

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE  
TELECOMUNICACIÓN  
UNIVERSIDAD POLITÉCNICA DE CARTAGENA



Trabajo Fin de Grado

**Diseño e implementación de una aplicación web para una tienda virtual**



AUTOR: Daniel Alarcón Amador

DIRECTOR: Dr. Fernando Losilla López



## Ficha de Propuesta Proyecto Final de Carrera/Trabajo Fin de Grado



Autor	Daniel Alarcon Amador
E-mail del Autor	Danychi24@gmail.com
Director	Fernando Losilla López
E-mail del Director	<a href="mailto:fernando.losilla@upct.es">fernando.losilla@upct.es</a>
Codirector(es)	
Título del PFC	Diseño e implementación de una aplicación web para una tienda virtual.
Descriptores	
Resumen	
<p>El trabajo que se muestra en la memoria de este proyecto muestra el diseño y desarrollo de una aplicación web, haciendo uso de las tecnologías que están teniendo mejor acogida entre los desarrolladores web. La aplicación implementa una tienda virtual en la que los usuarios pueden comprar una serie de productos haciendo uso de los servicios web. Entre las tecnologías usadas se encuentran HTML5, Bootstrap, CSS3, JavaScript y AngularJS, para el front-end y MySQL, Hibernate, Spring/Jersey y Maven para el back-end.</p> <p>Las principales claves del proyecto son el uso de una arquitectura REST con los servicios web de Spring/Jersey y una buena interfaz para el usuario gracias a la potencia de AngularJS</p>	
Titulación	Grado en Ingeniería Telemática
Intensificación	
Departamento	Tecnologías de la Información y las Comunicaciones
Fecha de Presentación	Octubre 2015



# Contenido

1. Introducción.....	7
1.1 Planteamiento del Proyecto .....	7
1.2 Objetivos del Proyecto .....	7
2. Tecnologías Usadas .....	8
2.1 HTML5.....	8
2.2 CSS3 .....	12
2.3 BOOTSTRAP[5] .....	14
2.4 PHP [6] .....	17
2.5 JAVASCRIPT[7][8] .....	18
2.6 ANGULAR JS .....	20
2.7 HTTP[13] .....	26
2.8 Hibernate[16].....	31
2.9 Maven .....	35
2.10 MySQL .....	36
2.11 Spring/Jersey.....	38
3. Entornos de Programación.....	42
3.1 XAMPP[22].....	42
3.2 Sublime Text [23] .....	44
3.3 Eclipse Mars[24] .....	45
¿Qué es?.....	45
4. Desarrollo del Cliente.....	50
4.1 Objetivo .....	50
4.2 Tecnologías Implicadas .....	52
4.3 Estructura.....	53

4.4 Explicación.....	54
5. Servidor .....	97
5.1 Objetivo .....	97
5.2 Tecnologías Implicadas .....	97
5.3 Estructura.....	98
5.4 Configuración.....	101
5.5 Explicación.....	109
6. Test.....	129
Usuarios.....	129
Tienda .....	137
7. Conclusiones y Líneas futuras.....	140
8. Bibliografía.....	141

# 1. Introducción

## 1.1 Planteamiento del Proyecto

Hoy en día el comercio electrónico está en auge con numerosas tiendas que ofrecen sus productos en internet. En este sentido existe una gran cantidad de tecnologías que permiten el desarrollo de aplicaciones web para la creación de tiendas on-line, las cuales se encuentran en constante evolución y cambio.

En el proyecto se optarán por las tecnologías que están teniendo mejores resultados y mayor acogida por parte de los desarrolladores y empresas. Entre las tecnologías que se usarán se encuentra HTML5, Bootstrap, CSS3, JavaScript y AngularJS para el Front-End y MySQL, Hibernate, Spring/Jersey y Maven para el Back-End. La comunicación cliente-servidor se hará siguiendo una arquitectura REST. La aplicación web permitirá al usuario comprar una serie de productos almacenados en una base de datos, tras iniciar una sesión en la tienda en la que tendrá acceso a información relacionada con sus pedidos y podrá gestionar sus datos de cliente.

El proyecto va estar separado en diferentes secciones, en primer lugar se documentará al usuario de la información básica para entender las tecnologías usadas, a continuación se detallarán los **entornos de programación** sobre los que se ha trabajado para, finalmente dar paso a la explicación del desarrollo del **cliente** y el **servidor**

## 1.2 Objetivos del Proyecto

Los objetivos del proyecto son:

- Aprender las principales tecnologías utilizadas en Front-End, como HTML5, Bootstrap, CSS3, Javascript, AngularJS para el Front-End
- Conocer las tecnologías de mayor utilización para Back-End, como MySQL, Hibernate, Spring/Jersey y Maven.
- Ser capaz de desarrollar una aplicación web para compra on-line desde cero, tratando todos los aspectos del diseño e implementación de la aplicación.

## 2. Tecnologías Usadas

### 2.1 HTML5



#### ¿Qué es HTML5? [1]

HTML5 es un lenguaje de marcas (sus siglas significan Hyper Text Markup Language) usado para estructurar y presentar el contenido en la web. HTML5 es la última revisión del estándar creado en 1990, hace pocos años la W3C la recomendó para ser el nuevo estándar usado en los próximos proyectos web. Con HTML5 entra en desuso el formato XHTML, ya que no será necesaria su implementación.

HTML4 fue su predecesor, declarado como lenguaje oficial de la web en el año 2000. Con esta nueva generación de HTML, se introducen algunos cambios muy significativos que comentaremos en las siguientes secciones.

Volviendo a qué es HTML5, se trata de una tecnología que nos permite formatear el **layout** de nuestra página, de esta manera nuestro navegador sabe **cómo** mostrar la **información**, donde se sitúan los elementos, donde poner un vídeo, imagen y como novedad como dividir las principales secciones de nuestra página fácilmente.

#### Cuáles son sus novedades

HTML5 se ha conseguido adaptar a los nuevos tiempos que corren, entre las novedades que trae, una de las más importantes es que permite introducir multimedia en la web sin



necesidad de **plugins** y el conocido **Adobe Flash**. Gracias a este cambio, se amplía notablemente el número de dispositivos que pueden navegar por la web sin la necesidad de estas extensiones, algunos navegadores como Chrome o incluso el SO Android querían eliminar la dependencia de algunos elementos como Adobe Flash, por lo tanto HTML5 supone una gran ventaja.

Otras novedades que trae es la posibilidad de acceder a sitios web de manera offline, funcionalidades como la de drag and drop, la edición de documentos online popularizados por Google Docs, geolocalización, entre otras muchas.

## Nuevas etiquetas introducidas

HTML funciona mediante la interpretación de marcas que cobran sentido, llamadas etiquetas. Los navegadores leen estas etiquetas e interpretan el código para poder así permitirnos visualizar el contenido de la página.

En HTML5 se introducen nuevas etiquetas que nos permiten hacer cosas que con HTML no eran posibles. Se ha pretendido con esta nueva revisión que la escritura de código sea más sencilla, lógica y comprensible para el programador. La idea que se buscaba con HTML5 era la correcta visualización de contenido multimedia en dispositivos de gama baja, intentando prescindir de la instalación de plugins u otras tecnologías que podrían ralentizar el dispositivo o incluso no permitir visualizar la web.

Algunas de las nuevas etiquetas son:

- **article**: esta etiqueta se usa para escribir un artículo, publicación, comentario, independiente del resto del contenido.
- **header, footer**: estas etiquetas evitan tener que definir un div con un id determinado, como se solía hacer anteriormente. Además, se pueden insertar en cada sección.
- **nav**: se utiliza para definir una sección que sólo contiene enlaces de navegación
- **section**: se utiliza para separar contenido dentro de la página, definiendo secciones. Tiene un funcionamiento similar al div, pero de esta forma se ve

claramente las separaciones entre secciones.

- **audio y video:** estas etiquetas definen el tipo de contenido multimedia que estará en su interior, además puede ser reproducido por casi todos los dispositivos.
- **embed:** con esta etiqueta se puede marcar la presencia de un contenido interactivo o aplicación externa que por lo general no es HTML.
- **canvas:** Representa un área de mapa de bits en el que se pueden utilizar scripts para renderizar gráficos como gráficas, gráficos de juegos o cualquier imagen visual al vuelo, necesitará el uso de JavaScript.

A continuación se verá un ejemplo de HTML5 [2]:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>HTML5 Skeleton</title>
<meta charset="utf-8">
</head>

<body>
<header>
<h1>HTML5 SKeleton</h1>
</header>

<nav>
<ul>
<li><a href="html5_semantic_elements.asp">HTML5 Semantic</a></li>
<li><a href="html5_geolocation.asp">HTML5 Geolocation</a></li>
<li><a href="html5_canvas.asp">HTML5 Graphics</a></li>
</ul>
</nav>

<section>
<h1>Famous Cities</h1>
<article>
<h2>London</h2>
<p>London is the capital city of England. It is the most populous city in the United Kingdom, with a metropolitan area of over 13 million inhabitants.</p>
</article>

<article>
<h2>Paris</h2>
<p>Paris is the capital and most populous city of France.</p>
</article>

<article>
<h2>Tokyo</h2>
<p>Tokyo is the capital of Japan, the center of the Greater Tokyo Area, and the most populous metropolitan area in the world.</p>
</article>
</section>

<Footer>
<p>&copy; 2014 W3Schools. All rights reserved.</p>
</Footer>

</body>
</html>
```

<b>Concepto</b>	<b>Descripción</b>
<code>&lt;html&gt;</code>	Representa la raíz de un documento HTML o XHTML. Todos los demás elementos deben ser descendientes de este elemento. Lang = “en”, especifica el idioma base de la página.
<code>&lt;head&gt;</code>	Representa una colección de metadatos acerca del documento, incluyendo enlaces a, o definiciones de, scripts y hojas de estilo.
<code>&lt;title&gt;</code>	Define el título del documento, el cual se muestra en la barra de título del navegador o en las pestañas de página. Solamente puede contener texto y cualquier otra etiqueta contenida no será interpretada.
<code>&lt;meta&gt;</code>	Define los metadatos que no pueden ser definidos usando otro elemento HTML.
<code>&lt;body&gt;</code>	Representa el contenido principal de un documento HTML. Solo hay un elemento <code>&lt;body&gt;</code> en un documento.
<code>&lt;header&gt;</code>	Define la cabecera de una página o sección. Usualmente contiene un logotipo, el título del sitio Web y una tabla de navegación de contenidos.
<code>&lt;h1&gt;</code> hasta <code>&lt;h6&gt;</code>	Los elemento de cabecera implementan seis niveles de cabeceras de documentos; <code>&lt;h1&gt;</code> es la de mayor y <code>&lt;h6&gt;</code> es la de menor importancia. Un elemento de cabecera describe brevemente el tema de la sección que introduce.
<code>&lt;ul&gt;</code>	Define una lista de artículos sin orden.
<code>&lt;li&gt;</code>	Define un artículo de una lista enumerada.
<code>&lt;nav&gt;</code>	Define una sección que solamente contiene enlaces de navegación
<code>&lt;section&gt;</code>	Define una sección en un documento.
<code>&lt;footer&gt;</code>	Define el pie de una página o sección. Usualmente contiene un mensaje de derechos de autoría, algunos enlaces a información legal o direcciones para dar información de retroalimentación.

---

# HTML5 SKeleton

- [HTML5 Semantic](#)
- [HTML5 Geolocation](#)
- [HTML5 Graphics](#)

## Famous Cities

### London

London is the capital city of England. It is the most populous city in the United Kingdom, with a metropolitan area of over 13 million inhabitants.

### Paris

Paris is the capital and most populous city of France.

### Tokyo

Tokyo is the capital of Japan, the center of the Greater Tokyo Area, and the most populous metropolitan area in the world.

© 2014 W3Schools. All rights reserved.

Resultado

## 2.2 CSS3



CSS3 nos ayuda a definir las reglas y estilos de representación para nuestra página definida en HTML. Las hojas de estilo en cascada (Cascading Style Sheets o CSS) nos permiten representar nuestro estilo en diferentes dispositivos tanto ordenadores, móviles o cualquier otro dispositivo que nos permita mostrar contenidos web.

La primera versión de CSS se publicó en 1996, fue logrando popularidad hasta la versión 2.1, convirtiéndose en un estándar durante mucho tiempo. A partir de 2005 se comenzó a trabajar en la actual versión CSS3 o Cascading Style Sheet Level 3. Esta versión introduce nuevos cambios, adaptándose a las nuevas necesidades de los diseñadores web. Desde opciones de sombreado y redondeado, hasta funciones de transición, transformaciones 2D/3D o movimiento.

## ¿Cómo funciona?[3]

Las definiciones de estilo se guardan normalmente en archivos .css externos.

Con este archivo podemos cambiar el estilo de un sitio Web; podemos ver un ejemplo de cómo adjuntar a nuestra página diferentes archivos .css mediante la etiqueta link dentro de head, aclarando con el atributo rel la relación entre el documento actual y el vinculado.

```
<head>
<meta charset="utf-8">
<title>Untitled</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
<meta name="description" content="" />
<meta name="author" content="http://bootstraptaste.com" />
<!-- css -->
<link href="../css/bootstrap.min.css" rel="stylesheet" />
<link href="../css/fancybox/jquery.fancybox.css" rel="stylesheet">
<link href="../css/flexslider.css" rel="stylesheet" />
<link href="../css/style.css" rel="stylesheet" />

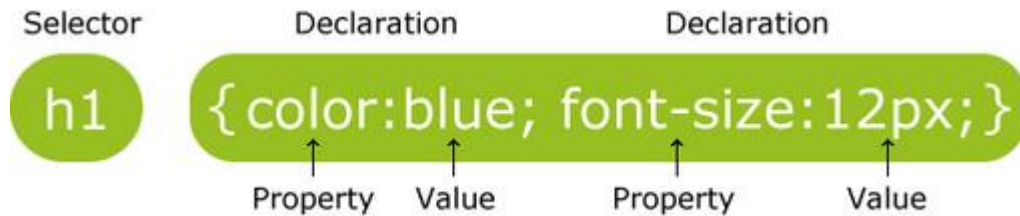
<!-- Theme skin -->
<link href="../skins/default.css" rel="stylesheet" />
</head>
```

Cuando un navegador muestra un documento, se debe combinar el contenido (HTML) con la información de estilo (CSS). Esto se hace en dos etapas:<sup>i</sup>

- 1- El navegador convierte el lenguaje de marcas y el CSS en el **DOM** (Document Object Model). El DOM representa el documento en la memoria del ordenador. Combina el contenido del documento con su estilo.
- 2- El navegador muestra el contenido del DOM.

## Sintaxis [4]

Un conjunto de reglas CSS consiste en un selector y un bloque de declaración:



El selector apunta al elemento HTML que va a aplicar el estilo. Éste puede apuntar a un elemento genérico como `<h1>`, `<body>`, `<p>` o a una clase o id que hayamos creado en nuestro documento HTML.

El bloque de declaración contiene una o más declaraciones separadas por punto y coma. Cada declaración incluye un nombre de propiedad y un valor, separados por dos puntos. En el ejemplo mencionado arriba, el elemento `h1` se mostraría de color azul y con un tamaño de fuente de 12 píxeles.

**Hello World!**

**This h1 is styled with CSS.**

Resultado

## 2.3 BOOTSTRAP[5]



Twitter Bootstrap es un entorno de trabajo o conjunto de herramientas que facilitan el desarrollo y diseño de sitios web. Está formado por plantillas de diseño con tipografía, botones, cuadros, formularios, barras de navegación, menús desplegables y otros elementos basados en HTML, CSS y JS (principalmente la librería jQuery).

## **Diseño responsive**

Mediante el sistema de disposición del contenido en Bootstrap, conseguimos que nuestra página web se visualice correctamente en diferentes dispositivos sin importar su resolución, ya sea un teléfono móvil, tableta o computadora con baja o alta resolución.

## **Entendiendo la hoja de estilo CSS**

Bootstrap trae por defecto un conjunto de hojas de estilo que proveen definiciones básicas para todos los elementos que se encuentran en HTML. Esto otorga una uniformidad a toda la página y al sistema de anchura, da una apariencia moderna para el formateo de los elementos de texto, tablas y formularios.

## **Componentes re-usables**

Bootstrap trae componentes por defecto que pueden ser de gran ayuda como botones con características avanzadas (botones con menú desplegable, listas de navegación, paginación, etc.), etiquetas, formatos para mensajes de alerta, barras de progreso, paneles. Todos estos componentes los podemos encontrar en su página oficial y usar dentro de nuestra página a nuestro gusto.

## **Plug-ins de JavaScript**

Los componentes de JavaScript para Bootstrap están basados en la librería jQuery.

En su página podemos encontrar todos los elementos a los que tenemos acceso, entre ellos diálogos, tooltips, carruseles, transiciones, etc.

La versión 2.0 soporta los siguientes plug-ins de JavaScript: Modal, Dropdown, Scrollspy, Tab, Tooltip, Popover, Alert, Button, Collapse, Carousel y Typeahead.

Ejemplo

```

<div class="row">
  <div class="col-lg-12">
    <div class="row">
      <section id="categorias">
        <ul id="thumbs">
          <!-- Item Project and Filter Name -->
          <li class="col-lg-4 design" data-id="id-0" data-type="web">
            <div class="item-thumbs link-index">
              <!-- Thumb Image and Description -->
              <a href="shop/shop.php/"></a>
              <h3><a href="shop/shop.php/">SHOP</a></h3>
              <p>The best selection of music. You will find all genres, with high quality in all our products.</p>
              <a href="shop/shop.php">Go now!</a>
            </div>
          </li>
          <!-- End Item Project -->

          <!-- Item Project and Filter Name -->
          <li class="col-lg-4 design" data-id="id-0" data-type="web">
            <div class="item-thumbs link-index">
              <!-- Thumb Image and Description -->
              <a href="shop/shop.php"></a>
              <h3><a href="shop/shop.php/">DIGITAL MUSIC</a></h3>
              <p>Get your favorite music has never been easier. Download it instantly!</p>
              <a href="shop/shop.php">Go now!</a>
            </div>
          </li>
          <!-- End Item Project -->

          <!-- Item Project and Filter Name -->
          <li class="col-lg-4 design" data-id="id-0" data-type="web">
            <div class="item-thumbs link-index">
              <!-- Thumb Image and Description -->
              <a href="shop/shop.php/"></a>
              <h3><a href="shop/shop.php/">NEWS</a></h3>
              <p>Taste the last music, the best selection.</p>
              <a href="shop/shop.php/">Go now!</a>
            </div>
          </li>
          <!-- End Item Project -->
        </ul>
      </section>
    </div>
  </div>
</div>

```



### SHOP

The best selection of music. You will find all genres, with high quality in all our products.

Go now!



### DIGITAL MUSIC

Get your favorite music has never been easier. Download it instantly!

Go now!



### NEWS

Taste the last music, the best selection.

Go now!

Gracias a Bootstrap podemos organizar de forma sencilla nuestra página.

Bootstrap divide cada fila en 12 columnas, en nuestro ejemplo cada elemento de la lista ocupa 4 columnas en formato large, usada para dispositivos cuya resolución es mayor o igual a 1200 píxeles.



## 2.4 PHP [6]



### ¿Qué es?

**PHP** es un lenguaje de programación diseñado para el **desarrollo web de contenido dinámico**, éste debe ser interpretado por un servidor.

Fue uno de los primeros lenguajes que podía ser incluido dentro del documento HTML. Puede ser usado en la mayoría de los servidores web y en casi todos los sistemas operativos y plataformas de forma gratuita.

### Implementación en el proyecto

Dentro de este proyecto se ha utilizado simplemente para inyectar código HTML de manera sencilla y rápida, que permitiese separar los documentos de una manera ordenada y eficaz.

```
<?php include 'inc/encabezado-index.php' ?>  
<?php include 'inc/pie-index.php' ?>
```

## 2.5 JAVASCRIPT[7][8]



### Para qué sirve

JavaScript (abreviado JS) es un lenguaje de programación que se utiliza principalmente para crear páginas **web dinámicas**. Con JS podemos incorporar efectos a nuestros textos, **animaciones** a imágenes, acciones que se activan al pulsar un botón, ventanas que envían un mensaje de aviso al usuario o **modificar** el **DOM** (Document Object Model).

### Nacimiento

JavaScript es un **lenguaje de programación interpretado**, no es necesario compilarlo para ejecutarlo en nuestra web. JavaScript se diseñó con una sintaxis similar al lenguaje de programación C, aunque adopta nombres y convenciones del lenguaje de programación Java. Sin embargo no está relacionado con Java y tienen semánticas y propósitos diferentes. Todos los navegadores modernos interpretan el código JavaScript.

### Actualidad JS

Habitualmente se utilizaba en páginas web HTML para realizar operaciones únicamente en el marco del cliente, sin hacer peticiones a un servidor. Actualmente JS se usa para enviar y recibir información del servidor junto con otras tecnologías como AJAX o AngularJS.

Desde el lanzamiento en junio de 1997 del primer estándar ECMAScript 1, han existido las versiones 2, 3 y 5, que es la más usada actualmente. En junio de 2015 se publicó la versión ECMAScript 6, que se implementará próximamente.

## Ejemplo

Algunas características del lenguaje que podemos encontrar son la creación de variables, condiciones, bucles, creación de funciones, etc.

```
getStatus: function (statusId) {
  if(statusId != null){
    var status;
    switch (statusId){
      case 1:
        status = "Confirmed";
        break;

      case 2:
        status = "Ready for Shipping";
        break;

      case 3:
        status = "Sent";
        break;

      case 4:
        status = "Dispatched";
        break;

      case 5:
        status = "Delivery Failure";
        break;

      case 6:
        status = "Canceled";
        break;

      default:
        status = "Unknown";
    }
  }
  return status;
},
```

En el ejemplo de arriba se crea una función que recibe una variable llamada “statusId”, posteriormente comprueba que ésta no tiene un valor nulo, finalmente ejecuta una sentencia switch para evaluar el valor de la variable y devuelve un resultado.

## Implementación de JavaScript en el proyecto

En el proyecto se ha integrado JavaScript con AngularJS, su principal cometido ha sido hacer llamadas al servidor, como podemos ver en este ejemplo:

```
getProducts: function (success, error) {
  $http.get(urls.BASE_API_PRODUCTS).success(success).error(error)
},
```

## 2.6 ANGULAR JS



### ¿Qué es AngularJS?[9]

**AngularJS** (también conocido como Angular) es un **framework** de **JavaScript** del tipo **MVC** (Modelo Vista Controlador), mantenido por Google. El objetivo es aumentar las aplicaciones basadas en navegadores con capacidad MVC, facilitando el desarrollo y pruebas.

Actualmente es muy usado en páginas del tipo **Single Page Application**[10] (aplicación de página única). La SPA es una aplicación web o un sitio web que cabe en una sola página con el propósito de dar una experiencia más fluida a los usuarios como una aplicación de escritorio

Una de las principales ventajas de Angular es que se integra perfectamente con **HTML**, para servir mejor **contenido dinámico** a través del **data-binding bidireccional** permitiendo la sincronización automática de **modelos** y **vistas**. El data-binding bidireccional consiste en[11]:

- Cuando las propiedades del modelo se actualizan, lo mismo ocurre con la interfaz del usuario.
- Cuando los elementos de la interfaz de usuario se actualizan, los cambios se propagan de vuelta al modelo.

## Descripción Conceptual[12]

En la siguiente tabla comentaremos los principales conceptos necesarios para entender cómo funciona AngularJS:

Concepto	Descripción
<b>Template</b>	HTML con markup adicional.
<b>Directives</b>	Extienden HTML con atributos y elementos adicionales.
<b>Model</b>	Los datos que se muestran al usuario en la vista y con los que el usuario interactúa.
<b>Scope</b>	Contexto en el que se almacena el modelo para que los controladores, directivas y expresiones puedan acceder a él.
<b>Expressions</b>	Las variables de acceso y funciones del scope.
<b>Compiler</b>	Analiza el template e instancia las directivas y expresiones.
<b>Filter</b>	Formatea el valor de una expresión para la visualización del usuario.
<b>View</b>	Lo que el usuario ve (el DOM).
<b>Data Binding</b>	Sincroniza los datos entre el modelo y la vista.
<b>Controller</b>	La lógica de negocio detrás de la vista.
<b>Dependency Injection</b>	Crea y conecta objetos y funciones.
<b>Injector</b>	Contenedor de inyección de dependencias.
<b>Module</b>	Un contenedor para las diferentes partes de una aplicación, incluyendo controladores, servicios, filtros, directivas que configuran el inyector
<b>Service</b>	Lógica de servicio independiente reusable de las vistas

## Un primer ejemplo

```
<div ng-app ng-init="qty=1;cost=2">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="qty">
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="cost">
  </div>
  <div>
    <b>Total:</b> {{qty * cost | currency}}
  </div>
</div>
```

Invoice:

Quantity:

Costs:

Total: \$2.00

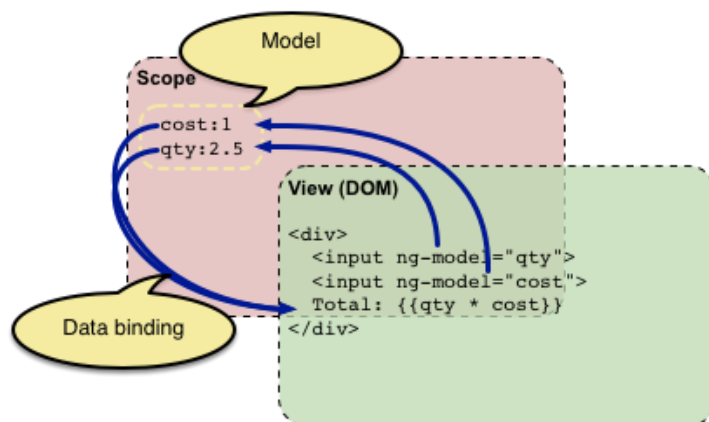
A simple vista parece un HTML normal, con markup diferente. En angular, este tipo de archivo se llama **template**. Cuando Angular inicia la aplicación, analiza y procesa este nuevo markup del template usando el **compiler**. Una vez que el **DOM** se ha cargado, transformado y renderizado pasa a llamarse **vista**.

En el ejemplo anterior empezamos usando el atributo **ng-app**, que está vinculado a una directiva que inicializa automáticamente nuestra **app**. Con **ng-init** damos un valor de inicio a las variables **qty** y **cost** del **modelo**. La directiva **ng-model** guarda/actualiza el valor de los campos **input** en una variable.

La segunda parte de nuestro nuevo markup está formado por **llaves dobles** del tipo **{{ expression | filter }}**, cuando el **compiler** encuentra este markup, va a reemplazarlo con el valor evaluado del markup. El código que se encuentra dentro es similar al de JavaScript, éste permite **leer** y **escribir variables**. Al igual que las variables de JavaScript funcionan sólo en un **scope** (ámbito), las variables que escribamos dentro deben pertenecer a nuestro scope, sino no se mostrará nada. En nuestro caso “**qty**” y “**cost**” se refieren a las variables creadas en **ng-model** que pertenecen a nuestro scope, que le hemos dado unos valores iniciales. A la multiplicación de nuestras variables se les aplica un filtro que trae angular

por defecto para mostrar el resultado como dinero.

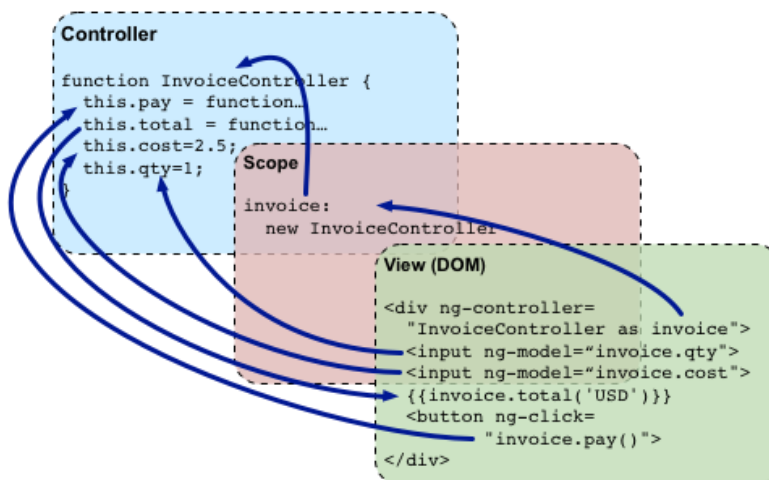
Por último, es importante decir que en este ejemplo Angular ofrece **binding** inmediato: Cuando los valores del **input** cambian, el valor de las **expresiones** se **recalcula** automáticamente y el **DOM** se actualiza con sus valores. El concepto detrás de esto es el enlace de datos bidireccional, que se ha explicado anteriormente.



## Segundo ejemplo

En este ejemplo vamos a separar el modelo de datos dentro de un controlador y vamos a implementar un servicio sencillo que luego inyectaremos en nuestro controller.

El **InvoiceController** contiene toda la lógica de nuestro ejemplo. Cuando la aplicación crece, es una buena práctica mover la lógica independiente desde el controlador a un servicio, por lo que puede ser inyectado en cualquier parte de la aplicación.



## Finance2.js Service

```
angular.module('finance2', [])
.factory('currencyConverter', function() {
  var currencies = ['USD', 'EUR', 'CNY'];
  var usdToForeignRates = {
    USD: 1,
    EUR: 0.74,
    CNY: 6.09
  };
  var convert = function (amount, inCurr, outCurr) {
    return amount * usdToForeignRates[outCurr] / usdToForeignRates[inCurr];
  };
  return {
    currencies: currencies,
    convert: convert
  };
});
```

El servicio `currencyConverter` se va a encargar fundamentalmente de recibir unos datos que son: Una cantidad de dinero (**amount**), una moneda de entrada (**inCurr**) y una moneda de salida (**outCurr**). Con estos datos va a hacer una conversión para obtener la cantidad de dinero en la moneda nueva. El servicio devuelve los tipos de conversión disponibles en este archivo y la función de conversión para que pueda ser usada en cualquier aplicación.

## Invoice2.js Controller

```
angular.module('invoice2', ['finance2'])
.controller('InvoiceController', ['currencyConverter', function(currencyConverter) {
  this.qty = 1;
  this.cost = 2;
  this.inCurr = 'EUR';
  this.currencies = currencyConverter.currencies;

  this.total = function total(outCurr) {
    return currencyConverter.convert(this.qty * this.cost, this.inCurr, outCurr);
  };
  this.pay = function pay() {
    window.alert("Thanks!");
  };
}]);
```

El controller va a usar el servicio que hemos creado arriba, aquí es donde entra en juego la inyección de dependencias. La **inyección de Dependencias (DI)**<sup>1</sup> es un patrón de diseño que se ocupa de cómo los objetos y funciones se crean y cómo consiguen mantener sus dependencias. Todos los elementos dentro de angular (directivas, filtros, controllers, servicios, etc) se crean y conectan mediante la inyección de dependencias. Al **contenedor**

---

<sup>1</sup> En Informática, Inyección de Dependencias[38] es un patrón de diseño orientado a objetos, en el que se suministran objetos a una clase en lugar de ser la propia clase quien cree el objeto.



de **DI** se le llama el **inyector**.

Para utilizar la DI, es necesario que haya un punto donde todos los elemento que necesiten trabajar juntos se registren. En angular, los **módulos** se encargan de este propósito. Cuando se inicia la aplicación, se utilizará la configuración del módulo con el nombre definido por la directiva **ng-app**.

Dentro de nuestro **controlador** se inicializan varias variables por defecto, se definen otras que extienden del **servicio** (como **currencias**) y se calculan otras haciendo uso de las funciones que trae el servicio (como **total**, que utiliza la función **currencyConverter**).

Finalmente se define una función que mostrará un pop-up diciendo “Thanks”. La sintaxis dentro del controlador es muy similar a JavaScript.

Template index.html

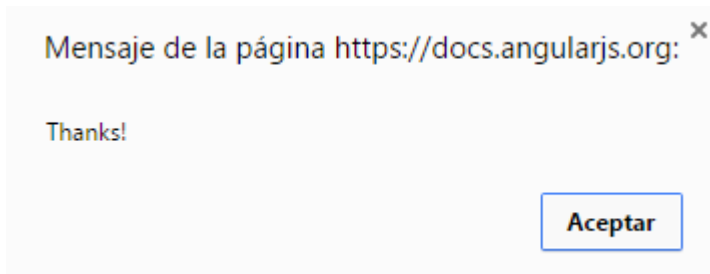
```
<div ng-app="invoice2" ng-controller="InvoiceController as invoice">
  <b>Invoice:</b>
  <div>
    Quantity: <input type="number" min="0" ng-model="invoice.qty" required >
  </div>
  <div>
    Costs: <input type="number" min="0" ng-model="invoice.cost" required >
    <select ng-model="invoice.inCurr">
      <option ng-repeat="c in invoice.currencias">{{c}}</option>
    </select>
  </div>
  <div>
    <b>Total:</b>
    <span ng-repeat="c in invoice.currencias">
      {{invoice.total(c) | currency:c}}
    </span>
    <button class="btn" ng-click="invoice.pay()">Pay</button>
  </div>
</div>
```

Para terminar, nos encontramos con el **template**, en el que inicializamos la aplicación con **ng-app**, invocamos al **controller** con **ng-controller** y le dotamos del nombre **invoice**, para que sea más sencillo acceder a sus variables en la aplicación. Se puede acceder a cualquier variable o función del controller con “**invoice.variable**”. Algunas variables traen un valor por defecto, aunque pueden ser actualizadas haciendo uso de los input que están conectados con **ng-model** al **controller**.

También nos encontramos con un objeto tipo **select** de HTML, que con la directiva **ng-repeat** a modo de bucle recorrerá las opciones dentro del **array currencies** para mostrar sus valores a elegir.

Finalmente el valor **total** se calculará usando la función definida en el **controller** como **total** (que a su vez usa la función **convert** del servicio **currencyConverter**) y mostrará el valor total en cada moneda haciendo uso de **ng-repeat** para recorrer todos los resultados posibles del array aplicándole el **filtro** adecuado en cada momento; si pulsamos el botón “Pay” saltará un pop-up diciendo “Thanks”.

**Invoice:**  
Quantity:   
Costs:  EUR ▼  
**Total:** USD2.70 EUR2.00 CNY16.46 Pay



## 2.7 HTTP[13]



**Hypertext Transfer Protocol** o **HTTP** es el protocolo más extendido para cada transacción de la **World Wide Web** (WWW). Fue desarrollado por el World Wide Web Consortium y la Internet Engineering Task Force, colaboración que culminó en 1999 con la publicación de una serie de RFC, el más importante es el RFC 2616 que especifica la versión 1.1.

HTTP se encarga de definir la arquitectura web necesaria para que clientes, servidores, proxies, etc. puedan comunicarse entre ellos. Es un protocolo orientado a transacciones y sigue el esquema **petición-respuesta** entre un cliente y un servidor.

Al cliente que se encarga de efectuar la petición se le conoce como "**user agent**". La información transmitida se la llama **recurso** y se la identifica mediante un **URL** (localizador uniforme de recursos). El servidor puede responder con el resultado de la ejecución de un programa, una consulta a una base de datos, la traducción automática de un documento, etc.

HTTP se le conoce como un protocolo **stateless** (sin estado), que no guarda ninguna información sobre conexiones anteriores. Para mantener el estado se suelen usar **cookies**, que es información que un servidor puede **almacenar** en el sistema **cliente** y recurrir a ella cuando sea necesario. Con las cookies podemos mantener una "**sesión**", y también permite **rastrear información** de los usuarios para que el servidor la interprete de una manera determinada.

## Métodos de HTTP

**HTTP** define **ocho métodos** (aunque nos interesan esencialmente cuatro) que indican la **acción** que se desea que se efectúe sobre el **recurso identificado**.

Normalmente, el recurso corresponde a un archivo, un objeto de la base de datos, la salida de un ejecutable que reside en un servidor, etc.

### GET

Pide un recurso especificado. Se suele usar para **leer** un dato del servidor.

Ejemplo:

GET /user/1 devuelve el usuario con Id número 1

## HEAD

Es similar a la petición GET, pero en este caso sólo necesita la **cabecera**, sin el body (cuerpo) de la respuesta. Es muy útil para recuperar la **meta-información** escrita en los encabezados de respuesta.

## POST [14]

Envía **datos** a una dirección del servidor para que sean procesados como el servidor crea oportuno. Estos datos se incluyen en el body de la petición. El servidor puede **crear** un recurso nuevo, guardar los datos en una base de datos, borrarlo, devolver una información; se conoce como el método comodín.

### Usos

- Se puede usar para crear:

POST /emp/ Crea un nuevo empleado y devuelve su URI (p. ej. /emp/125)

- O para crear/actualizar un recurso:

POST a la URL: /emp/120

En caso de que no existiera, esta operación creará el empleado 120 con los datos que se han pasado en la petición, si ya existía actualizará el empleado 120. POST no es idempotente. **Idempotente** significa que vamos a obtener el mismo resultado sin importar el número de veces que realicemos la petición.

## PUT

Se utiliza para poner un recurso en un lugar especificado (la URL), por ejemplo para actualizar o crear un **recurso**. Es el camino más eficiente para subir archivos a un servidor, esto es porque en POST utiliza un mensaje multiparte y el mensaje es decodificado por el servidor.

### Ejemplo:

PUT /emp/120 actualiza la información del empleado 120 con la información que se ha

pasado en el cuerpo de la petición. Estamos definiendo una operación **idempotente** sobre un recurso específico. No importa cuántas veces la lancemos, el resultado será siempre el mismo: el recurso identificado como "/emp/120" existirá con los datos que se han pasado en la petición. Si no existía se creará. Si ya existía se reemplazará de nuevo. No afecta a nada más.

## DELETE

**Borra** el recurso especificado.

## TRACE

Este método sirve para que el servidor devuelva toda la información que reciba del mensaje de solicitud, se utiliza a modo de **diagnóstica y prueba**.

## OPTIONS

Devuelve los **métodos** HTTP que el servidor **soporta**/permite.

## CONNECT

Se utiliza para **comprobar si un host está disponible**, no necesariamente la petición llega al servidor, se usa para comprobar si un proxy nos da acceso a un host bajo unas condiciones especiales; por ejemplo corrientes de datos bidireccionales encriptadas (como requiere SSL).

En el proyecto se han usado principalmente los métodos **POST, GET, PUT y DELETE**, que se corresponde con un **CRUD** (Create, Read, Update and Delete) que son las peticiones básicas a una base de datos.

## Códigos de estado de respuesta

Los códigos más comunes usados en HTTP son los siguientes [15]:

Response Code	HTTP Operation	Response Body Contents	Description
200	GET, PUT, DELETE	Resource	No error, operation successful.
201 Created	POST	Resource that was created	Successful creation of a resource.
202 Accepted	POST, PUT, DELETE	N/A	The request was received.
204 No Content	GET, PUT, DELETE	N/A	The request was processed successfully, but no response body is needed.
301 Moved Permanently	GET	XHTML with link	Resource has moved.
303 See Other	GET	XHTML with link	Redirection.
304 Not Modified	conditional GET	N/A	Resource has not been modified.
400 Bad Request	GET, POST, PUT, DELETE	Error Message	Malformed syntax or a bad query.
401 Unauthorized	GET, POST, PUT, DELETE	Error Message	Action requires user authentication.
403 Forbidden	GET, POST, PUT, DELETE	Error Message	Authentication failure or invalid Application ID.
404 Not Found	GET, POST, PUT, DELETE	Error Message	Resource not found.
405 Not Allowed	GET, POST, PUT, DELETE	Error Message	Method not allowed on resource.
406 Not Acceptable	GET	Error Message	Requested representation not available for the resource.
408 Request Timeout	GET, POST	Error Message	Request has timed out.
409 Resource Conflict	PUT, PUT, DELETE	Error Message	State of the resource doesn't permit request.
410 Gone	GET, PUT	Error Message	The URI used to refer to a resource.
411 Length Required	POST, PUT	Error Message	The server needs to know the size of the entity body and it should be specified in the Content Length header.
412 Precondition failed	GET	Error Message	Operation not completed because preconditions were not met.
413 Request Entity Too Large	POST, PUT	Error Message	The representation was too large for the server to handle.
414 Request URI too long	POST, PUT	Error Message	The URI has more than 2k characters.
415 Unsupported Type	POST, PUT	Error Message	Representation not supported for the resource.
416 Requested Range Not Satisfiable	GET	Error Message	Requested range not satisfiable.
500 Server Error	GET, POST, PUT	Error Message	Internal server error.
501 Not Implemented	POST, PUT, DELETE	Error Message	Requested HTTP operation not supported.
502 Bad Gateway	GET, POST, PUT, DELETE	Error Message	Backend service failure (data store failure).
505	GET	Error Message	HTTP version not supported.

## 2.8 Hibernate[16]



### ¿Para qué sirve Hibernate?

Hibernate es una herramienta usada para el Mapeo objeto-relacional (ORM). Facilita el mapeo de atributos en una base de datos relacional y el modelo de objetos en nuestra aplicación, mediante anotaciones en los beans<sup>2</sup> de las entidades que permiten establecer estas relaciones o archivos declarativos (XML o eXtensible Markup Language)

### Características liberadas

Como todas las herramientas de su tipo, Hibernate busca solucionar el problema de la diferencia entre los dos modelos de datos coexistentes en una aplicación: el usado en la memoria de la computadora (orientación a objetos) y el usado en las bases de datos (modelo relacional). Para lograr esto permite al desarrollador detallar cómo es su modelo de datos, qué relaciones existen y qué forma tienen. Con esta información Hibernate le permite a la aplicación manipular los datos en la base de datos operando sobre objetos, con todas las características de la POO. Hibernate convertirá los datos entre los tipos utilizados por Java y los definidos por SQL. Hibernate genera las sentencias **SQL** y libera al desarrollador del manejo manual de los datos que resultan de la ejecución de dichas sentencias, manteniendo la portabilidad entre todos los motores de bases de datos con un ligero incremento en el

---

<sup>2</sup> Un **Bean**[39] es un componente software que tiene la particularidad de ser reutilizable y así evitar la tediosa tarea de programar los distintos componentes uno a uno. Un **Bean** puede representar desde un botón, un grid de resultados, un panel contenedor o un simple campo de texto, hasta otras soluciones mucho más complejas como conexiones a bases de datos, etc.

tiempo de ejecución.

Hibernate está diseñado para ser flexible en cuanto al esquema de tablas utilizado, para poder adaptarse a su uso sobre una base de datos ya existente. También tiene la funcionalidad de crear la base de datos a partir de la información disponible.

Hibernate ofrece también un lenguaje de consulta de datos llamado **HQL** (*Hibernate Query Language*).

Hibernate para Java puede ser utilizado en aplicaciones Java independientes o en aplicaciones Java EE, mediante el componente **Hibernate Annotations** que implementa el estándar JPA, que es parte de esta plataforma.

## Ejemplo práctico[17]

Entity

@XmlElement

@Entity

@Table(name = "paymentdetails", schema = "untitled")

**public class** PaymentDetails **implements** Serializable {

private static final long *serialVersionUID* = 1L;

@Id

@Column(name = "PM\_ID")

@GeneratedValue(strategy = GenerationType.*AUTO*)

**private** Long paymentDetailId;

@Column(name = "DESCRIPTION")

**private** String description;

**public** Long getPaymentDetailId() {

return paymentDetailId;



```

    }
    public void setPaymentDetailId(Long paymentDetailId) {
        this.paymentDetailId = paymentDetailId;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
}

```

En el ejemplo de arriba podemos ver como relaciona la **base de datos** con nuestro modelo de datos de Java.

Anotación	Descripción
<b>@XmlRootElement</b>	Hace que esta clase se represente como elemento XML en un documento XML.
<b>@Entity</b>	Define nuestra clase como una entidad.
<b>@Table</b>	Conecta con la tabla <b>name="nombre de la tabla"</b> perteneciente la base de datos <b>schema="nombre de la BBDD"</b> .
<b>@Id</b>	La anotación <b>@Id</b> declara el siguiente objeto como la <b>columna</b> asignada para la <b>clave principal</b> de la <b>entidad</b> , se supone que es la clave principal de la tabla.
<b>@Column</b>	Hace referencia a la columna correspondiente a la tabla mencionada dentro de <b>@Table</b> .
<b>@GeneratedValue</b>	<b>@GeneratedValue(strategy = GenerationType.AUTO)</b> hace que cada vez que se cree un nuevo objeto (en nuestro caso un Id), se incremente automáticamente.
<b>@Transient</b>	Se utiliza para indicar que un campo no debe ser persistido en la base de datos.

Las demás líneas de código corresponden a getters/setters que podemos encontrar en cualquier aplicación de Java.

## DAO

```
@Override
public Users findUserByEmail(String login) {

    String sqlString = "SELECT u FROM Users u WHERE u.email = ?1";

    Query query = entityManager.createQuery(sqlString, Users.class);

    query.setParameter(1, login);

    return (Users) query.getSingleResult();

}
```

En este ejemplo se ha creado un método perteneciente al **DAO** (Data Access Object), del que se hablará en profundidad en el desarrollo.

Este método se encarga de hacer una consulta a la base de datos mediante Hibernate, la sintaxis que se ha usado es similar a SQL:

```
"SELECT u FROM Users u WHERE u.email = ?1";
```

Selecciona el usuario u de la tabla **Users** donde la característica de éste coincida con el email que recibe el método nombrado como “**login**” en este caso.

El **entityManager** se encarga de crear la **query**, que posteriormente ejecuta devolviendo un **único resultado** (getSingleResult) que en este caso corresponde con un usuario determinado.

## 2.9 Maven



### ¿Qué es Maven?[18]

Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Es similar en funcionalidad a Apache Ant, pero tiene un modelo de configuración de construcción más simple, basado en un formato XML. Maven utiliza un **Project Object Model (POM)** para describir el proyecto de software a construir, sus **dependencias** de otros módulos y **componentes externos**, y el orden de construcción de los elementos.

Una característica clave de Maven es que está listo para usar en red. El motor incluido en su núcleo puede dinámicamente descargar **plugins** de un **repositorio**, el mismo repositorio que provee acceso a muchas versiones de diferentes proyectos Open Source en Java, de Apache y otras organizaciones y desarrolladores. Este repositorio y su sucesor reorganizado, el **repositorio Maven 2**, pugnan por ser el mecanismo de facto de distribución de aplicaciones en Java, pero su adopción ha sido muy lenta.

Maven provee soporte no sólo para obtener archivos de su repositorio, sino también para subir artefactos al repositorio al final de la construcción de la aplicación, dejándola al acceso de todos los usuarios. Una caché local de artefactos actúa como la primera fuente para sincronizar la salida de los proyectos a un sistema local.

Maven está construido usando una arquitectura basada en plugins que permite que utilice cualquier aplicación controlable a través de la entrada estándar. En teoría, esto podría permitir a cualquiera escribir plugins para su interfaz con herramientas como compiladores, herramientas de pruebas unitarias, etcétera, para cualquier otro lenguaje. En realidad, el soporte y uso de lenguajes distintos de Java es mínimo.

## Ejemplo [19]

El **pom.xml** (Project Object Model) es un fichero del tipo **XML**, que es la **unidad principal** de un proyecto tipo **Maven**. Dentro de él se encuentra información relacionado con fuentes, plugins, versiones de paquetes, dependencias, test, seguridad, etc.

Gracias a esto, podemos resolver por ejemplo un tema que suele dar bastantes problemas a la hora de crear nuestra aplicación web, el conocido CORS (Cross Domain Resource Sharing). CORS es un mecanismo que permite a nuestra aplicación evitar que un cliente o servidor accedan a un recurso restringido, en caso de que se encuentren fuera del dominio que se originó el recurso. Incluyendo en nuestro pom.xml unas simples líneas, se instalan unas dependencias que ayudan a resolver este problema de raíz.

```
<!-- CORS -->
    <dependency>
        <groupId>com.thetransactioncompany</groupId>
        <artifactId>cors-filter</artifactId>
        <version>2.4</version>
    </dependency>
```

## 2.10 MySQL



## ¿Qué es?[20]

**MySQL** es un sistema de gestión de bases de datos relacional, multihilo y multiusuario con más de seis millones de instalaciones. MySQL AB —desde enero de 2008 una subsidiaria de Sun Microsystems y ésta a su vez de Oracle Corporation desde abril de 2009— desarrolla MySQL como software libre en un esquema de licenciamiento dual.

Por un lado se ofrece bajo la GNU GPL para cualquier uso compatible con esta licencia, pero para aquellas empresas que quieran incorporarlo en productos privativos deben comprar a la empresa una licencia específica que les permita este uso. Está desarrollado en su mayor parte en ANSI C.

MySQL es muy utilizado en aplicaciones web, como Joomla, Wordpress, Drupal o phpBB, en plataformas (Linux/Windows-Apache-MySQL-PHP/Perl/Python), y por herramientas de seguimiento de errores como Bugzilla. Su popularidad como aplicación web está muy ligada a PHP, que a menudo aparece en combinación con MySQL.

## Características

- Amplio subconjunto del lenguaje SQL. Algunas extensiones son incluidas igualmente.
- Disponibilidad en gran cantidad de plataformas y sistemas.
- Posibilidad de selección de mecanismos de almacenamiento que ofrecen diferentes velocidades de operación, soporte físico, capacidad, distribución geográfica, transacciones, etc.
- Transacciones y claves foráneas.
- Conectividad segura.
- Replicación.
- Búsqueda e indexación de campos de texto.

## Repercusión sobre el proyecto

En este proyecto se ha utilizado MySQL para mantener de forma ordenada y separada los datos relacionados con los usuarios, productos, pedidos, etc.

Más adelante veremos cómo se ha organizado la BBDD (base de datos).

## 2.11 Spring/Jersey



### ¿Qué es? [21]

**JAX-RS: Java API for RESTful Web Services** es una API del lenguaje de programación Java que proporciona soporte en la creación de servicios web de acuerdo con el estilo arquitectónico Representational State Transfer (REST). JAX-RS usa anotaciones, introducidas en Java SE 5, para simplificar el desarrollo y despliegue de los clientes y puntos finales de los servicios web.

A partir de la versión 1.1 en adelante, JAX-RS es una parte oficial de Java EE 6. Una característica notable de ser parte oficial de Java EE es que no se requiere configuración para comenzar a usar JAX-RS (por lo tanto no tendremos que agregar código en nuestro pom.xml para usar Jersey). Para los entornos que no son Java EE 6 se requiere una (pequeña) entrada en el descriptor de despliegue web.xml.

De acuerdo con el Tutorial de Java EE 6, Volumen 1: Jersey es la implementación de referencia de calidad de producción de Sun para JSR 311: JAX-RS: The Java API for RESTful Web Services. Jersey implementa soporte para las anotaciones definidas en la JSR-311, lo que facilita a los desarrolladores crear servicios web RESTful con Java y la JVM de Java. Jersey también añade características adicionales no especificadas por la JSR.

El principal objetivo de Jersey ha sido siempre la facilitación de creación de servicios web RESTful que utilicen Java y la máquina virtual de Java (JVM).

## Ejemplo

Servicio

@Autowired

```
private UsersDao usersDao;
```

@Override

```
public Users findUserByEmail(String email) {  
    return usersDao.findUserByEmail(email);  
}
```

Siguiendo el ejemplo mostrado en Hibernate, se crea un servicio que conecta con el DAO de Users con la etiqueta @Autowired.

El servicio se usa para llamar a una o más funciones del DAO y utilizar esta respuesta en el endpoint.

End-Point

@Autowired

```
private UsersService usersService;
```

@POST

@Path("/login")

@Produces("application/json")

@Consumes("application/json")

```
public Response login(@RequestParam Users userRequest) {  
    Users user = usersService.findUserByEmail(userRequest.getEmail());  
    String passRequest = userRequest.getPassword();  
  
    if(passRequest.equals(user.getPassword())){  
        return Response.status(200).build();  
    }else{  
return Response.status(HttpStatus.FORBIDDEN.value()).build();  
    }  
}
```

Anotación	Descripción
<b>@Autowired</b>	Conecta con el <b>Servicio</b> creado anteriormente.
<b>@MétodoHttp</b>	Con <b>@POST</b> especificamos que este método sólo funcionará cuando se reciba un POST
<b>@Path</b>	Definimos la dirección donde se espera ese <b>POST</b> , en este caso <b>“/login”</b> .
<b>@Produce</b> y <b>@Consumes</b>	Especifican el tipo de datos que va a consumir el método, en este caso va a producir y consumir datos tipo <b>“json”</b>
<b>@RequestParam</b>	Especifica el tipo de parámetro que espera.

El método va a devolver un objeto tipo Response al cliente, en esta ocasión se espera recibir en el request un objeto tipo Users (clase definida en el entity, que se encarga de realizar la conexión entre el modelo de datos de Java y la base de datos), se nombra como userRequest para poder utilizarlo dentro del método.

Invocando a la función findUserByEmail del servicio se extrae el **usuario** de la base de datos correspondiente con el email que se ha recibido en el **Request**. Se va a comparar la contraseña del **usuario extraído** con la del **usuario del request**, suponiendo que coincidan entonces el **login** es correcto, por lo tanto el método devolverá un **HttpStatus 200** que significa **OK!**, en caso contrario el método devolverá un **HttpStatus 403, FORBIDDEN**.

### *Response*

Una de las partes más importantes en el endpoint, es la respuesta HTTP que le damos al usuario. Analicemos cómo construir una respuesta:

Primero se crea un objeto del tipo **Response**, se utiliza el método **status(HttpStatus.State.value())** o directamente **status(StatusCode)** para establecer el código de estado que va a recibir el cliente. Con el método **entity(object)** es posible enviar un objeto en el cuerpo de la respuesta. Existe la opción de añadir una cabecera en la



respuesta con la función **header**, dentro de ésta se pueden realizar algunas configuraciones, introducir cookies o insertar objetos como con el método `entity`. Finalmente se debe utilizar el método `build` para construir la respuesta.

Ejemplo de una response:

```
Response response = Response.status(HttpStatus.CREATED.value())
    .header("Access-Control-Allow-Origin",
"http://localhost:8080")
    .header("Access-Control-Allow-Credentials", true)
    .header("Access-Control-Allow-Methods", "POST,
GET, OPTIONS, PUT, DELETE, HEAD")
    .header("Access-Control-Allow-Headers", "Content-
Type, authorization").header("token", token)
    .entity(token).build();
    return response;
```

## 3. Entornos de Programación

### 3.1 XAMPP[22]

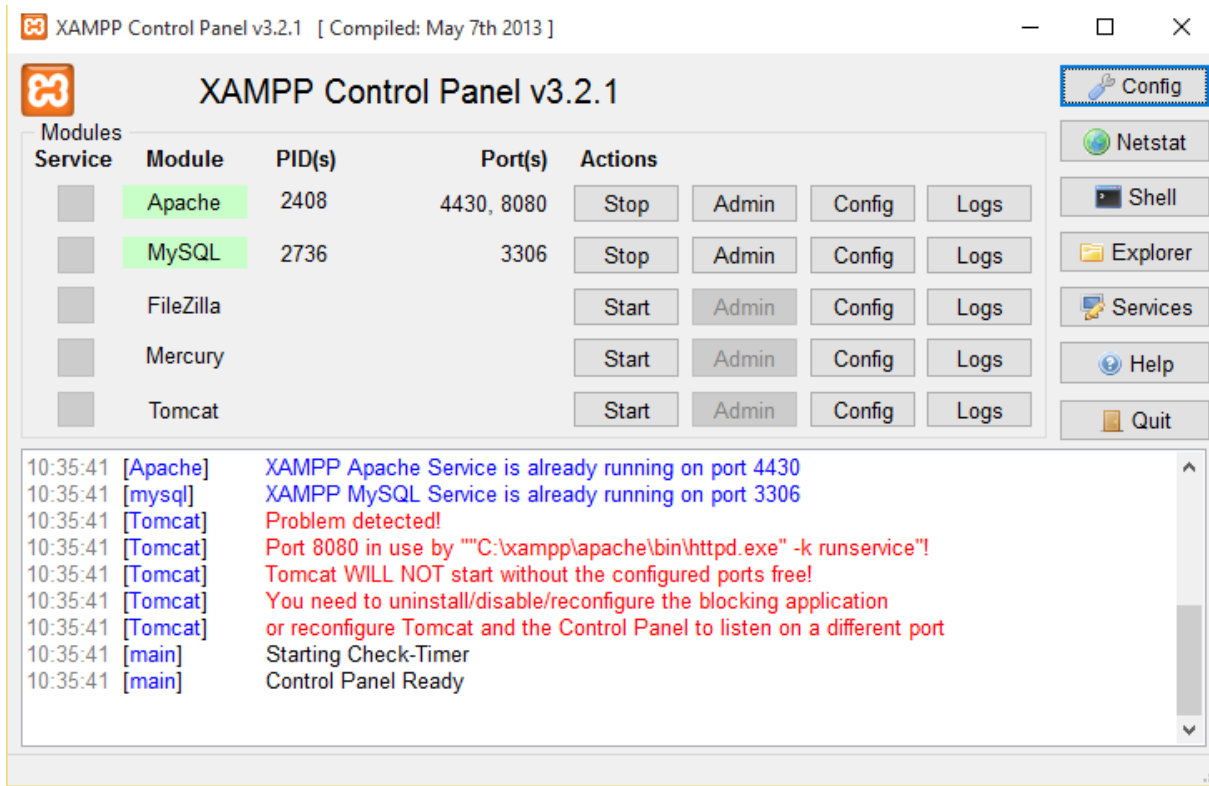


XAMPP es una distribución de Apache completamente gratuita y fácil de instalar que contiene MySQL, PHP y Perl. Xampp es gratuito tanto para usos comerciales como no comerciales; facilita enormemente la instalación y configuración de las tecnologías mencionadas anteriormente, por defecto viene configurado con todas las opciones activadas.

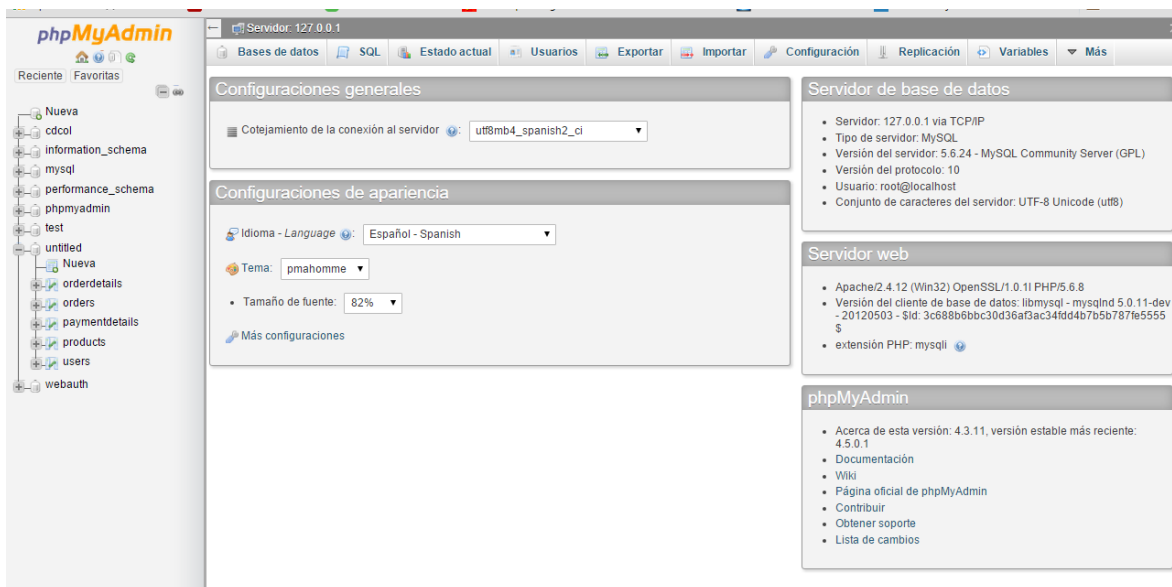
#### **Implicación en el proyecto**

Xampp se ha utilizado con dos cometidos:

- Servidor Apache que interpreta el código php de nuestra página.
- Base de datos MySQL.



Remarcar que se cambiaron los puertos del servidor Apache (4430 y 8080) para evitar incompatibilidades con los servicios instalados en la computadora.



## 3.2 Sublime Text [23]



### ¿Qué es?

Sublime Text es un editor de texto y editor de código fuente que destaca por su fácil uso, atractiva apariencia y ventajas a la hora del desarrollo y diseño web.

Es gratuito aunque no es software libre o de código abierto, aunque se puede usar de forma ilimitada.

### Implicación en el proyecto

Sublime Text trae un montón de opciones interesantes a la hora de crear documentos relacionados con el desarrollo o diseño web, por ejemplo permite la visualización de dos archivos a la vez, la edición multi línea de texto, plugins para facilitar la creación de código con tecnologías como AngularJS, opciones de autocompletar, fácil lectura de código web, etc. Todo el código de la parte del **cliente** se ha escrito usando Sublime Text.

```
65 <!-- PRODUCTOS-->
66
67 <div class="col-xs-3" dir-paginate="x in products |
68 filter:searchText | filter:categoryFilter | filter:typeFilter |
filter:priceFilter | itemsPerPage: 6">
69
70 <div class="pricing-box-alt" >
71
72 <div class="pricing-heading">
73 <a href="shop.php#product-details" ng-click="
74 setProductIdToShow(x.productId);setCategoryToShow(x.
category);"></a>
75 <h4>{{ x.productname }}</h4>
76 </div>
77 <div class="product-quantity"></div>
78 <div class="pricing-terms">
79 <h6>{{ x.price }}€</h6>
80 </div>
81 <div class="pricing-content">
82 </div>
83 <div class="pricing-action">
84 <ngcart-addtocart id="{{ x.productId }}" name="{{ x
85 .productname }}" price="{{ x.price }}" quantity="1"
quantity-max="30" data="item">comprar</ngcart-
86 addtocart>
87 </div>
88 </div>
89 </div>
90 </div>
91 </div>
92 </div>
93 </div><!-- PRODUCTS-->
94 </div>
95 <div class="pull-right" style="margin-right:2em;"><dir-paginat
96 ion-controls></dir-paginat
97 ion-controls></div>
98
99 <div class="row">
100 <div class="col-md-12">
101 <div class="col-xs-offset-10 col-xs-4">
102 <div class="header-order-details">Checkout</div>
103 <div class="row">
104 <div class="col-md-6">
```

```
1 (function () {
2 'use strict';
3
4 angular.module('app')
5 .factory('Auth', ['$http', '$localStorage', 'urls', function ($h
6
7 function urlBase64Decode(str) {
8 var output = str.replace('-', '+').replace('_', '/');
9 switch (output.length % 4) {
10 case 0:
11 break;
12 case 2:
13 output += '==';
14 break;
15 case 3:
16 output += '=';
17 break;
18 default:
19 throw 'Illegal base64url string!';
20 }
21 return window.atob(output);
22
23
24 return {
25 signup: function (data, success, error) {
26 $http.post(urls.BASE_API_USERS + 'signup', data).suc
27
28 },
29 signin: function (data, success, error) {
30 $http.post(urls.BASE_API_USERS + 'signin', data).suc
31
32 },
33 logout: function (success) {
34 delete $localStorage.token;
35 delete $localStorage.myOrders;
36 delete $localStorage.orderDetails;
37 delete $localStorage.orderId;
38 delete $localStorage.userInfo;
39 delete $localStorage.productsOrderDetail;
40 delete $localStorage.productsId;
41 delete $localStorage.paymentInfo;
42 delete $localStorage.orders;
43 delete $localStorage.status;
44 success();
45
46 },
47 update: function (data, success, error) {
48 $http.put(urls.BASE_API_USERS + 'update', data).succe
49
50 },
51 myOrders: function (success, error) {
52 $http.get(urls.BASE_API_ORDERS + 'userId').success(su
53
54 },
```

### 3.3 Eclipse Mars[24]

#### ¿Qué es?

**Eclipse** es un programa informático compuesto por un conjunto de herramientas de programación de código abierto multiplataforma. Esta plataforma, típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE), como el IDE de Java llamado *Java Development Toolkit* (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse).

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Eclipse fue liberado originalmente bajo la Common Public License, pero después fue relicenciado bajo la Eclipse Public License. La Free Software Foundation ha dicho que ambas

licencias son licencias de software libre, pero son incompatibles con Licencia pública general de GNU (GNU GPL). Podemos descargar Eclipse desde su página oficial.

## Principales características [25]

**Perspectivas, editores y vistas:** en Eclipse el concepto de trabajo está basado en las perspectivas, que no es otra cosa que una preconfiguración de ventanas y editores, relacionadas entre sí, y que nos permiten trabajar en un determinado entorno de trabajo de forma óptima.

**Gestión de proyectos:** el desarrollo sobre Eclipse se basa en los proyectos, que son el conjunto de recursos relacionados entre sí, como puede ser el código fuente, documentación, ficheros configuración, árbol de directorios, etc. El IDE nos proporcionará asistentes y ayudas para la creación de proyectos. Por ejemplo, cuando creamos uno, se abre la perspectiva adecuada al tipo de proyecto que estemos creando, con la colección de vistas, editores y ventanas preconfigurada por defecto.

**Depurador de código:** se incluye un potente depurador, de uso fácil e intuitivo, y que visualmente nos ayuda a mejorar nuestro código. Para ello sólo debemos ejecutar el programa en modo depuración (con un simple botón). De nuevo, tenemos una perspectiva específica para la depuración de código, la **perspectiva depuración**, donde se muestra de forma ordenada toda la información necesaria para realizar dicha tarea.

**Extensa colección de *plug-ins*:** están disponibles en una gran cantidad, unos publicados por Eclipse, otros por terceros. Al haber sido un estándar de facto durante tanto tiempo (no el único estándar, pero sí uno de ellos), la colección disponible es muy grande. Los hay gratuitos, de pago, bajo distintas licencias, pero casi para cualquier cosa que nos imaginemos tenemos el plug-in adecuado.

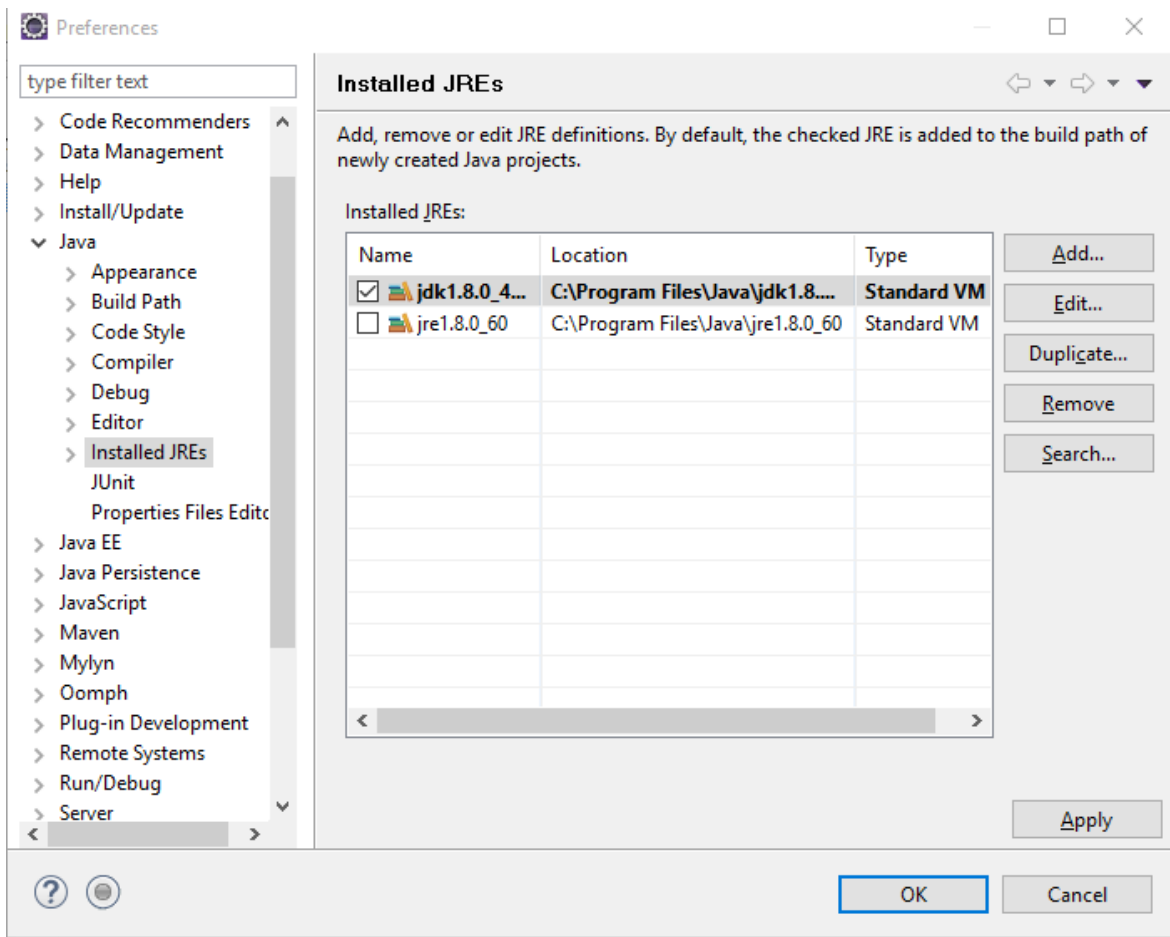
## Configuración

Eclipse Mars necesita que se instalen y configuren una serie de componentes:

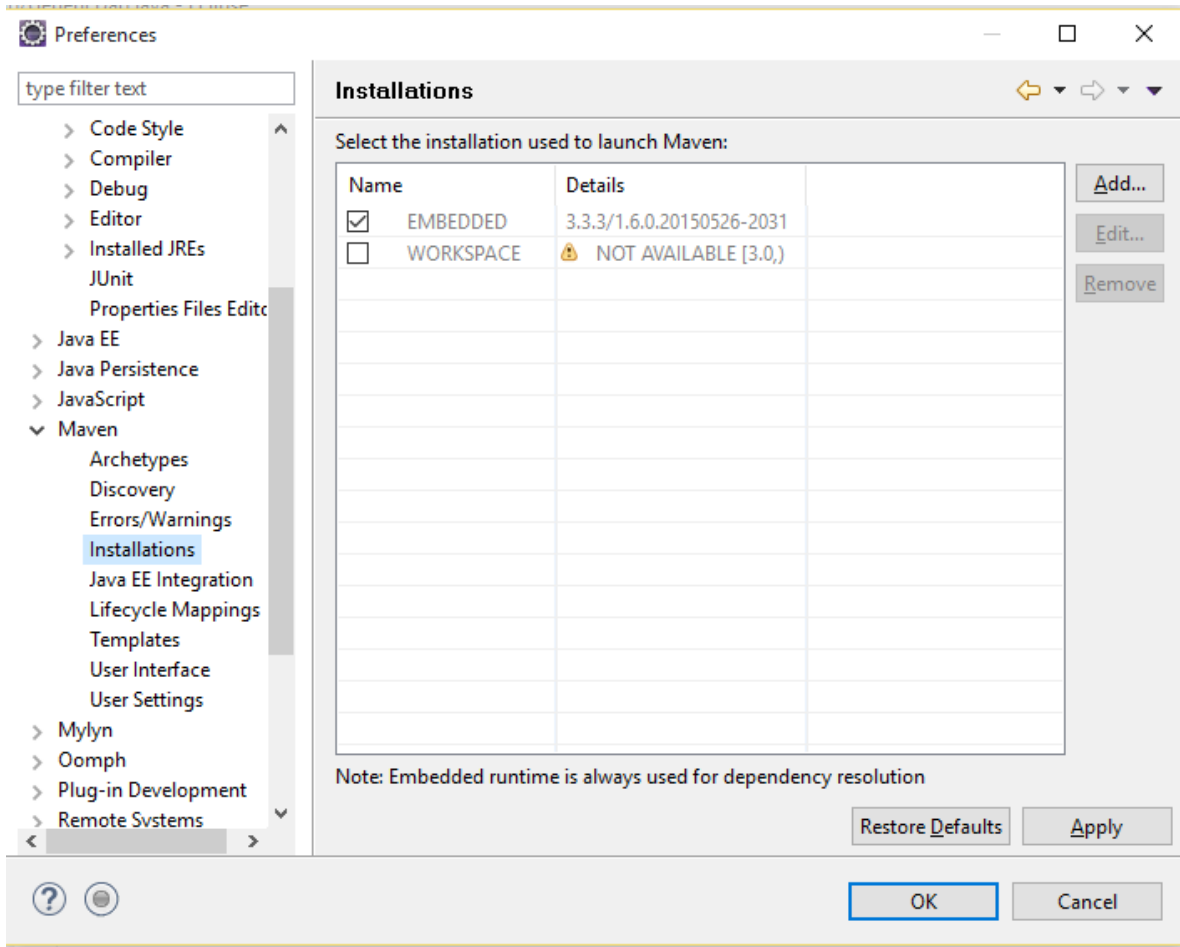
**-Java SE Development Kit:** El **JDK** es un entorno de desarrollo para crear aplicaciones, applets y componentes utilizando el lenguaje de programación Java. El JDK incluye herramientas útiles para desarrollar y probar programas escritos en el lenguaje de programación Java y se ejecuta en la plataforma Java.

Podemos encontrar los archivos de descarga aquí <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>.

Para configurarla en Eclipse, pulsamos en la ventana Window->Preferences->Java->Installed JREs y agregamos desde ahí la librería que acabamos de instalar como Standard VM.



**-Maven:** En nuestra versión de Eclipse Maven viene embebido, por lo tanto no será necesario instalarlo.



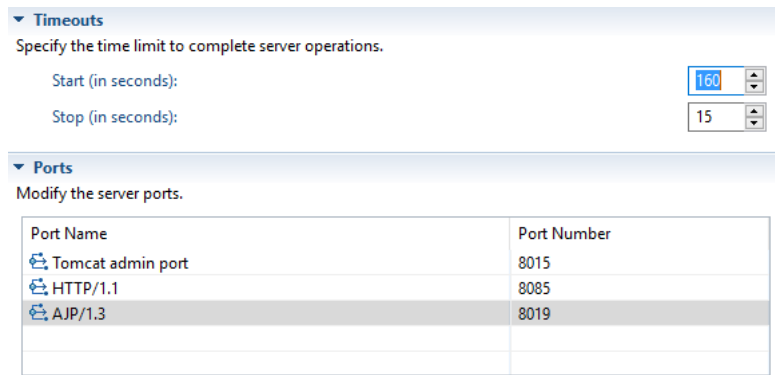
- **Apache Tomcat 8.0:** El servidor apache se va a encargar de cargar nuestra aplicación y desplegarla. Podemos descargar desde aquí el servidor <http://tomcat.apache.org/download-80.cgi>

Una vez descargado lo instalamos, desde Eclipse Mars pinchamos en Window->Show View->Server.



Abajo habrá aparecido una ventana llamada Servers, pinchamos en el texto que nos aparece para agregar un nuevo servidor, elegimos Tomcat v8.0 Server, Next, pinchamos en nuestra aplicación y add all, finalmente pinchamos finish. Le vamos a cambiar los puertos usados y los timeouts por los que aparecen en la siguiente imagen, para evitar conflictos.





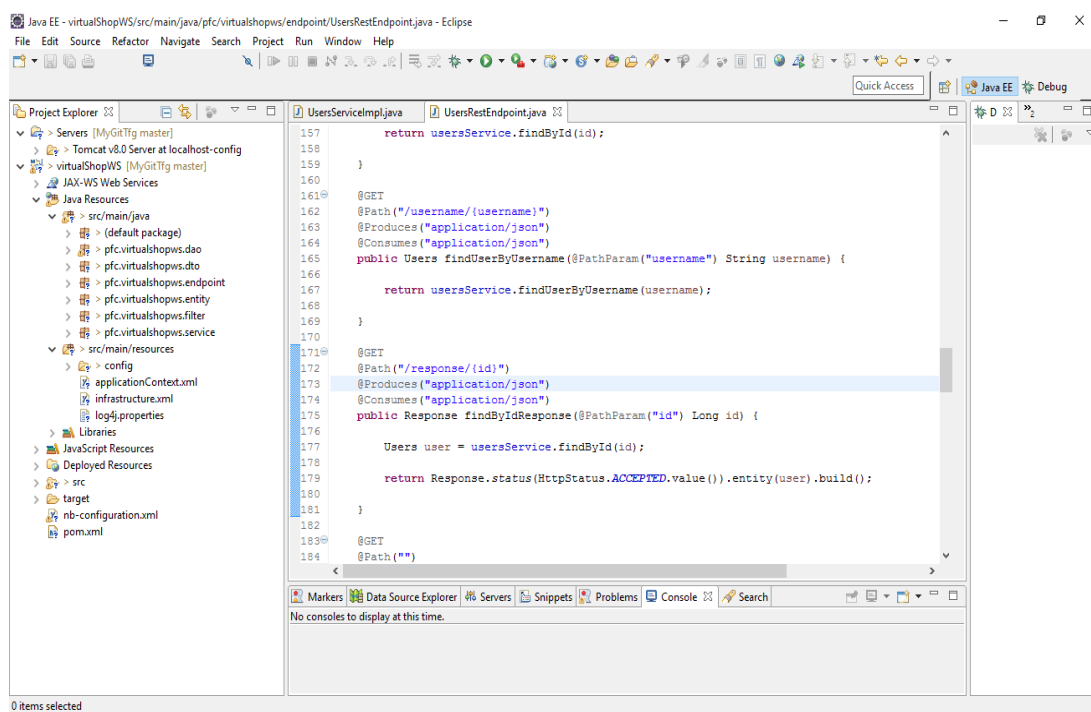
Para ejecutar el servidor pinchamos en el servidor y en el tercer botón empezando por la izquierda.



## Implicación en el proyecto

Todo el código de la parte del **servidor** está escrito sobre la versión **Mars** del **IDE Eclipse**, pertenece al tipo **J2EE** (Java Platform Enterprise Edition), esta versión está enfocada al **desarrollo** y **ejecución** de aplicaciones web en el lenguaje de programación **Java**.

Eclipse Mars J2EE, estructura de un proyecto web estándar:



## 4. Desarrollo del Cliente

### 4.1 Objetivo

El objetivo que se ha perseguido a la hora de diseñar y desarrollar el cliente, ha sido el de realizar una interfaz para el usuario que fuese clara, rápida y eficaz

UNTITLED

[HOME](#) [USER PROFILE](#) [SHOP](#) [CART](#)

[LOGOUT](#)



El cliente puede navegar por las diferentes secciones de la página: Loguearse o registrarse, consultar sus pedidos, actualizar su información de usuario, consultar la información acerca de un producto, añadir ítems a tu carrito, confirmar su pedido, etc.



*Life's short. Anything could happen, and it usually does, so there is no point in sitting around thinking about all the ifs, ands and buts.*

— Amy Winehouse

## Summary

16 items  
\$191.50



Type


- CD
- VINYL
- Digital Music

category

- Indie & Alternative

price


- 10
- 13
- 7
- 2



**English Graffiti**

10€


1  Item added!. Remove



**One**

13€

4  Item added!. Remove



**7th or St. Tammany**

7€

2  Item added!. Remove

## UNTITLED

[HOME](#) [USER PROFILE](#) [SHOP](#) [CART](#)

### User Info

Name: dani  
Billing Address: dani  
Telephone: 674855605  
Email: danychi@gmail.com  
City: dani  
Postal Code: 1234

Is your Info ok?  
You can update it from your user profile [Goto!](#)

### Payment Method



## 4.2 Tecnologías Implicadas

El cliente se ha hecho sobre seis tecnologías:

Se ha montado sobre un template de **Bootstrap** que se puede encontrar en el siguiente enlace <http://bootstraptaste.com/demo/Moderna/>. De este template se reciclaron varias secciones y se suprimieron otras quedando como resultado final nuestra página.

Se ha reutilizado principalmente la cabecera (**header**), pie (**footer**), el **slider** de la página **home** y algunos estilos predeterminados que trae Bootstrap como tamaño de letra, fuente, efectos, tamaño para botones, etc.

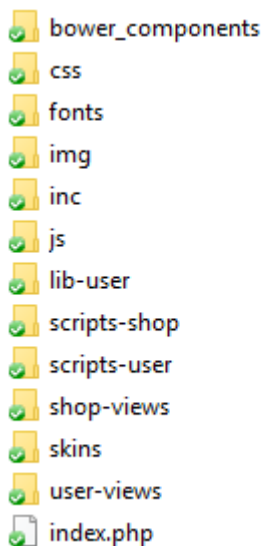
Con **HTML** se han creado las vistas necesarias para las diferentes partes de la página, como por ejemplo la lista de productos de la tienda, un formulario para actualizar la información del usuario o una vista para ver nuestros pedidos más recientes.

Con **CSS3** se ha modificado el estilo de botones, barras de navegación, colores principales de la página, estilo de letra, media queries para que se adapte según el tamaño del dispositivo, etc.

**PHP** se ha usado simplemente para reutilizar código que estaba presente en más de una parte de la página y así incluirlo cuando fuese necesario.

La clave del cliente ha sido la utilización de **AngularJS** combinada con la potencia de la tecnología **JavaScript**; de esta manera se han podido manejar vistas, hacer peticiones de recursos al servidor, hacer más interactiva la página, etc.

## 4.3 Estructura



El cliente está estructurado de la siguiente manera:

**-Bower\_components:** Dentro de esta carpeta se encuentran los archivos que se han instalado mediante bower (una herramienta que nos ayuda a instalar fácilmente librerías y archivos), por ejemplo la directiva ng-cart.

**-css:** Aquí se encuentran las hojas de estilo usadas en nuestra web.

**-fonts:** Conjunto de fuentes e iconos que utiliza bootstrap que pueden ser utilizados en nuestra página desde HTML.

**-img:** Imágenes que se usan dentro de la página como las de los productos, sliders, index, etc.

**-js:** En esta carpeta se almacenan todos los archivos del tipo JavaScript, como las librerías de jQuery, bootstrap, o archivos JavaScript que se han necesitado incluir a lo largo de la creación del proyecto.

**-lib-user:** Librerías descargadas usadas en AngularJS.

**-scripts-shop o scripts-user:** Siguiendo una filosofía de creación de aplicaciones de AngularJS, se han separado en esta carpeta la app, el controller y el service de la aplicación de Angular. Todos los archivos de esta carpeta utilizan la extensión .js (JavaScript).

**-shop-views:** Aquí se encuentran todas las vistas de la tienda.

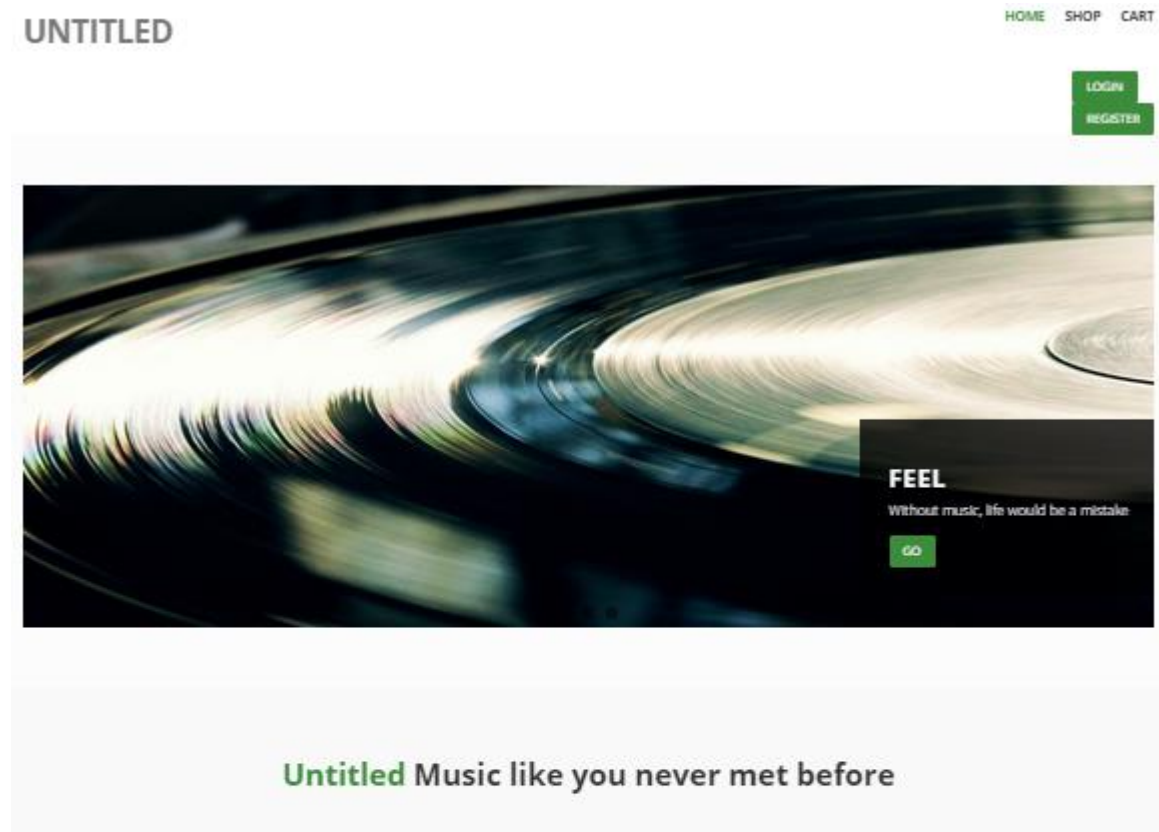
**-users-views:** Aquí se encuentran todas las vistas pertenecientes al apartado usuario.

## 4.4 Explicación

El cliente se podría dividir en 3 secciones esencialmente, Home, Users y Shop.

### 4.4.1 Home

Home es la primera vista que ve el usuario cuando entra a la página, desde ella puede acceder rápidamente a las diferentes partes de la página, registrarse, loguearse, etc.



#### SHOP

The best selection of music. You will find all genres, with high quality in all our products.

Go now!



#### DIGITAL MUSIC

Get your favorite music has never been easier. Download it instantly!

Go now!



#### NEWS

Taste the last music, the best selection.

Go now!

Lo primero que realiza Home es hacer **auto-bootstrap** (inicialización de una app) a la aplicación de Angular y asignarle un controller a esta app (posteriormente se explicará en detalle en qué consiste).

La página de inicio mediante la sentencia **“include”** introduce en la página dos documentos externos, correspondientes con el encabezado y pie de la página.

```
<?php include 'inc/encabezado-index.php' ?>
```

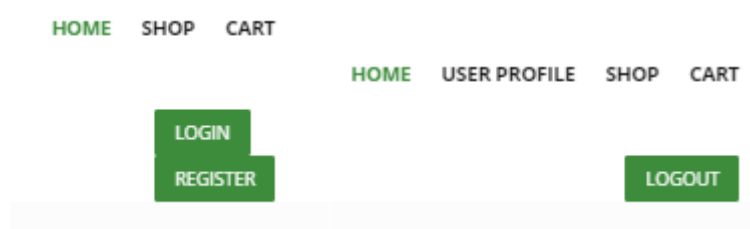
En el **pie** simplemente se encuentran los scripts necesarios para AngularJS y Bootstrap, que se incluyen al final de la página para que ésta cargue más rápido y un **<footer>** donde se ubican los derechos de autor y links a redes sociales.

El **encabezado** por otra parte, dentro de la etiqueta **<head>** incluye las hojas de estilo que va a usar la página y se define la cabecera de la página con la nueva etiqueta introducida en HTML5 **<header>**.

Se ha creado una cabecera personalizada, haciendo uso de la directiva **ng-hide** y **ng-show**, evaluamos una variable **token** (se hablará de ella durante el proyecto, sirve para el control de sesión) que valdrá true si hay una **sesión** iniciada y false en caso contrario. Ng-hide muestra el código HTML si la variable evaluada vale false, ng-show funciona de manera opuesta.

En caso de que no exista el token, será necesario **registrarse** o **loguearse** para poder comprar en la tienda o acceder a la sección de usuario.

En estas imágenes se pueden ver las diferencias de la cabecera en función de si existe el token o no:



Home está formado por un slider que traía el **template** Moderna de Bootstrap. Su uso es sencillo, se usa la clase **flexslider** para definir que los ítems que van dentro de un div van a

tener el efecto de un **slider**. Posteriormente se crea una lista tipo `<ul>` dándole el nombre de la clase “**slides**” y dentro se van creando los elementos de la lista con `<li>` con el contenido de cada **slide**; con la clase “**flex-caption**” dotamos a cada slide de un cuadro de texto con un enlace a una sección de la página. En la siguiente parte vemos un texto que se ha formateado adecuadamente para que resulte llamativo.

A continuación encontramos tres imágenes acompañadas de un texto, para conseguir que se muestren en tres columnas se ha usado la clase “`col-lg-4`”, ya que cada fila (**row**) está dividida en 12 **columns**, por lo tanto para mostrar tres elementos tenemos que dividir esta fila en tres columnas de tamaño cuatro. Remarcar que a estas imágenes se le añadió un efecto sencillo que permite Bootstrap mediante JavaScript:

```
.img-index:hover{
  transform:scale(1.1);
}

.img-index{
  transition: all 0.3s ease 0s;
  width:100%;
  max-height: 215px;
}
```

Al pasar el cursor sobre cualquiera de las tres imágenes, la imagen aumenta su tamaño en una transición de 0.3 segundos, al sacar el cursor de la imagen ésta vuelve a su tamaño original.

Al final de la página nos encontramos con una imagen acompañada de una cita, usamos `<blockquote>` para crear la cita.



*Life's short. Anything could happen, and it usually does, so there is no point in sitting around thinking about all the ifs, ands and buts.*  
— Amy Winehouse.



#### 4.4.2 Users[26]

La sección Users está formada fundamentalmente por login, register y user profile.

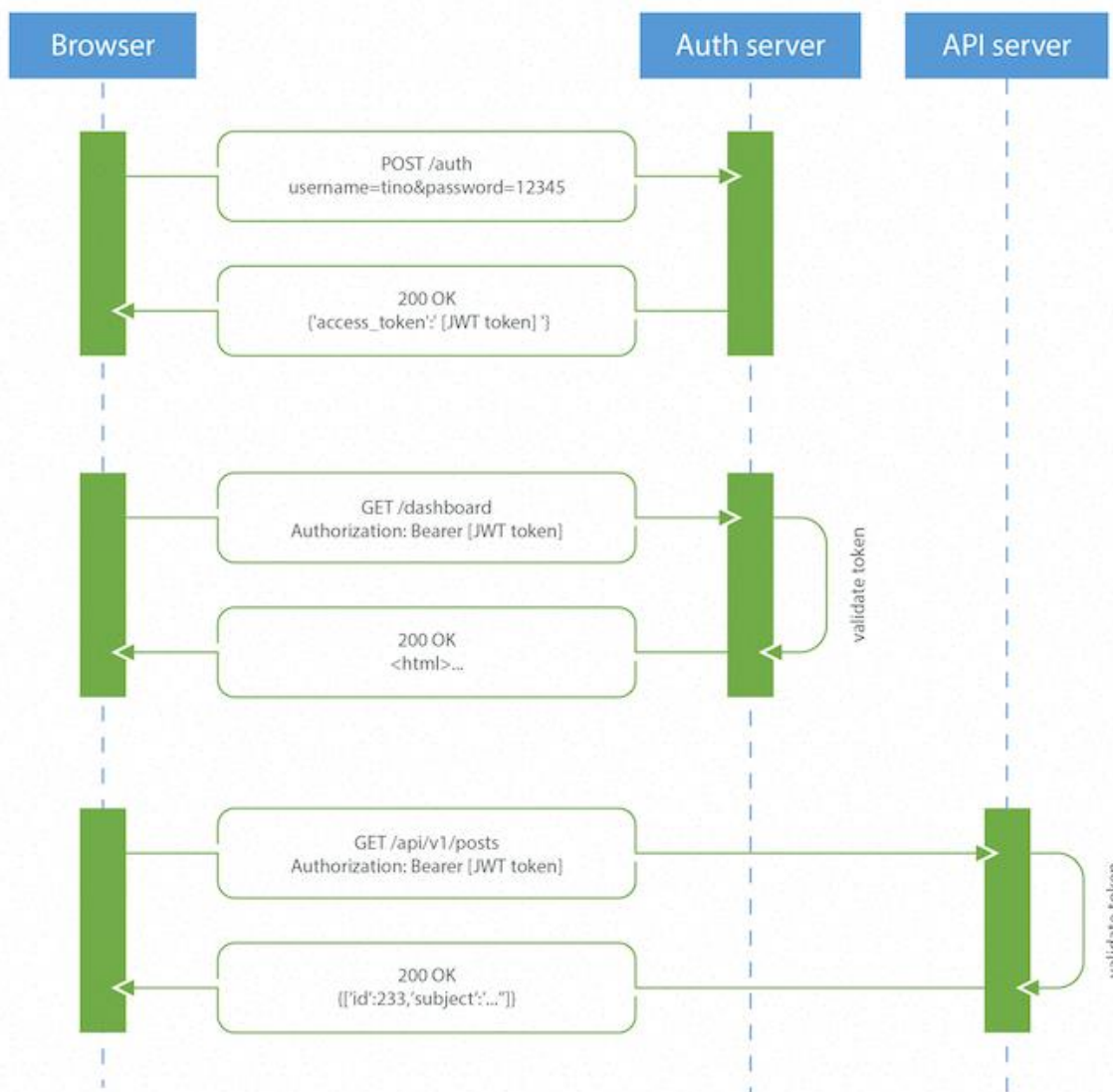
Dentro del perfil de usuario se puede actualizar nuestra información, consultar los pedidos, ver su estado, ver los detalles de un pedido, borrar alguno y cerrar la sesión.

La aplicación de Users está basada en el código y metodología de trabajo de este ejemplo: <http://www.toptal.com/web/cookie-free-authentication-with-json-web-tokens-an-example-in-laravel-and-angularjs>

#### Sesión

La autenticación es una de las partes más importantes de cualquier aplicación web. Durante décadas, las cookies y la autenticación basada en servidor fueron la solución más sencilla. Sin embargo con la aparición de los sistemas móviles y las Single Page Applications (SPA) puede resultar más complicado, y se exige un mejor enfoque.

Para la autenticación se ha usado una versión simplificada de la autenticación basada en JSON Web Token (JWT), en la siguiente figura se puede visualizar el funcionamiento de la autenticación entre el cliente y servidor:



Un navegador o cliente móvil hace una petición al servidor de autenticación que contiene la información de inicio de sesión de usuario. El servidor de autenticación genera un nuevo JWT y lo devuelve al cliente. En cada solicitud de un recurso limitado, el cliente envía el token de acceso en el query string o Authorization header. El servidor valida el token y si es válido, devuelve el recurso para el cliente.

## *Ventajas*

**Stateless**, más fácil de escalar: El token contiene toda la información para identificar al usuario, eliminando la necesidad de mantener un estado de la sesión. Si se utiliza un equilibrador de carga, podemos pasar el usuario a cualquier servidor, en lugar de estar atado al mismo servidor en el que hizo login.

**Reutilización:** Podemos tener muchos servidores distintos, que se ejecutan en múltiples plataformas y dominios, reutilizando el mismo token para autenticar al usuario.

**Seguridad:** Ya no estamos utilizando cookies, no tenemos que protegernos contra el cross-site request forgery (CSRF). Aun así tenemos que cifrar nuestros tokens en caso de tener información sensible en ellos, y transmitir los tokens a través de HTTPS para prevenir los ataques man-in-the-middle.

## Estructura de Users

```
|-- lib/
  |-- ngStorage.js
|-- user-views/
  |-- account-settings.html
  |-- home.html
  |-- logout.html
  |-- my-orders.html
  |-- order-details.html
  |-- home.html
  |-- signin.html
  |-- signup.html
  |-- user.php
  |-- user-profile.html
|-- scripts-user/
  |-- app-user.js
  |-- controllers.js
  |-- services.js
```

Users está formado por las vistas (user-views), los scripts necesarios para que funcione la app correspondiente a user y la librería ng-Storage que se utilizará para guardar datos en el Local Storage (a modo de caché).

## Cómo funciona la aplicación Users de AngularJS

Para entender cómo funciona la aplicación necesitamos explicar qué hace y para qué sirve app-user.js, controller.js y services.js

### *App-user.js*

App-user.js define el módulo principal de la aplicación y carga todas las configuraciones necesarias para correr la aplicación.

Lo primero que encontramos es la definición del módulo que llamamos “app”, con este nombre nos referiremos a él cuando llamemos a la aplicación con la directiva ng-app. Lo siguiente que vemos es como se incluyen dos librerías en el módulo, **ngStorage** y **ngRoute**, esta última sirve para crear las vistas de la aplicación que se mostrarán instantáneamente sin necesidad de recargar la página.

```
angular.module('app', [  
  'ngStorage',  
  'ngRoute'  
)
```

La siguiente parte del código corresponde a la definición de constantes, en este caso son todo urls, que se utilizarán habitualmente en el servicio para hacer llamadas al servidor

```
.constant('urls', {  
  BASE: 'http://localhost:8080',  
  BASE_API_USERS: 'http://localhost:8085/virtualShopWS/api/rest/users/',  
  BASE_API_ORDERS : 'http://localhost:8085/virtualShopWS/api/rest/orders/',  
  BASE_API_PAYMENT_DETAILS : 'http://localhost:8085/virtualShopWS/api/rest/paymentDetails/',  
  BASE_API_PRODUCTS : 'http://localhost:8085/virtualShopWS/api/rest/products/',  
  BASE_API_ORDER_DETAILS : 'http://localhost:8085/virtualShopWS/api/rest/orderDetails/'  
})
```

A continuación se inyectan los servicios routeProvider y httpProvider, el primero sirve para que funcione ngRoute y el segundo para realizar la configuración necesaria para evitar problemas con el CORS.

```
.config(['$routeProvider', '$httpProvider', function ($routeProvider, $httpProvider) {  
  $httpProvider.defaults.crossDomain=true;  
  $httpProvider.defaults.useXDomain = true;  
  $httpProvider.defaults.withCredentials = true;  
})
```

Posteriormente se configuran las vistas. Tiene un funcionamiento muy sencillo, cuando se pide una ruta sobre user.php, por ejemplo /signin entonces se muestra una vista y se le asigna un controlador. Con la directiva `<div ng-view></div>` ngRoute crea la vista sobre ese div.

```
$routeProvider.  
  when('/', {  
    templateUrl: 'home.html',  
    controller: 'HomeController'  
  }).  
  when('/signin', {  
    templateUrl: 'signin.html',  
    controller: 'HomeController'  
  }).  
  when('/signup', {  
    templateUrl: 'signup.html',  
    controller: 'HomeController'  
  }).  
  when('/user-profile', {  
    templateUrl: 'user-profile.html',  
    controller: 'HomeController'  
  }).  
  when('/account-settings', {  
    templateUrl: 'account-settings.html',  
    controller: 'HomeController'  
  }).  
  when('/my-orders', {  
    templateUrl: 'my-orders.html',  
    controller: 'HomeController'  
  }).
```

Después, se configura el request para que en cada petición se envíe el token en caso de que exista en nuestro Local Storage. Si se recibe un “responseError” con un código de estado HTTP 401(no autorizado) o 403 (prohibido), se borra automáticamente el token y nos envía a la pantalla de login.

```
$httpProvider.interceptors.push(['$q', '$location', '$localStorage', function ($q, $location, $localStorage)  
  return {  
    'request': function (config) {  
      config.headers = config.headers || {};  
      if ($localStorage.token) {  
        config.headers.Authorization = $localStorage.token;  
      }  
      return config;  
    },  
    'responseError': function (response) {  
      if (response.status === 401 || response.status === 403) {  
        delete $localStorage.token;  
        $location.path('/signin');  
      }  
      return $q.reject(response);  
    }  
  }  
});
```

Finalmente se inicializa la aplicación con `.run`, invocando a los servicios necesarios para que funcione la app como `scope`, `location` y `localStorage`.

```
}).run(function($rootScope, $location, $localStorage) {
  $rootScope.token=false;
  if($localStorage.token){$rootScope.token =true;}

  $rootScope.$on( "$routeChangeStart", function(event, next) {
    if ($localStorage.token == null) {
      if ( next.templateUrl === "user-views/signin.html") {
        $location.path("/signin");
      }
    }
  });
});
```

Se inicializa una variable en el scope llamada `token` (diferente al `token` en el `localStorage`) con valor `false` y se ejecuta una sentencia `if` comprobando si existe el `token` en la caché, en ese caso el `token` del scope valdrá `true`; esta variable se usa en el **encabezado** para comprobar si el usuario está logueado o no. Por defecto si no existe el `token` y queremos acceder a cualquier vista de usuarios, nos enviará a la pantalla de login.

### *Controller.js*

En el controller se encuentran todas las funciones a las que tiene acceso el scope, éstas invocan al servicio `Auth` cuando es necesario.

```
angular.module('app')
  .controller('HomeController', ['$rootScope', '$scope', '$location', '$filter', '$localStorage', 'Auth',
    function ($rootScope, $scope, $location, $filter, $localStorage, Auth) {
```

El controlador se conecta al módulo “app”, se define el nombre y se inyectan todos los servicios que serán necesarios en las funciones del controller, como `rootScope` (funciona de forma global en la aplicación, sólo hay uno y sigue el mismo ciclo de vida que la aplicación), `scope`, `location` (para redireccionar a una url), `filter` (para utilizar filtros sencillos de angular), `localStorage` y `Auth`, que se ha definido en `Service.js`.

Se ha seguido una filosofía muy marcada a la hora de diseñar el controller:

- 1- Creación de una función en el controller.
- 2- Llamada al servicio.
- 3- Si hay éxito en la petición, se llama a una función del tipo `successNombreFunción`.
- 4- Se crea una variable en el `localStorage` para almacenar el resultado, usado en el cliente.

Veamos un ejemplo con la función signin:

- 1- Se crea una variable llamada **formData** que tiene formato JSON, en ella se definen un email y una contraseña.
- 2- Se llama a la función signin del servicio Auth, se le pasa como argumentos formData, una función en caso de que haya éxito y se crea una función en caso de que haya error en la respuesta del servidor.
- 3- Si hay éxito, entonces se llama a la función successAuth, en la que se crea el token con la respuesta del servidor y se redirecciona a la página de inicio.
- 4- En caso de error, se mostraría un mensaje “Invalid credentials.”.

```
$scope.signin = function () {  
    var formData = {  
        email: $scope.email,  
        password: $scope.password,  
    };  
  
    Auth.signin(formData, successAuth, function () {  
        $rootScope.error = 'Invalid credentials.';  
    })  
};
```

```
function successAuth(res) {  
    $localStorage.token = res;  
    window.location = "/";  
}
```

Ahora se explicará el resto de funciones creadas en el controller, necesarias para el funcionamiento de la aplicación Users:

- **Signup**: Esta función sirve para registrar al cliente, recopila sus datos y se los pasa a la función del servicio correspondiente, en caso de éxito llama a successAuth, que actúa de igual manera que en signin. En caso de error muestra un mensaje diciendo que no ha sido posible registrar al usuario.

```
$scope.signup = function () {
```

-**Update**: De forma similar a signup, recibe unos datos del usuario para poder actualizar su perfil, se los pasa al servicio, si hay éxito actualiza el valor del token (ya que si hemos cambiado la información del usuario, también habrá que actualizar el token en el servidor) y actualiza la página mostrando un mensaje de “Updated”. Si ha ocurrido un error, muestra un mensaje informativo.

```
$scope.update = function () {
```

```
function successUpdate(res){
  localStorage.token = res;
  window.location.reload();
  window.alert("Updated!");
}
```

**-Logout:** Sirve para cerrar la sesión del usuario, llama al servicio para que se encargue de esta tarea.

```
$scope.logout = function () {
```

**-myOrders:** Esta función se encarga de inicializar un array donde se guardarán los estados de los pedidos posteriormente y de llamar a la función myOrders del servicio. En caso de que reciba una respuesta correcta del servidor, se llama a successGetOrders, se guarda en el scope y en el localStorage los pedidos asignados a un usuario con el nombre **orders**.

```
$scope.myOrders = function () {
  localStorage.status= [];
  $rootScope.status = [];
  Auth.myOrders(successGetOrders, function (res) {
    $rootScope.error = res.error || 'No orders to display.';
  })
};
```

```
function successGetOrders(res){
  localStorage.orders = res;
  $rootScope.orders = localStorage.orders;
}
```

**-getOrders:** Con esta función se pueden recuperar los pedidos asignados a un usuario que se encuentran almacenados en el localStorage, para guardarlos en el scope, así se evita volver a hacer una llamada a myOrders; también porque cuando se recarga la página los valores en el scope se reinician.

```
$scope.getOrders = function () {
  if(localStorage.orders!=null){
    $rootScope.orders = localStorage.orders;
  }
};
```

**-setOrderIdToShow:** Esta función sirve para almacenar el orderId de un pedido, que se utilizará para ver los detalles del mismo por ejemplo. Recibe un orderId en la cabecera de la función y se guarda en el localStorage en caso de que no sea nulo.



```
$scope.setOrderIdToShow = function (orderId) {  
  if(orderId!=null){  
    $localStorage.orderId = orderId;  
  }  
};
```

**-getOrderIdToShow:** Esta función recupera el valor que se almacenó en la anterior función, guardándola en el scope, previa verificación de que no se recibe un valor nulo.

```
$scope.getOrderIdToShow = function () {
```

**-getUserInfo:** Esta función se encarga de llamar al servicio para que haga una petición al servidor y recupere la información relacionada con nuestro usuario (el que está asignado al token). En caso de que la llamada sea exitosa se llama a `successGetUserInfo`, que guardará la respuesta del servidor en nuestro `localStorage` con el nombre de **userInfo**.

```
$scope.getUserInfo = function () {
```

**-recoverUserInfo:** Con esta función recuperamos la información obtenida en la función anterior, para que sea accesible desde el scope.

```
$scope.recoverUserInfo = function(){  
  $rootScope.userInfo = $localStorage.userInfo;  
}
```

**-getProductsIdByOrderId:** Con el `orderId` que se ha guardado con `getOrderIdToShow`, se realiza una llamada al servicio para que haga una petición al servidor con este ID, para así obtener todos los **IDs** de los **productos** relacionados con un **pedido**. En caso de que se reciba una respuesta correcta del servidor, se llama a `successGetProductsId`, esta función creará un formulario tipo JSON con los `productsId` obtenidos, hará una llamada al servicio para obtener la información de todos los productos que coincidan con los IDs que hemos enviado en la petición. Si la respuesta es exitosa, se llama a `successGetProductsByProductsId` guardando todos los productos en el `localStorage` y scope. Aquí no hace falta definir dos funciones diferentes, una para establecer la información de los productos recuperados del servidor y otra para recuperar la información de la llamada cuando se necesite la información, ya que esta información será necesaria sólo en este apartado de la página y por lo tanto no es información reusable.

```
$scope.getProductsIdByOrderId = function () {
```

```
function successGetProductsId(res){
  $localStorage.productsId = res;

  var formData = $localStorage.productsId ;
  Auth.getProductsByProductsId(formData,successGetProductsByProductsId , function (res) {
    $rootScope.error = res.error || 'Failed to get the Products.';
  })
}
```

```
function successGetProductsByProductsId(res){
  $localStorage.productsOrderDetail = res;
  $rootScope.productsOrderDetail = $localStorage.productsOrderDetail;
}
```

**-getPaymentInfo:** Esta función llama al servicio para obtener los detalles de pago de un pedido, se le pasa el orderId recogido anteriormente; en caso de éxito se llama a **successGetPaymentInfo**, función que guarda en el localStorage y el scope la respuesta del servidor con nombre **paymentInfo**.

```
$scope.getPaymentInfo = function () {
```

**-getOrderDetailsByOrderId:** En esta función se obtienen los detalles de un pedido haciendo una llamada al servicio enviando el orderId del pedido, en caso de que haya éxito se llama a **successGetOrderDetailsByOrderId** para guardar los detalles del pedido en el almacenamiento local y scope con nombre **productsOrderDetail**.

```
$scope.getOrderDetailsByOrderId = function () {
```

**-deleteOrder:** Esta función recibe un orderId (usando la directiva ng-click en el cliente) y su cometido es llamar al servicio para que borre un pedido, si se ha ejecutado la petición correctamente se invoca a **successDeleteOrder** que de forma similar a successUpdate recarga la página y muestra una alerta diciendo que el pedido se ha borrado correctamente.

```
$scope.deleteOrder = function (orderId) {
```

```
function successDeleteOrder(res){
  window.location.reload();
  window.alert("Order deleted Successfully!");
}
```

**-getStatus:** Esta función recibe un statusId (mediante ng-click), se encarga rellenar el array status inicializado con anterioridad con los valores de estado de cada pedido, lo hace invocando a la función getStatus del servicio Auth (que traduce un número por un código de estado de pedido), con **.push** se añade el resultado al array siempre y cuando se respete que el tamaño del array status no sea mayor que el de orders (lo cual es lógico).

```

$scope.getStatus = function (statusId) {
  if($localStorage.status[$rootScope.orders.length-1]!=null){
    $localStorage.status.push(Auth.getStatus(statusId));
  }
  $rootScope.status = $localStorage.status;
};

```

**-getSingleStatus:** Con esta función se obtiene el estado de un pedido singular llamando a la función `getStatus` del servicio, guardando esta información en el `localStorage` y `rootScope` con el nombre `ss` (se usa en los detalles del pedido).

```

$scope.getSingleStatus = function (statusId){
  $localStorage.ss = (Auth.getStatus(statusId));
  $rootScope.ss = $localStorage.ss;
}

```

### Service.js

Service.js está formado por servicios que pueden ser llamados desde cualquier aplicación de angular, simplemente inyectándolo mediante inyección de dependencias (DI).

```

angular.module('app')
  .factory('Auth', ['$http', '$localStorage', 'urls', function ($http, $localStorage, urls) {

```

Se crea como una factoría (`.factory`), se le nombra 'Auth' nombre con el que se invocará al servicio y finalmente se inyectan los servicios que va a necesitar, entre ellos destacar `$http` que es el servicio con el que se harán las llamadas al servidor y `$localStorage` que será necesario para crear/modificar variables en el almacenamiento interno.

```

  signup: function (data, success, error) {
    $http.post(urls.BASE_API_USERS + 'signup', data).success(success).error(error)
  },

```

En este ejemplo vemos lo sencillo que es hacer una llamada al servidor.

Con `$http.métodoHttp` se puede utilizar cualquier método que nos permita HTTP, dentro del método en paréntesis indicamos la dirección (path) a la que se quiere llamar, concatenando con "+" para obtener la url deseada. Si acabamos la dirección con una coma y **data** (datos que recibe el método del servicio), entonces esa información se adjuntará en el body del método HTTP, recordar que no se pueden adjuntar datos con el método GET.

Finalmente se definen las funciones a las que llamar en caso de success y error, con las funciones que reciba el servicio.

A continuación explicaremos los demás servicios que se han definido en service.js

**-signin:** Esta función hace un post a la dirección del servidor “users/signin”, enviando unos datos en el body de la petición. Sirve para registrar al usuario.

```
signin: function (data, success, error) {  
  $http.post(urls.BASE_API_USERS + 'signin', data).success(success).error(error)  
},
```

**-logout:** Se encarga de borrar todos los datos que se han creado en la sesión, relacionados con el usuario.

```
logout: function (success) {  
  delete $localStorage.token;  
  delete $localStorage.myOrders;  
  delete $localStorage.orderDetails;  
  delete $localStorage.orderId;  
  delete $localStorage.userInfo;  
  delete $localStorage.productsOrderDetail;  
  delete $localStorage.productsId;  
  delete $localStorage.paymentInfo;  
  delete $localStorage.orders;  
  delete $localStorage.status;  
  success();  
},
```

**-update:** Realiza un put a “users/update” con la información facilitada en data.

```
update: function (data, success, error) {  
  $http.put(urls.BASE_API_USERS + 'update', data).success(success).error(error)  
},
```

**-myOrders:** Hace un get a “orders/userId”.

```
myOrders: function (success, error) {  
  $http.get(urls.BASE_API_ORDERS + 'userId').success(success).error(error)  
},
```

**-getUserInfo:** Método get a “users/getUserInfo”.

```
getUserInfo: function (success, error) {  
  $http.get(urls.BASE_API_USERS + 'getUserInfo').success(success).error(error)  
},
```

**-getProductsIdByOrderId:** Método get a “ordersDetails/orderId/{data}”, sustituyendo data con el valor que recibe el método.

```
getProductsIdByOrderId: function (data, success, error) {  
  $http.get(urls.BASE_API_ORDER_DETAILS + 'orderId/' + data).success(success).error(error)  
},
```

**-getPaymentInfo:** Método get a “paymentDetails/orderId/{data}”, evaluando data de la misma manera que en la función anterior.

```
getPaymentInfo: function (data, success, error) {  
  $http.get(urls.BASE_API_PAYMENT_DETAILS + 'orderId/' + data).success(success).error(error)  
},
```

**-getProductsByProductsId:** Método post a “products/productsId”, introduciendo en el

body del request un data compuesto por un array de IDs de productos.

```
getProductsByProductsId: function (data, success, error) {
  $http.post(urls.BASE_API_PRODUCTS + 'productsId/' , data).success(success).error(error)
},
```

**-getOrderDetailsByOrderId:** Método get a “orderDetails/orderDetailsOrderId/{data}”.

```
getOrderDetailsByOrderId: function (data, success, error) {
  $http.get(urls.BASE_API_ORDER_DETAILS + 'orderDetailsOrderId/' + data).success(success).error(error)
},
```

**-deleteOrder:** Método delete a “orders/orderId”.

```
deleteOrder: function (data, success, error) {
  $http.delete(urls.BASE_API_ORDERS + 'orderId/' + data).success(success).error(error)
},
```

**-getStatus:** Esta función se encarga de devolver un código de estado según un ID de estado. Se explicó en el ejemplo del apartado 2.5.

```
getStatus: function (statusId) {
```

## Vistas de Users

|-- user-views/

    |-- account-settings.html

    |-- home.html

    |-- logout.html

    |-- my-orders.html

    |-- order-details.html

    |-- home.html

    |-- signin.html

    |-- signup.html

    |-- user.php

    |-- user-profile.html

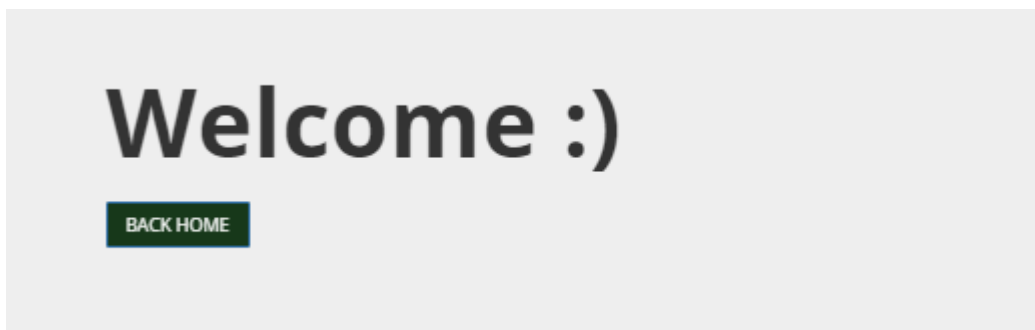
Finalmente, queda explicar las vistas del usuario y cómo funciona cada una.

La raíz de las vistas es **user.php**:

```
<body ng-app="app">
  </br>
  <div class="jumbotron">
    <div class="container">
      <div class="col-sm-8 col-sm-offset-2">
        <div ng-view></div>
      </div>
    </div>
  </div>
```

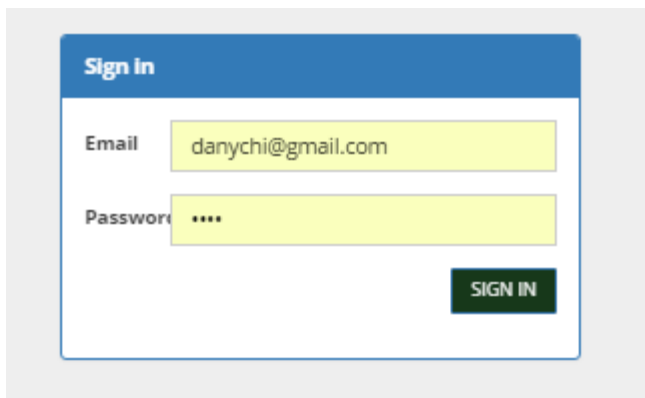
En este fichero se inicializa la aplicación con la directiva **ng-app**, se crean un par de div con diferentes clases para que tenga un estilo determinado y se inyecta la directiva **ng-view**, para crear las vistas.

### Home



Esta pantalla nos da un mensaje de bienvenida.

### Signin



Sign In es una vista sencilla que se encarga de que nuestro usuario pueda iniciar sesión, está formada por un formulario del tipo form (perteneciente a HTML) con unas pequeñas modificaciones para que funcione con AngularJS.

```
<form class="form-horizontal" role="form" ng-submit="signin()">
```

Cuando nuestro cliente pulse el botón SIGN IN, mediante la directiva ng-submit, Angular podrá recuperar la información que se ha introducido en el formulario; de esta manera se llamará a la función signin() del controller que rellenará un formulario del tipo JSON con los datos recopilados, éste a su vez llamará al servicio y el servicio al servidor, si la respuesta es correcta se habrá iniciado la sesión correctamente. Con la directiva ng-model Angular puede recopilar la información necesaria para la función.

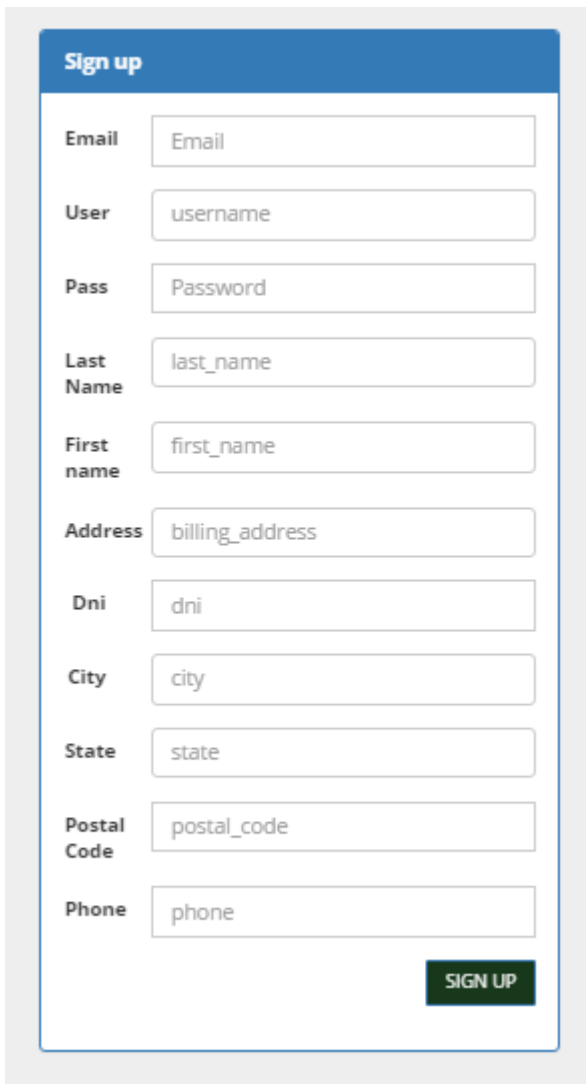
```
<div class="col-sm-10">
  <input type="email" class="form-control" id="email" placeholder="Email" ng-model="email">
</div>
```

```
$scope.signin = function () {
  var formData = {
    email: $scope.email,
    password: $scope.password,
  };
};
```

Finalmente en caso de error, se mostrará el error en un párrafo en la parte inferior a SIGN IN.

```
<p data-ng-show="error" class="error">{{ error }}</p>
```

### Signup



The image shows a 'Sign up' form with a blue header. The form contains the following fields:

- Email: Email
- User: username
- Pass: Password
- Last Name: last\_name
- First name: first\_name
- Address: billing\_address
- Dni: dni
- City: city
- State: state
- Postal Code: postal\_code
- Phone: phone

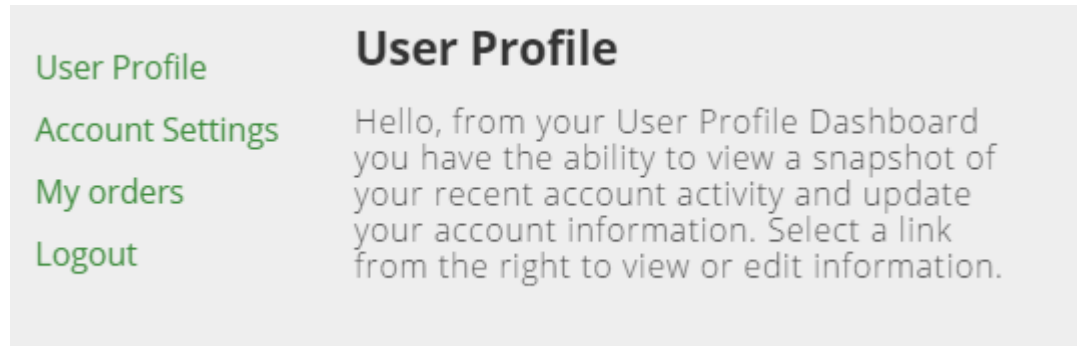
A green 'SIGN UP' button is located at the bottom right of the form.

De manera similar a Sign in, Sign up está formado por un formulario del tipo HTML que usa directivas de AngularJS (ng-model, la misma vista de errores, ng-submit, etc).

En este caso el objetivo es registrar al cliente. Al pulsar el botón SIGN UP se llama a la función signup() del controller que se encargará de que nuestro usuario quede registrado.

```
<form class="form-horizontal" role="form" ng-submit="signup()">
```

### User Profile



User Profile da la bienvenida al usuario y le muestra las opciones que tiene desde su perfil de usuario.

Tiene una barra de navegación a la izquierda que aparece en las diferentes secciones del perfil de usuario, que nos da acceso a las vistas disponibles de nuestro perfil. Al pulsar sobre Logout, mediante la directiva ng-click se llama a la función logout() que se encargará de borrar la información relacionada con la sesión de usuario.

```
<nav>
  <div class="col-lg-4 user-profile">
    <ul>
      <li><a href="user.php#/user-profile">User Profile</a></li>
      <li><a href="user.php#/account-settings">Account Settings</a></li>
      <li><a href="user.php#/my-orders">My orders</a></li>
      <li><a href="user.php#/logout" ng-click="logout()">Logout</a></li>
    </ul>
  </div>
</nav>
```



## Account Settings

The screenshot shows a form titled "Account Settings" with the following fields and labels:

- Email: Email
- User: username
- Pass: Password
- Last Name: last\_name
- First name: first\_name
- Address: billing\_address
- Dni: dni
- City: city
- State: state
- Postal Code: postal\_code
- Phone: phone

At the bottom right of the form is a blue button labeled "UPDATE".

Rellenando un formulario idéntico al de Sign up, se consigue actualizar la información del usuario. Hay unas pequeñas diferencias que remarcar:

Ng-submit en este caso llama a la función `update()`.

```
<form name="myForm" class="form-horizontal" role="form" ng-submit="update()">
```

Los campos del formulario contienen la etiqueta **required**, esta etiqueta obliga a que este campo esté cumplimentado para poder pinchar en submit.

```
<div class="col-sm-10">  
  <input type="email" class="form-control" id="email" name="email" placeholder="Email"  
    ng-model="email" required>  
</div>
```

Con la directiva **ng-disabled** se desactiva el botón suponiendo que no se haya rellenado algún campo.

```
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <button type="submit" ng-disabled="myForm.$invalid" class="btn btn-primary pull-right">
      Update</button>
    </div>
  </div>
</div>
```

### My orders

Dentro de My Orders se pueden ver los pedidos que ha hecho el usuario, consultar los detalles acerca de algún pedido o borrarlo.

My Orders				
Order	Total	Order Date	Status	Actions
#13	133\$	2015-09-14	Sent	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#14	63\$	2015-09-14	Delivery Failure	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#17	159\$	2015-09-15	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#18	159\$	2015-09-15	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#20	132\$	2015-09-16	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#21	192\$	2015-09-16	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>

El diseño de la tabla se ha obtenido en esta página <http://codepen.io/jackrugile/pen/EyABe>

La tabla muestra el número de pedido, el valor total en \$, la fecha del pedido, el estado de éste y dos acciones: Mostrar detalles y borrar.

My Orders llama a una serie de funciones cuando se pide la vista:

```
<div class="container" data-ng-init="myOrders();getUserInfo();" >
```

Con la directiva **data-ng-init**, se invocan a dos funciones al iniciar la vista:

- **myOrders**, hace una petición al servidor para obtener los pedidos relacionados con el usuario.
- **getUserInfo**, obtiene la información del usuario que será de utilidad a la hora de ver los detalles de los pedidos.

A continuación se rellenan los siguientes campos de la tabla con la directiva **ng-repeat**.

```
<tr>
  <th>Order</th>
  <th>Total</th>
  <th>Order Date</th>
  <th>Status</th>
  <th>Actions</th>
</tr>
```

```
<tr ng-repeat="x in orders">
  <td >#{{x.orderId}}</td>
  <td>{{x.total}}€</td>
  <td>{{x.orderDate}}</td>
  <td data-ng-init="getStatus(x.statusId)">{{status[$index]}}</td>
  <td>
    <a href="user.php#/order-details" ng-click="setOrderIdToShow(x.orderId)"><i class="fa fa-arrow-circle-right"> </i> Show Details </a><br>
    <a ng-click="deleteOrder(x.orderId)"> <i class="fa fa-arrow-circle-right"> </i> Delete Order</a></td>
</tr>
```

Se utiliza la variable que se ha nombrado como “x” para acceder a las propiedades de orders (array donde se guardan de los pedidos que se han obtenido mediante la función myOrders). Por lo tanto {{x.propiedad}} muestra la propiedad nombrada.

En cada fila se llama a la función getStatus(x.statusId), que traducirá el **statusId** de cada **pedido** a un código de estado de pedido. **\$index** sirve para mostrar la iteración justa que está haciendo ng-repeat, así se puede mostrar la posición exacta de cada pedido dentro del array status.

Código de estado	Estado
1	Confirmed
2	Ready for Shipping
3	Sent
4	Dispatched
5	Delivery Failure
6	Canceled
Default	Unknown

A la hora de definir las acciones disponibles en la tabla, se ha utilizado la directiva **ng-click** para llamar a **setOrderIdToShow(x.orderId)** o **deleteOrder(x.orderId)** para que en caso de que se pinche, se llame a la primera función para definir un orderId determinado (información que se usa para mostrar los detalles del pedido) o borrar ese pedido.

### *Order details*

En esta sección se pueden visualizar los detalles del pedido como: El número de pedido, el estado del mismo, la fecha en que se hizo, la información del usuario, los detalles de pago y la lista de productos que se encargaron.


### Order Details from Order # 13

**STATUS:** Sent

**ORDER DATE:** 2015-09-14

<b>User Info</b>	<b>Payment Method</b>
Name: dani	Payment Method: VISA
Billing Address: dani	Holder Name: DANIEL
Telephone: 674855605	ALARCON
Email: danychi@gmail.com	Number: 3021
City: dani	Exp. Date: 2016-07-10
Postal Code: 1234	

**Items Ordered**

Title	Quantity	Total	Cover
The Vaccines - English Graffiti	1	10€	
Tesseract - One	5	65€	
Dylan Ross - Hatch	4	32€	

Al principio del archivo se inicializan las siguientes funciones:

**-getOrders():** Se recuperan los pedidos.

**-getOrderIdToShow:** Se obtiene el orderId que se ha guardado al usar setOrderIdToShow(orderId).

**-getProductsIdByOrderId:** Se obtienen los IDs de los productos relacionados con el orderId.

**-getProductsByProductsId:** Se recuperan los productos que coincidan con el array de ProductsId que se ha obtenido en la función anterior.

**-getOrderDetailsByOrderId:** Se recopilan los detalles del pedido.

La vista Order Details empieza mostrando el ID del pedido, su estado y fecha.

Se ha utilizado un pequeño truco para presentar la info de este pedido, se recorre el array de pedidos aplicándole un filtro personalizado, se mostrará el pedido en el que el valor orderId coincida con nuestro orderId (obtenido con **getOrderIdToShow**); se usa la condición “:true” para mostrar el resultado que coincida al 100% con la condición.

## Order Details from Order # 13

STATUS: Sent

ORDER DATE: 2015-09-14

```
<div class="col-lg-8">
  <h3>Order Details from Order # {{orderId}}</h3>

  <div class "col-md-12 order-info" data-ng-init="getSingleStatus(x.statusId)"ng-repeat="x in orders
  | filter:{orderId:orderId}:true">
    <h5>STATUS:  {{s}} </h5>
    <h5>ORDER DATE:  {{x.orderDate}}</h5>
  </div>
```

A continuación se muestra la información del usuario, al iniciar el archivo se ha llamado a **recoverUserInfo()**, que nos devuelve un objeto con la información del usuario.

Al ser sólo un objeto no necesitamos recorrer un array con ng-repeat, por lo tanto directamente vamos mostrando la información en una lista ordenada.

### User Info

Name: dani

Billing Address: dani

Telephone: 674855605

Email:danychi@gmail.com

City: dani

Postal Code: 1234

```
<div class="col-md-6 order-details" data-ng-init="recoverUserInfo()">
  <div class="header-order-details">User Info</div>
  <ul>
    <li>Name: {{userInfo.firstName}} </li>
    <li>Billing Address: {{userInfo.billingAddress}} </li>
    <li>Telephone: {{userInfo.phone}} </li>
    <li>Email:{{userInfo.email}} </li>
    <li>City: {{userInfo.city}} </li>
    <li>Postal Code: {{userInfo.postalCode}}</li>
  </ul>
```

Seguidamente nos encontramos con la información de pago. De manera análoga a User Info, al iniciar el archivo se ha llamado a **getPaymentInfo()**, de esta manera se pueden mostrar los datos de la información de pago en una lista ordenada de una forma sencilla.

## Payment Method

Payment Method: VISA

Holder Name: DANIEL

ALARCON




Number: 3021

Exp. Date: 2016-07-10

```
<div class="col-md-6 order-details" data-ng-init="getPaymentInfo()">
```

Finalmente nos encontramos con una tabla que rellenaremos con el uso de varias directivas ng-repeat. Con la información almacenada en el array productsOrderDetails (obtenido mediante la llamada inicial a **getProductsByProductsId**) se muestra el autor y nombre de cada producto. A continuación se muestra la cantidad y total en € que se encuentran en el array orderDetails, usando ng-repeat con un filtro para que sólo se muestre el que coincida con el ID del producto.

```
<tr ng-repeat="x in productsOrderDetail">  
  <td>{{x.author}} - {{x.productname}} </td>  
  <td ng-repeat="y in orderDetails | filter:{productId:x.productId}:true">{{y.quantity}}</td>  
  <td ng-repeat="z in orderDetails | filter:{productId:x.productId}:true">{{z.total}}$</td>  
  <td></td>  
</tr>
```

Title	Quantity	Total	Cover
The Vaccines - English Graffiti	1	10€	
Tesseract - One	5	65€	
Dylan Ross - Hatch	4	32€	

## Logout

Al pinchar en el texto Logout de la barra de navegación del perfil de usuario o en el botón que se encuentra en la cabecera, se muestra un texto de despedida.



## Aclaración scope y rootScope[27]

Cada aplicación tiene un **único rootScope** y puede ser inyectado en cualquier parte de la aplicación. Todos los demás scope son scopes descendientes del rootScope.

Un ejemplo práctico sería:

Si tenemos dos controllers diferentes en nuestra aplicación y declaramos la variable:

```
$rootScope.age = 24;
```

Esta variable será accesible con `{{age}}` desde cualquier vista que tenga cualquiera de los dos controllers asignados a la aplicación, ya que rootScope sigue el mismo ciclo de vida que la aplicación.

## Diferencias entre rootScope y localStorage

El localStorage corresponde con el **almacenamiento interno del navegador** y el **rootScope** con el **almacenamiento global de la aplicación de AngularJS**.

Un ejemplo dentro de nuestra aplicación es la variable del localStorage “token”, esta variable es necesaria tanto en la aplicación Users como Shop, si se declarase en el rootScope de Users, sólo podría ser utilizada dentro de Users y Shop no tendría acceso a ella; por lo tanto se guarda en el localStorage.



### 4.4.3 Shop

La tienda ofrece distintas posibilidades al usuario, podemos ver todos los artículos que hay disponibles, aplicarles filtros según su género, precio y tipo; ver información detallada de un producto, añadir al carrito ítems o hacer checkout una vez la sesión esté iniciada.

La tienda está basada en el módulo ng-cart <http://ngcart.snapjay.com/>, que permite de forma sencilla implementar un carro en nuestra aplicación.

#### Estructura de Shop

```
|-- lib/  
    |-- ngStorage.js  
|-- shop-views/  
    |-- cart-view.html  
    |-- checkout-view.html  
    |-- product-view.html  
    |-- shop.php  
    |-- shop-view.html  
|-- scripts-shop /  
    |-- app-shop.js  
    |-- controllers.js  
    |-- services.js
```

Shop está formado por las vistas de la tienda (shop-views), los scripts necesarios para que funcione la aplicación (app-shop.js, controllers.js y services.js) y la librería ngStorage para tener acceso al almacenamiento local (así es posible acceder a la información de la aplicación Users también).

#### Cómo funciona la aplicación Shop en AngularJS

Para comprender cómo funciona nuestra tienda necesitamos entender el funcionamiento de la app, el controller y el service.

##### *App-shop.js*

App-shop.js define el módulo principal de la aplicación y carga todas las configuraciones

necesarias para correr la aplicación.

Lo primero que encontramos es la definición del módulo de la aplicación, nombrado con “app”. A continuación se incluyen las librerías que va a necesitar la aplicación, **dirPagination** que se utilizará para la paginación de los productos, **ngStorage**, **ngCart** que servirá para crear y gestionar un carrito de compra y **ngRoute** para manejar las vistas de la aplicación.

```
angular.module('app', [  
  'angularUtils.directives.dirPagination',  
  'ngStorage',  
  'ngCart',  
  'ngRoute'
```

Seguidamente se define un filtro llamado ‘unique’[28] que sirve para eliminar elementos duplicados en ng-repeat.

```
]).filter('unique', function() {  
  return function(collection, keyname) {  
    var output = [],  
        keys = [];  
  
    angular.forEach(collection, function(item) {  
      var key = item[keyname];  
      if(keys.indexOf(key) === -1) {  
        keys.push(key);  
        output.push(item);  
      }  
    });  
  
    return output;  
  };  
})
```

La siguiente parte del código corresponde a la definición de constantes, en este caso son todos urls, que se utilizarán habitualmente en el servicio para hacer llamadas al servidor

```
.constant('urls', {  
  BASE: 'http://localhost:8080',  
  BASE_API_USERS: 'http://localhost:8085/virtualShopWS/api/rest/users/',  
  BASE_API_ORDERS : 'http://localhost:8085/virtualShopWS/api/rest/orders/',  
  BASE_API_PAYMENT_DETAILS : 'http://localhost:8085/virtualShopWS/api/rest/paymentDetails/',  
  BASE_API_PRODUCTS : 'http://localhost:8085/virtualShopWS/api/rest/products/',  
  BASE_API_ORDER_DETAILS : 'http://localhost:8085/virtualShopWS/api/rest/orderDetails/'  
})
```

A continuación se inyectan los servicios routeProvider y httpProvider, el primero sirve para que funcione ngRoute y el segundo para realizar la configuración necesaria para evitar problemas con el CORS.

```
.config(['$routeProvider', '$httpProvider', function ($routeProvider, $httpProvider) {
    $httpProvider.defaults.crossDomain=true;
    $httpProvider.defaults.useXDomain = true;
    $httpProvider.defaults.withCredentials = true;

```

Posteriormente se configuran las vistas. Tiene un funcionamiento muy sencillo, cuando se pide una ruta sobre shop.php, por ejemplo /cart entonces se muestra una vista y se le asigna un controlador.

```
$routeProvider.
    when('/', {
        templateUrl: 'shop-view.html',
        controller: 'ShopController'
    }).
    when('/cart', {
        templateUrl: 'cart-view.html',
        controller: 'ShopController'
    }).
    when('/checkout', {
        templateUrl: 'checkout-view.html',
        controller: 'ShopController'
    }).
    when('/product-details', {
        templateUrl: 'product-view.html',
        controller: 'ShopController'
    }).
    otherwise({
        redirectTo: '/'
    });

```

Con la directiva `<div ng-view></div>` ngRoute crea la vista sobre ese div (en shop.php se incluirá esta directiva). El resto del código funciona de igual manera que en app-users.js.

```
$httpProvider.interceptors.push(['$q', '$location', '$localStorage', function ($q, $location, $localStorage) {
    return {
        'request': function (config) {
            config.headers = config.headers || {};
            if ($localStorage.token) {
                config.headers.Authorization = $localStorage.token;
            }
            return config;
        },
        'responseError': function (response) {
            if (response.status === 401 || response.status === 403) {
                delete $localStorage.token;
                $location.path('/signin');
            }
            return $q.reject(response);
        }
    };
}]);

```

```

}).run(function($rootScope, $location, $localStorage) {
  $rootScope.token=false;
  if($localStorage.token){$rootScope.token =true;}

  $rootScope.$on( "$routeChangeStart", function(event, next) {
    if ($localStorage.token == null) {
      if ( next.templateUrl === "user-views/signin.html") {
        $location.path("/signin");
      }
    }
  });
});
});

```

### Controllers.js

En el controller se encuentran todas las funciones a las que tiene acceso el scope, estas llaman al servicio **Shop** cuando necesita conectar con el servidor.

El controlador se conecta al módulo “app”, se define el nombre y se inyectan todos los servicios necesarios para las funciones del controller, como rootScope, scope, location, filter, localStorage, ngCart y Shop, que se ha definido en Services.js

```

angular.module('app')
  .controller('ShopController', ['$rootScope', '$scope', '$location', '$filter', '$localStorage', 'Shop', 'ngCart',
    function ($rootScope, $scope, $location, $filter, $localStorage, Shop, ngCart ) {

```

Se ha seguido la misma filosofía que en el controller de Users.

En el controller se declaran e inicializan algunas variables por defecto, que necesitará ngCart:

```

//##### SETTING STANDARD VARIABLES #####
ngCart.setTaxRate(21);
ngCart.setShipping(3.95);

```

Las tasas con un valor de un 21% y los gastos de envío 3.95\$.

A continuación explicaremos las funciones a las que va a tener acceso nuestro scope:

**-getProducts:** Esta función sirve para recopilar todos los productos existentes en la tienda, hace una llamada al servicio Shop, en caso de que la respuesta sea exitosa llama a successGetProducts, que guarda la respuesta del servidor en el localStorage y en el rootScope con el nombre **products**.

```
$scope.getProducts = function () {
    Shop.getProducts(successGetProducts, function (res) {
        $rootScope.error = res.error || 'Failed to get the Products.';
    })
}
```

```
function successGetProducts(res){
    $localStorage.products = res;
    $rootScope.products = $localStorage.products;
}
```

**-getUserInfo:** Tiene el mismo funcionamiento que getUserInfo del controller de Users, su cometido es obtener la información del usuario que está relacionado con el token de la sesión actual. Llama al servicio Shop, en caso de obtener una respuesta correcta del servidor invoca a **successGetUserInfo**, que guarda la información en la variable **userInfo** del localStorage y rootScope.

```
$scope.getUserInfo = function () {
```

**-randomPicture:** Esta función se creó para mostrar una imagen aleatoria en la foto de portada de la tienda, elige aleatoriamente un número del 1 al 5 para mostrar una de las 5 fotos disponibles.

```
$scope.randomPicture = function(){
    $rootScope.picture = Math.floor(Math.random() * 5) + 1 ;
}
```

**-createOrder:** El cometido de esta función es el de crear un pedido, la parte complicada es rellenar el JSON correctamente. Algunas variables toman el valor null porque se establecen automáticamente en el servidor, ya sea porque es una variable que se autoincrementa, porque tiene un valor por defecto o porque depende de la fecha actual. Otros valores se recogen gracias a los métodos que trae ngCart, como el coste total del pedido. También hay valores que se recogen desde un formulario presente en checkout.html.

Los detalles del pedido se recopilan haciendo una llamada a una función presente en el controller.

```
$scope.createOrder = function () {
```

```

var orderDetails = $scope.getOrderDetails();

var formData = {
  orderId:null,
  statusId:null,
  userId:null,
  total:ngCart.totalCost(),
  orderDate:null,
  active:null,
  orderDetails:orderDetails,
  paymentDetail:
  {paymentDetailId:null,
  description:$scope.description,
  orderId:null,
  number:$scope.number,
  expDate:$scope.expDate,
  holderName:$scope.holderName
  }
};

```

Se llama a la función createOrder de Shop, si la respuesta es exitosa se invoca a **sucesCreateOrder**, que nos redirige a la página de inicio mostrándonos una alerta de que el pedido ha sido creado. En caso de error, se muestra un mensaje diciendo que no ha sido posible crear el pedido.

```

function sucesCreateOrder(res){
  window.location = "/";
  window.alert("Created!");
}

```

**-getOrderDetails:** En esta función se recogen los detalles del pedido. Primero se recuperan todos los productos que están en el carrito, se inicializa un array llamado **orderDetails** y se va rellenando con un bucle; finalmente se devuelven los detalles del pedido.

```

$scope.getOrderDetails = function(){

  var items =ngCart.getItems();
  var orderDetails =[];
  for(var i in items)
  {
    orderDetails.push(
    {orderDetailId:null,
    active:1,
    orderId:null,
    productId:items[i]._id,
    quantity:items[i]._quantity,
    total:items[i]._quantity*items[i]._price
    })
  };
  return orderDetails;
}

```

**-setProductIdToShow:** La función recibe un **productId** que guardará en el localStorage y en el rootScope, se utiliza cuando se clickea en un producto para ver sus detalles.

```
$scope.setProductIdToShow = function(productId){
  $localStorage.productId = productId;
  $rootScope.productId = $localStorage.productId;
}
```

**-getProductsIdToShow:** Con esta función recuperamos el ID del producto que queremos mostrar, esta función es necesaria porque cuando se recarga la página de un producto se reinician los valores de rootScope y scope, por lo tanto nuestra variable productId se pierde y es necesaria recuperar su valor desde el localStorage.

```
$scope.getProductIdToShow = function(){
  $rootScope.productId = $localStorage.productId;
}
```

**-setCategoryToShow:** Igual que setProductIdToShow, recibe un **category** que guarda en el localStorage y rootScope, se utiliza para hacer recomendaciones de otros productos en la vista del producto.

```
$scope.setCategoryToShow = function(category){
  $localStorage.category = category;
  $rootScope.category = $localStorage.category;
}
```

**-getCategoryToShow:** Se recupera el valor de category por la misma razón que se explicó en getProductsIdToShow.

```
$scope.getCategoryToShow = function(){
  $rootScope.category = $localStorage.category;
}
```

Para terminar el controller, nos encontramos con la creación de tres filtros personalizados[29] que funcionan de la misma manera. Básicamente sirven para filtrar por un tipo de categoría nuestros productos.

```
$scope.categoryIncludes = [];

$scope.includecategory = function(category) {
  var i = $.inArray(category, $scope.categoryIncludes);
  if (i > -1) {
    $scope.categoryIncludes.splice(i, 1);
  } else {
    $scope.categoryIncludes.push(category);
  }
}

$scope.categoryFilter = function(products) {
  if ($scope.categoryIncludes.length > 0) {
    if ($.inArray(products.category, $scope.categoryIncludes) < 0)
      return;
  }

  return products;
}
```

## Services.js

El servicio está formado por una serie de funciones que pueden ser llamadas desde cualquier aplicación de AngularJS, siempre y cuando sea inyectado mediante inyección de dependencias (DI).

```
angular.module('app')
  .factory('Shop', ['$http', '$localStorage', 'urls', function ($http, $localStorage, urls) {
```

Se crea como una factoría (.factory), se le nombra “Shop” nombre con el que se invocará y finalmente se inyectan los servicios que va a necesitar, entre ellos destacar **\$http** que es el servicio con el que se harán las llamadas al servidor y **\$localStorage** que será necesario para modificar el almacenamiento interno.

El servicio dispone de tres llamadas al servidor simplemente:

**-getProducts:** Hace un get a la dirección “products/”, sirve para obtener un array con los productos de la tienda.

```
getProducts: function (success, error) {
  $http.get(urls.BASE_API_PRODUCTS).success(success).error(error)
},
```

**-getUserInfo:** Hace un get a “users/getUserInfo” para recuperar la información del usuario que tiene una sesión asociada con el token actual.

```
getUserInfo: function (success, error) {
  $http.get(urls.BASE_API_USERS + 'getUserInfo').success(success).error(error)
},
```

**-createOrder:** Hace un post a “orders/confirm/{data}”, siendo data un formulario del tipo JSON con los datos necesarios para crear un pedido.

```
createOrder: function (data, success, error) {
  $http.post(urls.BASE_API_ORDERS + 'confirm', data).success(success).error(error)
}
```

## Vistas de Shop

```
|-- shop-views/
   |-- cart-view.html
   |-- checkout-view.html
   |-- product-view.html
   |-- shop.php
   `-- shop-view.html
```



Finalmente queda explicar las vistas de la tienda y cómo funciona cada una.

La raíz de las vistas es **shop.php**:

```
<body ng-app="app">
  <div ng-view></div>
```

En este fichero se inicializa la aplicación con la directiva **ng-app**, se inyecta la directiva **ng-view** para mostrar las vistas y se incluyen los scripts necesarios para que funcione la aplicación (scripts-shop, ngStorage, dirPagination, etc.).

### Shop

Desde la vista de la tienda se pueden ver los productos que ésta nos ofrece, añadir productos al carrito, ordenar los productos según su tipo, categoría o precio, hacer checkout y ver más información acerca de un producto.



**Type**

- CD
- VINYL
- Digital Music

**category**

- Indie & Alternative
- Djent
- Hip-hop
- Rock
- Latin music

**price**

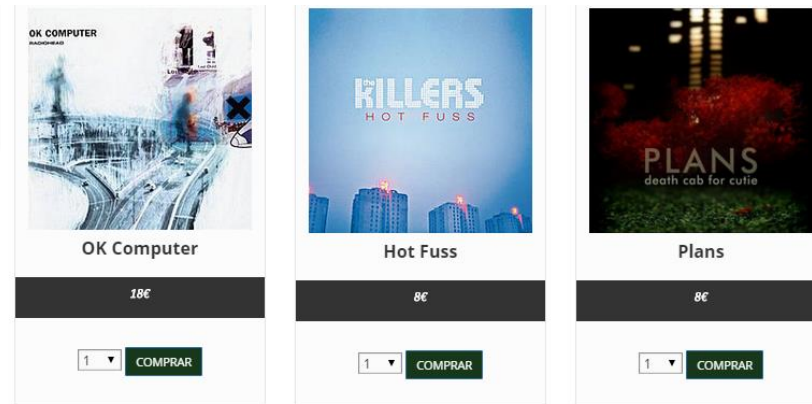
- 10€
- 13€

An End Has a Start  
10€  
1 ▼ COMPRAR

Arctic Monkeys  
Hambug  
10€  
1 ▼ COMPRAR

BEN HARPER AND WHITE LIES  
RELENTLESS  
White Lies for Dark Times  
13€  
1 ▼ COMPRAR

- 17€
- 8€
- 7€
- 15€
- 18€



Checkout



```
data-ng-init="getProducts();randomPicture()"
```

Lo primero que hace shop al inicializarse es llamar a **getProducts** y **randomPicture**. Con **getProducts** obtiene la lista de productos de la tienda y con **randomPicture** obtiene un número aleatorio (entre 1 y 5) para mostrar la imagen de portada.

### Cabecera

Summary nos informa del número de artículos que hay en nuestro carrito y del precio de todos los artículos en el carrito, con la directiva **ngcart-summary** se crea la vista automáticamente.

## Summary

```
<div class="col-xs-offset-10 col-xs-4">
  <h3>Summary</h3>
  <ngcart-summary></ngcart-summary>
</div>
```

11 items  
\$131.00

A continuación se muestra la imagen de portada gracias al número aleatorio guardado en la variable del scope “picture”.

```
<div class="row">
  <div class="col-lg-12">
    
  </div>
</div>
```

## Left Sidebar

Seguidamente se define un formulario de búsqueda al que se le asigna el modelo `searchText` con la directiva **ng-model**, posteriormente utilizando `filter` se podrán filtrar los productos según lo que se escriba en este formulario.

```
<div class="widget">
  <form class="form-search">
    <input class="form-control" ng-model="searchText" type="text" placeholder="Search">
  </form>
</div>
```

En la siguiente sección encontramos tres categorías para ordenar nuestros productos: Type, genre y price.

Mediante **ng-repeat** y el filtro **unique** mostramos las opciones posibles para cada categoría, para filtrar según una característica se ha utilizado un checkbox que responde a la directiva **ng-click**, ésta llama a uno de los tres filtros personalizados creados anteriormente.

```
<div class="widget">
  <h5 class="widgetheading">Type</h5>
  <ul>
    <li ng-repeat="x in products | unique:'type'"><input type="checkbox" ng-click="includetype(x.type)">
      {{ x.type}}</li>
  </ul>
</div>
```

Type	Genre	Price
<input type="checkbox"/> CD	<input type="checkbox"/> Indie & Alternative	<input type="checkbox"/> 10\$
<input type="checkbox"/> VINYL	<input type="checkbox"/> Djent	<input type="checkbox"/> 13\$
<input type="checkbox"/> Digital Music	<input type="checkbox"/> Hip-hop	<input type="checkbox"/> 17\$
	<input type="checkbox"/> Rock	<input type="checkbox"/> 8\$
	<input type="checkbox"/> Latin music	<input type="checkbox"/> 7\$
		<input type="checkbox"/> 15\$

## Products

En la siguiente parte se muestran los productos con la directiva de paginación incluida en el módulo **dirPaginate**, estos artículos se evalúan según los filtros nombrados con anterioridad y se ordenan en un máximo de seis artículos por página.

```
<div class="col-xs-3" dir-paginate="x in products | filter:searchText | filter:categoryFilter | filter:typeFilter | filter:priceFilter | itemsPerPage: 6">
```

Al recorrer el array de productos se van a mostrar las características de cada artículo, si se pincha en la imagen irá a la vista del producto, estableciendo el ID del producto y su género, propiedades necesarias para mostrar la vista de éste.

```

<div class="pricing-box-alt" >
  <div class="pricing-heading">
    <a href="shop.php#/product-details" ng-click="setProductIdToShow(x.productId);setCategoryToShow(x.category);"></a>
    <h4>{{ x.productname }}</h4>
  </div>

```



La directiva **ngcart-addtocart** nos permite agregar artículos al carrito que se almacenan en el localStorage en la variable **cart**.

```

<div class="pricing-action">
  <ngcart-addtocart id="{{ x.productId }}" name="{{ x.productname }}" price="{{ x.price }}"
  quantity="1" quantity-max="30" data="item">comprar</ngcart-addtocart>
</div>

```

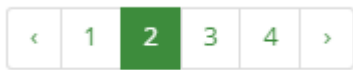
*Foot*

Insertando la directiva **dir-pagination-controls**, se crea la vista correspondiente a la paginación.

```

<div class="pull-right" style="margin-right:2em;"><dir-pagination-controls></dir-pagination-controls></div>

```



El botón de checkout actúa de manera diferente según si existe el **token** o no, si hay una sesión activa, cuando pulsemos checkout nos llevará a la vista correspondiente al carrito, en caso contrario nos llevará a la pantalla de login.

```
<div class="row">
  <div class="col-md-6">
    <div ng-show="{{token}}">
      <a href="#/cart"> <button class="btn btn-primary">Checkout</button></a>
    </div>

    <div ng-hide="{{token}}">
      <a href="http://localhost:8080/user-views/user.php?#/signin"> <button class="btn btn-primary">
Checkout</button></a>
    </div>
  </div>
</div>
```

Checkout



Product

```
data-ng-init="getProducts();randomPicture();getProductIdToShow();getCategoryToShow();">
```

Product llama a las siguientes funciones:

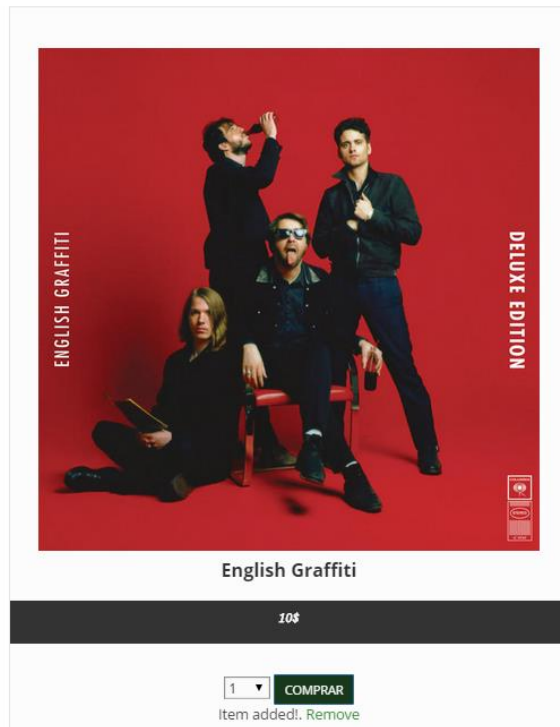
- getProducts:** Pide la lista de productos necesaria para hacer las recomendaciones al usuario y filtrar el producto que va a mostrar.
- randomPicture:** Se utiliza para mostrar una imagen de portada aleatoria.
- getProductIdToShow:** Sirve para recuperar el ID del producto que vamos a mostrar.
- getCategoryToShow:** Recupera el género del producto, para recomendar al usuario artistas de la misma categoría musical.

Lo primero que aparece en esta vista es la foto de portada y el Summary, al igual que en la vista de la tienda.

**Summary**

12 items  
\$143.10





The Vaccines - English Graffiti

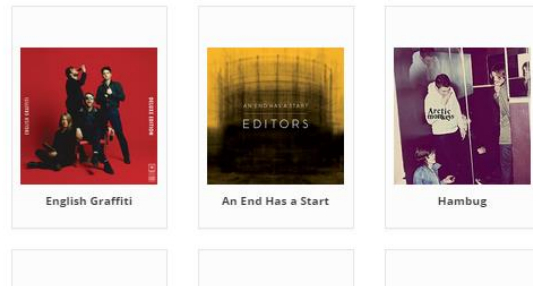
BACK

Category: Indie & Alternative

Type:CD

The Vaccines 'English Graffiti,' is their first release in three years. The album was recorded in upstate New York, and the 11-song collection is the UK foursome's most accomplished work to date. 'English Graffiti' was produced David Fridmann (Flaming Lips, Tame Impala, MGMT) and Cole MGN (Ariel Pink, Beck). 'English Graffiti' is The Vaccines' first record made in the US, and finds group at the top of their game tracks like "Minimal Affection" recall the glory days of 80s synth-pop, "20/20" fires with machine gun precision, while "Handsome" connects with their power pop roots. The Vaccines' last record, 'The Vaccines Come Of Age,' reached # 1 in the UK and has sold over 400k copies. They've toured with everyone from The Rolling Stones, Arcade Fire, RHCP, Arctic Monkeys

Perhaps you may be interested in the following artists:



A continuación se muestra el producto filtrando por el ID que coincida con nuestro artículo; además se muestra el tipo de producto que es (CD, Vinilo o música digital), una descripción de éste y también nos permite añadir desde aquí nuestro artículo al carrito.

```
<div class="col-md-6" ng-repeat="x in products | filter:{productId:productId}:true">
```

La última parte de esta vista es la recomendación de artistas similares al producto que estamos visualizando. Se ha conseguido repitiendo los productos que tengan el mismo género, aplicándole un **filter** con la categoría recogida en la variable “category” obtenida con **getCategoryToShow** y limitando a seis el número de ítems a mostrar, con el filtro que trae angular por defecto llamado **limitTo**.

```
<div class="col-xs-4 order-details" ng-repeat="x in products | filter:{category:category} | limitTo: 6">
```

### Cart

La vista del carrito se genera automáticamente gracias a la directiva:

```
<ngcart-cart></ngcart-cart>
```

Al final del carrito se ha añadido un botón del carrito de checkout que funciona igual que en Shop, sólo que suponiendo que exista una sesión activa nos lleva a la vista de checkout en vez de al carrito.

## Cart

	Title	Quantity	Amount	Total
✕	7th or St. Tammany	- 2 +	\$7.00	\$14.00
✕	A contra corriente	- 3 +	\$7.00	\$21.00
✕	One	- 4 +	\$13.00	\$52.00
✕	Hatch	- 1 +	\$8.00	\$8.00
✕	English Graffiti	- 1 +	\$10.00	\$10.00
✕	An End Has a Start	- 1 +	\$10.00	\$10.00
			Tax (21%):	\$24.15
			Shipping:	\$3.95
			Total:	\$143.10

## Checkout

Checkout es la última vista de la tienda y del cliente.

### User Info

Name: dani  
Billing Address: dani  
Telephone: 674855605  
Email: danychi@gmail.com  
City: dani  
Postal Code: 1234

Is your Info ok?  
You can update it from your user profile [Goto!](#)

### Payment Method

Payment Method

Holder Name

Number

Expiration Date

CONFIRM



### User Info

La primera parte muestra la información del usuario, que funciona exactamente igual que la sección User Info de la vista Order details (incluida en la aplicación Users), haciendo uso de la función **getUserInfo**.

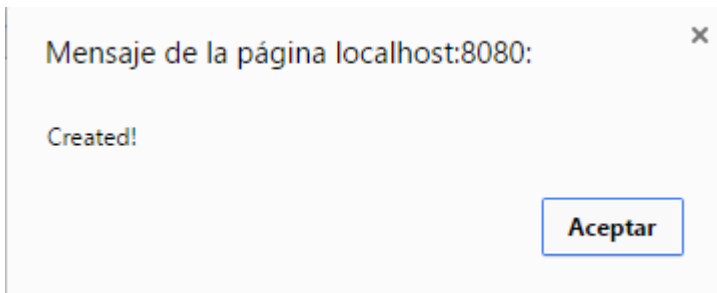
En caso de que la información correspondiente al usuario no fuese correcta, esta vista nos ofrece un link a Account Settings para actualizar la información.

Is your Info ok?  
You can update it from your user profile **Goto!**

### Payment Details

La siguiente sección muestra un formulario para introducir los detalles de pago, al pulsar el botón CONFIRM se llama a la función createOrder (se recopilan los datos introducidos en el formulario, con el mismo mecanismo utilizado en los formularios anteriores).

Si se ha rellenado el formulario adecuadamente, se mostrará un mensaje diciendo que el pedido se ha creado correctamente y nos redirigirá a la página de inicio, sino mostrará un mensaje de error diciendo que revisemos los datos introducidos.



### Items Ordered

Al final de la vista se muestra la información de los productos que se han encargado.

#### Items Ordered

Title	Quantity	Amount	Total
7th or St. Tammany	2	\$7.00	\$14.00
A contra corriente	3	\$7.00	\$21.00
One	4	\$13.00	\$52.00
Hatch	1	\$8.00	\$8.00
English Graffiti	1	\$10.00	\$10.00
An End Has a Start	1	\$10.00	\$10.00
		Tax (21%):	\$24.15
		Shipping:	\$3.95
		Total:	\$143.10



## 5. Servidor

### 5.1 Objetivo

El objetivo que se buscaba a la hora de crear el servidor era tener una plataforma que atendiese y gestionase las peticiones de los clientes, de la forma más eficaz y sencilla posible.

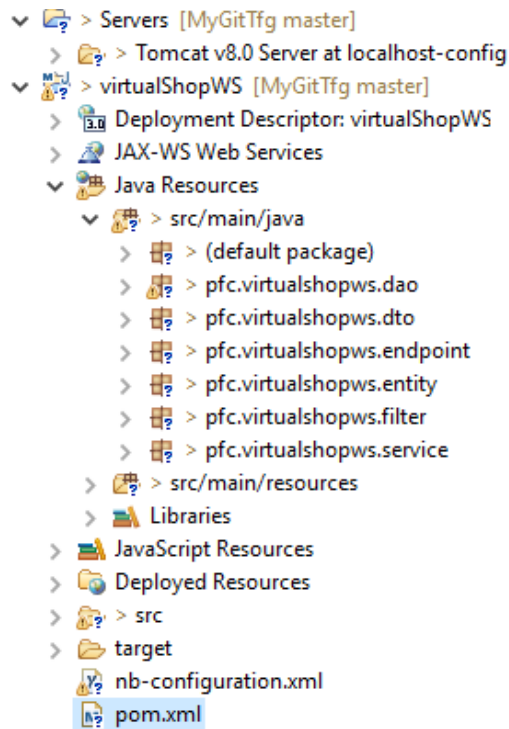
El servidor se va a encargar fundamentalmente de gestionar la funcionalidad del usuario, como la creación de un nuevo cliente, la gestión de su información o el mantenimiento de su sesión; y por otra parte gestiona las peticiones a la tienda, como la obtención de los productos disponibles, la creación y eliminación de pedidos, etc.

### 5.2 Tecnologías Implicadas

Las peticiones al servidor se realizan mediante el protocolo de comunicación **HTTP**, con **Hibernate** conectamos nuestro modelo de datos Java a la base de datos **MySQL** y con **Jersey** gestionamos las peticiones en el **end-point**.

## 5.3 Estructura

### 5.3.1 Estructura en Eclipse



El servidor está estructurado en cuatro capas que se mantienen conectadas entre ellas:

-**Entity**, capa encargada de realizar la conexión con la base de datos y de crear las relaciones entre los objetos Java y la BBDD. Con la anotación **@Entity** se define la entidad.

-**DAO**, en esta capa se crean y realizan las peticiones a la base de datos con Hibernate. Se utiliza **@Repository** para definir el repositorio.

-**Service**, se llaman a los métodos DAOs necesarios para el endpoint. **@Service** define la clase como un servicio.

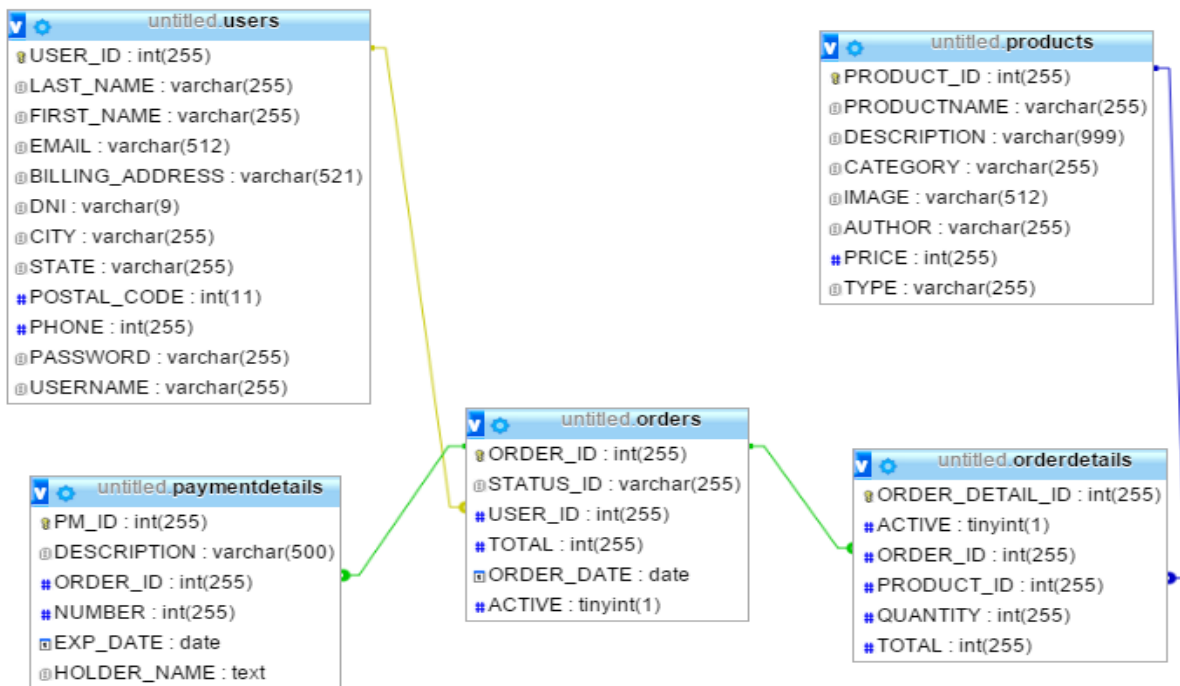
-**Endpoint**, es el punto final de la aplicación, donde se reciben y atienden las llamadas del cliente. La etiqueta **@Controller** define a la clase como endpoint.

En las capas DAO, Service y Endpoint se ha seguido el siguiente procedimiento para crear las clases de nuestro servidor:

- 1- Creación de la interfaz<sup>3</sup> donde se definen todos los métodos disponibles en ésta. Cada método informa del tipo de objeto que devuelve, el nombre del método y el objeto u objetos que necesita recibir para ejecutar correctamente la función.
- 2- Implementación de las funciones definidas en la interfaz.

El resto de la estructura del proyecto son fundamentalmente archivos como el pom.xml, web.xml, infrastructure.xml o applicationContext.xml; esenciales para la configuración del servidor, hablaremos de ellos más adelante.

### 5.3.2 Estructura de la BBDD



La base de datos está formada por cinco tablas **users**, **paymentdetails**, **orders**, **products** y **orderdetails**. Mediante el uso de foreign keys<sup>4</sup> se crean conexiones en algunas tablas. Una clave foránea[30] o clave ajena (o Foreign Key FK) es una limitación referencial entre dos tablas. La clave foránea identifica una columna o grupo de columnas en una tabla (tabla

<sup>3</sup> Una **interfaz** en Java es una colección de métodos abstractos y propiedades. En las interfaces se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describan la lógica del comportamiento de los métodos.

hija o referendo) que se refiere a una columna o grupo de columnas en otra tabla (tabla maestra o referenciada). Las columnas en la tabla referendo deben ser la clave primaria u otra clave candidata en la tabla referenciada.

Dentro de la tabla **users** encontramos la información relacionada con el usuario, como su ID, nombre, apellidos, dirección, email, contraseña y otros datos personales. El ID del usuario es una FK (Foreign Key) en orders.

En la tabla **paymentdetails** se almacenan los detalles de pago de un pedido, como el número de tarjeta, nombre del titular, fecha de expiración, el ID de los detalles de pago, el ID del pedido; este último ID es FK, por lo tanto debe ser un ID de pedido que exista para poder crear los detalles del pedido.

La tabla **orders** guarda datos esenciales para crear un pedido como el ID del pedido, su estado, el ID de usuario relacionado con el pedido, el precio total, la fecha que se creó y si está activo. El ID de usuario debe ser un ID que exista en la tabla users, ya que es FK.

**Orderdetails** es una tabla donde se almacenan los detalles de un pedido, como el ID de los detalles del pedido, un ID del pedido con el que está relacionado, un ID de producto, la cantidad de éste, el precio total (al multiplicar la cantidad por el precio del producto). El ID del pedido y del producto son **FK**, por lo tanto tienen que ser valores reales en las tablas orders y products.

Finalmente queda la tabla **products**, aquí se aloja la información relacionada con el producto como su ID, nombre, descripción, categoría, imagen, autor, precio y tipo.

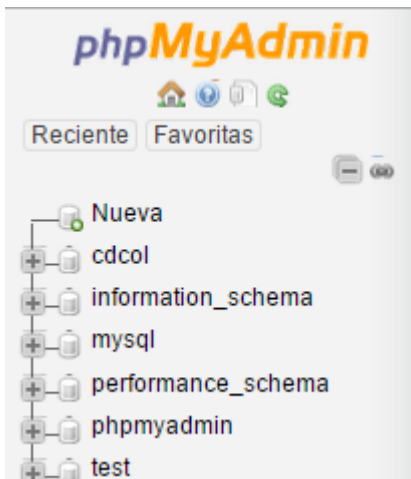
A la hora de crear un pedido hagamos algunas aclaraciones:

- Un **pedido** sólo va a tener **un usuario** asociado (relación 1 a 1).
- Un **pedido** sólo va a tener **unos detalles de pago** asociados (relación 1 a 1).
- Un **pedido** puede tener **uno o más detalles de pedido** asociados (relación 1 a n).
- Unos **detalles de pedido** sólo puede tener **un producto** asociado (relación 1 a 1).

## 5.4 Configuración

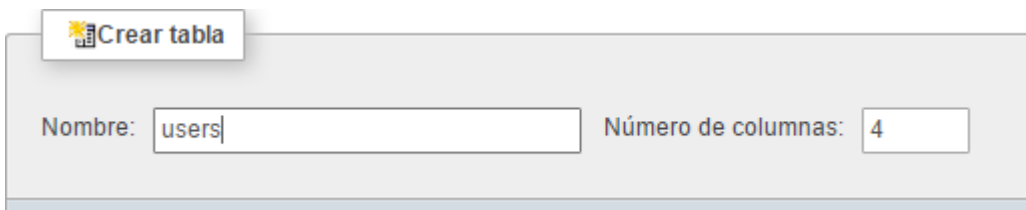
### Configuración de la Base de datos

Creación de una BBDD



Accediendo desde <http://localhost:8080/phpmyadmin/> a la base de datos, en la esquina superior derecha pinchar sobre Nueva, para crear una nueva base de datos. Darle el nombre deseado en modo cotejamiento y pinchar crear.

Creación de una tabla



Para crear una tabla por primera vez, debajo de crear tabla ponemos el nombre y el número de columnas deseado, después pulsamos continuar.

Nombre	Tipo	Longitud/Valores	Predeterminado	Cotejamiento	Atributos	Nulo	Índice	A_I
USER_ID	INT	255	Ninguno			<input type="checkbox"/>	PRIMARY	<input checked="" type="checkbox"/>
EMAIL	VARCHAR	255	Ninguno			<input type="checkbox"/>	---	<input type="checkbox"/>
PASSWORD	VARCHAR	255	Ninguno			<input type="checkbox"/>	---	<input type="checkbox"/>
USERNAME	VARCHAR	255	Ninguno			<input type="checkbox"/>	---	<input type="checkbox"/>

Se pueden crear los campos que se desee, en el ejemplo de arriba se ha definido un USER\_ID del tipo int con máxima longitud 255, va a ser la clave primaria de la tabla y además se va a incrementar automáticamente (A\_I). Los demás campos son: un email, contraseña y username, del tipo varchar (código de longitud variable). Pulsar guardar para almacenar esta tabla en la BBDD.

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra	Acción
1	USER_ID	int(255)			No	Ninguna	AUTO_INCREMENT	Cambiar Eliminar Primaria
2	EMAIL	varchar(255)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Primaria
3	PASSWORD	varchar(255)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Primaria
4	USERNAME	varchar(255)	latin1_swedish_ci		No	Ninguna		Cambiar Eliminar Primaria

## Creando un usuario

Pulsando insertar, se puede rellenar la tabla con usuarios de una forma sencilla.

Columna	Tipo	Función	Nulo	Valor
USER_ID	int(255)	<input type="text"/>		<input type="text"/>
EMAIL	varchar(255)	<input type="text"/>		danychi24@gmail.com
PASSWORD	varchar(255)	<input type="text"/>		1234
USERNAME	varchar(255)	<input type="text"/>		danychi

[Continuar](#)

Se rellenan los campos necesarios para crear un usuario, no es necesario darle un ID a USER\_ID, ya que se incrementa automáticamente, pulsar continuar y el usuario se habrá creado correctamente.

+ Opciones

	USER_ID	EMAIL	PASSWORD	USERNAME
<input type="checkbox"/> Editar Copiar Borrar	1	danychi24@gmail.com	1234	danychi

## Ejemplo de declaración de Foreign Key

**ALTER TABLE** orders

**ADD CONSTRAINT** fk\_UsersOrders

**FOREIGN KEY** (USER\_ID)

**REFERENCES** users(USER\_ID)

Desde la tabla orders, pinchando sobre *SQL* podemos insertar la petición para crear una clave foránea. En la tabla **orders** se añade la restricción **fk\_UsersOrders**, definiendo el campo **USER\_ID** como **foreign key** en orders, proveniente de la tabla **users** (REFERENCES), tabla donde **USER\_ID** es clave primaria.

## Creación de un proyecto Maven

Para crear nuestro proyecto en Eclipse, pinchar en File->New->Maven Project->Next->En filter buscar el arquetipo webapp->Next->Rellenar el campo Group Id, éste se utiliza para identificar el proyecto de forma única entre todos los proyectos (com.ws en este proyecto) y el Artifact Id es el nombre del jar<sup>5</sup> (virtualShopWS en el caso de este trabajo).

Group Id:	<input type="text" value="com.ws"/>
Artifact Id:	<input type="text" value="VirtualShopWS"/>
Version:	<input type="text" value="0.0.1-SNAPSHOT"/>
Package:	<input type="text" value="com.ws.VirtualShopWS"/>
Properties available from archetype:	

### Configuración del pom.xml

El **pom.xml** (Project Object Model) sirve para almacenar la información relacionada con fuentes, plugins, versiones de paquetes, dependencias, test, seguridad, etc.

Las dependencias que se han agregado siguen el siguiente modelo:

```
<!-- Name of Dependency -->
  <dependency>
    <groupId>nameOfGroupId</groupId>
    <artifactId>nameOfArtifact</artifactId>
    <version>numberOfVersion </version>
  </dependency>
```

---

<sup>5</sup> Un archivo **JAR**[40] (por sus siglas en inglés, **J**ava **A**Rchive) es un tipo de archivo que permite ejecutar aplicaciones escritas en el lenguaje Java.

Se han añadido las siguientes:

- **CORS**, resuelve el problema con el Cross-origin resource sharing, que se explicó en el apartado 2.9 del proyecto.
- **GSON**, sirve para serializar o deserializar un objeto Java y su representación en JSON.
- **Spring 3 dependencies**, aquí se incluyen todas las librerías necesarias para que Spring funcione en el entorno.
- **Jackson JSON Mapper y Fasterxml Jackson JSON Mapper**, Java lo utiliza internamente para serializar y deserializar objetos tipos JSON, la diferencia de estas librerías con Gson, es que Spring la utiliza con archivos XML, en Spring en las anotaciones Produces y Consumes, etc.
- **Log4j**: Log4j[31] es una biblioteca open source desarrollada en Java por la Apache Software Foundation que permite escribir mensajes de registro, cuyo propósito es dejar constancia de una determinada transacción en tiempo de ejecución.
- **Hibernate**: En esta librería se encuentran todos los archivos necesarios para que Hibernate funcione en nuestra aplicación.
- **Tomcat**: Librería necesaria para poder implementar nuestra aplicación en Tomcat.
- **MySQL**: Esta dependencia permite conectar Hibernate a la BBDD tipo MySQL.

### Configuración web.xml [32]

Un **descriptor de despliegue** (en inglés *Deployment Descriptor*) (DD) es un componente de aplicaciones J2EE que describe cómo se debe desplegar (o implantar) una aplicación web. Esto dirige una herramienta de despliegue (o publicación) para desplegar un módulo o aplicación con opciones de contenedor específicas y describe requisitos de configuración específicos que puede resolver un desplegador.

En el web.xml se va a configurar Logger, CORS y los Servlets.

### Configuración log.4j

```
<context-param>
    <param-name>log4jConfigLocation</param-name>
    <param-value>classpath:config/log4j.xml</param-value>
</context-param>
```



```

    <listener>
      <listener-
class>org.springframework.web.util.Log4jConfigListener</listener-class>
    </listener>

```

En el web.xml el logger actúa como un listener<sup>6</sup>, se carga su configuración presente en el archivo log4j.xml.

### *Configuración Spring*

Al igual que en el logger, se inicializa el listener de Spring y se carga su configuración desde el applicationContext.xml.

```

<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>
<listener>
  <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
  </listener>

```

### *Configuración servlet*

Se configura Spring para que todas las peticiones al servidor que pasen por “/api/rest/\*” las maneje el Servlet que se configura en este apartado.

```

<servlet>
  <servlet-name>jersey-servlet</servlet-name>
  <servlet-
class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-
name>jersey.config.server.provider.packages</param-name>
    <param-value>pfc.virtualshopws</param-value>
  </init-param>
  <init-param>
    <param-
name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
    <param-value>>true</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
  <enabled>true</enabled>
</servlet>
<servlet-mapping>
  <servlet-name>jersey-servlet</servlet-name>
  <url-pattern>/api/rest/*</url-pattern>
</servlet-mapping>

<context-param>

```

---

<sup>6</sup> Un listener[41] (también conocido como controlador de eventos) es una subrutina de devolución de llamada que se encarga de los inputs recibidos en un programa.

```

        <param-name>resteasy.scan</param-name>
        <param-value>>false</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.scan.resources</param-name>
        <param-value>>false</param-value>
    </context-param>

    <context-param>
        <param-name>resteasy.scan.providers</param-name>
        <param-value>>false</param-value>
    </context-param>

    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

```

### Configuración CORS

Mediante esta configuración, el servidor admitirá los métodos GET, POST, HEAD, PUT, DELETE y OPTIONS desde cualquier dominio de origen.

```

<!-- CORS -->
    <filter>
        <filter-name>CORS</filter-name>
        <filter-
class>com.thetransactioncompany.cors.CORSFilter</filter-class>
        <init-param>
            <param-name>cors.allowOrigin</param-name>
            <param-value>*</param-value>
        </init-param>

        <init-param>
            <param-name>cors.supportedMethods</param-name>
            <param-value> GET, POST, HEAD, PUT, DELETE, OPTIONS</param-
value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>CORS</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

```

### Configuración infrastructure.xml

Dentro de este archivo se crean y configuran los objetos necesarios para montar la infraestructura de la aplicación.

Las siguientes líneas de código validan la estructura del xml y permiten que se puedan

utilizar en este fichero las etiquetas bean, property, qualifier, etc .

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">
```

Carga el dialecto que utilizará Hibernate:

```
<bean id="jpaDialect"
class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />
```

Gestiona la transaccionalidad:

```
<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory"
ref="entityManagerFactory" />
  <qualifier value="pfc"/>
</bean>
```

En las siguientes líneas se define el **entityManagerFactory**, objeto que gestiona toda la persistencia.

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFacto
ryBean">
  <property name="persistenceXmlLocation"
value="classpath:config/persistenceConfig.xml" />
  <property name="persistenceUnitName"
value="restPersistenceDialect" />
  <property name="dataSource" ref="dataSourceAP"/>
  <property name="packagesToScan" value="pfc.virtualshopws.*"
/>
</bean>

<bean
class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostP
rocessor">
  <property name="defaultPersistenceUnitName"
value="restPersistenceDialect"/>
</bean>
```

Configuración del data source, este objeto gestiona el acceso a la base de datos, mediante sus propiedades configuramos la conexión a la base de datos.

```

    <bean id="dataSourceAP"
class="org.springframework.jdbc.datasource.SimpleDriverDataSource">
    <property name="driverClass" value="com.mysql.jdbc.Driver" />
    <property name="url"
value="jdbc:mysql://localhost:3306/untitled" />
    <property name="username" value="root" />
    <property name="password" value="" />
    </bean>

```

## Configuración applicationContext.xml

El fichero de configuración básico de Spring es el contexto de aplicación[33] (application context). Consiste en un fichero XML donde se añadirán todos los objetos que deberán existir en la aplicación al inicializarse la misma.

Se importa el infrastructure.xml, explicado en el apartado anterior, básicamente para cargar la configuración de la base de datos e Hibernate.

```
<import resource="infrastructure.xml"/>
```

Se carga la ruta de los paquetes que tendrán las anotaciones de las cuatro capas presentes en nuestro servidor: @entity, @repository, @service y @controller.

```
<context:component-scan base-package="pfc.virtualshopws.*" />
```

## Configuración de persistenceConfig.xml

El persistenceConfig.xml va a permitir que se realice el mapeo entre las entidades que se han creado en la capa entity y las tablas de la BBDD en MySQL.

```

<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
    <persistence-unit name="restPersistenceDialect">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <class>pfc.virtualshopws.entity.Users</class>
        <class>pfc.virtualshopws.entity.OrderDetails</class>
        <class>pfc.virtualshopws.entity.Orders</class>
        <class>pfc.virtualshopws.entity.PaymentDetails</class>
        <class>pfc.virtualshopws.entity.Products</class>
        <properties>
            <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL5InnoDBDialect" />
            <property name="hibernate.show_sql" value="true" />
        </properties>
    </persistence-unit>
</persistence>

```

## Configuración final

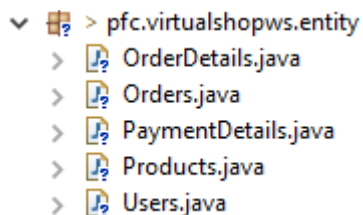
Cuando la aplicación esté terminada hay que realizar una configuración para poder ejecutarla en el servidor:

- 1- Pinchar con el botón derecho en el proyecto de la aplicación y pulsar Run as Maven Clean, para limpiar en anterior .war que se hubiese creado en el servidor.
- 2- Posteriormente Run as Maven install, para que se genere el nuevo archivo .war correspondiente con nuestra aplicación.

En el archivo .war se almacena toda la información de la aplicación de forma compacta, este archivo es necesario para que Tomcat pueda desplegar la aplicación.

## 5.5 Explicación

### Entity



**Entity** es el paquete que se ha creado para almacenar las clases encargadas de establecer las relaciones del modelo de datos Java, con el modelo de datos de la BBDD.

El ejemplo práctico del apartado 2.8 de este proyecto ejemplifica y explica a la perfección el funcionamiento de las clases: **OrderDetails**, **PaymentDetails**, **Products** y **Users**. Repasemos cómo funcionaba:

Cada clase se va a definir como entidad con la anotación **@Entity**, se va a conectar con la tabla correspondiente con la etiqueta **@Table**, posteriormente se definirá un objeto java por cada columna de la tabla con la etiqueta **@Column**. La clave primaria de la tabla llevará la anotación **@Id**, con **@GeneratedValue(strategy = GenerationType.AUTO)** autoincrementará este campo cada vez que se cree un nuevo objeto.

La única clase que varía respecto a las mencionadas arriba es **Orders**:

```
public Orders(long userId, BigDecimal total) {
    DateFormat df = new SimpleDateFormat("yyyy/MM/dd");
    Date today = Calendar.getInstance().getTime();
    this.orderDate = df.format(today);
    this.userId = userId;
    this.statusId = (long) 1;
    this.total = total;
}
```

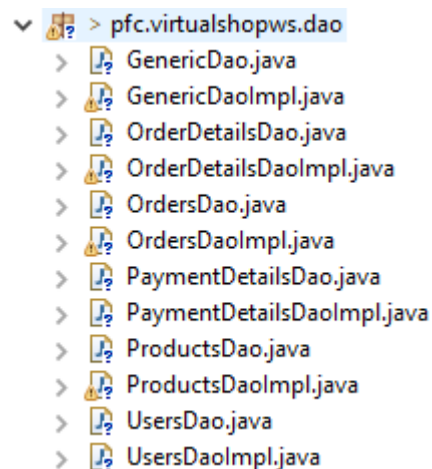
Orders tiene un constructor que se usará en el end-point para crear un pedido, éste establece la fecha del pedido a la fecha actual dándole el formato “yyyy/MM/dd”, el userId se crea según el “userId” que recibe el constructor, el estado del pedido por defecto valdrá 1 (en la tabla de estados significaba “Confirmed”) y el precio total lo establece con la variable “total” que recibe el constructor.

```
@Transient
private List<OrderDetails> orderDetails = new
ArrayList<OrderDetails>();

@Transient
private PaymentDetails paymentDetail = new PaymentDetails();
```

Dentro de la clase también se define una lista de detalles de pedido y un objeto del tipo detalles de pago, objetos necesarios para la creación de un pedido. Ambos tienen la etiqueta **@Transient**, que indica que estos campos no deben ser persistidos en la base de datos.

## DAO



La capa DAO o Data Access Object se va a encargar de hacer las peticiones a la base de datos, gracias a la sintaxis de Hibernate. Para hacer la tarea más sencilla, se ha

implementado un DAO genérico[34] que va a ocuparse de montar un CRUD para cada clase.

Los métodos que se han creado en el DAO por lo general siguen una estructura similar:

- 1- Creación de la petición a la base de datos, se guarda en un String con el nombre **sqlString**.
- 2- Creación de la query usando el método **createQuery(sqlString,Classname.class)** del **entityManager**. El primer término del método corresponde con la petición a la BBDD y el segundo con el tipo de la **clase** que se espera en el resultado de la consulta. En caso de que no devuelva nada el método, entonces se usa el método **createQuery(sqlString)**.
- 3- Ejecución de la query con el método **query.getSingleResult()** o **query.getResultList()**, según si se espera un solo objeto o una lista de objetos. Este resultado se devuelve con return, haciéndole previamente un type casting (conversión de tipo) con la clase que espera retornar el método.

## Implementación del DAO

Cada clase del DAO (excepto la GenericDao) extiende de GenericDaoImpl<Classname> e implementa la interfaz correspondiente a su clase.

Ejemplo de UsersDaoImpl:

```
extends GenericDaoImpl<Users>implements UsersDao
```

### *Genérico*

La clase GenericDao nos aporta de manera sencilla un CRUD para cualquier clase de nuestro paquete.

Primero importa el dialecto que se va a usar en el paquete:

```
@PersistenceContext(unitName = "restPersistenceDialect")
```

Crea un entityManager que se va a encargar de crear las queries para la BBDD

```
public EntityManager entityManager;
```

Define un objeto genérico que se cambiará por la clase que se vaya a utilizar.

```
private Class<T> type;
```

Con el constructor estándar de GenericDao es posible particularizar la clase que se va a manejar.

```

public GenericDaoImpl() {
    Type t = getClass().getGenericSuperclass();
    ParameterizedType pt = (ParameterizedType) t;
    type = (Class) pt.getActualTypeArguments()[0];
}

```

### **Create**[35]

```

public T create(final T t) {
    this.entityManager.detach(t);
    this.entityManager.persist(t);
    this.entityManager.flush();
    return t;
}

```

Create genérico para todas las clases del paquete. Con detach, quita cualquier relación que tenga el objeto t con alguna entidad, persist crea una instancia del objeto para que persista en la base de datos y flush sincroniza el contexto de persistencia con la base de datos subyacente.

### **Read**

```

@Override
public T find(final Object id) {
    return (T) this.entityManager.find(type, id);
}

```

Busca el objeto T según su clave primaria

```

@Override
public List<T> findAll() {
    return entityManager.createQuery("SELECT t from " +
type.getTypeName() + " t").getResultList();
}

```

Selecciona todos los elementos de la tabla de tipo T.

### **Update**

```

@Override
public T update(final T t) {
    T out = this.entityManager.merge(t);
    this.entityManager.flush();
    return out;
}

```

Con merge fusiona el estado de la entidad dada en el contexto de persistencia actual y con flush sincroniza el contexto de persistencia con la base de datos subyacente.

### **Delete**

```

@Override
public void delete(final Object id) {
}

```



```
        this.entityManager.remove(this.entityManager.getReference(type,
id));
    }
```

Elimina el objeto que coincida con la clave primaria que recibe el método.

### *UsersDAO*

UsersDAO dispone de dos métodos adicionales a los genéricos, `findUserByEmail` y `findUserByUsername`.

**findUserByEmail** se va a encargar de hacer una consulta a la base de datos, para devolver el usuario que coincida con el email que recibe el método mediante la variable `login`. La consulta es la siguiente:

```
"SELECT u FROM Users u WHERE u.email = ?login"
```

**findUserByUsername** funciona igual que `findUserByEmail`, sólo que el método recibe un `username` en vez de un email.

### *ProductsDAO*

Además de los métodos genéricos, `ProductsDAO` dispone de un método extra:

**getProductsByProductsId**, este método recibe una lista de IDs del tipo `long` con la que va a hacer una petición a la BBDD para encontrar una lista de productos que coincidan con los IDs. Aparece el operador "IN", se utiliza para especificar varios valores en una cláusula WHERE, ya que hacemos una consulta en base a una lista de IDs.

```
"SELECT u FROM Products u WHERE u.productId IN ?productsId"
```

Tras crear la query, se ejecuta y devuelve una lista de productos.

```
return (List<Products>) query.getResultList();
```

### *PaymentDetailsDAO*

La clase `PaymentDetailsDao` está formada por dos métodos adicionales:

**findPaymentDetailByOrderId**, va a encontrar los detalles de pagos relacionados con un `orderId`.

```
"SELECT u FROM PaymentDetails u WHERE u.orderId = ?orderId"
```

**deleteByOrderId**, este método borra los detalles de pago vinculados a un orderId.

```
"DELETE FROM PaymentDetails WHERE orderId = ?1"
```

La sintaxis para borrar es ligeramente diferente a SELECT. La creación y ejecución de la query también, no se espera un valor de vuelta por lo tanto cambia el método usado en createQuery, la ejecución no utiliza return y utiliza el método executeUpdate.

```
Query query = entityManager.createQuery(sqlString);  
query.executeUpdate();
```

### *OrdersDAO*

Esta clase sólo dispone de un método añadido:

**findOrdersByUserId**, va a encargarse de encontrar una lista de pedidos asociados a un ID de usuario que recibe el método.

```
"SELECT u FROM Orders u WHERE u.userId = ?1";
```

### *OrderDetailsDAO*

Esta clase es la que añade más métodos adicionales:

**getProductsIdByOrderId**, devuelve una lista de IDs de productos según un ID de pedido.

```
"SELECT u.productId FROM OrderDetails u WHERE u.orderId = ?orderId"
```












**getOrderDetailsByOrderId**, esta función retorna una lista de detalles de pago según un ID de pedido.

```
"SELECT u FROM OrderDetails u WHERE u.orderId = ?orderId"
```

**deleteByOrderId**, elimina todos los detalles de pedido en los que su ID de pedido coincidan con el que recibe el método.

```
"DELETE FROM OrderDetails WHERE orderId = ?orderId"
```

## Service

- ▼  > pfc.virtualshopws.service
  - >  OrderDetailsService.java
  - >  OrderDetailsServiceImpl.java
  - >  OrdersService.java
  - >  OrdersServiceImpl.java
  - >  PaymentDetailsService.java
  - >  PaymentDetailsServiceImpl.java
  - >  ProductsService.java
  - >  ProductsServiceImpl.java
  - >  UsersService.java
  - >  UsersServiceImpl.java

El servicio se encarga de crear los métodos a los que tiene acceso el end-point, estas funciones llaman a una función de un DAO para conseguir su cometido, en aplicaciones más complejas se llaman a más de una función de un o varios DAOs por servicio. Con la anotación **@Autowired** se realiza la conexión con el DAO.

## Implementación

Todas las clases implementan el CRUD y conectan con su DAO, ejemplo con UsersService:

```
@Autowired
private UsersDao usersDao;

@Override
public Users create(Users user) {
    return usersDao.create(user);
}

@Override
public Users findById(Long id) {
    return usersDao.find(id);
}

@Override
public List<Users> findAll() {
    return usersDao.findAll();
}

@Override
public Users update(Users user) {
    return usersDao.update(user);
}

@Override
public void delete(Users user) {
    usersDao.delete(user.getUserId());
}
}
```

Además de esto, cada clase debe implementar los métodos adicionales que se han agregado en cada DAO. La implementación en nuestro caso es sencilla, no va a existir ninguna

ocasión en la que se llame a más de una función del DAO por cada función del servicio.

La metodología que se ha seguido es la siguiente:

- 1- Creación del método del servicio, nombrándolo con el mismo nombre que la función del DAO que va a llamar y especificando el tipo de valor que va a devolver la función.
- 2- Devolución del valor o valores (en caso de ser una lista) que devuelve la ejecución de la función del DAO.

Ejemplo de implementación de métodos adicionales con OrdersService:

### *OrdersService*

```
@Override
public void delete(Orders order) {
    ordersDao.delete(order.getOrderid());
}

@Override
public List<Orders> findAll() {
    return ordersDao.findAll();
}

@Override
public List<Orders> findOrdersByUserId(long userId) {
    return ordersDao.findOrdersByUserId(userId);
}
```

De igual manera se va a realizar para ProductsService, PaymentDetailsService, UsersService y OrderDetailsService.

## **End-point**

### Control de sesión

En el apartado 4.4.2 se habló de cómo se mantenía la sesión en la aplicación, comentaremos cual es el cometido del servidor en este procedimiento.

El servidor es el encargado de crear los SSOTokens (objeto que lleva datos del usuario e información relacionada con la sesión) y de almacenarlos en el SSOTokenMap; este último incluye métodos para manejar los SSOTokens, a destacar el método que comprueba si

existe un SSOToken (a partir de ahora token) en el Map y el que permite agregarle tokens.

### *SSOToken*

La clase SSOToken sirve para crear un token que identifique de forma unívoca la sesión de un usuario. Tiene cuatro propiedades:

```
public SSOToken(UsersDto user, Date expiration, long userId)
{
    super();
    Random random = new Random();

    this.tokenId = new String("" + Math.abs(random.nextInt()));
    this.user = user;
    this.expiration = expiration;
    this.userId = userId;
}
```

- **tokenId**: Mediante un número aleatorio crea un identificador único, este objeto es el que se envía al cliente para mantener la sesión con el token.
- **user**: Este objeto es del tipo UsersDto se hablará en más detalle de él en el UserRestEndpoint. Básicamente lleva la información necesaria para el login, su email y password.
- **Expiration**: Tiempo de duración de la sesión.
- **userId**: ID del usuario.

El resto del código son setters y getters de las variables mencionadas arriba.

### *SSOTokenMap*

Esta clase funciona como un singleton<sup>7</sup>, este comportamiento se consigue declarando la clase como enum y escribiendo INSTANCE en la primera línea de código. En el singleton se van a almacenar los tokens para mantener la sesión de los usuarios. Destacar las cuatro primeras líneas del código:

---

<sup>7</sup> El patrón de diseño **singleton** (instancia única) está diseñado para 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.

### **INSTANCE;**

```
private static final Logger LOGGER =  
Logger.getLogger(SSOTokenMap.class);  
  
protected Map<String, SSOToken> tokens =  
Collections.synchronizedMap(new HashMap<String, SSOToken>());  
  
private String timeout = "3600000";
```

- **LOGGER** nos permite utilizar log4j para sacar por consola los cambios que se realicen en el singleton.
- **tokens**, es un Map donde se va a almacenar cada token de cada usuario que quiera mantener una sesión activa.
- **timeout**, es el tiempo de expiración de la sesión, como valor por defecto se establece a 1 hora (3600000 en milisegundos).

**SSOTokenMap** dispone de diferentes métodos como deleteToken (borra un token según su ID), getTokens (devuelve todos los tokens que hay en el map), cleanUpTokens, etc.

Pero en el endpoint se han utilizado fundamentalmente tres:

- **getUser(String tokenId)**: Se ha utilizado para comprobar si la sesión del usuario sigue activa, si la función devuelve null significa que la sesión ha expirado.
- **addSSOToken(SSOToken ssoToken)**: Agrega un token al singleton.
- **getSSOToken(ssoTokenReqId)**: Recupera el SSOToken, normalmente para usar setExpiration(expirationTime) y así renovar la sesión del usuario.

### *Comprobación de sesión estándar*

Se realiza en dos pasos:

Primero se comprueba la recepción de un token en la cabecera del request del cliente.

```
String ssoTokenReqId = req.getHeader("Authorization");  
if (ssoTokenReqId != null) {
```

En caso de que se reciba un token, se revisa que éste se encuentre en el singleton, si se cumple la condición nuestro usuario está logueado y se renueva el tiempo de expiración.

```
if (SSOTokenMap.INSTANCE.getUser(ssoTokenReqId) != null)  
Date expirationTime = new Date();  
expirationTime.setTime(expirationTime.getTime() + (3600 * 1000));
```

```
SSOTokenMap.INSTANCE.getSSOToken(ssoTokenReqId).setExpiration(expirationTime);
```

Si no se cumple cualquiera de las dos condiciones anteriores se devuelve un código de error 401 (no autorizado).

```
return Response.status(HttpStatus.UNAUTHORIZED.value()).build();
```

## Métodos Genéricos en el Endpoint

Todas las clases del endpoint comparten o partieron desde un CRUD que funcionaba igual para todas las clases, veamos un ejemplo con ProductsRestEndpoint:

### Create

```
@Transactional
@POST
@Path("/")
@Produces("application/json")
@Consumes("application/json")
public Products create(@RequestParam Products product) {

    return productsService.create(product);

}
```

Con la anotación **@POST** indica que espera un post en la dirección base de products (**@Path**), el método produce y consume datos del tipo JSON (**@Produces** y **@Consumes**), la etiqueta **@RequestParam** indica que espera un objeto del tipo Products, con éste se va a crear un nuevo producto usando el **servicio** asociado a products. La etiqueta **@Transactional** informa que se va a realizar una transacción en la que se pueden perder datos, con esta anotación se consigue que se cree una copia del objeto que se va a modificar, para recuperarla suponiendo que haya algún error en la operación.

### Read

```
@GET
@Path("/{id}")
@Produces("application/json")
@Consumes("application/json")
public Products findById(@PathParam("id") Long id) {

    return productsService.findById(id);

}
```

Cuando se recibe un GET a la dirección “products/{id}”, se recoge ese ID con la anotación **@PathParam** que sirve para inyectar el valor del parámetro de la URI que se define en el

Path, dentro del método. De esta manera se puede utilizar el servicio para devolver el producto asociado a ese ID.

```
@GET
@Path("")
@Produces("application/json")
@Consumes("application/json")
public List<Products> findAll() {

    return productService.findAll();

}
```

Si el cliente hace un **GET** a la dirección base de products, obtendrá una lista con todos los productos en la base de datos.

```
@GET
@Path("/response/{id}")
@Produces("application/json")
@Consumes("application/json")
public Response findByIdResponse(@PathParam("id") Long id) {

    Products product = productService.findById(id);

    return
Response.status(HttpStatus.ACCEPTED.value()).entity(product).build();

}
```

Un GET a la dirección “/response/{id}”, devolverá el producto asociado a ese ID en la respuesta HTTP con un código 200 (OK!).

### *Update*

```
@Transactional
@PUT
@Path("/{id}")
@Produces("application/json")
@Consumes("application/json")
public Products update(@RequestParam Products product,
@PathParam("id") Long id) {

    Products productRecovered = productService.findById(id);
    product.setProductId(id);
    BeanUtils.copyProperties(product, productRecovered);

    return productService.update(productRecovered);

}
```

Un **PUT** a la dirección “products/{id}” obtendrá el producto con el ID que se recibe en la URI y actualizará la información del producto recuperado de la base de datos, con el objeto que recibe el método con la anotación **@RequestParam**.



## Delete

```
@Transactional
@DELETE
@Path("/{id}")
@Produces("application/json")
@Consumes("application/json")
public Boolean delete(@PathParam("id") Long id) {

    Products productRecovered = productService.findById(id);

    productService.delete(productRecovered);

    return true;
}
```

Cuando el cliente haga un **DELETE** en la dirección “products/{id}”, se llamará al servicio para eliminar el producto que coincida con el ID que se recoge de la dirección con **@PathParam**.

## UsersDTO[36]

Primero definamos que es un DTO:

“**Objeto de Transferencia de Datos (DTO)** por sus siglas en inglés) es un objeto que transporta datos entre procesos. La motivación de su uso tiene relación con el hecho que la comunicación entre procesos es usualmente realizada mediante interfaces remotas, donde cada llamada es una operación costosa. Como la mayor parte del costo de cada llamada está relacionado con el tiempo round-trip entre el cliente y servidor, una forma de reducir el número de llamadas es usando un objeto (el DTO) que agrega los datos que habrían sido transferidos por cada llamada, pero que son entregados en una sola invocación.

La diferencia entre DTO y Objetos de Negocio (Business Objects) o Data Access Objects (DAO) es que un DTO no tiene más comportamiento que almacenar y entregar sus propios datos (accessors and mutators).”

UsersDTO va a almacenar la información justa y necesaria para poder realizar el inicio de sesión en nuestro servidor. Esta clase está formada por dos objetos: **email** y **password**. Se implementa un constructor y los getters/setters para estas variables.

Como se puede ver supone un ahorro de costo por llamada, ya que Users está formado por doce campos y UsersDTO sólo por dos. Se va a usar a menudo en el endpoint.

## UserRestEndpoint

UserRestEndpoint se va a encargar de mantener la sesión del usuario con signin, de devolver la información del usuario con getUserInfo y de redefinir los métodos genéricos create y update. Con la anotación **@Path** debajo de la etiqueta **@Controller**, se define la dirección base de este endpoint.

```
@Controller
@Path("users")
```

### Create

```
@Transactional
    @POST
    @Path("/signup")
    @Produces("application/json")
    @Consumes("application/json")
    public Response create(@RequestParam Users user) {
```

Este método recibe un usuario en la ruta “users/signup” mediante el método POST. Lo primero que hace es invocar al método create del servicio que tiene conectado. Tras esto, se da paso a crear la sesión:

```
Date expirationTime = new Date();
    expirationTime.setTime(expirationTime.getTime() + (3600 *
1000));

    UsersDto userDto = new UsersDto(user.getEmail(),
user.getPassword());
    Long userId =
(usersService.findUserByEmail(userDto.getEmail())).getUserId();

    SSOToken ssoToken = new SSOToken(userDto, expirationTime,
userId);

    SSOTokenMap.INSTANCE.addSSOToken(ssoToken);
```

Se genera la fecha de expiración de la sesión, se crea un user del tipo UsersDTO y se obtiene el ID del usuario a partir de la recuperación total del usuario mediante el email que recibe el método. Con estos datos ya se puede generar un nuevo token e introducir su ID en el singleton con el método addSSToken.

```
String token = ssoToken.getTokenId();
    System.out.println(token);
```

```

        Response response =
Response.status(HttpStatus.CREATED.value())
        .header("Access-Control-Allow-Origin",
"http://localhost:8080")
        .header("Access-Control-Allow-Credentials", true)
        .header("Access-Control-Allow-Methods", "POST,
GET, OPTIONS, PUT, DELETE, HEAD")
        .header("Access-Control-Allow-Headers", "Content-
Type, authorization").header("token", token)
        .entity(token).build();
        return response;

```

Finalmente se obtiene el ID del token que se acaba de crear para devolverlo en el body de la respuesta HTTP. En la cabecera de la respuesta se configura el CORS para evitar problemas de dominios y se añade el tokenId en el body con el método entity.

### Signin

```

@POST
@Path("/signin")
@Produces("application/json")
@Consumes("application/json")
public Response signin(@RequestParam UsersDto userDto, @Context
HttpServletRequest req)

```

Esta función recibe un usuario y accede al Request que envía el cliente gracias a la anotación **@Context**, que es comúnmente usada para tener acceso al request o response.

Se extraen los datos necesarios para comprobar la sesión como: el email, password, si el usuario envía un token en el request, etc. El token se extrae del header del request con el nombre **“Authorization”**.

```

String ssoTokenReqId = req.getHeader("Authorization");

Users user =
userService.findUserByEmail(userDto.getEmail());
String pass = userDto.getPassword();
Long userId =
(userService.findUserByEmail(userDto.getEmail())).getUserId();

Date expirationTime = new Date();
expirationTime.setTime(expirationTime.getTime() + (3600 *
1000));

```

La primera comprobación que hace el servidor es ver si la contraseña que envía el cliente coincide con la del usuario asociado al email, si no coincide el método termina y devuelve un response con un código HTTP no autorizado.

```

if (pass.equals(user.getPassword())) {

```

Suponiendo que la contraseña es correcta, la siguiente comprobación consiste en revisar si el cliente ha enviado un token, si no lo ha enviado crea uno nuevo, lo añade al singleton y devuelve un response con código HTTP OK! y el token en el body.

```
if (ssoTokenReqId == null) {  
    SSOToken ssoToken = new SSOToken(userDto, expirationTime, userId);  
    SSOTokenMap.INSTANCE.addSSOToken(ssoToken);  
}
```

En caso de que sí haya enviado un token el cliente, se comprueba que este token se encuentra dentro del singleton, si no es así, se crea un nuevo token y se devuelve al cliente en el response con un código HTTP OK!

```
if (SSOTokenMap.INSTANCE.getSSOToken(ssoTokenReqId) == null)
```

Si por el contrario el token sí que se encuentra dentro del singleton, entonces simplemente se renueva la sesión del usuario y se devuelve un código de estado OK!

```
SSOTokenMap.INSTANCE.getSSOToken(ssoTokenReqId).setExpiration(expirationTime);
```

### *Update*

```
@Transactional  
@PUT  
@Path("/update")  
@Produces("application/json")  
@Consumes("application/json")  
public Response update(@RequestParam Users user, @Context  
HttpServletRequest req)
```

Cuando se recibe un PUT a “users/update” con un usuario en el request, se procede a actualizar la información de este usuario. Se realiza la comprobación de sesión estándar en el servidor. Suponiendo que el cliente ha iniciado la sesión correctamente, se actualiza la información del usuario con la información que recibe en el cuerpo del request.

### *Get User Info*

```
@GET  
@Path("/getUserInfo")  
@Produces("application/json")  
@Consumes("application/json")  
public Response getUserInfo(@Context HttpServletRequest req) {
```

Get User Info va a devolver la información del usuario asociada con el token que recibe el servidor del cliente. Primero se hace una comprobación de sesión estándar. Suponiendo que el usuario está logueado, se recupera la información del usuario asociada al token y se devuelve en el response con un código HTTP OK!

```
UsersDto userDto = SSOTokenMap.INSTANCE.getUser(ssoTokenReqId);  
Users user = usersService.findUserByEmail(userDto.getEmail());
```

## ProductsRestEndpoint

ProductsRestEndpoint se va a encargar de implementar el CRUD para los productos y de devolver una lista de productos según una lista de IDs de productos. Con la anotación `@Path` debajo de la etiqueta `@Controller`, se define la dirección base de este endpoint.

```
@Controller  
@Path("products")  
  
@POST  
@Path("productsId")  
@Produces("application/json")  
@Consumes("application/json")  
public Response getProductsByProductsId(@RequestParam List<Long>  
productsId) {
```

Cuando se ejecute un POST sobre la dirección “products/productsId”, se comprobará primero que la lista no esté vacía (para evitar la null pointer exception), a continuación se invocará al servicio para que encuentre la lista de productos que coinciden con la lista de IDs recibidos; finalmente se devuelve un response con los productos y un código de estado ACCEPTED.

```
List<Products> products =  
productsService.getProductsByProductsId(productsId);  
Return  
Response.status(HttpStatus.ACCEPTED.value()).entity(products).build();
```

## PaymentDetailsRestEndpoint

La clase PaymentDetailsRestEndpoint se encarga de hacer el CRUD a los detalles de pago y además implementa un método adicional necesario para el cliente, este método sirve para encontrar los detalles de pago relacionados con un pedido

La dirección base de este endpoint es:

```
@Path("paymentDetails")
```

### *Find Payment Details By Order Id*

```
@GET  
@Path("orderId/{id}")  
@Produces("application/json")  
@Consumes("application/json")
```

```
    public Response findPaymentDetailByOrderId(@PathParam("id") Long
orderId) {
```

Al recibir un GET en “paymentDetails/orderId{id}”, se evaluará el ID de la dirección para encontrar los detalles de pagos asociados al ID de ese pedido. El resultado se devuelve en un response con un código HTTP ACCEPTED.

```
PaymentDetails paymentDetails =
paymentDetailsService.findPaymentDetailByOrderId(orderId);
return
Response.status(HttpStatus.ACCEPTED.value()).entity(paymentDetails).build
();
```

## OrderDetailsRestEndpoint

La clase OrderDetailsRestEndpoint va a llevar a cabo las peticiones del cliente relacionadas con los detalles de los pedidos, implementa el CRUD básico más otros dos métodos que se agregaron posteriormente. La dirección base de este endpoint es:

```
@Controller
@Path("orderDetails")
```

### *Get ProductsId By OrderId*

```
@GET
@Path("orderId/{id}")
@Produces("application/json")
@Consumes("application/json")
public Response getProductsIdByOrderId(@PathParam("id") Long orderId,
@Context HttpServletRequest req) {
```

Un método GET sobre la dirección “orderDetails/orderId/{id}”, comprobará que el usuario está logueado y en caso afirmativo, devolverá la lista de productos asociada con el ID de pedido con la ayuda del servicio.

```
List<Long>orderDetails=orderDetailsService.getProductsIdByOrderId(orderId
);
```

Para terminar devuelve la lista de productos en el body del response, con un código HTTP ACCEPTED.

```
Return
Response.status(HttpStatus.ACCEPTED.value()).entity(orderDetails).build()
;
```

### *Get OrderDetails By OrderId*

```
@GET
@Path("orderDetailsOrderId/{id}")
```

```

@Produces("application/json")
@Consumes("application/json")
public Response getOrderDetailsById(@PathParam("id") Long orderId) {
    Cuando el servidor recibe un GET sobre la dirección
    "orderDetails/orderDetailsOrderId/{id}", éste se encarga de llamar al servicio conectado a
    orderDetails, para devolver una lista con los detalles de pago asociados con el ID de pedido
    que recibe el método en la URL, comprobando que el ID que recibe no es nulo.

    if (orderId != null) {
        List<OrderDetails> orderDetails =
        orderDetailsService.getOrderDetailsById(orderId);
        En la respuesta devuelve la lista de detalles de pago con un código HTTP ACCEPTED.

    Return
        Response.status(HttpStatus.ACCEPTED.value()).entity(orderDetails).build()
    ;
}

```

## OrdersRestEndpoint

OrdersRestEndpoint atiende las peticiones del cliente relacionadas con los pedidos del usuario, implementa el CRUD básico y tres métodos adicionales que se explicarán en detalle. La dirección base es "orders". A diferencia de los demás endpoints, éste inyecta tres servicios necesarios para crear un pedido.

```

@Autowired
private OrdersService ordersService;
@Autowired
private OrderDetailsService orderDetailsService;
@Autowired
private PaymentDetailsService paymentDetailsService;

```

### Confirm

```

@Path("orders")
@Transactional
    @POST
    @Path("confirm")
    @Produces("application/json")
    @Consumes("application/json")
    public Response confirm(@RequestParam Orders order, @Context
    HttpServletRequest req) {
}

```

Este método es el responsable de crear un pedido mediante la información que recibe del cliente. El objeto orders que recibe incluye la información del pedido, una lista con los detalles del pedido y los detalles de pago.

Lo primero que hace este método es una comprobación de sesión estándar, posteriormente obtiene el `userId` (necesario para utilizar el constructor de `Orders` que definimos en el entity) y el total que extrae del objeto tipo `Orders` del request.

```
long userId =
(SSOTokenMap.INSTANCE.getSSOToken(ssoTokenReqId)).getUserId();
```

Con todo esto, crea el pedido y lo inserta en la base de datos mediante una llamada al servicio.

```
Orders orderInfo = new Orders(userId, order.getTotal());
Orders orderCreated = ordersService.create(orderInfo);
```

A continuación se recupera el ID del pedido que acabamos de crear y se procede a crear los detalles de pago con la información que recuperamos del request.

```
long orderId = orderCreated.getOrderId();
PaymentDetails paymentDetails = order.getPaymentDetail();
paymentDetails.setOrderId(orderId);
paymentDetailsService.create(paymentDetails);
```

Para terminar se crean los detalles del pedido con un bucle `for` y se devuelve un response con un código HTTP `CREATED`.

```
for (OrderDetails orderDetails : order.getOrderDetails()) {
    orderDetails.setOrderId(orderId);
    orderDetailsService.create(orderDetails);
}
return Response.status(HttpStatus.CREATED.value()).build();
```

### *Delete Order Roots*

```
@Transactional
@DELETE
@Path("orderId/{id}")
@Produces("application/json")
@Consumes("application/json")
public Response deleteOrderRoots(@PathParam("id") Long orderId,
@Context HttpServletRequest req) {
```

Este método se va a encargar de borrar todas las raíces de un pedido, eso conlleva borrar el pedido, los detalles del pedido y sus detalles de pago.

Tras hacer una comprobación de sesión estándar, el método va a recuperar el pedido relacionado con el ID del path y va a invocar a los servicios necesarios para borrar toda la información vinculada con el ID del pedido.

```
Orders orderRecovered = ordersService.findById(orderId);

orderDetailsService.deleteByOrderId(orderId);
ordersService.delete(orderRecovered);
paymentDetailsService.deleteByOrderId(orderId);
```



Finalmente devuelve un response con un código ACCEPTED.

```
return Response.status(HttpStatus.ACCEPTED.value()).build();
```

### *Find Order By UserId*

```
@GET
@Path("/userId/")
@Produces("application/json")
@Consumes("application/json")
public Response findOrdersByUserId(@Context HttpServletRequest req) {
```

El método findOrdersByUserId va a devolver los pedidos asociados al usuario que hace la petición. Primeramente se hace una comprobación estándar de sesión, a continuación se invoca al servicio para que recupere los pedidos vinculados con el userId del token. Este resultado se serializa a JSON y se envía en el body del response con un código OK!

```
String jsonResponse = new
Gson().toJson(ordersService.findOrdersByUserId(userId));
return
Response.status(HttpStatus.OK.value()).entity(jsonResponse).build();
```

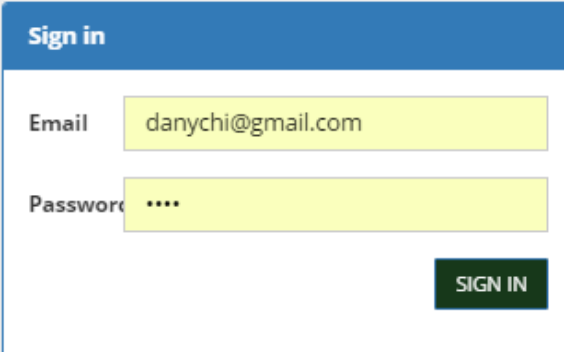
## 6. Test

Para la comprobación de la correcta funcionalidad se van a realizar una serie de tests funcionales en la aplicación web.

## Usuarios

### Sign In

Al introducir los datos de sesión correctamente la aplicación envía al usuario a la pantalla de inicio, la cabecera cambia permitiendo acceder al perfil de usuario y cerrar la sesión.



The image shows a web form titled "Sign in" with a blue header. It contains two input fields: "Email" with the value "danychi@gmail.com" and "Password" with masked characters "....". A dark green "SIGN IN" button is located at the bottom right of the form.



Si se introducen unos datos incorrectos, se mostrará un mensaje informando que los datos no son válidos.

**Sign in**

Email

Password

**SIGN IN**

Invalid credentials.

## Sign Up

Si se rellena el formulario para registrarse correctamente, la página nos envía a la pantalla de inicio con la sesión iniciada. En caso de que dejemos algún campo vacío, se mostrará un mensaje de error diciendo que no ha sido posible realizar el registro.

Sign up	
Email	<input type="text" value="fernando.losilla@gmail.com"/>
User	<input type="text" value="fernando"/>
Pass	<input type="password" value="****"/>
Last Name	<input type="text" value="López"/>
First name	<input type="text" value="Losilla"/>
Address	<input type="text" value="Antiguo Hospital de Marina, Campu"/>
Dni	<input type="text" value="41234567"/>
City	<input type="text" value="Cartagena (Murcia)"/>
State	<input type="text" value="Spain"/>
Postal Code	<input type="text" value="30202"/>
Phone	<input type="text" value="612345678"/>
<input type="button" value="SIGN UP"/>	

Sign up	
Email	<input type="text" value="Email"/>
User	<input type="text" value="username"/>
Pass	<input type="password" value="Password"/>
Last Name	<input type="text" value="last_name"/>
First name	<input type="text" value="first_name"/>
Address	<input type="text" value="billing_address"/>
Dni	<input type="text" value="dni"/>
City	<input type="text" value="city"/>
State	<input type="text" value="state"/>
Postal Code	<input type="text" value="postal_code"/>
Phone	<input type="text" value="phone"/>
<input type="button" value="SIGN UP"/>	

Failed to sign up.

LOGOUT



Hibernate se encarga de crear el usuario en la base de datos.

USER_ID	LAST_NAME	FIRST_NAME	EMAIL	BILLING_ADDRESS	DNI	CITY	STATE	POSTAL_CODE	PHONE	PASSWORD
1	26 López	Losilla	fernando.losilla@gmail.com	Antiguo Hospital de Marina, Campus Muralla del Mar	41234567	Cartagena (Murcia)	Spain	30202	612345678	1234

## User Profile

Si la sesión expira, desde cualquier sección de User Profile nos envía a la pantalla de login.

**Sign in**

Email

Password

**SIGN IN**

Failed to get the User Info.

## Account Settings

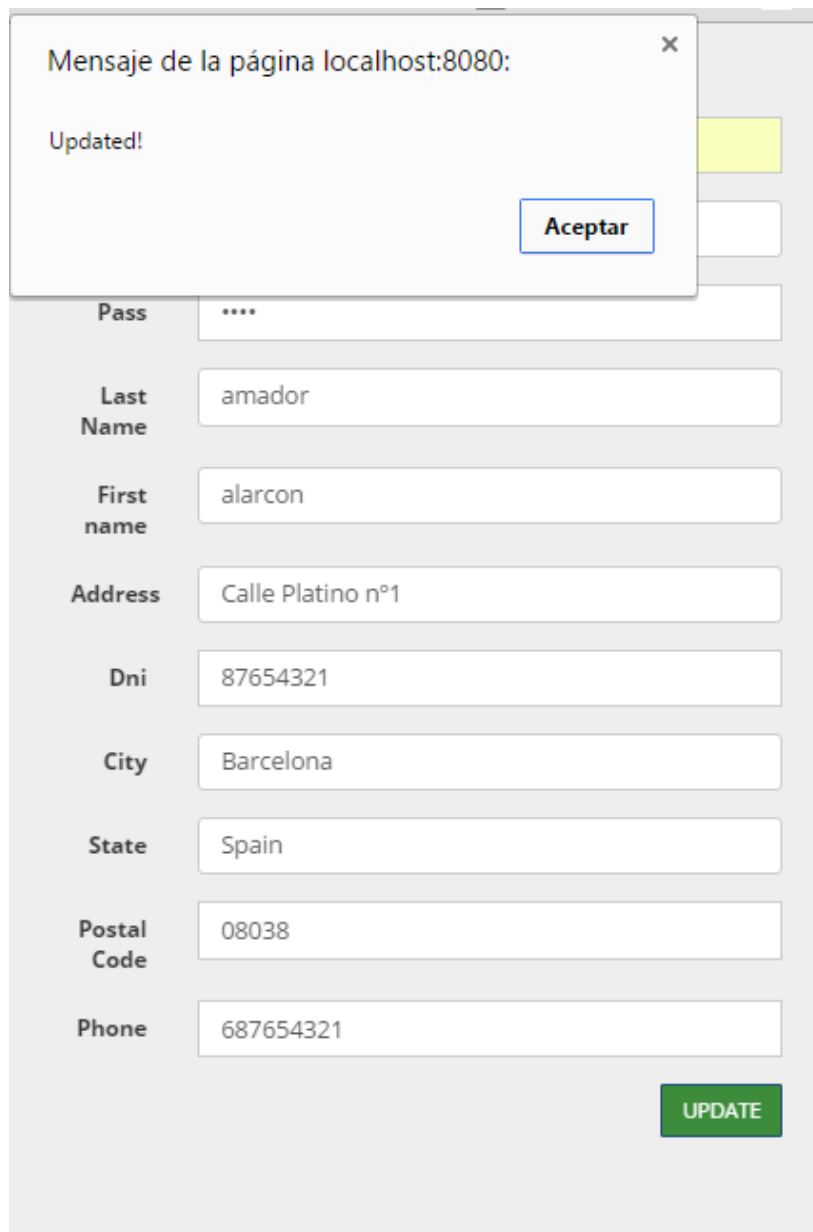
Si actualizamos la información de nuestro usuario, los cambios se verán reflejados en la base de datos al instante. En caso de que dejemos algún campo vacío, no podremos pulsar sobre el botón update.

USER_ID	LAST_NAME	FIRST_NAME	EMAIL	BILLING_ADDRESS	DNI	CITY	STATE	POSTAL_CODE	PHONE	PASSWORD
1	dani	dani	danychi@gmail.com	dani	1234	dani	dani	1234	674855605	1234

USER_ID	LAST_NAME	FIRST_NAME	EMAIL	BILLING_ADDRESS	DNI	CITY	STATE	POSTAL_CODE	PHONE	PASSWORD
1	amador	alarcon	danychi24@gmail.com	Calle Platino nº1	87654321	Barcelona	Spain	8038	687654321	4321

La página muestra una alerta reflejando que la operación se ha llevado a cabo con éxito.



Mensaje de la página localhost:8080:

Updated!

Aceptar

Pass: \*\*\*\*

Last Name: amador

First name: alarcon

Address: Calle Platino nº1

Dni: 87654321

City: Barcelona

State: Spain

Postal Code: 08038

Phone: 687654321

UPDATE

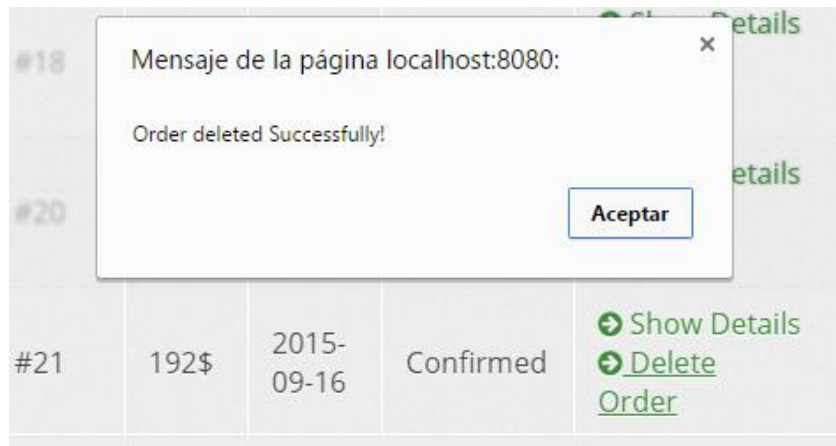
## My Orders

En My Orders van a aparecer todos los pedidos asociados a un usuario.

My Orders				
Order	Total	Order Date	Status	Actions
#13	133\$	2015-09-14	Sent	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#14	63\$	2015-09-14	Delivery Failure	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#17	159\$	2015-09-15	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#18	159\$	2015-09-15	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#20	132\$	2015-09-16	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
#21	192\$	2015-09-16	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>

<input type="checkbox"/>		Editar		Copiar		Borrar	ORDER_ID	STATUS_ID	USER_ID	TOTAL	ORDER_DATE	ACTIVE
<input type="checkbox"/>		Editar		Copiar		Borrar	13	3	1	133	2015-09-14	1
<input type="checkbox"/>		Editar		Copiar		Borrar	14	5	1	63	2015-09-14	1
<input type="checkbox"/>		Editar		Copiar		Borrar	17	1	1	159	2015-09-15	1
<input type="checkbox"/>		Editar		Copiar		Borrar	18	1	1	159	2015-09-15	1
<input type="checkbox"/>		Editar		Copiar		Borrar	20	1	1	132	2015-09-16	1
<input type="checkbox"/>		Editar		Copiar		Borrar	21	1	1	192	2015-09-16	1

Si borramos algún pedido nos aparecerá un mensaje diciendo que se ha borrado correctamente:



El pedido ya no aparece en la base de datos ni en la página:

### My Orders

Order	Total	Order Date	Status	Actions
#13	133\$	2015-09-14	Sent	<a href="#">Show Details</a> <a href="#">Delete Order</a>
#14	63\$	2015-09-14	Delivery Failure	<a href="#">Show Details</a> <a href="#">Delete Order</a>
#17	159\$	2015-09-15	Confirmed	<a href="#">Show Details</a> <a href="#">Delete Order</a>
#18	159\$	2015-09-15	Confirmed	<a href="#">Show Details</a> <a href="#">Delete Order</a>
#20	132\$	2015-09-16	Confirmed	<a href="#">Show Details</a> <a href="#">Delete Order</a>

	ORDER_ID	STATUS_ID	USER_ID	TOTAL	ORDER_DATE	ACTIVE
<input type="checkbox"/> Editar Copiar Borrar	13	3	1	133	2015-09-14	1
<input type="checkbox"/> Editar Copiar Borrar	14	5	1	63	2015-09-14	1
<input type="checkbox"/> Editar Copiar Borrar	17	1	1	159	2015-09-15	1
<input type="checkbox"/> Editar Copiar Borrar	18	1	1	159	2015-09-15	1
<input type="checkbox"/> Editar Copiar Borrar	20	1	1	132	2015-09-16	1

## Order Details

Aquí aparecen los detalles relacionados con un pedido, los detalles coinciden con los de la base de datos:



### Order Details from Order # 13

**STATUS:** Sent

**ORDER DATE:** 2015-09-14

User Info	Payment Method
Name: amador	Payment Method: VISA
Billing Address: jose de mora 42	Holder Name: DANIEL ALARCON
Telephone: 123456789	Number: 3021
Email:danychi@gmail.com	Exp. Date: 2016-07-10
City: baza	
Postal Code: 18800	

#### Items Ordered

Title	Quantity	Total	Cover
The Vaccines - English Graffiti	1	10\$	
Tesseract - One	5	65\$	
Dylan Ross - Hatch	4	32\$	



ORDER_ID	STATUS_ID	USER_ID	TOTAL	ORDER_DATE	ACTIVE
13	3	1	133	2015-09-14	1

ORDER_DETAIL_ID	ACTIVE	ORDER_ID	PRODUCT_ID	QUANTITY	TOTAL
13	1	13	1	1	10
14	1	13	2	5	65
15	1	13	4	4	32

PM_ID	DESCRIPTION	ORDER_ID	NUMBER	EXP_DATE	HOLDER_NAME
6	VISA	13	3021	2016-07-10	DANIEL ALARCON

## Tienda

En la tienda podremos consultar la información de cada producto, agregar diferentes artículos al carrito y finalmente hacer checkout. Los filtros funcionan correctamente:

**Type**

CD

VINYL

Digital Music

**Genre**


Indie & Alternative

Djent

Hip-hop

Rock


Latin music



**Periphery**

17\$


1



**Animals as Leaders**

17\$

1



**Måsstaden**


17\$

1

```
SELECT * FROM `products` WHERE `CATEGORY` LIKE 'DJENT' AND `TYPE` LIKE 'VINYL'
```

PRODUCT_ID	PRODUCTNAME	DESCRIPTION	CATEGORY	IMAGE	AUTHOR	PRICE	TYPE
17	Periphery	Periphery is the debut studio album by American pr...	Djent	periphery.jpg	Periphery	17	VINYL
18	Animals as Leaders	Animals as Leaders is an American instrumental pro...	Djent	aal.JPG	Animals as Leaders	17	VINYL
19	Måsstaden	Måsstaden is the debut album of the Swedish prog...	Djent	vil.jpg	Vildhjarta	17	VINYL

Se puede visualizar la información del producto, agregarlo al carrito, ver artistas similares (recomendados según el género del artículo), etc.



**English Graffiti**

10\$


1

The Vaccines - English Graffiti

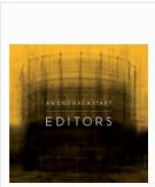
Category: Indie & Alternative  
Type: CD

The Vaccines 'English Graffiti,' is their first release in three years. The album was recorded in upstate New York, and the 11-song collection is the UK foursome's most accomplished work to date. 'English Graffiti' was produced David Fridmann (Flaming Lips, Tame Impala, MGMT) and Cole MGN (Ariel Pink, Beck). 'English Graffiti' is The Vaccines' first record made in the US, and finds group at the top of their game tracks like "Minimal Affection" recall the glory days of 80s synth-pop, "20/20" fires with machine gun precision, while "Handsome" connects with their power pop roots. The Vaccines' last record, 'The Vaccines Come Of Age,' reached # 1 in the UK and has sold over 400k copies. They've toured with everyone from The Rolling Stones, Arcade Fire, RHCP, Arctic Monkeys


**Perhaps you may be interested in the following artists:**



English Graffiti



An End Has a Start



Hamburg

En el carrito se van a mostrar los artículos que se hayan agregado.

**Cart**

	Title	Quantity	Amount	Total
✕	One	- 1 +	\$13.00	\$13.00
✕	White Lies for Dark Times	- 2 +	\$15.00	\$30.00
✕	Californication	- 1 +	\$10.00	\$10.00
✕	Nevermind	- 3 +	\$8.00	\$24.00
✕	A Rush of Blood to the Head	- 1 +	\$8.00	\$8.00
✕	English Graffiti	- 1 +	\$10.00	\$10.00
			Tax (21%):	\$19.95
			Shipping:	\$3.95
			<b>Total:</b>	<b>\$118.90</b>

Checkout

Al rellenar los datos de pago y confirmar el pedido, se mostrará un mensaje confirmando que se ha creado, al aceptar la notificación, la aplicación nos devolverá a la pantalla de inicio.

User Info

Name: amador  
Billing Address: jose de  
Telephone: 123456789  
Email: danychi@gmail.  
City: baza  
Postal Code: 18800

Mensaje de la página localhost:8080: x

Created!

Aceptar

Payment Details

Payment Method: VISA

Holder Name: DANIEL ALARCON AMADOR

Number: 3021

Expiration Date: 2016-07-16

LOGOUT

Is your info ok?  
You can update it from your user profile Goto!

CONFIRM



El pedido se ha creado, podemos comprobarlo en My Orders desde User Profile.

#34	119\$	2015-10-06	Confirmed	<a href="#">➔ Show Details</a> <a href="#">➔ Delete Order</a>
-----	-------	------------	-----------	--

Title	Quantity	Total	Cover
The Vaccines - English Graffiti	1	10\$	
Tesseract - One	1	13\$	
Ben Harper - White Lies for Dark Times	2	30\$	
Red Hot Chili Peppers - Californication	1	10\$	
Coldplay - A Rush of Blood to the Head	1	8\$	
Nivana - Nevermind	3	24\$	

## 7. Conclusiones y Líneas futuras

En este proyecto se han podido utilizar las tecnologías que están en auge y mejores resultados han dado a empresas y desarrolladores. Entre las tecnologías que se han usado se encuentran HTML5, Bootstrap, CSS3, JavaScript y AngularJS para el Front-End y MySQL, Hibernate, Spring/Jersey y Maven para el Back-End. La comunicación cliente-servidor ha seguido una arquitectura REST. La aplicación ha permitido que un usuario pudiese comprar productos que se habían almacenado previamente en una base de datos, además de gestionar la información relacionada con sus pedidos y actualizar sus datos de cliente.

Se ha aprendido a manejar las principales tecnologías de mayor utilización para el desarrollo web, pasando por montar y configurar un proyecto Maven desde cero, diseñar y desarrollar una interfaz dinámica para el cliente totalmente funcional, implementar servicios web con Jersey/Spring y consumir éstos en el cliente.

Entre las complicaciones que han aparecido, cabría destacar el aprendizaje de AngularJS, una tecnología sencilla para aplicaciones que ofrecen una funcionalidad básica, pero complicada para aplicaciones más complejas, requiriendo una comprensión más extensa del funcionamiento interno de Angular, sin embargo su uso está justificado en vista de los resultados obtenidos.

Tras la realización de este proyecto se abren nuevas líneas futuras que podrían ampliar y mejorar el trabajo.

Podría integrarse diferentes tests que ayudarían a testear la aplicación. La realización de **tests unitarios** ayudaría a comprobar el correcto funcionamiento de nuestras clases, se podría utilizar tecnologías como **JUnit** o **Mockito**. Por otra parte también se podrían realizar **tests de integración** en las que se combinarían las clases sobre las que previamente se habrían hecho tests unitarios, para comprobar su funcionamiento en grupo usando **Maven** para las pruebas.

Una mejora que se podría aplicar a este proyecto es la mejora de la seguridad con **Spring**

**Security**[37]. Spring Security es un framework que se centra en proporcionar la autenticación y control de acceso para aplicaciones Java, es el estándar para asegurar aplicaciones basadas en Spring. En el servidor se podría implementar la versión oficial de los **JWT** (JSON Web Tokens) utilizando la librería **jose.4.j**, que nos permitiría cifrar y verificar con una firma digital los datos relacionados con la autenticación.

## 8. Bibliografía

- [1] “Qué es HTML5.” [Online]. Available: <http://hipertextual.com/archivo/2013/05/entendiendo-html5-guia-para-principiantes/>. [Accessed: 03-Oct-2015].
- [2] “HTML5 Example.” [Online]. Available: [http://www.w3schools.com/html/html5\\_browsers.asp](http://www.w3schools.com/html/html5_browsers.asp). [Accessed: 03-Oct-2015].
- [3] “CSS Tutorial.” [Online]. Available: <http://www.w3schools.com/css/>. [Accessed: 03-Oct-2015].
- [4] “CSS Syntax.” [Online]. Available: [http://www.w3schools.com/css/css\\_syntax.asp](http://www.w3schools.com/css/css_syntax.asp). [Accessed: 03-Oct-2015].
- [5] Wikipedia, “Twitter Bootstrap.” [Online]. Available: [https://es.wikipedia.org/wiki/Twitter\\_Bootstrap](https://es.wikipedia.org/wiki/Twitter_Bootstrap).
- [6] Wikipedia, “Php.” [Online]. Available: <https://es.wikipedia.org/wiki/PHP>.
- [7] “Capítulo 1. Introducción (Introducción a JavaScript).” [Online]. Available: [http://librosweb.es/libro/javascript/capitulo\\_1.html](http://librosweb.es/libro/javascript/capitulo_1.html). [Accessed: 03-Oct-2015].
- [8] Wikipedia, “JS.” [Online]. Available: <https://es.wikipedia.org/wiki/JavaScript>.
- [9] Wikipedia, “AngularJS.” [Online]. Available: <https://es.wikipedia.org/wiki/AngularJS>.
- [10] Wikipedia, “Single Page Application.” [Online]. Available: [https://es.wikipedia.org/wiki/Single-page\\_application](https://es.wikipedia.org/wiki/Single-page_application).
- [11] “javascript - What is two way binding? - Stack Overflow.” [Online]. Available: <http://stackoverflow.com/questions/13504906/what-is-two-way-binding>. [Accessed: 03-Oct-2015].

- [12] “AngularJS: Developer Guide: Conceptual Overview.” [Online]. Available: <https://docs.angularjs.org/guide/concepts>. [Accessed: 03-Oct-2015].
- [13] Wikipedia, “Http.” [Online]. Available: [https://es.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol).
- [14] “JavaScript a boli: HTTP: diferencia entre POST y PUT.” [Online]. Available: <http://notasjs.blogspot.com.es/2013/07/http-diferencia-entre-post-y-put.html>. [Accessed: 03-Oct-2015].
- [15] “HTTP Response Status Codes - YDN.” [Online]. Available: [https://developer.yahoo.com/social/rest\\_api\\_guide/http-response-codes.html](https://developer.yahoo.com/social/rest_api_guide/http-response-codes.html). [Accessed: 03-Oct-2015].
- [16] Wikipedia, “Hibernate.” [Online]. Available: <https://es.wikipedia.org/wiki/Hibernate>.
- [17] “Home: Java Platform, Enterprise Edition (Java EE) 7 Release 7.” [Online]. Available: <http://docs.oracle.com/javaee/7/index.html>. [Accessed: 03-Oct-2015].
- [18] Wikipedia, “Maven.” [Online]. Available: <https://es.wikipedia.org/wiki/Maven>.
- [19] “¿Qué es un pom.xml? | Administrando en un iglú en WordPress.com.” [Online]. Available: <https://pierfinazzi.wordpress.com/2011/04/14/%C2%BFque-es-un-pom-xml/>. [Accessed: 03-Oct-2015].
- [20] Wikipedia, “MySQL.” [Online]. Available: <https://es.wikipedia.org/wiki/MySQL>.
- [21] Wikipedia, “Spring/Jersey.” [Online]. Available: <https://es.wikipedia.org/wiki/JAX-RS>.
- [22] “XAMPP .” [Online]. Available: <https://www.apachefriends.org/es/index.html>. [Accessed: 03-Oct-2015].
- [23] “Sublime Tex.” [Online]. Available: <http://www.sublimetext.com/>. [Accessed: 03-Oct-2015].
- [24] Wikipedia, “Eclipse.” [Online]. Available: [https://es.wikipedia.org/wiki/Eclipse\\_\(software\)](https://es.wikipedia.org/wiki/Eclipse_(software)).
- [25] calendamaia, “Eclipse IDE Jennic,” 2010. [Online]. Available: <http://www.genbetadev.com/herramientas/eclipse-ide>. [Accessed: 03-Oct-2015].
- [26] T. TKALEC, “JSON Web Token Tutorial: Example using AngularJS & Laravel | Toptal.” [Online]. Available: <http://www.toptal.com/web/cookie-free-authentication->

with-json-web-tokens-an-example-in-laravel-and-angularjs. [Accessed: 03-Oct-2015].

- [27] “AngularJS: API: \$rootScope.” [Online]. Available: [https://docs.angularjs.org/api/ng/service/\\$rootScope](https://docs.angularjs.org/api/ng/service/$rootScope). [Accessed: 05-Oct-2015].
- [28] “AngularJs Remove duplicate elements in ng-repeat - Stack Overflow.” [Online]. Available: <http://stackoverflow.com/questions/20222555/angularjs-remove-duplicate-elements-in-ng-repeat>. [Accessed: 04-Oct-2015].
- [29] “Angular Checkbox Filters - JSFiddle.” [Online]. Available: <http://jsfiddle.net/65Pyj/>. [Accessed: 04-Oct-2015].
- [30] Wikipedia, “Clave Foránea.” [Online]. Available: [https://es.wikipedia.org/wiki/Clave\\_for%C3%A1nea](https://es.wikipedia.org/wiki/Clave_for%C3%A1nea).
- [31] Wikipedia, “log4j.” [Online]. Available: <https://es.wikipedia.org/wiki/Log4j>.
- [32] Wikipedia, “Descriptor de despliegue.” [Online]. Available: [https://es.wikipedia.org/wiki/Descriptor\\_de\\_despliegue](https://es.wikipedia.org/wiki/Descriptor_de_despliegue).
- [33] “ApplicationContext - maven-spring-seminar - Introducción a Spring - Introduction to development with Maven and Spring - Google Project Hosting.” [Online]. Available: <https://code.google.com/p/maven-spring-seminar/wiki/ApplicationContext>. [Accessed: 04-Oct-2015].
- [34] “The Generic DAO pattern in Java with Spring 3 and JPA 2.0 - CodeProject.” [Online]. Available: <http://www.codeproject.com/Articles/251166/The-Generic-DAO-pattern-in-Java-with-Spring-and>. [Accessed: 04-Oct-2015].
- [35] “EntityManager (Java EE 6 ).” [Online]. Available: <https://docs.oracle.com/javaee/6/api/javax/persistence/EntityManager.html>. [Accessed: 04-Oct-2015].
- [36] Wikipedia, “Objeto de Transferencia de Datos (DTO).” [Online]. Available: [https://es.wikipedia.org/wiki/Objeto\\_de\\_Transferencia\\_de\\_Datos\\_\(DTO\)](https://es.wikipedia.org/wiki/Objeto_de_Transferencia_de_Datos_(DTO)).
- [37] “Spring Security.” [Online]. Available: <http://projects.spring.io/spring-security/>. [Accessed: 06-Oct-2015].
- [38] Wikipedia, “Inyección de Dependencias (DI).” [Online]. Available: [https://es.wikipedia.org/wiki/Inyecci%C3%B3n\\_de\\_dependencias](https://es.wikipedia.org/wiki/Inyecci%C3%B3n_de_dependencias).
- [39] Wikipedia, “Bean.” [Online]. Available: <https://es.wikipedia.org/wiki/Bean>.

[40] Wikipedia, “Java Archive (JAR).” [Online]. Available:  
[https://es.wikipedia.org/wiki/Java\\_Archive](https://es.wikipedia.org/wiki/Java_Archive).

[41] Wikipedia, “Listener.” [Online]. Available:  
[https://en.wikipedia.org/wiki/Event\\_\(computing\)](https://en.wikipedia.org/wiki/Event_(computing)).

---