**Curso de Java Spring**

**¿Java sigue siendo gratuito?**

Cambio de licencia del JDK de Java

* Oracle siempre ha tenido una relación amor/odio con el OpenSource
* Desde Java 9, cada 6 meses existe una versión “mayor”
* JDK 11
* Nueva licencia para uso en producción / comercial y LTS

**Alternativas al JDK de JAVA**

* **Open JDK**
* Alternativas basadas en OpenJDK
  + Amazon Corretto
  + RedHat OpenJDK
  + Otras

**Instalación de ambiente de desarrollo: Windows**

¿Qué vas a necesitar?

Para este curso vas a ocupar cuatro herramientas:

* El OpenJDK que ya vimos en la clase pasada y que nos dará todo lo necesario para construir aplicaciones usando Java.
* El IDE que usaremos en el curso será IntelliJ IDEA, la versión será la Community Edition que se acomoda sin problemas a nuestras necesidades.
* PostgreSQL será la base de datos encargada de gestionar toda la información que será expuesta en la API. Más adelante vamos a configurarla y a configurar un set de datos inicial.
* Postman, es un cliente donde vamos a probar de manera sencilla todos los servicios expuestos de nuestra API antes de publicarla.

**¿Qué es Spring?**

* Es el framework más popular de Java
* Usado por compañías como Netflix y Mercedes-Benz

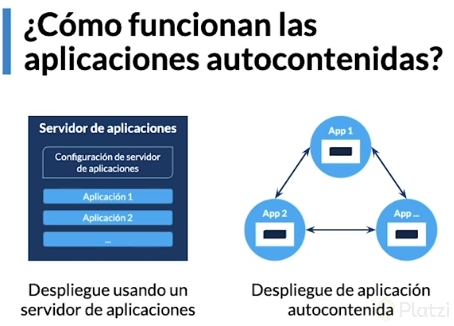
Spring es el framework más usado de Java. Nos ofrece herramientas que nos permite crear proyectos más avanzados, con mejores prácticas y en menor tiempo. También posee una gran comunidad, lo que nos brinda muchísima documentación y ayuda.

Spring tiene una estructura modular y flexible, lo que nos hace usar solo lo que necesitemos.

Vamos a usar 4 subproyectos de Sping:

* Spring Framework: Permite crear aplicaciones empresariales. Es transversal, ya que todos lo usan.
* Spring Boot: Con el que podemos crear aplicaciones autocontenidas y autoconfigurables.
* Spring Data: Gestionar e integrar bases de datos.
* Spring Security: Gestionar la seguridad de la aplicación.

**¿Qué es una aplicación autocontenida?**

Hasta hace algún tiempo las aplicaciones web empresariales lucían como el grafico de la izquierda. Teníamos un servidor de aplicaciones que poseía su configuración - donde van configuración de bases de datos, servicios externos, variables, etc. - Allí también iban desplegadas todas las aplicaciones que quisiéramos que interactuaran o colaboraran entre sí.

Ahora las tendencias han cambiado, las arquitecturas modernas nos sugieren tener algo como el diagrama de la derecha donde tengamos pequeñas aplicaciones o pequeños servicios que interactúen entre sí en vez de una gran aplicación. Esto nos da mucha facilidad al momento de desarrollo y al momento de mantener nuestra aplicación. Cada aplicación internamente contiene su propio servidor de aplicaciones, con una configuración totalmente independiente una de la otra.

**Spring Boot**

* Es el proyecto de Spring para aplicaciones autocontenidas
* Olvidarnos de la infraestructura y centrarnos en el desarrollo
* Puede funcionar con Tomcat (por defecto), Jetty o Undertow.
* Incluye gestión de dependencias iniciales, configuración automática y más

**¿Qué es Spring Initializr?**

[start.spring.io](https://start.spring.io/)

* Sitio oficial para generar un proyecto de Spring Boot
* En poco tiempo y a nuestra medida
* Con todo lo que necesitamos para empezar
* ¡Hagámoslo!

**Project:**

* Los projectos Maven gestionan las dependencias con archivos XML
* Los archivos Gradle son escritos en Groovy y permite crear tareas que se pueden ejecutar al momento de hacer despliegue o integración continua
* Utilizar la version estable de spring boot y llenar los metadatos
* Spring Web: permite genera APIs rest utilizando Apache Tomcat como servidor por defecto
* build.gradle: Es donde está escrita toda la configuración del proyecto (plugins, versión de Java, dependencias, software para pruebas etc.)
* Carpeta gradle: Archivos necesarios para que gradle funcione
* @SpringBootApplication: indica la clase que gestiona la aplicación

**Configurar Spring Boot**

**Propiedades de la aplicación**

* Application.properties, application.yml o línea de comando
* Posibilidad de añadir propiedades propias
* Gestión de perfiles según el tipo de despliegue.

¿Qué atributos podemos modificar? En la documentación de Spring lo vemos. [Documentacion Spring](https://docs.spring.io/spring-boot/docs/current/reference/html/application-properties.html)

**Crear la estructura del proyecto**

La estructura del proyecto será de la siguiente manera:

1. DOMINIO:
   * DTO y objetos del dominio (Contexto de la aplicación)
   * Servicios: Puente entre los controladores y la capa de persistencia.
   * Especificación de repositorios: Interfaces que determinan las reglas que debe cumplir la persistencia para actuar entre los objetos de dominio y la DB.
2. WEB:
   * Controladores de API Rest.
3. PERSISTENCIA:
   * Repositorios: Implementan las especificaciones que tiene la capa de DOMINIO.
   * Entities: Mapean y actúan como tablas de la DB.

Es similar a la estructura “Clean Arquitecture” donde las capas son:

* Domain
* Presentation
* Data

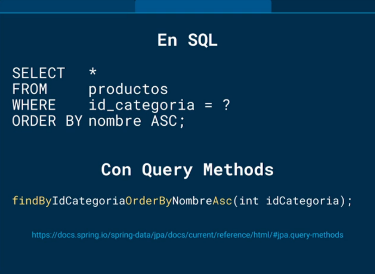


**¿Qué es JPA?**

* JPA es una especificación de Java (un estándar) para un framework ORM
* Interactuar con las tablas de la base de datos en forma de objetos Java
* Algunas de sus implementaciones son:
  + - * Hibernate
      * Eclipse
      * TopLink
      * ObjectDB

**Anotaciones JPA**

* @Entity: Indica a una clase de java que está representando una tabla de nuestra BD.
* @Table: Recibe el nombre de la tabla a la cual está mapeando la clase.
* @Column: Se le pone a los atributos de la clase, no es obligatoria, se indica sólo cuando el nombre de la columna es diferente al nombre del atributo de la tabla.
* @Id y EmbededId: Es el atributo como clave primaria de la tabla dentro de la clase. @id se utiliza cuando es clave primaria sencilla y @EmbededID cuando es una clave primaria compuesta.
* @GeneratedValue: Permite automáticamente generar valores para las clases primarias en nuestras clases
* @OneToMany & @ManyToOne representan relaciones.

****

**¿Qué es Spring Data?**

Spring Data **NO** es una implementación de JPA, más bien es un proyecto que usa JPA para ofrecer funcionalidades extra en la gestión de tareas desde JAVA a las base de datos.

Spring Data internamente tiene varios sub proyectos, entre ellos: Spring Data JPA y Spring Data JDBC, para conectarnos a BD relacionales (SQL). Spring Data MongoDB y Spring Data Cassandra, son proyectos para conectarnos a BD no relacionales.

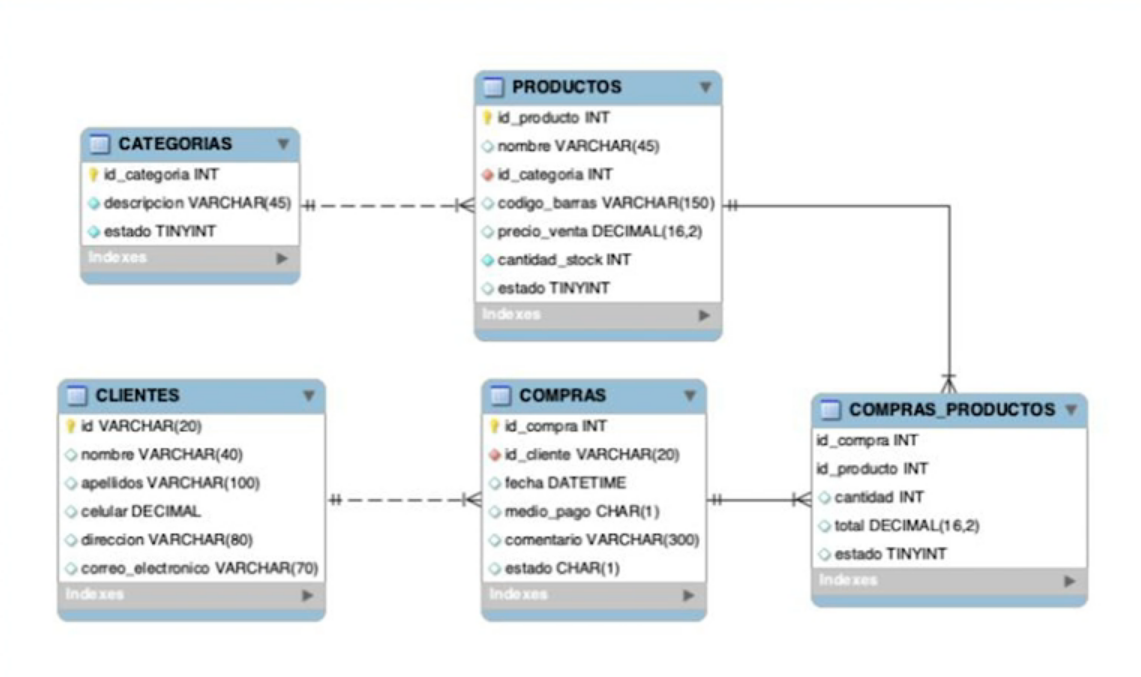
La tarea principal de Spring Data es optimizar tareas repetitivas.

Spring data nos provee de repositorios sin código, nos permiten hacer todo tipo de operaciones en BD (CRUD) sin utilizar una línea de código.

También nos provee de auditorías transparentes, por ello, posee un motor de auditorías que nos permite saber cuándo se insertó un registro, cuando se borró, cuando se actualizo en la BD, etc.

**Modelo E/R**

El modelo relacional que se seguirá para este proyecto será el siguiente:



**Spring Data Repositories**

* Ahorrar un MONTON de código y tiempo de implementación
* Operaciones SIN CODIGO en la BD
* Repositorios de Spring Data

Crud Repository: realiza el crud

PagingAndSortingRepository: incluye lo del Crud Repository además de paginación y ordenamiento

JPARepository: Ademas de tener CrudRepository y PagingAndSortingRepository nos permite tareas específicas como Flush

**Query Methods**

Los Query Method son muy potentes ya que permite realizar múltiples operaciones con:

* **Números:** mayores, menores, igual
* **Textos:** contiene cierta porción de texto, empieza o termina con una porción de texto, ignora case sensitive
* **Fechas:** Antes de cierta fecha, después de cierta fecha, entre cierta fecha.
* **Joins entre entidades:** si tenemos una entidad que se relaciona con otra, es posible realizar “joins” con esa relación para tener queries más específicas según nuestra necesidad. Por ejemplo: si tengo una relación de Producto y Categoría y quiero tener todos los productos de cierta categoría podría hacer:

findAllByCategoriasId(Integer categoriaId) y así poder llegar a esta relación. Esto puede mezclarse con múltiples relaciones en simultáneo

* **Comparación entre conjunto de datos:** Si por ejemplo quiero traerme los productos con varias categorías podría escribir findAllByCategoriasIdIn(List<Integer> categoriaIds) y así trabajar bajo un conjunto de Id de categorías.

Existen más funcionalidades: Pueden ver más detalle acá: [Spring Query Creation](https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation)

**Implementar la anotación @Repository**

Le indicamos a la clase que es la encargada de interactuar con la base de datos.

**Implementar la anotación @Component**

Le indicamos que es un componente de Spring.

**Pregunta acerca del proyecto:**

**¿Por qué no @Repository iría en la interfaz, y la clase que implementa la interfaz llevara @Service ya que prácticamente tiene todos los parámetros que se están colocando en clase? Parece que lo estuviéramos duplicando**

**Respuesta:** El @Repository es una capa de abstracción que se añade a tu sistema para desacoplar el motor de base de datos de tu aplicación, básicamente te ayuda a que tu aplicación no dependa del tipo de base de datos que estás utilizando. Si mañana quisieras cambiar Postgres por MySQL o MongoDB sólo tendrías que modificar el repositorio y el resto de la aplicación no debería verse afectada en gran medida (que sería lo ideal). Esta capa es **SÓLO** para interactuar con tu base de datos y **NO** posee lógica de negocio.

Un service por el contrario, es la capa que guarda la mayor parte de la lógica de negocio y su responsabilidad es esa, implementar lógica, no relacionarse con una base de datos.

**Patrón Data Mapper**

* Convertir o traducir dos objetos que pueden hacer una misma labor
* No exponer directamente la base datos mediante la API
* Esto garantiza que ningún agente externo, visualice la forma del diseño de la base de datos
* Desacoplar la API de una base de datos puntual
* En el caso que se desee integrar una nueva base de datos con otros campos, pero que sea para el mismo proyecto, no es necesario cambiar todo el código, simplemente se crea otro traductor que sirva para traducir la nueva tabla al dominio
* Evita tener campos innecesarios en la API
* Evitar mezclar idiomas en el dominio

**Orientar nuestro repositorio al dominio con MapStruct**

Reflexión: Con esto evitamos que nuestro proyecto se acople a una base de datos puntual, ¿Qué pasaría si el día de mañana estamos usando otra base de datos incluso que no sea SQL? Quizás Mongo, allí tenemos colecciones y estas colecciones harían que nuestro api cambiara completamente si la estuviéramos usando de manera directa. Con MapStruct nos evitamos esto, simplemente creamos un nuevo mapper que convierta Product en lo que sea que la colección tenga como nombre y el código seguirá funcionando tal cual como lo tenemos.

**Inyección de dependencias**

*Es uno de los 5 principios del diseño S.O.L.I.D*

* *Single responsibility Principle*
* *Open/Closed Principle*
* *Liskov Substitution Principle*
* *Interface Segregation Principle*
* *Dependency Inversion Principle*

**Consiste** en pasar la dependencia o clase que lo va a utilizar en lugar de crearla internamente dentro de esa clase. Esto con el fin de no acoplar la clase a la implementación que está utlizando.

**Inversión de control:** Un framework tiene el control sobre los objetos. Spring tiene un contenedor de inversión de control, donde se administran y crean instancias de objetos beans o components.

**@Autowired:** Anotación para hacer inyección de dependencias.

***Exponer nuestra API***

¿Qué anotaciones usaremos?

* Nuestra API se expone por @RestController
* Los métodos se exponen con @GetMapping, @PostMapping o @DeleteMapping

**Controlar las respuestas HTTP**

Objetivo: controlar de una mejor manera los llamados que reciben los endPoints, esto gracias a códigos HTTP particulares para las peticiones en cuestión. Para esto Spring utiliza **ResponseEntity.**

* ¿Qué es y en qué nos ayuda?

Una clase especialmente dedicada para controlar los llamados y respuesta que reciben nuestros controladores.

* HttpStatus

Para definir desde allí que código queremos retornar según cada caso puntual.

Los tipos de valores HttpStatus que devolvemos son los siguientes:

OK – NOT\_FOUND - CREATED

**Pregunta:** ¿Por qué devolvemos NOT\_FOUND en vez de NO\_CONTENT?

La razón por la que utilizamos el Not Found (404) en vez del No Content (204) es porque según su definición:

* No Content (204) se utiliza cuando la petición se ha completado con éxito pero su respuesta no tiene ningún contenido.
* Not Found (404) se usa cuando el servidor no pudo encontrar el contenido solicitado.

En este caso puntual el recurso que solicitamos no existe y por lo tanto el proceso no finalizó como se esperaba, por lo cual un 404 es lo ideal.

**Documentar nuestra API con Swagger**

Swagger es una herramienta que nos permite documentar nuestros servicios usando anotaciones como las que hemos visto hasta ahora. Documentar nuestra API **siempre es buena práctica.**

**¿Por qué documentar nuestra API?**

* Le agregaremos una capa de entendimiento
* Es más fácil de usar
* Es más profesional
* Quien consuma tendrá información oficial y de primera mano.

**Configurar la seguridad nuestra API con Spring Security**

La seguridad es una característica muy importante dentro de las aplicaciones modernas. Garantizar la protección ante ataques o agujeros de seguridad es algo que todas las aplicaciones deberían tener por defecto.

**Spring Security**

* Autenticación y autorización para aplicaciones de Spring
* Como todos los proyectos de Spring, es muy fácil de configurar
* Protección ante ataques como Session Fixation, clickjacking, cross site request forgery, entre otros.
* Configuración por defecto

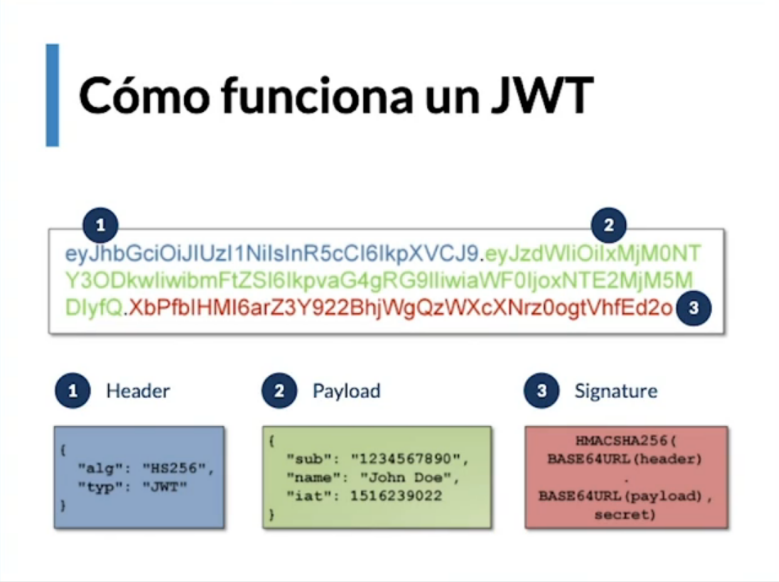
En este proyecto usaremos JSON Web Tokens para controlar las peticiones que recibe nuestra aplicación.

**Que es un JWT**

* Estándar de código abierto basado en JSON para crear tokens de seguridad
* La autenticación viaja en el header de la petición: Authorization: Bearer <token>

Un JWT por dentro está dividido en tres secciones:

1. Incluye el algoritmo y el tipo de token que se está generando
2. Toda la información del token: el usuario para el cual fue generado, cuando fue su fecha de vencimiento, entre otros.
3. La firma. Agarra los dos anteriores y los encripta según el algoritmo seleccionado

****

**Desplegar nuestra API desde la ventana de comandos**

*java –jar spring-market-1.0.jar*

***Algunas propiedades adicionales***

* *Xmx2048m:* Indica el tamaño de la memoria que se le asigna para usar la aplicación
* *-Dspring.profiles.active=pdn:* Indica a la aplicación que se comporte en un ambiente de produccion
* *-Dserver.port=88:* Cambiamos el puerto desde la terminal

**Finalmente, si ya tienes el jar y estás en la powershell deberás insertar este comando**: *java -jar "-Dspring.profiles.active=pdn" build/libs/spring-market-1.0.jar*

Con este comando accedes a la propiedades del modo producción.