TED UNIVERSITY

PROJE RAPORU

MAVI PROJE

TAMAMLANDI

Interbank Payment System

Yazar(lar):

Famura

İçerikler

1. Proje Tanımı

1.1. Giriş

Interbank Payment Service projesi, farklı bankalara ait kullanıcılar arasında **para yatırma (deposit)**, **para çekme (withdraw)** ve **para transferi (transfer)** işlemlerini gerçekleştirebilen, basitleştirilmiş bir **bankalar arası işlem platformunu** simüle eder.

Projenin temel amaçları şunlardır:

- Kurumsal Java uygulamalarında yaygın olarak kullanılan farklı teknolojileri entegre etmek (Spring, Hibernate, SOAP, Kafka, Jetty, MySQL)
- Frontend / API katmanları ile backend işlem katmanı arasında **Apache Kafka** kullanarak güvenilir ve asenkron iletişimi göstermek
- Kullanıcı ve işlem verilerini **ilişkisel bir veritabanında**, tutarlı ve transactional bir şekilde saklamak

Sistem; bankaların ve kullanıcıların merkezi bir servis üzerinden etkileşime girdiği bir senaryoyu modeller. Bu servis ödeme taleplerini kabul eder, mesaj olarak yayınlar ve bakiyeleri güncellemek ve işlem kayıtlarını tutmak için bu mesajları işler.

1.2. Proje Hedefleri

- **Hedef 1 – Bankalar Arası İşlem Simülasyonu:**
  Farklı bankalara ait olabilen kullanıcılar arasında temel bankacılık işlemlerini (deposit, withdraw, transfer) gerçekleştirmek.
- **Hedef 2 – Mesaj Tabanlı Mimari:**
  API katmanı ile iş mantığı arasında **Apache Kafka** kullanarak mesaj aracılı bir yapı kurmak.
- **Hedef 3 – Kurumsal Java Teknoloji Yığını:**
  Uygulamayı **Spring Boot kullanmadan**, aşağıdaki teknolojilerle geliştirmek:
  - Spring Core / Spring MVC / Spring WS
  - Hibernate ORM with MySQL
  - Embedded servlet container olarak Jetty
  - SOAP web servisleri
- **Hedef 4 – Tutarlılık ve Transaction Yönetimi:**
  Bakiye güncellemeleri ve işlem kayıtlarının **veritabanı transaction'ları** içerisinde çalışmasını sağlayarak tutarsız veri riskini en aza indirmek.
- **Hedef 5 – Genişletilebilirlik:**
  Kod yapısını modüler olacak şekilde tasarlayarak (API, messaging, database paketleri ayrı) yeni bankaların veya işlemlerin kolayca eklenebilmesini sağlamak.

2. Proje Teknolojileri

2.1. Modül(ler)

- **Windows (Windows 10):**
  Projenin tamamı Windows sistemler için tasarlanmıştır. Windows 7 gibi daha eski sürümlerde test edilmemiştir. Windows 11 üzerinde test edilmiş ve sorunsuz çalışmaktadır.

- **Android(Android 13+):**
  Proje mobil öncelikli olarak geliştirilmemiştir. Ancak SOAP API kullanıldığı için mobil bağlantılar otomatik olarak desteklenmektedir. Android üzerinde test edilmiştir. iOS üzerinde test edilmemiştir (muhtemelen çalışacaktır ancak test edilmediği için kesinlik yoktur).

2.2. Programlama Dilleri

- **Java (Java 21 / Maven - Spring) – IntelliJ IDEA Community Edition:**
  Projenin backend tarafı Java 21 kullanılarak geliştirilmiştir. Spring Framework kullanılmıştır (Spring Boot **kullanılmamıştır**). Derleme Maven üzerinden yapılmaktadır. Bu yapı proje gereksinimlerinden biridir. IDE olarak Git araçlarıyla uyumlu olduğu için IntelliJ IDEA tercih edilmiştir.

- **HTML/CSS /JavaScript - IntelliJ IDEA Community Edition / Opera GX & Microsoft Edge:**
  Client-side arayüz; HTML (yapı), CSS (stil) ve JavaScript (dinamik davranış) kullanılarak geliştirilmiştir. IntelliJ IDEA kod düzenleme için, Opera GX ve Microsoft Edge ise test ve debug amacıyla kullanılmıştır (makinede mevcut olan tarayıcılar oldukları için).

2.3. Kütüphaneler & Harici Eklentiler

2.3.1. Kullanılan Sistemler

- **Soap Api:**
  XML tabanlı, HTTP/HTTPS üzerinden çalışan, standart ve güvenli veri alışverişi sağlayan bir web servis protokolüdür.

- **Apache Kafka:**
  Gerçek zamanlı veri akışları ve asenkron mesajlaşma için kullanılan dağıtık bir event-streaming platformudur.

- **Hibernate:**
  Java nesneleri ile ilişkisel veritabanı tabloları arasında eşleme yapan bir ORM framework'üdür.

- **Docker:**
  Uygulamaların ve bağımlılıklarının izole container'lar içinde çalışmasını sağlar. Bu projede MySQL için kullanılmıştır.

- MySQL:
  İlişkisel veritabanı yönetim sistemidir. Yapılandırılmış verilerin güvenilir şekilde saklanmasını sağlar.

2.3.2. Diğerleri

- **Jetty 11:**
  Java tabanlı web uygulamaları çalıştırmak için kullanılan hafif ve yüksek performanslı bir servlet container'dır.

- **Jackson Databind:**
  Java nesneleri ile JSON verileri arasında dönüşüm sağlar.

- **SLF4J Simple:**
  Uygulama loglarını üretmek için kullanılan logging framework'üdür.

- **Spring Context, Spring Web, Spring Web MVC:**
  Dependency injection, MVC tabanlı web geliştirme ve request handling için kullanılmıştır.

- **Ngrok:**
  Local server'ı internet üzerinden erişilebilir hale getiren tunneling aracıdır. Ücretsiz sürüm kullanılmıştır.

3. Proje Mimarisi

Proje üç ana katmana ayrılmıştır. Bu **katmanlı mimari**, kullanıcı etkileşimi, iş mantığı ve veri saklama sorumluluklarını ayırarak bakım ve ölçeklenebilirliği artırır.

3.1. API Layer

Bu katman SOAP API kullanılarak geliştirilmiştir.

Alt katmanlar:

- **User Layer (UI – HTML/CSS/JS)**
- **SOAP Layer**

User Layer kullanıcıdan gelen talepleri XML formatında alır ve SOAP Layer'a gönderir. SOAP Layer bu talepleri JSON'a çevirerek Kafka'ya publish eder.

3.1.1. User Layer

3.1.1.1. Login

Kullanıcı bankacılık arayüzünü açar ve login olur. LoginServlet ve index.html kullanılır. Login servlet ile ilgili bilgi 3.1.2.1. Login başlığında bulunabilir.

Ana sayfada kullanıcı banka seçimi yapar ve ilgili bankanın HTML sayfasına yönlendirilir (raporda basitlik adına Bank A referans alınmıştır).

```html
<!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';
```

Login bilgileri login.js ile XML formatına çevrilir.

```
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';
    });
}
```

Eğer kullanıcı başarıyla giriş yaparsa bütün operasyonlara erişim sağlayabilir.

3.1.1.2. İşlem Gönderimi

Giriş başarılıysa kullanıcı işlem yapabilir. Gerekli parametreler doldurulur ve ilgili API endpoint'lerine gönderilir.

```
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. Yanıt

UI, backend'den gelen yanıtı bekler ve kullanıcıya gösterir.

```
    .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 başarıyla yatırıldı!`, 'success');

            setTimeout(() => {
                window.location.href = 'bankA.html';
            }, 2000);
        } else {
            showMessage('❌ İşlem başarısız: ' + data.message, 'error');
        }
    })
    .catch(error => {
        console.error('Error:', error);
        showMessage('❌ Bağlantı hatası!', 'error');
    });

    return false;
}
```

3.1.2. Soap Layer

SOAP Layer, kullanıcıdan veya harici sistemlerden gelen istekleri alan **ilk server-side katmandır**.

3.1.2.1. Login Initialize

Uygulama başlatıldığında LoginServlet.init() çalışır. ServletContext içerisinden Spring ApplicationContext (springContext attribute) alınır. Ardından UserDao bean'i elde edilir ve userDao değişkenine atanır. Context bulunamazsa ServletException fırlatılır ve hata loglanır.

```
@Override
public void init() throws ServletException {
    try {
        System.out.println("LoginServlet init() başlatılıyor...");

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

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

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

        System.out.println("LoginServlet başarıyla başlatıldı!");
    } catch (Exception e) {
        System.err.println("LoginServlet init() başarısız: " + e.getMessage());
        e.printStackTrace();
        throw new ServletException(message: "LoginServlet başlatılamadı", e);
    }
}
```

3.1.2.2. Login Request Handling

Response tipi JSON (application/json) ve UTF-8 encoding olarak ayarlanır. Ardından email, password ve bankName parametreleri okunur. userDao.login(email, password, bankName) çağrılarak HQL sorgusu çalıştırılır. Eşleşen bir kullanıcı bulunursa User entity'si döndürülür, aksi halde null döner.

```
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

init() metodunda, LoginServlet'te olduğu gibi ApplicationContext alınır. Ardından TransactionProducer ve UserDao bean'leri elde edilir. Başlatma başarılı veya başarısız ise loglanır.

```
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

Response tipi JSON ve UTF-8 olarak ayarlanır ve response için bir PrintWriter alınır. User Layer'dan gelen parametreler okunur: type – işlem türü (deposit, withdraw, transfer) / userEmail – işlemi yapan kullanıcı / amount – string olarak miktar / description – opsiyonel açıklama / bankName – işlemi yapan kullanıcının bankası / receiverName, receiverBank – yalnızca transfer işlemleri için zorunludur

```
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");
```

Bu noktada TransactionServlet, senkron kullanıcı isteği ile Kafka tabanlı asenkron işleme hattı arasında bir **köprü** görevi görür.

3.1.2.5. Transaction Validation and Lookup

Zorunlu parametrelerden herhangi biri eksikse (type, userEmail, amount, bankName) şu yanıt döner: {"success": false, "message": "Eksik parametreler!"}. Miktar double'a parse edilemezse: {"success": false, "message": "Geçersiz miktar!"}. Kullanıcı bulunamazsa: {"success": false, "message": "Kullanıcı bulunamadı!"}.

```
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\": \"Kullanıcı bulunamadı!\"}");
    return;
}
```

3.1.2.6. JSON Oluşturma ve Kafka'ya Asenkron Gönderim

Jackson ObjectMapper ve ObjectNode kullanılarak şu alanları içeren yapılandırılmış bir JSON oluşturulur: transactionId, type, userEmail, userId, userName, amount, bankName, description, timestamp. Eğer işlem türü transfer ise receiverName ve receiverBank alanları da eklenir. Bu JSON, UI'ya döndürülen yanıt **değildir**; Kafka'ya gönderilecek mesajdır. JSON string'e çevrilir ve transactionProducer.sendTransaction(transactionId, jsonString) çağrılır. Bu noktada API katmanı isteği kabul etmiş olur ve işlem messaging katmanına devredilir.

```
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 alıcı 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. Yanıt

SOAP katmanı, işlemin backend'e başarıyla iletildiğini belirten şu yanıtı döner: {"success": true, "message": "Transaction başarıyla gönderildi!", "transactionId": "..."}. Olası hatalar yakalanır, stack trace loglanır ve güvenli bir hata mesajı döndürülür.

```
    // 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

Kafka katmanı, sistemin **asenkron omurgasıdır**. API katmanını, bakiyeleri güncelleyen ve işlem kayıtlarını yazan iş mantığından ayırır. HTTP/SOAP isteği sırasında veritabanını doğrudan güncellemek yerine, API katmanı bir mesajı Kafka'ya yayınlar; Kafka katmanı ise bu mesajın güvenilir bir şekilde, gerçek işlemi gerçekleştiren consumer'a iletilmesini sağlar.

Bu projede Kafka katmanı şu bileşenlerden oluşmaktadır:

- Konfigürasyon sınıfları (KafkaConfig, KafkaTopicConfig)
- **Producer** servisi (TransactionProducer)
- **Consumer** bileşeni (TransactionConsumer)
- Payment-transactions adlı bir topic

3.2.1. Kafka Configuration

KafkaConfig sınıfı @Configuration ve @EnableKafka anotasyonları ile işaretlenmiştir. Bu sınıf, Kafka ile etkileşim kurmak için gerekli olan tüm bean'leri tanımlar.

```java
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, Kafka topic'lerinin idari konfigürasyonunu yönetir.

- KafkaAdmin bean'i tanımlanır.
- kafkaAdmin() metodu, aynı BOOTSTRAP_SERVERS ayarları ile yapılandırılmış bir KafkaAdmin nesnesi oluşturur.
- Bu admin client, uygulama başlatıldığında topic'lerin tanımlanmasına olanak sağlar.

Topic tanımı için, paymentTransactionsTopic() metodu bir NewTopic döndürür:

- **Topic adı:** payment-transactions
- **Partition sayısı:** 1
- **Replication factor:** 1

Uygulama başlatıldığında Spring, KafkaAdmin kullanarak payment-transactions topic'inin varlığını kontrol eder. Eğer topic yoksa otomatik olarak oluşturur. Bu otomatik topic oluşturma yaklaşımı, deploy sürecini kolaylaştırır ve producer ile consumer'ın her zaman tutarlı bir topic üzerinde çalışmasını garanti eder.

```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, Kafka'ya transaction verisi göndermekten sorumlu **tek giriş noktasıdır** ve bir Spring Service olarak tanımlanmıştır.

Bağımlılıklar ve Konfigürasyon için KafkaTemplate<String, String> bağımlılığı @Autowired ile inject edilir. Sonra TOPIC = "payment-transactions" sabiti tanımlanır. Bu şekilde temel metot beslenir. Temel metod sendTransaction(String transactionId, String transactionData) metodudur. Bu metot TransactionServlet (HTTP/JSON API) ve PaymentEndpoint (SOAP API) tarafından çağrılır.

Çalılması için kafkaTemplate.send(TOPIC, transactionId, transactionData) çağrısı yapılır. Sonra, transactionId, **message key** olarak kullanılır. Böylece aynı transaction'a ait tüm event'ler aynı partition'a yönlendirilir. transactionData, kullanıcı, miktar, banka, işlem türü gibi tüm iş alanlarını içeren JSON string'dir.

Gönderim sonrası topic, key ve payload bilgileri console'a loglanır. Eğer gönderim sırasında bir hata oluşursa, exception yakalanır ve System.err ile stack trace loglanır.

Bu producer yapısı, tüm Kafka yazma işlemlerini merkezileştirir ve her transaction mesajı için tutarlı bir topic, key ve veri formatı kullanılmasını sağlar.

```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, Kafka'dan mesajları okuyarak **çekirdek iş mantığını** çalıştıran bir Spring Component'tir. Şu DAO'lar inject edilir: UserDao, BankATransactionDao, BankBTransactionDao, BankCTransactionDao. Ayrıca gelen JSON payload'ları parse etmek için Jackson ObjectMapper ve JsonNode kullanılır. Bu bağımlılıklar sayesinde consumer kullanıcı bilgilerini okuyabilir ve bankaya göre farklı transaction tablolarına kayıt yazabilir

```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 Katmanı

Database katmanı, tüm iş verilerinin **kalıcı olarak saklanmasından** sorumludur. Kullanıcılar, bankalar ve işlemler bu katmanda tutulur. Bu katman Hibernate ORM, Jakarta Persistence ve MySQL üzerine inşa edilmiştir ve Spring'in transaction yönetimi ile **tam entegre** çalışır.

Ana bileşenleri; Hibernate ve data source konfigürasyonu (HibernateConfig), Database.Entities altında yer alan JPA entity sınıfları (ör. User, Transaction, banka transaction entity'leri), Database.DAO altındaki DAO sınıfları ve HibernateTransactionManager ve @Transactional kullanılarak transaction yönetimidir.

3.3.1. Configuration

HibernateConfig, şu anotasyonlarla işaretlenmiştir: @Configuration / @EnableTransactionManagement / @ComponentScan(basePackages = {"Database", "Message.Kafka"})

3.3.1.1. Data Source

dataSource() metodu bir DriverManagerDataSource oluşturur. Bu data'da **Driver:** com.mysql.cj.jdbc.Driver'a, URL:jdbc:mysql://localhost:3306/mydb?useSSL=false&serverTimezone =UTC&allowPublicKeyRetrieval=true'ya, **Username:** root'a ve **Password:** root'Bu bean, uygulamanın MySQL veritabanına nasıl bağlanacağını tanımlar.

```
@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() metodu, yapılandırılmış bir LocalSessionFactoryBean döndürür. Bu bean, yukarıda tanımlanan dataSource kullanılarak oluşturulur ve setPackagesToScan("Database.Entities") çağrısı ile ilgili paket altında bulunan tüm entity sınıflarını otomatik olarak keşfeder. Bu sayede sistemde tanımlı olan tüm JPA entity'leri manuel olarak eklenmeden Hibernate tarafından algılanır.

Hibernate için kullanılan temel ayarlar bu aşamada tanımlanır. hibernate.dialect değeri org.hibernate.dialect.MySQL8Dialect olarak ayarlanmıştır ve bu sayede Hibernate, MySQL 8 uyumlu SQL sorguları üretir. hibernate.show_sql özelliği true olarak ayarlanmıştır; bu da üretilen SQL sorgularının çalışma zamanında console'a yazdırılmasını sağlar ve debug sürecini kolaylaştırır. hibernate.hbm2ddl.auto ayarı update olarak belirlenmiştir; bu ayar, entity mapping'lerine göre veritabanı tablolarının otomatik olarak oluşturulmasını veya güncellenmesini sağlar.

Bu konfigürasyon sayesinde Hibernate, entity sınıfları üzerinde gerçekleştirilen save, update ve query gibi işlemleri otomatik olarak MySQL için uygun SQL sorgularına dönüştürür.

```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) metodu bir HibernateTransactionManager döndürür. Spring Framework, bu transaction manager'ı kullanarak veritabanı işlemlerinin transaction yaşam döngüsünü yönetir. Bu kapsamda transaction başlatma, commit etme ve rollback işlemleri Spring tarafından otomatik olarak gerçekleştirilir. DAO veya service katmanında @Transactional anotasyonu ile işaretlenmiş metotlar, bu transaction manager tarafından yönetilen transaction'lar içerisinde çalıştırılır. Böylece manuel olarak transaction başlatma veya sonlandırma ihtiyacı ortadan kalkar ve transaction yönetimi merkezi bir yapı altında toplanmış olur.

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

3.3.2. Entity Model

Entity sınıfları, veritabanındaki tabloları temsil eder ve bu tablolar arasındaki ilişkileri tanımlar. Hibernate, bu sınıfları JPA anotasyonları aracılığıyla ilişkisel veritabanı tablolarına map eder. Her entity sınıfı, sistemdeki iş verilerinin yapısal karşılığını oluşturur.

3.3.2.1. User Entity

User entity'si @Entity ve @Table(name = "users") anotasyonları ile işaretlenmiştir.

Ana alanlar; id birincil anahtardır ve @GeneratedValue(strategy = GenerationType.IDENTITY) ile otomatik üretilir. name alanı zorunludur (@Column(nullable = false)). email alanı hem zorunlu hem de benzersizdir (@Column(nullable = false, unique = true)). password alanı zorunludur. balance alanı mevcut hesap bakiyesini tutar. bank alanı ise bir Bank entity'si ile many-to-one ilişki kurar; bu ilişki @ManyToOne(fetch = FetchType.LAZY) ve @JoinColumn(name = "bank_id", nullable = false) anotasyonları ile tanımlanmıştır.

Bu entity, tek bir banka ve tek bir bakiye ile ilişkili bir banka müşterisini temsil eder.

```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;
```

3.3.2.2. Transaction Entity

Transaction entity'si @Entity ve @Table(name = "transactions") anotasyonları ile işaretlenmiştir.

Alanlar; id birincil anahtardır ve otomatik üretilir. sendingUser, parayı gönderen kullanıcıyı temsil eder ve User entity'si ile many-to-one ilişki kurar (@ManyToOne(fetch = FetchType.LAZY), @JoinColumn(name = "sending_user_id", nullable = false)). receivingUser, parayı alan kullanıcıyı temsil eder ve yine User entity'si ile ilişkilidir (@JoinColumn(name = "receiving_user_id", nullable = false)). amount alanı transfer edilen miktarı tutar. timestamp alanı @CreationTimestamp ve @Column(nullable = false, updatable = false) anotasyonları ile işaretlenmiştir ve kayıt eklendiği anda otomatik olarak doldurulur.

```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;
```

Bu entity, kullanıcılar arası para transferlerini kaydederek sistemin kimin kime, ne zaman ve ne kadar para gönderdiğini takip etmesini sağlar.

3.3.2.3. Bank Entity

```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;
```

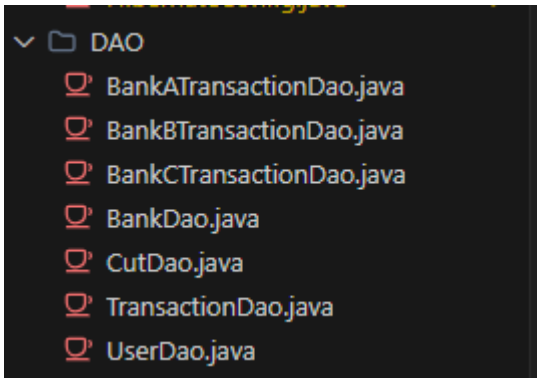
BankATransaction, BankBTransaction ve BankCTransaction gibi ek entity'ler (burada tamamı gösterilmemiştir) her banka için ayrı işlem loglarını tutar.

Bu entity'ler genellikle kullanıcıya veya kullanıcı ID'sine referans, işlem miktarı, işlem türü (DEPOSIT, WITHDRAW, TRANSFER) ve timestamp bilgilerini içerir.

Bu entity'ler Kafka consumer tarafından her banka için detaylı işlem geçmişlerini yazmak amacıyla kullanılır ve raporlama ile denetim süreçlerini mümkün kılar..

3.3.3. Data Access Layer

DAO sınıfları, veritabanı işlemlerini gerçekleştirmek için temiz ve nesne yönelimli bir arayüz sunar. Bu sınıflar, altta yatan Hibernate API'sini üst katmanlardan gizler.



Tüm DAO'lar @Repository anotasyonu ile işaretlenmiştir; bu sayede Spring component scanning ve exception translation sağlar. Ayrıca @Transactional anotasyonu ile işlemlerin transaction içerisinde çalışması garanti edilir. DAO'lar paylaşılan SessionFactory nesnesini @Autowired ile alır ve aktif session'ı sessionFactory.getCurrent Session() çağrısı üzerinden elde eder.

Bu tasarım, transaction yönetimini manuel begin/commit çağrıları yerine Spring'e merkezileştirir.

3.4. Web Layer

Son olarak, uygulama localhost:8080 üzerinde çalışırken Ngrok kullanılarak public hale getirilmiştir. Ngrok, local ortamda yayınlanan içeriği alarak https://unconsternated-theodore-nonedified.ngrok-free.dev adresine yansıtır.

4. Proje Analizi

4.1. Test

Testler Windows bilgisayarlar ve Android telefonlar üzerinde yapılmıştır. Sistem local ortamda, public erişimde ve telefon üzerinden public erişimde test edilmiştir ve her seferinde sorunsuz çalışmıştır. Ortalama yanıt süresi yaklaşık ~100 ms civarındadır ve bu tür bir proje için yeterlidir. Kullanılan ngrok'un ücretsiz sürümü nedeniyle aynı anda yalnızca bir kişinin bağlanabilmesi tek dezavantajdır.

4.2. Geliştirilebilir Alanlar

Öncelikle, proje için bir kesinti (cut) yapısı tasarlanmıştır ancak projeye eklenmemiştir. Bu yapı, bankalar arası bir işlem Bank C'yi içerdiğinde transfer edilen tutardan %8 kesinti yapılmasını hedeflemektedir.

Ayrıca proje bazı siber saldırılara karşı savunmasızdır. Gerçek bir sistem olarak kullanılmak üzere geliştirilmediği için yalnızca basit kullanım amaçlı tasarlanmıştır. Uygun siber güvenlik çalışmaları ile gerçek hayatta kullanılabilecek hale getirilebilir.

Son olarak, bazı durumlarda Kafka uygulama başlatılırken çökebilmektedir. Sebebi bilinmemektedir ancak Kafka yeniden başlatıldığında sistem sorunsuz şekilde çalışmaktadır.

5. Proje Yorumları

5.1. Olası Kullanım Alanları

Bu proje tek başına kullanılabilir bir ürün olmamakla birlikte, yapısı başka projelerde veya gelecekteki projeler için bir web sitesi mimarisi olarak kullanılabilir.

5.1.1. Kullanırken Yapılacaklar

Bu proje tek başına kullanılabilir bir ürün olmamakla birlikte, yapısı başka projelerde veya gelecekteki projeler için bir web sitesi mimarisi olarak kullanılabilir..

5.2. Creator Comments

Bu proje başlangıçta basit bir ödevdi. Ancak daha karmaşık, daha detaylı, daha zaman alıcı ve daha zor hale getirdim. Buna rağmen proje; liderlik, Maven ile Java kodlama ve daha büyük projelere hazırlanma konusunda bana çok şey kazandırdı. Bu proje, League of Legends Manager oyun ve web sitesi projem için bir itici güç oldu.

Hooah

– 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 İçerikleri

- Project kaynak kodu
- Bu raporun PDF formatında hali
- Demo Video
- Money for Nothing by Dire Straits