JAVA ACADEMY

XIDERAL

XIDERAL

FINALP.-JAVA

PRESENTATION

ABOUT PROJECT

The purpose of this project is to develop a simple Bank Transaction Processing System using Spring Boot and Maven with features for handling customer accounts, transactions between accounts, and ensuring proper management of account balances.

The project aims to simulate the core functionalities of a basic banking system where users can create accounts, check balances, transfer funds, and ensure that security protocols are in place to protect sensitive data and operations.

PROJECT DETAILS

- The project aims to develop a robust bank transaction processing system,
- ensuring data integrity and providing security for each operation.

Key Objectives:

- Create a system for managing bank accounts and transactions.
- Implement role-based security to protect sensitive data.
- Ensure secure transactions between accounts and manage user access.

Technologies Used:

- Spring Boot for building lightweight and efficient RESTful services.
- Spring Security for authentication and authorization implementation.
- Jakarta EE (instead of javax) for data persistence using JPA.
- MySQL as the database to store information about customers, accounts, and transactions.
- JUnit and Mockito for ensuring quality through unit and integration testing.

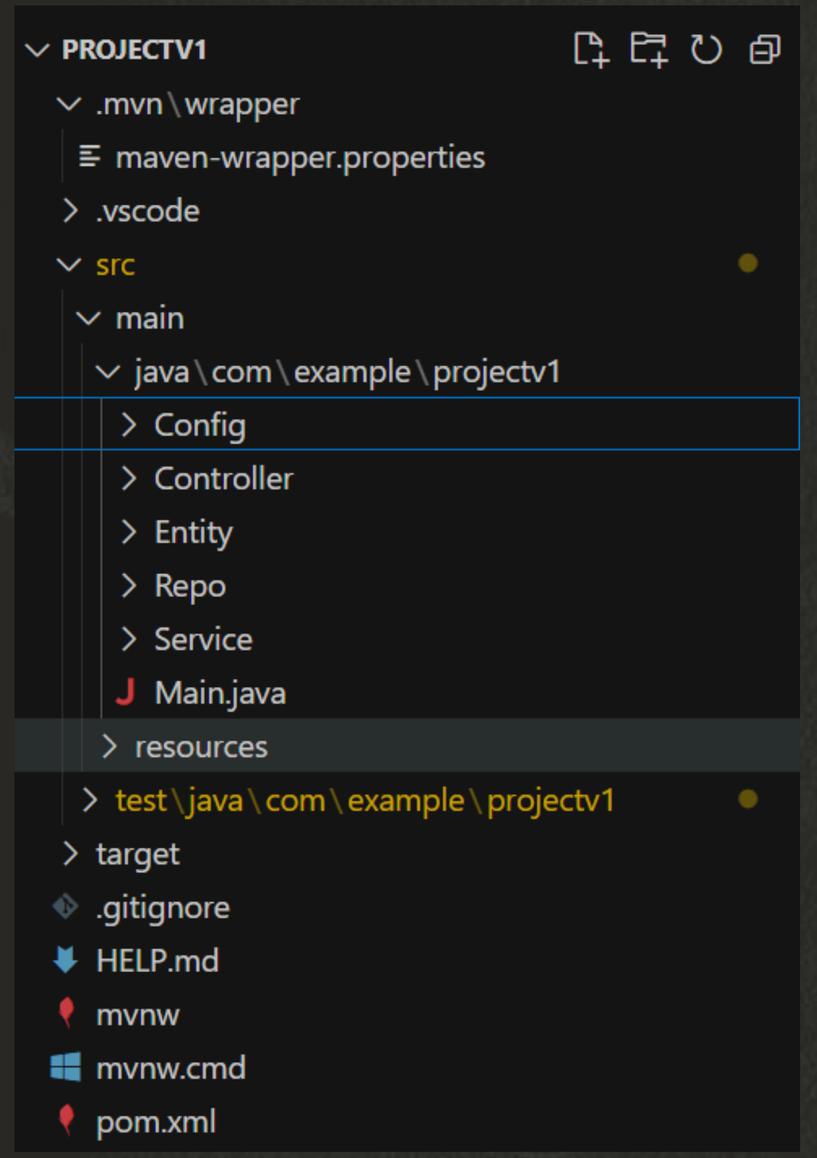
PROJECT STRUCTURE

Main Components:

- Customer Entity: Represents the bank customer with basic information such as name, address, and contact details.
- Account Entity: Manages the bank accounts linked to a customer, including balance and allowed operations.
- Transaction Entity: Handles all transactions between accounts, keeping a history and details of each transfer.

Project Architecture:

- Follows a Model-View-Controller (MVC)
- Entity relationships are defined using annotations like @ManyToOne



SECURITY

Spring Security:

- A simple security system
- Authentication: Users must log in with valid credentials
- Authorization: Role-based access control defines what actions each type of user can perform.
- Roles:
  - Admin: Has full control over accounts and transactions, can create and delete accounts, and perform critical operations.
  - User: Can only access their own accounts, check balances, and make authorized transfers between accounts.

Key Features:

- In-Memory Authentication is used at this initial stage for user authentication, but the system is flexible enough to switch to database-based or external authentication methods

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
        .csrf(csrf -> csrf.disable())
        .authorizeHttpRequests(auth -> auth
            // Definir primero las reglas más específicas
            .requestMatchers(...patterns:"/accounts/**").hasAnyRole(...roles:"USER", "ADMI
            .requestMatchers(...patterns:"/api/transactions/**").hasRole(role:"ADMIN") //
            // Then, allow any other request.
            .anyRequest().permitAll()  //Allow public access to other endpoints.
        )
        .httpBasic(withDefaults()); // Use basic authentication

    return http.build();
}
```

REST API

Main Endpoints:

- AccountController: Manages the creation, modification, and retrieval of bank accounts.
  It allows operations such as balance inquiries and account management for customers.
- TransactionController: Manages account transfers, ensuring that each operation is validated..
- CustomerController: Manages the creation of new customer who are the only ones that can create an account
- API Security: All operations are protected by a security layer to prevent unauthorized access,
  such as attempts to perform transactions without proper credentials.

User Workflow:

- The user logs in.
- Accesses their account to check balances or make transfers.
- Authentication and authorization ensure that they can only access their own information.

```
    // Endpoint to create a new account
    @PostMapping
    public ResponseEntity<Account> createAccount(@RequestBody Account account) {
    try {
        Account createdAccount = accountService.createAccount(account);
        return ResponseEntity.ok(createdAccount);
    } catch (Exception e) {
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(body:null);
    }
}
    // Endpoint to get an account by number
    @GetMapping("/{accountNumber}")
    public ResponseEntity<Account> getAccountByNumber(@PathVariable String accountNumber) {
        Optional<Account> account = accountService.getAccountByNumber(accountNumber);
        return account.map(ResponseEntity::ok)
                      .orElseGet(() -> ResponseEntity.notFound().build());
    }
```

UNIT TESTING - COVERAGE

JUnit and Mockito:

- Unit tests are implemented for each service, controller, and repository to ensure that each system component works as expected.
- Mockito is used to mock the interaction between dependencies, verifying that services operate correctly without connecting to the real database.
- Test Coverage: Includes security validation, ensuring roles and access controls are correctly configured, and verifying critical operations such as transactions and account management.

ProjectV1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.example.projectv1.Service		94%		100%	2	15	1	37	2	14	0	3
com.example.projectv1.Controller		93%		n/a	1	13	2	33	1	13	0	3
com.example.projectv1		37%		n/a	1	2	2	3	1	2	0	1
com.example.projectv1.Config		100%		n/a	0	5	0	18	0	5	0	1
com.example.projectv1.Entity		100%		100%	0	7	0	15	0	4	0	3
Total	21 of 446	95%	0 of 8	100%	4	42	5	106	4	38	0	11

CHALLENGES & LESSONS

Challenges:

- The main challenge was correctly setting up the relationships between entities, ensuring that transactions are linked to accounts without compromising data integrity.
- Ensuring the security implementation does not interfere with the system's functionality and that roles are properly enforced on each endpoint.

Lessons Learned:

- The importance of designing security from the initial stage to avoid issues later on.
- Properly structuring the project in layers (MVC) improves maintainability and simplifies unit testing.

FUTURE IMPROVEMENTS

- OAuth2 Authentication: Expand the system to support OAuth2, allowing users to authenticate via third parties such as Google or Facebook.
- Batch Processing: Implement batch processing to handle large volumes of transactions efficiently.
- Test Enhancements: Expand the range of unit and integration tests, covering more complex cases and simulating real scenarios with large amounts of data.