



## Épica 5→ Los Autenticadores

Stack tecnológicos

- firebase
- outh 2.0
- springboot → rapides y simplicidad
- 
- Arquitectura  
microservicios

ecociclo

## ¿ Que es Github Actions?

GitHub Actions es una herramienta de automatización de CI/CD (Continuous Integration/Continuous Delivery) integrada directamente en GitHub. Permite a los desarrolladores automatizar tareas comunes en el ciclo de vida del desarrollo de software, como la construcción, prueba y despliegue de aplicaciones.

## Proyecto de Midudev

<https://github.com/midudev/pokedex-for-ci.git>

### Principales Características de GitHub Actions:

#### 1. Workflows:

- Los workflows son archivos de configuración escritos en YAML que definen una serie de pasos para automatizar procesos. Estos workflows se ejecutan en respuesta a eventos específicos en el repositorio, como `push`, `pull_request`, o `release`.

#### 2. Actions:

- Las acciones son las tareas individuales que se ejecutan como parte de un workflow. Puedes usar acciones predefinidas disponibles en el GitHub Marketplace o crear las tuyas propias.

#### 3. Jobs:

- Un workflow se compone de uno o más jobs. Cada job se ejecuta en un runner, que puede ser una máquina virtual o un contenedor.

#### 4. Steps:

- Los steps son comandos individuales que se ejecutan dentro de un job. Un step puede ejecutar una acción o un comando de shell.

### Usos Comunes:

- Automatizar Builds:** Construcción automática de tu aplicación cada vez que haces un push.
- Ejecución de Pruebas:** Ejecutar pruebas unitarias, de integración y de aceptación.
- Despliegue:** Desplegar tu aplicación en diferentes entornos, como servidores de producción o staging.
- CI/CD:** Implementar pipelines completos de CI/CD para asegurar la calidad y entrega continua del software.

## Sistemas de integración Continua(son maquinas virtuales)

La Integración Continua (CI) es una práctica de ingeniería de software que implica la fusión regular del código de desarrollo en un repositorio compartido, como Git. Tras cada integración, se ejecuta automáticamente un pipeline CI que realiza compilaciones y ejecuta pruebas unitarias y de integración. Esto se hace utilizando herramientas como Jenkins, Travis CI o GitHub Actions para asegurar que cada cambio de código sea verificable y no introduzca errores en la base de código existente.

Implementar CI ayuda a mejorar la colaboración entre los desarrolladores, detectar errores temprano en el ciclo de desarrollo y mantener una base de código más estable y de alta calidad.

- jenkins

**Jenkins**

Jenkins – an open source automation server which enables developers around the world to reliably build, test, and deploy their software

🔗 <https://www.jenkins.io/>



- travis CI

**Simple, Flexible, Trustworthy CI/CD Tools - Travis CI**

Travis CI is the most simple and flexible ci/cd tool available today. Find out how Travis CI can help with continuous integration and continuous delivery.

🔗 <https://www.travis-ci.com/>

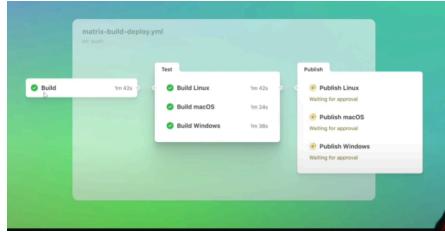


- github action

**GitHub Actions**

Easily build, package, release, update, and deploy your project in any language—on GitHub or any external system—without having to run code yourself.

🔗 <https://github.com/features/actions>



## Referencias.

CI/CD GRATIS con GITHUB ACTIONS y SPRING BOOT: Tu código NUNCA fallará sin que lo sepas  
Buenas a todos, en este video veréis como usar GitHub Actions para una aplicación de Spring Boot

Repository: <https://github.com/David-DAM/ecommerce-api>  
➡ [https://www.youtube.com/watch?v=El\\_a3ax26x0](https://www.youtube.com/watch?v=El_a3ax26x0)



## Worflows

- se definirá el yml para hacer un flujo de trabajo para crear y probar la aplicación cada vez que una solicitud de grupo se abre o se crea.

## Análisis del archivo ( maven-build.yml)

```
# This workflow will build a Java project with Maven, and cache/restore any dependencies to improve the work
# For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-a-java-project-with-maven
```

```
# nombre del flujo de trabajo
name: Java CI with Maven
```

### Sección 1

```
# en esta sección se indica el disparador para el flujo;
# el cual es una solicitud de grupo que se abre o se crea
```

```
# ¿ que significa
##→ que el workflows se activara cuando la solicitud de grupo se abra, sincronice o reopened
```

```
---
```

```
on:
```

```
  pull_request:
    branches:
      - main
```

```
---
```

```

# Sección 2:
se agrupan todo los grupos de trabajo que estan mal, en el flujo de trabajo actual

# por lo tanto → One workflows has one or more jobs: es decir que en un mismo flujo de trabajo se peden ejec

jobs:
  build:
    # propiedades de este trabajo de compilación

    runs-on: ubuntu-latest # el trabajo se ejecutara en la ultima versión de linux
      ## → se pueden ejecutar trabajos en contenedores de docker o un ejecutor a

# sección 3:
se agrupan todos los pasos que estan mal en el trabajo de compilación
steps:
  - name: Check out the repository #verifica el repositorio
    uses: actions/checkout@v4    #

  - name: Set up JDK 17      #configurar el entorno de Java para que tu workflow pueda ejecutarse con JD
    uses: actions/setup-java@v4
    with:
      java-version: '17' #Esta línea especifica la versión de Java (JDK) que se va a instalar.
      distribution: 'temurin' #Esta línea define la distribución específica de Java que se utilizará. en este caso te
      cache: maven #Esta línea configura la caché para las dependencias de Maven.
  - name: Build and test with Maven
    run: mvn --batch-mode package

```

worflow → 1 or more jobs

1 job → 1 or more steps

```
name: Deploy to VPS
```

```
on:
```

```
push:
  branches:
    - main
```

```
jobs:
```

```

deploy:
  runs-on: ubuntu-latest

  steps:
    # 1. Checkout del código
    - name: Checkout code
      uses: actions/checkout@v3

    # 2. Configurar Java
    - name: Set up Java
      uses: actions/setup-java@v3
      with:
        java-version: '21'
        distribution: 'temurin'

    # 3. Construir el Jar con Maven
    - name: Build JAR file with Maven
      run: |
        mvn clean package
        ls -l target/*.jar

    # 4. Añadir el host SSH a known_hosts
    - name: Add SSH Key to Known Hosts
      run: |
        mkdir -p ~/.ssh
        ssh-keyscan -p ${{ secrets.PORT }} -H ${{ secrets.SERVER_HOST }} >> ~/.ssh/known_hosts

    # 5. Configurar la clave SSH para el acceso (agente SSH)
    - name: Set up SSH
      uses: webfactory/ssh-agent@v0.5.3
      with:
        ssh-private-key: ${{ secrets.SERVER_KEY }}

    # 6. Desplegar la aplicación en la VPS
    - name: Deploy to VPS
      run: |
        DEPLOY_DIR="/home/admin_jb_3"
        LOGS_DIR="$DEPLOY_DIR/logs"
        APP_JAR="Bussiness-0.0.1-SNAPSHOT.jar"

        echo "Matando proceso anterior (si existe)..."
        ssh -p ${{ secrets.PORT }} ${{ secrets.SERVER_USER }}@${{ secrets.SERVER_HOST }} "sudo pkill -f \$APP_JAR"

        echo "Copiando el nuevo JAR..."
        scp -P ${{ secrets.PORT }} target/*.jar ${{ secrets.SERVER_USER }}@${{ secrets.SERVER_HOST }}:$DEPLOY_DIR

        echo "Iniciando la aplicación en segundo plano..."
        ssh -p ${{ secrets.PORT }} ${{ secrets.SERVER_USER }}@${{ secrets.SERVER_HOST }} "\"
          nohup java -jar $DEPLOY_DIR/$APP_JAR > $LOGS_DIR/output.log 2>&1 &

```

"

```
echo "Deploy completed"
```

### Introducción a Github Actions

Github Actions es una de las características de Github como plataforma que permite ejecutar tareas en un servidor para poder hacer Testing, o ejecutar comandos de programas en cada push, pull request o otro evento que hagamos en nuestro repositorio de código de Git. En este

▶ [https://www.youtube.com/watch?v=ePXwmNXG\\_nA](https://www.youtube.com/watch?v=ePXwmNXG_nA)



### [Episode 46] Spring Boot Continuous Integration with GitHub Actions

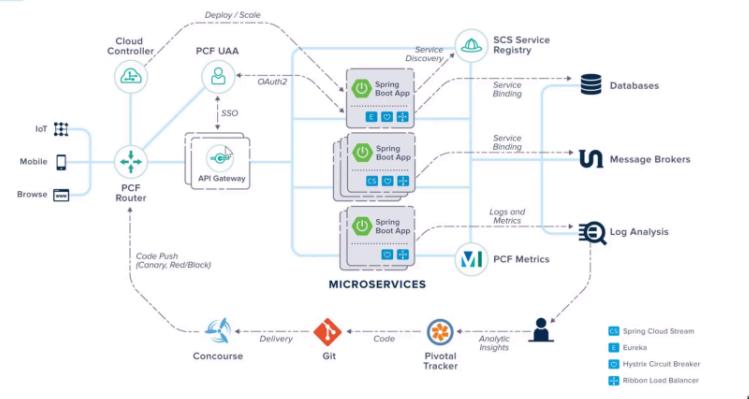
🔗 .github/workflows/maven-build.yml: <https://github.com/Washingtonwei/hogwarts-artifacts-online/blob/main/.github/workflows/maven-build.yml>

▶ <https://www.youtube.com/watch?v=0QyBY1tMoVk>



## Microservicio con Springboot

- ▼ Spring Cloud Config 
- ▼ Spring Cloud Circuit Breaker
- ▼ Spring Cloud Bus
- ▼ Spring Cloud Netflix
- ▼ Spring Cloud Gateway
- ▼ Spring Cloud Sleuth
- ▼ Spring Cloud Stream



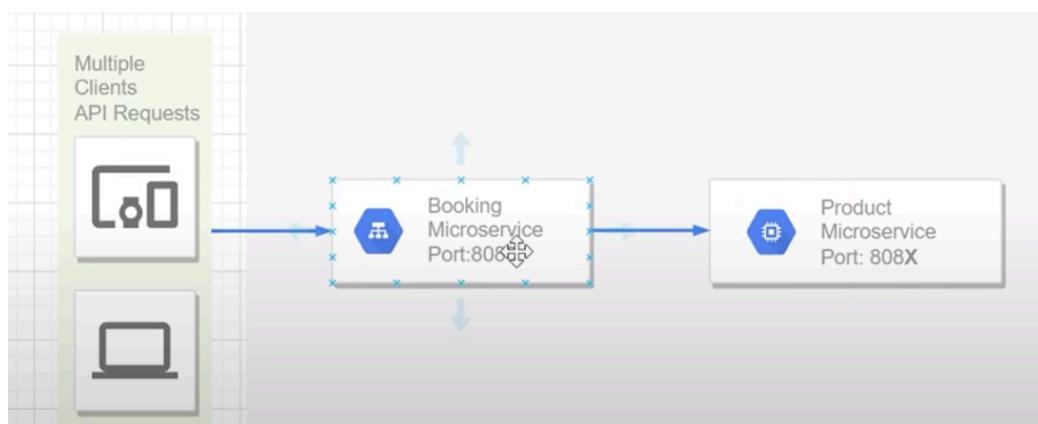
## creando el primer Microservicio

- se llama product-microserver

## Implementación Discovery Service. → también es un microservicio

- que es harcodear una url?

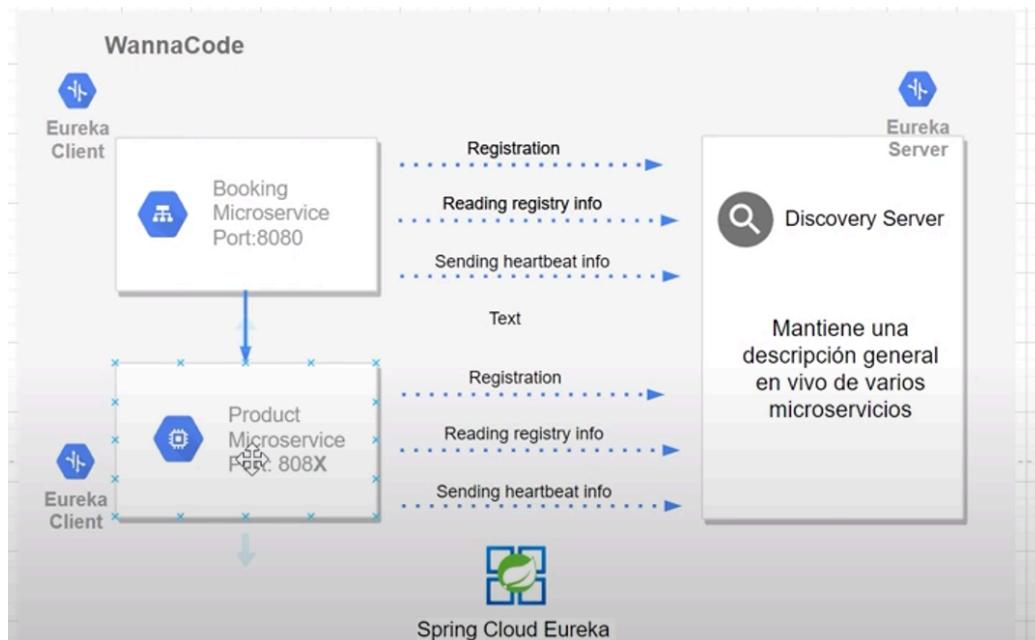
"Hardcodear" una URL en el contexto de microservicios se refiere a definir la URL de un servicio de manera fija en el código, en lugar de obtenerla dinámicamente o configurarla externamente. Esto puede tener varias implicaciones:



## Para que sirve un discovery server?

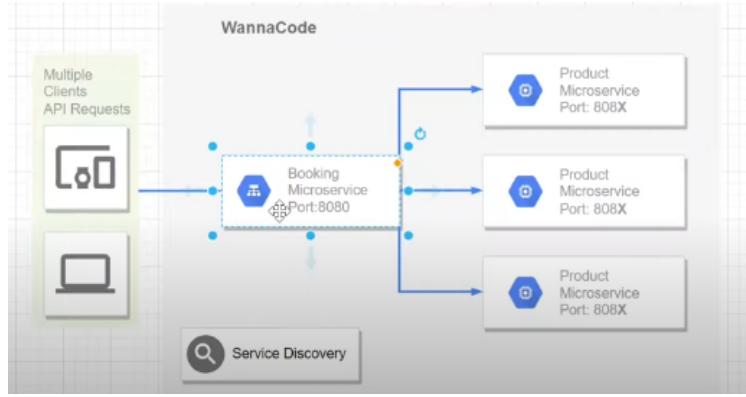
•

**Service Discovery:** Implementa un mecanismo de descubrimiento de servicios (como Consul, Eureka, o Kubernetes) que permite que los microservicios encuentren y se comuniquen entre sí sin URLs fijas.



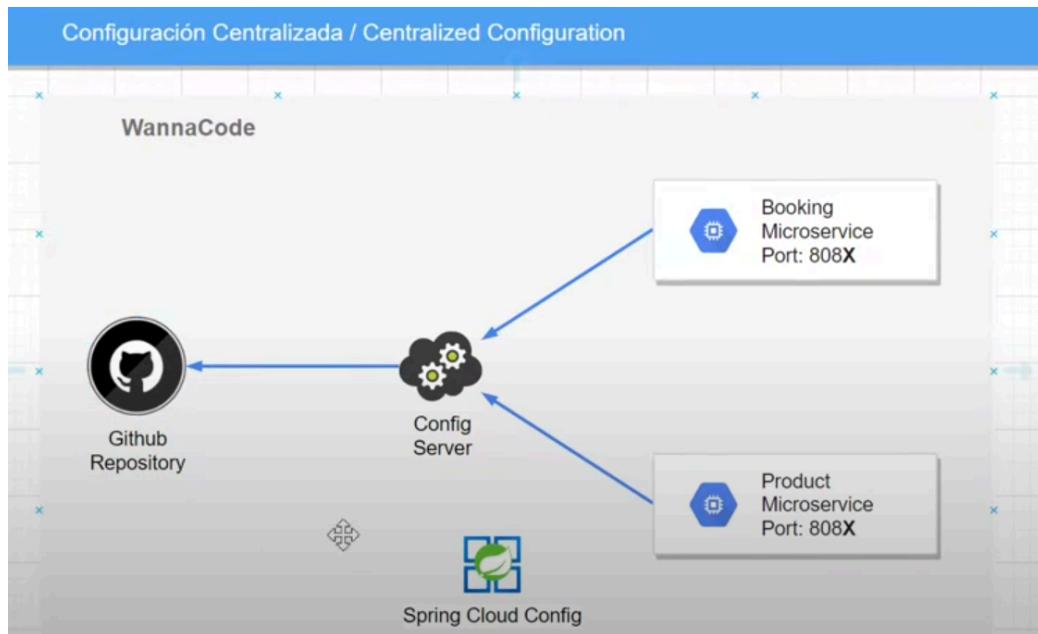
un microservicio se registra en el Discovery Server, puede leer información de los microservicio que ya están registrados y puede pasar un diagnostico de como se encuentra su estado(funciona o no)

## Implementando un Config server



En el caso que existan muchas instancias de diferentes microservicios y necesitemos cambiar las credenciales o configuraciones en el properties... Para no pausar su ejecución usamos el siguiente mecanismo.

## Configuración centralizada



## Refresh configuration

La función de "Refresh Configuration" en Spring Cloud Config permite actualizar dinámicamente la configuración de las aplicaciones sin tener que reiniciarlas. Esto es muy útil cuando deseas cambiar alguna configuración de tu aplicación en tiempo real y quieres que esos cambios se reflejen inmediatamente sin necesidad de interrumpir el funcionamiento de la aplicación.

Aquí te explico cómo funciona y para qué sirve:

### 1. Cambio de Configuración Dinámica:

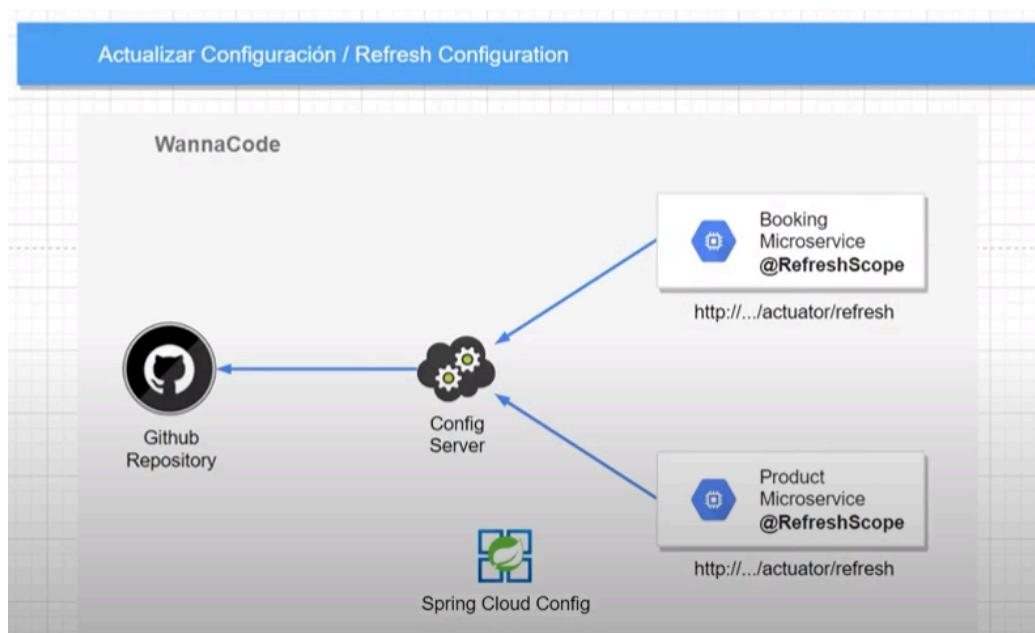
- Permite aplicar cambios en la configuración desde el Config Server sin reiniciar la aplicación.
- Es útil para ajustar parámetros de configuración, como límites de tiempo, configuraciones de logging, URLs de servicios externos, etc., de manera dinámica.

### 2. EndPoint de `/refresh`:

- Spring Actuator proporciona un endpoint `/refresh` que se puede invocar para recargar la configuración.
- Puedes habilitar este endpoint añadiendo la dependencia `spring-boot-starter-actuator` y configurando las propiedades necesarias.

### 3. Uso de `@RefreshScope`:

- Utilizando la anotación `@RefreshScope`, puedes marcar beans específicos que deben ser refrescados cuando se invoca el endpoint `/refresh`.
- Esta anotación puede ser aplicada a clases de configuración o componentes específicos que contengan propiedades que deseas refrescar dinámicamente.



## Hascore Vault

**HashiCorp Vault** es una herramienta de gestión de secretos basada en identidad desarrollada por HashiCorp. Su propósito principal es **almacenar, gestionar y proteger** tokens, contraseñas, certificados y claves de cifrado de manera segura. Vault permite a las aplicaciones y servicios acceder a estos secretos de manera controlada y segura.

Algunas características clave de HashiCorp Vault incluyen:

- **Gestión de secretos dinámicos:** Genera secretos en tiempo real y únicos para cada cliente.
- **Rotación automática de credenciales:** Administra la rotación de contraseñas y certificados automáticamente.
- **Integración con identidades:** Autentica y controla el acceso a secretos utilizando fuentes de identidad confiables como Active Directory, LDAP, Kubernetes, entre otros.
- **Alta disponibilidad:** Permite configurar múltiples servidores Vault para garantizar la disponibilidad y la recuperación ante desastres

guía para la configuración de vault usando springboot

### Getting Started | Vault Configuration

Learn how to store and retrieve application configuration details in HashiCorp Vault



 <https://spring.io/guides/gs/vault-config>

Toda la documentacion y practicas recomendadas de springboot

### Guides

Level up your Java code and explore what Spring can do for you.



 <https://spring.io/guides>

generar un servidor de vault local.

```
vault server --dev --dev-root-token-id="00000000-0000-0000-0000-000000000000"
```

tambien se puede ejecutar vault server --dev → aqui vault asignara un token aleatorio

\*Para agregar las claves secretas al vault\*

```
vault kv put secret/booking-microservice @booking-microservice.json
```

## Hardcodear

El término "hardcodear" (del inglés "hardcoding") se refiere a la práctica de **insertar directamente datos específicos o valores fijos dentro del código fuente de un programa** en lugar de utilizar variables, archivos de configuración o bases de datos.

Por ejemplo, en lugar de usar una variable que almacene una URL, un programador podría escribir la URL directamente en el código:

- eliminar .json desde el control de versiones

```
git rm --cached *.json
```

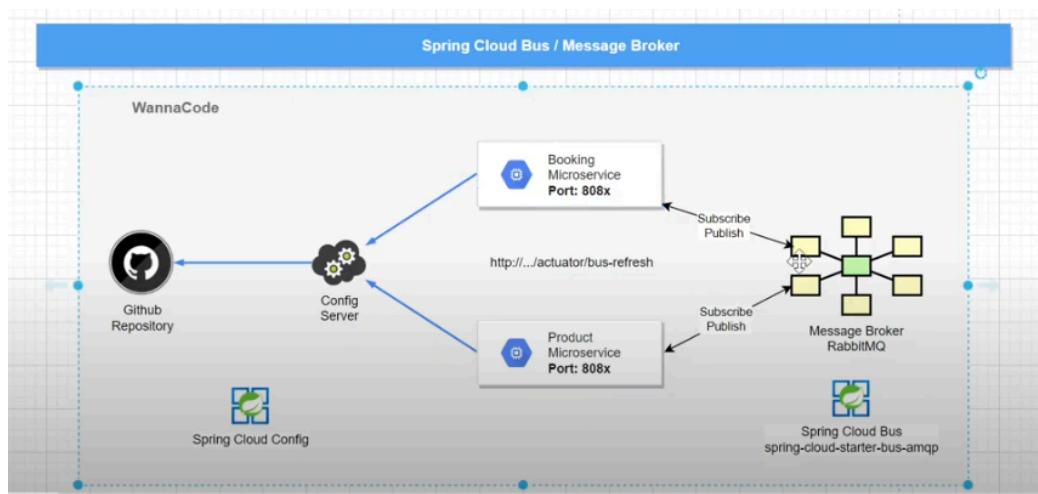
## Microservicios con Spring Boot - Message Broker, Refresh Configuration, RabbitMQ, Docker

Un **message broker** (o **intermediario de mensajes**) es un software que actúa como intermediario entre aplicaciones, sistemas o servicios que necesitan comunicarse entre sí. Su función principal es **traducir y enrutar mensajes** entre diferentes sistemas, permitiendo que estos se comuniquen de manera eficiente y segura, incluso si están escritos en diferentes lenguajes o implementados en diferentes plataformas.

### Funciones principales de un message broker:

- **Validación de mensajes:** Asegura que los mensajes recibidos cumplan con ciertos criterios antes de ser procesados.
- **Almacenamiento de mensajes:** Guarda los mensajes en una cola hasta que el sistema receptor esté listo para procesarlos.

- **Enrutamiento de mensajes:** Dirige los mensajes a los sistemas o servicios correctos.
- **Transformación de mensajes:** Convierte los mensajes a diferentes formatos según sea necesario.
- **Mensajería asíncrona:** Permite que las aplicaciones se comuniquen sin necesidad de una conexión constante, lo que mejora la eficiencia y la tolerancia a fallos.



como funciona el mecanismo de spring cloud bus?

<http://localhost:49399/actuator/busrefresh> → postman, refleja la actualización en otros microservicios

La anterior solicitud es cirtamente muy diferente a la solicitud usada para actualizar un solo microservicio, la cu

<http://localhost:49399/actuator/refresh>

## Vault server

HashiCorp Vault | Identity-based secrets management  
Standardize secrets management with identity-based security from Vault that lets you centrally discover, store, access, rotate, and distribute dynamic secrets.

<https://www.hashicorp.com/es/products/vault>



el siguiente comando de la terminal, me permitirá correr el servidor de vault en mi local

```
vault server --dev --dev-root-token-id="00000000-0000-0000-0000-000000000000"
```

se esta definiendo el token pero puede ser aleatorio

- Para guardar harcodeando las credenciales de la base de datos, se vuelve necesario desde el archivo de config-server crear archivos .json donde se guarden esas credenciales, uno por microservicio.

```
vault kv put secret/booking-microservice @booking-microservice.json
```

## Rabbitmq

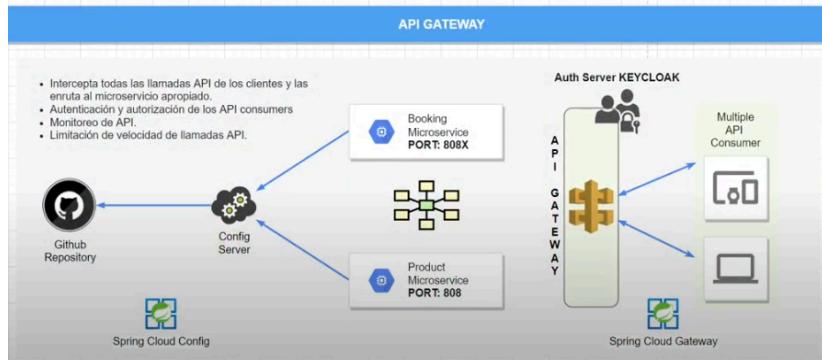
Installing RabbitMQ | RabbitMQ  
<!--  
👉 <https://www.rabbitmq.com/docs/download>



→ encuento a RABBITMQ se volvio necesario correr desde el powershell encender el servidor:

```
docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:4.0-management
```

## Api GateWay



### Relación con el Discovery Server:

- **Eureka como servidor de descubrimiento:** Eureka actúa como el servidor de descubrimiento donde todos los microservicios se registran y mantienen su información de estado.
- **Spring Cloud Gateway:** Actúa como un punto de entrada para tus microservicios y puede enrutar las solicitudes a los microservicios adecuados basándose en las rutas.
- **DiscoveryClientRouteDefinitionLocator:** Esta clase en Spring Cloud Gateway consulta el servidor de Eureka para obtener la lista de servicios registrados y crea rutas dinámicas en función de esta información.

### Eureka (Discovery Server):

- **Registro de Servicios:** Permite a los microservicios registrarse y mantenerse actualizados en el servidor de descubrimiento.
- **Descubrimiento de Servicios:** Permite a los microservicios descubrir otros microservicios mediante consultas al servidor de descubrimiento.
- **Equilibrio de Carga:** Proporciona la lista de instancias disponibles de un servicio para permitir el equilibrio de carga a nivel de cliente (cada microservicio elige una instancia de la lista proporcionada).

### API Gateway:

- **Punto de Entrada Unificado:** Actúa como un único punto de entrada para todas las solicitudes externas que llegan al sistema. Los clientes interactúan con el API Gateway en lugar de hacerlo directamente con los microservicios.
- **Enrutamiento de Solicitudes:** Enruta las solicitudes de los clientes a los microservicios correspondientes. Puede enrutar automáticamente basándose en la información de Eureka.
- **Seguridad y Autorización:** Gestiona la autenticación y autorización de las solicitudes antes de que lleguen a los microservicios.
- **Balanceo de Carga:** Realiza el balanceo de carga a nivel de gateway, distribuyendo las solicitudes entre las diferentes instancias de microservicios.
- **Transformación de Solicitudes y Respuestas:** Puede modificar las solicitudes antes de enviarlas a los microservicios y transformar las respuestas antes de devolverlas a los clientes.

- **Gestión de API:** Proporciona capacidades adicionales como limitación de tasas (rate limiting), caché, y manejo de fallos (circuit breaking).

### Ejemplo:

- **Eureka:**
  - `booking-microservice` y `product-microservice` se registran en Eureka.
  - `booking-microservice` puede descubrir `product-microservice` consultando a Eureka.
- **API Gateway:**
  - Los clientes envían solicitudes a `http://api-gateway/api/bookings` o `http://api-gateway/api/products`.
  - El API Gateway consulta a Eureka para encontrar las instancias correctas de `booking-microservice` o `product-microservice`.
  - El API Gateway enruta las solicitudes a los microservicios correspondientes y gestiona la autenticación, autorización y balanceo de carga.

## Microservicios con Spring Boot - Autenticación y Autorización, Keycloak Server - Parte 1

### Docker - Keycloak

Keycloak is an open source identity and access management solution

 <https://www.keycloak.org/getting-started/getting-started-docker>

**Keycloak** es una solución de gestión de identidad y acceso (IAM) de código abierto desarrollada por Red Hat. Su propósito principal es facilitar la autenticación y autorización para aplicaciones y servicios modernos<sup>1</sup>. Aquí tienes algunos puntos clave sobre Keycloak:

1. **Inicio de Sesión Único (SSO):** Permite a los usuarios iniciar sesión una sola vez y acceder a múltiples aplicaciones sin necesidad de volver a autenticarse.
2. **Sopporte para Protocolos Estándar:** Es compatible con protocolos de federación de identidad como SAML v2, OpenID Connect (OIDC) y OAuth2.
3. **Gestión Centralizada:** Permite centralizar la gestión de identidades y accesos, facilitando la administración y el control de usuarios y permisos.
4. **Integraciones Fáciles:** Ofrece integraciones listas para usar con servicios de inicio de sesión social, bases de datos LDAP y otros proveedores de identidad.

5. **Escalabilidad y Alta Disponibilidad:** Puede ser escalado y configurado para alta disponibilidad mediante clústeres de servidores.
6. **Facilidad de Implementación:** Facilita la implementación de autenticación fuerte y métodos de autenticación adicionales como OTP (One-Time Password) a través de aplicaciones como FreeOTP o Google Authenticator.

Keycloak es una herramienta poderosa que ayuda a los desarrolladores a centrarse en la funcionalidad empresarial sin preocuparse por los aspectos de seguridad de la autenticación

comando que sirve para crear una imagen de docker en mi pc, se puede cambiar el puerto de exposición, en este caso tengo la apigateway corriendo en el puerto 8080, por lo tanto se cambio a 9090.

```
docker run -p 8080:8080 -e KC_BOOTSTRAP_ADMIN_USERNAME=admin -e KC_BOOTSTRAP_ADMIN_PASSWORD=passw0rd
```

Un

**realm** en Keycloak es equivalente a un **tenant** porque ambos conceptos implican la creación de entornos aislados y autónomos dentro de una única instancia de un sistema, permitiendo la gestión independiente de usuarios, roles, configuraciones y recursos. Esta capacidad de aislamiento y personalización es clave para implementar soluciones de autenticación y autorización seguras y escalables.

## Realms ≠ client(keycloak)

cree un realm y un cliente en keycloak cual es su diferencia y para que sirve cada uno. explicame en base a esa estructura del api gateway

En Keycloak, un **realm** y un **cliente** son dos conceptos importantes que sirven para gestionar la autenticación y autorización de usuarios en diferentes aplicaciones y servicios. Vamos a explicarlos en el contexto de tu estructura de API Gateway.

### ¿Qué es un Realm?

Un **realm** en Keycloak es un espacio de gestión donde puedes definir y administrar usuarios, roles, grupos y configuraciones de autenticación y autorización. Cada realm es una entidad independiente y proporciona un entorno aislado dentro de Keycloak.

- **Propósito:** Un realm se utiliza para agrupar usuarios y configuraciones de seguridad específicas para una o varias aplicaciones.
- **Aislamiento:** Los datos y configuraciones dentro de un realm son completamente independientes de otros realms, permitiendo una separación clara entre diferentes aplicaciones o servicios.

### ¿Qué es un Cliente en Keycloak?

Un **cliente** en Keycloak representa una aplicación o servicio que desea utilizar Keycloak para autenticar y autorizar usuarios. Los clientes pueden ser aplicaciones web, aplicaciones móviles u otros servicios que necesiten autenticación.

- **Propósito:** Un cliente define las configuraciones específicas para la autenticación y autorización de una aplicación. Incluye detalles como las URLs de redirección, las credenciales y los roles permitidos.
- **Interacción:** Los clientes interactúan con Keycloak para verificar la identidad de los usuarios y obtener tokens de acceso para permitir el acceso seguro a recursos protegidos.

Sí, en Keycloak, un **realm**, un **cliente** y un **usuario** son conceptos diferentes, cada uno con su propio propósito y funcionalidad. Vamos a desglosar estos términos para entender mejor sus diferencias y cómo se relacionan entre sí:

## Realm

- **Definición:** Un realm es un espacio de gestión dentro de Keycloak que proporciona aislamiento completo de recursos y datos.
- **Propósito:** Permite agrupar y gestionar usuarios, roles, grupos, y configuraciones de autenticación y autorización para una o varias aplicaciones.
- **Ejemplo:** Podrías tener un realm llamado `example-realm` que gestiona la autenticación y autorización de todos los usuarios de una empresa o una colección de aplicaciones.

## Cliente

- **Definición:** Un cliente en Keycloak representa una aplicación o servicio que utiliza Keycloak para la autenticación y autorización.
- **Propósito:** Define las configuraciones específicas para la autenticación y autorización de una aplicación, incluyendo detalles como las URLs de redirección, las credenciales y los roles permitidos.
- **Ejemplo:** Dentro del realm `example-realm`, podrías tener un cliente llamado `api-gateway` para tu API Gateway, y otro cliente llamado `product-microservice` para tu microservicio de productos.

## Usuario

- **Definición:** Un usuario en Keycloak es una persona o entidad que se autentica y accede a las aplicaciones gestionadas por Keycloak.
- **Propósito:** Los usuarios tienen credenciales (como nombre de usuario y contraseña) y pueden ser asignados a roles y grupos para definir sus permisos y acceso dentro de las aplicaciones.
- **Ejemplo:** Un usuario llamado `john.doe` podría ser parte del realm `example-realm` y tener acceso a los clientes `api-gateway` y `product-microservice` con roles específicos como `admin` o `user`.

## Que es un mecanismo de Token Relay

Un

**mecanismo de token relay** se refiere a la transmisión de un token de autenticación a través de diferentes partes de un sistema, por ejemplo, desde un cliente hasta un API y luego a un microservicio. Sirve para mantener la seguridad y la autenticación entre las distintas capas, asegurando que el token se valide y propague

correctamente a lo largo del flujo de trabajo.

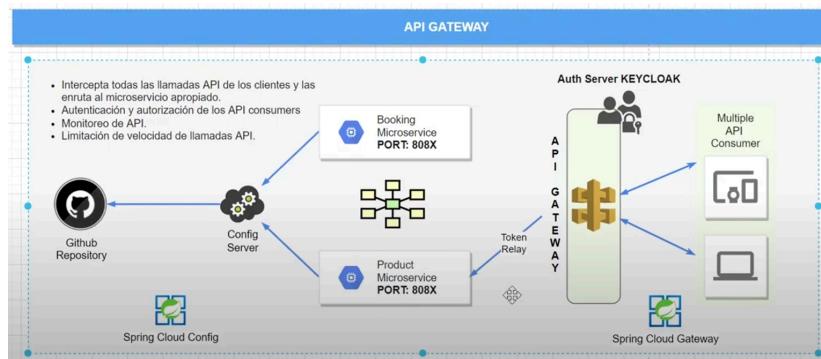
## Microservicios con Spring Boot - Autenticación y Autorización, Keycloak Server - Parte 2

```
<!-- https://mvnrepository.com/artifact/org.springframework.security/spring-security-oauth2-jose -->
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
    <version>6.4.2</version>
</dependency>
```

El nombre

**JOSE** se refiere al **JSON Object Signing and Encryption**. Es una familia de especificaciones para la firma y el cifrado de datos estructurados como JSON. Incluye varias especificaciones estándar que forman parte de un grupo de tecnologías diseñadas para asegurar datos JSON. Aquí hay una breve descripción de las componentes principales:

Token Relay → me transmite el token al microservicio invocado:



acabe de loguearme pero, no tengo credenciales, ya que aun no esta funcionando el token realy.



## Problemas de credenciales al acceder al microservicio product-microservice

ya me funciono, porque es necesaria esa propiedad?

`spring.security.oauth2.client.registration.spring-cloud-gateway-client.scope=openid`  
que funciona cumple?

¡Me alegra saber que funcionó! La propiedad `spring.security.oauth2.client.registration.spring-cloud-gateway-client.scope=openid` es crucial en la configuración de tu cliente OAuth2 porque define los alcances (scopes) solicitados al servidor de autorización (Keycloak) cuando se autentica al usuario.

### ¿Qué es un Scope en OAuth2?

En OAuth2, un **scope** es un mecanismo que permite a un cliente solicitar acceso a recursos específicos en nombre de un usuario. Los scopes limitan el acceso que un token puede otorgar, ofreciendo una capa adicional de seguridad y control sobre los recursos a los que se puede acceder.

#### Función del Scope `openid`

El scope `openid` es una parte fundamental del protocolo **OpenID Connect** (OIDC), que se basa en OAuth2. Cuando se incluye el scope `openid` en la solicitud de autenticación, estás indicando que deseas utilizar las capacidades de autenticación de OpenID Connect además de las de autorización de OAuth2. Aquí hay algunas funciones clave del scope `openid`:

##### 1. Autenticación del Usuario:

- Permite que el cliente autentique al usuario y obtenga información sobre su identidad.

##### 2. ID Token:

- Al incluir `openid` como scope, el servidor de autorización (Keycloak) devolverá un ID Token además de un Access Token. El ID Token contiene información del usuario, como su nombre, dirección de correo electrónico, etc.
- Este ID Token es crucial para que el cliente y otros servicios backend puedan autenticar y autorizar al usuario de manera segura.

##### 3. Información del Usuario:

- Facilita el acceso a los endpoints del usuario en el servidor de autorización (como `/userinfo`), lo que permite obtener información adicional sobre el usuario autenticado.

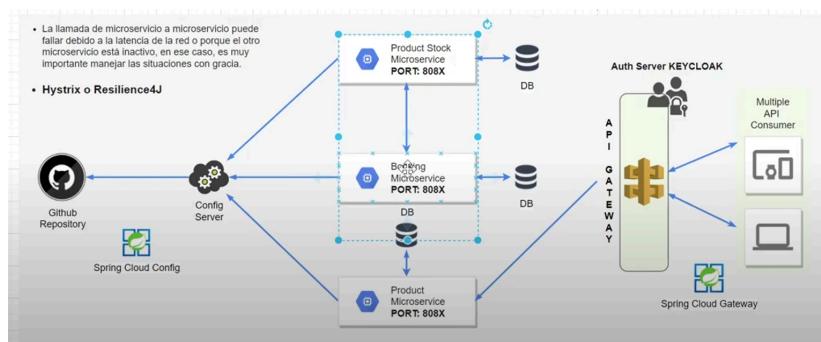
### Ejemplo de Solicitud de Token:

Cuando `spring.security.oauth2.client.registration.spring-cloud-gateway-client.scope=openid` se incluye en tu configuración, la solicitud de autenticación que se envía a Keycloak incluirá el scope `openid`. Keycloak reconocerá esto y proporcionará un ID Token junto con el Access Token en la respuesta.

### Resumen:

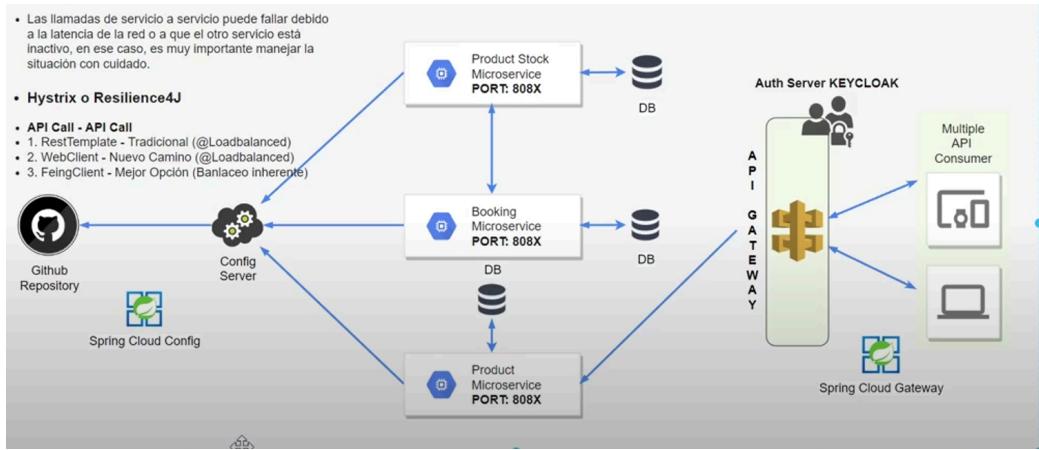
- **Scope `openid`**: Indica que deseas utilizar OpenID Connect para autenticar al usuario además de autorizar el acceso a recursos protegidos.
- **ID Token**: Proporciona un token adicional que contiene información sobre la identidad del usuario.
- **Seguridad Mejorada**: Ofrece una capa adicional de control y seguridad sobre los recursos a los que puede acceder el token.

## Microservicios con Spring Boot - Patrón Circuit Breaker - Parte 1



7Tdqh:WAmrNLvZk → contraseña empleabilidad

## Microservicios con Spring Boot - Patrón Circuit Breaker - Parte 2



## página de compatibilidad de springboot cloud