resumen

Temario

Core Java
Spring Boot
Spring Core - IoC
Spring MVC
Spring AOP
Spring Data JPA
Spring REST
Spring Security
Docker
Microservices
Spring AI

Java basico

javac

Es el compilador que nos provee el JDK, convierte el codigo de java en bytecode, que luego es utilizado por la JVM para correr el programa. La JVM convierte el codigo bytecode en codigo maquina, que puede ser ejecutado por el procesador

jshell

Consola de comandos de java que permite ejecutar codigo sin crear clases o un metodo principal, sirve probar fragmentos de codigo u aprender.

Como corre java?

Compilador: se encarga de convertir el human-readable code en bytecode.

JVM: Java Virtual Machine, transforma el btecodigo en machine code y lo ejecuta.

JRE: Provee a la JVM de las librerias y configuraciones necesarias para funcionar.

java -> **COMPILADOR** -> bytecode -> **JVM** -> machine code -> **RUN**

JRE vs JVM

JRE es un paquete que incluye a la JVM, pero ademas tiene archivos de configuracion y las librerias necesarias para que la JVM pueda correr el codigo.

JDK

Paquete que contiene un JRE, por lo tanto una JVM, y tambien un compilador de java (javac).

Compilar y ejecutar un .java

```
javac ${nombreArchivo}.java

java ${nombreArchivo}
```

Ejecutar un .jar

`java -jar app.jar`

Paquetes

El codigo .java corre dentro de paquetes. Es recomendable para vscode que el proyecto tenga una carpeta "src" en la raiz, en donde esten contenidos los archivos .java u otros paquetes.

Variables primitivas y sus wrappers

Tipo primitivo	Tamaño	Rango / Descripción	Clase Wrapper
`byte`	1 byte	-128 a 127	`Byte`
`short`	2 bytes	-32,768 a 32,767	`Short`
`int`	4 bytes	±2 mil millones aprox	`Integer`
`long`	8 bytes	±9 quintillones aprox	`Long`
`float`	4 bytes	Precisión simple (32 bits IEEE 754)	`Float`
`double`	8 bytes	Precisión doble (64 bits IEEE 754)	`Double`
`char`	2 bytes	Representa un carácter Unicode	`Character`
`boolean`	1 bit	Técnicamente usa 1 bit, pero en objetos puede ocupar 1 byte o más	`Boolean`

En java, todo es un objeto, excepto los tipos primitivos. Estos son mas eficientes de utilizar, pero no tienen las funcionalidades que nos otorgan sus wrappers. Por ejemplo, int es mas rapido, porque no es necesario alojar un objeto Int en el heap. Pero su wrapper, Integer, es mas lenta de utilizar, pero nos provee de metodos como el toString() y muchos mas.

Method overloading

Una clase puede tener metodos con los mismos nombres siempre y cuando tengan distintos parametros.

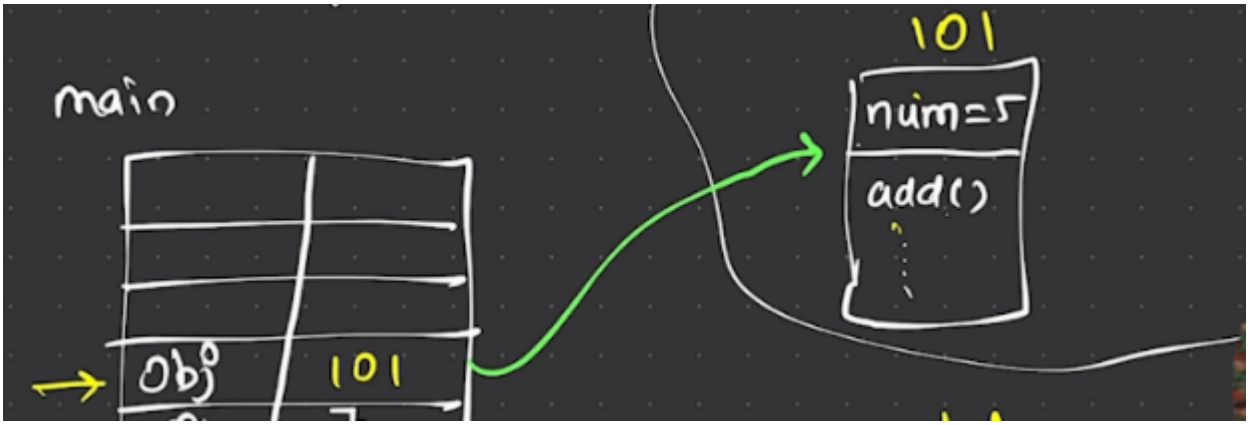
Method overriding

Es cuando reemplazamos el comportamiento de un metodo existente al heredar una clase, o implementamos el metodo de una interfaz.

Stack

Es una pila con disciplina LIFO, en formato KEY:VALUE . Cada key es el nombre de una variable, y el value es su valor. Cada metodo de una clase, va a tener su propio stack.

Al instanciar un objeto y asignarlo a una variable “obj” , el objeto, junto con sus variables de instancia y metodos, seran guardados en el heap, mientras que el value de “obj” será la direccion en memoria de este objeto en el heap. Entonces las variables objeto, son en realidad, referencias al objeto real.



Heap

Es un espacio en memoria que puede variar en tamaño, ahi se almacenan los objetos instanciados. Las variables de referencia, contienen la direccion del objeto instanciado en el heap.

Jagged array

An array that contains other arrays that vary in size.

3D array

An array that contains an array of arrays. int[][][]

String y StringBuffer

Al instanciar un objeto de tipo String, una variable en el stack va a hacer referencia a la direccion del objeto en el heap. La cadena de caracteres del string va a estar en el heap, por ejemplo “hola”. Si yo creo una

Metodos y variables estaticos

Lo estatico pertenece a la clase y no al objeto. Se pueden acceder a valores y metodos directamente de una clase, sin la necesidad de instanciarla. No se pueden utilizar variables de instancia en un metodo estatico.

- clases: no pueden ser estaticas menos que sean una clase dentro de otra clase (clase interna).
- variables y metodos: pueden ser estaticos, osea que se puede acceder a ellos sin la necesidad de instanciar la clase.

Static block

El bloque estatico ejecuta codigo estatico de una clase, justo antes de ser instanciada, y solo UNA vez en todo el programa. Sirve para inicializar recursos, asignar valores a las variables estaticas, etc. Si se crea un segundo objeto de esta clase, el codigo no correrá. Si queremos ejecutar el bloque estatico sin instanciar la clase, usamos el Class.forName()

Palabra reservada "final" (como constant en C)

Una variable, un metodo o una clase pueden ser constantes con FINAL:

- variable final: no puede ser modificada
- metodo final: no puede ser reemplazado (override) pero si puede ser sobrecargado (overloaded)
- clase final: no puede ser heredada

Clases y Herencia

No lo voy a resumir porque ya esta muy visto, pero voy a remarcar conceptos importantes.

- java no soporta herencia multiple. Supongamos las clases independientes A,B,C. etonces C puede heredar de A o de B, pero no de ambas a la vez.
- El metodo super() es llamado implicitamente aunque no lo hayamos escrito.
- Todas las clases en java extienden la clase Object

equals y hashCode()

- **equals**(), por defecto, calcula si dos referencias apuntan al mismo objeto en memoria, es decir, si dos variables, o dos objetos representan el mismo. Este metodo puede ser reemplazado, para comparar si dos objetos distintos, tienen las mismas variables internas, como el mismo nombre.
- **hashCode**() calcula el hash de un objeto en funcion de sus variables.

Upcasting y Downcasting en clases

Supongamos que tengo dos clases A y B que extiende A.

- **upcasting**: creo B y lo casteo en una referencia a una clase A. Esto se puede hacer implicitamente y es seguro, ya que B es un tipo de A.
- **downcasting**: creo una clase de A y lo casteo en una referencia a una clase B. Esto se debe castear de forma explicita y es inseguro, ya que A no conoce a B.

Clases abstractas

Son clases que no se pueden instanciar por si solas, pero si se pueden extender. Pueden contener metodos comunes para sobreescribir opcionalmente, o metodos abstractos que solo estan declarados, y obligatoriamente deben ser implementados. Por ejemplo, la clase Animal, debe ser abstracta, ya que solo es posible instanciar por ejemplo, un Perro o un Gato, pero un animal, es un concepto abstracto.

Si la clase abstracta "animal" tiene un constructor (String nombre), entonces al extender esta clase con una concreta, deberemos llamar a super("nombre") para inicializar correctamente la clase.

El metodo abstracto se declara asi: **public abstract void method();**

Clases internas

Una clase puede contener dentro otras clases, ya sean abstractas o concretas. Su forma de instanciacion varía dependiendo de eso:

- A a = new A();
- A.B b = new A.B(); // For static inner classes
- A.C c = a.new C(); // For non static inner classes

Interfaces

Son estructuras, en donde se declaran metodos (semejante a una clase abstracta) solo que no son clases, y por defecto, todos los metodos declarados son publicos y abstractos. Sus variables son FINAL y STATIC por defecto.

Hay 3 tipos de interfaces:

- comunes: solo declaran metodos
- funcionales: implementan metodos
- mock: estan vacias

Enum

Es un tipo de clase, que extiende de la clase java.lang.Enum que sirve para definir constantes, y reutilizarlas facilmente evitando errores.

Anotaciones

Son maneras de aclarar nuestras intenciones en el codigo para prevenir errores logicos

Serializacion / Deserializacion

La primera consiste en persistir un objeto en el almacenamiento persistente, la segunda consiste en cargar un objeto del almacenamiento persistente al heap.

Throwables

Las excepciones y los errores extienden los Throwables. Los errores detienen obligatoriamente la ejecucion del codigo, mientras que las excepciones, nos permiten ser manejadas con el bloque try-catch. Podemos usar el catch multiples veces, para atrapar y manejar distintos tipos de excepciones que se arrojen durante la ejecucion del bloque.

Por otro lado, las excepciones se divicen en checked y unchecked. Cuando una linea de codigo puede arrojar una excepcion chequeada, el compilador nos obliga a manejar esa excepcion. Si no es chequeada, no nos obliga.

Exception ducking

Cuando una clase tiene metodos que tiran exceptiones, podemos hacer que estos metodos, en vez de interrumpir la ejecucion del sistema en alguna linea, arrojen la excepcion a quien esta llamandola, para que este maneje la excepcion. Se usa la palabra reservada “throws” en el metodo, y el que llama tiene que manejar tal excepcion con un bloque try-catch.

Entrada y salida

La entrada y salida legacy en java es un poco mas complicada que el input("") en python. Pero se puede estudiar:

Buffer: Es un espacio en memoria en donde se guarda la secuencia de caracteres que nosotros ingresamos en consola, la cual luego sera interpretada por el programa. Cuando presionamos enter, la cadena ingresa al buffer junto con un salto de linea al final: \n

System.in.read(); obtiene el primer caracter del buffer y lo devuelve en su codigo ascii, pero deja abandonado el salto de linea dentro del buffer. Si luego intentarmos usarlo nuevamente, o por ejemplo, el buffered reader, se va a leer el salto de linea del comienzo y nos va a devolver una linea vacia. Por esto, se recomienda consumir el salto de linea restante nuevamente con: System.in.read();

Ejemplo

- Cuando escribís en consola y apretás **Enter**, todo lo que tipeaste (incluido el `\n` final) queda en un **buffer de entrada**.

- Los métodos de `Scanner` “consumen” del buffer:
  - `nextInt()`, `nextDouble()`, etc. leen **solo el número** y **dejan el** `\n` en el buffer.
  - `next()` lee **una palabra** (hasta espacio o `\n`) y **deja** el `\n` si lo hubiera.
  - `nextLine()` lee **toda la línea completa**, incluido el `\n` (que descarta), y devuelve el texto **sin** el `\n`.

```
Scanner input = new Scanner(System.in);
// Buffer: []
System.out.print("Enter first number: ");
int number1 = input.nextInt();
// Buffer: [\n]
input.nextLine(); // Vacío el buffer para poder ingresar un valor
proximamente.
// Buffer: []
System.out.print("Enter a string: ");
String str = input.nextLine();
// Buffer: []
```

Métodos que no consumen el salto de línea final:

Cualquier método de `Scanner` que **no sea** `nextLine()`, deja el `\n` en el buffer cuando presionás Enter.

- `nextInt()`
- `nextDouble()`
- `nextFloat()`
- `nextLong()`
- `nextByte()`
- `nextShort()`
- `next()`

Hilos (Threads)

Para crear hilos, y ejecutar codigo de manera concurrente debemos crear una clase "x" que extienda a Thread. Esta clase debe tener un metodo run(), que es el que va a ser ejecutado automaticamente al llamar al metodo start() de la clase "x" (ese metodo start lo hereda, no debemos crearlo). Al llamar a multiples start() de distintos threads, se llega a la concurrencia, en donde el procesador puede ejecutar al mismo tiempo, la misma cantidad de hilos que nucleos tenga. Generalmente tienen 4 u 8 nucleos. Que pasa si hay mas hilos que nucleos? Quedan en espera, y el sistema operativo tiene algoritmos de planificacion para el procesador (scheduling), para poder ejecutar una parte de un hilo, una parte de otro y asi, siguiendo una disciplina, como RR (Round Robin), SJF (small jobs first), FCFS (first come first server), etc. Entonces, no podemos saber o elegir el orden en el que se va a

ejecutar nuestro codigo, si esta corriendo en forma de hilos, ya que el SO va a decidirlo de acuerdo a su politica de planificacion.

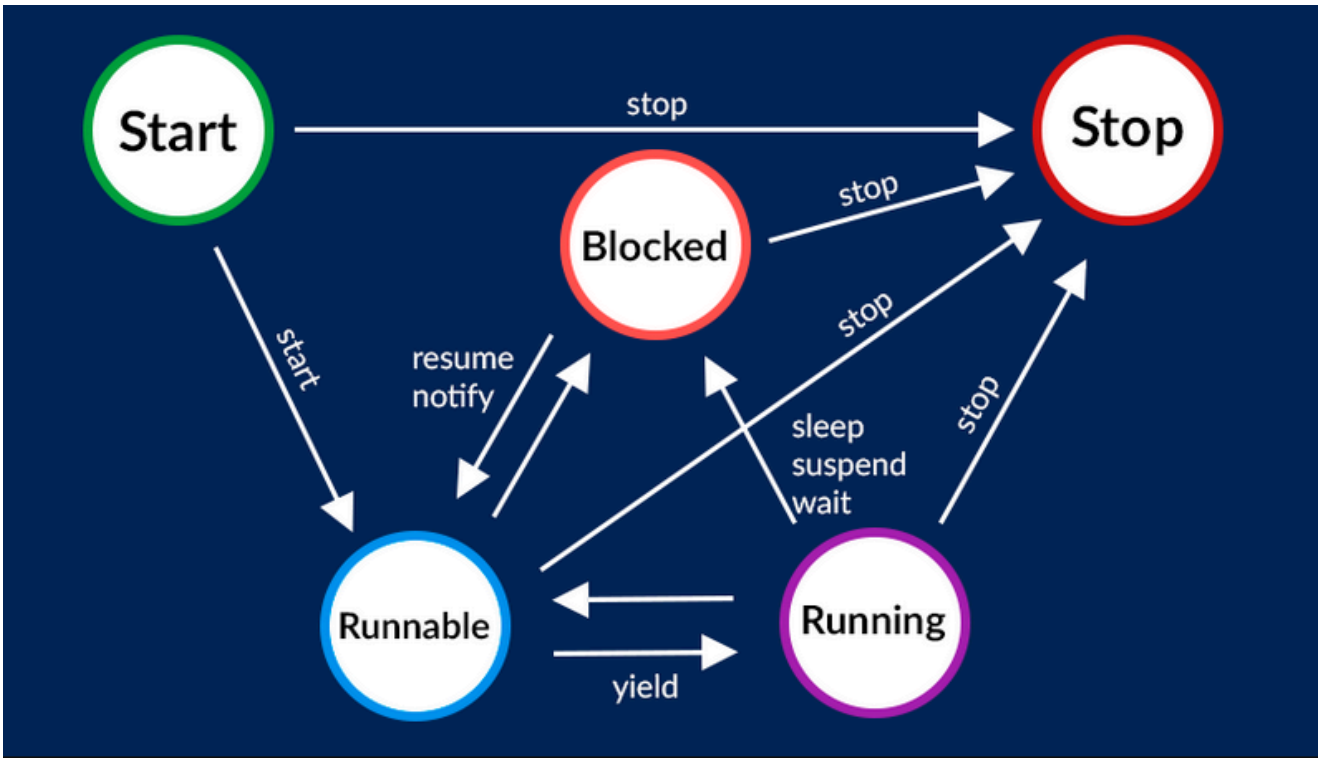
Ejemplo

```
// Definir clase que implementa runnable (metodo run())
public class PrinterTask implements Runnable {
    public void run() {
        System.out.println(Thread.currentThread().getName());
    }
}

// Crear objeto thread y le damos como parametro nuestra implementacion de
Runnable y un nombre para el trabajador (hilo).
Thread t1 = new Thread(new PrinterTask(), "Worker-1"); // (Runnable, Name)
t1.start(); // Ejecuta el thread.

try {
    t1.join(); // Quien ejecutó el thread, ahora debe esperar a que este
finalize. Caso contrario, se puede terminar el programa antes de que el
thread termine la ejecución.
} catch (Exception e) {
}
```

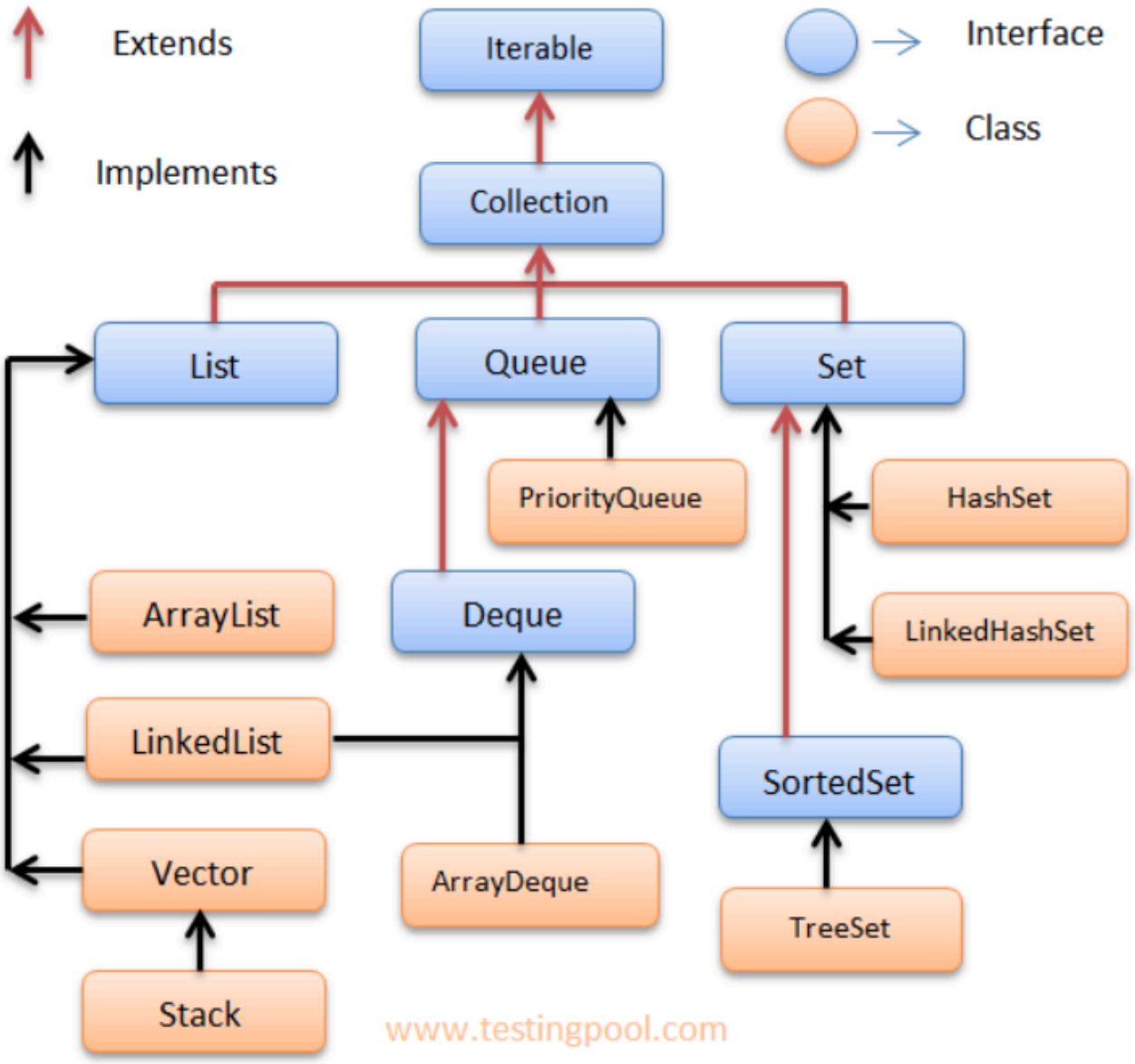
Concepto de thread safe

Cuando dos hilos tocan funciones o variables a la vez, se puede producir comportamiento inesperado. El clasico ejemplo del banco: una cuenta bancaria tiene $7000 y dos personas sacan dinero en el mismo instante, desde dos cajeros distintos. En total retiraron $14000, excediendo el dinero que habia en la cuenta. Para solucionar esto, podemos hacer que una variable o funcion, solo pueda ser accedida por un hilo a la vez, y luego por otro una vez este libre. Para esto, se utiliza la palabra reservada **synchronized**.

Estados de un hilo



Colecciones (Collection)



List: Almacena Objetos ordenados y se permiten duplicados

Set: Almacena objetos unicos (no se permiten duplicados)

Map: Almacena pares key:value

Stream api

Un stream es una secuencia de elementos que se pueden cada uno con los siguientes metodos:

- coleccion.stream()
  - .filter(...) // paso 1: filtrar
  - .map(...) // paso 2: transformar
  - .sorted(...) // paso 3: ordenar (opcional)
  - .collect(...) // paso 4: recoger resultado

Tambien se puede utilizar el parallelStream() para que se opere utilizando varios hilos, y obtener mas performance. No vale la pena utilizarlo con arrays cortos, ya que la diferencia de tiempo se vuelve despreciable, incluso la creacion de hilos toma un tiempo extra.

Ejemplo

```
List<String> products = Arrays.asList("Laptop", "Pen", "Notebook",
"Headphones", "Smartphone");

List<String> filtered = products.stream()
        .filter(x -> x.length() > 5) // Quita los menores a 5 caracteres
        .map(x -> x.toUpperCase()) // Convierte todo a mayusculas
        .sorted() // Ordena todo alfabeticamente (String es comparable)
        .collect(Collectors.toList()); // Devuelve el resultado como una
lista

System.out.println(filtered);
```

CheatSheet

> CheatSheet
>
> ```
> // metodos de ArrayList<>()
> list.add(element);
> list.add(index, element);
> list.get(index);
> list.set(index, newElement);
> list.remove(index);
> list.remove(Object obj);
> list.contains(element);
> ```

```
list.size();
list.clear();
list.isEmpty();
list.indexOf(element);
list.subList(fromIndex, toIndex);
list.toArray(array);

// pares ordenados
AbstractMap.SimpleEntry<String,Integer> a = new
AbstractMap.SimpleEntry<>("monitor",300);
a.getValue()
a.setValue()
a.getKey()

// metodos de String
.toCharArray()
.charAt(index)
.contains("texto")
.length()
.toLowerCase()
.split("\\s+"); // devuelve un String[] con cada palabra, con la regex
separa por espacios

// arrays de datos primitivos
int[] arr = new int[3];
int[] arr1 = {1,2,3};
char[] arr1 = {'a','b'};

// imprimir array primitivo
Arrays.toString(arr)

// copiar arrays primitivos
Arrays.copyOf(arr,arr.length);

// atributos de array
.size
```

Dependencias

Son archivos .jar que podemos importar a nuestro proyecto y uilizar llamando a sus clases.

Apache Maven

Es una herramienta open source que nos ayuda a gestionar las dependencias (librerias externas) de un proyecto de java.

Por defecto, nos descarga las dependencias en los siguientes directorios:

- En **Linux/Mac**: `/home/usuario/.m2/repository`
- En **Windows**: `C:\Users\usuario\.m2\repository`
  Una vez descargada la dependencia, es globalmente accesible para todos los proyectos que utilizen maven.

Comandos

instalacion (linux): `sudo apt install maven`

ejecucion: `mvn`

creacion de proyecto basico:

```
mvn archetype:generate -DgroupId=com.ejemplo.app \
    -DartifactId=app-name \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DinteractiveMode=false
```

Compilar proyecto a .jar (target/app.jar)
`mvn package`

Limpiar target antes de compilar (toma mas tiempo pero es mas seguro)
`mvn clean package`

Pom.xml

Algunos proyectos requieren librerias externas, que son archivos .jar que podremos utilizar. Maven, nos permite gestionar estas librerias de una forma comoda por medio de un archivo llamado pom.xml Este archivo se traduce en el effective pom al momento de la ejecucion, el cual contiene datos mas detallados.

Las dependencias descargadas por maven, se guardan en la carpeta oculta “.m2” en el home directory.

Se pueden descargar manualmente los .jar en https://mvnrepository.com/

Arquetipos

Son configuraciones iniciales de maven, para distintos tipos de proyectos. Incluye archivos de configuracion y estructura de carpetas.

Springboot

Generalmente, para crear un proyecto springboot accedemos a https://start.spring.io/ realizamos la configuracion necesaria y descargamos el proyecto. Pero ¿que pasa si el servidor esta caido o no queremos depender de un sitio web?

Inicializar proyecto springboot sin spring initializer

Se inicializa con el siguiente codigo:
inicializar springboot sin spring initializer

Lombok (org.projectlombok)

Es una libreria que nos permite generar automaticamente los getters, setters, contructores, y demas funciones boilerplate de una clase, para ahorrarnos tiempo.

Solo hay que escribir las anotaciones elegidas encima de la clase:

- @**Getter**: genera todos los getters
- @**Setter**: genera todos los setters
- @**AllArgsConstructor**: genera un constructor con todos los argumentos
- @**NoArgsConstructor**: genera un constructor vacío
- @**RequiredArgsConstructor**: genera un constructor con los campos final o anotados con @NonNull
- @**Data**: incluye @Getter, @Setter, @ToString, @EqualsAndHashCode y @**RequiredArgsConstructor**
- @**Builder**: genera el patrón builder

Se recomienda utilizar las anotaciones **Data** y **AllArgsConstructor**, para reducir la escritura de codigo. Incluir **NoArgsConstructor** si se esta utilizando **hibernate**.

H2 (com.h2database)

Es una base de datos embebida que se ejecuta en memoria (RAM) y no persiste información por defecto. Es útil para pruebas rápidas, ya que Spring Boot la detecta automáticamente si está en el classpath y puede inicializarla usando los archivos schema.sql y data.sql ubicados en el directorio resources.

Ejemplo

resources/schema.sql

```
create table student(
    id INT PRIMARY KEY,
    name VARCHAR(50),
    avg INT
);
```

resources/data.sql

```
insert into student (id, name, avg) values (352798,'ian', 100);
```

Swagger

Herramienta para estandarizar y documentar APIs.

springdoc-openapi-starter-webmvc-ui

Capa de abstraccion para swagger core. Nos provee de una documentacion automatica para nuestra API, integrandose con **springboot**. Podemos acceder a ella mediante: http://localhost:8080/swagger-ui/index.html

https://mvnrepository.com/artifact/org.springdoc/springdoc-openapi-starter-webmvc-ui
Ejemplo:

```
<dependency>
  <groupId>org.springdoc</groupId>
  <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
  <version>2.8.9</version> <!-- Usa la última disponible -->
</dependency>
```

Spring Data Rest

Spring Data REST es un proyecto del ecosistema **Spring** que permite exponer automáticamente repositorios de datos como servicios RESTful, sin necesidad de escribir controladores (`@RestController`) manualmente. Crea automáticamente endpoints REST a partir de interfaces `Repository` . Internamente usa **HATEOAS** para estructurar las respuestas y sigue convenciones RESTful.

Bases de datos

JDBC (Java Database Connectivity)

Es una api de java, que se compone por un conjunto de interfaces y clases que permiten gestionar las conexiones a bases de datos sin importar su tipo. El JDBC debe ser implementado por cada driver especifico, como por ejemplo el driver de **postgresql**.

Jakarta persistence (JPA)

Es una interfaz que nos permite implementarla para persistir objetos en la base de datos, y abstraernos del manejo de consultas complejas (no requiere SQL). Una famosa implementacion de JPA es **hibernate**.

Hibernate

Implementacion de la interfaz JPA. Utiliza JDBC para realizar las consultas.

Spring data JPA

Capa de abstraccion que nos permite utilizar hibernate con menos boilerplate, y evitando realizar consultas SQL manuales. Utiliza internamente las librerias:

- `org.hibernate:hibernate-core`
- `jakarta.persistence:jakarta.persistence-api`
- `org.springframework.data:spring-data-jpa`

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

Para pruebas rapidas, podemos simplemente agregar la dependencia h2 y spring data lo va a reconocer y utilizar automaticamente sin configuracion, pero h2 debe ser la unica base de datos presente.

Metodos que nos provee la interfaz `JpaRepository` por defecto

```
List<T> findAll();
Optional<T> findById(ID id);
T save(T entity);
void deleteById(ID id);
boolean existsById(ID id);
long count();
```

Metodos personalizados que podemos agregar

```
public interface UserRepository extends JpaRepository<AppUser, Long> {

    Optional<AppUser> findByUsername(String username);

    Optional<AppUser> findByEmail(String email);

    List<AppUser> findByActive(boolean active);

    List<AppUser> findByUsernameContaining(String partOfUsername);

    List<AppUser> findByCreatedAtAfter(LocalDateTime date);

    List<AppUser> findByActiveTrueOrderByCreatedAtDesc();

    boolean existsByEmail(String email);

    long countByActiveTrue();

    void deleteByUsername(String username);
```

```
}
```

Anotaciones de persistencia jakarta e hibernate

Cuando trabajamos con entidades en hibernate, usamos anotaciones, que sirven para configurar las entidades, relaciones y tablas.

`@Entity`
Especificamos que una clase es una entidad para persistir. Se crea una tabla automaticamente con el nombre de la entidad en minuscula.

`@Table(name="student")`
Cuando queremos que la tabla tenga otro nombre, usamos esto.

`@Id`
Especificamos cual va a ser la Primary Key de la entidad. Cuando construyamos la clase, nos va a solicitar que le asignemos un ID (clave primaria) manualemente

`@Column(length = 600)`
Cantidad de caracteres maxima de una columna de tipo String

`@GeneratedValue(strategy = GenerationType.IDENTITY)`
Se declara debajo de @Id y sirve para que hibernate se encargue de asignarle un id automaticamente. No debemos especificarlo al construir la clase.

`@Column(name="student_name")`
Cambiar el nombre de una columna, cuando no queremos que tome el nombre del atributo en minuscula.

`@Transient`
Evita que se cree una columna para ese atributo de la entidad

`@ElementCollection
Crea una tabla auxiliar para cuando el atributo es una lista. La tabla contiene la llave foranea de la clase, y el elemento de la lista

`@Embeddable`
Sirve para que una clase no sea una entidad ni sea mapeada a una tabla, sino que se pueda embeber dentro de otra clase. Por ejemplo la clase Laptop es embeddable y la entidad Alumno posee el atributo Laptop, entonces cuando guardemos un Alumno, se van a mapear las columnas de la Laptop a parte de las del Alumno

`@OneToOne`
La entidad A tiene como atributo a la entidad B (pero solo un objeto B, no varios). Entonces si agregamos esta anotacion arribe de la declaracion de la variable B, en la tabla mapeada de A va a aparecer una FK correspondiente a B.

`@OneToMany`
La entidad A tiene como atributo una lista de entidades B, entonces se crea una nueva tabla de asociacion, en donde se asocia el id de A con el id de B.

Pero... por que una tabla de asociacion cuando cada objeto B podria tener una FK hacia A ?

En este ejemplo, un Student puede tener varias Laptops

Tenemos que especificarlo en ambas entidades:

```
@OneToMany(mappedBy = "student")
Laptop laptop;

@ManyToOne
@JoinColumn(name = "student_id")
Student student;
```

En la base de datos, cada laptop va a tener una columna que contiene el id del estudiante. Notar que en **mappedBy**, se coloca el nombre de la variable de referencia de la entidad opuesta, y en **name**, el nombre de la columna en la DB. El **mappedBy** sirve para declarar quien es el dueño. Al utilizarlo de un lado, declara que la entidad **NO** es dueña de la relacion, por lo tanto no debera poseer su FK, pero SI la otra entidad.

`@ManyToMany`
Sirve cuando ambas entidades poseen una lista de la entidad opuesta. Por ejemplo un classRoom posee una lista de alumnos, y cada alumnos tiene una lista de varios classRoom a los que asiste. Esto va a generar una tabla intermedia de asociacion que va a contener dos columnas, con la respectiva PK de cada entidad. Hay que notar que, aunque no haya dueño de la relacion (ninguna entidad tiene la PK de la otra en su tabla), hay que declarar un dueño igual para que hibernate esté contento (mappedBy).

```
@ManyToMany
private List<ClassRoom> classRooms;

@ManyToMany(mappedBy = "classRooms")
private List<Student> students;
```

Notas

- User es una palabra reservada en algunas bases de datos, por lo que no conviene tener una entidad o tabla propia llamada User

Spring framework

Spring

Spring, es un framework y a su vez un ecosistema de tecnologias para desarrollar aplicaciones en java, kotlin o groovy.

Nucleo

Spring nacio como un framework de inversion de control (IoC) e inyeccion de dependencias (DI). Su objetivo principal era facilitar la creacion de aplicaciones desacopladas y modulares.

Proyecto	Descripción breve
Spring Framework	El núcleo del ecosistema. Proporciona IoC (Inversión de Control), DI (Inyección de Dependencias), AOP (Programación Orientada a Aspectos), y soporte para programación modular.
Spring Boot	Facilita la creación de aplicaciones Spring listas para producción con mínima configuración. Incluye servidor embebido, autoconfiguración y dependencias agrupadas.
Spring MVC / Web	Módulo para crear aplicaciones web con el patrón Modelo-Vista-Controlador. Base para construir APIs REST.
Spring Data	Simplifica el acceso a bases de datos relacionales y NoSQL (JPA, MongoDB, Redis, etc.). Reduce el código repetitivo del DAO.
Spring Security	Provee autenticación, autorización y otras funcionalidades de seguridad para aplicaciones Java.
Spring Batch	Framework robusto para el procesamiento de datos en lote (batch), útil para tareas repetitivas y procesamiento de grandes volúmenes de datos.
Spring Cloud	Conjunto de herramientas para construir sistemas distribuidos y microservicios (configuración centralizada, descubrimiento de servicios, circuit breakers, etc.).
Spring Integration	Facilita la integración de sistemas usando patrones de integración empresarial (mensajería, colas, adaptadores, etc.).
Spring AMQP	Abstracción para trabajar con sistemas de mensajería como RabbitMQ, utilizando el protocolo AMQP.
Spring for GraphQL	Soporte para construir APIs GraphQL con Spring. Reemplaza al viejo `spring-graphql-java`.
Spring HATEOAS	Facilita la creación de APIs RESTful que siguen el principio HATEOAS (Hypermedia as the Engine of Application State).
Spring Shell	Permite crear interfaces de línea de comandos interactivas usando Spring. Útil para herramientas internas.
Spring Mobile (Deprecated)	Fue usado para detectar dispositivos móviles y adaptar el contenido, ya no se mantiene.

Proyecto	Descripción breve
Spring Web Services	Permite crear y consumir servicios web SOAP con contrato basado en WSDL/XSD.
Spring Session	Maneja sesiones de usuarios fuera del contenedor, útil para escalabilidad (e.g. Redis, JDBC, etc.).

Conceptos fundamentales

IOC (Inversion Of Control)

Antes de hablar sobre spring , hay que saber unos conceptos, como el de inversion de control (IoC). IoC es un **PRINCIPIO** que refiere a como las creacion y el control de objetos es transferido desde el programador a spring. Lo que suele diferenciar a las aplicaciones **es** la logica de negocio, por lo que seriamos mas productivos si no tuviesemos que estar creando y manteniendo objetos por nuestra cuenta. El control de inversion permite que delegemos este trabajo y nos enfoquemos en la logica del negocio.

DI (Dependency Injection)

Es un patron de diseño que utiliza el principio de IoC.

Una dependencia, es un objeto que es requerido por otro para poder funcionar. Por ejemplo, la clase auto tiene un objeto motor.

Sin inyeccion de dependencias, el auto o el programador, se encarga de crear un motor para poder funcionar y esto genera acoplamiento. Con DI, el sistema se encarga de inyectar el motor dentro del auto, sin que nos debamos preocupar por como implementarlo, en spring, usamos anotaciones como @**Component** y @**Autowired**

Beans

Existen gracias al concepto de inversion de control. **Un bean, es un objeto que es gestionado por el contenedor de Spring**. Mas tecnicamente, es una instancia de una clase que spring crea, configura y mantiene durante su ciclo de vida.

Pueden tener distintos scopes, como **singleton** (te da el mismo objeto al buscarlo), o **prototype**, que genera un nuevo objeto cada vez que se busca.

A veces, tenemos una interfaz implementada por varios beans, y una clase que utiliza a la interfaz, es decir, puede usar cualquiera de los beans que la implementan. Si usamos el @**Autowired**, springb no va a saber cual bean elegir ya que todos son posibles candidatos. Por este motivo, se asigna un bean por defecto con la anotacion @**Primary** en el bean. Tambien, se puede agregar una anotacion en la clase en donde especifique cual tipo de bean se debe inyectar por defecto: @**Qualifier("nombreBean")**

Declaracion de Beans

Spring Framework nos proporciona 3 maneras distintas de declarar beans:

1. **XML (Forma antigua)**:** Se crea un archivo xml de contexto en donde se referencian las clases que seran beans, y de ser necesario se les otorgan atributos de inicializacion.

   Declaracion del xml de configuracion context.xml

   Alien es una clase personalizada que nosotros creamos. En este caso tiene un nombre y una edad. La clase debe tener un constructor sin parametros, y sus setters y getters.

   Accediendo al bean anteriormente declarado:

2. **Configuracion basada en java**

   En vez de un .xml, se crea un archivo de configuracion .java para declarar y configurar beans. En este ejemplo se declara el bean Laptop. En este caso se obtiene el bean por tipo, pero se puede obtener por nombre tambien.

```java
@Configuration
public class AppConfig {
    @Bean
    @Primary
    public Laptop laptop() {
        return new Laptop();
    }

    @Bean
    public Alien alien(Computer laptop) {
        Alien alien = new Alien();
        alien.setLap1(laptop);
        return alien;
    }
}
```

```java
// Java based bean config
ApplicationContext context = new AnnotationConfigApplicationContext(...
componentClasses:AppConfig.class);
Alien alien = (Alien) context.getBean(name:"alien");
alien.code();
alien.useComputer();
}
```

3. Anotaciones

Usamos un archivo de configuracion como en el punto 2, solo que en vez de declarar beans en ese archivo, se declaran en el codigo mismo, en cada clase correspondiente. Tambien, se pueden conectar estos objetos entre sí mediante la anotacion @autowired. El AppConfig requiere la anotacion de ComponentScan, para poder buscar los beans declarados mediante la anotacion @Component.

El codigo en main sigue intacto.

```java
@Configuration
@ComponentScan("com.example")
public class AppConfig {
    // @Bean
    // @Primary
```

```java
@Component
public class Alien {
    private String name;
    private int age;

    @Autowired
    private Computer lap1;
```

```java
// Java based bean config
ApplicationContext context = new AnnotationConfigApplicationContext(...
componentClasses:AppConfig.class);
Alien alien = (Alien) context.getBean(name:"alien");
alien.code();
alien.useComputer();
```

Spring MVC (Modelo vista controlador)

Descripcion

Es un framework del ecosistema de spring que se utiliza para desarrollar aplicaciones web siguiendo el patron de arquitectura MVC. Antiguamente se combinaba con JSP, pero hoy en dia se utiliza mas comunmente thymeleaf.

Posee los siguientes componentes:

- **DispatcherServlet**: Enrutador central que recibe las peticiones HTTP.
- **Controller** (`@Controller`, `@RestController`): Maneja la lógica de negocio según la petición.
- **Model**: Datos que se envían a la vista o como respuesta JSON.
- **View** (`.jsp`, `.html`, Thymeleaf, etc.): Representación visual de los datos.
- **ViewResolver**: Decide cómo transformar los datos del modelo en una vista.

Flujo tipico:

- El usuario entra a `http://localhost:8080/hola`.
- Spring ejecuta el método del controlador que maneja esa ruta.
- El controlador **consulta o genera datos** desde el modelo.
- Devuelve el nombre de la vista a mostrar (`hola.html`).
- Spring procesa la vista con Thymeleaf y devuelve el **HTML final** al navegador.

JSP

JSP (JavaServer Pages) es una tecnología antigua que permite generar contenido HTML dinámico desde el servidor Java, mezclando HTML con código Java incrustado. Se usaba mucho antes de que existiera Spring Boot. Hoy en dia se considera obsoleto y en su lugar se utillizan tecnologias como thymeleaf o freemarker, tambien se suele utilizar un frontend por separado como react, angular, vue.

Spring AOP (Aspect Oriented Programming)

Es un modulo de spring y un paradigma de programacion que nos permite separar el codigo de logica del negocio, de las preocupaciones transversales (cross-cutting concerns) para tener un codigo mas limpio, legible y mas facil de mantener.

Ejemplos de preocupaciones transversales:

- Logging
- Seguridad
- Excepciones
- Metricas

Generalmente para cada metodo de un service (en la Arquitectura por capas) se requiere de la implementacoin de excepciones, metricas, etc. Y nos quedamos con metodos muy largos y dificiles de leer. Esto es poco conveniente cuando nuestro interes principal es la logica del negocio.

Conceptos teoricos

- Aspect: La clase que contiene la logica transversal
- Advice: El codigo que se ejecuta en un punto especifico, segun el momento en el que se debe ejecutar, tenemos Tipos de advice:
- Join Point: Punto de ejecucion en donde se pueden insertar advices (siempre son llamadas a metodos en este contexto)
- Pointcut: Expresion que selecciona los join points en donde se aplicara el advice. Se define con una notacion y una sintaxis especial en formato string.
- Target object: El metodo que estamos interceptando
- Proxy: Es el objeto que spring crea por sobre el target para poder acceder a traves de el al metodo original del target.

Tipos de advice

Tipo de advice	Anotación	¿Cuándo se ejecuta?
Before	`@Before`	Antes del método objetivo
After	`@After`	Después (haya fallado o no)
AfterReturning	`@AfterReturning`	Solo si el método termina sin errores

Tipo de advice	Anotación	¿Cuándo se ejecuta?
AfterThrowing	`@AfterThrowing`	Solo si el método lanza una excepción
Around	`@Around`	Antes y después, e incluso puede abortar

Ejemplo

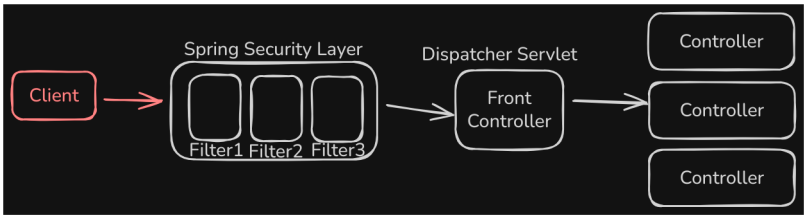
Ejemplo de una clase que imprime un log cada vez que un metodo de service es ejecutado:

```
package com.example.springbootproject5.aop;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
@Aspect
public class LoggingAspect {
    private static final Logger LOGGER =
LoggerFactory.getLogger(LoggingAspect.class);

    // return type, className.method(args)
        @Before("execution(*
com.example.springbootproject5.service.JobService.*(..))")
    public void logService() {
        LOGGER.info("Service method called");
    }
}
```

Spring Security

Es un modulo de spring que nos sirve para autenticar y autorizar usuarios.

Al agregar la dependencia de spring security, automaticamente se coloca una capa que recibe todas las solicitudes. Esta capa de spring security esta compuesta por filtros.



Configuracion por defecto

Usuario y contraeña

Por defecto, spring security bloquea todos los endpoints y los proteje bajo un solo usuario y contraseña:

- usuario: `user`
- contraseña: impresa en consola

Esto se puede configurar desde `application.properties

```
spring.security.user.name=user
spring.security.user.password=123
```

CSRF token (Cross-Site Request Forgery)

Este token se exige por defecto para todas las solicitudes HTTP que **modifican estado** (como `POST`, `PUT`, `DELETE`), excepto para métodos seguros como `GET`. Si no enviamos este token, el servidor responde con un error **403 Forbidden**, incluso si ya estamos autenticados.

Spring Security suele **inyectar el token CSRF en los formularios HTML que genera el servidor** (por ejemplo con Thymeleaf o JSP), para que al enviar una solicitud que modifica datos, se incluya automáticamente y el servidor pueda validarlo.

Para obtener este token en el navegador es necesario iniciar sesion por una unica vez.

El objetivo es proteger la sesión del usuario contra ataques CSRF. Si un atacante intenta usar tu `sessionId` para enviar una solicitud maliciosa desde otro sitio, no podrá hacerlo con éxito porque **no tiene acceso al token CSRF**, que solo está disponible en el entorno de tu navegador legítimo y se envía explícitamente en la solicitud.

Configuracion de seguridad

Ejemplo explicado de una clase de configuracion de seguridad

> Configuracion spring security

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {

        http
                .csrf(customizer -> customizer.disable())
                .httpBasic(Customizer.withDefaults())
                .authorizeHttpRequests(request -> request
                        .requestMatchers("/registrar").permitAll()
                        .anyRequest().authenticated())
```

```
                .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));

        return http.build();
    }

    @Autowired
    private MyUserDetailsService myUserDetailsService;

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider provider = new
DaoAuthenticationProvider();
        provider.setUserDetailsService(myUserDetailsService);
        provider.setPasswordEncoder(new BCryptPasswordEncoder(10));
        return provider;
    }

}
```

Agregar usuarios en memoria (UserDetailsService)

Aparte de los usuarios y contraseña configurados en application properties, se pueden agregar mas, de manera manual dentro de la clase de Configuracion de seguridad, agregando un nuevo bean dentro de la clase SecurityConfig:

```
@Bean
public UserDetailsService userDetailsService() {

    UserDetails user = User
            .withDefaultPasswordEncoder()
            .username("navin")
            .password("n@123")
            .roles("USER")
            .build();

    UserDetails admin = User
            .withDefaultPasswordEncoder()
            .username("admin")
            .password("admin@789")
            .roles("ADMIN")
            .build();

    return new InMemoryUserDetailsManager(user, admin);
}
```

Proveedor de autenticaciones (AuthProvider)

Cuando nos logeamos con spring security, el Authentication Provider recibe un objeto llamado Authentication Object que contiene las credenciales del usuario, y se encarga de corroborar que las credenciales sean correctas y autenticar al usuario. Por defecto, el auth. provider, busca las credenciales por defecto de spring security (las que se encuentran en memoria o en la configuracion por defecto) para comparar, pero se puede modificar para que busque las credenciales en una base de datos.

Bean de configuracion (en SecurityConfig):

```
@Autowired
private MyUserDetailsService myUserDetailsService;

@Bean
public AuthenticationProvider authenticationProvider() {
    DaoAuthenticationProvider provider = new DaoAuthenticationProvider();
    provider.setUserDetailsService(myUserDetailsService);
    provider.setPasswordEncoder(NoOpPasswordEncoder.getInstance());
    return provider;
}
```

Pero debemos implementar UserDetailsService!
Dentro de la implementacion, generalmente accedemos a la base de datos mediante el repositorio, para buscar el usuario y retornar un objeto UserDetails, el cual tambien requiere de su propia implementacion

UserDetailsService para AuthProvider

Se encarga de acceder al repositorio, armar un objeto de UserDetails con los datos requeridos por este, y devolverlo al authentication provider.

- Notar que deberemos implementar UserDetails ya que es una interfaz.
- Utilize pseudocodigo para que sea mas entendible el proceso

```
@Service
public class MyUserDetailsService implements UserDetailsService {

    // Repositorio donde tenemos los usuarios y contrasenias
    @Autowired
```

```
    Repositorio repositorio;

    // Auth provider llama a MyUserDetailsService
    @Override
    public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {

        // Buscamos el usuario que coincida en la DB
        User MiUsuario = repositorio.buscarusuario(username);

        // Devolvermos el objeto requerido con los datos de nuestro
usuario (deberemos implementar esta clase antes)
        return new UserDetailsImplementation(MiUsuario);
    }

}
```

UserDetails

Implementamos UserDetails para poder devolver este objeto habiendolo generado con los datos obtenidos de nuestro usuario de la base de datos. En mi caso, hice que directamente obtenga mi entidad User, y de ahi extraiga los datos necesarios. User es una entidad personalizada.

En esta implementacion, se deja el rol como USER por defecto para todos los usuarios, pero el rol deberia ser cargado desde la DB. Tambien se pueden implementar otras caracteristicas como `isAccountNonLocked`, `isCredentialsNonExpired`, `isAccountNonExpired`, `isEnabled`, que por defecto no requieren de implementacion.

```
public class UserDetailsImplementation implements UserDetails {

    private User user;

    public UserDetailsImplementation(User user) {
        this.user = user;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return Collections.singleton(new SimpleGrantedAuthority("USER"));
    }

    @Override
    public String getPassword() {
        return user.getPassword();
    }

    @Override
```

```
    public String getUsername() {
        return user.getUsername();
    }

}
```

User

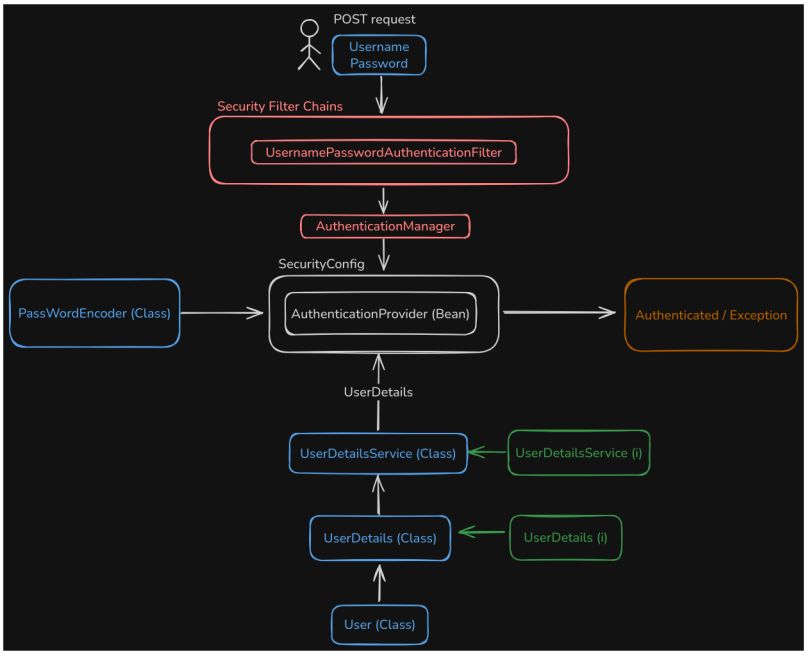
```
@Component
@AllArgsConstructor
@NoArgsConstructor
@Data
public class User {
    private String username;
    private String password;
}
```

Encoder

Usamos el BCryptPasswordEncoder para hashear y guardar las contraseñas en la base de datos. Hay que usar el mismo encoder en el AuthenticationProvider para que pueda comparar los hashes.

```
// Al momento de guardar en nuevo usuario registrado, su contrasenia se
almacena hasheada con Bcrypt.
// Crea un codificador con fuerza=10
private BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(10);
encoder.encode("password");
```

Flujo de registro y login



- **Usuario envía** `POST /login` con sus credenciales.
- **Spring Security intercepta la petición** con `UsernamePasswordAuthenticationFilter`.
- Se crea un `UsernamePasswordAuthenticationToken` con los datos del request.
- El `AuthenticationManager` delega en el `DaoAuthenticationProvider`.
- `DaoAuthenticationProvider` llama a `UserDetailsService` para buscar al usuario:
- `UserDetailsService` busca en la DB y devuelve al Auth Provider un objeto `UserDetails` que contiene el nombre de usuario y la contraseña hasheada.
- El `PasswordEncoder` del Auth Provider compara ambos hashes:
- Si coinciden los hashes:
  - Se autentica al usuario.
  - Se genera un `SecurityContext` con los roles y datos.
- Si no coinciden:
  - Se lanza `BadCredentialsException`.

JWT Token (Json Web Token)

Es un estandar para transmitir informacion de manera segura. Se suele utilizar para la autenticacion del cliente hacia el servidor en cada solicitud, y que este no deba iniciar sesion nuevamente hasta que el token expire.

Caracteristicas:

- Firmado digitalmente
- Es un estandar abierto
- Usa una firma privada a partir de un secreto
- Contiene un objeto Json

Ventajas:

- El servidor no debe guardar ningun dato extra
- Es transportable para el cliente, como una api key

Cuando el cliente inicia sesion, se devuelve este token, que contiene la siguiente estructura de datos:

`JwtService.java`

```
return Jwts.builder()
        .claims(claims) // Declaraciones (rol, email, etc)
        .subject(username) // Nombre de usuario
        .issuedAt(new Date(System.currentTimeMillis()))
        .expiration(new Date(System.currentTimeMillis() + 1000 * 60 * 4))
```

```
            .signWith(getKey(), SignatureAlgorithm.HS256)
            .compact(); // Construye el token y lo devuelve en String
```

Cuando el cliente quiere acceder a un endpoint protegido, debe enviarnos el token incluido en el header `Authorization`.

Ejemplo:

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I
kpvaG4gRG9lIiwiYWRtaW4iOnRydWUsImlhdCI6MTUxNjIzOTAyMn0.KMUFsIDTnFmyG3nMiGM
6H9FNFUROf3wh7SmqJp-QV30
```

El servidor se encarga de validar la firma del token (revisar que él haya emitido el token), y extraer el nombre de usuario. Luego se decide si está o no autorizado para acceder al endpoint.

Dependencias

```
jjwt-api (pom.xml)
jjwt-impl (pom.xml)
jjwt-jackson (pom.xml)
```

Implementacion en springboot

Login

- Añadir jwt service para generar el token
- Generar y devolver el token en el login si las credenciales son correctas (para validar las credenciales usamos `authenticationManager`)

Solicitud autenticada

- Añadir un filtro de tipo `OncePerRequestFilter` en SecurityFilterChain (addFilterBefore) `UsernamePasswordAuthenticationFilter.class` para validar el token antes de que spring pueda autenticar el usuario. Este filtro va a extraer el token del header, el nombre de usuario del token, y lo va a validar con ayuda de jwt service (valida que la firma haya sido del servidor, y que el usuario exista en la db). Si el token esta validado, el filtro genera un `UsernamePasswordAuthenticationToken` con los datos del usuario y lo autentica en `SecurityContextHolder`.

Spring AI

Es una capa de abstraccion que nos permite interactuar con distintas APIs de inteligencia artificial (OpenAI, Ollama, Azure OpenAI, HuggingFace, etc) sin t ener que modificar nuestro

codigo.

- Se instala la dependencia en `pom.xml` (ya sea para OpeinAi, azure, etc)
  Ejemplo: `spring-ai-starter-model-openai`
- Se ingresa la API key en `application.properties` (spring.ai.openai.api-key)
- Ya se puede interactuar con la API.

`Ejemplo`

```
import org.springframework.ai.openai.OpenAiChatModel;
@RestController
public class MainController {

    @Autowired
    private OpenAiChatModel chatModel;

    @GetMapping("/askGPT")
    public String askGPT(@RequestParam String prompt) {
        String response = chatModel.call(prompt);
        return response;
    }
}
```

Podemos cambiar de modelo sin modificar gran parte del codigo. En este caso, me quedé sin tokens de OpenAI, por lo que voy a correr un modelo local usando Ollama. Solo hay que cambiar la dependencia de OpeinAI por la de Ollama, y la inyeccion de dependencia.

Ollama

```
ollama pull modelo # Descargar modelo

ollama list # Ver modelos instalados

# Abrir ollama a la red local
ollama serve

# Por defecto abre en el puerto 11434 (spring AI se conecta
automaticamente)
curl [http://localhost:11434/api/tags](http://localhost:11434/api/tags)
```

```
<dependency>
    <groupId>[org.springframework.ai](http://org.springframework.ai)
</groupId>
    <artifactId>spring-ai-starter-model-ollama</artifactId>
    <version>1.0.1</version>
</dependency>
```

```
spring.ai.ollama.chat.options.model=qwen3:4b
```

```
package com.example.spring_ai2.controller;

import org.springframework.ai.chat.client.ChatClient;
import org.springframework.ai.chat.model.ChatResponse;
import org.springframework.ai.ollama.OllamaChatModel;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class OllamaController {

    private ChatClient chatClient;

    public OllamaController(OllamaChatModel chatModel){
        this.chatClient = ChatClient.create(chatModel);
    }

    @GetMapping("/ask")
    public String ask(@RequestParam String prompt) {
        ChatResponse chatResponse =
chatClient.prompt(prompt).call().chatResponse();
        System.out.println(chatResponse.getMetadata()); // Tokens, uso,
rate limit...
        String response = chatResponse.getResult().getOutput().getText();
// Pensamiento + respuesta final
        System.out.println(response);
        return response;
    }
}
```

Embeddings

Un embedding en inteligencia artificial es una representación matemática de datos (palabras, imágenes, sonidos, usuarios, etc.) en forma de un vector de números reales en un espacio de alta dimensión.

La idea principal es que estos vectores capturen las características o significados relevantes de los datos, de manera que objetos "similares" en el mundo real estén cercanos entre sí en el espacio vectorial.

```
@Autowired
EmbeddingModel embeddingModel;

@PostMapping("/embedding/{text}")
```

```
public float[] embedding(@PathVariable String text) {
    return embeddingModel.embed(text);
}
```

Devuelve algo como:

```
[
  -0.012133638,
  0.015329252,
  0.016223365,
  -0.011653082,
  -0.04637007,
  ...
]
```

Similaridad coseno

La similaridad coseno es una métrica muy usada en Inteligencia Artificial (IA), machine learning y procesamiento de lenguaje natural (PLN) para medir el grado de similitud entre dos vectores en un espacio n-dimensional.

Se define como el **coseno del ángulo** entre dos vectores A y B

$$sim(A, B) = cos(\theta) = \frac{A * B}{||A|| * ||B||}$$

Casos de uso

Podemos mejorar los mecanismos de busqueda (con ayuda de una base de datos de vectores). Por ejemplo, al buscar "headphones" en una base de datos relacional, podriamos tener 0 resultados para "airpods bluetooth", pero usando similaridad, podemos devolver resultados similares, incluyendo dichos airpods.

```
    public double similarity(String text1, String text2) {
        float[] embedding1 = embeddingModel.embed(text1);
        float[] embedding2 = embeddingModel.embed(text2);

        double dotProduct = 0;
        double norm1 = 0;
        double norm2 = 0;

        for (int i = 0; i < embedding1.length; i++) {
            dotProduct += embedding1[i] * embedding2[i];
            norm1 += Math.pow(embedding1[i], 2);
            norm2 += Math.pow(embedding2[i], 2);
        }
```

```
        return dotProduct * 100 / (Math.sqrt(norm1) * Math.sqrt(norm2));
    }
```

SpringBoot

Al crear un proyecto de spring, es muy dificil llegar al primer "hello world" ya que requiere de muchas configuraciones. SpringBoot, no es mas que Spring, pero preconfigurado.

Caracteristicas que nos provee:

- Ahorra configuración manual.
- Viene con "convenciones sobre configuración".
- Incluye un servidor embebido (como Tomcat o Jetty).
- Permite que la app sea auto-ejecutable (como un .jar que se ejecuta con java -jar).

Configuracion (application.properties)

Propiedad	Descripción
`spring.datasource.url`	Define la URL de conexión con la datos.
`spring.datasource.username`	Define el nombre de usuario para conectarse a la base de datos.
`spring.datasource.password`	Define la contraseña del usuario de datos.
`spring.datasource.driver-class-name`	Especifica el driver JDBC a utiliza conectarse a la base de datos.
`spring.jpa.database-platform`	Define el dialecto de JPA (por eje `org.hibernate.dialect.MySQL8`
`spring.jpa.show-sql`	Muestra las sentencias SQL gene JPA en consola (`true` o `false`)
`spring.jpa.hibernate.ddl-auto`	Controla la estrategia de creación/actualización de esquen (`none` , `update` , `create` , `creat` `validate`).
`spring.jpa.properties.hibernate.format_sql`	Formatea las sentencias SQL mo
`spring.jpa.open-in-view`	Permite mantener la sesión abier la vista (recomendado en `false` apps REST).
`spring.jpa.defer-datasource-initialization`	Permite que el esquema se inicia de ejecutar scripts SQL (útil con `data.sql`).

Propiedad	Descripción
`spring.sql.init.mode`	Determina cuándo ejecutar script (`always`, `embedded`, `never`).
`spring.jpa.properties.hibernate.jdbc.time_zone`	Define la zona horaria usada por (`UTC`, `America/Argentina/Buenos_Ai`
`spring.datasource.initialization-mode`	Controla si los scripts SQL se eje automáticamente al iniciar (`alwa` `never`).

Arquitectura por capas

Es una arquitectura de desarrollo web y patron de dise;o para aumentar la cohesion de las clases, y reducir su acomplamiento.

(FrontEnd) Cliente → (Backend) Controller → Service → Repository

Cliente: No es parte del backend, sino, codigo que se ejecua del lado del cliente. Es el que se comunica con el servidor y consume la API, puede ser un navegador web, una aplicación movil, una instancia de postman, etc.

Controller: Ya en el backend, es el que recibe las solicitudes del cliente y las pasa al service. Luego, envia los datos de respuesta nuevamente al cliente.

Service: Contiene mayor parte de la logica de negocio. Es el que ejecuta la logica necesaria (incluso para verificacion y vaidacion) y hace las consultas al repositorio para interactuar con la base de datos. En springboot, es conveniente declararlos con la anotacion **@Service**

Repository: Es generalmente una interfaz, con metodos que nos permiten interactuar con la base de datos directamente. Spring genera automaticamente su implementacion, por lo que solo nos preocupamos por declarar los metodos.

Flujo completo:

- El cliente hace una petición HTTP
- El controller la recibe y delega al servicio.
- El servicio ejecuta lógica de negocio y accede a los datos a través del repositorio.
- El repositorio consulta la base de datos y devuelve la entidad al servicio
- El servicio transforma el resultado si es necesario (a un DTO, por ejemplo).
- El controller devuelve la respuesta al cliente.

Controladores

Anotaciones del controlador

Anotación	Uso
`@RestController`	Marca la clase como controlador REST (devuelve JSON)
`@Controller`	Marca como controlador MVC (devuelve vistas HTML)
`@RequestMapping`	Mapea rutas HTTP (general)
`@GetMapping`	Mapea GET `/ruta`
`@PostMapping`	Mapea POST `/ruta`
`@PutMapping`	Mapea PUT `/ruta`
`@DeleteMapping`	Mapea DELETE `/ruta`
`@PatchMapping`	Mapea PATCH `/ruta`

Ejemplo de un controlador:

```
package com.example.springbootproject3;

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

@Controller
public class HomeController {

    @RequestMapping("/")
    public String home() {
        System.out.println("home called");
        return "index";
    }
}
```

Request con parametros:
http://localhost:8080/sum?a=1&b=2

```
@RequestMapping("/sum")
public String sum(@RequestParam String a, @RequestParam String b) {
    Integer result = Integer.parseInt(a) + Integer.parseInt(b);
    String returnString = a + " + " + b + " is: " + result.toString();
    System.out.println(returnString);
    return returnString;
}
```

Mapeo de parametros del controlador

Hay distintas formas de mapear parametros al controlador (Ejemplos con @RestController):

- PathVariable

```
@GetMapping("/jobs/{id}")
public Job getJob(@PathVariable long id) {
    return jobService.getJob(id);
}
```

- http://localhost:8080/jobs/1
- RequestParam

```
@GetMapping("/jobs")
public Job getJob(@RequestParam long id) {
    return jobService.getJob(id);
}
```

- http://localhost:8080/jobs?id=1
- RequestBody

```
@PostMapping("/register")
public AppUser register(@RequestBody AppUser user) {
    return userService.save(user);
}
```

- http://localhost:8080/register (AppUser se envía como json)

Utilizacion

Se suele utilizar pathvariable en modelos REST, pero si utilizamos filtros, conviene mas usar requestparam.

Parametros de solicitud del controlador

Podemos ingresar parametros en las funciones de los controladores, los cuales spring detectara automaticamente e inyectara estas variables:

Objeto	Descripción
`HttpServletRequest`	Representa la solicitud HTTP (Servlet API).
`HttpServletResponse`	Representa la respuesta HTTP (Servlet API).
`HttpSession`	Sesión del usuario en el servidor.
`Principal`	Usuario autenticado.
`Locale`	Localización (idioma preferido del cliente).
`InputStream` / `Reader`	Leer directamente el cuerpo de la solicitud.

Objeto	Descripción
`OutputStream` / `Writer`	Escribir directamente la respuesta.

Ejemplo:

```
//Este codigo devuelve el sessionId del cliente.
@GetMapping("/")
public String greet(HttpServletRequest request) {
    return "Hello world! " + request.getSession().getId();
}
```

Apache Tomcat

Apache Tomcat es un servidor web y contenedor de servlets de código abierto, desarrollado por la Apache Software Foundation. Su propósito principal es ejecutar aplicaciones web **Java EE** (actualmente Jakarta EE), soportando tecnologías como:

- **Servlets**
- **JSP (JavaServer Pages)**
- **WebSocket**

Tomcat se encarga de recibir peticiones HTTP, mapearlas a los servlets correspondientes y retornar las respuestas generadas por estos componentes Java.

Un **servlet** es una clase Java que extiende la funcionalidad de los servidores web. Esta clase debe estar compilada previamente (antes de desplegarla). Tomcat se encarga de cargar la clase servlet, crear instancias y ejecutar sus métodos para procesar las solicitudes y generar respuestas dinámicas.

Se encuentra como una dependencia en maven:
https://mvnrepository.com/artifact/org.apache.tomcat.embed/tomcat-embed-core

Datos utiles

- Viene embebido en springboot por defecto
- Utiliza el puerto 8080 por defecto
- Springboot provee una capa de abstraccion de apache tomcat, por lo que no deberemos hacer las configuraciones manuales

Tipos de API

Rest (Representational State Transfer)

REST (Representational State Transfer) es un **estilo de arquitectura** para el diseño de servicios web. Fue propuesto por Roy Fielding en su tesis doctoral en el año 2000.

REST define un conjunto de **principios y restricciones** para cómo deben interactuar los sistemas distribuidos, especialmente en la web. Utiliza el protocolo **HTTP** y se basa en operaciones estándar como:

- `GET` : Obtener recursos
- `POST` : Crear recursos
- `PUT` : Actualizar recurso (completamente)
- `PATCH` : Actualiar recurso (parcialmente)
- `DELETE` : Eliminar recursos

Los **recursos** (como usuarios, productos, etc.) se identifican mediante **URLs**, y su representación se transfiere usualmente en formato JSON o XML.

Los **verbos de la acción** (como obtener, crear, eliminar) ya están definidos por el **método HTTP** (`GET` , `POST` , `PUT` , `DELETE` , etc.). Por eso **no se deben repetir en la URL**.

Ejemplo:

```
GET    /usuarios        → obtener lista de usuarios
GET    /usuarios/5      → obtener el usuario con ID 5
POST   /usuarios        → crear un nuevo usuario
PUT    /usuarios/5      → actualizar el usuario 5
DELETE /usuarios/5      → eliminar el usuario 5
```

¿ Por que se dice que se transfiere el estado ?
Porque en cada interaccion utilizando la arquitectura rest, se transfiere el estado de un objeto o endad para cada una de las operaciones (CRUD).

RESTful Naming Conventions (in English)

Nombres estandar de los metodos del codigo para programacion web api rest.

HTTP Method	Purpose	Method name
`GET`	Get all resources	`getAllEmployees()` or `listEmployees()`
`GET`	Get one by ID	`getEmployeeById(Long id)`
`POST`	Create new	`createEmployee(EmployeeDTO dto)`
`PUT`	Full update	`updateEmployee(Long id, EmployeeDTO dto)`
`PATCH`	Partial update	`patchEmployee(Long id, EmployeeDTO dto)`
`DELETE`	Delete by ID	`deleteEmployee(Long id)`

RPC (Remote Procedure Call)

RPC significa **Llamada a Procedimiento Remoto**. Es un estilo de comunicación donde un cliente **invoca funciones (procedimientos)** que están en otro sistema (generalmente un servidor), **como si fueran funciones locales**.

ejemplo:
`POST /crearUsuario`

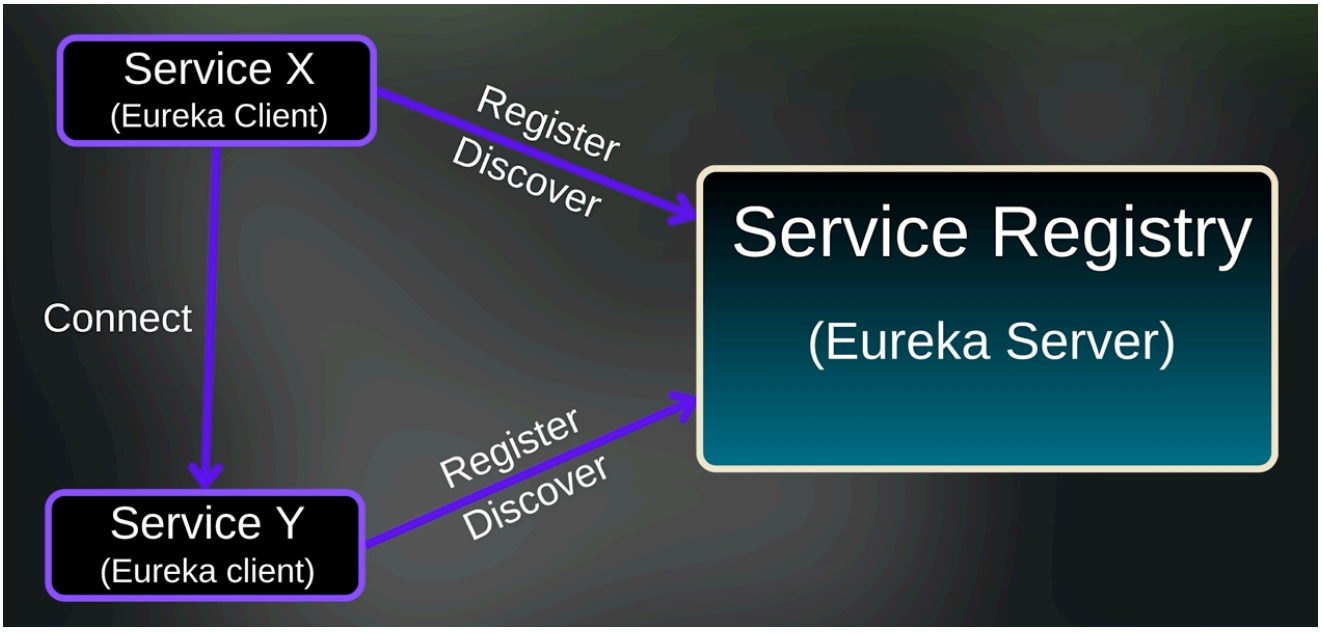
SOAP (Simple Object Access Protocol)

SOAP es un **protocolo de mensajería** basado en **XML**, que se utiliza para hacer **llamadas RPC sobre HTTP o SMTP**, entre otros. Fue muy usado en servicios web empresariales antes de REST.

Microservicios

Consiste en separar una aplicación monolítica en servicios que se ejecutan independientemente e interactúan entre sí.
Funciona como una aplicacion springboot normal, pero son varios proyectos, en donde cada uno representa un servicio, como userService, etc. Esto genera desacoplamiento y nos da la posibilidad de escalar los servicios mas requeridos por el usuario, optimizando los recursos. También funciona como seguro ante fallos, ya que si un servicio cae, podemos buscar un reemplazo.

Para conectar y descubrir servicios, usamos `Eureka Server`, en donde cada microservicio se conecta a un servidor eureka para, dado un nombre, ver cual es la ip del microservicio con el cual se quiere comunicar.



Eureka es el servidor que se encarga de registrar y proveer las IP de los microservicios.
Un microservicio se conecta al servidor eureka para ver los microservicios que hay disponibles, y mostrarse a estos.

Eureka (servidor)

Setup del servidor eureka

No es más que una app springboot:

`pom.xml`

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

`main.java`

```
@SpringBootApplication
@EnableEurekaServer
// http://localhost:8761/
public class EurekaserverApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaserverApplication.class, args);
    }

}
```

`application.properties`

```
spring.application.name=eurekaserver
server.port=8761

eureka.instance.hostname=localhost
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

Eureka (microservicio 1)

Setup del primer microservicio

- **Cliente eureka**: ver y ser descubierto por microservicios
- **Feign**: comunicarse con microservicios

Nota: podemos usar feign sin el cliente eureka, pero deberíamos especificar manualmente la IP del microservicio a conectarse, lo que dificultaría mucho el trabajo

`pom.xml` (microservicio1)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

En este caso la conexión a eureka se realizó de manera automática por estar en la misma red.
Está todo listo, solo falta conectarse al microservicio y llamar a su endpoint. Para ello, debemos declarar una interfaz para Feing, en donde se especifiquen los endpoints del microservicio objetivo, en este caso, se llama `MICROSERVICE2`, nombre el cual se encuentra registrado en Eureka.

`FeingInterface.java` (microservicio1)

```
// El nombre de los microservicios se pueden obtener en la interfaz web de
eurekaServer
@FeignClient("MICROSERVICE2")
public interface FeingInterface {

    @GetMapping("/micro2")
    public String micro2(); // Mapeo perteneciente al controlador del
microservicio 2
}
```

Debemos decirle a spring que tenemos el bean `feing` para que lo escanee
`main.java` (microservicio1)

```
@SpringBootApplication
@EnableFeignClients(basePackages = "com.example.microservice1.feing")
public class Microservice1Application {
    public static void main(String[] args) {
        SpringApplication.run(Microservice1Application.class, args);
    }
}
```

Feing interactúa con el cliente de eureka para obtener la IP del servidor, y nos va a abstraer de realizar las solicitudes manualmente. Ya podemos interactuar con el microservicio objetivo (`MICROSERVICE2`) haciendo uso de la interfaz.

`controller.java` (microservicio1)

```
@Autowired
FeingInterface feingInterface;

@GetMapping("/micro2")
public String micro2() {
    return feingInterface.micro2();
}
```

Eureka (microservicio 2)

Setup del primer microservicio

No necesitamos Feign si el microservicio 2 no va a llamar a otro microservicio.
`pom.xml` (microservicio2)

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

El resto, funciona como una aplicación normal springboot:

`controller.java` (microservicio2)

```
@RestController
// http://localhost:8082/swagger-ui/index.html
public class Controller {
    @GetMapping("/micro2")
    public String micro2() {
        return "Hello World - Service 2";
    }
}
```

Ya podemos llamar al endpoint `/micro2` desde el microservicio1.

Notas

- No se configuraron los microservicios en redes externas, el codigo fue provado en LAN, y en el mismo host.
- Hay que cambiar el puerto de los microservicios al correrlos en el mismo host:
  `application.properties`

  ```
  server.port=8081
  ```

Notas utiles

Instalar JDK 21 en ubuntu

sudo apt install openjdk-21-jdk
sudo apt install maven

Software recomendado

- IDE intellij idea o vscode
- maven instalado
- java JDK instalado

Extensiones recomendadas para VSCODE

- Extension Pack for Java
- Git graph
- XML Language support by red hat
- Spring Boot Extension Pack - VMware
- Spring Boot Tools - VMware

Configuracion recomendada para VSCODE

Formateo automatico

in vscode:

- ctrl + shift + x
- install "Extension Pack for Java"
- Open vscode settings (Ctrl + ,).
- Search for: Format On Save
- Check option: Editor: Format On Save

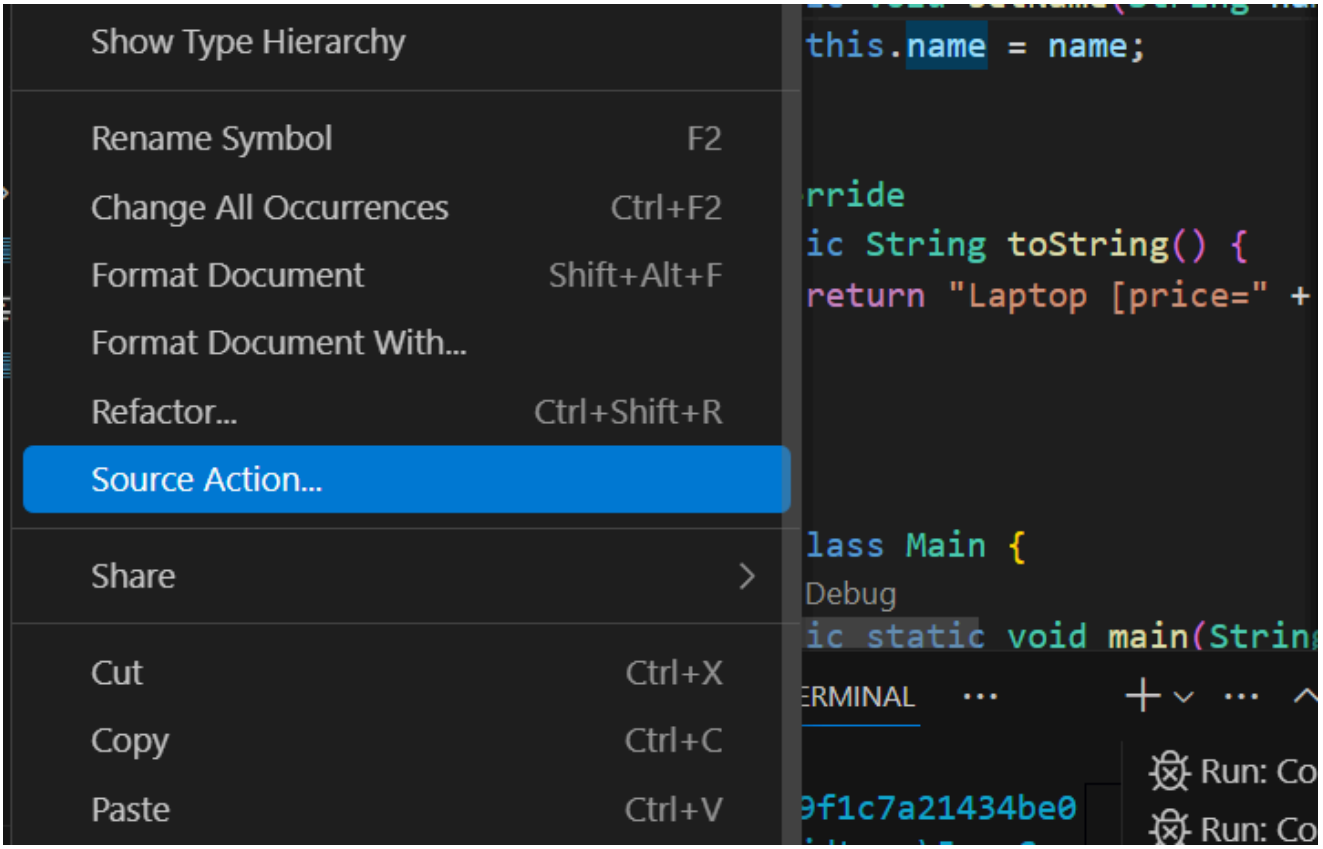
or mannualy:
download: https://github.com/google/google-java-format/releases
java -jar google-java-format-1.27.0-all-deps.jar --replace YourClass.java

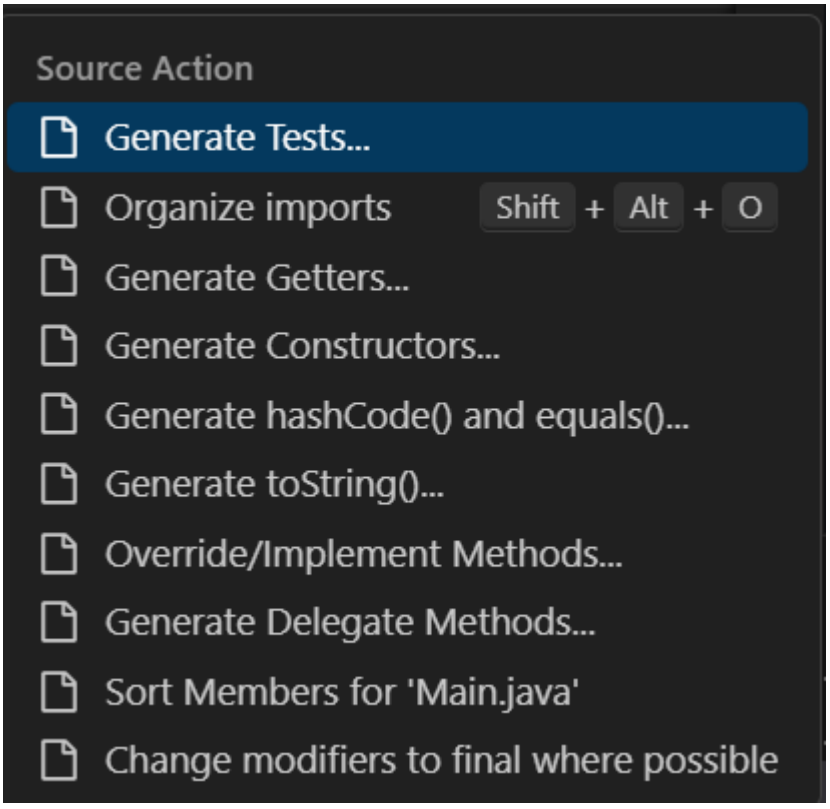
Atajo para comentar lineas

`ctrl + shift + /`
o
`ctrl + /`

Source Action (vscode)

Nos deja realizar acciones repetitivas de manera automatica.





Trucos / Preguntas dificiles

1 Caracteres

```
char c = ‘a’;
char result = (char) c + 1;
```

cuanto vale result ?

result vale ‘b’ porque ‘a’ es 98 en ascii, por lo que al sumarle 1 y castearlo como char, es la letra ‘b’.

2 Condicionales con post increment

What will be the value of variables ‘x’ and ‘y’ from the given code?

```
int x = 5;

int y = 10;

int z = (x++ > 5 && y-- < 10) ? x-- : y;
```

La respuesta es (6,10) porque al evaluar 5 > 5 es false entonces no se ejecuta el resto del condicional, luego a X se le asigna el valor de 6 por ser un post increment, y el valor de Y queda intacto.

2 Condicionales que modifican variables

```
int i, j;

i = 100;

j = 300;

while(++i < --j);
```

Por mas que sea un condicional, las variables SI son modificadas, por el hecho de estar usando “++” y “—“ en cambio de j + 1, en donde no se modificaria la variable.