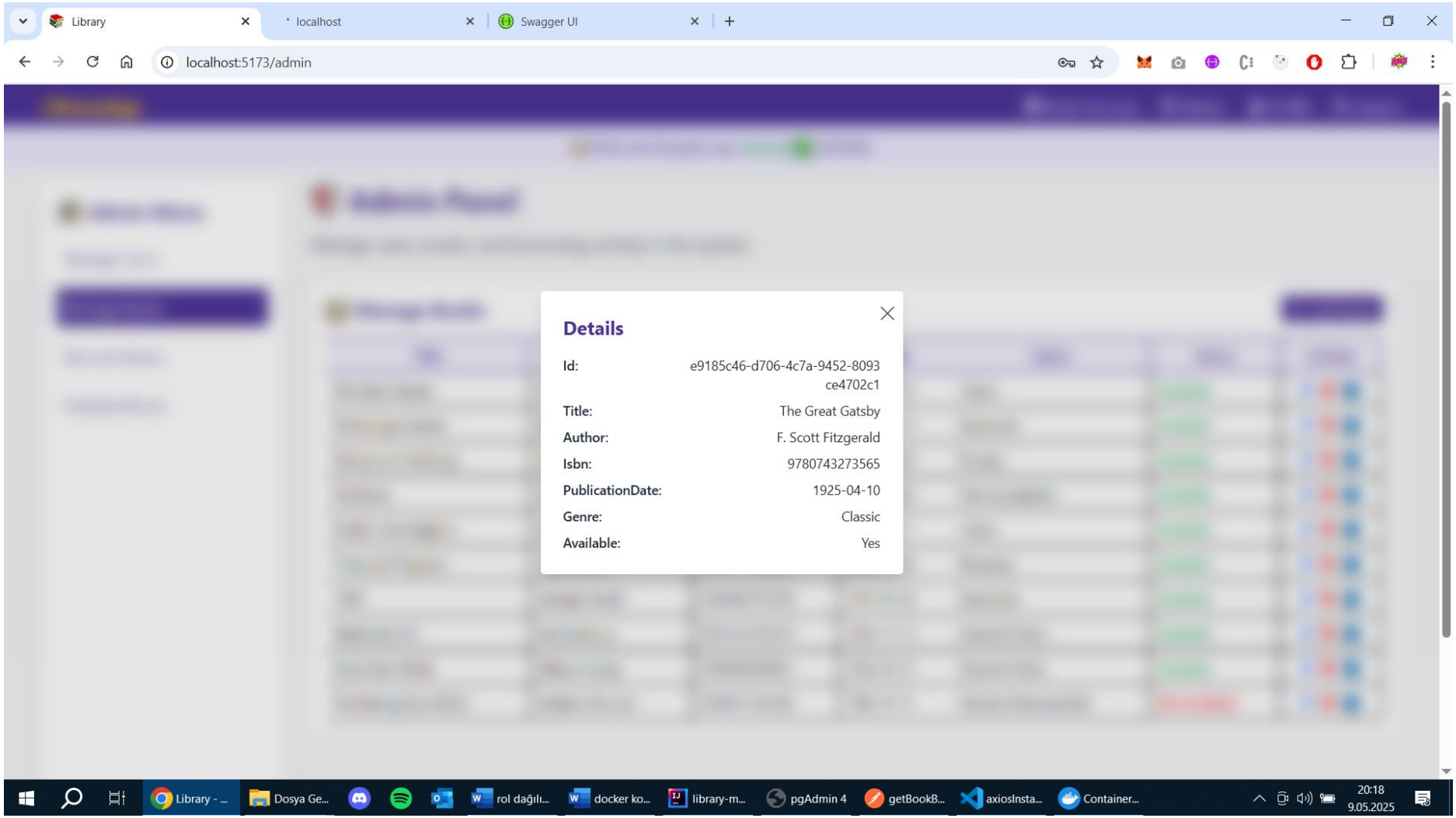
GET /books/{id} with LIBRARIAN Token – (200 OK)



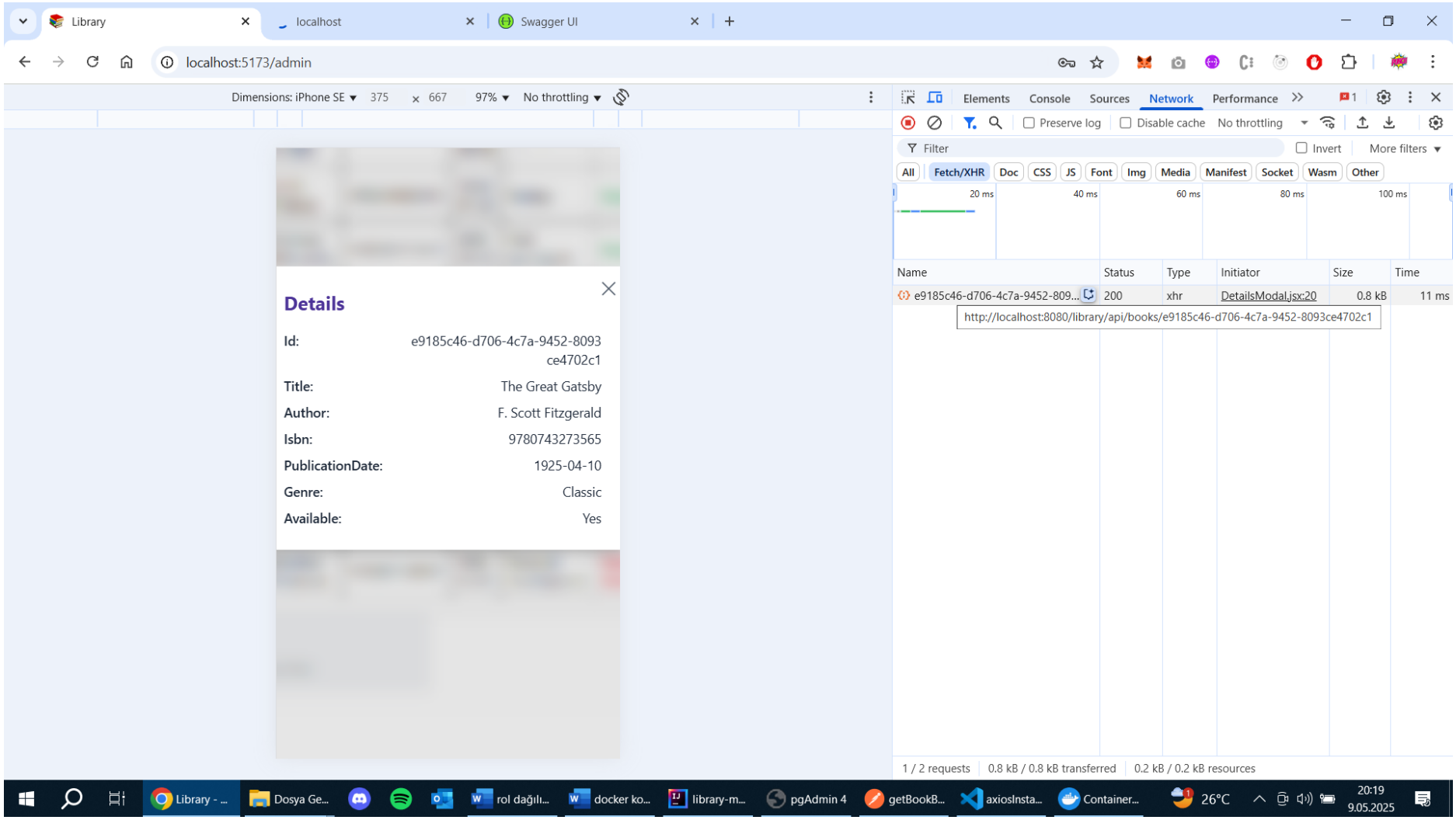
GET /books/{id} with PATRON Token – (200 OK)



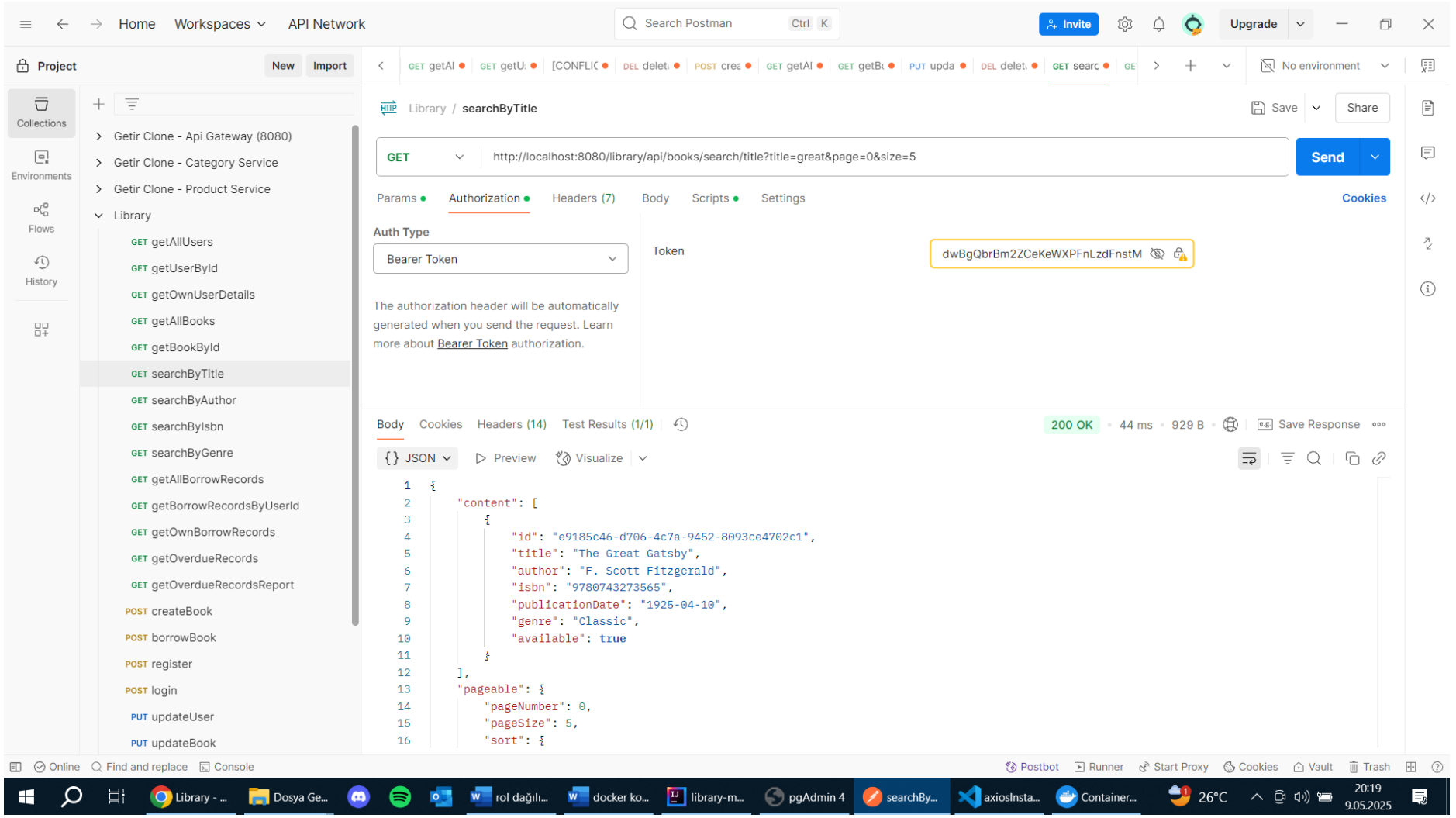
GET /books/{id} – Admin Panel Manage Books Details



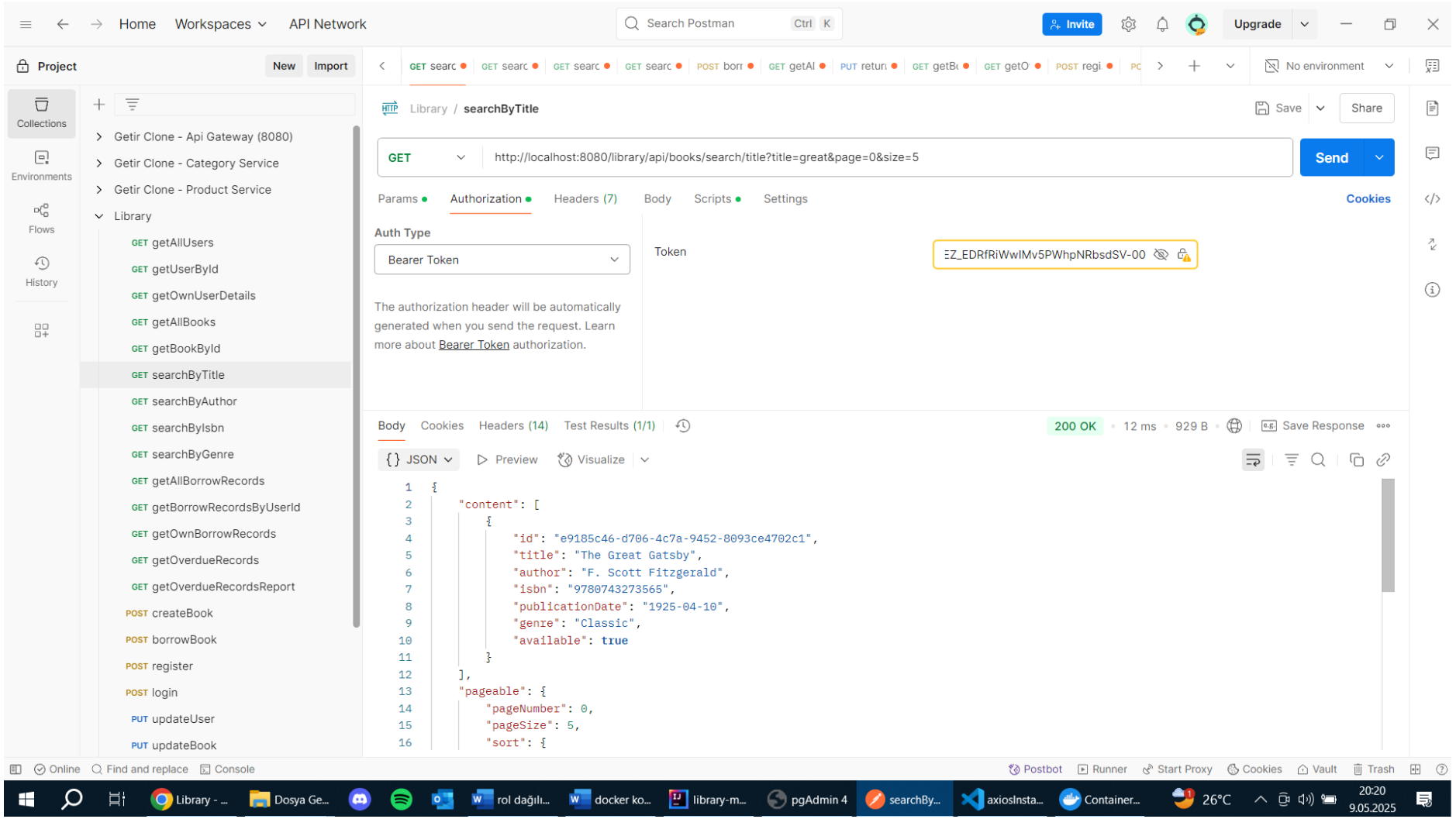
GET /books/{id} – Admin Panel Manage Books Details 2



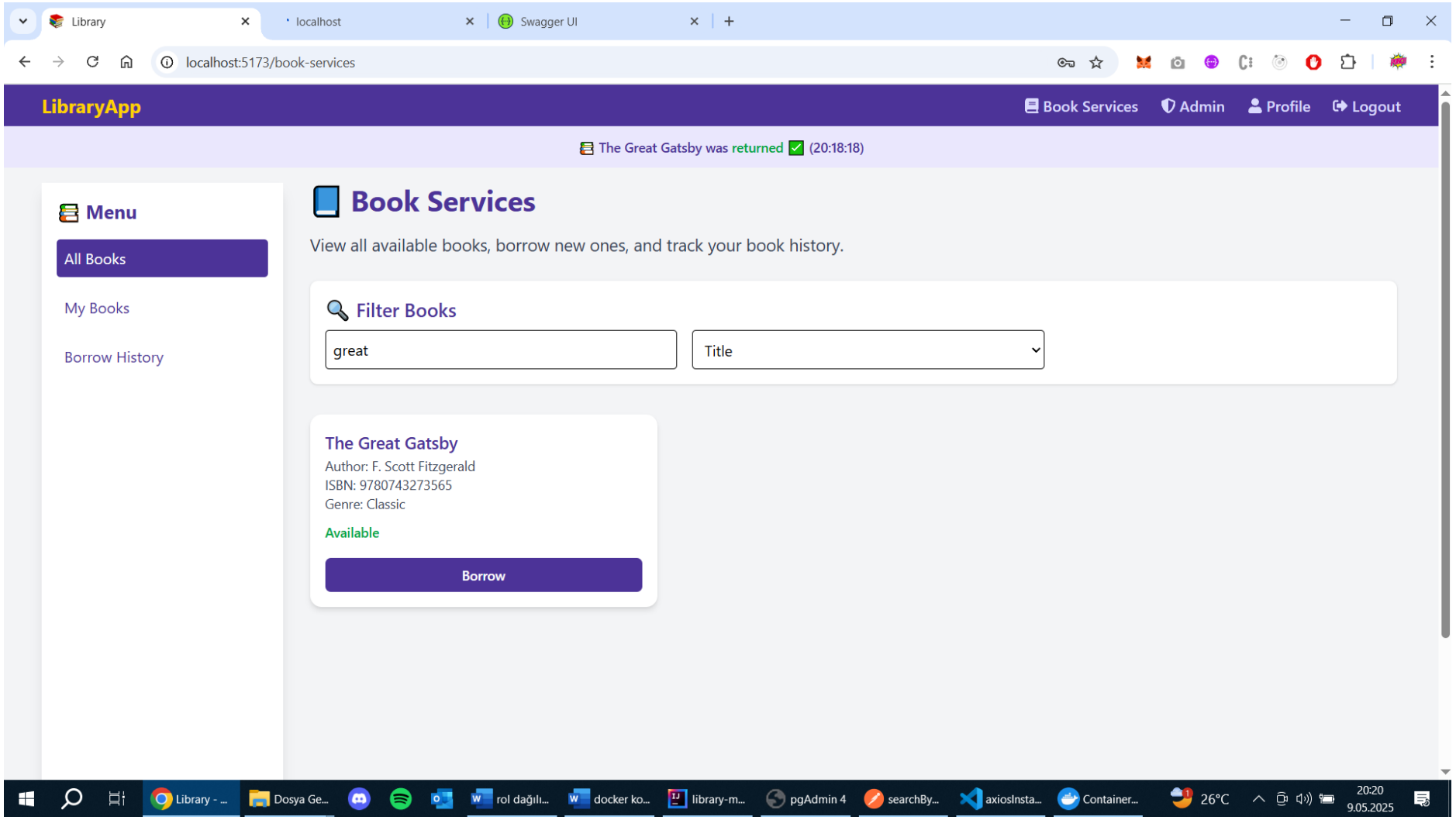
GET /books/search/title with LIBRARIAN Token – (200 OK)



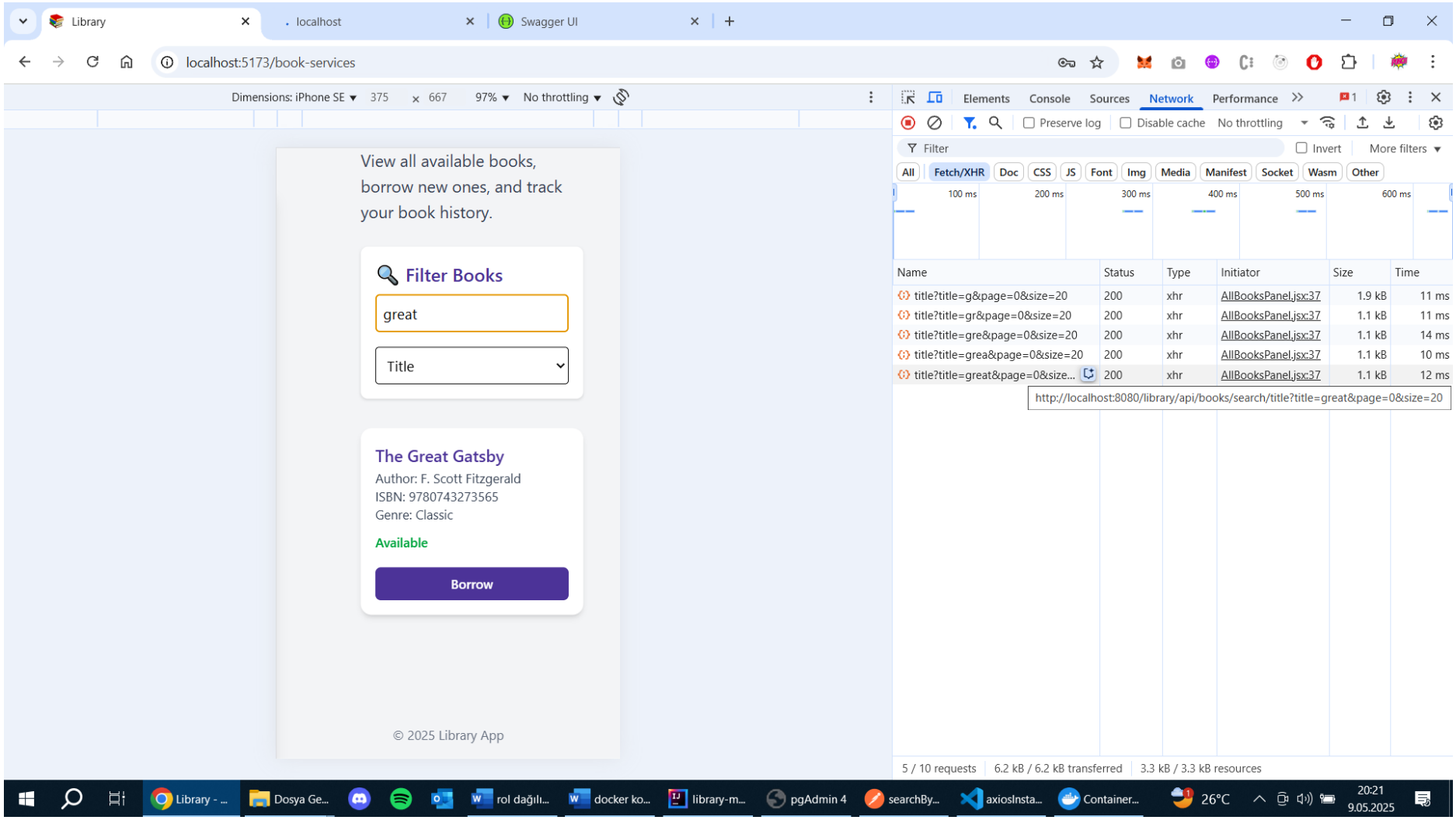
GET /books/search/title with PATRON Token – (200 OK)



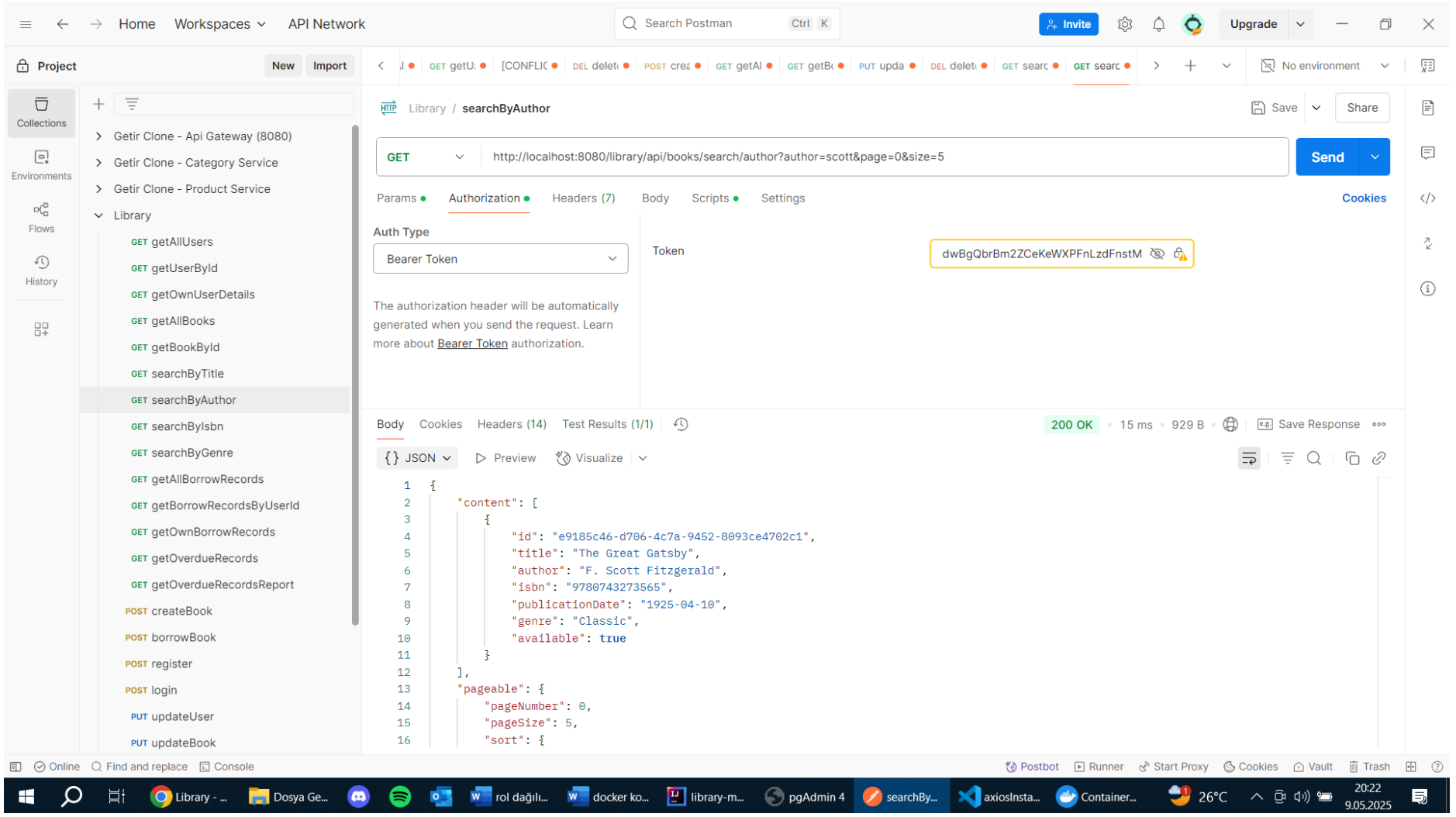
GET /books/search/title – All Books filter books



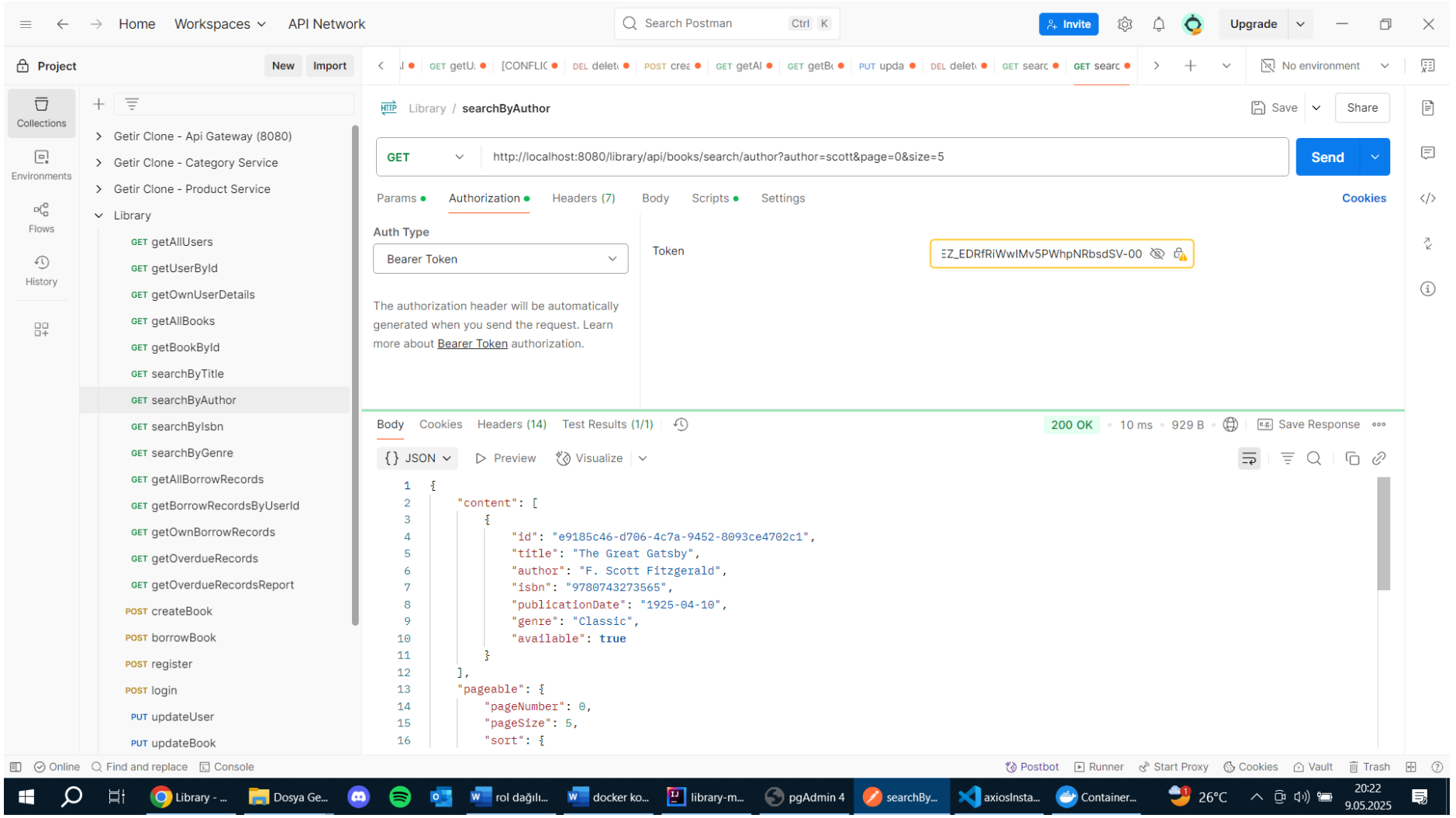
GET /books/search/title – All Books filter books 2



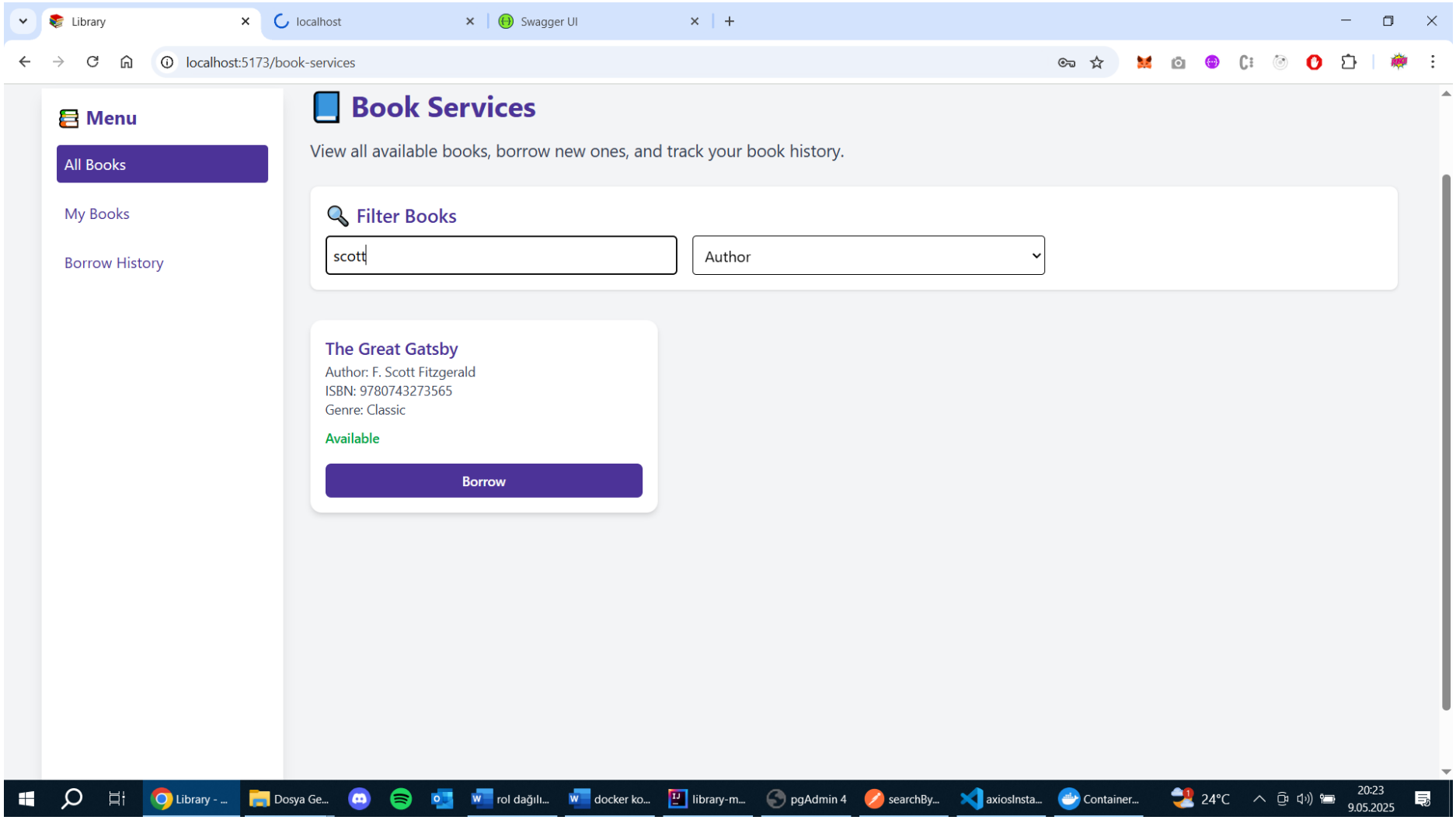
GET /books/search/author with LIBRARIAN Token – (200 OK)



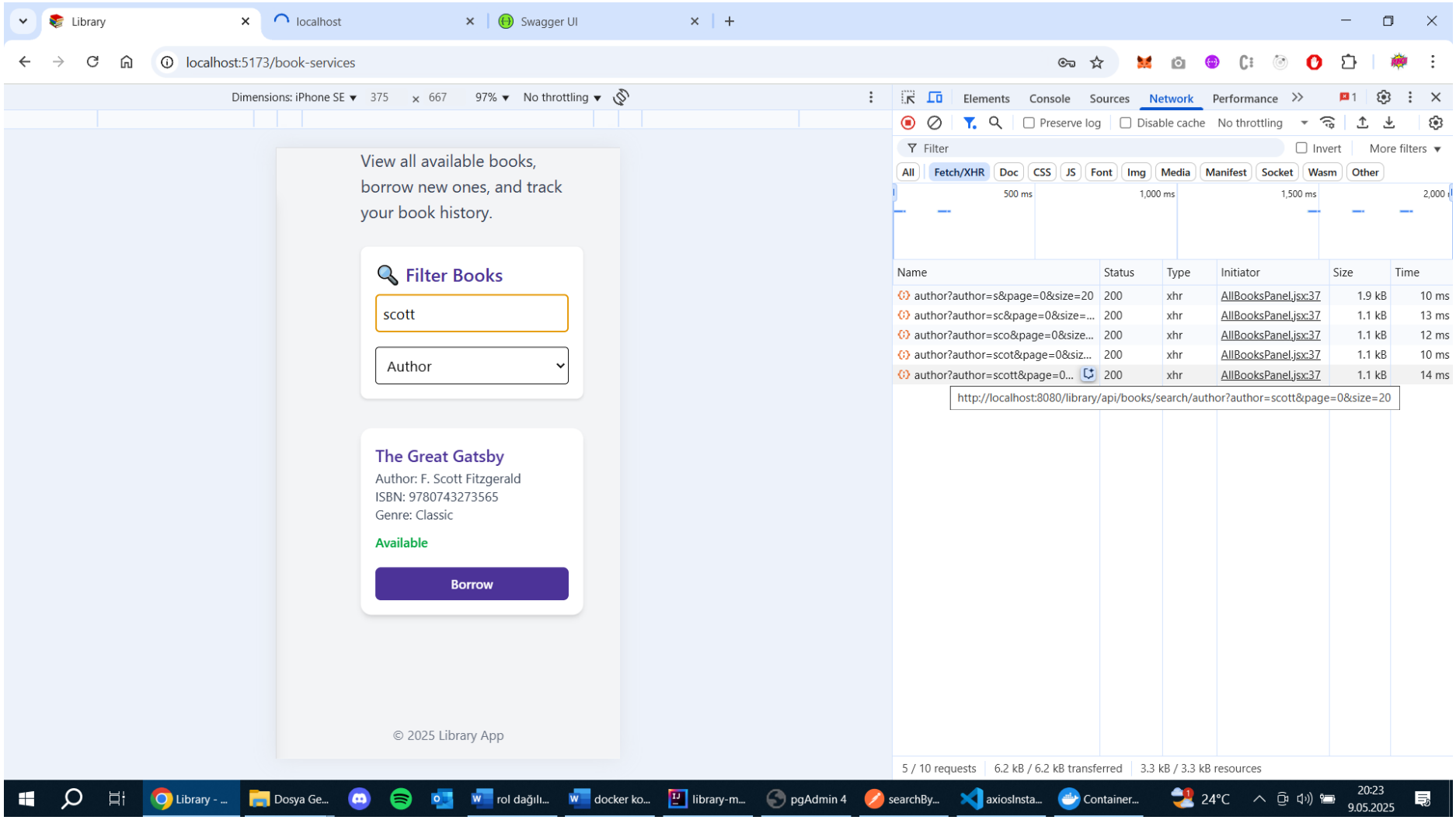
GET /books/search/author with PATRON Token – (200 OK)



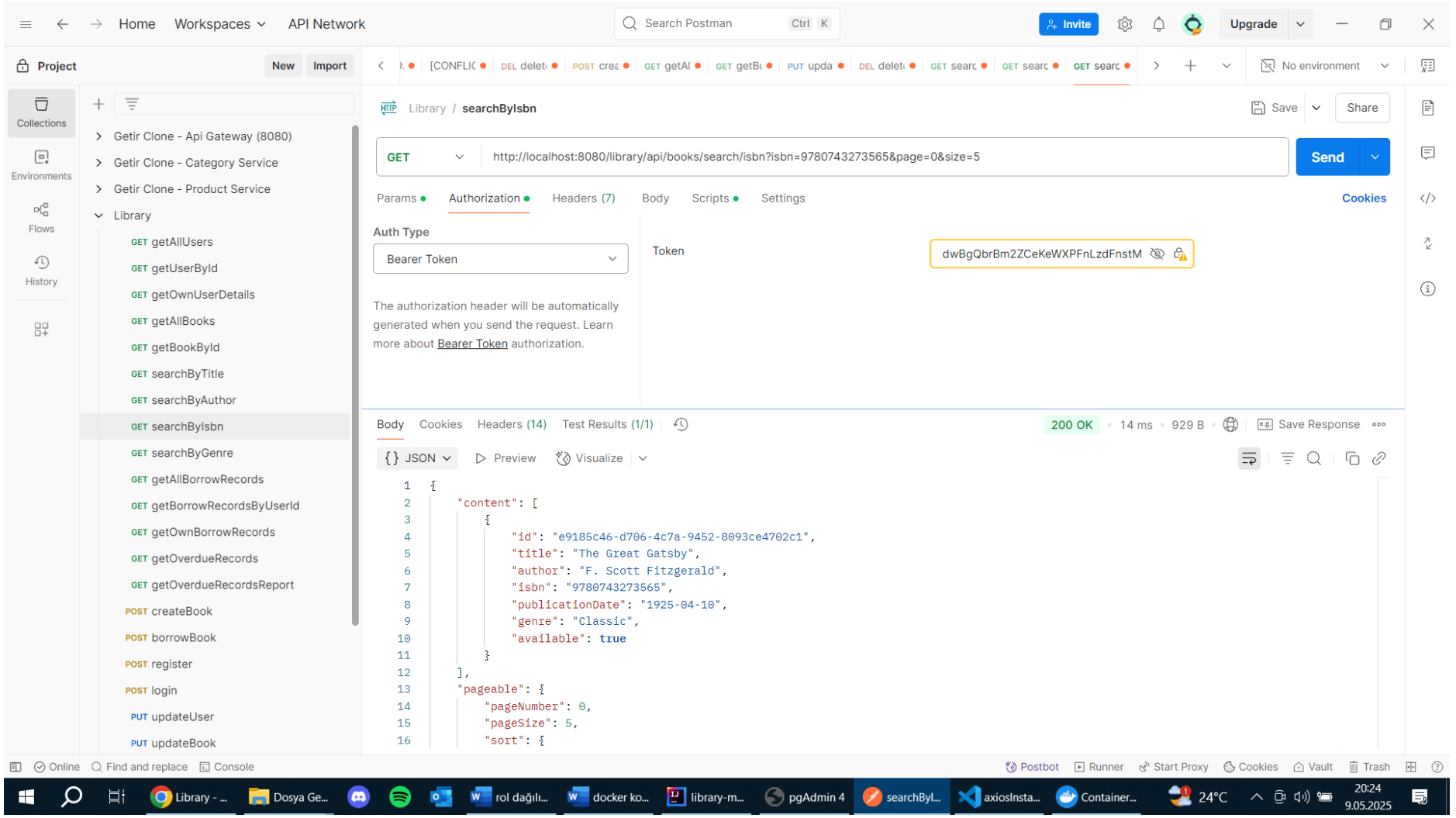
GET /books/search/author – All Books filter books



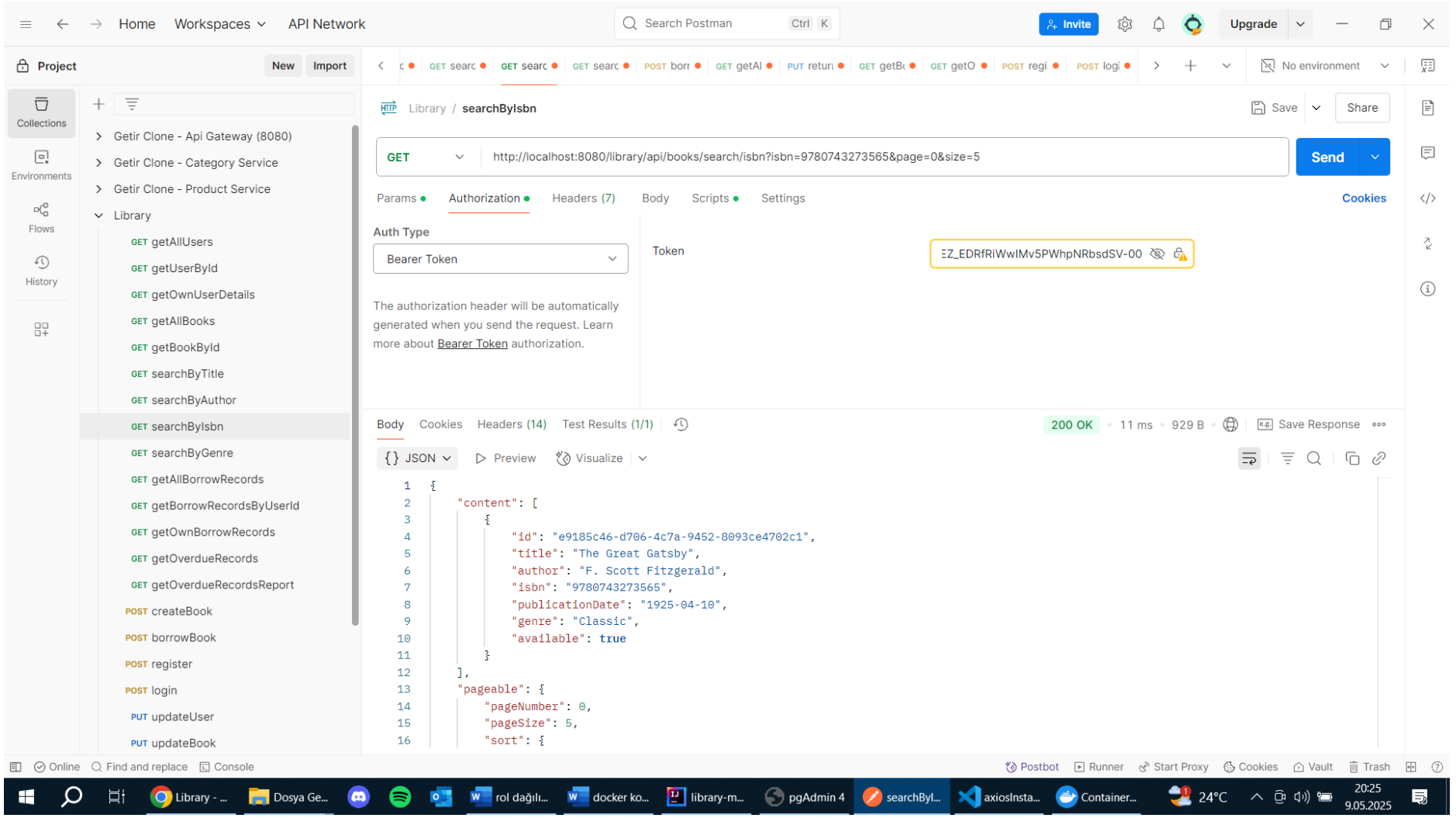
GET /books/search/author – All Books filter books 2



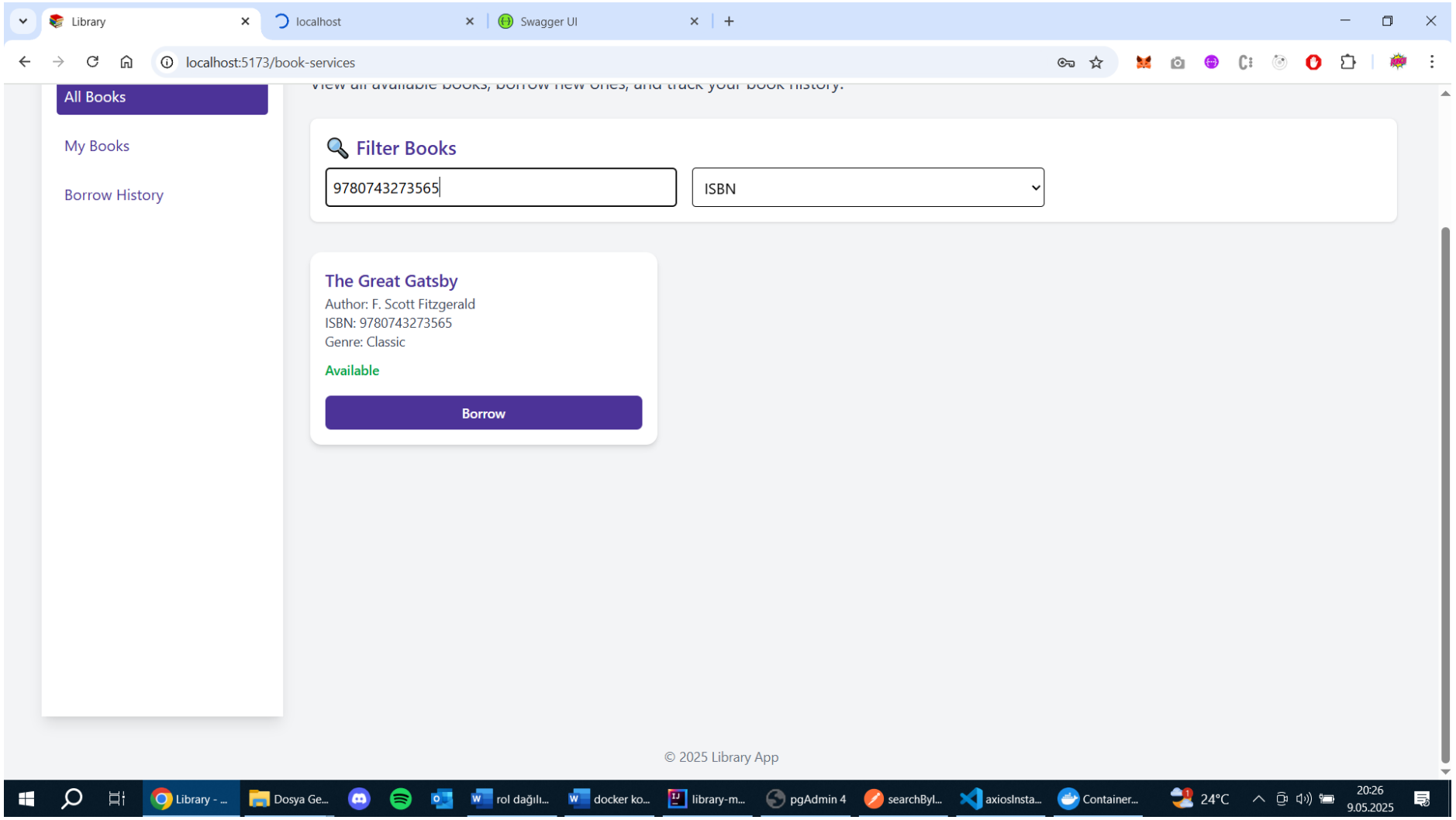
GET /books/search/isbn with LIBRARIAN Token – (200 OK)



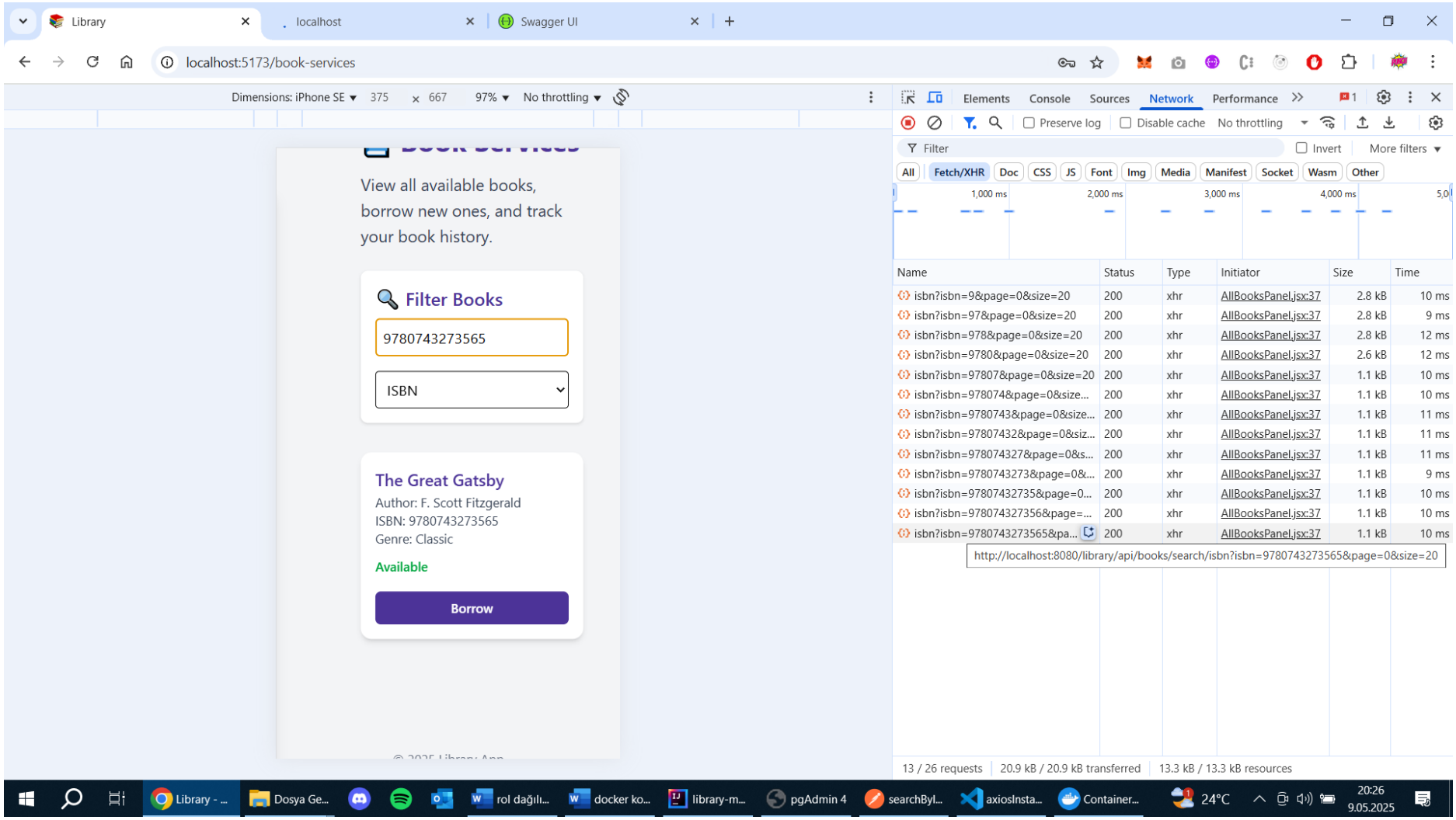
GET /books/search/isbn with PATRON Token – (200 OK)



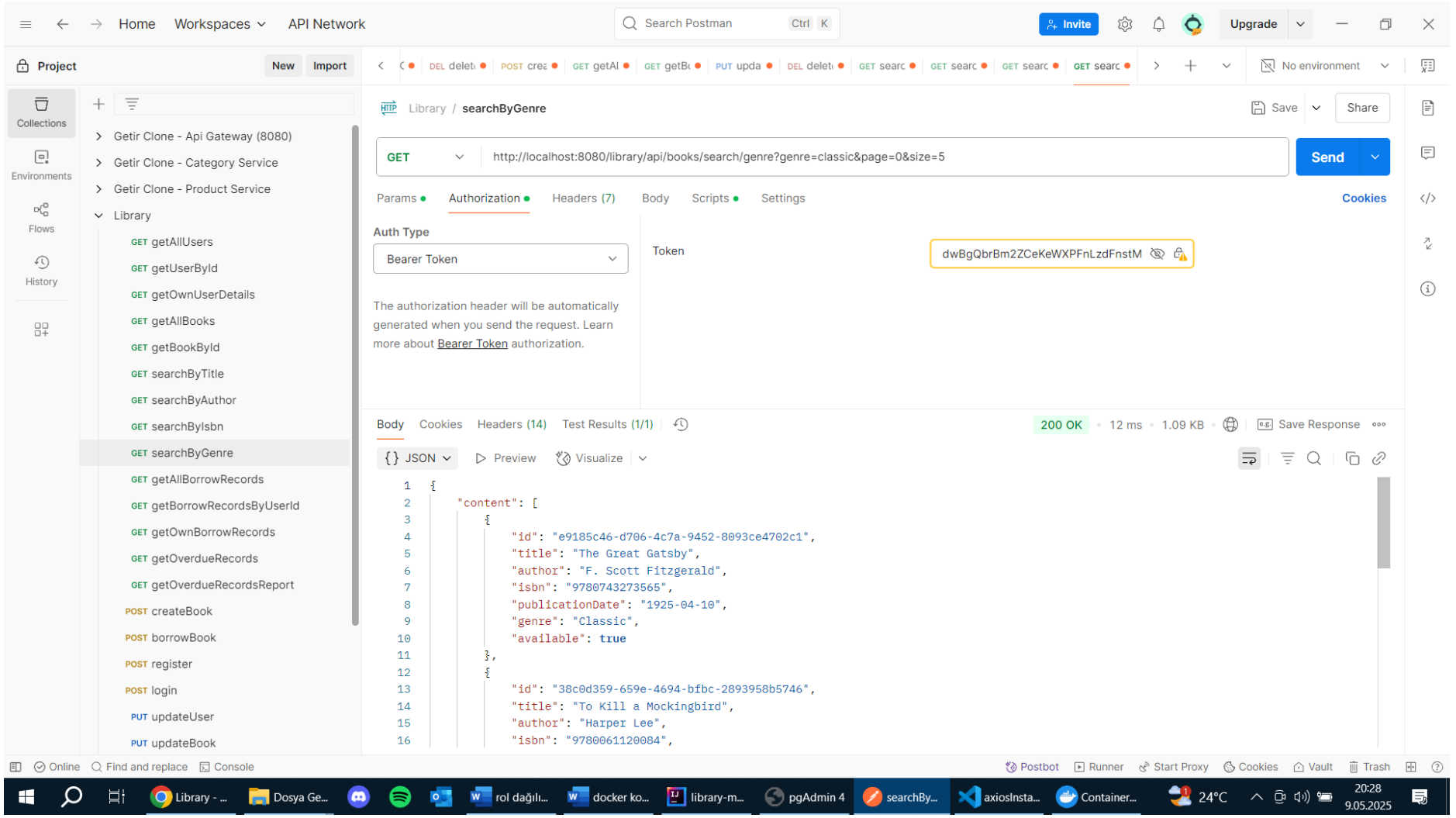
GET /books/search/isbn – All Books filter books



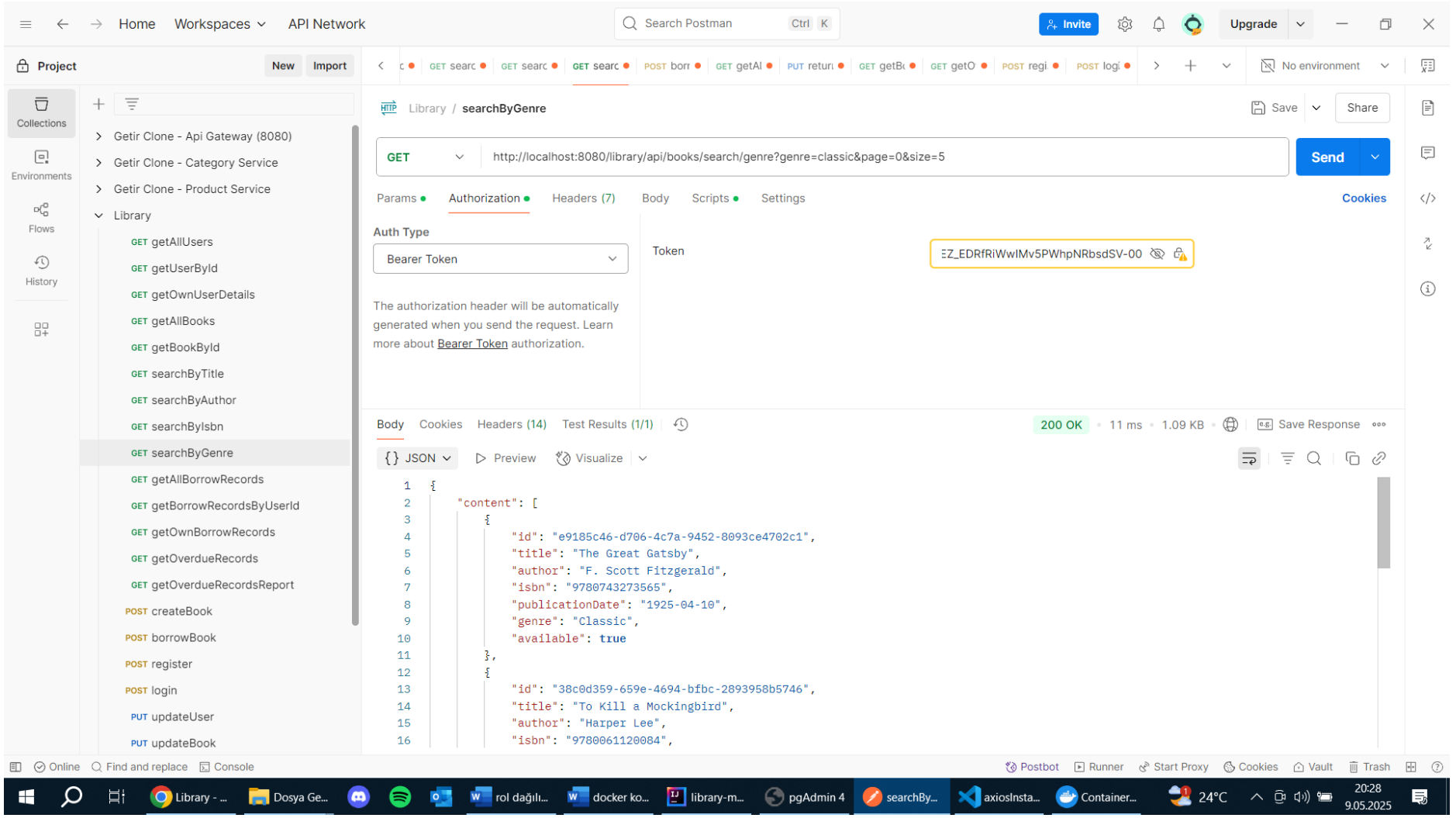
GET /books/search/isbn – All Books filter books 2



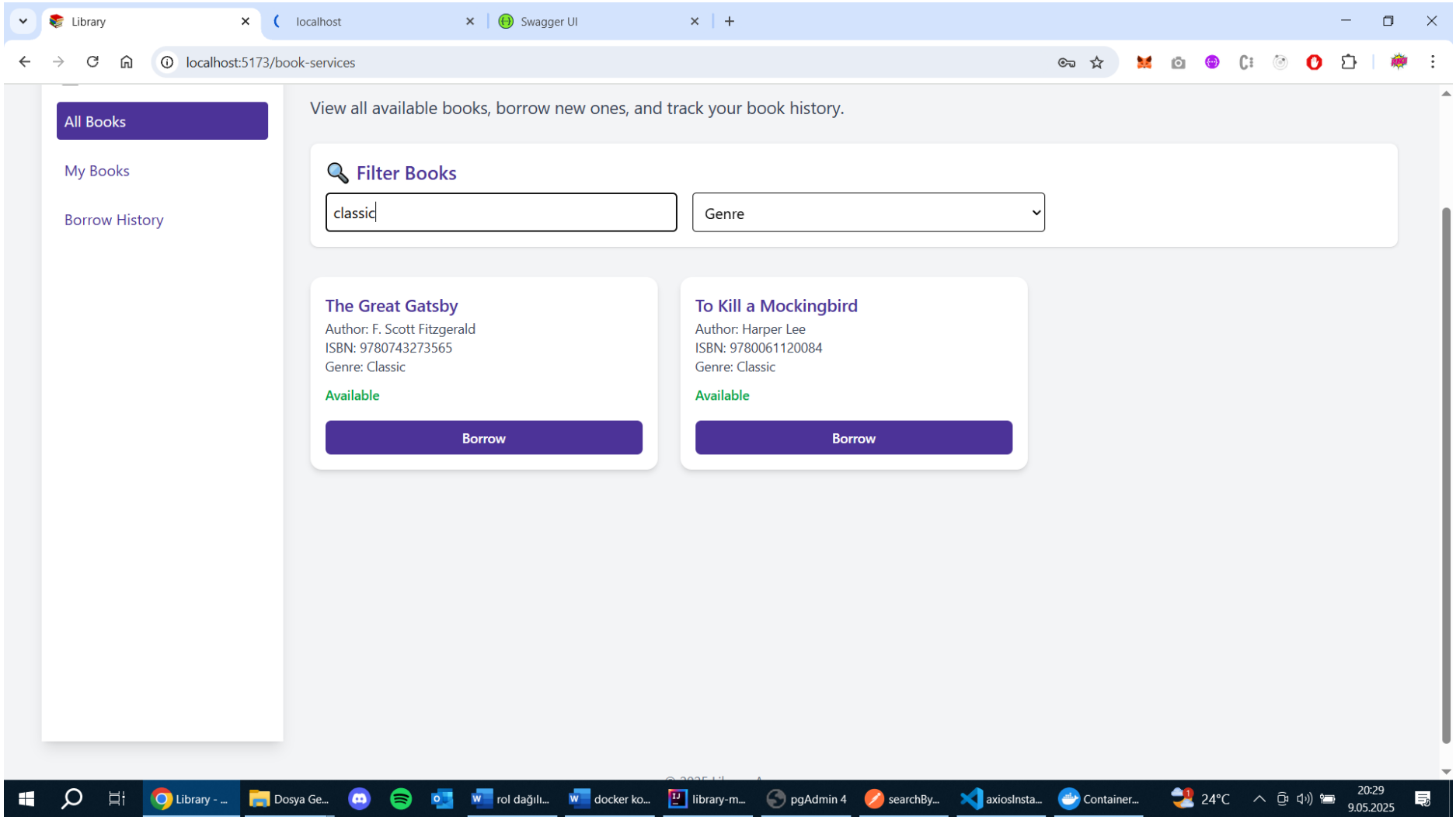
GET /books/search/genre with LIBRARIAN Token – (200 OK)



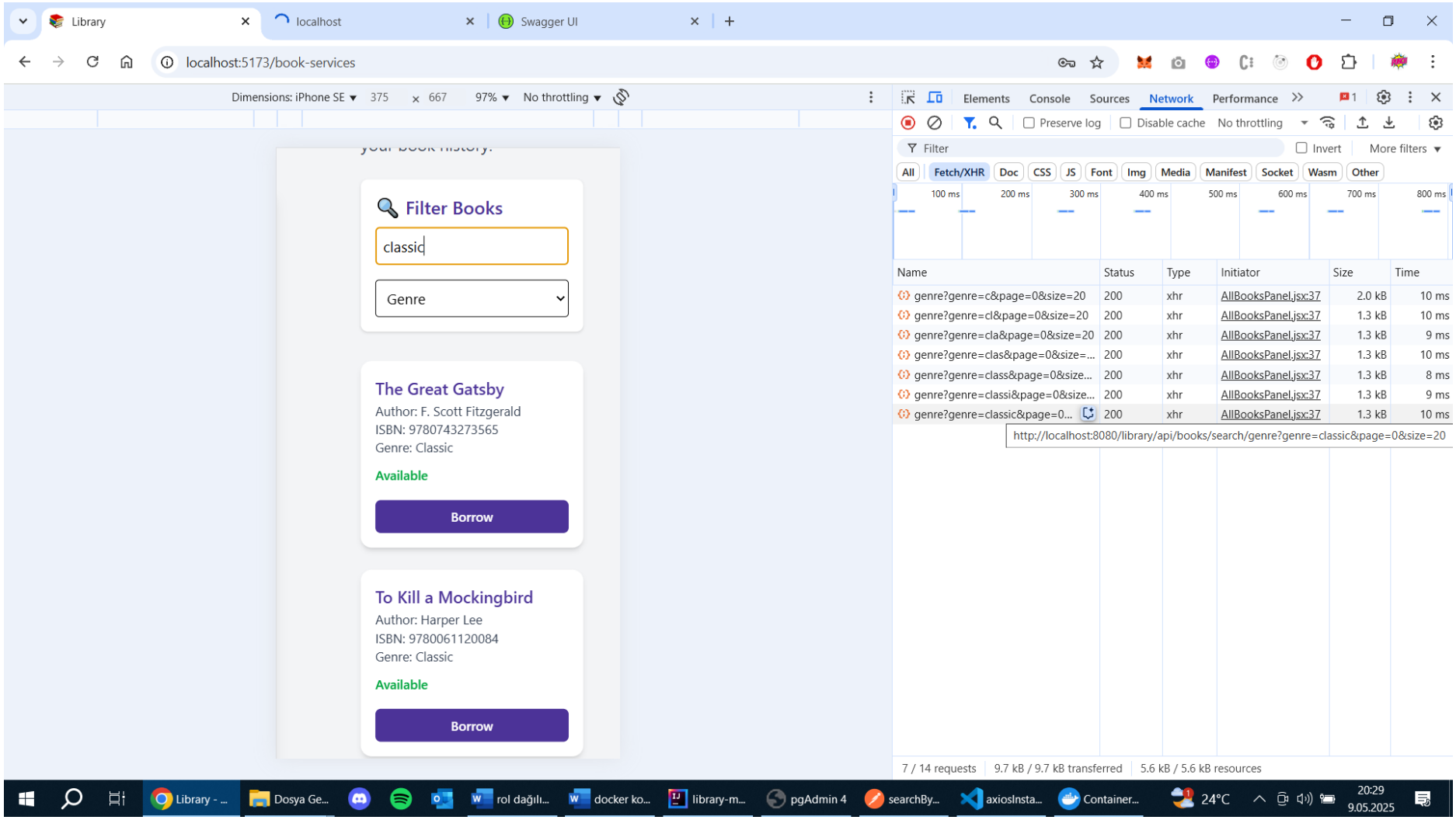
GET /books/search/genre with PATRON Token – (200 OK)



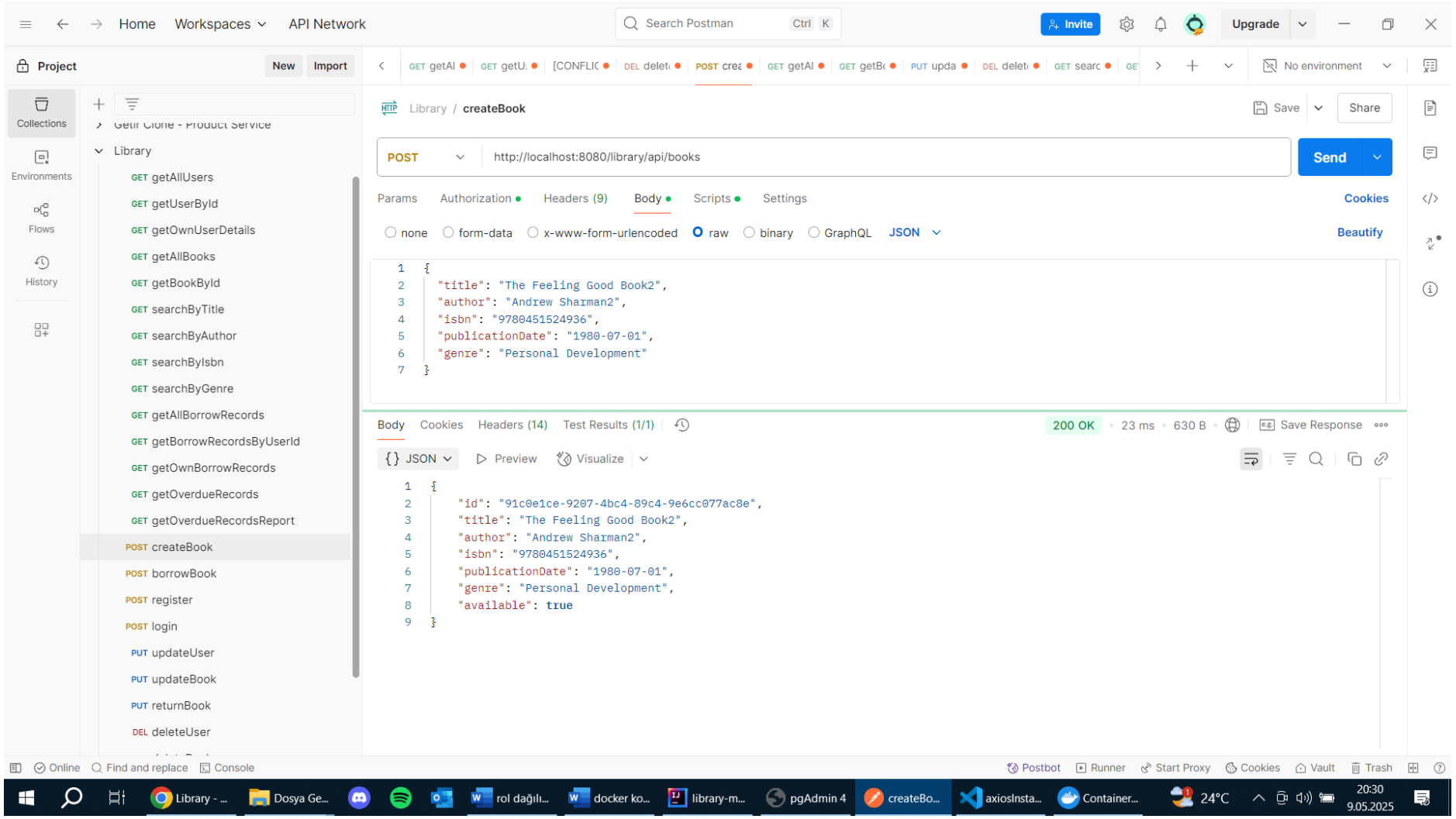
GET /books/search/genre – All Books filter books



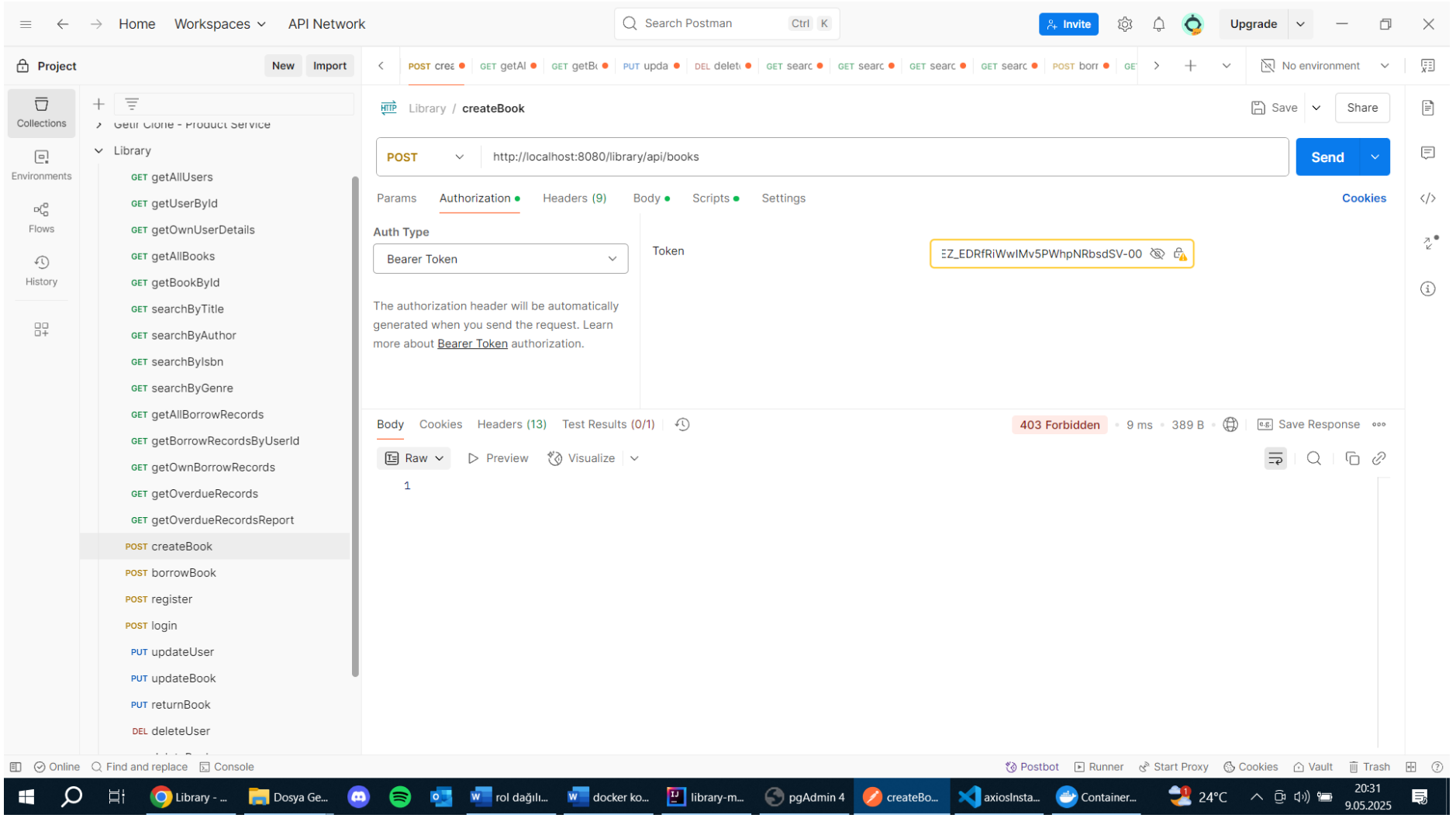
GET /books/search/genre – All Books filter books 2



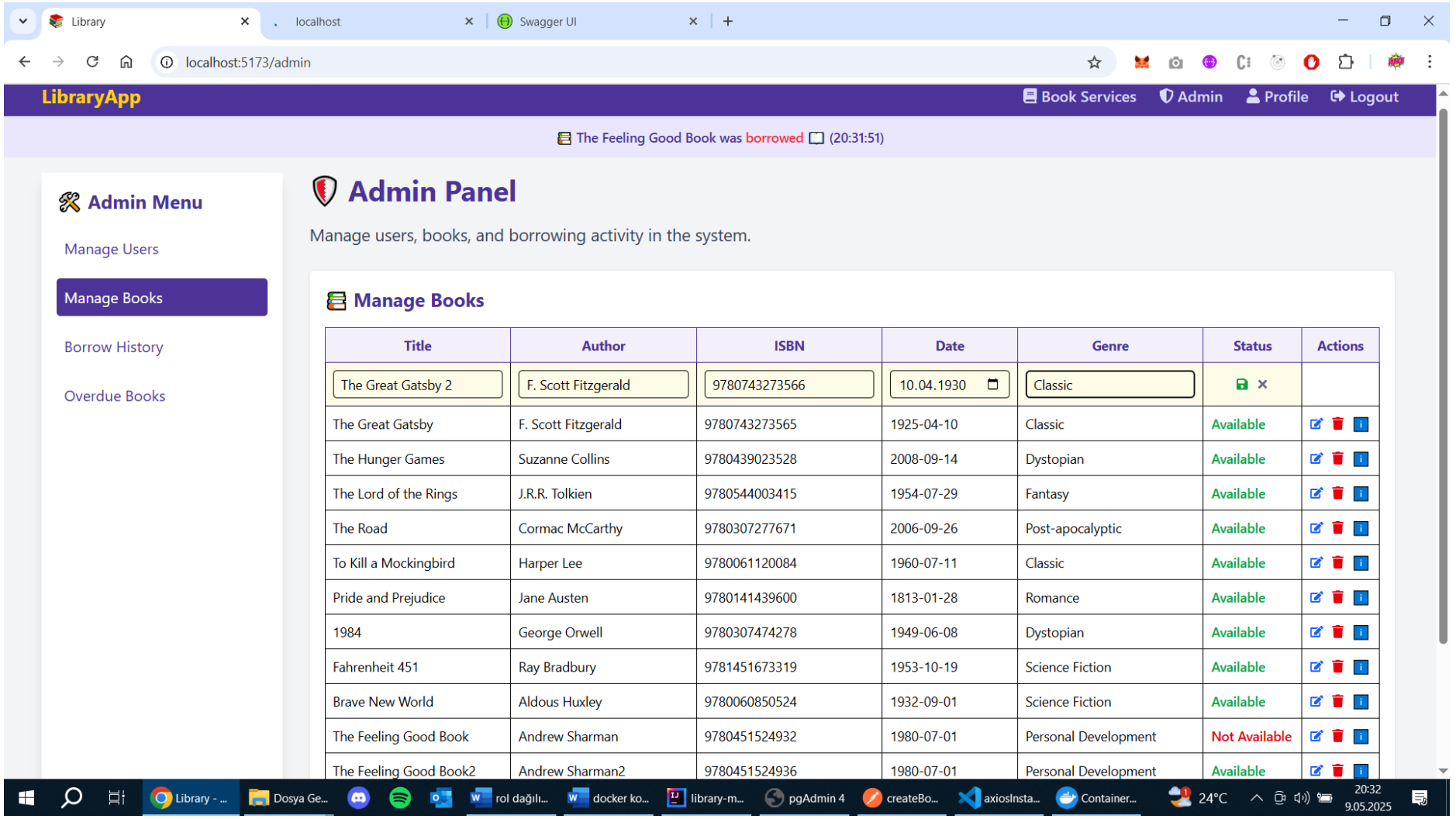
POST /books with LIBRARIAN Token – (200 OK)



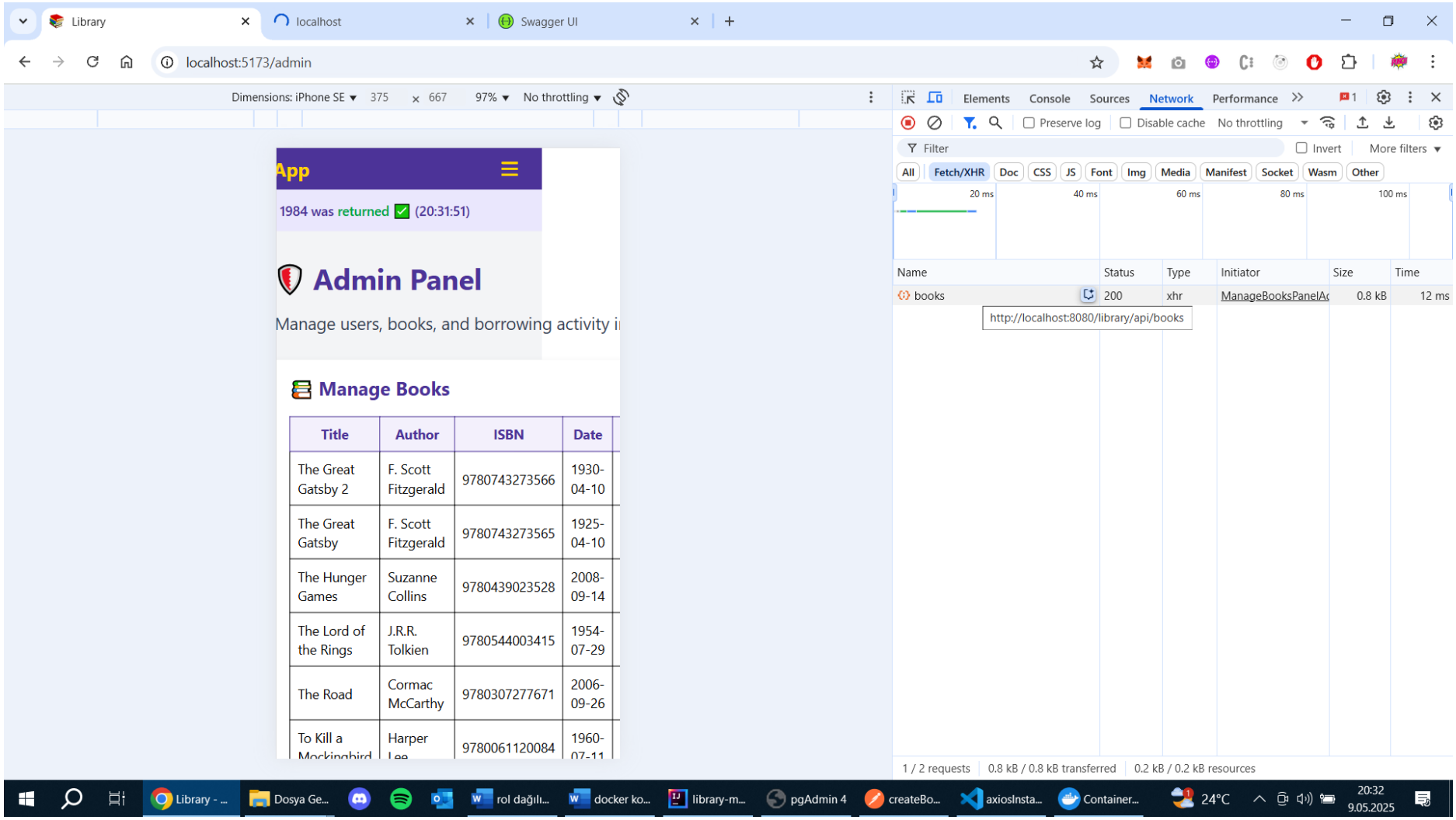
POST /books with PATRON Token – (403 Forbidden)



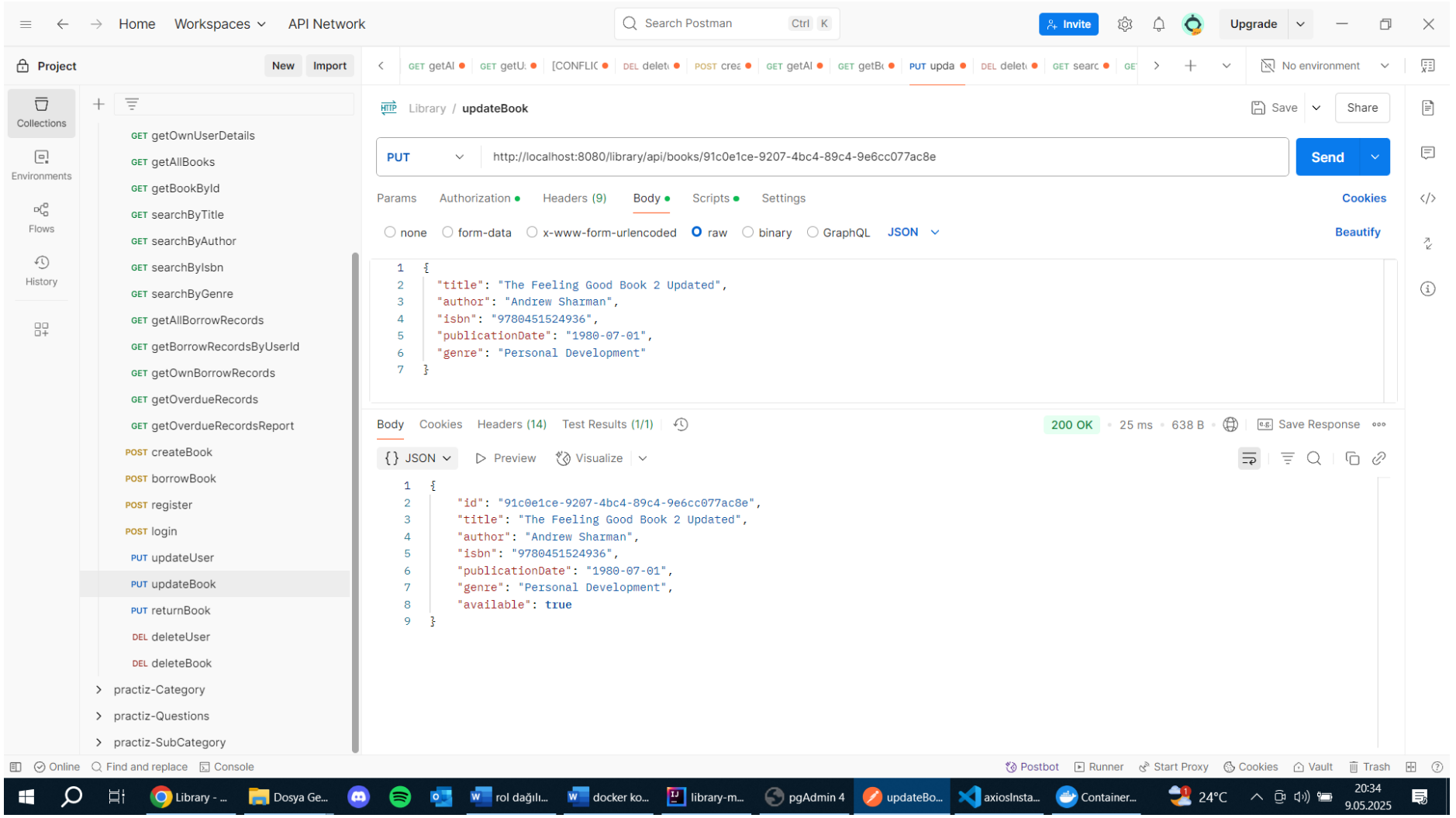
POST /books – Admin panel manage books



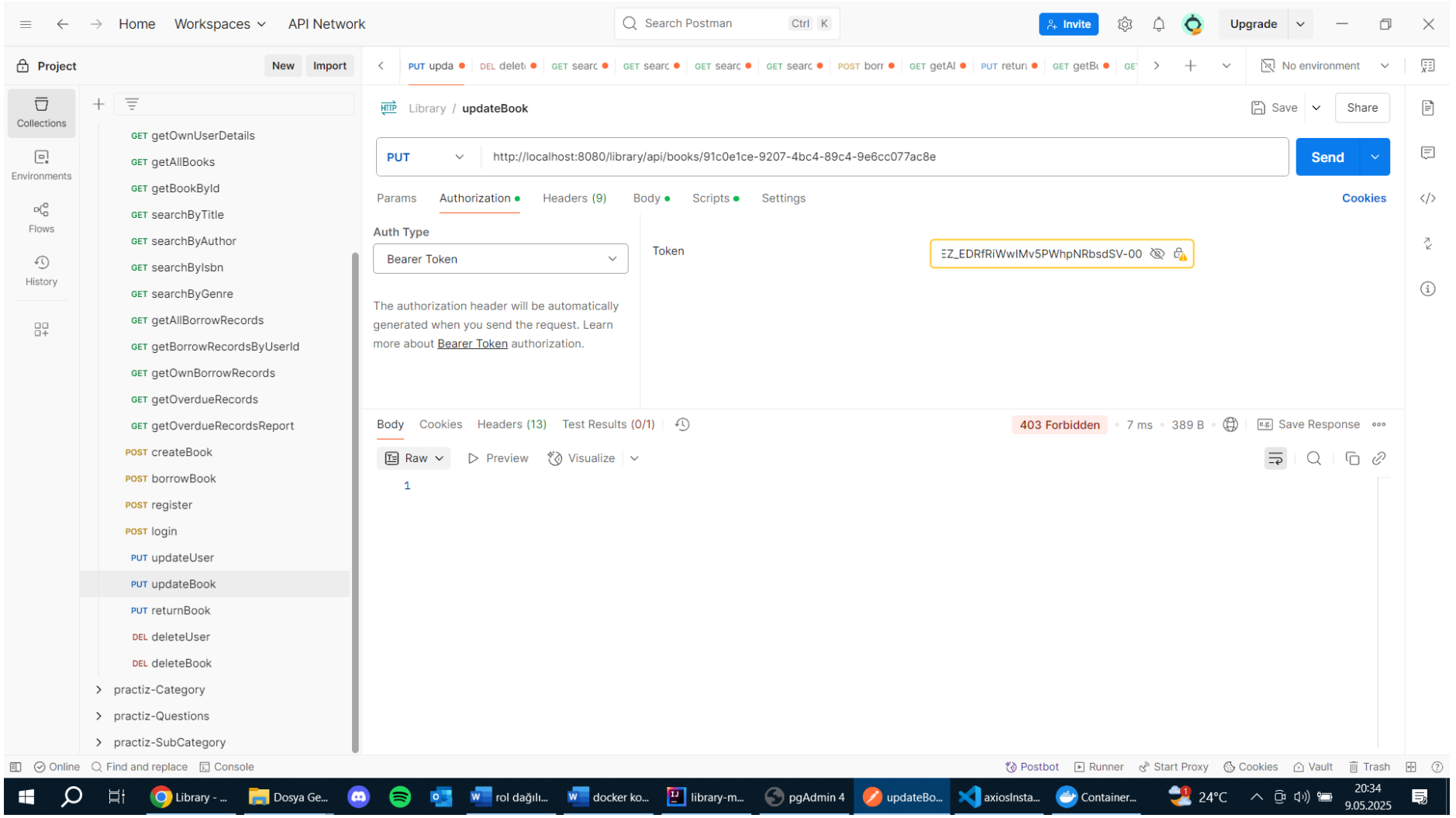
POST /books – Admin panel manage books 2



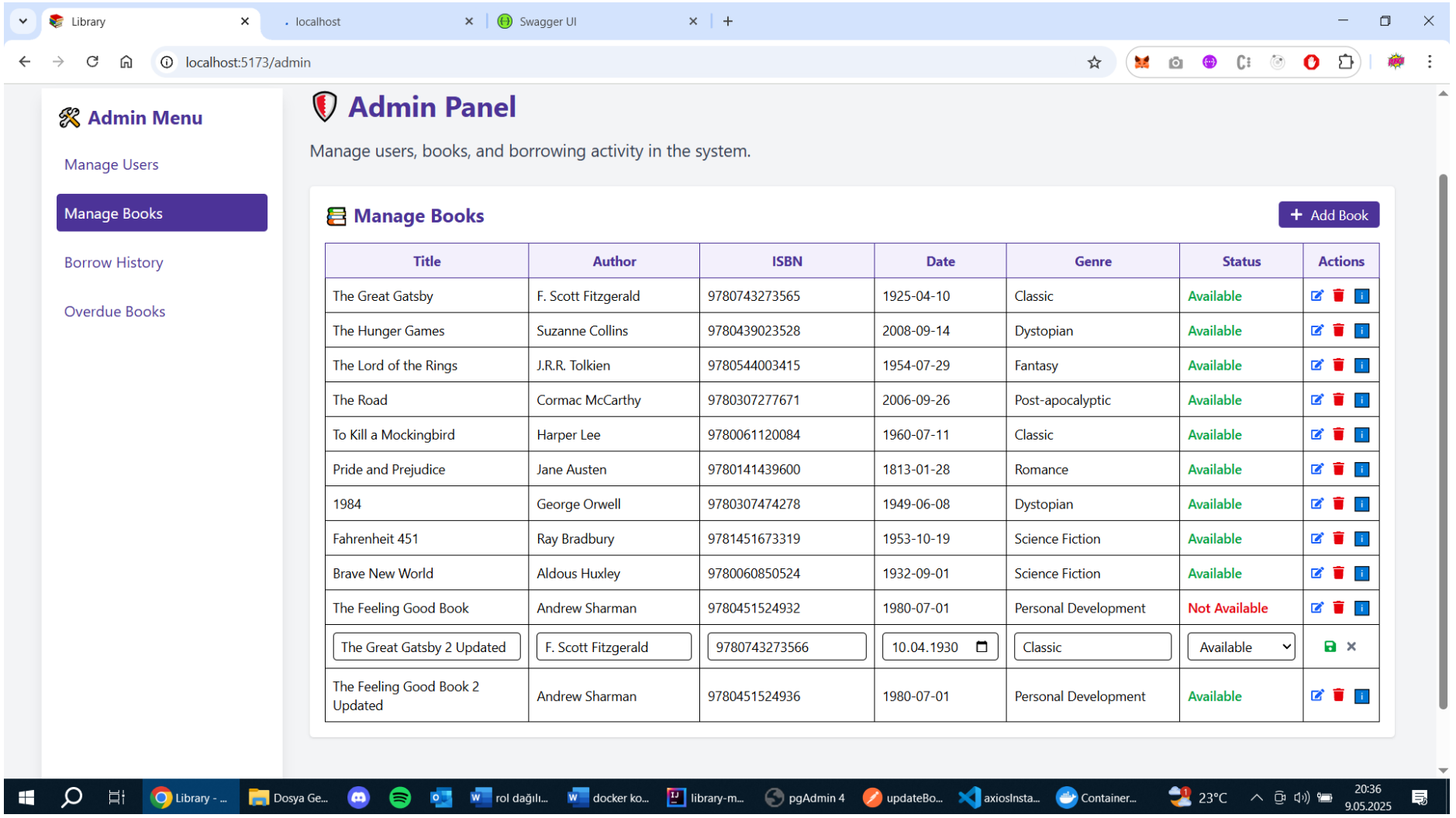
PUT /books/{id} with LIBRARIAN Token – (200 OK)



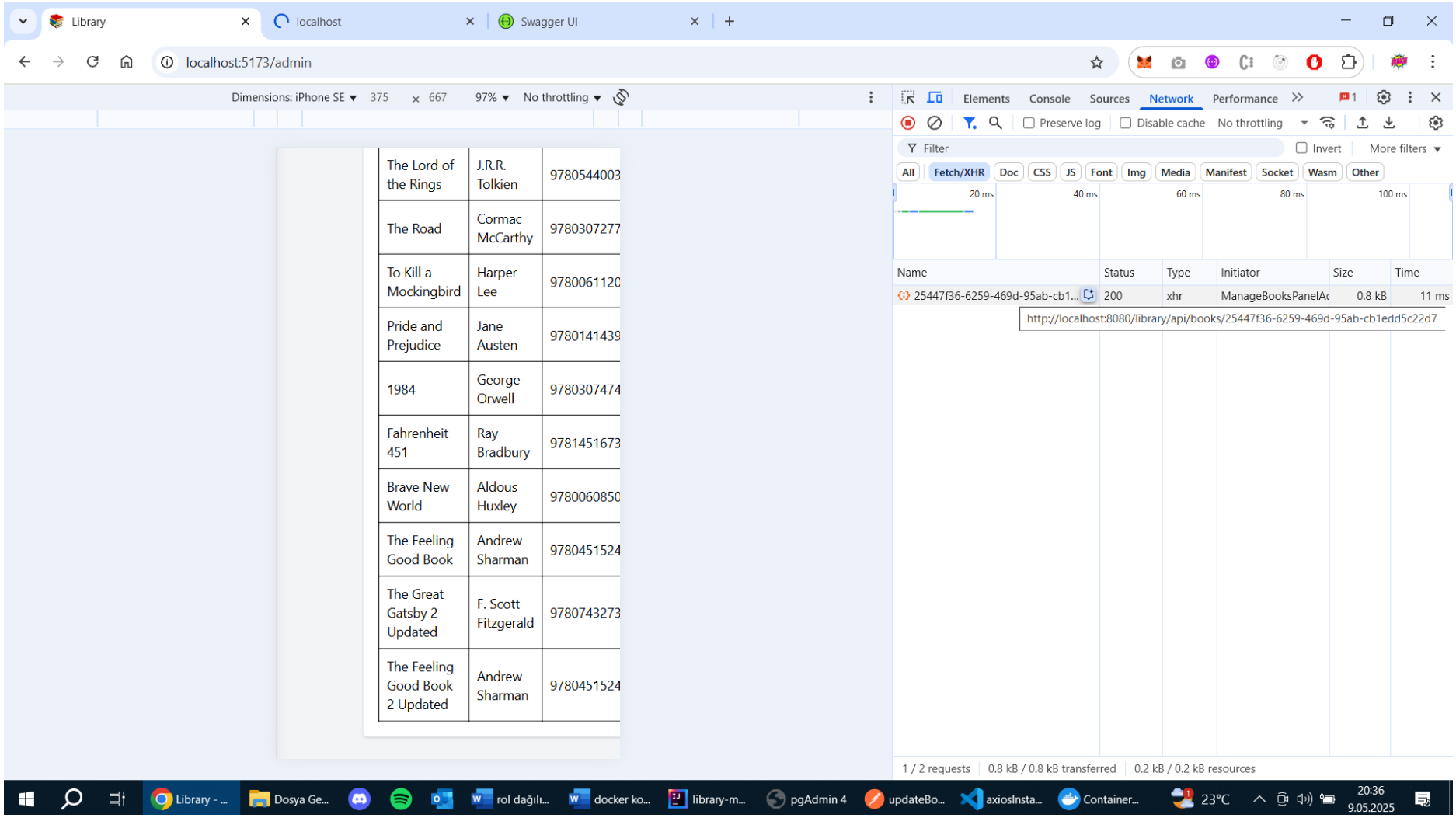
PUT /books/{id} with PATRON Token – (403 Forbidden)



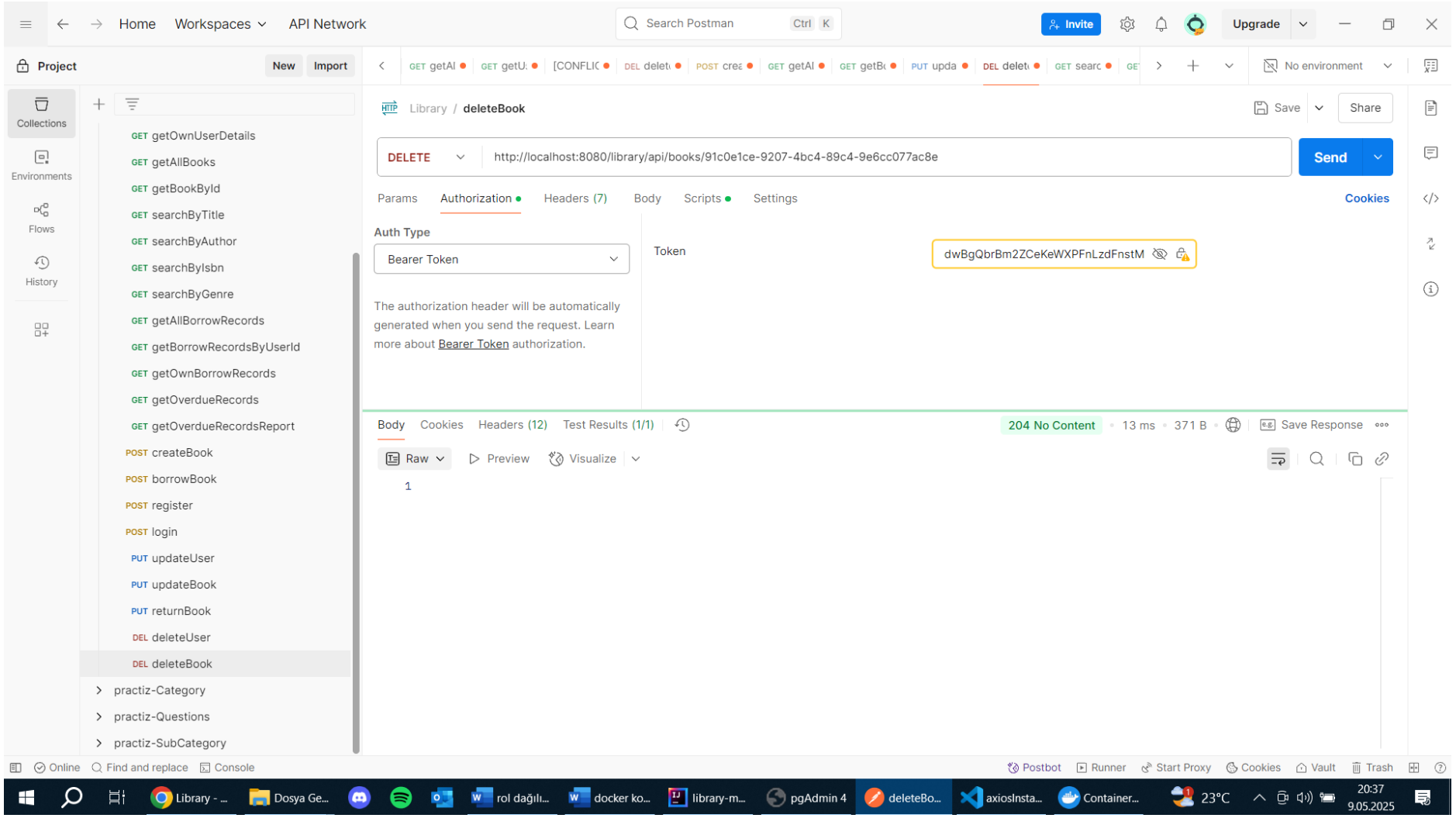
PUT /books/{id} – Admin panel manage books



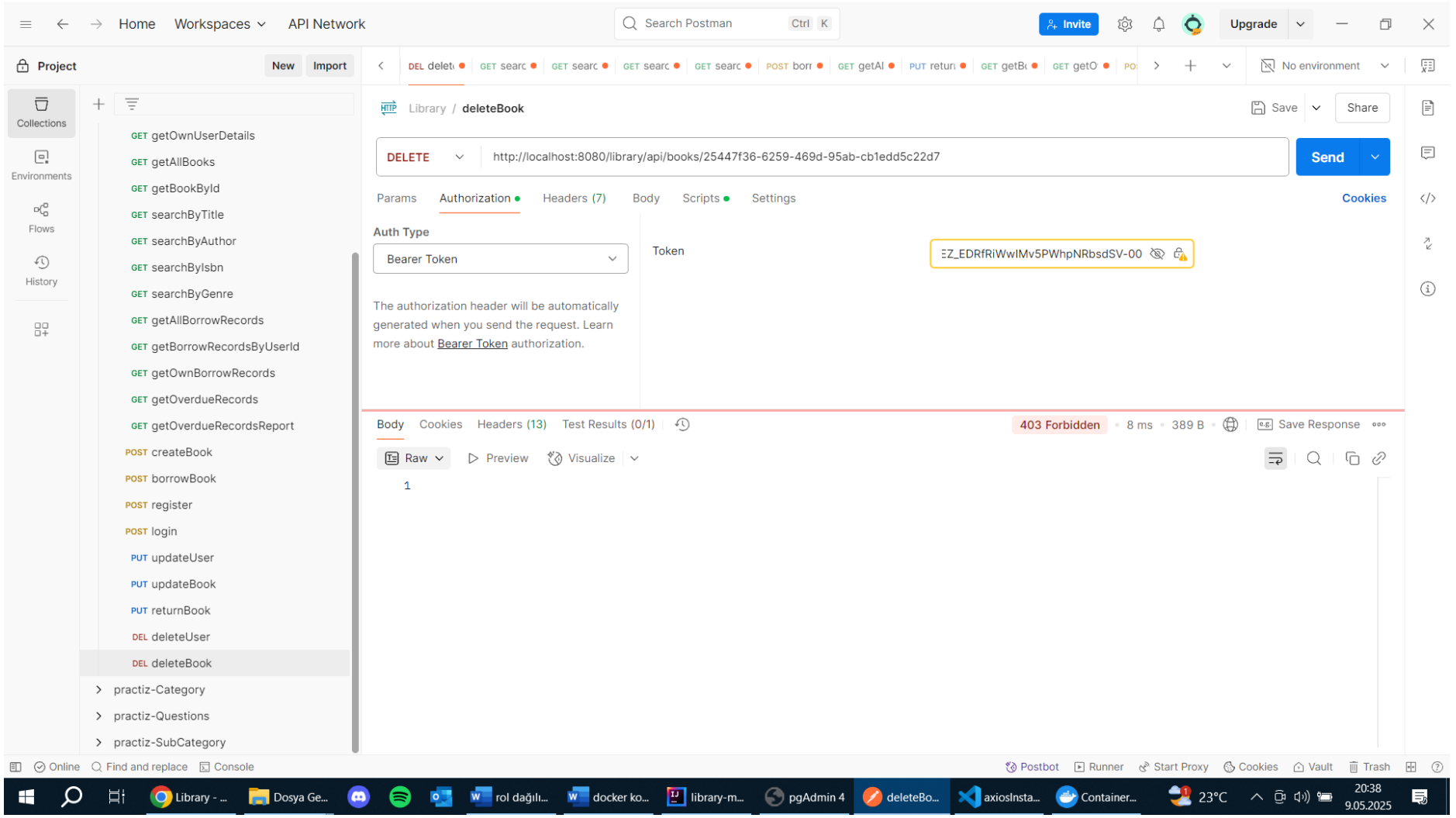
PUT /books/{id} – Admin panel manage books 2



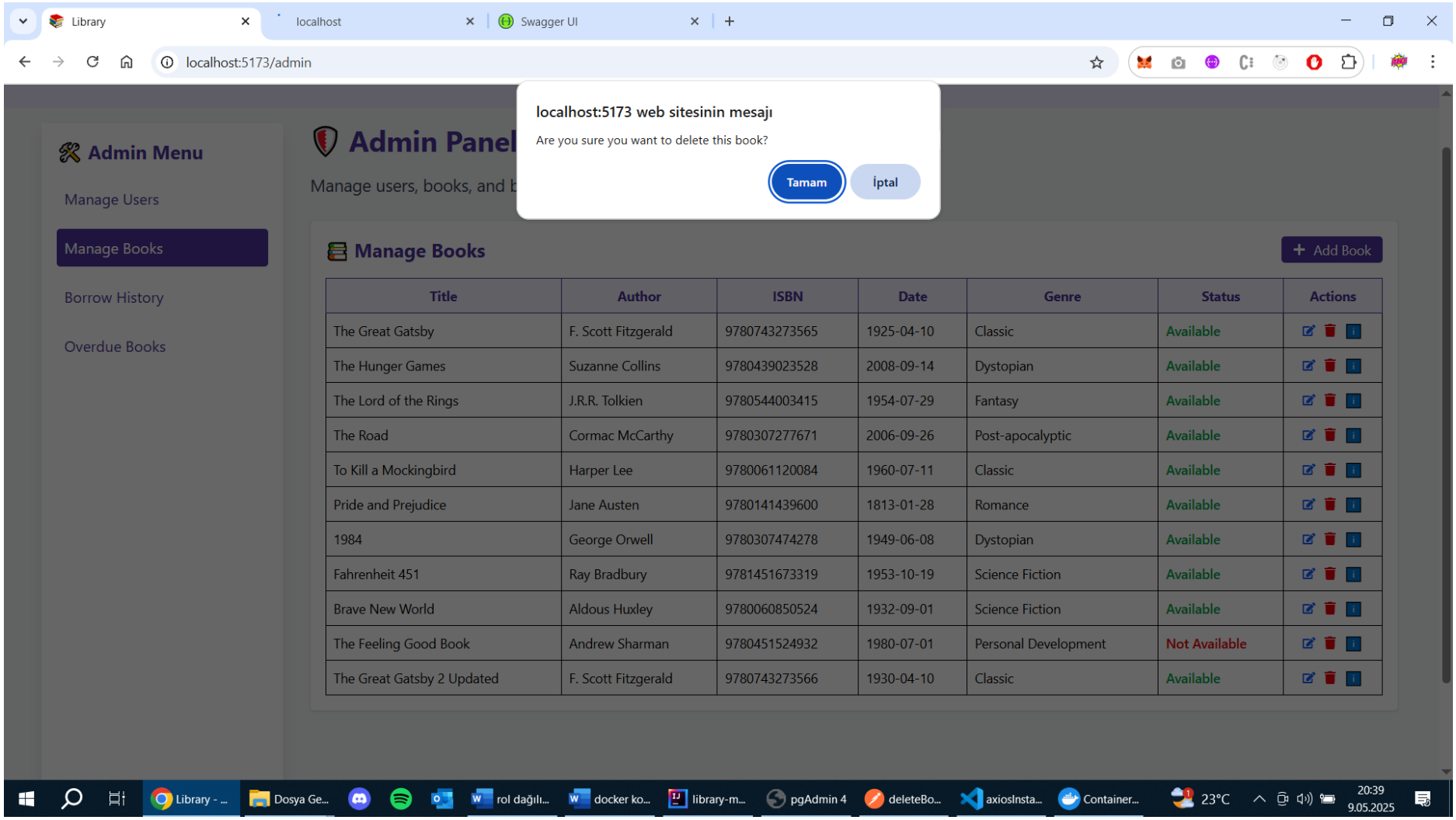
DELETE /books/{id} with LIBRARIAN Token – (200 OK)



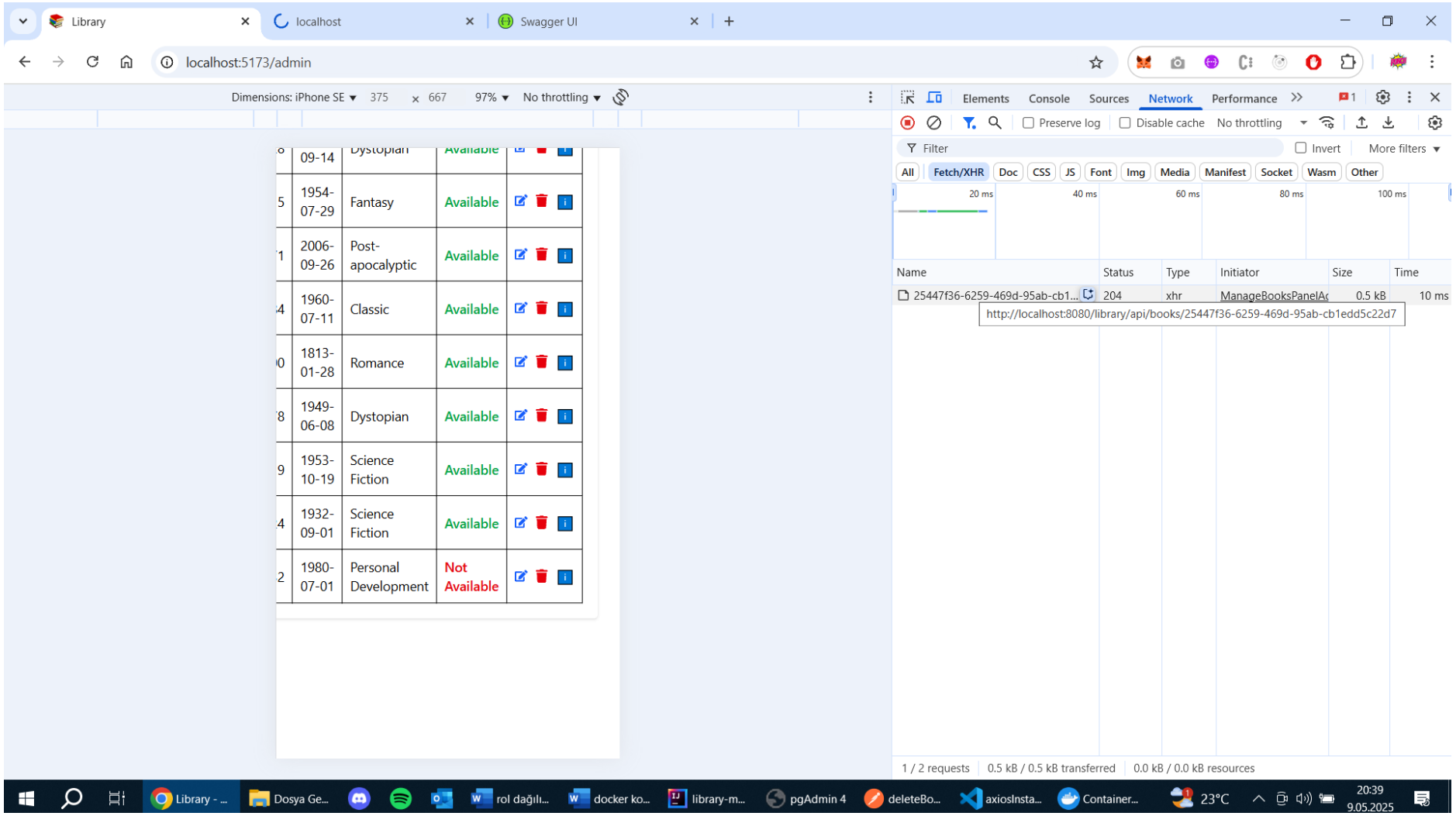
DELETE /books/{id} with PATRON Token – (403 Forbidden)



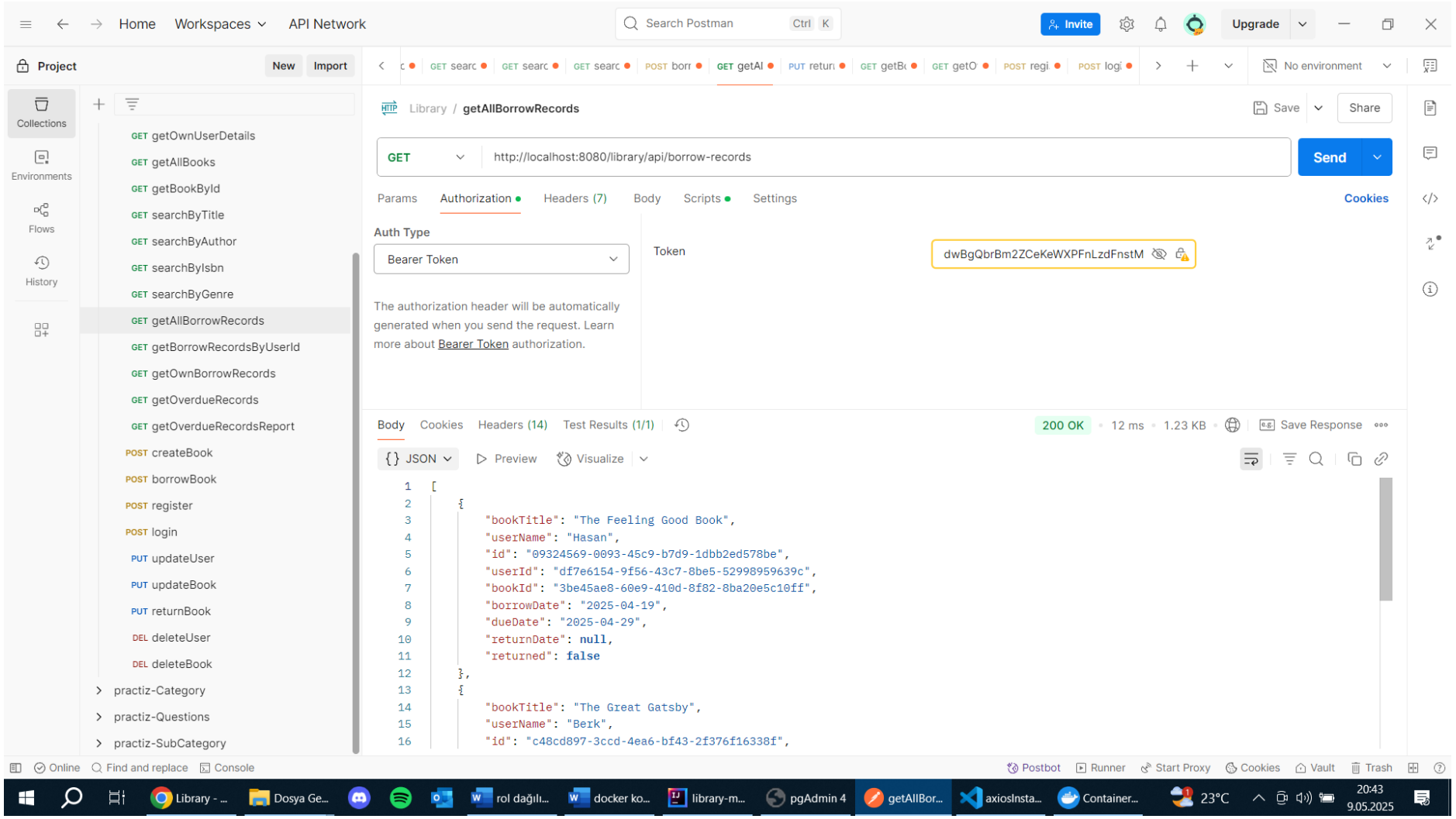
DELETE /books/{id} – Admin panel manage books



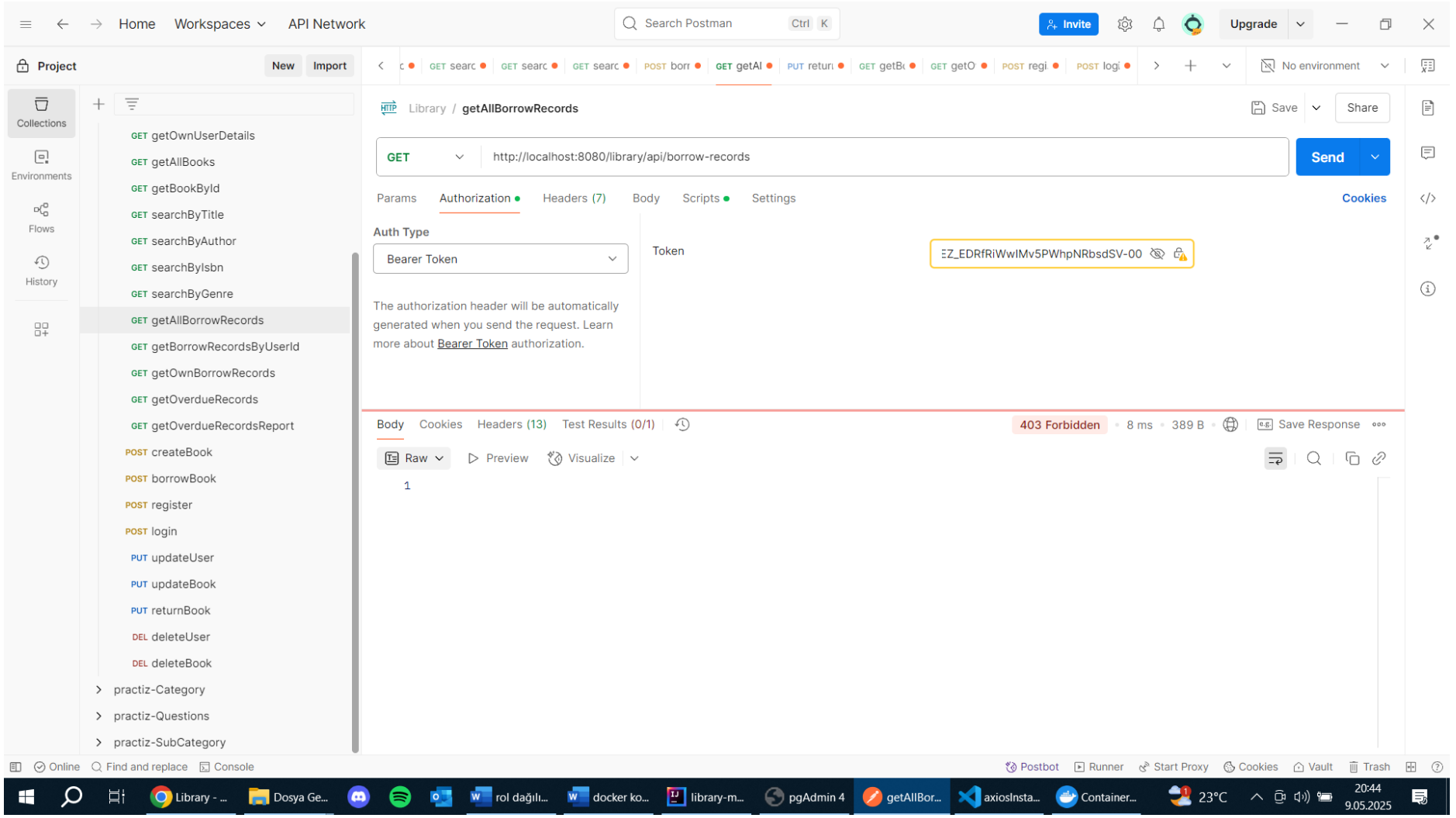
DELETE /books/{id} – Admin panel manage books 2



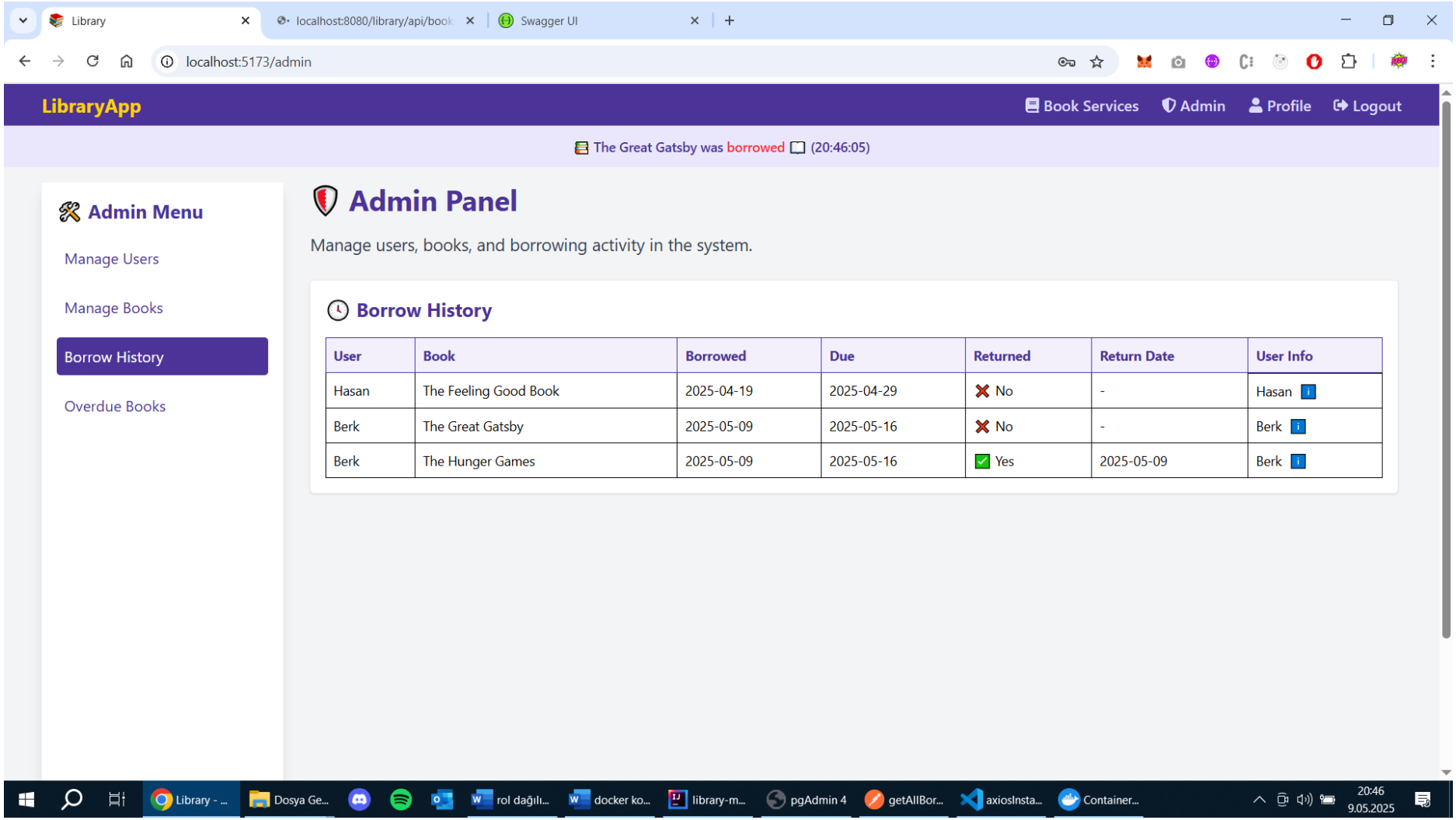
GET /borrow-records with LIBRARIAN Token – (200 OK)



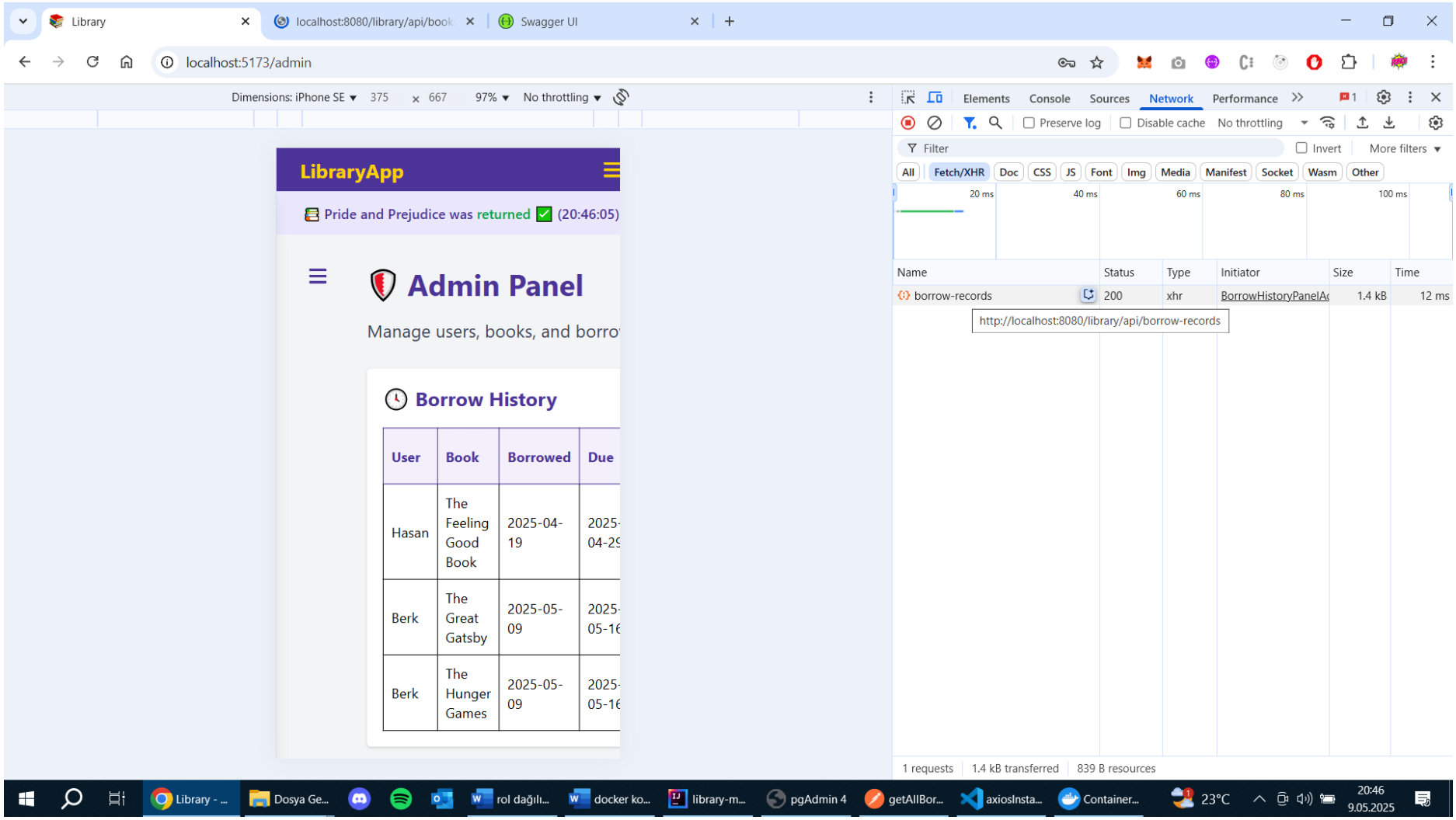
GET /borrow-records with PATRON Token – (403 Forbidden)



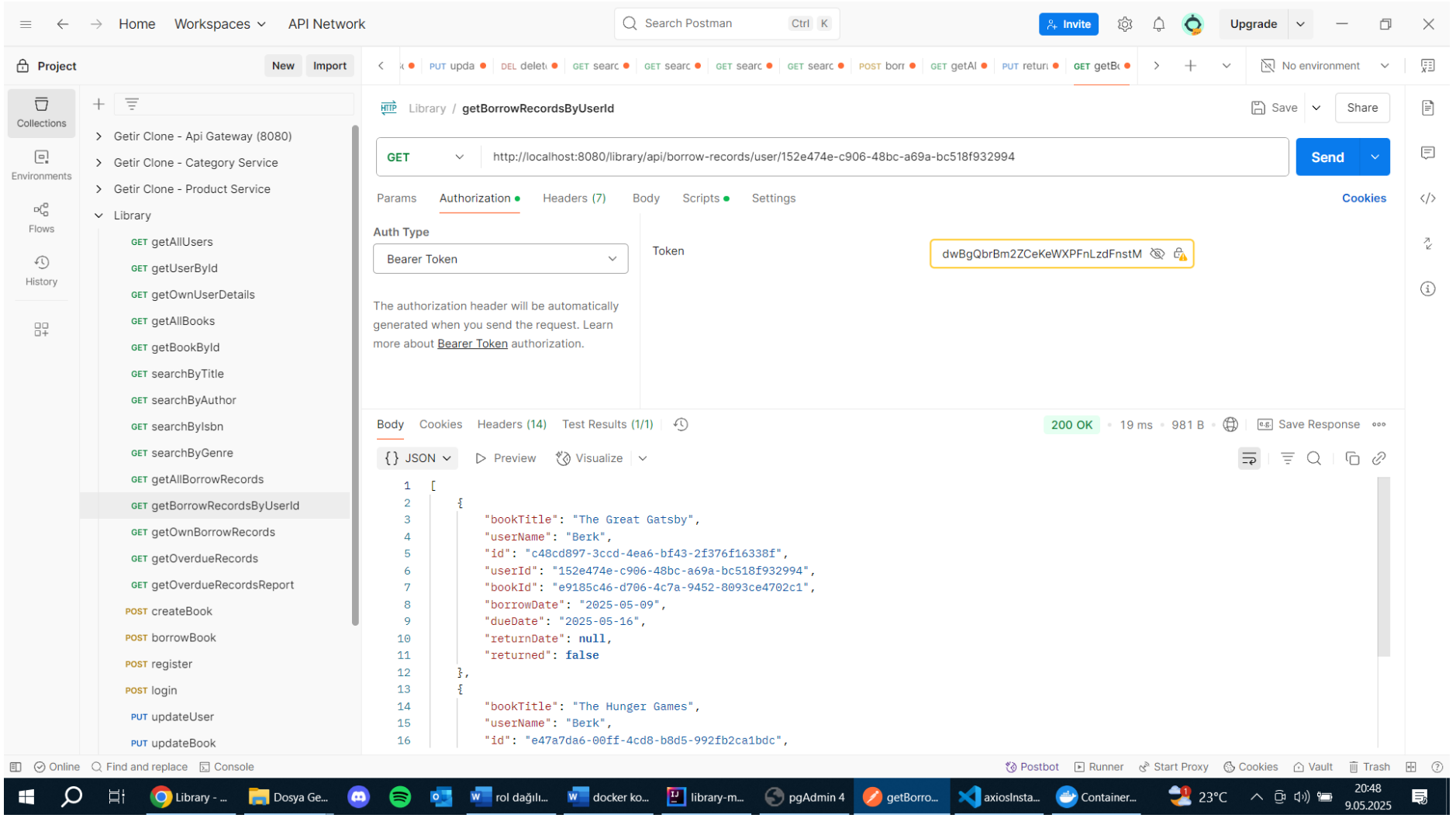
GET /borrow-records – Admin panel borrow history



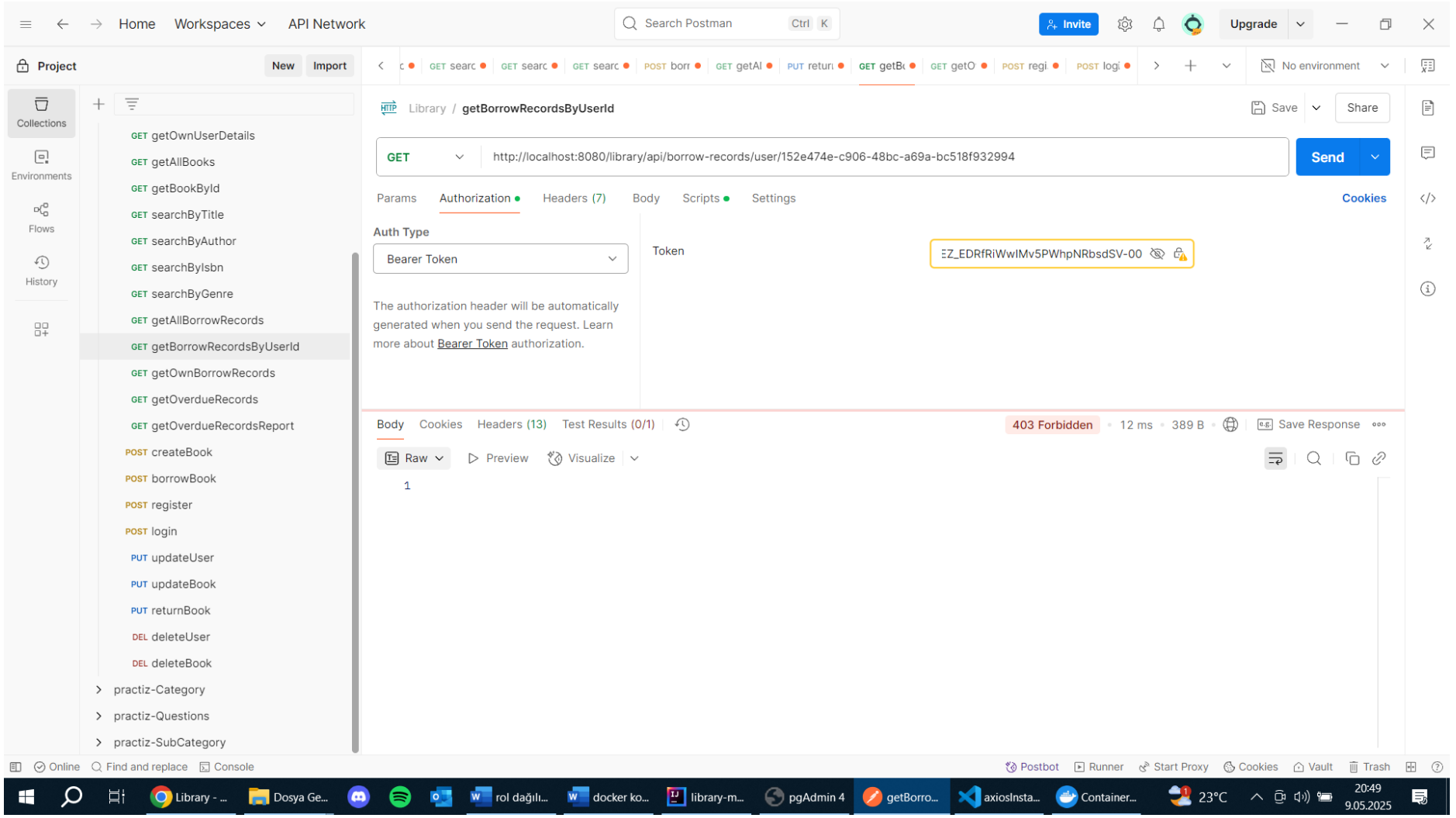
GET /borrow-records – Admin panel borrow history 2



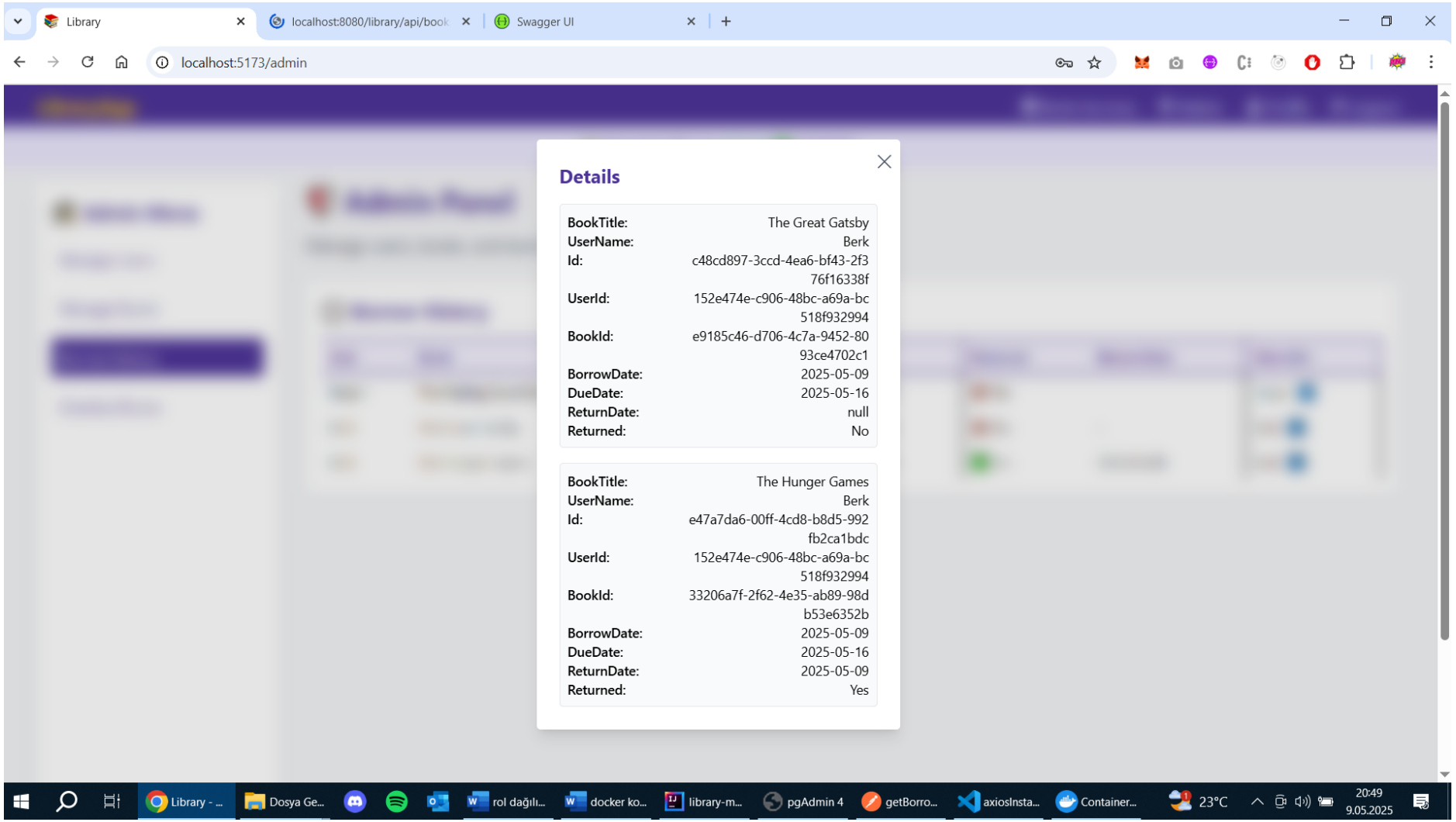
GET /borrow-records/user/{userId} with LIBRARIAN Token – (200 OK)



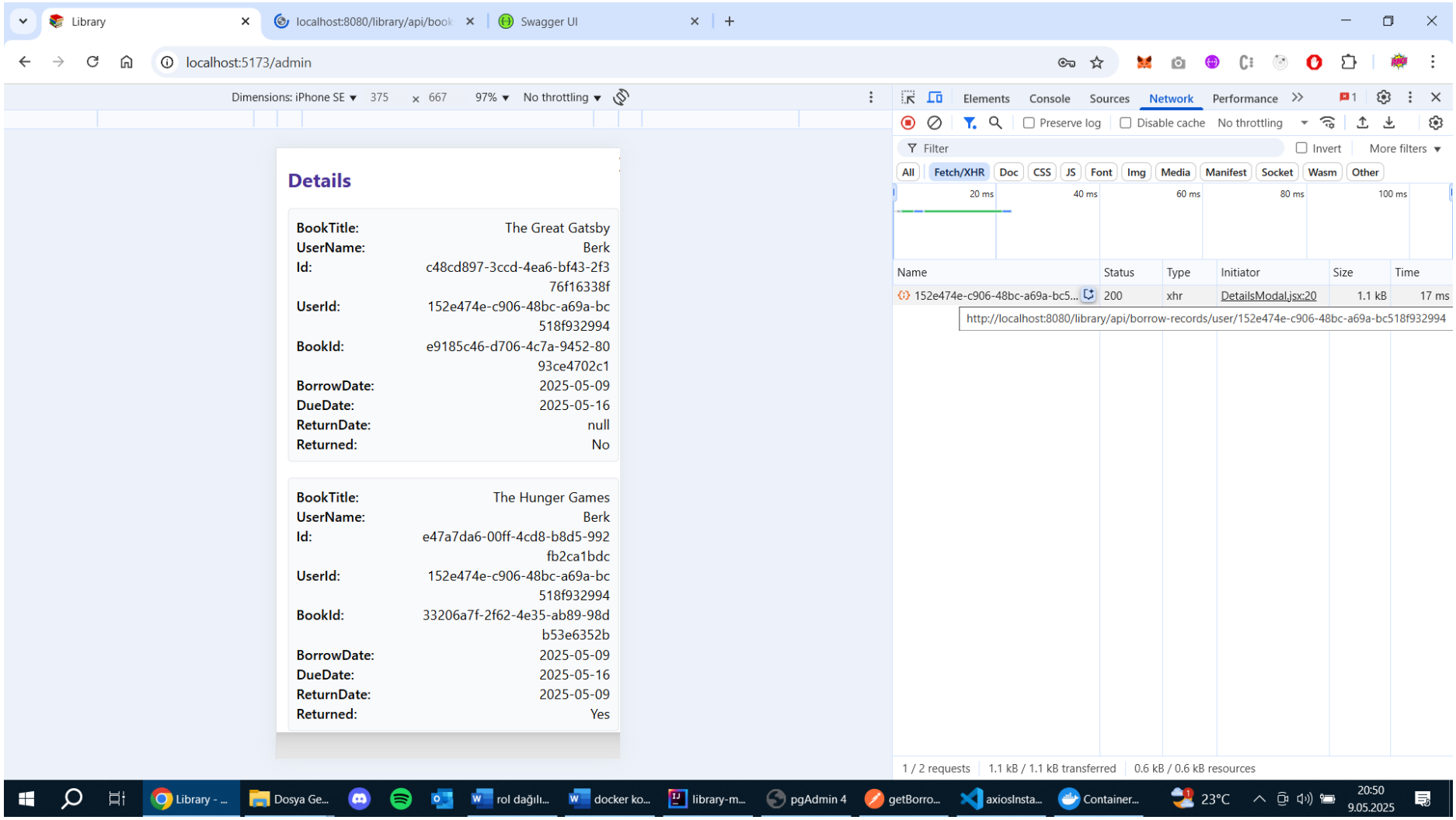
GET /borrow-records/user/{userId} with PATRON Token – (403 Forbidden)



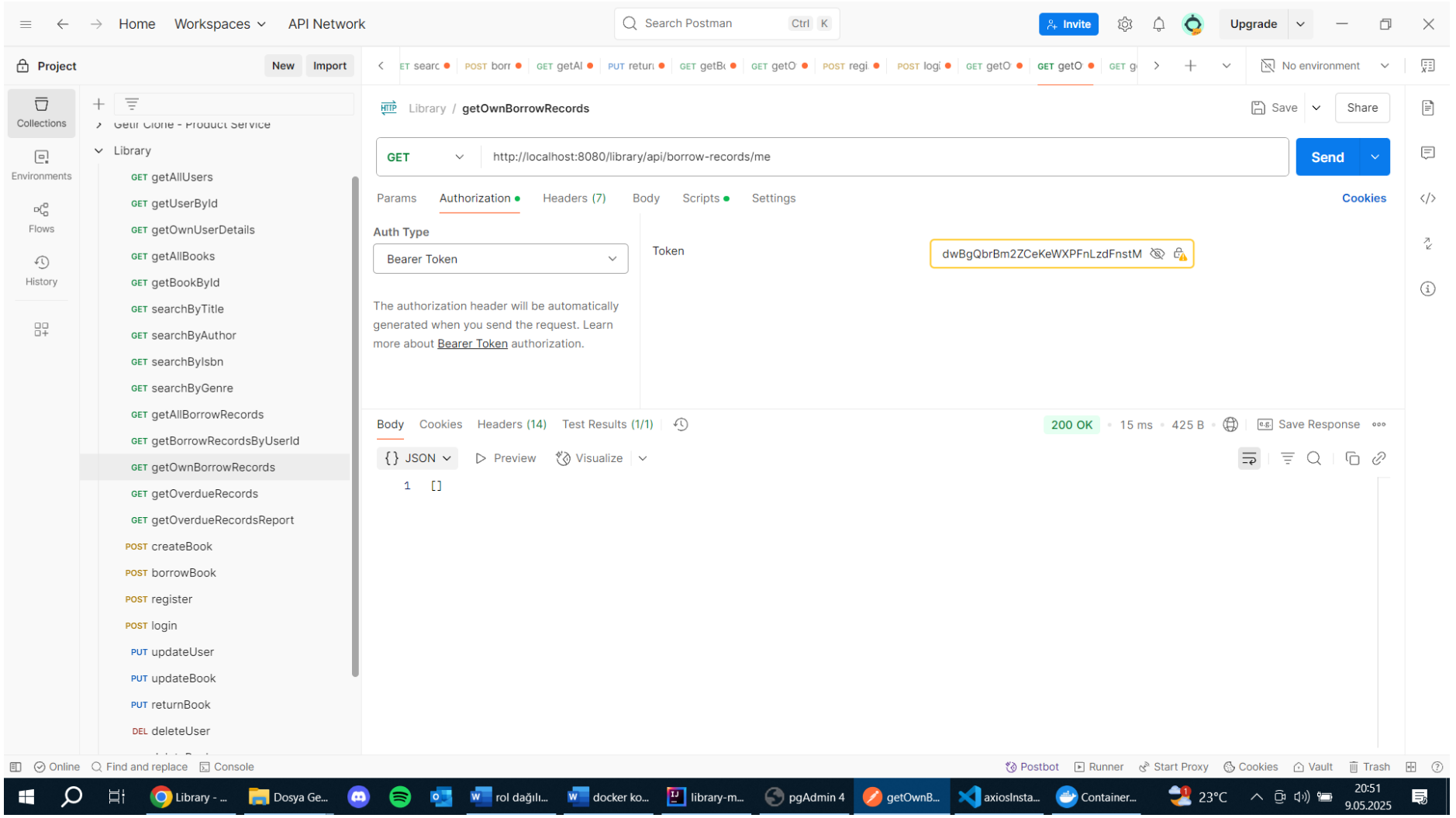
GET /borrow-records/user/{userId} – Admin panel borrow history details



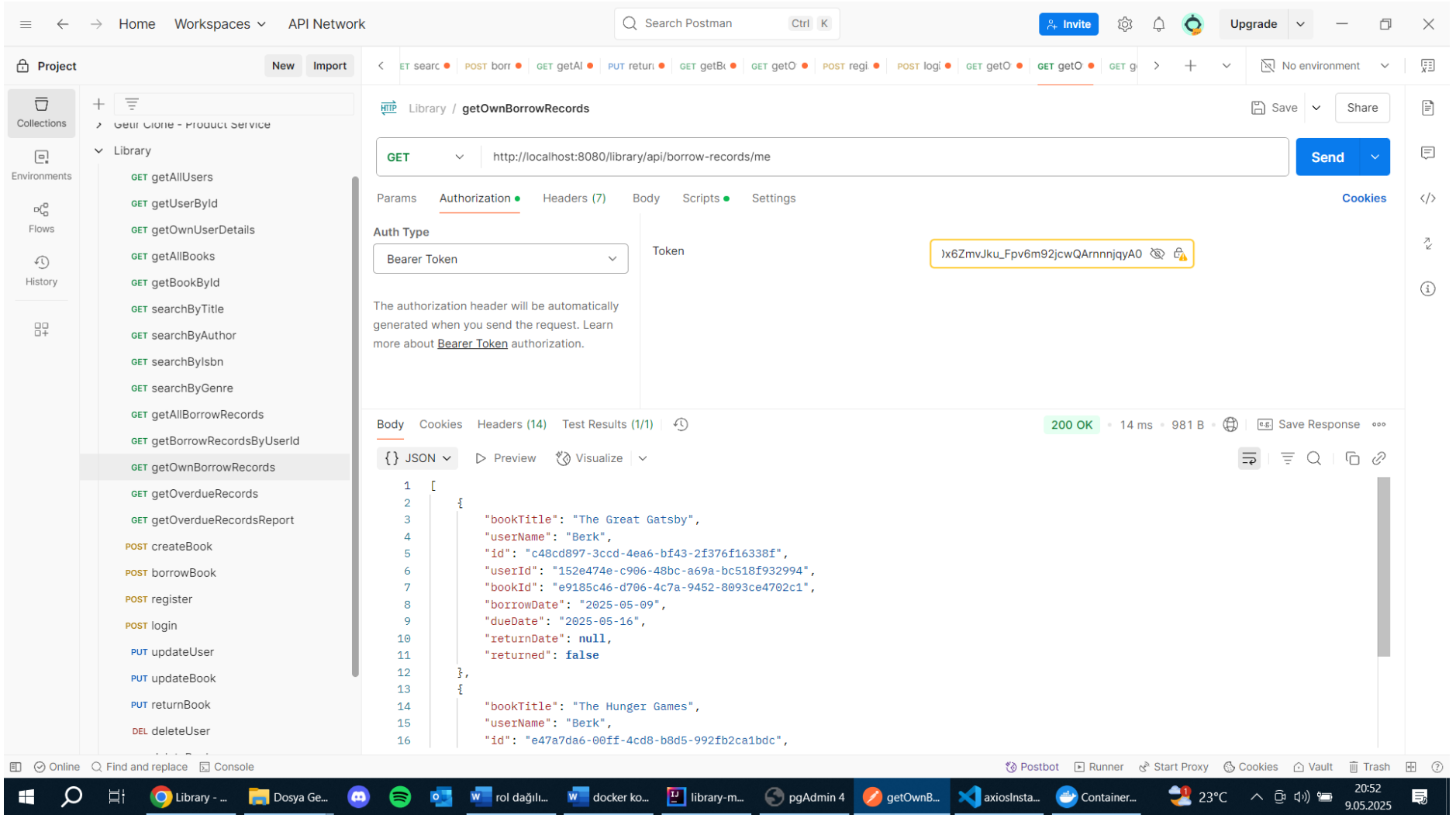
GET /borrow-records/user/{userId} – Admin panel borrow history details 2



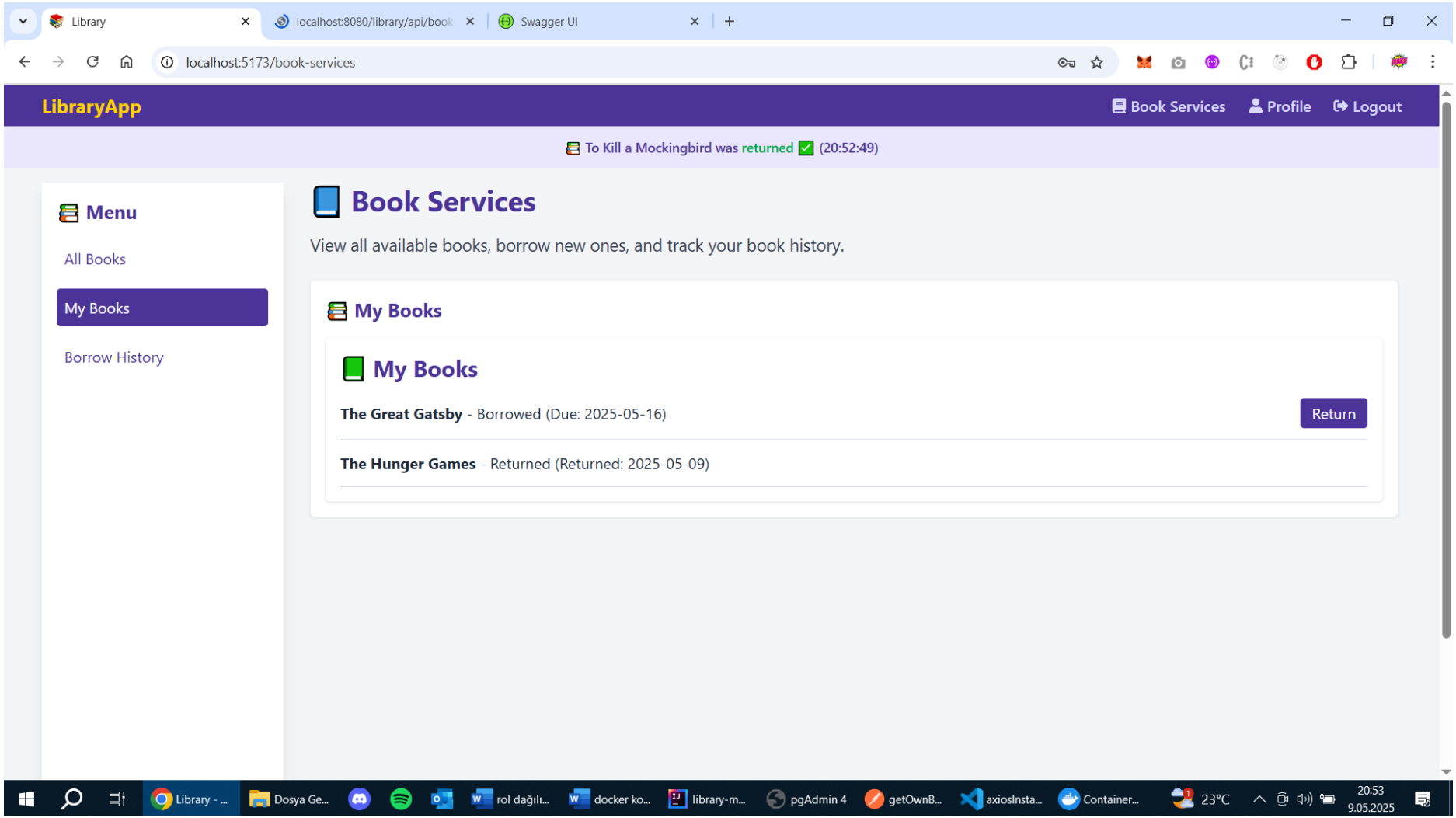
GET /borrow-records/me with LIBRARIAN Token – (200 OK)



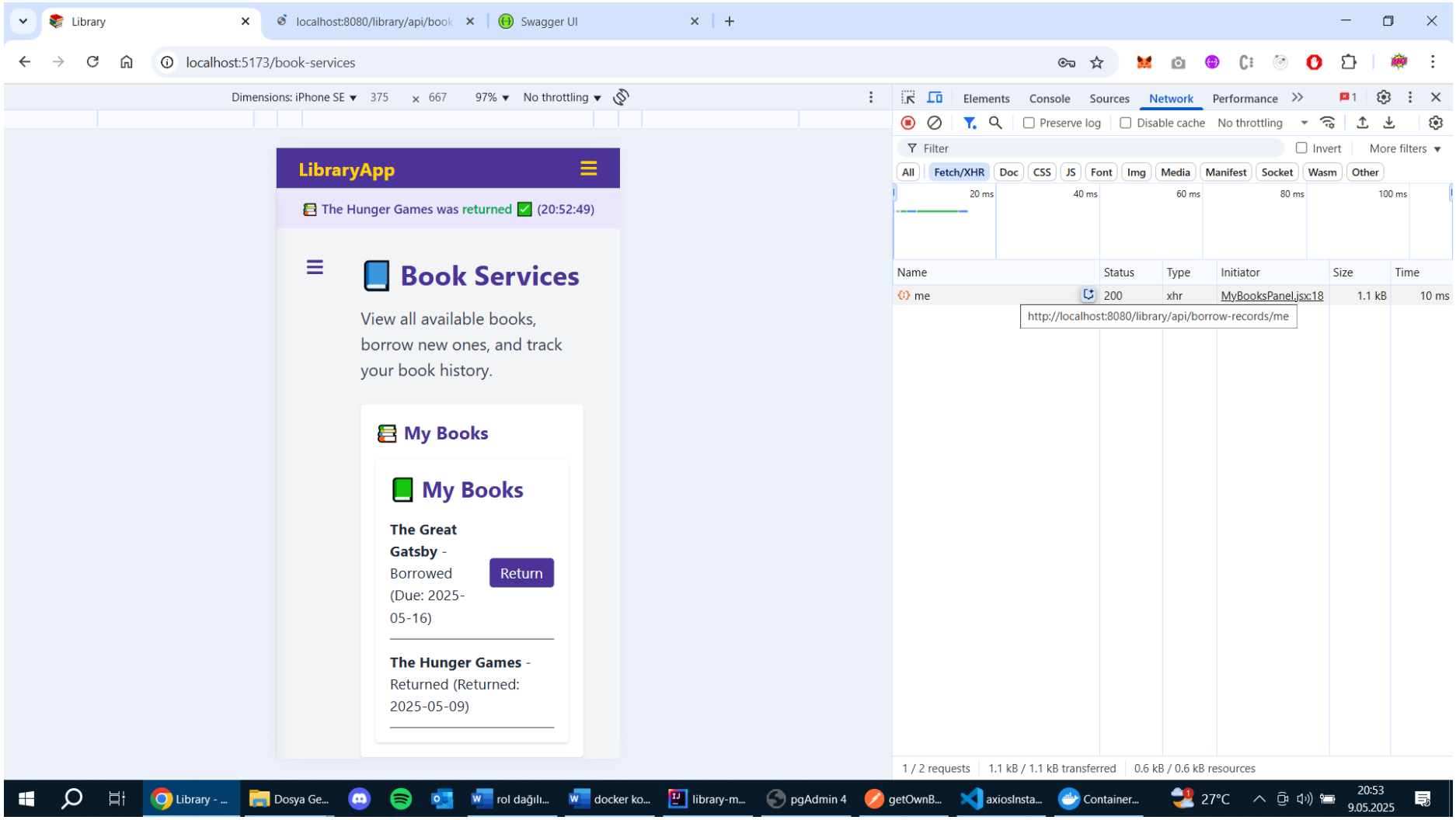
GET /borrow-records/me with PATRON Token – (200 OK)



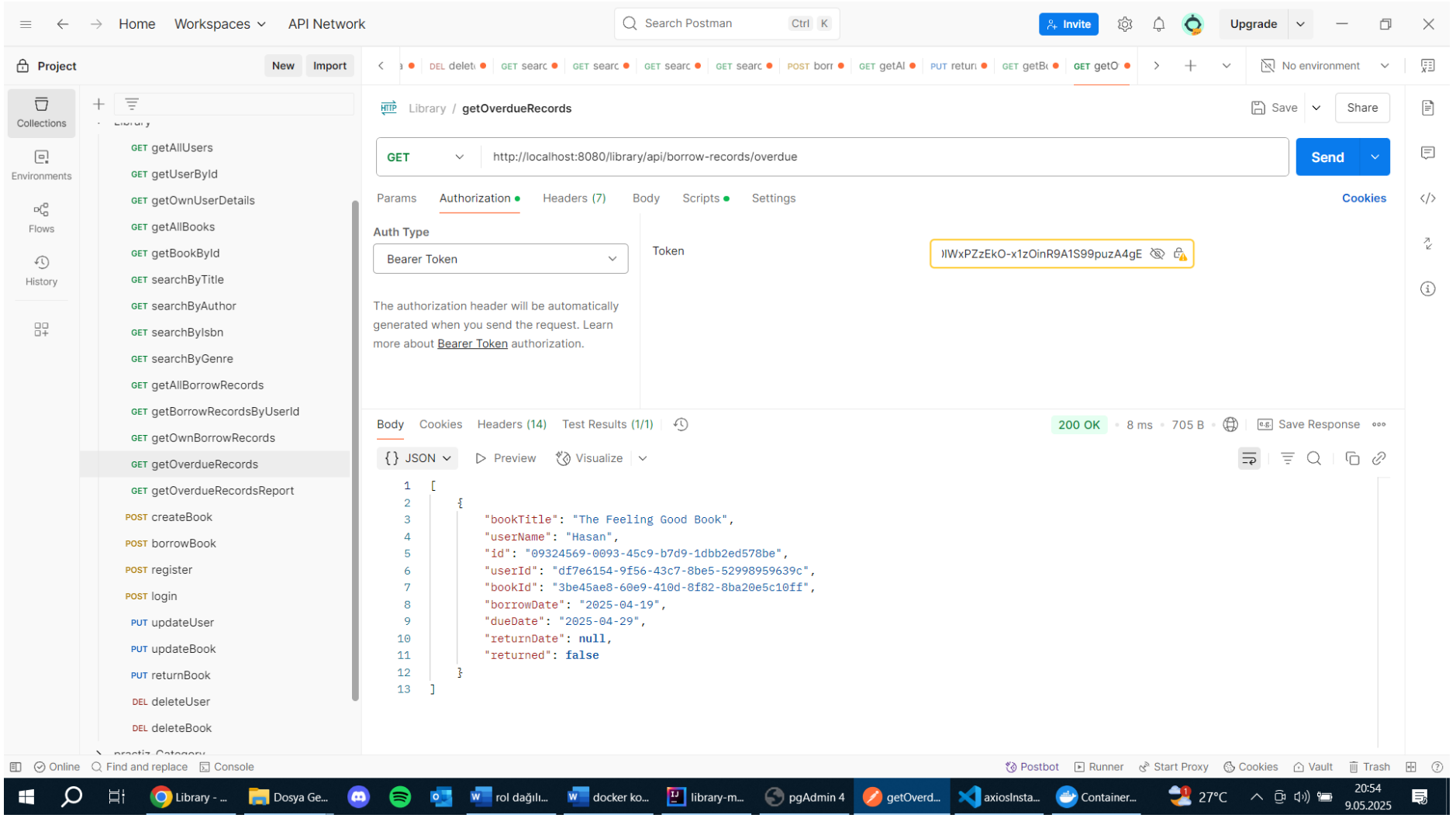
GET /borrow-records/me – My Books



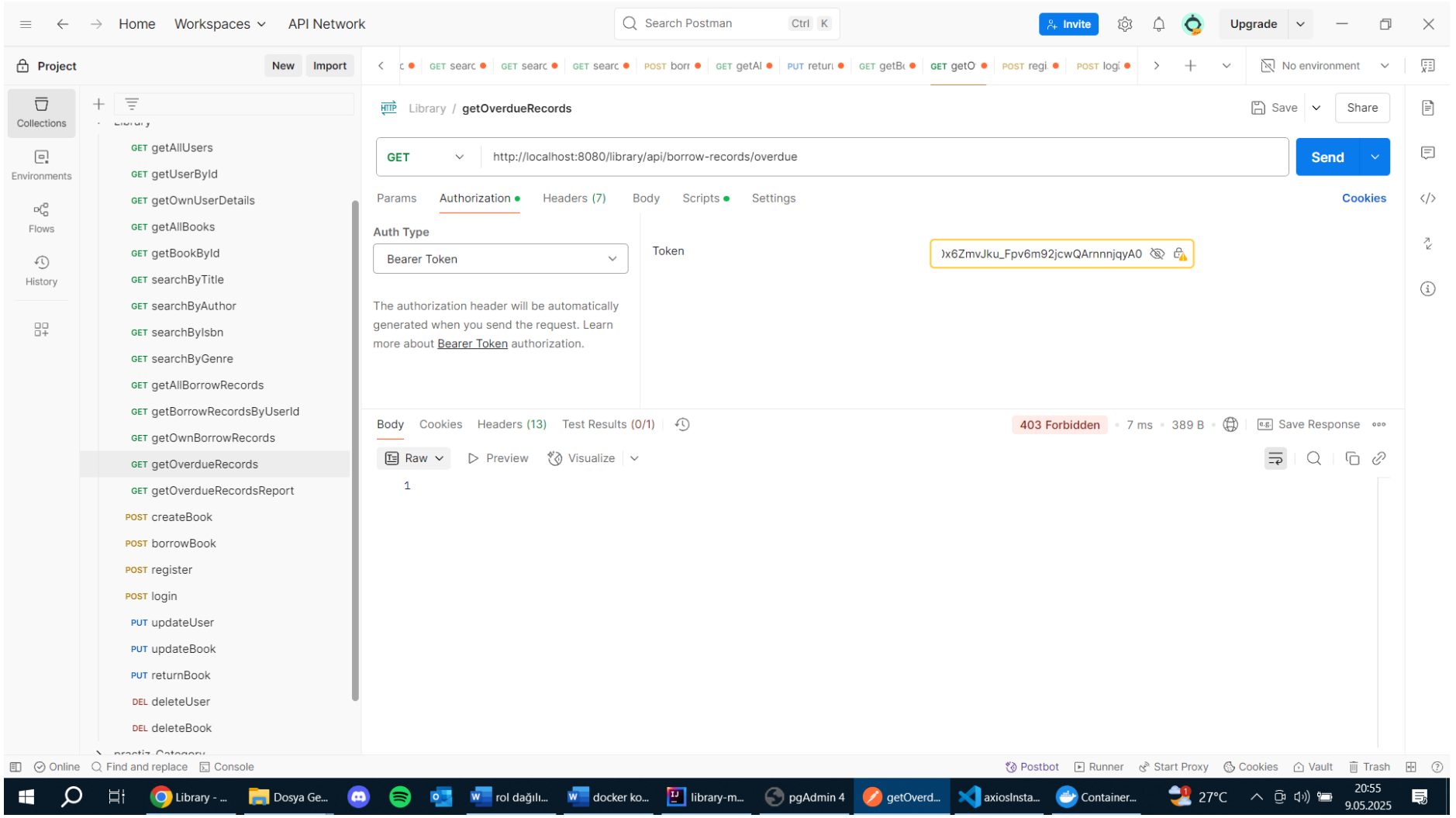
GET /borrow-records/me – My Books 2



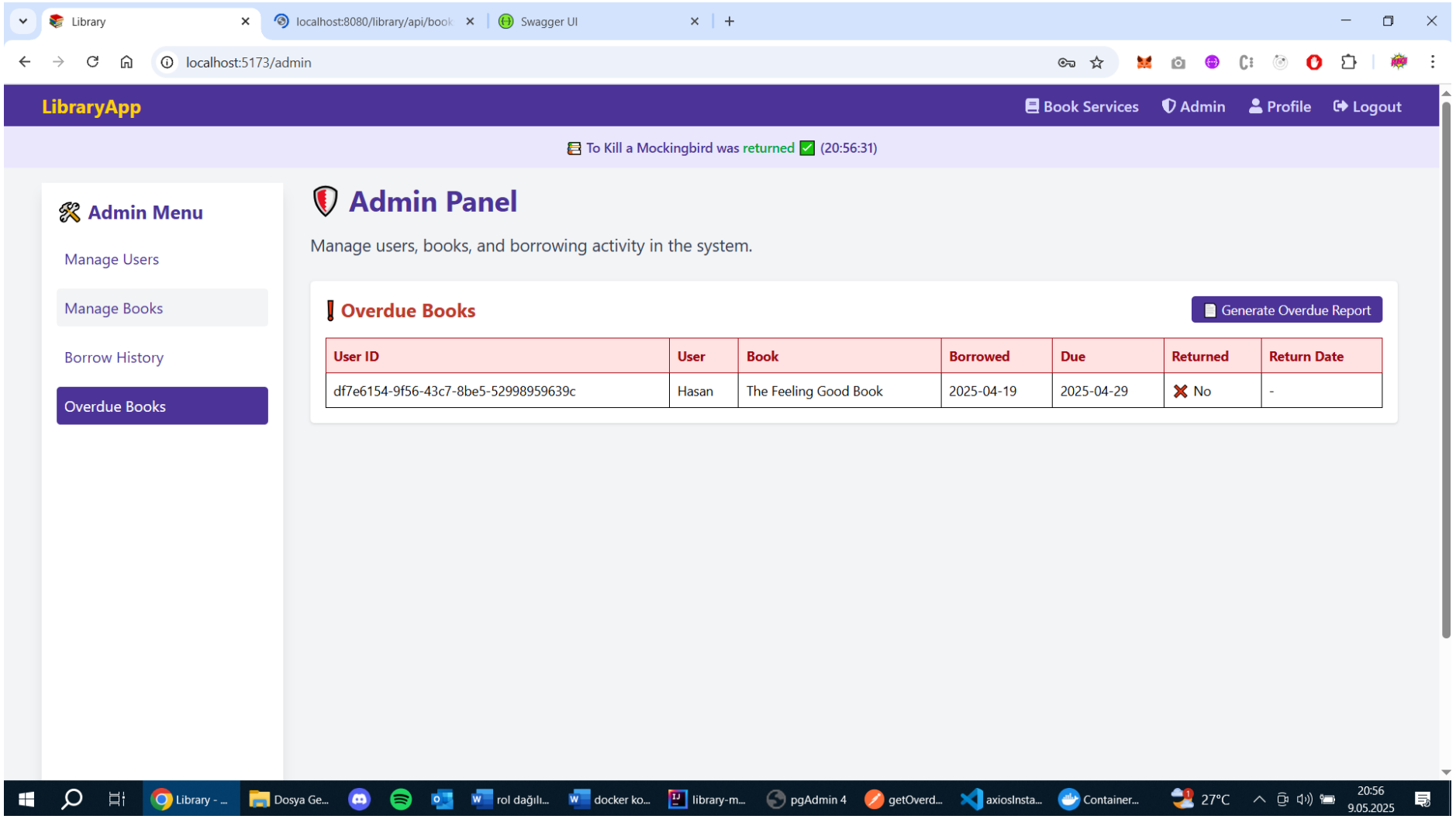
GET /borrow-records/overdue with LIBRARIAN Token – (200 OK)



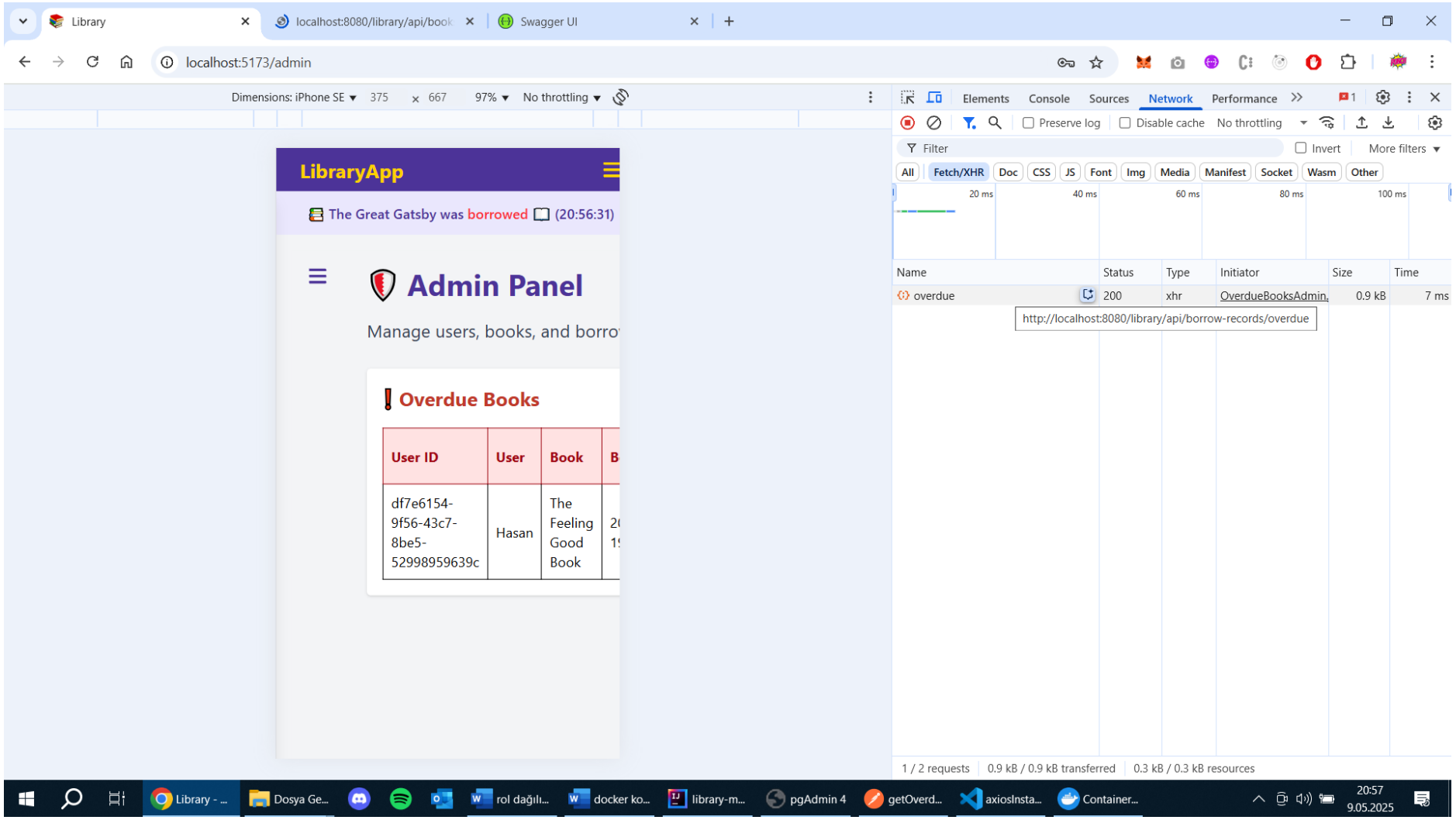
GET /borrow-records/overdue with PATRON Token – (403 Forbidden)



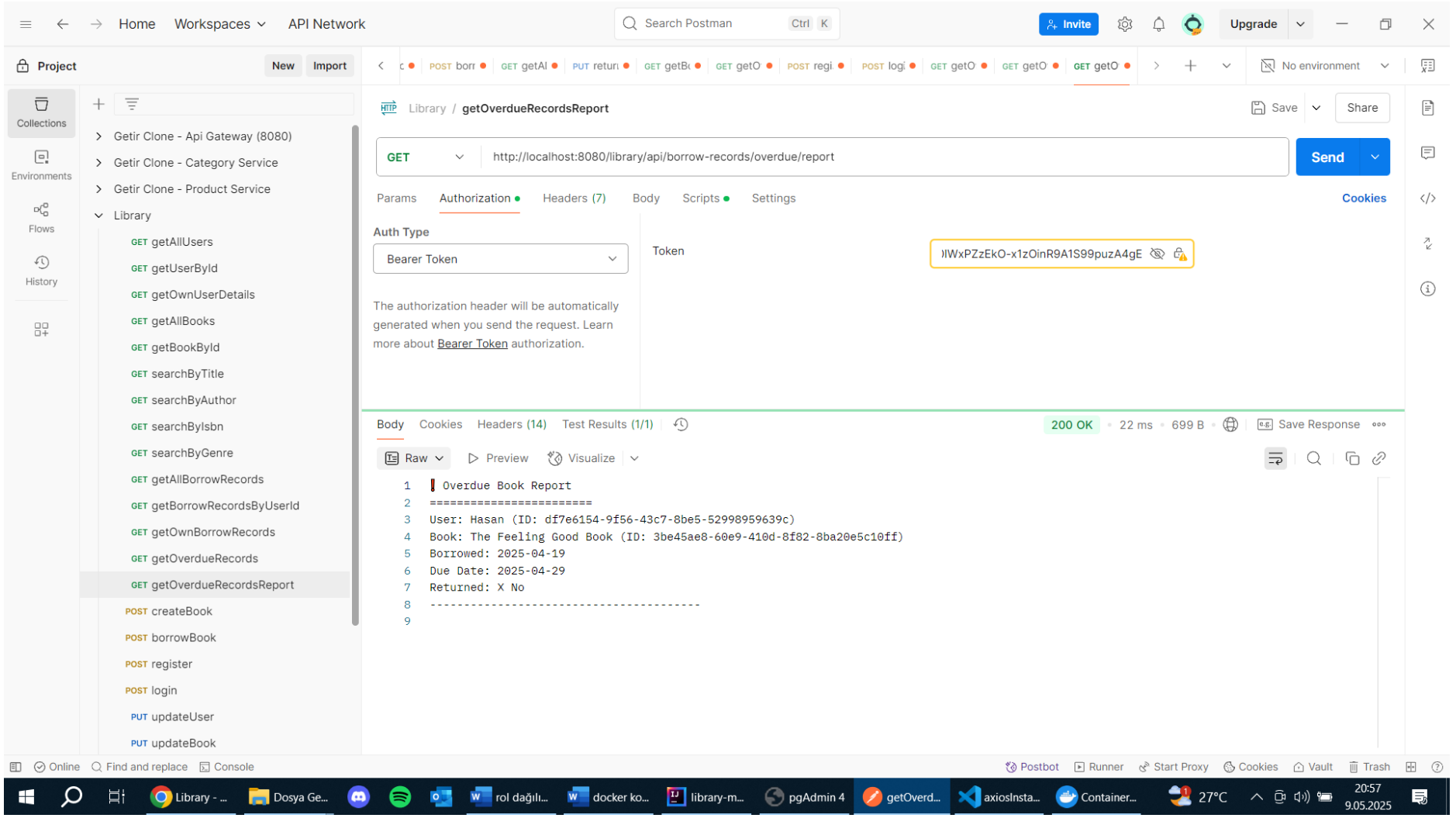
GET /borrow-records/overdue – Admin panel overdue books



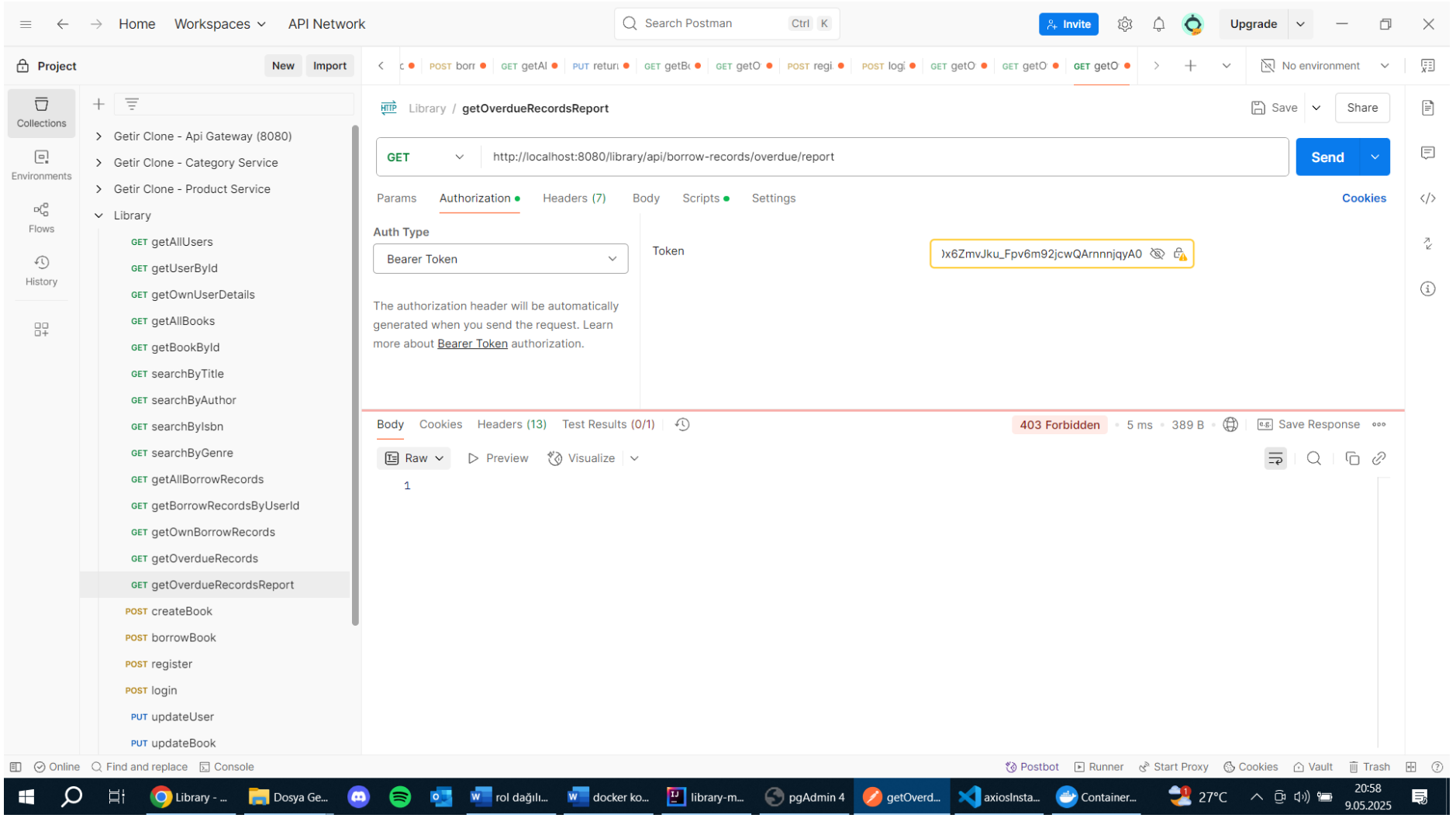
GET /borrow-records/overdue – Admin panel overdue books 2



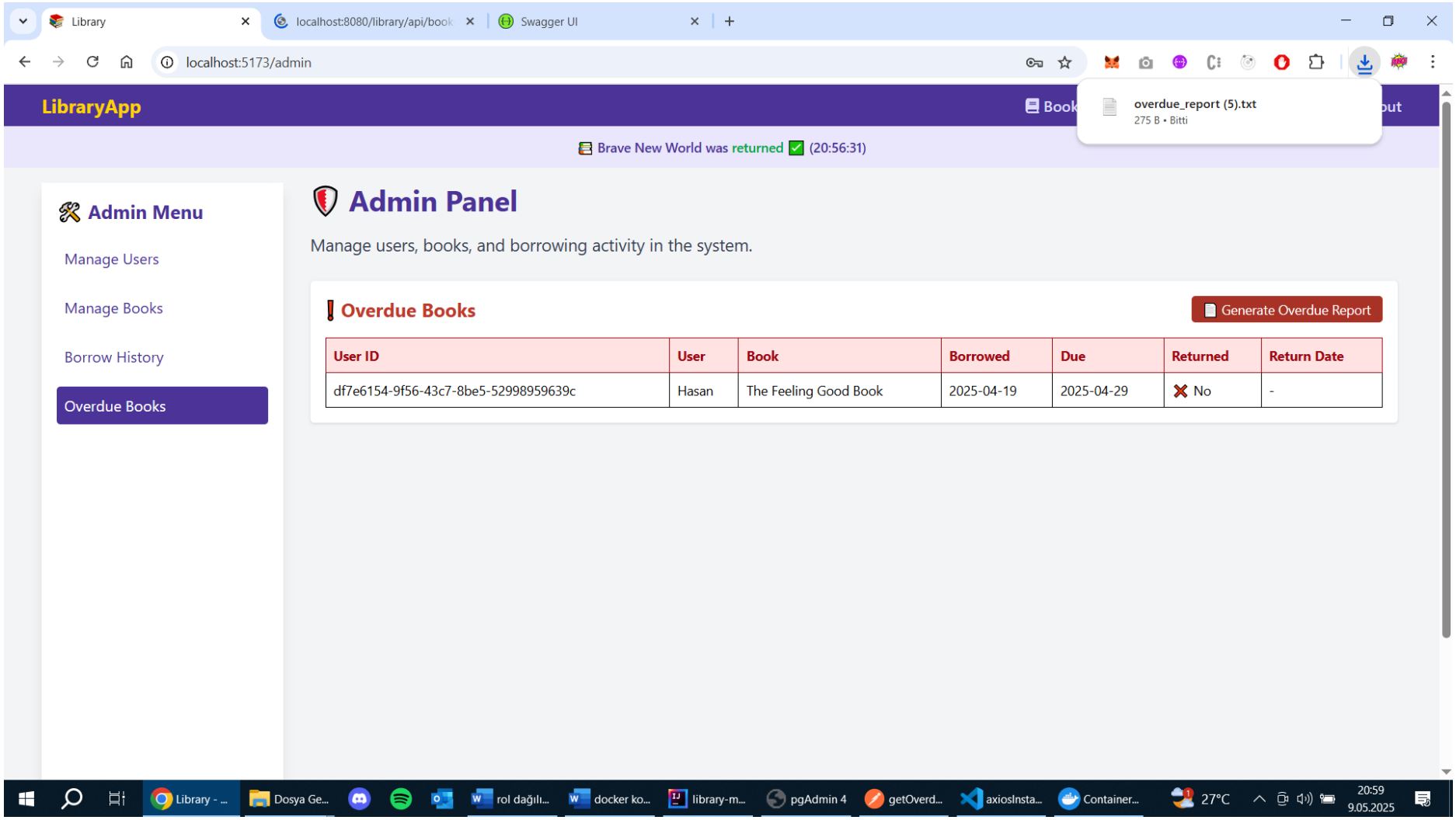
GET /borrow-records/overdue/report with LIBRARIAN Token – (200 OK)



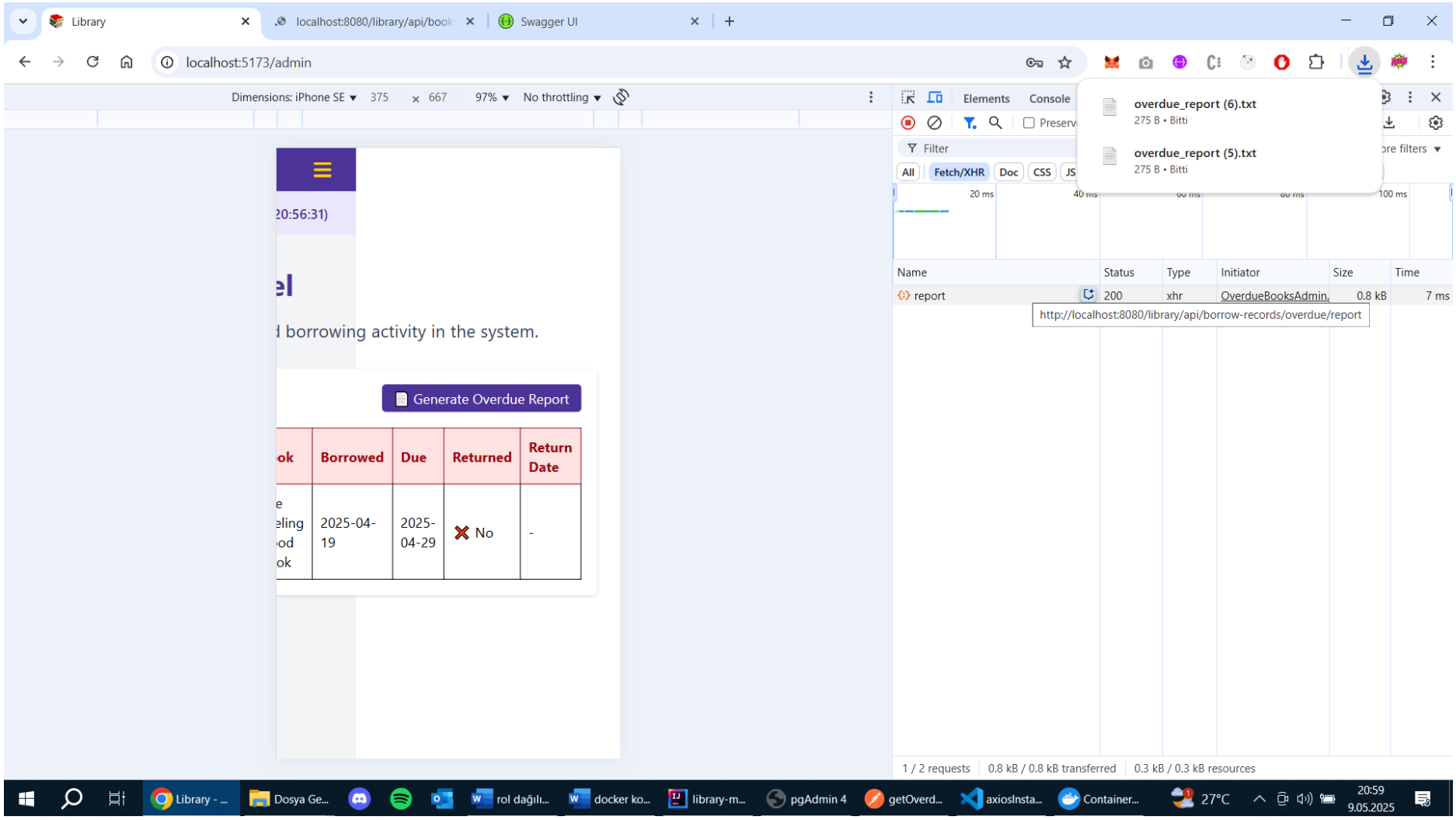
GET /borrow-records/overdue/report with PATRON Token – (403 Forbidden)



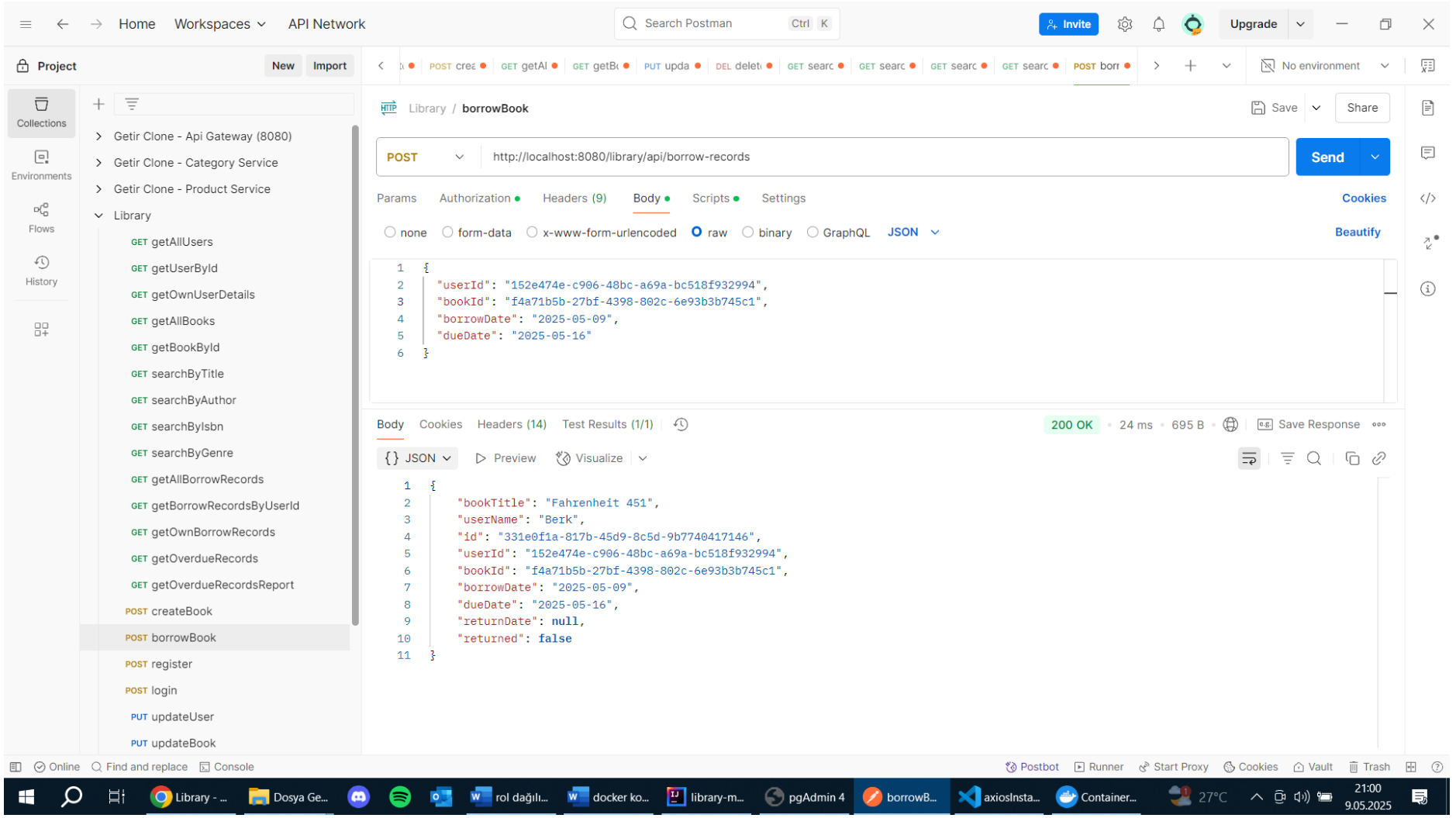
GET /borrow-records/overdue/report – Admin panel overdue books



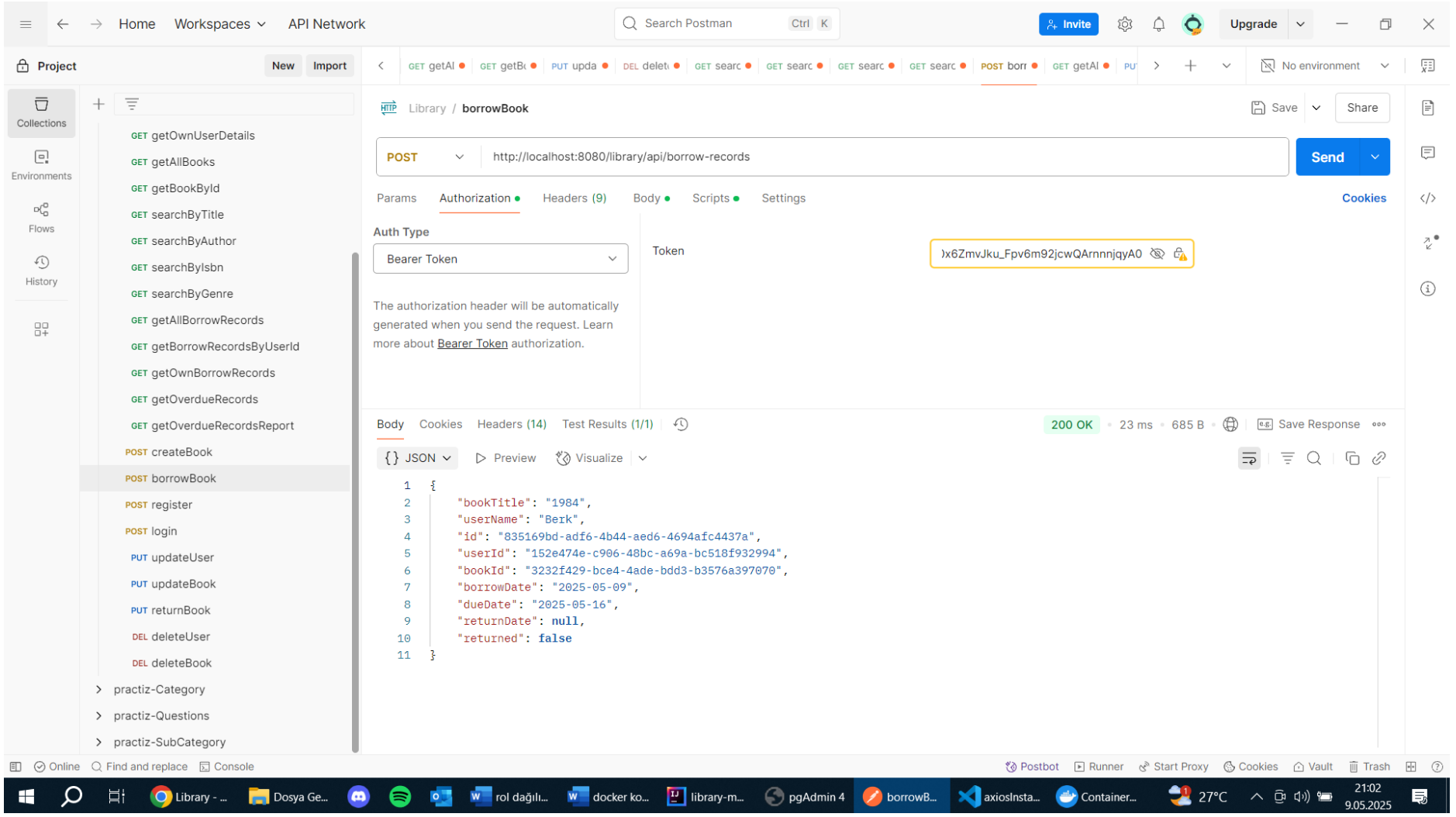
GET /borrow-records/overdue/report – Admin panel overdue books 2



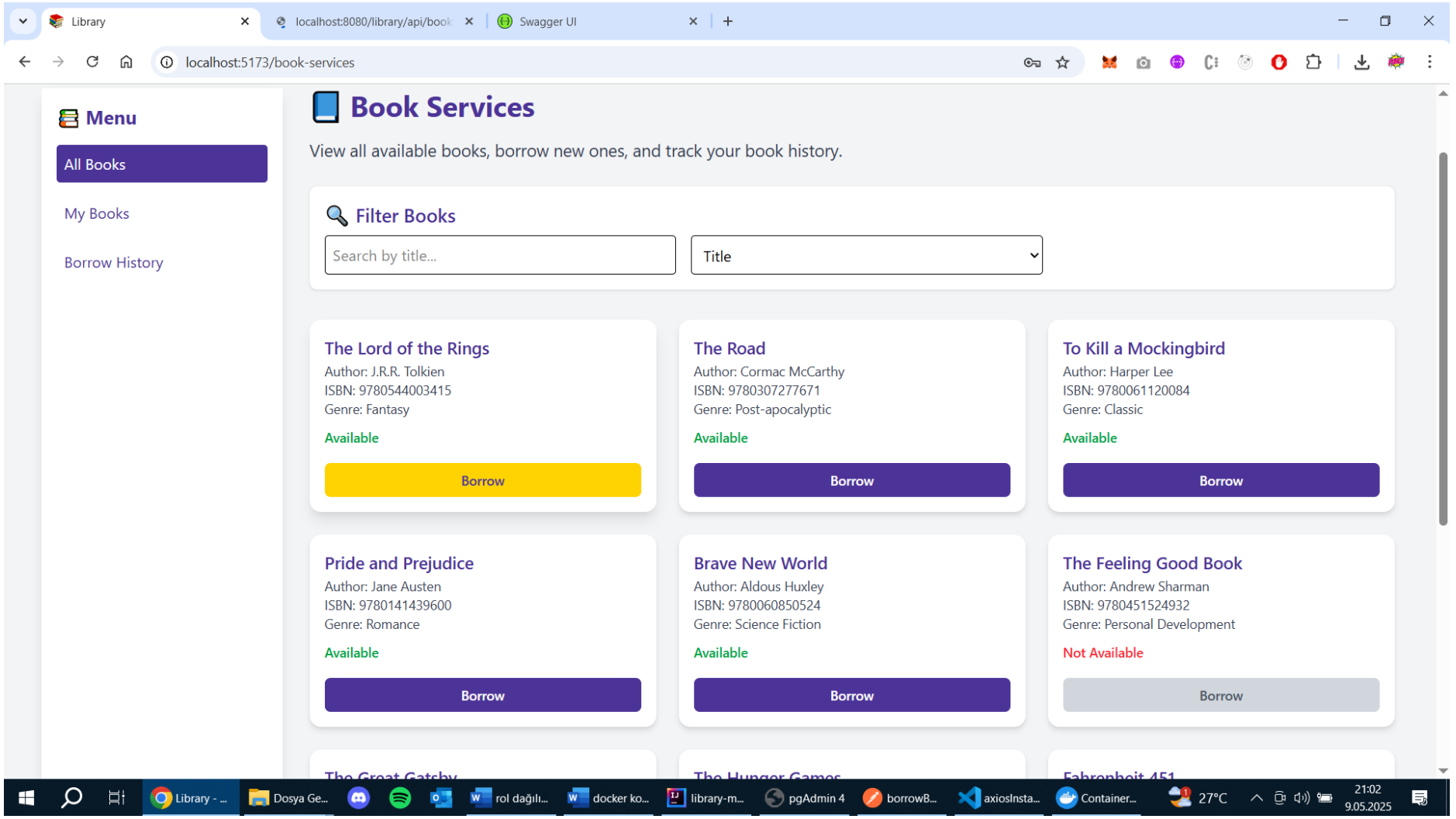
POST /borrow-records with LIBRARIAN Token – (200 OK)



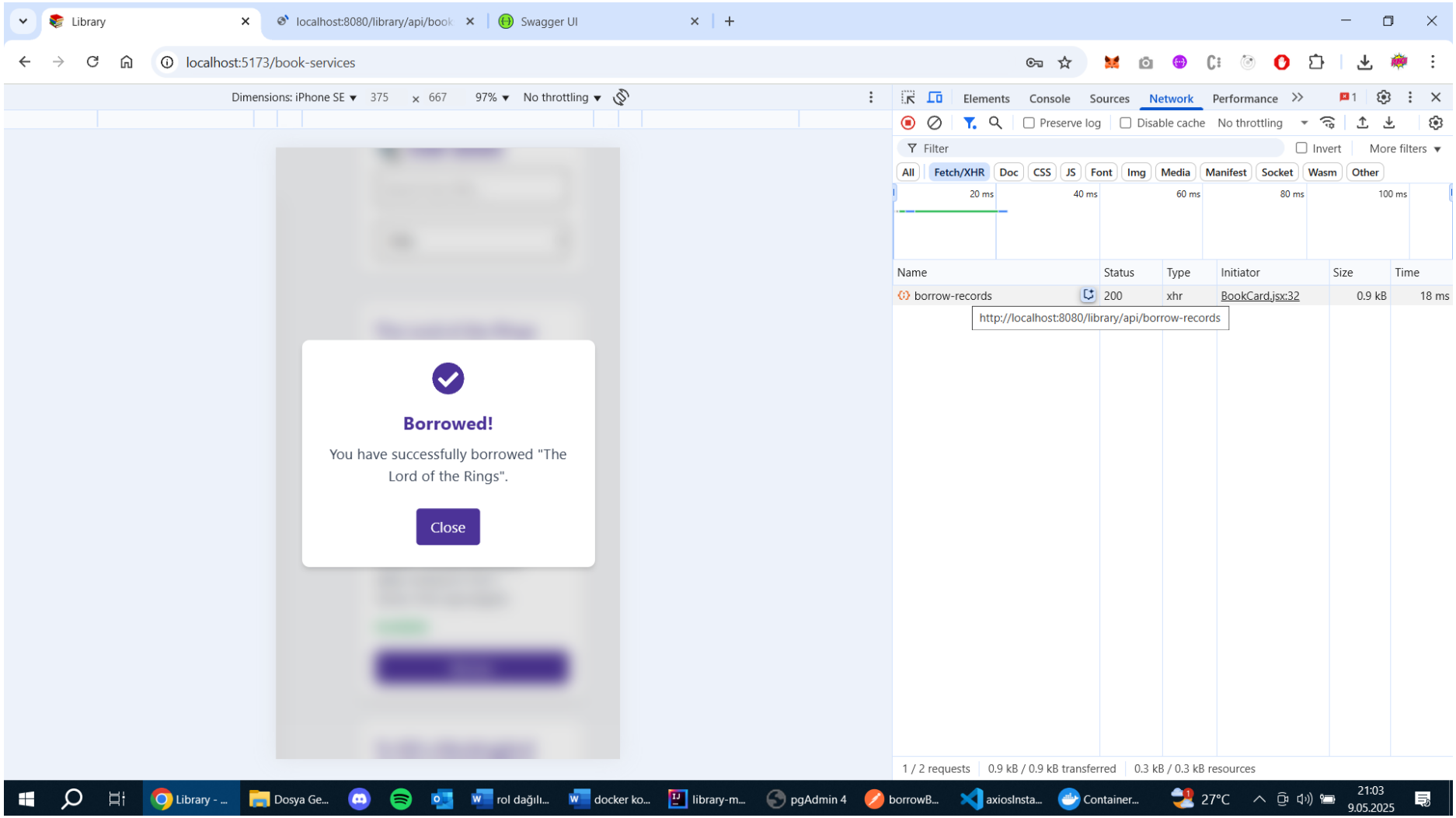
POST /borrow-records with PATRON Token – (200 OK)



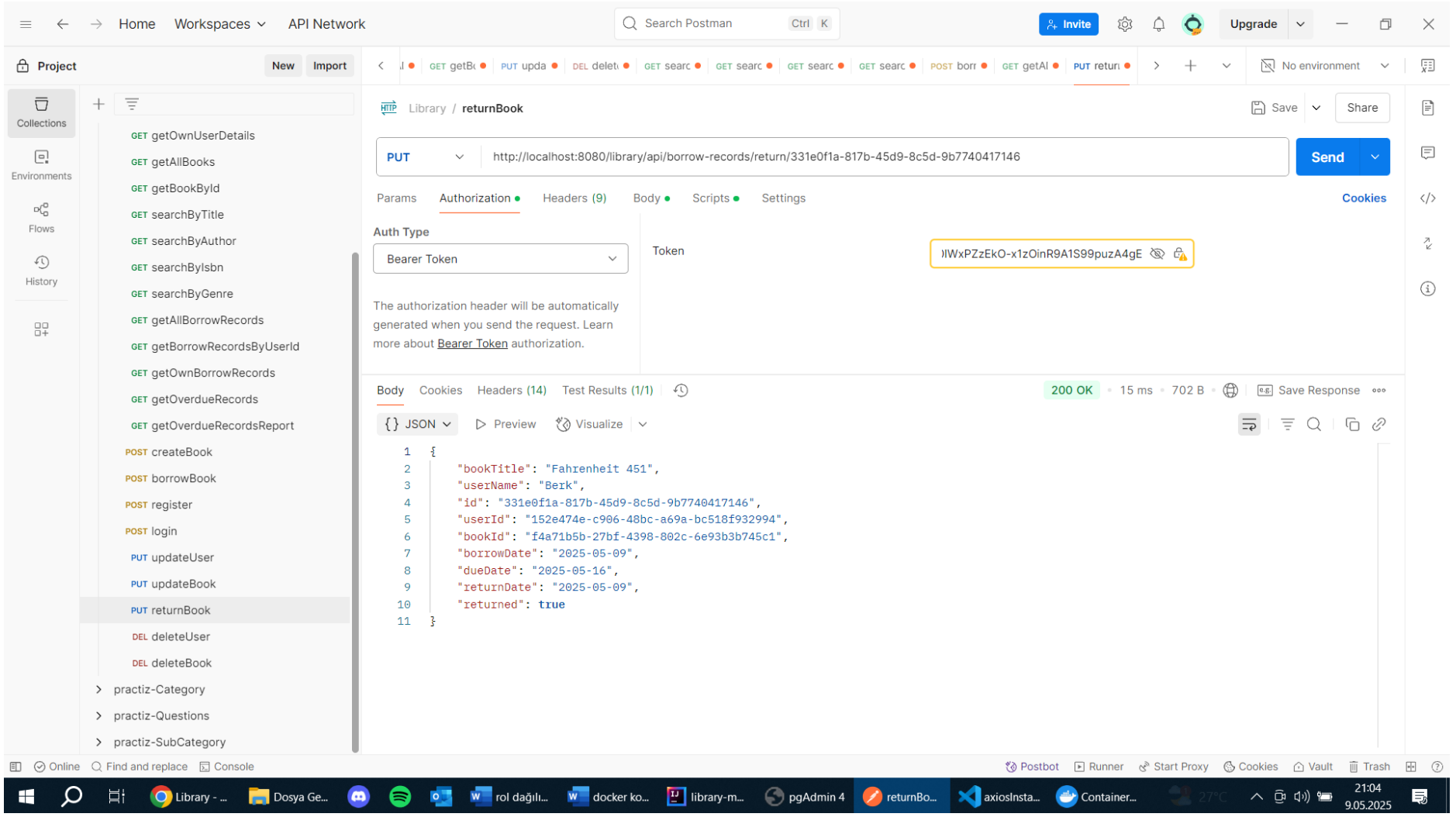
POST /borrow-records – All Books & Home



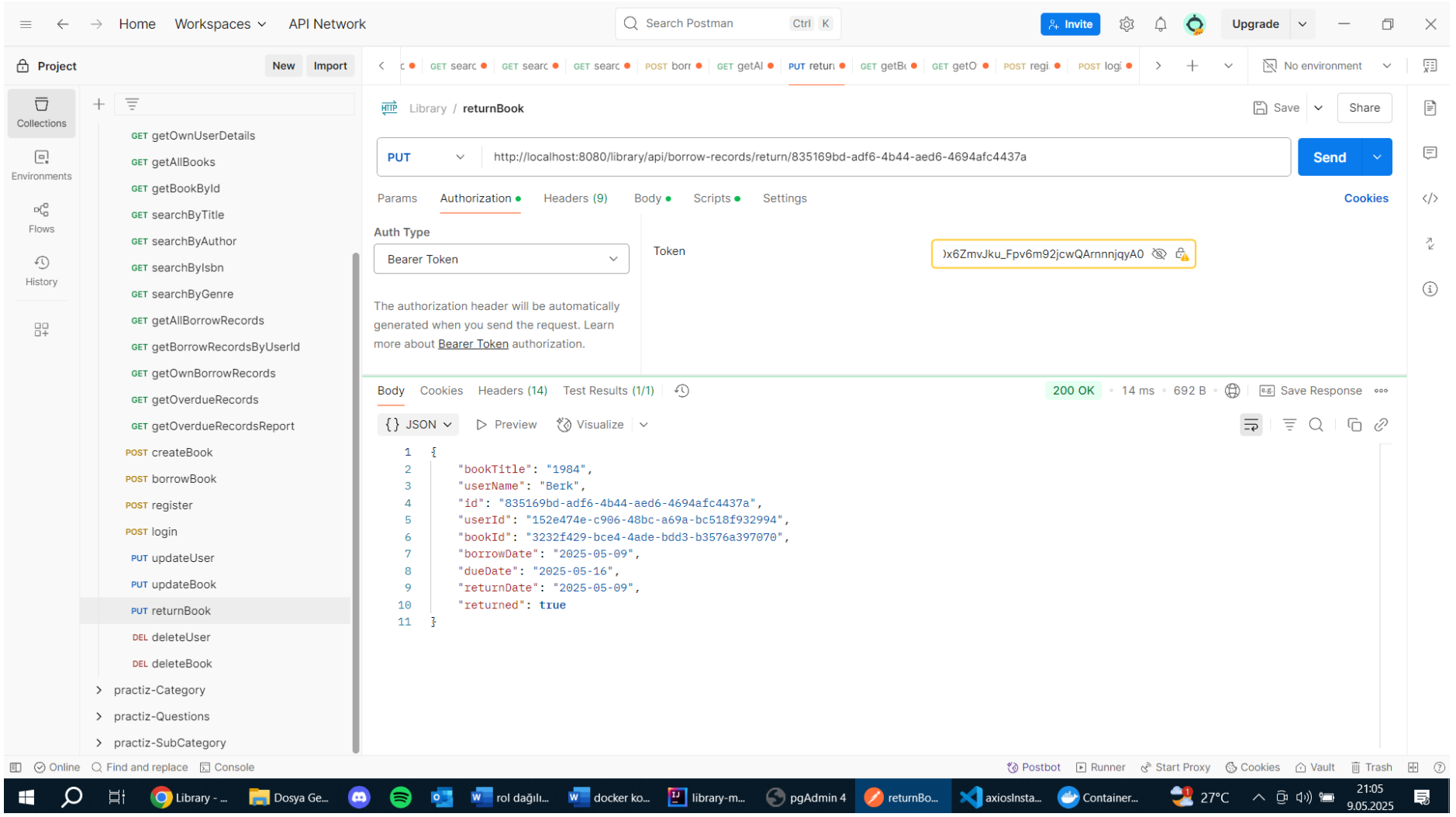
POST /borrow-records – All Books & Home 2



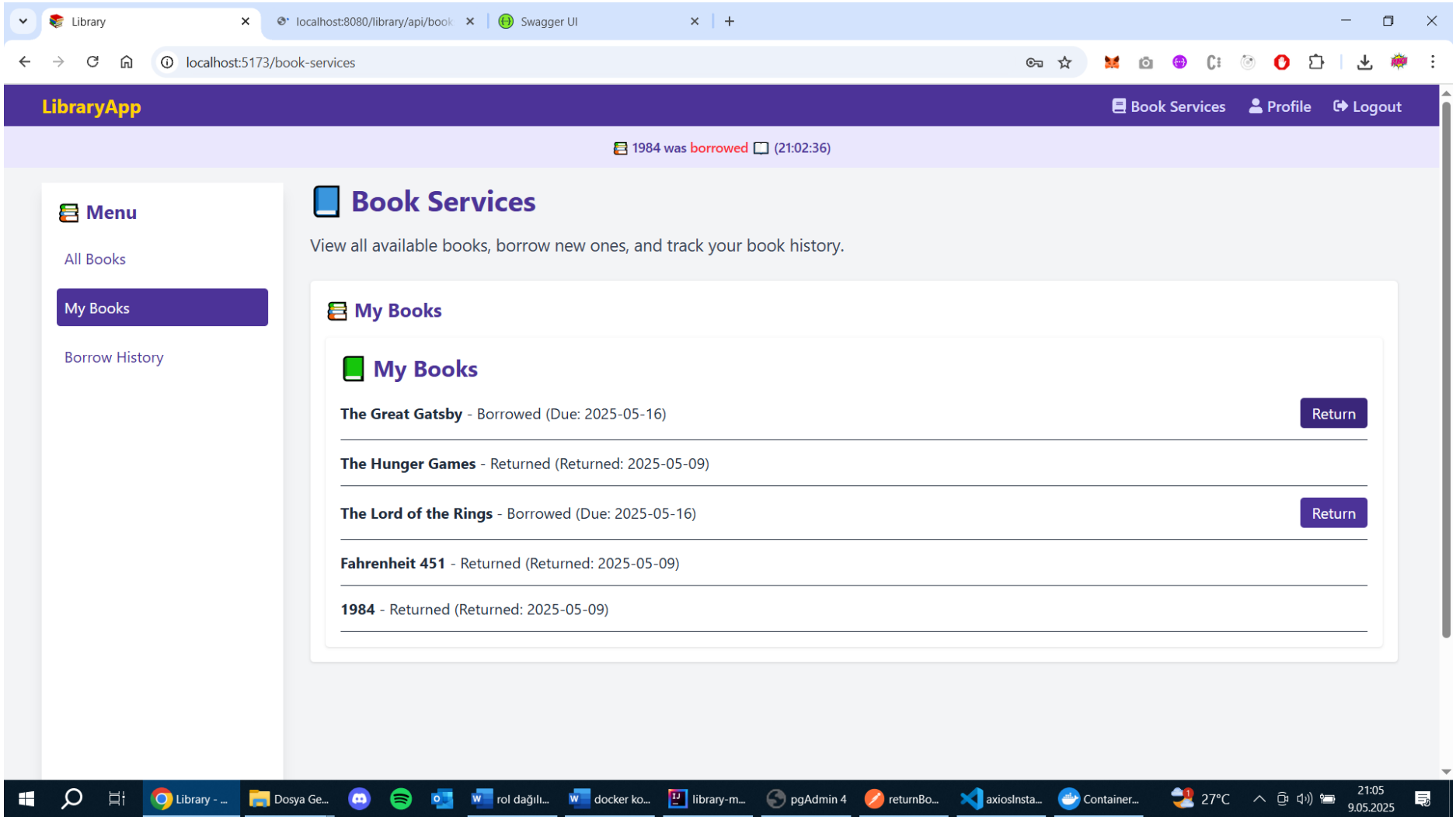
PUT /borrow-records/return/{id} with LIBRARIAN Token – (200 OK)



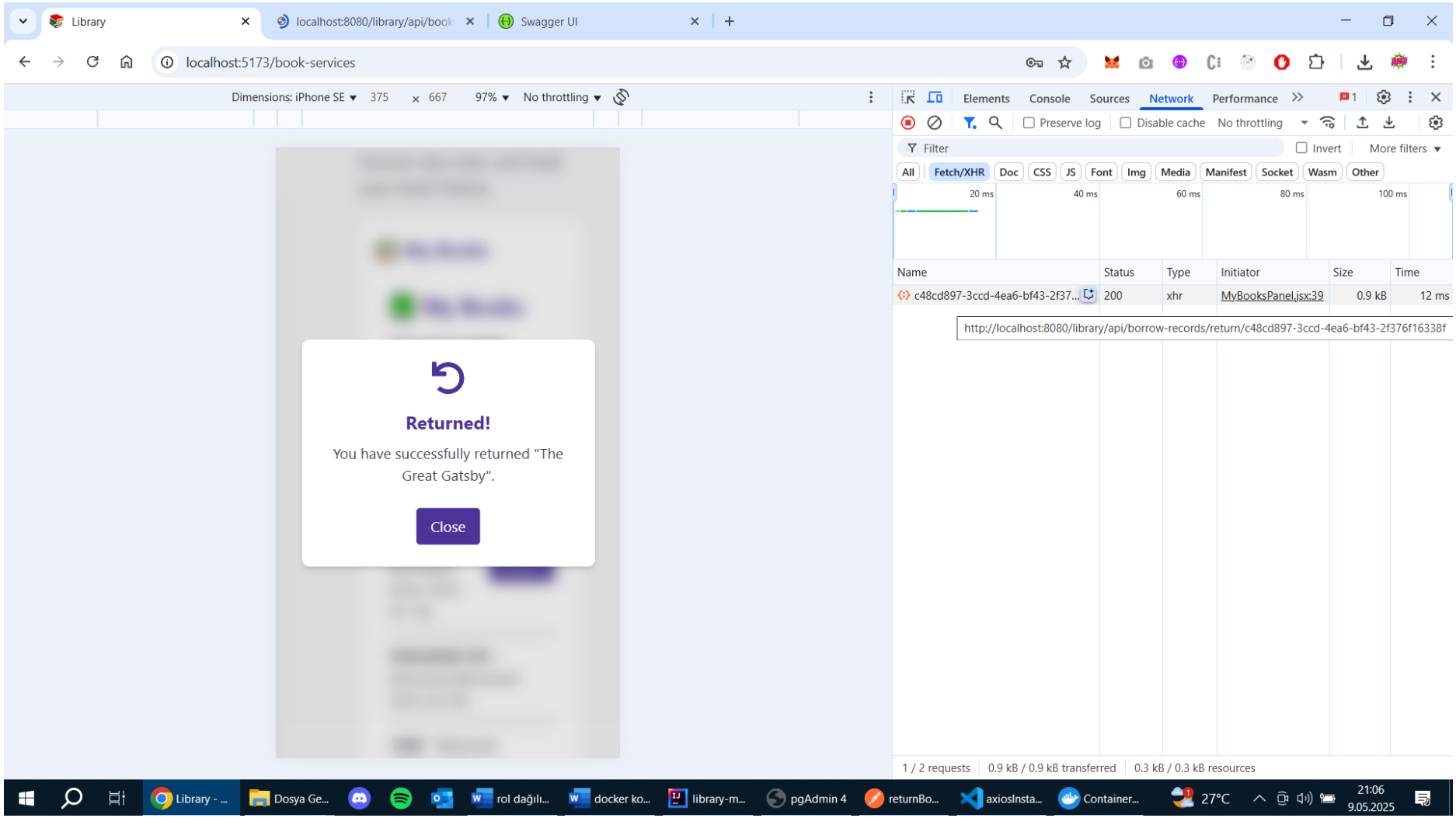
PUT /borrow-records/return/{id} with PATRON Token – (200 OK)



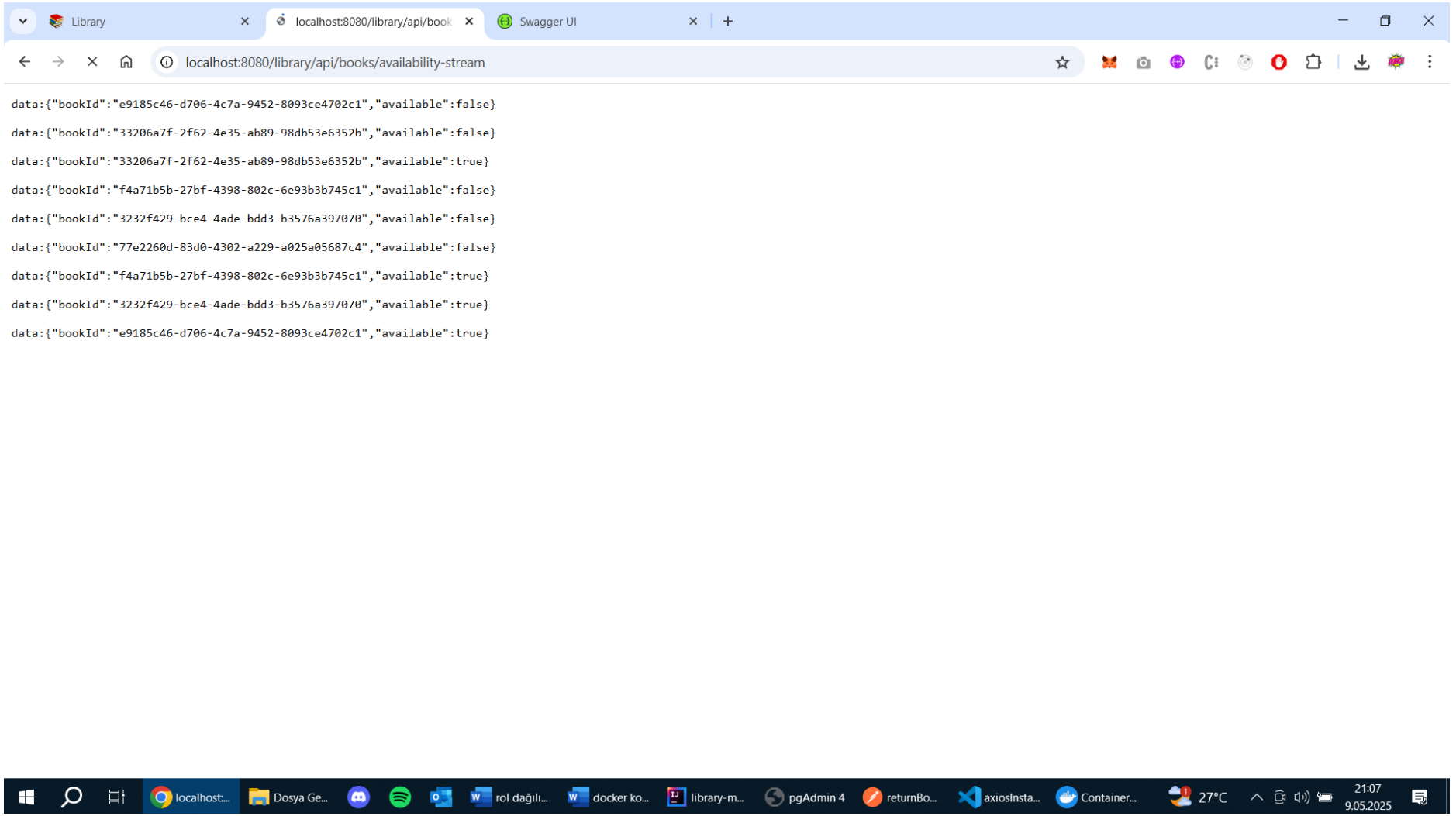
PUT /borrow-records/return/{id} – My Books



PUT /borrow-records/return/{id} – My Books 2

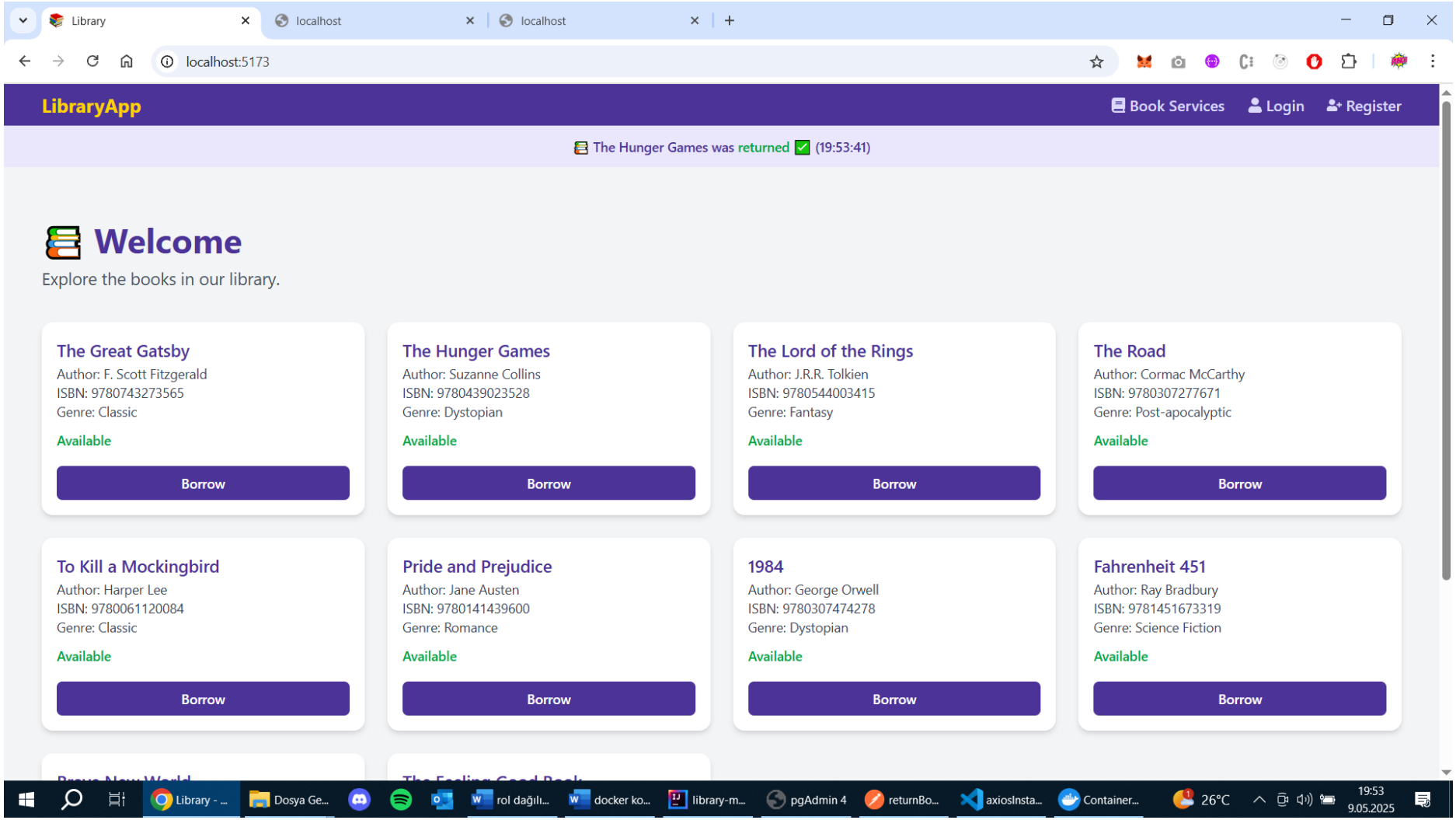


GET /books/availability-stream (reactive)

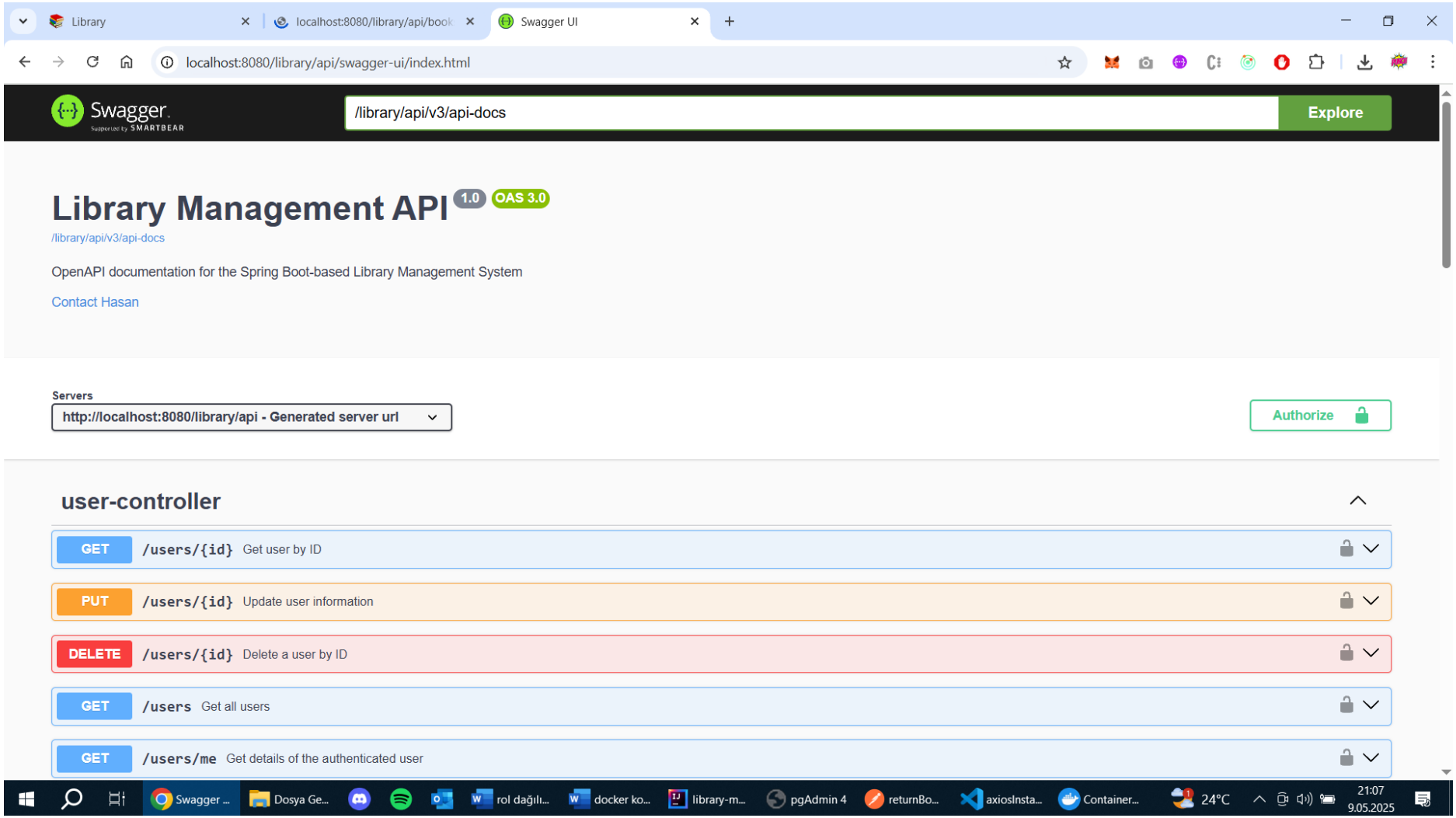


GET /books/availability-stream (reactive) 2

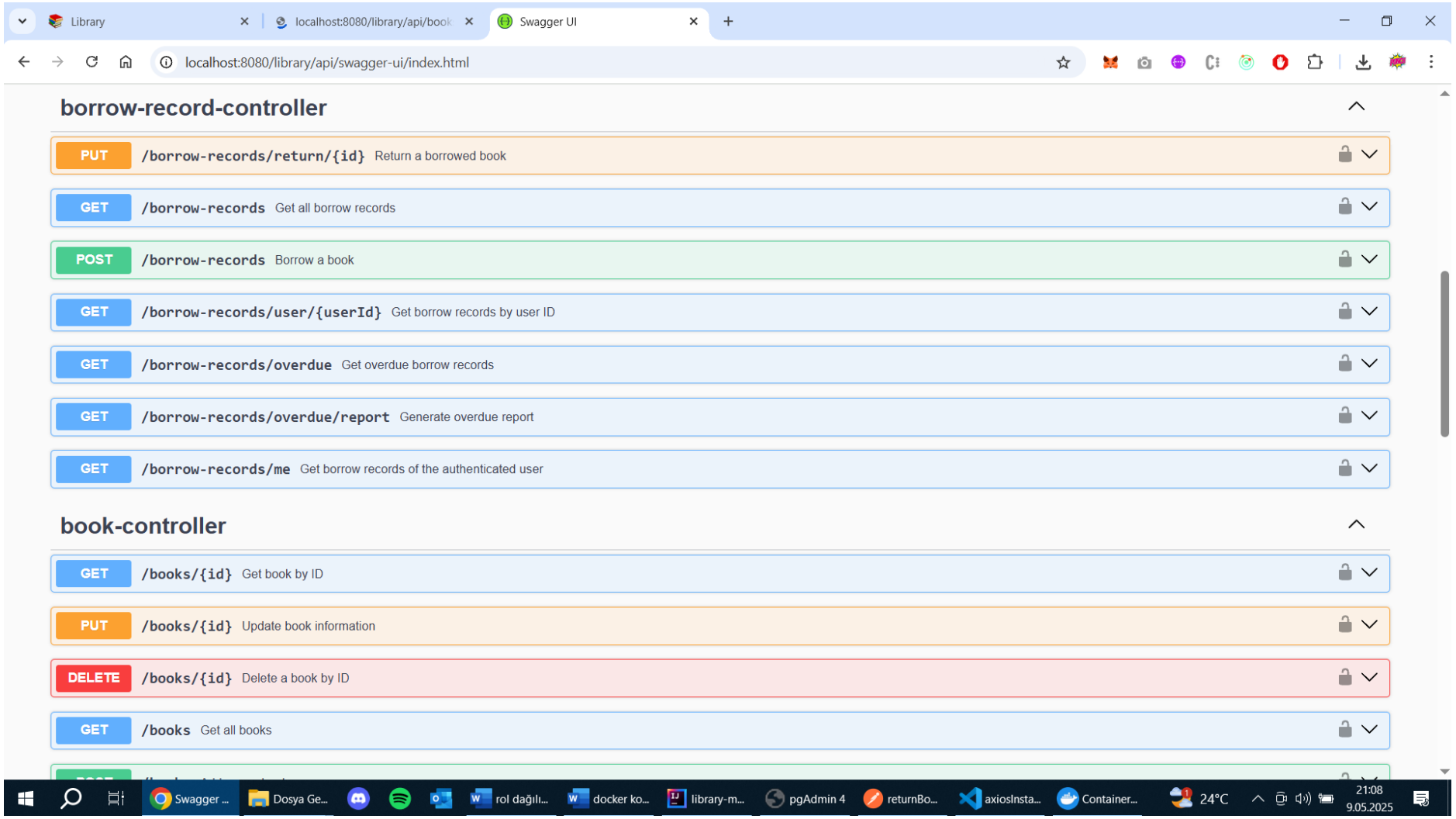
A real-time banner below the navbar displays book borrow/return events using Server-Sent Events (SSE).



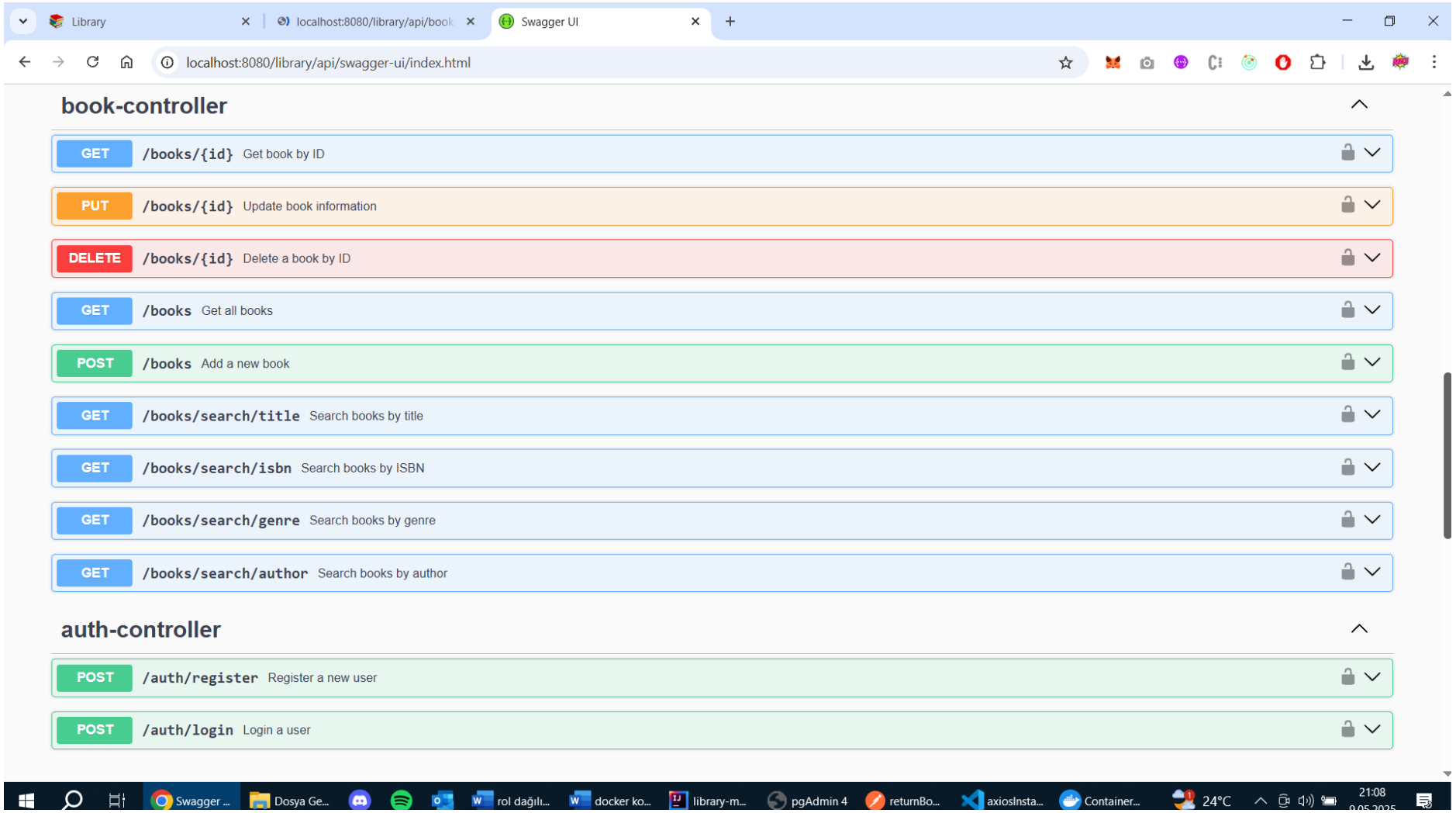
Swagger UI – Interactive API Documentation (http://localhost:8080/library/api/swagger-ui/index.html)



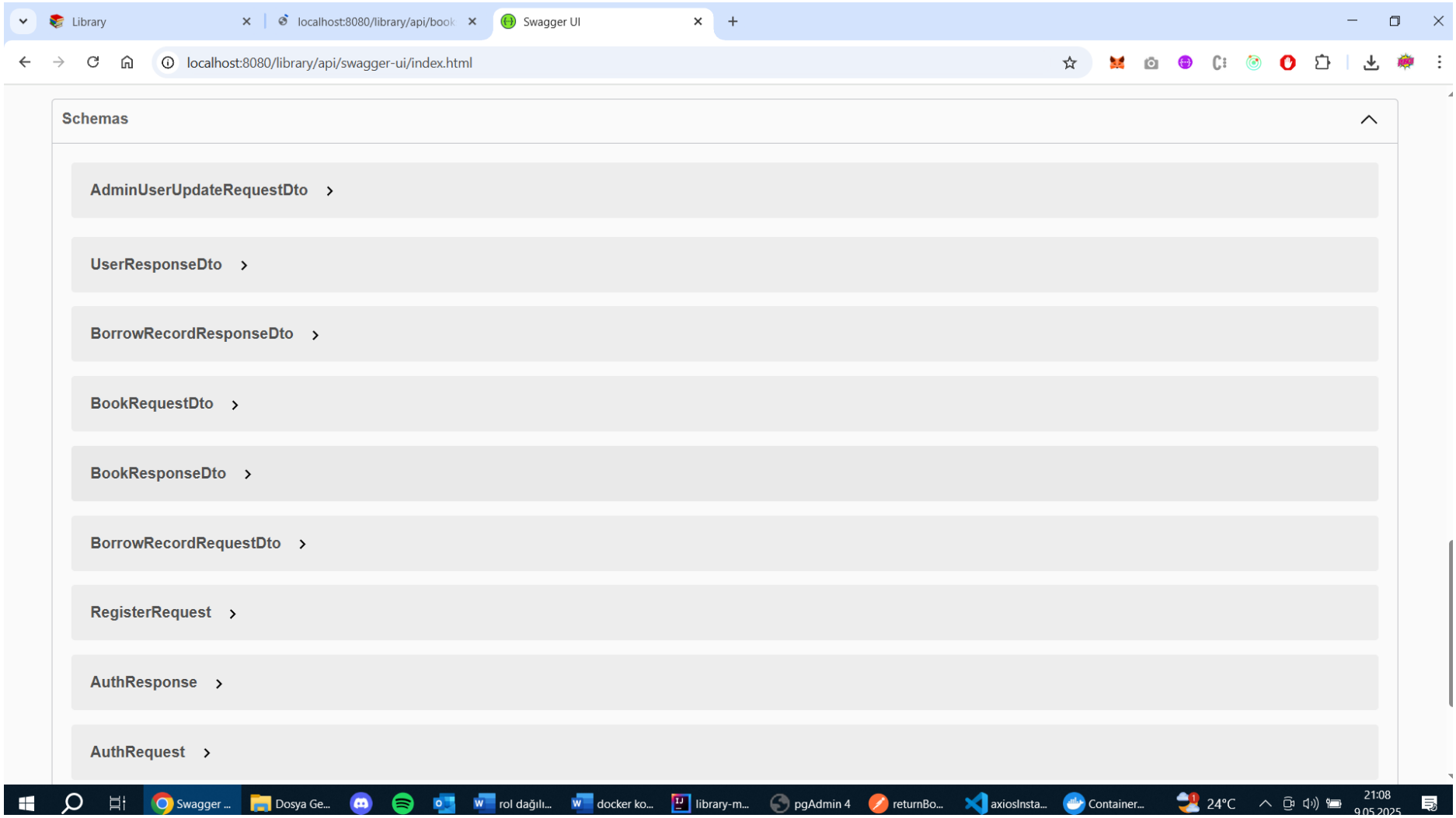
Swagger UI 2


Library
localhost:8080/library/api/book
Swagger UI
localhost:8080/library/api/swagger-ui/index.html
borrow-record-controller
PUT /borrow-records/return/{id} Return a borrowed book
GET /borrow-records Get all borrow records
POST /borrow-records Borrow a book
GET /borrow-records/user/{userId} Get borrow records by user ID
GET /borrow-records/overdue Get overdue borrow records
GET /borrow-records/overdue/report Generate overdue report
GET /borrow-records/me Get borrow records of the authenticated user
book-controller
GET /books/{id} Get book by ID
PUT /books/{id} Update book information
DELETE /books/{id} Delete a book by ID
GET /books Get all books
Swagger ...
Dosya Ge...
rol dağılı...
docker ko...
library-m...
pgAdmin 4
returnBo...
axiosInsta...
Container...
24°C
21:08
9.05.2025

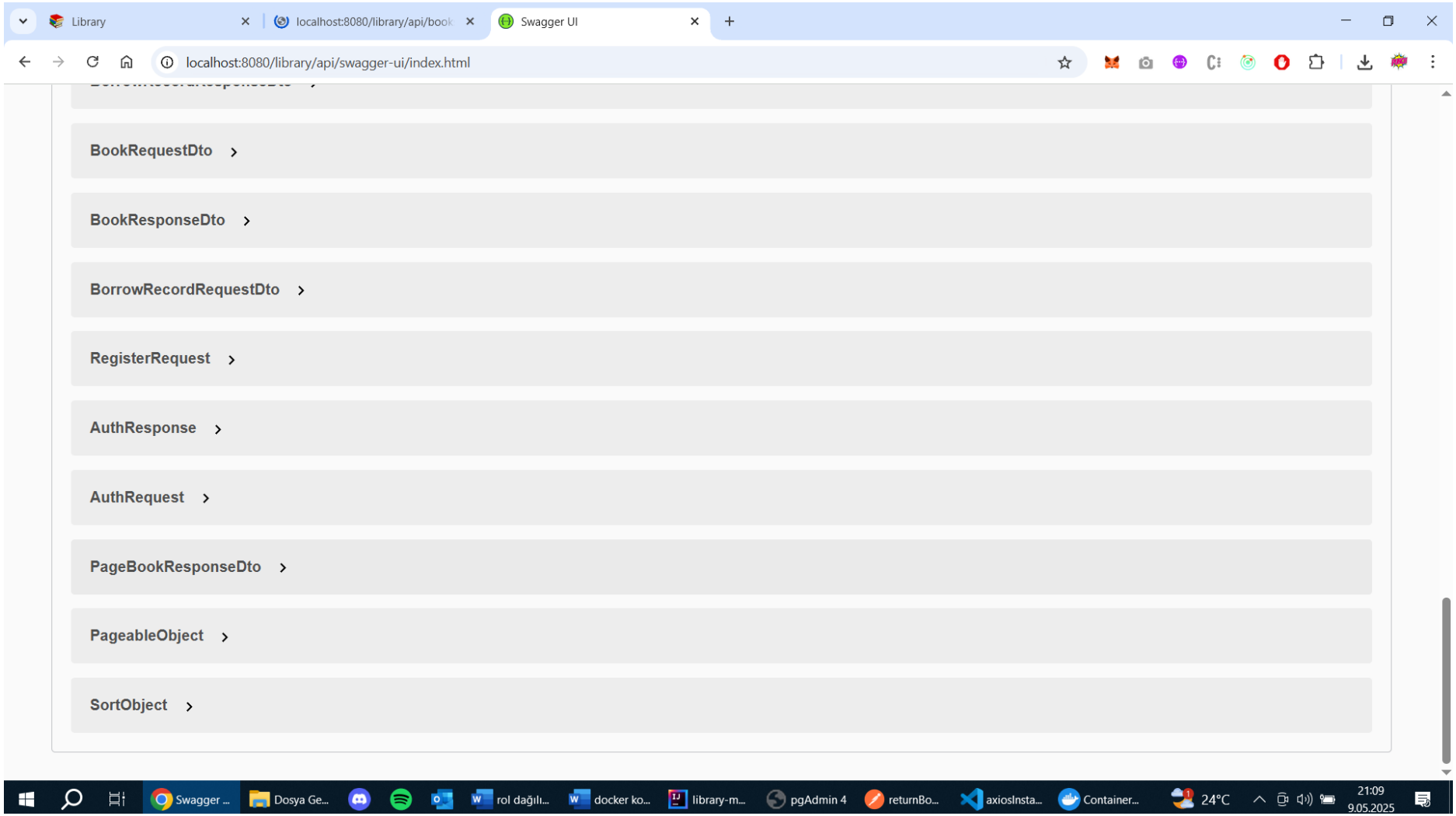
Swagger UI 3


Library
localhost:8080/library/api/book
Swagger UI
localhost:8080/library/api/swagger-ui/index.html
book-controller
GET /books/{id} Get book by ID
PUT /books/{id} Update book information
DELETE /books/{id} Delete a book by ID
GET /books Get all books
POST /books Add a new book
GET /books/search/title Search books by title
GET /books/search/isbn Search books by ISBN
GET /books/search/genre Search books by genre
GET /books/search/author Search books by author
auth-controller
POST /auth/register Register a new user
POST /auth/login Login a user
Swagger ...
Dosya Ge...
rol dağılı...
docker ko...
library-m...
pgAdmin 4
returnBo...
axiosInsta...
Container...
24°C
21:08

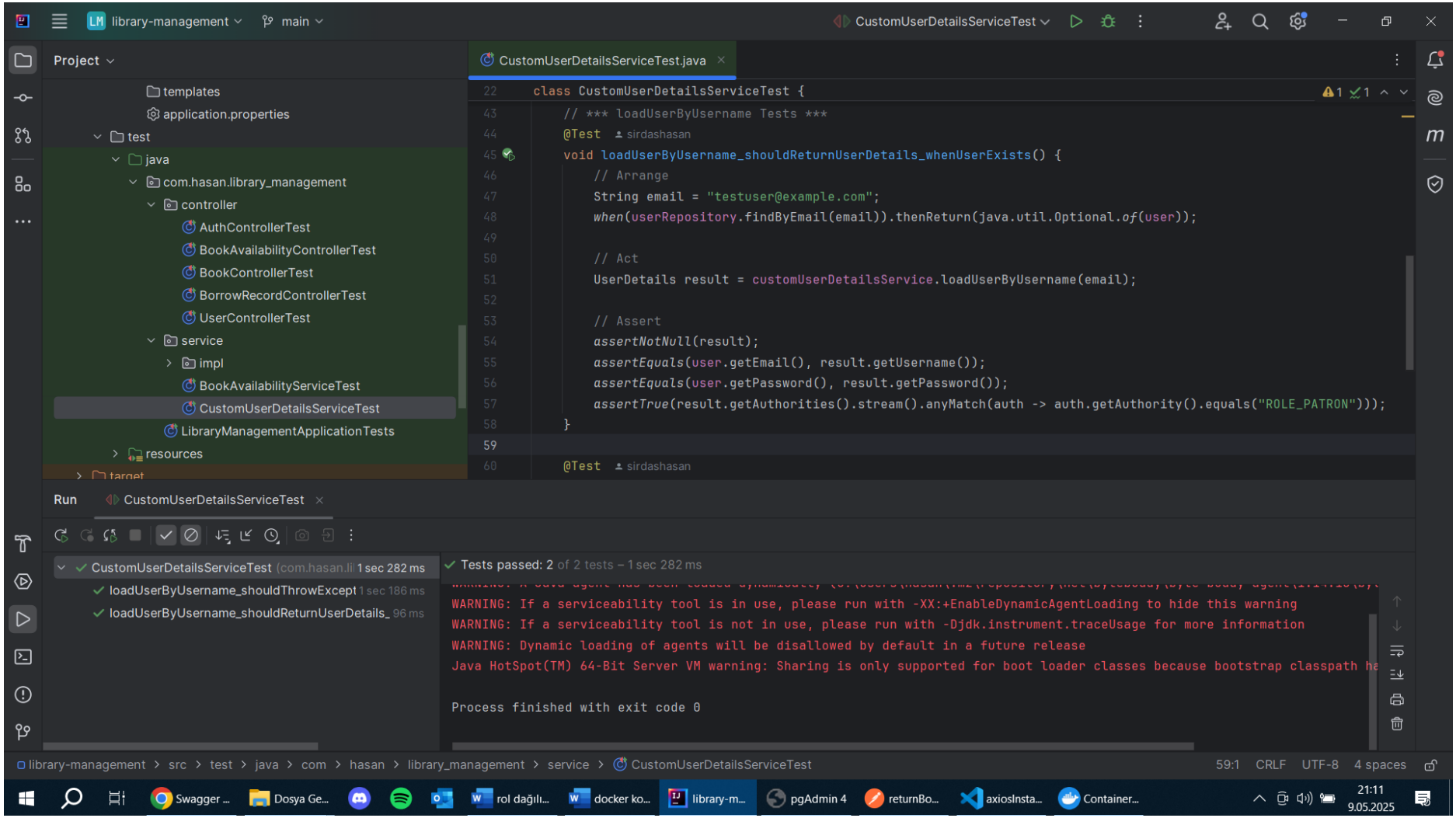
Swagger UI 4


Library
localhost:8080/library/api/book
Swagger UI
localhost:8080/library/api/swagger-ui/index.html
Schemas
AdminUserUpdateRequestDto
UserResponseDto
BorrowRecordResponseDto
BookRequestDto
BookResponseDto
BorrowRecordRequestDto
RegisterRequest
AuthResponse
AuthRequest
Swagger ...
Dosya Ge...
rol dağılı...
docker ko...
library-m...
pgAdmin 4
returnBo...
axiosInsta...
Container...
24°C
21:08
9.05.2025

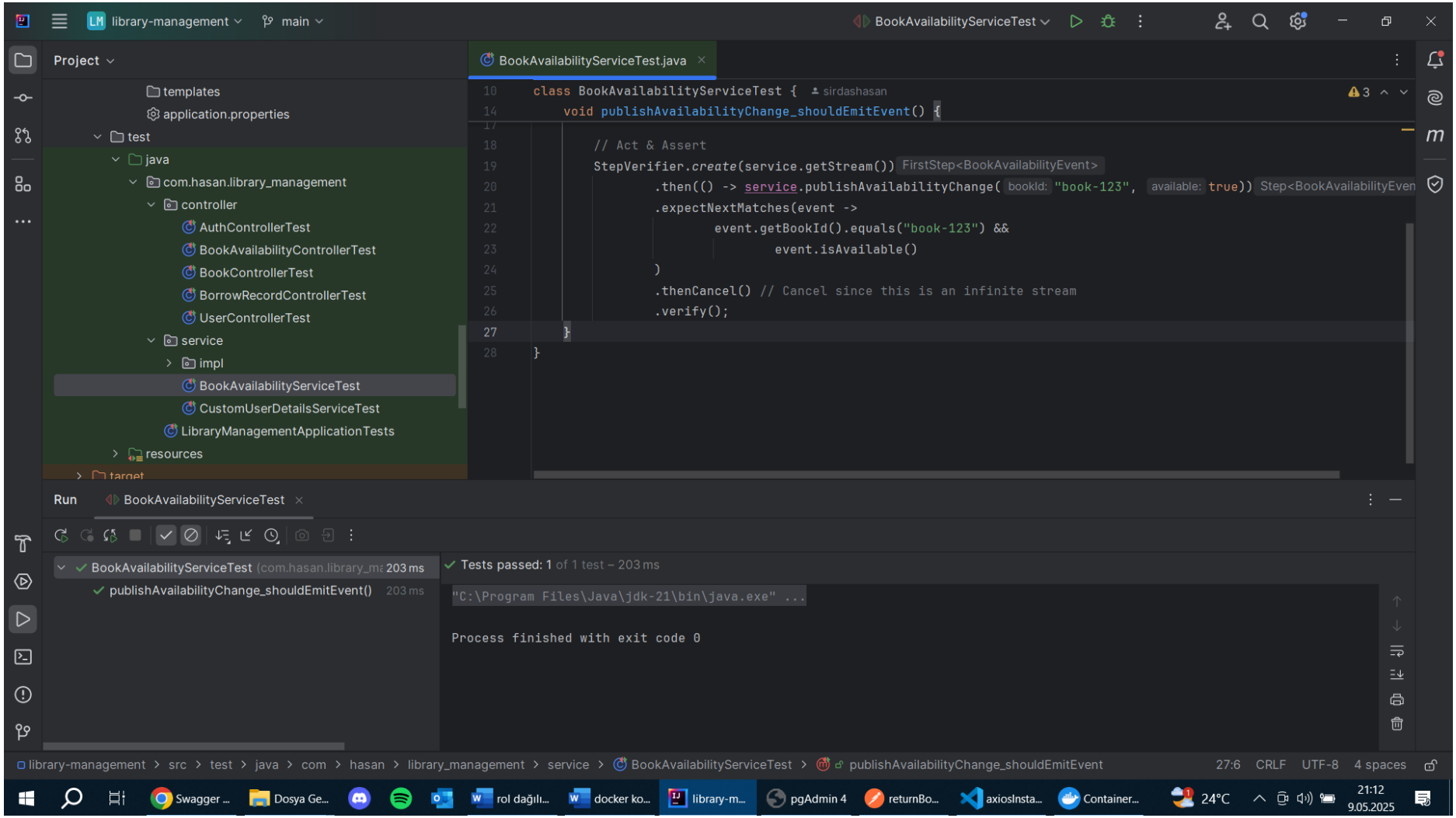
Swagger UI 5


Library
localhost:8080/library/api/book
Swagger UI
localhost:8080/library/api/swagger-ui/index.html
BookRequestDto
BookResponseDto
BorrowRecordRequestDto
RegisterRequest
AuthResponse
AuthRequest
PageBookResponseDto
PageableObject
SortObject
Swagger ...
Dosya Ge...
rol dağılı...
docker ko...
library-m...
pgAdmin 4
returnBo...
axiosInsta...
Container...
24°C
21:09
9.05.2025

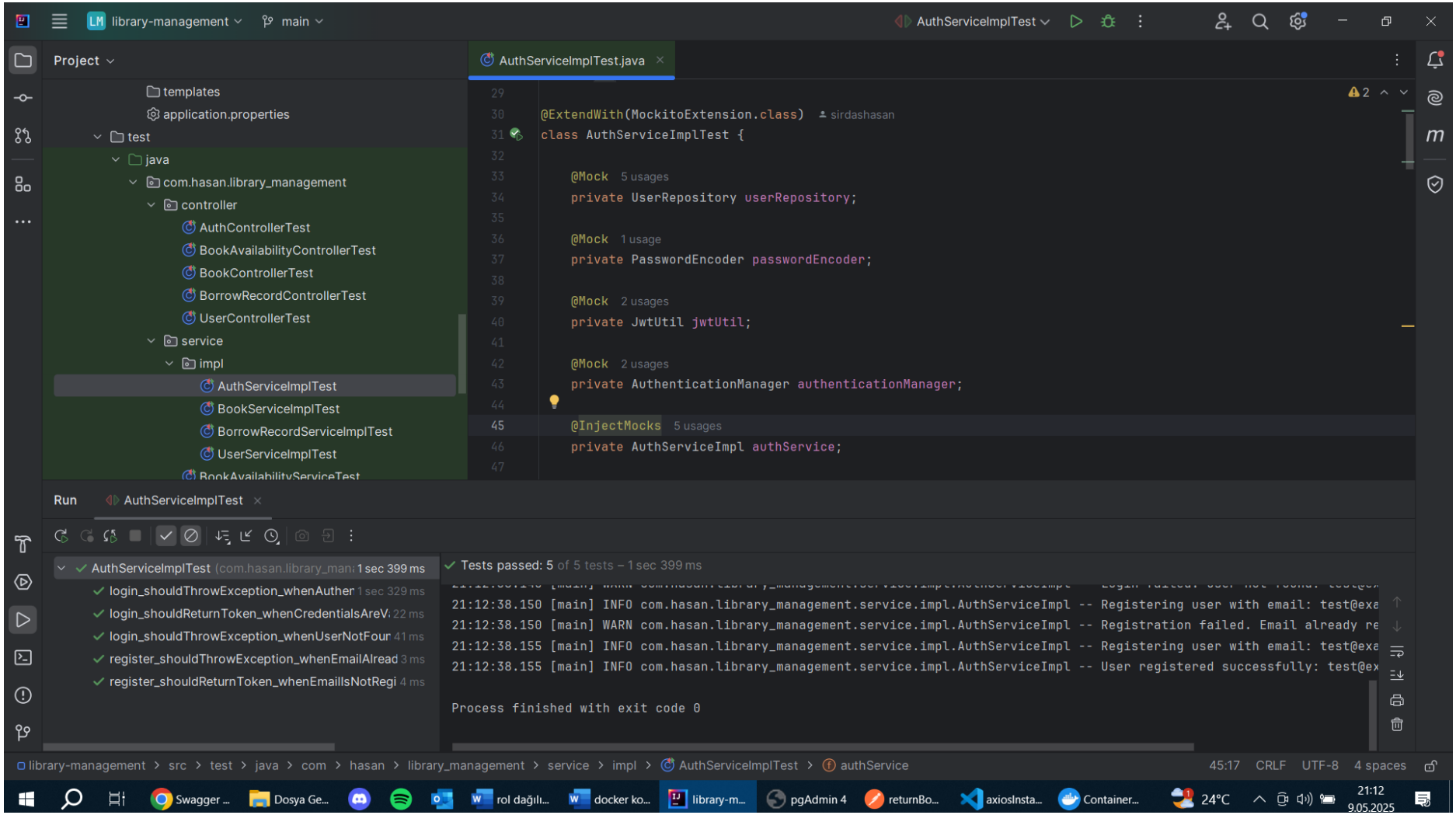
Unit Test – CustomUserDetailsServiceTest



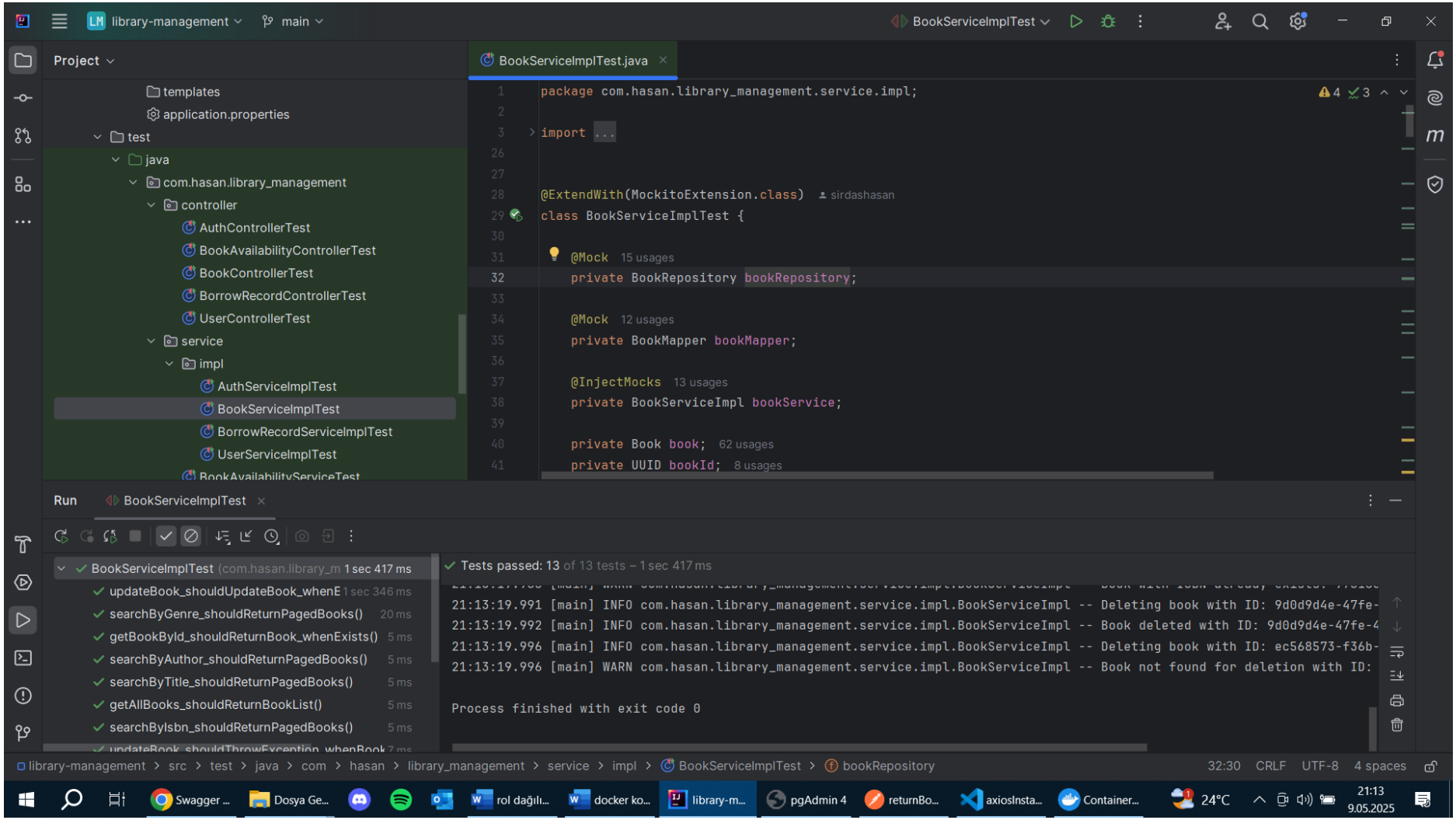
Unit Test – BookAvailabilityServiceTest



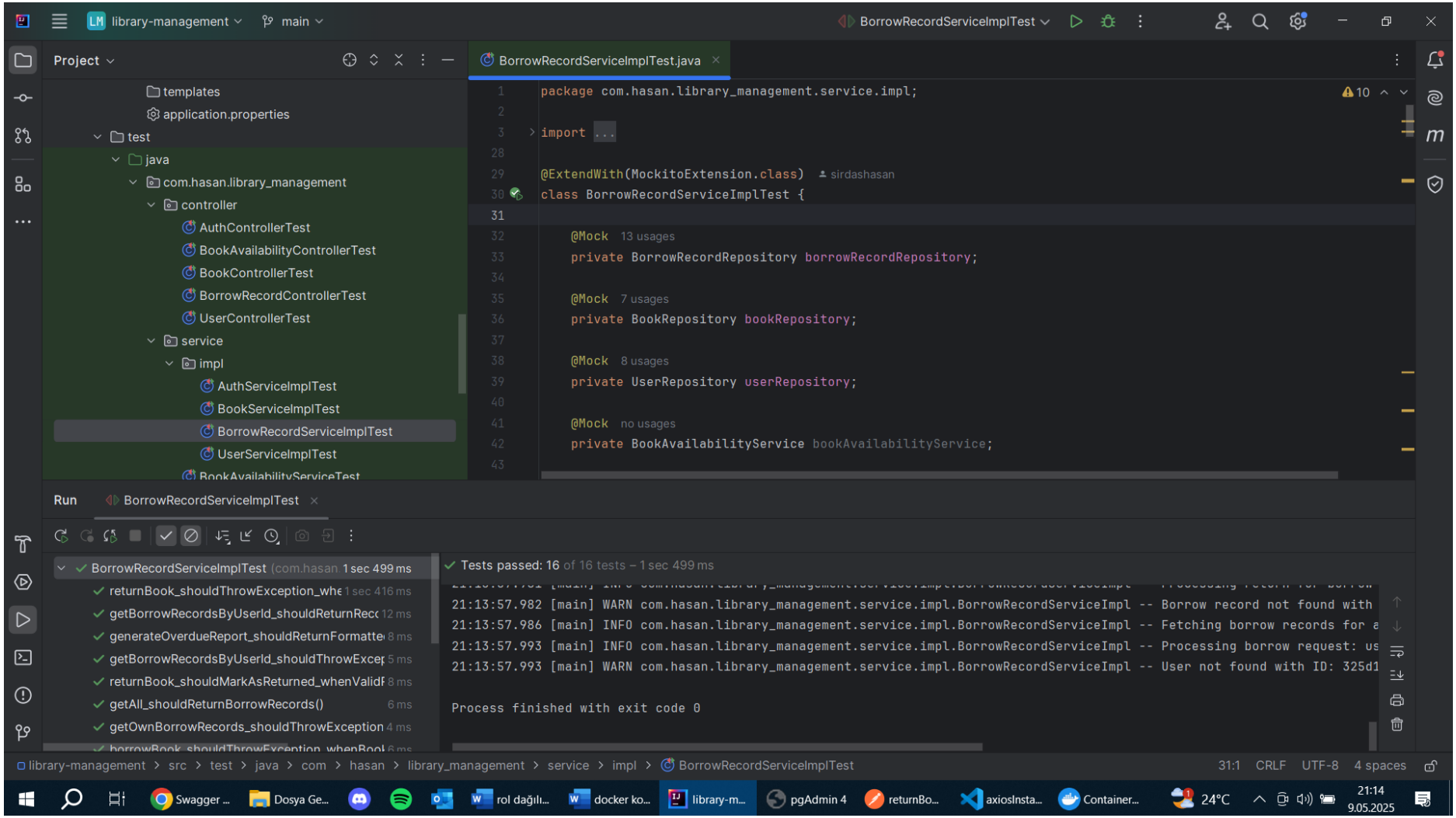
Unit Test – AuthServiceImplTest



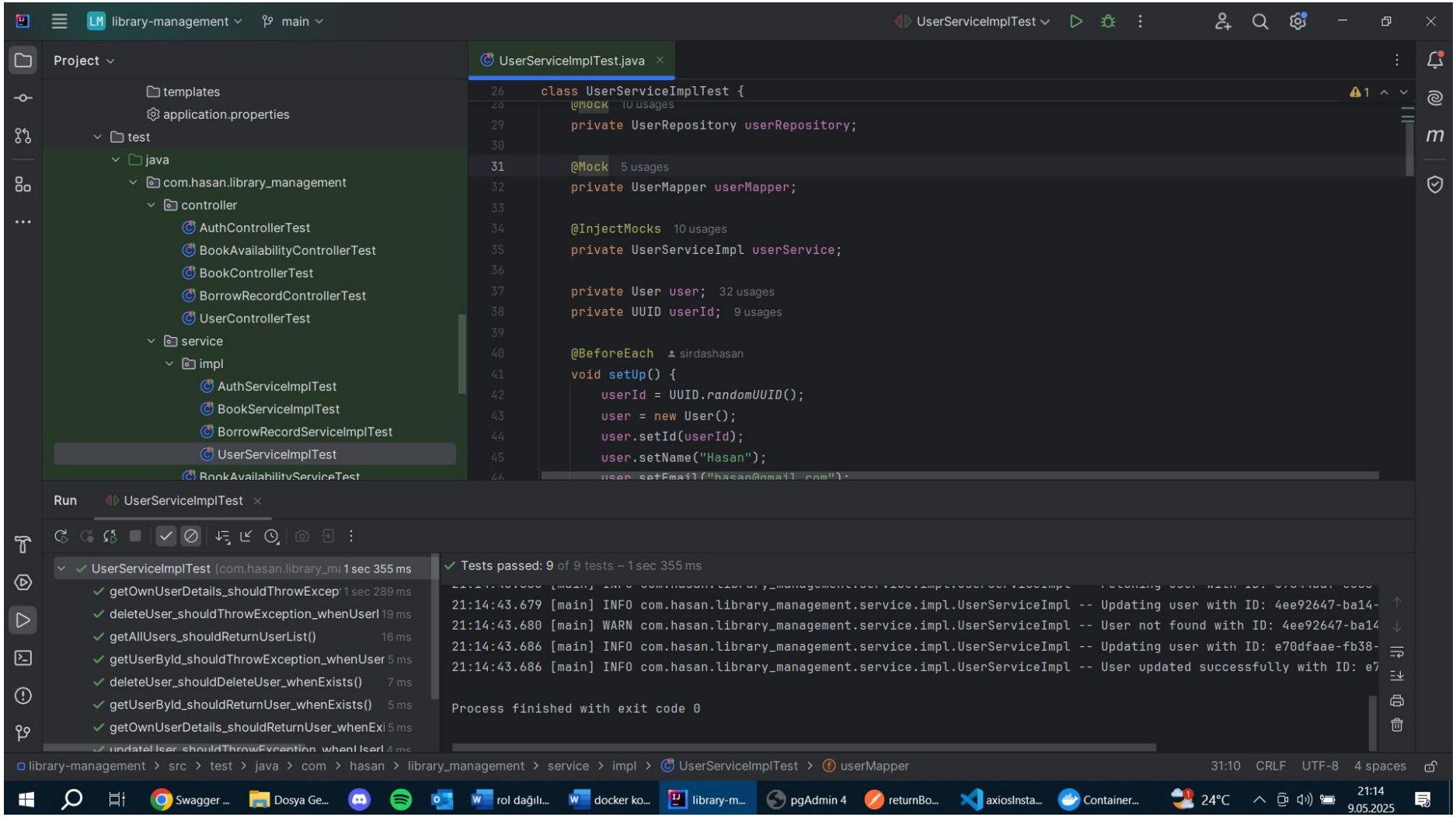
Unit Test – BookServiceImplTest



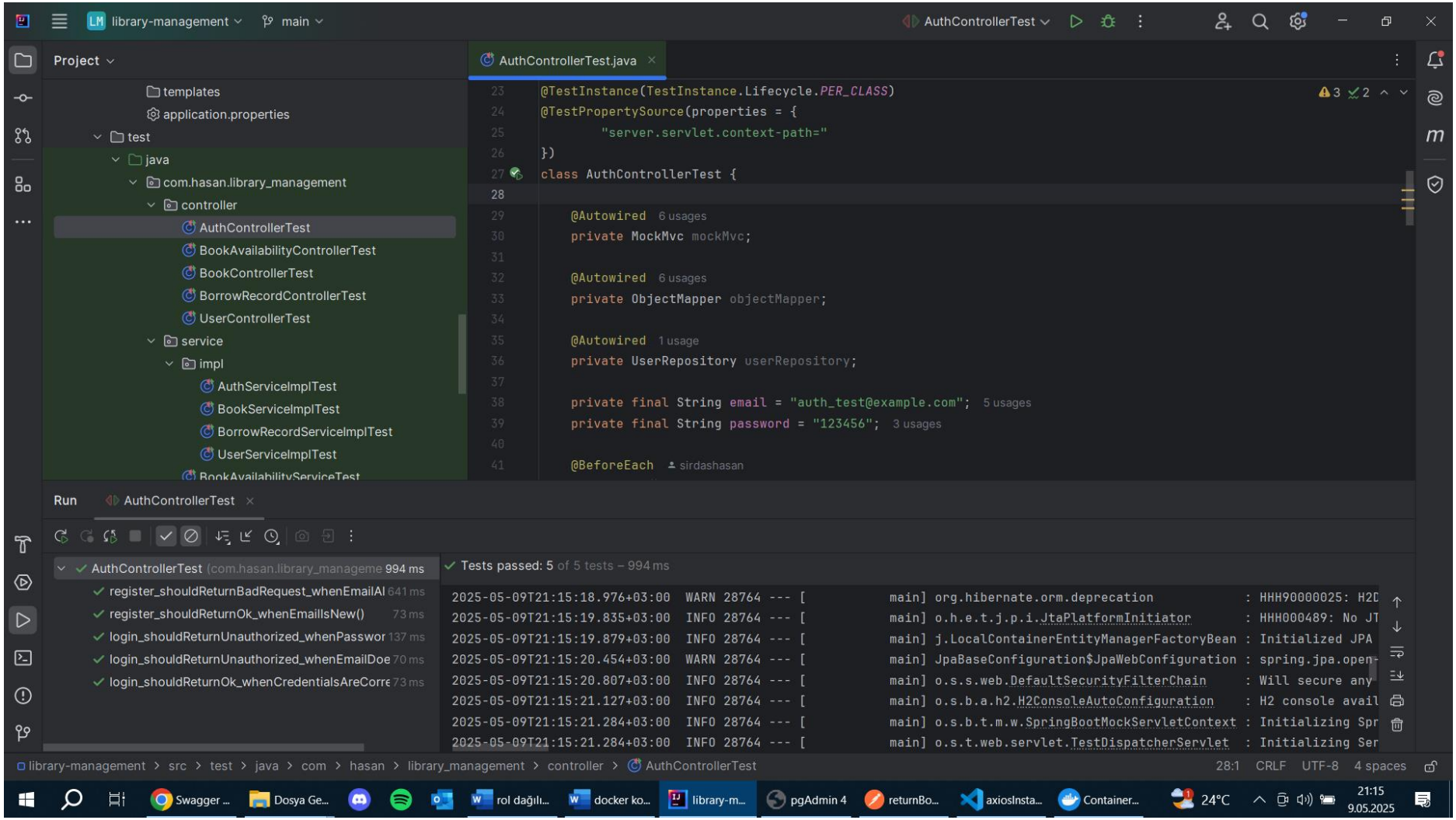
Unit Test – BorrowRecordServiceImplTest



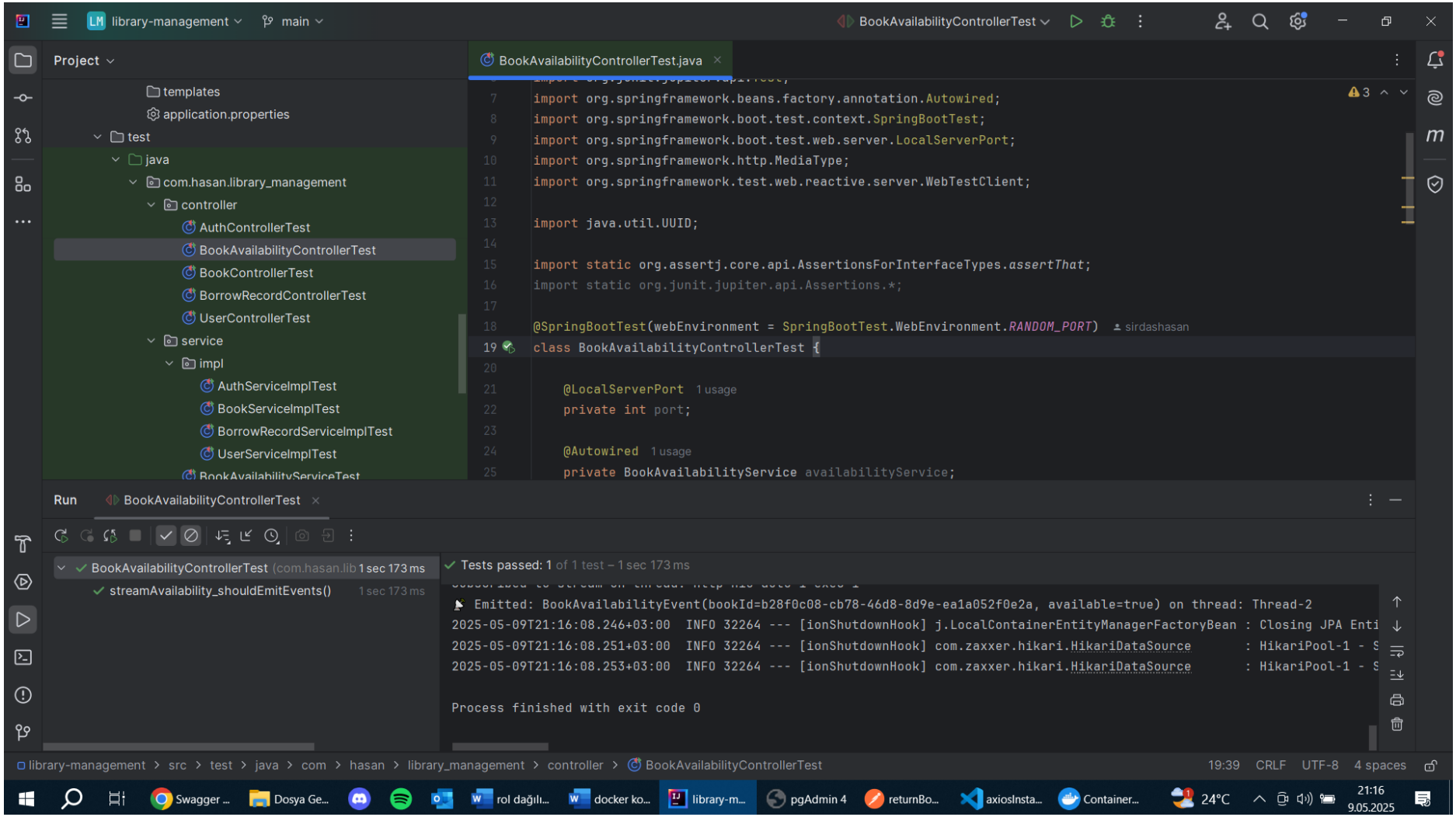
Unit Test – UserServiceImplTest



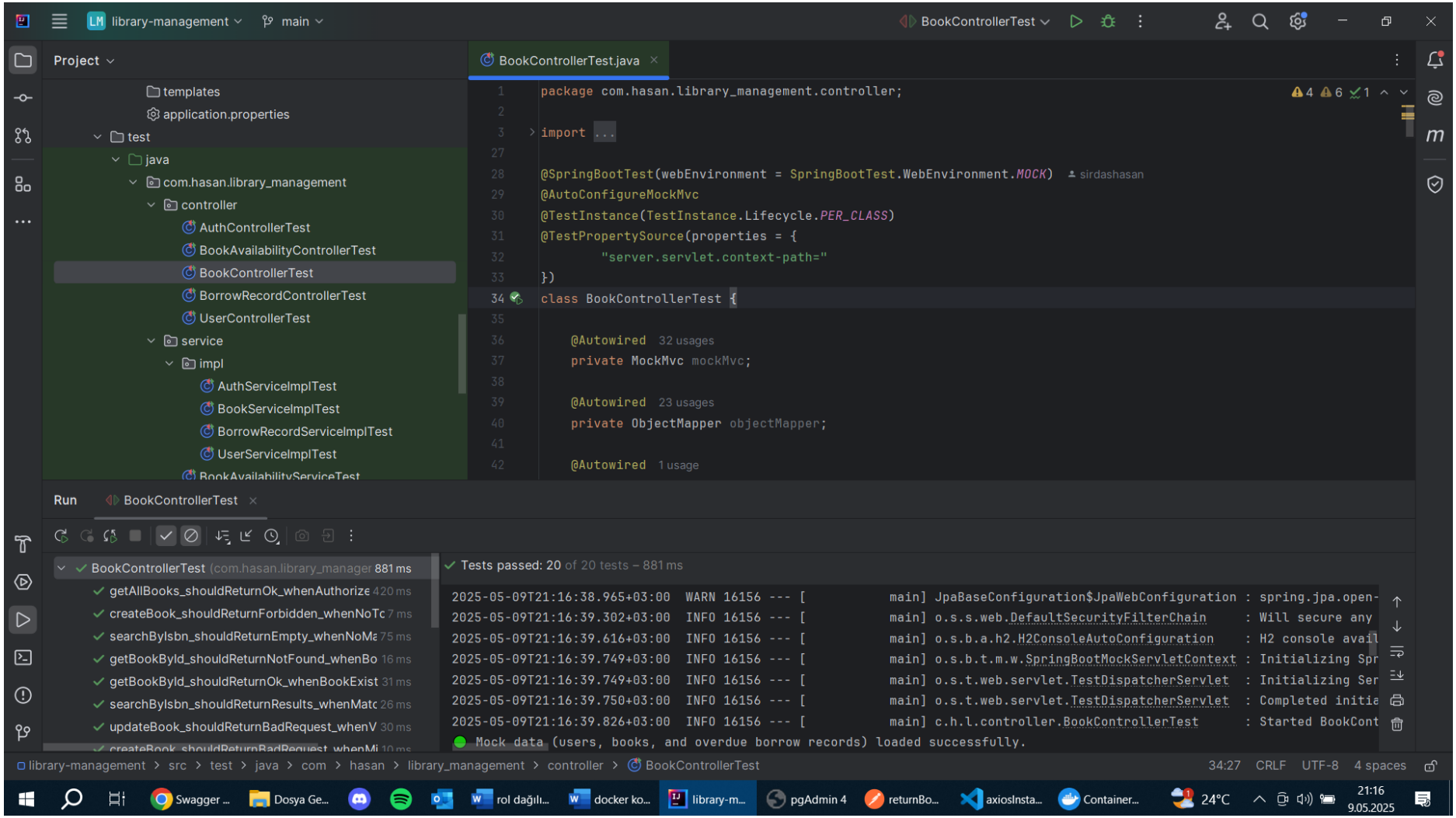
Integration Test – AuthControllerTest



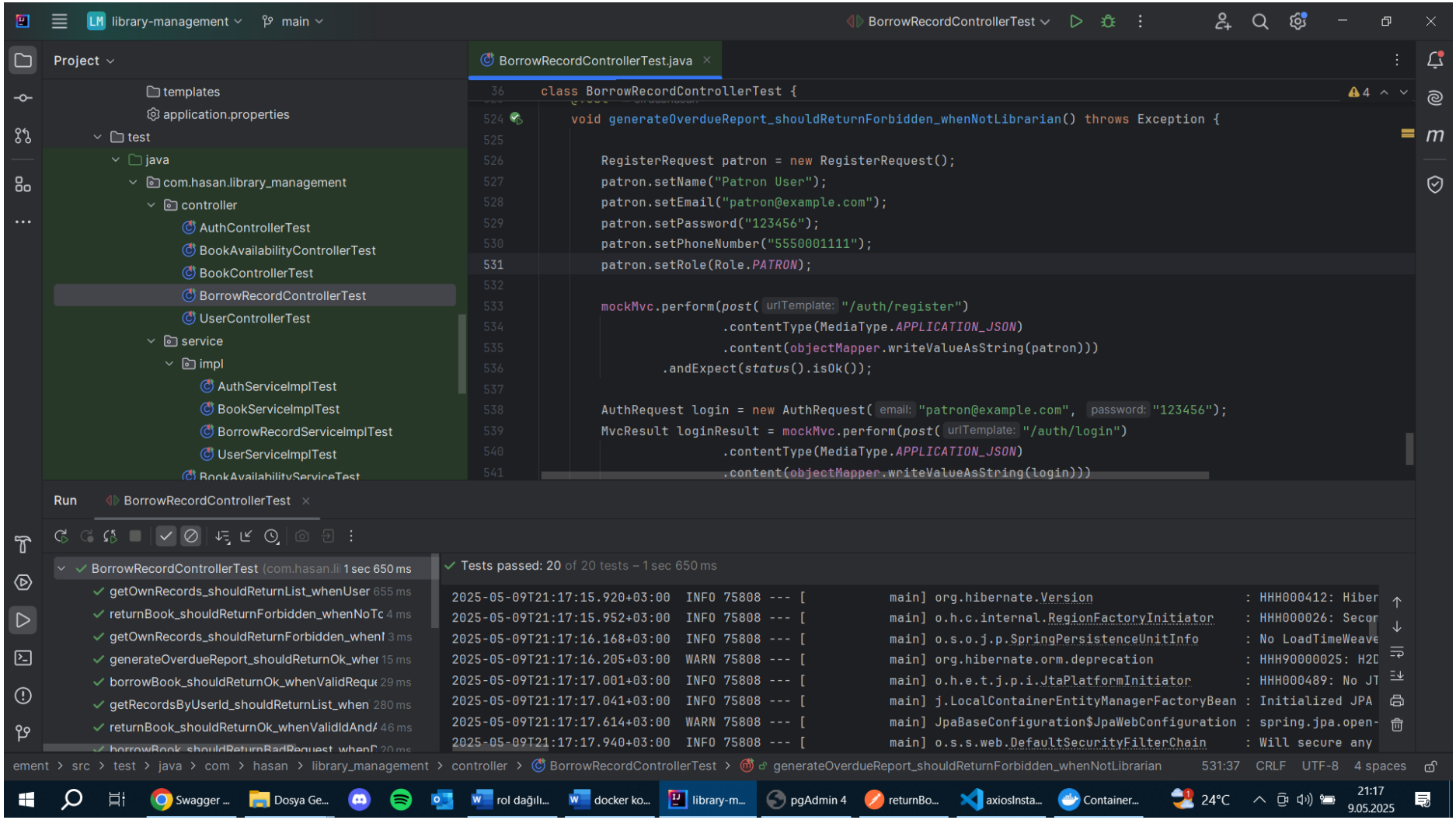
Integration Test – BookAvailabilityControllerTest



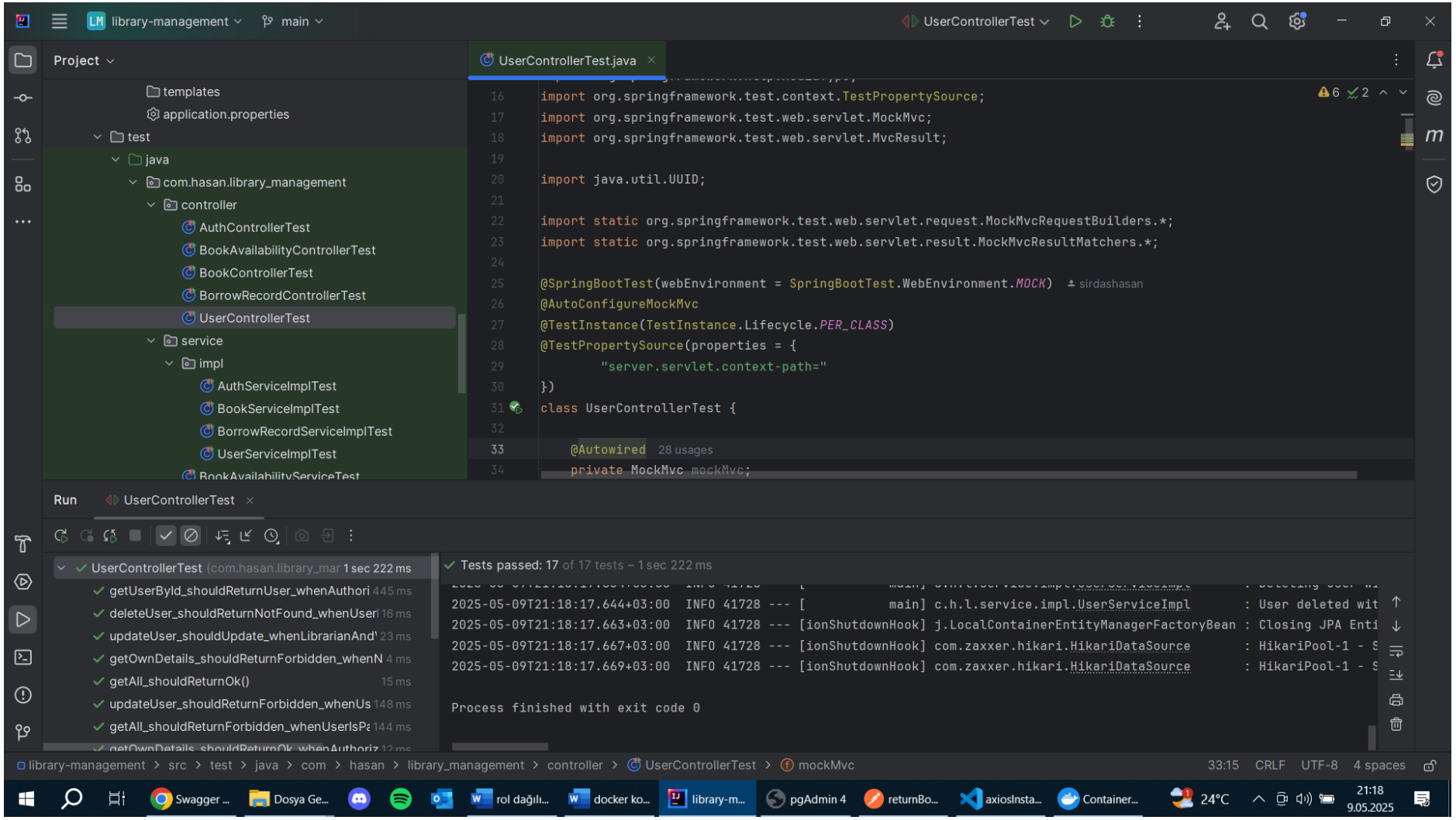
Integration Test – BookControllerTest



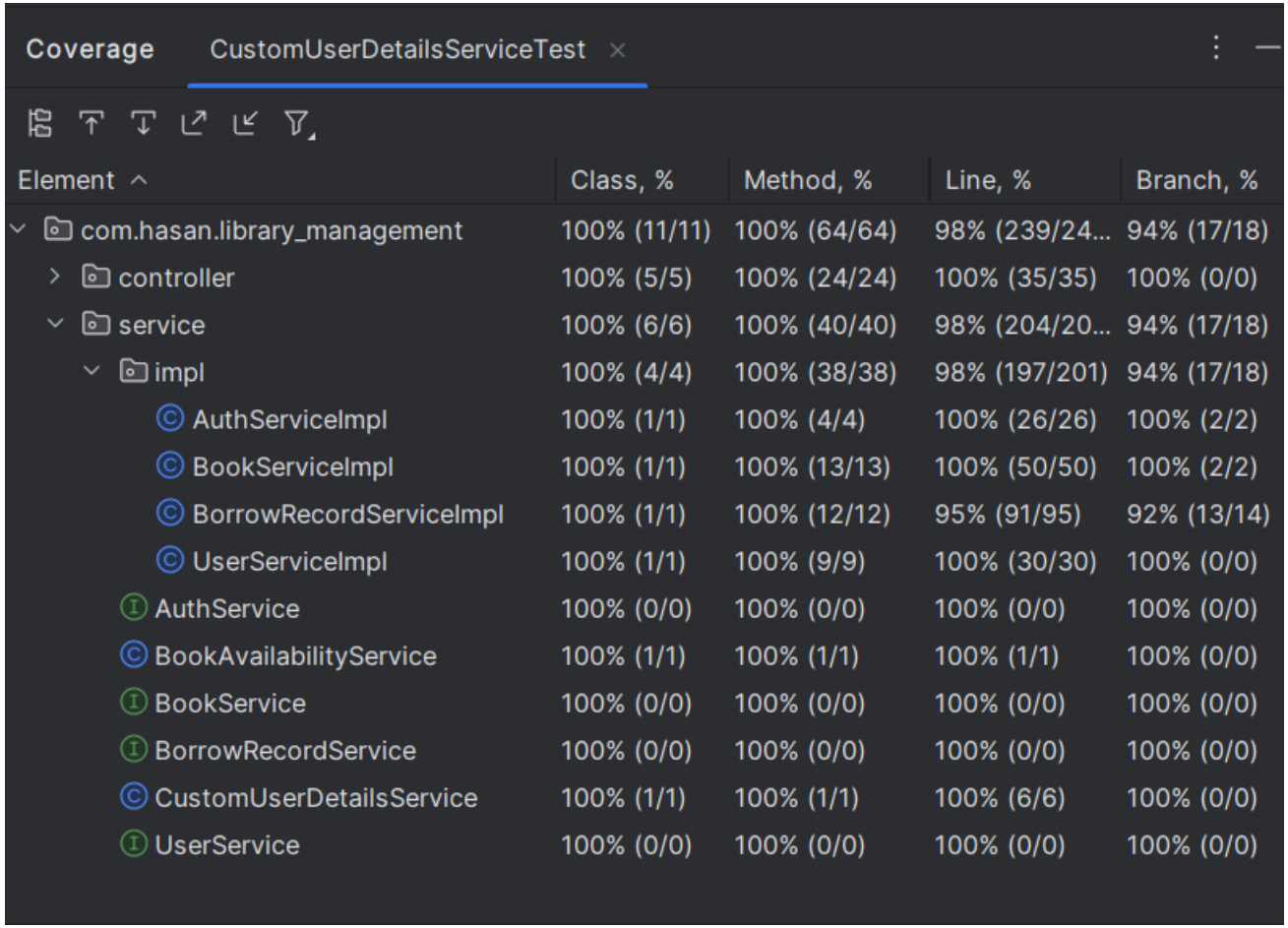
Integration Test – BorrowRecordControllerTest



Integration Test – UserControllerTest

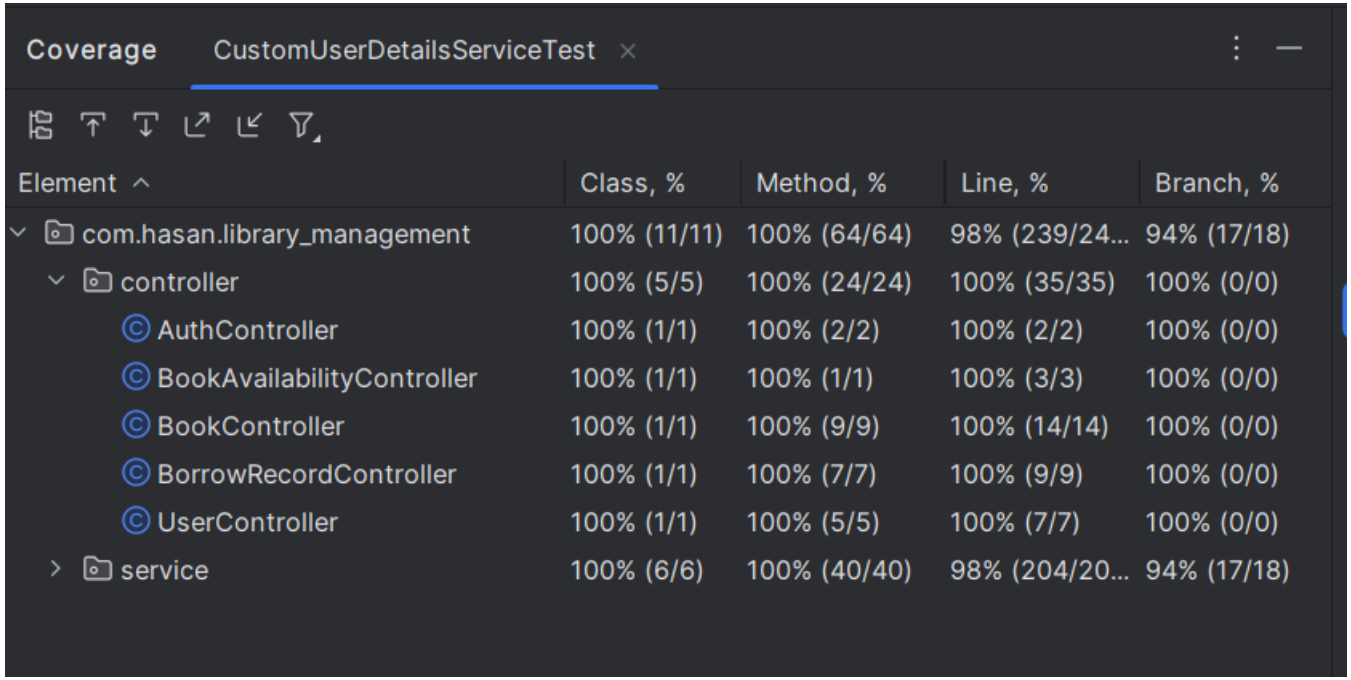


Service Layer Test Coverage

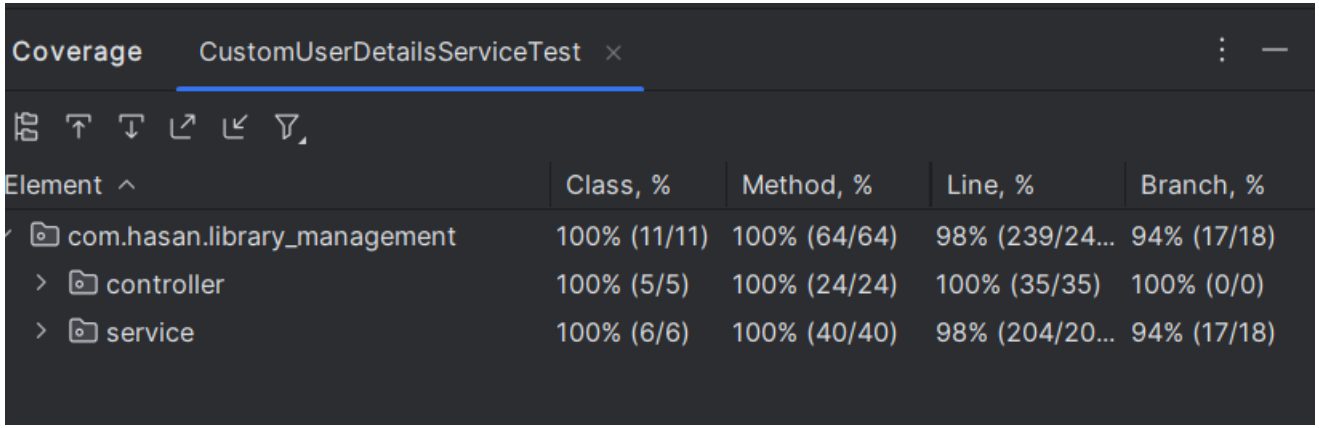


Element	Class, %	Method, %	Line, %	Branch, %
com.hasan.library_management	100% (11/11)	100% (64/64)	98% (239/24...	94% (17/18)
controller	100% (5/5)	100% (24/24)	100% (35/35)	100% (0/0)
service	100% (6/6)	100% (40/40)	98% (204/20...	94% (17/18)
impl	100% (4/4)	100% (38/38)	98% (197/201)	94% (17/18)
AuthServiceImpl	100% (1/1)	100% (4/4)	100% (26/26)	100% (2/2)
BookServiceImpl	100% (1/1)	100% (13/13)	100% (50/50)	100% (2/2)
BorrowRecordServiceImpl	100% (1/1)	100% (12/12)	95% (91/95)	92% (13/14)
UserServiceImpl	100% (1/1)	100% (9/9)	100% (30/30)	100% (0/0)
AuthService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
BookAvailabilityService	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
BookService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
BorrowRecordService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
CustomUserDetailsService	100% (1/1)	100% (1/1)	100% (6/6)	100% (0/0)
UserService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)

Controller Layer Test Coverage

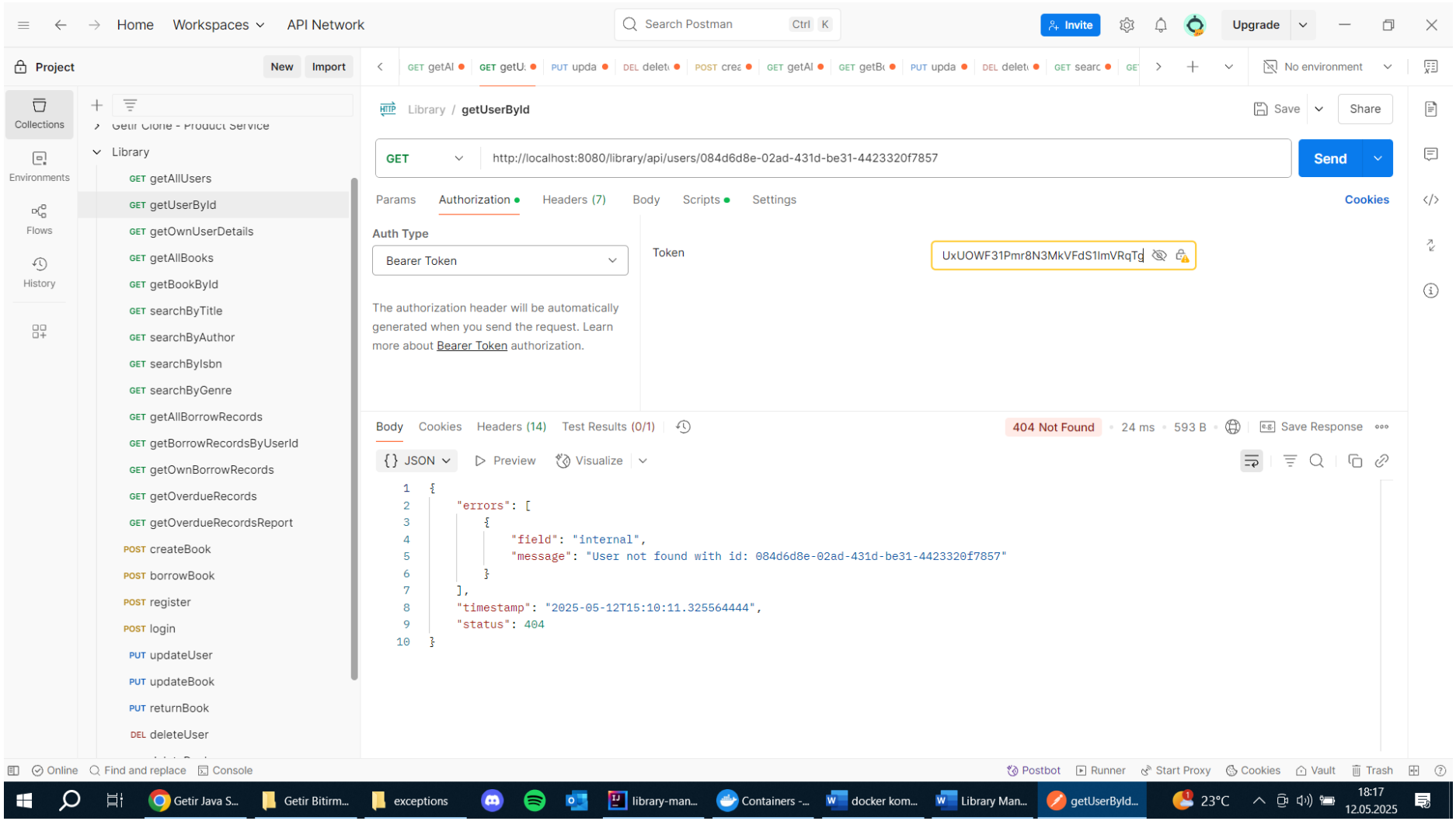


Service & Controller Layers Test Coverage

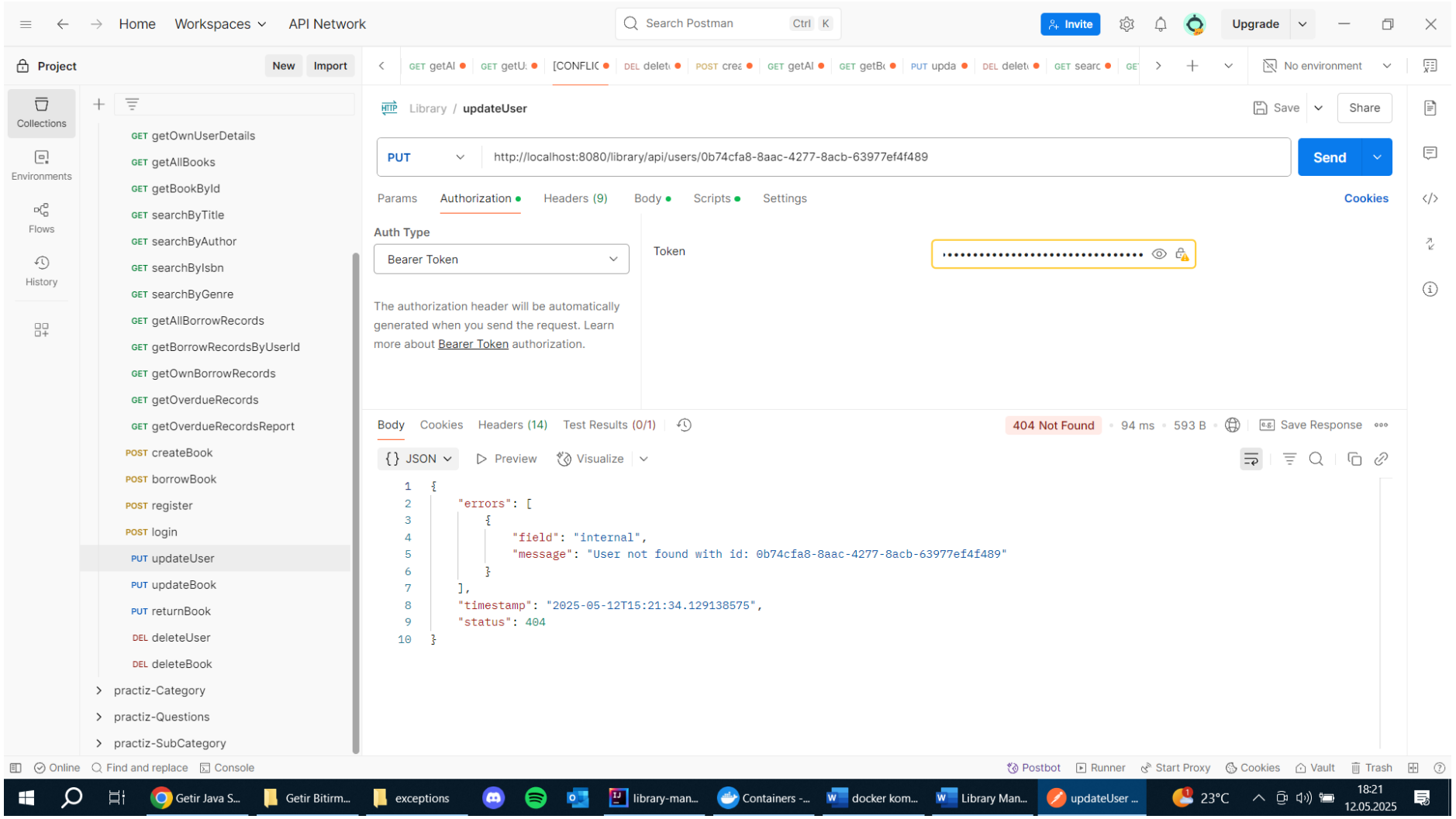


Exceptions

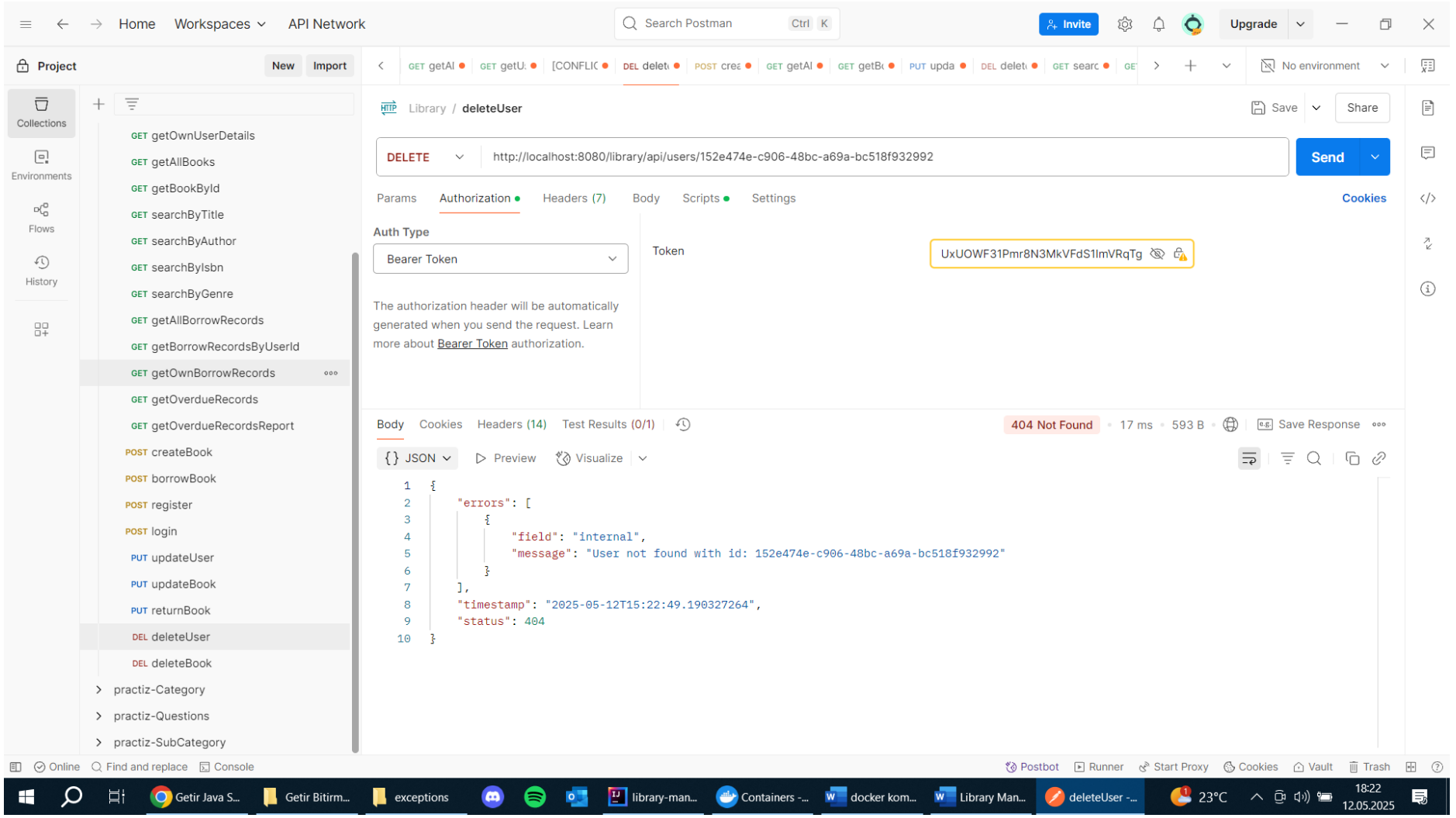
GET /users/{id} – Query user with invalid ID



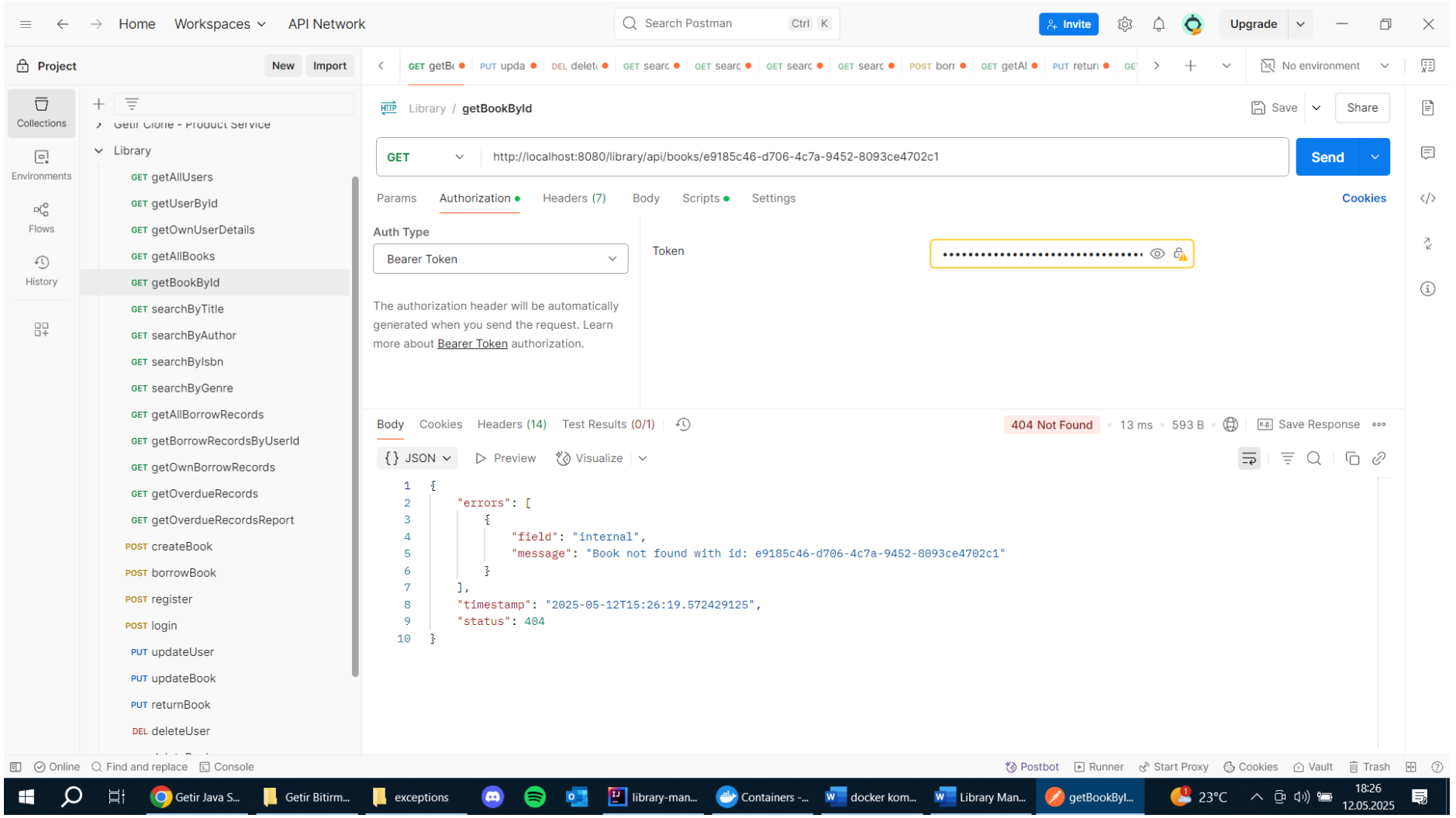
PUT /admin/users/{id} – Update non-existent user



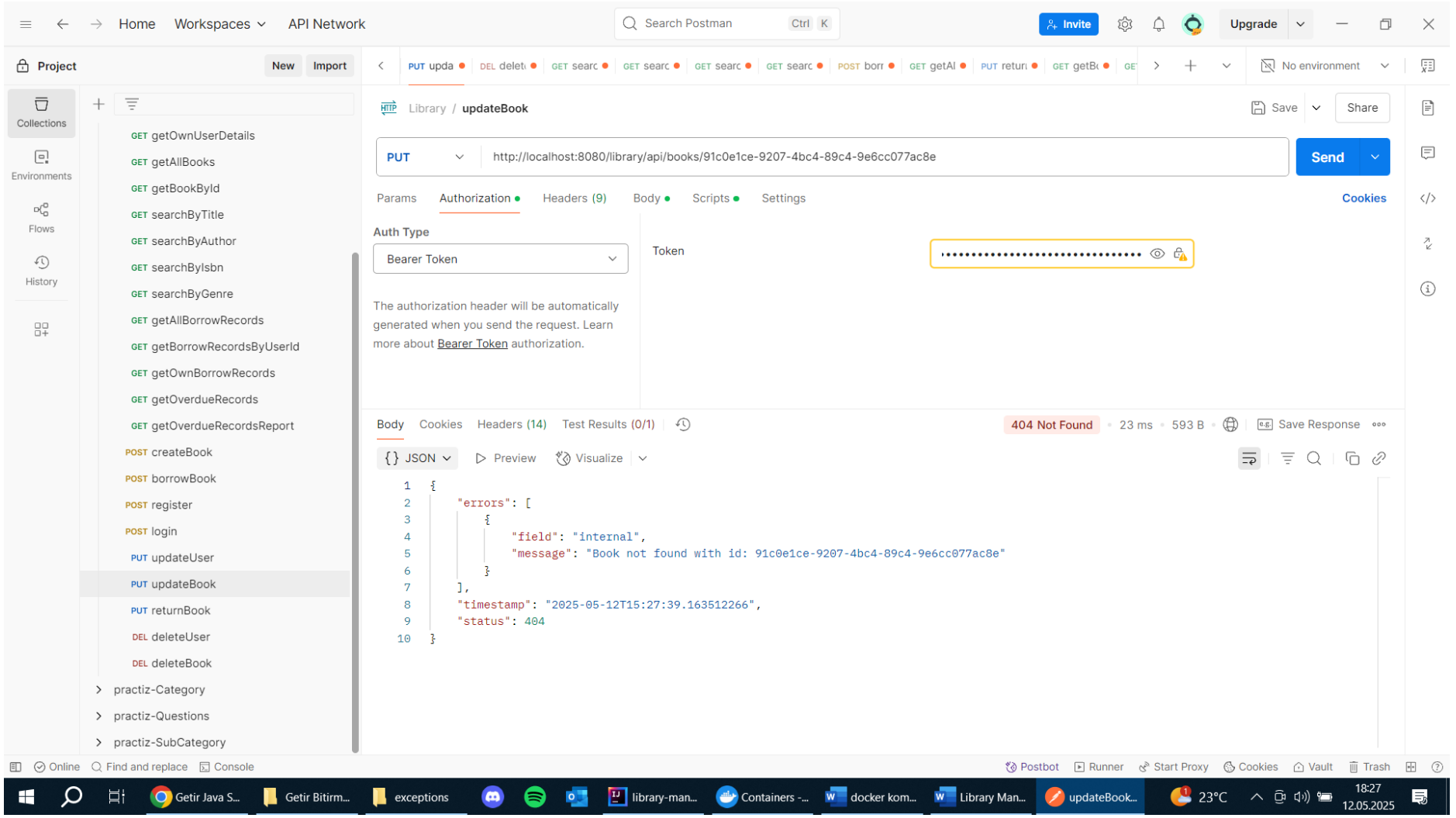
DELETE /admin/users/{id} – Delete a user that does not exist



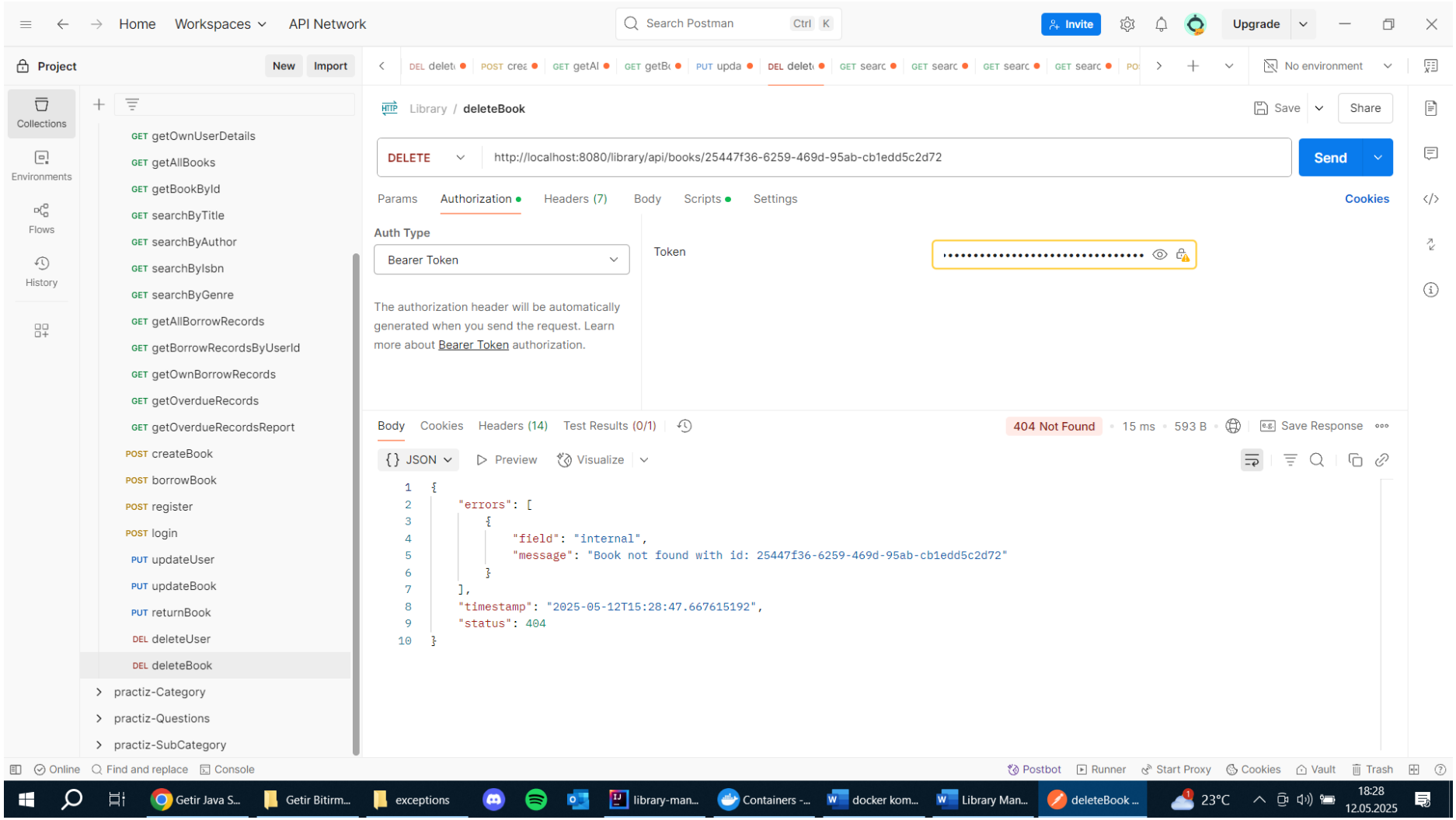
GET /books/{id} – Book not found



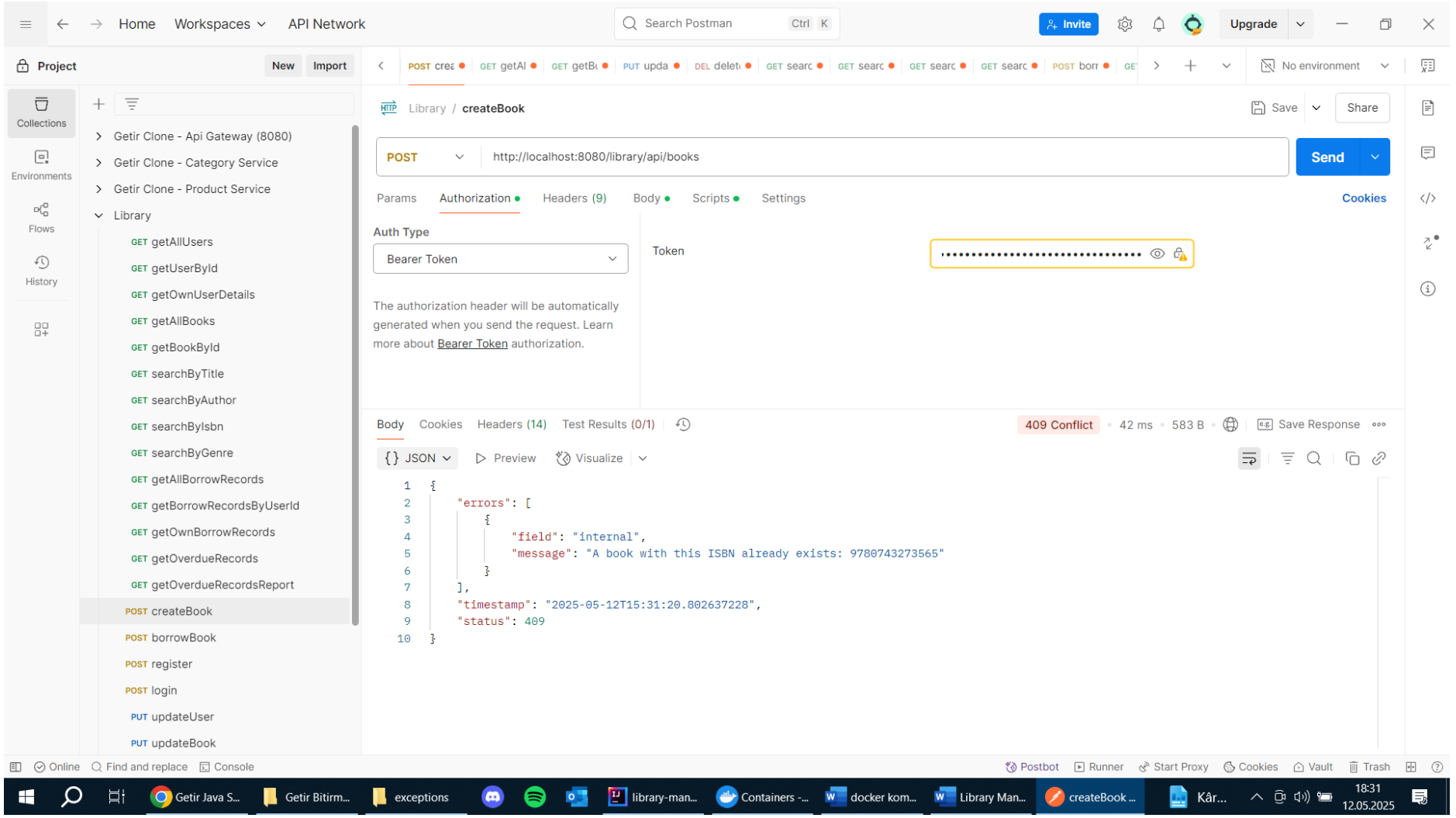
PUT /books/{id} – Book not found during update



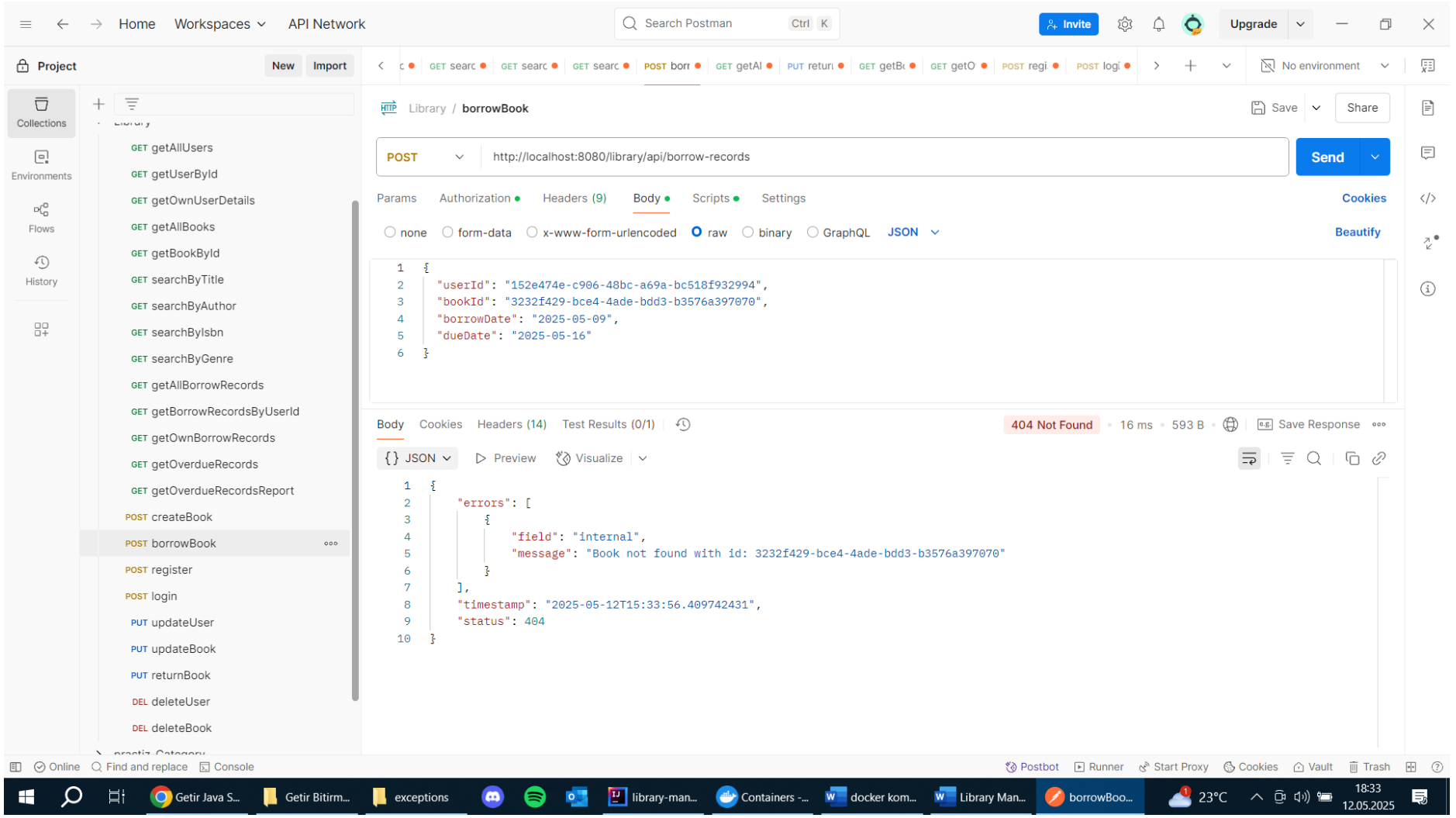
DELETE /books/{id} – Book not found during delete



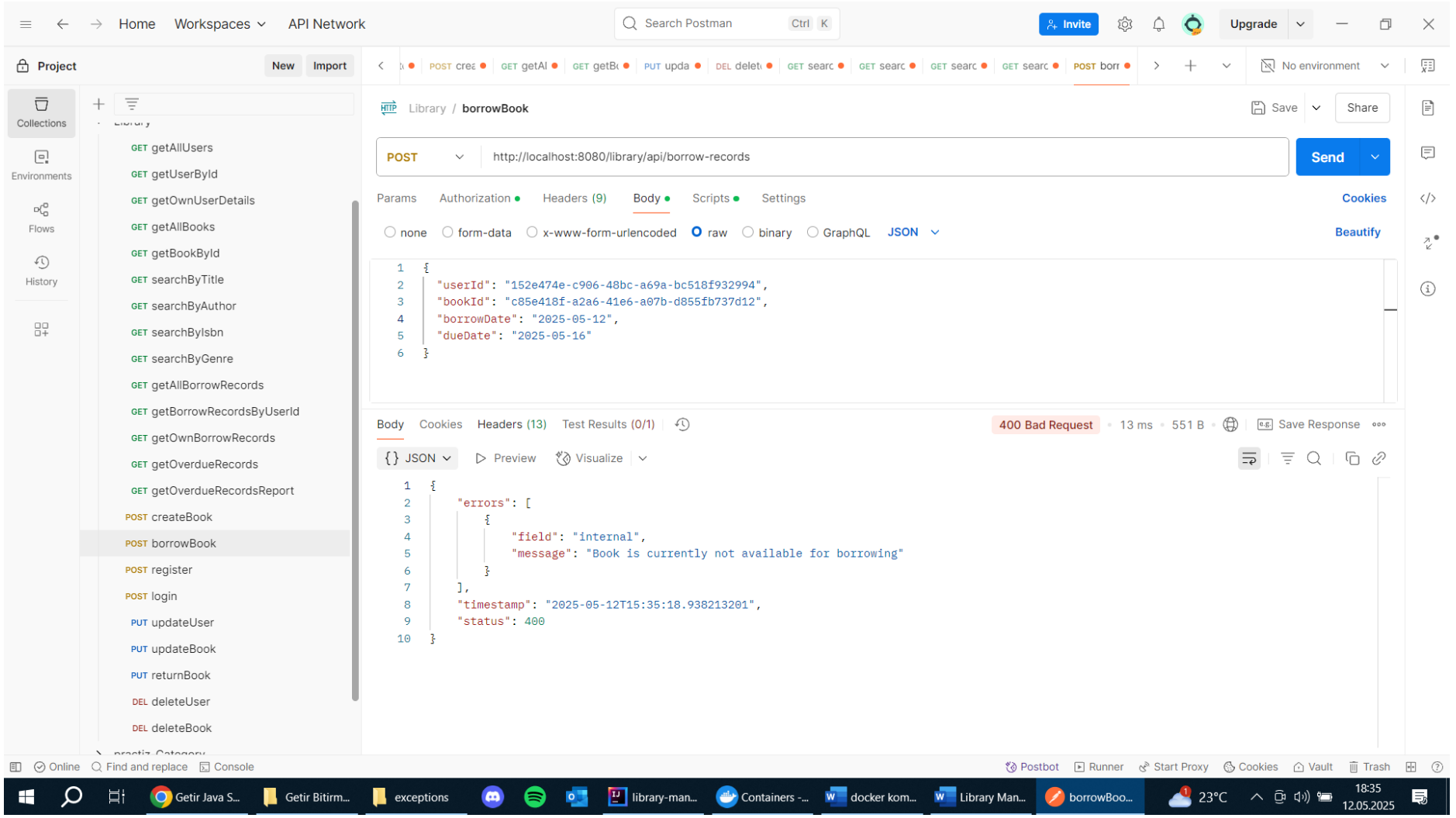
POST /books – Duplicate ISBN conflict



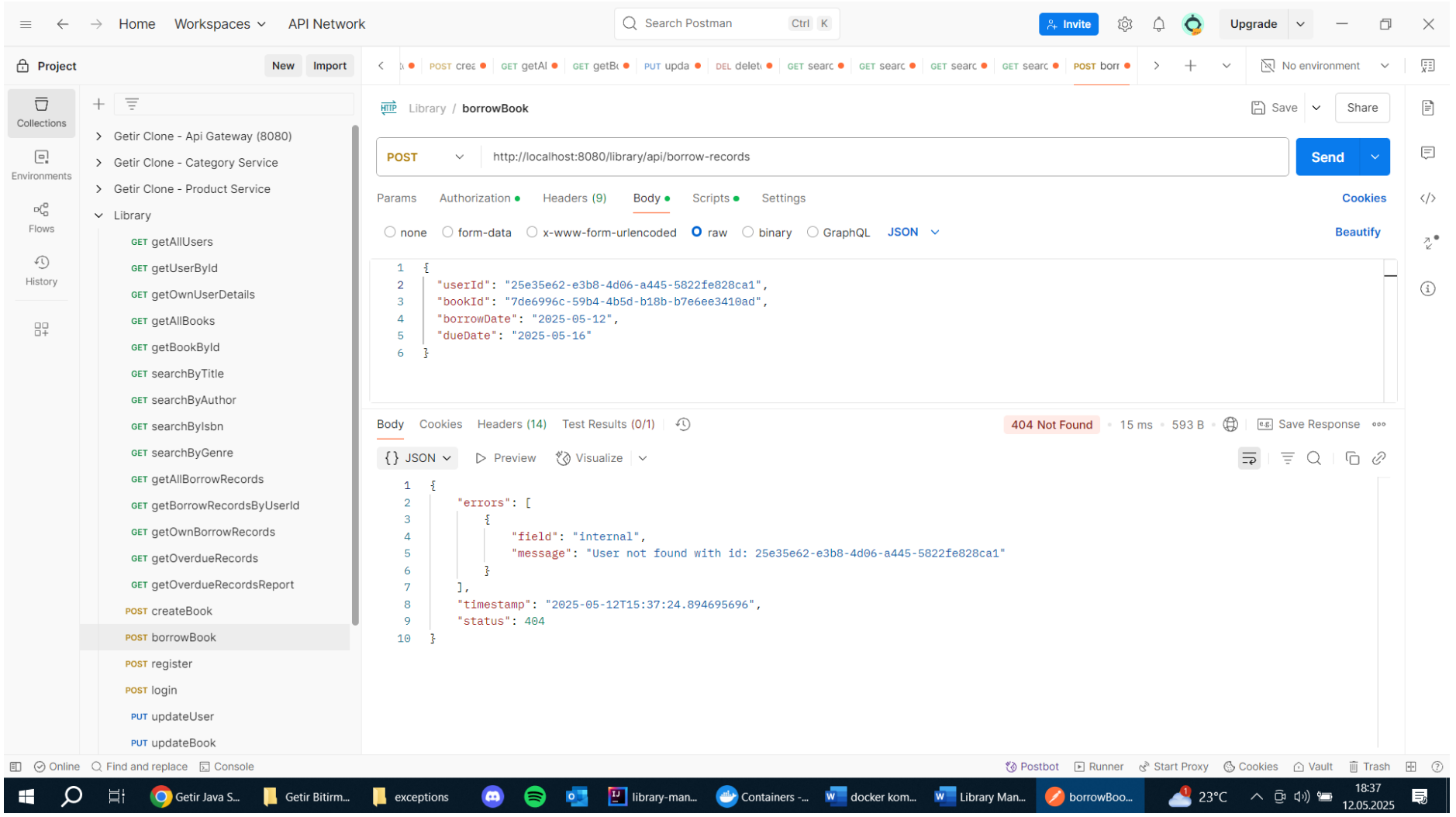
POST /borrow-records – Book not found



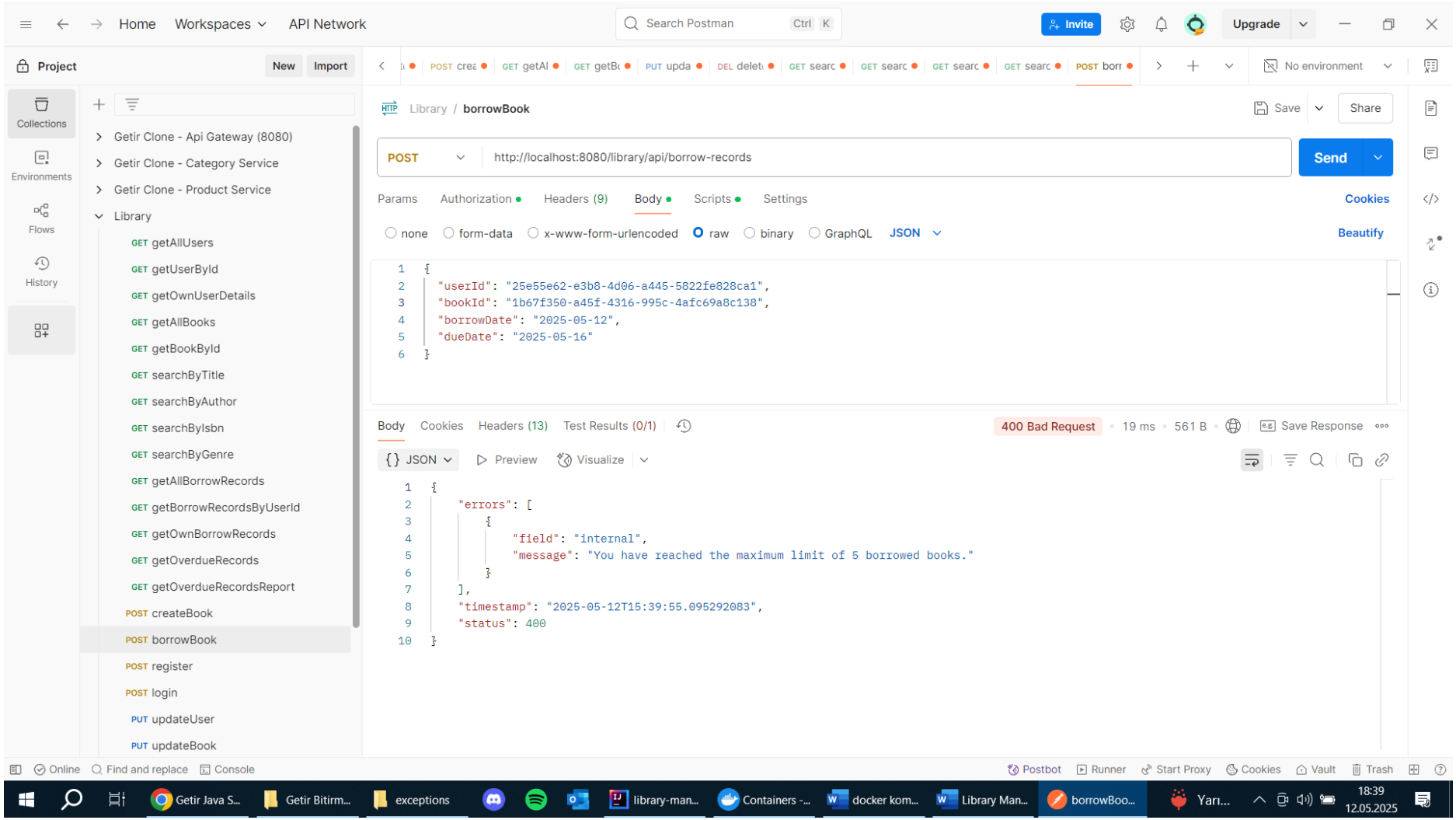
POST /borrow-records – Book not available



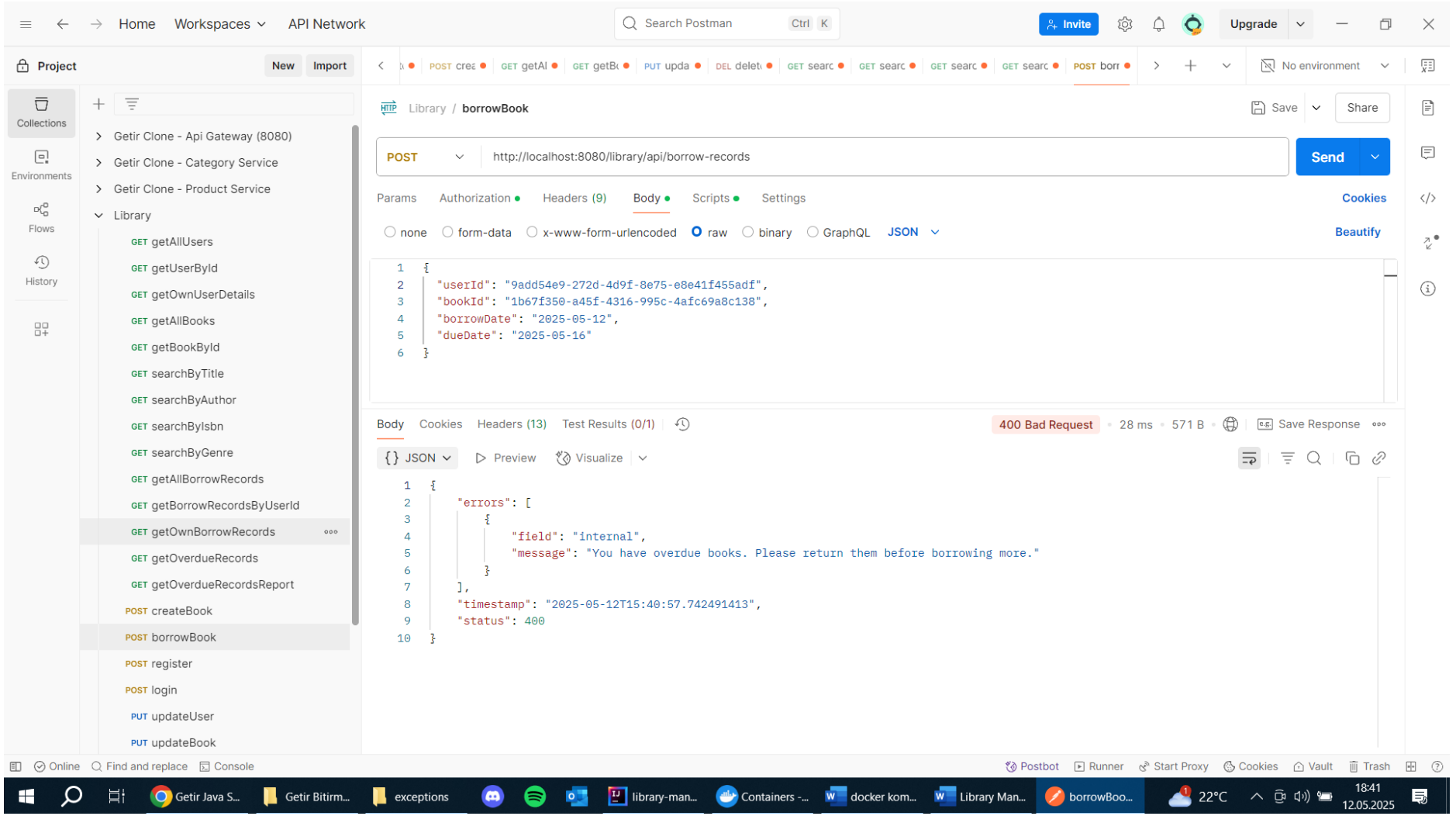
POST /borrow-records – User not found



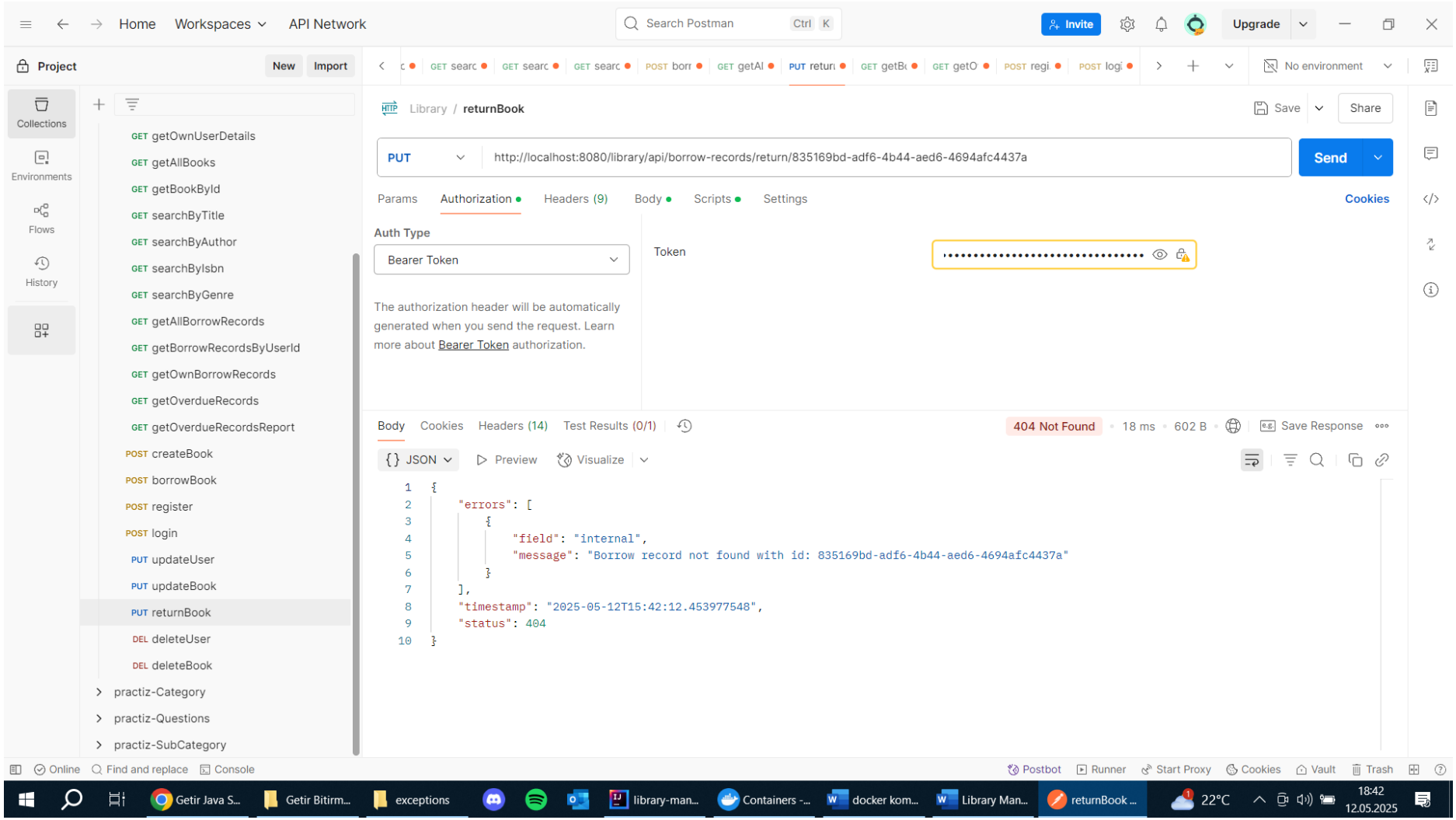
POST /borrow-records – Borrow limit exceeded



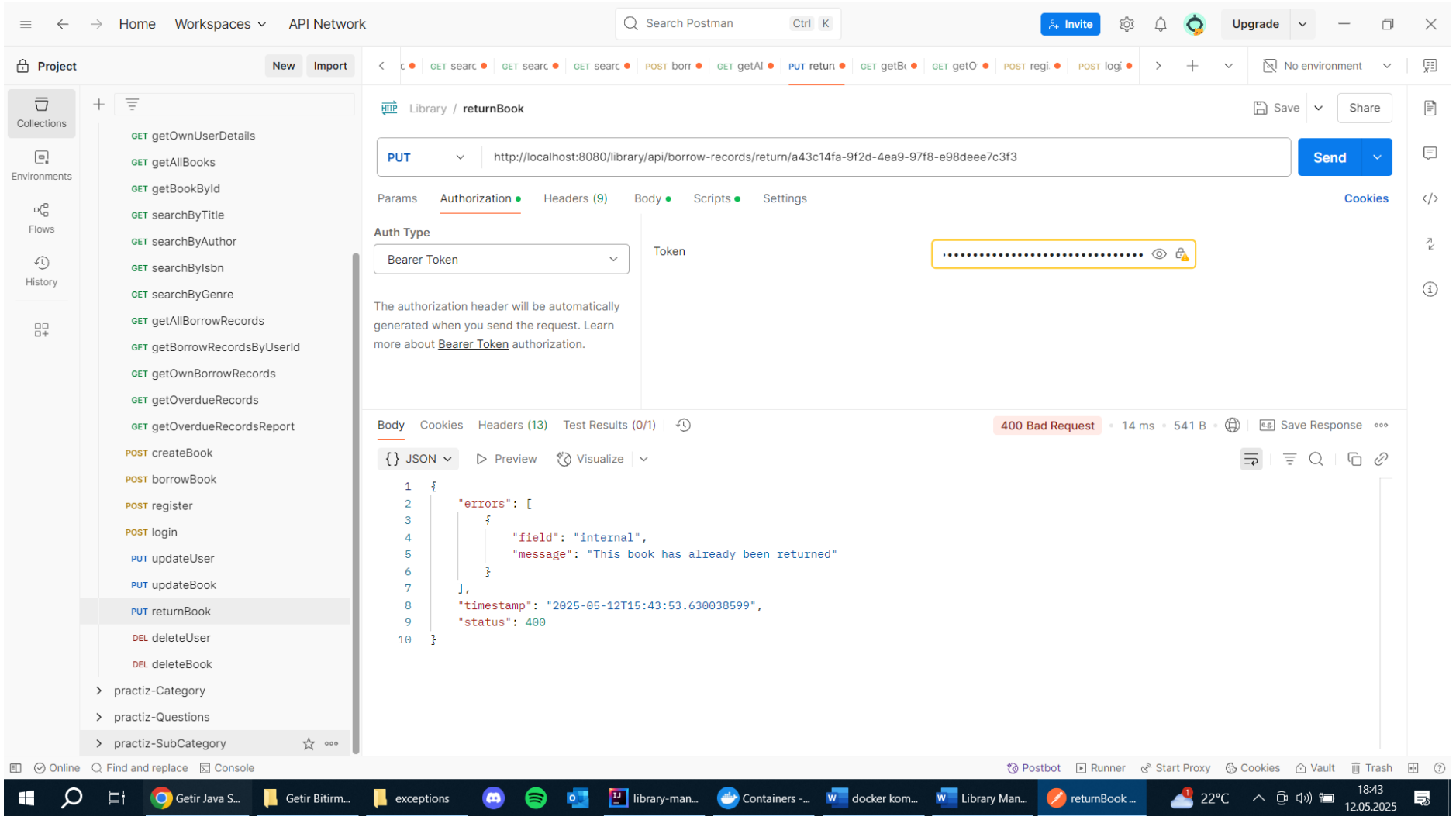
POST /borrow-records – User has overdue books



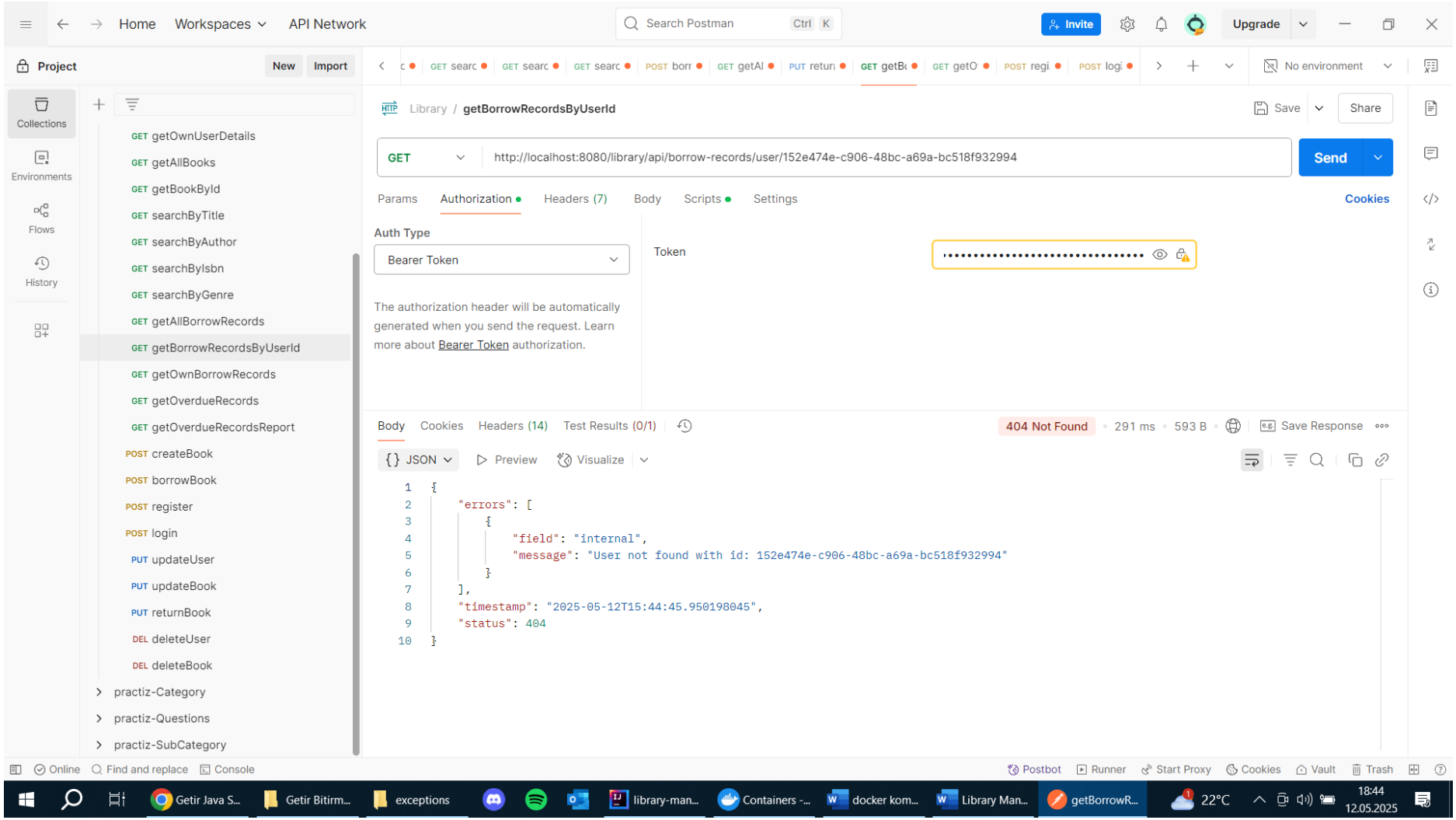
PUT /borrow-records/return/{id} – Borrow record not found



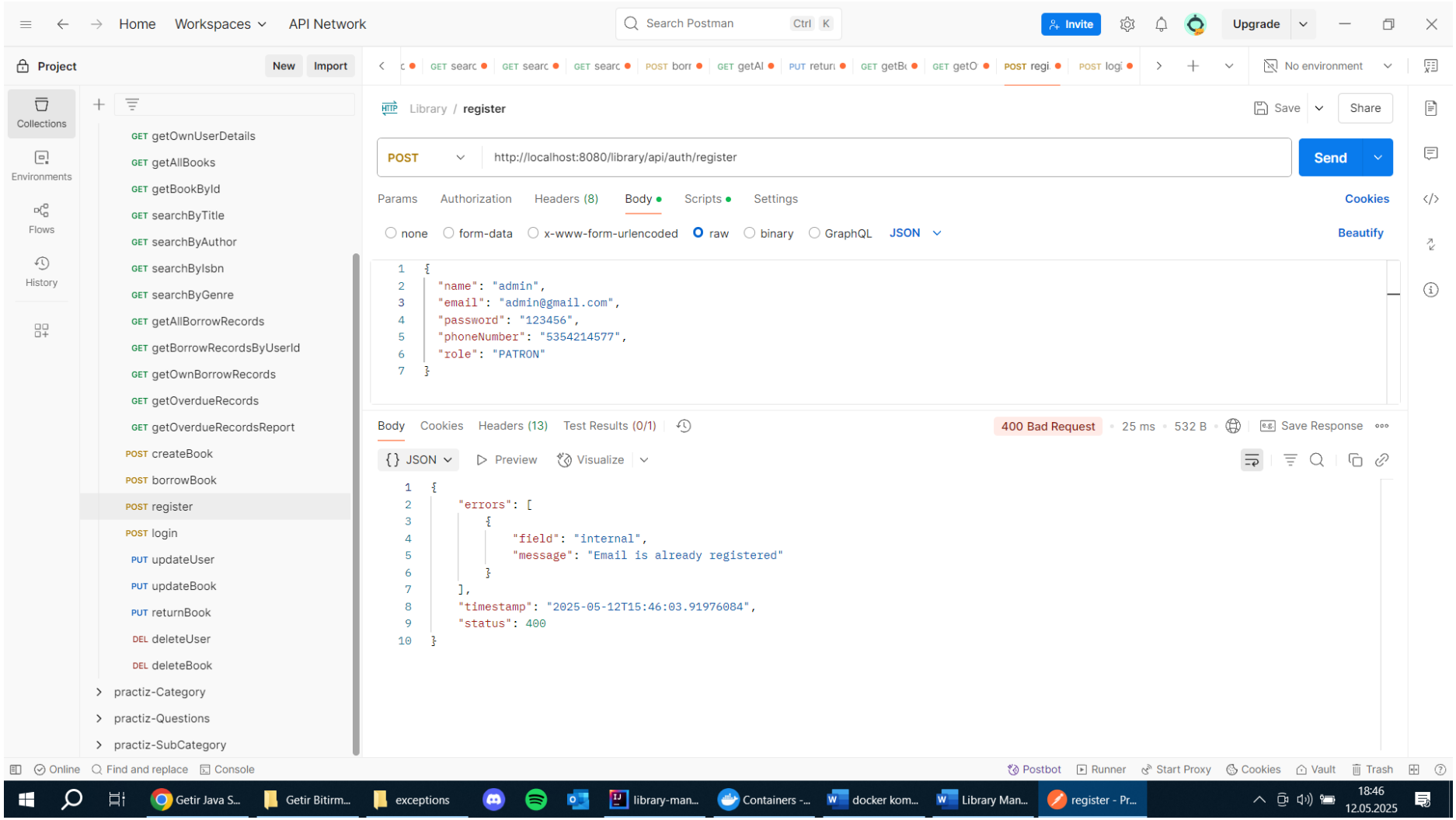
PUT /borrow-records/return/{id} – Book already returned



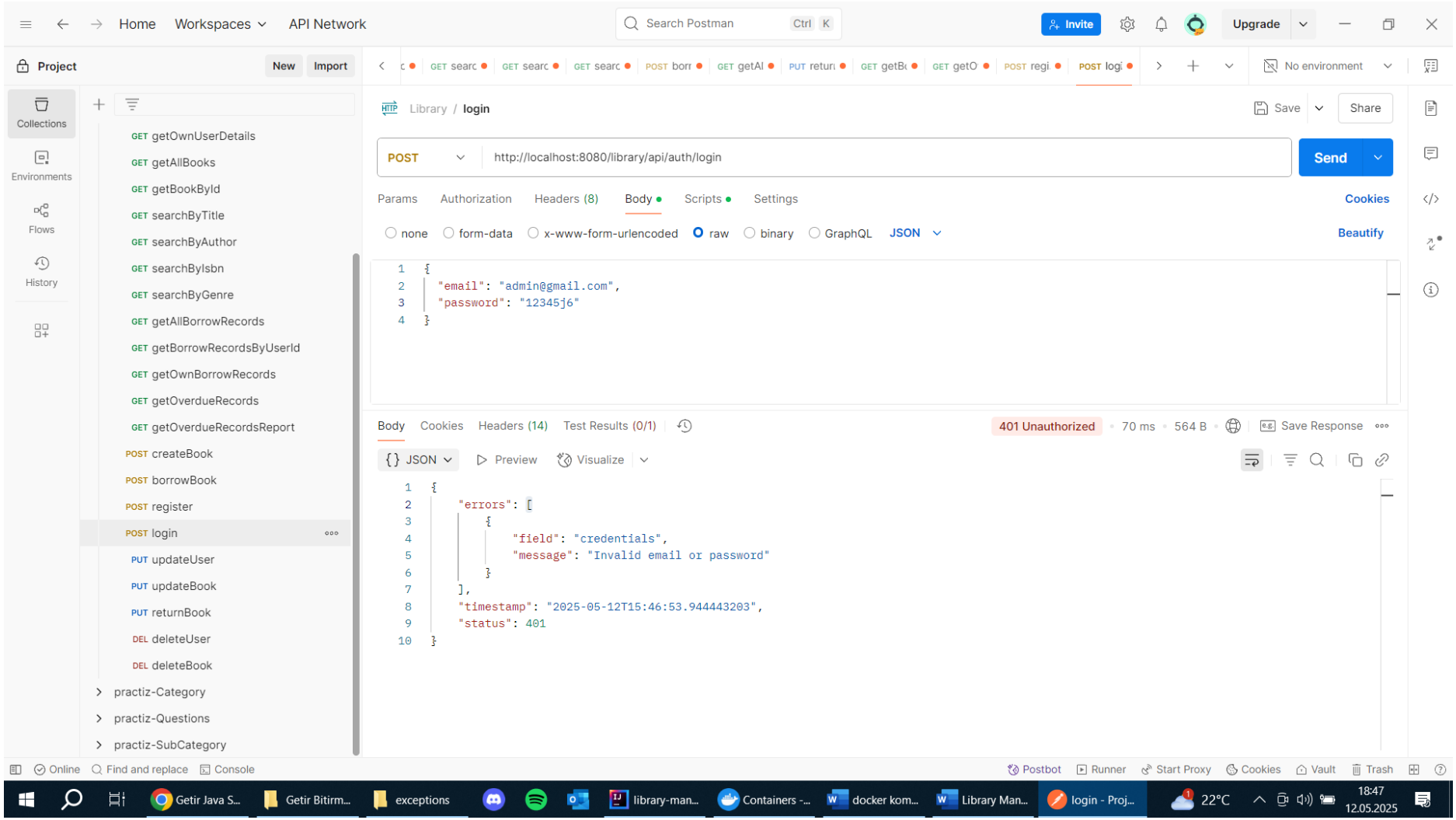
GET /borrow-records/user/{userId} – User not found



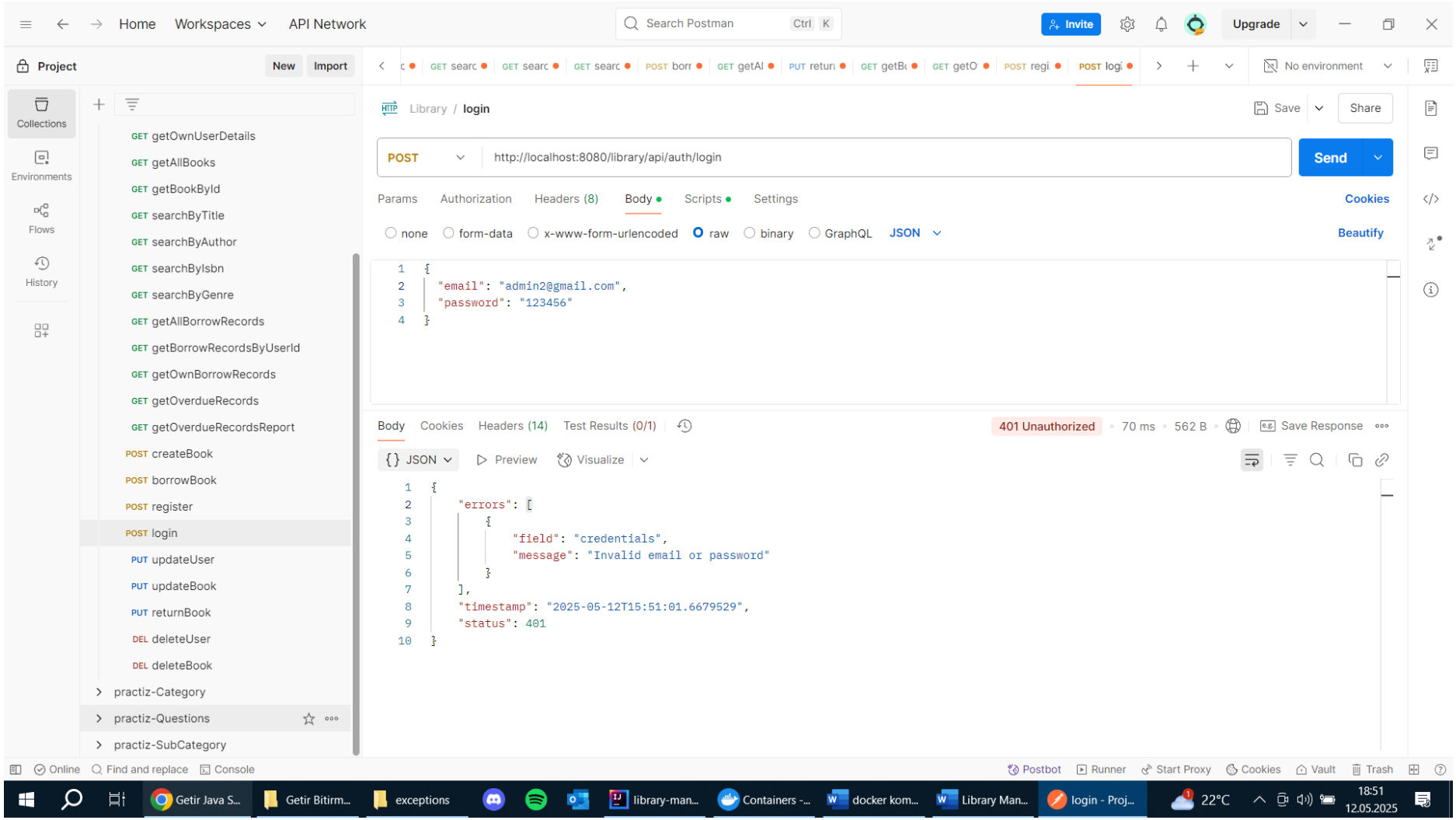
POST /auth/register – Email already registered



POST /auth/login – User exists but wrong password

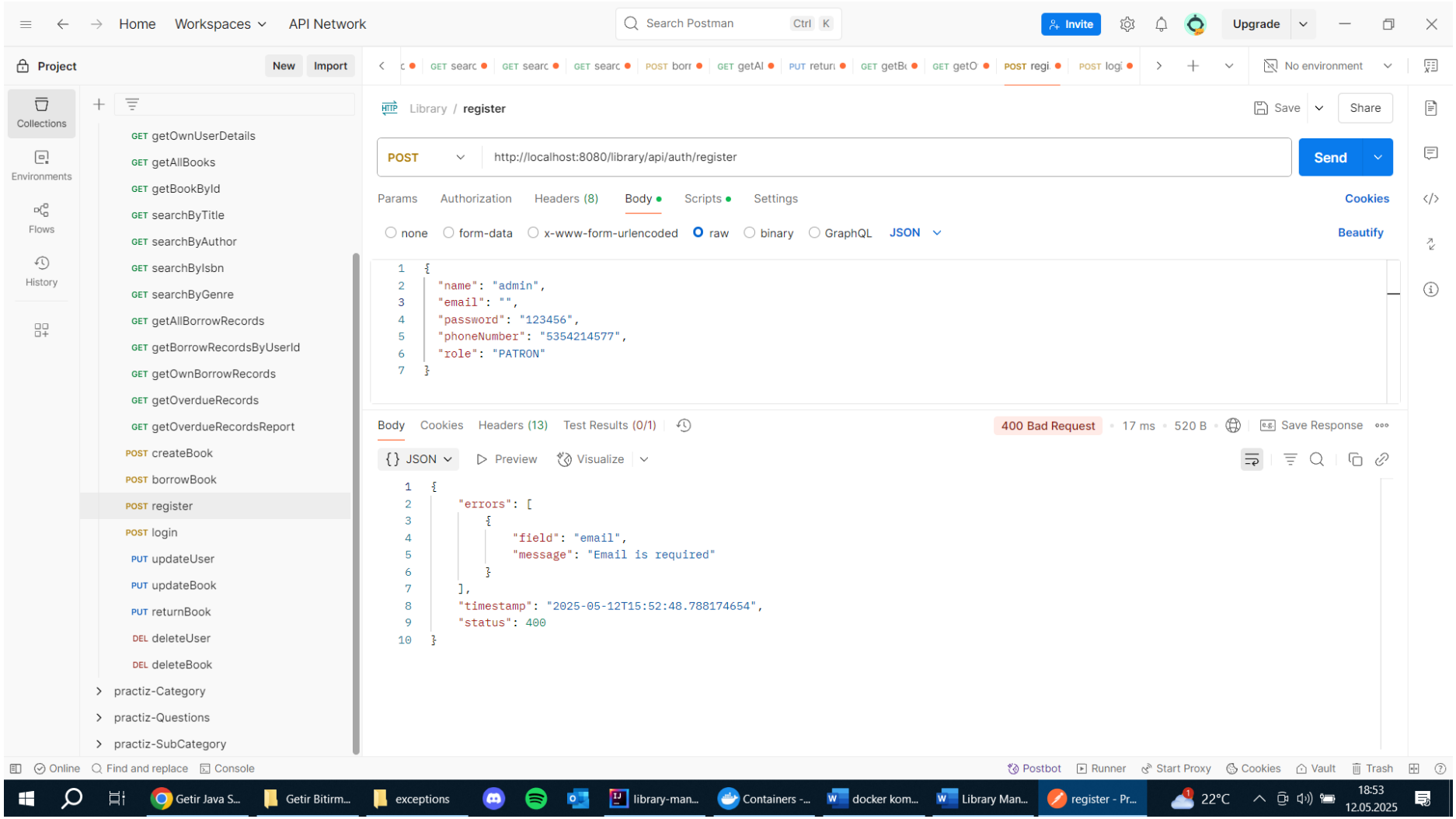


POST /auth/login –

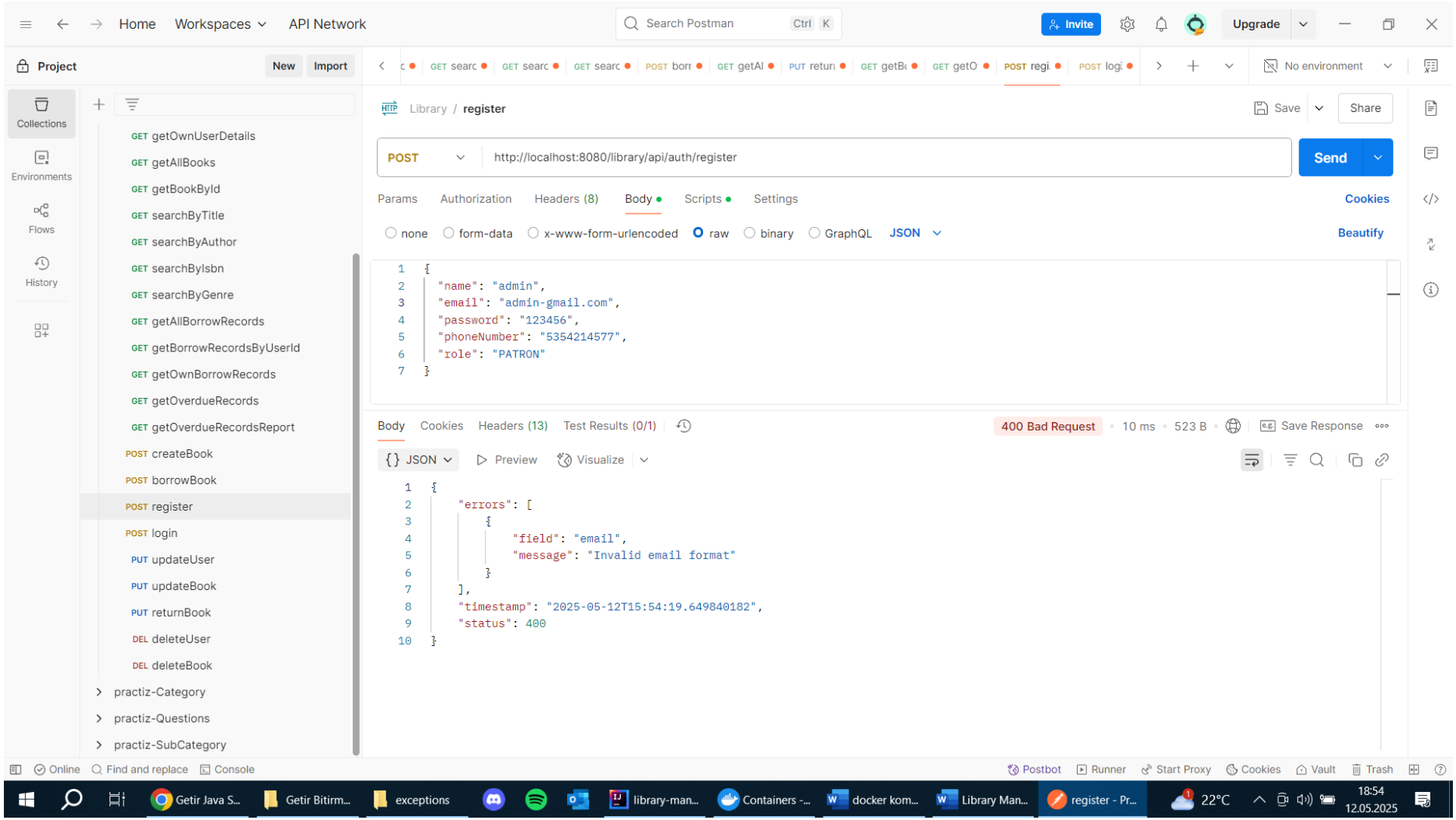


Validations

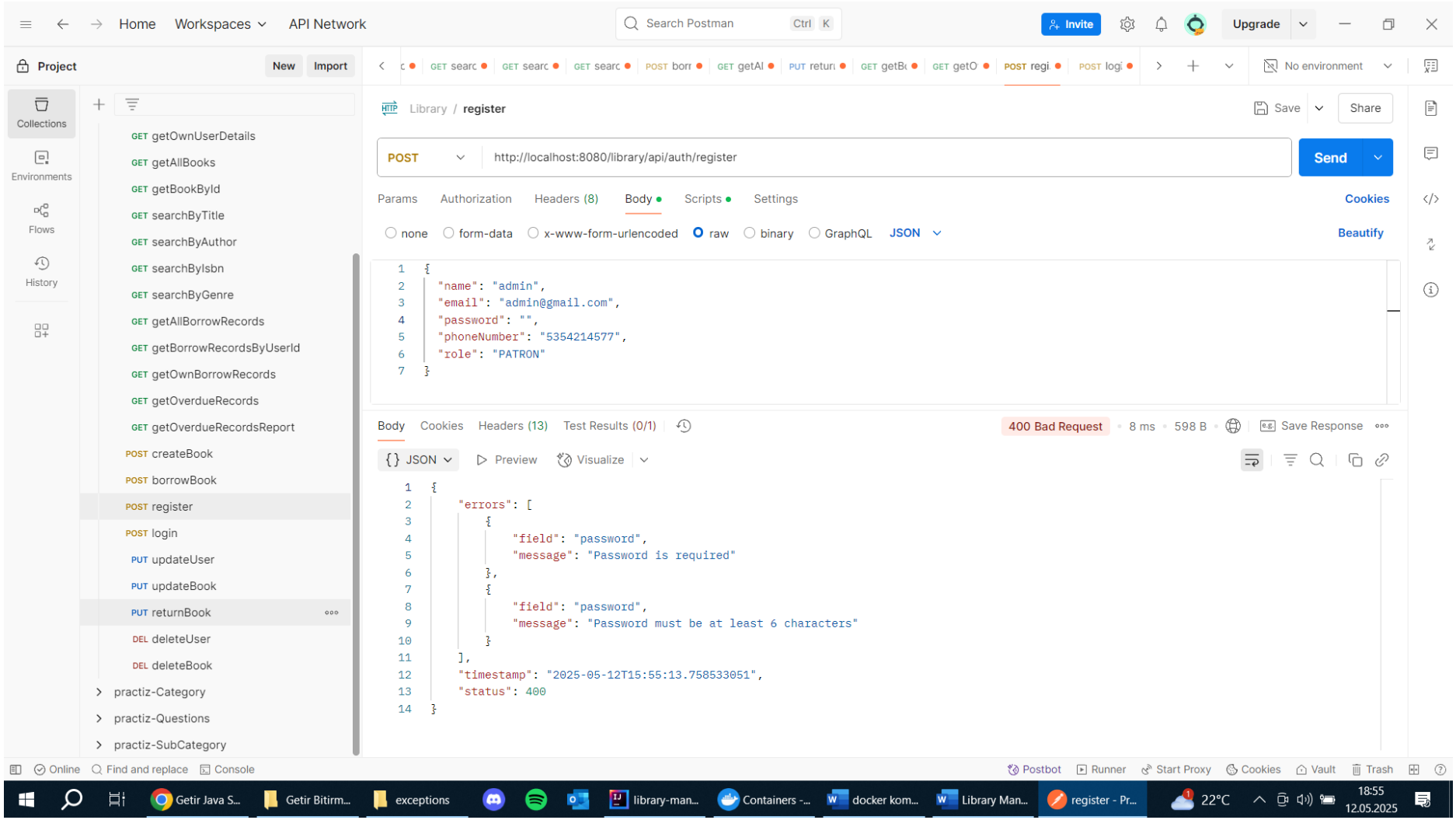
POST /auth/login – Missing email



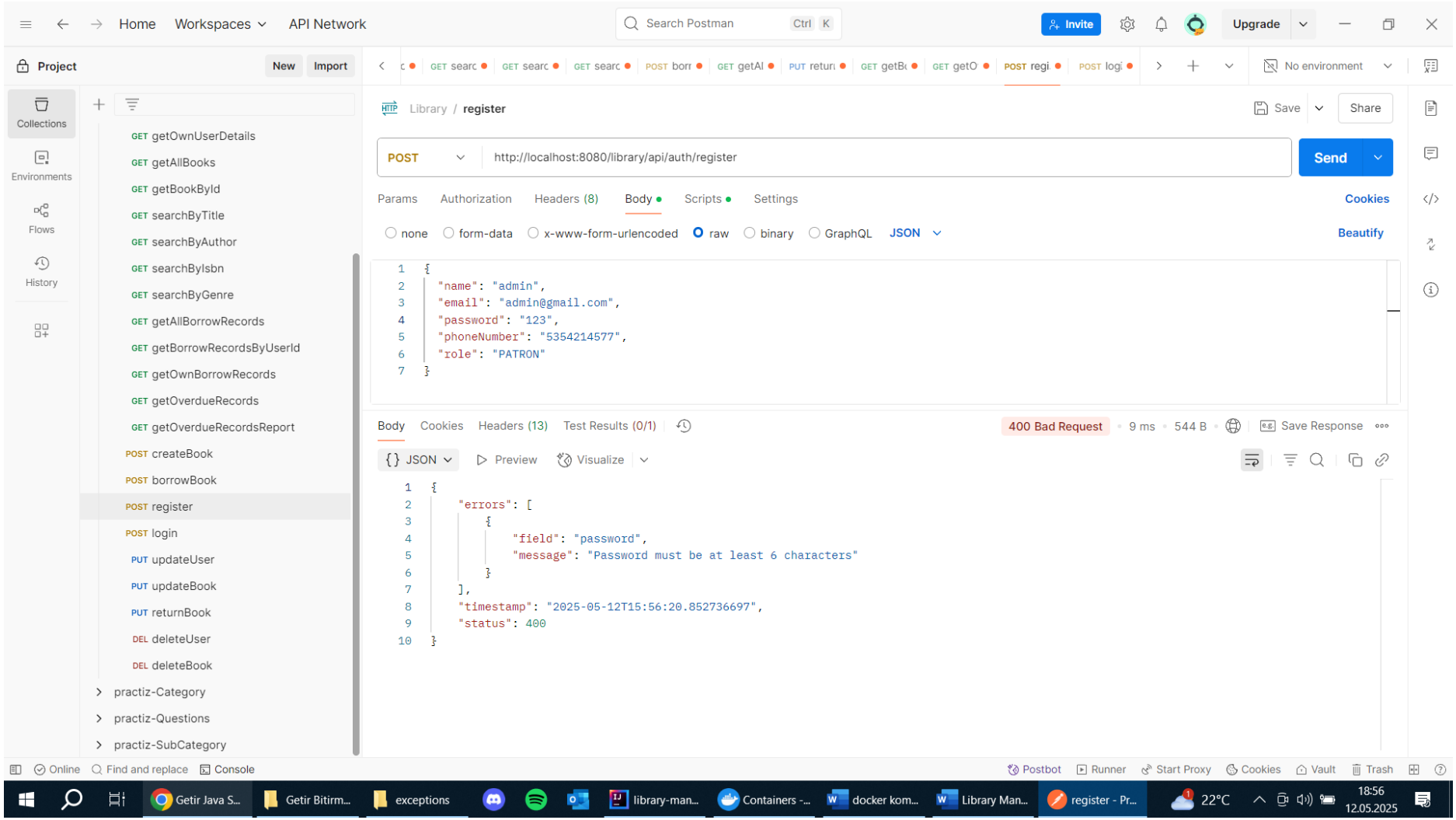
POST /auth/login – Invalid email format



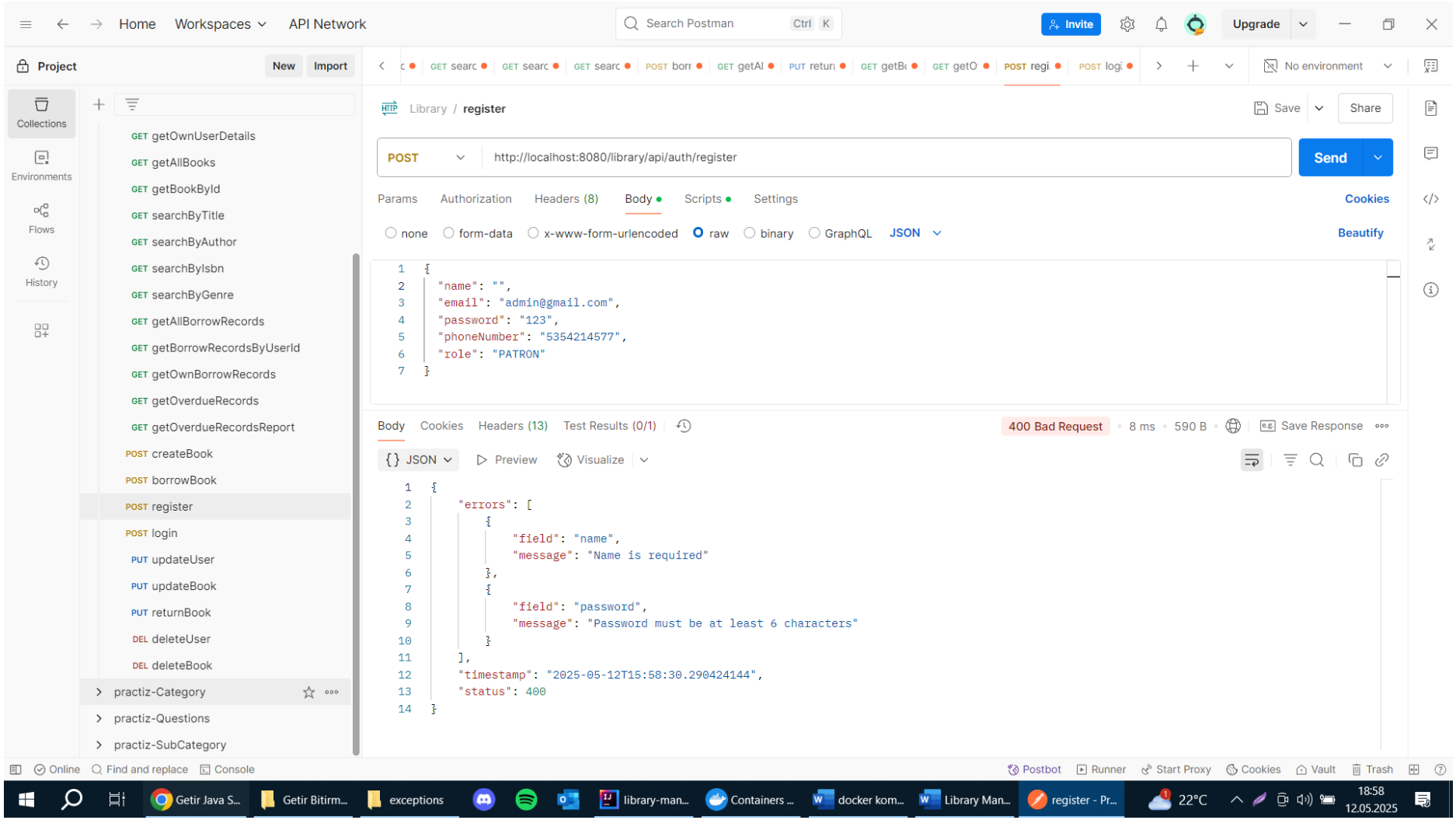
POST /auth/login – Missing password



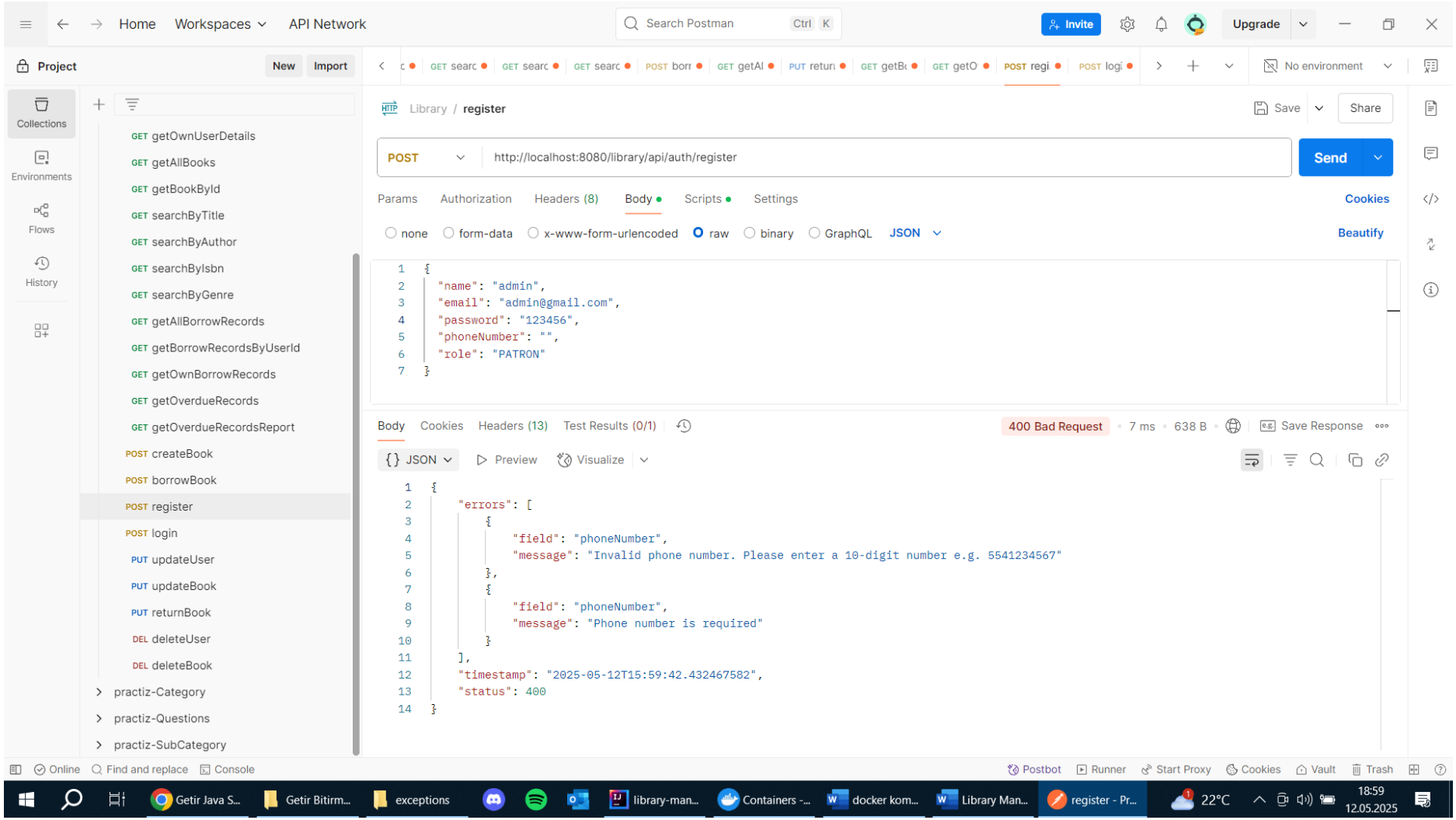
POST /auth/login – Password too short



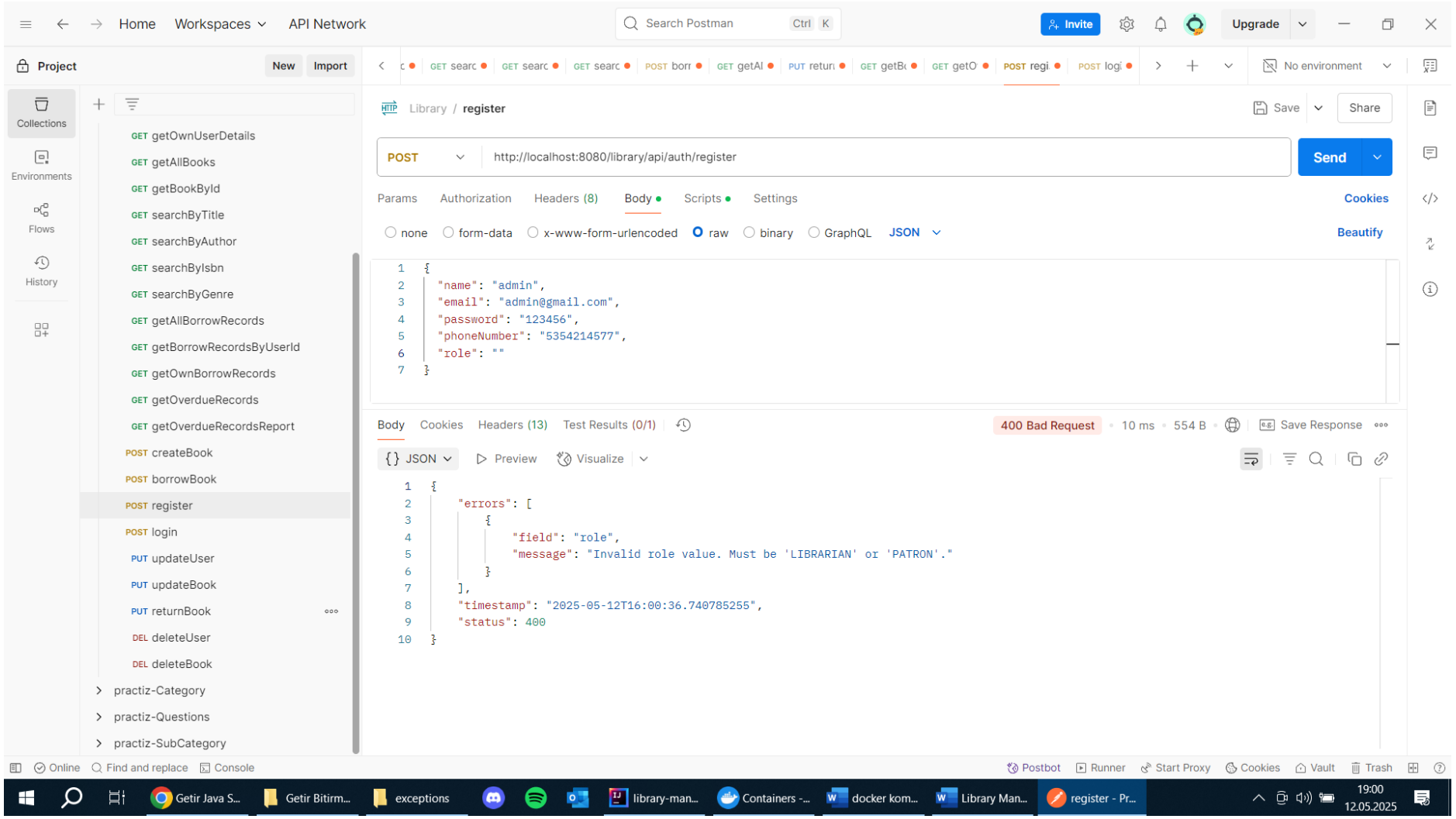
POST /users – Missing name



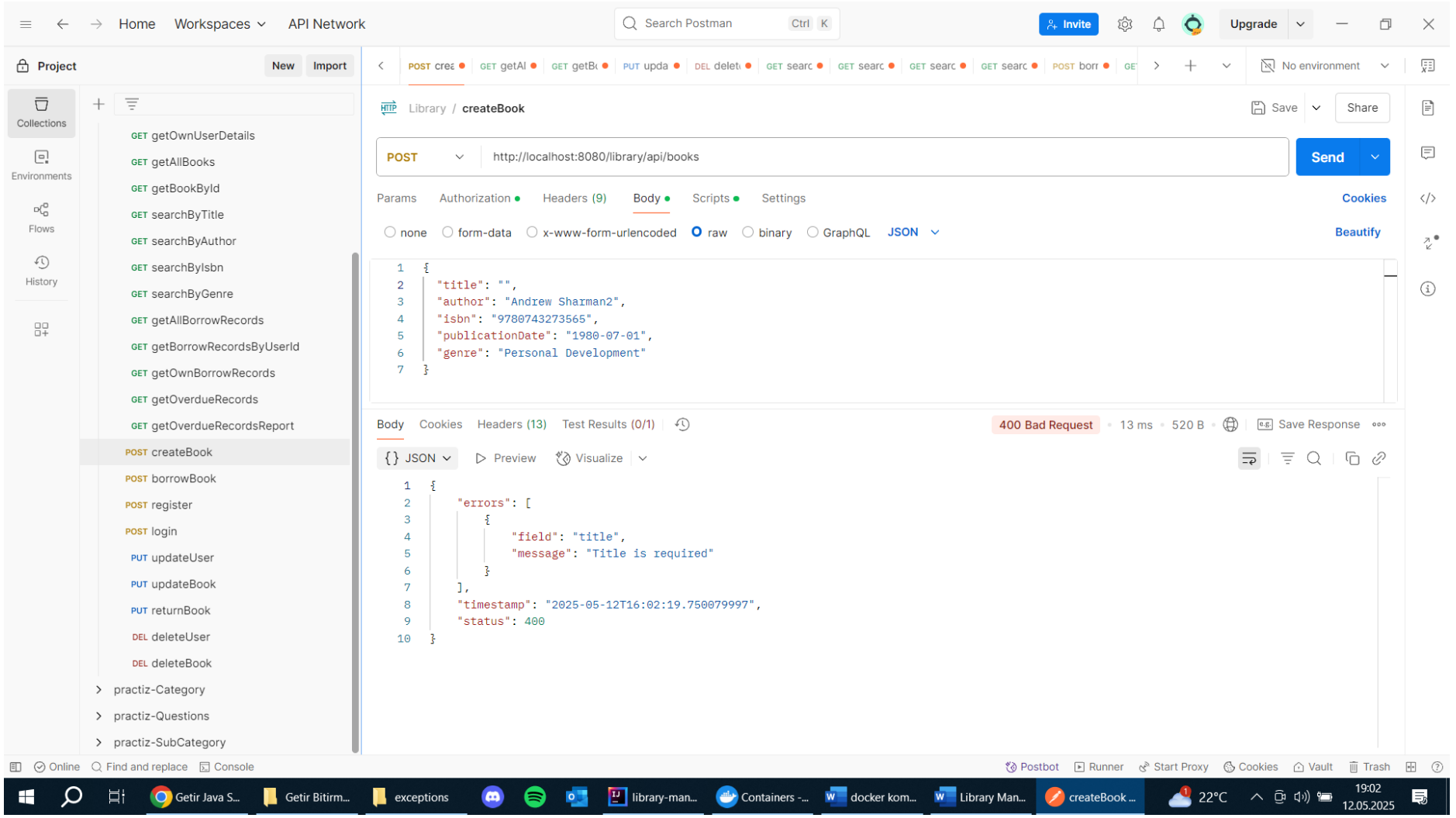
POST /users – Invalid phone number format



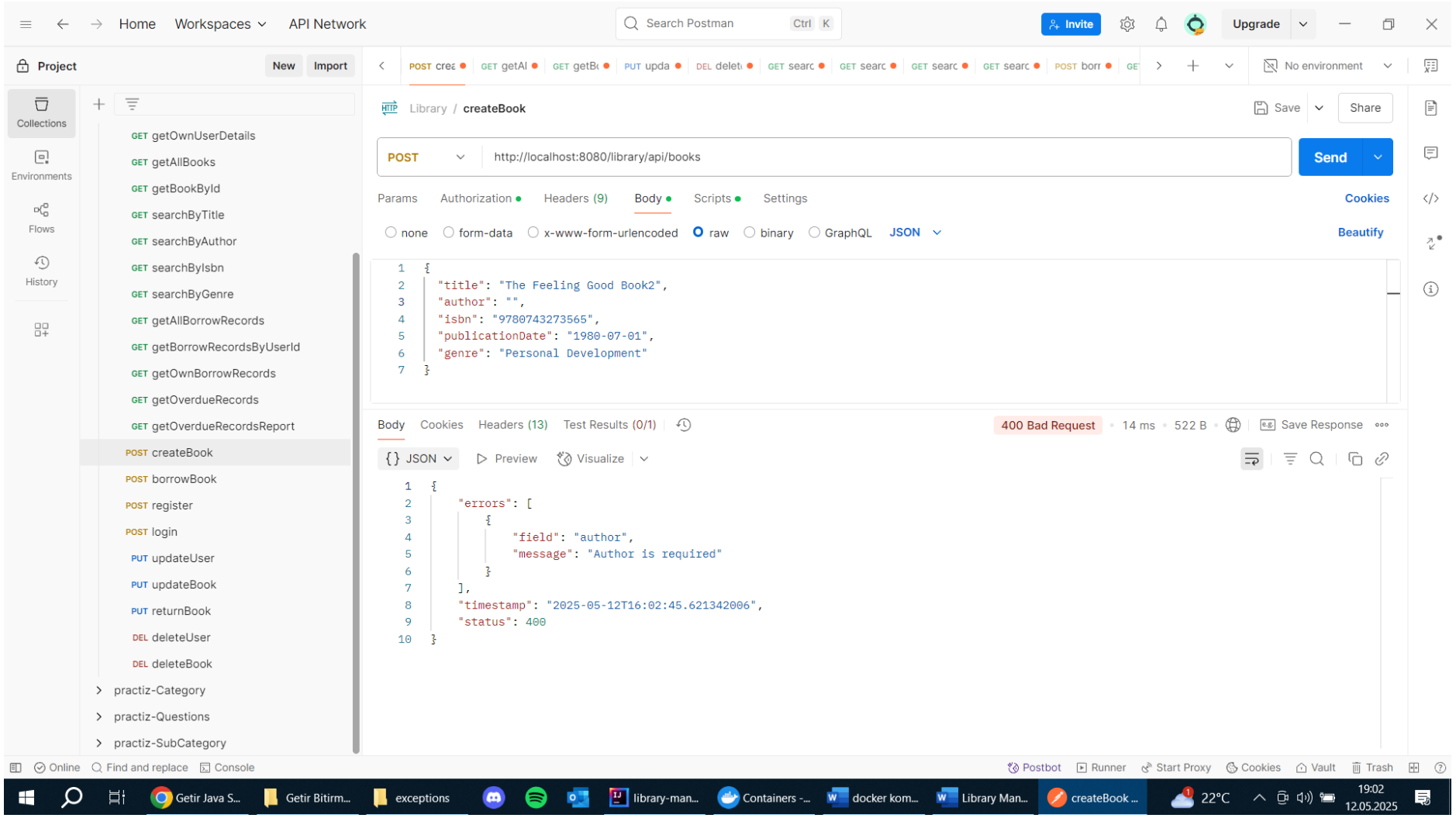
POST /users – Missing role



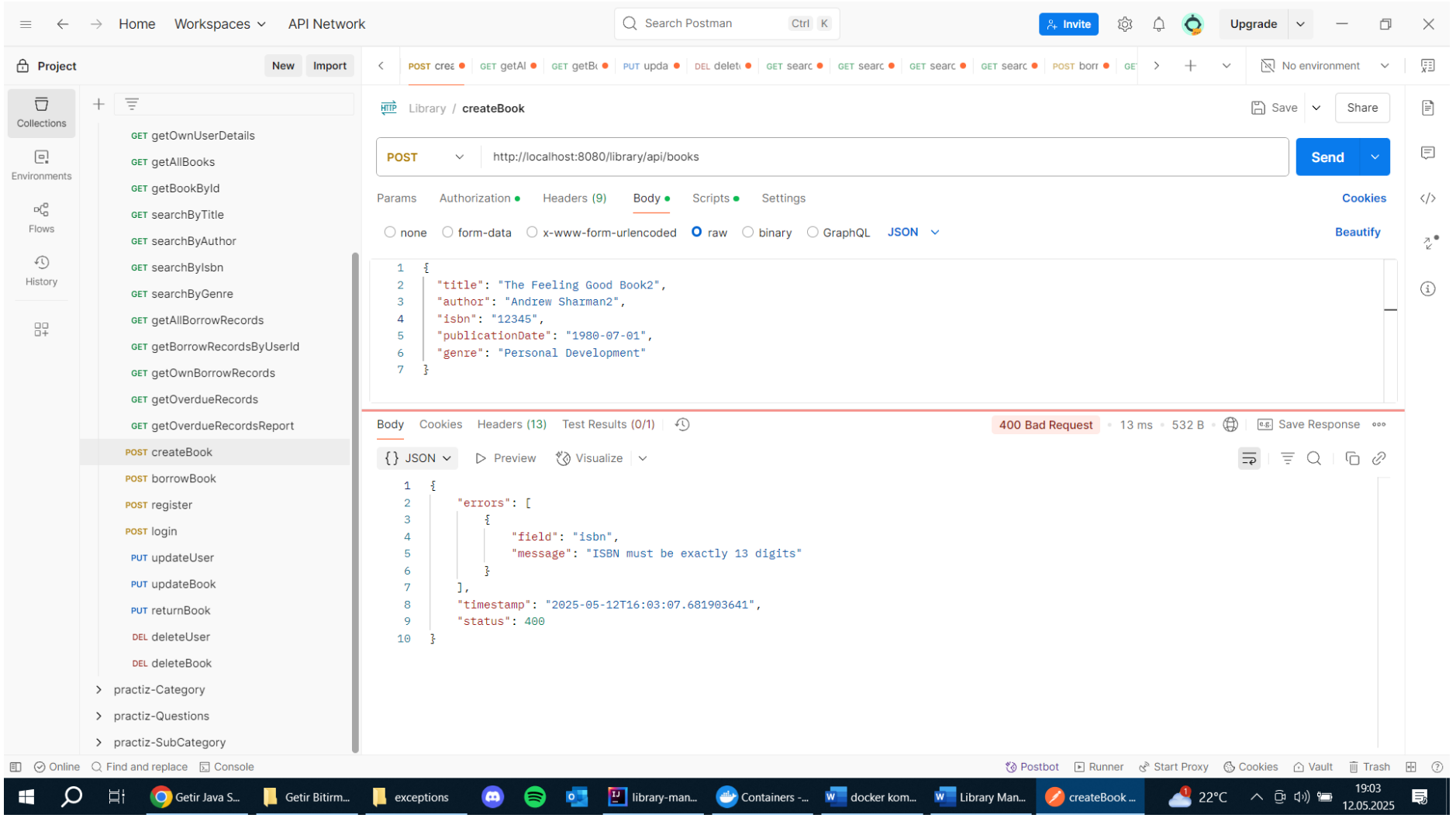
POST /books – Missing title



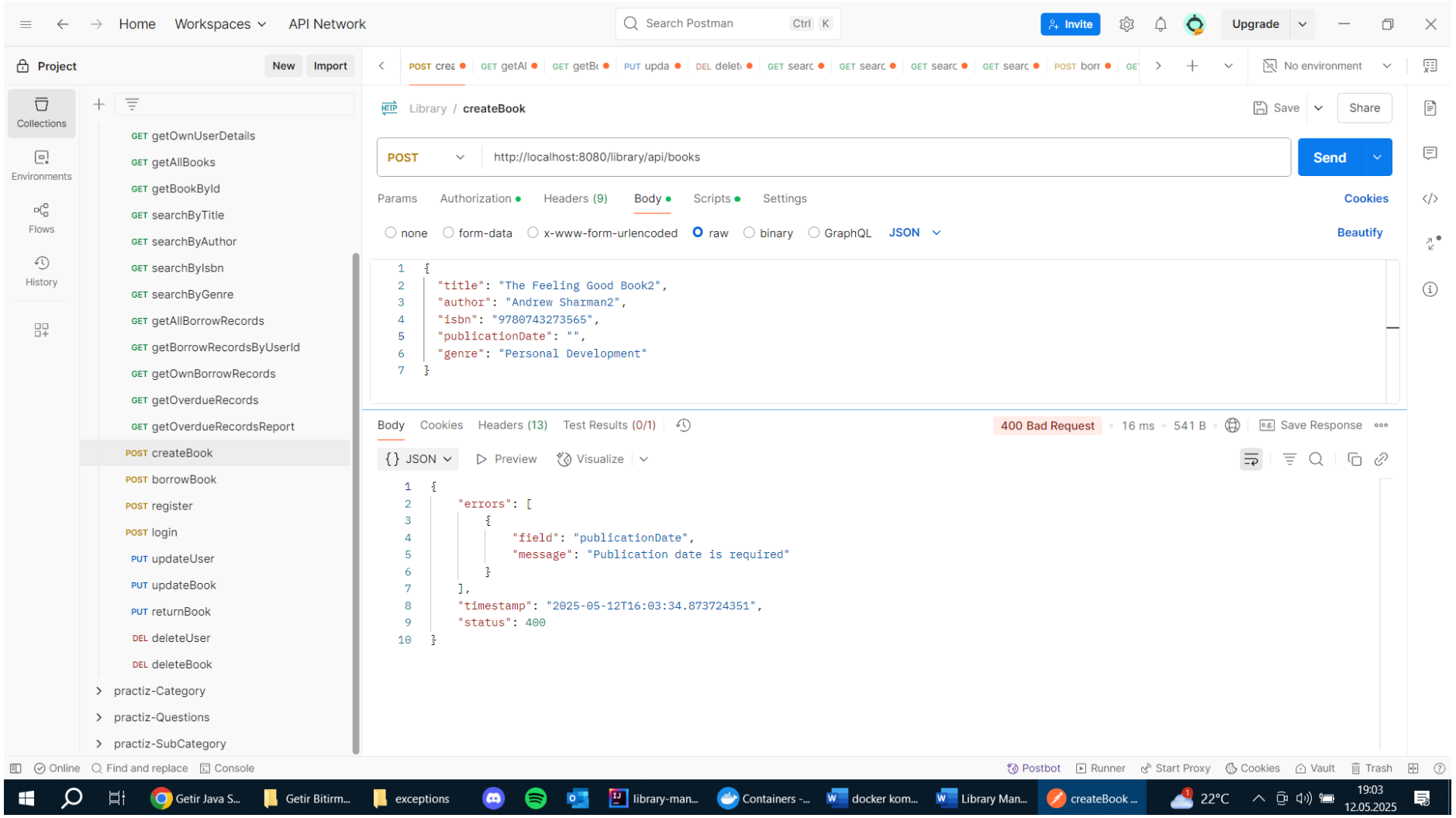
POST /books – Missing author



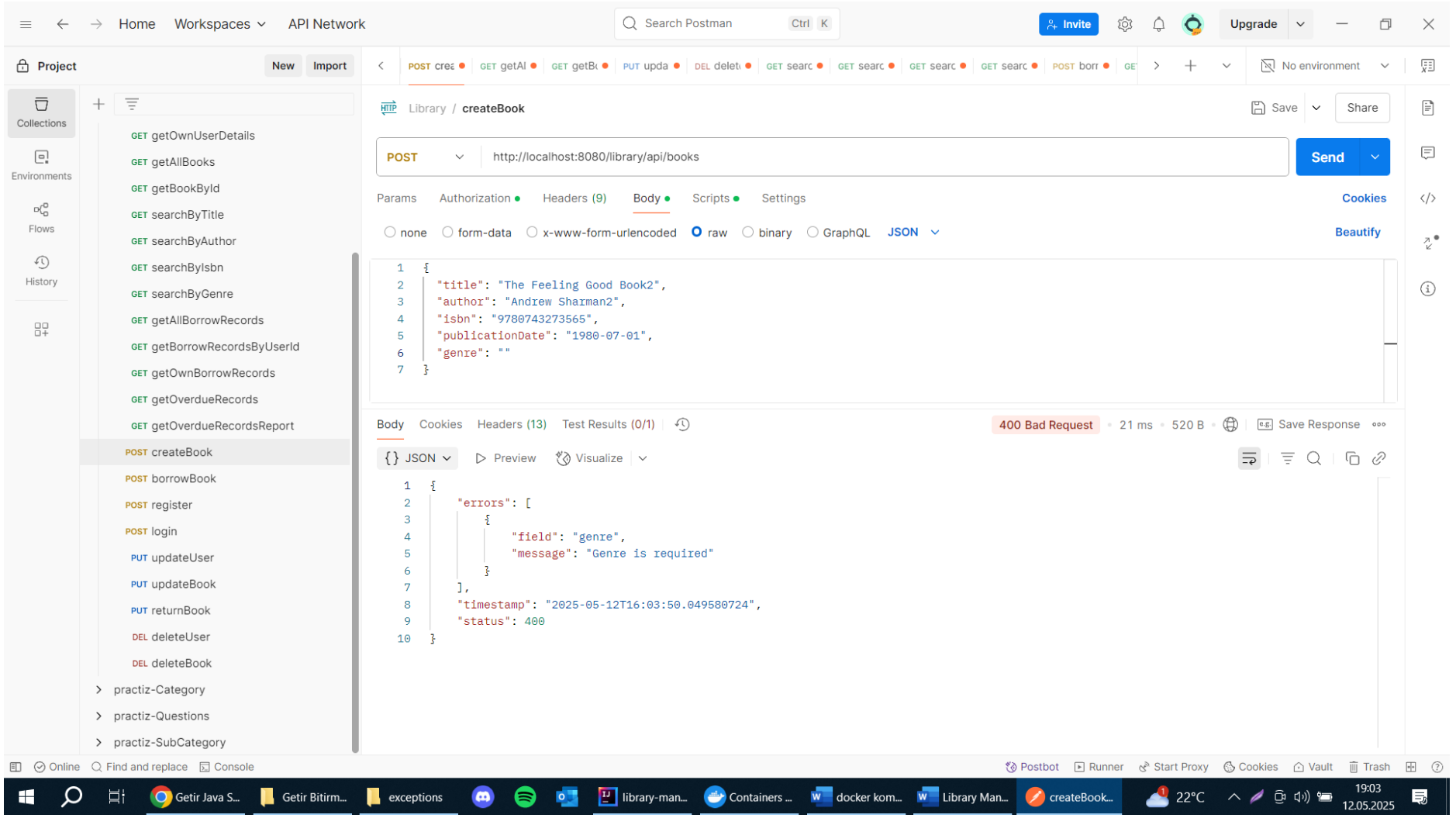
POST /books – Invalid ISBN format



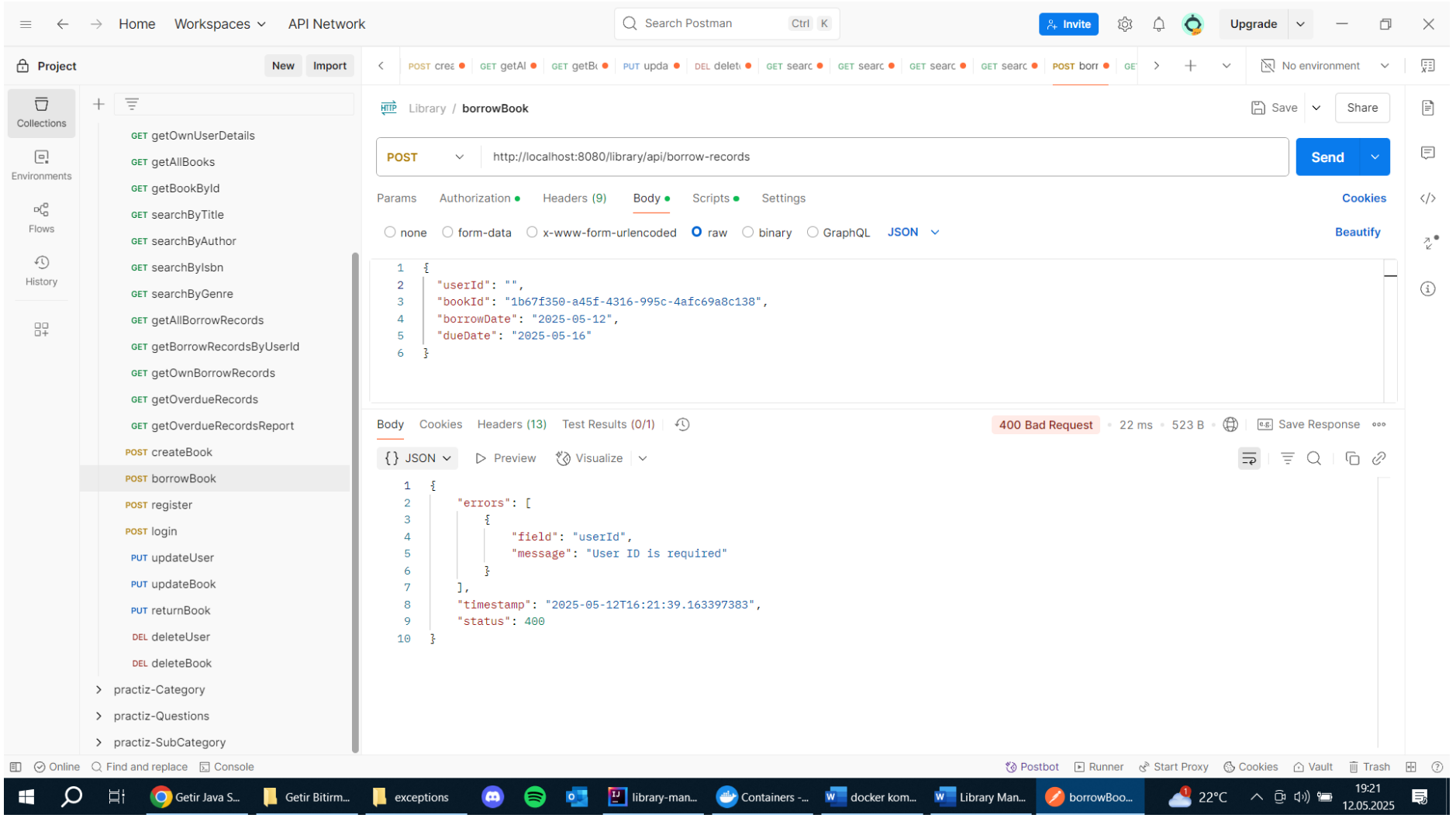
POST /books – Missing publication date



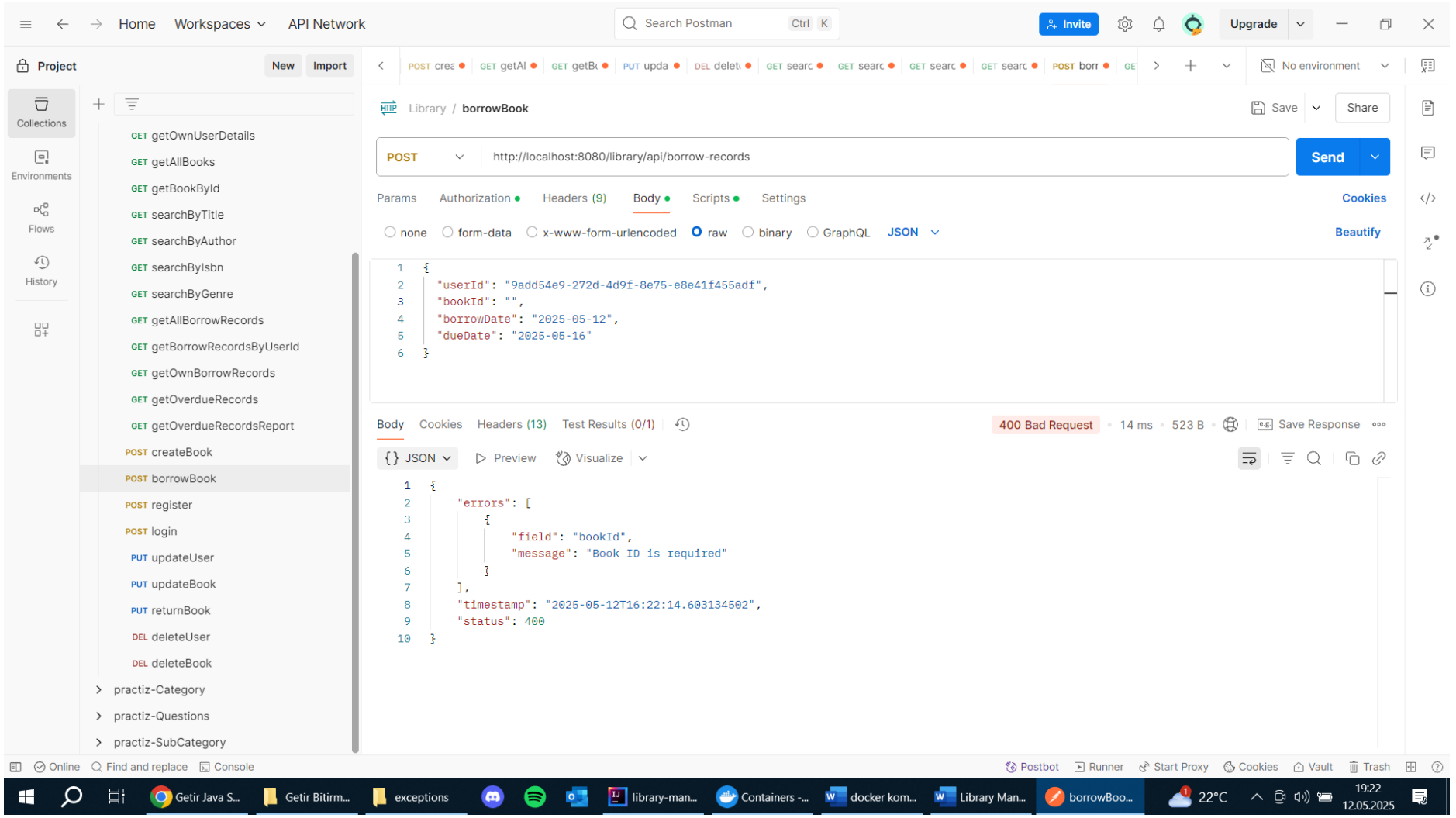
POST /books – Missing genre



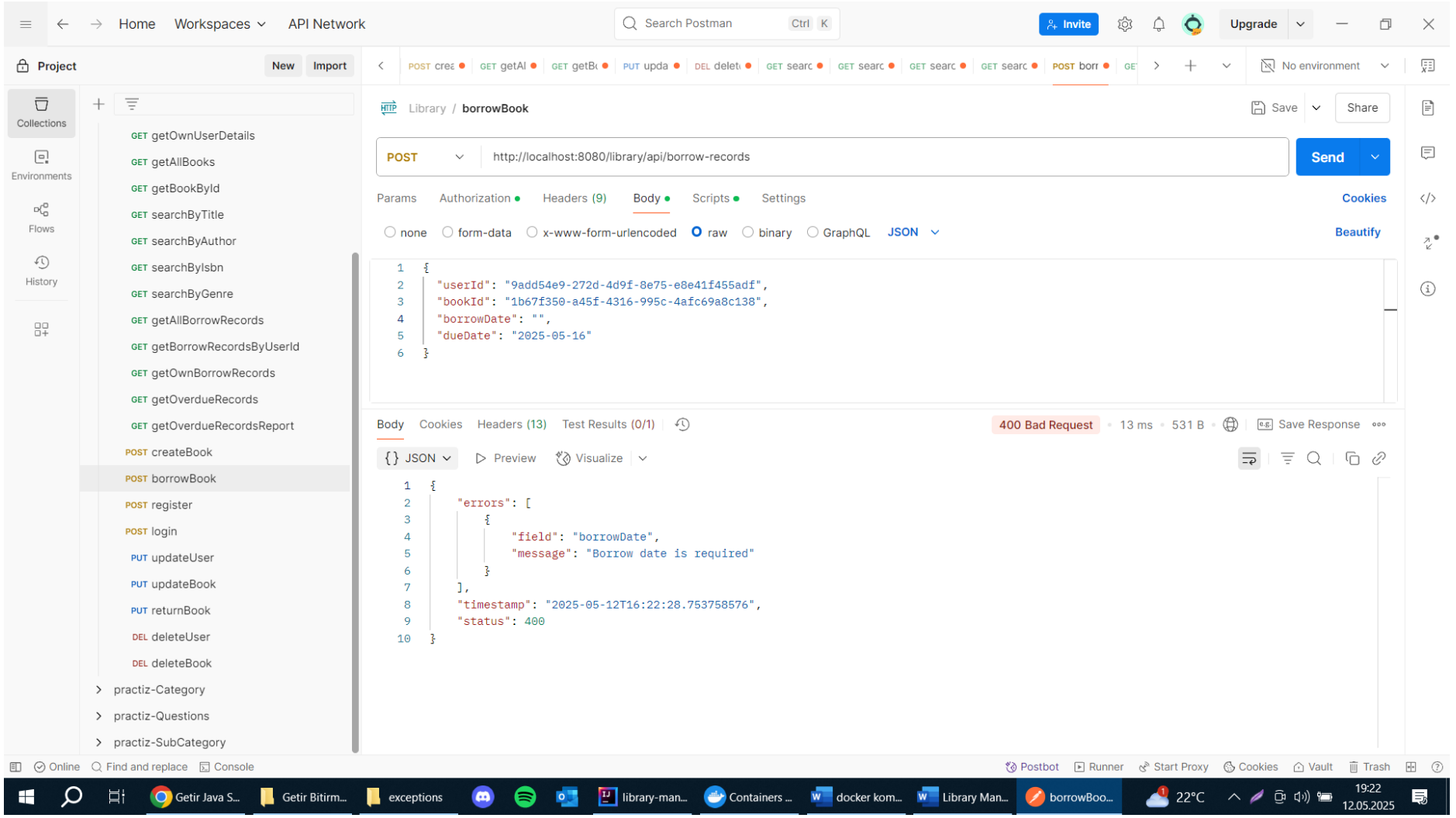
POST /borrow-records – Missing userId



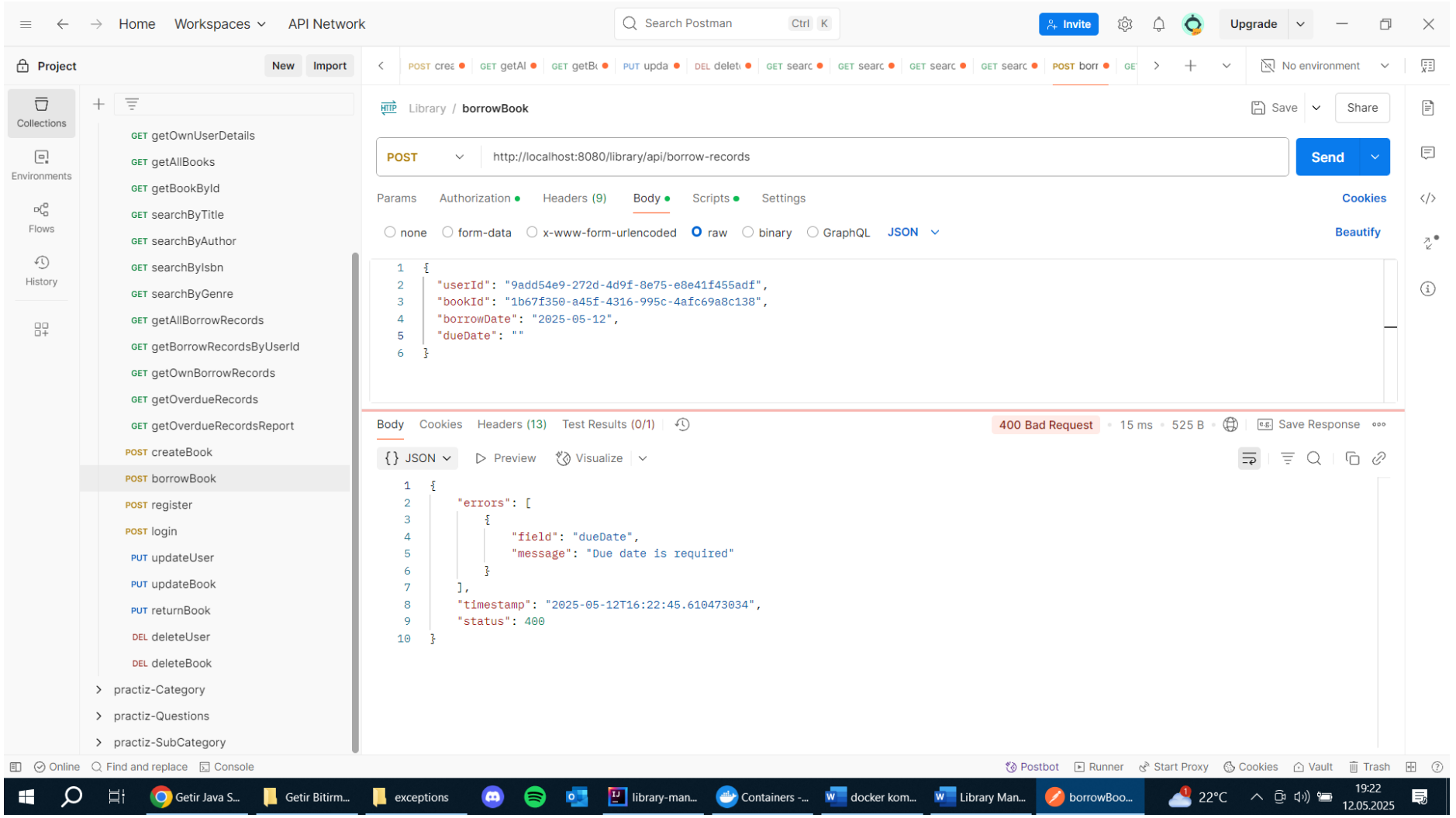
POST /borrow-records – Missing bookId



POST /borrow-records – Missing borrowDate



POST /borrow-records – Missing dueDate



POST /borrow-records – Due date not in future