Jafar Beldar

Week 3 Assignment

GITHUB LINK

1. Explain the Entity relationship with an example

Entity relationships define how different entities (tables) in a database are related to each other. In a Library Management System, for example:
- A 'Book' can be borrowed multiple times, so one Book has a One-to-Many relationship with 'BorrowRecord'.
- A 'Member' can borrow many books, so one Member has a One-to-Many relationship with 'BorrowRecord'.
These relationships help organize data and ensure integrity between related records.

2. Explain the Dependency Injection mechanism in NestJS.

Dependency Injection (DI) in NestJS is a design pattern where classes receive their dependencies from external sources rather than creating them.
This is done using the @Injectable() decorator and constructor injection. For example, a service can be injected into a controller:

```
class BookService {}
class BookController {
  constructor(private bookService: BookService) {}
}
```

This makes code easier to manage and test since dependencies are controlled by the framework.

3. How does NestJS DI resolve circular dependencies between services?

NestJS uses the forwardRef() function to resolve circular dependencies. This tells NestJS to resolve the reference later.
For example, if Service A uses Service B, and Service B also uses Service A:

```
constructor(@Inject(forwardRef(() => ServiceB)) private serviceB: ServiceB) {}
```

This approach prevents infinite loops during injection and allows mutual access between services safely.

4. What is the purpose of a Module in NestJS?

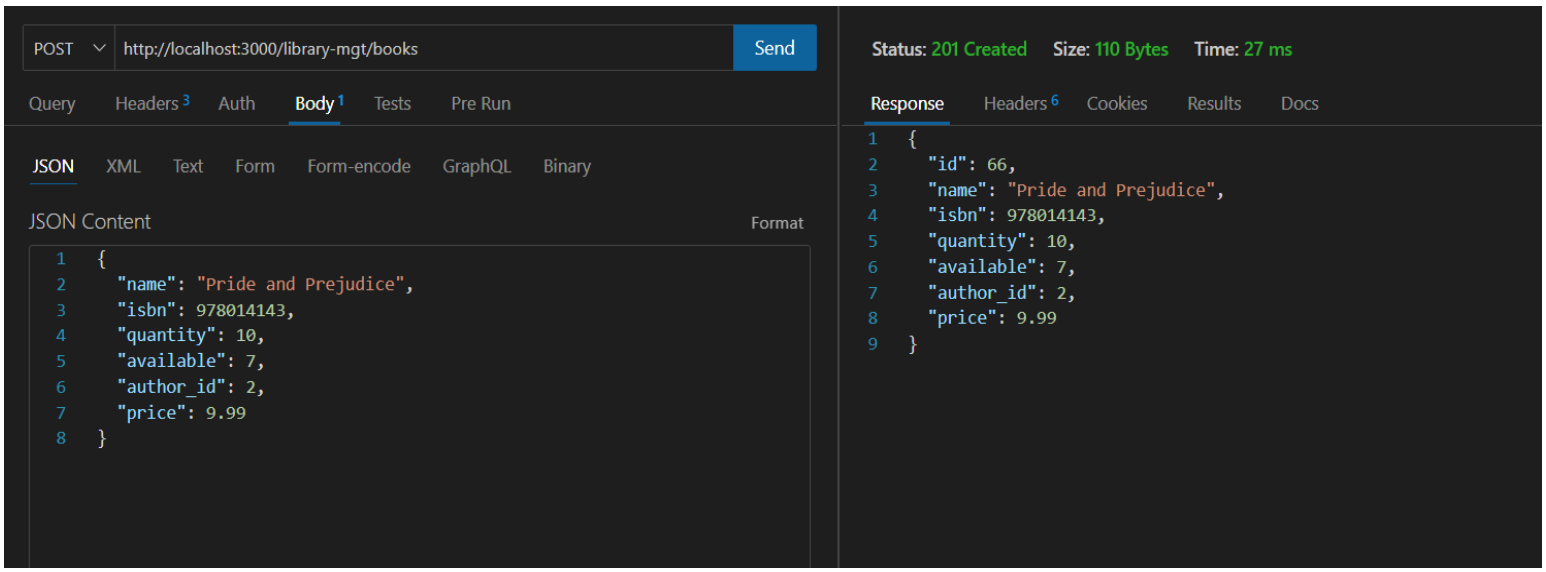
A Module in NestJS is used to organize related components (controllers, services, etc.) into a single unit. It helps group functionalities, making the app scalable and easier to maintain. Each module is defined using the @Module() decorator.
For example, a 'LibraryModule' could include controllers and services related to books and members.

5. Why would you use a Custom Validation Pipe instead of class-validator decorators alone?

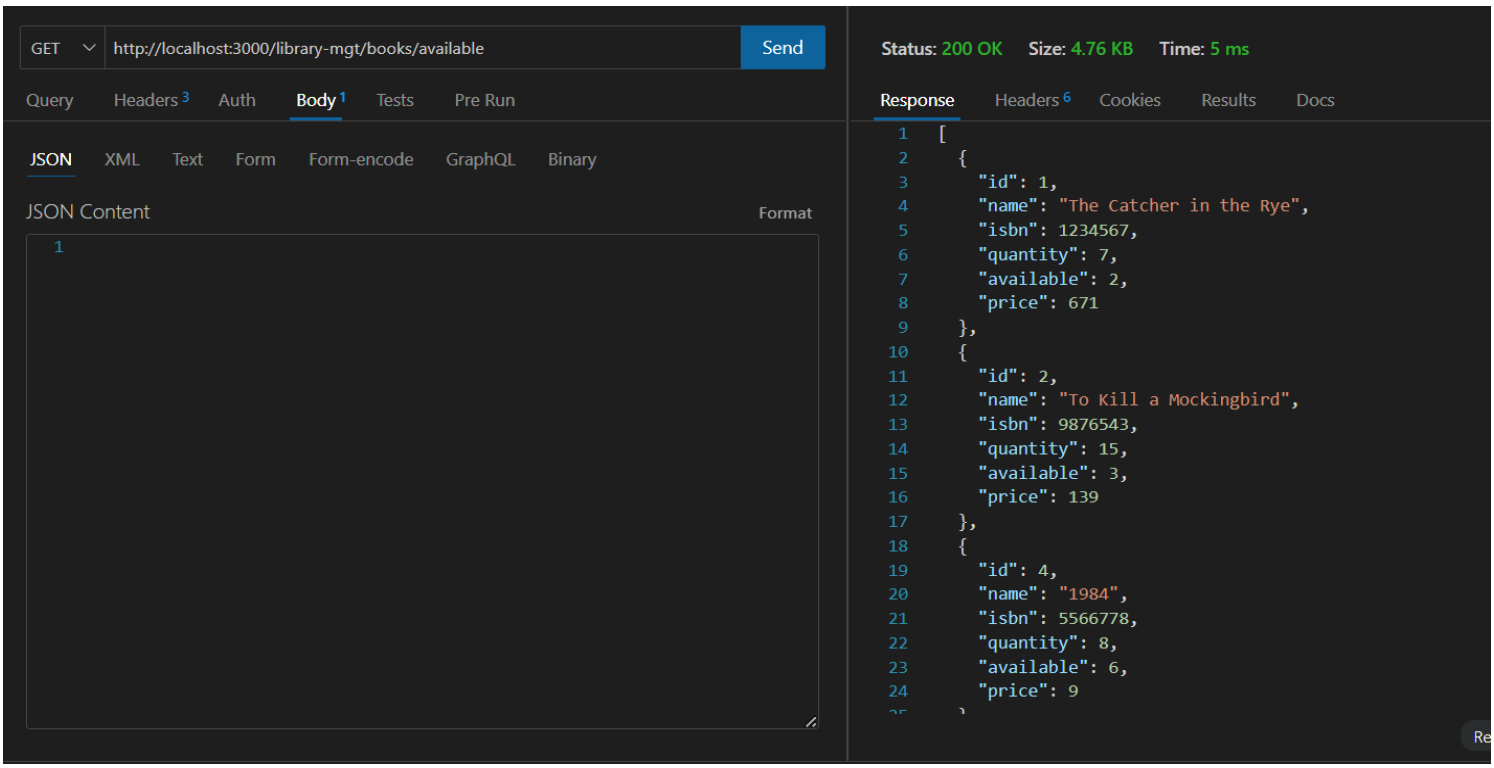
Class-validator decorators are great for simple validation like checking email or string length.
However, Custom Validation Pipes are used when you need more advanced logic, like checking if a book's quantity is greater than 0 before borrowing.
Custom pipes can use services and throw custom exceptions, making them powerful for complex validations.

Project 1: Library Management System –

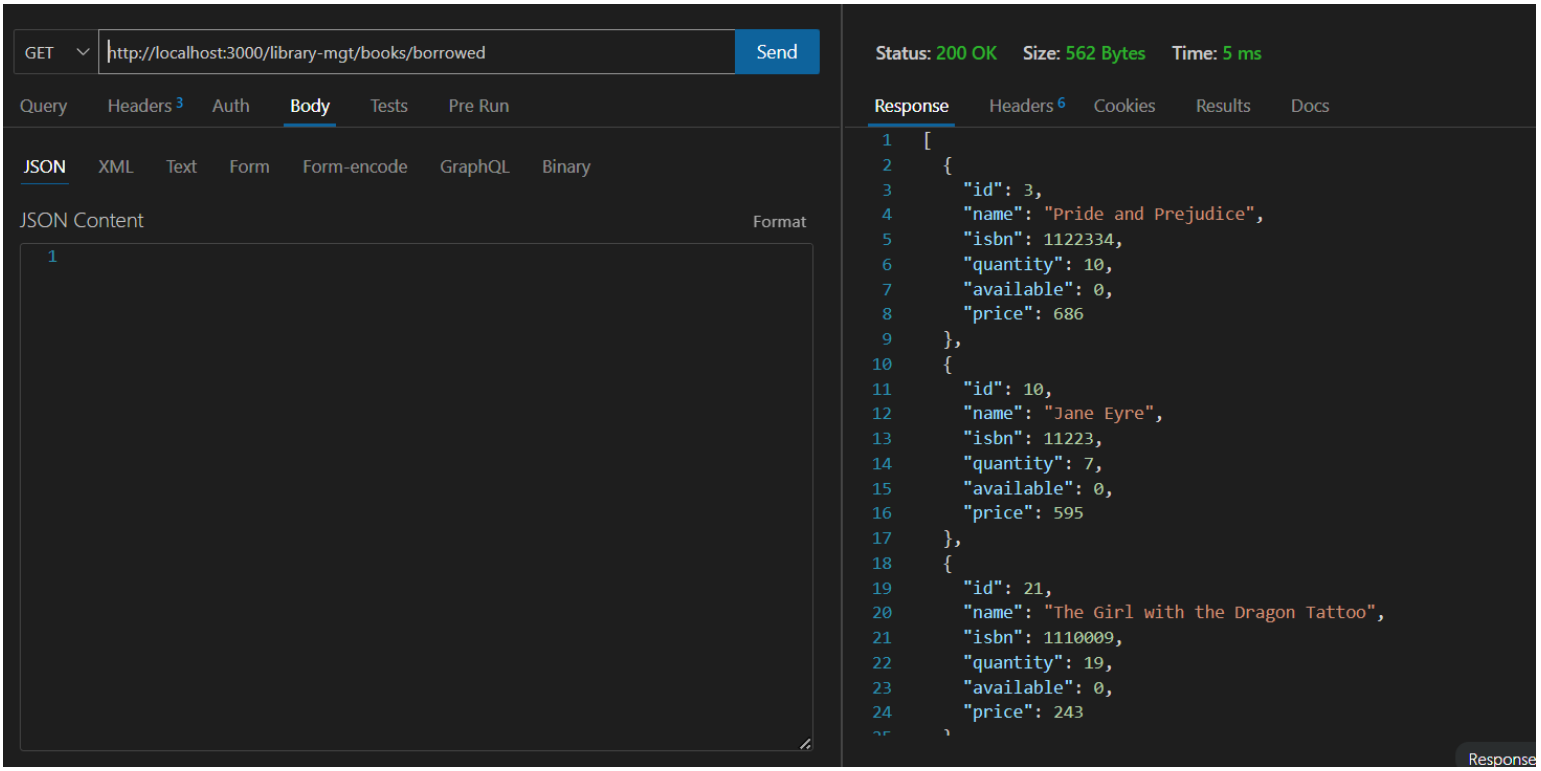
POST /books:



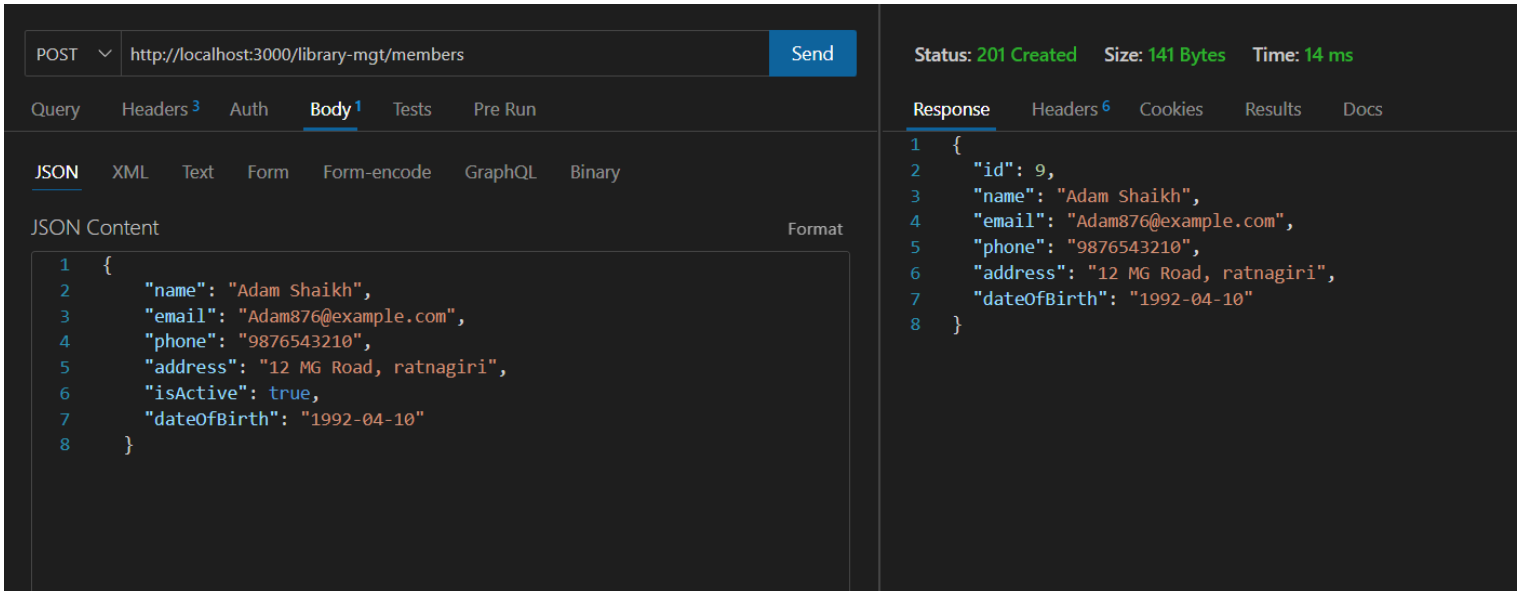
GET /books/available # List available books :



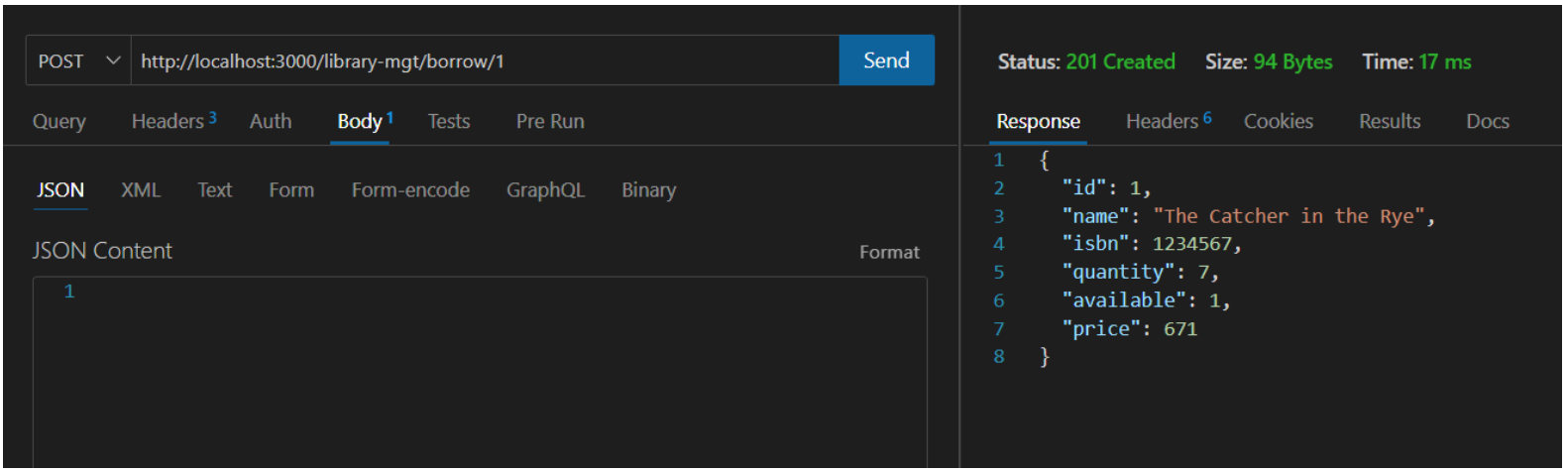
GET /books/borrowed # List borrowed books :



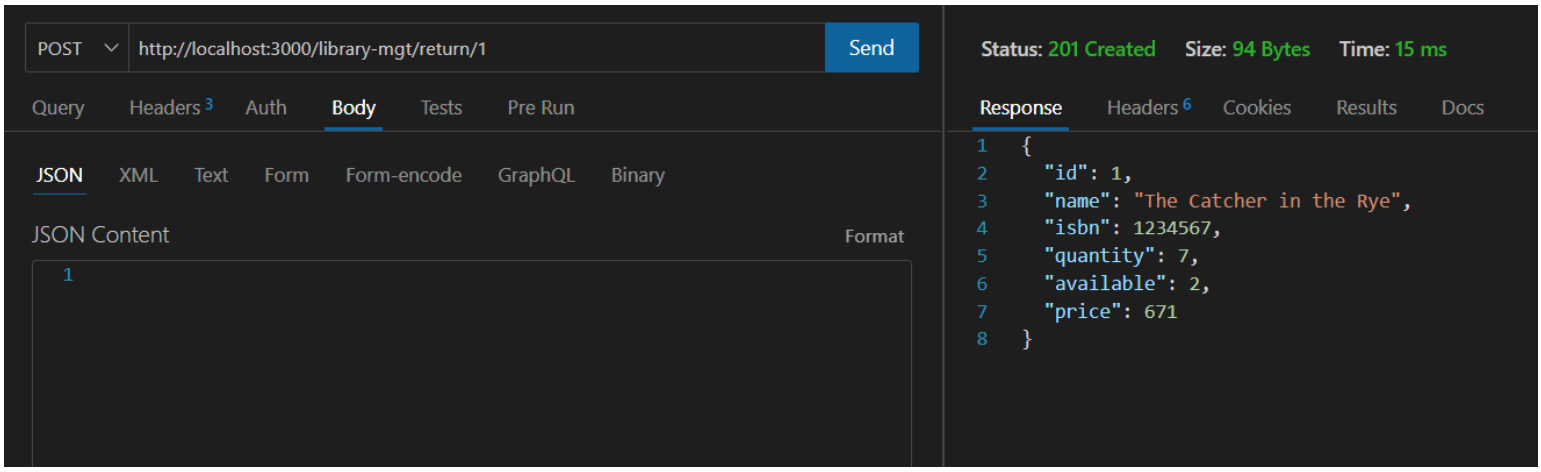
POST /members # Register member



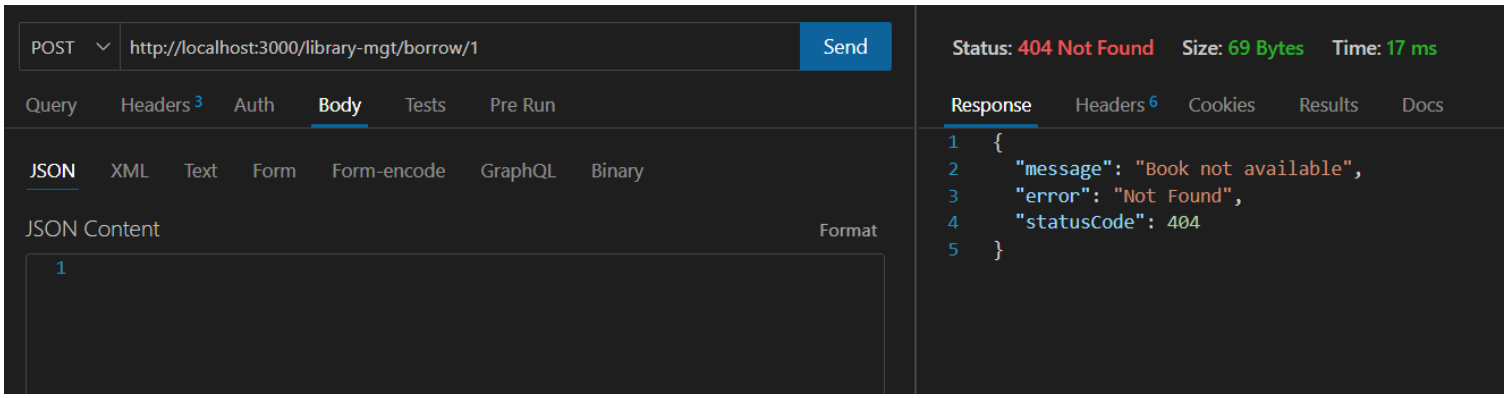
POST /borrow # Borrow a book (validate stock)



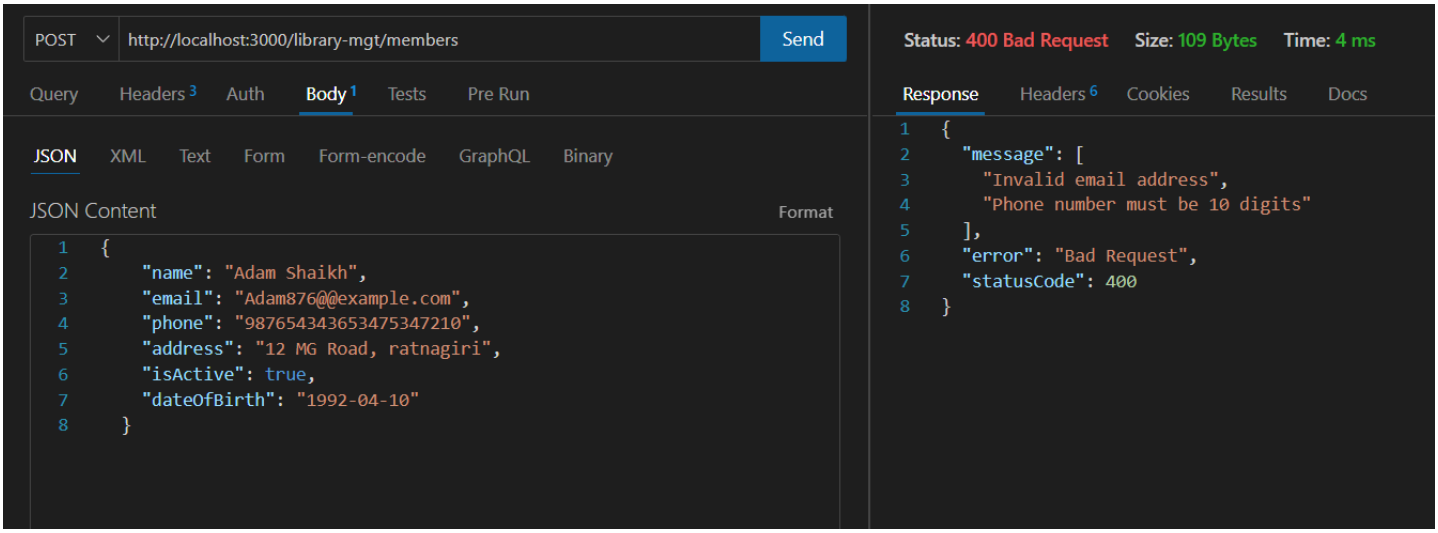
POST /return/:id # Return a book : note that the available value is increased when book is returned.



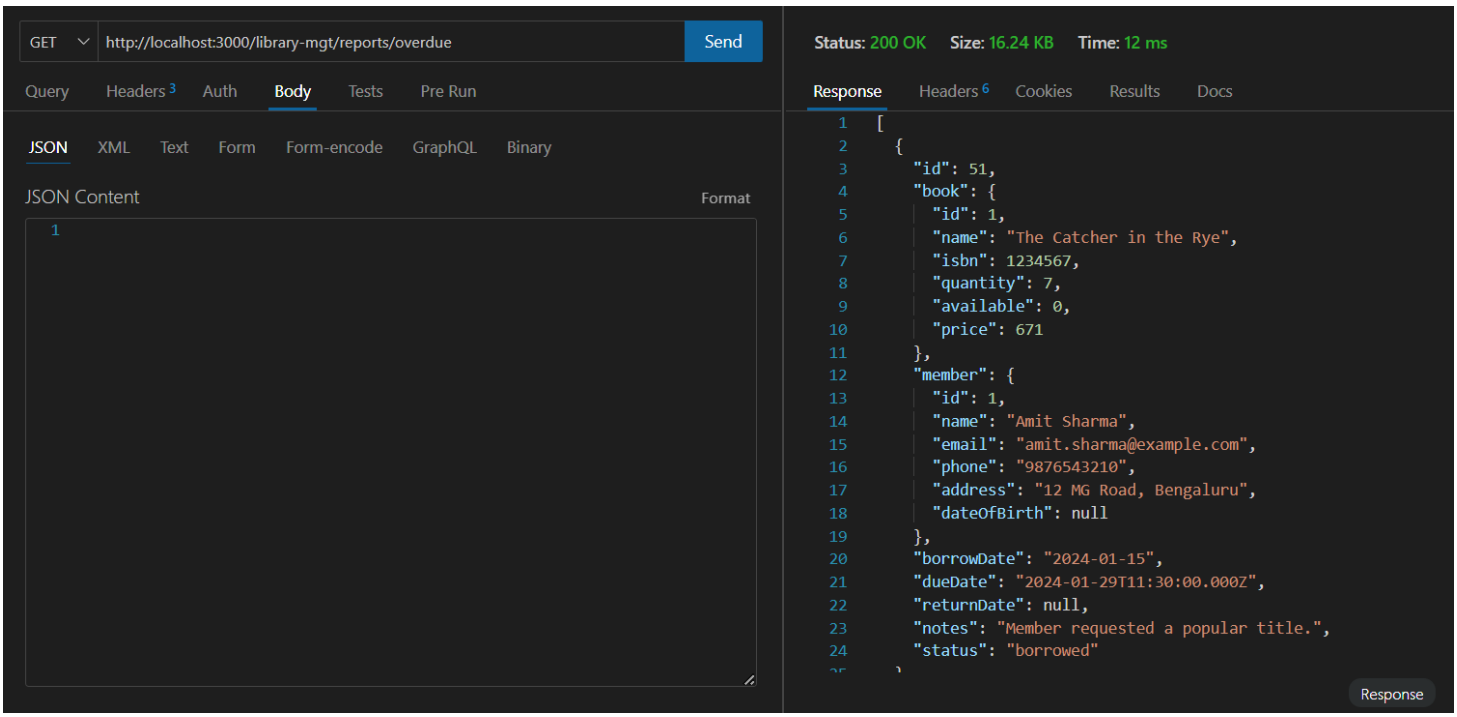
Custom pipe to check quantity >= 0 & Exception filter for "Book not available"



Class-validator for email/phone in Member

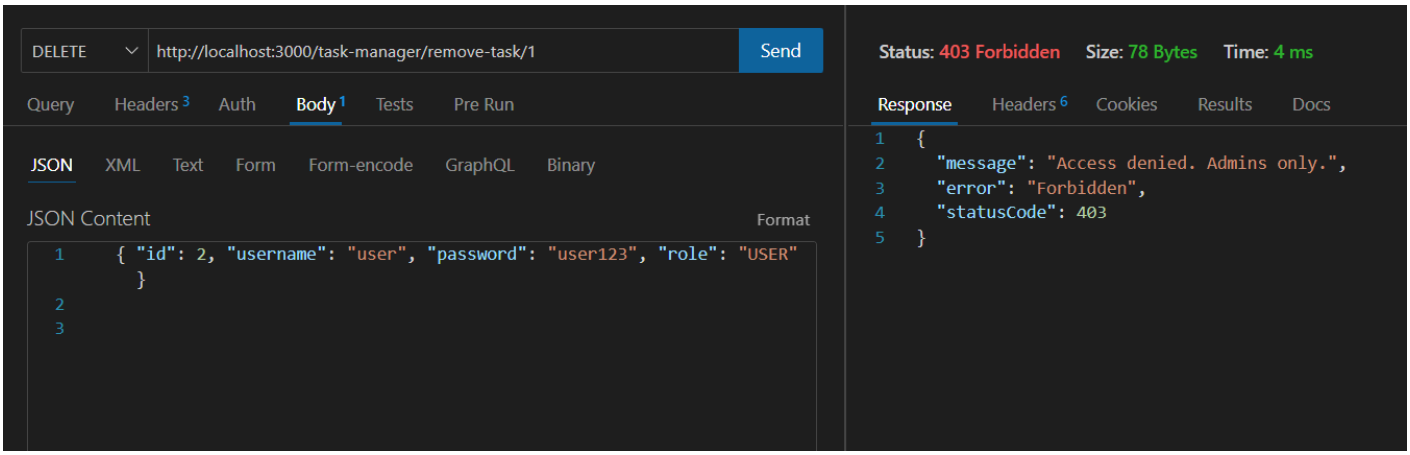


Overdue books report (GET /reports/overdue)

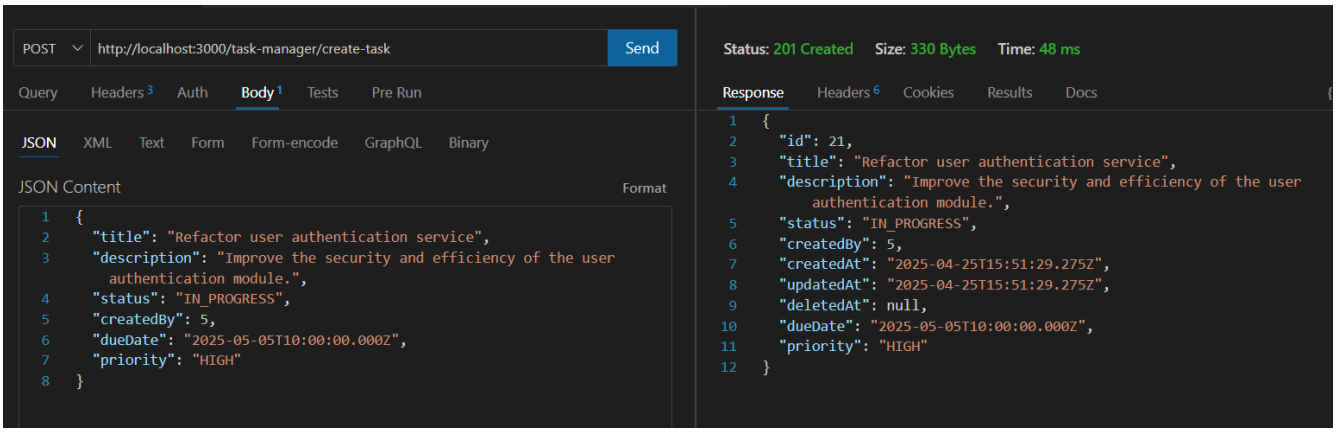


Project 2: Task Manager with RBAC

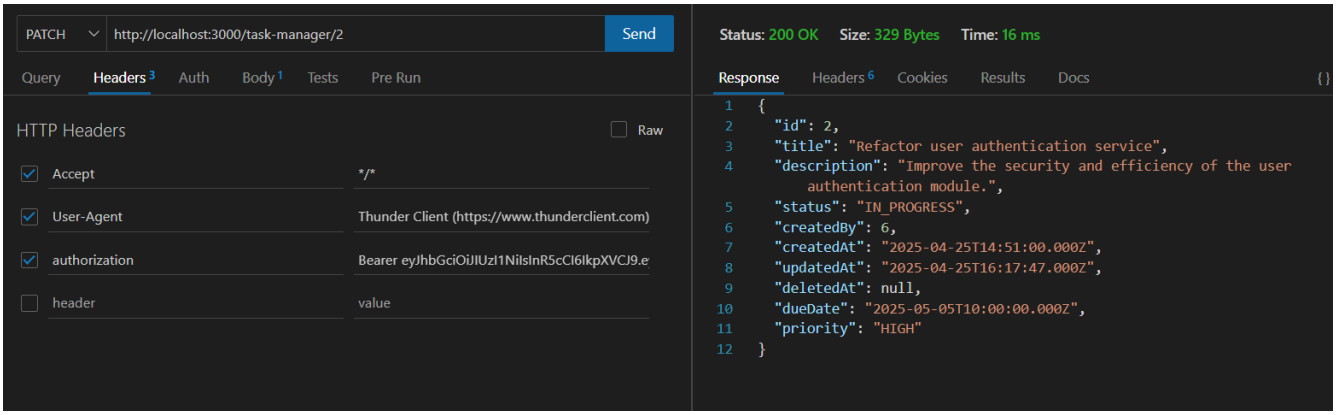
Static Role-Based Access :

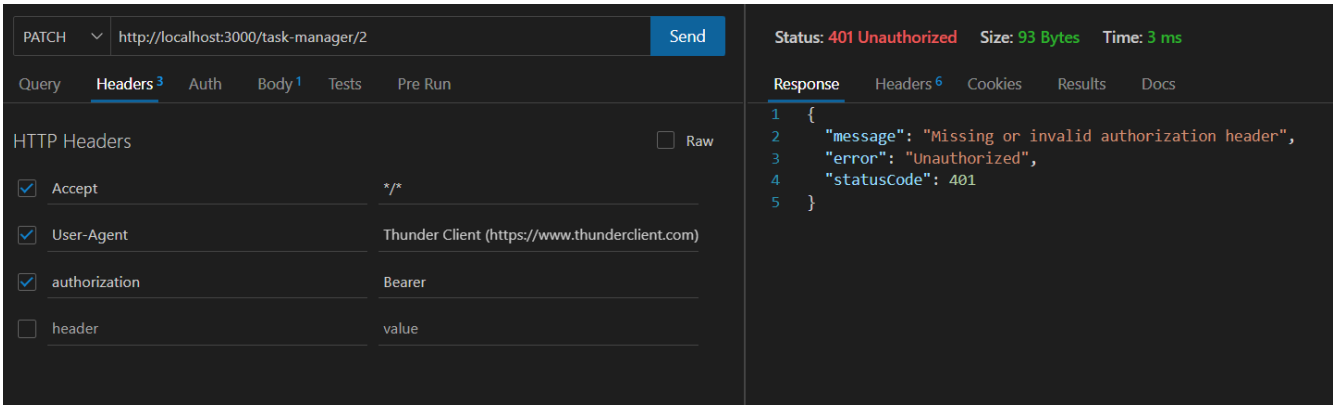


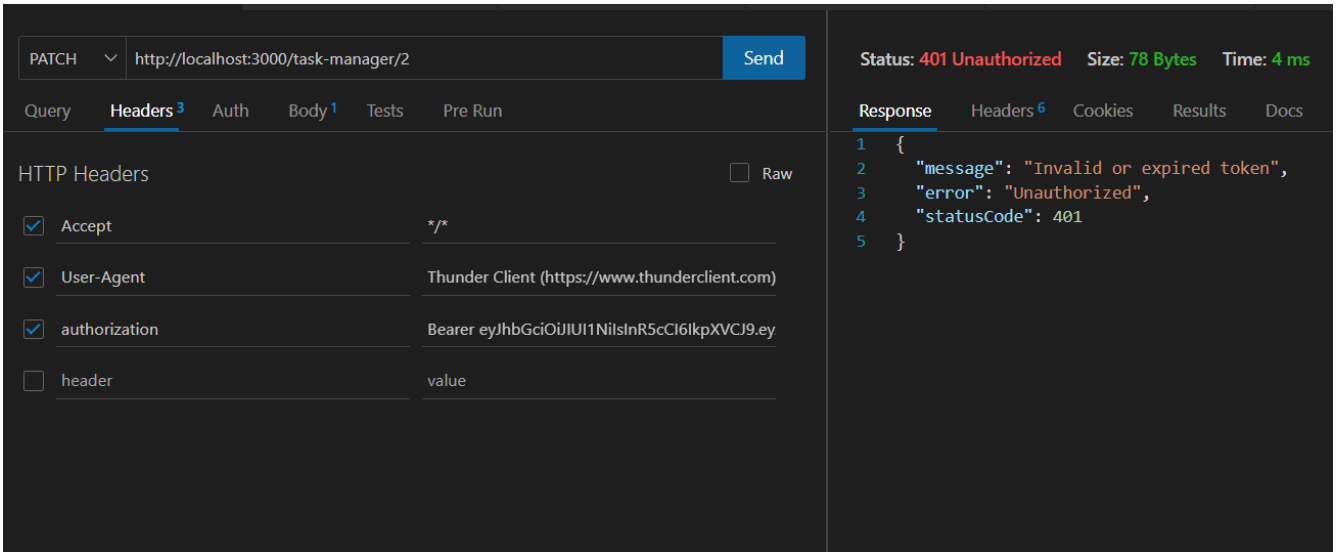
Create Task :



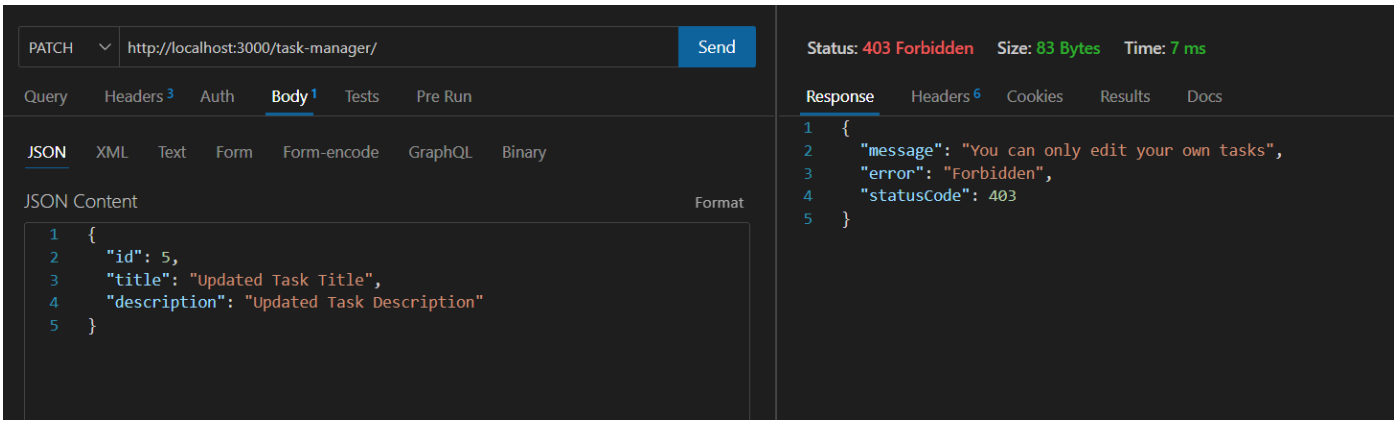
editable only by current user:



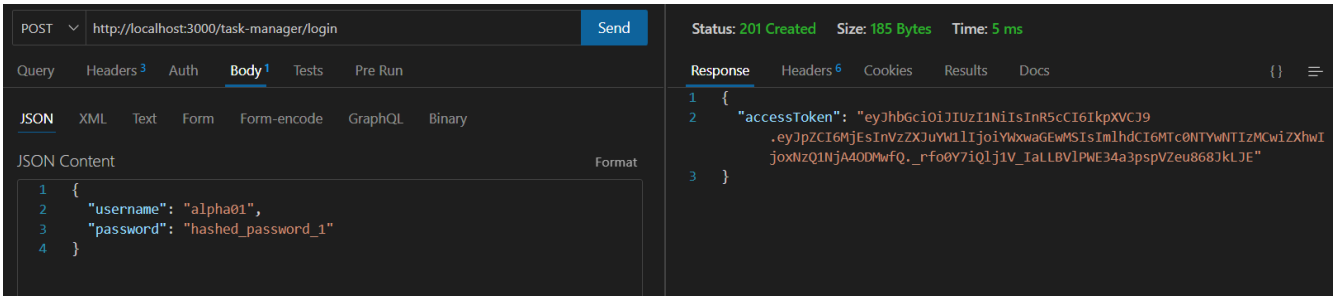




You can only edit your own tasks:



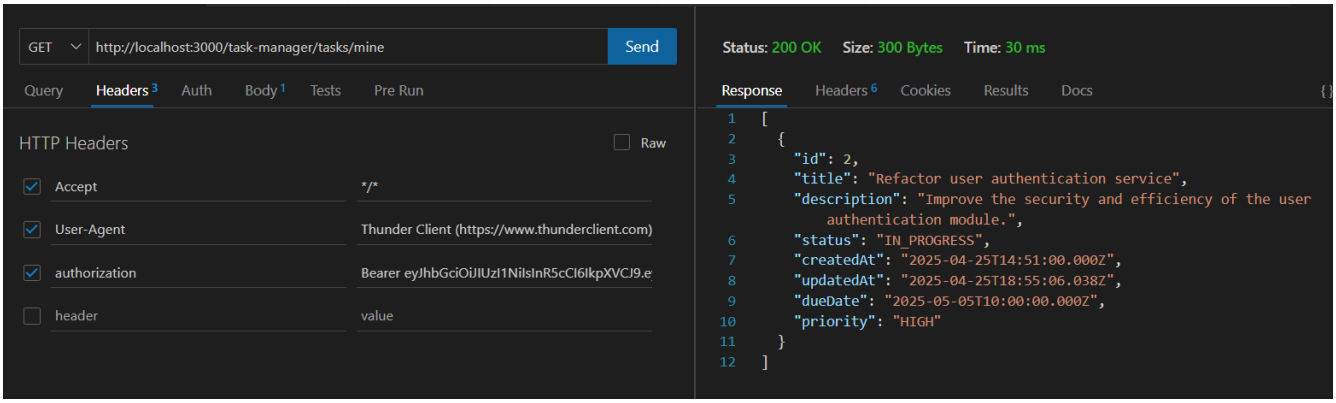
Login:



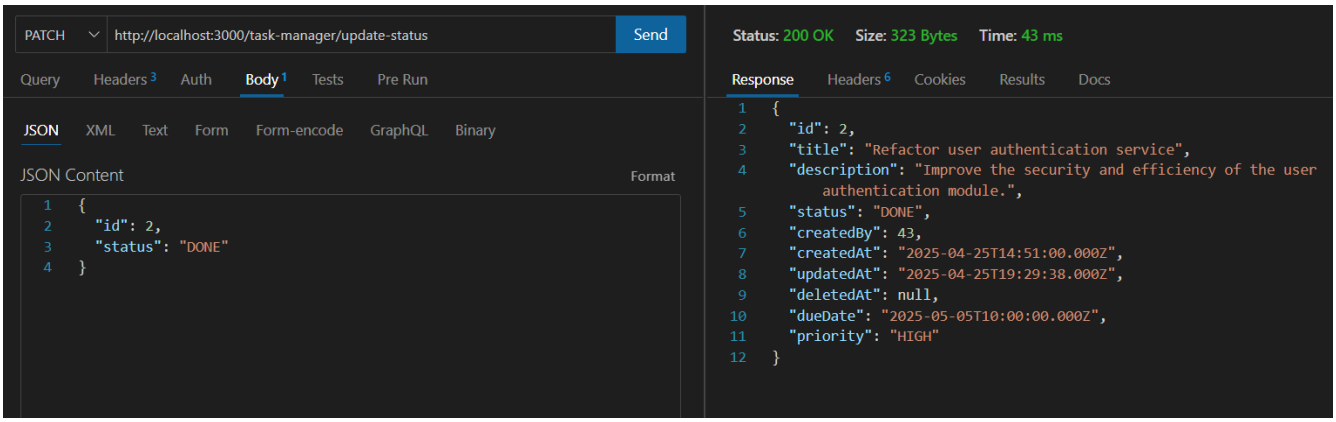
● GET /tasks/mine (Returns tasks by current user)

for user with userId:43

extracted from jwt token



Status Validation Pipe:



Request logger (log method, URL, timestamp)

Task Search: