**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**.

# 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).**

**🎯 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**

# 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.