JAVA
EXAMEN FINAL

- Programación orientada a objetos (POO).
- JUnit
- Colecciones
- Relaciones entre clases
- Herencia
- Polimorfismo
- Clases Abstractas
- Interfaz
- Manejo de excepciones(falta)
- Base de datos(falta)
- JDBC
- JPA
- Spring - MVC
- RESTful
- SpringSecurity
- Spring Security JWT.

POO

Es un paradigma de programación, es decir, un modelo o un estilo de programación que se basa en el concepto de clases y objetos. Este tipo de programación se utiliza para estructurar un programa de software en piezas simples y reutilizables de código (clases) para crear instancias individuales de objetos.
Permite que el código sea reutilizable, organizado y fácil de mantener. Sigue el principio de desarrollo de software utilizado por muchos programadores DRY (Don't Repeat Yourself), para evitar duplicar el código y crear de esta manera programas eficientes. Además, evita el acceso no deseado a los datos o la exposición de código propietario mediante la encapsulación y la abstracción.

Un objeto es una abstracción conceptual del mundo real que se puede traducir a un lenguaje de programación orientado a objetos. Los objetos del mundo real comparten dos características: Todos poseen estado y comportamiento.

El estado de un objeto es una lista de variables conocidas como sus atributos, cuyos valores representan el estado que caracteriza al objeto.
El comportamiento es una lista de métodos, procedimientos, funciones u operaciones que un objeto puede ejecutar a solicitud de otros objetos. Los objetos también se conocen como instancias.

ELEMENTOS DE UNA CLASE

Una clase describe un tipo de objetos con características comunes. Es necesario definir la información que almacena el objeto y su comportamiento.

¿QUÉ SON LOS ATRIBUTOS?

Los atributos son características comunes a todos los objetos. Son los "espacios" donde alojaremos información que cambiará en cada objeto pero que corresponda a una descripción
El estado o información de un objeto se almacena en atributos.

¿QUÉ SON LOS CONSTRUCTORES?

Los constructores con métodos propios del objeto que nos permiten CREARLO. A la creación de un objeto se le denomina INSTANCIACIÓN.
Además de definir los atributos de un objeto, es necesario definir los métodos que determinan su comportamiento. Toda clase debe definir un método especial denominado constructor para instanciar los objetos de la clase. Este método tiene el mismo nombre de la clase.

Una vez que se ha declarado una clase, se pueden crear objetos a partir de ella. A la creación de un objeto se le denomina instanciación. Por esta razón que se dice que **un objeto es una instancia de una clase** y el término instancia y objeto se utilizan indistintamente

ABSTRACCIÓN

La abstracción es la propiedad que considera los aspectos más significativos o notables de un problema y expresa una solución en esos términos. La abstracción posee diversos grados o niveles de abstracción, los cuales ayudan a estructurar la complejidad intrínseca que poseen los sistemas del mundo real. La abstracción encarada desde el punto de vista de la programación orientada a objetos es el mecanismo por el cual se proveen los límites conceptuales de los objetos y se expresan sus características esenciales, dejando de lado sus características no esenciales. Si un objeto tiene más características de las necesarias los mismos resultan difíciles de usar, modificar, construir y comprender. En el análisis hay que concentrarse en ¿Qué hace? y no en ¿Cómo lo hace?

ENCAPSULAMIENTO

La encapsulación o encapsulamiento significa reunir en una cierta estructura a todos los elementos que a un cierto nivel de abstracción se pueden considerar pertenecientes a una misma entidad, y es el proceso de agrupamiento de datos y operaciones relacionadas bajo una misma unidad de programación, lo que permite aumentar la cohesión de los componentes del sistema.
El encapsulamiento oculta lo que hace un objeto de lo que hacen otros objetos y del mundo exterior por lo que se denomina también ocultación de datos. Un objeto tiene que presentar "una cara" al mundo exterior de modo que se puedan iniciar sus operaciones.
Los métodos operan sobre el estado interno de un objeto y sirven como el mecanismo primario de comunicación entre objetos. Ocultar el estado interno y hacer que toda interacción sea através de los métodos del objeto es un mecanismo conocido como encapsulación de datos.
el uso correcto del encapsulamiento vamos a utilizar los modificadores de acceso:
Public: Este modificador permite a acceder a los elementos desde cualquier clase, independientemente de que esta pertenezca o no al paquete en que se encuentra el elemento.
Private: Es el modificador más restrictivo y especifica que los elementos que lo utilizan sólo pueden ser accedidos desde la clase en la que se encuentran.

Protected: Este modificador indica que los elementos sólo pueden ser accedidos desde su mismo paquete y desde cualquier clase que extienda la clase

ATRIBUTOS Y MÉTODOS ESTÁTICOS

Un atributo o un método de una clase se puede modificar con la palabra reservada static para indicar que este atributo o método no pertenece a las instancias de la clase si no a la propia clase.
Se dice que son atributos de clase si se usa la palabra clave static: en ese caso la variable es única para todas las instancias (objetos) de la clase (**ocupa un único lugar en memoria**), es decir que, **si se poseen múltiples instancias de una clase, cada una de ellas no tendrán una copia propia de este atributo, si no que todas estas instancias compartirán una misma copia del atributo.**

En el caso de una constante no tiene sentido crear un nuevo lugar de memoria por cada objeto de una clase que se cree. Por ello es adecuado el uso de la palabra clave static. Cuando usamos "static final" se dice que creamos una constante de clase, un atributo común a todos los objetos de esa clase.

En este contexto indica que una variable es de tipo constante: no admitirá cambios después de su declaración y asignación de valor. La palabra reservada **final** determina que un atributo no puede ser sobrescrito o redefinido, es decir, no funcionará como una variable "tradicional", sino como una constante. Toda constante declarada con final ha de ser inicializada en el mismo momento de declararla. El modificador final también se usa como palabra clave en otro contexto: una clase final es aquella que no puede tener clases que la hereden.

CLASE SERVICIO

Es una clase común y corriente pero que se va a encargar de crear los objetos y va a tener todos los métodos necesarios para la utilización de ese objeto.

JUnit

Una prueba unitaria es una técnica de prueba que se utiliza en el desarrollo de software para verificar el funcionamiento individual y aislado de una unidad de código, generalmente un método o una función. El objetivo principal de una prueba unitaria es garantizar que cada unidad funcione correctamente de manera independiente antes de integrarla con otras partes del sistema.
Las pruebas unitarias son pruebas automatizadas que se centran en verificar el correcto funcionamiento de las unidades de código de forma individual. Proporcionan beneficios como la detección temprana de errores, facilitar el mantenimiento del código, actuar como documentación viva, mejorar la colaboración entre desarrolladores y elevar la calidad del software en general.

● @Test: Esta anotación se utiliza para marcar un método como una prueba unitaria. Los métodos anotados con @Test deben tener una firma pública y no devolver un valor.

Una vez que se ha llamado al método bajo prueba y se ha obtenido un resultado, puedes utilizar las aserciones (assertions) de JUnit para verificar si el resultado es el esperado. Las aserciones comparan el resultado obtenido con el resultado esperado y muestran un mensaje de error si no coinciden.

● @Before: Esta anotación se utiliza para marcar un método que se ejecutará antes de cada prueba. Puedes utilizar este método para realizar la configuración necesaria antes de cada prueba.
Por ejemplo, puedes crear instancias de objetos,establecer conexiones con bases de datos, cargar datos de prueba, inicializar variables státicas,entre otros. (Ahorro de tiempo y recursos: Al ejecutar una configuración previa una vez antes de todas las pruebas, se evita la repetición de la misma configuración en cada prueba individual.)

● @BeforeClass: Esta anotación se utiliza para marcar un método que se ejecutará una vez antes de todas las pruebas en la clase. Puedes utilizar este método para realizar configuraciones que sean comunes a todas las pruebas.
Si hay algún estado o datos que deben ser compartidos entre varias pruebas, la configuración previa con @BeforeClass puede ser útil. Puedes preparar el estado compartido en el método @BeforeClass y acceder a él desde diferentes pruebas en la clase.

Mayor legibilidad y mantenibilidad: Al colocar la configuración común en un método anotado con @BeforeClass, mejora la legibilidad del código de prueba.
La anotación @BeforeClass es útil cuando tienes configuraciones que son comunes a todas las pruebas en la clase y que solo necesitan ejecutarse una vez antes de todas las pruebas. Esto evita la necesidad de repetir la misma configuración en cada método de prueba individual.
Recuerda que el método anotado con @BeforeClass debe ser estático para que se ejecute correctamente antes de las pruebas.

● **@After**: Esta anotación se utiliza para marcar un método que se ejecutará después de cada prueba. Puedes utilizar este método para realizar la limpieza o restauración de estados después de cada prueba.

● **@AfterClass**: Esta anotación se utiliza para marcar un método que se ejecutará una vez después de todas las pruebas en la clase. Puedes utilizar este método para realizar tareas de limpieza o liberación de recursos después de todas las pruebas.
Necesidad: Es útil para liberar recursos, cerrar conexiones a bases de datos, eliminar archivos
temporales, revertir cambios en el estado, entre otras tareas de limpieza.
Importancia: Sin @After, no se realizarían las tareas de limpieza necesarias después de cada prueba. Esto podría llevar a una acumulación de recursos no liberados o a un estado incorrecto que podría afectar a otras pruebas y a la precisión de los resultados.

● **@Ignore**: Esta anotación se utiliza para marcar una prueba para ser ignorada. La prueba no se
ejecutará y se marcará como pasada automáticamente.

● **@RunWith**: se utiliza para personalizar el comportamiento de jecución de las pruebas. Permite especificar un corredor (runner) personalizado para ejecutar las pruebas en lugar del corredor predeterminado de JUnit.

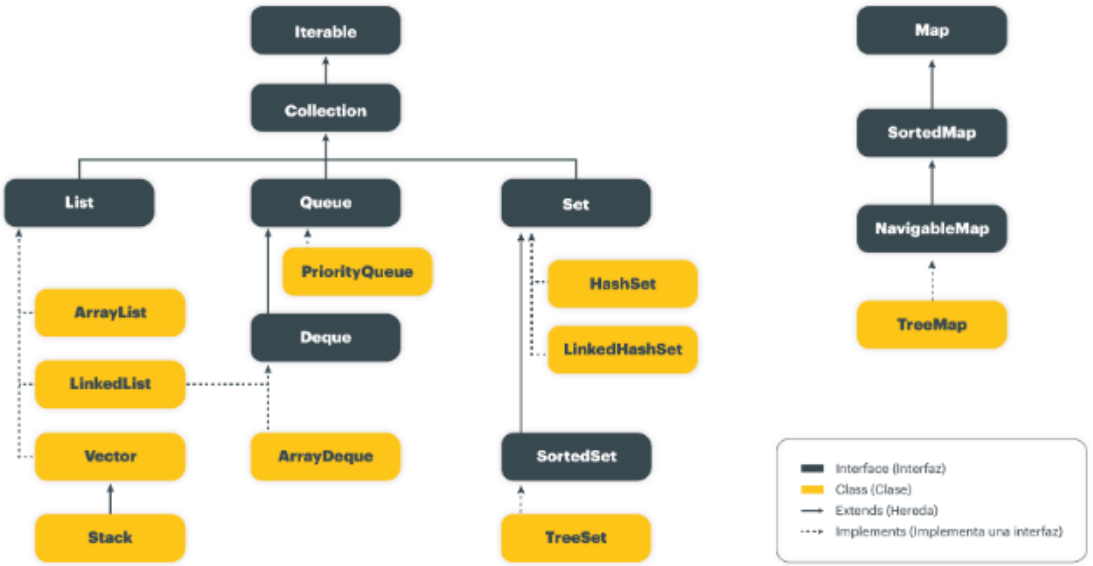
● **@Category**: Permite agrupar pruebas en categorías para poder ejecutar subconjuntos de pruebas en función de esas categorías.
Uso: Define una interfaz o una clase de categoría y anota las pruebas con @Category para asignarlas a una o varias categorías.

Colecciones

Una colección es un grupo de objetos, pero para obtener una colección vamos a utilizar unas clases propias de Java. Estas clases, que van a ser el almacén de los objetos, nos proveen con una serie de herramientas (métodos) comunes, para trabajar con los elementos de la colección
La principal diferencia entre las colecciones y los arreglos tradicionales es que las colecciones crecen de manera dinámica a medida que se van agregando objetos. No necesitamos saber de antemano la cantidad de elementos con la que vamos a trabajar.

Usaremos el Java Collections Framework dentro del paquete java.util. El Collections Framework es una arquitectura compuesta de interfaces y clases.

¿QUÉ ES UN FRAMEWORK?
Un framework es un marco de trabajo el cual contiene un conjunto estandarizado de conceptos, prácticas, criterios y herramientas para hacer frente a un tipo de problemática particular y resolver nuevos problemas de índole similar.



listas son un tipo de colección que nos permiten tener un control preciso sobre el lugar que ocupa cada elemento. Es decir, sus elementos están ordenados y podemos elegir en qué lugar colocar un elemento mediante su índice(lugar que ocupa). Esto nos da una de las características más importantes de las listas: **pueden guardar elementos duplicados.**

ARRAYLIST

Se implementa como un arreglo de tamaño variable. A medida que se agregan más elementos, su tamaño aumenta dinámicamente. Es el tipo más común. Básicamente es un array o vector de tamaño dinámico, con las características propias de las listas.

LINKEDLIST

Se implementa como una lista de doble enlace. Su rendimiento al agregar y quitar es mejor que Arraylist, pero peor en los métodos set y get. ¿Qué es una lista de doble enlace?
Básicamente una lista de doble enlace es un tipo de lista enlazada que permite moverse hacia delante y hacia atrás.

CONJUNTOS - SET

Los conjuntos / Set modelan una colección de objetos de una misma clase donde cada elemento aparece solo una vez, **no puede tener duplicados**, a diferencia de una lista donde los elementos podían repetirse.
El framework trae varias implementaciones de distintos tipos de conjuntos:

HASHSET

Se implementa utilizando una tabla hash para darle un valor único a cada elemento y de esa manera evitar los duplicados. Es decir, el HashSet crea un código hash para cada valor, evitando que hayan dos valores iguales o duplicados y a diferencia del TreeSet sus elementos no están ordenados.

¿Qué es un Hash?

Una función criptográfica hash- usualmente conocida como "hash"- es un algoritmo matemático que transforma cualquier bloque arbitrario de datos en una nueva serie de caracteres alfanuméricos (mezcla entre letras y números) con una longitud fija. Independientemente de la longitud de los datos de entrada, el valor hash de salida tendrá siempre la misma longitud

TREESET

Se implementa utilizando una estructura de árbol. La gran diferencia entre el HashSet y el TreeSet, es que el TreeSet **mantiene todos sus elementos de manera ordenada** (forma ascendente), pero los métodos de agregar, eliminar son más lentos que el HashSet ya que cada vez que le entra un elemento debe posicionarlo para que quede ordenado. Además de ordenarlos el TreeSet **tampoco admite duplicados.**

LINKEDHASHSET

Está entre HashSet y TreeSet. Se implementa como una tabla hash con una lista vinculada que se ejecuta a través de ella, por lo que proporciona el orden de inserción.

MAPAS

Los mapas modelan un objeto a través de una llave y un valor. Esto significa que cada valor de nuestro mapa, va a tener una llave única para representar dicho valor. Las llaves de nuestro mapa no pueden repetirse, pero los valores sí.
Los mapas al tener dos datos, también vamos a tener que especificar el tipo de dato tanto de la llave y del valor, pueden ser de tipos de datos distintos.

HASHMAP

Es un mapa implementado a través de una tabla hash, las llaves se almacenan utilizando un algoritmo de hash solo para las llaves y evitar que se repitan.

TREEMAP

Es un mapa que ordena los elementos de manera ascendente a través de las llaves.

LINKEDHASHMAP

Es un HashMap que conserva el orden de inserción

Relaciones entre clases.

Las relaciones entre clases realmente significan que una clase contiene una referencia a un objeto u objetos, de la otra clase en la forma de un atributo.

ASOCIACIÓN

En las relaciones de asociación se puede establecer una relación bidireccional, donde los objetos que están al extremo de una relación pueden "conocerse "entre sí, o una relación unidireccional donde solamente uno de ellos "conoce" a otro.
Dentro de la asociación simple existe la composición y la agregación, que son las dos formas de relaciones entre clases.

AGREGACIÓN

Representa un tipo de relación muy particular, en la que una clase es instanciada por otro objeto y clase. La agregación se podría definir como el momento en que dos objetos se unen para trabajar juntos y así, alcanzar una meta. Un punto a tomar muy en cuenta es que **ambos objetos son independientes entre sí.**
En agregación, ambos objetos pueden sobrevivir individualmente, lo que significa que al borrar un objeto no afectará a la otra entidad.

COMPOSICIÓN

La composición es una relación más fuerte que la agregación, es una "relación de vida", es decir, que **el tiempo de vida de un objeto está condicionado por el tiempo de vida del objeto que lo incluye**. La composición es un tipo de relación dependiente en dónde un objeto más complejo es conformado por objetos más pequeños. En esta situación, la frase **"Tiene un"**, debe tener sentido, por ejemplo: el cliente tiene una cuenta bancaria. Esta relación es una composición, debido a que al eliminar el cliente la cuenta bancaria no tiene sentido, y también se debe eliminar, es decir, la cuenta existe sólo mientras exista el cliente.

Herencia

Es un pilar importante de la POO. Es el mecanismo mediante el cual una clase es capaz de heredar todas las características (atributos y métodos) de otra clase.
Las propiedades comunes se definen en la superclase (clase padre/madre) y las subclases heredan estas propiedades (Clase hija/o).

La herencia apoya el concepto de "reutilización", es decir, cuando queremos crear una nueva clase y ya existe una clase que incluye parte del código que queremos, podemos utilizar esa clase que ya tiene el código que queremos y hacer de la nueva clase una subclase. Al hacer esto, estamos reutilizando los campos y métodos de la clase existente.

HERENCIA Y ATRIBUTOS

La subclase (Hija) como hemos dicho recibe todos los atributos de la superclase (Madre), y además la subclase puede tener atributos propios.

En la superclase podemos observar que los atributos están creados con el modificador de acceso protected y no private. Esto es porque el modificador de acceso **protected** permite que las subclases puedan acceder a los atributos sin la necesidad de getters y setters.
Los atributos se trabajan como protected también, porque una subclase no hereda los miembros privados de su clase principal. Sin embargo, si la superclase tiene métodos públicos o protegidos (como getters y setters) para acceder a sus campos privados, estos también pueden ser utilizados por la subclase. Entonces, para evitar esto, usamos atributos protected.

HERENCIA Y CONSTRUCTORES

Una diferencia entre los constructores y los métodos es que los constructores no se heredan, pero los métodos sí. Todos los constructores definidos en una superclase pueden ser usados desde constructores de las subclases a través de la palabra clave super. La palabra clave super es la que me permite elegir qué constructor, entre los que tiene definida la clase padre, es el que debo usar.
Si la superclase tiene definido el constructor vacío y no colocamos una llamada explícita super, se llamará el constructor vacío de la superclase.

HERENCIA Y MÉTODOS

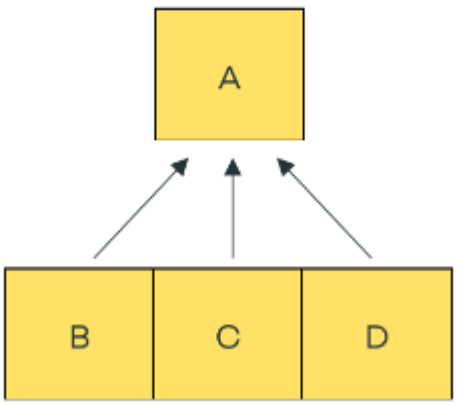
Los métodos heredados pueden ser redefinidos en las clases hijas. Este mecanismo se lo llama **sobreescritura**. La sobreescritura permite a las clases hijas utilizar un método definido en la superclase.

Una subclase sobreescribe un método de su superclase cuando define un método con las mismas características (nombre, número y tipo de argumentos) que el método de la superclase. Las subclases emplean la sobreescritura de métodos la mayoría de las veces para agregar o modificar la funcionalidad del método heredado de la clase padre.
La sobreescritura permite que las clases hijas sumen sus métodos en torno al funcionamiento y Esto se logra poniendo la anotación **@Override** arriba del método que queremos sobreescribir, el método debe llamarse igual en la subclase como en la superclase.

POLIMORFISMO significa "muchas formas".

Este término se utiliza en POO para referirse a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos, es decir, que la misma operación se realiza en las clases de diferente forma. Estas operaciones tienen el mismo significado y comportamiento,pero internamente, cada operación se realiza de diferente forma. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía. **Esto hace referencia a la idea de que podemos tener un método definido en la superclase y que las subclases tengan el mismo método, pero con distintas funcionalidades.**

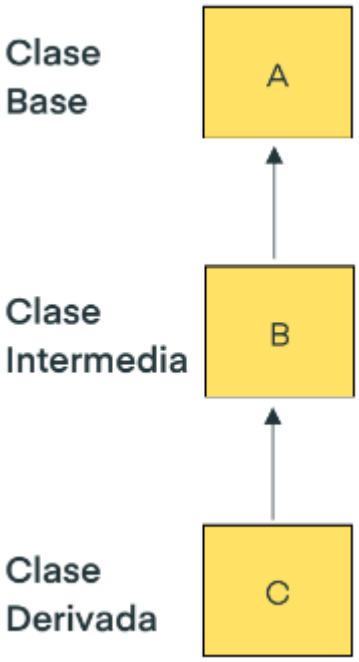
HERENCIA JERÁRQUICA
En la herencia jerárquica, una clase sirve como una superclase (clase base) para más de una
subclase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, C y D.



HERENCIA MULTINIVEL

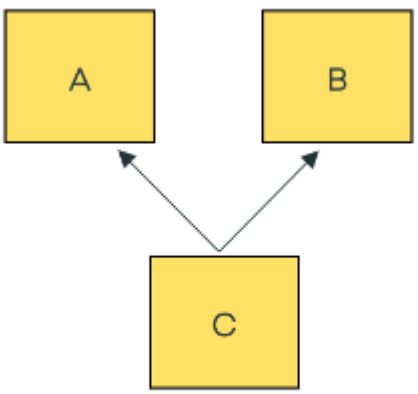
En la herencia multinivel, una clase derivada heredará una clase base y, además, la clase derivada también actuará como la clase base de otra clase. En la imagen inferior, la clase A sirve como clase base para la clase derivada B, que a su vez sirve como clase base para la clase derivada C.

En Java, **una clase no puede acceder directamente a los miembros de los "abuelos"**.



HERENCIA MÚLTIPLE (A TRAVÉS DE INTERFACES)

En Herencia múltiple, una clase puede tener más de una superclase y heredar características de todas las clases principales. Tenga en cuenta que **Java no admite herencia múltipl**e con clases. En Java, podemos lograr herencia múltiple solo a través de Interfaces. En la imagen a continuación, la Clase C se deriva de la interfaz A y B.



CLASES FINALES

El modificador final puede utilizarse también como modificador de clases. Al marcar una clase como final impedimos que se generen hijos a partir de esta clase, es decir, cortamos la jerarquía de herencia.
public final class Animal{ }

MÉTODOS FINALES

El modificador final puede utilizarse también como modificador de métodos. La herencia nos permite reutilizar el código existente y uno de los mecanismos es la crear una subclase y sobrescribir alguno de los métodos de la clase padre. Cuando un método es marcado como final en una clase, evitamos que sus clases hijas puedan sobrescribir estos métodos.
public final void método(){ }

CLASES ABSTRACTAS

En java se dice que una clase es abstracta cuando no se permiten instancias de esa clase, es decir que **no se pueden crear objetos.**
Usualmente las clases abstractas suelen ser las superclases, esto lo hacemos porque creemos que la superclase o clase padre, no debería poder instanciarse.
Otra razón es porque decidimos hacer métodos abstractos en nuestra superclase. **Cuando una clase posee al menos un método abstracto esa clase necesariamente debe ser marcada como abstracta.**

MÉTODOS ABSTRACTOS

Un método abstracto es un método declarado, pero no implementado, es decir, es un método del que solo se escribe su nombre, parámetros, y tipo devuelto, pero no su código de implementación. Estos métodos se heredan y se sobreescriben por las clases hijas quienes son las responsables de implementar sus funcionalidades.

INTERFACES

Una interfaz es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo. Esos métodos deben ser implementados por una clase para que se definan sus acciones.
Por lo tanto, una interfaz especifica qué se debe hacer, pero no cómo hacerlo. Una vez que se define una interfaz, cualquier cantidad de clases puede implementarla. Además, **una clase puede implementar cualquier cantidad de interfaces.**
Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) para los métodos descritos por la interfaz. Cada

clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero cada clase aún admite el mismo conjunto de métodos.

INSTANCIAR UNA INTERFAZ

Aunque las interfaces van a ser implementadas por clases y van a tener métodos, al igual que una clase abstracta, esta, no se va a poder instanciar. La definición de un interfaz no tiene constructor, por lo que no es posible invocar el operador new sobre un tipo interfaz, por lo que no podemos crear objetos del tipo interfaz.

Manejo de excepciones

Una excepción es un evento que ocurre durante la ejecución de un programa que interrumpe el flujo normal de las instrucciones del programa.

JDBC: (JavaTM Database Connectivity)
Con estas interfaces y clases estándar (API JDBC), los programadores pueden escribir aplicaciones que se conecten con bases de datos, envíen consultas escritas en el lenguaje de consulta estructurada (SQL) y procesen los resultados.
Siempre y cuando haya un driver para dicho DBMS (DataBase Management System) en concreto.

Componentes principales de JDBC o JDBC Driver Manager: Es el enlace de comunicaciones de la base de datos que maneja toda la comunicación con la base de datos.
La clase permite obtener objetos Connection con la base de datos.

Para conectarse es necesario proporcionar:
• URL de conexión, que incluye:
• Nombre del host donde está la base de datos.
• Nombre de la base de datos a usar.
• Nombre del usuario en la base de datos.
• Contraseña del usuario en la base de datos

Las API (Aplication Programming Interface) son mecanismos que permiten a dos componentes de software comunicarse entre sí mediante un conjunto de definiciones y protocolos.

API JDBC: Es un conjunto de interfaces y clases, que proporciona varios métodos e interfaces para una fácil comunicación con la base de datos.

Connection: Es una interfaz con todos los métodos para contactar una base de datos. El objeto de conexión representa el contexto de comunicación.

Statement: Encapsula una instrucción SQL que se pasa a la base de datos para ser analizada, compilada, planificada y ejecutada.

ResultSet: Los ResultSet representan un conjunto de filas recuperadas debido a la ejecución de una consulta.

Estos paquetes son:
1. java.sql.*;
2. javax.sql.*;

Las clases e interfaces principales de JDBC son:
• java.sql.DriverManager
• java.sql.Connection
• java.sql.Statement
• java.sql.ResultSet
• java.sql.PreparedStatement
• javax.sql.DataSource

El concepto de **Driver** hace referencia al conjunto de clases necesarias que implementa de forma nativa el protocolo de comunicación con la base de datos.

DAO (Data Access Object): representa una capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos. Esta clase va a ser la encargada de comunicarse con la base de datos, de conectarse con la base de datos, enviar las consultas y recuperar la información de la base de datos.

JPA: (Java Persistence API)
Implementar un Framework Object Relational Mapping (ORM), que permite interactuar con la base de datos por medio de objetos,el encargado de convertir los objetos Java en instrucciones para el Manejador de Base de Datos (DBMS).el diseño de esta API es no perder las ventajas de la orientación a objetos al interactuar con una base de datos (siguiendo el patrón de mapeo objeto-relacional).

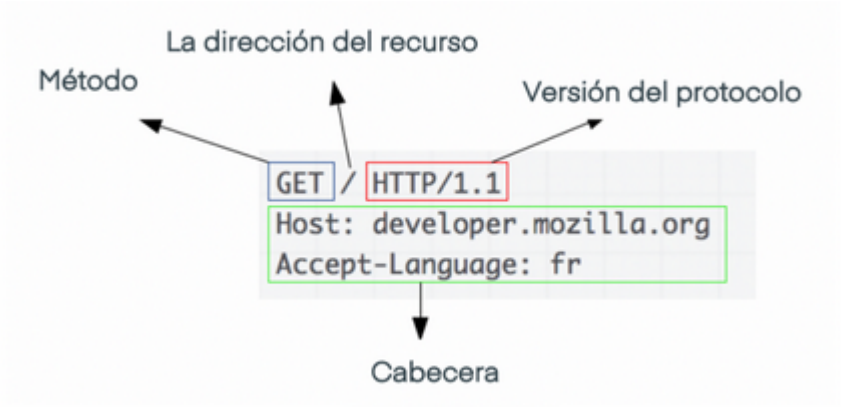
JDBC por mucho tiempo fue la única forma de interactuar con las bases de datos, pero representaba un gran problema y es que Java es un lenguaje orientado a objetos y se tenían que convertir los atributos de las clases en una consulta JPA es una especificación, es decir, no es más que un documento en el cual se plasman las reglas que debe de cumplir cualquier proveedor que desee desarrollar una Implementación de JPA.

WEB:
Se basa en dos factores fundamentales: el **protocolo HTTP** y el lenguaje de marcado **HTML**. El primero permite una implementación sencilla de un sistema de comunicaciones que permite enviar cualquier archivo de forma fácil, simplificando el funcionamiento del servidor y posibilitando que servidores poco potentes atiendan cientos o miles de peticiones y reduzcan de este modo los costes de despliegue.
El segundo, el lenguaje HTML, proporciona un mecanismo sencillo y muy eficiente de creación de páginas enlazadas.

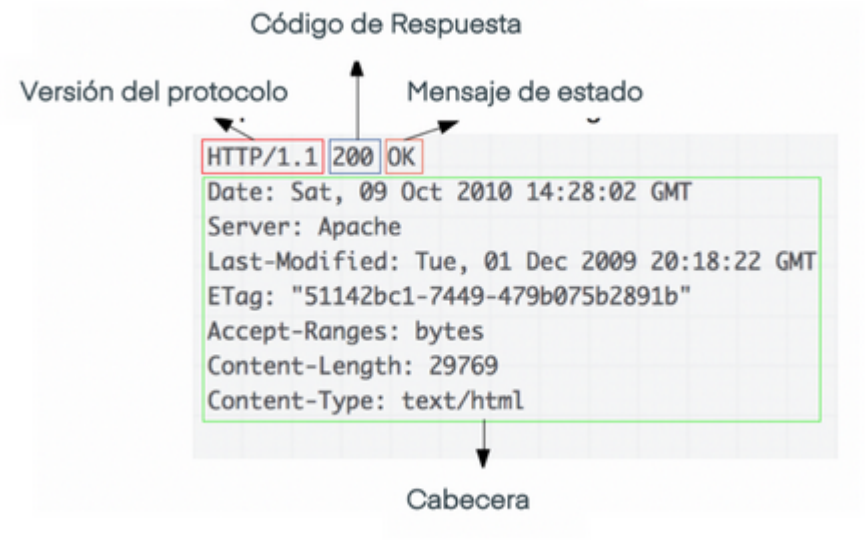
Un **protocolo** es una **PETICIÓN** o SOLICITUD desde el cliente hacia un servidor.
El **protocolo HTTP** (Hypertext Transfer Protocol) es el protocolo principal de la World Wide Web. Es un protocolo simple, orientado a conexión y sin estado. Es el código o lenguaje en el que el navegador le comunica al servidor qué página quiere visualizar.
HTTPS (S de "secure", o "seguro") que utiliza el protocolo de seguridad SSL (o "Secure Socket Layer") para cifrar y autenticar el tráfico de datos.

Funcionamiento: el cliente establece una conexión TCP con el servidor, hacia el puerto por defecto para el protocolo HTTP envía una orden HTTP de solicitud de un recurso (añadiendo algunas cabeceras con información) y, utilizando la misma conexión, el servidor responde enviando los datos solicitados y, además, añadiendo algunas cabeceras con información.
Existen dos tipos de mensajes HTTP: peticiones y respuestas, cada uno sigue su propio formato.

Una petición:



Una respuesta:



API: (Application Programing interface) que en español significa interfaz de programación de aplicaciones. Las aplicaciones son cualquier tipo de software, y la interfaz es un contrato de comunicación que una aplicación habilita para permitir que otro software le solicite información o acciones a realizar, de ahí la parte de "programación".

REST (Representational State Transfer) es un estilo de arquitectura de software que se basa en principios clave, diseñados para permitir la comunicación entre sistemas a través de internet. Estos son los conceptos fundamentales:

- Recursos: En una API REST, todo es un recurso, como objetos, datos o entidades. Cada recurso se identifica de manera única mediante una URL.

- Verbos HTTP: Las operaciones CRUD se mapean a los métodos HTTP:
- GET: Obtener información sobre un recurso.
- POST: Crear un nuevo recurso.
- PUT: Actualizar un recurso existente.
- DELETE: Eliminar un recurso.

Operaciones RESTful

Operación	Método HTTP	URI	Parámetros	Resultado
Listar	GET	/{recurso}	No aplica	Lista del tipo de recurso
Crear	POST	/{recurso}	Dentro del cuerpo en el POST	Se crea un nuevo recurso
Leer	GET	/{recurso}/{recurso_id}	No aplica	Recurso en función al id
Actualizar	PATCH/PUT	/{recurso}/{recurso_id}	Se pasan usando una cadena de consulta	Se actualiza/reemplaza el recurso
Borrar	DELETE	/{recurso}/{recurso_id}	No aplica	Se elimina el recurso en función al id

- Sin estado: Cada solicitud del cliente al servidor debe contener toda la información necesaria para procesar la solicitud, sin depender de estados anteriores.
-Transferencia de Representación: Los recursos se transfieren entre el cliente y el servidor en formatos como JSON, XML o YAM

Así como REST es el estilo de Arquitectura, RESTful es la Implementación de dicha arquitectura.

Las API REST pueden categorizarse en diferentes niveles de madurez,

Nivel 2 - Verbos HTTP: Las operaciones que se realizan sobre los recursos son las operaciones de creación, obtención, actualización y eliminación o CRUD. Usando los diferentes verbos del protocolo HTTP es posible asignar a cada uno de ellos las diferentes operaciones básicas de manipulación de datos. Se usan los códigos de estado HTTP, que son números que indican el resultado de la operación:

- 200: la operación se ha procesado correctamente.
- 201, CREATED: un nuevo recurso ha sido creado.
- 400, BAD REQUEST: la respuesta es inválida y no puede ser procesada.

La descripción del mensaje de error puede ser devuelta en lo datos retornados.

- 401, UNAUTHORIZED: acceder o manipular el recurso requiere autenticación.
- 403, FORBIDDEN: el servidor entiende la petición pero las credenciales proporcionadas
no permiten el acceso.
- 404, NOT FOUND: el recurso de la URL no existe.
- 500, INTERNAL SERVER ERROR: se ha producido un error interno al procesar la petición por un fallo de programación. En la respuesta no siempre se devuelve una descripción del error, sin embargo en las trazas del servidor debería haber información detallada del error.

Nivel 3 - HATEOAS (Hypermedia as the Engine of Application State / Hipertexto como motor del estado de la aplicación): Este nivel consta de dos partes: negociación de contenido y descubrimiento de enlaces en recursos.y desea los datos en formato JSON debe incluir la cabecera "Accept: application/json". Si prefiere los datos en formato XML, utilizará "Accept: application/xml".

Esas capas se mantienen igual a como veníamos trabajando. La diferencia estará en los controladores que ya no devolverán una página HTML, sino que deberán recibir y proveer información en formato JSON

Spring tiene la anotación **@RestController** que suplanta a @Controller, esta anotación permite convertir automáticamente objetos de Java a JSON, de esta manera en nuestros métodos solo debemos poner el tipo de dato que queremos devolver y Spring se encargará de devolver la respuesta en formato JSON:

@REQUESTBODY
En el caso de que queramos recibir datos para guardar o modificar una entidad ahora deberemos usar la anotación @RequestBody acompañado del tipo de objeto java que esperamos recibir,
Spring se encargará automáticamente de serializar el JSON recibido y convertirlo a un objeto Java.

Los **DTO** (data transfer object) son objetos diseñados específicamente para transportar datos entre diferentes capas de una aplicación o entre distintas aplicaciones. Proporcionan una manera eficiente y estructurada de mover datos, evitando la exposición innecesaria de detalles internos que hacen referencia a la base de datos.
Al usar DTOs también deberemos usar clases **"converter"** que se encargarán de tener métodos que conviertan un UserDTO en un User y viceversa.

Se recomienda no eliminar registros de la base de datos, en su lugar tener un atributo como por ejemplo “active” para darlo de baja y en las consultas a las bases de datos ignorar los inactivos. BAJA LÓGICA.-

Para el **manejo de excepciones** se pueda usar un bloque try catch en los controladores para
atrapar las excepciones que se producen.
Se recomienda utilizar la clase **ResponseStatusException** de Spring que te permite lanzar excepciones con códigos de estado HTTP.

@RestControllerAdvice es una anotación en Spring que te permite definir una clase que manejan excepciones globalmente en toda tu aplicación. Puedes crear una clase anotada con @RestControllerAdvice y métodos anotados con **@ExceptionHandler** para manejar excepciones específicas.

La clase **ErrorMessage es una clase DTO** que tiene los atributos statusCode de tipo int y message de tipo String.

ResponseEntity es una clase de Spring que permite controlar y personalizar la respuesta que se envía al cliente cuando se procesa una solicitud web, incluyendo el estado HTTP, el cuerpo de la respuesta y las cabeceras, en este caso solo estamos usando el cuerpo y el estado HTTP.

CORS significa "Cross-Origin Resource Sharing" (Compartir recursos entre distintos orígenes) y es un mecanismo de seguridad que se aplica en los navegadores web para controlar las solicitudes y respuestas entre dos dominios diferentes. CORS es necesario para garantizar que los navegadores restrinjan las solicitudes de recursos (como datos JSON o imágenes) desde un dominio web a otro, a menos que se especifiquen explícitamente como seguras y permitidas.

Guia de Seguridad Egg:

Spring Security es el framework (marco de trabajo) encargado de gestionar todo lo relativo a la seguridad dentro de Spring/Spring Boot. Algunas de las funciones/características principales las que se encarga son Spring Securiy son:
o Protocolos de seguridad.
o Roles para los accesos a los recursos o no.
o Autentificación y autorización con Spring Security.

AUTORIZACIÓN Y AUTENTICACIÓN BÁSICA
Autenticación: verificamos la identidad del usuario.
Autorización: tipo de permisos que tiene ese usuario.

Para crear una clase de seguridad personalizada, necesitamos usar @EnableWebSecurity y
extender la clase con @WebSecurityConfigurerAdapter para que podamos redefinir algunos de los
métodos proporcionados. Spring Security te fuerza a hashear las contraseñas para que no se
guarden en texto plano. Para los siguientes ejemplos, vamos a usar BCryptPasswordEncoder.

Crearemos una clase "WebSecurity" donde ubicaremos la anotación @Configuration.

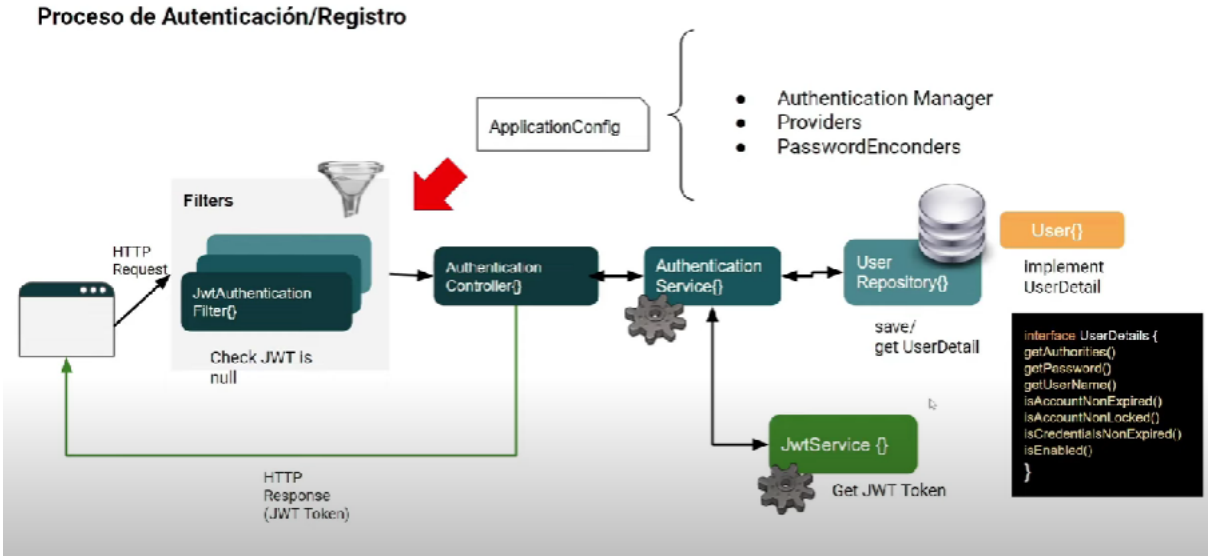
AUTENTICACIÓN
A continuación del código anterior, vamos a utilizar el método configureGlobal que recibe como
parámetro AuthenticationManagerBuilder auth y tiene el método userDetailsService, además del
encriptador de contraseñas.

Para obtener los usuarios, utilizaremos un Service que se conectará a un repositorio que
implementa la interfaz JPA para acceder a nuestra base de datos.
Para que el Service funcione, tiene que estar anotado como @Service y debe implementar la
interfaz UserDetailsService.
La interfaz UserDetailsService se utiliza para recuperar datos relacionados con el usuario. Tiene

un método llamado loadUserByUsername () que se puede sobreescribir para personalizar el
proceso de búsqueda del usuario.

SECURITY con JWT

@Bean en Spring se utiliza para indicarle al contenedor de Spring que un método específico en una clase de configuración produce un bean que debe gestionarse y estar disponible para ser utilizado en otras partes de la aplicación.



Tutorial de Spring Security - JWT - Roles de usuario - CORS

- Implementación de **Spring Security**
- Login y Registro con **Json Web Token**
- Autorizar el acceso a un endpoint a usuarios con un **ROL** determinado
- Configuración de **CORS** para que el Front-end pueda acceder a tu proyecto

Introducción - Autenticación con Json Web Token

Al loguearse el usuario, si las credenciales (usuario y contraseña) son válidas, le enviará al front un JWT.

El front almacenará ese JWT en una cookie o sessionStorage, y lo incluirá en las posteriores requests para acceder a los endpoints protegidos.
El JWT contiene:

- HEADER: tipo de token y algoritmo de firma utilizado
- PAYLOAD: id usuario, roles, permisos. Se le pueden agregar más datos
- SIGNATURE: para garantizar que no haya sido manipulado

1 - Crear Proyecto Spring / Agregar dependencias a pom.xml

Al crear el proyecto con Spring Initializr debes agregar las siguientes dependencias:

- Spring Web
- Spring Security
- Spring Data JPA
- MySQL Driver
- Lombok (librería de anotaciones para ahorrarse el código de getters, setters y constructores)
- Validation (anotaciones para validar atributos @NotNull, @NotBlank, etc.)

Si estás trabajando en un proyecto ya iniciado, revisa en tu archivo pom.xml que tenga todo lo que figura a continuación.
Además hay que agregar manualmente las 3 dependencias de JWT.

```
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-api</artifactId>
			<version>0.11.5</version>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-impl</artifactId>
			<version>0.11.5</version>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>io.jsonwebtoken</groupId>
			<artifactId>jjwt-jackson</artifactId>
			<version>0.11.5</version>
			<scope>runtime</scope>
		</dependency>
<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-data-jpa</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-security</artifactId>
		</dependency>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-web</artifactId>
		</dependency>

		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-devtools</artifactId>
			<scope>runtime</scope>
			<optional>true</optional>
		</dependency>
		<dependency>
			<groupId>com.mysql</groupId>
			<artifactId>mysql-connector-j</artifactId>
			<scope>runtime</scope>
		</dependency>
		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
```

```
                <optional>true</optional>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-test</artifactId>
                <scope>test</scope>
            </dependency>
            <dependency>
                <groupId>org.springframework.security</groupId>
                <artifactId>spring-security-test</artifactId>
                <scope>test</scope>
            </dependency>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-validation</artifactId>
            </dependency>

        </dependencies>

        <build>
            <plugins>
                <plugin>
                    <groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>
                    <configuration>
                        <excludes>
                            <exclude>

<groupId>org.projectlombok</groupId>
                                <artifactId>lombok</artifactId>
                            </exclude>
                        </excludes>
                    </configuration>
                </plugin>
            </plugins>
        </build>
```

2 - Crear Clase Controlador AuthController

En esta clase están los endpoints para autenticación (Login y Registro). Estos métodos no estarán protegidos (se podrán acceder por un usuario no autenticado), esto lo indicaremos posteriormente en una clase de Configuración. Ambos devolverán un JWT.

Llama a los métodos de la Clase Servicio Authservice, los cuales desarrollaremos más adelante.

```
@RestController
@RequestMapping("/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthService authService;

    @PostMapping("login")
    public ResponseEntity<AuthResponse> login(@RequestBody LoginDto
datos) {
          try {
                  return ResponseEntity.ok(authService.login(datos));
        } catch (RuntimeException e) {
            return new ResponseEntity(e.getMessage(),
HttpStatus.BAD_REQUEST);
        }
    }

    @PostMapping("registro")
    public ResponseEntity<AuthResponse> registro(@RequestBody
RegistroDto datos) {
          try {
                  return ResponseEntity.ok(authService.registro(datos));
        } catch (RuntimeException e) {
            return new ResponseEntity(e.getMessage(),
HttpStatus.BAD_REQUEST);
        }
    }

}
```

3 - Crear Clases DTO

Creamos las clases **LoginDto** y **RegistroDto** que habiamos indicado como parámetros en los endpoints Login y Registro. También **AuthResponse**, la respuesta que retornarán esos endpoints: el JWT como String.

```
@Data
@Builder
```

```
@NoArgsConstructor
@AllArgsConstructor
public class RegistroDto {
    String email;
    String password;
    String nombre;
    String apellido;
    String pais;
    String rol;
}
@Data
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class LoginDto {
    String email;
    String password;
}
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class AuthResponse {
    String token;
}
```

4 - Implementar UserDetails en tu Entidad Usuario

- UserDetails es un usuario de Spring Security. Es una interfaz, y debes implementarla en la entidad que será el usuario de tu app (User, Usuario, Persona, etc.). Al hacerlo, tu IDE te pedirá sobreescribir sus métodos.
- UserDetails tiene como atributos **username** y **password**. Aquí sobreescribimos el método getUsername y le indicamos que usaremos el email como username.
- En este caso no hizo falta sobreescribir el método getPassword porque ya tenemos un atributo "password" en la entidad User, y Lombok se está encargando de crear el getter por la anotación @Data. Si al campo le pusiste otro nombre (ej: contrasena) tu IDE te forzará a implementar el método getPassword, al cual habrá que pasarle el atributo contrasena.
- Le agregamos como atributo el Rol. **Los roles estarán listados en una Clase Enumerador** (ver más abajo).

- En el método getAuthorities le pasamos el rol, serán los permisos que tiene ese usuario.
- A los métodos de expiración le ponemos todo true. No los usaremos, ya que eso se manejará con el JWT.

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
@Entity
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    Integer id;
    String email;
    String apellido;
    String nombre;
    String pais;
    String password;

    @Enumerated(EnumType.STRING)
    Role role;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
      return List.of(new SimpleGrantedAuthority((role.name())));
    }
    @Override
    public String getUsername() {
        return email;
    }
    @Override
    public boolean isAccountNonExpired() {
       return true;
    }
    @Override
    public boolean isAccountNonLocked() {
       return true;
    }
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }
    @Override
    public boolean isEnabled() {
```

```
        return true;
    }
}
public enum Role {
        COMPRADOR,
    VENDEDOR
}
```

5 - Agregar query en Clase Repositorio

En el repositorio de tu clase usuario agregamos un método para buscar por el atributo que habíamos decidido utilizar como **username**, en este caso el **email**.

```
public interface UserRepository extends JpaRepository<User,Integer> {
    Optional<User> findByEmail(String email);
}
```

6 - Crear Clase de Configuración SecurityConfig

- Esta clase contiene la **SecurityFilterChain**. Todas las requests que reciba nuestra API pasarán por esta cadena de filtros.
- Le indicamos que los endpoints en la ruta /auth/ (login y registro) serán públicos (son los de la clase AuthController, que hicimos en el punto 2#) .
- Para acceder a los demás endpoints, el usuario deberá estar autenticado (.anyRequest().authenticated())
- Deshabilitamos csrf y session. Son métodos predeterminados de Spring Security que no usaremos, porque la autenticación la haremos con JWT.
- Agregamos el **jwtAuthenticationFilter** (lo desarrollaremos luego).
- El authenticationProvider es el responsable de recibir una solicitud de autorización y decidir si es válida o no. Más adelante, en otra clase de configuración indicaremos cuál provider implementaremos.
- La anotación @EnableMethodSecurity(securedEnabled = true) nos permitirá incluir en los controladores la anotación **@Secured** para indicar el **ROL** de los usuarios que tendrán acceso a los mismos.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(securedEnabled = true)
@RequiredArgsConstructor
```

```
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtAuthenticationFilter;
    private final AuthenticationProvider authProvider;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception
    {
        return http
            .csrf(csrf ->
                csrf
                .disable())
            .authorizeHttpRequests(authRequest ->
              authRequest
                .requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
                )
            .sessionManagement(sessionManager->
                sessionManager
                  .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authenticationProvider(authProvider)
            .addFilterBefore(jwtAuthenticationFilter,
UsernamePasswordAuthenticationFilter.class)
            .build();
    }
}
```

7 - Crear Filtro JwtAuthenticationFilter

Ya indicamos anteriormente en SecurityConfig que todas las peticiones deben pasar por este filtro.

- El filtro hereda de OncePerRequestFilter (se ejecutará una vez sola por cada request).
- Obtenemos el token que viene incluido en la request llamando al método getTokenFromRequest (ver más abajo). El mismo busca el token que está en el HEADER de la request y le quita la palabra "Bearer".
- Si la request no tiene JWT, continuamos con la cadena de filtros, donde habíamos indicado que solo podría acceder al login y registro en /auth/.
- Si la request viene con un JWT, buscará el usuario en nuestra Base de Datos. Luego lo validará (credenciales correctas, no

expirado) y si está todo ok lo guardará en el SecurityContextHolder.

- SecurityContextHolder es un método estático para recuperar los datos del usuario. Permitirá llamarlo desde cualquier parte de nuestro código sin pasarle ningún parámetro.

```
@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
HttpServletResponse response, FilterChain filterChain)
            throws ServletException, IOException {

        final String token = getTokenFromRequest(request);
        final String username;

        if (token==null)
        {
            filterChain.doFilter(request, response);
            return;
        }

        username=jwtService.getUsernameFromToken(token);

        if (username!=null &&
SecurityContextHolder.getContext().getAuthentication()==null)
        {
            UserDetails
userDetails=userDetailsService.loadUserByUsername(username);

            if (jwtService.isTokenValid(token, userDetails))
            {
                UsernamePasswordAuthenticationToken authToken= new
UsernamePasswordAuthenticationToken(
                    userDetails,
                    null,
                    userDetails.getAuthorities());

                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
```

```
SecurityContextHolder.getContext().setAuthentication(authToken);
            }

        }

        filterChain.doFilter(request, response);
    }

    private String getTokenFromRequest(HttpServletRequest request) {
        final String
authHeader=request.getHeader(HttpHeaders.AUTHORIZATION);

        if(StringUtils.hasText(authHeader) && authHeader.startsWith("Bearer
"))
        {
            return authHeader.substring(7);
        }
        return null;
    }
}
```

8 - Crear Clase de Configuración AppConfig

- **AuthenticationManager** es una interfaz de de Spring Security, responsable de manejar el proceso de autenticación de usuarios.
- El proveedor de autenticación a implementar será **DaoAuthenticationProvider**, que valida las credenciales (usuario y contraseña) contra una Base de Datos. Otro proveedor utilizado comúnmente es OAuth2Login, que sirve para inciar sesión con Google, Facebook, etc.
- Para encriptar las contraseñas utilizaremos el algoritmo **Bycrypt**.
- **UserDetailsService** se encargará de buscar el usuario en la base de datos. Recordemos que habíamos definido que utilizaríamos como username el **email**.
- **CORS** (Cross-Origin Resource Sharing) es un mecanismo de seguridad que tienen los navegadores web para restringir peticiones HTTP entre distintos servidores. Es necesario agregar esta configuración para que el Front pueda acceder a nuestra API. Completa la línea de .allowedOrigins(...) con la URL que utilizará el front-end.

```
@Configuration
@RequiredArgsConstructor
```

```
public class AppConfig {

    private final UserRepository userRepository;

    @Bean
    public AuthenticationManager
authenticationManager(AuthenticationConfiguration config) throws
Exception
    {
        return config.getAuthenticationManager();
    }

    @Bean
    public AuthenticationProvider authenticationProvider()
    {
        DaoAuthenticationProvider authenticationProvider= new
DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());
        return authenticationProvider;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public UserDetailsService userDetailService() {
        return username -> userRepository.findByEmail(username)
        .orElseThrow(()-> new UsernameNotFoundException("User not
fournd"));
    }

    @Bean
    public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override
            public void addCorsMappings(@NotNull CorsRegistry registry) {
                registry.addMapping("/**")
                        .allowedOrigins("http://localhost:5173")  // URL del
Front-end
                        .allowedMethods("GET", "POST", "PUT", "DELETE","OPTIONS")
                        .allowCredentials(true);
            }
```

```
        };
    }
}
```

9 - Crear Clase Servicio JWTService

Este servicio contendrá métodos para generar el JWT, verificar su validez y extraer información del mismo.

- La SECRET_KEY sirve para validar la firma del token. Con la anotación @Value le asignamos el valor de la variable jwt.secret, que guardamos en el archivo application.properties (ver más abajo).
- El método getToken recibirá por parámetro un usuario de Spring Security (UserDetails), y construirá un JWT. Su firma se realiza con la SECRET_KEY y el algoritmo HS256.
- La expiración del token se expresa en milisegundos. Un día tiene 86400 segundos (60 seg x 60 min x 24 hs.). Este token expirará en un día.
- El método **isTokenValid** verifica si el token es válido comprobando el username (getUsernameFromToken) y su expiración (isTokenExpired)

```
@Service
public class JwtService {

    @Value("${jwt.secret}")
    private String SECRET_KEY;

    public String getToken(UserDetails user) {
        return getToken(new HashMap<>(), user);
    }

    private String getToken(Map<String,Object> extraClaims, UserDetails
user) {
        return Jwts
            .builder()
            .setClaims(extraClaims)
            .setSubject(user.getUsername())
            .setIssuedAt(new Date(System.currentTimeMillis()))
            .setExpiration(new Date(System.currentTimeMillis()+1000*86400))
            .signWith(getKey(), SignatureAlgorithm.HS256)
            .compact();
    }
```

```
    private Key getKey() {
       byte[] keyBytes=Decoders.BASE64.decode(SECRET_KEY);
       return Keys.hmacShaKeyFor(keyBytes);
    }

    public String getUsernameFromToken(String token) {
        return getClaim(token, Claims::getSubject);
    }

    public boolean isTokenValid(String token, UserDetails userDetails) {
        final String username=getUsernameFromToken(token);
        return (username.equals(userDetails.getUsername())&&
!isTokenExpired(token));
    }

    private Claims getAllClaims(String token)
    {
        return Jwts
            .parserBuilder()
            .setSigningKey(getKey())
            .build()
            .parseClaimsJws(token)
            .getBody();
    }

    public <T> T getClaim(String token, Function<Claims,T> claimsResolver)
    {
        final Claims claims=getAllClaims(token);
        return claimsResolver.apply(claims);
    }

    private Date getExpiration(String token)
    {
        return getClaim(token, Claims::getExpiration);
    }

    private boolean isTokenExpired(String token)
    {
        return getExpiration(token).before(new Date());
    }

}
```

Application.properties :

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=
jdbc:mysql://localhost:3306/securityJWT?useSSL=false&serverTimezone=
UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
jwt.secret=
123456789654564564dsa65f4s56d4f65sdf56sd564f65sdf65sd6f54sd6f
```

10 - Crear Clase Servicio AuthService

Finalmente podemos desarrollar aquí los métodos de login y registro invocados por el AuthController que hicimos en el paso #2

- Registro: Recibe el DTO con los datos de registro, el cual incluye el email. Si ya existe un usuario en la Base de Datos con ese email, lanzará un mensaje de error. De lo contrario guardará el usuario en la BD y devolverá el JWT llamando al JWTService del paso anterior.
- Login: Autentica al usuario con las credenciales que recibe dentro del LoginDto. Busca al usuario en la BD y genera el JWT.

```
@Service
@RequiredArgsConstructor
public class AuthService {

    private final UserRepository userRepository;
    private final JwtService jwtService;
    private final PasswordEncoder passwordEncoder;
    private final AuthenticationManager authenticationManager;

    public AuthResponse login(LoginDto datos) {
        authenticationManager.authenticate(new
UsernamePasswordAuthenticationToken(datos.getEmail(),
datos.getPassword()));
        UserDetails user =
userRepository.findByEmail(datos.getEmail()).orElseThrow();
        String token = jwtService.getToken(user);
        return AuthResponse.builder()
            .token(token)
            .build();

    }
```

```
    public AuthResponse registro(RegistroDto datos) {

        Optional<User> userOptional =
userRepository.findByEmail(datos.getEmail());
        if (userOptional.isPresent()) {
            throw new RuntimeException("Ya existe un usuario con ese email");
        }

        User user = User.builder()
            .email(datos.getEmail())
            .password(passwordEncoder.encode(datos.getPassword()))
            .nombre(datos.getNombre())
            .apellido(datos.getApellido())
            .pais(datos.getPais())
            .role(Role.valueOf(datos.getRol()))
            .build();

        userRepository.save(user);

        return AuthResponse.builder()
            .token(jwtService.getToken(user))
            .build();

    }
}
```

11 - Asignar Roles de acceso a los endpoints

Mediante la anotación @Secured("ROL") indicamos el rol que debe tener el usuario para poder acceder a cada endpoint. Recordemos que para que esta anotación funcione, pusimos esta otra anotación en SecurityConfig: @EnableMethodSecurity(securedEnabled = true).
Si varios roles tienen permiso a ese endpoint se puede poner así: @Secured({"ADMIN", "ROL1", "ROL2"})
Aquí tenemos una clase controlador con unos endpoints de ejemplo:

- probando: podrá ser accedido por cualquier usuario que esté logueado, independientemente de su rol, ya que no utilizamos la anotación @Secured.
- endpointComprador: solo podrá ser accedido por un usuario con rol "COMPRADOR". Si el usuario tiene otro rol, devolverá un 403.
- endpointVendedor: solo podrá ser accedido por un usuario con rol "VENDEDOR". Si el usuario tiene otro rol, devolverá un 403.

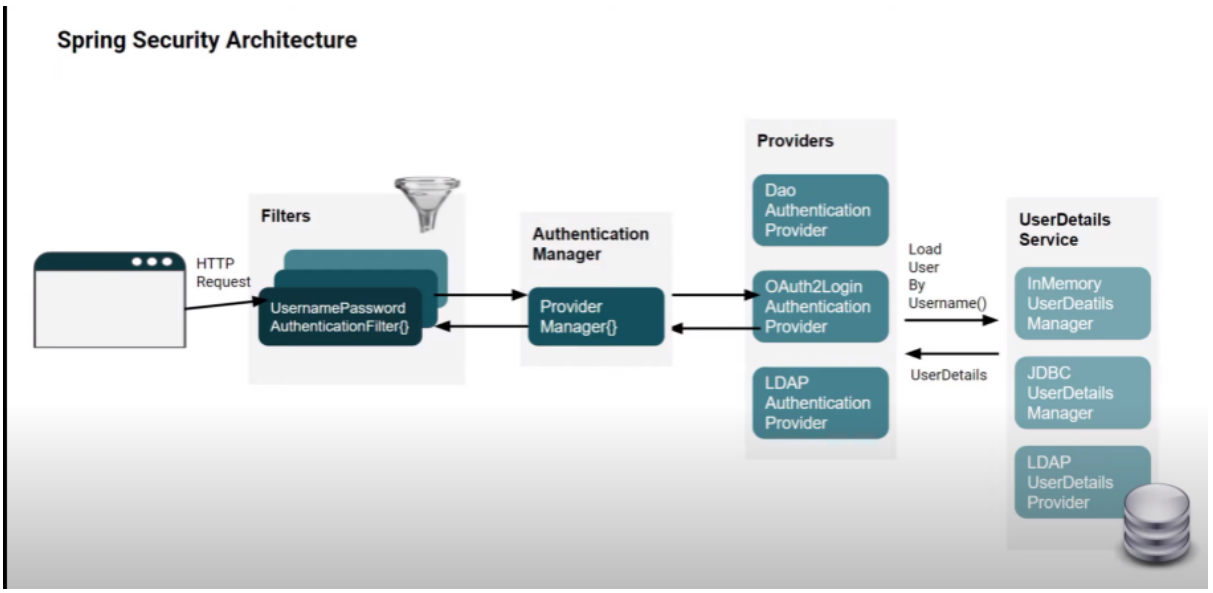
```
@RestController
@RequestMapping("/test")
@RequiredArgsConstructor
public class TestController {

    @GetMapping()
    public String probando() {
        return "Hola Mundo";
    }

    @Secured("COMPRADOR")
    @GetMapping("endpointComprador")
    public String endpointComprador() {
        return "Hola, soy un comprador";
    }

    @Secured("VENDEDOR")
    @GetMapping("endpointVendedor")
    public String endpointVendedor() {
        return "Hola, soy un vendedor";
    }

}
```

12 - Testear endpoints en Postman

Los endpoints de Login y Registro devolverán un JWT.

Para acceder a los métodos protegidos, copiar y pegar el token en Authorization - Bearer Token. Sin ese token, cualquier petición a un endpoint protegido devolverá un 403.

En el sitio Web JWT.IO se puede decodificar el JWT y ver su contenido.


Spring Security Architecture
Filters
HTTP
Request
UsernamePassword
AuthenticationFilter{}
Authentication
Manager
Provider
Manager{}
Providers
Dao
Authentication
Provider
OAuth2Login
Authentication
Provider
LDAP
Authentication
Provider
Load
User
By
Username()
UserDetails
UserDetails
Service
InMemory
UserDeatils
Manager
JDBC
UserDetails
Manager
LDAP
UserDetails
Provider