Escola Universitària
Politècnica de Mataró

Centre adscrit a:

UNIVERSITAT POLITÈCNICA
DE CATALUNYA

Grado en Ingeniería Informática

APICONSOLA: Control y gestión de equipos de telecomunicaciones

Memoria

JONATAN NIETO AGIS

PONENTE: LÉONARD JANER GARCÍA

INVIERNO 2016

TecnoCampus
Mataró-Maresme

Dedicatoria

Dedico este proyecto a mi familia por haber confiado en mí y por su apoyo económico.
También a mi novia, Claudia, por la ayuda en momentos difíciles durante estos años.

Agradecimientos

Agradezco a Dycec y Telefónica la oportunidad de formar parte de su equipo y darme la oportunidad de llevar el análisis y desarrollo de una aplicación en un entorno de producción real.

También mencionar a John Thompson, mi referencia en arquitectura de software y desarrollo en Spring, ya que con sus cursos he aprendido mucho y he sacado provecho para mi proyecto.

Por último agradecer a los docentes de Tecnocampus por su enseñanza, en especial a Léonard por tutorizar mi proyecto y su ayuda.

Resumen

El proyecto consiste en desarrollar una aplicación para controlar equipos de la red de transporte de Telefónica. La plataforma se desarrolla en Spring Framework, Thymeleaf, Javascript, HTML, CSS, PostgreSQL. Con esta aplicación los técnicos y otras aplicaciones de la compañía podrán acceder y gestionar los equipos que controlan la red de transporte de Telefónica.

Resum

El projecte tracta a desenvolupar una aplicació per controlar equips de la xarxa de transport de Telefónica. La plataforma es desenvolupa en Spring Framework, Thymeleaf, Javascript, HTML, CSS, PostgreSQL. Amb aquesta aplicació els tècnics i altres aplicacions de la companyia podran accedir i gestionar els equips que controlen la xarxa de transport de Telefónica.

Abstract

This project consist of developing an application to control Telefonica's transport network system. It is developed by using Spring Framework, Thymeleaf, Javascript, HTML, CSS, and PostgreSQL. The application will help other applications and technicians in the organisation have access and manage the system that control Telefonica's transport network.

Índice

Índice de figuras

Índice de tablas

Glosario de términos

Términos técnicos

API Application Programming Interface

CRUD Acrónimo de Crear, Restaurar, Actualizar y Eliminar elementos de una aplicación.

CSS Cascading Style Sheets

Confluence Aplicación para la documentación de la aplicación.

Framework Conjunto de librerías y clases que nos facilitan el trabajo para programar cosas comunes en aplicaciones (ejemplo: Seguridad de la aplicación).

GIT Control de código fuente.

HTML5 HyperText Markup Language 5

IntelliJ Entorno de desarrollo.

Java Lenguaje de programación (compilado).

Jira El software de seguimiento de incidencias flexible y escalable para equipos de software.

Maven Herramienta de gestión y construcción de proyectos. Mediante XML asignamos las dependencias de nuestro proyecto y construye los elementos.

PostgreSQL Base de datos SQL.

Python Lenguaje de programación (interpretado).

Red Hat Sistema operativo basado en Linux donde vamos a desplegar nuestro servicio.

Spring Framework Java. Basándose en ficheros xml o anotaciones el encargado de construir todos los objetos que la aplicación va a utilizar.

Stash Interfaz para un mejor control de código.

Thymeleaf Librería que permite definir un conjunto de plantillas para presentar la capa de Java a capa web. Se acopla muy bien para trabajar en la capa vista del MVC de aplicaciones web.

Tomcat Servidor de aplicaciones Java.

Términos CMIP

Alarma Aviso del estado del equipo.

Conexión Es necesaria para poder comunicarnos con un gestor y los recursos del mismo.

Gestor Distribuidos en zonas, controlan los recursos de un punto de la red CNT de Telefónica.

Equipo Controlan, a nivel físico la capa de transporte de Telefónica.

Emlim Cuelgan de los gestores y gestionan los equipos que tienen por debajo.

Radio enlace Conexión punto a punto vía radio.

Recurso Cualquier objeto de la CMIP se considera un recurso.

1. Objetivos

1.1. Propósito

Desarrollar una aplicación WEB para controlar los equipos de la capa de transporte de Telefónica. La plataforma puede servir el contenido en formato JSON como en una interfaz web, según como sea la petición. El proyecto ha sido encargado por el CNT (Control Nacional de Transporte) de Telefónica.

1.2. Finalidad

Conseguir una aplicación WEB para un mejor acceso y control de los equipos. Facilitará el acceso tanto para operarios de la compañía como otros automatismos que controlan la infraestructura. Actualmente el acceso a los equipos se realiza de manera individualizada por cada fabricante y en cambio API Consola hace homogéneo el servicio.

1.3. Objeto

Una vez finalizado el proyecto se dispondrá de una aplicación REST en la red interna, con acceso restringido a aquellos usuarios y automatismos que tengan responsabilidades sobre el control que se quiere llevar.

1.4. Abastecimiento

La aplicación se desarrolla principalmente con Spring Framework, ya que permite operar directamente en la lógica de negocio y poniendo a disposición del programador aquellas tareas comunes que se han de programar en todas las aplicaciones.

2. Estudio previo

2.1. Aplicaciones actuales para el control de equipos

Se ha realizado un estudio de las aplicaciones que dispone la compañía actualmente y vamos a exponer las aplicaciones de cada cliente. (Por razones de confidencialidad no voy a nombrar qué fabricantes y de qué aplicaciones hablamos, así que llamaremos a estas Aplicación-X donde X será un número distintivo entre cada aplicación)

2.1.1. Aplicación-1

Controla una serie de equipos, dando el detalle de los mismos mediante una interfaz gráfica. El servicio no es homogéneo ya que sólo da la información de los equipos del fabricante en concreto. La aplicación debido a las conexiones que hay que establecer, el usuario debe navegar por un árbol sin poder atacar al equipo directamente.

2.1.2. Aplicación-2

Controla los equipos del fabricante, vía conexiones por consola. El servicio no es homogéneo ya que sólo da la información de los equipos del fabricante en concreto. Es difícil de manejar por los operadores de Telefónica y requiere mucha formación.

2.1.3. Consola

Consola a la cual vamos a atacar para obtener la información para nuestra aplicación. Esta es compleja, ya que para obtener información hay que hacer una serie de peticiones vía línea de comandos y mediante CURL.

2.2. Análisis tecnológico

2.2.1. Introducción

Antes de comenzar el proyecto, estaba en marcha otro proyecto realizado en Django Rest framework, pero después de supervisar el código y estudiar los requerimientos de la aplicación nos reunimos y propuse el cambio para realizar el proyecto de nuevo en Spring, porque con el actual framework había problemas que no nos permite resolverlos de manera tan eficaz. El realizar el estudio y decidir si lo que hay actualmente lo cambiamos o no es una decisión importante ya que está en juego el tiempo y la futura evolución de la aplicación.

En este apartado se muestran las utilidades de los dos *frameworks*, el que se estaba utilizando anteriormente y el actual. Se explica por qué se ha escogido Spring Framework.

2.2.2. Estudio frameworks

2.2.2.1 Django Rest Framework

Está basado en ***Python***, y esto hace que la programación sea fácil y práctica, además si en el proyecto participan otros programadores es un lenguaje que es de rápido aprendizaje. Permite rápidamente realizar aplicaciones web. Al instalar el framework nos da las siguientes características ya implementadas y codificadas:

Aplicación web navegable: Crea automáticamente una aplicación web navegable y a medida que implementas lógica de negocio el te muestra los resultados en el navegador de forma fácil.

Políticas de autenticación: Implementa automáticamente los paquetes de autenticación OAuth1 y OAuth2, es decir que controla los usuarios y permisos asignados a cada usuario.

Serialización: Convierte a objetos datos almacenados en bases de datos. Hace doble función, serializar y deserializar.

Implementa otras funciones y tiene una extensa documentación.

Antes de utilizar el framework, debemos tener en cuenta que hay una serie de requerimientos y preparación del entorno antes de poder programar. Enumeramos lo que nos hace falta:

S.O: sistema operativo basado en Linux, puede funcionar sobre otras plataformas, pero actualmente el 80% de aplicaciones se implementan bajo estos sistemas operativos.

Python: En la nueva versión de Django hay que tener instalada la versión 3.3.3.

Base de datos: Hay que configurar la máquina con un motor de BBDD y preparar la base de datos para ser usada por Django. Podemos usar SQLite, MySQL, PostgreSQL u Oracle.

Django: para usar Django Rest Framework antes debemos instalar Django.

Entorno virtual: en algunas ocasiones y según requerimientos debemos instalar un entorno virtual para no entrar en conflicto con otros servicios que estén corriendo en esa máquina. (*Personalmente, no me gusta tener servicios de esta manera y es un punto crítico en cuanto a la decisión de usar un software u otro)

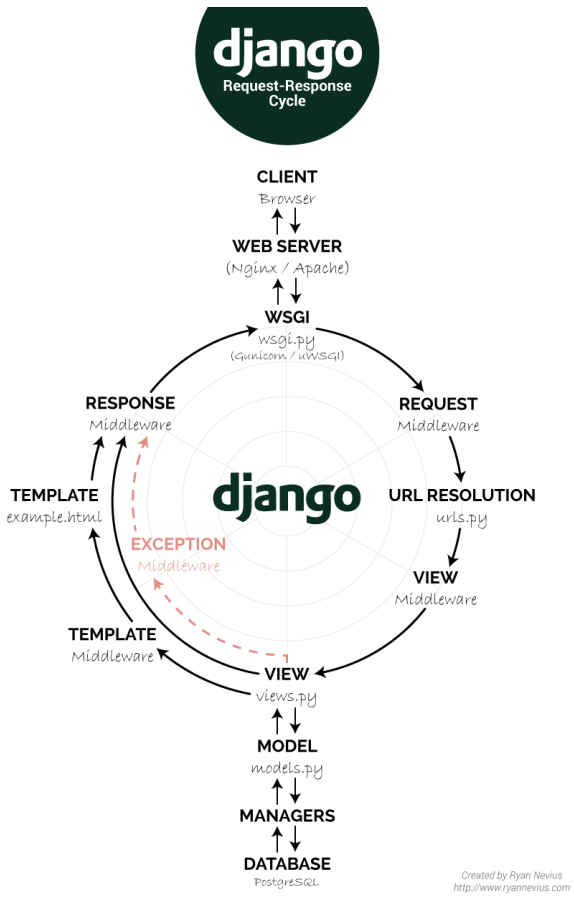

Fig. 1 Ciclo petición-respuesta Django

Después de analizar el framework y probar todo lo que había hecho hasta ahora, la conclusión es que Django REST Framework está orientado a realizar un servicio **REST** sobre una base de datos, donde la gestión principal son objetos y persistencia de los mismos, pero cuando queremos hacer operaciones que no estén dentro del **CRUD**, el framework a diferencia de otros no gestiona tan fácilmente la modificación de la aplicación para implementar una lógica de negocio que se base en los requisitos de la aplicación que nos piden. Lo que pasaba hasta el momento es que por falta de recursos se habían implementado cosas fuera de la capa que proporciona el framework y esto hacía difícil la aplicación y su futuro mantenimiento.

2.2.2.2 Spring Framework (Spring Boot)

Basado en Java, la programación es más estricta y el tiempo de aprendizaje es más elevado. Es más difícil de lanzar un proyecto, debido a que debemos configurar el framework según los requerimientos que tenemos marcados, pero una vez configurado las herramientas que nos proporciona son potentes y nos permiten centrarnos mucho más en la lógica de negocio de nuestra aplicación. Al usar Spring Boot lo que conseguimos es que, además de usar todas las herramientas de Spring Framework, creamos un proyecto autónomo y listo para entrar en producción y lista para correr. Las características principales de Spring Boot son:

- Crear proyectos autónomos basados en Spring.
- Tomcat incrustado, quiere decir que directamente corre sobre el proyecto, no necesitamos empaquetar y desplegar ficheros WAR en Tomcat.
- Proporciona configuraciones '*Starter'* para facilitar la configuración previa de la aplicación (se requiere conocimiento previo de Spring Framework para saber lo que está realizando). La configuración la realiza mediante Maven y su archivo de configuración POM, donde se definen las dependencias del proyecto que van a ser ejecutadas.
- Cuando es posible, aplica automáticamente la configuración de Spring.
- No requiere ningún tipo de generación de código ni configuración XML para arrancar la aplicación, siempre y cuando se usen los valores por defecto.

El salto a Spring Boot ha sido un avance considerable para todos aquellos que ya programaban en Spring Framework, ya que han puesto a disposición un entorno donde crear nuevos proyectos es más fácil, pero requiere un aprendizaje previo bastante elevado.

Una de las grandes ventajas de utilizar Spring Framework es que en la máquina no hay que tener instalado ningún servicio para poder desarrollar ni nada complejo, simplemente debes tener:

- **JDK** (Java ***Development*** Kit)
- **Entorno de desarrollo** (Usado: **Intellij Community Edition**)

2.2.3. Comparativa de los diferentes frameworks

A continuación, y como resumen, haré la comparativa de los 2 *frameworks*, el que se estaba utilizando inicialmente (**Django**) y el actual (**Spring**).

Django aparenta implementar el patrón **MVC**, pero el controlador es llamado vista y la plantilla.

Primero, debemos aclarar que al momento de diseñar Django, no buscaron apegarse a nada en particular, sino desarrollar una herramienta que funcione lo mejor posible.

Si bien es cierto que se asemeja mucho a la implementación del patrón MVC, en Django la Vista describe "qué" datos serán presentados y no "cómo" se verán los mismos. Aquí es donde entran en juego los ***templates***, que describen "cómo los datos son presentados".

Se dice que el "***controller***" de un MVC clásico está representado por el propio framework. Es decir, el sistema que envía una petición a la vista correspondiente, de acuerdo a la configuración de URL de Django (archivo de configuración).

Entonces diríamos que éste es un framework "MTV": modelo, ***template***, vista.

EN CONCLUSIÓN, nosotros no teníamos tanta accesibilidad sobre los controladores de Django y poder programar el funcionamiento de los mismos, como en un sistema MVC tradicional.

En cambio con Spring, además de muchas otras opciones, el patrón MVC se puede implementar y tener control absoluto sobre los modelos, vistas y controladores. Pero el gran potencial del framework no se encuentra aquí, si no lo encontramos en la Inyección de dependencias y su control unidad de control de objetos (***IoC***), donde los veremos explicados más adelante, pero en resumen la inyección es un patrón de diseño donde objetos dependientes se inyectan dentro de tus clases y la ***IoC*** administra los componentes del software y la inyección de los mismos en Objetos dependientes.

Spring es usado en el 50% de las aplicaciones web y va creciendo. Para combatir la complejidad de desarrollo, sigue las siguientes filosofías.

- Desarrollo simplificado mediante **objetos Java** y **POJOs**.
- Poco acoplamiento entre componente mediante el uso de Inyección de dependencias y programación para interfaces.

- Enfoque de programación declarativa usando costumbres y aspectos convencionales.
- Reducción de código repetitivo haciendo uso de plantillas y aspectos.

3. Spring Framework

A continuación voy a explicar los conceptos principales y funcionamiento del framework, es un punto muy importante, ya que quiero remarcar que en las ingenierías no se profundiza en el tema y es un concepto importante. Cada framework es un mundo, pero el propósito de todos es el mismo, combatir la complejidad de desarrollo.

3.1. Spring Core

3.1.1 Beans

Objetos java simples, centro de **Spring *Core*** y la filosofía general de Spring. Spring está diseñado para utilizar desde el núcleo **POJO's** (*Plain Old Java Objects*) or 'Spring Beans'. Como desarrollador se desaconseja crear objetos muy complejos.

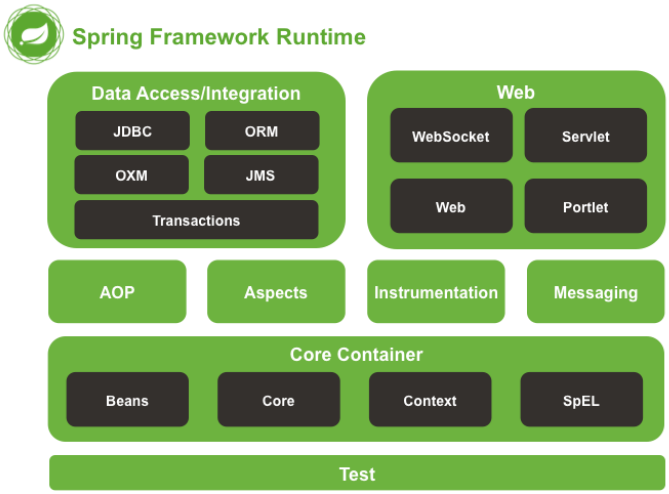

Fig. 2 Estructura de Spring Framework

3.1.2 Inyección de dependencias (DI)

Fortaleza principal de Spring Framework. Patrón de diseño donde objetos dependientes se inyectan dentro de tus clases.

Un claro **ejemplo** es el de un controlador web que realiza el post de un formulario, siguiendo el principio de responsabilidad individual no se quiere que la clase controlador interactúe con la BBDD. En su lugar tendrías algún tipo de clase servicio que se encarga de ello. El controlador haría el trato del formulario y luego llamaría a un método de la clase inyectada. El controlador no tiene que preocuparse, por la conexión a BBDD, agrupación de conexiones o qué tablas necesita actualizar. Del mismo modo el servicio no necesita saber nada acerca de la solicitud web. El servicio no se preocupa si los datos llegan en forma de formulario o JSON.

3.1.3 Inversión de Control (IoC)

El concepto de núcleo de Spring Framework es *IoC*. Para los desarrolladores, es la verdadera fuerza de Spring. A través de este concepto Spring administra los componentes del software y la inyección de los mismos en Objetos dependientes. La inyección de dependencias es tal y como suena: las dependencias se inyectan en tu clase. *IoC* es la inyección real de las dependencias.

3.1.4 Inyección de dependencias en IoC

Para los desarrolladores proporciona dos potentes conceptos, puedes construir objetos ligeramente acoplados. Cada uno tendrá un propósito muy específico.

Por ejemplo , un patrón de diseño muy común en la construcción de aplicaciones web es utilizar un controlador para interactuar con la capa web, e inyectar un objeto de servicio para interactuar con el nivel de base de datos. Como he dicho anteriormente, el controlador no necesita saber nada acerca de la base de datos , y el objeto servicio no necesita saber nada acerca de la capa web. Si se necesitan hacer cambios y mantenimiento de la clase de BBDD no se requieren cambios en el controlador.

3.1.5 Contexto Spring

Es lo que contiene todos los conceptos. En el contexto se cargan los ***Beans*** y la configuración del entorno. Asegura la inyección de las dependencias en las clases especificadas.

3.1.6 SpEL - Spring lenguaje expresivo

Lenguaje conciso utilizado para dar una gran flexibilidad al configurar Spring Framework.

3.2 Inyección de dependencias

Es una característica principal del *Core* de Spring Framework, En tiempo de ejecución Spring determina las dependencias para inyectar en tus clases (IoC).

3.2.1 Código sin vs con inyección de dependencias

Voy a comentar las ventajas de la inyección de dependencias y las desventajas de no usarlas. No hay inconveniente alguno, solo que su uso es más complejo y complica el código de nuestra aplicación.

El código **sin inyección** de dependencias:

- Los objetos asumen la responsabilidad de la gestión de sus propias dependencias.
- Generalmente conduce a un código más acoplado (Acoplamiento: Una unidad de software depende de otra para realizar una función.)
- Requiere cambios de código para gestionar los cambios con las dependencias.
- Las clases de test son más complejas.

El código **con inyección** de dependencias presenta las siguientes ventajas:

- Conduce minimizar el código acoplado.
- La clase no es responsable de determinar sus dependencias.
- Permite componer la clase en tiempo de ejecución.

3.2.2 Tipos de inyección de dependencias

- Basadas en el constructor, preferiblemente para aquellas clases que no puedan ser instanciadas sin estas dependencias.
- ***Setter Based***, preferibles en aplicaciones de Spring, un poco más flexibles que las basadas en constructor.

3.2.3 Interfaces basada en la inyección de dependencias

Se considera una **buena práctica** codificar dependencias en una interfaz.

- Esta práctica está relacionada estrechamente con los SÓLIDOS principios de la Programación Orientada a Objetos.
- Permite flexibilidad en la composición del comportamiento de la clase. (**Ejemplo**: Se pueden inyectar diferentes dependencias para soportar diversos comportamientos, y los ***Mock*** pueden ser usados para facilitar el testeo).

3.3 Entornos

Cuando comienzas a programar una aplicación empresarial, se necesitan múltiples entornos de despliegue. No sólo es testear el código en el entorno local y luego desplegarlo en el servidor de producción. Frecuentemente, en las empresas, los desarrolladores no tienen acceso a los entornos de producción. Las empresas que quieren cumplir las

regulaciones SOX, PCI o SAS-70 deben tener equipos especializados en desplegar el código para su testeo y una vez comprobado desplegarlo en el entorno de producción.

3.3.1 Desarrollo local

Este es el entorno de desarrollo del programador, corriendo en su máquina. Debería ser capaz de correr en local sin necesidad de conectarse a otros servidores. Se deberían cumplir las siguientes condiciones:

- No usar BBDD externas.
- No interactuar con otros servicios web.
- No JMS
- Los artefactos necesitados deberían estar en local (*hello Maven, Ivy!!*)

3.3.2 Integración continua

La **integración continua** es un modelo informático propuesto inicialmente por Martin Fowler que consiste en hacer **integraciones automáticas** de un proyecto lo más a menudo posible para así poder detectar fallos cuanto antes. Entendemos por integración la compilación y ejecución de pruebas de todo un proyecto.

El proceso suele ser cada cierto tiempo (horas), descargarse las fuentes desde el control de versiones (por ejemplo **CVS, Git, Subversion, Mercurial**) compilarlo, ejecutar pruebas y generar informes.

Para esto se utilizan aplicaciones como **Bamboo**(para proyectos **Java**), que se encargan de controlar las ejecuciones, apoyadas en otras herramientas como Ant o **Maven** (también para proyectos Java), que se encargan de realizar las compilaciones, ejecutar las pruebas y realizar los informes.

A menudo la integración continua está asociada con las metodologías de programación extrema y desarrollo ágil.

Los servidores de integración continua pueden llegar a ser pequeñas bestias difíciles de configurar. En cada uno de los diferentes escenarios es probable que se necesite configurar específicamente la aplicación.

3.3.3 Desarrollo

Algunas empresas tienen entornos de desarrollo. Tiene las mismas características que el servidor de desarrollo, pero a diferencia el equipo tiene control sobre él. Cuando se despliega sobre este entorno, tienes que configurar la aplicación para interactuar con servidores dentro de ese entorno.

3.3.4 QA / UAT

Entorno de testeo del software para su aceptación. Hay un equipo específico de ingenieros que trabajan en este entorno y todos los cambios de software pasan para ser verificados.
Por **ejemplo** en un caso de pruebas de un método de una aplicación primero se definen 4 preguntas a las cuales deberíamos dar respuesta:

1. ¿Qué se va a probar?
2. ¿Desde qué perspectiva se va a probar?
3. ¿A qué nivel se va a probar?
4. ¿Con qué técnicas voy a probar?

Responder las anteriores preguntas da una visión más clara del objetivo de las pruebas, para saber cuando se ha terminado. Una vez testeada y respondidas estas preguntas podemos hacer la valoración y dar su aceptación, o de lo contrarío marcar las revisiones que hagan falta.

3.3.5 Pre-Producción o *Stage*

Es un entorno que funciona con los servicios de producción y bases de datos de apoyo. Es un entorno que permite el despliegue del código, pero limita el acceso a ese código.
Las ventajas de trabajar con este entorno son:

- Podemos probar la aplicación en un entorno igual al real, así asegurando el funcionamiento del código cuando lo despleguemos en producción.
- Podemos probar la reacción de los servicios de producción en base a nuestro código.

3.3.6 Producción

Es el entorno que van a utilizar los usuarios finales.

3.3.7 Soporte de Spring Framework al Multi-Entorno

El framework ha sido desarrollado para dar soporte a los diferentes entornos de desarrollo. Tiene diversas características para dar soporte a aplicaciones de clase empresarial.

3.3.7.1 Propiedades

Tiene un excelente soporte de propiedades. Las propiedades son simples *Strings* externalizados a tu aplicación. Estas propiedades se pueden establecer de las siguientes maneras.

- En un **fichero de propiedades**: <nombre_del_fichero>.properties
- Usando **variables de entorno** del sistema operativo
- **Variables de línea de comando**, al ejecutar la aplicación tenemos la opción de pasar comandos con las variables con las que queremos que arranque la aplicación.

3.3.7.2 Inyección de dependencias

Las aplicaciones de clase empresarial tendrán cambios de comportamiento que son más complejos que una simple propiedad. Una de las características principales de Spring es el apoyo de inyección de dependencias. El código se vuelve más modular al realizar inyección de dependencias y las clases cumplirán con el principio de responsabilidad individual. Al usarlas se hace muy fácil intercambiar componentes.

3.3.7.3 Composición de configuración

Un error al comenzar con Spring es colocar la configuración de la aplicación en un fichero XML. Un grave error debido a que limita las opciones de configuración. En los ejemplos todas las opciones de configuración podrán ser soportadas por la configuración de Spring. La configuración debe colocarse en un fichero independiente o un paquete de configuración. Tu propia configuración se utilizará en la configuración padre, que se carga en el contexto de Spring en tiempo de ejecución.

3.3.7.4 Perfiles

Los perfiles de Spring permiten definir los *Beans* que van a ser cargados en el contexto de Spring, es decir según el perfil de configuración cargaremos una opción u otra (Ejemplo: Una aplicación configurada con el Perfil inglés cargará los mensajes en inglés y configurada con el perfil en español cargará los mensajes en español.)

3.4 Detalle sobre Inyección de dependencias

A continuación vamos a detallar cómo se inyectan las dependencias. Primero veremos código que no usa inyección de dependencias, seguidamente veremos el mismo código refactorizado con inyección de dependencias. El objetivo es entender cómo construir las clases para permitir la inyección de dependencias.

3.4.1 Código sin inyección de dependencias

Cogemos el siguiente ejemplo, donde no vemos esta característica implementada.

```
public class MyController {
    private MyService service;
    public MyController() {
        service = new MyService();
    }
}
```

Fig. 3 Clase sin inyección de dependencias

Tenemos un controlador, donde la lógica de negocio se ha movido a un servicio. En este ejemplo el servicio se construye en el constructor de la clase controladora, pero podría estar dentro de un método. Si pensamos, vemos que esta clase está de forma permanente codificada como la clase ***MyService*** (***HARD CODED***). Esto es lo que se entiende como código estrechamente acoplado. No podríamos ejecutar esta clase sin una instancia anteriormente.

¿Qué pasa si queremos probar la clase? ¿Qué pasa si el servicio necesita conectarse a una base de datos? Bajo este escenario, el controlador no se puede ser testeado por ***Unit Test***. Una vez que esté por medio una conexión a base de datos, ya no tendremos una prueba unitaria. La inyección de dependencias soluciona el problema.

3.4.2 Usando la inyección de dependencias

Según lo que hemos visto anteriormente, es un patrón de diseño donde los objetos dependientes se inyectan en tu clase. Spring Framework gestiona la inyección por ti, pero debes preparar las clases para que acepten la inyección. Hay dos tipos de inyección, basadas en el constructor y basadas en ***Setters***. Las dos tienen ventajas y desventajas.

3.4.2.1 Basadas en constructor

Los puristas de la orientación a objetos están a favor y usan la inyección basada en constructor, la clase no puede ser instanciada sin proveer las dependencias necesarias.

```
public class MyController {
    private MyService myService;
    public MyController(MyService myService) {
        this.myService = myService;
    }
    public Object controllerMethod(){
        return new Object();
    }
}
```

Fig. 4 Inyección de dependencias en constructor

3.4.2.2 Basadas en *Setters*

Simplifican el código y ofrecen una mayor flexibilidad a diferencia de las anteriores.

```
public class MyController {
    private MyService myService;
    public void setMyService(MyService myService) {
        this.myService = myService;
    }
    public Object controllerMethod(){
        return new Object();
    }
}
```

Fig. 5 Inyección de dependencias basada en Setter

3.4.2.3 ¿Cual usamos?

Encontramos que es un área que ha sido muy debatida por los ingenieros de software. La mayoría de desarrolladores de Spring usan Setter DI. No crear una instancia sin las dependencias adecuadas es un argumento muy válido. Sin embargo, hablando de escribir código para trabajar con Spring, el framework gestiona las dependencias por ti. No debemos preocuparnos acerca si la clase se utiliza sin las dependencias. Si Spring por cualquier razón no localiza la dependencia, nos dará un error al iniciar la aplicación y se cerrará.

```
private GestorService gestorService;
private EmlimService emlimService;

@Autowired
public void setGestorService(GestorService gestorService) {
    this.gestorService = gestorService;
}

@Autowired
public void setEmlimService(EmlimService emlimService) {
    this.emlimService = emlimService;
}
```

Fig. 6 Ejemplo inyección de dependencias en Spring

3.4.2.4 Inyección de dependencias basada en interfaz

Al instanciar un objeto concreto, sea en el constructor o en el setter, usaremos una instancia del tipo interfaz. Este enfoque es muy recomendado. Nos proporciona una gran flexibilidad en el código. Cualquier objeto que implemente la interfaz específica puede ser inyectado en tu clase. Hace más fácil el testeo de la unidad.

Creamos la interfaz:

```
public interface MyService {
    Object someMethod();
}
```

Fig. 7 Creación interfaz

Implementamos el servicio en nuestra clase:

```
public class MyServiceImpl implements MyService {
    @Override public Object someMethod() {
        return new Object();
    }
}
```

Fig. 8 Implementación interfaz de un servicio

Nuestro controlador en el ***Setter*** está usando la interfaz, no una clase concreta.

```
public class MyController {
    private MyService myService;
    public void setMyService(MyService myService) {
        this.myService = myService;
    }
    public Object controllerMethod(){
        return new Object();
    }
}
```

Fig. 9 Inyección de dependencia en controlador

3.5 Spring *Boot*

Herramienta que nos permite crear un proyecto de Spring, rápidamente. Hay cuatro áreas clave:

- Configuración automática
- Dependencias para arrancar
- Interfaz Línea de comandos
- Actuador

Spring Boot es un **subproyecto** de **Spring**, pero busca facilitarnos la creación de proyectos de Spring Framework eliminando la necesidad de crear largos archivos de configuración xml, Spring *Boot* provee configuraciones por defecto para Spring y otra gran cantidad de librerías, además provee un modelo de programación parecido a las aplicaciones Java tradicionales que se inician en el método ***main***.

3.6 Inyección de dependencias usando Spring

Vamos a enseñar como usar en Spring la inyección de dependencias. Se pueden configurar de dos maneras, mediante fichero **XML** o basándose en **anotaciones**.

3.6.1 Spring Beans

Los *Beans* son el núcleo de nuestra aplicación, componentes que Spring administra por ti. Hay cuatro tipo de componentes:

1. *Component*
2. *Service*
3. *Controller*
4. *Repository*

@Component

Esta anotación se usa en la clase que se define como Spring *Bean*.

@Service

La lógica de negocio debe estar en la clases de Servicio. Una operación ofrecida como una interfaz que se encuentra en el modelo, sin un estado encapsulado. Funcionalmente, no hay ninguna diferencia entre usar la anotación servicio o componente. El propósito de usar una manera u otra es de definir la intención de tu clase.

@Repository

La interfaz repositorio se utiliza para indicar qué clase se utiliza para la persistencia. Es un mecanismo para encapsular el almacenamiento, recuperación y proceder a buscar en una colección de objetos emulada.

@Controller

Se utiliza para indicar qué clase se utiliza como controlador web. Muy usada en Spring MVC y es usada con otras anotaciones que definen el comportamiento del controlador.

3.6.2 Wiring Spring Beans

La mejor manera de conectar dependencias de Spring es mediante el uso de la anotación ***@Autowired***. Puede ser usada en **constructores**, **métodos** y **propiedades**. Mencionan mucho el debate entre usar inyección basada en constructor o setter, pero **los profesionales de Spring prefieren la inyección de dependencias en Setter**. Especialmente cuando tenemos múltiples propiedades para establecer vía inyección de dependencias.

4. Entorno de desarrollo

4.1 Plataforma de desarrollo

4.1.1 Centos 7

He escogido este sistema operativo debido a que en Telefónica trabajan con servidores Red Hat y veía conveniente usar una versión desktop que derive del mismo. No obstante las incorporaciones en el equipo podrán usar este SO u otro alternativo (ej. Windows, MAC OS X).

Con un sistema basado en **UNIX** podemos conectar a los servidores de la empresa sin necesidad de instalar ningún software para conectarse por SSH a un servidor o hacer traspasos de archivos mediante SCP a los servidores. En empresas puede ser una restricción el instalar software de terceros y por eso mejor optar por un SO que no se vea comprometido en esta norma.

4.1.2 IntelliJ IDEA

Software para la codificación del proyecto de la empresa **JetBrains**. Son referentes debido a su buena asistencia al programador. Voy a describir algunas de las ayudas que nos ofrece este programa:

- **Finalización inteligente**: analiza el contexto de programación y en todo momento sugiere las opciones de terminación y así poder codificar más rápido, debido a que nos ayuda a autocompletar la codificación.

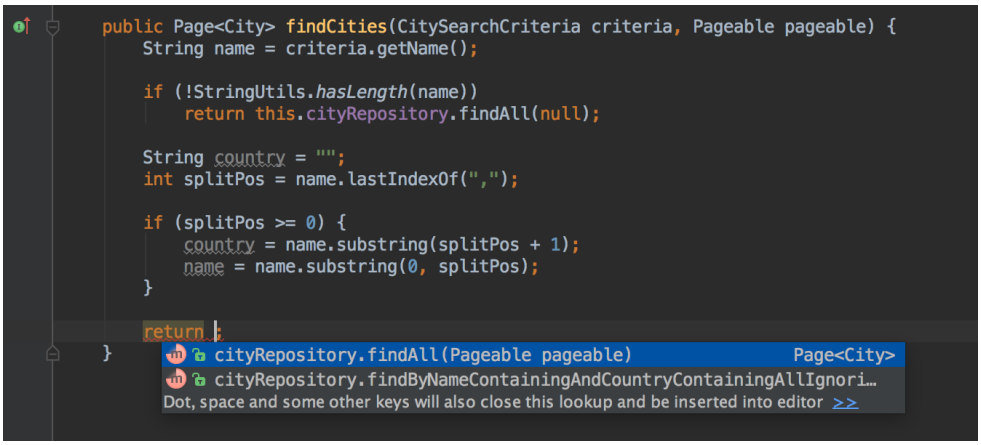

Fig. 10 Finalización inteligente de código IntelliJ IDEA

- **Navegación por el código**: nos facilita la navegación por el código de clase en clase, es decir que con un solo clic podemos ir a la clase que estamos haciendo referencia para hacer cambios o consultar información.
- **Refactorización segura**: al cambiar el nombre de cualquier clase, nos renombra la misma y se encarga de aplicarlo en todo el código donde mencionamos a la clase que ha cambiado.
- **Inspección de código**: Muy avanzado e integrado con Spring Framework, donde nos inspecciona el código en busca de errores y nos da la solución/es más adecuadas al fallo.

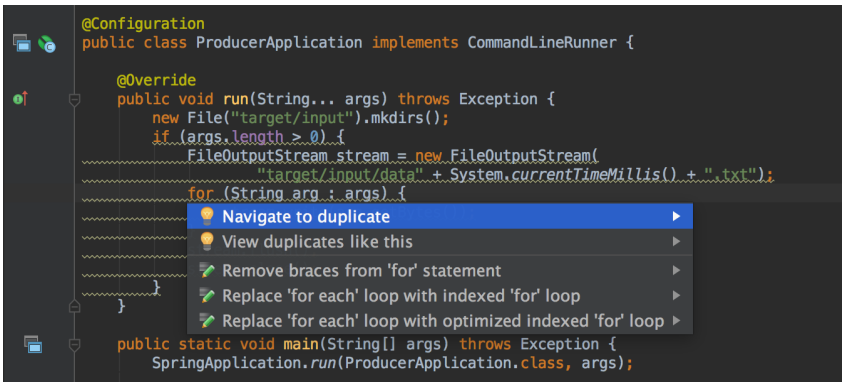

Fig. 11 Navegación por el código en IntelliJ IDEA

- **Herramientas e integraciones**: Esta es una de las características muy ventajosa frente a otros software para desarrollar(como Eclipse o Netbeans). Se integra fácilmente con herramientas como:

- Control de dependencias: **GRADLE**, **MAVEN**.
- Control de versiones: **Git**, Mercurial.
- Herramientas de bases de datos: Oracle, SQL Server, **PostgreSQL**, **MySQL**.
- **Testing**: interfaz única para realizar pruebas y mediciones correspondientes.

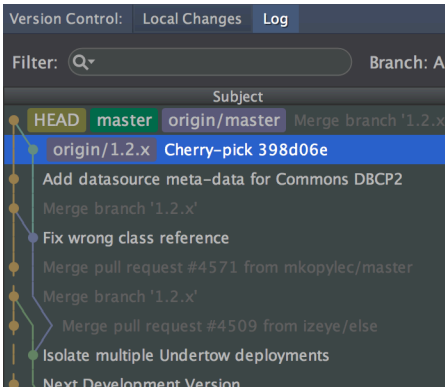

Fig. 12 Integración de herramientas de IntelliJ IDEA

- **Depurador avanzado:** cuando se usa el depurador de código en todo momento nos muestra las variables de entorno y en cada línea los valores de las variables afectadas (en su última actualización vemos que nos muestra los resultados hasta de las **LAMBDA** de Java 1.8).

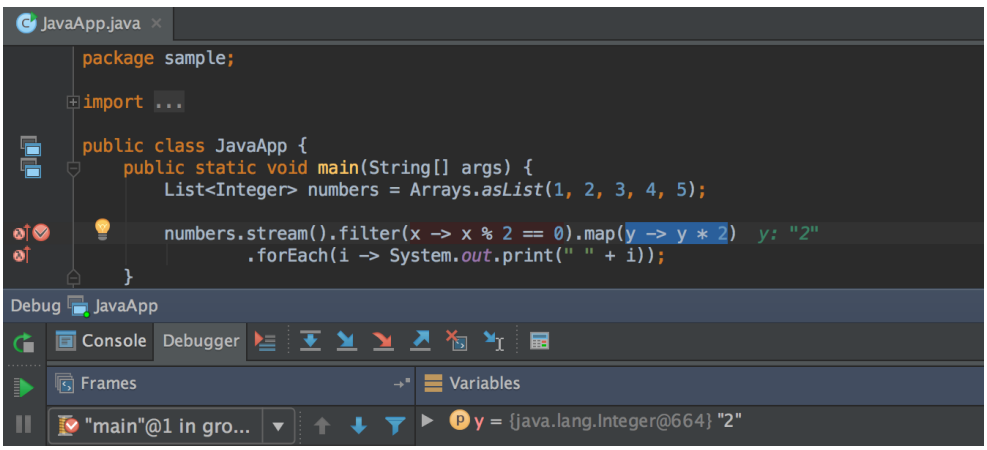

Fig. 13 Depurador IntelliJ IDEA

- **Integración con Spring Framework:** Asistencia en la codificación con Spring Framework y Spring *Boot*.

4.1.3 Java 1.8

Uso la versión **JDK 1.8** en el equipo de desarrollo y así mismo la programo con la versión 1.8 en el proyecto. Esta decisión es debido a que la versión ya es estable y que

posteriormente vemos el uso de **Lambdas** y funciones para hacer **paralelismo** en el proyecto, estas funciones han sido integradas en esta versión.

4.2 Tecnología de desarrollo escogida

4.2.1 Spring Boot

Permite crear proyectos independientes de manera rápida, basados en Spring y listos para ser puestos en producción. Incorpora librerías propias y de terceros que nos permiten empezar a trabajar en el proyecto con el mínimo esfuerzo. La mayoría de aplicaciones de Spring *Boot* necesitan muy poca configuración para arrancar. Se caracteriza por:

- Crear aplicaciones independientes.
- **Tomcat** incrustado, es decir que al arrancar el proyecto nos arranca un servidor automáticamente y así desplegar nuestra aplicación (Muy útil en entornos de desarrollo ya que el proyecto lo ponemos en marcha con un solo clic).
- Proporciona un **POM** por defecto para simplificar la configuración de MAVEN.
- Configura Spring automáticamente siempre que sea posible y no se necesite configuración específica.
- Proporciona las características para poner el proyecto en producción.
- No es necesario configurar ningún XML como en Spring.

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.1.RELEASE</version>
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Fig. 14 Ejemplo fichero Pom

```
package hello;

import org.springframework.boot.*;
import org.springframework.boot.autoconfigure.*;
import org.springframework.stereotype.*;
import org.springframework.web.bind.annotation.*;

@Controller
@EnableAutoConfiguration
public class SampleController {

    @RequestMapping("/")
    @ResponseBody
    String home() {
        return "Hello World!";
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(SampleController.class, args);
    }
}
```

Fig. 15 *Hello World* en Spring

4.2.2 Thymeleaf: generador vista HTML

Es una librería de Java, es un motor que genera HTML, XML, JavaScript, CSS y texto según la plantilla que estipulamos. Lo uso como capa de presentación para la aplicación web.

En nuestra aplicación después de aplicar la lógica de negocio que toca y al querer generar una vista de la información, creamos un **MAV** (*Model and View*) en Java donde se le asignan los objetos que queremos mostrar y como se llama la correspondiente vista que va a mostrar la información.

Thymeleaf, genera el código según la vista y los objetos que enviamos, se ejecuta en el servidor y es totalmente transparente al cliente final, ya que a este solo le llega código HTML.

Disponen de un módulo para **Spring MVC** y substituye por completo a **JSP**. La creación de plantillas es muy buena, ya que solo hay que etiquetar el código de aquellas vistas HTML donde van a ir los datos dinámicos.

Es software libre y se distribuye bajo la licencia de Apache 2.0 por lo que no tiene coste alguno.

Para que se entienda mejor voy a poner un fragmento de código del controlador, la correspondiente vista creada con Thymeleaf y el código HTML generado. En este procedimiento vemos la impresión de un **gestor** de Telefónica (Nodo de comunicación donde cuelgan los **Emlims** de Telefónica) y su listado de **Emlims**:

- **Controlador**: observamos en el controlador, que para generar una vista HTML declaramos un **ModelAndView** y a este, mediante sus correspondientes métodos le añadimos los objetos correspondiente y su vista (marcados en amarillo), los objetos se llaman de una manera dentro del **MAV** porque luego mediante **Thymeleaf** haremos referencia a ellos. Marcado en rojo es el servicio de nuestra aplicación el cual nos está proporcionando la información solicitada, más adelante veremos cómo se comporta esta capa.

```java
/**
 * Este método devuelve el gestor y sus emlims.
 * @param contentType Según este parámetro la respuesta será JSON o en formato WEB
 * @param gestor Variable en el path que determina el gestor del cual queremos recuperar la información
 * @return Devuelve un JSON o el template + la información del gestor y sus emlims.
 */
@RequestMapping(value="/gestor/{gestor}", method = RequestMethod.GET)
public @ResponseBody Object conectarGestorNombre(@RequestHeader(value="Content-Type", required=false,
                                                  defaultValue="text/html") String contentType,
                                                 @PathVariable String gestor) throws Exception {
    if(contentType.equals("application/json")){
        return gestorService.getGestorByName(gestor);
    }else{
        ModelAndView mav = new ModelAndView();
        mav.addObject("gestor", gestorService.getGestorByName(gestor));
        mav.addObject("emlims", emlimService.getEmlimsGestor(gestor, gestorService.abrirConexionGestor(gestor)));
        mav.setViewName("gestor");
        return mav;
    }
}
```

Fig. 16 *ModelAndView* de Spring Framework

- **Thymeleaf** (Etiquetado de la vista): vemos los fragmentos de la vista HTML etiquetados para mostrar la información tanto del gestor como los Emlims que cuelgan de él. Thymeleaf genera la vista según los objetos añadidos al MAV:

  Por cada atributo del gestor que queremos mostrar con Thymeleaf dentro del HTML usamos el siguiente tipo de etiquetado: **th:text**=”${[**objeto**].[**atributo**]}”.

```html
<h3>Información gestor</h3>
<div class="table-responsive">
    <table class="table table-bordered table-striped">
        <tbody>
        <tr>
            <th>Address</th>
            <td th:text="${gestor.address}">-</td>
        </tr>
        <tr...>
        <tr...>
        <tr...>
        <tr>
            <th>MIB</th>
            <td th:text="${gestor.mib}">-</td>
        </tr>
        <tr...>
        <tr...>
        <tr...>
        <tr>
            <th>TSEL</th>
            <td th:text="${gestor.tsel}">-</td>
        </tr>
        </tbody>
    </table>
</div>
```

Fig. 17 Etiquetado Thymeleaf en HTML

A continuación vemos que para mostrar la lista de Emlims, con **Thymeleaf** hacemos un bucle mediante **th:each** (marcado en amarillo) y por cada iteración crearemos una fila con sus correspondientes columnas dentro de nuestra tabla HTML.

```
<div class="panel-body">
    <h3>Lista EMLIMS</h3>
    <div class="dataTable_wrapper">
        <table class="table table-striped table-bordered table-hover" id="dataTables-example">
            <thead>
                <tr>
                    <th>userLabel</th>
                    <th>object_instance</th>
                    <th>object_instance_raw</th>
                    <th>object_class</th>
                    <th>Acceso</th>
                </tr>
            </thead>
            <tbody>
                <tr class="odd gradeX" th:each="emlim : ${emlims}">
                    <td th:text="${emlim.attributes.get('userLabel')}">userLabel</td>
                    <td th:text="${emlim.object_instance}">object_instance</td>
                    <td th:text="${emlim.object_instance_raw}">object_instance_raw</td>
                    <td th:text="${emlim.object_class}">object_class</td>
                    <td><center><a th:href="@{/gestor/} + ${gestor.name} + @{/emlim} + ${emlim.object_instance_raw} + @{/}"
                                   class="btn btn-info btn-sm" role="button">Acceder</a></center></td>
                </tr>
            </tbody>
        </table>
    </div>
```

Fig. 18 Bucle en Thymeleaf

- Una vez procesado el código en **Thymeleaf**, el fichero que se genera es **HTML**, sin ningún índice de todo lo que está en la parte de servidor y totalmente trasparente al usuario.

```
<div class="panel panel-default">
    <div class="panel-body">
        <h3>Información gestor</h3>
        <div class="table-responsive">
            <table class="table table-bordered table-striped">
                <tbody>
                    <tr>
                        <th>Address</th>
                        <td>172.26.11.29</td>
                    </tr>
                    <tr>
                        <th>Aequalifier</th>
                        <td>4</td>
                    </tr>
                    <tr>
                        <th>Aptitle</th>
                        <td>{0 1 2 3}</td>
                    </tr>
                    <tr>
                        <th>Description</th>
                        <td></td>
                    </tr>
                    <tr>
                        <th>MIB</th>
                        <td>alu395</td>
                    </tr>
                    <tr>
                        <th>Protocol</th>
                        <td>cmip</td>
                    </tr>
                    <tr>
                        <th>PSEL</th>
                        <td>5050</td>
                    </tr>
                    <tr>
                        <th>SSEL</th>
                        <td>5345</td>
                    </tr>
                    <tr>
                        <th>TSEL</th>
                        <td>4152</td>
                    </tr>
```

Fig. 19 HTML generado con Thymeleaf

```
<div class="panel-body">
    <h3>Lista EMLIMS</h3>
    <div class="dataTable_wrapper">
        <table class="table table-striped table-bordered table-hover" id="dataTables-example">
            <thead>
                <tr>
                    <th>userLabel</th>
                    <th>object_instance</th>
                    <th>object_instance_raw</th>
                    <th>object_class</th>
                    <th>Acceso</th>
                </tr>
            </thead>
            <tbody>
                <tr class="odd gradeX">
                    <td>neGroup1461</td>
                    <td>/neGroupId=1461</td>
                    <td>/0.3.0.2.7.7=1461</td>
                    <td>neGroup</td>
                    <td>
                        <center>
                            <a class="btn btn-info btn-sm" role="button" href="/gestor/sh14/emlim/0.3.0.2.7.7=1461/">Acceder</a>
                        </center>
                    </td>
                </tr>
                <tr class="odd gradeX">
                    <td>neGroup1424</td>
                    <td>/neGroupId=1424</td>
                    <td>/0.3.0.2.7.7=1424</td>
                    <td>neGroup</td>
                    <td>
                        <center>
                            <a class="btn btn-info btn-sm" role="button" href="/gestor/sh14/emlim/0.3.0.2.7.7=1424/">Acceder</a>
                        </center>
                    </td>
                </tr>
```

Fig. 20 HTML generado con un bucle en Thymeleaf

4.2.3 SB Admin 2 (Bootstrap, Ajax, jQuery)

La compañía "*Iron Summit Media Strategies*" de Orlando es la creadora de startbootstrap.com, una web donde mediante Bootstrap crean *templates* tanto personalizados como bajo licencia apache para que la gente no tenga que diseñar los HTML de su aplicación web, que en ocasiones es un coste alto y no solo requiere programadores, sino también diseñadores.

He escogido un tema de un panel de control (**SB Admin 2**) bajo la licencia Apache 2.0 de *Start* **Bootstrap**. En la creación de este tema participan varias instituciones que menciono en cada componente. En este tema ya nos vienen definidos los siguientes componentes y estilos:

- Barra lateral con desplegables multi-nivel.
- Ficheros comprimidos incluidos.
- Menú responsivo con elementos desplegables.
- Tres estilos de paneles: rojo, amarillo y verde. Se puede personalizar, nuestro proyecto es un claro ejemplo.
- Dos plugins jQuery para crear gráficos, *Flot Charts* y *Morris.js*.

- Tablas con filtros, buscador y paginado usando *DataTables jQuery plugin.*
- Botones personalizados de *Bootsnipp.*
- Librería para botones de redes sociales de *Bootstrap Social.*
- Línea temporal responsiva de *Bootsnipp.*
- Chat *widget* de *Bootsnipp.*
- Página de *Login Bootsnipp.*

A continuación muestro la integración de **SB Admin 2** en mi proyecto, mostrando varias vistas basadas en el tema escogido.

Versión web de la interfaz

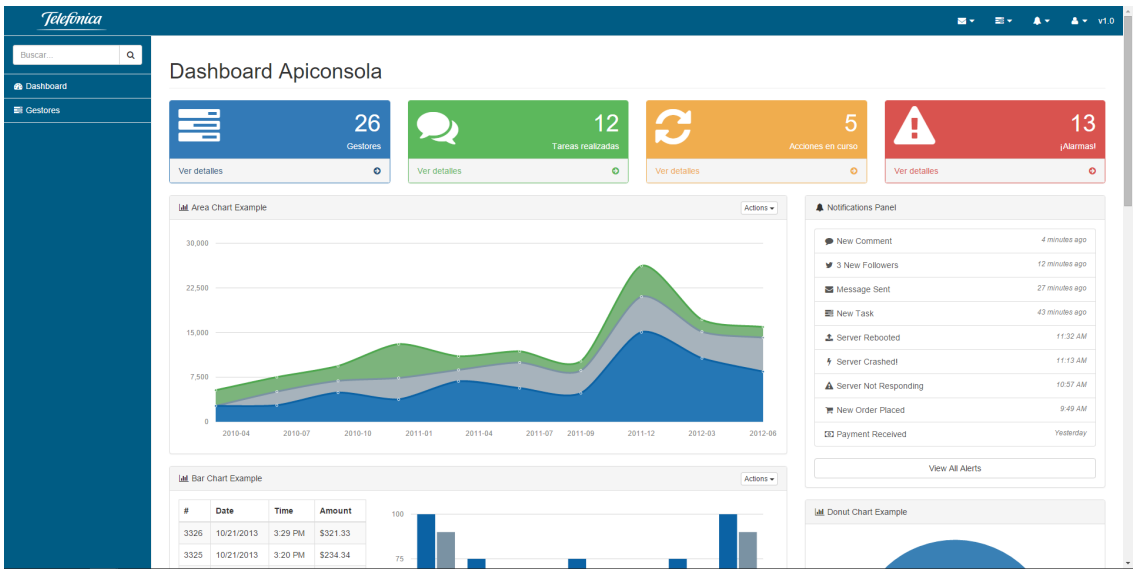

Fig. 21 Interfaz web APICONSOLA

Versión móvil

Como vemos una de las cosas que nos facilita trabajar con este tipo de plantillas web, es que hay cosas importantes que ya nos resuelven, como la adaptación de nuestra aplicación a todos los dispositivos.

Mediante la edición de los archivos CSS podemos dar el toque corporativo a la aplicación, así respetando los colores de la compañía.

Así también disponemos de un etiquetado interno para diferenciar cada componente y poder adaptar la plantilla a nuestra manera y así crear las vistas y ajustarlas a los requisitos del proyecto.

El trabajar de esta manera también nos proporciona nuevas ideas al crear el proyecto por ejemplo como mostraremos la información en nuestra aplicación.

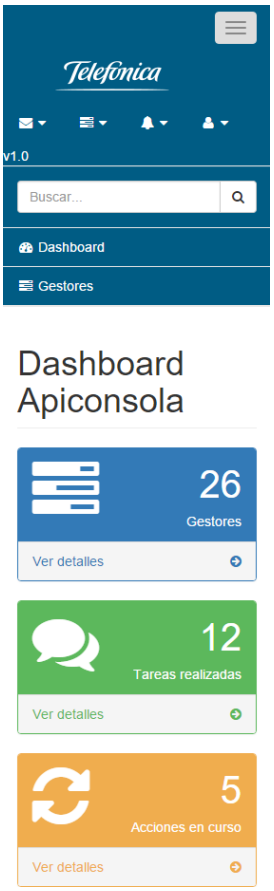

Fig. 22 Versión responsiva APICONSOLA

4.2.4 Maven

Maven es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl. La construcción está basado en un formato XML. Está bajo la licencia de Apache Software *Foundation*.

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. Viene con objetivos predefinidos para realizar ciertas tareas claramente definidas, como la compilación del código y su empaquetado.

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. Maven provee soporte

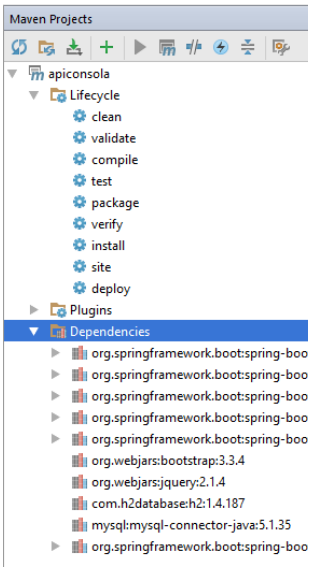

Fig. 23 Plugin MAVEN IntelliJ IDEA

no solo 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.

Al crear el proyecto mediante **MAVEN** se crea un fichero que controla todas las dependencias del proyecto, en el caso de agregar nuevas dependencias o actualizaciones este las descarga y deja el proyecto preparado para ser para ser ejecutado. Las partes del ciclo de vida principal del proyecto son:

1. ***compile***: Genera los ficheros .class compilando los fuentes .java
2. ***test***: Ejecuta los test automáticos de JUnit existentes, abortando el proceso si alguno de ellos falla.
3. ***package***: Genera el fichero .war con los .class compilados.
4. ***install***: Copia el fichero .war a un directorio de nuestro ordenador donde Maven deja todos los .war

5. ***deploy***: Copia el fichero .war a un servidor remoto, poniéndolo disponible para cualquier proyecto Maven con acceso a ese servidor remoto.

Fig. 24 Empaquetado de la aplicación con MAVEN

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.telefonica</groupId> <!-- Bajo que dominio se encuentra la aplicación -->
    <artifactId>apiconsola</artifactId> <!-- Como se llama la aplicación -->
    <version>1.0</version> <!-- Versión ACTUAL de la aplicación -->
    <packaging>war</packaging> <!-- Extensión del fichero al crear un paquete de nuestra aplicación -->

    <name>apiconsola</name> <!-- Nombre de la aplicación -->
    <description>Proyecto CNT: Apiconsola</description> <!-- Descripción de la aplicación -->

    <parent> <!-- Proyecto del que deriba nuestra aplicación, en este caso SPRING BOOT -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.4.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <properties> <!-- Propiedades de nuestra aplicación, en nuestro caso la codificación y version de JAVA -->
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <java.version>1.8</java.version>
    </properties>

    <dependencies> <!-- Listado de dependencias que usamos en nuestra aplicación -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId> <!-- Gestión BBDD mediante JPA -->
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId> <!-- Implementación de la seguridad de Spring -->
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId> <!-- Motor de generación HTML -->
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId> <!-- Dependencia de proyecto web -->
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId> <!-- Tomcat incrustado, auto despliegue de servidor -->
        <!--<scope>provided</scope>-->
    </dependency>

    <!--WebJars-->
        <dependency...>
        <dependency...>
        <dependency...>
        <dependency...>
        <dependency...>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Fig. 25 Fichero POM APICONSOLA

4.3 Control de código y documentación

En todo proyecto es fundamental hoy en día el control de código y la documentación. Para ello he escogido el software de Atlassian ya que todas sus herramientas están integradas entre ellas.

4.3.1 Stash (Git)

Control de código basado en Git o Mercurial (en mi proyecto uso Git). Este software está pensado para la eficiencia y confiabilidad del mantenimiento de versiones de aplicaciones cuando tenemos un extenso código fuente. Entre las características más relevantes se encuentran:

- **Desarrollo no lineal**, rapidez en la gestión de ramas y mezclado de diferentes versiones. El control basado en Git incluye herramientas específicas para navegar y visualizar un historial de desarrollo no lineal. Un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.

- **Gestión distribuida**: Git proporciona a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en la misma manera que se hace con la rama local.

- Los almacenes de información pueden publicarse por **HTTP**, FTP, rsync o mediante un protocolo nativo, ya sea a través de una conexión **TCP/IP** simple o a través de cifrado **SSH**.

- **Gestión eficiente** de proyectos grandes, dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución.

- **Todas las versiones previas a un cambio determinado**, implican la notificación de un cambio posterior en cualquiera de ellas a ese cambio (denominado autenticación criptográfica de historial).

- Los renombrados se trabajan basándose en similitudes entre ficheros, aparte de nombres de ficheros, pero no se hacen marcas explícitas de cambios de nombre con base en supuestos nombres únicos de nodos de sistema de ficheros, lo que evita posibles, y posiblemente desastrosas, coincidencias de ficheros diferentes en un único nombre.

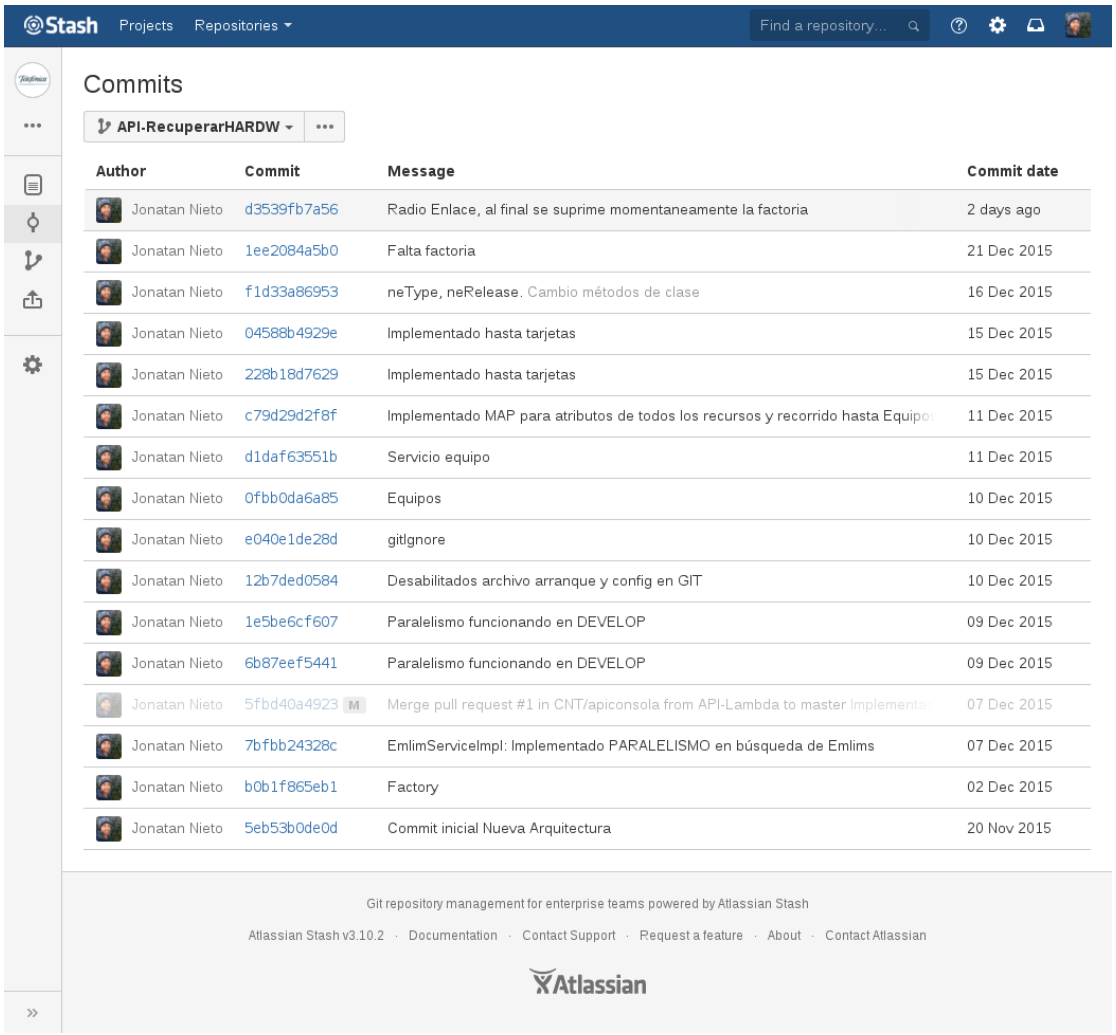

Fig. 26 Interfaz web Stash para la gestión de ramas

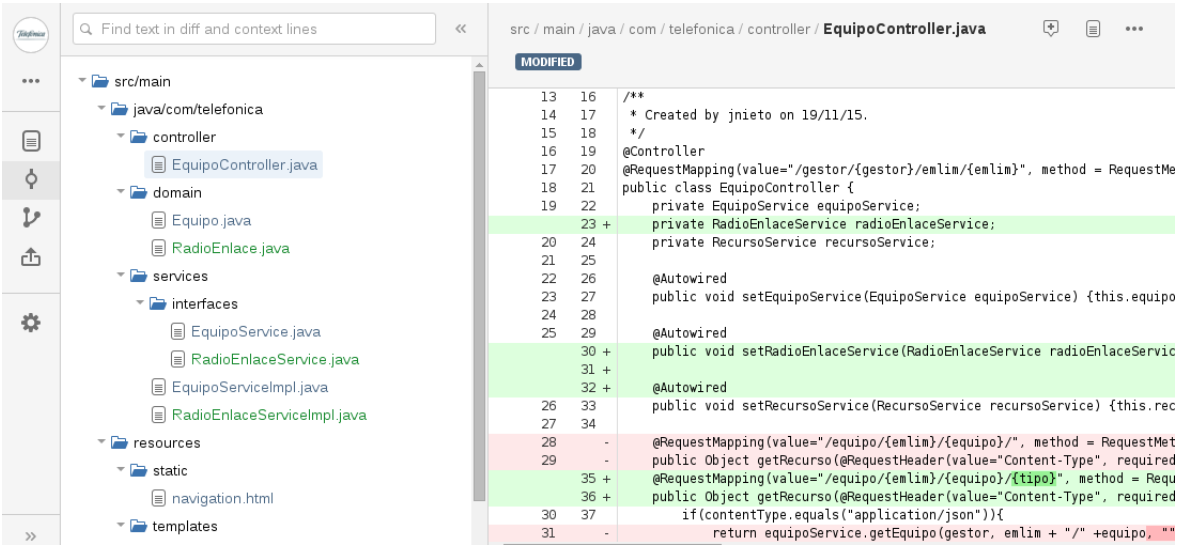

Fig. 27 Ejemplo commit en Stash

Para realizar los *commits* dentro de una rama utilizo la integración de GIT dentro de IntelliJ IDEA. Para subir los cambios de versión uso la línea de comandos, debido a que como para acceder a los servidores hay un proxy por medio y así no hay tantos problemas desde la consola para realizar las acciones pertinentes.

4.3.2 Jira (Control de *ticketing*)

Seguidamente después de integrar el control de código, una vez que tenemos el core de la aplicación lo que va a ir haciendo Telefónica es ir añadiendo nuevos requisitos y funcionalidades en la aplicación, entonces este proyecto lo voy a gestionar con Jira, un control de *ticketing* donde se van a ir añadiendo las tareas a realizar y posteriormente se asignan los trabajos.

La metodología que vamos a usar es **SCRUM** (desarrollo ágil), donde lo que se quiere conseguir es una continua actualización de la aplicación y a vista del cliente ir añadiendo pequeños requisitos de una manera constante. Con esto lo que consigo es de cara al cliente siempre tiene constancia de actualización en la aplicación y el nivel de cambios en ella es más asumible.

Conceptos básicos SCRUM:

Principio básico: entrega temprana al cliente de software (entre 2 semanas y 2 meses) con valor para su satisfacción. No se alienta la resistencia al cambio, como es la tónica habitual en los proyectos, sino que pretende aprovechar para aumentar la ventaja competitiva del cliente y su satisfacción. Por otra parte, se pretende obtener un ritmo constante de desarrollo.

***Sprint 0**:* fase inicial donde se crea el *Product Backlog*. Cuanto más breve mejor. A partir de ahí, comienzan los sprint. Puede durar unas semanas o un mes aprox.

Sprint: iteración corta de desarrollo entre 1-4 semanas generalmente. Se suceden uno tras otro.

Product Backlog: Lista priorizada y ordenada y estimada de requisitos de alto nivel o historias de usuario. Representa el alcance y la planificación del proyecto. La Prioridad en la lista: puede deberse a mitigar riesgos, necesidades técnicas, dependencias, satisfacción de áreas corporativas, alineación con estrategia de la empresa, etc.. en ningún caso será capricho del ***Product Owner***.

Gráfico de *Burndown*: es un gráfico que muestra que alcance hay que entregar sprint por sprint para alcanzar el objetivo. Lo ideal es que la cantidad pendiente de hacer vaya disminuyendo, por lo que será una gráfica de un arco descendente, aunque se pueden producir paradas. Se crea durante la planificación del reléase y es actualizado en cada sprint por el ***Product Owner***.

Taskboard: es un instrumento de autogestión del equipo. Contiene un *post-it* por cada historia de usuario que incluye el sprint. Se van colocando en la columna que corresponda: no comenzado, comenzado y terminado. Debe estar siempre actualizado. Refleja fielmente quien está haciendo qué cosa.

A continuación muestro nuestro primer ***Backlog*** y ***Sprint*** 1, donde vemos dos desarrolladores activos y las tareas que ha marcado el cliente en el ***Backlog*** para ir introduciéndolas en diferentes ***Sprints***. También vemos el detalle de una tarea.

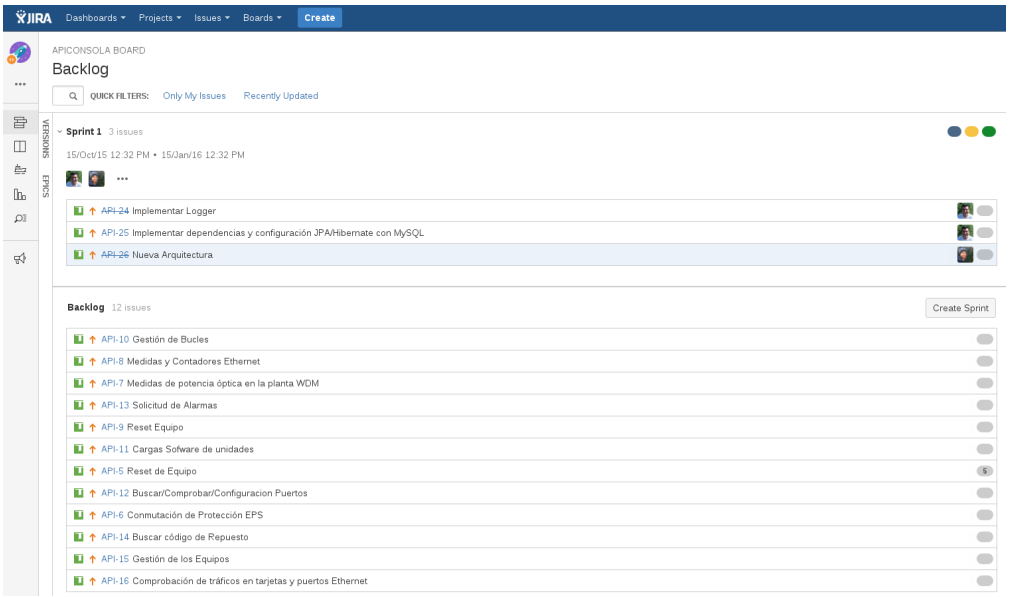

Fig. 28 Backlog en Stash

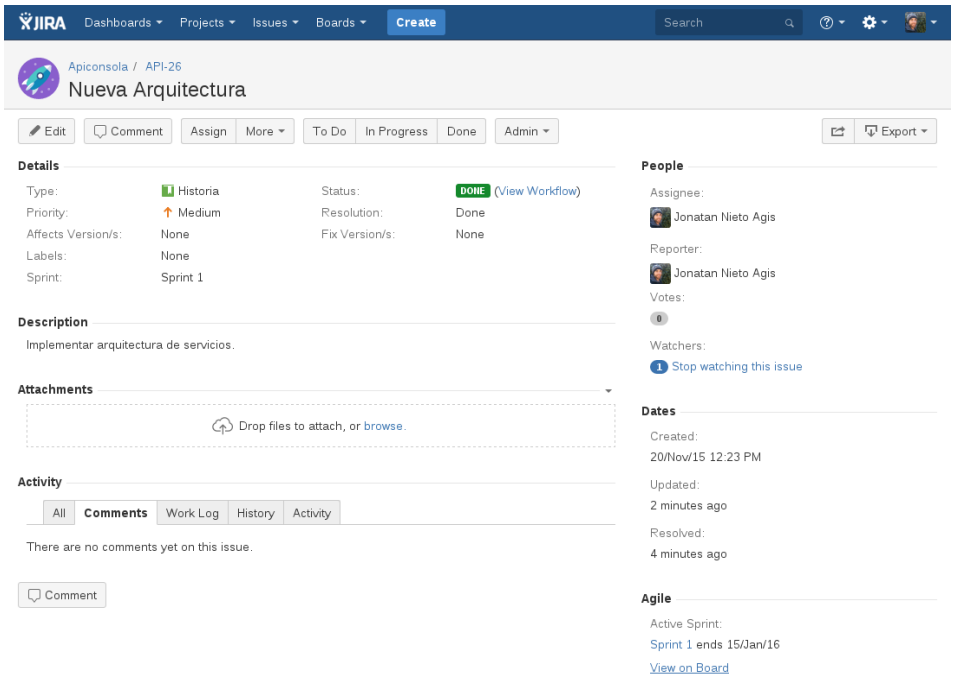

Fig. 29 Ejemplo tarea en Stash

4.3.3 Confluence (Documentación)

Software para centralizar la documentación de un proyecto y la colaboración del equipo. Al trabajar en un solo lugar lo que se consigue es documentar al momento todo y tenerlo en una plataforma donde puede acceder todo el equipo de desarrollo.

Fig. 30 Antes y después de Confluence

Por cada proyecto podemos crear un espacio de trabajo y añadir a los miembros de equipo, también administramos los permisos según las funciones que tenga cada desarrollador.

Dispone de un editor en el propio navegador, así que podemos hacer cualquier cosa.

Una de las características es que en toda la plataforma de documentación, podemos mencionar a cualquier miembro del equipo y es muy útil para en aquellos puntos importantes poder "avisar" a los desarrolladores y dejar ese punto como importante y que lo deben consultar.

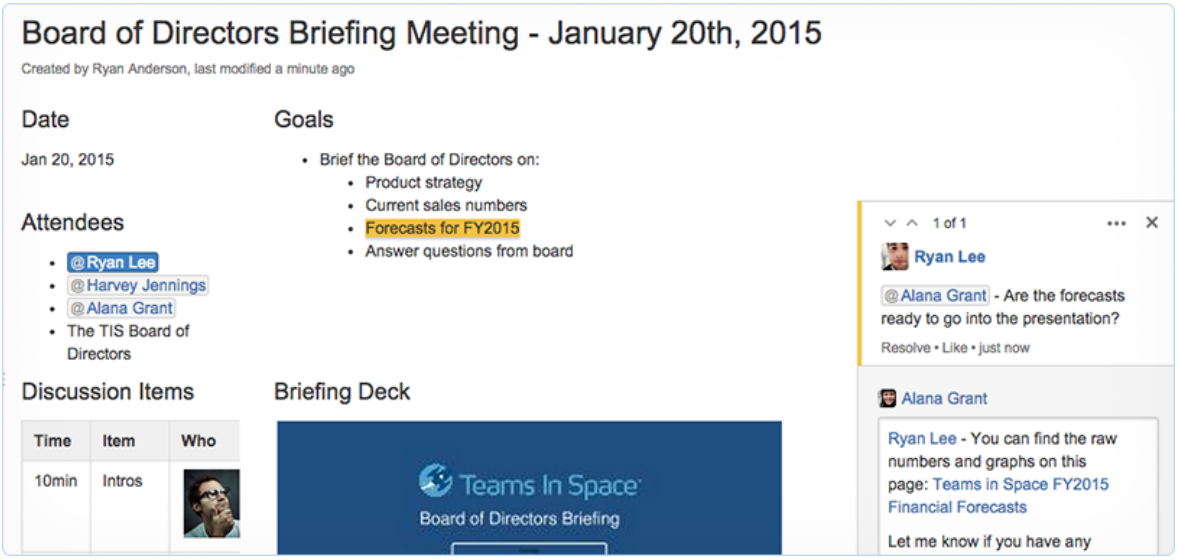

Fig. 31 Ejemplo tablero Confluence

También está preparado para el desarrollo ágil e integrado con Jira de manera que todo en lo que tu equipo está trabajando en **JIRA** (tareas, *sprints*, versiones) aparecerá a cualquiera que esté en **Confluence** con **Team Calendars**. Evita sorpresas visualizando los viajes del equipo junto al trabajo programado.

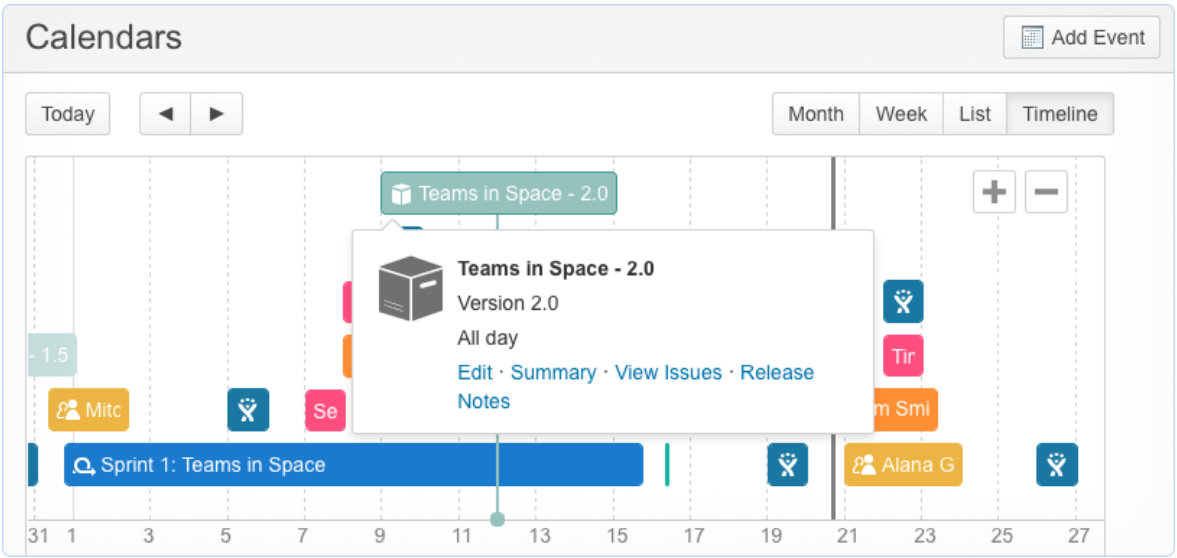

Fig. 32 Calendario Confluence

4.4 Despliegue de la aplicación

En este capítulo explico los diferente entornos de ejecución de la aplicación, en una aplicación empresarial siempre encontramos diferentes entornos donde vamos a ver corriendo nuestra aplicación. Cada empresa tiene sus diferencias pero en la gran mayoría podemos encontrar que destacan los tres entornos que explicaremos en los puntos de este capítulo.

Al trabajar con control de código, como hemos visto en los capítulos anteriores, nos facilita mucho este punto dentro de nuestro proyecto. Con las herramientas Git es fácil poner en marcha los diferentes "entornos" del proyecto, tenemos las siguientes ramas:

- **Master:** aquí tenemos el código fuente de la aplicación que está en producción, este se caracteriza porque está atacando a un entorno real, en nuestro caso gestionando los equipos de la red que están funcionando para el cliente final.
- **DEV**: encontramos la versión que actualmente está en desarrollo e incorpora cambios de otras ramas que han sido desarrolladas localmente y se están testeando para verificar su funcionamiento y los resultados que espera el cliente.
- **Ramas locales**: es cada rama que se va creando en local, es decir cuando tenemos que añadir un nuevo requerimiento al proyecto hacemos un clon de la master y seguidamente desarrollamos el nuevo código sobre esta nueva rama. Finalmente acabada de probará en desarrollo y si se aprueba entrarán los cambios en la siguiente **RELEASE** que hagamos en producción.

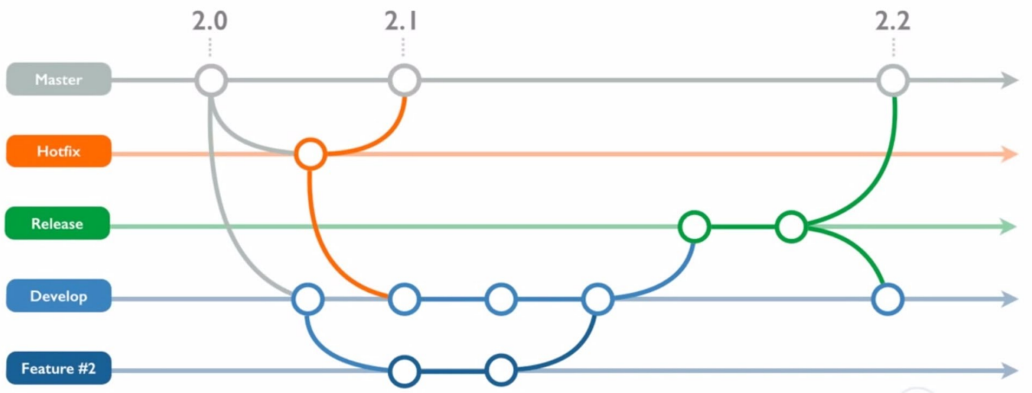

Fig. 33 Ramas GIT

- **Release**: se crea una rama con los nuevos cambios y se pone en un servidor alternativo al de producción pero con las mismas características. Lo que

normalmente se hace es derivar tráfico a la nueva versión ***Release*** para que los usuarios la usen o simplemente que ciertas personas puedan acceder y probar esta versión.

- **Hotfix**: cuando se detectan bugs en la aplicación se realiza un *hotfix*, esto es un cambio en el código puntual y se despliega rápidamente en las versiones de **DEV** y **Producción**, si tenemos alguna ***Release*** en marcha también se aplica a esta.

4.4.1 Local

Es el entorno de desarrollo del programador, donde se codifican todas las nuevas ramas con las nuevas implementaciones que quiere el cliente. Las características en cuanto a software del equipo son las siguientes:

- Centos 7
- Escritorio Gnome
- IntelliJ IDEA 15
- JDK 1.8

Los equipos de desarrollo local los tengo conectados a la red de Telefónica mediante una máquina de salto y un proxy (que controlan todas las acciones que se realizan todos los usuarios) debido a que aunque estemos desarrollando localmente en este proyecto necesitamos conectarnos al entorno de desarrollo para poder probar contra equipos que son iguales que los originales, es decir que no se simula un entorno de producción, sino que Telefónica tiene X **gestores** que se usan para hacer pruebas.

Una de las cosas importantes del proyecto es que cuando codificamos en local, tenemos un un **Tomcat 8** incrustado en la aplicación, que nos permite probar la misma de manera muy rápida y sin necesidad de crear un paquete (**.war**) del proyecto y tener que ponerlo en un servidor. En las siguientes imágenes podemos ver las diferentes líneas que cambian en el entorno local para poder desplegar el proyecto tal y como he explicado anteriormente:

- Vemos que en la clase principal de la aplicación, lo que está comentado es solo para cuando no tenemos el **Tomcat 8** incrustado y lo ejecutamos en un servidor:

ApiconsolaApplication.java

```
package com.telefonica;

import ...

@SpringBootApplication
public class ApiconsolaApplication /*extends SpringBootServletInitializer*/ {


    /*@Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        // Customize the application or call application.sources(...) to add sources
        // Since our example is itself a @Configuration class we actually don't
        // need to override this method.
           return application;
    }*/

    public static void main(String[] args) {
        SpringApplication.run(ApiconsolaApplication.class, args);
    }
}
```

Fig. 34 Main Spring Boot con Tomcat incrustado

- Vemos que cuando el **Tomcat 8** lo quiero incrustado le debo quitar la etiqueta provided, qué quiere decir cuando ejecutamos la aplicación en un servidor:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId> <!-- Tomcat incrustado, auto despliegue de servidor -->
    <!--<scope>provided</scope>-->
</dependency>
```

Fig. 35 Diferencias POM Tomcat incrustado

Esto es considerada una **BEST PRACTICE** ya que trabajar así nos ahorra mucho tiempo en mientras desarrollamos, ya que cada cambio lo podemos probar al momento con un solo click. Previamente lo único que debemos configurar es el entorno de desarrollo para que nos despliegue la aplicación. En IntelliJ se configura en dos pasos:

- Agregando una nueva configuración de Run/Debug:

Fig. 36 Nueva configuración para desplegar aplicación

- Configurando un nuevo ejecutable de **Aplicación**, donde se ha de marcar la **clase principal** y el ***path*** de nuestro **JRE** local:

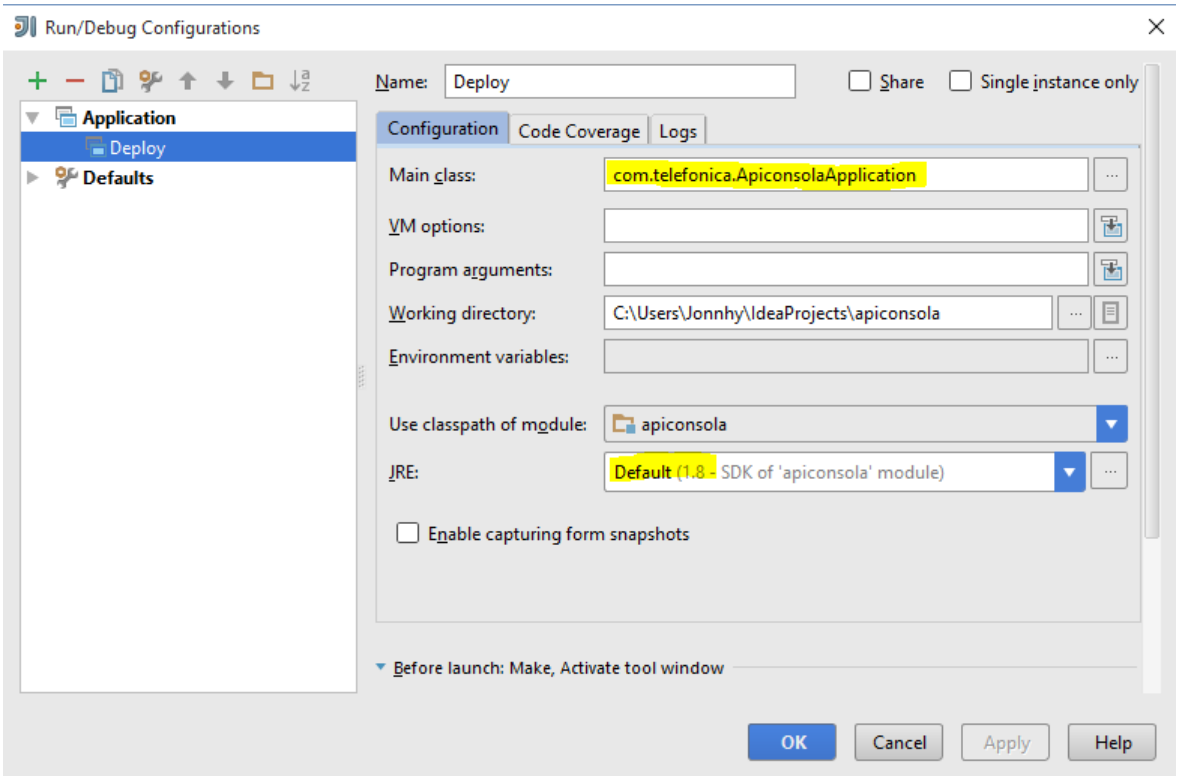

Fig. 37 Configuración ejecutable aplicación IntelliJ IDEA

Otra característica del entorno local y al trabajar con editores profesionales es la facilidad con la que puedo depurar la aplicación, como había mencionado antes, **IntelliJ IDEA** dispone de un depurador muy avanzado que permite al programador saber en todo momento los valores de las variables de la aplicación y hacer seguimiento del código.

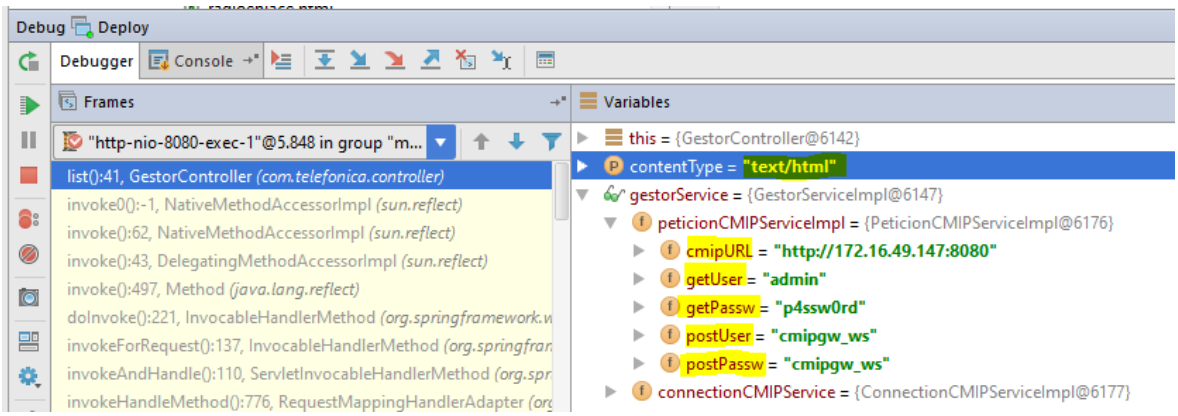

Fig. 38 Depurador IntelliJ IDEA

4.4.2 Desarrollo/Certificación

Las características del servidor de pruebas son las siguientes:

- S.O. Red Hat
- Java 1.8
- Tomcat 8

Este servidor tiene conectividad con el servidor de pruebas **CMIP de Telefónica** (el que permite comunicarnos con los gestores y equipos), donde atacamos a unos gestores de prueba.

Cuando están acabadas nuevas ramas en local, la siguiente tarea es probarlas en desarrollo. Para ello subimos la rama al servidor de control de código y la juntamos a la rama de **DEV** para así poder crear un paquete con los nuevos cambios y subirlos al servidor de pruebas.

The Apache Software Foundation
http://www.apache.org/

Gestor de Aplicaciones Web de Tomcat

Mensaje:	OK

Gestor			
Listar Aplicaciones	Ayuda HTML de Gestor	Ayuda de Gestor	Estado de Servidor

Aplicaciones					
Trayectoria	Versión	Nombre a Mostrar	Ejecutándose	Sesiones	Comandos
/	Ninguno especificado		true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/docs	Ninguno especificado	Tomcat Documentation	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/examples	Ninguno especificado	Servlet and JSP Examples	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/host-manager	Ninguno especificado	Tomcat Host Manager Application	true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/apiconsola-1.0	Ninguno especificado		true	0	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos
/manager	Ninguno especificado	Tomcat Manager Application	true	1	Arrancar Parar Recargar Replegar Expirar sesiones sin trabajar ≥ 30 minutos

Fig. 39 Interfaz web Apache Tomcat 8

4.4.3 Producción

El servidor de producción es similar al de desarrollo pero ataca al entorno real. Las características del servidor son:

- Red Hat
- Java 1.8
- Tomcat 8
- **SSL**
- **Se registran todos los movimientos** de los usuarios y las aplicaciones de terceros que usan nuestro software.
- Todas las peticiones se derivan al puerto **HTTPS.**
- Conectado directamente con el entorno de producción de Telefónica, por lo tanto tenemos acceso directo a la **CMIP**.

En cada nueva versión de la rama master se despliega el **.war** en el Tomcat 8 del servidor.

5. Definición de la aplicación

5.1 Interfaz

5.1.1 Interfaz web

Login

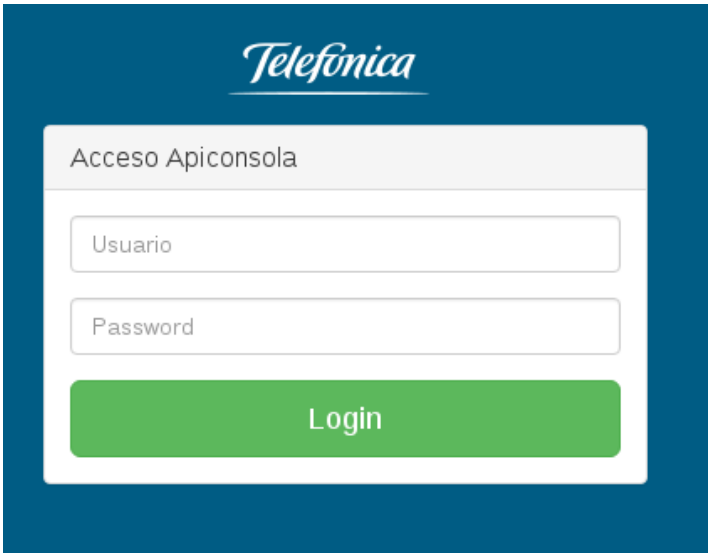

Fig. 40 Login

Dashboard

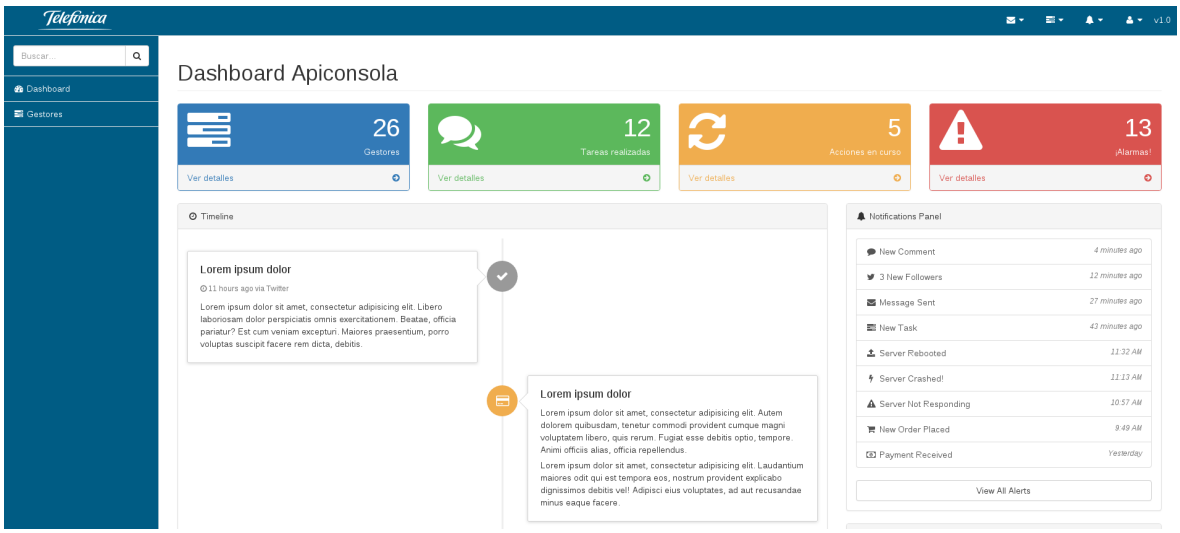

Fig. 41 Dashboard

Gestores

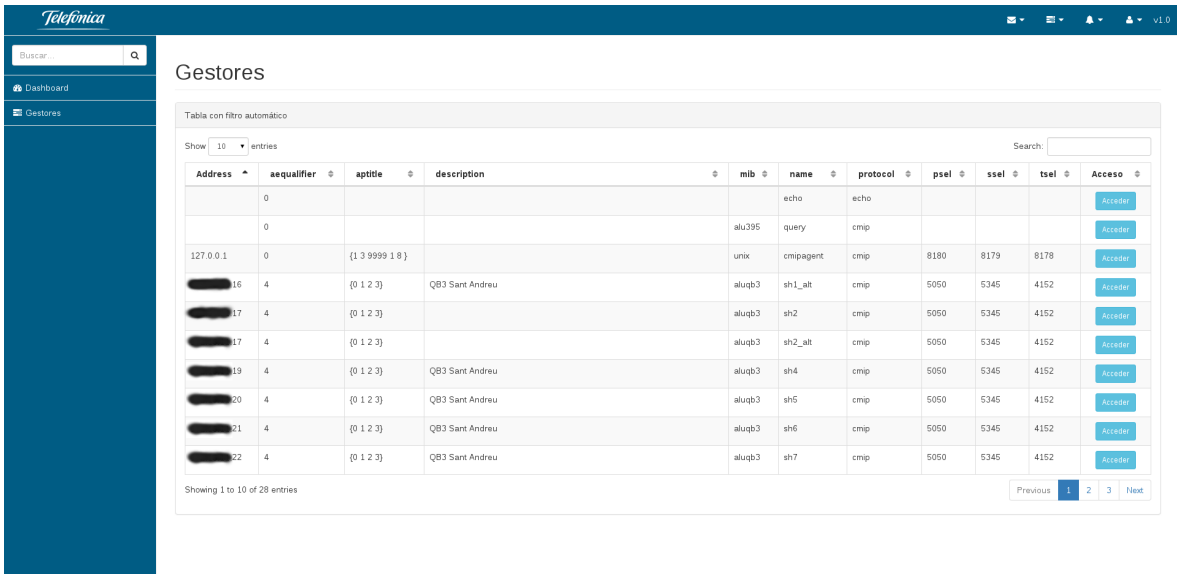

Fig. 42 Interfaz web gestores

Gestor

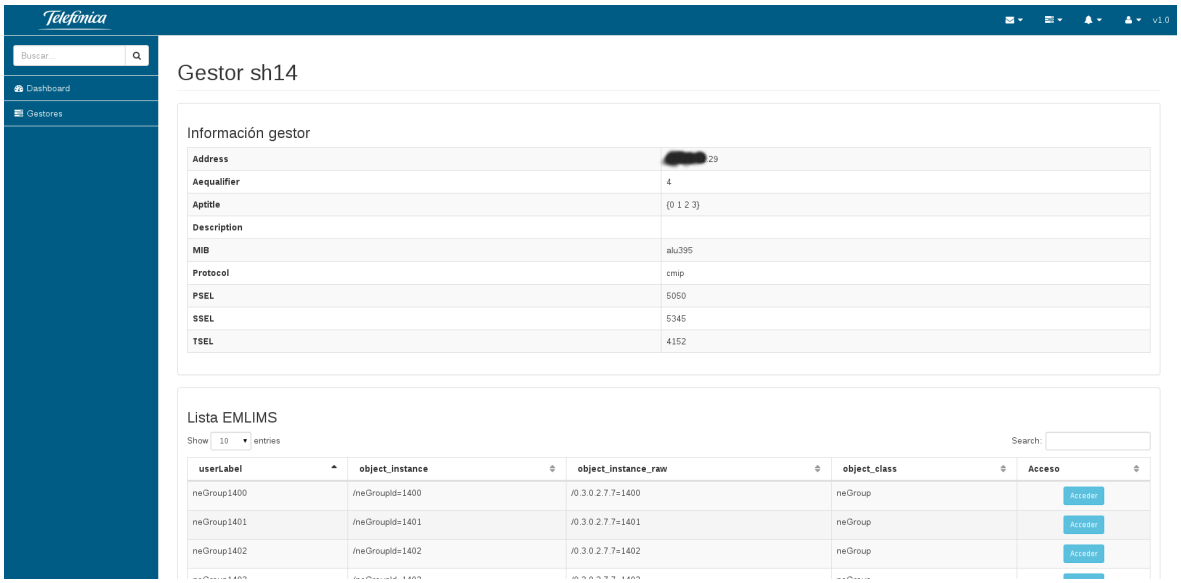

Fig. 43 Pantalla web gestor

Emlim

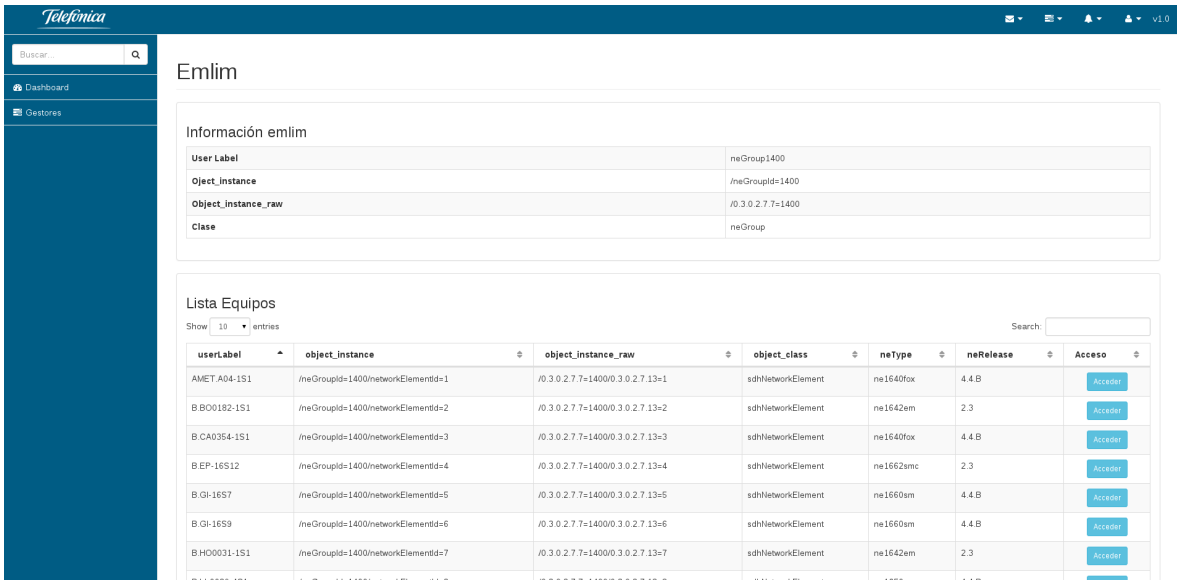

Fig. 44 Pantalla web Emlim

Equipo

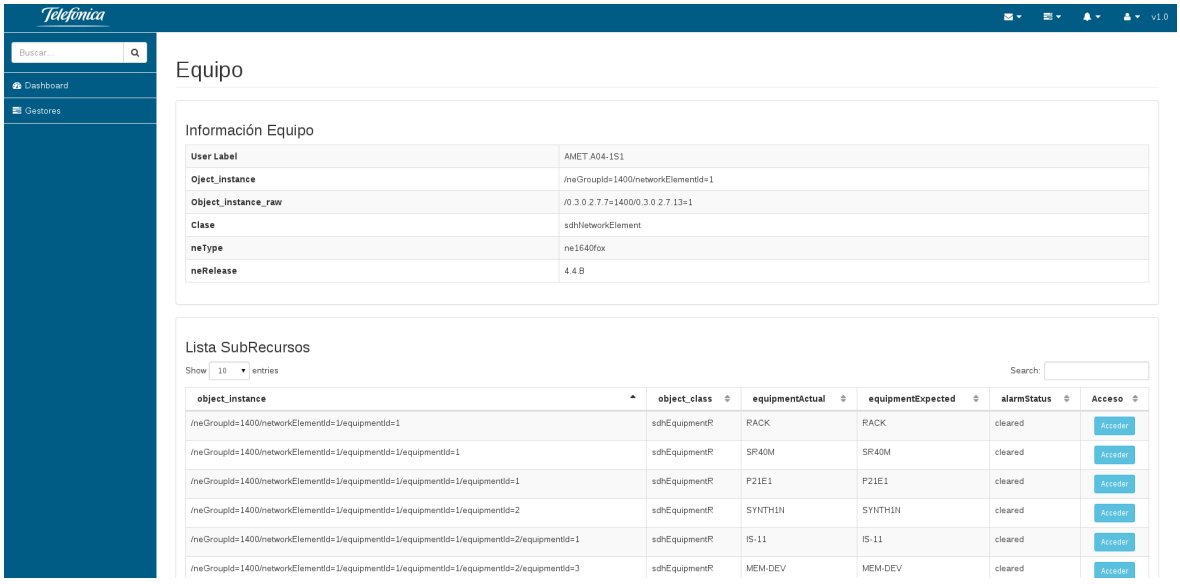

Fig. 45 Interfaz web Equipo

Radio enlace

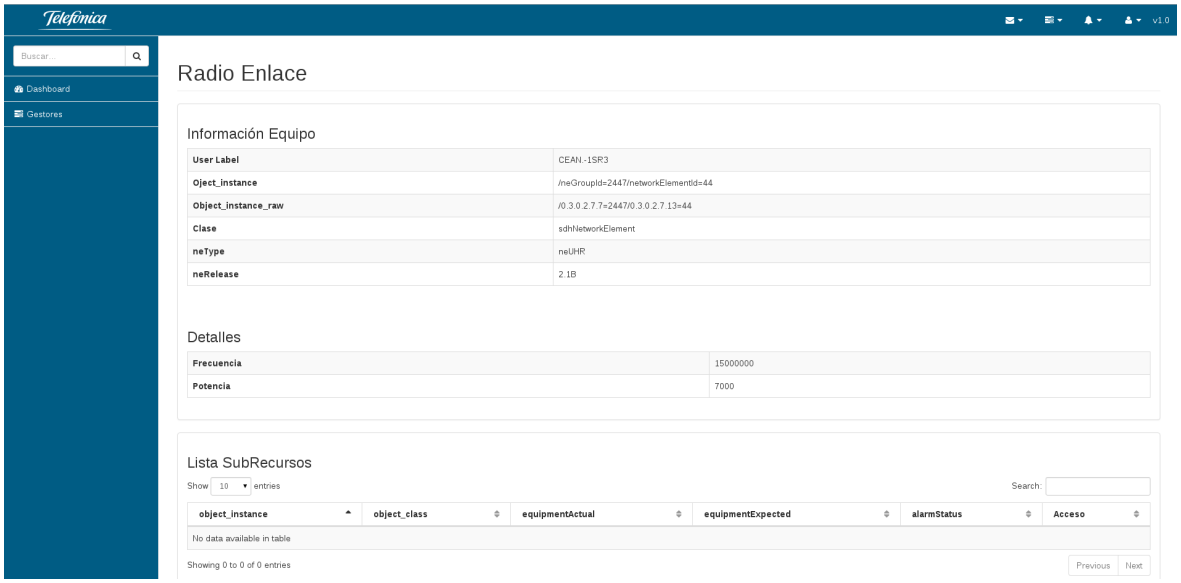

Fig. 46 Interfaz web Radio enlace

5.1.2 Interfaz móvil

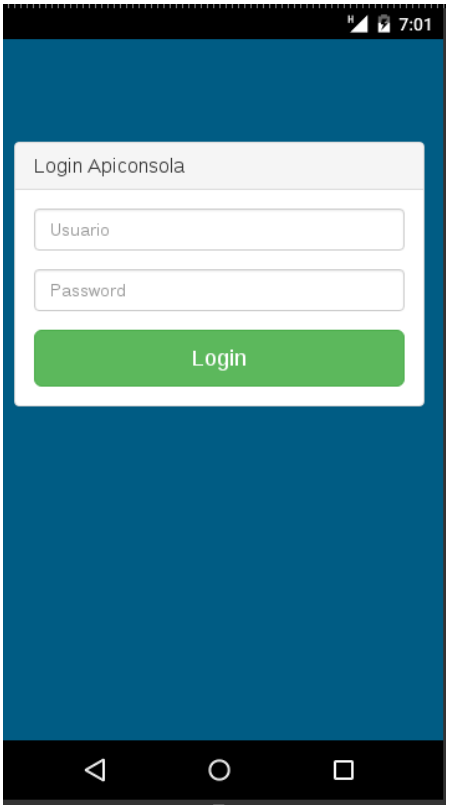

Fig. 47 Interfaz móvil Login

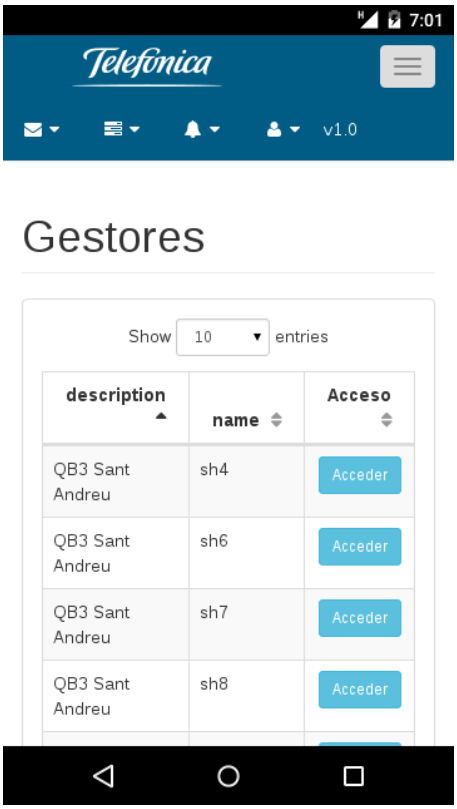

Fig. 48 Interfaz móvil gestores

Fig. 49 Interfaz móvil gestor

Fig. 50 Interfaz móvil emlim

Fig. 51 Interfaz móvil Equipo

Fig. 52 Interfaz móvil radio enlace

5.1.3 Peticiones sin interfaz

Este punto es importante ya que nuestra aplicación la van a usar tanto empleados de la compañía como aplicaciones autómatas que deciden acciones a realizar en el entorno de Telefónica, la mayoría de peticiones hacía nuestra aplicación van a venir por terceras aplicaciones.

5.2 Requerimientos

5.2.1 Login

La primera pantalla de la aplicación consistirá en una pantalla de *login*, sólo pueden acceder los usuarios que disponen acceso. Cada usuario tendrá un perfil diferente y sólo podrá acceder a las funciones que tenga disponible y un administrador le haya asignado previamente. El intento de acceso a cualquier página sin previamente estar *logueado* dirige a esta pantalla.

5.2.2 Dashboard

Al acceder a la aplicación se mostrará un *Dashboard* con datos sobre los gestores, Emlims y equipos del sistema. Estos datos cambiarán y se ampliarán a medida que se vaya desarrollando la aplicación. Cada usuario, según su perfil tendrá un *Dashboard* diferente y podrá ver información específica relacionada con su puesto de trabajo.

En esta pantalla también se mostrará un menú según el rol de usuario.

5.2.3 Recuperar gestores de la CMIP

Debemos poder recuperar los gestores, para ello se debe hacer una petición para que el servidor CMIP devuelva la información de los gestores en línea y la aplicación muestre la información. De cada gestor se ha de mostrar:

- **Address:** dirección IP del gestor.
- **Aequalifier:** calificador de aplicación.
- **Aptitle:** identificador capa aplicación.
- **Description:** referencia que describe donde se encuentra este gestor.

- **MIB:** tipo de MIB que tiene.
- **Name:** nombre del gestor.
- **Protocol:** protocolo de comunicación del gestor.
- **PSEL:** Selector capa presentación.
- **SSEL:** Selector capa sesión.
- **TSEL:** Selector capa transporte.
- **Acceso:** Link de acceso al gestor.

5.2.4 Mostrar gestor y su lista de Emlims

Al acceder a un gestor debemos mostrar la información del gestor y todos los Emlims que cuelgan de él. La información del gestor se mostrará en una tabla de Atributo-Valor y el listado de los Emlims en una tabla, donde mostraremos la siguiente información de cada uno:

- **UserLabel:** Identificador de red.
- **Object_instance:** dirección de un Emlim, más legible por el usuario.
- **Object_instance_raw:** dirección de un Emlim en formato RAW.
- **Object_class:** tipo de objeto o familia.
- **Acceso:** Link de acceso al Emlim.

5.2.5 Mostrar Emlim y sus equipos

Mostraremos la información del Emlim y los equipos que están colgando de él. La información que hay que mostrar es:

- **User Label:** Identificador de red.
- **Oject_instance:** dirección de un Emlim, más legible por el usuario.
- **Object_instance_raw:** dirección de un Emlim en formato RAW.
- **Clase:** tipo de objeto o familia.

La información de los equipos que cuelgan hay que mostrarlos en formato tabla y es la siguiente:

- **UserLabel:** Identificador de red.
- **Object_instance:** dirección del equipo, más legible por el usuario.

- **Object_instance_raw:** dirección del Equipo en formato RAW.
- **Object_class:** tipo de objeto o familia.
- **NeType:** atributo importante en esta parte de la aplicación ya que diferencia la clase de equipo al cual vamos a acceder y según este tipo el usuario de la aplicación ya espera una información específica que va relacionada a este valor.
- **NeRelease:** versión del equipo.
- **Acceso:** Link de acceso al equipo.

5.2.6 Mostrar equipo y sus recursos

Aquí está el objetivo de la aplicación y es la parte que va a crecer a medida que el personal de Telefónica vaya estableciendo nuevos requerimientos. Según el ***neType*** se va a mostrar cierta información importante sobre el equipo y, en el caso que disponga, se mostrarán los recursos que cuelgan del mismo. La información que se va a mostrar de cada equipo es la siguiente:

- **UserLabel:** Identificador de red.
- **Object_instance:** dirección del equipo, más legible por el usuario.
- **Object_instance_raw:** dirección de un Equipo en formato RAW.
- **Object_class:** tipo de objeto o familia.
- **NeType:** clase de equipo.
- **NeRelease:** versión del equipo.

Según que equipo se muestre se van a detallar una serie de parámetros, de momento los equipos que encontramos son:

- **Defecto**: cualquier equipo que aún no esté detallado.
- **Radio Enlace:** enlace punto a punto
- **Tarjeta:** los equipos tienen tarjetas que controlan varios parámetros de la red.
- **Alarma:** indica el estado de un equipo o recursos del mismo.

Se implementarán más a medida que vayamos evolucionando la aplicación.

5.2.7 Equipo por defecto

Se mostrará la información detallada en el equipo y la lista de recursos que tiene por debajo.

5.2.8 Radio Enlace

Además de la información mostrada en el equipo por defecto se mostrarán los detalles de los siguientes atributos del radio enlace:

- **Frecuencia**: Hz
- **Potencia**: W

Podrán implementarse más a medida que vayamos evolucionando la aplicación.

5.2.9 Alarma

La aplicación ha de mostrar la información del estado de las alarmas de los equipos, en los casos que se especifiquen estas alarmas deben mostrarse y el usuario podrá tomar acciones sobre el estado de la misma. El atributo principal de la alarma de un recurso es ***alarmStatus*****,** que define el estado actual del recurso.

5.2.10 Conectar con gestor

Un punto importante de la aplicación, es que **cualquier acción** que se hace contra un gestor o los recursos que cuelgan del mismo se ha de **crear una conexión** y todas las acciones deben ir mediante esta, ya sea para recuperar información o hacer acciones.

Las conexiones **tienen una caducidad**, así que por cada petición debemos comprobar la conexión pidiendo otra a la **CMIP** (si existe conexión, nos devuelve esta, de lo contrario crea una nueva conexión y nos la proporciona).

5.2.11 Peticiones CMIP

Debemos tener una clase en la aplicación que se comunique con la CMIP. Las peticiones que se pueden realizar son dos tipos:

1. **GET**: utilizada para recuperar los gestores o información de un gestor en concreto.
2. **POST**: para las demás peticiones o acciones. Se deben pasar en el cuerpo del mensaje los siguientes valores en formato **JSON**:
    - **Method**: GET o POST, según si es consulta o acción.
    - **Params**: Parámetros que indican el recurso de la **CMIP** que queremos recuperar y qué queremos que nos devuelva. Principalmente trabajaremos con los siguientes parámetros:
        - **object_instance:** Recurso del gestor que queremos recuperar.
        - **object_class:** tipo de recurso que queremos recuperar.
        - **Scope**: Indica los niveles de objetos que queremos recuperar.
        - **Attributes**: Información del recurso que queremos recuperar.

5.2.12 Peticiones de terceros

Debemos preparar nuestra aplicación para que las peticiones que vengan de terceras aplicaciones tengan una **API de acceso** y les devolvamos las respuestas en formato **JSON**. Esta **API solo puede tener una URL de acceso** y en el cuerpo del mensaje vendrá la información de la acción que quiere realizar el autómata que nos hace la llamada.

5.3 Casos de uso

5.3.1 Login aplicación

Actor: Usuario

Propósito: Control de acceso a nuestra aplicación, el usuario que quiere acceder debe tener un usuario en nuestro sistema y un rol asignado.

Precondición: Los administradores de la aplicación deben dar de alta al usuario y asignarle un rol.

Flujo normal de eventos:

1. Acceder a la aplicación web, mediante móvil o navegador web.
2. Introducir usuario.
3. Introducir contraseña.
4. Presionar botón *Login*.
5. Si el *login* es correcto accede al Dashboard de la aplicación.

Fig. 53 Flujo normal del Login

Flujo alternativo:

- *Login* incorrecto:
  - 4.1. Si al presionar el botón indicado en el paso 4 el *login* es incorrecto volveremos al paso 2, donde el usuario volverá a repetir el proceso.
- *Login* terceras aplicaciones:
  - Las aplicaciones de terceros que hagan peticiones a nuestro entorno deberán incluir la autorización en el cuerpo del mensaje.

Postcondición: Al acceder a la aplicación cualquier movimiento puede ser registrado.

5.3.2 Dashboard

Actor: Usuario

Propósito: Acceso a las funciones que tiene cada usuario.

Precondición: Según el rol que se le asigna al usuario tendrás más o menos acciones en el ***Dashboard***. También se muestra información y notificaciones.

Flujo normal de eventos:

1. Acceso al ***Dashboard*** (/).
2. Buscar la acción que se desea realizar (Ej. **Gestores** para visualizar los gestores del sistema).
3. Pulsar sobre la acción.
4. El sistema redirige a la página donde se muestra la información.

Flujo alternativo:

- Acceso terceras aplicaciones:
    - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar las acciones disponibles a través de la API.

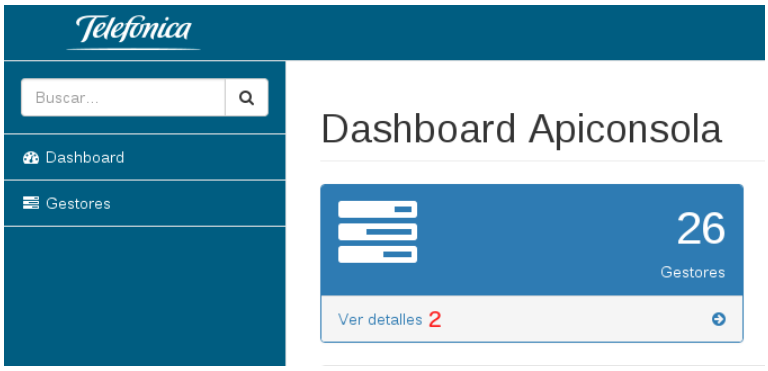

Fig. 54 Acceso a gestores desde Dashboard

5.3.3 Gestores

Actor: Usuario

Propósito: Se muestra la información de los gestores a los que el usuario puede acceder.

Precondición: Asignar rol a usuario, según el rol del usuario tendrá restricciones sobre los gestores que puede visualizar.

Flujo normal de eventos:

1. Acceso al listado de gestores (/gestores).
2. Se muestra una lista de gestores con los atributos principales de los mismos.
3. Puede acceder al gestor pulsando el botón "**Acceder**".
4. Si pulsa el botón del paso 3 el sistema redirige a la página donde se muestra la información.

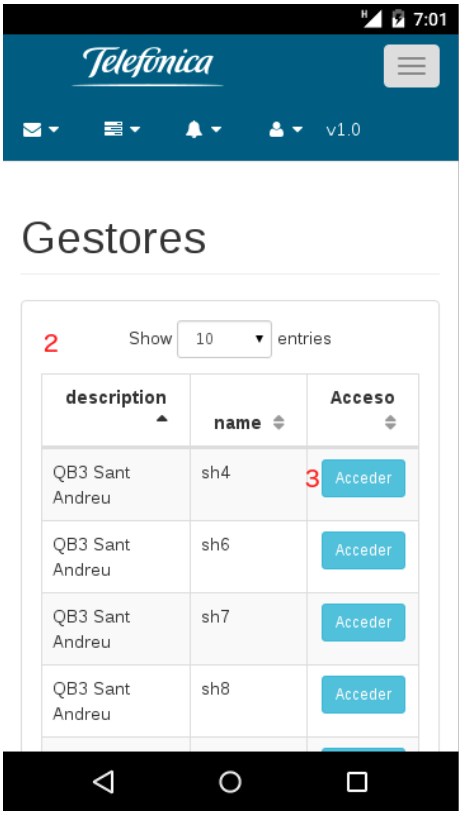

Fig. 55 Flujo normal gestores

Flujo alternativo:

- Búsqueda:
  - 2.1 El usuario puede filtrar la lista de gestores introduciendo en el buscador cualquier referencia de los atributos mostrados.
- Acceso terceras aplicaciones:
  - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar la información disponible a través de la API.

Postcondición: El usuario puede tener visualización del gestor pero si no tiene permisos a la información interna no podrá acceder.

5.3.4 Gestor

Actor: Usuario

Propósito: Se muestra la información del gestor al que se ha accedido y la lista de Emlims que cuelgan del mismo.

Precondición: Asignar rol a usuario, según el rol del usuario tendrá restricciones sobre los gestor que quiere visualizar.

Flujo normal de eventos:

1. Acceso al gestor (/gestor/{nombre_gestor}).
2. Se muestra la información y una lista de Emlims a los que puede acceder.
3. Puede acceder a los Emlims mostrados pulsando el botón "**Acceder**" de cualquiera de los elementos.
4. Si pulsa el botón del paso 3 el sistema redirige a la página donde se muestra la información del elemento.

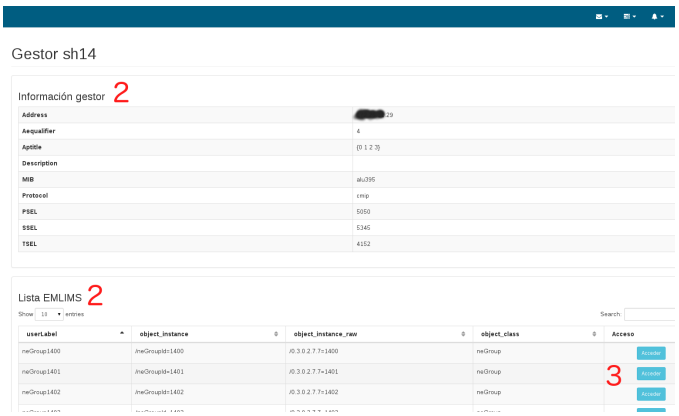

Fig. 56 Flujo normal gestor

Flujo alternativo:

- Búsqueda:
  - 2.1 El usuario puede filtrar la lista de Emlims introduciendo en el buscador cualquier referencia de los atributos mostrados.
- Acceso terceras aplicaciones:
  - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar la información disponible a través de la API.

Postcondición: El usuario puede tener visualización del Emlim pero si no tiene permisos a la información interna no podrá acceder.

5.3.5 Emlim

Actor: Usuario

Propósito: Se muestra la información del Emlim al que se ha accedido y la lista de equipos que cuelgan del mismo.

Precondición: Asignar rol a usuario, según el rol del usuario tendrá restricciones sobre los Emlim que quiere visualizar.

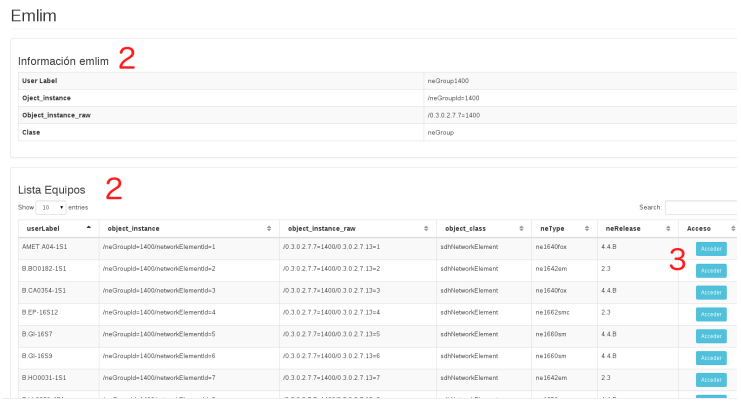

Fig. 57 Flujo normal Emlim

Flujo normal de eventos:

1. Acceso al Emlim (/gestor/{nombre_gestor}/Emlim/{object_instance_raw del Emlim}).
2. Se muestra la información y una lista de equipos a los que puede acceder.
3. Puede acceder a los equipos mostrados pulsando el botón "**Acceder**" de cualquiera de los elementos.
4. Si pulsa el botón del paso 3 el sistema redirige a la página donde se muestra la información del elemento.

Flujo alternativo:

- Búsqueda:
  - 2.1 El usuario puede filtrar la lista de equipos introduciendo en el buscador cualquier referencia de los atributos mostrados.
- Acceso terceras aplicaciones:
  - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar la información disponible a través de la API.

Postcondición: El usuario puede tener visualización del equipo pero si no tiene permisos a la información interna no podrá acceder.

5.3.6 Equipo

Actor: Usuario

Propósito: Se muestra la información del equipo al que se ha accedido y la lista de recursos que cuelgan del mismo.

Precondición: Asignar rol a usuario, según el rol del usuario tendrá restricciones sobre los equipos que quiere visualizar.

Flujo normal de eventos:

1. Acceso al equipo:
   /gestor/{nombre_gestor}/Emlim/{object_instance_raw del Emlim}/equipo/{object_instance_raw del equipo}
2. Se muestra la información y una lista de recursos que cuelgan del equipo(4 niveles por debajo) a los que puede acceder.
3. Puede acceder a los recursos mostrados pulsando el botón "**Acceder**" de cualquiera de los elementos.
4. Si pulsa el botón del paso 3 el sistema redirige a la página donde se muestra la información del elemento.

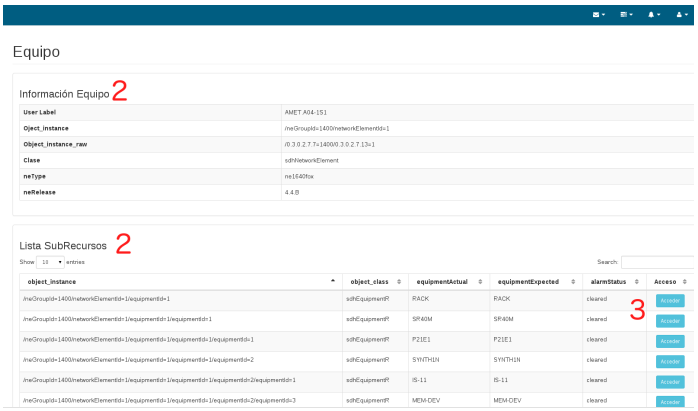

Fig. 58 Flujo normal equipo

Flujo alternativo:

- Búsqueda:
  - 2.1 El usuario puede filtrar la lista de recursos introduciendo en el buscador cualquier referencia de los atributos mostrados.
- Acceso terceras aplicaciones:
  - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar la información disponible a través de la API.

Postcondición: El usuario puede tener visualización los recursos pero si no tiene permisos a la información interna no podrá acceder.

5.3.7 Radio enlace

Actor: Usuario

Propósito: Se muestra la información del radio enlace(tipo concreto **equipo**) al que se ha accedido y la lista de recursos que cuelgan del mismo. A diferencia del equipo veremos los siguientes detalles:

- Frecuencia
- Potencia

Precondición: Asignar rol a usuario, según el rol del usuario tendrá restricciones sobre los radio enlaces que quiere visualizar.

Flujo normal de eventos:

1. Acceso al radio enlace: /gestor/{nombre_gestor}/Emlim/{object_instance_raw del Emlim}/equipo/{object_instance_raw del RE}
2. Se muestra la información y una lista de recursos que cuelgan del radio enlace, siempre que tenga disponibles.
3. Puede acceder a los recursos mostrados pulsando el botón "**Acceder**" de cualquiera de los elementos.
4. Si pulsa el botón del paso 3 el sistema redirige a la página donde se muestra la información del elemento.

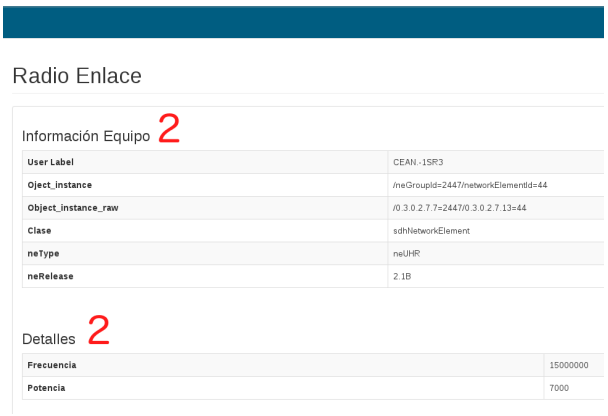

Fig. 59 Detalles radio enlace

Flujo alternativo:

- Búsqueda:
  - 2.1 El usuario puede filtrar la lista de recursos introduciendo en el buscador cualquier referencia de los atributos mostrados.
- Acceso terceras aplicaciones:
  - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar la información disponible a través de la API.

Postcondición: El usuario puede tener visualización los recursos pero si no tiene permisos a la información interna no podrá acceder.

5.3.8 Otros recursos

Actor: Usuario

Propósito: Se muestra la información del recurso que se especifique al que se ha accedido y la lista de subrecursos que cuelgan del mismo.

Precondición: Asignar rol a usuario, según el rol del usuario tendrá restricciones sobre los recursos que quiere visualizar. Se debe programar cada caso específico, la aplicación a medida que crezca irá añadiendo nuevos recursos específicos que deberán seguir este caso de uso.

Flujo normal de eventos:

1. Acceso al recurso:
   /gestor/{nombre_gestor}/Emlim/{object_instance_raw del Emlim}/equipo/{object_instance_raw del recurso}
2. Se muestra la información y una lista de subrecursos que cuelgan del recurso a los que puede acceder, siempre que tenga disponibles.
3. Puede acceder a los recursos mostrados pulsando el botón "**Acceder**" de cualquiera de los elementos.
4. Si pulsa el botón del paso 3 el sistema redirige a la página donde se muestra la información del elemento.

Flujo alternativo:

- Búsqueda:
  2.1 El usuario puede filtrar la lista de subrecursos introduciendo en el buscador cualquier referencia de los atributos mostrados.
- Acceso terceras aplicaciones:
  - Siempre y cuando tengan autorización, pueden lanzar una petición directa para visualizar la información disponible a través de la API.

Postcondición: El usuario puede tener visualización los subrecursos pero si no tiene permisos a la información interna no podrá acceder.

5.3.9 Conexión con el gestor

Actor: Aplicación

Propósito: Para recuperar cualquier recurso de un gestor se ha de conectar al mismo para poder lanzar las peticiones de nuestra aplicación.

Precondición: Debemos tener un acceso para la APICONSOLA en la consola CMIP.

Flujo normal de eventos:

1. Llamamos a la CMIP solicitando conexión a un gestor:
   /admin/templates/{nombre_gestor}
2. Si hay una conexión abierta nos devolverá la misma, si no crea una nueva.
3. Cuando disponemos de conexión ya podemos hacer cualquier llamada desde nuestra aplicación.

Flujo alternativo:

- Alternativas:
  2.1 Hay recursos que tienen dos gestores disponibles, el normal y el alternativo, debemos llamar a un proceso para determinar cuál es el que está funcionando en este momento para crear la conexión con uno o con otro.

Postcondición: Una vez hemos accedido podremos acceder a cualquier información del gestor.

```
/*Abrir conexión a SH*/
public Connection nuevaConectionGestor(String gestor) throws Exception{
    String url = cmipURL.concat("/connections");
    URL obj = new URL(url);
    HttpURLConnection con = (HttpURLConnection) obj.openConnection();
```

Fig. 60 Fragmento del código de conexión con gestor

5.3.10 Peticiones CMIP

Actor: Aplicación

Propósito: Para recuperar cualquier información de un recurso dispondremos de un clase con dos tipos de peticiones a la CMIP (GET y POST).

Precondición: Debemos tener un acceso para la APICONSOLA en la consola CMIP. Para consultar las peticiones que podemos/debemos realizar hay que mirar la **documentación(PRIVADA) de la CMIP de Telefónica**.

Flujo normal de eventos:

1. La petición puede ser GET o POST.
2. Si la petición es POST, podemos enviar la siguiente información:
    a. **Method**
    b. **Params** -> los atributos más usados dentro del mismo son:
        i. **cmip_header_raw**
        ii. **object_instance:** recurso CMIP que recuperamos.
        iii. **scope:** niveles que queremos recuperar.
        iv. **object_class:** tipo objetos.
        v. **attributes:** atributos que queremos recuperar.
3. Después de realizar la petición, la CMIP nos devuelve un JSON con el resultado.

Flujo alternativo:

- Si la petición es errónea o incompatible, la CMIP nos devolverá un error.

Postcondición: Una vez que nos devuelve la información debemos tratar el JSON para traducirlo a objetos de nuestra aplicación.

```
@Service
public class PeticionCMIPServiceImpl implements PeticionCMIPService {
    //Variables del fichero properties
    @Value("${cmip.url}")
    private String cmipURL;

    @Value("${cmip.peticionGET.user}")
    private String getUser;
    @Value("${cmip.peticionGET.passw}")
    private String getPassw;

    @Value("${cmip.peticionPOST.user}")
    private String postUser;
    @Value("${cmip.peticionPOST.passw}")
    private String postPassw;

    //Constructor
    public PeticionCMIPServiceImpl() throws Exception {}

    public String peticionGET(String url, String body) throws Exception{...}

    public String peticionPOST(String url, String body) throws Exception{...}
}
```

Fig. 61 Código peticiones CMIP

6. Desarrollo de la aplicación

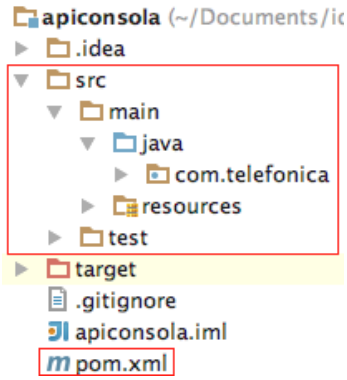

Fig. 62 Arquitectura del proyecto

En este capítulo explico cómo está estructurado el código y la función de los archivos del proyecto. Vemos la carpeta principal del proyecto(*main*) donde se encuentran los ficheros **Java**(codificación de la aplicación) y los **recursos**(ficheros estáticos y *templates*). Al ser un proyecto Spring vemos el archivo Maven **pom.xml**, donde asignamos todas las dependencias de nuestro proyecto(librerías que usamos).

Comenzaré explicando el fichero **pom.xml** y luego entraremos más en detalle sobre el código. En el fichero **Maven** de nuestro proyecto una de las partes importantes es que definimos que el ***parent*** del proyecto es de **Spring *Boot*:**

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.2.4.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Fig. 63 Dependencia *Parent* para Spring *Boot*

Vemos que el proyecto se lanza con la versión **Spring Boot Starter 1.2.4.RELEASE** y que estas librerías nos las proporciona Maven vía **org.springframework.boot.**

Otra cosa importante en este fichero es que debemos especificar la **versión de Java** que vamos a utilizar en el proyecto (los equipos de desarrollo y servidores deben tener instalada la misma versión que se asigna en el fichero):

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
</properties>
```

Fig. 64 Propiedades proyecto fichero POM

El resto de configuración son todas aquellas dependencias que usamos en nuestro proyecto, podemos ver ***Thymeleaf***(capa presentación), ***Security***(implementación de la seguridad de

Spring), ***Starter Web***(conjunto de clases para proyectos web), ***Tomcat***(dependencia para incrustar el Tomcat en el proyecto o definirlo como proporcionado por el servidor). Vemos otras dependencias que se van a implementar como el conector ***MySQL*** y la **unidad de test**.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Fig. 65 Dependencias del proyecto

Seguidamente voy a explicar la arquitectura del proyecto, donde como hemos dicho antes se divide en código Java, *templates*, ficheros estáticos y propiedades de la aplicación, definimos cada uno de ellos:

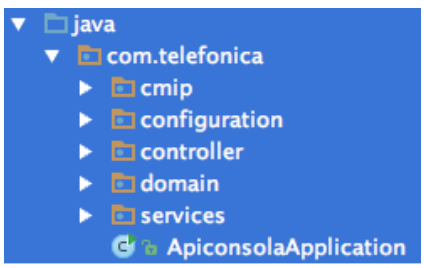

Fig. 66 Directorios Java del proyecto

- **Java**: donde se encuentra de codificación de la lógica de negocio de nuestra aplicación, y la resolución del problema planteado.
- ***Templates***: ficheros *HTML* con etiquetas *Thymeleaf* que van a generar nuestras vistas de la aplicación.

- **Ficheros estáticos:** lo componen los CSS, Javascripts, JQuery, imágenes, entre otros.
- **Propiedades aplicación**: donde definimos las propiedades del proyecto y es donde se diferencian los distintos entornos de nuestra aplicación.

La estructura de del proyecto está compuesta por los siguientes conjuntos de ficheros:
CMIP: Clases que controlan la conexión y comunicación con la consola CMIP.
Configuración: Clases Java donde cambiamos parámetros de configuración y seguridad de la aplicación.
Controladores: clases que reciben las peticiones de los clientes de la aplicación, se comunican con los servicios y devuelven una respuesta.
Dominio: clases principales de nuestra aplicación.
Servicios: los métodos de estas clases son los encargados crear acciones sobre la CMIP y tratar su información, mediante las correspondientes clases de comunicación.

Para entender el recorrido que se hace en la aplicación voy a poner un ejemplo, la recuperación de un gestor. Para situarnos, cuando el usuario de la aplicación pide la información de un gestor, se le proporcionan los valores del mismo y la lista de Emlims del gestor. En esta petición el usuario hace una petición a la URL **url_apiconsola/gestor/{nombre_gestor}** y el controlador ***GestorController*** es el que resuelve esta petición. Esta clase controlador tiene inyectados los servicios ***GestorService*** y ***EmlimService*** que van a proporcionar la información del gestor y el listado de Emlims que cuelgan del mismo. La inyección de dependencias la vemos identificada con la etiqueta ***@Autowired***, donde se hace referencia a una interfaz, pero en tiempo de ejecución se llama a las clases que implementan esta interfaz, en este caso ***GestorServiceImpl*** y ***EmlimServiceImpl***. Remarcar no siempre encontramos que un esta asociación, sino que también puede ser que varias clases implementan la interfaz y según el archivo de configuración, en tiempo de ejecución se use una u otra clase.

```
@Autowired
public void setGestorService(GestorService gestorService) {
    this.gestorService = gestorService;
}

@Autowired
public void setEmlimService(EmlimService emlimService) {
    this.emlimService = emlimService;
}
```

Fig. 67 Inyección de dependencias

Vemos que lo que marca que ese método va a resolver la petición es el "@RequestMapping(value="/gestor/{gestor}", method = RequestMethod.*GET*)" y el controlador lo que va a servir es un *JSON* con la información del gestor, si se trata de una petición de terceros, o en el caso de ser una petición con *Content-Type text/html* va a servir un ***ModelAndView*** con la información y la plantilla correspondiente.

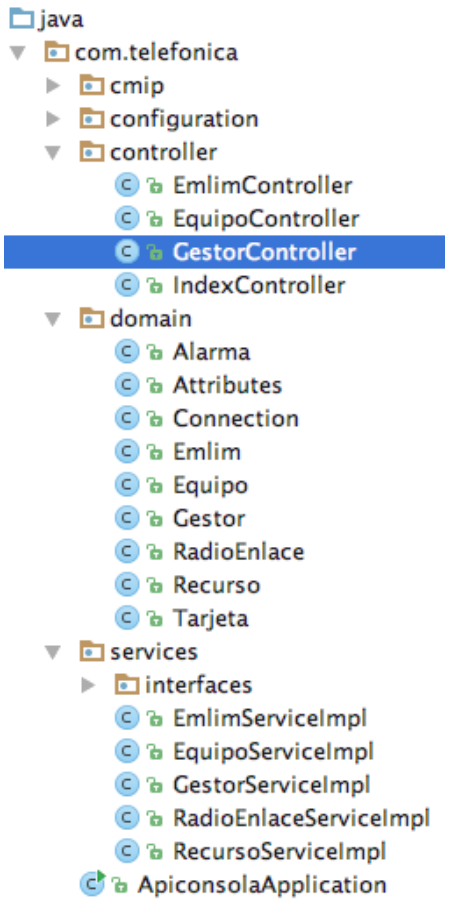

Fig. 68 Ficheros Java del proyecto

Analizando más a fondo la parte donde se devuelve el ***ModelAndView***, vemos que se añade la información del gestor cuando llamamos al método ***getGestorByName(gestor)*** del servicio ***gestorService*** y el listado de sus Emlims lo proporciona ***EmlimService.getEmlimsGestor(gestor, conexión_gestor)***. Por último añadimos la vista en el MAV.

En detalle, los servicios, estos se comunican con la **CMIP** para proporcionarnos el JSON de los datos que hemos solicitado y lo convierto en objetos a través de la librería **Jackson.**

```
/**
 * Este método devuelve el gestor y sus emlims.
 * @param contentType Según este parámetro la respuesta será JSON o en formato WEB
 * @param gestor Variable en el path que determina el gestor del cual queremos recuperar la información
 * @return Devuelve un JSON o el template + la información del gestor y sus emlims.
 */
@RequestMapping(value="/gestor/{gestor}", method = RequestMethod.GET)
public @ResponseBody Object conectarGestorNombre(@RequestHeader(value="Content-Type", required=false,
                                                  defaultValue="text/html") String contentType,
                                                  @PathVariable String gestor) throws Exception {
    if(contentType.equals("application/json")){
        return gestorService.getGestorByName(gestor);
    }else{
        ModelAndView mav = new ModelAndView();
        mav.addObject("gestor", gestorService.getGestorByName(gestor));
        mav.addObject("emlims", emlimService.getEmlimsGestor(gestor, gestorService.abrirConexionGestor(gestor)));
        mav.setViewName("gestor");
        return mav;
    }
}
```

Fig. 69 Método del controlador para recuperar un gestor

```
@Override
public Gestor getGestorByName(String gestor) throws Exception {
    ObjectMapper mapper = new ObjectMapper();

    return mapper.readValue(peticionCMIPServiceImpl.peticionGET("/admin/templates/"+gestor,
            "{\"template_name\":\"" + gestor + "\",\"name\":\"" + gestor +"\"}"), new TypeReference<Gestor>() {});
}
```

Fig. 70 Método del servicio para recuperar un gestor

Entro en detalles del ***EmlimService*** porque es una clase donde vemos una clara implementación de lo que he mencionado en capítulos anteriores, paralelismo, gracias a las a las nuevas funcionalidades de Java 1.8 (***parallel*** y ***lambdas***). Como para cada Emlim de un Gestor debemos conectarnos y hacer una petición independiente, lanza una función *lambda* en paralelo que recupera el Emlim y si este no es null lo añade a una lista.

```
@Override
public Iterable<Emlim> getEmlimsGestor(String gestor, Connection connection) throws Exception {
    List<Emlim> listaemlims = new Vector<>();

    //CON PARALELISMO usando
    IntStream.range(0,99).parallel().forEach(value -> {
        Emlim e = recuperarEmlim(value, gestor, connection.getUrl());
        if(e!=null)listaemlims.add(e);
    });

    //SIN PROCESOS PARALELOS
    /*for(int i=1; i<100; i++){
        Emlim e = recuperarEmlim(i, gestor, connection.getUrl());
        if(e!=null)listaemlims.add(e);
    }*/
    return listaemlims;
}
```

Fig. 71 Método del servicio para recuperar la lista de Emlims

Antes era una petición que tardaba unos 2 minutos y lanzando el mismo método con procesos paralelos se ha conseguido bajar a 15 segundos.

Finalmente volviendo al ***ModelAndView*** del método del controlador de este recorrido vemos que añadimos un *template* ***HTML*** con etiquetas ***Thymeleaf*** para dibujar la vista del gestor y listar los ***Emlims*** en el código ***HTML***.

```
<h3>Información gestor</h3>
<div class="table-responsive">
    <table class="table table-bordered table-striped">
        <tbody>
        <tr>
            <th>Address</th>
            <td th:text="${gestor.address}">-</td>
        </tr>
        <tr...>
        <tr...>
        <tr>
            <th>Description</th>
            <td th:text="${gestor.description}">-</td>
        </tr>
```

Fig. 72 Código Thymeleaf del gestor

```
<thead>
    <tr>
        <th>userLabel</th>
        <th>object_instance</th>
        <th>object_instance_raw</th>
        <th>object_class</th>
        <th>Acceso</th>
    </tr>
</thead>
<tbody>
    <tr class="odd gradeX" th:each="emlim : ${emlims}">
        <td th:text="${emlim.attributes.get('userLabel')}">userLabel</td>
        <td th:text="${emlim.object_instance}">object_instance</td>
        <td th:text="${emlim.object_instance_raw}">object_instance_raw</td>
        <td th:text="${emlim.object_class}">object_class</td>
        <td><center><a th:href="@{/gestor/} + ${gestor.name} + @{/emlim} +
        ${emlim.object_instance_raw} + @{/}" class="btn btn-info btn-sm"
                    role="button">Acceder</a></center></td>
    </tr>
</tbody>
```

Fig. 73 Código Thymeleaf de la lista de Emlims

Con las etiquetas **th** lo que hacemos un **bucle** en de **filas** y en cada **columna** con otro etiquetado **imprimimos** los valores de **cada atributo** del Emlim.

Finalmente decir que cada recorrido de nuestra aplicación es similar y vemos que el código gracias a un sistema MVC con servicios es muy ordenado y podemos seguir el código fácilmente.

7. Estudio económico

En este capítulo voy a detallar los costes económicos relacionados con el proyecto, hay que tener en cuenta que se ha realizado formando parte de la plantilla de Telefónica a través de una consultoría.

7.1. Coste de amortización

Coste de amortización del material técnico usado hasta ahora, para la puesta en marcha del núcleo de la aplicación.

Concepto	Precio	Vida útil	Tiempo utilizado	Amortización
Equipo informático	800€	48 meses	6 meses	100€
TOTAL				**100€**

Tabla 1 Costes de amortización

*El puesto de trabajo es un precio que tiene acordado Telefónica con la empresa consultora, esta última paga por cada persona un x mensual para que cada trabajador tenga en las oficinas de trabajo de Telefónica un puesto.

El coste total de amortización es **100€.**

7.2. Costes directos

En este apartado se detallan los gastos relacionados con la creación del producto. Se valora el contrato hasta la fecha actual, los 6 meses trabajados en el proyecto.

Concepto	Anual	Coste
Nómina jornada completa	24.000€*	(24.000/12)*6
TOTAL		**12.000€***

Tabla 2 Costes directos

*A estos costes falta añadir lo que la empresa paga de más por el trabajador, se estima un 30%, esto es igual a **31.200€, en estos 6 meses 15.600€.**

7.3. Costes indirectos

Son costes que afectan en el desarrollo del producto, en mi caso, al estar en "Casa del cliente", mi empresa paga a Telefónica un alquiler por el puesto de trabajo donde ya entran todos los servicios.

Concepto	Precio	Tiempo utilizado	Amortización
Puesto trabajo*	500€/mes	6 meses	3.000€
TOTAL			**3.000€**

Tabla 3 Costes indirectos

7.4. Costes totales

Concepto	Precio
Costes amortización	100€
Costes directos	15.600€
Costes indirectos	3.000€
TOTAL	**18.600€**

Tabla 4 Costes totales

El coste total aproximado de la empresa en estos 6 meses es de **18.600€**.

Si hablamos del cliente, Telefónica, el coste del proyecto para ellos es el coste de la empresa consultora más el margen de beneficio que aplican por cada trabajador. También añadir que los servidores los proporciona Telefónica y es un coste que asumen ellos internamente.

8. Conclusiones

Para acabar podemos decir que el resultado del proyecto es adecuado, ya que hemos conseguido una estructura sólida para poder seguir trabajando y añadiendo nuevos requerimientos especificados por **Telefónica**. Actualmente está en proceso de desarrollo/certificación, para así poder disponer de una primera versión en el entorno de producción.

El proyecto es completamente funcional y podemos ver que tenemos una mejor accesibilidad a la consola **CMIP** y donde se han reducido los tiempos de acceso a los recursos que queremos consultar o realizar acciones sobre ellos.

Debido a la evolución de las tecnologías móviles hemos preparado nuestra aplicación para que pueda ser accesible desde estos dispositivos, para así poder ampliar el uso de nuestra aplicación desde puntos donde los operadores no disponen de ordenadores.

Usar un entorno de desarrollo como **IntelliJ IDEA** proporciona ayudas al programador que mejoran y disminuyen los tiempos de programación de la aplicación. El tener una herramienta con todo integrado tenemos un control de nuestro software en la misma pantalla, sin necesidad de tener que abrir otros programas, por ejemplo una herramienta **GIT** para el control de código.

Para mejorar el esqueleto de nuestra aplicación, se ha de implementar:

- Control de usuarios y roles más complejo, persistido en BBDD, que hasta el momento no estaba especificado.
- **Persistir en base de datos** aquellas **acciones** importantes que realizan los usuarios mediante la **APICONSOLA**.
- Implementar **nuevos requerimientos** definidos por **Telefónica**.

Al trabajar con tecnologías de última generación podemos asegurar una evolución de la aplicación y gracias a los servicios la tenemos dividida y poco acoplada, para poder asignarlos en distintos servidores según la carga de trabajo.

9. Referencias

[1] https://www.jetbrains.com/idea/

[2] http://projects.spring.io/spring-boot/

[3] http://courses.springframework.guru/

[4] http://www.thymeleaf.org/

[5] http://startbootstrap.com/template-overviews/sb-admin-2/

[6] https://es.atlassian.com/software/

[7] https://Maven.apache.org/pom.html

[8] http://www.redhat.com/es/global/spain

[9] https://www.centos.org/

[10] https://es.wikipedia.org/wiki/Integraci%C3%B3n_continua

[11] http://acodigo.blogspot.com.es/2014/11/spring-boot-introduccion.html

[12] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[13] https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html