**Máster Universitario en Ingeniería del Software para la Web**



**Trabajo Final Asignatura:**

**Herramientas de Desarrollo Web**

**Profesores:**

Dr. D. Salvador Otón Tortosa – [salvador.oton@uah.es](mailto:salvador.oton@uah.es)

Dr. D. Roberto Barchino Plata – [roberto.barchino@uah.es](mailto:roberto.barchino@uah.es)

**Alumno:**

Piero Rospigliosi Beltrán – [piero.rospigliosi@uah.es](mailto:piero.rospigliosi@uah.es), [pierinho13@gmail.com](mailto:pierinho13@gmail.com)

**SOLUCION ENCONTRADA:**

Dado que el enunciado del trabajo final nos pide crear una nueva aplicación que sea un API REST. Y dado que normalmente nosotros conocemos de antemano la dirección del API de donde vamos a pedir la información he considerado que la API, además de ser un proyecto nuevo, va tener una dirección diferente y una base de datos diferente que el proyecto intermedio.

Por tanto nuestra API será solicitada a <http://localhost:8081/proveedoresAPI/producto/porCantidadYFecha> . Sin embargo, dado que existe la posibilidad de que se desee lanzar la aplicación en el mismo Tomcat que el proyecto intermedio de tienda Virtual, he dado opción en el formulario a que se edite la raíz. Es decir, la parte de <http://localhost:8081> . Esto se explicará mas adelante en el manual de uso e instalación.

Dado que el trabajo intermedio se ha creado ser lanzado en la misma raíz, este proyecto se configurado para que abra en raíz/proveedoresAPI. Siendo compatible desplegar los dos proyectos en el mismo Tomcat pero recomendable que se desplieguen por separado.

En adelante, se supondrá que el proyecto intermedio será desplegado en localhost:8080 y el API en localhost:8081, siendo su URL de acceso localhost:8081/proveedoresAPI

Para realizar este trabajo se ha trabajado en Spring Framework que está basado en Eclipse y nos permite trabajar con proyectos Maven con dependencias de Spring, como Spring MVC por ejemplo.

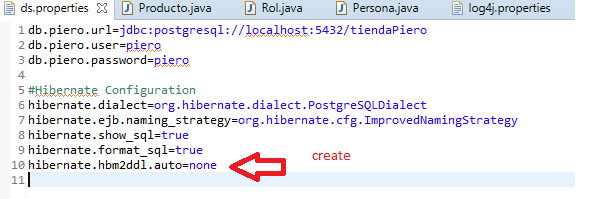
Se han utilizado las siguientes tecnologías:

* Maven, que es una herramienta para la gestión construcción de proyectos Java.
* Spring MVC, Spring Security (para la gestión de los usuarios).
* Spring JPA-DATA.
* En el capa de cliente, JSP, JSTL y Javasctipt.
* Postgres SQL.
* Para hacer las llamadas a nuestro API se ha usado la clase RestTemplate y HtttpComponentsClientHttpRequestFactory de Spring.

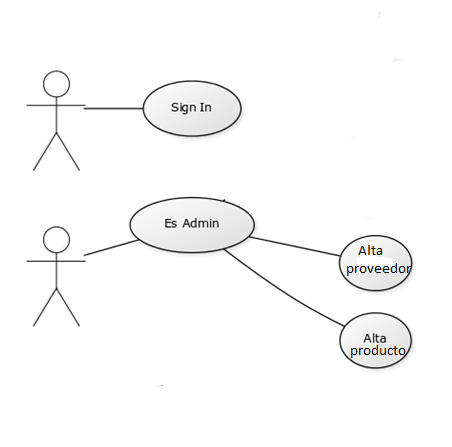
Se ha adjuntado el archivo script\_final.sql, el proyecto Maven a importar (si ese fuera el caso), el WAR llamado proveedoresAPI y el proyecto intermedio actualizado con el servicio de conexión a nuestra API REST.

Para lanzar correctamente script\_final.sql se debe haber lanzado correctamente antes script.sql del trabajo intermedio (puesto que comparten el usuario piero).

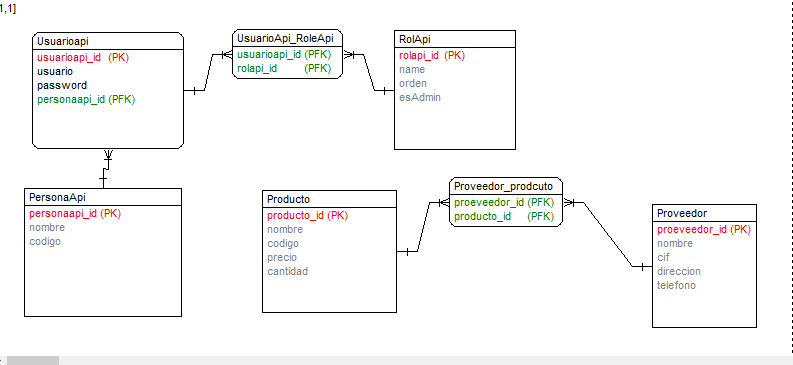
La aplicación también ha configurado para que si quisiéramos, solo ejecutar el script de crear base de datos y usuario. Y que Hibernate sea quien cree tablas y realice los insert del archivo import.sql del proyecto. Para ello tendríamos que cambiar el valor que podemos ver abajo por create.



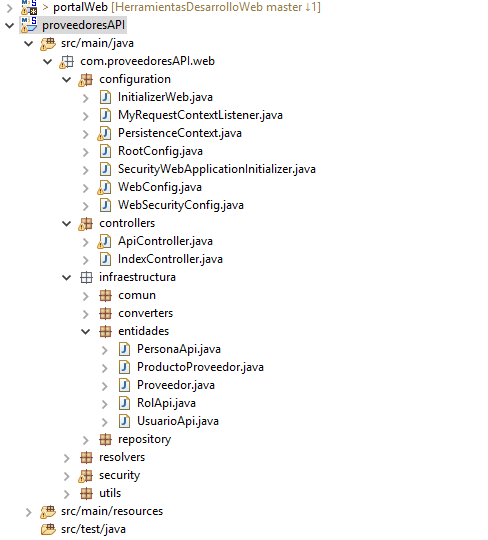
**CASOS DE USO:**



**DIAGRAMA ENTIDAD RELACIÓN:**



Esta es la estructura de nuestro proyecto Maven.



**Los usuarios de la aplicación son:**

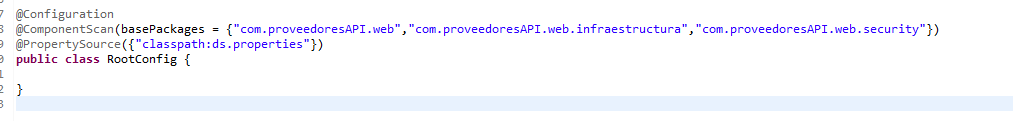
Para el ROL ADMIN : admin 1234

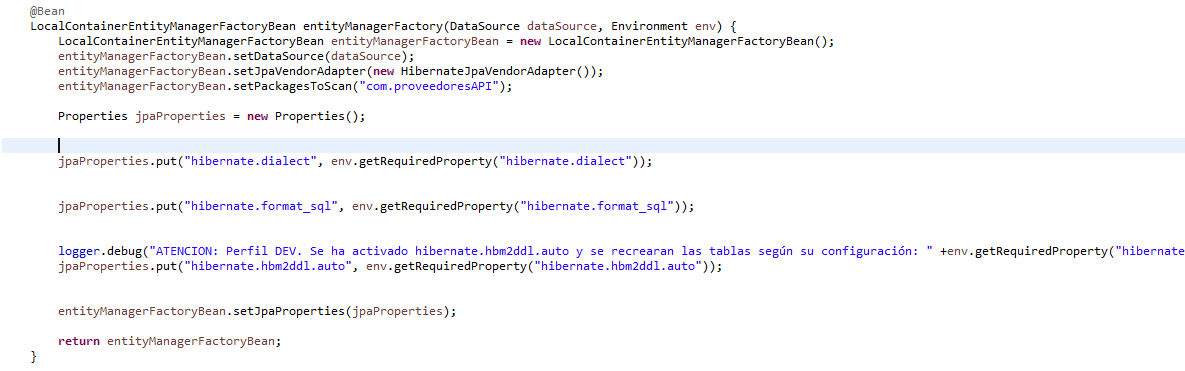
Para el ROL CLIENTE : cliente 1234

Donde admin y cliente son los usuarios y 1234 las contraseñas.

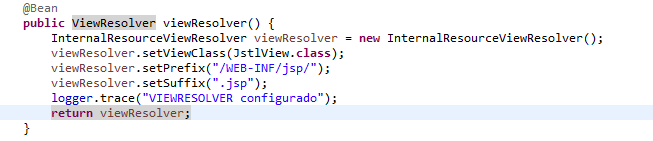
**Configuración principal de proyecto:**

Le decimos a Spring que en que paquetes buscar componentes



Vamos creando los beans necesarios, entre ellos el entityManagerFactory  
  


O el ViewResolver que nos dice donde buscar los archivos que usaremos en la capa de cliente y además le hemos agregado un sufijo por defecto.



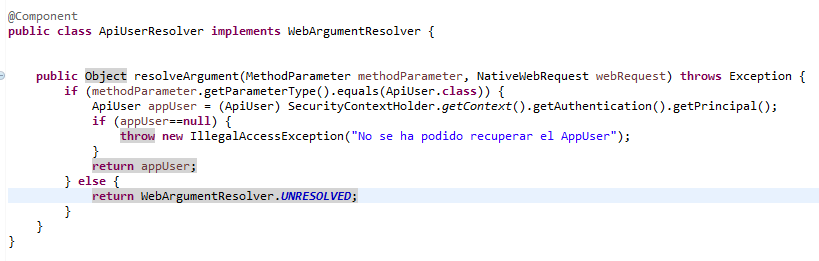
**Configuración Gestión de Usuarios con Spring Security**



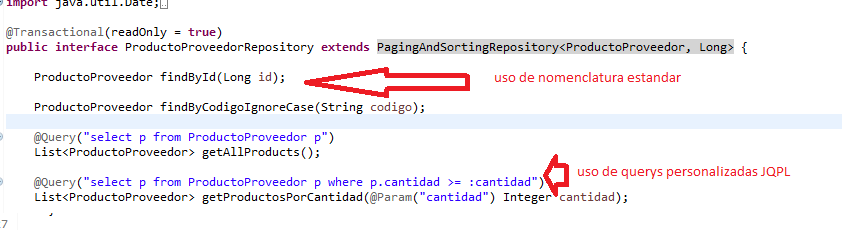
Nuestro MyUserDetailsService que obtiene el usuario de la base de datos y almacena sus datos en memoria.



ApiUserResolver para poder acceder a esos datos desde cualquier parte del código

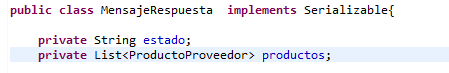


**Acceso a base de datos con JPA mediante la anotación repositorios:**



**Resolución del problema de convertir nuestro controlador de productos en uno que devuelva una respuesta a nuestra tienda:**

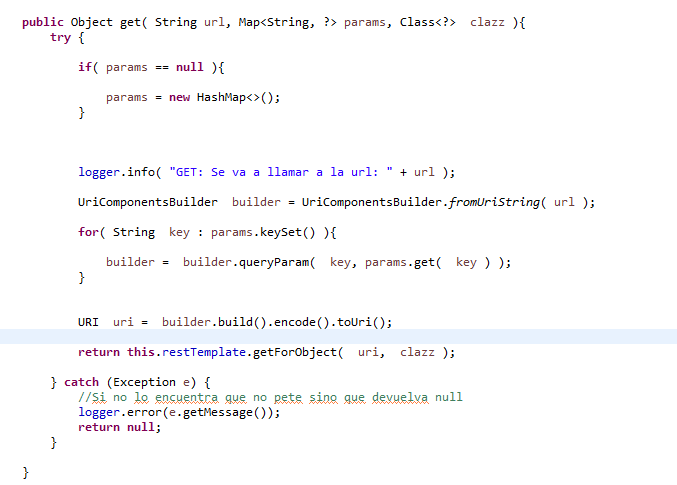
Mediante la creación de la clase MensajeRespuesta con atributos de estado y los datos



Uso de la anotación @ResponseBody en el controlador de nuestra API, que nos permite devolver un JSON en lugar de una vista como habitualmente.



Creación de nuestro servicio RestService en el proyecto Tienda que utiliza a RestTemplate de Spring:



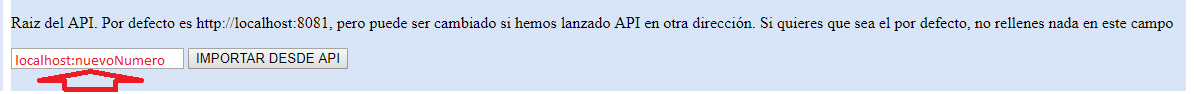
**Manual de instalación:**

* Crear la base de datos tiendaPiero del trabajo intermedio con el script de dicho trabajo. Se presupone el puerto localhost:5432 en PostgreSQL
* Crear la base de datos apiPiero con el sql que se adjunta para este trabajo.

**Si levantas los dos proyectos en puerto distinto (es decir en dos tomcat distintos)**

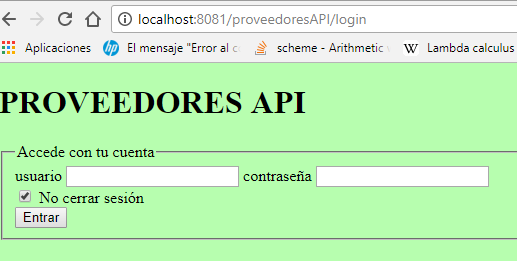
* Desplegar el proyecto del trabajo intermedio
* Desplegar el proyecto del trabajo final. Se presupone que estará en localhost:8081 pero se da opción a modificar la raíz del API en el formulario de importación.

**Si levantas en un mismo puerto (un solo tomcat)**

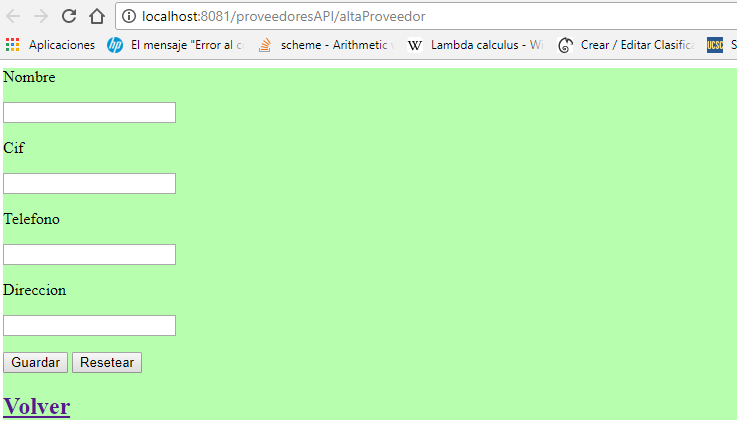
-Desplegar ambos proyectos. Pero tener en cuenta que la tienda se abre en localhost:8080, la API en localhost:8080/proveedoresAPI y que entonces debemos rellenar el campo de especificar url raíz en el formulario de importación.  


**Pantallas de la aplicación:**

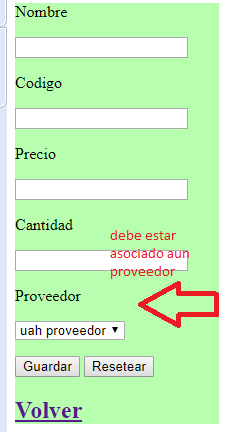
**Login**:



**Alta Proveedor:**



**Alta producto:**

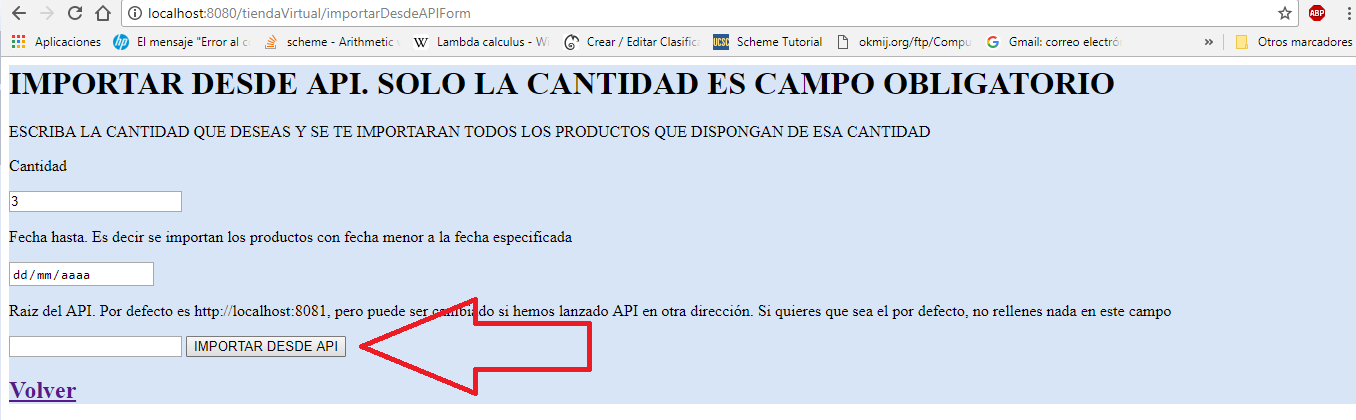


**Importar producto desde nuestra tienda:**

Para ello debemos loguearnos con un usuario con rol ADMIN (admin 1234)

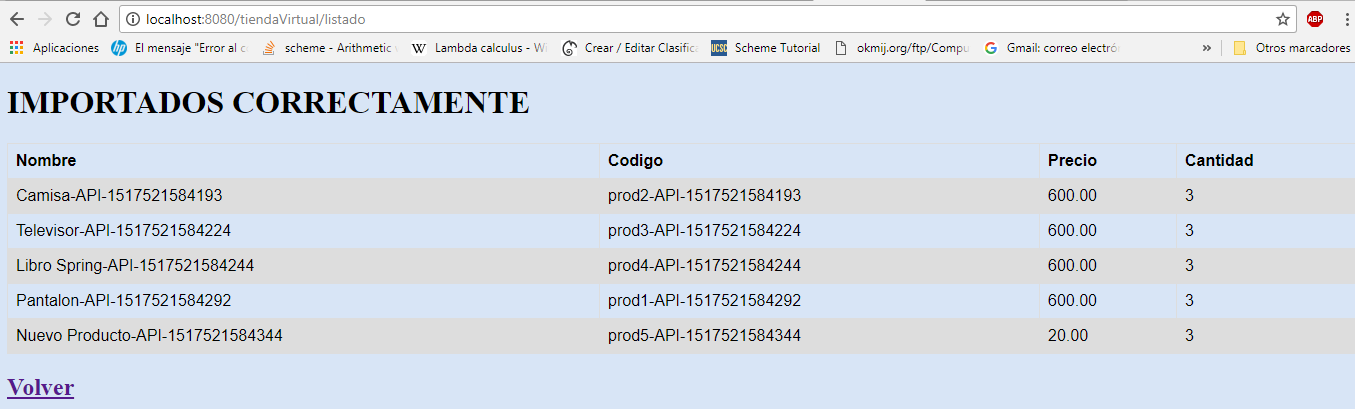


Entonces debemos pasar la cantidad, la fecha (opcional), y la raíz del API (si no has levantado el API en localhost:8081)

Y se nos importarán los productos que tengan más que la cantidad indicada en stock, y si le has pasado la fecha los productos cuya fecha de entrada sea menor a la indicada.  


De esta manera se nos importarán y además se reducirá la cantidad indicada en el API.

**Resultado:**



Podemos ver que efectivamente se ha llamado al API en los logs y comprando que ahora existe menos cantidad en el API (Por ejemplo solicitando una cantidad que sabemos ya no debería existir).

