Final Project - Bank Transaction Processing Project

MIGUEL ANGEL RAMIREZ JUAREZ

JAVA ACADEMY

SEP/ 10/ 2024

Project Overview

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.

The project utilizes modern technologies, including **Spring Boot**, **Spring Security**, **Jakarta EE**, and **MySQL**, while following industry best practices for secure application development.

Objectives and Implementation

The project aimed to achieve the following:

1. **Entity Management**: Developed entities for **Customer**, **Account**, and **Transaction**. The Customer entity holds user details, Account represents different types of bank accounts, and Transaction records fund transfers between accounts. These entities are managed through a MySQL database using Jakarta Persistence (JPA).
2. **Business Logic**: Implemented in the service layer, which includes:
    - **AccountService**: Handles account creation, balance retrieval, and account management.
    - **TransactionService**: Manages fund transfers and updates transaction records.
    - **CustomerService**: Manages customer registration and information retrieval.
3. **Security Configuration**: Utilized **Spring Security** to enforce authentication and authorization. The security setup includes:
    - **Authentication**: Users must log in with valid credentials. An in-memory authentication provider is used for simplicity.
    - **Authorization**: Role-based access control ensures that only users with the appropriate roles (e.g., ADMIN, USER) can access certain functionalities. Admins can perform all operations, while regular users have limited access.
4. **REST API**: Created RESTful endpoints for account management and transactions. The AccountController and TransactionController expose operations like creating accounts, checking balances, and transferring funds. Security is enforced using role-based access control.

5. Testing

- **Unit Testing**: The service layer is tested to ensure business logic is correct. Tests cover account management and transaction processing.
- **Integration Testing**: Ensures that controllers interact correctly with services and repositories. Includes testing of REST API endpoints for functionality and security.
- **Security Testing**: Verifies that the security configuration correctly restricts access based on user roles. Includes tests to ensure that only authorized users can access restricted endpoints.

Entity Management

Entity Management in a Spring Boot project using JPA (Java Persistence API) is responsible for interacting with the database through entities, which are object representations of persistent data stored in the database. Entity management handles CRUD (Create, Read, Update, Delete) operations and manages the persistence and querying logic of these objects.

In JPA, an entity is a class mapped to a database table. Each entity represents a table, and its fields represent the columns in the table. Entities are the core objects managed by JPA.

Heres i show we management the entity Account:

```
package com.example.projectv1.Entity;

import java.math.BigDecimal;

import jakarta.persistence.Entity;
import jakarta.persistence.EnumType;
import jakarta.persistence.Enumerated;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.ManyToOne;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
```

```
public class Account {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String accountNumber;
    private String accountHolderName;

    @Enumerated(EnumType.STRING)
    private AccountType accountType;

    private BigDecimal balance;

    @ManyToOne
    private Customer customer;



    // Example of custom methods, if needed
    public void deposit(BigDecimal amount) {
        if (amount.compareTo(BigDecimal.ZERO) > 0) {
            this.balance = this.balance.add(amount);
        } else {
            throw new IllegalArgumentException("Deposit amount must be positive");
        }
    }

    public void withdraw(BigDecimal amount) {
         if (amount.compareTo(BigDecimal.ZERO) > 0 && amount.compareTo(this.balance)
<= 0) {
            this.balance = this.balance.subtract(amount);
        } else {
              throw new IllegalArgumentException("Invalid withdrawal amount or insufficient
funds");
        }
    }
}
```

@Entity: Marks this class as a JPA entity that will be mapped to a database table.
@Id: Specifies the primary key of the entity.
@GeneratedValue: Defines the strategy for automatically generating the value of the primary key (id).
Other fields, like accountNumber and balance, map directly to columns in the database table.

Repository

A **repository** is an interface that extends JpaRepository or CrudRepository. These interfaces provide predefined CRUD operations for entities. Spring Data JPA handles the implementation and database interactions automatically.

```
package com.example.projectv1.Repo;

import com.example.projectv1.Entity.Account;
import org.springframework.data.jpa.repository.JpaRepository;
import java.util.Optional;

public interface AccountRepo extends JpaRepository<Account, Long> {
    Optional<Account> findByAccountNumber(String accountNumber);

}
```

AccountRepository: Manages CRUD operations for the Account entity.
JpaRepository<Account, Long>: Extends CRUD functionality, adding pagination and sorting features. Account is the entity being managed, and Long is the type of its primary key.
findByAccountNumber: A custom query method to find an account by its account number.

Service Layer

The **service layer** contains business logic and interacts with repositories to manage entities. Services may include validations and other operations before making changes to persistent data.

```
package com.example.projectv1.Service;

import java.math.BigDecimal;
import java.util.List;
import java.util.Optional;

import org.springframework.stereotype.Service;

import com.example.projectv1.Entity.Account;
import com.example.projectv1.Repo.AccountRepo;


@Service
public class AccountService {
```

```
    private final AccountRepo accountRepo;

    public AccountService(AccountRepo accountRepo) {
        this.accountRepo = accountRepo;
    }

    // service to create a new account
    public Account createAccount(Account account) {
        account.setBalance(BigDecimal.ZERO); // Inicia con saldo cero
        return accountRepo.save(account);
    }

    // aget account by number of account
    public Optional<Account> getAccountByNumber(String accountNumber) {
        return accountRepo.findByAccountNumber(accountNumber);
    }

    // get de ammount of a account
    public BigDecimal getAccountBalance(String accountNumber) {
        return accountRepo.findByAccountNumber(accountNumber)
                .map(Account::getBalance)
                    .orElseThrow(() -> new RuntimeException("DIDNT FIN THE ACCOUNT,
PLEASE CHECK THE INFO AND TRY AGAIN"));
    }

    public List<Account> getAllAccounts() {
        return accountRepo.findAll();
    }
}
```

AccountService: Handles business logic related to account management.
getAccountByAccountNumber: Calls the repository to retrieve an account by its account number, throwing an exception if the account is not found.
createAccount: Uses the repository to create (persist) a new account.

Entity Lifecycle Management

JPA automatically manages the lifecycle of entities, which includes states like:

- **Transient**: The entity is not yet persisted.

- **Managed**: The entity is being managed by the **EntityManager** (retrieved from the database or newly persisted).
- **Detached**: The entity is no longer managed, typically after the session has closed.
- **Removed**: The entity is marked for deletion from the database.

Security Configuration

Spring Security is an essential part of the project that handles authentication, authorization, and security configurations. It is seamlessly integrated with the Spring Boot application to ensure secure access to the APIs, endpoints, and resources based on user roles and permissions. Below is a detailed explanation of how Spring Security is integrated and configured in the project.

The Spring Security setup includes:

- **In-Memory Authentication**: Uses hardcoded credentials for admin and user roles.
- **Role-Based Authorization**: Configures endpoint access based on user roles. Admin endpoints are restricted to users with the ADMIN role, while user endpoints are accessible to USERS and ADMINs.
-

```
@Configuration
public class SecurityConfig {

    @Bean
        public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                // Definir primero las reglas más específicas
                    .requestMatchers("/accounts/**").hasAnyRole("USER", "ADMIN") // Protect these endpoints for users and administrators
                    .requestMatchers("/api/transactions/**").hasRole("ADMIN") // Only accessible by administrators.
                // Then, allow any other request.
                .anyRequest().permitAll()  //Allow public access to other endpoints.
            )
            .httpBasic(withDefaults()); // Use basic authentication

        return http.build();
    }

    @Bean
public UserDetailsService userDetailsService() {
    UserDetails user = User.withUsername("user")
        .password("{noop}password")
        .roles("USER")
        .build();

```

```
    UserDetails admin = User.withUsername("admin")
        .password("{noop}admin")
        .roles("ADMIN")
        .build();

    return new InMemoryUserDetailsManager(user, admin);
}
}
```

Results and Achievements

The project successfully implements a basic banking system where:

- Accounts and transactions are managed efficiently.
- Security measures are in place to protect sensitive operations.
- A functional REST API allows users to interact with the system while ensuring proper access control.

Explanation:

- **HttpSecurity**: Configures HTTP-level security. It defines rules such as:
  - **/accounts/****: Accessible by both "USER" and "ADMIN" roles.
  - **/api/transactions/****: Restricted to only "ADMIN" role.
  - **csrf.disable()**: Disables CSRF protection (which might be used in environments where it's not needed, like API-only applications).
  - **httpBasic()**: Implements Basic Authentication.
- **In-memory Authentication**: Two users are defined with hardcoded usernames and passwords: one with the "USER" role and one with the "ADMIN" role.

Authentication Mechanism

Spring Security uses an in-memory authentication mechanism in this project. This is configured through the UserDetailsService bean, which defines the users (user and admin) and their respective roles and passwords.

In this project:

- The **"USER" role** has access to account-related endpoints (/accounts/**).
- The **"ADMIN" role** has additional access to transaction-related APIs (/api/transactions/**).

The user authentication process works as follows:

- When a user tries to access a secured endpoint, Spring Security checks their credentials using the in-memory user details (in a real-world scenario, this could be connected to a database or an external identity provider).
- Based on their role, the user is either granted or denied access.

Authorization

Authorization is managed through **role-based access control (RBAC)**, where different users are assigned specific roles that dictate which resources or endpoints they can access.

Role Configuration:

- **hasAnyRole("USER", "ADMIN")**: This allows users with either the USER or ADMIN role to access certain endpoints (like /accounts/**).
- **hasRole("ADMIN")**: This restricts some sensitive endpoints (like transaction APIs) to only administrators.

Spring Security checks the user's role after authentication to ensure they have the necessary permissions to access the requested resource.

HTTP Basic Authentication

In this project, **HTTP Basic Authentication** is used, which is a simple and widely supported protocol where users provide their credentials (username and password) in each HTTP request header. While it's not the most secure method (compared to OAuth or JWT), it is adequate for simpler applications or scenarios where transport layer security (HTTPS) is enforced.

- **How it works**: The client includes an **Authorization** header in each HTTP request containing the Base64-encoded username and password.

Spring Security Test Integration

The project integrates **Spring Security testing** to ensure that roles and permissions are properly enforced during authentication and authorization.

```
@SpringBootTest
@AutoConfigureMockMvc
class SecurityConfigTest {

    @Autowired
    private MockMvc mockMvc;
```

```
    @Test
    @WithMockUser(username = "user", roles = {"USER"}, password = "password") // Usuario "USER" no debería acceder a transacciones
    void testTransactionAccessAccessForUserRole() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post("/api/transactions/transfer")
                .param("sourceAccountNumber", "18131049")
                .param("destinationAccountNumber", "1234567890")
                .param("amount", "10.00")) // Parámetros necesarios
                .andExpect(status().isForbidden()); // Verificamos que sea 403 Forbidden
    }

    @Test
    @WithMockUser(username = "admin", roles = {"ADMIN"}, password = "admin")
    void testTransactionAccessWithAdminRole() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post("/api/transactions/transfer")
                .param("sourceAccountNumber", "18131049")
                .param("destinationAccountNumber", "1234567890")
                .param("amount", "10.00")) // Parámetros necesarios
                .andExpect(status().isOk()); // Espera un 200 OK
    }

}
```

Explanation:

- **@WithMockUser**: This annotation allows us to simulate a logged-in user with specific roles during testing. In this case:
    - The first test checks that a user with the "USER" role cannot access the transaction endpoint (403 Forbidden).
    - The second test checks that an "ADMIN" can access the same endpoint (200 OK).
- **MockMvc**: This is used to perform requests and validate the response status. The mockMvc object simulates HTTP requests to the Spring application without starting the entire server.

Handling Authorization Failures

If a user tries to access an endpoint without the necessary permissions:

- Spring Security returns a **403 Forbidden** status, indicating that the user is authenticated but does not have the correct role to access the resource.

For example, in the test testTransactionAccessDeniedForUserRole, a user with the role "USER" cannot access the /api/transactions/transfer endpoint, which is restricted to admins. This leads to a 403 status code.

Transaction and Account Endpoint Protection

- **Transaction API** (/api/transactions/): Only admins can access or initiate transactions. Regular users are restricted from making any transfers or viewing sensitive transaction information.
- **Accounts API** (/accounts/): Both "USER" and "ADMIN" roles are granted access to account-related operations. Users can view their account details, and administrators have full access as well.

Database Design

The database design for this project is centered around three key entities: customer, account, and transaction. These tables establish a relationship between customers and their financial accounts and record the transactions that occur between different accounts. The design ensures data integrity through foreign key constraints and maintains uniqueness and relationships between entities using primary keys and foreign keys.

Customer Table

The customer table stores the basic information of the bank's clients, such as their name, email, and the timestamp of when their record was created.

```
CREATE TABLE IF NOT EXISTS customer (

id BIGINT AUTO_INCREMENT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

email VARCHAR(255) UNIQUE,

created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);
```

Columns Explanation:

- **id**: This is the primary key, a unique identifier for each customer. It is an AUTO_INCREMENT column, meaning the database will automatically generate a unique ID for each new record.

- **name**: The name of the customer. It is a VARCHAR field, restricted to 255 characters, and is required (NOT NULL).
- **email**: The email address of the customer, which is unique across the table. This ensures that no two customers can register with the same email.
- **created_at**: A timestamp that records when the customer was added to the database. It defaults to the current time (CURRENT_TIMESTAMP), so it automatically tracks when the customer record is created.

Testing and Code Coverage

In the context of this project, testing is crucial to ensure that all parts of the application, including the Spring Security configuration and the REST endpoints, work as expected. The goal is to achieve high code coverage, ensuring that a large percentage of the codebase is tested, minimizing the risk of unexpected behavior during runtime.

1. Unit Testing

The project uses **JUnit 5** for writing unit tests. Unit tests are responsible for testing individual components of the application in isolation, ensuring that methods and logic behave as expected under various conditions. The core focus is on testing controllers, services, and Spring Security configuration.

For example, in this project, the unit tests validate:

- **Security Access Control**: To ensure that users with different roles (like "USER" and "ADMIN") have appropriate access to different endpoints.
- **REST API Controllers**: Tests ensure that the endpoints work as expected, validate the status codes returned, and confirm that correct data is passed and retrieved.

MockMVC for Controller Testing

In the project, **MockMVC** is used to simulate HTTP requests to the controllers and verify their responses without needing a fully running web server. MockMVC allows for the testing of the entire flow of an HTTP request, including authentication and authorization.

Example of using **MockMVC** in a test:

```
@Autowired
  private MockMvc mockMvc;
```

```
    @Test
    @WithMockUser(username = "user", roles = {"USER"}, password = "password") // Usuario "USER" no debería acceder a transacciones
    void testTransactionAccessAccessForUserRole() throws Exception {
        mockMvc.perform(MockMvcRequestBuilders.post("/api/transactions/transfer")
                .param("sourceAccountNumber", "18131049")
                .param("destinationAccountNumber", "1234567890")
                .param("amount", "10.00")) // Parámetros necesarios
                .andExpect(status().isForbidden()); // Verificamos que sea 403 Forbidden
    }
```

Test Coverage Tools

In addition to writing tests, it's essential to measure **test coverage**. This ensures that all important parts of the code are being tested. A popular tool for measuring code coverage in Java is **JaCoCo**.

Code Coverage Goals:

- **Controller Coverage**: Ensure all the REST endpoints are covered, verifying that the right HTTP status codes and responses are returned based on user input.
- **Service Layer**: Test the business logic in the service layer, ensuring methods perform as expected.
- **Security Layer**: Verify that Spring Security is configured correctly, and ensure that access control is functioning properly based on user roles.

With JaCoCo, a **coverage report** can be generated, showing the percentage of code that is covered by unit tests. JaCoCo provides granular details like line coverage, branch coverage (for conditionals), and method coverage.

Generating Coverage Reports:

In Maven, the following plugin can be added to the pom.xml file to generate coverage reports:

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.7</version>
    <executions>
        <execution>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>test</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Running mvn test will not only run the tests but also generate a coverage report that can be found in the target/site/jacoco directory.

Edge Case Testing

It's also important to ensure that edge cases are covered in the tests. This includes:

- **Invalid input handling**: Testing how the system handles missing or incorrect data (e.g., missing account numbers, invalid email formats).
- **Boundary testing**: Testing limits of allowed input, such as the maximum account balance or the minimum amount for transactions.
- **Security breaches**: Trying to access restricted endpoints without proper authentication or authorization.

Here is the result of the coverage of us code in JaCoCo:

ProjectV1

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		0%		n/a	2	2	3	3	2	2	1	1
com.example.projectv1.Controller		93%		n/a	1	13	2	33	1	13	0	3
com.example.projectv1.Entity		100%		100%	0	7	0	15	0	4	0	3
Total	24 of 352	93%	0 of 8	100%	5	37	6	88	5	33	1	10

Postman Testing for the Project

Postman is an API client that allows developers to test, document, and monitor APIs easily. For this project, Postman was used to test the REST endpoints, verifying the security configuration, and ensuring that each endpoint works as expected with proper role-based access control.

1. Setting Up Postman for API Testing

Before testing, ensure that your application is running locally (e.g., on http://localhost:8080). You will be testing endpoints like /api/transactions/transfer, /api/accounts, and others related to accounts and transactions.

1. **Base URL**: http://localhost:8080 (or whichever port your Spring Boot application is running on).
2. **Authorization**: The project uses Basic Authentication for user roles, so you need to pass valid credentials for users with the required roles (e.g., "USER" or "ADMIN").
    - **Username and Password** for testing:
        - user with password password (ROLE: USER)
        - admin with password admin (ROLE: ADMIN)

Testing the /api/transactions/transfer Endpoint

This endpoint allows users to perform transactions between accounts. Only administrators (users with the "ADMIN" role) are allowed to access this endpoint.

Testing the /api/transactions/transfer Endpoint

This endpoint allows users to perform transactions between accounts. Only administrators (users with the "ADMIN" role) are allowed to access this endpoint.

Request Type: POST

Endpoint: /api/transactions/transfer

Steps:

1. **Authorization**: In the Authorization tab, select "Basic Auth" and enter the credentials:
    - **Username**: admin
    - **Password**: admin
2. **Request Type**: Set the method to POST.
3. **Endpoint**: Enter the endpoint URL http://localhost:8080/api/transactions/transfer.
4. **Query Parameters**:
    - In the "Params" section, add the following parameters:
        - sourceAccountNumber: The account number from which the money is transferred (e.g., 1234567890).

        - destinationAccountNumber: The account number to which the money is transferred (e.g., 18131049).
        - amount: The amount to be transferred (e.g., 1000).

1. **Authorization**: In the Authorization tab, select "Basic Auth" and enter the credentials:
    - **Username**: admin
    - **Password**: admin
2. **Request Type**: Set the method to POST.
3. **Endpoint**: Enter the endpoint URL http://localhost:8080/api/transactions/transfer.
4. **Query Parameters**:
    - In the "Params" section, add the following parameters:
        - sourceAccountNumber: The account number from which the money is transferred (e.g., 1234567890).
        - destinationAccountNumber: The account number to which the money is transferred (e.g., 18131049).
        - amount: The amount to be transferred (e.g., 1000).

Testing the /api/transactions/transfer Endpoint

This endpoint allows users to perform transactions between accounts. Only administrators (users with the "ADMIN" role) are allowed to access this endpoint.

Request Type: POST

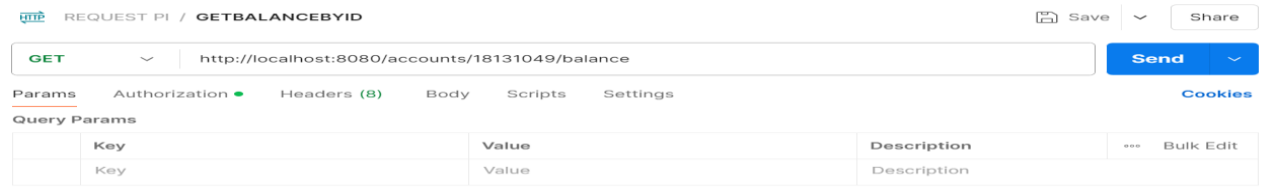
Endpoint: /api/transactions/transfer

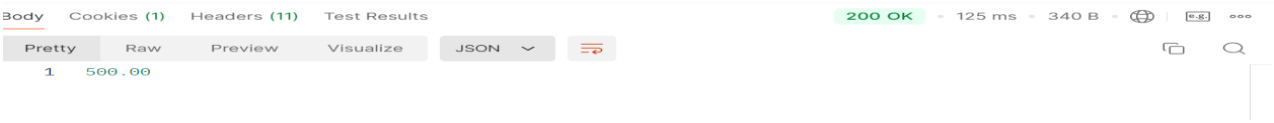
Steps:

1. **Authorization**: In the Authorization tab, select "Basic Auth" and enter the credentials:
    - **Username**: admin
    - **Password**: admin
2. **Request Type**: Set the method to POST.
3. **Endpoint**: Enter the endpoint URL http://localhost:8080/api/transactions/transfer.
4. **Query Parameters**:
    - In the "Params" section, add the following parameters:
        - sourceAccountNumber: The account number from which the money is transferred (e.g., 1234567890).
        - destinationAccountNumber: The account number to which the money is transferred (e.g., 18131049).
        - amount: The amount to be transferred (e.g., 1000).
        -
5. **Send the Request**: After setting the parameters and authorization, click "Send."
6. **Expected Response**: If the request is successful, you should receive an HTTP status code 200 OK with a success message confirming the transaction.

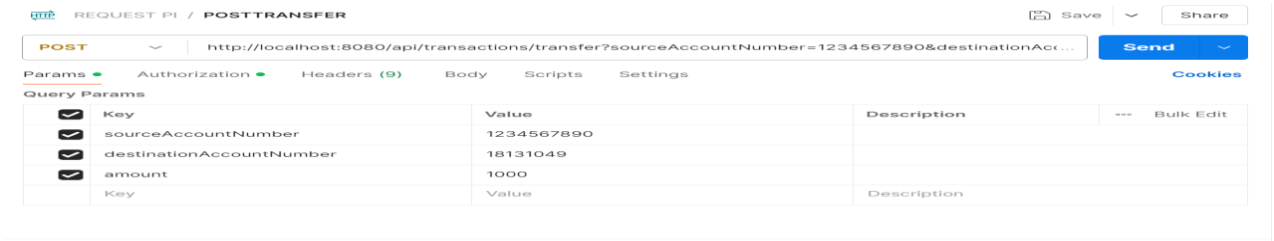
EXAMPLE OF TRANSACTION IN POSTMAN:

BALANCE BEFORE TRANSACTION:

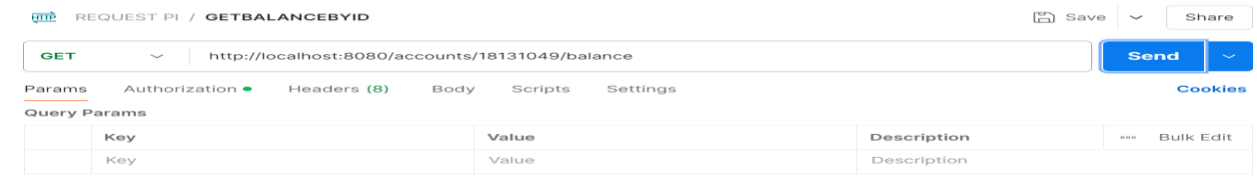


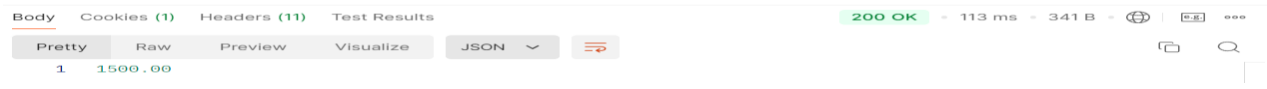


THE TRANSACTION TRANSFER:



BALANCE AFTER TRANSACTION:





Conclusion

This project demonstrates a comprehensive implementation of a Spring Boot application with essential functionalities like entity management, role-based security using Spring Security, a well-designed database schema, and API testing using Postman.

- **Entity Management**: The entities, such as `Customer`, `Account`, and `Transaction`, are well-structured, following relational database principles and leveraging Spring JPA for data management. Relationships between entities, such as one-to-many and foreign keys, are appropriately used, ensuring data integrity.
- **Spring Security Integration**: The project integrates Spring Security to enforce role-based access control. Users with specific roles (e.g., `USER` or `ADMIN`) have different levels of access to various endpoints, providing a robust layer of security. The use of in-memory authentication for demonstration purposes allows the easy assignment of roles and testing of access restrictions.
- **Database Design**: The database design adheres to best practices, normalizing data and using constraints like `FOREIGN KEY` to maintain relationships between tables. With tables like `customer`, `account`, and `transaction`, the design efficiently handles account management and transaction tracking, ensuring scalability and maintainability.
- **Testing and Code Coverage**: The project includes unit tests using JUnit and MockMVC to ensure that each functionality, particularly security restrictions, is properly validated. The tests simulate various user roles, ensuring that unauthorized users cannot access protected endpoints. This testing framework ensures a high level of code coverage, increasing confidence in the robustness of the application.
- **Postman Testing**: Postman was instrumental in manually testing the APIs. Through detailed requests with appropriate parameters and role-based authentication, we validated the behavior of each endpoint, ensuring that the security policies and business logic were functioning as expected.

Overall, the project showcases a solid understanding of Spring Boot fundamentals, security mechanisms, and database design. It highlights the importance of testing (both unit and integration) and demonstrates how Spring's various modules can be effectively used to build secure, scalable, and maintainable applications.