

Baku Engineering University
INHA University

ECOMMERCE

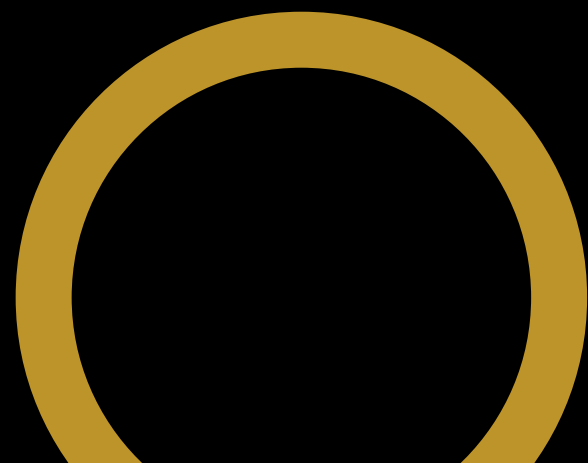
Presentation by

Said Gulizada

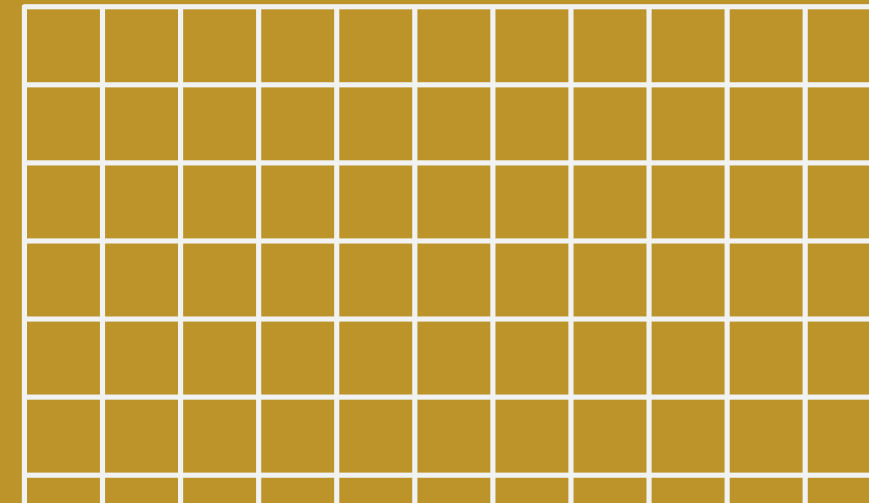
Teacher

Yusif Yusifov

Information Technologies
(INHA 3)



Overview



- Introduction
- Database Connection
- CRUD operations
- Logging
- Error Handling
- Structured Response
- Using Custom Address
- Using methods: get, post, put, delete, patch
- Swagger
- Security
- Git commit
- Conclusion

Introduction

I am delighted to present to you today our exciting e-commerce project. In this project, our goal is to redefine the online shopping experience from the ground up, providing users with a more robust, fast, and user-friendly platform.

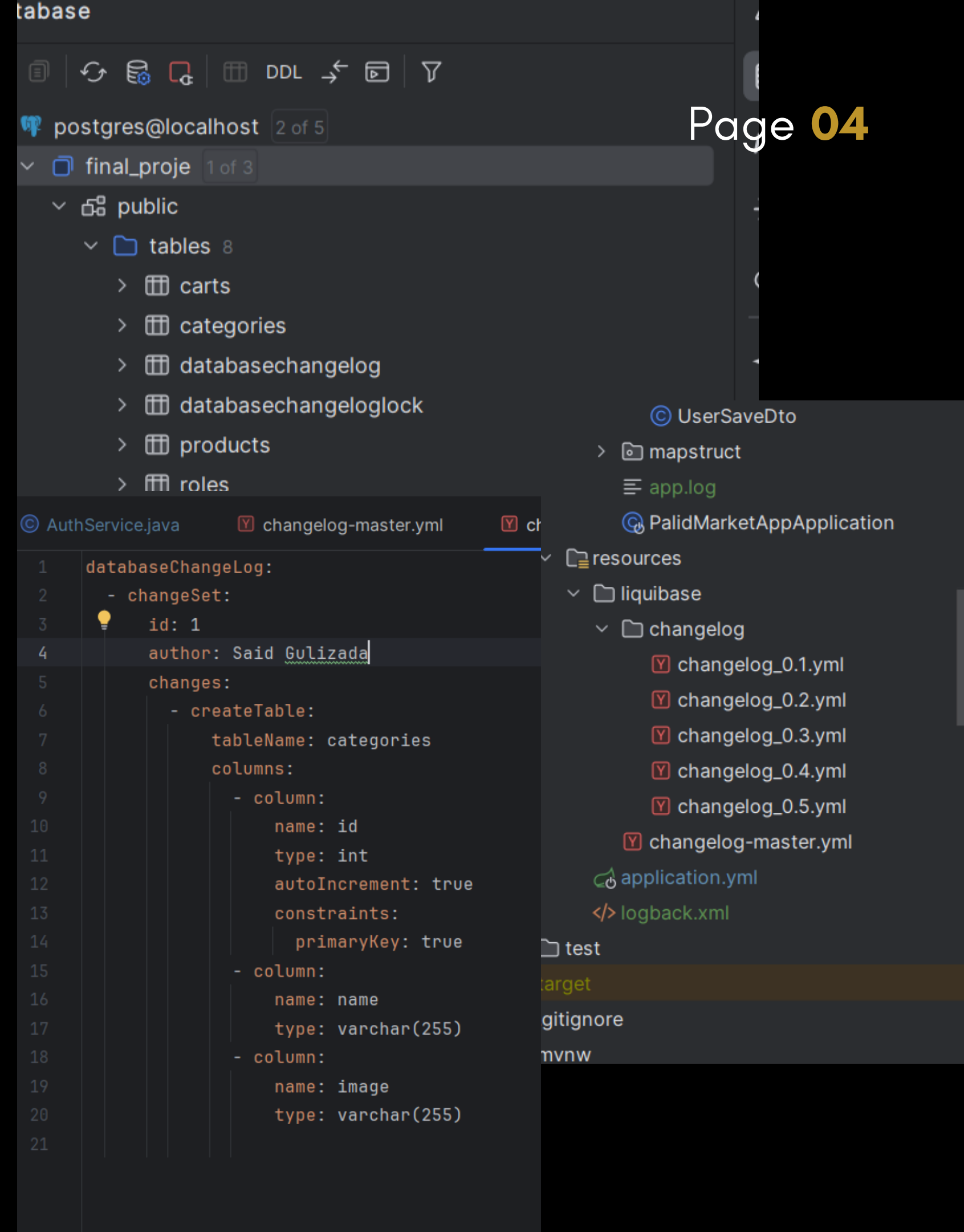
Now, I will provide a brief demonstration of our project. Please watch attentively, and afterward, we will delve into more in-depth details about our project.

Database connection

I opted for PostgreSQL as my database of choice for several reasons. Firstly, it offers an easy-to-use interface, making it particularly user-friendly for personal projects. The simplicity of interaction and the availability of a variety of tools contribute to a smoother development experience.

Additionally, I integrated Liquibase into the system to enhance the security and reliability of the database. Liquibase provides a robust version control system for database changes. Each modification made to the database is logged with user details, allowing for a transparent view of who made what changes and when. This not only ensures data integrity but also provides an added layer of accountability in case of any issues or updates.

In summary, the combination of PostgreSQL and Liquibase not only provides a user-friendly environment for personal projects but also ensures the security and traceability of every database change.



CRUD Operations

In my project, I implemented all CRUD operations, following the advice given by our instructor. While some developers might opt for using only GET and POST, I took a more comprehensive approach. By incorporating PUT and PATCH instead of creating everytime POST endpoints, I aimed to ensure that each operation is used appropriately in its intended context.

This strategy not only aligns with recommended best practices but also contributes to a more complete and maintainable project. It reflects a commitment to utilizing each HTTP method in its native form, adhering to the principles of RESTful API design.

user-controller

PUT /api/users/user/update

PATCH /api/users/user/{phoneNumber}/{lastName}

PATCH /api/users/user/{phoneNumber}/{firstName}

GET /api/users/user/{phoneNumber}

DELETE /api/users/user/{phoneNumber}

GET /api/users/admin/getAll

auth-controller

POST /v1/auth/register

POST /v1/auth/login

product-controller

POST /api/products/admin/add

PATCH /api/products/admin/{productId}/{price}

GET /api/products/user/{getByCategoryId}

GET /api/products/admin/getAll

DELETE /api/products/admin/{id}

GET /api/products/user/{getByCategoryId}

GET /api/products/admin/getAll

DELETE /api/products/admin/{id}

category-controller

POST /api/categories/admin/save

PATCH /api/categories/admin/{categoryId}/{name}

PATCH /api/categories/admin/{categoryId}/{img}

GET /api/categories/user/getAll

DELETE /api/categories/admin/{id}

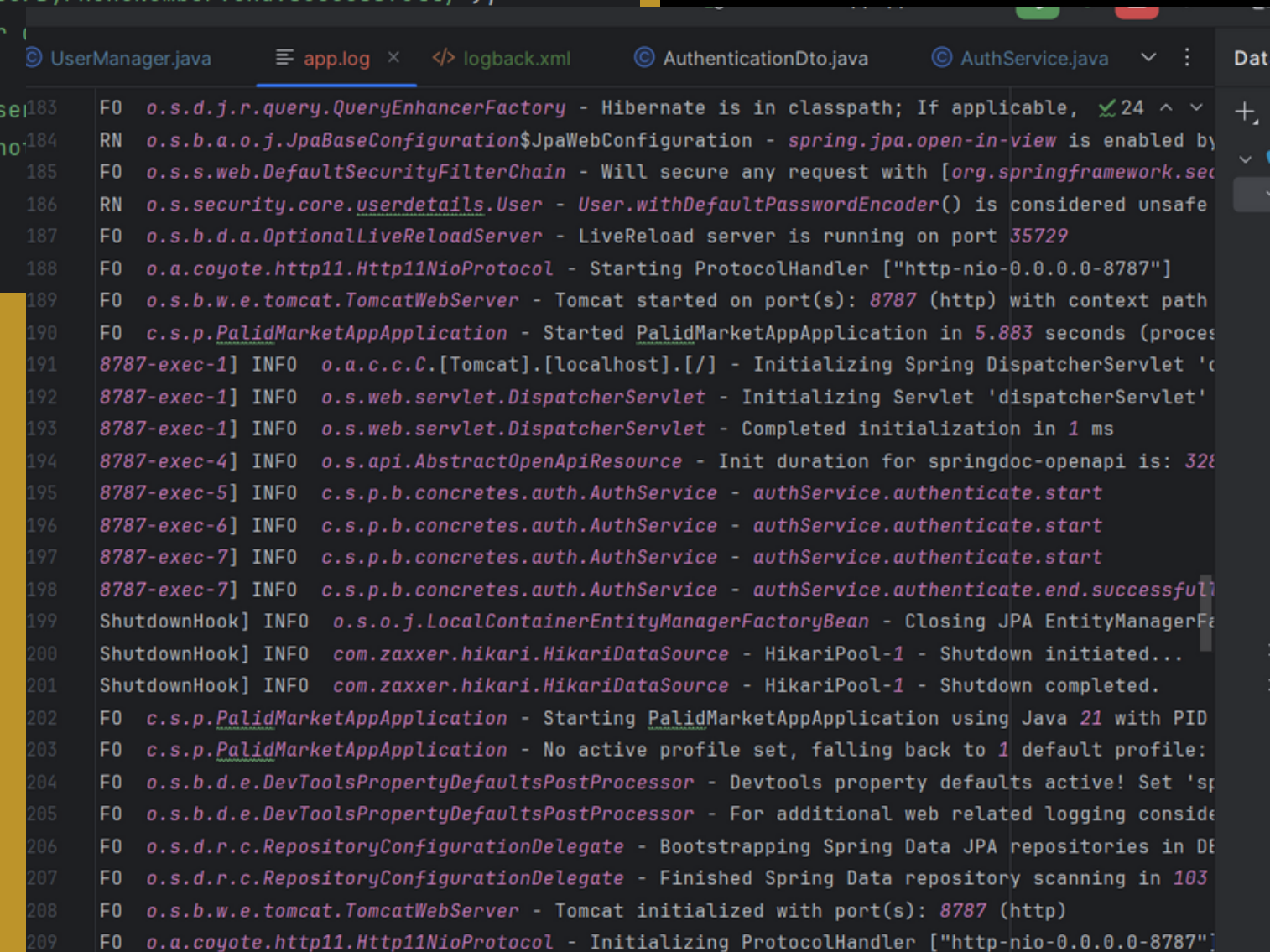
Logging

In my project, I have implemented logging within the methods of service classes to provide a detailed record of the execution flow. For every method, logs are incorporated at the beginning and end, specifying whether the operation was successful or unsuccessful. Moreover, these logs are not only directed to the console but are also stored in an external file. This dual-logging approach ensures that a comprehensive record of the project's activities is maintained. Storing logs in an external file facilitates better tracking and analysis, allowing for a clearer understanding of when and what events occurred in the project.

This logging strategy not only aids in debugging and error identification but also contributes to a more informed and transparent project development process.

```
1 usage  👤 said4shi
@Override
public DataResult<List<UserDto>> getAll() {
    log.info("application.getAllUser.start");
    List <User> users = userDao.findAll();
    log.info("application.getAllUser.end");
    return new SuccessDataResult<>(users.stream().map(user -> modelMapper.map(user,
}

1 usage  👤 said4shi
@Transactional
public Result deleteUserByPhoneNumber(String phoneNumber) {
    User user = userDao.findUserByPhoneNumber(phoneNumber);
    Role role = (Role) user.getRoles();
    log.info("application.deleteUserByPhoneNumber.start");
    if (userDao.existsByPhoneNumber(phoneNumber)) {
        roleDao.delete(role);
        userDao.deleteByPhoneNumber(phoneNumber);
        log.info("application.deleteUserByPhoneNumber.end.successfully");
        return new SuccessResult("User deleted successfully");
    } else {
        log.info("application.deleteUserByPhoneNumber.end.failed");
        return new ErrorResult("User not found");
    }
}
```



The screenshot shows an IDE with a logback.xml file open. The file contains the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.core.filter.DefaultFilter">
            <value>INFO</value>
        </filter>
        <encoder class="ch.qos.logback.core.encoder.DefaultEncoder"/>
    </appender>

    <root>
        <level>INFO</level>
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

The console window displays the following logs:

```
2025-01-01 10:00:00.000 INFO o.s.d.j.r.query.QueryEnhancerFactory - Hibernate is in classpath; If applicable, 24 ^ v
2025-01-01 10:00:00.000 INFO o.s.b.a.o.j.JpaBaseConfiguration$JpaWebConfiguration - spring.jpa.open-in-view is enabled by
2025-01-01 10:00:00.000 INFO o.s.s.web.DefaultSecurityFilterChain - Will secure any request with [org.springframework.se
2025-01-01 10:00:00.000 INFO o.s.security.core.userdetails.User - User.withDefaultPasswordEncoder() is considered unsafe
2025-01-01 10:00:00.000 INFO o.s.b.d.a.OptionalLiveReloadServer - LiveReload server is running on port 35729
2025-01-01 10:00:00.000 INFO o.a.coyote.http11.Http11NioProtocol - Starting ProtocolHandler ["http-nio-0.0.0.0-8787"]
2025-01-01 10:00:00.000 INFO o.s.b.w.e.tomcat.TomcatWebServer - Tomcat started on port(s): 8787 (http) with context path
2025-01-01 10:00:00.000 INFO c.s.p.PalidMarketAppApplication - Started PalidMarketAppApplication in 5.883 seconds (proces
2025-01-01 10:00:00.000 INFO 8787-exec-1] INFO o.a.c.c.C.[Tomcat].[localhost].[/] - Initializing Spring DispatcherServlet 'c
2025-01-01 10:00:00.000 INFO 8787-exec-1] INFO o.s.web.servlet.DispatcherServlet - Initializing Servlet 'dispatcherServlet'
2025-01-01 10:00:00.000 INFO 8787-exec-1] INFO o.s.web.servlet.DispatcherServlet - Completed initialization in 1 ms
2025-01-01 10:00:00.000 INFO 8787-exec-4] INFO o.s.api.AbstractOpenApiResource - Init duration for springdoc-openapi is: 328
2025-01-01 10:00:00.000 INFO 8787-exec-5] INFO c.s.p.b.concretes.auth.AuthService - authService.authenticate.start
2025-01-01 10:00:00.000 INFO 8787-exec-6] INFO c.s.p.b.concretes.auth.AuthService - authService.authenticate.start
2025-01-01 10:00:00.000 INFO 8787-exec-7] INFO c.s.p.b.concretes.auth.AuthService - authService.authenticate.start
2025-01-01 10:00:00.000 INFO 8787-exec-7] INFO c.s.p.b.concretes.auth.AuthService - authService.authenticate.end.successful
2025-01-01 10:00:00.000 INFO ShutdownHook] INFO o.s.o.j.LocalContainerEntityManagerFactoryBean - Closing JPA EntityManagerFa
2025-01-01 10:00:00.000 INFO ShutdownHook] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Shutdown initiated...
2025-01-01 10:00:00.000 INFO ShutdownHook] INFO com.zaxxer.hikari.HikariDataSource - HikariPool-1 - Shutdown completed.
2025-01-01 10:00:00.000 INFO c.s.p.PalidMarketAppApplication - Starting PalidMarketAppApplication using Java 21 with PID
2025-01-01 10:00:00.000 INFO c.s.p.PalidMarketAppApplication - No active profile set, falling back to 1 default profile: sp
2025-01-01 10:00:00.000 INFO o.s.b.d.e.DevToolsPropertyDefaultsPostProcessor - Devtools property defaults active! Set 'sp
2025-01-01 10:00:00.000 INFO o.s.b.d.e.DevToolsPropertyDefaultsPostProcessor - For additional web related logging conside
2025-01-01 10:00:00.000 INFO o.s.d.r.c.RepositoryConfigurationDelegate - Bootstrapping Spring Data JPA repositories in DE
2025-01-01 10:00:00.000 INFO o.s.d.r.c.RepositoryConfigurationDelegate - Finished Spring Data repository scanning in 103
2025-01-01 10:00:00.000 INFO o.s.b.w.e.tomcat.TomcatWebServer - Tomcat initialized with port(s): 8787 (http)
2025-01-01 10:00:00.000 INFO o.a.coyote.http11.Http11NioProtocol - Initializing ProtocolHandler ["http-nio-0.0.0.0-8787"]
```

Error Handling

In my project, I prioritize robust error handling to ensure that any potential vulnerabilities are identified and addressed promptly. To achieve this, I've implemented try-catch blocks within the API controllers.

When an error occurs, I don't rely on the lengthy default Java error messages. Instead, I return custom error messages that provide meaningful information about the error's location and nature. This approach aids in understanding how and where the error occurred, making the debugging process more efficient.

By customizing error messages, I aim to provide users and developers with clear and concise information about the issues they may encounter. This not only contributes to a better user experience but also facilitates a more streamlined debugging and issue resolution process.

```
said4shi
PatchMapping(🔗"/admin/{productId}/{price}")
public ResponseEntity<Result> updateName(@PathVariable int productId, @PathVaria
    try {
        Result result = productService.updatePrice(productId, price);
        if (result.isSuccess()) {
            return ResponseEntity.ok(result);
        } else {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(result);
        }
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null)
    }
}
```

```
said4shi
DeleteMapping(🔗"/admin/{id}")
public ResponseEntity<Result> deleteProduct(@PathVariable int id) {
    try {
        Result result = productService.deleteProduct(id);
        if (result.isSuccess()) {
            return ResponseEntity.ok(result);
        } else {
            return ResponseEntity.status(HttpStatus.NOT_FOUND).body(result);
        }
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(null)
    }
}
```

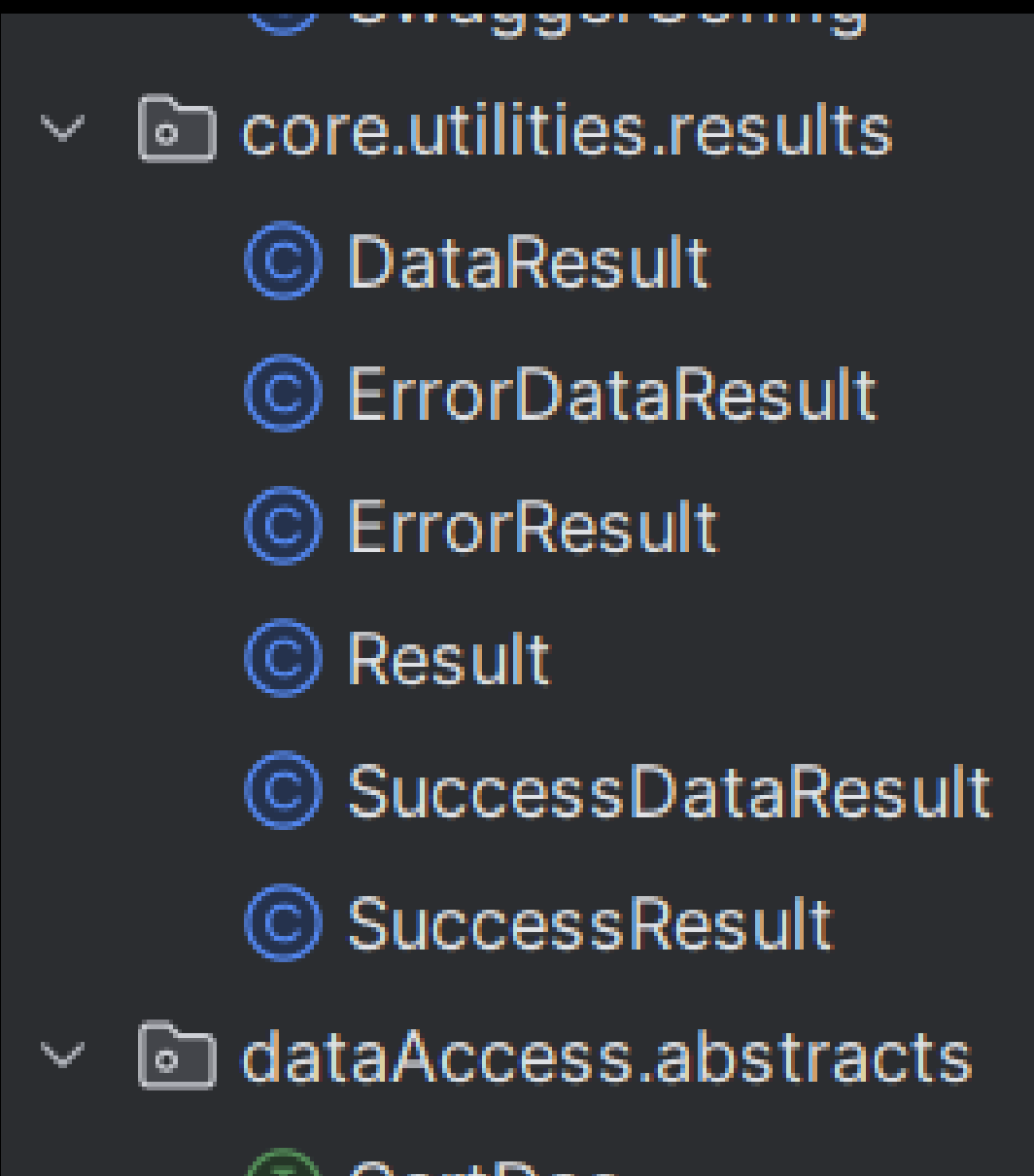

Structured Response

In my project, I've adopted a structured approach for API responses to ensure clarity and ease of understanding for the front-end. This structure encompasses two main categories: one for successful responses (data or result) and the other for errors.

As you may have noticed in the provided screenshot, I have organized the response into distinct key-value pairs, allowing for straightforward interpretation by the front-end. Whether the API call is expected to return data, a result without data, or an error, the response structure remains consistent.

This intentional design choice simplifies the front-end's interaction with the APIs. If the response were overly complex or lacked a uniform structure, it could pose challenges for the front-end developers in handling different scenarios. By maintaining consistency in the response structure across all APIs, I aim to enhance the overall development experience and facilitate a smoother integration process with the front-end.

This structured approach not only promotes code readability but also contributes to the maintainability of the project, making it easier for both current and future developers to work with the APIs.

A screenshot of an IDE showing the source code of the SuccessDataResult class. The class is defined in the package com.said.palidmarketapp.core.utilities.results and extends DataResult<T>. It has two constructors: one with parameters (T data, String message) and another with parameter (T data).

```
1 package com.said.palidmarketapp.core.utilities.results;
2
3 12 usages said4shi
4 public class SuccessDataResult <T> extends DataResult<T>{
5     10 usages said4shi
6     public SuccessDataResult(T data,String message) { super(
7
8     no usages said4shi
9     public SuccessDataResult(T data) { super(data, success: t
10 }
11
12
```