

IES POLITÉCNICO
HERMENEGILDO LANZ
GRANADA

DEPARTAMENTO
INFORMÁTICA Y COMUNICACIONES



PROYECTO FINAL CICLO SUPERIOR DESARROLLO DE
APLICACIONES WEB

AUTOR:
MARÍA PRIETO ROSALES
Granada, 13 de Junio de 2018

Contenido

Resumen	2
Justificación del proyecto	2
Objetivos	2
Desarrollo	2
Fundamentación teórica	2
Materiales y métodos	3
Entorno servidor	3
Entorno cliente	8
Diagramas de casos de uso	12
Arquitectura de la aplicación.....	13
Mapa de navegación	14
Guía de estilos	14
Modelo relacional	18
Servidor web	18
Cross-browser.....	21
Responsive.....	22
Resultados y análisis	23
Conclusiones.....	26
Líneas de investigación futuras	26
Bibliografía	27

Resumen

En el proyecto encontraremos una banca online en la que los usuarios podrán navegar entre las diferentes noticias, cuentas y tarjetas. También tendrán la opción de darse de alta como clientes. Estos pueden, tanto ver los movimientos que se han realizado en sus cuentas y el saldo del que disponen en este momento, como transferir dinero.

El administrador tiene los privilegios de ingresar dinero a cualquiera de los clientes, modificar sus datos o darlos de alta. En cuanto a la vista pública podrá modificar la sección de *Novedades* o ingresar nuevas.

Justificación del proyecto

Debido a los pocos recursos económicos de los que disponen hoy en día los estudiantes, se plantea la solución de poner en marcha una entidad bancaria online dedicada exclusivamente a ellos.

Objetivos

La finalidad de este proyecto es ofrecerles una forma de vida más económica adaptada a sus necesidades y escasos recursos, en la que puedan fácilmente realizar una matrícula en la universidad, obtener descuentos en el transporte o incluso fomentar la iniciativa empresarial de los jóvenes.

Desarrollo

Fundamentación teórica

El proyecto consta de dos partes:

- En la parte del servidor, un proyecto Maven en el que se encuentran tres módulos:
 - o Módulo business: En él se encuentran las clases **domain** (tablas BD → objetos), las **DAO**, donde se realizan las consultas a la BD **y** los **gestores**.
 - o Módulo ws (webservices): Aquí llegarán las peticiones del cliente.
 - o Módulo web: En este módulo incluiremos el proyecto de Angular una vez compilado.
- En el cliente, un proyecto de Angular en el que se diferencian los diferentes componentes, servicios y módulos.

- Cada componente consta de un fichero **html** en el que implementamos la vista, un fichero **typeScript** y un fichero de **css**, o en este caso, **scss (Sass)** en el que indicaremos los estilos que se aplicarán solo al componente al que pertenece.

Materiales y métodos

Entorno servidor

- Lenguaje **Java**, acompañado de **Hibernate** que nos da la posibilidad de transformar el modelo de bases de datos en objetos.

En el siguiente fichero configuramos la conexión con la base de datos.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</property>
    <property name="hibernate.default.catalog">hlantz_bank</property>
    <property name="hibernate.username">root</property>
    <property name="hibernate.password">password</property>
    <property name="hibernate.url">jdbc:mysql://localhost:3306/hlantz_bank</property>
  </session-factory>
</hibernate-configuration>
```

Una vez conectado con la base de datos, **Hibernate** nos permite a través de la ingeniería inversa generar las clases correspondientes a cada tabla de la base de datos. Aquí un ejemplo:

```

package com.hlanz.bank.business.domain;

/**
    IMPORTS
**/
@Entity
@Table(name = "usuarios", catalog = "hlanz_bank", uniqueConstraints =
@UniqueConstraint(columnNames = "dni"))
public class Usuarios implements java.io.Serializable {

    private Integer idUsuario;
    private String nombre;
    private String apellidos;
    private String dni;
    private Integer telefono;
    private String email;
    private int pin;
    private Set<Cuentas> cuentas = new HashSet<Cuentas>();

    public Usuarios() {
    }

    public Usuarios(String nombre, String apellidos, String dni, int pin) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.dni = dni;
        this.pin = pin;
    }

    public Usuarios(String nombre, String apellidos, String dni, Integer
telefono, String email, int pin,
        Set<Cuentas> cuentas) {
        this.nombre = nombre;
        this.apellidos = apellidos;
        this.dni = dni;
        this.telefono = telefono;
        this.email = email;
        this.pin = pin;
        this.cuentas = cuentas;
    }

    @Id
    @GeneratedValue(strategy = IDENTITY)

    @Column(name = "id_usuario", unique = true, nullable = false)
    public Integer getIdUsuario() {
        return this.idUsuario;
    }
}
/** ... Setters y Getters ... */

```

- **Spring**. Nos permitirá realizar la persistencia con la base de datos a través de peticiones a los **webservices** en la que podremos ejecutar consultas, modificaciones de los datos ya insertados, inserciones nuevas ...

Aquí el fichero de configuración de Spring para el **módulo business**.

(**Persistencia y transacciones**).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  ...
>
  <context:annotation-config />
  <!-- CHANGE BASE PACKAGE FOR APPLICATION NAME -->
  <context:component-scan base-package="com.hlanz.bank" />
  <!-- enable the configuration of transactional behavior based on
annotations -->
  <tx:annotation-driven transaction-manager="transactionManager"/>

  <bean id="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalContainerEntityManagerFactory
Bean">
    <property name="persistenceUnitName" value="hlantzPU"/>
  </bean>

  <bean
    class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanP
ostProcessor">
  </bean>

  <bean id="txManager"
    class="org.springframework.orm.jpa.JpaTransactionManager" >
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="*" timeout="400" propagation="REQUIRED" />
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="hlantzMethods"
      expression="execution(* com.hlanz.bank..*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="hlantzMethods" />
  </aop:config>
</beans>
```

En este fichero establecemos los servicios **REST** para poder hacer las peticiones posteriormente desde el cliente. En él configuramos el nombre de las rutas.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jaxrs="http://cxf.apache.org/jaxrs"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
  http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>

  <jaxrs:server id="hlanzServices" address="/rest/">
    <jaxrs:serviceBeans>
      <ref bean="servicioUsuarios"/>
      <ref bean="servicioCuentas"/>
      <ref bean="servicioMovimientos"/>
      <ref bean="servicioInfo"/>
    </jaxrs:serviceBeans>
    <jaxrs:providers>
      <bean class="org.codehaus.jackson.jaxrs.JacksonJsonProvider"/>
    </jaxrs:providers>
  </jaxrs:server>
  <bean id="servicioUsuarios"
class="com.hlanz.bank.ws.services.impl.ServicioUsuariosImpl" />
  <bean id="servicioInfo"
class="com.hlanz.bank.ws.services.impl.ServicioInfoImpl" />
  <bean id="servicioCuentas"
class="com.hlanz.bank.ws.services.impl.ServicioCuentasImpl" />
  <bean id="servicioMovimientos"
class="com.hlanz.bank.ws.services.impl.ServicioMovimientosImpl" />
</beans>
```

```

<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:spring/applicationContext.xml</param-value>
  </context-param>

  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

  <servlet>
    <servlet-name>CXFServlet</servlet-name>
    <display-name>CXF Servlet</display-name>
    <servlet-class>
      org.apache.cxf.transport.servlet.CXFServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
</web-app>

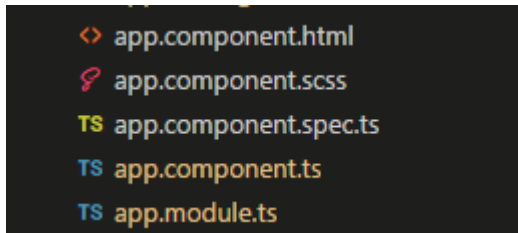
```

En el módulo web establecemos que cualquier petición que se realice a:

<host>:<puerto>/services/* pasará por CXF. Si además a la ruta añadimos **/rest** en la petición accederemos a nuestros servicios web que configuramos anteriormente.

Entorno cliente

- Angular 5 con la ayuda de Angular Material y Bootstrap.
 - o De forma predeterminada al generar el proyecto de Angular, este trae:



```
<> app.component.html
🔗 app.component.scss
TS app.component.spec.ts
TS app.component.ts
TS app.module.ts
```

En el `app.module.ts` (módulo general de la aplicación) se importarán los demás módulos y componentes a utilizar en la aplicación. Por ejemplo, cada vez que generemos un componente nuevo, este se añadirá a los ***declarations***.

Los módulos o componentes que queremos importar, en este caso los componentes de ***Angular Material***, el módulo ***Http*** para poder realizar las peticiones, etc los incluiremos en ***imports***.

Y por último para poder utilizar los servicios, los añadiremos en el apartado de ***providers***.

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule, ErrorHandler } from '@angular/core';

import {BrowserAnimationsModule} from '@angular/platform-
browser/animations';
import {MatToolbarModule,
      MatFormFieldModule,
      MatInputModule,
      . . .
      } from '@angular/material';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import {FlexLayoutModule} from '@angular/flex-layout';

import {HttpClientModule} from '@angular/common/http';

import { AppComponent } from './app.component';
import { NavComponent } from './nav/nav.component';
import { AppRoutingModuleModule } from './app-routing.module';
import { HomeComponent } from './home/home.component';
import { CuentasComponent } from './cuentas/cuentas.component';
import { CuentasService } from './shared/services/cuentas.service';
import { MovimientosComponent } from
'./movimientos/movimientos.component';
import { NoticiasComponent } from './noticias/noticias.component';
...
@NgModule({
  declarations: [
    AppComponent,
    NavComponent,
    HomeComponent,
    CuentasComponent,
    MovimientosComponent,
    NoticiasComponent,
    ...
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    MatProgressBarModule,
    MatButtonModule,
    MatFormFieldModule,
    FormsModule,
    ReactiveFormsModule,
    HttpClientModule,
    FlexLayoutModule,
    AppRoutingModuleModule
  ],
  providers: [UserService, CuentasService, MovimientosService ...

```

En el `app.component.html` definimos con sus selectores los componentes que se van a fijar y con el selector `<router-outlet>` los componentes que van a variar en función de la ruta. A continuación la explicación:

```
<app-nav></app-nav>
<router-outlet></router-outlet>
<app-footer></app-footer>
```

En nuestro caso, el **navbar** y el **footer** lo dejaremos fijo y lo que irá cambiando será el contenido.

Creamos un fichero en el que definiremos nuestras rutas y al componente al que hace referencia:

```
const routes: Routes = [
  { path: '', redirectTo: 'inicio', pathMatch : 'full' },
  { path: 'inicio', component: HomeComponent},
  { path: 'info/:id', component: NoticiasComponent},
  { path: 'movimientos', component: CuentasComponent, canActivate:
[AuthGuard] },
  { path : 'transferencia' , component : TransferenciaComponent,
canActivate: [AuthGuard]},
  { path: 'registro' , component: RegistroComponent},
  { path: 'admin', component: AdminComponent, canActivate:
[AdminGuard]},
  { path: 'perfil', component: PerfilComponent, canActivate:
[AuthGuard]},
  { path : 'cuentas', component: TarjetasComponent},
  { path: '**', redirectTo: 'inicio' }
];

@NgModule({
  exports: [ RouterModule ],
  imports: [
    CommonModule,
    RouterModule.forRoot(routes, {useHash: true})
  ],
  declarations: []
})
export class AppRoutingModule { }
```

Si en la ruta no indicamos nada o la ruta no existe, nos redirigirá al inicio.

En cada ruta podemos definir la directiva **CanActivate** que tomará como valor el resultado de un método. A esto se le llama **Routes Guard**.

El **auth-guard** devolverá *true* cuando la variable de sesión *user* esté creada. Solo así podremos acceder a las rutas que pertenecen a la parte privada de un usuario.

```

import { Injectable } from '@angular/core';
import { Router, CanActivate } from '@angular/router';
import { UserService } from './user.service';
@Injectable()
export class AuthGuardService implements CanActivate {
  constructor(public router: Router) {}
  canActivate(): boolean {
    if (!UserService.isAuthenticated()) {
      this.router.navigate(['']);
      return false;
    }
    return true;
  }
}

```

```

import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { HttpClient, HttpHeaders, HttpResponse } from '@angular/common/http';
import { Usuario } from '../model/usuario.model';
import { Login } from '../model/login.model';
import { CuentasService } from './cuentas.service';
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';

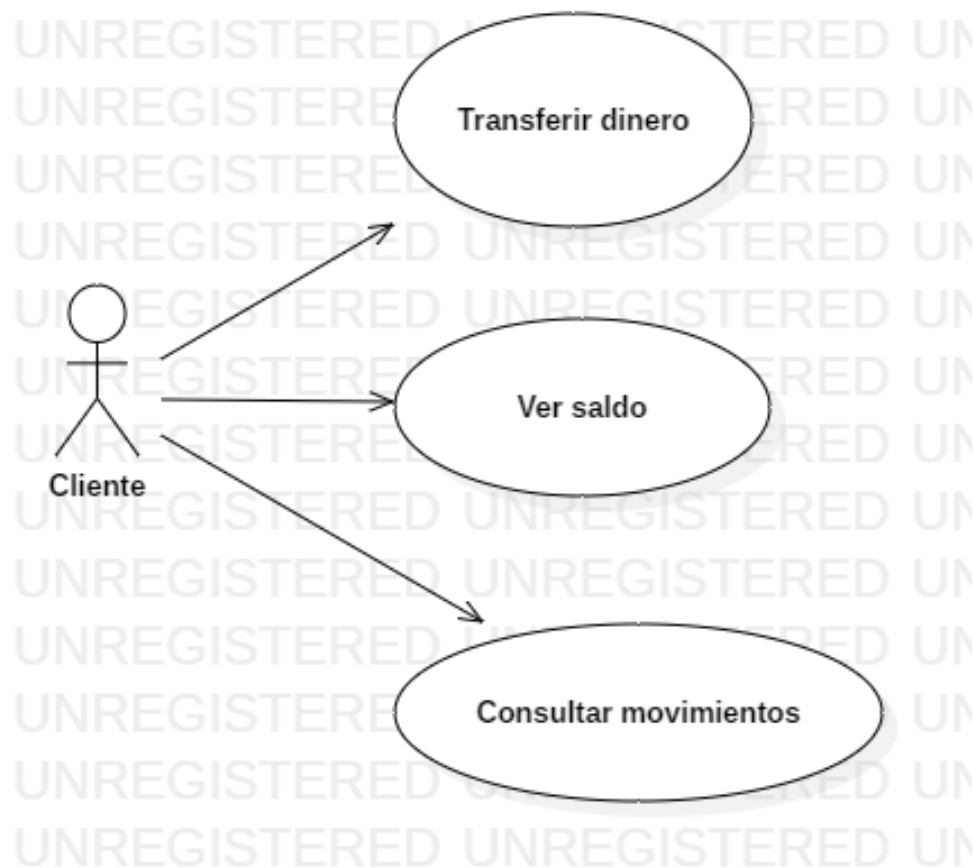
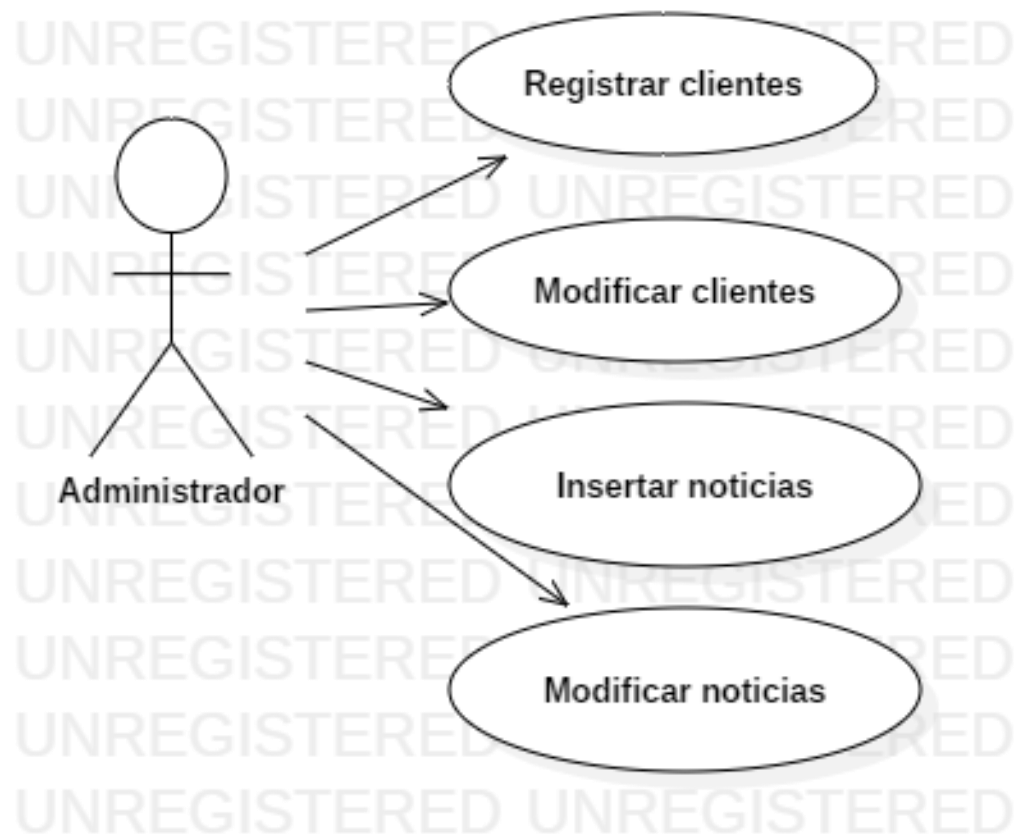
@Injectable()
export class UserService {
  encode : String;
  constructor(private http : HttpClient){
  }

  [ . . . ]

  public static isAuthenticated(): boolean {
    const token = localStorage.getItem('user');
    if(token != undefined)
      return true;
    return false;
  }
}

```

Diagramas de casos de uso

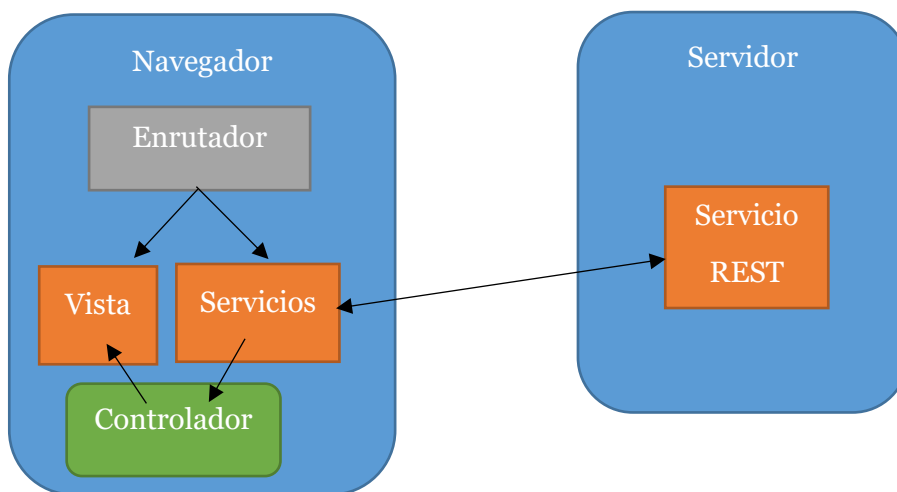




Arquitectura de la aplicación

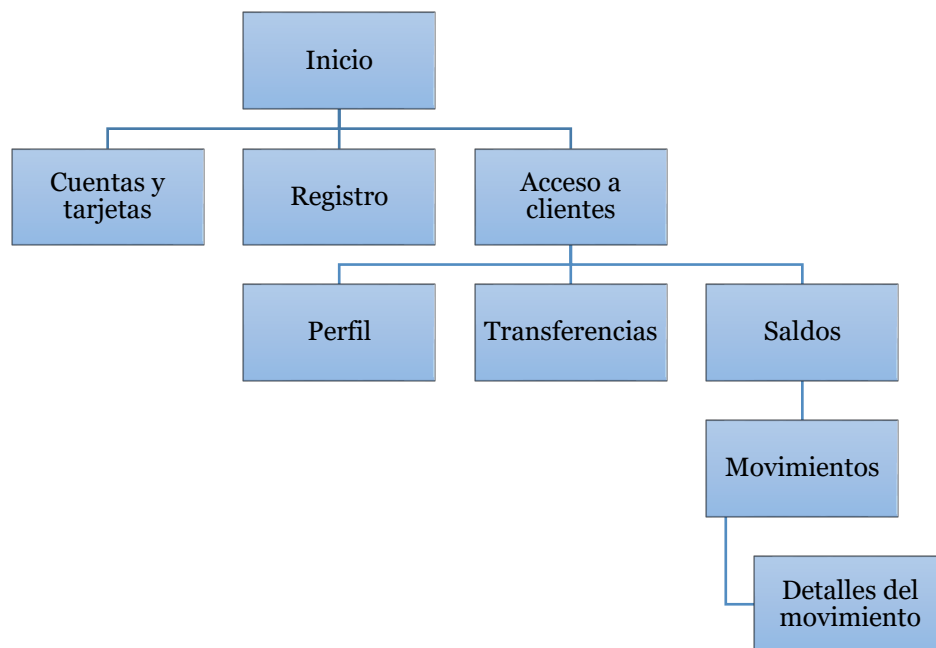
Una aplicación SPA consta de las siguientes partes:

- Router: Se encarga de seleccionar las vistas que solicita el cliente y navegar sin recargar la página, manteniendo los elementos y datos.
- Templates: Las vistas en HTML a las que accede el router.
- Controllers: Median entre el modelo y la vista.



El cliente hace una primera petición al servidor que trae el *index*. A partir de ahí, es el navegador el que se encarga de mostrar las vistas en función de la ruta.


Mapa de navegación



Guía de estilos

Paleta de colores

#00ADB5 → 

#303841 → 

#EEEEEE → 

Tipografía

- Títulos y cuerpo : Raleway
- Título h4 : **darkblue** – **bold**

Header

- Background-color : #EEEEEE
- Border-bottom : #00ADB5

Botones

- Background-color : #303841
- Color : #FFFFFF



Tablas

Cabecera:

- Background-color : #EEEEEE
- Color: #303841

Filas :hover

- Background-color: : #EEEEEE

Cabecera	Cabecera
Filas	Filas
Filas	Filas

Footer

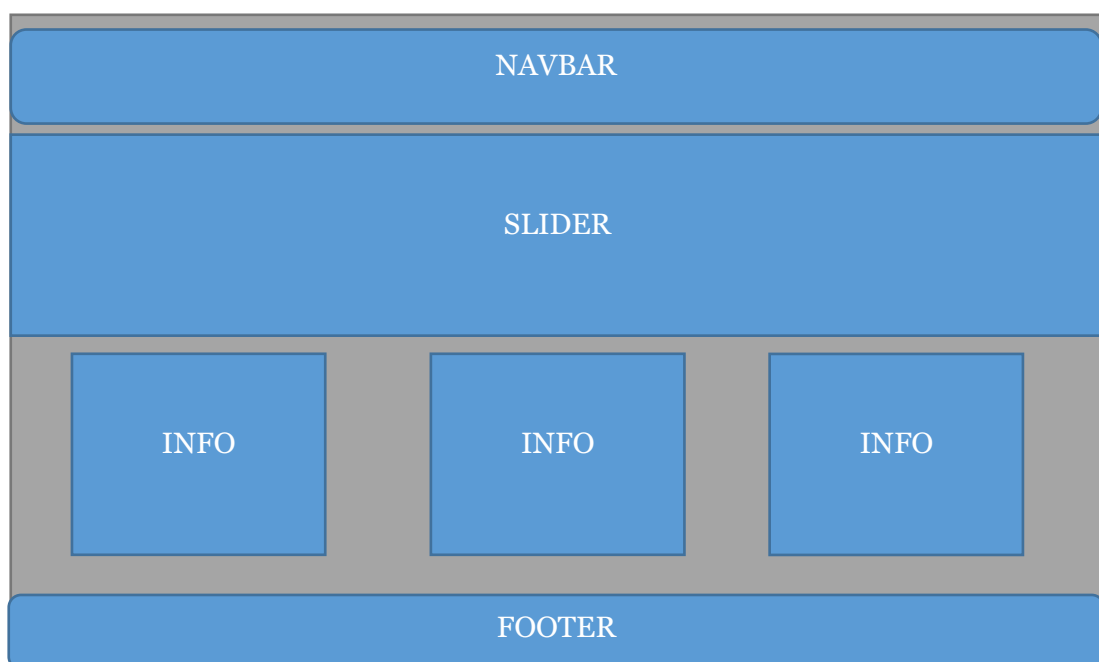
Background-color: #303841

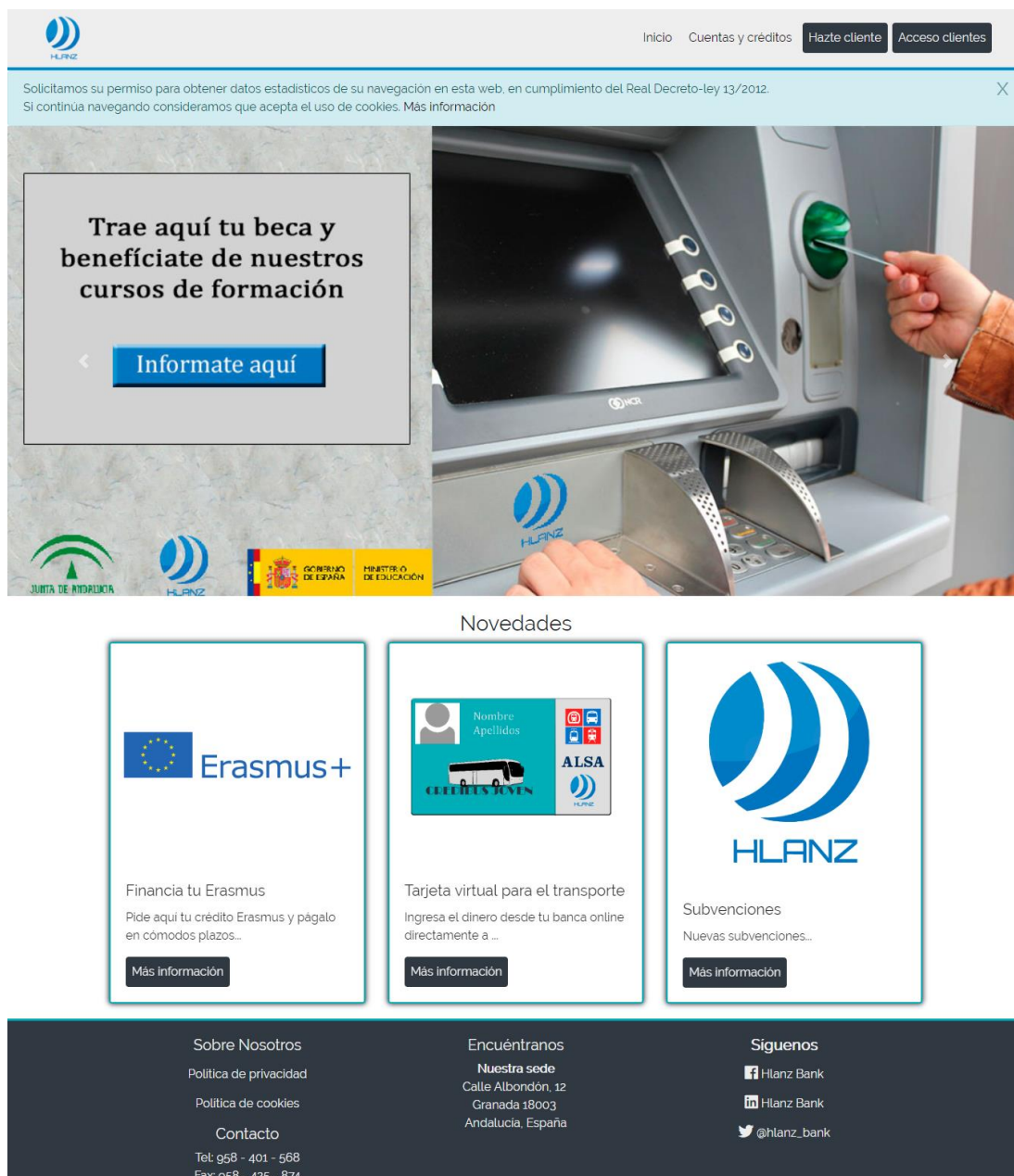
Color: #EEEEEE

Border-top: 3px solid #00ADB5

Column 1	Column 2	Column 3
----------	----------	----------

Diseño Inicio





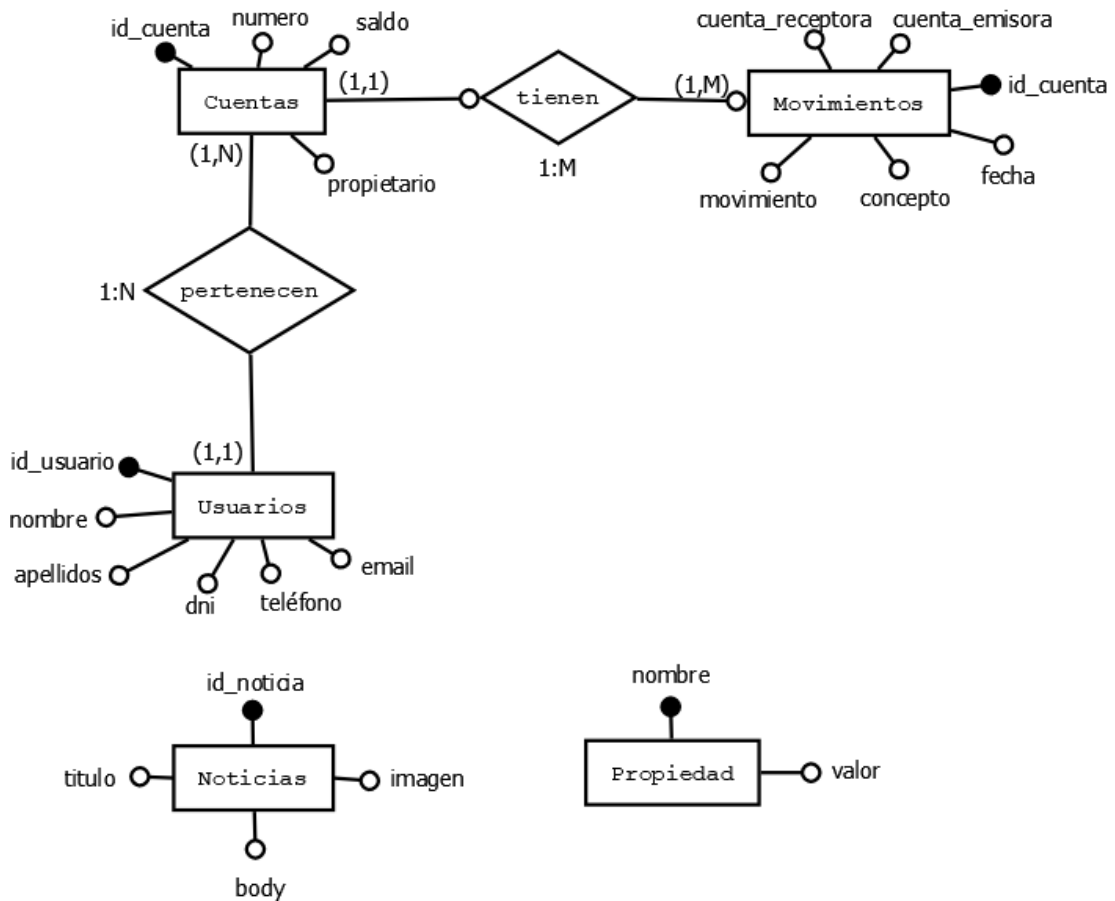
Modelo Entidad – Relación

En Hlanz Bank se necesita almacenar los datos de los usuarios que se registran y sus cuentas correspondientes.

De los usuarios nos gustaría saber su nombre, apellidos, DNI, teléfono de contacto y email. Nunca varios usuarios podrán tener el mismo DNI. Estos podrán tener las cuentas que les sean necesarias, con la condición de que las cuentas tendrán un único titular.

En las cuentas tendremos un número o IBAN, el saldo actual del que disponen y el propietario al que pertenecen.

Los usuarios podrán realizar transferencias y consultar sus movimientos, de los cuales se almacenará el número de cuenta que ha realizado la transferencia, el que la ha recibido, en qué fecha se efectuó el movimiento, el importe y un concepto informativo.



Las entidades *Noticias* y *Propiedad* son independientes. No guardan relación alguna con los Usuarios.

En la entidad de *Propiedad* será usada para guardar datos de “configuración” que pueden variar. Por ejemplo, el correo electrónico desde el que se envía el pin al registrar un nuevo usuario, el servidor de correo...

De esta manera, se puede configurar la aplicación sin necesidad de abrir código, tan solo modificando la base de datos.

Modelo relacional

Usuarios (id_usuario, nombre, apellidos, teléfono, email)

Cuentas (id_cuenta, numero, saldo, propietario)

Movimientos (id_movimiento, cuenta_emisora, cuenta_receptora, movimiento, concepto, fecha)

Noticias (id_noticia, titulo, body, imagen)

Propiedad (nombre, valor)

Servidor web

Configuración

Para el despliegue de mi aplicación web (WAR) se ha usado un servidor [Tomcat](#), un servidor o “contenedor” de Servlets.

La configuración de este se hace a través de ficheros .xml que encontramos en la carpeta **conf** una vez instalado el servidor.

do > Acer (C:) > Archivos de programa (x86) > Apache Software Foundation > Tomcat 8.5 > conf >			
Nombre	Fecha de modifica...	Tipo	Tamaño
Catalina	12/04/2018 17:24	Carpeta de archivos	
catalina.policy	31/08/2016 21:52	Archivo POLICY	13 KB
catalina.properties	31/08/2016 21:52	Archivo PROPERTI...	8 KB
context.xml	30/05/2018 19:01	Archivo XML	2 KB
jaspic-providers.xml	31/08/2016 21:52	Archivo XML	2 KB
jaspic-providers.xsd	31/08/2016 21:52	XML Schema File	3 KB
logging.properties	31/08/2016 21:52	Archivo PROPERTI...	4 KB
server.xml	04/06/2018 19:10	Archivo XML	8 KB
tomcat-users.xml	12/04/2018 17:23	Archivo XML	3 KB
tomcat-users.xsd	31/08/2016 21:52	XML Schema File	3 KB
web.xml	05/06/2018 16:42	Archivo XML	170 KB

En el **context.xml** configuramos la conexión con la base de datos con la etiqueta **<Resource>** y los siguientes parámetros.

```
<Resource name="jdbc/hlanz_bank" auth="Container"
type="javax.sql.DataSource"
maxTotal="100" maxIdle="30" maxWaitMillis="100000"
username="root" password="password"
driverClassName="com.mysql.jdbc.Driver"
url="jdbc:mysql://localhost:3306/hlanz_bank"/>
```

El parámetro **name** no podrá ser diferente de **“jdbc/hlanz_bank”** si no,

tendríamos que cambiar la configuración de Hibernate que configuramos en puntos anteriores.

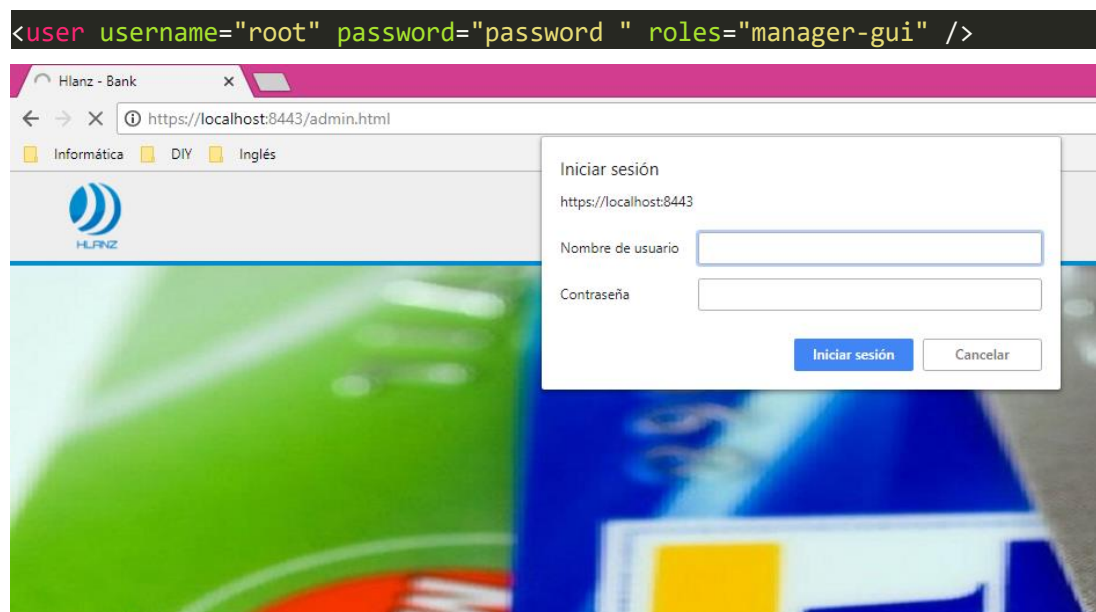
Si en algún momento la base de datos cambia de usuario, contraseña o host solo tendríamos que cambiar este fichero y no haría falta modificar el código.

Autenticación Básica

Para la autenticación básica el fichero es el siguiente: **web.xml**

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>
      Admin
    </web-resource-name>
    <url-pattern>/admin.html</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>manager-gui</role-name>
  </auth-constraint>
</security-constraint>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Basic Authentication</realm-name>
</login-config>
```

Cuando el usuario acceda a la ruta **/admin.html** nos encontraremos con la ventana de autenticación a la que solo podrán acceder los usuarios que definamos en el fichero **tomcat-users.xml** con el rol de **manager-gui**



Certificado SSL

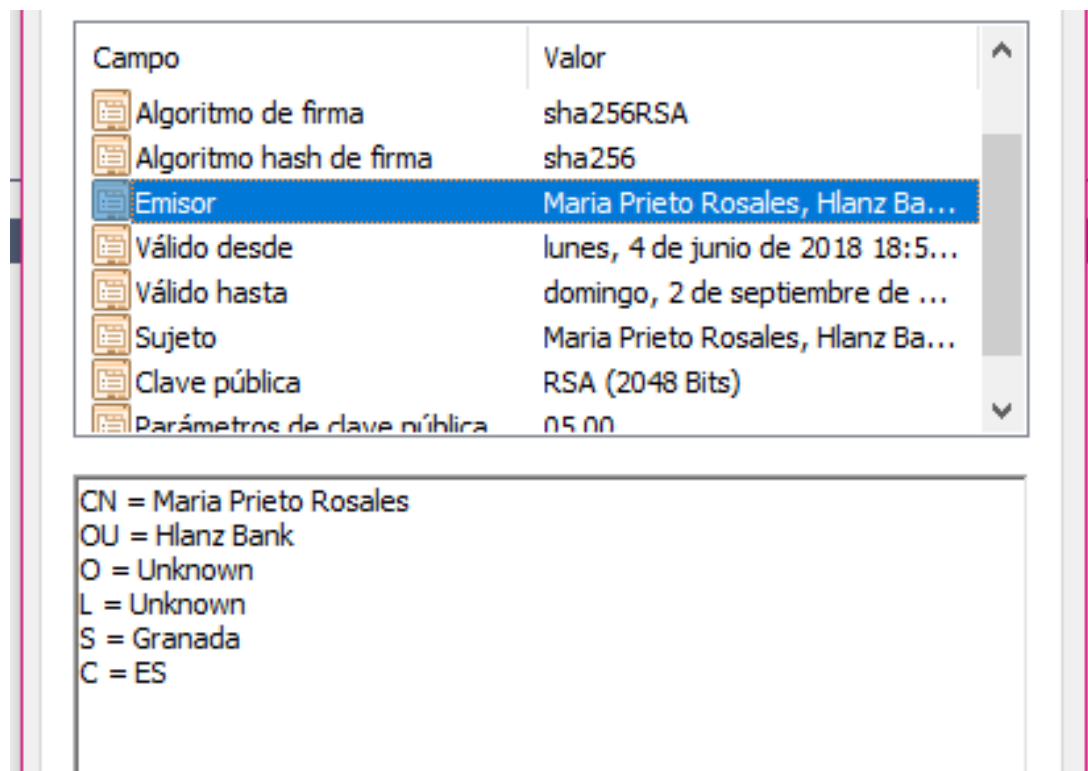
Para generar el certificado SSL debemos de ejecutar el siguiente comando:

```
"%JAVA_HOME%\bin\keytool" -genkey -alias tomcat -keyalg RSA  
-keystore \path\keystore
```

En la opción **-keystore** escribiremos la ruta del directorio donde deseamos crear el almacén de keys. Se solicitará una contraseña que posteriormente vamos usar para configurar el **Connector**.

En el fichero **server.xml** configuramos el puerto SSL de Tomcat que, por defecto, es el 8443.

```
<Connector  
    protocol="org.apache.coyote.http11.Http11NioProtocol"  
    port="8443" maxThreads="200"  
    scheme="https" secure="true" SSLEnabled="true"  
    keystoreFile="C:\Users\Maria\Documents\Proyecto\keys"  
    keystorePass="h14nzB4nk"  
    clientAuth="false" sslProtocol="TLS"/>
```



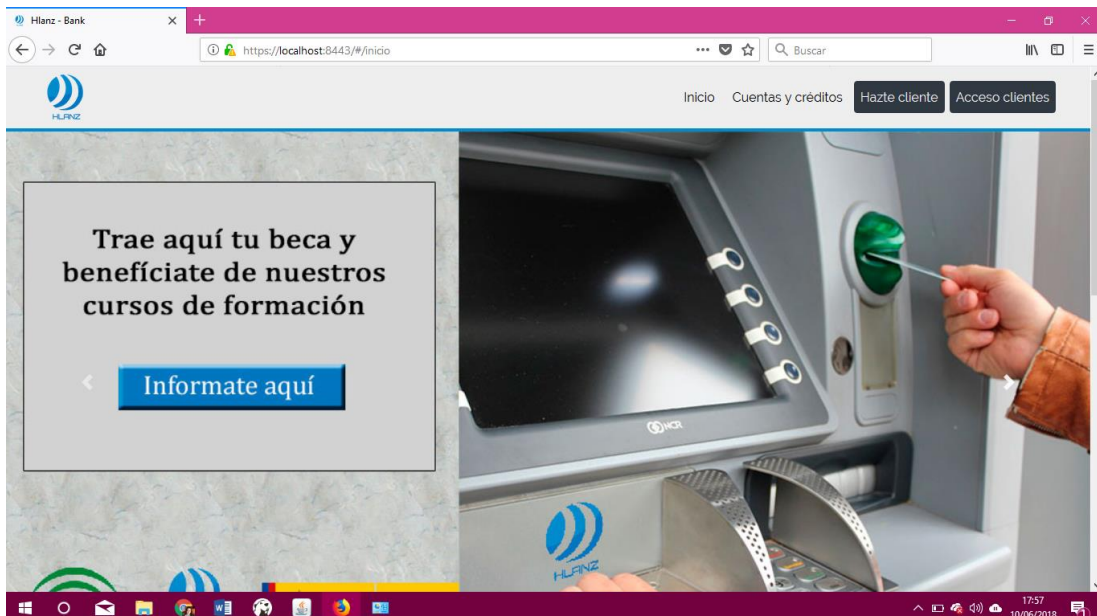
Cross-browser

La aplicación se comporta igual en cada navegador excluyendo los comportamientos añadidos de cada uno de ellos. Por ejemplo, Microsoft Edge añade a los input de tipo password la opción de poder visualizar la contraseña, las diferentes formas de validar los formularios, etc.

Chrome



Firefox



Microsoft Edge



Responsive

Para hacer responsiva la aplicación se ha recurrido tanto a Bootstrap 4, como FlexLayout de Angular.

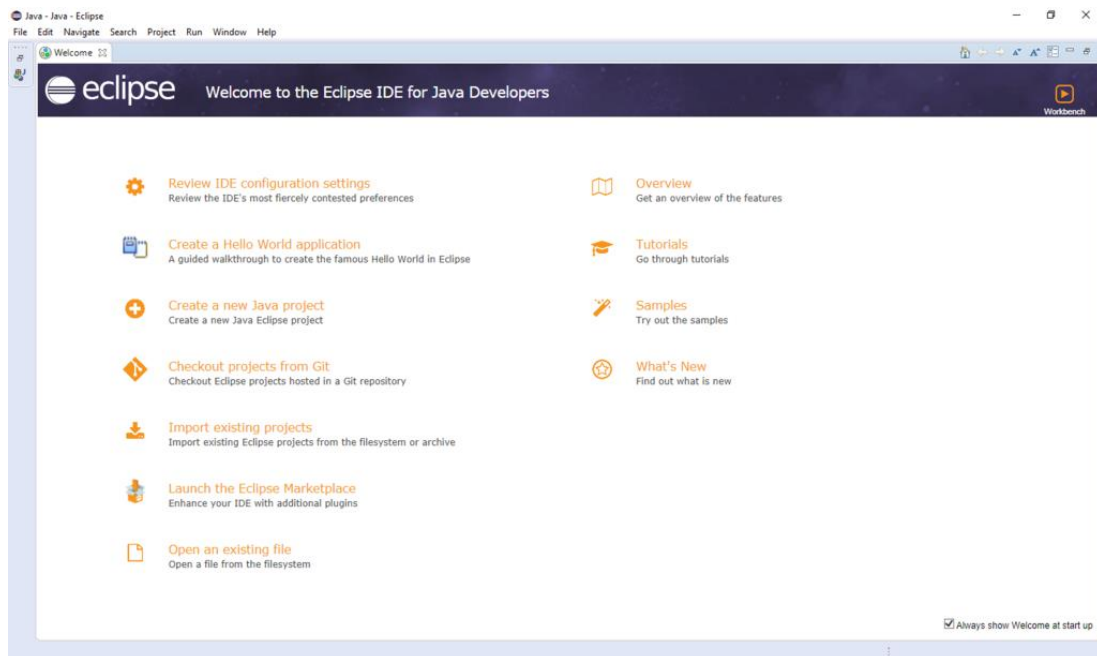
A screenshot of the Hlanz Bank "Hazte Cliente" (Become a Client) form, displayed on a mobile device (labeled "Iphone 5" at the bottom). The form is titled "Hazte Cliente" and includes input fields for "Nombre", "Apellidos", "DNI", "Teléfono", and "Correo electrónico". At the bottom, there is a checkbox labeled "Acepto los términos y condiciones". The HLANZ logo and a hamburger menu icon are visible in the top header. The browser's address bar shows "Hlanz - Bank".

Resultados y análisis

Para la creación del sitio web se ha utilizado el software siguiente:

- [Eclipse](#), en el entorno servidor a la hora de realizar el proyecto Maven en Java.

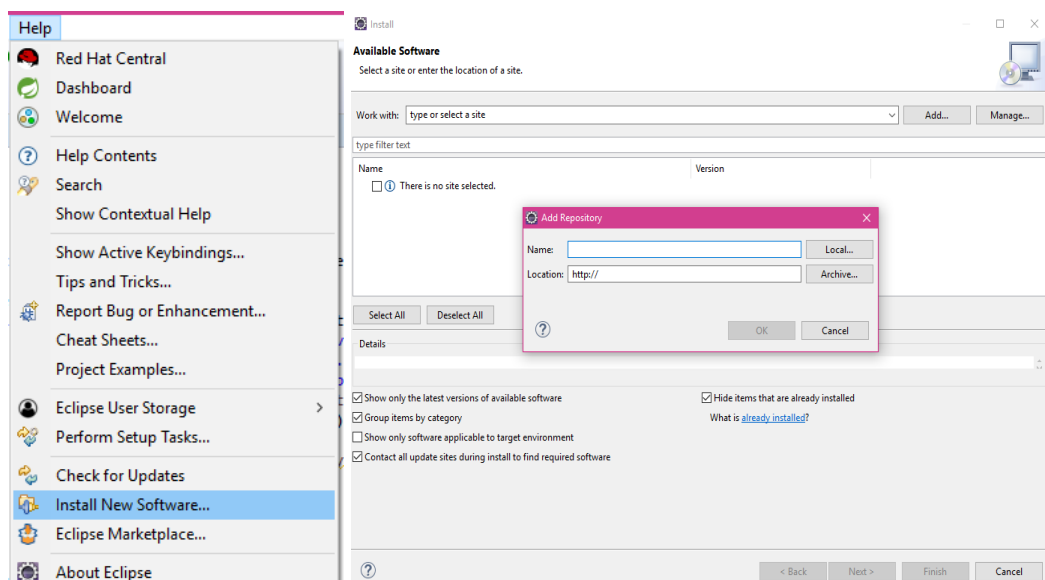
Para esto es necesario tener instalado [JDK 1.6](#) o superior.



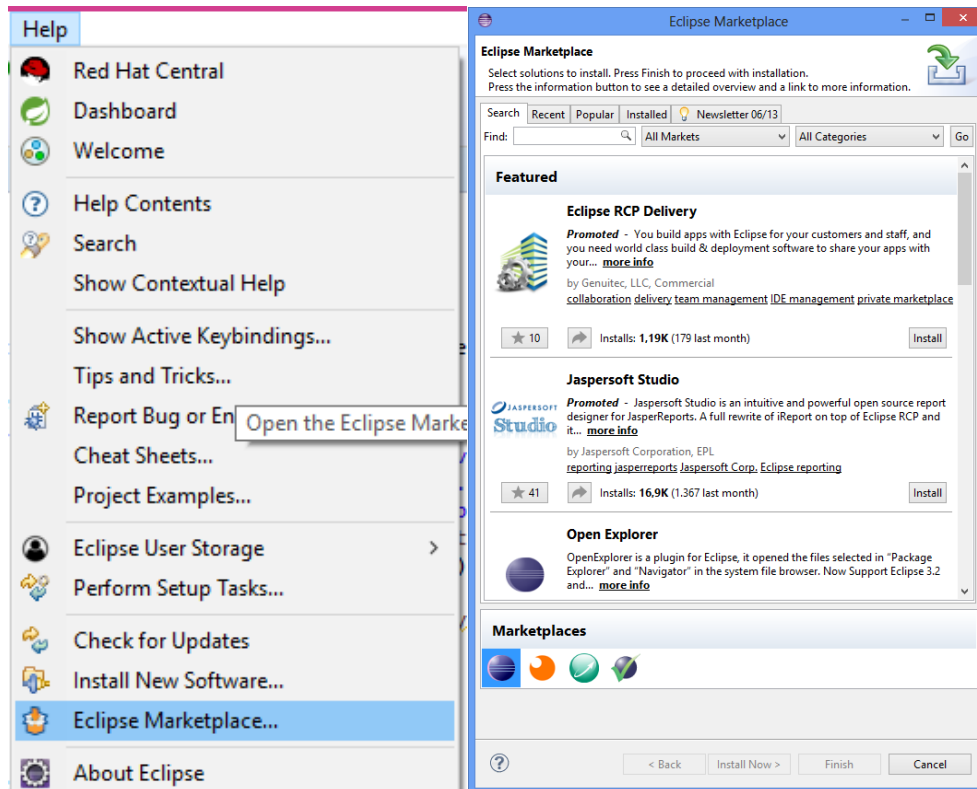
Instalaremos los plugins para Eclipse:

- [Hibernate](#). Este plugin nos permite hacer la ingeniería inversa para convertir las tablas en objetos Java.
- [Spring](#). (Gestión de transacciones, acceso a datos...)

Para instalar los plugins en Eclipse:

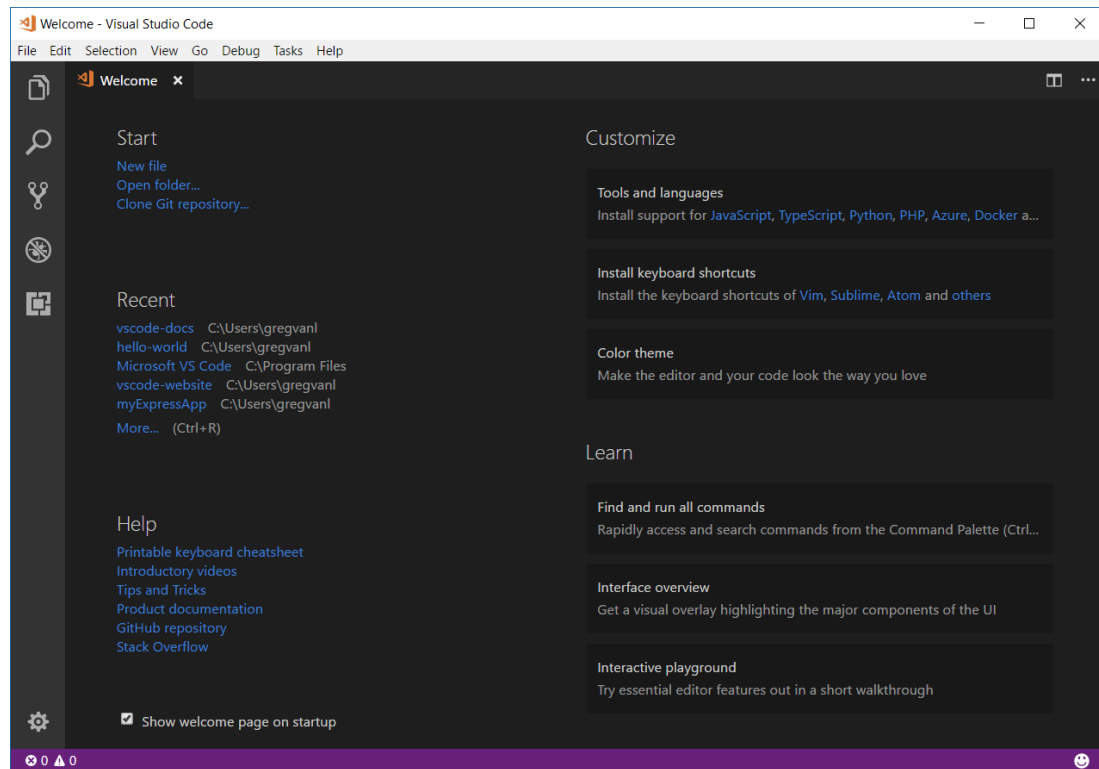


Eclipse ya incorpora un **Marketplace** en el también podemos encontrar los plugins anteriores.



Por otro lado tenemos la parte del cliente en la que hemos utilizado:

- [Visual Studio Code](#), para la edición de código, gestión de componentes y por su fácil acceso a un terminal que nos permite compilar, iniciar su propio servidor, etc.



- Antes de nada, necesitamos tener instalado [NodeJS](#) el cual, gestionará los paquetes que vamos a utilizar en Angular activando el comando ***npm*** y por consecuente, podemos utilizar el servidor de NodeJS para el desarrollo de Angular.
- Una vez hecho el paso anterior, procederemos a instalar [Angular 5](#).
npm install -g @angular/cli
- Iniciamos un nuevo proyecto (***ng new <name>***) y posteriormente instalaremos los componentes que consideremos necesarios. Por ejemplo, en nuestro caso [Angular Material](#):
npm install --save @angular/material

Conclusiones

El proyecto ha contribuido de manera importante para reforzar los conocimientos adquiridos durante el curso, ser capaz de detectar y cubrir una necesidad y saber implementar exitosamente las distintas tecnologías que se mencionan anteriormente.

Entre los puntos a destacar están las ventajas de dividir la aplicación en “dos proyectos”. Es decir, al contrario que en otros lenguajes como PHP, no mezclar el código de la parte del servidor con etiquetas HTML ya que si en un futuro deseáramos modificar solamente la interfaz o incluso reutilizar los servicios web para otra aplicación, podríamos hacerlo sin necesidad que cambiar nada.

Otra de las ventajas es que el entorno cliente al ser *Single Page Application (SPA)* es mucho menos pesada ya que consta de una sola página (index.html) y se reducen las peticiones al servidor, haciendo más rápida la navegación.

Entre las desventajas encontramos que, primero, el software utilizado es lento y se requiere un equipo más potente. Y, segundo, cada uno de los entornos utiliza un servidor para el desarrollo que hace más complicadas las peticiones entre ambos por el conocido *Cross-origin resource sharing (CORS)*

Líneas de investigación futuras

Se pretende mejorar la autenticación del cliente, ya que esta está hecha a través del body de la petición http y no utilizando la cabecera **Authorization** e incluso generar un token. En relación a esto, el id del usuario es autoincremental y una vez logueado se almacena en una variable de sesión en el navegador que podría ser modificada por cualquier usuario que acceda al navegador. Por lo que se plantea modificar este por un UID, más seguro y menos predecible, para evitar posibles fraudes.

Mejorar el patrón a la hora de validar el DNI en el registro o modificación de un usuario. Ya que este solo controla que se introduzcan 8 números y una letra, pero no que la letra coincida con el número del DNI siguiendo el algoritmo correspondiente.

Bibliografía

- [Tutorial ingeniería inversa con Hibernate](#)
- [Repositorio Maven \(dependencias \)](#)
- [Tutorial Angular](#)
- [Tutorial Angular Material](#)
- [Wikimedia Commons \(imágenes sin derechos\)](#)
- [Tutorial Instalación Tomcat](#)
- [Certificado SSL Tomcat](#)