TED UNIVERSITY

PROJECT REPORT

BLUE PROJECT

DONE

Interbank Payment System

Author(s):

Famura

Contents

1. Project Description

1.1. Introduction

The Interbank Payment Service project simulates a simplified interbank transaction platform that allows deposits, withdrawals, and transfers between users belonging to different banks.

The main goals are:

- To integrate different technologies typically used in enterprise Java applications (Spring, Hibernate, SOAP, Kafka, Jetty, MySQL).
- To demonstrate reliable, asynchronous communication between front-end/API layers and the back-end processing layer using Apache Kafka.
- To persist user and transaction data in a relational database in a consistent and transactional way.

The system models a scenario where banks and users interact through a central service that accepts payment requests, publishes them as messages, and processes them to update balances and record transactions.

1.2. Project Objectives

- **Objective 1 – Interbank transaction simulation:**
  Implement basic banking operations (deposit, withdraw, transfer) between users that may belong to different banks.
- **Objective 2 – Message-driven architecture:**
  Use Apache Kafka as a message broker between the API layer and the business logic that updates balances and writes transaction logs.
- **Objective 3 – Enterprise Java stack:**
  Build the application without Spring Boot, using:
  - Spring Core / Spring MVC / Spring WS
  - Hibernate ORM with MySQL
  - Jetty as an embedded servlet container
  - SOAP web services
- **Objective 4 – Consistency and transactions:**
  Ensure that balance updates and transaction logs are executed within database transactions, minimizing the risk of inconsistent data.
- **Objective 5 – Extensibility:**
  Design the code in a modular way (separate packages for API, messaging, database) so that new banks or operations can be added easily.

2. Project Technologies

2.1. Module(s)

- **Windows (Windows 10):**
  The entire project is designed to run in Windows systems. Did not tested in lower Windows versions like Windows 7. But tested in Windows 11 and running flawlessly.

- **Android(Android 13+):**
  The project was not developed mobile side kept in mind. But since we use SOAP API, it automatically handles mobile connections. Thus, allowing us to use and test our product in mobile as well. But it has not been tested in the IOS(probably running flawlessly there as well but just to be sure, it has not been tested there).

2.2. Coding Language(s)

- **Java (Java 21 / Maven - Spring) – IntelliJ IDEA Community Edition:**
  The entire project is written in Java 21 Spring(not Spring boot just Spring) and is set to compile with the Maven. This was one of the requirements of the project. IntelliJ IDEA is chosen as IDE for easy-to-handle Git tools.

- **HTML/CSS /JavaScript - IntelliJ IDEA Community Edition / Opera GX & Microsoft Edge:**
  The client-side interface is developed using HTML for structure, CSS for styling, and JavaScript for dynamic behaviour. These technologies are used to build a responsive and interactive user interface. IntelliJ IDEA Community Edition is utilized for code editing and project organization, while Opera GX and Microsoft Edge are used for testing and debugging the web interface. Opera GX and Microsoft Edge is used because they were the only available web browsers on my machine.

2.3. Libraries & External Additions

2.3.1. Used Systems

- **Soap Api:**
  Provides a protocol-based web service communication mechanism using XML messages over HTTP/HTTPS. SOAP ensures structured, standardized, and secure data exchange between distributed systems.

- **Apache Kafka:**
  A distributed event-streaming platform used for real-time data pipelines and asynchronous message communication between services. Kafka enables high-throughput, fault-tolerant, and scalable messaging.

- **Hibernate:**
  An Object-Relational Mapping (ORM) framework that simplifies database interaction by mapping Java objects to relational database tables, reducing the need for manual SQL queries.

- **Docker:**
  A containerization platform used to package applications and their dependencies into isolated containers, ensuring consistent execution across different environments. Docker simplifies deployment, scaling, and environment management in software development.

- MySQL:
  An open-source relational database management system (RDBMS) used for storing, managing, and querying structured data. MySQL supports SQL-based data manipulation and is widely used in backend systems for reliable and efficient data persistence. Used with Docker in this project.

2.3.2. Others

- **Jetty 11:**
  A lightweight and high-performance web server and servlet container used to deploy and run Java-based web applications. Jetty 11 supports modern Java standards and is commonly used for embedded server configurations.

- **Jackson Databind:**
  A Java library is used for JSON serialization and deserialization, enabling efficient conversion between Java objects and JSON data structures. It is widely used for handling data exchange in REST and web service applications.

- **SLF4J Simple:**
  A logging framework implementation is used to produce runtime log messages for monitoring application behavior and debugging purposes. SLF4J provides a simple and unified logging abstraction.

- **Spring Context, Spring Web, Spring Web MVC:**
  Core Spring Framework modules used for dependency injection, web application development, and request handling based on the Model-View-Controller (MVC) pattern. These modules enable structured, scalable, and maintainable server-side application architecture.

- **Ngrok:**
  A tunneling tool used to expose local servers to the internet through secure public URLs. ngrok is commonly used for testing and debugging web services, APIs, and webhook integrations without deploying the application to a production environment. Free version was used.

3. Project Architecture

As can be understood in the content of the previous sections, the project is divided into three parts.

This layered architecture is designed to separate concerns between user interaction, business logic, and data persistence, thereby improving maintainability and scalability. By decoupling synchronous request handling from asynchronous processing, the system ensures reliability under concurrent operations and allows each layer to evolve independently.

3.1. API Layer

This part is developed using Soap API. And it works like this:

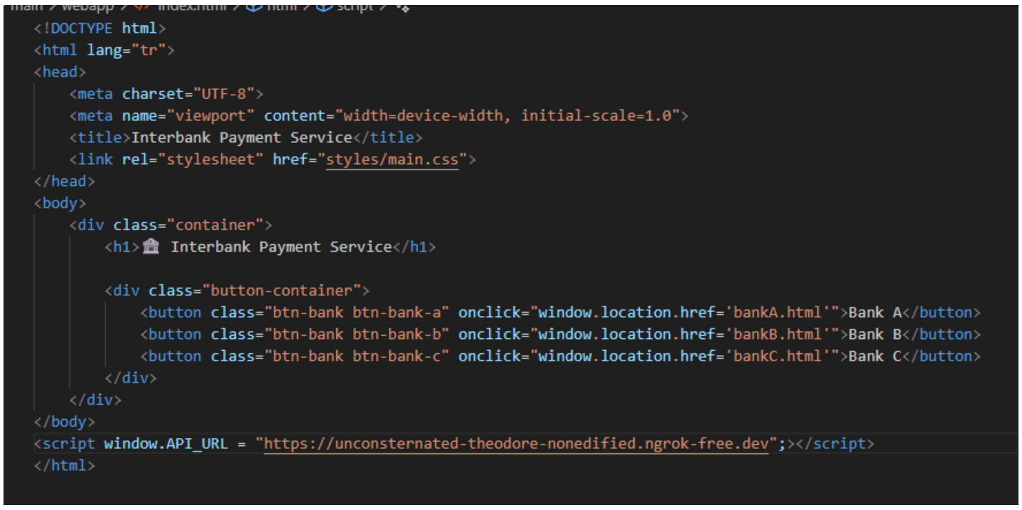
There is a lower layer which is user layer that uses Html, Css and Javascript files to function and acting as a UI to the system. This part also collects requests sent by user as ".xml" format and sends them to second lower layer which is Soap Layer. Soap Layer collects these requests in its endpoints and converts these requests into ".json" format so that Kafka can read it. Then it publishes them.

3.1.1. User Layer

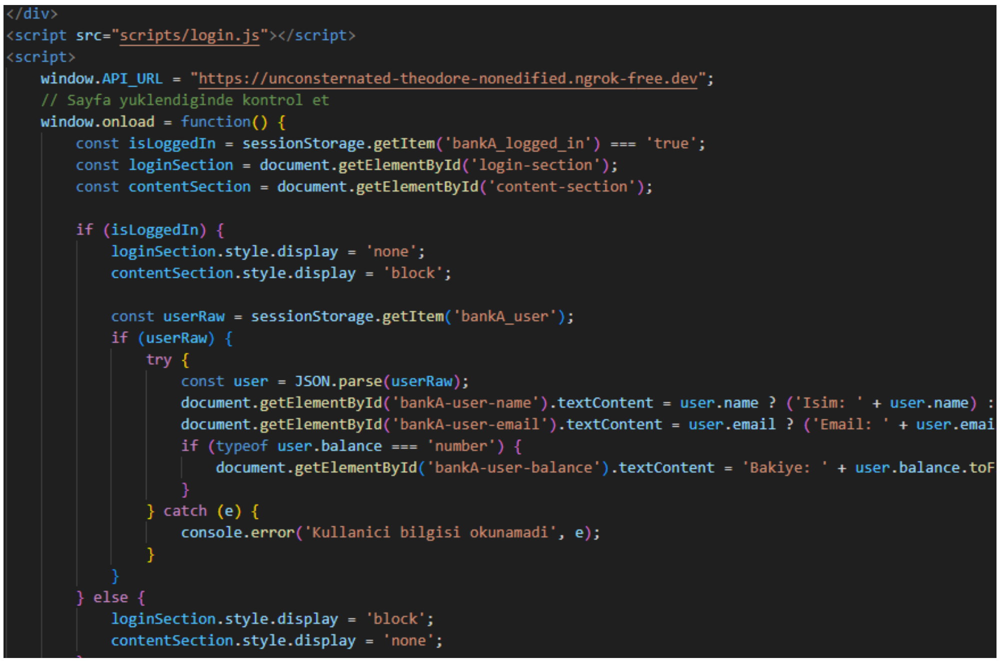
3.1.1.1. Login

User opens the banking interface and logs in using the login form. Login Servlet and index.html used here. Information about login servlet can be found on 3.1.2.1. Login header.

Onto the html part, firstly a main page opens. In that page user can choose any of the banks. And the application redirects users to the required html page that are all indicated to their respective banks. Bank A's codes will be used in the remaining of the report for simplicity.

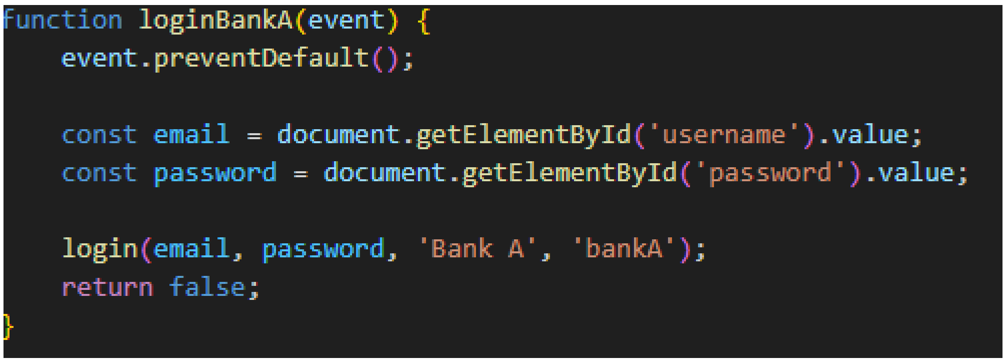


```
main > webapp > index.html > html > script
<!DOCTYPE html>
<html lang="tr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Interbank Payment Service</title>
    <link rel="stylesheet" href="styles/main.css">
</head>
<body>
    <div class="container">
        <h1>🏦 Interbank Payment Service</h1>

        <div class="button-container">
            <button class="btn-bank btn-bank-a" onclick="window.location.href='bankA.html'">Bank A</button>
            <button class="btn-bank btn-bank-b" onclick="window.location.href='bankB.html'">Bank B</button>
            <button class="btn-bank btn-bank-c" onclick="window.location.href='bankC.html'">Bank C</button>
        </div>
    </div>
</body>
<script window.API_URL = "https://unconsternated-theodore-nonedified.ngrok-free.dev";></script>
</html>
```

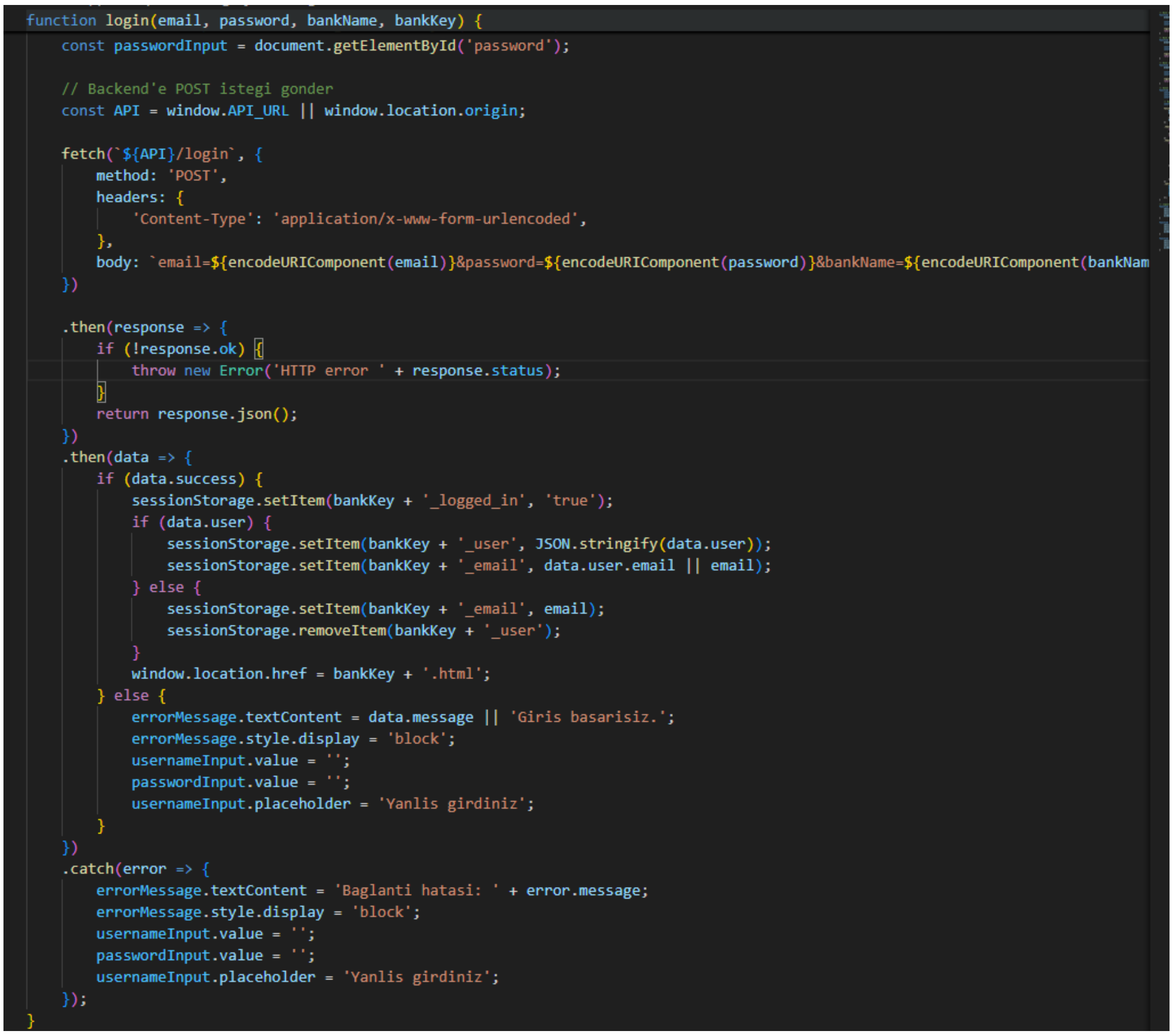


```
</div>
<script src="scripts/login.js"></script>
<script>
    window.API_URL = "https://unconsternated-theodore-nonedified.ngrok-free.dev";
    // Sayfa yuklendiginde kontrol et
    window.onload = function() {
        const isLoggedIn = sessionStorage.getItem('bankA_logged_in') === 'true';
        const loginSection = document.getElementById('login-section');
        const contentSection = document.getElementById('content-section');

        if (isLoggedIn) {
            loginSection.style.display = 'none';
            contentSection.style.display = 'block';

            const userRaw = sessionStorage.getItem('bankA_user');
            if (userRaw) {
                try {
                    const user = JSON.parse(userRaw);
                    document.getElementById('bankA-user-name').textContent = user.name ? ('Isim: ' + user.name) :
                    document.getElementById('bankA-user-email').textContent = user.email ? ('Email: ' + user.emai
                    if (typeof user.balance === 'number') {
                        document.getElementById('bankA-user-balance').textContent = 'Bakiye: ' + user.balance.toF
                    }
                } catch (e) {
                    console.error('Kullanici bilgisi okunamadi', e);
                }
            }
        } else {
            loginSection.style.display = 'block';
            contentSection.style.display = 'none';
```

And in here the user is directed to login.js so that user’s input can be converted into xml format.



```
function loginBankA(event) {
    event.preventDefault();

    const email = document.getElementById('username').value;
    const password = document.getElementById('password').value;

    login(email, password, 'Bank A', 'bankA');
    return false;
}
```

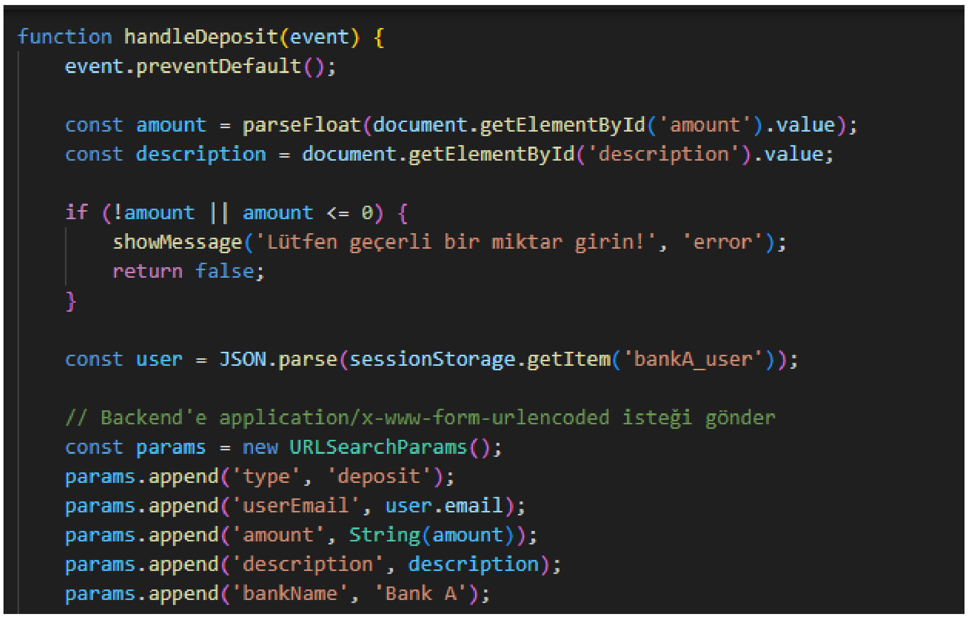


```
function login(email, password, bankName, bankKey) {
    const passwordInput = document.getElementById('password');

    // Backend'e POST istegi gonder
    const API = window.API_URL || window.location.origin;

    fetch(`${API}/login`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
        },
        body: `email=${encodeURIComponent(email)}&password=${encodeURIComponent(password)}&bankName=${encodeURIComponent(bankNam
    })

    .then(response => {
        if (!response.ok) {
            throw new Error('HTTP error ' + response.status);
        }
        return response.json();
    })
    .then(data => {
        if (data.success) {
            sessionStorage.setItem(bankKey + '_logged_in', 'true');
            if (data.user) {
                sessionStorage.setItem(bankKey + '_user', JSON.stringify(data.user));
                sessionStorage.setItem(bankKey + '_email', data.user.email || email);
            } else {
                sessionStorage.setItem(bankKey + '_email', email);
                sessionStorage.removeItem(bankKey + '_user');
            }
            window.location.href = bankKey + '.html';
        } else {
            errorMessage.textContent = data.message || 'Giris basarisiz.';
            errorMessage.style.display = 'block';
            usernameInput.value = '';
            passwordInput.value = '';
            usernameInput.placeholder = 'Yanlis girdiniz';
        }
    })
    .catch(error => {
        errorMessage.textContent = 'Baglanti hatasi: ' + error.message;
        errorMessage.style.display = 'block';
        usernameInput.value = '';
        passwordInput.value = '';
        usernameInput.placeholder = 'Yanlis girdiniz';
    });
}
```

If the user successfully logs in, all the operations are allowed like deposit.

3.1.1.2. Submitted Transaction

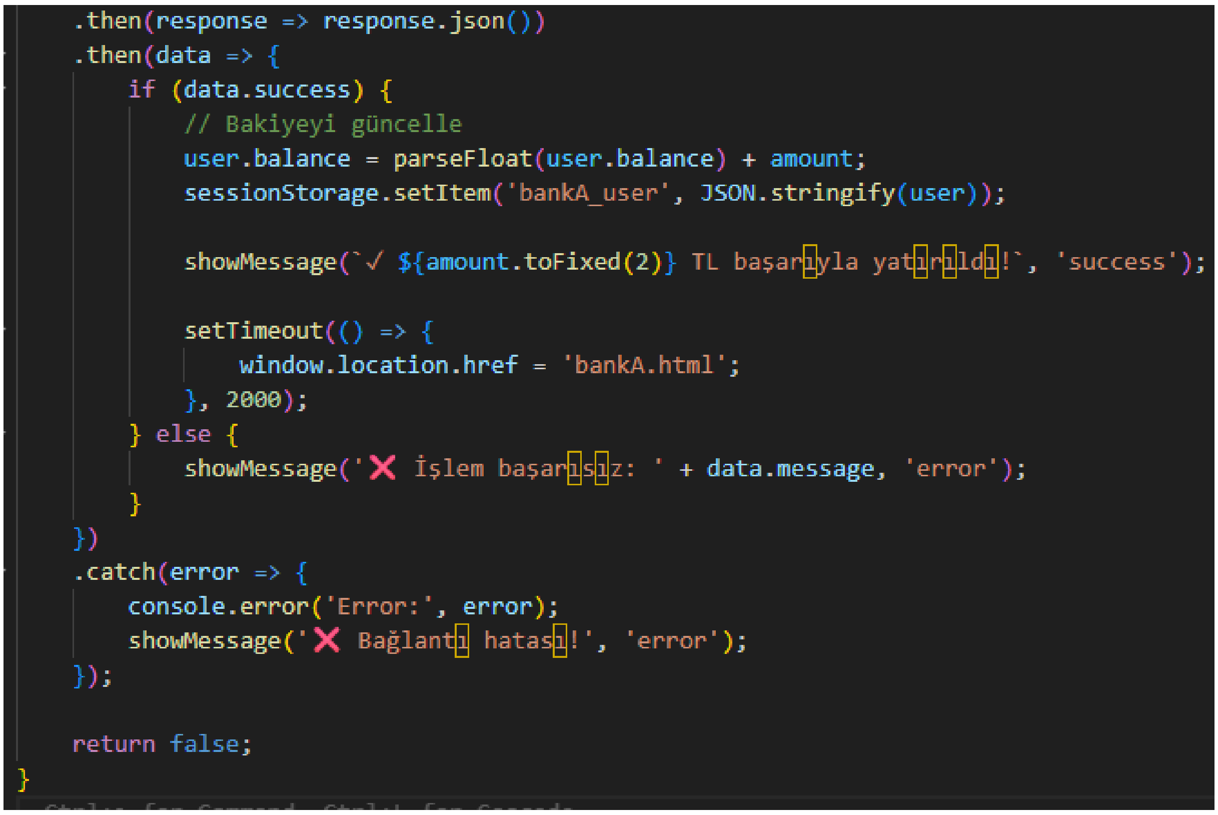
Now that user can do anything, they will do some transactions. And after filling required parameters regarding their operation, all of these parameters are send to corresponding API endpoints.



```
function handleDeposit(event) {
    event.preventDefault();

    const amount = parseFloat(document.getElementById('amount').value);
    const description = document.getElementById('description').value;

    if (!amount || amount <= 0) {
        showMessage('Lütfen geçerli bir miktar girin!', 'error');
        return false;
    }

    const user = JSON.parse(sessionStorage.getItem('bankA_user'));

    // Backend'e application/x-www-form-urlencoded isteği gönder
    const params = new URLSearchParams();
    params.append('type', 'deposit');
    params.append('userEmail', user.email);
    params.append('amount', String(amount));
    params.append('description', description);
    params.append('bankName', 'Bank A');
```

3.1.1.3. Response

Lastly, UI waits for response and shows it to the user.



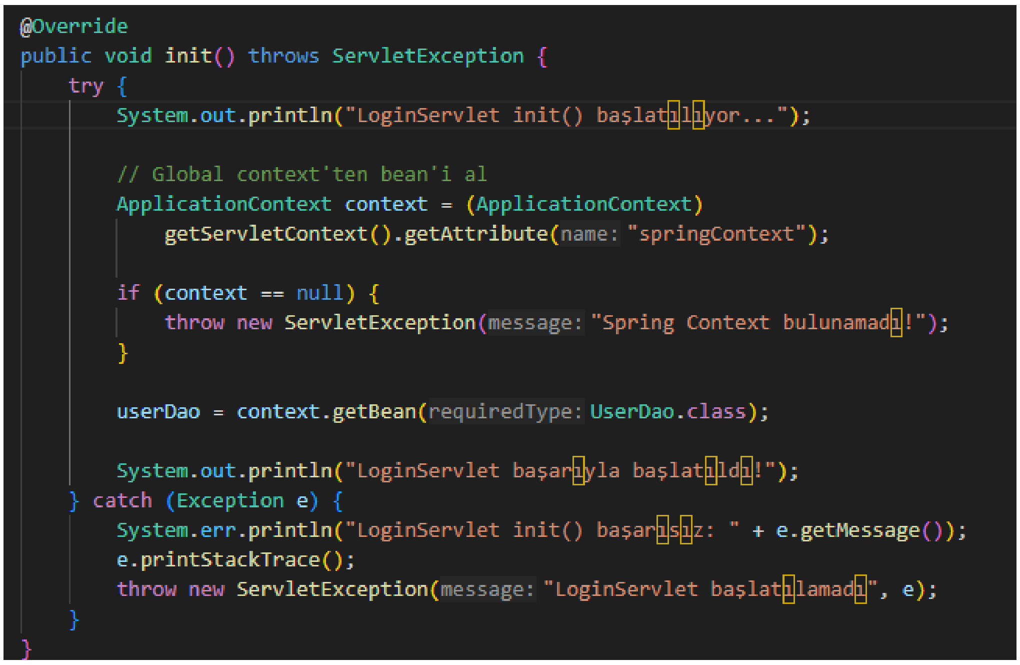
```
    .then(response => response.json())
    .then(data => {
        if (data.success) {
            // Bakiyeyi güncelle
            user.balance = parseFloat(user.balance) + amount;
            sessionStorage.setItem('bankA_user', JSON.stringify(user));

            showMessage(`✓ ${amount.toFixed(2)} TL başar□yla yat□r□ld□!`, 'success');

            setTimeout(() => {
                window.location.href = 'bankA.html';
            }, 2000);
        } else {
            showMessage('❌ İşlem başar□s□z: ' + data.message, 'error');
        }
    })
    .catch(error => {
        console.error('Error:', error);
        showMessage('❌ Bağlant□ hatas□!', 'error');
    });

    return false;
}
```

3.1.2. Soap Layer

The **Soap layer** is the first server-side layer that receives input from the user layer or from external systems. In this project, it has two main entry points:

3.1.2.1. Login Initialize

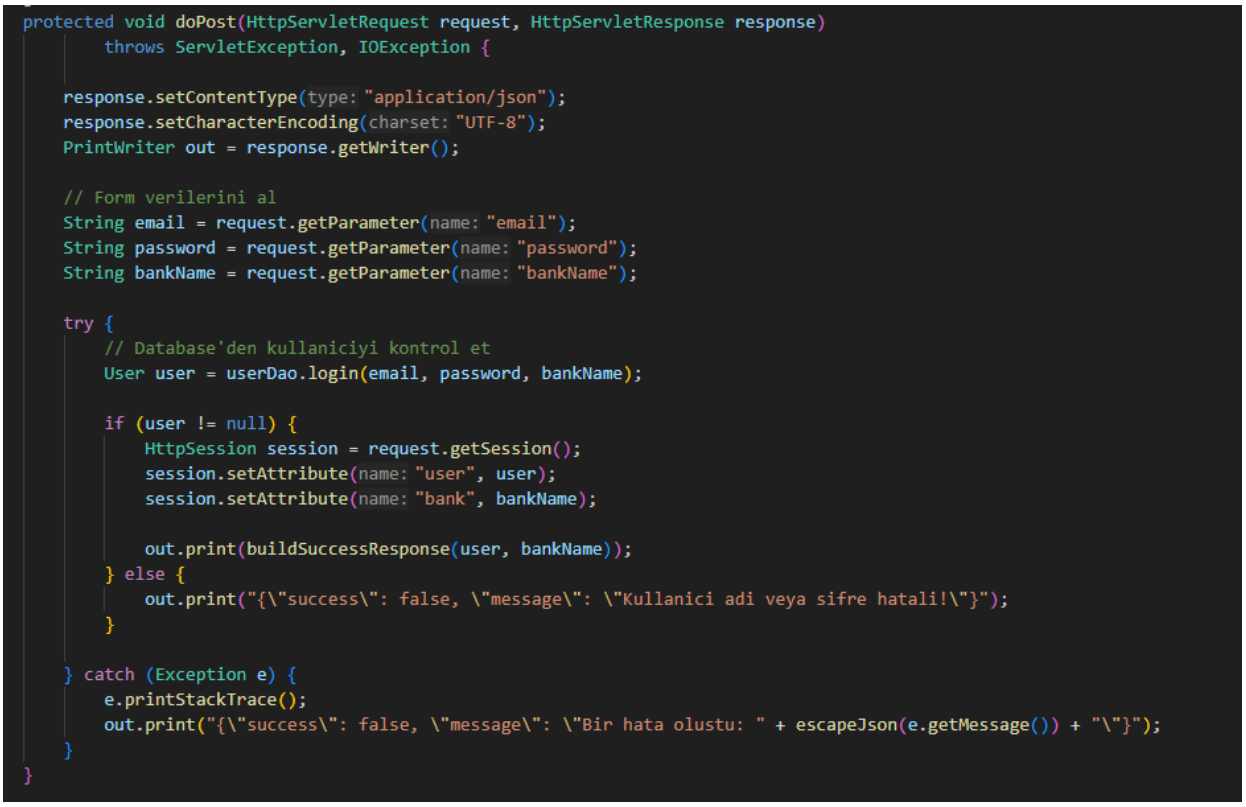
On application startup, LoginServlet.init() runs. It retrieves the Spring ApplicationContext from ServletContext (springContext attribute). It obtains the UserDao bean from the context and stores it in userDao. If the context is missing, it throws a ServletException and logs an error.



```
@Override
public void init() throws ServletException {
    try {
        System.out.println("LoginServlet init() başlat□l□yor...");

        // Global context'ten bean'i al
        ApplicationContext context = (ApplicationContext)
            getServletContext().getAttribute(name: "springContext");

        if (context == null) {
            throw new ServletException(message: "Spring Context bulunamad□!");
        }

        userDao = context.getBean(requiredType: UserDao.class);

        System.out.println("LoginServlet başar□yla başlat□ld□!");
    } catch (Exception e) {
        System.err.println("LoginServlet init() başar□s□z: " + e.getMessage());
        e.printStackTrace();
        throw new ServletException(message: "LoginServlet başlat□lamad□", e);
    }
}
```

3.1.2.2. Login Request Handling

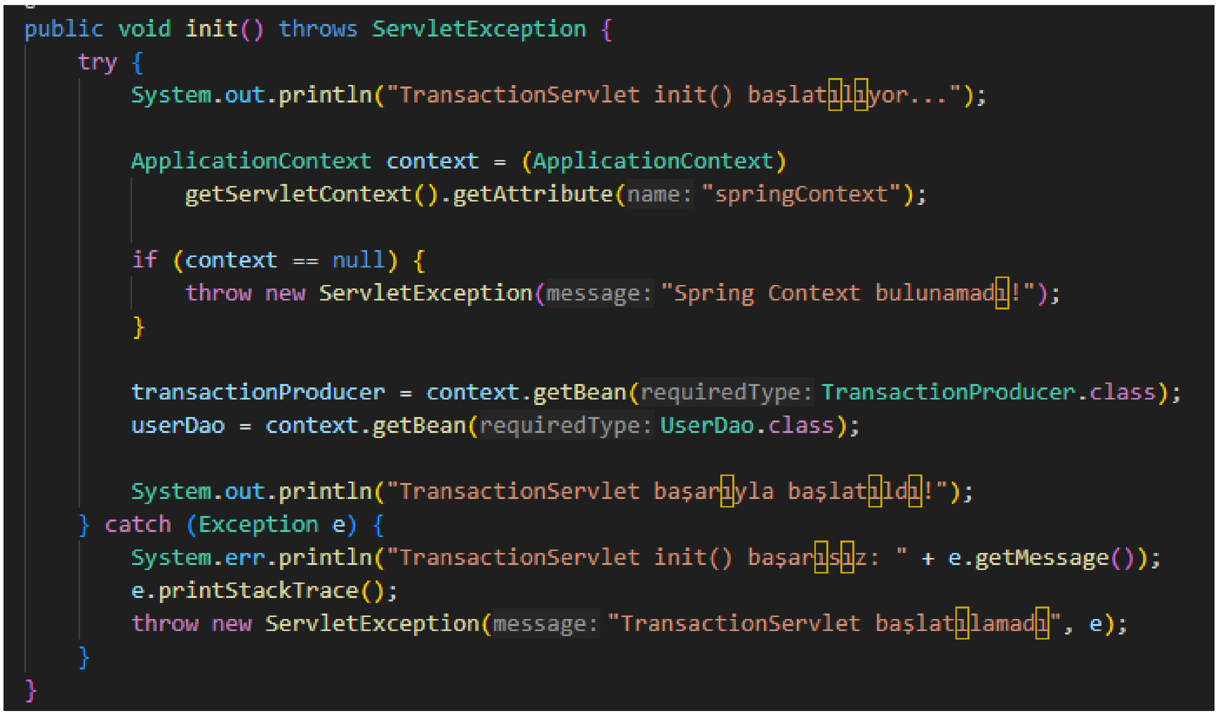
It Sets the response type to JSON (application/json) and UTF-8 encoding. Then Reads form parameters email, password and bankName. Calls userDao.login(email, password, bankName) to execute an HQL query to check the user's credentials and associated bank and if a matching user exists, it returns a User entity; otherwise returns null.



```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    response.setContentType(type: "application/json");
    response.setCharacterEncoding(charset: "UTF-8");
    PrintWriter out = response.getWriter();

    // Form verilerini al
    String email = request.getParameter(name: "email");
    String password = request.getParameter(name: "password");
    String bankName = request.getParameter(name: "bankName");

    try {
        // Database'den kullaniciyi kontrol et
        User user = userDao.login(email, password, bankName);

        if (user != null) {
            HttpSession session = request.getSession();
            session.setAttribute(name: "user", user);
            session.setAttribute(name: "bank", bankName);

            out.print(buildSuccessResponse(user, bankName));
        } else {
            out.print("{\"success\": false, \"message\": \"Kullanici adi veya sifre hatali!\"}");
        }

    } catch (Exception e) {
        e.printStackTrace();
        out.print("{\"success\": false, \"message\": \"Bir hata olustu: " + escapeJson(e.getMessage()) + "\"}");
    }
}
```

3.1.2.3. Transaction Initialize

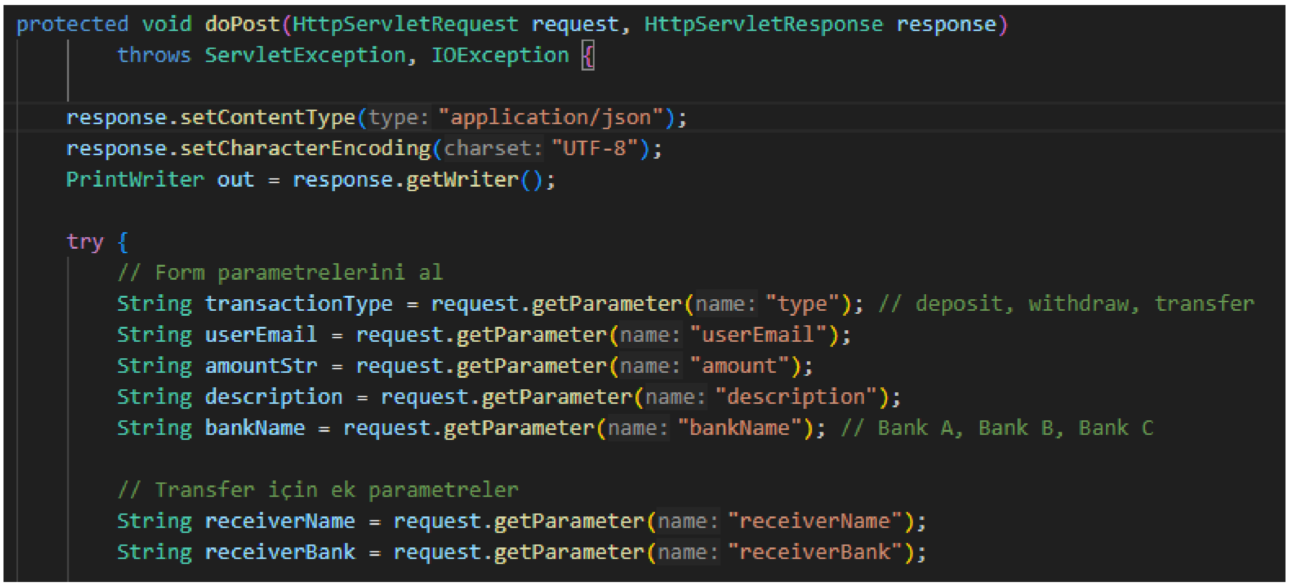
In init(), similar to LoginServlet it gets the Spring ApplicationContext from ServletContext. Then retrieves TransactionProducer and UserDao beans. Logs success or failure of initialization.



```java
public void init() throws ServletException {
    try {
        System.out.println("TransactionServlet init() başlat□l□yor...");

        ApplicationContext context = (ApplicationContext)
            getServletContext().getAttribute(name: "springContext");

        if (context == null) {
            throw new ServletException(message: "Spring Context bulunamad□!");
        }

        transactionProducer = context.getBean(requiredType: TransactionProducer.class);
        userDao = context.getBean(requiredType: UserDao.class);

        System.out.println("TransactionServlet başar□yla başlat□ld□!");
    } catch (Exception e) {
        System.err.println("TransactionServlet init() başar□s□z: " + e.getMessage());
        e.printStackTrace();
        throw new ServletException(message: "TransactionServlet başlat□lamad□", e);
    }
}
```

3.1.2.4. Transaction Request Handling

Sets content type to JSON and UTF-8 encoding; obtains a PrintWriter for the response. Read parameters from the user layer; type – transaction type (deposit, withdraw, transfer), userEmail – the executing user, amount – amount as string, description – optional description, bankName – the bank of the executing user, receiverName, receiverBank – required only for transfer.

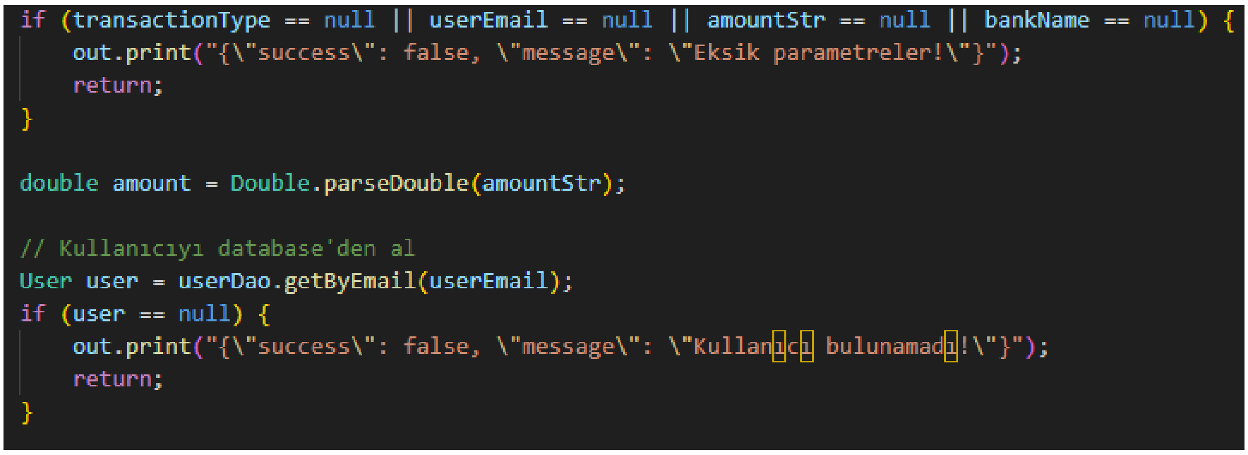


```java
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    response.setContentType(type: "application/json");
    response.setCharacterEncoding(charset: "UTF-8");
    PrintWriter out = response.getWriter();

    try {
        // Form parametrelerini al
        String transactionType = request.getParameter(name: "type"); // deposit, withdraw, transfer
        String userEmail = request.getParameter(name: "userEmail");
        String amountStr = request.getParameter(name: "amount");
        String description = request.getParameter(name: "description");
        String bankName = request.getParameter(name: "bankName"); // Bank A, Bank B, Bank C

        // Transfer için ek parametreler
        String receiverName = request.getParameter(name: "receiverName");
        String receiverBank = request.getParameter(name: "receiverBank");
```

The TransactionServlet therefore acts as a bridge between the synchronous user request and the asynchronous Kafka-based processing pipeline.

3.1.2.5. Transaction Validation and Lookup

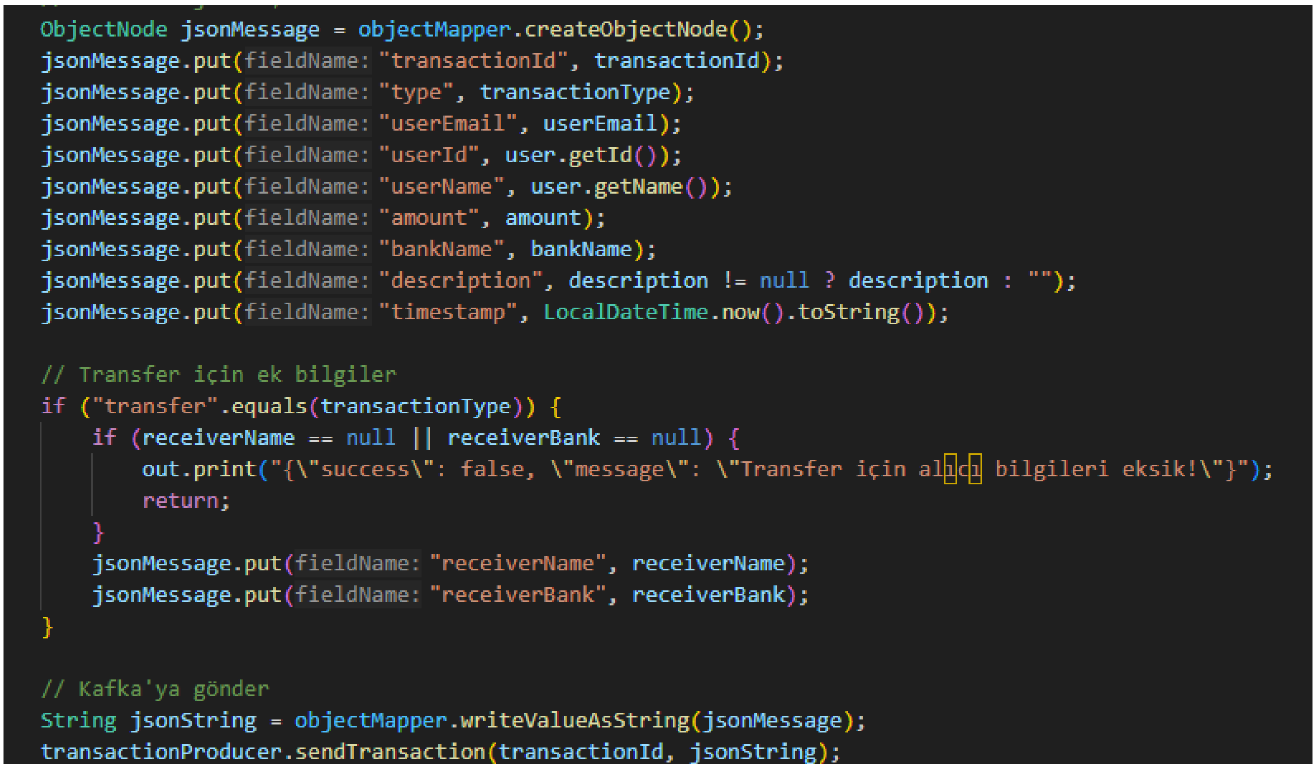
If any mandatory parameter is missing (type, userEmail, amount, bankName), it immediately returns{"success": false, "message": "Eksik parametreler!"}. Then, converts amount to double. If parsing fails, returns {"success": false, "message": "Geçersiz miktar!"}. After that, it calls userDao.getByEmail(userEmail) to fetch the executing user. If the user cannot be found returns {"success": false, "message": "Kullanıcı bulunamadı!"}.



```java
if (transactionType == null || userEmail == null || amountStr == null || bankName == null) {
    out.print("{\"success\": false, \"message\": \"Eksik parametreler!\"}");
    return;
}

double amount = Double.parseDouble(amountStr);

// Kullanıcıyı database'den al
User user = userDao.getByEmail(userEmail);
if (user == null) {
    out.print("{\"success\": false, \"message\": \"Kullan□c□ bulunamad□!\"}");
    return;
}
```

3.1.2.6. Create Json and Asynchronous Dispatch to Kafka

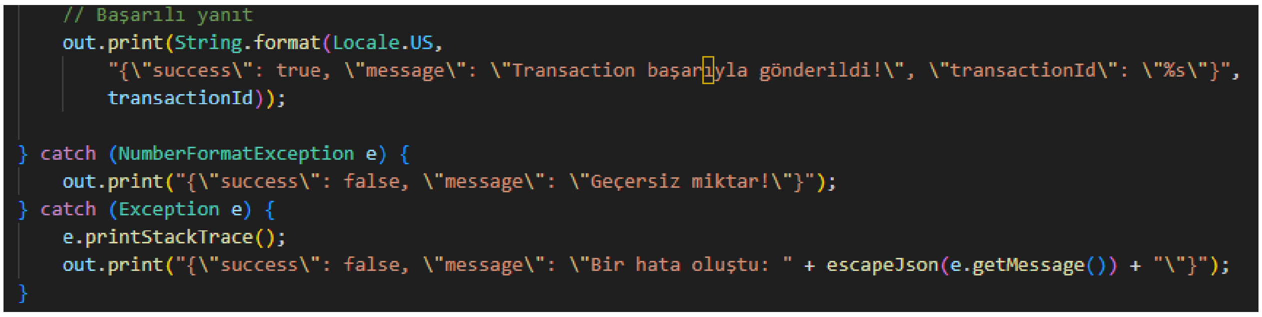
Uses Jackson ObjectMapper and ObjectNode to build a structured JSON containing transactionId, type, userEmail, userId, userName, amount, bankName, description, timestamp. If type is transfer, it also requires and adds receiverName, receiverBank. This JSON is not the UI response; it is the message to be sent to Kafka. After that; it serializes the ObjectNode to a string, jsonString. It calls transactionProducer.sendTransaction(transactionId, jsonString).At this point, the API layer has accepted the request and handed it over to the messaging layer for processing.



```java
ObjectNode jsonMessage = objectMapper.createObjectNode();
jsonMessage.put(fieldName: "transactionId", transactionId);
jsonMessage.put(fieldName: "type", transactionType);
jsonMessage.put(fieldName: "userEmail", userEmail);
jsonMessage.put(fieldName: "userId", user.getId());
jsonMessage.put(fieldName: "userName", user.getName());
jsonMessage.put(fieldName: "amount", amount);
jsonMessage.put(fieldName: "bankName", bankName);
jsonMessage.put(fieldName: "description", description != null ? description : "");
jsonMessage.put(fieldName: "timestamp", LocalDateTime.now().toString());

// Transfer için ek bilgiler
if ("transfer".equals(transactionType)) {
    if (receiverName == null || receiverBank == null) {
        out.print("{\"success\": false, \"message\": \"Transfer için al□c□ bilgileri eksik!\"}");
        return;
    }
    jsonMessage.put(fieldName: "receiverName", receiverName);
    jsonMessage.put(fieldName: "receiverBank", receiverBank);
}

// Kafka'ya gönder
String jsonString = objectMapper.writeValueAsString(jsonMessage);
transactionProducer.sendTransaction(transactionId, jsonString);
```

3.1.2.7. *Response*

Soap Returns a JSON response indicating that the transaction message was successfully sent to the processing backend {"success": true, "message": "Transaction başarıyla gönderildi!", "transactionId": "..."}. Any exceptions are caught; the stack trace is logged and an error response with a safe message is returned.



```
    // Başarılı yanıt
    out.print(String.format(Locale.US,
        "{\"success\": true, \"message\": \"Transaction başarıyla gönderildi!\", \"transactionId\": \"%s\"}",
        transactionId));

} catch (NumberFormatException e) {
    out.print("{\"success\": false, \"message\": \"Geçersiz miktar!\"}");
} catch (Exception e) {
    e.printStackTrace();
    out.print("{\"success\": false, \"message\": \"Bir hata oluştu: " + escapeJson(e.getMessage()) + "\"}");
}
```

3.2. Messaging (Kafka) Layer

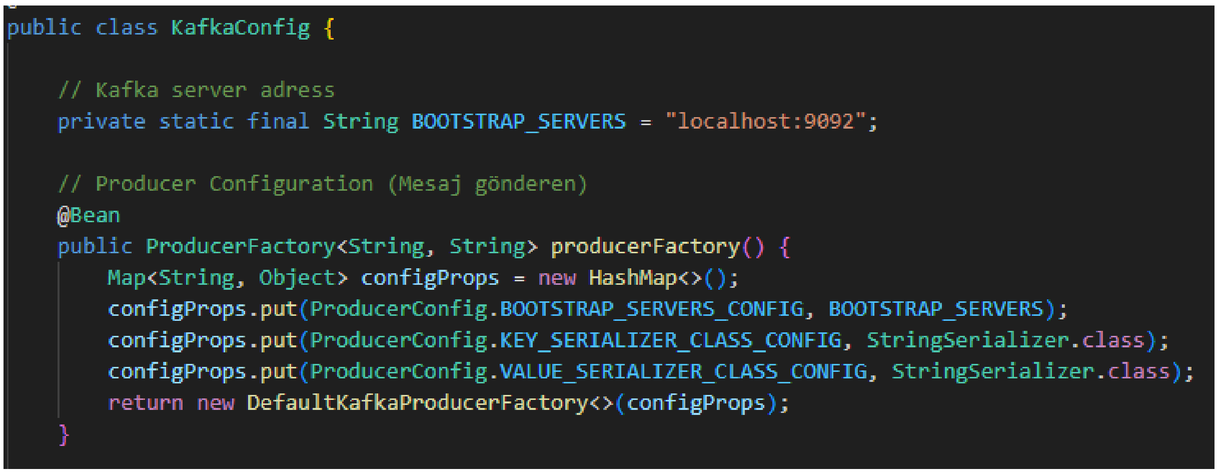
The Kafka layer is the **asynchronous backbone** of the system. It decouples the API layer from the business logic that updates balances and writes transaction logs. Instead of directly updating the database during an HTTP/SOAP request, the API layer publishes a message to Kafka; the Kafka layer then ensures reliable delivery to a consumer that performs the actual processing.

The Kafka layer in this project consists of:

- Configuration classes (KafkaConfig, KafkaTopicConfig)
- A **producer** service (TransactionProducer)
- A **consumer** component (TransactionConsumer)
- A **topic** named payment-transactions

3.2.1. Kafka Configuration

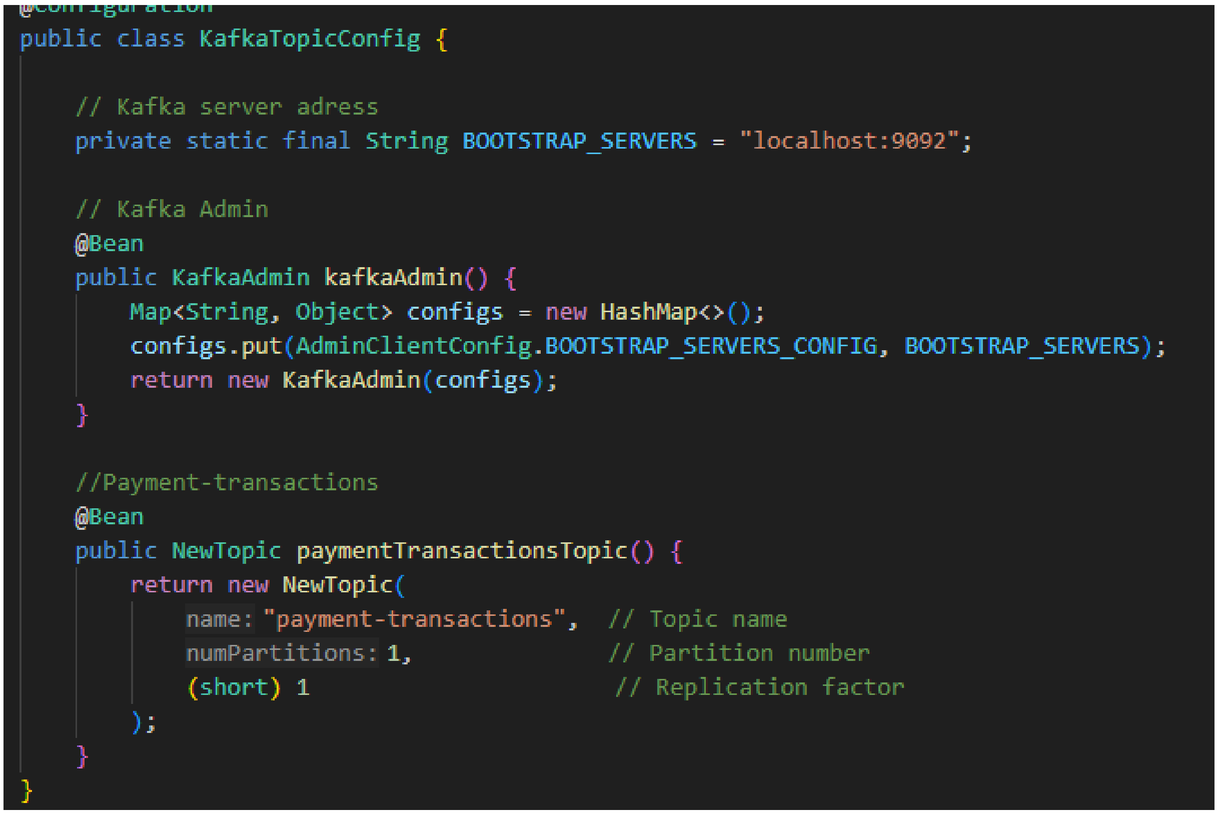
KafkaConfig is annotated with Configuration and EnableKafka. It defines all necessary beans to interact with Kafka.



```
public class KafkaConfig {

    // Kafka server adress
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";

    // Producer Configuration (Mesaj gönderen)
    @Bean
    public ProducerFactory<String, String> producerFactory() {
        Map<String, Object> configProps = new HashMap<>();
        configProps.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configProps.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configProps.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        return new DefaultKafkaProducerFactory<>(configProps);
    }
```

3.2.2. Topic Management

KafkaTopicConfig handles administrative configuration of Kafka topics: KafkaAdmin bean; kafkaAdmin() creates a KafkaAdmin configured with the same BOOTSTRAP_SERVERS, then this admin client allows the application to declare topics at startup. Topic definition; paymentTransactionsTopic() returns a NewTopic, Name: payment-transactions, Partitions: 1, Replication factor: 1. When the application starts, Spring uses KafkaAdmin to ensure the payment-transactions topic exists (creating it if necessary). This automatic topic creation simplifies deployment and ensures that the producer and consumer always have a consistent topic to work with.



```java
@Configuration
public class KafkaTopicConfig {

    // Kafka server adress
    private static final String BOOTSTRAP_SERVERS = "localhost:9092";

    // Kafka Admin
    @Bean
    public KafkaAdmin kafkaAdmin() {
        Map<String, Object> configs = new HashMap<>();
        configs.put(AdminClientConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        return new KafkaAdmin(configs);
    }

    //Payment-transactions
    @Bean
    public NewTopic paymentTransactionsTopic() {
        return new NewTopic(
            name: "payment-transactions",  // Topic name
            numPartitions: 1,              // Partition number
            (short) 1                       // Replication factor
        );
    }
}
```

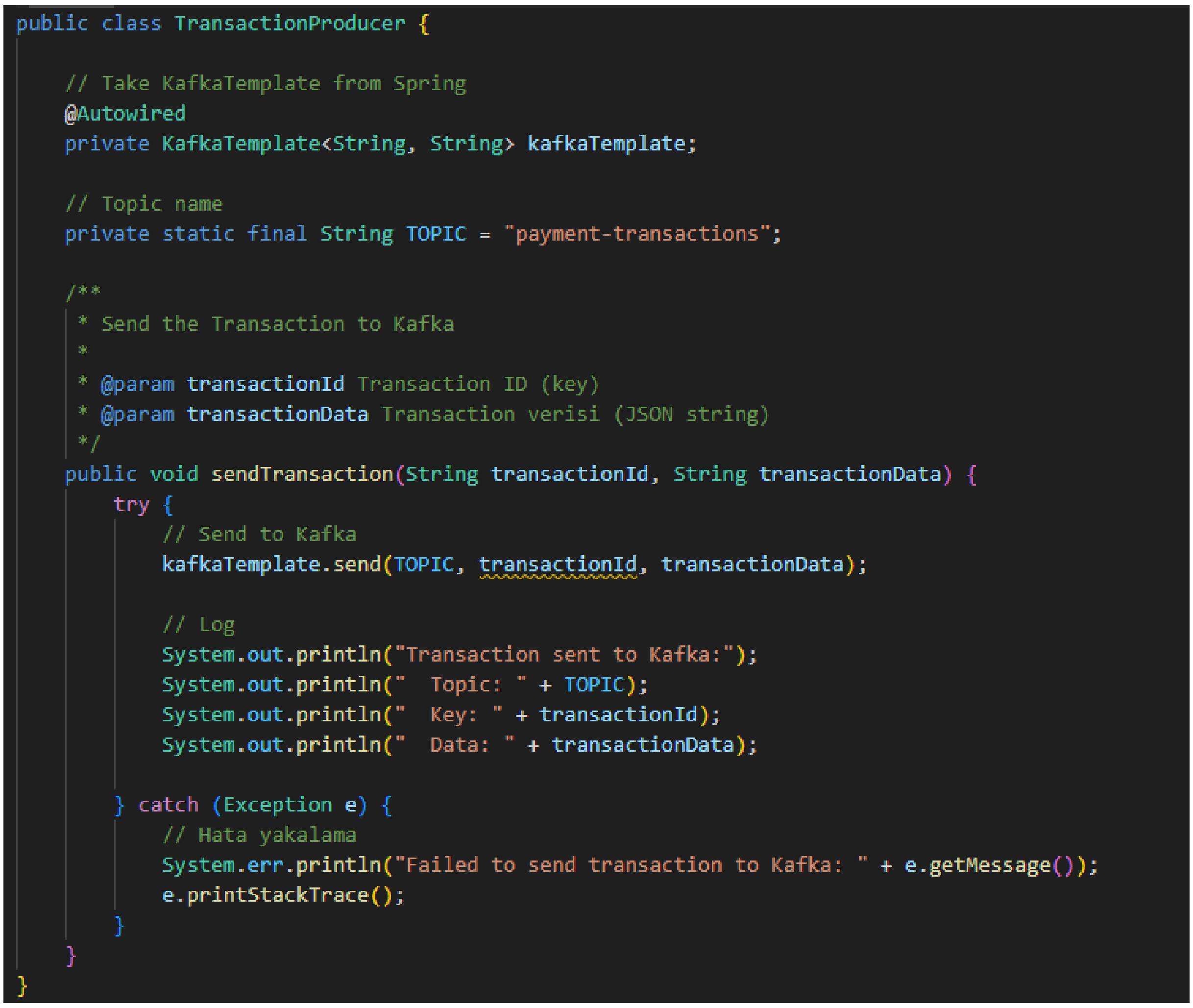
3.2.3. Message Production

TransactionProducer is a Spring Service that acts as the single entry point for publishing transaction data to Kafka.

It’s dependencies and configuration injects KafkaTemplate<String, String> via Autowired. Then, it defines a constant topic name TOPIC = "payment-transactions". It’s core method: sendTransaction(String transactionId, String transactionData) and it is called by TransactionServlet (HTTP/JSON API) & PaymentEndpoint (SOAP API).

It uses kafkaTemplate.send(TOPIC, transactionId, transactionData) as transactionId is used as the message key so that all events for the same transaction go to the same partition or transactionData is a JSON string containing all business fields (user, amount, bank, type, etc.). Then it logs the action to the console, printing the topic, key, and payload.

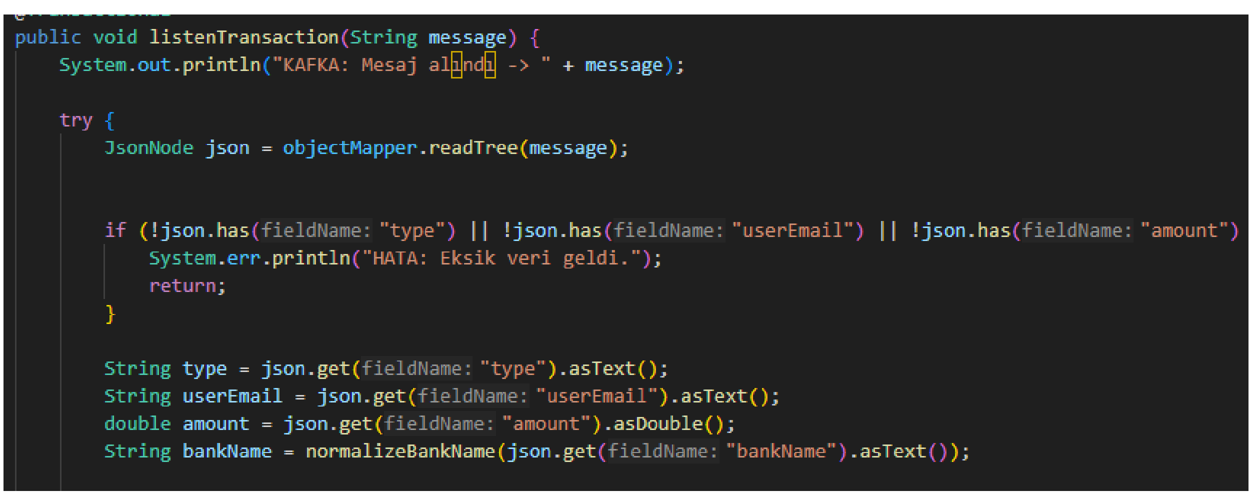
Lastly, it catches and logs any exceptions if sending fails (System.err and stack trace). This producer centralizes all Kafka writing operations, enforcing a consistent topic, key, and data format for every transaction message.



```java
public class TransactionProducer {

    // Take KafkaTemplate from Spring
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    // Topic name
    private static final String TOPIC = "payment-transactions";

    /**
     * Send the Transaction to Kafka
     *
     * @param transactionId Transaction ID (key)
     * @param transactionData Transaction verisi (JSON string)
     */
    public void sendTransaction(String transactionId, String transactionData) {
        try {
            // Send to Kafka
            kafkaTemplate.send(TOPIC, transactionId, transactionData);

            // Log
            System.out.println("Transaction sent to Kafka:");
            System.out.println("  Topic: " + TOPIC);
            System.out.println("  Key: " + transactionId);
            System.out.println("  Data: " + transactionData);

        } catch (Exception e) {
            // Hata yakalama
            System.err.println("Failed to send transaction to Kafka: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

3.2.4. *Message Consumption and Processing*

TransactionConsumer is a Spring Component responsible for reading messages from Kafka and executing the core business logic. Dependencies inject; UserDao, BankATransactionDao, BankBTransactionDao, BankCTransactionDao. It uses,Jackson ObjectMapper and JsonNode to parse incoming JSON payloads. These dependencies allow it to read user information and write transaction logs to different tables depending on the bank.



```java
public void listenTransaction(String message) {
    System.out.println("KAFKA: Mesaj alındı -> " + message);

    try {
        JsonNode json = objectMapper.readTree(message);


        if (!json.has(fieldName: "type") || !json.has(fieldName: "userEmail") || !json.has(fieldName: "amount")
            System.err.println("HATA: Eksik veri geldi.");
            return;
        }

        String type = json.get(fieldName: "type").asText();
        String userEmail = json.get(fieldName: "userEmail").asText();
        double amount = json.get(fieldName: "amount").asDouble();
        String bankName = normalizeBankName(json.get(fieldName: "bankName").asText());
```

3.3. Database Layer

The database layer is responsible for **persisting all business data**: users, banks, and transactions. It is built with **Hibernate ORM** and **Jakarta Persistence** on top of **MySQL** and is fully integrated with Spring's transaction management.

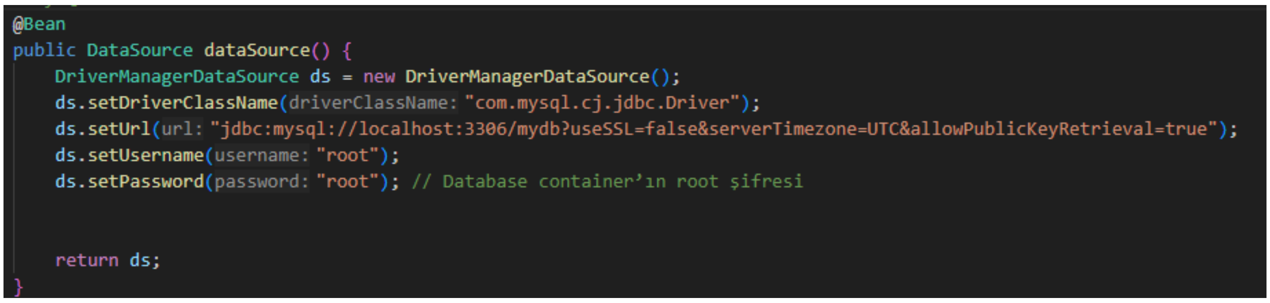
The main components are: Hibernate and data source configuration (HibernateConfig), JPA entity classes under Database.Entities (e.g. User, Transaction, bank transaction entities), DAO (Data Access Object) classes under Database.DAO (e.g. UserDao, BankATransactionDao, etc.), Transaction management using HibernateTransactionManager and Transactional.

3.3.1. Configuration

HibernateConfig is annotated with Configuration, EnableTransactionManagement, and @ComponentScan(basePackages = {"Database", "Message.Kafka"}).

3.3.1.1. Data Source

dataSource() method creates a DriverManagerDataSource with Driver: com.mysql.cj.jdbc.Driver URL: jdbc:mysql://localhost:3306/mydb?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true. Username: root, Password: root. This bean defines how the application connects to the MySQL database.

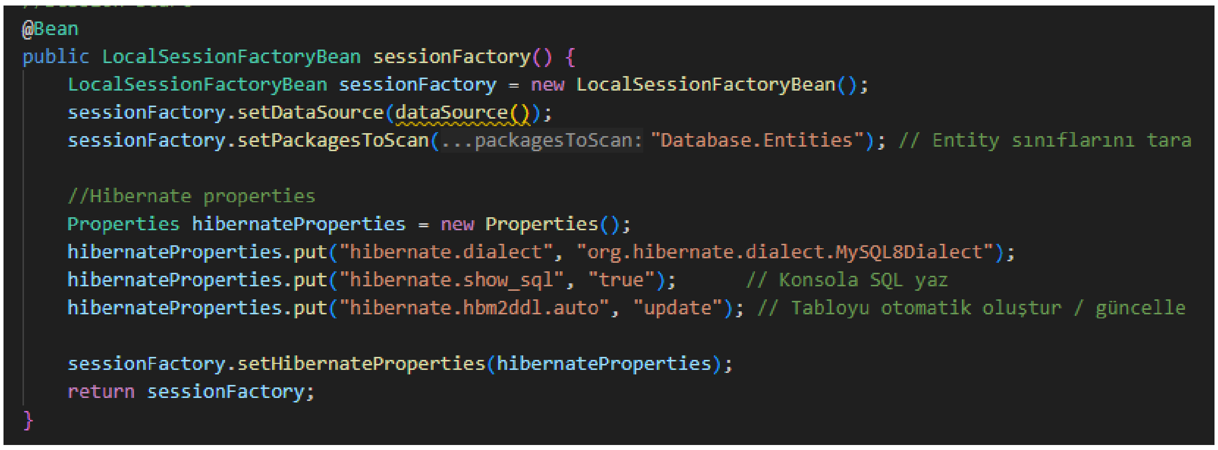


```java
@Bean
public DataSource dataSource() {
    DriverManagerDataSource ds = new DriverManagerDataSource();
    ds.setDriverClassName(driverClassName: "com.mysql.cj.jdbc.Driver");
    ds.setUrl(url: "jdbc:mysql://localhost:3306/mydb?useSSL=false&serverTimezone=UTC&allowPublicKeyRetrieval=true");
    ds.setUsername(username: "root");
    ds.setPassword(password: "root"); // Database container'ın root şifresi


    return ds;
}
```

3.3.1.2. Session Factory

sessionFactory() returns a LocalSessionFactoryBean configured with dataSource from above setPackagesToScan("Database.Entities") – automatically discovers all entity classes in that package. Hibernate properties hibernate.dialect = org.hibernate.dialect.MySQL8Dialect, hibernate.show_sql = true (prints SQL statements to the console),hibernate.hbm2ddl.auto = update (creates/updates tables automatically based on entity mappings).

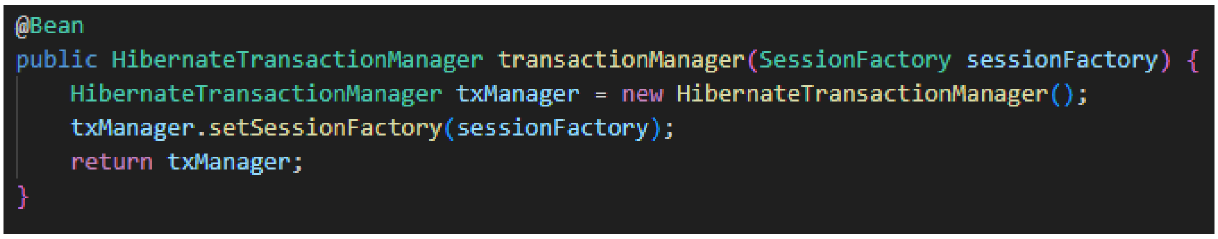
This configuration lets Hibernate convert entity operations (save, update, query) into SQL statements for MySQL.



```java
@Bean
public LocalSessionFactoryBean sessionFactory() {
    LocalSessionFactoryBean sessionFactory = new LocalSessionFactoryBean();
    sessionFactory.setDataSource(dataSource());
    sessionFactory.setPackagesToScan(...packagesToScan: "Database.Entities"); // Entity sınıflarını tara

    //Hibernate properties
    Properties hibernateProperties = new Properties();
    hibernateProperties.put("hibernate.dialect", "org.hibernate.dialect.MySQL8Dialect");
    hibernateProperties.put("hibernate.show_sql", "true");      // Konsola SQL yaz
    hibernateProperties.put("hibernate.hbm2ddl.auto", "update"); // Tabloyu otomatik oluştur / güncelle

    sessionFactory.setHibernateProperties(hibernateProperties);
    return sessionFactory;
}
```

3.3.1.3. Transaction Manager

transactionManager(SessionFactory sessionFactory) returns a HibernateTransactionManager. Spring uses this manager to start, commit, and roll back database transactions.Methods annotated with Transactional (in DAOs or services) run inside these managed transactions.



```java
@Bean
public HibernateTransactionManager transactionManager(SessionFactory sessionFactory) {
    HibernateTransactionManager txManager = new HibernateTransactionManager();
    txManager.setSessionFactory(sessionFactory);
    return txManager;
}
```

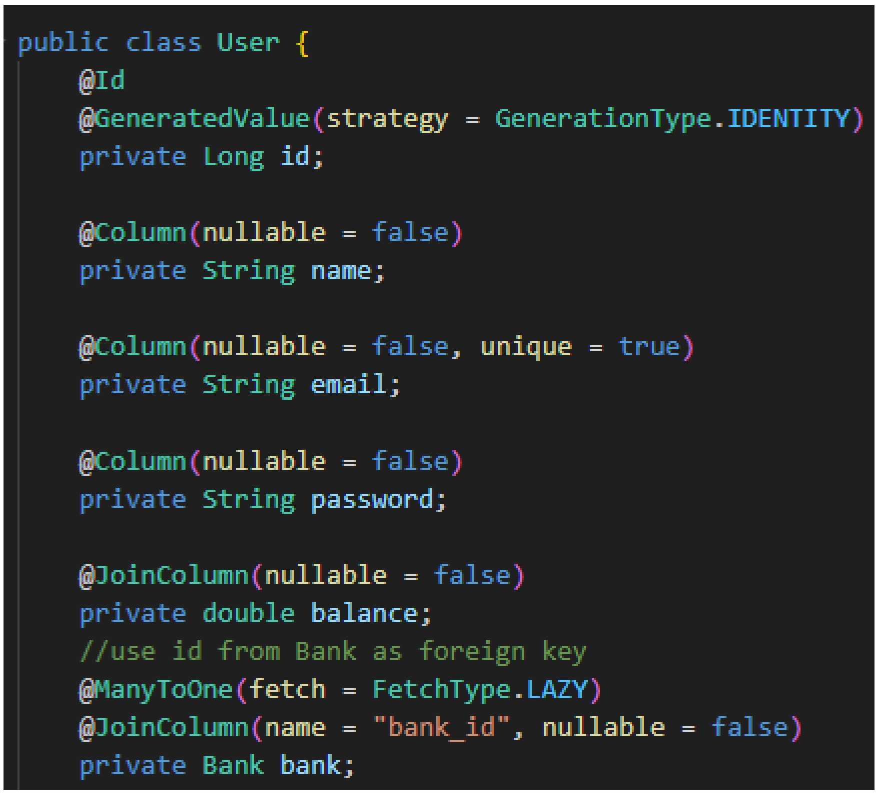
3.3.2. Entity Model

The entity classes represent database tables and define relationships between them. Hibernate maps these classes to tables using JPA annotations.

3.3.2.1. User Entity

Annotated with Entity and @Table(name = "users").

Main fields: id – primary key, Id, GeneratedValue(strategy = GenerationType.IDENTITY), name – required (@Column(nullable = false)), email – required and unique (@Column(nullable = false, unique = true)), password – required, balance – current account balance, bank – many-to-one relation to a Bank entity; @ManyToOne(fetch = FetchType.LAZY), @JoinColumn(name = "bank_id", nullable = false)



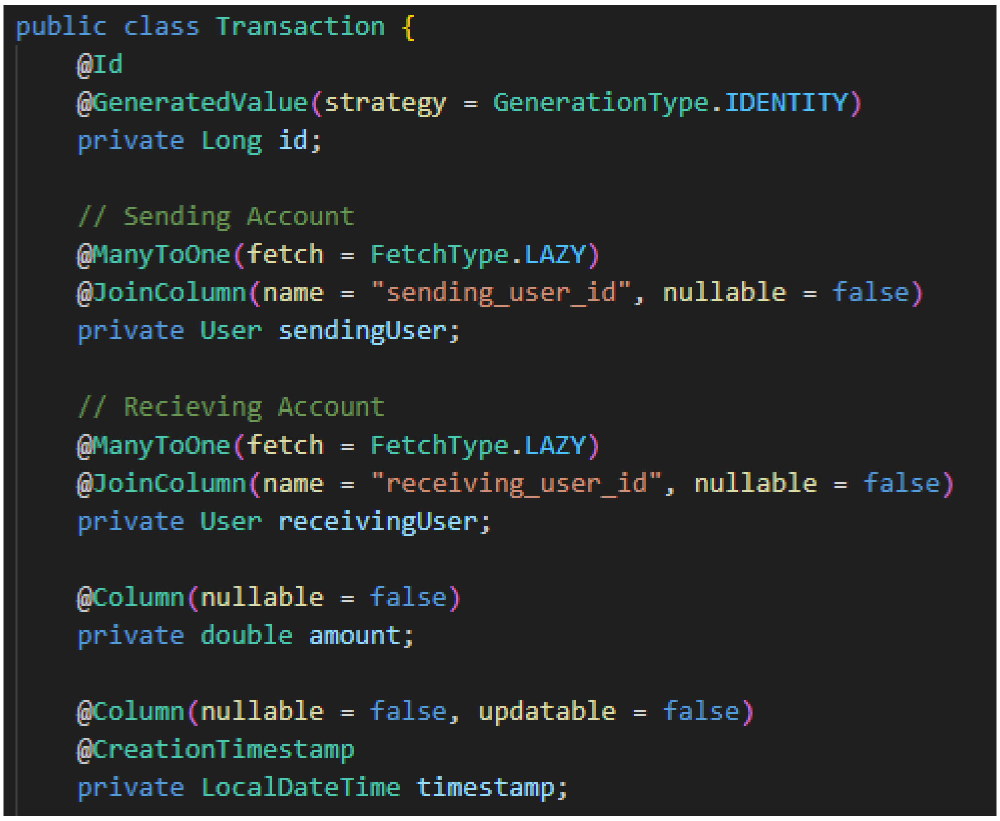
```java
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @JoinColumn(nullable = false)
    private double balance;
    //use id from Bank as foreign key
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "bank_id", nullable = false)
    private Bank bank;
```

This entity models a customer of a particular bank with a single account and balance.

3.3.2.2. Transaction Entity

Annotated with Entity and @Table(name = "transactions").

Fields: id – primary key, auto-generated, sendingUser – many-to-one relationship to the User who sends money; @ManyToOne(fetch = FetchType.LAZY), @JoinColumn(name = "sending_user_id", nullable = false), receivingUser – many-to-one relationship to the User who receives money @JoinColumn(name = "receiving_user_id", nullable = false), amount – transferred amount, timestamp – CreationTimestamp, and @Column(nullable = false, updatable = false). Automatically filled with the current time when the row is inserted.



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

    // Sending Account
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "sending_user_id", nullable = false)
    private User sendingUser;

    // Recieving Account
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "receiving_user_id", nullable = false)
    private User receivingUser;

    @Column(nullable = false)
    private double amount;

    @Column(nullable = false, updatable = false)
    @CreationTimestamp
    private LocalDateTime timestamp;
```

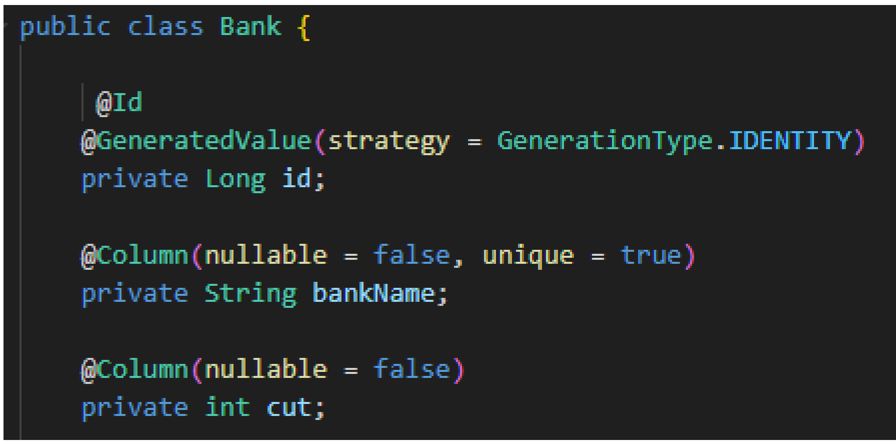
This entity records **inter-user transfers**, allowing the system to track who sent money to whom, when, and how much.

3.3.2.3. Bank Entity

Additional entities such as BankATransaction, BankBTransaction, and BankCTransaction (not fully shown here) store logs per bank.

Each typically contains: Reference to the User or user ID, Amount, Operation type (DEPOSIT, WITHDRAW, TRANSFER), Timestamp

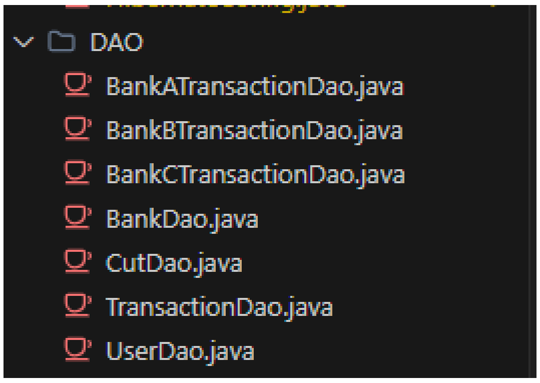
These entities are used by the Kafka consumer to write detailed transaction histories for each bank, enabling reporting and auditing.



```java
public class Bank {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true)
    private String bankName;

    @Column(nullable = false)
    private int cut;
```

3.3.3. Data Access Layer

DAO classes provide a clean, object-oriented interface for performing database operations. They hide the underlying Hibernate API from the upper layers.



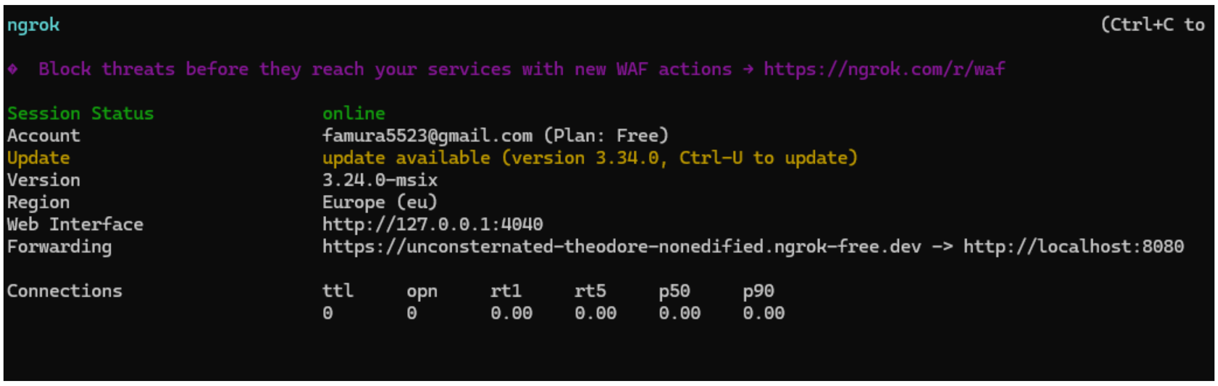
```
DAO
  BankATransactionDao.java
  BankBTransactionDao.java
  BankCTransactionDao.java
  BankDao.java
  CutDao.java
  TransactionDao.java
  UserDao.java
```

All DAOs are annotated with; Repository (for Spring's component scanning and exception translation), Transactional to ensure that operations run inside a transaction and inject a shared SessionFactory (Autowired) and obtain the current Session with sessionFactory.getCurrentSession().

This design centralizes transaction handling in Spring instead of manual begin/commit calls.

3.4. Web Layer

Lastly, everything is given to Ngrok so that it can take what posted on the localhost:8080 and reflect it onto https://unconsternated-theodore-nonedified.ngrok-free.dev.



```
ngrok                                                                                         (Ctrl+C to

◆ Block threats before they reach your services with new WAF actions → https://ngrok.com/r/waf

Session Status                online
Account                       famura5523@gmail.com (Plan: Free)
Update                        update available (version 3.34.0, Ctrl-U to update)
Version                       3.24.0-msix
Region                        Europe (eu)
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://unconsternated-theodore-nonedified.ngrok-free.dev -> http://localhost:8080

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

4. Project Analysis

4.1. Test

The tests were done on Windows computers and Android phones. The system has been tested locally, public, public by phone and every time it works perfectly. It always responded around ~100 ms which is perfectly fine on a project like this. Only downside is since we use free version of ngrok, only one person can connect at one time.

4.2. Upgradable Things

Firstly, a cut structure was designed and was meant to be added to the project. Which meant to work like, if any interbank transaction includes Bank C, cut %8 from the money.

Also, the project is vulnerable to some cyber-attacks. Since it was not meant to be used as real system, it was only developed for simple usage. It can be still further developed with help of some cyber-security crew and can be used in real life. No held backs there.

Lastly, sometimes Kafka can crash on launch. No idea why, but it can be restarted and the system works perfectly.

5. Project Comments

5.1. Possible Usages

While this project is not usable alone, its structure can be used on other projects or maybe as a website structure for future projects.

5.1.1. To-Do's When Using

Do not forget that only one person can use it.

5.2. Creator Comments

This project was originally a simple homework assignment. But I made it more; more complicated, more detailed, more time consuming, more challenging. But this project taught me a lot about being a leader and a lot about writing Java code with Maven. This pushed me for League of Legends Manager game and website project. – Famura

6. Credits

Famura – API Layer & Web Layer

Aybüke Yaman – Messaging Layer

Tuna Münir Telatar – Database Layer

7. References

- Apache Kafka documentation. (2024). Apache Kafka: A distributed streaming platform. Apache Software Foundation. https://kafka.apache.org/documentation/
- Docker Inc. (2024). Docker documentation. https://docs.docker.com/
- Hibernate ORM documentation. (2024). Hibernate ORM user guide. Red Hat.
- https://hibernate.org/orm/documentation/
- Jackson Project. (2024). Jackson databind documentation. FasterXML. https://github.com/FasterXML/jackson-databind
- Jetty Project. (2024). Eclipse Jetty documentation. Eclipse Foundation. https://www.eclipse.org/jetty/documentation/
- MySQL documentation. (2024). MySQL 8.0 reference manual. Oracle Corporation. https://dev.mysql.com/doc/
- Spring Framework documentation. (2024). Spring Framework reference documentation. VMware, Inc. https://docs.spring.io/spring-framework/docs/current/reference/html/
- Spring Web Services. (2024). Spring Web Services reference documentation. VMware, Inc. https://docs.spring.io/spring-ws/docs/current/reference/html/
- W3C. (2007). SOAP version 1.2 part 1: Messaging framework (Second edition). World Wide Web Consortium. https://www.w3.org/TR/soap12/
- Ngrok documentation. (2024). Ngrok secure tunneling documentation. https://ngrok.com/docs
- Oracle. (2024). Java Platform, Standard Edition documentation. https://docs.oracle.com/en/java/

8. Disk Contents

- Project source code (whole Java Project in .rar) contains this report on .docx format
- This report in PDF format
- Demo Video
- Money for Nothing by Dire Straits