# SPL

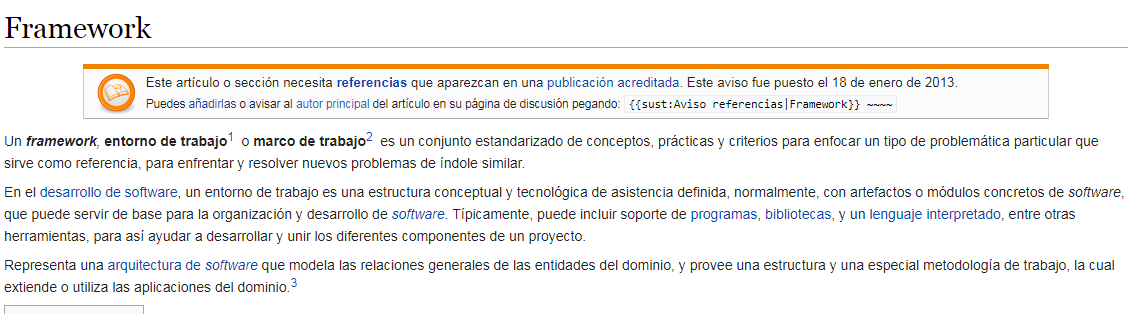
Debo antes que nada hablarles un poco de mí.

Soy Alejandro Lindarte Castro, una persona normal (no vuelo ni respiro bajo el agua), solo soy una persona fielmente enamorada de mis habilidades, tengo claro, por qué estoy aquí y a donde quiero estar, por eso, con mucho esfuerzo y sobre todo con el corazón en la izquierda (soy derecho y además hombre (Uniproceso)), debo presentarles por ahora, el primer paso para cumplir los sueños (si los sueños, esos que confundimos cuando cerramos los ojos y vemos a ese familiar que ya no está, o a esa chica por la que tanto nos protesta el corazón) y mi primer sueño se llamaba **SPL.**

# ¿Qué tal si empezamos?

El framework de persistencia SPL tiene ciertas configuraciones a nivel de software que es importante tocarlas a fondo en este documento.

Vamos a definir en primera instancia que es un framework, nuestra amada Wikipedia nos dice que:



***Imagen 1. Definición de framework según Wikipedia.***

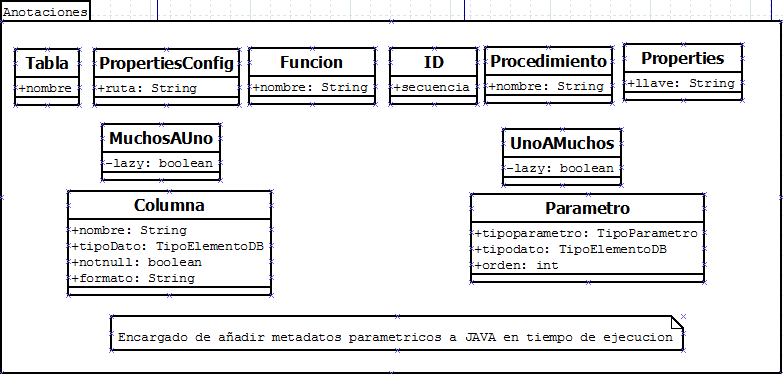
Claro!! Es un marco de trabajo que nos ayuda como punto de referencia para resolver N problemas sin necesidad de hacer N veces lo mismo. ¿Simple no?

Ahora si vamos a definir a **SPL** como un marco de trabajo para poder interactuar con nuestros sistemas de información continua de datos (Base de datos) de manera simple, transparente y con funcionalidades ya predefinidas para interactuar de manera concurrente y simple con nuestro sistema.

Ahora vamos a lo bonito, vamos a ver cómo está formado nuestro SPL

# Diagrama de clases

## Anotaciones



***Imagen 1. Anotaciones***

Vamos a hablar de los metadatos principales del framework, las anotaciones.

Pero, ¿Que es una anotación? ¿Qué es un metadato?

Tomando en cuenta nuestra amada Wikipedia de nuevo, vamos a definir primero un **Metadato**.

**Metadato:**

Son datos que describen otros datos. En general, un grupo de metadatos se refiere a un grupo de datos que describen el contenido informativo de un objeto al que se denomina *recurso.*

Interesante no? Datos que describen otros datos!! Ya me confundí otrav… NO!! Es simple, es tan simple que se vuelve algo importante y necesario de aplicar, los metadatos en **SPL** van a hacer Datos que nos ayuden a darle características particulares a nuestros datos o atributos, y tranquilos, ya le vamos dando más forma al termino **Metadato**.

**Anotación:**

Es una forma de añadir metadatos al código fuente **Java** que están disponibles para la aplicación en tiempo de ejecución.

¿Este es más claro no? Es decir, una **anotación** termina siendo un metadato en JAVA que nos ayuda a darle un valor agregado a nuestros componentes en JAVA, podemos “marcar” nuestras clases o atributos con **Anotaciones** y luego en tiempo de ejecución se resuelven X o Y acciones dependiendo de la “marca” (**Anotación**) que le pusiste (cabe aclarar que nuestras **X** o **Y** acciones deben ser programadas por nosotros, java nos ofrece un **API** introspectiva para poder incluirlas en tiempo de ejecución y dinámicamente (**java.lang.reflect**) ).

### Perfecto! ¿Que aprendimos de nuevo?

* **Metadatos:** Son datos que describen otros datos
* **Anotación:** Es una forma de añadir metadatos al código fuente

## Conexión



***Imagen 2. Diagrama clases paquete conexión***

El paquete conexión contiene todos los componentes necesarios que nos sirven para realizar conexiones a la base de datos.

Necesitamos como siempre aclarar algunos términos antes de continuar con la definición del paquete.

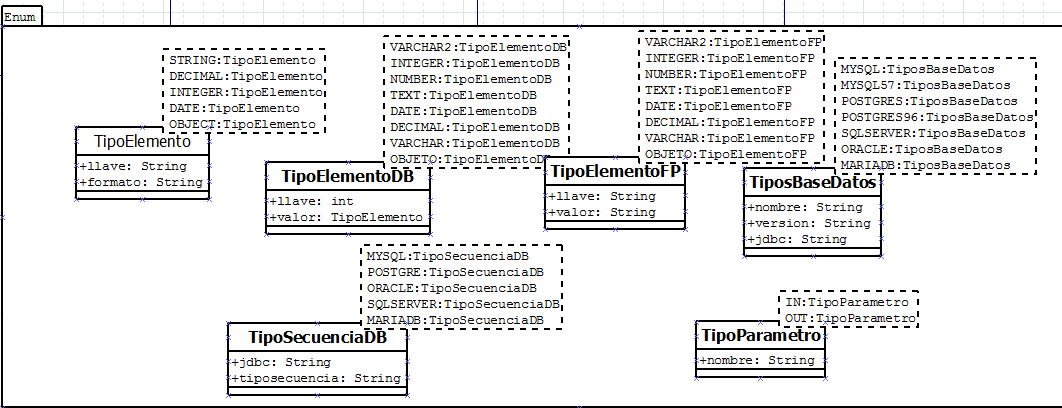
Los **DSN (Data source name)** que representa todo lo relativo a una fuente de datos configurada por el usuario para conectarse a una [Base de datos](https://es.wikipedia.org/wiki/Base_de_datos).

En este paquete es posible crear todo tipo de conexión a la base de datos por medio de JDBC, gestiona solamente la creación de este tipo de objetos, funciona como una clase abstracta builder (constructora de objetos funcionales).

### Perfecto! ¿Qué aprendimos de nuevo?

* **DSN,** fuente de datos configurada para gestionar la conexión a una base de datos
* **Patrón builder:** Es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente.
* **Conexión:** es el enlace que se establece entre el emisor y el receptor a través del que se envía el mensaje.

## Enum



***Imagen 3. Diagrama de clases del paquete ENUM***

Volviendo con nuestro interdisciplinar compañero “Wikipedia” vamos a definir que es un enum.

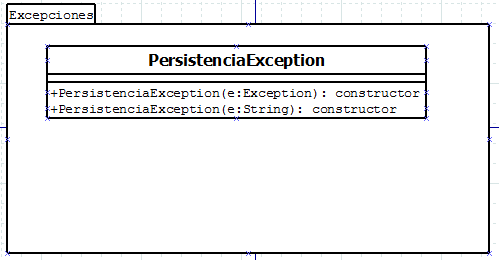
**ENUM:** Un enum es un tipo especial de datos, cuya finalidad es predefinir variables u objetos constantes.

¿Esto qué quiere decir? Los enum en SPL son utilizados para almacenar objetos constantes que sirven de referencia para construir la infraestructura lógica de SPL (esto suena a **metadato)** y si Maso menos, solo que enum es un tipo de dato más flexible y puede ser usado para almacenar más que un objeto como constante (**que no puede cambiar en toda la aplicación**).

### Perfecto! ¿Qué aprendimos de nuevo?

* **ENUM**:  proporciona una manera eficaz de definir un conjunto de constantes enteras con nombre que se pueden asignar a una variable.
* Los enum pueden ser usados como metadatos, como puntos de referencia para tratar comportamientos.

## Excepciones



***Imagen 4. Paquete excepciones.***

Primero vamos a definir que es una excepción para nuestro amigo en común (si, estoy hablando de Wikipedia)

**Excepción**: Algo fuera de lo común en el flujo normal y correcto de nuestro algoritmo.

¿Fácil no? Manejar las olvidadizas excepciones que nos pueden dar una línea de trazabilidad mucho más entera y fácil de entender si diseñamos bien el software, ¿pero es fácil no?

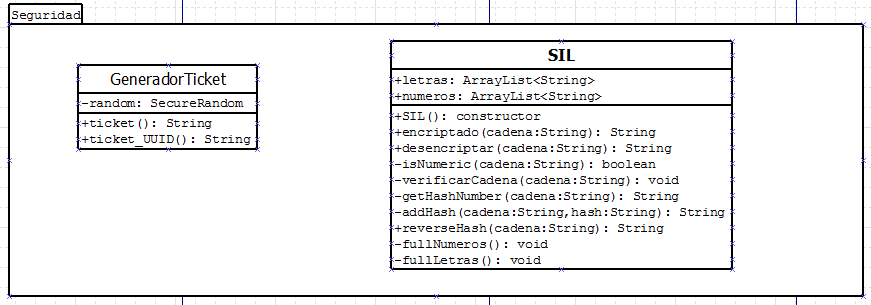
Si no me equivoco, mi experiencia me ha dicho que del 100% de software que he podido desarrollar y tocar con mi conocimiento, el **101%** no las maneja bien y lo más grave aún, **NO ES UN PROBLEMA RECONOCIDO E IDENTIFICADO EN LA APLICACIÓN.** (También cabe aclarar que yo también fallaba en estos problemas). Es por eso que las excepciones y errores en SPL tienen un tratamiento bastante explícito y muy intensivo, para tener un control más centralizado de los errores y saber en qué momentos suceden los problemas.

¿Fácil no?

### Perfecto! ¿Que aprendimos de nuevo?

* **Excepciones: Algo fuera de lo común en el flujo normal y correcto de nuestro algoritmo.**

## Seguridad



***Imagen 5. Paquete que contiene la seguridad en SPL.***

Bueno, yo y mi súper amigo volvemos de nuevo con las fastidiosas pero importantes definiciones, ahora vamos con Seguridad.

**Seguridad**: se puede referir a la ausencia de riesgo o a la confianza en algo o en alguien (Como definen de bonito).

Fácil y muy bonito, Seguridad = confianza, no hay nada más que tratar aquí, el SPL intenta ser un framework a nivel de seguridad un poco más robusto implementando criptografía propia y también un generador propio de token.

QUIETO!!! Donde esta Wikipedia y su término token???

Hay vamos.

**Token**: una unidad de valor la cual permite almacenar contraseñas y certificados y, además, llevar la identidad digital de la persona.

Excelente, un término muy directamente proporcional al de seguridad no?.

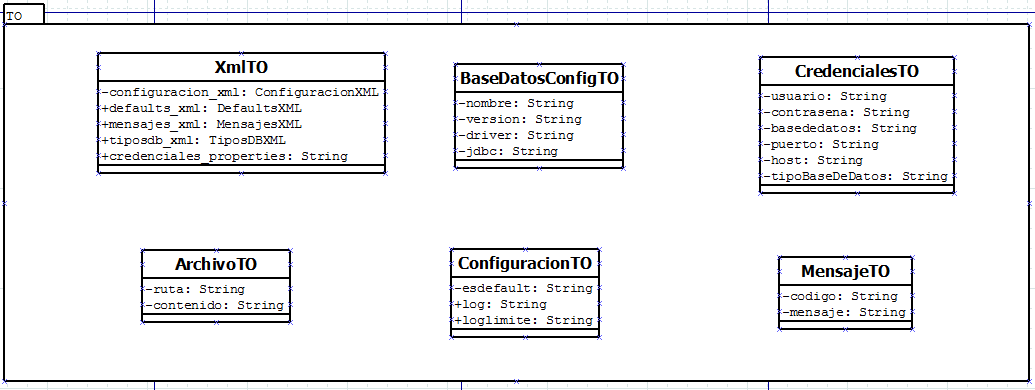
Si concluimos de una manera centralizada y sin sentimientos, nos damos cuenta que los **token** son dispositivos lógicos en nuestra arquitectura de seguridad, que nos da autenticidad entre más de un sistema con el que se quiera interactuar (el token, nos da el estado de nuestra identidad, y nos dice si “si se puede” o “no se puede” hacer una acción en nuestro sistema.

SPL nos provee API’s de seguridad para poder brindar una seguridad transparente a nuestros software construidos basado en el framework, partiendo de la programación orientada a objetos (**POO**) como su pilar más grande y fuerte.

### Perfecto! ¿Que aprendimos de nuevo?

* **Seguridad**: confianza.
* **Token**: una unidad de valor la cual permite llevar la identidad digital de la persona.

## TO



***Imagen 6. Transfer Objects necesarios para encapsular configuración de SPL***

Los transfer que ¿?

Ayúdame Wikipedia:

~~Data~~ **transfer object**: es un objeto que transporta ~~datos~~ (Información) entre procesos.

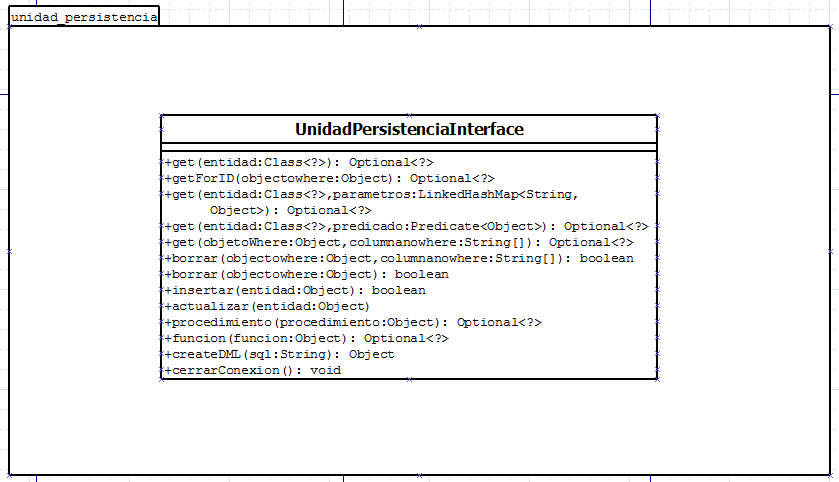
Hay pero que bonito, un objeto que transporta lo que yo quiera a nivel de datos o información en general, suena como importante no?¿

Así es amigo mío, los Transfer object son los objetos que encapsulan toda la información de valor que tenemos en SPL, nos sirven para acceder a la información de los archivos de SPL (**configuraciones.xml, mensajes.xml, defaults.xml, tipodb.xml, credenciales.properties**) y toda la información parametrizable del framework, la arquitectura del mismo(framework) está hecha para que todos los sistemas construidos con el framework crezcan es a nivel de **TO (**Objetos de transferencia como entidades (Clases que representan un objeto en la base de datos)) y la lógica del mismo siempre sea la misma como para los TO ya soportados como para los nuevos creados.

### Perfecto! ¿Que aprendimos de nuevo?

* **transfer object**: es un objeto que transporta Información entre procesos.
* **Entidades**: es la representación de un objeto o concepto del mundo real que se describe en una base de datos.

## Unidad persistencia



***Imagen 7. Paquete que encapsula toda la lógica fuertemente añadida del SPL.***

Excelente, llegamos a nuestra clase insignia, encontramos la unidad de persistencia.

Pero vamos a definir primero (Wikipedia) que nos dice sobre una unidad de persistencia.

**~~Unidad de persistencia~~**: Representa un conjunto de entidades que pueden ser mapeadas a una base de datos, así como la información necesaria para que la aplicación JPA pueda acceder a dicha base de datos.

**Unidad de persistencia (en SPL):** Representa un conjunto de métodos y rutinas que nos permite interactuar directamente con la base de datos.

Que diferencia vemos?¿

**Diferencia:**

Para SPL la unidad de persistencia **no es un contenedor de entidades** sino el **canal** por el cual nos comunicamos con la base de datos.

**Compartimos:**

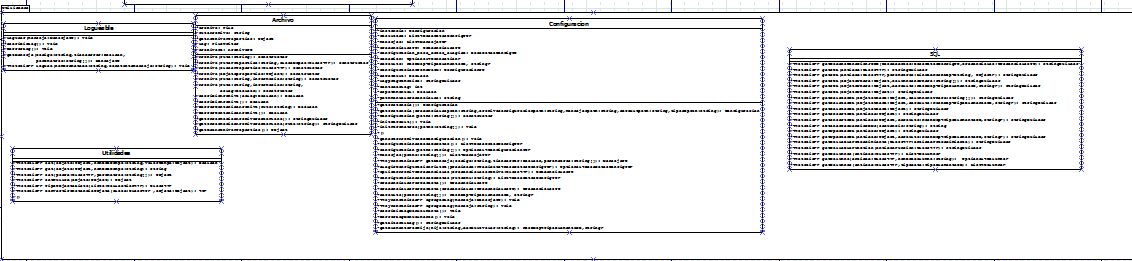
Contiene la información necesaria para acceder a la base de datos, tanto como para **JPA** (Java Persistence API) como para **SPL**.

Y si queremos concluir algo, podemos concluir que la **unidad de persistencia** en SPL, tiene la tarea de comunicarse de manera íntegra y dinámica con la base de datos, nos provee herramientas para poder interactuar de N formas con la base de datos.

### Perfecto! ¿Que aprendimos de nuevo?

* **Unidad de persistencia en JPA**: Representa un conjunto de entidades que pueden ser mapeadas a una base de datos, así como la información necesaria para que la aplicación JPA pueda acceder a dicha base de datos.
* **Unidad de persistencia en SPL:** Representa un conjunto de métodos y rutinas que nos permite interactuar directamente con la base de datos.
* **JPA:** un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard Y enterprise.
* **API:** Es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.

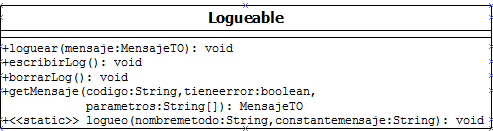
## Utilidades



***Imagen 8. Clases (Utilidades, Logueable, Archivo, Configuracion, SQL).***

El paquete utilidades es uno de los más concurrentes y créanme que mas importantes en SPL, en el paquete de utilidades encontramos la mayoría de la arquitectura robusta del SPL, vamos a verla (Cara de emocionado).

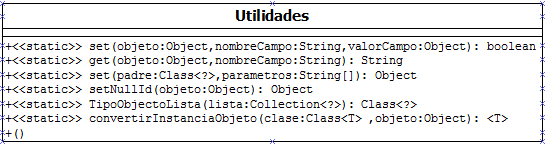
**Logueable:** Esta clase abstracta nos permite darle el comportamiento de Logueable a todas las clases que hereden de ella.



**Imagen 9. *Diagrama de clase de Logueable***

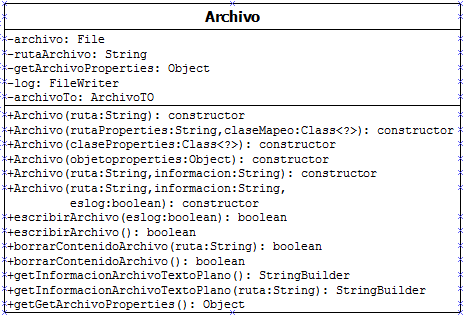


**Utilidades:** Esta clase nos expone la mayoría de métodos más recurrentes del SPL, es decir, en utilidades encontraremos la mayoría de métodos introspectivos (**java.lang.reflect)** que nos genera dinamismo introspectivo a nuestros resultados de la base de datos.



***Imagen 10. Diagrama de clase de Utilidades.***

**Archivo:** la clase archivo nos da todas las herramientas necesarias para realizar un tratamiento óptimo de los archivos en SPL (Y cualquier archivo accesible por JAVA). (Abrir, cerrar, escribir, eliminar).

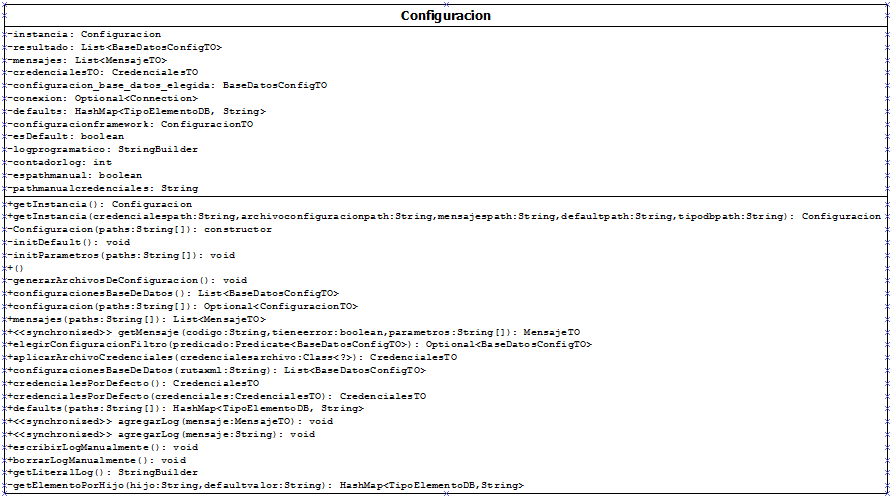


***Imagen 11. Diagrama de clases de la clase Archivo.***

**Configuracion:** debe ser la clase más importante del SPL, ya que ella guarda el estado inicial de la configuración del SPL de manera dinámica, y me permite acceder en cualquier momento de la aplicación a las configuraciones planteadas desde un inicio en SPL (información de archivos, flags)

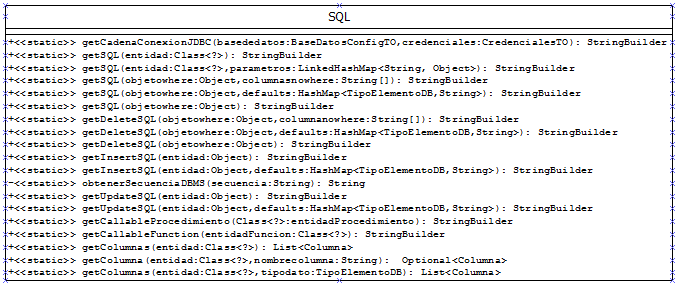
Este patrón de diseño es conocido como **Singleton (instancia única de una clase).**

***Nota:*** *cabe recordar que tuve en cuenta tanto las ventajas como desventajas de este patrón y se tomaron las medidas necesarias para que concurrentemente funcione correctamente.*



***Imagen 12. Diagrama de clases de Configuracion.***

**SQL:** La clase SQL es la encargada de construir introspectivamente el SQL basado en modelos y estructura de datos en JAVA (TO).



***Imagen 13. Diagrama de clases de SQL.***

### Perfecto! ¿Que aprendimos de nuevo?

* **Concurrencia:** es una propiedad de los sistemas en la cual los procesos de un cómputo se hacen simultáneamente, y pueden interactuar entre ellos.
* **Introspección:** capacidad de algunos [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) [orientado a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos) de determinar el tipo de un [objeto](https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)) en [tiempo de ejecución](https://es.wikipedia.org/wiki/Tiempo_de_ejecuci%C3%B3n).
* **Clase abstracta:** Este Tipo de Clases nos permiten crear “método generales”.
* **Java.lang.reflect:** Paquete en java que provee clases e interfaces que obtienen en tiempo de ejecución características propias de los objetos.
* **Singleton:** instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

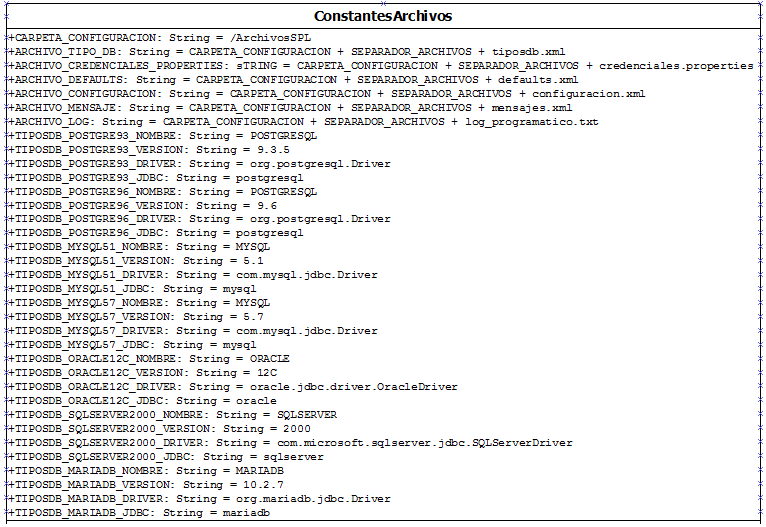
## Constantes

Las constantes es un recurso demasiado importante en nuestras aplicaciones, ya que declara recursos que **NO** van a cambiar en la misma, pero, porque no dejamos que mi amigo (tengo que decirlo (WIKIPEDIA)) me diga que es una constante.

**Constante:** Es un valor que no puede ser alterado/modificado durante la ejecución de un programa, únicamente puede ser leído.

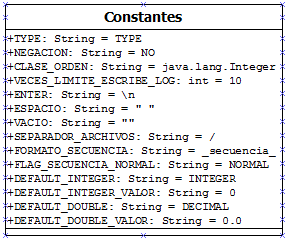
Muy bonita descripción y clara, nuestras constantes en SPL se encuentran muy bien discretas (es decir que cada sección de constantes tienen bien definidos sus límites a nivel de valores (que valores debe tomar y cuales otros no)), lo que nos da un entendimiento mucho mejor definido y claro, veremos que significa todo esto.

**ConstantesArchivos:** Esta clase contiene todas las constantes relacionadas con los archivos de configuración del SPL (Path de configuración, Nombre de archivos, Constantes de hash del sistema… etc).



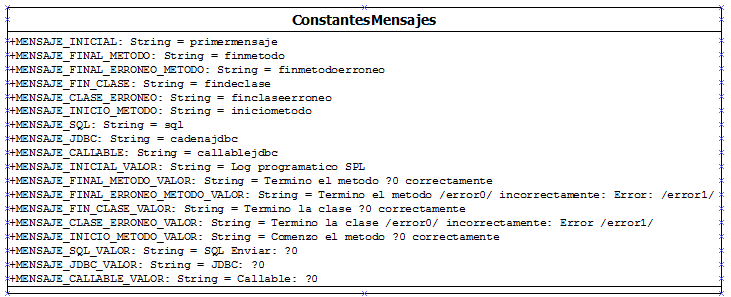
***Imagen 14. Diagrama de clases Constantes archivos.***

**Constantes:** Contiene las constantes que se usan lógicamente (en los algoritmos) en SPL.



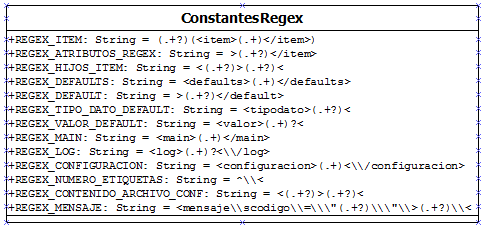
***Imagen 15. Diagrama de clases Constantes lógicas del sistema***

**ConstantesMensajes:** Esta clase contiene las constantes relacionadas con el archivo **mensajes.xml.**



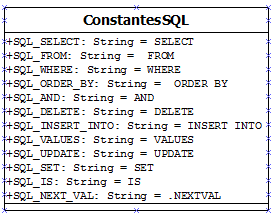
***Imagen 16. Diagrama de clases de ConstantesMensajes.***

**ConstantesRegex:** Clase que contiene todas las expresiones regulares usadas en la aplicación.



***Imagen 17. Diagrama de clases de ConstantesRegex.***

**ConstantesSQL:** Clase que contiene todas las constantes SQL utilizadas para construir el SQL final.

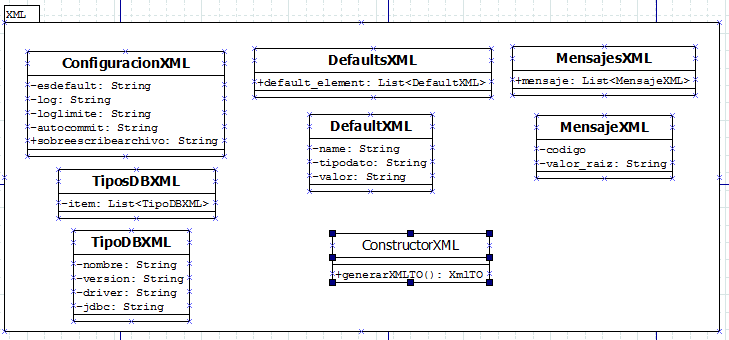


***Imagen 18. Diagrama de clases de ConstantesSQL.***

### Perfecto! ¿Que aprendimos de nuevo?

* **Constante:** Es un valor que no puede ser alterado/modificado durante la ejecución de un programa, únicamente puede ser leído.
* **Variables discretas:** es una variable que no puede tomar algunos valores dentro de un mínimo conjunto numerable, quiere decir, no acepta cualquier valor, sea programáticamente o contextualmente.
* **Expresiones regulares:** es una secuencia de [caracteres](https://es.wikipedia.org/wiki/Car%C3%A1cter_(tipo_de_dato)) que forma un patrón de búsqueda, principalmente utilizada para la [búsqueda de patrones](https://es.wikipedia.org/wiki/B%C3%BAsqueda_de_patrones) de cadenas de caracteres u operaciones de sustituciones.
* **SQL:** es un lenguaje específico del dominio que da acceso a un sistema de gestión de bases de datos relacionales que permite especificar diversos tipos de operaciones en ellos.

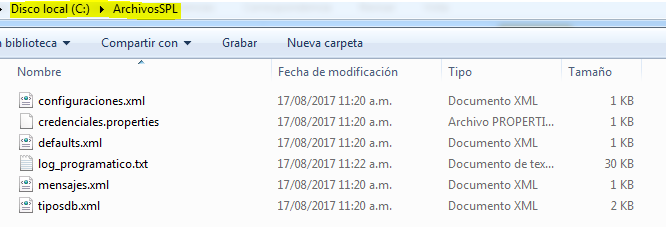
## XML



***Imagen 19. Diagrama de clases paquete XML.***

El paquete XML es el puente que existe entre SPL y su sistema de archivos **/ArchivosSPL** la carpeta que se crea en la raíz de sistema de archivos **Usualmente es la unidad “C:”** gracias a esta carpeta, tenemos acceso a los diferentes archivos de configuracion que tenemos en SPL (vamos no seas flojo y explícanoslo Alejo), vamos a explicar (☺) la estructura de los archivos y sus funcionalidades.

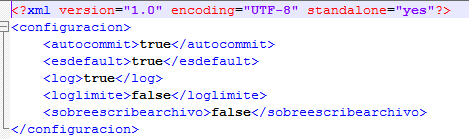
### SPL sistema de archivos.



**Imagen 20. Sistema de archivos SPL.**

El sistema de archivos del SPL se va crear por default en **C://ArchivosSPL//(configuraciones|defaults|mensajes|tiposdb).xml&log\_programatico.txt|credenciales.properties**

#### Configuraciones.xml



***Imagen 21. Contenido archivo configuraciones.xml.***

El archivo configuraciones tiene **5** configuraciones principales parametrizables que vamos a explicar:

**Autocommit (**true o false**):** este flag nos da la opción de prender (true) o apagar (false) el autocommit en las transacciones de la base de datos.

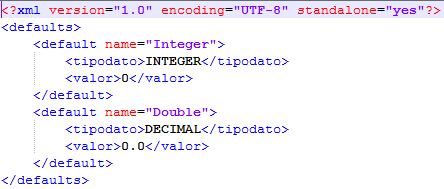
**Esdefault (**true o false**):** este flag nos indica si se van a usar la configuración del archivo **defaults.xml.**

**Log(**true o false**):** este flag nos indica si SPL escribe o no en el archivo **log\_programatico.txt**.

**Loglimite(**n > 0 & n <= 10000 o false**):** este flag nos indica si existe un límite de líneas a escribir en el log(n > 0 & n < 10000) donde n es un número real mayor a **0** y menor o igual a **10000.**

**Sobreescribearchivo(**true o false**):** este flag nos indica si los archivos van a sobrescribirse con la configuración default cada vez que se ejecute el SPL(true) o solo el archivo **log\_programatico.txt** (false).

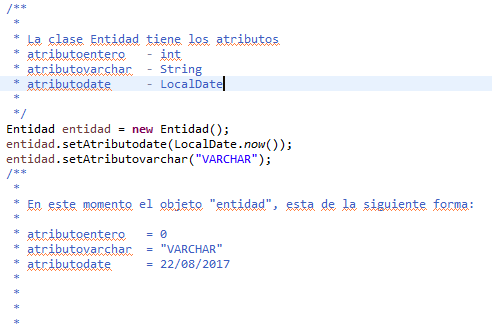
#### Defaults.xml



***Imagen 23. Contenido archivo defaults.xml.***

El archivo **defaults.xml** es muy importante cuando hablamos de la introspección de lenguajes y componentes del lenguaje, para SPL el archivo defaults nos sirve para realizar una introspección más precisa y un poco más parametrizable. Realmente esto es claro solamente con un ejemplo.

**Ejemplo:**



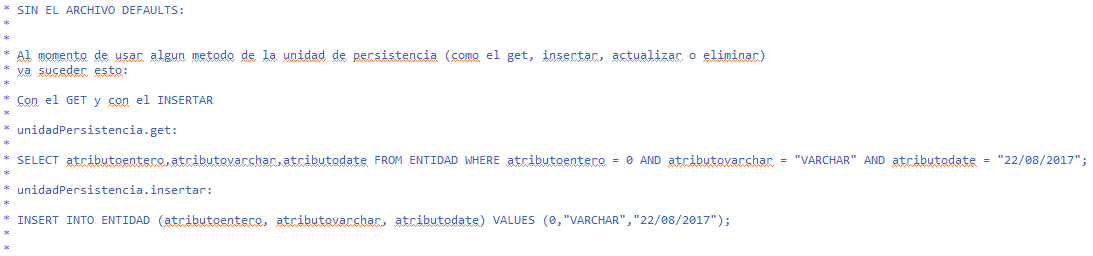
***Imagen 24. Creación del objeto Entidad.***

Como podemos ver en este instante el objeto entidad se encuentra “lleno”, aunque solo hayamos asignado valor a las variables “atributovarchar” y “atributodate”, el “atributoentero” es un **int,** esto significa que no puede obtener el valor **nulo** por ser un objeto **primitivo.**

Por ende viéndolo de la forma SPL, el “default” para el **int** y para **doublé** son **0** y **0.0** respectivamente, por ende parametrizamos en nuestro archivo **defaults.xml** si queremos que este “**0**” para el **int** y este “**0.0**” para el **doublé** sean considerados como **nulos** en nuestro SPL.

Lindo no? Primitivos nulos? Estos manes…

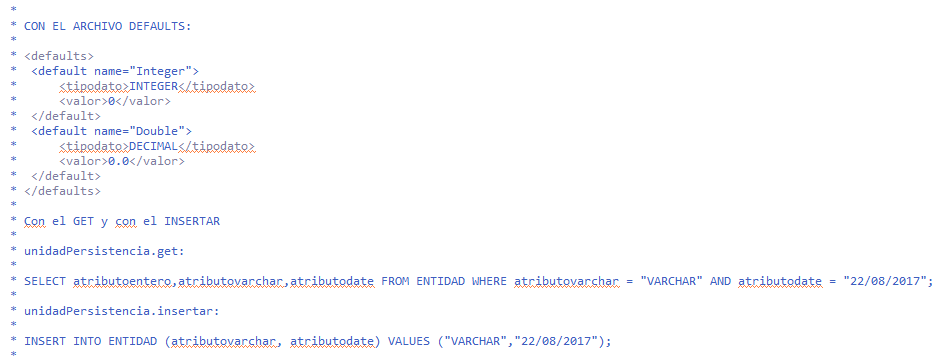
Ahora, vamos a ver un ejemplo de cómo sería una “introspección SPL” sin el archivo **defaults.xml**



***Imagen 25. Ejemplo sin el archivo defaults.xml***

Vamos a explicarlo para que sea más simple, sin el archivo **defaults.xml**  el **SELECT** que resulto fue un select que tuvo en cuenta **todos los atributos** que no fueran considerados como nulos, por eso tenemos el **WHERE atributoentero = 0**

En el Insert vemos como incluye la columna **atributoentero** también en el **INSERT.**



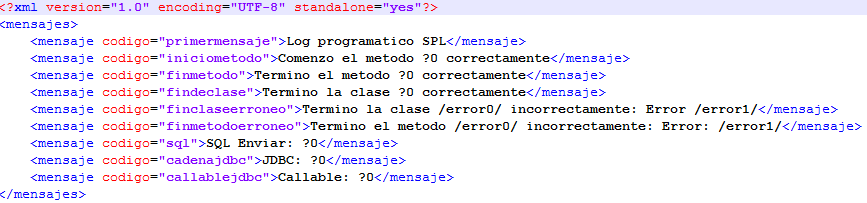
***Imagen 26. Ejemplo con el archivo defaults.xml***

Ahora simplemente con el archivo **defaults.xml** tenemos que nuestro **SELECT** va seguir funcionando de la misma manera, es decir, los atributos nulos van a seguir tomándose en la introspección PERO en este caso el **0** del **int** es considerado como **nulo**, por eso el atributo

**atributoentero = 0** es considerado como nulo y el **SELECT** no lo toma en la cláusula **WHERE** pero si en la **SELECT (select atributoentero FROM ENTIDAD (SIN WHERE)).**

En el insert pasa igual, ahora como se consideran que todos los **int** que valgan **0** van a ser nulos, no lo incluye ni el el **INSERT** ni en el **VALUES.**

#### MensajesXML



***Imagen 27. Contenido archivo mensajes.xml.***

El archivo mensajes contiene (y tentativamente queremos que CONTENGA) todos los mensajes de la aplicación. Aquí encontramos algunos mensajes por default que ya trae SPL para tipar los mensajes propios del framework.

Vamos a ver algunas particularidades de los flags de los mensajes para este archivo.

|  |  |
| --- | --- |
| Flag | Valor |
| *?0* | Identificador para reemplazar mensaje donde **0** es un número que aumenta (Ejemplo)  El método ***?0*** *termino de manera correcta en la clase* ***?1*** |
| /error0/ | Identificador para reemplazar mensaje donde **0** es un número que aumenta (Ejemplo)  *Termino la clase* **/error0/** incorrectamente: Error **/error1/** |

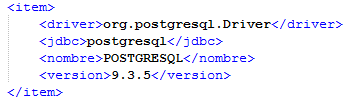
***Tabla 1. Flags archivo mensajes.***

#### TiposDB.xml



***Imagen 28. Tipo de base de datos relacionadas con sus componentes de conexión lógicos (driver, jdbc, etc…).***

El archivo tiposdb.xml contiene las bases de datos con las que va interactuar o con las que puede interactuar nuestro SPL, explicamos la estructura de cada elemento.



***Imagen 29. Item configuración base de datos.***

**Driver:** contiene el paquete donde se encuentra la clase “driver” para conectarnos a la base de datos.

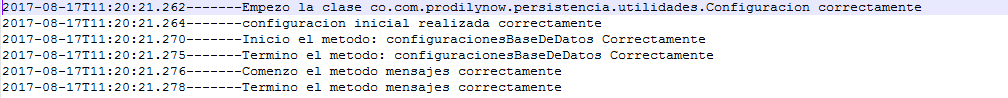
**Jdbc:** nombre clave que se le da a la base de datos en java (este lo provee cada fabricante del driver)

**Nombre:** nombre lógico que le queremos dar en SPL.

**Versión:** versión numérica separada por puntos donde se registra la versión de dicho driver (normalmente concuerda con la versión de la base de datos).

Las bases de datos que no se encuentren registradas no van a poder ser consumidas por el SPL.

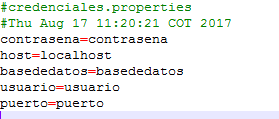
#### log\_programatico.txt



***Imagen 30. Contenido archivo log\_programatico.txt.***

En este archivo se registran por todos los métodos donde se esté logueando en el SPL, este archivo es plano y contiene fecha y hora de las consultas a los métodos, bueno no?

#### Credenciales.properties



***Imagen 31. Contenido archivo credenciales.properties.***

El archivo credenciales.properties nos sirve para poder ingresar los parámetros de conexión de la base de datos de manera manual y así utilizar toda el API SPL para gestionar la conexión.

Sus flags principales son:

**Contraseña:** Aquí tendremos nuestra clave de acceso.

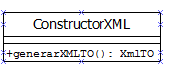
**Host**: Dirección ip o DNS donde se encuentra nuestra BD.

**Basededatos**: Nombre de la base de datos a conectar.

**Usuario:** usuario para ingresar a la base de datos.

**Puerto:** Puerto por donde ingresa la base de datos.

### Constructor XML



***Imagen 32. Constructor XML.***

La clase constructor XML nos brinda la función de generar nuestros archivos planos, esta clase tiene la funcionalidad de generar la configuración **inicial** de nuestro framework.

### Perfecto! ¿Que aprendimos de nuevo?

* **XML:** es un meta-lenguaje que permite definir lenguajes de marcas utilizado para almacenar datos en forma legible.
* **Commit:** Confirmación de una transacción.
* **Log:** se usa el término log, historial de log o registro a la grabación secuencial en un archivo o en una base de datos de todos los acontecimientos (eventos o acciones) que afectan a un proceso particular (aplicación, actividad de una red informática, etc.). (Waooo, me gustan esas definiciones!).
* **Default:** es una palabra de origen anglosajón que literalmente quiere decir o alude a hacer algo por defecto o de manera predeterminada.
* **Nulo:** Inexistente, que no ocurre.
* **Driver:** controlador de dispositivo, software realizado para controlar o simplemente conectarse un dispositivo ajeno.
* **Flag:** se refiere a uno o más bits que se utilizan para almacenar un valor binario o código que tiene asignado un significado.

# ¿Cómo usar SPL?

Todo es hermoso y aprendimos seguramente muchas cosas aquí en la sección llamada **Diagrama de clases.**

Ahora lo más importante es muy simple pero es lo más importante, como vamos a usar SPL?

Vamos a aclarar algo muy funcional, aquí vamos a explicar y a tener muy en cuenta cómo usar SPL con sus funcionalidades insignia, luego vamos bajando por las opciones de manera jerárquica, hasta llegar a la más irrelevante (PERO TODAS SON FUNCIONALES), con esto quiero que aprendas como usar el framework bien (Cuando hay caritas felices) y cuando entras en pánico (stackoverflow en chino).

## Dependencias directas

Para explicar paso a paso las funcionalidades del SPL, vamos a explicar paso a paso de manera simple y verbal las principales funcionalidades de SPL.

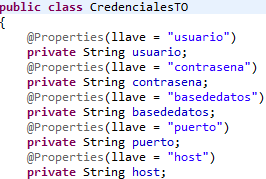
* Lo primero que debemos obtener es una conexión, pero para una conexión es necesario tener unas credenciales de autenticación necesarias para acceder a la base de datos, en este orden de cosas la primera dependencia antes de conseguir la conexión son las **CREDENCIALES.**

### Credenciales

En SPL las credenciales se manejan mediante el archivo **credenciales.properties**, puedes llenarlo a mano (editando el archivo manualmente) o simplemente lo llenas programáticamente (esta es chévere), vamos a ver.

Para esto vamos a usar el objeto **CredencialesTO** que contiene los mismos atributos que el archivo **credenciales.properties.**

Vamos a ver la clase **CredencialesTO**

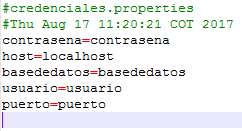


***Imagen 33. Credenciales transfer object JAVA.***

Como podemos observar nuestra clase tiene **5** atributos

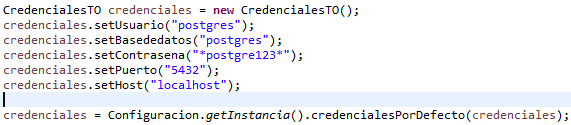
* Usuario
* Contraseña
* Basededatos
* Puerto
* Host

Ahora vamos a ver nuestro archivo **credenciales.properties.**



***Imagen 34. Contenido Credenciales.properties.***

Como podemos observar, el archivo **credenciales.properties** y la clase **CredencialesTO** tienen la misma estructura en que se guarda la información, por ende al ver esto, puedo afirmarte que el archivo **credenciales.properties** se “llenado” o “construido” con el objeto **CredencialesTO. ¿**Quieres ver?



***Imagen 35. Como llenar el archivo credenciales.properties.***

¿Que vemos aquí?

* El objeto **CredencialesTO** llenándose con unas credenciales (en este caso es una conexión a postgreSQL)
* Luego referenciamos el objeto consumiendo un método de nuestro **Singleton de configuración** llamado **credencialesPorDefecto.**
* El objeto nos retorna la información del archivo **credenciales.properties.**

Fácil o no compañeros? Ya sabemos cómo contextualizar nuestras credenciales en primera instancia.

Qué tal si vamos a nuestro siguiente nivel, los **tipos de base de datos.**

### Tipo de base de datos

Otro de los Plus de nuestro hermoso framework, recuerdas más arriba que te nombre los diferentes archivos que manejamos en SPL. Vamos a recordarlos.

* **Configuraciones.xml:** Nos sirve para las configuraciones globales o generales en el SPL.
* **Credenciales.properties:** nos ayuda a tener centralizadas nuestras credenciales
* **Defaults.xml:** manera de manejar los valores **nulos** en los primitivos.
* **Log\_programatico.txt:** escribe el paso a paso de SPL (donde fue, donde rompió, que género, en general, los eventos).
* **Mensajes.xml:** forma de manejar los mensajes el SPL.
* **Tiposdb.xml:** Tipos de base de datos que soporta el SPL.

Simple, a simple descarte logramos reconocer que el archivo que usa esta funcionalidad es llamado **tiposdb.xml**, pe pe pero como lo supiste? Vamos a verlo de nuevo.

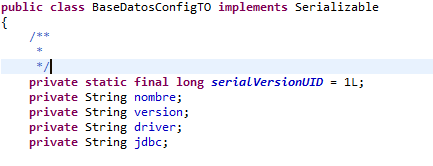


***Imagen 36. Tipos de base de datos soportada por SPL (contenido archivo tiposdb.xml).***

Como podemos ver, encontramos la simple configuración de una base de datos, vamos a ver qué elementos contrae un **ítem.**

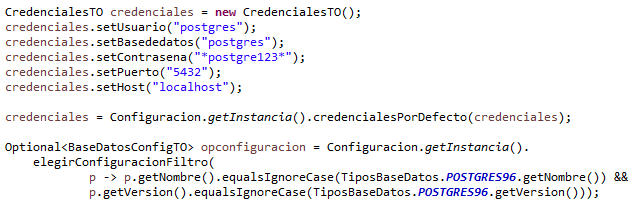
* **Driver:** clase que nos sirve para conectar nuestro **dbms** con nuestro lenguaje de alto nivel (en este caso hablamos de JAVA).
* **JDBC:** flag JAVA para reconocer lógicamente entre DBMS.
* **Nombre:** Nombre lógico asignado en SPL a el DBMS.
* **Versión:** Versión de nuestro DBMS.

Esto ya me suena parecido, en credenciales también teníamos un archivo, si esto funciona de la misma manera, deberíamos tener un **TO** que contenga la misma estructura de ¿nuestro archivo? Oye y si, miremos.



***Imagen 37. Estructura TO tiposdb.xml.***

Hay pero que beshesa (belleza en tono alegre), todo funciona casi de la misma forma, es decir, también tendremos que llenar un objeto para poder obtener nuestro DBMS a utilizar? Mmm, mejor miremos.



***Imagen 38. Obtener configuración del archivo tiposdb.xml.***

Pero qué clase de JAVA es este (->) ¿flechas? Miércoles, a menos de que sea un corrimiento de bytes, y no creo… VAMOS con un nuevo termino (si,si siiii), **las expresiones lambda.**

Lo siento mucho con ustedes, pero yo si confió en Wikipedia, hay vamos una vez más.

**Expresiones lambda**: es una [subrutina](https://es.wikipedia.org/wiki/Subrutina) definida que no está [enlazada](https://es.wikipedia.org/wiki/Enlace_simb%C3%B3lico) a un [identificador](https://es.wikipedia.org/wiki/Identificador).

(Para serles sincero, yo tampoco entendí)

Dividamos nuestra oración en partes, la primera parte a tratar es **subrutina**

**Subrutina:** Se denomina un subalgoritmo que sirve como insumo para un algoritmo padre.

Ahora sigamos con identificador:

**Identificador:** los identificadores son elementos textuales (también llamados símbolos) que nombran entidades del lenguaje. Algunas de las entidades que un identificador puede denotar son las variables, las constantes, los tipos de dato, las etiquetas, las subrutinas (procedimientos y funciones) y los paquetes.

Ósea es un subalgoritmo que no es ni una constante ni un tipo de dato ni una etiqueta ni una palabra reservada de nuestro lenguaje, pero que nos sirve para resolver un problema mayor? Esto me suena a otro concepto… Veámoslo!

Primero programación orientada a objetos y luego programación funcional… NO TE ASUSTES, probablemente ya lo has usado, vamos al concepto.

**Programación funcional:**

Es un [paradigma de programación](https://es.wikipedia.org/wiki/Paradigma_de_programaci%C3%B3n) [declarativa](https://es.wikipedia.org/wiki/Programaci%C3%B3n_declarativa) basado en el uso de [funciones matemáticas](https://es.wikipedia.org/wiki/Funci%C3%B3n_matem%C3%A1tica), en contraste con la [programación imperativa](https://es.wikipedia.org/wiki/Programaci%C3%B3n_imperativa), que enfatiza los cambios de estado mediante la mutación de variables. La programación funcional tiene sus raíces en el [cálculo lambda](https://es.wikipedia.org/wiki/C%C3%A1lculo_lambda), un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión.

Es decir, para concluir, una **expresión lambda** es un subprograma que va afuera de nuestros identificadores (palabras reservadas del sistema (JAVA)) que nos sirve para resolver varios problemas de manera general, solamente con una expresión, al carecer de transparencia referencial, nos ayuda a resolver N problemas con una misma expresión de manera transparente.

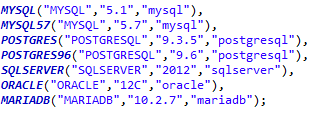
Es este caso, vamos a usar una expresión lambda para filtrar en nuestros **tipos de base de datos.**

Nuestro método de **configuración** llamado **elegirConfiguracionFiltro** nos recibe un objeto de tipo **Predicate** que nos sirve para filtrar mediante una expresión lambda, vamos a explicar el **Predicate** que enviamos como parámetro de nuestro método.



***Imagen 39. Primera expresión lambda.***

Aquí le estamos diciendo, tráeme de todos esos **dbms** uno que tenga como nombre el **enum TipoBaseDatos.POSTGRES96.NOMBRE**



***Imagen 40. Contenido ENUM TipoBaseDatos.***



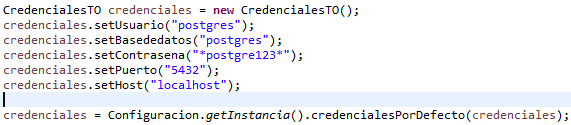
***Imagen 41. Unión expresiones Lambda.***

Aquí le estamos diciendo, tráeme de todos esos **dbms** del archivo **tiposdb.xml** el que tenga como nombre:

* **TipoBaseDatos.POSTGRES96.NOMBRE = POSTGRESQL**
* **&& (Y)**
* **TipoBaseDatos.POSTGRES96.VERSION = 9.6**

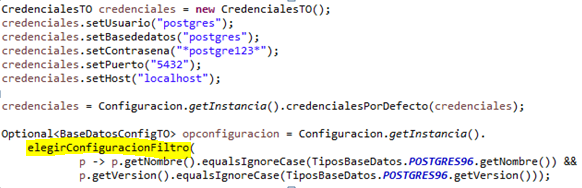
Ahora SPL ira y traerá del archivo **tipodb.xml** y filtrara los **dbms** que le indicaste en la expresión lambda.

En este orden de ideas compáñeros, ya sabemos como vamos a obtener las **Credenciales.**

******

***Imagen 42. Obtener credenciales.***

Obtener **Tipo de base de datos**



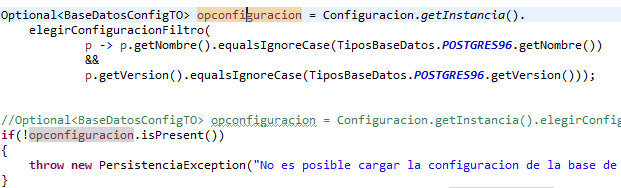
***Imagen 43. Obtener configuración base de datos.***

Ahora, vamos al final de nuestro ciclo completo.

Ya tenemos las credenciales de conexión, tenemos que tipo de base de datos vamos a conectar, YA NO NECESITAMOS MAS!!! Vamos a unir estas partes y a crear nuestra conexión, pero antes vamos a ver un término bastante importante que llevamos usando en esta contextualización, se llama el opcional.

El patrón **option** nos permite mostrar de manera explícita (mediante el sistema de tipos) la posibilidad de que un método pueda no devolver el valor deseado. Esto nos obliga a controlar la posible ausencia de valor de manera explícita, permitiéndonos elegir un valor alternativo en caso de dicha ausencia o simplemente realizar otra acción.

Es decir, ahora vamos a tener algo tipo.



***Imagen 44. Patrón Option en SPL.***

es simple compañeros, si nuestra expresión Lambda no retorna nada, no deberíamos tener un objeto **nulo** esto nos generaría muchos **NullPointerException.** Ahora con el patrón option implementado en SPL, si alguna consulta nos retorna **Null** con el método **isPresent** sabemos si viene o no **Nulo,** si viene nulo el método **isPresent** nos retorna un **false,** si existe algo diferente a **nulo** que se pueda usar, **isPresent** nos retorna true.

Como ya sabemos cómo funciona bien el patrón Option, vamos a seguir con la creación de nuestra **conexión.**



***Imagen 45. Crear conexión.***

¿Ya lo vemos?

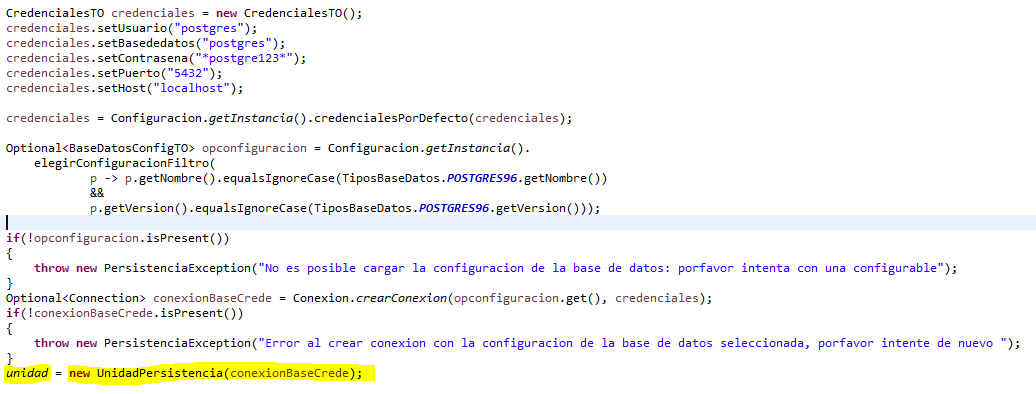
Para crear la conexión usamos el metodo **crearConexion** que nos recibe como parámetros, un objeto de tipo **BaseDatosConfigTO (Contiene la información del dbms a utilizar).**

Y como segundo parámetro un objeto **CredencialesTO,** la forma en que vamos a autenticar contra nuestra base de datos.

Vamos para nuestro último y más fácil paso, pero antes vamos a resumir un poquito que hemos visto.

* **Contextualizamos credenciales.**
* **Contextualizamos con expresiones lambda nuestro tipo de base de datos.**
* **Enviamos estos dos parámetros (TipoBaseDeDatos y credenciales) para crear nuestra conexión.**

Ahora si recordamos nuestro patrón de diseño de SPL, vemos que la clase encargada de realizar el dinamismo y la magia sobre nuestros objetos y nuestra base de datos se llama **Unidad de persistencia,** así que en el orden de las cosas, solo necesitamos contextualizar ahora nuestra unidad de persistencia.



***Imagen 46. Crear unidad de persistencia.***

Este es más simple aun amigos míos, la unidad de persistencia recibe en su constructor o un **Optional** con la conexión o un objeto **Connection** . Y ya, ESO ES TODO!! Lo logramos, ya lo tenemos, SPL funciona, funciona, FUNCIONA!!!

(tan chévere que funcione, pero si tan solo nos explicara para que sirven los métodos de la unidad de persi…) Vamos ahora a explicar los métodos de la unidad de persistencia para saber cómo interactuar con nuestra base de datos.

## Unidad de persistencia

Bueno, que bonito es llegar a un nivel de detalle en el que las cosas son entendibles, no es bonito?.

Antes de continuar con la práctica dinámica que llevamos, quiero hacerles una recomendación.

Los únicos que estamos aprendiendo de SPL aquí son **ustedes,** claramente mis conceptos referentes a SPL están claros y estoy plasmándolos (o intentándolo) de una manera clara, si en algún momento se tergiversa el conocimiento, te recomiendo que vuelvas y leas antes que lo sigas intentando, si al final no es totalmente claro, te recomiendo que me escribas, con mucho gusto yo puedo solucionarte las dudas técnicas referentes a SPL.

Y ya termine la charla mamona y de regaño, ahora si seguimos.

Como tenemos claro, nuestra **unidad de persistencia** en SPL es la clase que nos permite realizar distintos tipos de dinamismo con nuestra base de datos, es decir, es la clase que no permite **interactuar** con nuestra base de datos.

Vamos a empezar con los tipos de métodos que encontraremos en SPL y sus principales usos (wii).

### Métodos de consulta

Que es esto de “métodos de consulta”, lo siento pero yo soy muy conceptual y amigo del amigo de todos (WIKIPEDIA), definámoslo, déjenme definirlo:

**Métodos de consulta:** (La segunda cosa en el mundo que más me gusta respecto a las palabras es dividirlas) Empezaremos dividiendo nuestra palabra para darle una definición más clara y estrecha basada en nuestro contexto actual (Programación).

**Método:** es un conjunto de procedimientos, establecidos a partir de un enfoque, para determinar el programa de enseñanza, sus objetivos, sus contenidos, las técnicas de trabajo referentes.

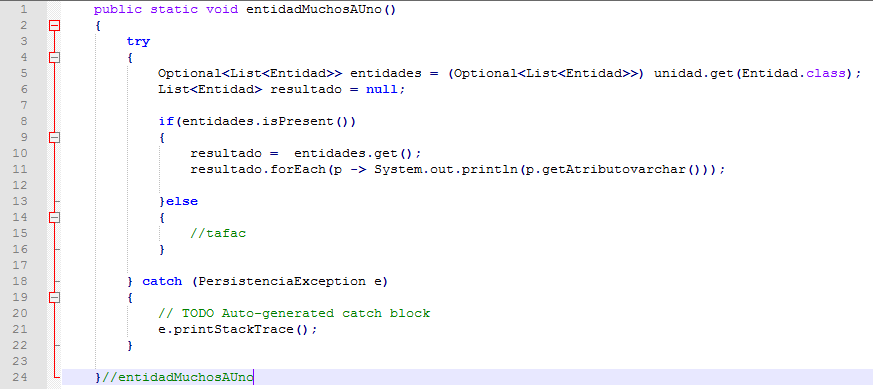
Excelente, yo pude concluir que un método es un conjunto de pasos que define el comportamiento de algo (ese algo en nuestro contexto, puede ser un **algoritmo**), entonces contextualizando, un **método de consulta** es un conjunto de pasos que define un **algoritmo** capaz de realizar **consultas** a la base de datos, no más alargue, vamos a verlos:

**Métodos “GET”.**



***Imagen 47. Método GET.***

Esta clase de métodos “GET” nos sirven para consultar entidades sobre referencia, es decir, solo debemos referenciar la clase a la cual queremos obtener los resultados, veamos un ejemplo más claro.



***Imagen 48. Ejemplo método “GET”.***

Simple, manejamos todos los conceptos actuales de arquitectura de SPL (**Patrón Option, Unidad de persistencia, Expresiones Lambda).**

En este ejemplo, le solicitamos a la unidad de persistencia su método **get** y por referencia pasamos la entidad a la que queremos referenciar (en este caso **Entidad**). Él nos retorna un opcional con una posible Lista de entidades consultadas en la base de datos. (**Linea 5)**

Desde la línea **8** hasta la línea **12** tenemos la validación de patrón opcional



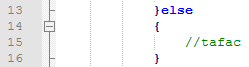
***Imagen 49. Patrón opcional.***

En palabras más naturales, preguntamos si la consulta contiene algo, entonces **obtenemos el valor de las entidades** y luego imprimimos un atributo de la entidad **Entidad.**



***Imagen 50. Obtener datos del opcional e imprimir un atributo de la entidad.***

Pero si no contiene nada, podemos controlarlo en un bloque simple **else.**



***Imagen 51. Si NO retorna nada nuestra consulta.***

Pero qué bonito, que bonito que bonito, es muy fácil de verdad, los métodos nos dan variedad de opcionas para usar, vamos a verlos.

**Get for ID:**



***Imagen 52. Entidad por ID.***

Quietos, súper mega quietos!!! “objetowhere”? QUE ES ESO???.

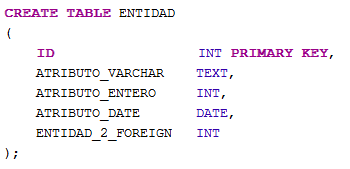
Llegamos a otro concepto del SPL que son los **objeto where.**

### Objeto where (Ejemplo mapeo)

Algo de esto habíamos explicado implícitamente arriba la explicación del paquete utilidades para ser exacta la clase **SQL.**

Vamos a explicarlo simple y con un ejemplo fácil, hagamos de cuenta que tenemos una entidad en nuestra base de datos de la siguiente forma.

**SQL:**

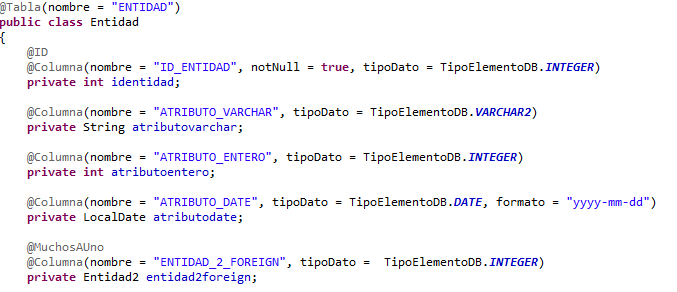


**Imagen 53. SQL tabla “Entidad” PGADMIN 4 PostgreSQL 9.6.**

Tenemos nuestra base de datos que tiene 5 atributos,

* Un identificador único: **ID.**
* Una columna literal (texto plano): **ATRIBUTO\_VARCHAR.**
* Una columna numérica: **ATRIBUTO\_ENTERO.**
* Una columna fecha: **ATRIBUTO\_DATE.**
* Una columna foránea: **ENTIDAD\_2\_FOREIGN.**

Tenemos una entidad que abarca muchos casos del comportamiento de una tabla en su **DDL (Data Definition Language),** por ende vamos a mapearla como una clase en JAVA para usarla a través de nuestro SPL.



***Imagen 54. Tabla “Entidad” mapeada como clase en JAVA.***

Aclaremos antes algo:

Como lo nombramos anteriormente SPL tiene algunos enum que nos sirven para poder realizar un mapeo correcto.

Para los datos enteros, tenemos los siguientes valores del enum **TipoElementoDB** para poder mapear en nuestra clase:

**Enteros:**







**Literales:**









**Fecha:**



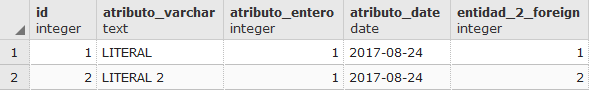
Ahora que tenemos nuestra tabla **(Entidad**), Nuestra clase (**Entidad.java**) y nuestra conexión a la base de datos (Nuestra unidad de persistencia lista).

Vamos a explicar el funcionamiento del objeto WHERE.

Si en nuestra base de datos encontramos el siguiente caso:



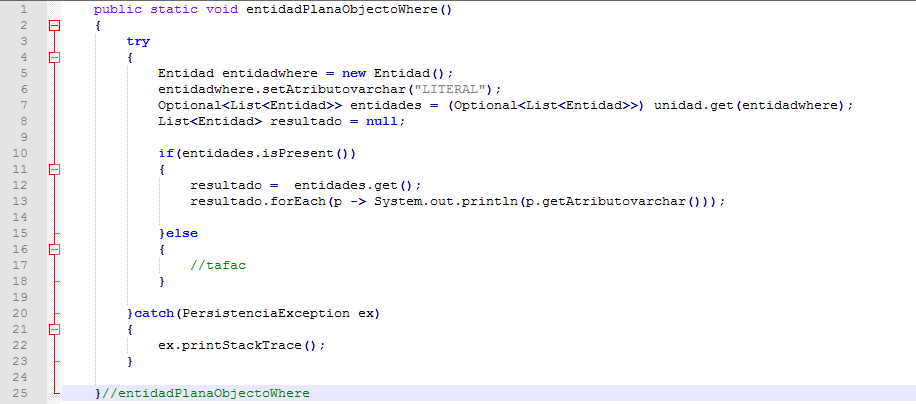
***Imagen 55. Select a la tabla “Entidad”.***



***Imagen 56. Resultados QUERY entidad.***

Como hacemos para poder consultar **MEDIANTE** SPL solamente el que tiene el que tiene el **atributo\_varchar** como “**LITERAL”.**

Simple, vámonos para JAVA.



***Imagen 57. Objeto where Entidad.***

Observamos como en la línea **5** a la **6,** creamos un objeto de tipo **Entidad** y le asignamos a su atributo **atributo\_varchar “Literal”.**

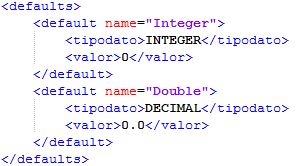
Excelente, antes de nada, vamos a entender como esta nuestro objeto where en este momento

**Entidadwhere:**

|  |  |  |
| --- | --- | --- |
| Columna | Tipo | Valor |
| ID | INT | 0 |
| ATRIBUTO\_VARCHAR | TEXT | “LITERAL” |
| ATRIBUTO\_ENTERO | INT | 0 |
| ATRIBUTO\_DATE | DATE | NULL |
| ENTIDAD\_2\_FOREIGN | INT | 0 |

***Tabla 2. Objeto where “entidadwhere”.***

Teniendo en cuenta nuestro archivo **defaults.xml,** vamos a verlo:



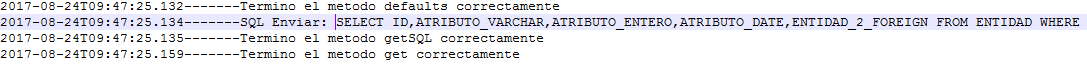
***Imagen 58. Archivo defaults.***

El archivo defaults nos dice que todos los **Integer** que tengan 0, van a ser considerados como **nulos** no igualables (No van en el WHERE) igualmente con los decimales.

En este orden de ideas, ejecutamos, y nos damos cuenta de que… NO TRAJO NADA.

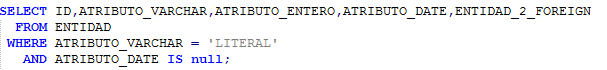
WTF.

Tranquilos, cuando suceda esto y estén interactuando con SPL, el tiene un archivo llamado **log\_programatico.txt** que te indica todo (hasta el SQL que arma dinámicamente a raíz de tus objetos WHERE o tus variables de referencia), entonces en este orden de ideas, vámonos para el log.

******

***Imagen 59. Log programático del objeto where “entidadwhere”.***

Cuando lo formateamos, encontramos la consulta de esta forma:



***Imagen 60. Consulta resultante del objeto “entidadwhere”.***





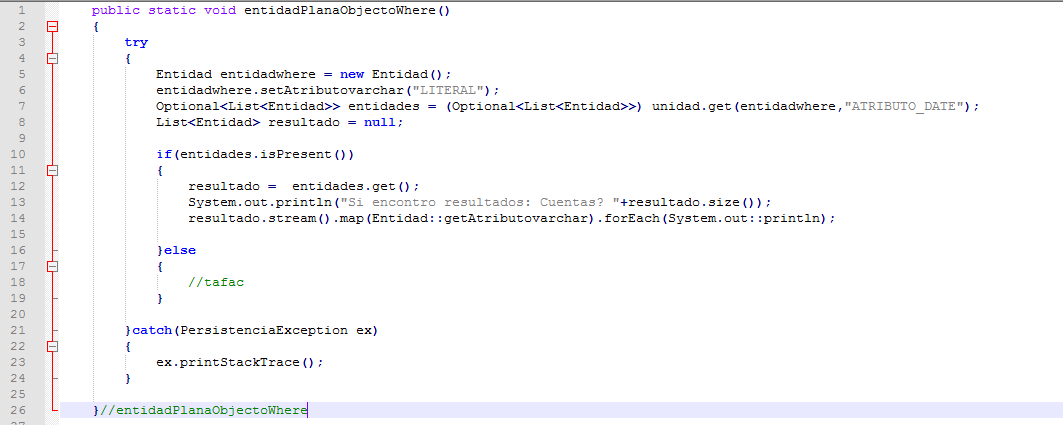


No puede ser… **atributo\_date** es nulo, como hacemos para que no iguale nuestro query a nulo?

Al ser un objeto where, él toma **todos los valores** que encuentre en el objeto y los iguala, pero tenemos un parámetro opcional de **todos** los métodos de consulta, se llama **columnanowhere.**

Si pasa como en este caso, donde necesitas solamente **igualar** una columna, y nos esta igualando algún **nulo** que no deseamos, debemos usar este parámetro, que nos recibe un **arreglo de lista de longitud variable** (**String…**) en donde podemos enviar todas las columnas que no “vayan” en nuestro **where.**

¿Un poco confuso? Apliquémoslo al ejemplo y entenderás.



***Imagen 60. Envió de parámetro columnanowhere en un método de consulta “get”.***

Simple y hermoso, solamente un **String** con el nombre de la columna y listo!

Nuestra consola nos retorna el **atributo\_varchar** y lo imprime.



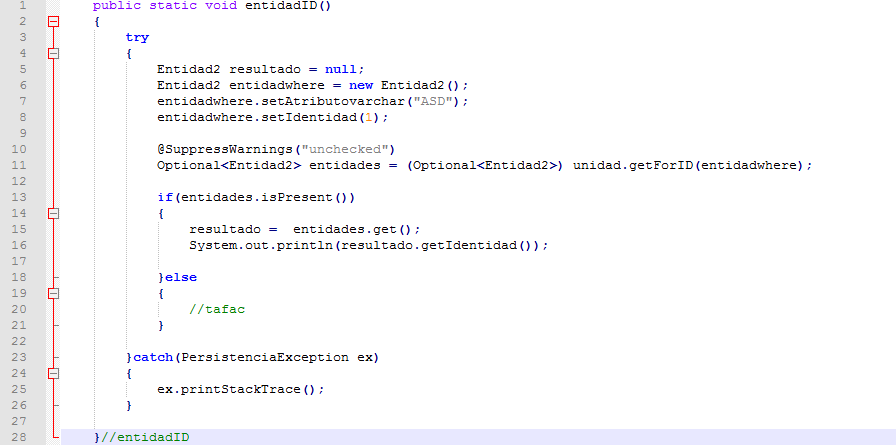
***Imagen 61. Impresión resultado final.***

Excelente, esto quiere decir que si retomamos a nuestro método **getForId**



***Imagen 62. GetForID.***

Pasamos nuestro objeto lleno y el solamente va tomar como valor único su ID.



***Imagen 63. Get for ID implementación.***

Esto quiere decir que el atributo de la linea **7,** que equivale a **atributo\_varchar** no será tenido en cuenta por la magia y así, solamente filtrar el objeto en la base de datos por su atributo **ID.**

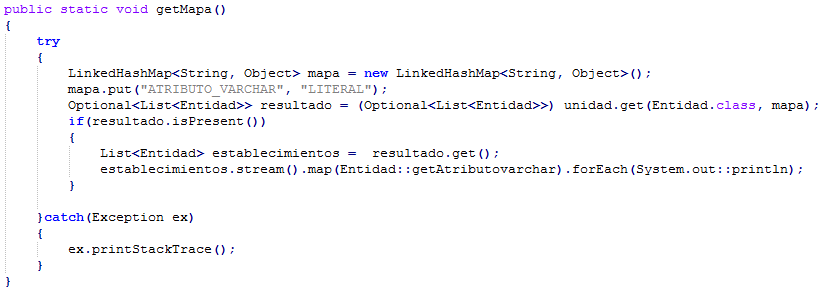
Súper, súper muy bueno, me parece a mí que vamos bien no les parece? Veamos otra variedad del método **get.**

**Get (con mapa)**



***Imagen 64. Método get con el parámetro LinkedHashMap.***

El método get con el atributo LinkedHashMap nos da la flexibilidad de guardar en una colección tipo “llave” -> “valor” los parámetros que queremos en nuestro WHERE, veámoslo en un ejemplo.



***Imagen 65. Método de consulta “get” implementando LinkedHashMap.***

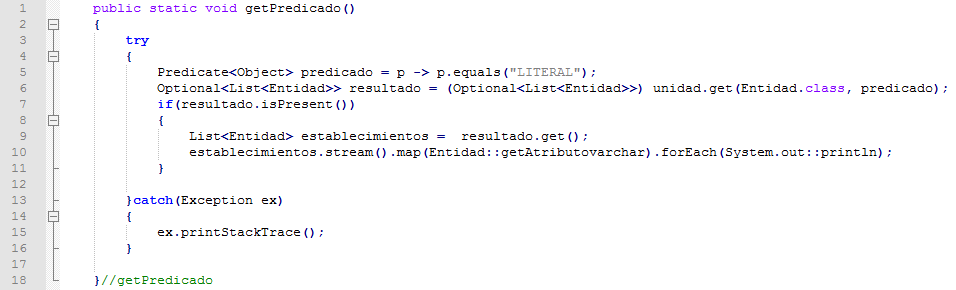
Como vemos en la imagen, usamos ya no el **objeto where** como insumo, sino una estructura de datos de tipo llave -> valor, donde las llaves son **los nombres de las columnas** y los valores son **los valores (:D).**

Vamos a nuestro próximo método.



***Imagen 66. Método de consulta “get” implementando el predicado.***

Anteriormente hablábamos del predicado como **una clase java** por la cual puedo usar las **expresiones lambda** para realizar un filtrado a una estructura de datos, en esta implementación, usamos el predicado para filtrar la lista resultante. Veamos el ejemplo.



***Imagen 67. Método de consulta get con objeto de filtrado Predicate.***

Solamente aquí, debemos tener cuidado construyendo el objeto Predicate, que es el que nos regala la magia final.

Lo logramos compañeros, nuestros métodos de consulta finalizan con esta última especificación.

¿Que sigue señor Alejo?

Métodos de manipulación de datos.

### Métodos de manipulación de datos.

Ya sabemos que un método es una serie de pasos que le dan estructura a algún tipo de definición (definimos que le daban orden en nuestro contexto, al algoritmo de búsqueda dinámica) , ahora vamos a definir que es la **manipulación de datos.**

Cuando nos referimos a **manipulación de datos** nos referimos a que podemos cambiar el estado de un dato (sea eliminado, actualizado o un dato nuevo) (**borrar, actualizar, insertar**).

Sin ir más, vamos a ver nuestros métodos.

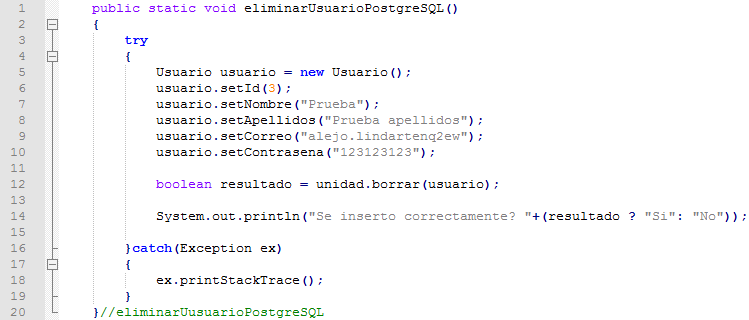
**Borrar:**



***Imagen 68. Método borrar unidad de persistencia.***

La implementación de borrar, funciona de la misma manera que el **get.**

Tenemos la opción de ingresar un **objetowhere** para su borrado y también las **columnanowhere,** que queremos que no vaya en nuestro script resultado, ahora, vamos a ver su **prueba unitaria** como hicimos con todos.



***Imagen 69. Borrar entidad mediante objeto where.***

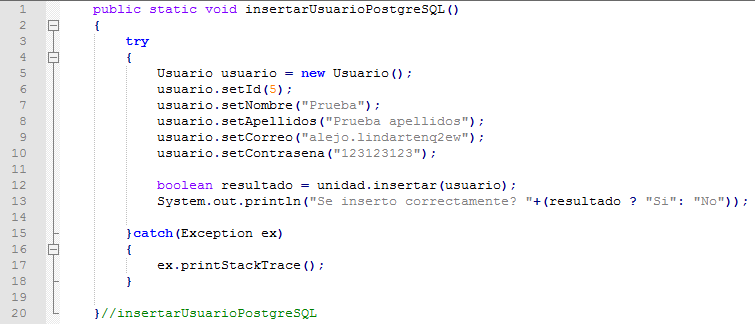
El método **borrar** de nuestra unidad de persistencia nos retorna un booleano (true si pudo, false si no).

**Insert.**



***Imagen 70. Insertar entidad mediante objeto where.***

Creo que sería muyyyyyy repetitivo si volvemos a explicar el objeto where, vamos a la **prueba unitaria.**



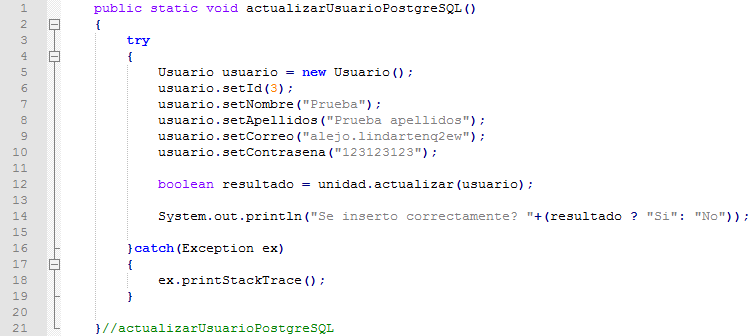
***Imagen 71. Prueba unitaria Insertar entidad objeto where SPL.***

El método **insertar** de nuestra unidad de persistencia nos retorna un booleano (true si pudo, false si no).

**Actualizar.**



**Imagen 72. Método Actualizar**



**Imagen 72. Prueba unitaria Actualizar entidad objeto where SPL.**

Ya se nos va acabar el SPL Alejo, queremos ver algo mas raro!!

Se les tiene.

**createDML.**

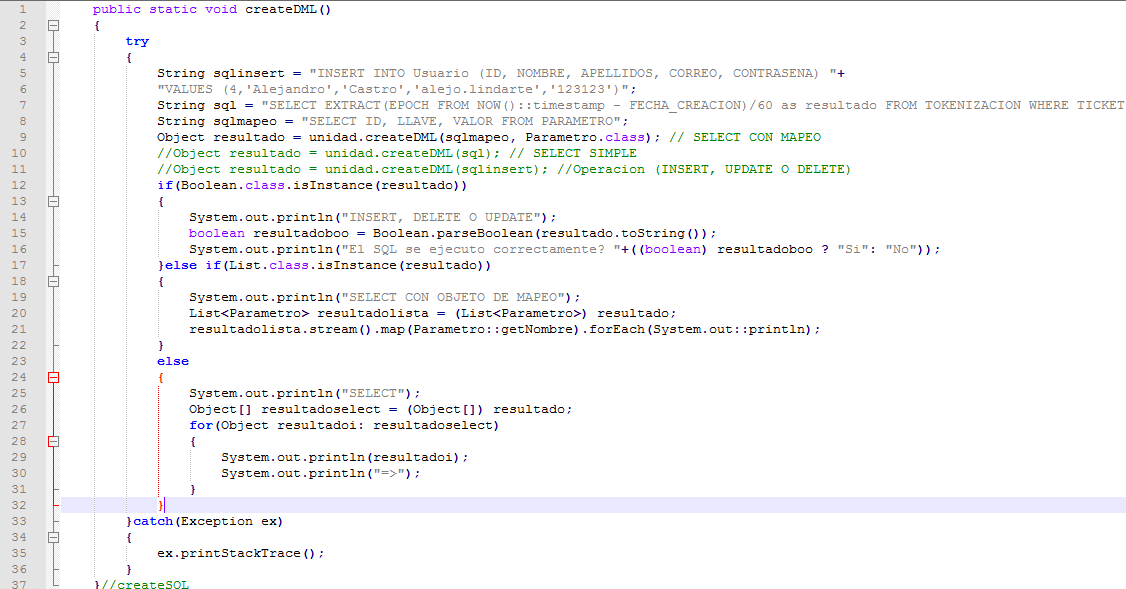
****

***Imagen 73. Crear DML función.***

Bastante pero bastante interesante la función **createDML,** esta función nos permite crear cualquier **SQL** sea **INSERT, UPDATE, DELETE** o un **SELECT.**

Esto nos da mucha flexibilidad, nos permite recrear un **SQL** sin necesidad de entidades, que bonito no?

Veamos la prueba unitaria e indagamos un poquito más:



***Imagen 74. Prueba unitaria createDML.***

Vamos a recrear las opciones que tiene nuestro método.

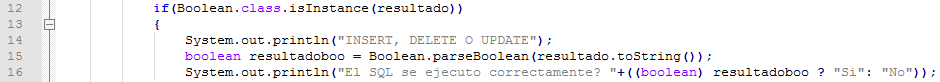
**Insert, Update o Delete.**

Como lo vemos en la línea **5.** Definimos un **INSERT.**



***Imagen 75. SQL Insert créateDML.***

**SI NUESTRO METODO CREATEDML** nos retorna un **BOOLEAN** (true o false), quiere decir que realizo un SQL como **insert, update o delete**.



***Imagen 76. Validación si es un Insert, Update o Delete.***

**Select con mapeo**

¿Qué es esto de **select con mapeo?**

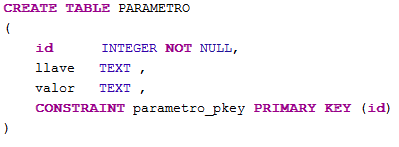
Nuestro SPL tiene la opción en nuestro método **createDML** de enviarle la consulta **SQL** y como segundo parámetro enviarle una **Entidad** a la cual debemos **mapear** su estructura resultado.

Veamos como esta en nuestro ejemplo:



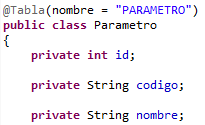
***Imagen 77. SQL con mapeo.***

Vamos a ver como esta nuestra tabla “parámetro” en la base de datos.



***Imagen 78. Tabla parámetro.***

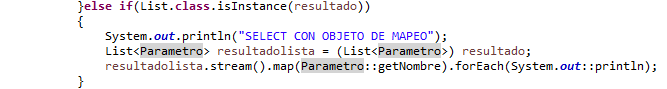
Ahora vamos a ver como se encuentra nuestro objeto **Parametro.java**



***Imagen 79. Entidad representativa de parámetro.***

Encontramos que nuestra **Entidad** representa de manera correcta a la Tabla **Parametro** en la base de datos, por ende podemos decirle a nuestro método **createDML** que los resultados de la consulta los mapee a objetos de este mismo tipo.

Ahora, el retorno. Cuando la respuesta de nuestro método **createDML** nos retorna la **Instancia de una lista**, quiere decir que es una consulta con un resultado de mapeo previo. Veamoslo.



***Imagen 80. Validación si es una entidad de mapeo.***

Simplemente hermoso, no hay nada que decir, solo nos falta un caso y terminamos:

**SELECT sin mapeo.**

Un select sin mapeo es un SQL bruto o plano, es decir, puede tener lo que tú quieras, desde sumar dos números “**SELECT 1 + 2”** o lo que necesites, vamos a ver el ejemplo.



***Imagen 81. SQL sin mapeo.***

Este SQL es un resultado de una operación entre fechas, y el método es capaz de ejecutarla de manera transparente.



***Imagen 82. Ejecución SQL sin mapeo.***

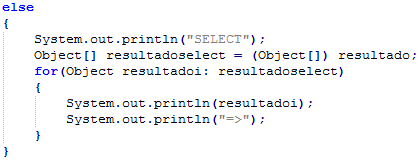
Ahora, como nos retorna este objeto si no es un **boolean** para los **INSERT, UPDATE, DELETE** o una lista si es un **SELECT CON MAPEO.** Este método nos devuelve una **lista de objetos (Object) DONDE:**

Las posiciones **PARES (numero % 2 = 0)** contiene los nombres de la columna que solicitaste (en este caso es **resultado:**



Y las posiciones **IMPARES (numero % 2 ¡= 0)** contiene los valores de la columna que solicitaste(en este caso el valor de **resultado**

Entonces nuestro resultado sería un **arreglo de 2 posiciones** la primera posición contiene el nombre **resultado** y la segunda posición contiene el **valor de la consulta**, veámoslo.



***Imagen 83. Validación select sin mapeo.***

Aquí podemos ver que como se debe tratar un objeto sin mapeo cuando es un **query.**

Aquí terminamos nuestros amados **métodos de manipulación,** métodos que nos permitían realizar cambios de estado (**INSERT, UPDATE O DELETE**) O métodos dinámicos híbridos que nos permiten ejecutar cualquier cosa **DML** en nuestro **dbms** (**createDML**).

Faltan exactamente 2 métodos.

### Métodos de consultada a objetos complejos en la base de datos (procedimientos y/o funciones).

Y que dijeron? No vamos a poder llamar procedimientos y funciones, pero CLARO!

Empecemos con las **funciones. (**Déjenme voy y busco a WIKIPEDIA)

**Funciones:** conjunto de líneas de código que realizan una tarea específica y puede **retornar un valor.**

Alejo subrayo **retorna un valor**, creo que debería tenerlo en cuenta.

Ahora vamos a definir **procedimiento:**

**Procedimiento:** Programa almacenado físicamente en una [base de datos](https://es.wikipedia.org/wiki/Base_de_datos).

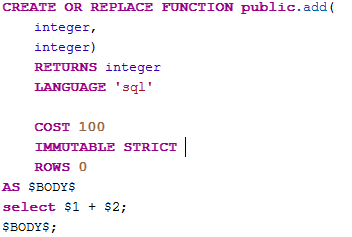
No entiendo, ¿teóricamente son lo mismo no?

La diferencia está en lo que **subraye con negro,** un procedimiento y una función pueden en teoría hacer lo mismo, **solo** que la función es restrictiva, y solo permite **retornar un valor SIEMPRE.** El procedimiento tiene la ventaja de retornar los valores que desees (**Parametros OUT)** o no retornar nada, la **función** SIEMPRE debe retornar un valor.

Ahora, ya veámoslo y no sigamos con Wikipedia.

**FUNCION**

Vamos a definir una función en primera instancia en nuestro **dbms.** Veámosla

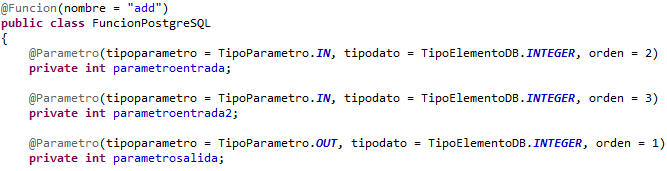


***Imagen 84. Función en postgreSQL.***

Tenemos definida una función en POSTGRESQL, que nos toma 2 números por Parametro y los suma.

Veamos el **mapeo** en JAVA.

**FuncionPostgreSQL.java**



***Imagen 85. Mapeo función en JAVA.***

¿Qué podemos observar?

En nuestra clase **FuncionPostgreSQL** tenemos **3** campos a mapear.

* **Parametroentrada:** equivale al **primer** parámetro de entrada (**tiene orden 2)**
* **Parametroentrada2:** equivale al **segundo** parámetro de entrada **(tiene orden 3)**
* **Parametrosalida:** equivale al parámetro de salida **(tiene orden 1).**

Si recordamos, cuando llamaos una función mediante **CallableStatement** tenemos algo como esto:

{ ? = call nombre\_funcion(?,?) }

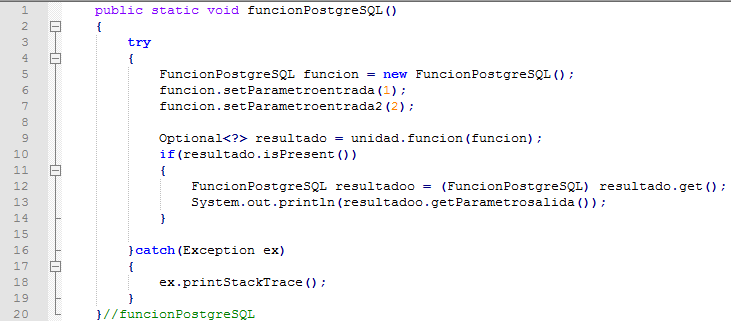
El signo de interrogación de color rojo (?) Representa el valor que va retorna.

(El|los) signo(s) de interrogación de color naranja (?) Representa los parámetros de la función.

En este orden de Ideas, nuestro **primer parámetro** siempre debe ser el valor **a retornar,** por eso en este caso (**FuncionPostgreSQL.java**) nuestro parámetro de salida se llama **parametrosalida** y tiene **orden 1,** luego ya los parámetros que contenga nuestra función, en este caso (**FuncionPostgreSQL.java**) **2 parámetros (Parametroentrada -> orden 2 y parametroentrada2 -> orden 3)**

¿Fácil no?

Vamos a nuestra prueba unitaria.



***Imagen 86. Prueba unitaria funciones.***

Encontramos que después de construir la estructura de nuestra función, vamos a crear un **objeto** de función, asignamos los valores de entrada, y luego la consumimos!!

**Vamos a los procedimientos.**

**PROCEDIMIENTO:**

Retornando la definición de **procedimiento:**

**Procedimiento:** Programa almacenado físicamente en una [base de datos](https://es.wikipedia.org/wiki/Base_de_datos).

Esto quiere decir que este es nuestro objeto más volátil (puede que retorne algo o simplemente no haga nada más allá que procesamiento en la base de datos).

Funciona de la misma forma que una función, maneja parámetros de entrada y de salida, y su callable funciona de la misma forma:

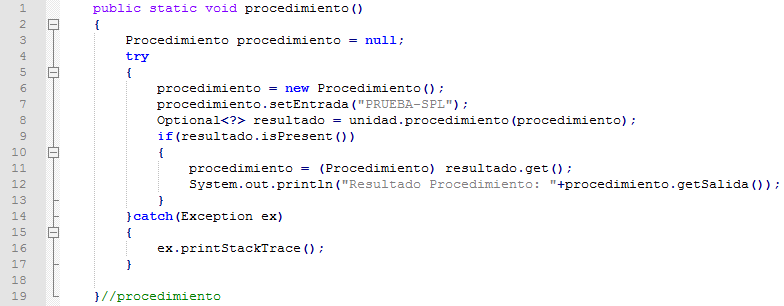
{ ? = call nombre\_procedimiento (?,?) }

El signo de interrogación de color rojo (?) Representa el valor que va retorna. (ES OPCIONAL)

(El|los) signo(s) de interrogación de color naranja (?) Representa los parámetros de la función.

¿Mucho más fácil no?

Pónganos la prueba unitaria Alejo.



***Imagen 87. Prueba unitaria procedimientos.***

Encontramos que después de construir la estructura de nuestro procedimiento almacenado, vamos a crear un **objeto** de procedimiento, asignamos los valores de entrada, y luego la consumimos y se acabó.

### Fechas

Vamos a hablar de las fechas de una manera mas “genérica”. Las fechas son objetos complejamente manejables, ya que existen muchos sistemas que manejan sus fechas a “su forma”. SPL maneja 2 tipos de fecha, vamos a verlos.



***Imagen 88. Fecha tipo Date.***

Las fechas tipo **date,** son las únicas fechas que no requieren del campo **formato.** Las fechas en SPL de tipo **date** solo contienen la información **gruesa** de la fecha (**año-mes-dia**) por ende el formato solo nos serviría para cambiarle el “maquillado” (**año-mes-dia**),(**año/mes/dia**) etc.

*Nota:* ***DEBEN*** *SI O SI, estar en un objeto de tipo* ***LocalDate.***



***Imagen 89. Fecha tipo timestamp.***

Ahora nuestras fechas tipo **timestamp** son fechas que por primera instancia, deben ir con el campo formato lleno en su totalidad, esto para que nuestra fecha con su factor tiempo disponible (**timestamp**) pueda ser “maquillado” o formateado (**año-mes-dia horas:minutos:segundos),** (**año/mes/dia’T’horas:minutos:segundos**).

*Nota:* ***DEBEN*** *SI O SI, estar en un objeto de tipo* ***LocalDateTime.***

Que les parece un “**Que aprendimos en total**” (siiii)

# ¿Perfecto! Que fue TODO lo que aprendimos?

1. **Anotaciones**
2. **Metadatos:** Son datos que describen otros datos
3. **Anotación:** Es una forma de añadir metadatos al código fuente
4. **Conexión**
   1. **DSN,** fuente de datos configurada para gestionar la conexión a una base de datos
   2. **Patrón builder:** Es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente.
   3. **Conexión:** es el enlace que se establece entre el emisor y el receptor a través del que se envía el mensaje.
5. **Enum**
   1. **ENUM**:  proporciona una manera eficaz de definir un conjunto de constantes enteras con nombre que se pueden asignar a una variable.
   2. Los enum pueden ser usados como metadatos, como puntos de referencia para tratar comportamientos.
6. **Excepciones**
   1. **Excepciones:** Algo fuera de lo común en el flujo normal y correcto de nuestro algoritmo.
7. **Seguridad**
   1. **Seguridad**: confianza.
   2. **Token**: una unidad de valor la cual permite llevar la identidad digital de la persona.
8. **TO**
   1. **transfer object**: es un objeto que transporta Información entre procesos.
   2. **Entidades**: es la representación de un objeto o concepto del mundo real que se describe en una base de datos.
9. **Unidad persistencia**
10. **Unidad de persistencia en JPA**: Representa un conjunto de entidades que pueden ser mapeadas a una base de datos, así como la información necesaria para que la aplicación JPA pueda acceder a dicha base de datos.
11. **Unidad de persistencia en SPL:** Representa un conjunto de métodos y rutinas que nos permite interactuar directamente con la base de datos.
12. **JPA:** un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard Y enterprise.
13. **API:** Es un conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción.
14. **Utilidades**
    1. **Concurrencia:** es una propiedad de los sistemas en la cual los procesos de un cómputo se hacen simultáneamente, y pueden interactuar entre ellos.
    2. **Introspección:** capacidad de algunos [lenguaje de programación](https://es.wikipedia.org/wiki/Lenguaje_de_programaci%C3%B3n) [orientado a objetos](https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos) de determinar el tipo de un [objeto](https://es.wikipedia.org/wiki/Objeto_(programaci%C3%B3n)) en [tiempo de ejecución](https://es.wikipedia.org/wiki/Tiempo_de_ejecuci%C3%B3n).
    3. **Clase abstracta:** Este Tipo de Clases nos permiten crear “método generales”.
    4. **Java.lang.reflect:** Paquete en java que provee clases e interfaces que obtienen en tiempo de ejecución características propias de los objetos.
    5. **Singleton:** instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.
15. **Constantes**
    1. **Constante:** Es un valor que no puede ser alterado/modificado durante la ejecución de un programa, únicamente puede ser leído.
    2. **Variables discretas:** es una variable que no puede tomar algunos valores dentro de un mínimo conjunto numerable, quiere decir, no acepta cualquier valor, sea programáticamente o contextualmente.
    3. **Expresiones regulares:** es una secuencia de [caracteres](https://es.wikipedia.org/wiki/Car%C3%A1cter_(tipo_de_dato)) que forma un patrón de búsqueda, principalmente utilizada para la [búsqueda de patrones](https://es.wikipedia.org/wiki/B%C3%BAsqueda_de_patrones) de cadenas de caracteres u operaciones de sustituciones.
    4. **SQL:** es un lenguaje específico del dominio que da acceso a un sistema de gestión de bases de datos relacionales que permite especificar diversos tipos de operaciones en ellos.
16. **XML**
    1. **XML:** es un meta-lenguaje que permite definir lenguajes de marcas utilizado para almacenar datos en forma legible.
    2. **Commit:** Confirmación de una transacción.
    3. **Log:** se usa el término log, historial de log o registro a la grabación secuencial en un archivo o en una base de datos de todos los acontecimientos (eventos o acciones) que afectan a un proceso particular (aplicación, actividad de una red informática, etc.). (Waooo, me gustan esas definiciones!).
    4. **Default:** es una palabra de origen anglosajón que literalmente quiere decir o alude a hacer algo por defecto o de manera predeterminada.
    5. **Nulo:** Inexistente, que no ocurre.
    6. **Driver:** controlador de dispositivo, software realizado para controlar o simplemente conectarse un dispositivo ajeno.
    7. **Flag:** se refiere a uno o más bits que se utilizan para almacenar un valor binario o código que tiene asignado un significado.

FIN TEORICO

# Instalación y pruebas de SPL.

Pero por supuesto que existen unos ejemplos y un **JAR** predeterminado con todas las pruebas unitarias correspondientes (no solo existe el **JAR,** también está todo el código fuente).

Vamos brevemente a explicar cómo está construido nuestro cliente.

Contiene **3** paquetes, los cuales van a ser descritos.

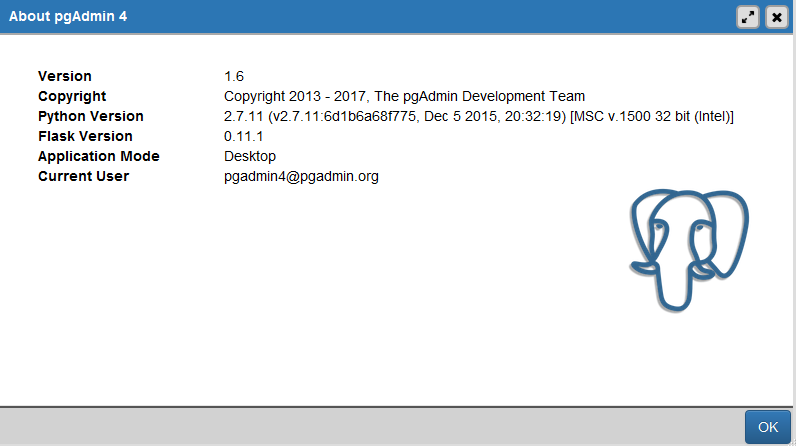
* **Dbobject:** Este paquete contiene todos los elementos con los que queremos comunicarnos por medio de SPL **QUE NO SEAN** tablas (Funciones, procedimientos)
* **Entidades:** Este paquete encapsula todas las tablas con las que queremos relacionarnos por medio de SPL.
* **Main:** clase donde encontramos nuestro **public static void main** o mejor conocido como inicio de programa en JAVA.

Ahora vamos a guiarlos a cómo deben hacer la prueba completa.

En primera instanciala única dependencia (y para serles sincero fue porque yo lo decidi asi, fue por simple gusto) las pruebas se realizaron en 4 **dbms** (SQL SERVER, ORACLE, MYSQL, Y **POSTGRE)**

Me imagino que ya se imaginaron que **dbms** es mi Wikipedia.

Las pruebas que se encuentran de aquí en adelante fueron adelantadas con **PostgreSQL 9.6** con el cliente **PGADMIN 4.**



***Imagen 88. Versión PGAdmin pruebas de SPL.***

Y así mismo la versión de **PostgreSQL.**

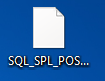


***Imagen 89. Instancia postgreSQL 9.6 en PGADMIN 4;***

Ahora va lo más discutido, si vas hacer lo que vas hacer, porque no puedo usarlo donde yo quiera?

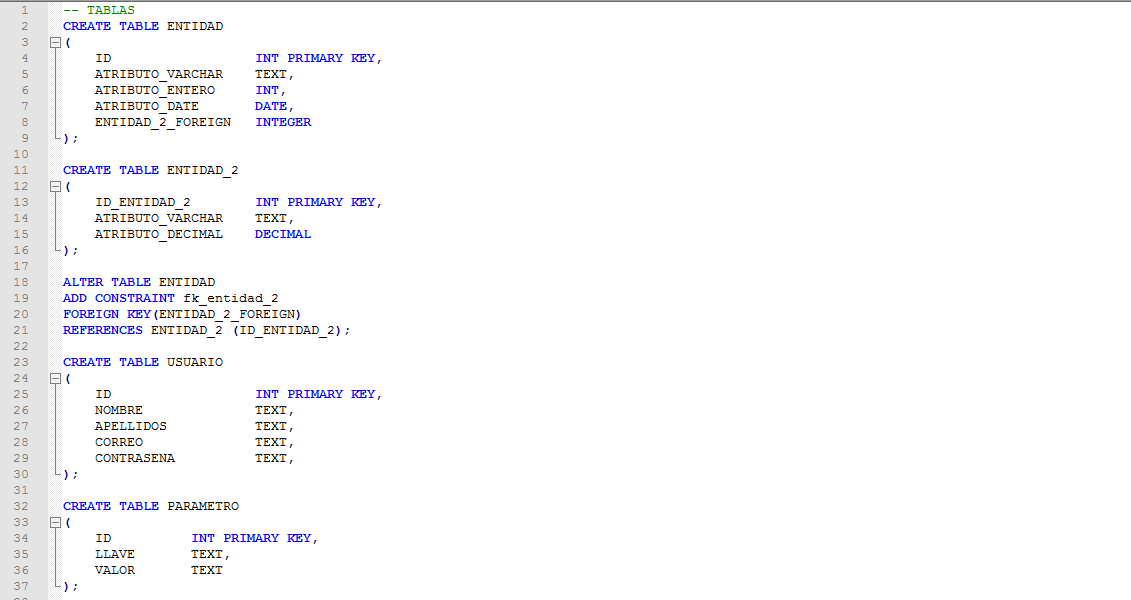
**AMI ME GUSTA POSTGRESQL. Esa es mi única y ultima respuesta.**

Ahora ya con el software previamente instalado, vamos a ir a un pequeño archivo que viene adjunto en el cliente, llamado **SQL\_SPL\_POSTGRE9.6**



***Imagen 90. SQL DDL Pruebas unitarias SPL.***

Es un archivo como uni corriente en el cual vamos a encontrar las definiciones de nuestras tablas y objetos de prueba para poder ejecutar el cliente de manera correcta, miremos el archivo.



***Imagen 91. Contenido archivo SQL\_SPL\_POSTGRE9.6.sql.***

Es un SQL cualquiera que puede ser fácilmente traducido entre **dbms’s** distintos, tiene sintaxis de **PostgreSQL 9.6.**

**No es ningún archivo de backup para PostgreSQL ni PGAdmin ni nada respectivo.**

# Léeme

Aunque no lo creas, para mi es lo más importante de este documento.

Impresionante como el aprendizaje y conocimiento que puede conglomerarse en un artefacto o entorno de trabajo simple como este (SPL), pero sobre todo, me da muchísimo gusto tener la cabeza, la paciencia y sobre todo el **amor** de poder construir algo en este mundo actual de la tecnología, donde la mayoría de las cosas se encuentran ya bien definidas.

Debo confesarles que me siento muy feliz, ahora, observando por la pequeña ventana de mi escritorio y contemplando a 24/08/17 las 66 hojas que representa la documentación simple de SPL, que puedo decirles que más que sentirme satisfecho, creo que necesito más de esto, cuando empiezas algo, debes terminarlo, y si lo terminas, porque no hacerlo mucho mejor? No estamos acostumbrados a la evolución y a que las cosas cambien de sabor y color como sea? O como yo quiera?.

DEBO invitarlos obligatoriamente a que por favor, aprendamos a compartir, pero no a compartir cosas ni momentos, también es importante aprender a compartir sentimientos, como te sientes ahora, como te quieres sentir, y como debes hacer sentir mejor a los demás, es por eso, que cordialmente y muy encarecidamente, te invito a que el framework, sea un elemento para abrir un canal entre tu (Mi amado lector) y yo (Alejo y wikipedia), para que juntos, hagamos que SPL crezca y sea parte de cada uno de nosotros PERO DE MANERA CONSTRUCTIVA, si quieres aprender sobre esto, deberías intentar pensar algo muy simple pero efectivo.

**“El amor es hacer por los demás, lo que quisiéramos que hicieran con nosotros”.**

No soy el “mejor” compañeros, pero en el camino, me encontré con la excelencia.

Muchas gracias y estoy completamente abierto a las retroalimentaciones para poder crecer como persona e ingeniero a diario.

Sin ser mas, que tengan buen día, y en lo que sean que crean, pueda acompañarlos siempre.

Sinceramente de **vosotros**.

**Alejandro Lindarte Castro**.