**<https://chatgpt.com/c/68651ba4-32e4-8012-9e41-e293317873b0>**

**خلاصه مسیر پیشنهادی پروژه:**

1. ساخت پروژه Spring Boot
2. راه‌اندازی Keycloak و اتصال به پروژه
3. اضافه کردن JWT با Keycloak
4. تعریف state machine (مثلاً lifecycle سفارش)
5. مدل‌سازی همین فرآیند در Camunda
6. پیاده‌سازی Saga Orchestration با Camunda یا State Machine

اگه موافقی، می‌تونم مرحله‌به‌مرحله همراهیت کنم. دوست داری از کدوم بخش شروع کنیم؟ ساخت پروژه و تنظیم Keycloak؟

عالیه! حالا که پروژه Spring Boot آماده‌ست، بریم سراغ مرحله بعد:

# ✅ مرحله 2: راه‌اندازی Keycloak با Docker

**🎯 هدف:**

* اجرای Keycloak روی سیستم محلی با Docker
* ساخت Realm، Client و کاربر تست
* آماده‌سازی برای اتصال پروژه Spring Boot به Keycloak

**🔧 1. اجرای Keycloak در Docker**

در پوشه‌ای دلخواه، فایل زیر به اسم docker-compose.yml بساز:

version: '3.8'

services:

keycloak:

image: quay.io/keycloak/keycloak:24.0.2

container\_name: keycloak

command: start-dev

ports:

- "8081:8080"

environment:

KEYCLOAK\_ADMIN: admin

KEYCLOAK\_ADMIN\_PASSWORD: admin

و سپس اجرا کن:

docker-compose up -d

این کار Keycloak رو در آدرس زیر بالا میاره:  
[http://localhost:8081](http://localhost:8081/)

با یوزر و پسورد:

* **Username:** admin
* **Password:** admin

**🏗️ 2. پیکربندی اولیه در Keycloak**

حالا داخل پنل ادمین Keycloak (<http://localhost:8081/admin>) وارد شو و مراحل زیر رو انجام بده:

**✅ ساخت Realm جدید:**

1. از منوی بالا سمت چپ، روی "Master" کلیک کن و "Create Realm" رو بزن.
2. اسم مثلاً بذار: demo-realm

**✅ ساخت Client:**

1. داخل Realm، برو به "Clients"
2. روی "Create Client" کلیک کن
3. تنظیمات:
   * Client ID: springboot-client
   * Client Type: **OpenID Connect**
   * Root URL: http://localhost:8080
   * بقیه موارد پیش‌فرض

بعد از ایجاد:

* داخل تنظیمات Client، به تب **"Settings"** برو:
  + Access Type: confidential
  + Standard Flow Enabled: ✅
  + Direct Access Grants Enabled: ✅
  + Valid Redirect URIs: http://localhost:8080/\*
  + Save کن

بعد برو تب **Credentials** و **Client Secret** رو کپی کن. بعداً لازم می‌شه.

**✅ ساخت User تست:**

1. برو به "Users"
2. "Add User" بزن
   * Username: testuser
   * Enabled: ✅
3. بعد از ایجاد، برو به تب **Credentials**
   * Password: testpass
   * Temporary: ❌

**🎯 مرحله بعد:**

وصل کردن پروژه Spring Boot به Keycloak و گرفتن JWT بعد از لاگین.

آیا Keycloak بالا اومد و این موارد (Realm, Client, User) رو تونستی بسازی؟  
اگر بله، بریم سراغ مرحله اتصال Spring Boot به Keycloak.

سؤال بسیار خوبی پرسیدی، چون Keycloak از نسخه 17 به بعد (مخصوصاً نسخه‌های 22 به بالا) رابط کاربری جدیدی داره و بعضی گزینه‌ها **تغییر مکان دادن** یا **نام‌هاشون عوض شده**. در نسخه 24 که الان داری، مواردی مثل Access Type و Credentials دیگه اون شکلی که توی آموزش‌های قدیمی هست دیده نمی‌شن.

اجازه بده سریع راهنمایت کنم:

**🧭 آموزش تنظیم Client در Keycloak 24**

**🧩 قدم 1: ساخت Client**

1. وارد Realm دلخواهت شو (مثل demo-realm)
2. از منوی سمت چپ روی **Clients** کلیک کن
3. روی **Create client** کلیک کن

تنظیمات اولیه:

| **گزینه** | **مقدار** |
| --- | --- |
| Client ID | springboot-client |
| Client type | OpenID Connect |
| Root URL (اختیاری) | http://localhost:8080 |

روی **Next** کلیک کن.

**🧩 قدم 2: تنظیم دسترسی‌ها**

در صفحه بعدی (یا بعد از ساخت)، این تنظیمات رو انجام بده:

1. در تب **Capability config**:
   * تیک **Client authentication** رو فعال کن ✅ (این همون "Access Type: confidential" قدیمیه)
   * باقی موارد رو پیش‌فرض بذار
2. در تب **Credentials**:
   * این تب الان جزو منوی سمت چپ داخل صفحه client هست، نه تب جدا
   * برو به منوی سمت چپ همون صفحه و روی **Credentials** کلیک کن
   * اونجا **Client secret** رو می‌بینی و می‌تونی کپی کنی

lGx26eb1HeX3LYWubUyR2qLrWVK6ucOv

**🧩 قدم 3: تنظیم Redirect URI**

1. در تب **Settings** یا همون صفحه اصلی client:
   * برو پایین تا بخش **Valid redirect URIs**
   * مقدار زیر رو وارد کن:
   * http://localhost:8080/\*
   * Save رو بزن

اگه دوست داشتی، می‌تونم یه اسکرین‌شات از مسیر دقیق توی رابط جدید Keycloak برات طراحی کنم.

⬇️ وقتی این مرحله‌ها رو انجام دادی و client secret رو گرفتی، بگو تا بریم سراغ اتصال Spring Boot به Keycloak و گرفتن JWT.

خیلی خوبه 👌 حالا می‌ریم سراغ:

# ✅ مرحله 3: اتصال Spring Boot به Keycloak

**🎯 هدف:**

* پیکربندی Spring Boot برای احراز هویت با Keycloak
* محافظت از endpointها با توکن JWT

**🧩 1. افزودن وابستگی‌های Keycloak و امنیت**

در pom.xml این dependencyها رو اضافه کن:

<dependencies>

<!-- سایر وابستگی‌ها -->

<!-- Spring Security OIDC -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-oauth2-resource-server</artifactId>

</dependency>

<!-- استفاده از JWT -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<!-- برای خواندن مقادیر در تنظیمات -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

</dependencies>

**🧩 2. پیکربندی در application.yml (یا application.properties)**

ترجیحاً فایل application.yml بساز و تنظیمات زیر رو بذار:

spring.application.name=demo

spring.datasource.url=jdbc:mysql://localhost:3306/demo\_db?useSSL=false&serverTimezone=UTC

spring.datasource.username=root

spring.datasource.password=Bahman2462

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

server.port= 8082

spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:8081/realms/demo-realm

spring.security.oauth2.resourceserver.jwt.jwk-set-uri=http://localhost:8081/realms/demo-realm/protocol/openid-connect/certs

اگر فقط issuer-uri رو بدی، Spring Boot خودش کلید عمومی Realm رو از Keycloak می‌گیره.

**🧩 3. ساخت کنترلر تست**

مثلاً یک HelloController برای تست احراز هویت:

package com.example.demo;

import java.security.Principal;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

@RestController

@RequestMapping("/api")

public class HelloController {

    @GetMapping("/public")

    public String publicHello() {

        return "Hello Public!";

    }

    @GetMapping("/private")

    public String privateHello(Principal principal) {

        return "Hello Private, " + principal.getName();

    }

}

**🧩 4. تنظیم امنیت HTTP**

کلاسی مثل زیر بساز (مثلاً SecurityConfig):

package com.example.demo;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.security.config.annotation.web.builders.HttpSecurity;

import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;

import org.springframework.security.web.SecurityFilterChain;

@Configuration

@EnableWebSecurity

public class SecurityConfig {

    @Bean

    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http

            .authorizeHttpRequests(authz -> authz

                .requestMatchers("/api/public").permitAll()   // آزاد

                .requestMatchers("/api/private").authenticated() // نیازمند لاگین

                .anyRequest().permitAll()

            )

            .oauth2ResourceServer(oauth2 -> oauth2

                .jwt()

            );

        return http.build();

    }

}

بایستی کلاس امنیتی داخل پکیج controller باشه و کنار هم باشن

**🧪 5. تست با توکن Keycloak**

**بعدش پروژه رو اجرا میکنیم:**

* + 1. **نصب وابستگی های جدید Po**
       1. **mvnw.cmd clean install**
    2. **اجرای پروژه بعد از اصلاح و بیلد**
       1. **mvnw.cmd spring-boot:run**

برای گرفتن توکن، از Postman یا curl استفاده کن:

POST http://localhost:8081/realms/demo-realm/protocol/openid-connect/token

Content-Type: application/x-www-form-urlencoded

client\_id=springboot-client

client\_secret=<<< client secret >>>

username=testuser

password=testpass

grant\_type=password

خروجی:

{

"access\_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",

...

}

توکن رو در درخواست به /api/private بفرست:

GET http://localhost:8082/api/private

Authorization: Bearer eyJhbGciOi...

باید جواب بده:

Hello Private, testuser

✅ مرحله بعدی؟  
می‌تونیم بریم سراغ:

* تعریف Roleها در Keycloak و دسترسی مبتنی بر نقش
* یا
* شروع Camunda و اتصال به پروژه

# مرحله 4 - **تعریف Roleها و اعمال دسترسی بر اساس نقش در Keycloak + Spring Boot**

منطقیه که اول Keycloak رو کامل و اصولی راه بندازی، بعد بریم سراغ Camunda، State Machine، Saga و بقیه موارد. چون هویت و سطوح دسترسی پایه بسیاری از چیزهای بعدی هست.

**✅ فازهای بعدی Keycloak که با هم انجام می‌دیم:**

1. **تعریف Roleها در Realm و تخصیص به کاربران**
2. اعمال **دسترسی‌ها در Spring Security** بر اساس نقش
3. استخراج نقش‌ها از توکن JWT
4. تعریف endpointهای مختلف با نقش‌های مختلف (مثلاً فقط Admin یا فقط User)
5. ساختن یک client با role-based access (برای frontend یا microservice)
6. تست با Postman با نقش‌های مختلف

**📌 گام بعد: تعریف Role در Keycloak**

اگر موافقی، از همین الان شروع کنیم:

**✅ گام اول – تعریف Role:**

1. وارد ادمین کنسول Keycloak شو (<http://localhost:8081/>)
2. وارد Realm خودت شو (demo-realm)
3. از منوی سمت چپ، برو به **Roles**
4. روی **Create Role** کلیک کن
   * **Role name**: مثلا ROLE\_ADMIN
   * ذخیره کن
   * دوباره یکی دیگه با نام مثلاً ROLE\_USER بساز

**✅ گام دوم – تخصیص Role به کاربر:**

1. از منوی سمت چپ برو به **Users**
2. روی testuser کلیک کن
3. برو به تب **Role Mappings**
4. در قسمت **Available Roles**:
   * ROLE\_USER رو انتخاب کن و به سمت راست (Assigned roles) ببر

اگر این کارها رو انجام دادی، بگو تا بریم سراغ پیکربندی Spring Security که نقش‌ها رو بخونه و ازش استفاده کنه برای کنترل دسترسی.

# مرحله 5 - استفاده از Roleهای Keycloak در Spring Boot (Role-Based Access Control)

**🎯 هدف:**

* بخونیم نقش‌ها (Roles) رو از JWT
* و از اون‌ها برای محدود کردن دسترسی به endpointها استفاده کنیم (مثلاً فقط ROLE\_ADMIN بتونه یک مسیر خاص رو ببینه)

**🧩 1. نقش‌ها در Keycloak کجای توکن JWT میان؟**

Keycloak توکن JWT رو به شکل زیر تولید می‌کنه:

{

...

"realm\_access": {

"roles": [

"ROLE\_USER",

"ROLE\_ADMIN"

]

},

...

}

ما باید Spring رو طوری پیکربندی کنیم که این قسمت رو بخونه.

**🛠️ 2. پیکربندی خواندن نقش‌ها از realm\_access.roles**

**✅ ایجاد کلاس KeycloakRoleConverter**

**یک کلاس ساده برای استخراج نقش‌ها**

**🧩 3. اتصال KeycloakRoleConverter به Spring Security**

در SecurityConfig.java به این شکل اضافه‌اش کن:

@Configuration

@EnableWebSecurity

public class SecurityConfig {

    @Bean

    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

        http

            .authorizeHttpRequests(authz -> authz

                .requestMatchers("/api/public").permitAll()

                .requestMatchers("/api/private").authenticated()

                .requestMatchers("/api/admin").hasRole("ADMIN")

                .anyRequest().permitAll()

            )

            .oauth2ResourceServer(oauth2 -> oauth2

                .jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthenticationConverter()))

            );

        return http.build();

    }

    @Bean

    public JwtAuthenticationConverter jwtAuthenticationConverter() {

        JwtAuthenticationConverter converter = new JwtAuthenticationConverter();

        converter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleConverter());

        return converter;

    }

}

**🧪 4. تست endpoint با Role**

فرض کن این endpoint فقط برای ROLE\_ADMIN باز باشه:

@GetMapping("/api/admin")

public ResponseEntity<String> admin() {

return ResponseEntity.ok("Admin content");

}

**🧪 تست با Postman**

1. با کاربری که فقط ROLE\_USER داره، توکن بگیر → 401 Unauthorized
2. با کاربری که ROLE\_ADMIN داره، توکن بگیر → موفق

اگر تا اینجا پیاده‌سازی کردی، بهم بگو تا برایت:

* راهنمایی ساخت کاربر admin در Keycloak
* و تست درستش رو بفرستم.

بریم جلو؟ ✅

**📦 ساختار پوشه‌هات:**

src/

└── main/

└── java/

└── com/

└── example/

├── demo/

│ └── SecurityConfig.java

└── KeycloakRoleConverter.java ✅ این فایله

# خلاصه کار تا اینجا

عالیه! 👏 یعنی تا اینجا با موفقیت انجام دادی:

✅ اتصال Spring Boot به Keycloak  
✅ تعریف Endpointهای عمومی و خصوصی  
✅ تعریف نقش‌ها در Keycloak  
✅ گرفتن Token از Keycloak و ارسال با Postman  
✅ محافظت از APIها بر اساس نقش (hasRole("ADMIN"))

**🎯 مرحله بعد: انتخاب با تو**

حالا می‌تونیم به یکی از مراحل زیر وارد بشیم:

**🔄 ۱. اتصال Keycloak به Camunda**

* استفاده از همون Keycloak برای احراز هویت کاربران در Camunda
* بررسی اینکه چطور Roleهای Camunda رو از طریق Keycloak مدیریت کنیم

**🧩 ۲. راه‌اندازی Camunda در پروژه**

* نصب و اجرای Camunda (با Docker یا embedded)
* اجرای اولین Process Model ساده
* REST APIهای Camunda و تست با Postman

**🎛 ۳. پیاده‌سازی یک State Machine یا Saga (مرحله‌های بعدی‌تر)**

بیایم مرحله اول یعنی **راه‌اندازی مستقل Camunda** رو شروع کنیم. در این مرحله:

**🔹 سوال: شما کدوم نسخه Camunda رو مدنظر دارید؟**

* **Camunda 7** (جاوا مبتنی بر Servlet/Embedded و UI ساده‌تر)
* **Camunda 8** (معماری مبتنی بر Zeebe، microservices-friendly و نیازمند بیشتر منابع)

اگر برای شروع با Camunda آشنا نیستید یا هدف یادگیری مفاهیم اصلیه، پیشنهاد من: **Camunda 7**

کدوم رو ترجیح می‌دی؟

* ✅ docker-compose با Camunda 7
* ⬜ اجرای دستی فایل WAR در Tomcat

# مرحله 6 – راه اندازی Camunda

عالیه، این تقسیم‌بندی کاملاً منطقیه. بیایم مرحله اول یعنی **راه‌اندازی مستقل Camunda** رو شروع کنیم. در این مرحله:

**✅ مرحله ۱: راه‌اندازی Camunda به‌صورت مستقل**

**مقایسه سریع دو گزینه‌ی اجرایی:**

| **روش** | **مزایا** | **معایب** |
| --- | --- | --- |
| ✅ **Docker Compose** (پیشنهاد من) | سریع‌ترین راه، بدون نیاز به نصب Tomcat یا تنظیم محیط دستی | نیاز به Docker نصب‌شده (که شما داری 👍) |
| ⬜ اجرای فایل WAR در Tomcat | اگر بخوای مستقیماً روی یک سرور Java اجرا کنی مفیده | نیاز به تنظیم دستی Tomcat، deploy فایل WAR، و پیکربندی JVM |

**✅ نتیجه: Docker Compose بهترین گزینه برای یادگیری و راه‌اندازی سریع هست.**

بریم سراغ راه‌اندازی Camunda 7 با Docker Compose:

**⚙️ مرحله راه‌اندازی Camunda 7 با Docker Compose**

۱. یک فایل docker-compose.yml بساز با محتوای زیر:

version: '3.7'

services:

camunda:

image: camunda/camunda-bpm-platform:run-latest

ports:

- "8083:8083"

۲. در پوشه‌ای که این فایل هست، اجرا کن:

docker compose up -d

۳. سپس در مرورگر باز کن:

http://localhost:8083

* یوزرنیم پیش‌فرض: demo
* پسورد پیش‌فرض: demo

✅ وقتی تونستی بالا بیاری، بهم بگو تا بریم سراغ اتصال Camunda به پروژه جاوای خودت (مرحله ۲).

اگه خواستی به جای run-latest از نسخه خاصی استفاده کنی (مثلاً 7.20.0)، می‌تونم نسخه دقیق‌تر رو هم بهت بگم.

نکته‌ی دقیقی رو اشاره کردی، و مشکلت هم واضح مشخصه:

**🔴 مشکل:**

در فایل docker-compose.yml نوشتی:

ports:

- "8083:8083"

اما تصویر Docker camunda/camunda-bpm-platform:run-latest، به‌صورت پیش‌فرض **سرویس رو روی پورت 8080 داخل کانتینر اجرا می‌کنه**.

**✅ راه‌حل:**

باید پورت **داخلی کانتینر (8080)** رو به **پورت دلخواه روی سیستم (مثلاً 8083)** نگاشت کنی. یعنی باید بنویسی:

ports:

- "8083:8080"

فرمت نگاشت پورت در Docker Compose:

"پورت\_لوکال:پورت\_کانتینر"

**🔁 پس فایل درست:**

version: '3.7'

services:

camunda:

image: camunda/camunda-bpm-platform:run-latest

ports:

- "8083:8080"

**⏭ مراحل بعد از اصلاح:**

1. فایل رو ذخیره کن
2. کانتینر قبلی رو stop و remove کن:

docker compose down

1. مجدد بالا بیار:

docker compose up -d

1. در مرورگر باز کن:

http://localhost:8083

باید صفحه مدیریت Camunda با یوزر demo و پسورد demo بیاد بالا.  
وقتی آماده شدی، بریم مرحله بعد: **اتصال پروژه جاوا به Camunda**.

## خطای isolation level موقع اجرای کاموندا در داکر

Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'processEngineFactoryBean': FactoryBean threw exception on object creation

at org.springframework.beans.factory.support.FactoryBeanRegistrySupport.doGetObjectFromFactoryBean(FactoryBeanRegistrySupport.java:192) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.FactoryBeanRegistrySupport.getObjectFromFactoryBean(FactoryBeanRegistrySupport.java:125) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.AbstractBeanFactory.getObjectForBeanInstance(AbstractBeanFactory.java:1884) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.getObjectForBeanInstance(AbstractAutowireCapableBeanFactory.java:1302) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:261) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:202) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.config.DependencyDescriptor.resolveCandidate(DependencyDescriptor.java:254) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.DefaultListableBeanFactory.addCandidateEntry(DefaultListableBeanFactory.java:1926) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.DefaultListableBeanFactory.findAutowireCandidates(DefaultListableBeanFactory.java:1883) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultListableBeanFactory.java:1621) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListableBeanFactory.java:1555) ~[spring-beans-6.2.5.jar!/:6.2.5]

at org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement.resolveFieldValue(AutowiredAnnotationBeanPostProcessor.java:785) ~[spring-beans-6.2.5.jar!/:6.2.5]

... 29 common frames omitted

Caused by: org.camunda.bpm.engine.ProcessEngineException: ENGINE-12019 The transaction isolation level set for the database is 'REPEATABLE\_READ' which differs from the recommended value. Please change the isolation level to 'READ\_COMMITTED' or set property 'skipIsolationLevelCheck' to true. Please keep in mind that some levels are known to cause deadlocks and other unexpected behaviours.

راه حل:

مشکل دقیقاً از isolation level دیتابیس است. Camunda نیاز دارد که isolation level دیتابیس روی `READ\_COMMITTED` باشد، اما در حال حاضر روی `REPEATABLE\_READ` است.

## راه‌حل‌های ممکن:

### 1. \*\*تغییر isolation level دیتابیس\*\* (راه‌حل توصیه شده)

برای MySQL:

```sql

-- برای کل دیتابیس

ALTER DATABASE your\_database\_name CHARACTER SET utf8 COLLATE utf8\_general\_ci;

-- یا برای session جاری

SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;

SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

### 2. \*\*اضافه کردن property در application.yml\*\* (راه‌حل موقت)

```yaml

camunda:

bpm:

generic-properties:

properties:

skipIsolationLevelCheck: true

```

### 3. \*\*استفاده از connection string با پارامتر isolation level\*\*

برای MySQL در connection string:

```

jdbc:mysql://localhost:3306/camunda?sessionVariables=transaction\_isolation='READ-COMMITTED'

```

### 4. \*\*بررسی و تغییر در داکر کامپوز\*\*

اگر از داکر استفاده می‌کنید، در فایل `docker-compose.yml` برای سرویس دیتابیس:

```yaml

services:

database:

image: postgres:13

environment:

- POSTGRES\_DB=camunda

- POSTGRES\_USER=camunda

- POSTGRES\_PASSWORD=camunda

command: >

postgres -c default\_transaction\_isolation=read\_committed

```

## بهترین راه‌حل:

\*\*راه‌حل شماره 1 (تغییر isolation level دیتابیس)\*\* را توصیه می‌کنم چون:

- مشکل ریشه‌ای را حل می‌کند

- با توصیه‌های Camunda مطابقت دارد

- از مشکلات آینده مانند deadlock جلوگیری می‌کند

پس از اعمال تغییرات، دیتابیس و اپلیکیشن را restart کنید.

# مرحله 7 – **اتصال پروژه Spring Boot به Camunda**

برای اینکه پروژه‌ت بتونه با Camunda تعامل کنه (مثل deploy کردن فرایند BPMN یا شروع کردن process instance)، دو مسیر وجود داره:

**🔹 مسیر ۱: استفاده از REST API Camunda**

اگر فقط می‌خوای از بیرون با Camunda تعامل داشته باشی (مثلاً از پروژه‌ات فایل BPMN بفرستی، یا یک process instance رو با POST راه‌اندازی کنی)، نیازی به اضافه کردن dependency داخل پروژه نیست، فقط:

* از آدرس‌های REST استفاده می‌کنی مثل:
  + POST http://localhost:8083/engine-rest/process-definition/key/{key}/start
  + GET http://localhost:8083/engine-rest/task
* این روش سبک‌تر و decoupled هست.

**🔹 مسیر ۲: ادغام داخلی Camunda در پروژه Spring Boot**

اگر می‌خوای:

* داخل پروژه‌ی Spring Boot:
  + BPMN تعریف و deploy کنی
  + Delegate و Service Task بنویسی (کدهای Java برای Taskها)
  + مستقیماً فرآیندها رو اجرا کنی یا با Spring Bean ارتباط داشته باشن

باید Camunda رو به پروژه اضافه کنیم.

**✅ پیشنهاد من:**

فعلاً بریم سراغ مسیر ۲ (ادغام در Spring Boot) تا بتونی فرآیند بنویسی، تست کنی و به keycloak هم وصلش کنیم.

**🧱 مرحله بعد: اضافه کردن Camunda به پروژه**

بیا ابتدا Camunda رو به پروژه اضافه کنیم. فرض می‌کنم پروژه‌ی Spring Boot با Maven هست.

**1. اضافه کردن dependency به pom.xml:**

<dependency>

<groupId>org.camunda.bpm.springboot</groupId>

<artifactId>camunda-bpm-spring-boot-starter</artifactId>

<version>7.21.0</version>

</dependency>

**2. فایل application.properties یا application.yml رو ساده نگه دار:**

spring.datasource.url=jdbc:h2:mem:camunda;DB\_CLOSE\_DELAY=1000

spring.datasource.driverClassName=org.h2.Driver

spring.datasource.username=sa

spring.datasource.password=

camunda.bpm.admin-user.id=demo

camunda.bpm.admin-user.password=demo

camunda.bpm.admin-user.first-name=Demo

(بعداً دیتابیس رو می‌تونیم بذاریم روی MySQL یا PostgreSQL)

**3. تست ساده با فایل BPMN**

در مسیر src/main/resources یک فایل ساده process.bpmn بذار که یه Start Event، یک User Task، و یک End Event داشته باشه.

<?xml version="1.0" encoding="UTF-8"?>

<bpmn:definitions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"

xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"

xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"

id="Definitions\_1"

targetNamespace="http://camunda.org/examples">

<bpmn:process id="simple\_process" name="Simple Process" isExecutable="true">

<bpmn:startEvent id="startEvent" name="Start">

<bpmn:outgoing>flow1</bpmn:outgoing>

</bpmn:startEvent>

<bpmn:userTask id="userTask" name="Do something">

<bpmn:incoming>flow1</bpmn:incoming>

<bpmn:outgoing>flow2</bpmn:outgoing>

</bpmn:userTask>

<bpmn:endEvent id="endEvent" name="End">

<bpmn:incoming>flow2</bpmn:incoming>

</bpmn:endEvent>

<bpmn:sequenceFlow id="flow1" sourceRef="startEvent" targetRef="userTask"/>

<bpmn:sequenceFlow id="flow2" sourceRef="userTask" targetRef="endEvent"/>

</bpmn:process>

<bpmndi:BPMNDiagram id="BPMNDiagram\_1">

<bpmndi:BPMNPlane id="BPMNPlane\_1" bpmnElement="simple\_process">

<!-- Optional visual layout: حذف شده برای سادگی -->

</bpmndi:BPMNPlane>

</bpmndi:BPMNDiagram>

</bpmn:definitions>

بعد از گذاشتن این فایل:

1. پروژه رو اجرا کن.
2. اگر همه چیز درست باشه، این فرآیند به صورت خودکار deploy میشه.
3. می‌تونی از طریق Camunda Cockpit (در آدرس: <http://localhost:8083/camunda>) بری و بررسیش کنی.

## نکات:

1. خیلی سخت وصل شد
2. مشکل میتونست از کانفیگ باشه در application.properties
3. میتونست از نسخه Camunda که روی داکر بود با نسخه ای که در dependency های pom گذاشته بودیم باشه

## دلیل نیاز به تنظیم `historyTimeToLive` در Camunda

مربوط به تغییرات امنیتی و مدیریت داده‌های تاریخی در نسخه‌های جدید Camunda (از نسخه 7.13 به بعد) می‌شود.

## چرا این تنظیم اجباری شد؟

1. \*\*مدیریت حجم داده‌های تاریخی\*\*: Camunda به طور پیش‌فرض تمام رویدادهای فرآیند را در دیتابیس ذخیره می‌کند. بدون تنظیم TTL، این داده‌ها می‌توانند به مرور زمان بسیار حجیم شوند.

2. \*\*امنیت داده‌ها\*\*: با تعیین زمان انقضا، از ذخیره‌سازی نامحدود داده‌های حساس جلوگیری می‌شود.

3. \*\*بهبود عملکرد\*\*: داده‌های تاریخی قدیمی می‌توانند بر عملکرد سیستم تأثیر منفی بگذارند.

## مقادیر پیشنهادی برای `historyTimeToLive`:

- `P7D`: 7 روز (مناسب برای محیط‌های تست)

- `P30D`: 30 روز (پیش‌فرض مناسب برای اکثر محیط‌ها)

- `P365D`: 1 سال (برای فرآیندهای با طول عمر طولانی)

- `P0D`: برای غیرفعال کردن ذخیره‌سازی تاریخچه (غیر توصیه شده)

## اگر این تنظیم را انجام ندهید:

- در نسخه‌های جدید Camunda، استقرار فرآیندها با خطا مواجه می‌شود (همانطور که شما تجربه کردید)

- در نسخه‌های قدیمی‌تر، داده‌های تاریخی به صورت نامحدود ذخیره می‌شدند که می‌توانست باعث مشکلات عملکردی شود

## تنظیمات پیشرفته‌تر:

اگر نیاز به کنترل دقیق‌تر دارید، می‌توانید برای قسمت‌های مختلف فرآیند تنظیمات جداگانه تعریف کنید:

```xml

<bpmn:process id="my-process" name="My Process" isExecutable="true"

camunda:historyTimeToLive="P30D"

camunda:historyTimeToLiveCaseInstance="P60D"

camunda:historyTimeToLiveDecisionInstance="P90D">

```

این رفتار اگرچه در ابتدا ممکن است عجیب به نظر برسد، اما در واقع یک بهبود امنیتی و مدیریتی مهم در Camunda محسوب می‌شود.

عالیه 👏 کاملاً درست پیش رفتی و الان شرایطت برای ادامه‌ی مرحله‌ی **"اتصال Camunda به پروژه جاوایی"** کاملاً آماده‌ست.  
بریم مرحله‌به‌مرحله جلو تا هم Camunda رو خوب درک کنی، هم بتونی با Spring Boot کنترلش کنی.

## اجرای یک فرآیند ساده (BPMN) با Camunda از طریق پروژه‌ی Spring Boot.

**✅ قدم 1: اضافه کردن Camunda به پروژه Spring Boot**

چون Camunda رو **به‌صورت مستقل با Docker** اجرا کردی، توی پروژه جاوا فقط باید یه سری dependency و تنظیمات بذاری تا پروژه‌ات بتونه با اون ارتباط بگیره.

اما یه نکته:

تو داری از **Camunda Platform 7 (کلاسیک)** استفاده می‌کنی، نه Camunda 8. پس باید از Camunda 7 Spring Boot Starter استفاده کنیم.

**✅ 1.1 تغییرات در pom.xml**

<properties>

<java.version>17</java.version>

<camunda.version>7.23.0</camunda.version>

</properties>

<dependencies>

<!-- Camunda Process Engine -->

<dependency>

<groupId>org.camunda.bpm.springboot</groupId>

<artifactId>camunda-bpm-spring-boot-starter</artifactId>

<version>${camunda.version}</version>

</dependency>

<!-- Web & Security (قبلاً هستن احتمالاً) -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<!-- سایر dependencyهای پروژه‌ات هم اینجا باشن -->

</dependencies>

**✅ 1.2 تنظیمات در application.properties یا application.yml**

اگر از application.properties استفاده می‌کنی:

# برای فعال‌سازی UI (اختیاری)

camunda.bpm.admin-user.id=demo

camunda.bpm.admin-user.password=demo

camunda.bpm.admin-user.first-name=Demo

# غیر فعال‌کردن دیتابیس درونی و اتصال به Camunda مستقل

camunda.bpm.auto-deployment-enabled=false

ما چون Camunda رو با Docker اجرا کردیم، این تنظیم بالا باعث میشه Spring Boot دیگه Engine جداگانه نیاره و سعی کنه از REST API به اون متصل بشه.

**✅ قدم 2: اضافه کردن فایل BPMN**

داخل پوشه src/main/resources/processes/ یک فایل BPMN بساز.

مثلاً: sample-process.bpmn

محتوای ساده برای شروع:

<?xml version="1.0" encoding="UTF-8"?>

<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

id="Definitions\_1"

targetNamespace="http://camunda.org/examples">

<bpmn:process id="sample\_process" name="Sample Process" isExecutable="true">

<bpmn:startEvent id="StartEvent\_1" name="Start">

<bpmn:outgoing>Flow\_1</bpmn:outgoing>

</bpmn:startEvent>

<bpmn:sequenceFlow id="Flow\_1" sourceRef="StartEvent\_1" targetRef="Task\_1" />

<bpmn:userTask id="Task\_1" name="Review Task">

<bpmn:incoming>Flow\_1</bpmn:incoming>

<bpmn:outgoing>Flow\_2</bpmn:outgoing>

</bpmn:userTask>

<bpmn:sequenceFlow id="Flow\_2" sourceRef="Task\_1" targetRef="EndEvent\_1" />

<bpmn:endEvent id="EndEvent\_1" name="End">

<bpmn:incoming>Flow\_2</bpmn:incoming>

</bpmn:endEvent>

</bpmn:process>

</bpmn:definitions>

**✅ قدم 3: ساخت یک Controller برای شروع فرآیند**

@RestController

@RequestMapping("/process")

public class ProcessController {

private final RuntimeService runtimeService;

public ProcessController(RuntimeService runtimeService) {

this.runtimeService = runtimeService;

}

@PostMapping("/start")

public ResponseEntity<String> startProcess() {

ProcessInstance processInstance = runtimeService.startProcessInstanceByKey("sample\_process");

return ResponseEntity.ok("Started process instance with ID: " + processInstance.getId());

}

}

**✅ قدم 4: اجرا و تست**

1. پروژه رو با mvnw spring-boot:run اجرا کن.
2. با Postman یه POST بفرست به:

POST http://localhost:8082/process/start

اگر همه چیز درست باشه، یه Process Instance در Camunda ساخته میشه.

1. توی **Camunda UI در آدرس** http://localhost:8083/camunda/app/tasklist می‌تونی تسک ایجاد شده رو ببینی.

**❓ مرحله بعد**

اگر این بخش بالا درست پیش رفت، مرحله بعدی می‌تونه یکی از اینا باشه:

* اتصال Camunda به Keycloak (احراز هویت کاربران در tasklist)
* اجرای فرم‌های کاربر داخل task
* ارتباط فرآیند با موجودیت‌های واقعی پروژه (مثلاً سفارش، پرداخت و ...)

بذار اول این مرحله رو تا انتها برسونیم، بعد انتخاب کنیم.

می‌خوای من برات فایل sample-process.bpmn رو آماده بفرستم یا خودت می‌سازی؟

## ارسال درخواست به Camunda از طریق اپلیکیشن

حالا که Camunda راه‌اندازی شده، مرحله‌ی بعد اینه که از سمت اپلیکیشن بتونی:

1. ✅ یک process instance رو شروع کنی.
2. 🟡 یک task رو بگیری و complete کنی.
3. (در صورت نیاز) از مفاهیم User Task، Service Task، و Eventها استفاده کنی.

**🔸 گام 1: ساخت یک BPMN ساده**

یک فایل BPMN خیلی ساده با این ساختار بساز (یا از مدل‌سازی Camunda Modeler استفاده کن):

* StartEvent
* UserTask (مثلاً: "Approve Request")
* EndEvent

نام process رو هم approve-process بزار.

فایل رو بذار توی src/main/resources/approve-process.bpmn.

**🔸 گام 2: استارت کردن یک Process Instance با REST**

در Spring Boot یک کنترلر ساده بساز:

@RestController

@RequestMapping("/api/process")

public class ProcessController {

@Autowired

private RuntimeService runtimeService;

@PostMapping("/start")

public ResponseEntity<String> startProcess(@RequestParam String businessKey) {

ProcessInstance instance = runtimeService.startProcessInstanceByKey("approve-process", businessKey);

return ResponseEntity.ok("Process started with ID: " + instance.getId());

}

}

اگر فایل BPMN درسته و اسم process approve-process باشه، این کار:

* یک instance از فرایند ایجاد می‌کنه
* در Camunda Cockpit قابل مشاهده میشه ✅

**🔸 گام 3: لیست کردن Taskها**

@Autowired

private TaskService taskService;

@GetMapping("/tasks")

public List<String> getTasks() {

return taskService.createTaskQuery()

.list()

.stream()

.map(Task::getName)

.collect(Collectors.toList());

}

**🔸 گام 4: انجام Task**

@PostMapping("/complete")

public ResponseEntity<String> completeTask(@RequestParam String taskId) {

taskService.complete(taskId);

return ResponseEntity.ok("Task completed");

}

**🔁 نتیجه این مرحله**

با این کار، از داخل اپلیکیشن Spring Boot:

* یک process رو start می‌کنی
* task هاشو می‌گیری
* و complete می‌کنی

همه چیز همزمان در Camunda Cockpit هم نمایش داده میشه.

# مرحله بعد: اتصال به Keycloak برای احراز هویت کاربران Taskها (با role/assignment)

**اجرای احراز هویت و مجوزدهی (Authentication + Authorization) روی Camunda به کمک Keycloak**

* کاربران باید با توکن Keycloak احراز هویت شوند
* Role-based Authorization انجام شود
* توکن باید هم برای کنترلرهای معمولی و هم برای کنترلرهای مرتبط با Camunda معتبر باشد

**🔜 مراحل پیشنهادی:**

**1. [اگر انجام نشده] نصب Camunda Identity Plugin برای Keycloak**

این پلاگین باعث می‌شود Camunda مستقیماً از Keycloak برای احراز هویت و مجوزدهی استفاده کند.

**2. پیکربندی Camunda برای اتصال به Keycloak:**

* فعال‌سازی Identity Plugin در فایل bpm-platform.xml یا به صورت محیطی (در Docker)
* دادن URL، realm، clientId، clientSecret و …

**3. تعریف کاربران و نقش‌ها در Keycloak:**

* تعریف roleهای camunda-admin, camunda-user, task-assignee و ...
* تعریف گروه‌ها (Groups) در Keycloak (مثلاً managers, approvers)

**4. ایجاد mapping نقش‌ها و گروه‌ها به Resourceهای Camunda:**

* مثلاً کسی که عضو managers هست فقط بتونه تسک‌هایی با candidate group = managers رو بگیره

**5. در پروژه Spring Boot:**

* اطمینان از اینکه توکن JWT تولیدشده از Keycloak برای تمام endpointها از جمله موارد مرتبط با Camunda معتبره
* اضافه کردن role-check در کنترلرهای خودت با @PreAuthorize, @RolesAllowed, یا SecurityContextHolder

خیلی خوب شد که این رو شفاف گفتی 👏  
هدفت الان **استفاده از Keycloak فقط در سمت اپلیکیشن Spring Boot خودته** — و نه در خود UI و هسته Camunda (مثل Cockpit یا Tasklist).  
یعنی می‌خوای فقط:

✅ توکن Keycloak بیاد توی Header  
✅ توی کنترلرهایی مثل /process/start, /task/claim, /task/list که خودت نوشتی  
✅ بررسی بشه آیا یوزر توی یک Role خاص هست یا نه (مثلاً ROLE\_ADMIN, ROLE\_MANAGER و ...)

**بدون نیاز به نصب پلاگین خاصی در Camunda.**

**✅ نتیجه: در این حالت اصلاً نیاز به نصب پلاگین Keycloak برای Camunda نداری**

تو در واقع داری Camunda رو به‌عنوان یک موتور Process استفاده می‌کنی  
و تمام Logic دسترسی و کنترل رو در Spring Boot می‌نویسی. پس:

🟢 توکن Keycloak → در سمت Spring Boot با فیلترهای Security بررسی می‌شه  
🟢 پردازش‌ها با REST API Camunda (یا با ProcessEngine) انجام می‌شن  
🔴 ولی Camunda خودش نمی‌فهمه این یوزر کیه و هیچ احرازی توی Cockpit انجام نمی‌ده (که برات مهم هم نیست)

**🎯 پس چی کار باید بکنی؟**

**1. اطمینان از تنظیم درست Security در Spring Boot**

در SecurityConfig مطمئن شو که توکن JWT به درستی بررسی می‌شه و roles/realm roles درست به دست میان:

@EnableWebSecurity

public class SecurityConfig {

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http

.csrf().disable()

.authorizeHttpRequests(authorize -> authorize

.requestMatchers("/process/start").hasRole("MANAGER")

.requestMatchers("/task/assign").hasRole("ADMIN")

.anyRequest().authenticated()

)

.oauth2ResourceServer(oauth2 -> oauth2

.jwt(jwt -> jwt

.jwtAuthenticationConverter(jwtAuthenticationConverter())

)

);

return http.build();

}

private JwtAuthenticationConverter jwtAuthenticationConverter() {

JwtGrantedAuthoritiesConverter authoritiesConverter = new JwtGrantedAuthoritiesConverter();

authoritiesConverter.setAuthorityPrefix("ROLE\_");

authoritiesConverter.setAuthoritiesClaimName("realm\_access.roles");

JwtAuthenticationConverter converter = new JwtAuthenticationConverter();

converter.setJwtGrantedAuthoritiesConverter(authoritiesConverter);

return converter;

}

}

**2. کنترل دسترسی دقیق در Controllerهایی که نوشتی**

مثلاً:

@RestController

@RequestMapping("/process")

public class ProcessController {

@PreAuthorize("hasRole('MANAGER')")

@PostMapping("/start")

public ResponseEntity<?> startProcess(@RequestParam String key) {

// اجرای فرآیند با Camunda

...

}

@PreAuthorize("hasRole('ADMIN')")

@PostMapping("/task/claim")

public ResponseEntity<?> claimTask(@RequestParam String taskId) {

// گرفتن تسک

...

}

}

**🧠 مزایا و معایب این رویکرد:**

| **مورد** | **وضعیت** |
| --- | --- |
| مدیریت دسترسی‌ها | ✔ کنترل کامل سمت Spring |
| عدم نیاز به پلاگین اضافه | ✔ ساده‌تر |
| کاربرها در Cockpit دیده نمی‌شن | ❌ بله، ولی برات مهم نیست |
| امنیت مبتنی بر توکن JWT | ✔ کاملاً قابل اجرا |
| نمایش کاربران و گروه‌ها در Camunda UI | ❌ نیستن |

**✅ نتیجه نهایی:**

با توجه به هدفت،  
**اصلاً نیازی به پلاگین Keycloak برای Camunda نداری.**  
تنها کاری که لازمه اینه که در اپلیکیشن Spring Boot، توکن JWT که از Keycloak گرفتی رو بخونی و با @PreAuthorize یا @Secured نقش‌ها رو چک کنی.

## ✅ سناریو: فرآیند ثبت سفارش

**مراحل:**

1. **Start Event** — شروع فرآیند سفارش
2. **User Task** — تأیید سفارش توسط اپراتور
3. **Service Task** — ارسال پیامک یا ثبت در سیستم دیگر
4. **End Event** — پایان فرآیند

در ادامه: هم گراف فرآیند رو می‌سازیم، هم Controller جاوا برای استارت و تکمیلش، و بعد می‌ریم سراغ پیشرفته‌تر کردنش.

**🎨 فایل BPMN: order-process.bpmn**

این فایل رو با Camunda Modeler بساز یا همینو مستقیم ذخیره کن و بازش کن:

<bpmn:definitions xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"

xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"

xmlns:camunda="http://camunda.org/schema/1.0/bpmn"

xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"

xmlns:di="http://www.omg.org/spec/DD/20100524/DI"

id="Definitions\_1"

targetNamespace="http://bpmn.io/schema/bpmn">

<bpmn:process id="order\_process" name="Order Process" isExecutable="true">

<bpmn:startEvent id="StartEvent\_Order" name="شروع سفارش">

<bpmn:outgoing>Flow1</bpmn:outgoing>

</bpmn:startEvent>

<bpmn:userTask id="Task\_ConfirmOrder" name="تأیید سفارش" camunda:assignee="operator">

<bpmn:incoming>Flow1</bpmn:incoming>

<bpmn:outgoing>Flow2</bpmn:outgoing>

</bpmn:userTask>

<bpmn:serviceTask id="Task\_SendNotification" name="ارسال پیامک" camunda:class="com.example.workflow.SendSmsDelegate">

<bpmn:incoming>Flow2</bpmn:incoming>

<bpmn:outgoing>Flow3</bpmn:outgoing>

</bpmn:serviceTask>

<bpmn:endEvent id="EndEvent\_Order" name="پایان">

<bpmn:incoming>Flow3</bpmn:incoming>

</bpmn:endEvent>

<bpmn:sequenceFlow id="Flow1" sourceRef="StartEvent\_Order" targetRef="Task\_ConfirmOrder"/>

<bpmn:sequenceFlow id="Flow2" sourceRef="Task\_ConfirmOrder" targetRef="Task\_SendNotification"/>

<bpmn:sequenceFlow id="Flow3" sourceRef="Task\_SendNotification" targetRef="EndEvent\_Order"/>

</bpmn:process>

</bpmn:definitions>

**✅ فایل رو در پروژه قرار بده**

در مسیر:

src/main/resources/bpmn/order-process.bpmn

**✅ کلاس Delegate برای Service Task**

package com.example.workflow;

import org.camunda.bpm.engine.delegate.DelegateExecution;

import org.camunda.bpm.engine.delegate.JavaDelegate;

import org.springframework.stereotype.Component;

@Component

public class SendSmsDelegate implements JavaDelegate {

@Override

public void execute(DelegateExecution execution) {

String customer = (String) execution.getVariable("customer");

System.out.println("📱 پیامک ارسال شد برای مشتری: " + customer);

}

}

**✅ Controller برای استارت و تکمیل تسک**

@RestController

@RequestMapping("/order")

public class OrderController {

private final RuntimeService runtimeService;

private final TaskService taskService;

public OrderController(RuntimeService runtimeService, TaskService taskService) {

this.runtimeService = runtimeService;

this.taskService = taskService;

}

@PostMapping("/start")

public ResponseEntity<String> startOrder(@RequestParam String customer) {

ProcessInstance instance = runtimeService.startProcessInstanceByKey(

"order\_process",

Map.of("customer", customer)

);

return ResponseEntity.ok("Started process ID: " + instance.getId());

}

@GetMapping("/tasks")

public List<String> getTasks() {

return taskService.createTaskQuery()

.taskAssignee("operator")

.list()

.stream()

.map(task -> task.getId() + ": " + task.getName())

.toList();

}

@PostMapping("/complete")

public ResponseEntity<String> completeTask(@RequestParam String taskId) {

taskService.complete(taskId);

return ResponseEntity.ok("Task " + taskId + " completed.");

}

}

**✅ مشاهده در Camunda Cockpit**

1. اپلیکیشن رو اجرا کن.
2. فایل BPMN باید deploy بشه (اگر نشد از کلاس Deployer استفاده کن).
3. از /order/start?customer=Ali فرآیند رو شروع کن.
4. داخل Cockpit یا Tasklist تسک تأیید سفارش برای operator ظاهر می‌شه.
5. با /order/complete?taskId=... تسک رو کامل کن.
6. Delegate اجرا می‌شه (System.out.println یا لاگ پیامک).
7. فرآیند تمام می‌شه و در Cockpit می‌تونی **History و مراحل اجرا** رو ببینی.

# 🔄 چی هست Saga Pattern؟

عالیه، رسیدیم به یکی از مهم‌ترین و حرفه‌ای‌ترین مفاهیم در معماری‌های مدرن مبتنی بر میکروسرویس‌ها:

**🔹 تعریف:**

**Saga** یک **الگوی مدیریت تراکنش توزیع‌شده (Distributed Transaction Management)** هست، مخصوصاً برای معماری Microservices که در اون نمی‌تونیم از تراکنش‌های سنتی (مثلاً با دیتابیس مشترک و ACID) استفاده کنیم.

**🔹 مثال ملموس:**

فرض کن در فرآیند **ثبت سفارش** چند سرویس درگیر هستند:

1. کسر موجودی از انبار (Inventory Service)
2. رزرو مبلغ در کیف پول مشتری (Payment Service)
3. ارسال سفارش به پست (Shipping Service)

اگه یکی از این مراحل شکست بخوره، باید بقیه مراحل لغو بشن (یا "جبران" بشن – **Compensation**).

**🤝 Saga + Camunda + Spring Boot + Keycloak**

**🔸 Camunda چه نقشی داره؟**

Camunda می‌تونه به عنوان **Orchestrator** برای پیاده‌سازی Saga عمل کنه:

* وظیفه اجرای گام‌به‌گام فرآیندها رو بر عهده می‌گیره
* در صورت شکست هر مرحله، به‌صورت خودکار فرآیند جبرانی (compensation) رو اجرا می‌کنه

**🔸 Spring Boot چه می‌کنه؟**

سرویس‌های مستقل (inventory, payment, shipping) می‌تونن با Spring Boot پیاده‌سازی بشن. Camunda هم داخل یکی از این سرویس‌ها هست یا جداست و با REST API با این سرویس‌ها صحبت می‌کنه.

**🔸 Keycloak چه نقشی داره؟**

برای کنترل دسترسی:

* چه کسی حق داره فرآیند رو شروع کنه؟
* چه کسی می‌تونه تسک‌ها رو ببینه یا کامل کنه؟
* توکن Keycloak رو روی APIهای Spring Boot و کنترلرهای Camunda بررسی می‌کنیم (همون کاری که الان داریم انجام می‌دیم)

**💡 دو مدل پیاده‌سازی Saga:**

**1. Orchestration-based Saga (با Camunda) ✅ توصیه‌شده**

* یک سرویس مرکزی (Camunda) کل فرآیند را کنترل می‌کنه.
* ساده‌تر، قابل‌تست‌تر، قابل‌نظارت‌تر

**2. Choreography-based Saga (بدون Camunda)**

* هر سرویس، بعد از اتمام کار خودش، به سرویس بعدی پیام می‌ده
* پیچیده، ولی بدون نیاز به Orchestrator مرکزی

**🎯 سناریویی که پیاده‌سازی می‌کنیم:**

**ثبت سفارش با Saga و Camunda Orchestration**

مراحل:

1. شروع سفارش (Start Event)
2. سرویس Inventory: بررسی و رزرو کالا
3. سرویس Payment: رزرو مبلغ از کیف پول
4. سرویس Shipping: آماده‌سازی ارسال
5. پایان موفق

**اگر هر مرحله‌ای شکست بخوره:**

* اجرا می‌ره به **Compensation Flow**
  + آزاد کردن کالا
  + بازگرداندن پول

**🧩 ارتباط اجزا:**

| **مؤلفه** | **نقش** |
| --- | --- |
| Camunda | اجرای گام‌های سفارش، تصمیم‌گیری در مورد شکست/موفقیت |
| Spring Boot | پیاده‌سازی سرویس‌های مورد استفاده در فرآیند |
| Keycloak | کنترل دسترسی به APIهای سفارش و کاموندا |
| REST API | ارتباط بین Camunda و سرویس‌های دیگر |

بسیار عالی. اینجا وارد یکی از مهم‌ترین مباحث معماری نرم‌افزار می‌شیم. چون می‌خوای از Camunda و Saga تو پروژه واقعی استفاده کنی، درک عمیق مفاهیم **Microservices Architecture** و نگرانی‌هاش واقعاً حیاتی هست.

## 🧱 معماری Microservices – پایه‌ی تفکر مدرن نرم‌افزار

**🟢 1. تعریف Microservices**

معماری میکروسرویس یعنی شکستن یک نرم‌افزار بزرگ (Monolith) به چند سرویس کوچک و مستقل که هرکدام مسئول یک وظیفه خاص هستن.

**ویژگی‌ها:**

* هر سرویس **مستقل deploy می‌شه**
* هر سرویس می‌تونه **دیتابیس اختصاصی** خودش رو داشته باشه
* سرویس‌ها از طریق **HTTP REST** یا **پیام (messaging)** با هم صحبت می‌کنن
* می‌تونن با **زبان‌ها/تکنولوژی‌های مختلف** نوشته بشن (Polyglot)

**🔥 2. مزایای Microservices**

| **مزیت** | **توضیح** |
| --- | --- |
| **مقیاس‌پذیری بهتر** | می‌تونی فقط بخشی از سیستم که فشار داره (مثلاً فقط payment) رو scale کنی |
| **استقلال تیم‌ها** | تیم‌ها می‌تونن مستقل روی سرویس خودشون کار کنن |
| **استقرار سریع‌تر** | چون هر سرویس جداگانه deploy می‌شه |
| **تحمل‌پذیری خطا** | خطای یک سرویس روی کل سیستم اثر نداره (مثلاً اگر سرویس shipping داون بشه، payment همچنان کار می‌کنه) |

**🚨 3. نگرانی‌ها (Challenges) در معماری میکروسرویس**

اینجا جاهایی هست که ما وارد بازی‌های سنگینی می‌شیم و نیاز به ابزارهایی مثل Camunda، Keycloak، API Gateway، Message Broker و ... داریم.

**⚠️ A. Data Consistency (یکپارچگی داده‌ها)**

**مشکل:**

ما نمی‌تونیم از **تراکنش‌های دیتابیس سنتی (ACID)** استفاده کنیم چون هر سرویس دیتابیس خودش رو داره.

**راه‌حل:**

* استفاده از **Saga Pattern** (Orchestration یا Choreography)
* استفاده از **Eventual Consistency** (پذیرفتن تأخیر در همگام‌سازی)

**⚠️ B. Distributed Communication (ارتباط بین سرویس‌ها)**

**مشکل:**

سرویس‌ها باید بتونن با هم صحبت کنن اما چون جدا از هم هستن، این ارتباط پیچیده می‌شه.

**دو نوع ارتباط:**

1. **Synchronous (هم‌زمان)**  ➝ REST API
   * سادست، ولی tightly-coupled و مشکل‌ساز در زمان داون بودن سرویس مقابل
2. **Asynchronous (غیرهم‌زمان)**  ➝ Message Broker (Kafka, RabbitMQ, etc.)
   * مقاوم در برابر خطا و مقیاس‌پذیرتر، ولی پیچیده‌تر برای پیاده‌سازی و دیباگ

**⚠️ C. Distributed Authentication & Authorization**

**مشکل:**

کاربر باید فقط یک‌بار لاگین کنه و بعد به تمام سرویس‌ها دسترسی امن داشته باشه.

**راه‌حل:**

* استفاده از **Identity Provider** مثل **Keycloak**
* استفاده از **JWT Token**
* پیاده‌سازی **Role-Based Access Control (RBAC)**

**⚠️ D. Service Discovery & Routing**

**مشکل:**

سرویس‌ها آدرس ثابتی ندارن، ممکنه پویا باشن (مثلاً در Kubernetes)

**راه‌حل:**

* استفاده از **Service Registry** مثل Eureka, Consul
* استفاده از **API Gateway** مثل NGINX, Kong, Spring Cloud Gateway

**⚠️ E. Observability (نظارت و لاگ‌گیری)**

**مشکل:**

در یک سیستم توزیع‌شده، خطاها و لاگ‌ها بین چند سرویس پخش می‌شن و پیدا کردن منبع مشکل سخته

**راه‌حل:**

* استفاده از **Centralized Logging** (ELK Stack)
* **Distributed Tracing** (Jaeger, Zipkin)
* **Monitoring** (Prometheus + Grafana)

**⚠️ F. Deployment & CI/CD**

**مشکل:**

چند سرویس مستقل باید به‌صورت هماهنگ و اتوماتیک deploy بشن

**راه‌حل:**

* استفاده از **Docker** برای containerization
* استفاده از **Kubernetes** برای orchestration
* استفاده از CI/CD Pipeline مثل GitLab CI, GitHub Actions

**💬 حالا برویم سراغ پاسخ دقیق به سوالاتت:**

**❓ آیا مشکل اصلی در "صحبت کردن میکروسرویس‌ها با هم" است؟**

**بله، کاملاً.**  
یکی از پیچیده‌ترین قسمت‌ها در Microservices، چگونگی ارتباط بین سرویس‌هاست:

* چطور مطمئن بشیم که پیام به مقصد رسیده؟
* اگر سرویس گیرنده داون بود چی؟
* اگه یه درخواست باید چند سرویس رو شامل بشه (مثل ثبت سفارش)، چطور اون رو Atomic مدیریت کنیم؟

**❓ آیا API Gateway و Message Broker در Saga نقش دارند؟**

بله، هر دو بسیار مؤثرند.

**✅ API Gateway:**

نقشش:

* تمام درخواست‌های کلاینت اول به Gateway می‌رن
* اون‌ها رو به سرویس مربوطه هدایت می‌کنه
* احراز هویت و Rate Limiting هم می‌تونه انجام بده

در **Saga**، معمولاً فرآیند از طریق API Gateway شروع می‌شه. مثلاً وقتی کاربر سفارش ثبت می‌کنه، API Gateway درخواست رو به سرویس Order (که شامل Camunda هست) می‌فرسته.

**✅ Message Broker:**

در **Choreography-based Saga** حیاتی هست.

مثلاً:

* Inventory آماده می‌شه ➝ پیام می‌فرسته ➝ Payment پیام رو دریافت می‌کنه ➝ کار خودش رو انجام می‌ده ➝ پیام می‌فرسته ➝ Shipping و ...

در **Orchestration-based Saga** (یعنی با Camunda)، از پیام هم می‌تونی استفاده کنی (مثلاً با External Task) ولی ضروری نیست، چون Camunda خودش orchestrator اصلیه.

**❓ مفاهیم مهم دیگه‌ای هست؟**

بله، چند مفهوم کلیدی هست که حتماً باید بشناسی:

**🔸 Bounded Context**

هر سرویس باید مرز مشخص و واضحی داشته باشه. مثلاً:

* Order Service فقط مربوط به ثبت و مدیریت سفارش است
* Payment کاری به جز پرداخت نداره

این باعث کاهش coupling می‌شه.

**🔸 Database per Service**

هر سرویس دیتابیس خودش رو داره. این یکی از اصول میکروسرویسه. ارتباط بین داده‌های سرویس‌ها از طریق API یا Message انجام می‌شه، نه join کردن بین جدول‌ها.

**🔸 Eventual Consistency**

در میکروسرویس‌ها نباید انتظار داشته باشی که داده‌ها فوری در تمام سرویس‌ها sync باشن. باید بپذیری که تاخیر وجود داره (و این مشکلی نیست اگه مدیریت‌ش درست باشه).

**🔸 Idempotency**

سرویس‌ها باید idempotent باشن؛ یعنی اگر یک درخواست دوبار ارسال شد، نتیجه مشابه باشه.  
این برای جبران خطاها در Saga حیاتی هست.

**🔸 Resilience & Circuit Breaker**

وقتی یک سرویس داون یا کند می‌شه، باید از Circuit Breaker (مثلاً با Resilience4j) استفاده کنیم تا سیستم کل از کار نیفته.

**🔸 Retry Policy**

در ارتباط بین سرویس‌ها (مخصوصاً غیرهم‌زمان)، باید Retry هوشمندانه داشته باشی تا با یک failure ساده، سیستم crash نکنه.

**✅ جمع‌بندی تصویری:**

User → API Gateway → [Order Service (Camunda)] →

→ Inventory Service

→ Payment Service

→ Shipping Service

[Camunda] → orchestrates flow

[Keycloak] → auth for users & services

[Message Broker] → (اختیاری در صورت async comm)

**⏭️ قدم بعدی**

الان آماده‌ایم که یک **Saga واقعی** (مثلاً ثبت سفارش) رو با Camunda شروع کنیم، به‌صورت گام‌به‌گام و با در نظر گرفتن تمام این مفاهیم.

آیا بریم سراغ طراحی **BPMN دیزاین Orchestration Saga** برای سناریو سفارش؟  
یا سوال دیگه‌ای داری؟

## مثال سیستم بانکی از میکروسرویس

تصویر **سیستم واقعی** رو ببینیم و هم بفهمیم چرا باید از میکروسرویس استفاده کنیم، چه **تکنولوژی‌هایی** بر اساس نیاز انتخاب می‌شن، و بعدش می‌تونیم روی یک سناریوی واقعی کار کنیم.

**✅ مثال واقعی: سیستم بانکی مدرن (Modern Banking System)**

فرض کن می‌خوایم یک **سامانه بانک دیجیتال** بسازیم. مشابه چیزی که بانک‌های امروزی مثل "Revolut"، "Monzo" یا حتی در ایران "بلوبانک" و "نئوبانک‌ها" ارائه می‌دن.

**🧩 ساختار ماژولار میکروسرویس‌ها**

**می‌تونیم سیستم رو به میکروسرویس‌های زیر تقسیم کنیم:**

| **Microservice** | **وظیفه** |
| --- | --- |
| **Customer Service** | ثبت‌نام، احراز هویت مشتری، KYC |
| **Account Service** | ایجاد حساب، تغییر وضعیت حساب |
| **Transaction Service** | انتقال وجه، برداشت، واریز |
| **Payment Service** | پرداخت قبض، درگاه پرداخت، کارت‌به‌کارت |
| **Loan Service** | درخواست وام، بررسی اعتبار، تایید/رد |
| **Notification Service** | ارسال پیامک/ایمیل/اعلان |
| **Audit & Logging** | ثبت لاگ‌های امنیتی و عملکردی |
| **Fraud Detection** | بررسی تقلب و رفتار مشکوک |
| **Report Service** | گزارش‌گیری، مانیتورینگ |

**🛠️ تکنولوژی‌ها و ابزارهایی که می‌تونی در این سیستم استفاده کنی**

**📌 1. Backend - Java + Spring Boot**

* سریع، مقیاس‌پذیر، قابل ترکیب با Camunda، Resilience4j، Spring Cloud
* برای هر سرویس جداگانه یک اپلیکیشن Spring Boot مستقل

**📌 2. Camunda 7/8 (Orchestration Engine)**

* مدیریت فرآیندهای پیچیده بین سرویس‌ها (مثل افتتاح حساب یا دریافت وام)
* پیاده‌سازی الگوهای **Saga Orchestration** برای حفظ consistency

**📌 3. Keycloak (IAM و احراز هویت)**

* مدیریت کاربران، سرویس‌ها، Role-based Access
* صدور JWT برای احراز هویت بین سرویس‌ها

**📌 4. API Gateway (مثل Spring Cloud Gateway / Kong / NGINX)**

* نقطه ورودی برای کاربران
* مدیریت مسیرها، احراز هویت، ریت لیمیتینگ، لاگ‌گیری

**📌 5. Message Broker (Kafka یا RabbitMQ)**

* برای ارتباط غیرهم‌زمان بین سرویس‌ها (مخصوصاً در Fraud Detection یا Notification)
* مناسب برای Choreography-based Saga

**📌 6. Database per service (PostgreSQL, MongoDB, Cassandra, ...)**

* هر سرویس دیتابیس جدا داره
* مثلاً:
  + Transaction: PostgreSQL
  + Audit: MongoDB
  + Notification: Redis + PostgreSQL

**📌 7. Monitoring و Logging (ELK, Prometheus, Grafana, Zipkin)**

* ELK برای لاگ مرکزی
* Prometheus + Grafana برای مانیتورینگ متریک‌ها
* Zipkin برای Distributed Tracing

**📌 8. Deployment - Docker + Kubernetes**

* هر سرویس در یک کانتینر جداگانه
* Kubernetes برای orchestration و مقیاس‌پذیری

**📌 9. CI/CD - GitLab CI / Jenkins / GitHub Actions**

* برای تست خودکار، build و deploy

**🔁 مثال یک فرآیند قابل پیاده‌سازی با Camunda**

**فرآیند: درخواست وام (Loan Request)**

فرض کن کاربر درخواست وام می‌ده.

**سرویس‌ها دخیل:**

1. Customer Service → اعتبارسنجی هویت
2. Credit Scoring Service → بررسی امتیاز اعتباری
3. Loan Service → ثبت درخواست وام
4. Fraud Detection → بررسی تقلب
5. Notification Service → اطلاع‌رسانی نتیجه به کاربر

**🧠 اینجا چرا به Orchestration نیاز داریم؟**

چون می‌خوایم یک فرآیند چندمرحله‌ای با قوانین پیچیده، در صورت شکست یک مرحله، قابلیت جبران داشته باشه:

* اگر امتیاز اعتباری ضعیفه → توقف
* اگر رفتار مشکوک (fraud) باشه → reject
* اگر همه تایید بودن → وام ثبت و پول واریز بشه

**🔧 تکنولوژی‌ها در این سناریو**

| **مرحله** | **تکنولوژی پیشنهادی** |
| --- | --- |
| مدیریت فرآیند | Camunda |
| احراز هویت کاربران و API | Keycloak |
| ارتباط بین سرویس‌ها | REST + Kafka |
| اجرای خدمات | Spring Boot |
| وضعیت‌ها و تغییرات مرحله‌ای | Camunda BPMN + Task API |
| مانیتورینگ و گزارش‌گیری | ELK + Prometheus |
| ذخیره داده‌ها | PostgreSQL |
| ارتباط با مشتری | Notification (SMS/Email) Service |

## پیاده سازی یک مثال

عالی، وضعیت پروژه‌ات به‌خوبی مشخصه و ساختار پایه‌گذاری‌شده‌ای داری. حالا با توجه به این وضعیت، بیایم سناریوی **درخواست وام (Loan Application)** رو به‌صورت یک فرآیند واقعی در Camunda طراحی کنیم و از Spring Boot برای مدیریت درخواست‌ها و Keycloak برای کنترل دسترسی استفاده کنیم.

**🎯 سناریوی انتخابی: درخواست وام در یک سیستم بانکی میکروسرویسی**

**📌 توضیح کلی فرآیند (Loan Application Process):**

کاربر درخواست وامی را ثبت می‌کند و این درخواست مراحل زیر را طی می‌کند:

1. **ثبت درخواست وام**
2. **بررسی اعتبار (Credit Check)** توسط سیستم/کارمند
3. **بررسی اسناد (Document Verification)** توسط کارمند
4. **تایید نهایی یا رد درخواست**
5. **پرداخت مبلغ وام در صورت تایید**

**🧩 اجزای مرتبط در پروژه فعلی تو:**

| **مولفه** | **نقش در پروژه** |
| --- | --- |
| **Spring Boot App** | کنترلرها برای شروع فرآیند، گرفتن تسک‌ها، و Complete کردن آن‌ها |
| **Camunda (Docker)** | اجرای فرآیندها و نگهداری History |
| **Keycloak (Docker)** | کنترل احراز هویت کاربران و تعیین نقش‌ها (مثلاً user، clerk، manager) |
| **Postman / Frontend** | برای تست یا فراخوانی endpointها |
| **BPMN File** | مدل فرآیند loan-application.bpmn که در پروژه قرار می‌گیرد |

**📥 نقش‌ها و کاربران پیشنهادی در Keycloak**

| **نقش (Role)** | **توضیحات** |
| --- | --- |
| loan-applicant | کسی که درخواست وام را ثبت می‌کند |
| loan-clerk | کسی که اعتبار و مدارک را بررسی می‌کند |
| loan-manager | تصمیم‌گیرنده نهایی برای تایید یا رد وام |

**🎯 مرحله بعدی: تعریف BPMN برای فرآیند وام**

ما باید یک فایل BPMN طراحی کنیم شامل:

1. **Start Event**
2. **User Task: ثبت اطلاعات وام (Loan Request)**
3. **Service Task: بررسی اعتبار (Credit Check)** ← می‌تونه با Delegate Java پیاده‌سازی بشه
4. **User Task: بررسی مدارک**
5. **User Task: تایید نهایی وام**
6. **Exclusive Gateway: بررسی تایید یا رد**
7. **End Event (Approved/Rejected)**

من می‌تونم برات این فایل BPMN رو تولید کنم یا ساختارش رو همین‌جا بگم که بسازی.

**🔧 کنترلرهای پیشنهادی در Spring Boot**

@PostMapping("/loan/start")

public ResponseEntity<String> startLoanProcess(@RequestHeader("Authorization") String token) {

// فراخوانی camunda برای شروع فرآیند loan-application

}

@GetMapping("/loan/tasks")

public ResponseEntity<List<TaskDto>> getLoanTasks(@RequestHeader("Authorization") String token) {

// گرفتن لیست تسک‌ها بر اساس نقش کاربر

}

@PostMapping("/loan/tasks/{taskId}/complete")

public ResponseEntity<Void> completeLoanTask(@PathVariable String taskId, @RequestBody Map<String, Object> variables) {

// کامل کردن تسک خاص

}

**✅ مرحله بعد**

اگر موافقی:

1. فایل BPMN این فرآیند رو طراحی کنیم (من می‌تونم تولیدش کنم).
2. کنترلرهای مربوط به شروع و اجرای فرآیند رو بنویسیم.
3. تنظیمات Keycloak برای نقش‌ها رو انجام بدیم.
4. تست با Postman.

# مرحله 8- پیاده سازی مثال از درخواست وام

عالی، حالا که وضعیت پروژه کاملاً مشخصه و می‌خوای مرحله‌به‌مرحله سناریوی **درخواست وام** رو با Camunda، Spring Boot و Keycloak جلو ببریم، بریم سراغ مرحله اول:

**✅ مرحله 1: طراحی فایل BPMN فرآیند درخواست وام**

**چون پروژه‌ات Camunda 7.23 هست، باید از Camunda Desktop Modeler استفاده کنی (نه Web Modeler).**

**Camunda 7 و Camunda 8 (SaaS) دو پلتفرم متفاوت هستن:**

| **ویژگی** | **Camunda 7** | **Camunda 8 (SaaS)** |
| --- | --- | --- |
| **زبان اجرا** | **Java** | **Zeebe (Event-driven)** |
| **مدلینگ قابل اجرا** | **Executable BPMN با Delegate Expression** | **Task Definition + Connector Templates** |
| **نوع‌های Task** | **Delegate / Class / Expression** | **Task Type / gRPC / REST Connectors** |
| **ابزاری که استفاده کردی** | **Web Modeler (مربوط به Camunda 8)** | **نیاز به Camunda Modeler دسکتاپ برای 7** |

**چون پروژه‌ات Camunda 7.23 هست، باید از Camunda Desktop Modeler استفاده کنی (نه Web Modeler). در این ابزار، در تنظیمات Service Task گزینه‌هایی مثل Java Class یا Delegate Expression وجود دارن.**

**دانلود Camunda Modeler دسکتاپ**

1. **برو به:  
   https://camunda.com/download/modeler/**
2. **نسخه Desktop رو برای ویندوز (یا لینوکس/مک) دانلود کن.**
3. **فایل .bpmn رو که ساختی، در این ابزار باز کن.**
4. **حالا می‌تونی در Service Task، مقدار Delegate Expression مثل ${checkCreditScoreDelegate} وارد کنی.**

**🎯 هدف:**

ایجاد یک فرآیند کامل "درخواست وام" در قالب یک فایل BPMN که در Camunda قابل مشاهده، اجرا، و کنترل باشد.

**🧩 ساختار کلی فرآیند:**

Start → ثبت درخواست وام → بررسی اعتبار → تأیید مدیر → پرداخت وام → End

**💡 گره‌ها (Nodes) در BPMN:**

1. **Start Event** – شروع فرآیند
2. **User Task**: ثبت درخواست وام
3. **Service Task**: بررسی اعتبار (مثلاً امتیاز اعتباری)
4. **User Task**: تأیید مدیر
5. **Service Task**: پرداخت وام
6. **End Event** – پایان فرآیند

**📁 مرحله اول: ساخت فایل BPMN**

1. در پروژه Spring Boot خود، داخل مسیر:
2. src/main/resources/processes/

یک فایل جدید با نام مثلاً loan-request.bpmn ایجاد کن.

1. از Camunda Modeler استفاده کن( یا Modeler آنلاین) و مراحل بالا رو به صورت گرافیکی طراحی کن:
   * اسم Process رو بزار: loanRequestProcess
   * برای هر Task یک **ID یکتا** مشخص کن. مثلاً:
     + start\_event
     + submit\_application\_task (User Task)
     + check\_credit\_service (Service Task)
     + manager\_approval\_task (User Task)
     + disburse\_loan\_service (Service Task)
     + end\_event
2. برای user task ها باید مشخص کنی که به چه کسی assign می‌شن:
   * مثلاً:
   * <camunda:assignee>${initiator}</camunda:assignee>

یا با نقش:

<camunda:candidateGroups>managers</camunda:candidateGroups>

**🧪 مرحله دوم: Deploy کردن روی Camunda**

با توجه به اینکه پروژه‌ات به‌صورت خودکار فایل‌های داخل پوشه resources/processes/ رو deploy می‌کنه، فقط کافیه فایل رو اونجا بذاری.

برای اطمینان از deploy:

1. اپلیکیشن رو ران کن.
2. برو به **Camunda Cockpit** در مرورگر:
3. http://localhost:8080
4. در قسمت **Processes**، باید فرآیند loanRequestProcess رو ببینی.

✅ مرحله اول تموم شد. حالا اگر آماده‌ای، بریم سراغ مرحله دوم:

**▶️ مرحله دوم: پیاده‌سازی کنترلر در Spring Boot برای شروع فرآیند و مدیریت تسک‌ها**

**🧭 مراحل کامل طراحی فرآیند "درخواست وام"**

**🔹 مرحله 1: ایجاد فرآیند جدید**

1. **New Project → New BPMN Diagram**
2. **یک صفحه سفید برای طراحی فرآیند باز شده.**

**🔹 مرحله 2: تنظیمات اولیه فرآیند**

1. **روی فضای خالی کلیک کن تا کل فرآیند انتخاب شود.**
2. **در پنل سمت راست (Properties Panel):**
   * **Process ID را بگذار: loanRequestProcess**
   * **Name را بگذار: Loan Request Process**
   * **Executable را فعال کن (تیک بزن) — این خیلی مهمه چون Camunda فقط فرآیندهای Executable رو Deploy می‌کنه.**

**🔹 مرحله 3: اضافه کردن عناصر فرآیند**

**اکنون عناصر مختلف فرآیند را اضافه کن:**

**✅ Start Event:**

1. **از پنل سمت چپ، دایره‌ی شروع (Start Event) را بکش وسط صفحه.**
2. **روش کلیک کن و در سمت راست:**
   * **ID را بگذار: startEvent**
   * **Name را بگذار: Start Loan Request**

**✅ User Task: ثبت درخواست وام**

1. **روی Start Event کلیک کن و از آیکون کنارش (فلش)، یک User Task اضافه کن.**
2. **روش کلیک کن و در Properties Panel:**
   * **ID: submitApplicationTask**
   * **Name: Submit Loan Application**
   * **Assignee: applicant**

**✅ Service Task: بررسی اعتبار**

1. **دوباره از User Task قبلی، یک Service Task اضافه کن.**
2. **روش کلیک کن:**
   * **ID: checkCreditService**
   * **Name: Check Credit Score**
   * **Type: در Camunda 7 باید Delegate Expression باشه (مثلاً: ${checkCreditScoreDelegate})**

**✅ User Task: تأیید مدیر**

1. **بعد از بررسی اعتبار، یک User Task بکش.**
2. **تنظیماتش:**
   * **ID: managerApprovalTask**
   * **Name: Manager Approval**
   * **Candidate Groups: managers (نقشی که در Keycloak تعریف کردی)**

**✅ Service Task: پرداخت وام**

1. **یک Service Task اضافه کن.**
2. **تنظیمات:**
   * **ID: disburseLoanService**
   * **Name: Disburse Loan**
   * **Delegate Expression: ${disburseLoanDelegate}**

**✅ End Event**

1. **در انتهای فرآیند یک End Event بکش.**
2. **Name: End Loan Process**

**🔹 مرحله 4: اتصال مراحل با فلش (Sequence Flow)**

**فلش‌هایی بکش به ترتیب زیر:**

**Start → Submit Application → Check Credit Score → Manager Approval → Disburse Loan → End**

**🔹 مرحله 5: ذخیره فایل**

1. **روی آیکون "Save" بالا بزن.**
2. **اسم فایل را بگذار: loan-request.bpmn**
3. **دانلودش کن و داخل پروژه Spring Boot، در مسیر:**
4. **src/main/resources/processes/**

**قرار بده.**

**✅ حالا وقتی پروژه رو اجرا کنی، این فرآیند به صورت خودکار Deploy می‌شه و در Camunda Cockpit قابل مشاهده است.**

**اگر دوست داشتی من همین فایل XML خام رو برات بسازم که مستقیم داخل پروژه‌ات بذاری، فقط بگو.**

**مرحله بعدی هم می‌تونه نوشتن Delegate‌ها و کنترلرها باشه برای شروع فرآیند و تکمیل تسک‌ها. آماده‌ای؟**

# مرحله 9

خیلی عالی، بریم سراغ **مرحله دوم: پیاده‌سازی کنترلر در Spring Boot** برای کنترل فرآیند و تسک‌ها در Camunda 7.23.

**✅ مرحله دوم: کنترلر Spring Boot برای فرآیند و تسک‌ها**

ما در این مرحله 3 بخش اصلی خواهیم داشت:

**🧩 A. شروع فرآیند وام (Start Loan Process)**

**🧩 B. دریافت لیست تسک‌های باز (Get User Tasks)**

**🧩 C. Complete کردن تسک‌ها (Complete Task)**

**🛠️ A. شروع فرآیند وام**

**کنترلر:**

@RestController

@RequestMapping("/loan")

@RequiredArgsConstructor

public class LoanProcessController {

private final RuntimeService runtimeService;

@PostMapping("/start")

public ResponseEntity<String> startLoanProcess(@RequestBody LoanRequestDto dto) {

Map<String, Object> variables = new HashMap<>();

variables.put("applicantName", dto.getApplicantName());

variables.put("amount", dto.getAmount());

variables.put("creditScore", dto.getCreditScore());

ProcessInstance instance = runtimeService.startProcessInstanceByKey("loan-approval-process", variables);

return ResponseEntity.ok("Started process instance with ID: " + instance.getId());

}

}

**DTO:**

@Data

public class LoanRequestDto {

private String applicantName;

private double amount;

private int creditScore;

}

**🛠️ B. دریافت لیست تسک‌های باز برای کاربر**

@GetMapping("/tasks")

public List<TaskDto> getUserTasks(@RequestParam String assignee) {

List<Task> tasks = taskService.createTaskQuery()

.taskAssignee(assignee)

.list();

return tasks.stream()

.map(task -> new TaskDto(task.getId(), task.getName(), task.getProcessInstanceId()))

.collect(Collectors.toList());

}

**TaskDto:**

@AllArgsConstructor

@Data

public class TaskDto {

private String id;

private String name;

private String processInstanceId;

}

**🛠️ C. Complete کردن تسک**

@PostMapping("/tasks/{taskId}/complete")

public ResponseEntity<String> completeTask(@PathVariable String taskId, @RequestBody(required = false) Map<String, Object> variables) {

if (variables == null) {

variables = new HashMap<>();

}

taskService.complete(taskId, variables);

return ResponseEntity.ok("Task " + taskId + " completed successfully.");

}

**📦 وابستگی‌های Inject شده**

در این کلاس از سرویس‌های زیر استفاده کردیم:

* RuntimeService: برای شروع فرآیندها
* TaskService: برای مدیریت تسک‌ها

مطمئن شو که کلاس LoanProcessController به‌درستی annotate شده (@RestController) و سرویس‌ها با @Autowired یا @RequiredArgsConstructor تزریق می‌شن.

**🧪 تست کردن با Postman**

1. **شروع فرآیند:**

POST /loan/start

Content-Type: application/json

{

"applicantName": "Ali",

"amount": 12000,

"creditScore": 650

}

1. **لیست تسک‌ها برای کاربر:**

GET /loan/tasks?assignee=creditOfficer

1. **تکمیل تسک:**

POST /loan/tasks/123456/complete

Content-Type: application/json

{

"approved": true

}

**✅ مرحله بعد**

حالا اگر موافقی بریم سراغ مرحله سوم:

تعریف Delegate های Java برای تسک‌های Camunda (مثل Check Credit Score) و لینک کردن اونها با Service Taskهای موجود در فایل BPMN.

## نکته: خطای پیدا نشدن deletegate مورد نظر

@SpringBootApplication

**@ComponentScan(basePackages = {"com.example.demo", "com.example.delegate"})**

public class DemoApplication {

    public static void main(String[] args) {

        SpringApplication.run(DemoApplication.class, args);

    }

}

## جمع بندی وضعیت فعلی

**مرور وضعیت فعلی پروژه**

1. **زیرساخت‌ها:**
   * Camunda 7.23 روی Docker بالا اومده.
   * Keycloak روی Docker بالا اومده (ولی فعلاً ارتباط مستقیم با Camunda نداره).
   * Spring Boot پروژه اصلیه که Camunda و Keycloak رو مصرف می‌کنه.
2. **Camunda BPMN:**
   * یک فرآیند "Loan Request" طراحی شده.
   * فرآیند شامل مراحل:
     + Start Event
     + Service Task → checkCreditScoreDelegate
     + User Task (برای بررسی توسط کاربر)
     + End Event
   * فایل BPMN در resources پروژه هست و موقع اجرای اپلیکیشن به صورت خودکار Deploy می‌شه.
3. **Spring Boot Controller:**
   * POST /loan/start برای شروع فرآیند.
   * GET /loan/tasks/{processInstanceId} برای دریافت لیست تسک‌ها.
   * POST /loan/tasks/{taskId}/complete برای تکمیل تسک‌ها.
4. **Delegate:**
   * کلاس CheckCreditScoreDelegate به‌عنوان یک Spring Bean پیاده‌سازی شده.
   * این Delegate در Service Task اجرا می‌شه و متغیر creditScore رو ست می‌کنه.
   * در تست‌ها Delegate **در زمان complete شدن User Task اجرا شد**، نه هنگام start.

🔎 این رفتار به خاطر مدلی هست که کشیدی:

* + وقتی فرآیند start می‌شه، Service Task هنوز اجرا نشده، چون Camunda ابتدا Token رو به سمت User Task نفرستاده.
  + وقتی User Task رو complete کردی، Token از User Task به سمت Flow بعدی حرکت کرده و تازه اونجا Delegate اجرا شده.
  + در واقع این موضوع نشون می‌ده که توی BPMN ترتیب Flowها دقیقاً تعیین‌کننده‌ی زمان اجرای Delegate هست.

**✅ نتیجه تست**

* Start فرآیند → Process Instance ساخته شد.
* User Task ایجاد شد → تو لیست تسک‌ها قابل مشاهده.
* Complete کردن User Task → Delegate اجرا شد و متغیر creditScore ست شد.
* لاگ و خروجی درست بود. 🎉

**🔜 مرحله بعد**

حالا که چرخه کامل **start → task → complete → delegate → end** رو داریم، قدم بعدی می‌تونه یکی از این‌ها باشه:

1. **غنی‌تر کردن فرآیند**
   * اضافه کردن چند Service Task دیگه (مثلاً "بررسی درآمد"، "بررسی بدهی"، "تایید مدیر").
   * استفاده از **Gateway** برای تصمیم‌گیری (مثلاً اگر creditScore < 600 → رد درخواست).
2. **ذخیره و مشاهده History**
   * فعال کردن History در Camunda.
   * اضافه کردن یک Controller برای گرفتن تاریخچه‌ی فرآیندها (چه کسی چه تسکی رو تکمیل کرده و چه زمانی).
3. **اتصال با Keycloak** (در لایه‌ی Controller که داریم)
   * طوری که Complete کردن Task فقط توسط یوزر با Role مناسب مجاز باشه.

# مرحله 10 – ایجاد یک exclusive gateway

پس بریم مرحله بعدی که هم ملموس‌تر می‌کنه و هم قدرت Camunda رو نشون می‌ده:  
اضافه کردن **Gateway برای رد یا قبول وام** بر اساس creditScore.

**🔹 مرحله ۱: تغییر BPMN در Camunda Modeler**

1. فایل BPMN (loan-request.bpmn) رو در **Camunda Modeler** باز کن.
2. بعد از Service Task (Check Credit Score) یک **Exclusive Gateway** اضافه کن.
3. از Gateway دو شاخه بگیر:
   * **اگر creditScore >= 600 → Approved**
   * **اگر creditScore < 600 → Rejected**
4. به هر شاخه یک **User Task** اضافه کن:
   * Approve Loan
   * Reject Loan
5. هر دو User Task به یک **End Event** ختم بشن.

ساختار نهایی چیزی شبیه این می‌شه:

(Start) → [Check Credit Score] → (Gateway)

├── creditScore >= 600 → [Approve Loan Task] → (End Approved)

└── creditScore < 600 → [Reject Loan Task] → (End Rejected)

**🔹 مرحله ۲: تعریف Conditions روی Gateway**

روی هر Flow خروجی از Gateway باید شرط بنویسی:

* برای مسیر Approve:

<bpmn:sequenceFlow id="Flow\_approve" sourceRef="gateway" targetRef="approveTask">

<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression"><![CDATA[${creditScore >= 600}]]></bpmn:conditionExpression>

</bpmn:sequenceFlow>

* برای مسیر Reject:

<bpmn:sequenceFlow id="Flow\_reject" sourceRef="gateway" targetRef="rejectTask">

<bpmn:conditionExpression xsi:type="bpmn:tFormalExpression"><![CDATA[${creditScore < 600}]]></bpmn:conditionExpression>

</bpmn:sequenceFlow>

**🔹 مرحله ۳: Delegate تغییر نمی‌کنه**

کلاس CheckCreditScoreDelegate همونطور متغیر creditScore رو ست می‌کنه (بین 300 تا 800).

**🔹 مرحله ۴: تست فرآیند**

1. POST /loan/start رو بزن → یک Process Instance ساخته می‌شه.
2. برو GET /loan/tasks/{processInstanceId} بزن:
   * اگر creditScore >= 600 → تسک **Approve Loan** ساخته می‌شه.
   * اگر creditScore < 600 → تسک **Reject Loan** ساخته می‌شه.
3. POST /loan/tasks/{taskId}/complete رو بزن → فرآیند به End Event می‌رسه.
4. توی Cockpit Camunda هم می‌تونی ببینی کدوم مسیر طی شده.

**🔹 مرحله ۵: آماده‌سازی برای بعد**

بعد از اینکه این رو درست کردی، ما می‌تونیم:

* **History** رو فعال کنیم و لاگ اجرای کل فرآیند رو ببینیم.
* یا Roleها رو از Keycloak وارد کنیم تا مثلاً فقط کاربر با Role loan\_officer بتونه Approve کنه.

## حالا می‌ریم مرحله سوم: مدیریت Gateway تو کد و تست مسیرها

**1. وظیفه‌ی Delegate checkCreditScoreDelegate**

* باید مقدار creditScore رو توی ExecutionContext ست کنه.
* این متغیره که تصمیم Gateway رو تعیین می‌کنه.

@Component("checkCreditScoreDelegate")

public class CheckCreditScoreDelegate implements JavaDelegate {

@Override

public void execute(DelegateExecution execution) throws Exception {

// فرض کنیم امتیاز کاربر رو تصادفی بسازیم (برای تست)

int creditScore = (int) (Math.random() \* 1000);

System.out.println("Calculated Credit Score: " + creditScore);

// متغیر رو تو Context ذخیره می‌کنیم

execution.setVariable("creditScore", creditScore);

}

}

Delegate برای Reject Loan

@Component("rejectLoanDelegate")

public class RejectLoanDelegate implements JavaDelegate {

@Override

public void execute(DelegateExecution execution) throws Exception {

System.out.println("Loan application rejected. Credit Score: "

+ execution.getVariable("creditScore"));

}

}

**Delegate برای Disburse Loan**

@Component("disburseLoanDelegate")

public class DisburseLoanDelegate implements JavaDelegate {

@Override

public void execute(DelegateExecution execution) throws Exception {

System.out.println("Loan approved and disbursed. Credit Score: "

+ execution.getVariable("creditScore"));

}

}

**4. تست با Postman**

1. فرآیند رو استارت کنید (/loan/start).  
   → Camunda یه ProcessInstanceId بهتون برمی‌گردونه.
2. UserTask "Submit Loan Application" رو complete کنید.  
   → بعد از اون، Camunda خودش میره سراغ checkCreditScoreDelegate.
3. Delegate عدد creditScore رو ست می‌کنه.
   * اگر کمتر از 600 بود → Reject Loan Delegate اجرا میشه.
   * اگر بیشتر یا مساوی 600 بود → Disburse Loan Delegate اجرا میشه.
4. در آخر به **End Loan Process** می‌رسیم و می‌تونید History رو تو Camunda Cockpit ببینید.

## مرحله چهارم: اضافه کردن API برای Query کردن History و Status.

این مرحله بهت کمک می‌کنه که:

* بفهمی کدوم ProcessInstanceها در حال اجرا هستن.
* چه Taskهایی فعال هستن.
* فرآیندهای کامل‌شده یا Reject شده رو تو History ببینی.

**✨ پیاده‌سازی API ها در Spring Boot**

**1. گرفتن لیست Process Instance های فعال**

@RestController

@RequestMapping("/loan/history")

public class LoanHistoryController {

@Autowired

private RuntimeService runtimeService;

@Autowired

private HistoryService historyService;

@Autowired

private TaskService taskService;

// دریافت فرآیندهای در حال اجرا

@GetMapping("/running")

public List<Map<String, Object>> getRunningInstances() {

return runtimeService.createProcessInstanceQuery()

.list()

.stream()

.map(pi -> Map.of(

"id", pi.getId(),

"processDefinitionId", pi.getProcessDefinitionId(),

"businessKey", pi.getBusinessKey(),

"isEnded", pi.isEnded()

))

.toList();

}

**2. دریافت Taskهای فعال برای یک Process Instance**

// گرفتن Task های فعال بر اساس ProcessInstanceId

@GetMapping("/{processInstanceId}/tasks")

public List<Map<String, Object>> getActiveTasks(@PathVariable String processInstanceId) {

return taskService.createTaskQuery()

.processInstanceId(processInstanceId)

.list()

.stream()

.map(task -> Map.of(

"id", task.getId(),

"name", task.getName(),

"assignee", task.getAssignee()

))

.toList();

}

**3. گرفتن History کامل (شامل Activity های طی‌شده)**

// گرفتن History کل فرآیند

@GetMapping("/{processInstanceId}/history")

public List<Map<String, Object>> getHistory(@PathVariable String processInstanceId) {

return historyService.createHistoricActivityInstanceQuery()

.processInstanceId(processInstanceId)

.orderByHistoricActivityInstanceStartTime().asc()

.list()

.stream()

.map(act -> Map.of(

"activityId", act.getActivityId(),

"activityName", act.getActivityName(),

"activityType", act.getActivityType(),

"startTime", act.getStartTime(),

"endTime", act.getEndTime()

))

.toList();

}

}

**📌 حالا می‌تونی این API ها رو تو Postman تست کنی:**

1. **استارت فرآیند** → /loan/start
2. **Complete کردن UserTask** → /loan/tasks/{id}/complete
3. **گرفتن Instanceهای در حال اجرا** → /loan/history/running
4. **گرفتن Taskهای فعال** → /loan/history/{processInstanceId}/tasks
5. **گرفتن History کامل** → /loan/history/{processInstanceId}/history

## **سناریوی تست کامل** بچینیم که تو Postman اجراش کنی.

فرض می‌کنیم همون **Loan Request Process** رو داریم که شامل:

* Start → Check Credit Score (ServiceTask) → Exclusive Gateway →
  + اگر رد شد → End →
  + اگر قبول شد → UserTask: Manager Approval → بعدش Disburse Loan → End

**🛠️ سناریوی تست گام‌به‌گام**

**۱. شروع فرآیند**

📌 Endpoint:

POST http://localhost:8082/loan/start

📌 Request Body (JSON):

{

"applicantName": "Ali Rezaei",

"amount": 5000

}

📌 Expected Response:

{

"processInstanceId": "606a4272-8b01-11f0-83e3-9a541b66fc6a",

"message": "Loan process started"

}

**۲. بررسی Process Instanceهای در حال اجرا**

📌 Endpoint:

GET http://localhost:8082/loan/history/running

📌 Expected Response:

[

{

"id": "606a4272-8b01-11f0-83e3-9a541b66fc6a",

"processDefinitionId": "loan\_process:1:9a541b66fc6a",

"businessKey": null,

"isEnded": false

}

]

**۳. بررسی Taskهای فعال برای همون Instance**

📌 Endpoint:

GET http://localhost:8082/loan/history/606a4272-8b01-11f0-83e3-9a541b66fc6a/tasks

📌 Expected Response (مثلاً تسک "Manager Approval" فعال شده):

[

{

"id": "8d32b7f0-8b01-11f0-83e3-9a541b66fc6a",

"name": "Manager Approval",

"assignee": null

}

]

**۴. Complete کردن Task**

📌 Endpoint:

POST http://localhost:8082/loan/tasks/8d32b7f0-8b01-11f0-83e3-9a541b66fc6a/complete

📌 Request Body:

{

"approved": true

}

📌 Expected Response:

{

"message": "Task 8d32b7f0-8b01-11f0-83e3-9a541b66fc6a completed successfully."

}

**۵. بررسی History فرآیند**

📌 Endpoint:

GET http://localhost:8082/loan/history/606a4272-8b01-11f0-83e3-9a541b66fc6a/history

📌 Expected Response (نمونه):

[

{

"activityId": "StartEvent\_1",

"activityName": "Loan Request Started",

"activityType": "startEvent",

"startTime": "2025-09-06T10:30:00.000+0000",

"endTime": "2025-09-06T10:30:01.000+0000"

},

{

"activityId": "checkCreditService",

"activityName": "Check Credit Score",

"activityType": "serviceTask",

"startTime": "2025-09-06T10:30:01.000+0000",

"endTime": "2025-09-06T10:30:01.500+0000"

},

{

"activityId": "managerApprovalTask",

"activityName": "Manager Approval",

"activityType": "userTask",

"startTime": "2025-09-06T10:30:01.500+0000",

"endTime": "2025-09-06T10:31:10.000+0000"

},

{

"activityId": "disburseLoanService",

"activityName": "Disburse Loan",

"activityType": "serviceTask",

"startTime": "2025-09-06T10:31:10.000+0000",

"endTime": "2025-09-06T10:31:10.500+0000"

},

{

"activityId": "EndEvent\_1",

"activityName": "Loan Process Completed",

"activityType": "endEvent",

"startTime": "2025-09-06T10:31:10.500+0000",

"endTime": "2025-09-06T10:31:10.500+0000"

}

]

✅ با این ۵ مرحله، تو Postman می‌تونی کل چرخه‌ی فرآیند رو از **Start → User Task → Complete → End → History** ببینی.

پس تا اینجا یه نمونه‌ی end-to-end ساده از orchestration با **Spring Boot + Camunda + Keycloak** رو جلو بردیم و درست کار کرده.

# تفاوت پیاده سازی پروژه با Saga Orchestration و یا Choreography

## کاربرد Saga Orchestration در این پروژه

کدهای شما یک نمونه عالی از \*\*Saga Orchestration\*\* با استفاده از Camunda است. در ادامه ارتباط آن با الگوی Saga را توضیح می‌دهم:

# 🎯 Saga Orchestration در کد شما

## ۱. \*\*ساختار Saga\*\*:

فرآیند وام شما دقیقاً یک Saga است که شامل چندین مرحله (مرحله‌های تراکنش) می‌باشد:

- \*\*شروع درخواست وام\*\* (Start Event)

- \*\*ثبت درخواست\*\* (User Task)

- \*\*بررسی اعتبار\*\* (Service Task)

- \*\*تصمیم‌گیری\*\* (Exclusive Gateway)

- \*\*تأیید وام\*\* (Service Task) - \*\*Compensating Transaction\*\*

- \*\*رد وام\*\* (Service Task) - \*\*Compensating Transaction\*\*

## ۲. \*\*ارکستراسیون (Orchestration)\*\*:

Camunda به عنوان \*\*Saga Orchestrator\*\* عمل می‌کند:

```java

// Orchestrator کنترل جریان را در دست دارد

ProcessInstance instance = runtimeService.startProcessInstanceByKey(

"loanRequestProcess", variables // 🎼 ارکستراسیون

);

```

## ۳. \*\*مراحل Saga\*\*:

### 🔹 Transaction 1: ثبت درخواست

```xml

<userTask id="submitApplicationTask" name="Submit Loan Application"/>

```

### 🔹 Transaction 2: بررسی اعتبار

```java

@Component("checkCreditScoreDelegate")

public class CheckCreditScoreDelegate implements JavaDelegate {

public void execute(DelegateExecution execution) {

// منطق کسب وکار

execution.setVariable("creditScore", creditScore);

}

}

```

### 🔹 Decision Point: دروازه تصمیم‌گیری

```xml

<exclusiveGateway id="Gateway\_0v1zb74">

<conditionExpression>${creditScore >= 600}</conditionExpression>

<conditionExpression>${creditScore < 600}</conditionExpression>

</exclusiveGateway>

```

### 🔹 Compensating Transactions: تراکنش‌های جبرانی

\*\*اگر وام تأیید شود:\*\*

```java

@Component("disburseLoanDelegate")

public class DisburseLoanDelegate implements JavaDelegate {

public void execute(DelegateExecution execution){

// تراکنش اصلی - پرداخت وام

}

}

```

\*\*اگر وام رد شود:\*\*

```java

@Component("rejectLoanDelegate")

public class RejectLoanDelegate implements JavaDelegate {

public void execute(DelegateExecution execution) {

// تراکنش جبرانی - اطلاع‌رسانی رد وام

}

}

```

## ۴. \*\*مزایای Saga Orchestration در این پیاده‌سازی\*\*:

### ✅ \*\*یکپارچگی داده‌ها\*\*:

- اگر بررسی اعتبار失敗 شود، کل فرآیند متوقف می‌شود

- هیچ پرداختی بدون تأیید اعتبار انجام نمی‌شود

### ✅ \*\*قابلیت رهگیری\*\*:

```java

@GetMapping("/tasks")

public List<TaskDto> getUserTasks(@RequestParam String assignee) {

// رهگیری وضعیت Saga

}

```

### ✅ \*\*مدیریت خطا\*\*:

- خطاها در سطح فرآیند مدیریت می‌شوند

- امکان retry و compensation وجود دارد

### ✅ \*\*انعطاف‌پذیری\*\*:

- به راحتی می‌توان مراحل جدید اضافه کرد

- منطق کسب وکار از گردش کار جدا شده است

## ۵. \*\*مقایسه با Choreography\*\*:

| ویژگی | Orchestration (کد شما) | Choreography |

|--------|------------------------|-------------|

| \*\*کنترل\*\* | متمرکز (Camunda) | توزیع شده |

| \*\*پیچیدگی\*\* | مدیریت آسان‌تر | پیچیده‌تر |

| \*\*رهگیری\*\* | آسان | سخت‌تر |

| \*\*انعطاف\*\* | تغییرات آسان | تغییرات سخت |

## ۶. \*\*سناریوهای واقعی\*\*:

### 💳 پرداخت آنلاین:

1. بررسی موجودی (Transaction)

2. کسر مبلغ (Transaction)

3. واریز به مقصد (Transaction)

4. اگر خطایی رخ داد: برگشت مبلغ (Compensation)

### 🛒 سفارش آنلاین:

1. ثبت سفارش

2. کاهش موجودی

3. ایجاد حواله حمل

4. اگر موجودی نبود: لغو سفارش (Compensation)

## ۷. \*\*نکات تکمیلی\*\*:

### 🔧 بهبود کد:

```java

// اضافه کردن لغو تراکنش‌ها

@Component("compensateApplicationDelegate")

public class CompensateApplicationDelegate implements JavaDelegate {

public void execute(DelegateExecution execution) {

// لغو درخواست و پاک کردن داده‌ها

}

}

```

### 📊 مانیتورینگ:

```java

@GetMapping("/history/{processInstanceId}")

public List<Map<String, Object>> getHistory(@PathVariable String processInstanceId) {

// مانیتورینگ وضعیت Saga

}

```

## 🎯 نتیجه‌گیری:

کد شما یک \*\*Saga Orchestration\*\* کامل است که:

1. \*\*✅ تراکنش‌های توزیع شده\*\* را مدیریت می‌کند

2. \*\*✅ یکپارچگی داده‌ها\*\* را تضمین می‌کند

3. \*\*✅补偿 تراکنش‌ها\*\* دارد

4. \*\*✅ قابلیت رهگیری\*\* دارد

5. \*\*✅ انعطاف‌پذیر\*\* و قابل توسعه است

این دقیقاً همان الگویی است که در سیستم‌های میکروسرویس برای مدیریت تراکنش‌های توزیع شده استفاده می‌شود! 🚀

## نحوه پیاده سازی با Choreography

عالی سوال پرسیدی! برای پیاده‌سازی \*\*Saga Choreography\*\* به جای Orchestration، باید ساختار کاملاً متفاوتی داشته باشیم. در اینجا مقایسه و روش پیاده‌سازی را توضیح می‌دهم:

# 🎭 Saga Choreography vs Orchestration

## 🔄 تفاوت اصلی:

|  |  |  |
| --- | --- | --- |
|  | **Orchestration (کد فعلی شما)** | **Choreography** |
| **کنترل** | متمرکز (Camunda) | توزیع شده |
| **مسئولیت** | Orcherator تصمیم می‌گیرد | هر سرویس خودش تصمیم می‌گیرد |
| **ارتباط** | Command-based | Event-based |
| **وابستگی** | به Camunda وابسته است | مستقل از orchestration engine |

# 🏗️ پیاده‌سازی Choreography

## ۱. \*\*سرویس‌های مستقل\*\*:

```java

// Service 1: Application Service

@RestController

@RequestMapping("/applications")

public class ApplicationService {

@PostMapping

public ResponseEntity<String> submitApplication(@RequestBody LoanApplication application) {

//ذخیره درخواست

applicationRepository.save(application);

//انتشار event

kafkaTemplate.send("loan-application-submitted",

new ApplicationSubmittedEvent(application.getId(), application.getApplicantName(), application.getAmount()));

return ResponseEntity.ok("Application submitted");

}

}

```

## ۲. \*\*Event Classes\*\*:

```java

// 📨 Event Classes

public class ApplicationSubmittedEvent {

private String applicationId;

private String applicantName;

private double amount;

// getters, setters, constructor

}

public class CreditScoreCheckedEvent {

private String applicationId;

private int creditScore;

private boolean approved;

// getters, setters, constructor

}

public class LoanDisbursedEvent {

private String applicationId;

private double amount;

// getters, setters, constructor

}

public class LoanRejectedEvent {

private String applicationId;

private String reason;

// getters, setters, constructor

}

```

## ۳. \*\*Credit Check Service\*\*:

```java

// 📊 Service 2: Credit Check Service

@Service

public class CreditCheckService {

@KafkaListener(topics = "loan-application-submitted")

public void handleApplicationSubmitted(ApplicationSubmittedEvent event) {

//بررسی اعتبار

int creditScore = (int) (Math.random() \* 1000);

boolean approved = creditScore >= 600;

//انتشار event نتیجه

if (approved) }

kafkaTemplate.send("credit-check-approved",

new CreditScoreCheckedEvent(event.getApplicationId(), creditScore, true));

} else{

kafkaTemplate.send("credit-check-rejected",

new CreditScoreCheckedEvent(event.getApplicationId(), creditScore, false));

}

}

}

```

## ۴. \*\*Loan Service\*\*:

```java

// 💰 Service 3: Loan Service

@Service

public class LoanService {

@KafkaListener(topics = "credit-check-approved")

public void handleCreditApproved(CreditScoreCheckedEvent event) {

// پرداخت وام

System.out.println("Disbursing loan for application: " + event.getApplicationId());

// انتشار event پرداخت

kafkaTemplate.send("loan-disbursed",

new LoanDisbursedEvent(event.getApplicationId(), /\* amount \*/));

}

@KafkaListener(topics = "credit-check-rejected")

public void handleCreditRejected(CreditScoreCheckedEvent event) {

// رد وام

System.out.println("Rejecting loan for application: " + event.getApplicationId());

// انتشار event رد

kafkaTemplate.send("loan-rejected",

new LoanRejectedEvent(event.getApplicationId(), "Credit score too low: " + event.getCreditScore()));

}

}

```

## ۵. \*\*Notification Service\*\*:

```java

// 📧 Service 4: Notification Service

@Service

public class NotificationService {

@KafkaListener(topics = "loan-disbursed")

public void handleLoanDisbursed(LoanDisbursedEvent event) {

// ارسال ایمیل تأیید

System.out.println("Sending disbursement notification for: " + event.getApplicationId());

}

@KafkaListener(topics = "loan-rejected")

public void handleLoanRejected(LoanRejectedEvent event) {

// ارسال ایمیل رد

System.out.println("Sending rejection notification for: " + event.getApplicationId());

}

}

```

# 📊 نمودار جریان Choreography:

```

Application Service

│

↓ (ApplicationSubmittedEvent)

Credit Check Service

│

├─→ (CreditScoreCheckedEvent - approved) → Loan Service → (LoanDisbursedEvent) → Notification Service

│

└─→ (CreditScoreCheckedEvent - rejected) → Loan Service → (LoanRejectedEvent) → Notification Service

```

# ⚖️ مقایسه کامل:

## ✅ مزایای Choreography:

- \*\*عدم وابستگی\*\* به orchestration engine

- \*\*مقیاس‌پذیری\*\* بهتر

- \*\*انعطاف‌پذیری\*\* بیشتر

- \*\*کاهش single point of failure\*\*

## ❌ معایب Choreography:

- \*\*پیچیدگی debug\*\* بیشتر

- \*\*ردیابی فرآیند\*\* سخت‌تر

- \*\*مدیریت خطا\*\* پیچیده‌تر

- \*\*پیچیدگی درک\*\* جریان کلی

## 🔧 ابزارهای مورد نیاز برای Choreography:

```xml

<!-- Kafka برای event streaming -->

<dependency>

<groupId>org.springframework.kafka</groupId>

<artifactId>spring-kafka</artifactId>

</dependency>

<!-- Optional: Axon Framework برای Saga -->

<dependency>

<groupId>org.axonframework</groupId>

<artifactId>axon-spring-boot-starter</artifactId>

</dependency>

```

## 🎯 چه زمانی از Choreography استفاده کنیم؟

- وقتی \*\*سرویس‌ها بسیار مستقل\*\* هستند

- وقتی \*\*نیاز به مقیاس‌پذیری بالا\*\* داریم

- وقتی \*\*می‌خواهیم از vendor lock-in جلوگیری کنیم\*\*

- وقتی \*\*تیم‌های مختلف\*\* روی سرویس‌ها کار می‌کنند

## 🎯 چه زمانی از Orchestration استفاده کنیم؟

- وقتی \*\*نیاز به visibility و control کامل\*\* داریم

- وقتی \*\*مدیریت خطاهای پیچیده\*\* نیاز است

- وقتی \*\*ردیابی فرآیند\*\* مهم است

- وقتی \*\*تغییرات مکرر\*\* در flow داریم

# 💡 نتیجه‌گیری:

کد فعلی شما با \*\*Orchestration\*\* عالی است برای:

- ✅ سیستم‌های متمرکز

- ✅ فرآیندهای business-critical

- ✅ نیاز به مانیتورینگ و کنترل

\*\*Choreography\*\* بهتر است وقتی:

- 🚀 معماری میکروسرویس کاملاً توزیع شده داریم

- 🚀 نیاز به استقلال کامل سرویس‌ها داریم

- 🚀 مقیاس‌پذیری بسیار بالا مهم است

هر دو الگو valid هستند و انتخاب بستگی به نیازهای خاص پروژه دارد! 🎯

## آیا کاموندا فقط به درد saga orchestration میخوره؟

\*\*کاموندا عمدتاً برای پیاده‌سازی Saga Orchestration طراحی شده است\*\*، اما می‌توان تا حدی Choreography را نیز شبیه‌سازی کرد. در ادامه توضیح کامل می‌دهم:

# 🎯 کاموندا و Saga Patterns

## ۱. \*\*ذات کاموندا: Orchestration\*\* ✅

کاموندا ذاتاً یک \*\*Orchestration Engine\*\* است:

```java

//典型的 Orchestration

ProcessInstance instance = runtimeService.startProcessInstanceByKey(

"loanRequestProcess", variables // 🎼 کاموندا نقش Orchestrator را بازی می‌کند

);

```

## ۲. \*\*چرا کاموندا برای Orchestration مناسب است\*\*:

### ✅ مزایای کاموندا در Orchestration:

- \*\*Visibility کامل\*\* بر کل فرآیند

- \*\*مدیریت متمرکز\*\* state و flow

- \*\*ابزارهای monitoring\*\* built-in

- \*\*مدیریت خطا\*\* یکپارچه

- \*\*compensation\*\* داخلی

### ❌ محدودیت‌های کاموندا برای Choreography:

- \*\*وابستگی به کاموندا\*\* به عنوان single point of failure

- \*\*کمی overhead\*\* برای سرویس‌های بسیار سبک

- \*\*vendor lock-in\*\* نسبی

## ۳. \*\*شبیه‌سازی Choreography با کاموندا\*\* ⚡

اگر بخواهید با کاموندا Choreography-like داشته باشید:

### روش ۱: استفاده از Message Events

```xml

<!-- Event-Based Gateway برای شبیه‌سازی Choreography -->

<intermediateThrowEvent id="applicationSubmittedEvent">

<messageEventDefinition messageRef="ApplicationSubmittedMessage"/>

</intermediateThrowEvent>

<intermediateCatchEvent id="waitForCreditCheck">

<messageEventDefinition messageRef="CreditCheckResultMessage"/>

</intermediateCatchEvent>

```

### روش ۲: ترکیب با Event Bus

```java

@Component

public class KafkaCamundaBridge {

@Autowired

private RuntimeService runtimeService;

@KafkaListener(topics = "credit-check-completed")

public void handleCreditCheckResult(CreditCheckEvent event) {

// correlate message به کاموندا

runtimeService.createMessageCorrelation("CreditCheckResultMessage")

.processInstanceBusinessKey(event.getApplicationId())

.setVariable("creditScore", event.getScore())

.correlate();

}

}

```

## ۴. \*\*مقایسه دو رویکرد با کاموندا\*\*:

### 🎼 Pure Orchestration (توصیه شده با کاموندا)

```java

// کاموندا کاملاً کنترل می‌کند

ProcessInstance process = runtimeService.startProcessInstanceByKey("sagaProcess");

// تمام تصمیم‌گیری‌ها در کاموندا

```

### 💃 Hybrid Approach (ترکیبی)

```java

// سرویس‌ها events منتشر می‌کنند

kafkaTemplate.send("process-started", event);

// کاموندا فقط بر اساس events تصمیم می‌گیرد

runtimeService.correlateMessage("process-event", businessKey, variables);

```

## ۵. \*\*ابزارهای بهتر برای Choreography\*\*:

اگر واقعاً می‌خواهید \*\*Pure Choreography\*\* داشته باشید، این ابزارها بهتر هستند:

### 🚀 Event-Driven Frameworks:

- \*\*Apache Kafka + Spring Cloud Stream\*\*

- \*\*RabbitMQ with Event Sourcing\*\*

- \*\*Axon Framework\*\* (تخصصی Saga)

- \*\*NServiceBus\*\* (برای .NET)

### 📊 تفاوت معماری:

```mermaid

graph LR

A[Orchestration] --> B[کاموندا کنترل می‌کند]

C[Choreography] --> D[سرویس‌ها مستقل تصمیم می‌گیرند]

```

## ۶. \*\*جمع‌بندی: کاموندا برای چه چیزی مناسب است؟\*\*

### ✅ کاموندا عالی برای:

- \*\*Business Process Orchestration\*\*

- \*\*Long-Running Transactions\*\*

- \*\*Complex Workflows\*\*

- \*\*Cases needing visibility\*\*

### ⚡ کاموندا کمتر مناسب برای:

- \*\*Highly decentralized systems\*\*

- \*\*Extremely high-throughput event streaming\*\*

- \*\*When you want zero vendor dependency\*\*

## 🎯 توصیه نهایی:

- اگر \*\*نیاز به کنترل و visibility\*\* دارید → \*\*Orchestration با کاموندا\*\* ✅

- اگر \*\*استقلال کامل سرویس‌ها\*\* اولویت است → \*\*Pure Choreography با Kafka\*\* 🚀

- اگر \*\*ترکیبی\*\* می‌خواهید → \*\*کاموندا + Event Bridge\*\* ⚡

\*\*کاموندا یک Orchestration Engine عالی است\*\*، اما اگر واقعاً Choreography می‌خواهید، بهتر است از ابزارهای event-native استفاده کنید!

# آماده کردن اسلایدها

من پیشنهاد می‌کنم roadmap رو به این شکل ببریم جلو:

**1. گسترش فرآیند Loan Request**

* اضافه کردن چند **ServiceTask/UserTask** دیگه (مثلا استعلام از سرویس‌های خارجی، امضای دیجیتال، آپلود مدرک و ...)
* یاد گرفتن **Parallel Gateway** و **Inclusive Gateway** برای سناریوهای واقعی‌تر.

**2. پیاده‌سازی الگوهای Orchestration**

* معرفی **Saga Pattern** با Camunda برای مدیریت تراکنش‌های توزیع‌شده.
* پیاده‌سازی یک فرآیند که شامل چند میکروسرویس باشه (مثلا Payment Service، Notification Service).
* مدیریت **compensation** (برگشت تراکنش وقتی خطا پیش اومد).

**3. Query و Monitoring حرفه‌ای‌تر**

* اضافه کردن API برای گرفتن **Process Instance State**
* اضافه کردن API برای گرفتن **Variables** ذخیره‌شده داخل هر فرآیند
* استفاده از Camunda Cockpit برای تحلیل فرآیندها.

**4. امنیت و Keycloak**

* محدود کردن UserTaskها بر اساس نقش کاربر (Role-based Assignment).
* اضافه کردن delegation برای assign شدن task به role درست.

**5. Integration با ابزارهای اطراف**

* اضافه کردن **Message Event** و اتصال به Message Broker (Kafka / RabbitMQ).
* نمایش dashboard ساده از وضعیت فرآیندها (با chart یا جدول).

**🔹 برای ارائه به سرپرستت**

سرپرستت گفته یک سمپل تستی برای orchestration آماده کنی و براش پرزنت بدی.  
پیشنهاد می‌کنم ارائه‌ات شامل ۴ بخش باشه:

**1. مقدمه ساده**

* Camunda چی هست و چرا برای orchestration مناسبه.
* تفاوت choreography و orchestration.

**2. معماری نمونه‌ای که ساختی**

* یه اسلاید ساده که نشون بده:
  + Spring Boot API
  + Camunda
  + Keycloak
  + ارتباطشون با هم (با Docker Compose بالا اومدن)

**3. دمو**

* شروع یک فرآیند (Loan Request)
* نمایش Task فعال
* Complete کردن Task با Postman
* نمایش تاریخچه فرآیند (History API)
* نشون دادن گرافیک پروسه در Camunda Cockpit

**4. نکات پایانی**

* چرا این روش برای پروژه‌های آینده مفیده (انعطاف، مدیریت خطا، مانیتورینگ بهتر).
* roadmap خیلی کوتاه (مثلا مرحله بعدی: Saga، Integration با سرویس‌های خارجی).

👉 به نظرم اگر همین Loan Request ساده رو که ساختیم پرزنت کنی، خیلی خوبه چون end-to-end هست و همه چی رو پوشش داده.  
ولی اگر بخوای impression قوی‌تر بذاری، یه **compensation ساده (یعنی rollback)** هم اضافه کن که مثلا اگر disburse loan موفق نشد، یه تسک "Rollback Transaction" اجرا بشه.

# سوالات بعد از ارائه و جواب های ai

تو ارائه چند تا سوال از من پرسیدن 1- برای جبران ساز کاری کردی که گفتم نه 2- نحوه ارتباط camunda با application ی که با spring boot نوشتی چطوری هست؟ مثلا الان برناه از کجا میفهمه که باید چه trigger ی رو اجرا کنه؟ و یا آیا camunda engine هم به برنامه دستور میده یا فقط برنامه به اون دستور میده؟ جبران ساز: خطاهایی که میخوره کدام خطا اکشن rollback رو trigger میکنه و کدام خطا کل فرآیند رو failed میکنه پچیک دریافت کارمزد: جبران ساز هم باشه اگر یعنی کارمزد گرفتیم و خطا دور بشه کارمزد رو برگدانیم چطوری state رو پیگیری کنیم؟ (وضعیت های میانی) -< در حالی هستند که مغایرت خوردند امکان پیگیری داشته باشیم. چالش ها چیست؟

این سوال‌هایی که ازت پرسیدن خیلی به نقطه‌های حساس **Orchestration** و **Compensation** در معماری میکروسرویس‌ها مربوط می‌شه. من مرحله به مرحله جواب می‌دم تا برای پرزنت بعدی آماده باشی:

**1. جبران‌ساز (Compensation Mechanism)**

* **چی هست؟**  
  در میکروسرویس‌ها وقتی چند اکشن پشت سر هم انجام می‌دی (مثلاً Debit از حساب، گرفتن کارمزد، واریز به حساب مقصد)، اگه یکی از این‌ها Fail بشه باید قبلی‌ها رو **Rollback منطقی** کنی (نه Rollback دیتابیس، چون سرویس‌ها مستقل هستن).
* **دو حالت اصلی:**
  1. **Recoverable Error** → خطاهای موقت (مثلاً Timeout، مشکل شبکه).  
     → این خطاها معمولاً Retry می‌شن (Camunda می‌تونه Retry policy داشته باشه).
  2. **Business Error** → خطاهای منطقی (مثلاً موجودی کافی نیست).  
     → اینجا جبران‌ساز (Compensation) اجرا می‌شه. مثلاً اگر کارمزد برداشت شده بود باید Refund بشه.
* **مثال کارمزد (Compensation Task):**
  1. Task 1: برداشت مبلغ اصلی
  2. Task 2: برداشت کارمزد
  3. Task 3: انتقال به حساب مقصد
  4. اگر Task 3 Fail شد → باید Task 2 (Refund Fee) و Task 1 (Refund Amount) اجرا بشه.

**2. نحوه ارتباط Camunda با Application (Spring Boot)**

* **از سمت برنامه → Camunda:**
  + برنامه با Camunda API صحبت می‌کنه (runtimeService.startProcessInstanceByKey, taskService.complete, …).
  + یعنی **Application تصمیم می‌گیره** کی فرآیند رو شروع کنه یا Task رو Complete کنه.
* **از سمت Camunda → برنامه:**
  + وقتی در مدل BPMN یک **Service Task** تعریف می‌کنی با Delegate یا External Task:
    - **Delegate (Java Delegate):** کدی درون همون Spring Boot پروژه اجرا می‌شه (Sync call).
    - **External Task:** Camunda یک Task در لیست ایجاد می‌کنه و برنامه‌های خارجی Poll می‌کنن و وقتی Task اومد، اجرا می‌کنن (Async, Decoupled).

پس هم Application می‌تونه Camunda رو Trigger کنه، هم Camunda می‌تونه به Application بگه «این Task باید انجام بشه».

**3. State Tracking (پیگیری وضعیت میانی)**

* **Camunda History**: همه Eventها و Activityها در جدول History ذخیره می‌شن.
  + API → /history/process-instance و /history/activity-instance
  + می‌تونی بدون اینکه دستی Log بگیری، وضعیت میانی رو Query کنی.
* **Variables**: در هر مرحله متغیرها ذخیره می‌شن (مثل creditScore, feeStatus) → قابل Query.
* **Business-Level States:** می‌تونی برای هر مرحله یک UserTask یا Marker بذاری که در UI گزارش بشه (مثلاً "در حال بررسی مغایرت"، "منتظر پرداخت کارمزد").

**4. چالش‌ها**

* **همگام‌سازی (Synchronization):** اگه چند سرویس همزمان روی یک موجودیت کار کنن (مثلاً دو وام روی یک حساب)، مغایرت پیش میاد.
* **Idempotency:** هر سرویس باید Idempotent باشه (مثلاً اگر Refund دوبار صدا شد، دوبار پول برنگردونه).
* **Partial Failure:** بعضی سرویس‌ها موفق می‌شن، بعضی Fail → نیاز به Compensation دقیق.
* **Visibility:** مدیر سیستم باید همیشه بتونه بفهمه الان پروسه در چه وضعیتی هست (Camunda Cockpit کمک می‌کنه).
* **Security:** نقش‌ها و مجوزها (Keycloak) باید روی Taskها و APIها enforce بشن.

📌 پیشنهاد من برای پرزنت:

1. یک **BPMN ساده از سناریوی Fee + Loan Transfer** بکش (با Compensation Task).
2. توضیح بده:
   * Recoverable vs Business Errors
   * Compensation Flow (Refund)
   * ارتباط App ↔ Camunda (Delegate vs External Task)
   * Query کردن History برای دیدن State.
3. چالش‌ها و Best Practiceها رو روی اسلاید آخر لیست کن.

می‌خوای من یه **BPMN دیاگرام جبران‌ساز با Task Refund Fee** برات آماده کنم تا تو پرزنت بذاری؟

## جبران ساز

ببین ما واقعا تو شرایط واقعی بهش نیاز داریم

1- از محیط بیرونی مثلا یه third party اول سیستم cheque orchestration رو کال میکنه

2- بعد از داخل پروژه cheque orchestration اول سیستم کارمزد رو کال میکنیم و کارمزد رو از حساب طرف کم میکنیم (ما یه میکروسرویس کارمزد داریم که مستقل هست و دیتابیس مستقل داره)

3- بعد میکروسرویس پیچک بانک مرکزی رو کال میکنیم.

4- بعد سرویس چک داخلی رو کال میکنیم و اینجا چند تا عملیات در قالب یه تراکنش انجام میشه و یکی از زیر مجموعه های تراکنش مثلا ثبت دیتا تو دیتابیس عملیات چک به خطا میخوره و کل تراکنش رول بک میشه

5- انتظار داریم که کارمزد به حساب کاربر بر گرده و اگر پیچک بانک مرکزی هم سرویسی داره که بشه باهاش رول بک کرد و نیاز باشه قابل فراخوانی باشه

تو همه این حالت ها یه مورد هم که نیاز داریم این هست که state های میانی رو هم حفظ کنیم

راهکار چی هست؟

میخوام سناریویی که جلو میریم رو بر مبنای این بچینیم

عالی — سناریوی واقعی و ضروری‌ای تعریف کردی. می‌سازیمش طوری که **قابل پیاده‌سازی، قابل تست و قابل ارائه** باشه. پایین هم **طراحی مفهومی (BPMN + رفتارها)** هم **راهنمای پیاده‌سازی مرحله‌به‌مرحله** و هم **نمونه کُد / الگوها** رو آوردم تا ابیات بعدی رو مستقیم پیاده‌سازی کنی.

**🔥 مسئله خلاصه‌شده**

جریان تراکنش چک:

1. یک سرویس خارجی (third-party) درخواست orchestration رو به ما ارسال می‌کنه.
2. سرویس ما ابتدا **کارمزد** را می‌کِشد (microservice مستقل با DB خودش).
3. سپس به **بانک مرکزی (Peychak)** سرویس می‌زند.
4. سپس **سرویس داخلیِ عملیات چک** را اجرا می‌کنیم — داخل این سرویس چند عملیات دیتابیسی انجام میشه و یکی از آنها خطا می‌زند → transaction محلی rollback می‌شود.
5. انتظار: وقتی خطا در مرحله 4 رخ داد، کارمزد باید برگشت داده شود؛ همچنین اگر بانک مرکزی امکان rollback دارد باید آنرا فراخوانی کنیم.
6. در تمام مراحل باید **وضعیت‌های میانی** (state) قابل ردیابی و audit باشند.

**کلیت راه‌حل پیشنهادی (معماری)**

* از **Camunda** به‌عنوان Orchestrator استفاده می‌کنیم.
* هر interaction با یک میکروسرویس را به‌صورت **External Task** یا **Service Task (via REST / External Task)** مدل می‌کنیم. (External Task توصیه می‌شود چون میکروسرویس‌ها مستقل‌اند.)
* برای عملیات‌هایی که باید قابل جبران باشند، برای هر عملیات **Compensation Handler** تعریف می‌کنیم (مثلاً refundFee, peychakRollback, rollbackInternalOperations).
* وقتی مرحله‌ای Fail شد (Business Error یا unrecoverable)، Camunda **Compensate** را اجرا می‌کند تا جبران‌ها را در ترتیب معکوس انجام دهد.
* برای خطاهای موقتی از **Retries** و الگوهای backoff استفاده می‌کنیم؛ اگر پس از تلاش‌‌ها موفق نشد، به حالت خطای نهایی (failed) و یا Compensation ورود می‌کنیم.
* برای state tracking از **Process Variables + Camunda History** و همچنین یک جدول Audit در میکروسرویس‌ها استفاده می‌کنیم (correlationId).

**BPMN پیشنهادی (شرح گام به گام)**

(توصیفی تا بتوانی سریع در Modeler بکشیش)

1. **Start Event** (incoming request, contain correlationId, amount, payer, ...)
2. **Service Task (External)** — Charge Fee
   * External Task topic = charge-fee
   * On success: sets process variable feeCharged=true, feeTransactionId
   * Compensation handler: Refund Fee (topic refund-fee)
3. **Service Task (External)** — Call Peychak
   * topic = peychak-call
   * sets peychakTransactionId, peychakStatus
   * If peychak returns recoverable error → retry; if business error and peychak supports rollback later, record peychakRollbackPossible=true and peychakRollbackId.
   * Compensation handler: PeychakRollback (calls peychak rollback api) — only invoked if peychak reported it supports rollback.
4. **Subprocess (Transactional logic internal)** — Internal Check Processing
   * This is a **call activity** or **embedded subprocess** that encapsulates multiple DB ops in service internal-check-service.
   * Implemented as External Task internal-check which the internal service performs in its own DB transaction. If internal service fails with business error → it will rollback its DB transaction (local). Then it should **throw a BPMN Error** (business error) back to the process (use External Task failure with error boundary so Camunda gets notified).
   * Compensation handler for internal operations if needed: internalRollback (but usually local DB rolled back, so compensation may be no-op or for compensating side-effects).
5. **Exclusive Gateway**
   * On success → Complete → End Event.
   * On failure (business error from step 4) → **compensate** sequence triggered — Camunda executes compensation handlers in reverse order (internal -> peychakRollback -> refundFee).
6. **End Event**

**نکات BPMN پیاده‌سازی:**

* برای هر Service Task که نیاز به جبران دارد، در مدل BPMN به آن Activity یک **"compensation boundary"** یا حتماً **compensation handler** متصل کن (BPMN: add a separate ServiceTask with isForCompensation="true" and connect it as the compensationActivity).
* استفاده از **Transaction Subprocess**: اگر می‌خوایم رفتارهای تراکنشی‌تری داشته باشیم (Camunda supports BPMN Transaction), می‌توانیم یک Transaction SubProcess بگذاریم که در صورت خطای داخلی، boundary compensation یا cancel event اجرا شود. اما نکته: Transaction subprocess semantics پیچیده‌اند و با external-services ترکیب‌شده باید خیلی مراقب بود. معمولاً با External Task + Compensation ساده‌تر و قابل کنترل‌تر است.

**رفتار در مواجهه با خطا — چه چیزی جبران می‌شود و چه چیزی باعث fail کل پروسه می‌شود**

* **خطاهای قابل بازیابی (Transient / Technical)**
  + مثال: timeout شبکه به Peychak، DB deadlock، 5xx از remote.
  + رفتار: retry (via External Task retries / backoff). اگر بعد از N تلاش هنوز fail شد → escalation: mark as failed or invoke Compensation depending on business rule. (معمولاً ابتدا retry، سپس compensate).
* **خطاهای کسب‌وکاری (Business Error)**
  + مثال: internal service: validation failure, insufficient funds.
  + رفتار: اگر منجر به توقف و برگشت محلی شد → Camunda باید **Compensate** را اجرا کند تا کارمزد را refund کند و در صورت امکان peychak rollback را صدا بزند.
* **کدام خطا، کدام action را trigger می‌کند؟**
  + internal-check: اگر خطای Business ⇒ throw BPMN Error (External Task worker reports BPMN error) → catch by process → trigger compensation sequence.
  + peychak error: اگر peychak اعلام کند عملیات درج انجام شده اما قابل rollback است ⇒ set peychakRollbackPossible=true و در compensation call peychak rollback.
  + charge-fee failure: اگر charge-fee fail (مثلاً insufficient funds) ⇒ خود process باید متوقف شود و user notification، یا compensation (ندارد چون fee not taken). If fee taken but later need refund → refund-fee compensation.

**پیاده‌سازی External Task Workers (الگوها)**

**الف) Charge Fee Worker (Java using Camunda External Task Client)**

// using org.camunda:camunda-external-task-client-java

extern class ChargeFeeWorker {

ExternalTaskClient client = ExternalTaskClient.create()

.baseUrl("http://localhost:8083/engine-rest")

.asyncResponseTimeout(10000)

.build();

client.subscribe("charge-fee")

.lockDuration(10000)

.handler((externalTask, externalTaskService) -> {

String payerId = externalTask.getVariable("payerId");

double amount = externalTask.getVariable("amount");

try {

// call fee microservice REST (idempotent!)

FeeResponse r = feeService.charge(payerId, amount);

externalTaskService.complete(externalTask, Map.of(

"feeCharged", true,

"feeTransactionId", r.getTransactionId()

));

} catch (BusinessException be) {

// business failure — report BPMN error to process

externalTaskService.handleBpmnError(externalTask, "FEE\_CHARGE\_FAILED", be.getMessage());

} catch (Exception e) {

// transient -> fail and let retry

externalTaskService.handleFailure(externalTask, e.getMessage(), e.toString(), 0, 10000);

}

}).open();

}

**ب) Refund Fee Worker (compensation)**

client.subscribe("refund-fee")

.handler((t, s) -> {

String txId = t.getVariable("feeTransactionId");

try {

feeService.refund(txId); // idempotent

s.complete(t);

} catch (Exception e) {

s.handleFailure(t, e.getMessage(), "", 0, 5000);

}

}).open();

**ج) Internal Check Worker**

* این سرویس خودش DB transaction محلی داره. اگر خطا بخورد، rollback داخلی انجام می‌دهد و سپس به Camunda handleBpmnError می‌زند تا فرآیند orchestration متوجه شود و compensation را اجرا کند.

**نحوه فراخوانی Compensation در Camunda**

* اگر در مدل هر activity که نیاز به جبران دارد، یک **activity برای compensation** تعریف شده و آن را به عنوان isForCompensation=true مشخص کرده باشید، Camunda هنگام فراخوانی compensate() آن را اجرا می‌کند.
* در practice، وقتی internal-check worker handleBpmnError را صدا می‌زند یا ما از داخل مدل مسیر خطا را گرفته‌ایم، در مسیر خطا می‌گذاریم task Compensate را اجرا کنیم یا از API جاوا runtimeService.createProcessInstanceModification(processInstanceId).cancelAllForActivity(...).startBeforeActivity(...).execute() استفاده کنیم تا کارهای جبرانی را آغاز کنیم.
* ساده‌ترین راه: طراحی BPMN به‌گونه‌ای که مسیر خطا مستقیماً به Service Taskهای جبرانی (که External Task topic دارند) برود.

**حفظ Stateهای میانی و پیگیری (observability)**

* **Process Variables**: استفاده از process variables برای نگهداری feeTransactionId, peychakTransactionId, status، attempts و ...
* **Camunda History**: HistoricActivityInstance, HistoricVariableInstance برای audit.
* **Audit Table در هر میکروسرویس**: هر میکروسرویس هم local log/audit نگهداری کند (با correlationId مشترک) تا در صورت نیاز بتوان کارمزد / تراکنش داخلی را در DB محلی پیگیری کرد.
* **CorrelationId**: هر درخواست orchestration یک correlationId داشته باشد که به همه سرویس‌ها پاس داده می‌شود تا بتوان tracing انجام داد.
* **Distributed Tracing**: اضافه کردن OpenTelemetry/Jaeger برای دنبال کردن call chain.
* **Dashboard**: Cockpit + custom API برای جهت‌گیری admin.

**نکات مهندسی مهم (Best Practices)**

1. **Idempotency**: تمام endpointهایی که توسط External Workers صدا زده می‌شوند باید idempotent باشند (با transactionId، requestId).
2. **Retries و Backoff**: برای transient errors، external task client و یا سرویس خودت retry policy داشته باشد.
3. **Dead Letter / Escalation**: پس از چند بار تلاش ناموفق، task را به dead letter یا alert منتقل کن و نیاز به intervention دستی داشته باشد.
4. **Security**: تمام calls با توکن Keycloak و مصداق‌سنجی.
5. **Time-outs & Compensation windows**: برخی سرویس‌ها ممکن است بعدا قابل rollback نباشند — make this explicit (peychakRollbackPossible flag).
6. **Monitoring & Alerts**: alert on compensation failures — compensation هم باید پایدار و قابل retry باشد.

**مرحله‌به‌مرحله عملی برای شما (قابل اجرا)**

1. **مدل BPMN**: مدل مرحله‌ای که گفتم را در Camunda Modeler بساز (activities: charge-fee, peychak-call, internal-check; compensation tasks: refund-fee, peychak-rollback, internal-compensate).
2. **تعریف External Task Workers**: برای هر topic یک worker جاوا یا node بنویس (نمونه کد بالا). تست محلی با Camunda engine embedded یا docker Camunda که REST فعال داره.
3. **توسعه میکروسرویس‌ها**: fee-service, peychak-adapter, internal-check-service — هر کدام endpoint مناسب و idempotent پیاده کن و log/audit با correlationId بنویس.
4. **مکانیسم خطا**: در internal-check-service اگر خطای جدی رخ داد، آن را به Camunda با handleBpmnError یا reply error به external task برگردان.
5. **Compensation flows**: پیاده‌سازی refund-fee و peychak-rollback workers؛ تست کن که در صورت خطا، اینها اجرا می‌شوند و نتیجه را لاگ و به process variable منتقل می‌کنند.
6. **تست end-to-end**: با Postman یا تست اتوماتیک:
   * Start process
   * Force internal-check to throw business error → بررسی کن کارمزد refunded و peychak rollback invoked (اگر ممکن بود)
   * تست حالات موفق، transient (network), و failure نهایی
7. **Observability**: اضافه کردن queries:
   * GET /orchestration/{correlationId}/status → aggregate data from process variables + microservices audit tables
   * GET /orchestration/{processInstanceId}/history → uses Camunda history endpoints

**مثال ساده‌ی جریان خطا و compensation (روالی)**

* Start → charge-fee → peychak-call → internal-check → (internal fails)
* internal fails → internal service rolled back local DB and notifies Camunda (BPMN Error)
* Camunda receives error → route to compensation path:
  1. start external task peychak-rollback (if supported)
  2. start external task refund-fee
* After compensation complete → mark orchestration as FAILED\_WITH\_COMPENSATION and notify caller (third-party).

**مستندات/اسلاید پرزنت برای سرپرست**

برای پرزنت، پیشنهاد می‌کنم اسلایدهای زیر رو اضافه کنی:

1. UML sequence یا BPMN of full flow (with compensation path highlighted).
2. جدول «خطا → رفتار» (مثلاً: transient -> retry, business -> compensate, internal DB error -> local rollback + compensate).
3. نمونه log traces (correlationId) که نشان می‌دهد fee charged → internal failed → refund successful.
4. نکات عملی: idempotency, retries, DLQ, monitoring.

اگر می‌خوای، همین الآن برات:

* ۱) یک فایل BPMN نمونه با این ساختار (compensation tasks و external task topics) بسازم و بفرستم؛
* ۲) یا نمونه کد Java External Task Worker برای charge-fee، refund-fee، peychak-call، internal-check آماده کنم؛
* ۳) یا یک Postman collection / تست سناریو E2E آماده کنم.

کدوم یکیشو اول بخوای انجام بدم تا مستقیم پیاده‌سازی شروع کنی؟

## خطا در فراخوانی rollback

ببین اگر در مرحله فراخوانی خود جبران ساز به خطا خوردیم چی؟ مثلا سرویس پایین اومده بود یا به هر دلیلی عملیات rollback اون به خطا خورد؟

سؤال خیلی خوبی پرسیدی 👌  
واقعیت اینه که تو دنیای واقعی همیشه احتمال اینکه **خود جبران‌ساز هم fail کنه** وجود داره (مثلاً سرویس پایین باشه، یا rollback از سمت سرویس upstream پشتیبانی نشه).

**✅ راهکارهای برخورد با خطای Compensation**

**1. Retry Policy برای Compensation**

* دقیقاً مثل سرویس اصلی، جبران‌ساز هم باید با **retry/backoff** کار کنه.
* Camunda External Task API اجازه می‌ده برای compensation task هم handleFailure صدا بزنی و دوباره در صف بمونه.
* یعنی اگر سرویس refund-fee موقتاً در دسترس نبود، دوباره تلاش بشه.

**2. Idempotency در سرویس جبران‌ساز**

* همه جبران‌سازها باید idempotent طراحی بشن:
  + اگر دوبار refundFee(txId) صدا زده شد، فقط یک بار اثر بگذارد.
  + همین باعث میشه retries بدون خطر باشن.

**3. Dead Letter Queue (DLQ) / Incident Handling**

* اگر بعد از N بار retry باز هم fail بود → task میره توی **Incident / Dead Letter Queue**.
* اونجا alert زده میشه که اپراتور انسانی بررسی کنه (manual resolution).
* مثلاً اپراتور بره دستی refund کنه و بعد incident رو resolve کنه.

**4. Partial Compensation Handling**

* ممکنه بعضی جبران‌سازها موفق بشن و بعضی نه.
* راه‌حل: state process رو بذار روی FAILED\_WITH\_PARTIAL\_COMPENSATION.
* این وضعیت رو به upstream گزارش بده (مثلاً: "کارمزد برگشت داده شد ولی Peychak هنوز rollback نشده").
* بعداً یک job background یا اپراتور می‌تونه دوباره سعی کنه عملیات ناقص رو جبران کنه.

**5. Timeout & Escalation**

* اگر سرویس rollback خیلی طولانی fail بمونه (مثلاً روزها) →
  + در Camunda میشه escalation یا message boundary گذاشت که incident رو escalate کنه.
  + مثلاً: "Peychak rollback failed > escalate to manual support".

**6. Business Rules**

* خیلی وقت‌ها rollback واقعی در دسترس نیست (مثلاً کارمزد منتقل شده به حساب بانکی بیرون).
* در این موارد باید تصمیم بگیری:
  + یا خطا رو log کنیم و alert بدیم → manual refund توسط اپراتور.
  + یا مسیری برای "compensation failed" توی BPMN در نظر بگیریم.

**🔎 سناریوی عملی روی همون Flow شما**

1. internal-check fail میشه → Compensation trigger میشه.
2. Camunda صدا میزنه:
   * peychak-rollback (fail می‌کنه چون سرویس peychak down هست)
   * worker با handleFailure و retry policy ثبت می‌کنه → Camunda دوباره تلاش می‌کنه (مثلاً هر 1 دقیقه تا 5 بار).
   * اگر بعد از 5 بار موفق نشد → incident میره تو Cockpit. Process روی state FAILED\_WITH\_PARTIAL\_COMPENSATION می‌مونه.
3. همزمان refund-fee موفق میشه.
4. آخر کار: وضعیت process = "Failed, fee refunded, peychak rollback pending"

**📌 نتیجه**

* هیچ وقت نباید assume کنیم جبران‌ساز همیشه موفق میشه.
* باید **Retry + Idempotency + Incident Management + Partial Status** داشته باشیم.
* از دید کسب‌وکار → وقتی compensation fail میشه، **human intervention** لازمه.

می‌خوای برات **BPMN مدل‌شده**‌ی همین حالت (با compensation + error boundary روی compensation task برای incident) بکشم تا دقیقا ببینی توی Camunda چطور ترسیم میشه؟

# پیاده سازی به صورت میکروسرویس و سرویس های مجزا

این حالت و سناریویی که گفتیم رو شبیه سازی کنیم

میخوام یه میکروسرویس درست کنیم که از چند تا پروژه تشکیل شده باشه

* + 1. میخوام یه سرویس باشه که کار کارمزد رو انجام میده
    2. یه سرویس کار پیچک
    3. یه سرویس کار چک داخلی
    4. یه سرویس برای orchestarto که الان داریم و داریم روش کار میکنیم
    5. بعد یه bpmn هم برام طراحی کن که سناریویی که گفتم رو پیاده سازی کنه

یعنی سرویس کارمزد، پیچک و چک رو که الان نداریم راه اندازی کنیم. حالا نمیدونم چطوری هست مثلا میشه با وایرماک این کار رو کرد یا روش های بهتری هست یا اینکه باید بیایم یه پروژه ایجاد کنیم

اول میخوام نیازمندی ها رو بهم بگی و روش انجام و چالش ها رو بگی و تو مرحله بعد بیایم کار رو جلو ببریم

عالیه 🚀  
این دقیقاً همون چیزی هست که توی تیم‌های بزرگ هم برای **Orchestration PoC** یا **Integration Sandbox** درست می‌کنن. بذار مرحله اول رو با هم بچینیم:

**✅ نیازمندی‌ها و Setup**

تو داری روی Windows 10 با Java 17 و Docker Desktop کار می‌کنی → همین برای شروع کفایت می‌کنه.

**1. Microservices**

ما می‌خوایم ۴ سرویس داشته باشیم:

1. **Fee Service**
   * مسئول کم کردن/برگشت کارمزد.
   * APIها:
     + POST /fee/charge → کم کردن
     + POST /fee/refund → جبران‌ساز
2. **Peychak Service** (شبیه‌ساز بانک مرکزی)
   * APIها:
     + POST /peychak/register → ثبت چک
     + POST /peychak/rollback → لغو ثبت
3. **Internal Check Service**
   * APIها:
     + POST /check/process → پردازش چک (با تراکنش DB)
   * شبیه‌سازی خطا: بعضی وقت‌ها fail کنه (مثلاً random یا وقتی amount > 5000 باشه).
4. **Orchestrator Service** (Spring Boot + Camunda)
   * همونی که الان داریم.
   * پروسه BPMN روی این سوار میشه و orchestration می‌کنه.

**2. روش پیاده‌سازی**

اینجا سه گزینه داریم:

1. **WireMock برای Mock API**
   * سریع‌ترین راه برای شبیه‌سازی سرویس‌ها.
   * ولی چون می‌خوای سناریوی compensation و state واقعی رو تست کنی، بهتره **سرویس واقعی (Spring Boot ساده)** بسازیم.
2. **Spring Boot Mini Services**
   * هر سرویس = یک پروژه Spring Boot کوچک با REST Controller.
   * دیتا = H2 DB یا In-Memory Map (نیازی به DB سنگین نیست).
   * قابل گسترش به آینده (مثلاً اگه خواستی فیچر اضافه کنی).
3. **Docker Compose**
   * بعداً همه رو با Compose بالا بیاریم تا orchestrator بتونه همه سرویس‌ها رو call کنه.

**3. BPMN (High-Level)**

سناریوی تو → مرحله به مرحله:

(Start)

↓

[Fee Service → chargeFee]

↓

[Peychak Service → registerCheque]

↓

[Internal Check Service → processCheck]

↳ ❌ اگر fail شد → compensation

- refundFee

- peychakRollback

↓

[End]

**4. چالش‌ها**

* **Idempotency**: refundFee دوبار صدا نشه.
* **Error Simulation**: Internal Check باید fail بشه تا compensation رو تست کنیم.
* **Transactionality**: Internal Check باید rollback کنه (استفاده از DB transaction).
* **Compensation Errors**: اگر refundFee خودش fail کرد → retry/incident.
* **Service Discovery/Ports**: باید همه سرویس‌ها روی پورت جدا باشن (مثلاً 8081، 8082، 8083، orchestrator روی 8080).

**🎯 پیشنهاد قدم بعدی**

مرحله ۱: ساخت **Fee Service** به‌عنوان یک Spring Boot پروژه ساده (با دو endpoint: charge/refund).

* دیتا رو تو یه map نگه می‌داریم (مثلاً userId → balance).
* خطاها رو هم می‌تونیم شبیه‌سازی کنیم (مثلاً اگر مبلغ > 1000 باشه refund fail بشه).

بعد بریم سراغ Peychak و Internal Check و در نهایت Orchestrator BPMN.

## مرحله 1 - ساختار پروژه‌ی **Fee Service** (Spring Boot با API و simulation ساده

خوبه، بریم سراغ **Fee Service** 👇

**✅ مرحله اول: Fee Service**

**🎯 هدف**

* سرویس مستقلی باشه (Spring Boot)
* روی یه پورت جدا (مثلاً 8081) بالا بیاد
* دو API داشته باشه:
  + POST /fee/charge → کم کردن کارمزد
  + POST /fee/refund → برگرداندن کارمزد

**📂 ساختار پروژه پیشنهادی**

fee-service

├─ src

│ ├─ main

│ │ ├─ java/com/example/fee

│ │ │ ├─ FeeServiceApplication.java

│ │ │ ├─ controller/FeeController.java

│ │ │ ├─ model/FeeRequest.java

│ │ │ ├─ model/FeeResponse.java

│ │ │ └─ service/FeeService.java

│ │ └─ resources/application.yml

└─ pom.xml

**🚀 تست با Postman**

بعد از اجرای پروژه (mvn spring-boot:run):

1. **Charge Fee**

POST http://localhost:8081/fee/charge

{

"userId": "u1",

"amount": 200

}

1. **Refund Fee**

POST http://localhost:8081/fee/refund

{

"userId": "u1",

"amount": 200

}

1. **Check Balance**

GET http://localhost:8081/fee/balance/u1

## مرحله 2 – ساخت پروژه PichakService

عالیه 👌  
پس Fee Service رو داریم و تست هم کردی. بریم سراغ **Pichak Service**.

**✅ مرحله دوم: Pichak Service**

**🎯 هدف**

* یک سرویس مستقل (Spring Boot) روی پورت **8082**
* شبیه‌سازی سرویس بانک مرکزی (Pichak)
* دو API داشته باشه:
  + POST /pichak/validateCheque → بررسی و اعتبارسنجی چک
  + POST /pichak/cancelValidation → جبران‌ساز در صورت نیاز (لغو اعتبارسنجی)

**📂 ساختار پروژه پیشنهادی**

pichak

├─ src

│ ├─ main

│ │ ├─ java/com/example/pichak

│ │ │ ├─ PichakApplication.java

│ │ │ ├─ controller/PichakController.java

│ │ │ ├─ model/PichakRequest.java

│ │ │ ├─ model/PichakResponse.java

│ │ │ └─ service/PichakService.java

│ │ └─ resources/application.yml

└─ pom.xml

**🚀 تست با Postman**

1. **Validate Cheque**

POST http://localhost:8082/pichak/validateCheque

{

"chequeId": "c123",

"userId": "u1"

}

1. **Cancel Validation**

POST http://localhost:8082/pichak/cancelValidation

{

"chequeId": "c123",

"userId": "u1"

}

## مرحله 3 – پروژه چک

آفرین خیلی خوب دیباگ کردی 👌  
پس الان دو سرویس داریم:

* **FeeService** → مدیریت کارمزد
* **PichakService** → اعتبارسنجی با بانک مرکزی

حالا نوبت **ChequeService** هست.  
این سرویس یه مقدار مهم‌تره چون خودش باید چند مرحله (مثلاً شبیه تراکنش داخلی) انجام بده و بتونیم بعداً سناریوی خطا و rollback رو تست کنیم.

**✅ مرحله ایجاد پروژه Cheque**

**1. ساخت پروژه**

مثل Fee و Pichak:

* **Name:** cheque
* **Artifact:** cheque-service
* **Dependencies:**
  + Spring Web
  + Spring Boot Actuator (اختیاری برای health check)
  + Lombok (مثل قبل)

**4. تست با Postman**

✅ موفق:

curl -X POST http://localhost:8083/cheque/process \

-H "Content-Type: application/json" \

-d '{"chequeId":"c123","userId":"u1","amount":200}'

✅ رول‌بک:

curl -X POST http://localhost:8083/cheque/rollback \

-H "Content-Type: application/json" \

-d '{"chequeId":"c123","userId":"u1","amount":200}'

**5. نکته مهم**

این سرویس رو بعداً می‌تونیم طوری تنظیم کنیم که:

* **عمداً خطا بده** (مثلاً اگر amount > 1000) → تا orchestration بفهمه باید جبران ساز (rollback) اجرا بشه.
* rollback رو هم بتونیم تست کنیم (برگردوندن پول و …).

### نسخه موفق و خطاسازی (failure simulation)

Endpoints:

1. **پردازش چک**  
   POST /cheque/process

* ورودی: { "amount": 500, "forceFail": false }
* خروجی:
  + اگر forceFail = true یا amount > 1000 → شبیه‌سازی خطا (DB failure).
  + در غیر این صورت → ✅ processed = true.

1. **Rollback**  
   POST /cheque/rollback

* ورودی: { "transactionId": "xyz" }
* خروجی:
  + اگر forceFail = true → ❌ rollback fail.
  + در غیر این صورت → ✅ rolledBack = true.

**🌐 تست با Postman**

✅ Happy Path

POST http://localhost:8084/cheque/process

{

"amount": 500,

"forceFail": false

}

⛔ Fail Simulation

POST http://localhost:8084/cheque/process

{

"amount": 1500,

"forceFail": false

}

♻️ Rollback موفق

POST http://localhost:8084/cheque/rollback

{

"transactionId": "abc123",

"forceFail": false

}

🚨 Rollback Fail Simulation

POST http://localhost:8084/cheque/rollback

{

"transactionId": "abc123",

"forceFail": true

}

## مرحله 4 - طراحی Orchestration واقعی با Camunda

در این مرحله قراره همه سرویس‌ها رو کنار هم بیاریم:

1. **شروع فرآیند** (Start Event)
   * درخواست از بیرون (third-party) میاد به Orchestration API.
2. **کسر کارمزد** (Fee Service)
   * سرویس کارمزد صدا زده میشه.
   * اگر موفق بود → ادامه.
   * اگر خطا داد → فرآیند Fail + کاملا پایان.
3. **فراخوانی Pichak (بانک مرکزی)**
   * سرویس pichak صدا زده میشه.
   * اگر خطا داد → Rollback Fee (برگردوندن کارمزد).
4. **پردازش چک داخلی** (Cheque Service)
   * سرویس cheque صدا زده میشه.
   * اگر خطا داد → Rollback Cheque → Rollback Pichak (اگر API داره) → Rollback Fee.
5. **End Event**
   * اگر همه موفق بودن → ✅ Process Done.
   * اگر هرکدوم fail کردن → ❌ Compensations اجرا بشن و بعد End Event.

**📊 BPMN پیشنهادی**

* Start Event → Fee Service → Pichak Service → Cheque Service → End Event
* برای هر سرویس یک **Compensation Task** تعریف می‌کنیم:
  + Rollback Fee
  + Rollback Pichak
  + Rollback Cheque

این میشه یک مدل **Saga Orchestration با Compensation**.

**⚡ مرحله‌های بعدی توسعه**

1. طراحی **BPMN** کامل این سناریو.
2. اضافه کردن Delegateها برای call کردن سرویس‌ها (RestTemplate یا WebClient).
3. پیاده‌سازی Compensation Task ها برای rollback.
4. اضافه کردن API برای start process و query status/history.
5. شبیه‌سازی چند سناریو (Happy Path و Fail Path).

## پیشنهادات ارتقا پروژه

سؤال خیلی خوبی پرسیدی 👌  
الان ما یک سناریوی ساده‌ی میکروسرویسی داریم (Fee, Pichak, Cheque, Orchestrator) که همه چیز **request-response** هست (REST API).  
این برای **PoC و تست** عالیه، ولی اگر بخوای به سمت **پروژه واقعی** بری، چند تا قابلیت حرفه‌ای و best practice هست که ارزش داره همین الان روش فکر کنیم:

**✨ ارتقای معماری میکروسرویس‌ها**

**1. Messaging / Event-Driven**

* به جای اینکه همه‌چیز فقط Sync (REST) باشه، می‌تونیم بعضی callها رو **Async** کنیم.
* مثلاً:
  + وقتی کارمزد گرفته شد، یک **event** مثل FeeCollectedEvent روی Kafka/RabbitMQ منتشر بشه.
  + سرویس orchestrator یا سرویس‌های دیگر اون event رو consume کنن و مرحله بعدی رو شروع کنن.
* مزیت: decoupling، resiliency بیشتر، و قابلیت replay events.

**2. Resilience Patterns**

* استفاده از **Resilience4j** یا **Spring Cloud Circuit Breaker**:
  + **Retry**: اگر سرویس پایین بود دوباره امتحان بشه.
  + **Timeout**: اگر جواب نداد، fail بشه.
  + **Circuit Breaker**: اگر سرویس چند بار fail داد، قطع کنیم.
* این خیلی مهمه توی سرویس‌هایی مثل pichak که dependency بیرونی هست.

**3. Observability**

* **Logging مرکزی** (ELK / Loki + Grafana).
* **Metrics** (Micrometer + Prometheus + Grafana).
* **Tracing** (OpenTelemetry + Jaeger/Zipkin).
  + خیلی مهمه توی orchestrator بفهمیم الان دقیقا process تو کدوم مرحله است و چرا fail شده.

**4. Database Strategy**

* هر سرویس دیتابیس خودش رو داره (که رعایت کردی 👏).
* ولی می‌تونیم **Transactional Outbox** یا **Change Data Capture (CDC)** اضافه کنیم.
  + مثلاً وقتی کارمزد ثبت شد تو DB → event هم روی Kafka منتشر بشه.
  + این باعث میشه event و دیتابیس sync بمونن.

**5. Security**

* الان Keycloak داری → می‌تونیم JWT رو برای همه سرویس‌ها enforce کنیم.
* سرویس orchestrator به جای اینکه مستقیماً call کنه، با **client credential token** از Keycloak بره سمت سرویس Fee/Pichak/Cheque.
* این باعث میشه همه سرویس‌ها امن بشن.

**6. Error Handling & Compensation**

* همین بحثی که خودت گفتی (Saga + Compensation).
* می‌تونیم retry policy بذاریم برای compensation هم.
* اگه compensation fail کرد → یه event CompensationFailed منتشر بشه و توی log مرکزی یا alerting بیاد.

**7. Testing & Simulation**

* اضافه کردن **WireMock** یا **Testcontainers** برای شبیه‌سازی سرویس‌های خارجی.
* Integration Test های end-to-end روی orchestrator.

✅ پیشنهاد من اینه:  
قبل از اینکه bpmn رو حرفه‌ای کنیم، بیایم یک **لایه resilience + observability + security** به همین معماری اضافه کنیم.  
بعد بریم سراغ bpmn و orchestration، چون اون موقع سناریوی ما خیلی واقعی‌تر میشه.

عالیه 👌 پس اجازه بده برات یک **نقشه راه تکمیلی** بچینم، بعد مرحله به مرحله بریم جلو:

**🚀 نقشه راه ارتقای معماری**

ما چهار سرویس داریم (Fee, Pichak, Cheque, Orchestrator).  
روی این‌ها به ترتیب این لایه‌ها رو اضافه می‌کنیم:

**🔹 مرحله ۱ – Resilience**

* اضافه کردن **Retry, Timeout, Circuit Breaker** با Resilience4j.
* هدف: وقتی یکی از سرویس‌ها در دسترس نبود، orchestrator یا سرویس caller بتونه graceful fail کنه.

**🔹 مرحله ۲ – Messaging**

* راه‌اندازی Kafka (یا RabbitMQ) با Docker.
* سرویس Fee بعد از کم کردن کارمزد → یک event FeeCollected منتشر کنه.
* Orchestrator به جای REST polling، اون event رو گوش بده.

**🔹 مرحله ۳ – Observability**

* Logging → متمرکز کنیم (MDC برای traceId).
* Metrics → Micrometer + Prometheus.
* Distributed tracing → OpenTelemetry + Jaeger.
* هدف: ببینیم process end-to-end چه اتفاقی افتاده.

**🔹 مرحله ۴ – Security**

* همه سرویس‌ها JWT از Keycloak بخوان.
* Orchestrator با client-credential token از Keycloak احراز هویت کنه.

**🔹 مرحله ۵ – Compensation & Saga**

* BPMN رو توسعه بدیم.
* توی delegate ها retry + compensation پیاده کنیم.
* مثلا اگه Cheque fail شد → rollback Fee و Pichak.

**🔑 دلیل این ترتیب**

* اول resiliency (بدون اون تو هر مرحله fail می‌کنیم).
* بعد messaging (برای decouple شدن).
* بعد observability (برای مانیتور کردن).
* بعد security (برای production readiness).
* آخر compensation چون به همه این‌ها نیاز داره.

❓ به نظرت از همین **مرحله ۱ (Resilience)** شروع کنیم و Resilience4j رو توی Orchestrator اضافه کنیم (مثلا روی call به FeeService)؟

؟