

# EdOz



## Servidor REST y Cliente Web



**I.E.S. ABASTOS**  
Memoria del Proyecto de  
Desarrollo de Aplicaciones Multiplataforma  
**Lucas Cerveró Beltrán**  
Curso 2019/20 Grupo 7U  
Tutor: Pau Villanueva  
17 de Marzo de 2020

## **AGRADECIMIENTOS**

A mi familia, por su apoyo y ayuda durante la realización de las prácticas y el proyecto. A mi tutor de prácticas, Vicente Pons por ayudarme y ser tan paciente conmigo. A mi tutor de proyecto, Pau Villanueva por aguantarme, guiarme y responderme los correos hasta los fines de semana. A mis compañeros de trabajo, Paco Bonillo, Francis Llobell y Cristina Llosa por cambiarme los turnos que necesité para completar las prácticas.

## **LICENCIAS**

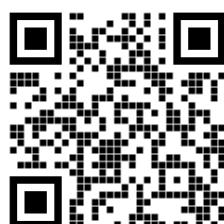
Este proyecto tiene licencia GNU FDL. Puede consultar la licencia al completo en el siguiente [enlace](#). El propósito de esta Licencia es permitir que un manual, libro de texto, u otro documento escrito sea *libre* en el sentido de libertad. Asegurar a todo el mundo la libertad efectiva de copiarlo y redistribuirlo, con o sin modificaciones, de manera comercial o no. En segundo término, esta Licencia proporciona al autor y al editor una manera de obtener reconocimiento por su trabajo, sin que se le considere responsable de las modificaciones realizadas por otros.

Esta Licencia es de tipo *copyleft*, lo que significa que los trabajos derivados del documento deben a su vez ser libres en el mismo sentido. Complementa la Licencia Pública General de GNU, que es una licencia tipo copyleft diseñada para el software libre.

Un programa libre debe venir con manuales que ofrezcan la mismas libertades que el software. Pero esta licencia no se limita a manuales de software. Puede usarse para cualquier texto, sin tener en cuenta su temática o si se publica como libro impreso o no. Se recomienda esta licencia principalmente para trabajos cuyo fin sea instructivo o de referencia.

## **PRESENTACIÓN**

El siguiente código QR enlaza con las diapositivas que se utilizaron en la presentación del proyecto. Puede usarse un lector de códigos en el móvil o visitar la siguiente dirección al pie.



<https://llucbrell.github.io/proyecto-dam/>

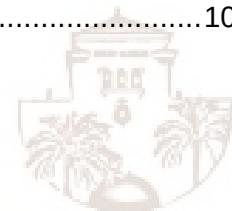
# Índice de Contenidos

1. Introducción.....	4
1.1. Introducción.....	4
1.2. Identificación.....	4
1.3. Viabilidad del Proyecto.....	4
1.4. Objetivos del Proyecto.....	4
1.4.1. Objetivos del Servidor.....	4
1.4.2. Objetivos del Cliente.....	4
1.4.3. Conclusiones Finales.....	5
2. Diseño del proyecto.....	6
2.1. Diseño Conceptual.....	6
2.1.1. Nombre de la aplicación.....	6
2.1.2. Iconos para la aplicación.....	6
2.2. Diseño de la interfaz gráfica.....	6
2.2.1. Interfaces del usuario de docencia o administrador.....	7
2.2.2. Interfaces del usuario del estudiante.....	8
2.3. Diseño del modelo de datos.....	10
2.3.1. Visión general del modelo.....	10
2.3.2. Estructura de la Base de Datos y diseño lógico.....	10
2.4. Introducción a REST.....	13
2.4.1. Acciones básicas de REST.....	13
2.5. Diseño del A.P.I.....	15
2.5.1. Análisis de Dominio o descripción de la API.....	15
2.5.2. Análisis de los requisitos de la Arquitectura.....	16
2.5.3. Diseño básico de la Arquitectura.....	16
2.5.4. Herramientas para el diseño y documentación del API.....	17
2.5.5. Conceptos básicos de Open Api.....	17
2.5.6. Publicación del API usando Spectacle.....	18
2.5.7. Desarrollo del API en el proyecto.....	18
2.6. Diseño y Estructura del Servidor.....	19
2.7. Estructura de paquetes y clases.....	19
2.8. Análisis de las herramientas.....	20
3. Desarrollo del proyecto.....	21
3.1. Implementación del modelo de la B.B.D.D.....	21
3.1.1. Scripts de creación de la estructura de la base de datos.....	27
3.2. Implementación de la arquitectura REST.....	28
3.2.1. ¿Qué es JAX-RS?.....	28
3.2.2. ¿Qué es Jersey?.....	29
3.2.3. La representación de los datos cliente-servidor.....	30
3.2.4. Librerías Java para serializar JSON.....	30
3.2.5. Google Gson.....	30
3.3. Implementar herramientas en el servidor.....	31

3.3.1. El log de la aplicación.....	31
3.3.2. Para qué utilizar los logs.....	31
3.3.3. Tecnologías.....	31
3.4. ¿Qué es DAO?.....	32
3.4.1. ¿Cómo implementar un DAO?.....	32
3.5. Implementación básica de Tomcat y servicio REST.....	33
3.5.1. Añadiendo las librerías del framework Jersey.....	35
3.5.2. Código de implementación básica con Tomcat y Jersey.....	36
3.5.3. Comprobación básica del funcionamiento de la aplicación.....	38
3.5.4. Alternativas de testeo de la aplicación.....	38
3.6. Login y Seguridad en la Aplicación.....	39
3.6.1. Login con el Directorio Activo.....	39
3.6.2. Asegurar nodos o “endpoints” en REST.....	41
3.6.3. Trampas, HoneyPots en el login y los formularios.....	41
3.7. Implementación del cliente Web.....	43
3.7.1. Angular.....	43
3.7.2. Angular CLI.....	43
3.7.3. NodeJS.....	43
3.7.4. NPM.....	43
3.7.5. TSC.....	43
3.7.6. WebPack.....	43
3.7.7. Editor de texto.....	43
3.7.1. Ejemplo de creación de un cliente con Angular 2.....	44
3.7.1.1. Crear la aplicación.....	44
3.7.1.1.1. Desarrollar el código.....	44
3.7.1.2. Testear la aplicación.....	45
3.7.1.3. Construir la aplicación.....	47
3.7.1.4. Desplegar la aplicación cliente.....	47
3.7.1.1.1. Construir el Servlet.....	48
4. Evaluación y conclusiones finales.....	49
4.1. Evaluación.....	49
4.1.1. Análisis del Servidor.....	49
4.1.2. Análisis del Cliente.....	49
4.1.3. Posibles mejoras.....	49
4.2. Conclusiones finales.....	50
Bibliografía.....	51
Índice de Figuras.....	52
A. Anexo de NodeJs.....	54
i. NodeJS nociones básicas.....	54
ii. Instalación de NodeJS en Unix.....	54
iii. Hola mundo en NodeJS.....	55
iv. El API de NodeJS.....	56
v. El concepto de módulos de NodeJS.....	56
B. Anexo de npm.....	58



i. Instalación de npm en UNIX.....	58
ii. Creación de un proyecto npm.....	58
iii. Fundamentos de NPM.....	59
iv. Automatizando con NPM.....	61
v. ¿Por qué usar NPM?.....	61
C. Anexo de TypeScript.....	62
i. TypeScript y TSC.....	62
ii. Instalación del compilador.....	62
iii. Hola mundo con TypeScript.....	62
iv. Variables y TypeScript.....	63
v. Funciones flecha.....	65
vi. Interfaces de TypeScript.....	65
vii. Clases en TypeScript.....	66
viii. Modulos en TypeScript.....	68
D. Anexo de Angular 2.....	70
i. Fundamentos de Angular.....	70
ii. Versiones del Framework.....	70
iii. Arquitectura de Angular.....	70
iv. Estructura de un proyecto Angular.....	71
a. Estructura de ficheros.....	71
b. Creación de componentes.....	73
c. Templates y renderizado dinámico.....	75
d. El DOM del navegador y las Directivas de Angular.....	76
e. Módulo HttpClient.....	77
f. Más sobre Angular.....	79
E. Anexo de Swagger.....	80
i. Publicación del API usando Swagger y openAPI.....	80
F. Anexo de Jersey.....	83
i. Anotaciones de Java con Jersey.....	83
ii. Manejo de respuestas.....	87
iii. Manejo de cabeceras.....	87
G. Anexo de figuras.....	89
i. Imágenes varias.....	89
ii. Imágenes extra para explicaciones del modelo.....	90
iii. Imágenes extra del servidor tomcat y eclipse.....	92
iv. Imágenes extra del cliente.....	94
v. Imágenes extra de envío de peticiones.....	96
H. anexo de ejemplos de código.....	97
i. Ejemplos de código de anotaciones.....	97
ii. Ejemplos de configuración de recursos JNDI.....	101
iii. Configuración Log4J.....	104
iv. Códigos alternativos de Angular-cli.....	105



# 1. INTRODUCCIÓN

## 1.1. Introducción

El presente documento intentará documentar y especificar los pasos seguidos para la realización del proyecto de editor de cursos y jornadas para docencia, ideado por el departamento de Informática del Hospital Universitario La Fe.

## 1.2. Identificación

El departamento de Docencia del Hospital Universitario La Fe, propuso al departamento de Sistemas de la Información del mismo centro, la implantación de una aplicación que controlara el número de asistentes a conferencias, jornadas y cursos, además de permitir realizar encuestas a los mismos, a fin de obtener datos y estadísticas sobre los mismos.

Tras realizar un primer intento de instalación de una herramienta de código privado un tanto tosca, y tras comprobar la no viabilidad de otros sistemas abiertos, la gerencia de Sistemas de Información decidió crear una aplicación que realizara dicha tarea. Dicho proyecto, tras una reunión fue encomendado al presente alumno, guiado por su tutor de prácticas Vicente Pons.

## 1.3. Viabilidad del Proyecto

El presente proyecto tiene una viabilidad alta. Es así, debido a la gran cantidad de editores y otros tipos de aplicaciones con funcionalidades muy similares. También se piensa que cubre esta aplicación una necesidad importante, puesto que en dicho hospital se celebran numerosas conferencias, jornadas y cursos por ser el hospital de referencia de la Comunidad y no poseer de una aplicación para la ya comentada gestión.

## 1.4. Objetivos del Proyecto

La aplicación desarrollada según gerencia, debería de cumplir con una serie de requisitos para ser apta. Se planteó el proyecto en dos partes. Una la relativa a los requisitos y objetivos del servidor y otra que hiciera lo propio con la parte del cliente.

### 1.4.1. Objetivos del Servidor

El servidor debería cumplir los requisitos más difíciles. Aunque en principio no había una limitación importante a la hora de utilizar nuevas tecnologías como Spring o incluso Node. La realidad en entornos de producción, imprime una necesidad de aportar cierta mantenibilidad y seguridad probada por el equipo de ingenieros del Hospital. Esto crea, como se verá más adelante en el proyecto una serie de necesidades y obligaciones a la hora de elegir entre tecnologías y lenguajes.

### 1.4.2. Objetivos del Cliente

Al ser una herramienta orientada exclusivamente en uso a los usuarios del hospital, debería ser accesible desde la intranet del mismo, por tanto debería ser una aplicación cliente-servidor que usara algún tipo de lenguaje web en el cliente. Esta decisión venía marcada por las limitaciones a la hora de usar tecnologías por estar la intranet del hospital muy limitada dada la enorme importancia de la seguridad y la dificultad de instalación de nuevo software.

La aplicación también debería utilizar algún tipo de framework que aportara, seguridad, estabilidad, reusabilidad y estructurara la aplicación a fin de ser mantenida por otro/s



desarrolladores del hospital. Se realizó una reunión con el tutor y se expusieron una serie de opciones y posibilidades en cuanto a la elección de framework web para el cliente.

#### **1.4.3. Conclusiones Finales**

Finalmente se realizaron una serie de reuniones y una presentación que trataba de arrojar algo de luz y orientar al estudiante en la dirección idónea para la realización del proyecto.

Dicha presentación se puede consultar en el siguiente [enlace](#).

Tras mucho indagar, y conversar acerca del tema, se llegó a la conclusión de utilizar un servidor REST, utilizando la especificación JAX-RS de Java y la implementación<sup>1</sup> JERSEY por parte del servidor.

Para la parte del cliente, se debería desarrollar con TypeScript y el framework Angular.

---

<sup>1</sup> También podría denominarse Framework



## 2. DISEÑO DEL PROYECTO

### 2.1. Diseño Conceptual

En este apartado se trata el diseño conceptual previo a la creación del programa.

#### 2.1.1. Nombre de la aplicación

A veces resulta complicado encontrar un nombre para el programa que se está desarrollando. No obstante, se siguieron una serie de técnicas a fin de crear baterías de nombres para proporcionar variabilidad y conseguir así cumplir con los objetivos de esta sección. Se utilizaron técnicas como el "brainstorming" y otras, la relación y semejanza de conceptos y la asociación libre de ideas<sup>2</sup>. Aunque la tarea resultó ser algo más difícil de lo esperado, a continuación se muestran los que finalmente fueron seleccionados.

*Jornadero, Jactos, EduFe, EduJ, DozEdu, SeminarIOs, EdEd – Editor de Educación,  
LaFedEd – La Fe, Editor, de Educación, EdOz*

Tras varios análisis y discusiones se seleccionó el último, EdOz, como nombre comercial de la herramienta, por su facilidad de asociación entre Editor y Docencia. Aunque como se verá más adelante, el cambio de nombre siempre puede realizarse de manera muy sencilla simplemente alterando el código en el cliente.

#### 2.1.2. Iconos para la aplicación

También se crearon una serie de iconos que podrían servir para la aplicación. Se buscó algo del estilo "Flat design" por su simplicidad y facilidad para la creación de diseños. Para el resto de iconografía de la aplicación, se optó por dos opciones principalmente. Se podría utilizar [Material Design](#) de Google, por ser libre y fácilmente adaptable al framework del cliente que se escogió, o como finalmente se decidió, aplicar una iconografía llamada [Line Awesome](#). Esta se acabó eligiendo por ser lo más sencillo de implementar, cosa que para el proyecto elegido significaría una clara ventaja.

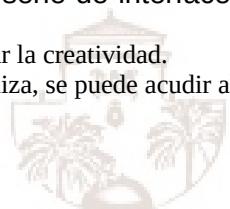


### 2.2. Diseño de la interfaz gráfica

Para el proyecto, una vez se decidió el framework a utilizar y en función del conocimiento de la estructura, organización y funcionamiento del mismo<sup>3</sup>, se diseñan una serie de interfaces

2 Estas son técnicas utilizadas para ayudar a la generación espontánea de ideas y mejorar la creatividad.

3 Se escogió el framework Angular 2 y para saber más sobre cómo se estructura y organiza, se puede acudir al Anexo D del presente proyecto.



de usuario para el manejo del programa. Como implícito en el concepto del programa hay varios tipos de usuario, debemos por tanto crear también varias interfaces para responder las necesidades particulares de cada uno de ellos. A continuación se definen las interfaces más importantes del programa.

### **2.2.1. Interfaces del usuario de docencia o administrador**

El usuario de docencia es el administrador del programa. Tiene la potestad de crear, editar y borrar los datos de los cursos o jornadas. Se puede ver aquí, el diseño de la interfaz principal del editor de cursos. Este tipo de usuario, debe además poder ver los diferentes usuarios inscritos en los cursos, las estadísticas de las respuestas a los cuestionarios y los cursos que estén públicos.

Nombre de la app sirve para ir al inicio (link)

Imagen del curso

Editable al pulsar

Controles de opciones del campo personalizable

Nombre de la App

Datos de dónde y como se desarrolla el curso

Nombre del Curso

Nombre del Curso

Descripción

Aula Salón de Actores

Plazas 225

Profesor Anacleto Careto

Pregunta o cuestionario A

Option 1 Option 2 Option 3

Pregunta o cuestionario B Pregunta o cuestionario B

Opción 1 Opción 2 Opción 3 Opción 4

Pregunta o cuestionario C

Option 1 Option 2 Option 3 Option 4

Pregunta o cuestionario D

Option 1 Option 2 Otro ...

Borra el campo

Crea un nuevo campo del cuestionario

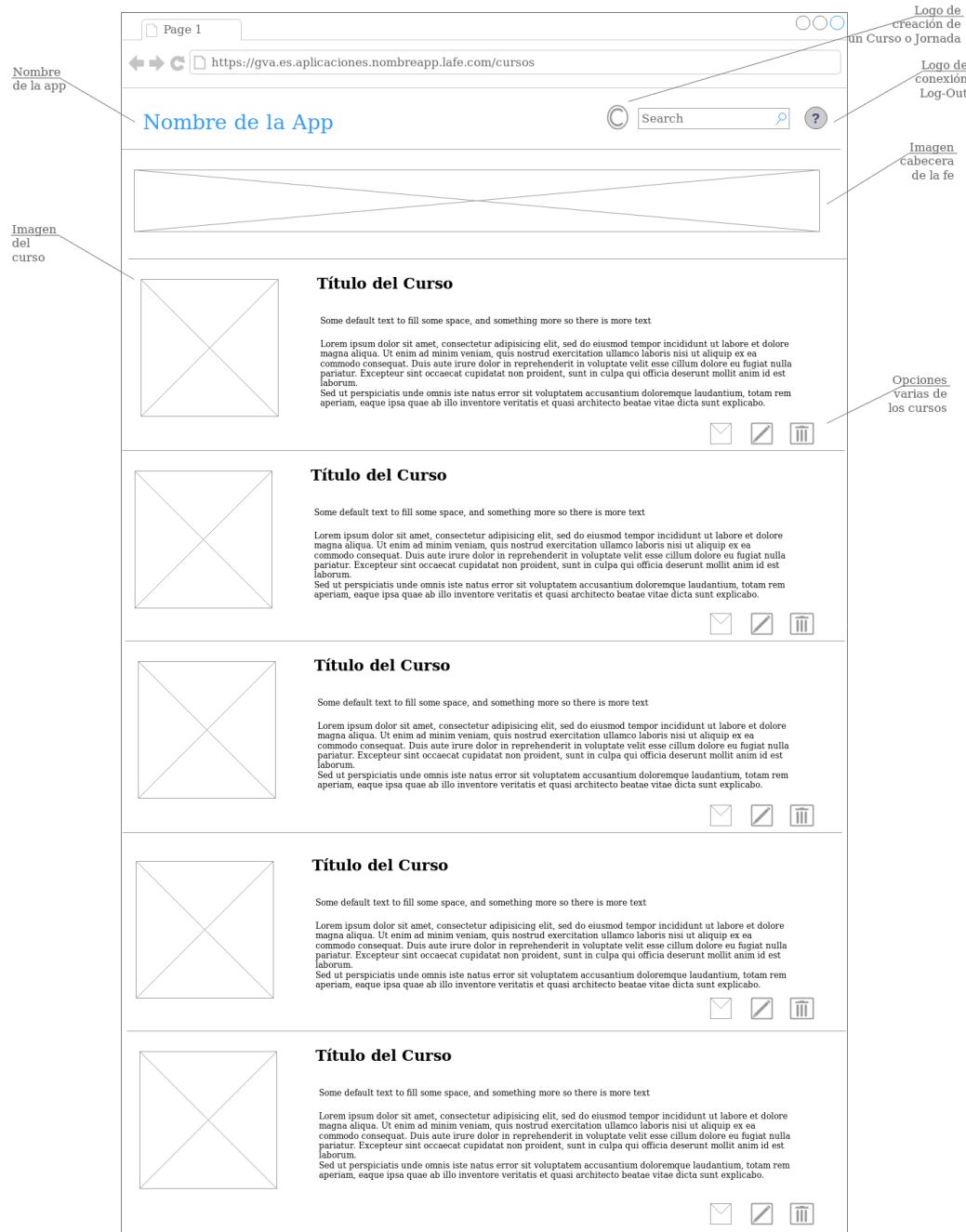
Cancelar Guardar

Errores de comprobación del formulario

**Figura 1: Diseño de interfaz de usuario para la edición de cursos**



El usuario de docencia, podrá administrar los cursos, borrando, editando o creando nuevos cursos a voluntad, como se observa en la siguiente imagen.



**Figura 2: Diseño de interfaz de usuario básica de gestión de cursos.**

### **2.2.2. Interfaces del usuario del estudiante**

Este tipo de usuario puede visualizar los cursos y apuntarse a los mismos en unos pocos pasos. No tiene la potestad de alterar los cursos como en la fig 1 pero si puede responder a las cuestiones y apuntarse a los mismos.



Nombre de la app sirve para ir al inicio (link)

Nombre de la App

Logo de conexión Log-In

Imagen cabecera de la fe

Datos opcionales

Descripción del curso

Imagen del curso

Título del Curso

Datos de dónde y como se desarrolla el curso

Cuestionario

Nombre

Apellidos

Correo electrónico

DNI

Pregunta o cuestionario A

Pregunta o cuestionario B

Pregunta o cuestionario C

Pregunta o cuestionario D

Inscribirse

Errors de comprobación del formulario

Mo	Tu	We	Th	Fr	Sa	Su
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4
5	6	7	8	9	10	11

**Figura 3:** Diseño de interfaz de usuario para la inscripción al curso o jornada

Como se observa, varios de los interfaces son similares, con ligeras variantes, este tipo de diseño se ha elegido por su sencillez y por ser de los que generalmente observamos en la mayoría de aplicaciones web del mercado. Un ejemplo de esto es la figura 2, en la que el estudiante visualizaría el listado de cursos con menor número de opciones para el curso que el administrador, que goza de plenos poderes para los mismos. Esta interfaz, ocultaría los iconos de creación de curso, de edición o eliminación de curso manteniendo las opciones de inscribirse y borrarse del curso, pero la estructura y los bloques constructivos serán los mismos.



El diseño de interfaces queda en última instancia supeditado a las necesidades de la arquitectura y de las tecnologías usadas. Así pues, puede que se realicen cambios en la misma, pero los fundamentos se mantendrán.

### **2.3. Diseño del modelo de datos**

Para el diseño del modelo de datos se siguieron las metodologías y apuntes de diversas fuentes, con el fin de realizar la tarea de la manera más correcta y metódica posible. Para consultar las referencias usadas, por favor acude a la sección de la Bibliografía.

#### **2.3.1. Visión general del modelo**

Siguiendo la línea marcada por las especificaciones, el modelo de datos se tiene que realizar usando una base de datos relacional.

El modelo se podría entender de manera muy sencilla como una relación entre dos objetos. Un objeto sería la abstracción “usuario” y el otro la que corresponde a “curso”. Aunque este modelo (fig. 5) conceptualmente es válido, en la práctica se queda muy limitado y complica la adición de otras abstracciones y relaciones más complicadas. Así, lo más lógico sería moverse hacia un modelo más explícito y cercano a la realidad. Es decir, que represente más fielmente lo que es la realidad (fig. 4). Esto deriva también en un modelo más sencillo de implementar.

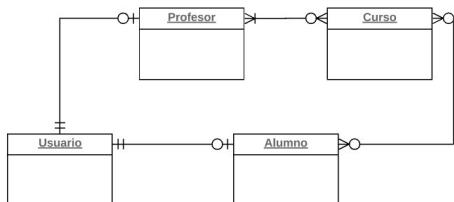


Figura 4: Vista general estrategia a 4

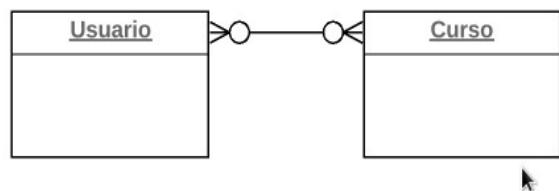


Figura 5: Vista general estrategia a 2

No obstante, esta aproximación es sólo conceptual. En el siguiente apartado se explicarán con mayor detalle el modelo. Se realizará un diseño lógico de la estructura de la base de datos y se explicarán sus ventajas e inconvenientes.

#### **2.3.2. Estructura de la Base de Datos y diseño lógico**

El diseño lógico es lo más complicado de realizar. Es aquí donde se encuentran posibles fallos y puntos muertos de concepto. Si se desarrollara la base de datos siguiendo el concepto mostrado en la figura 5 encontraríamos enormes dificultades para desarrollar el editor. Es decir, no encontraríamos forma posible de que el usuario pudiera incluir tipos de controles al estilo de la web de [encuestas](#). Se puede apreciar la limitación en la figura 56 y cómo sería necesaria la inclusión de más tablas en el diseño.

En cambio, el diseño lógico más explícito nos proporciona una notable mejora. Se puede por ejemplo usar un diseño de 3, 4 o más tablas. Así, como se ve en la fig. 57 la inclusión de estas nuevas tablas nos aporta un mayor control sobre los usuarios y sus roles. Además también se puede mejorar las relaciones entre los cursos y los usuarios, haciendo más lógica y comprensible las relaciones entre cursos, usuarios y profesores quedando un diseño como el de la figura 6.



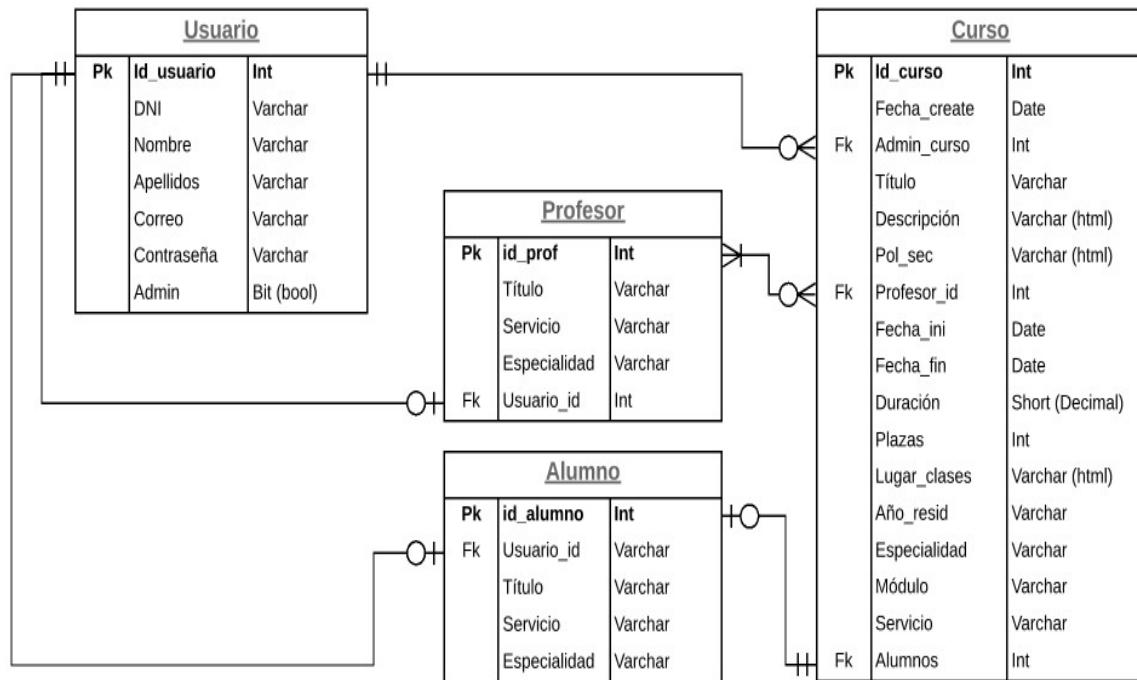


Figura 6: Modelo de datos usando un acercamiento a 4

A pesar de que ahora, podríamos decir que se cuenta con un diseño lógico cercano a la realidad, es necesario añadir una capa extra de complejidad para poder llevar a cabo los requisitos marcados por las especificaciones. Es por tanto necesario analizar la forma en que se almacenarán los datos en la base de datos. Estos datos son fundamentales a la hora de crear las funcionalidades que integran la utilidad principal de la aplicación, lo que realmente hace que la aplicación sea útil para los usuarios de docencia.

Como el editor pretende dejar que sea el usuario quien incluya partes de la interfaz gráfica (radio-buttons, checklists, listas-desplegables, etc.,) Se deberá tomar la decisión trascendental de separar o no la interfaz gráfica de la información contenida en las mismas. De esta manera, se podrían llevar a cabo dos aproximaciones diferenciadas.

Se podría pensar en un cliente web que genere los componentes personalizables utilizando html y los envíe al servidor, que los almacenaría en una columna de la tabla, como se aprecia en la imagen 7. Así, al realizar las peticiones al servidor, el cliente web solamente tendría que pintarlas en el lugar correspondiente.

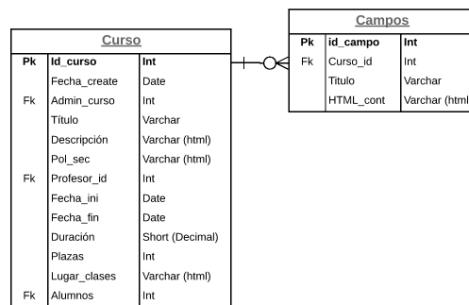


Figura 7: Posible aproximación con almacenamiento de html en crudo

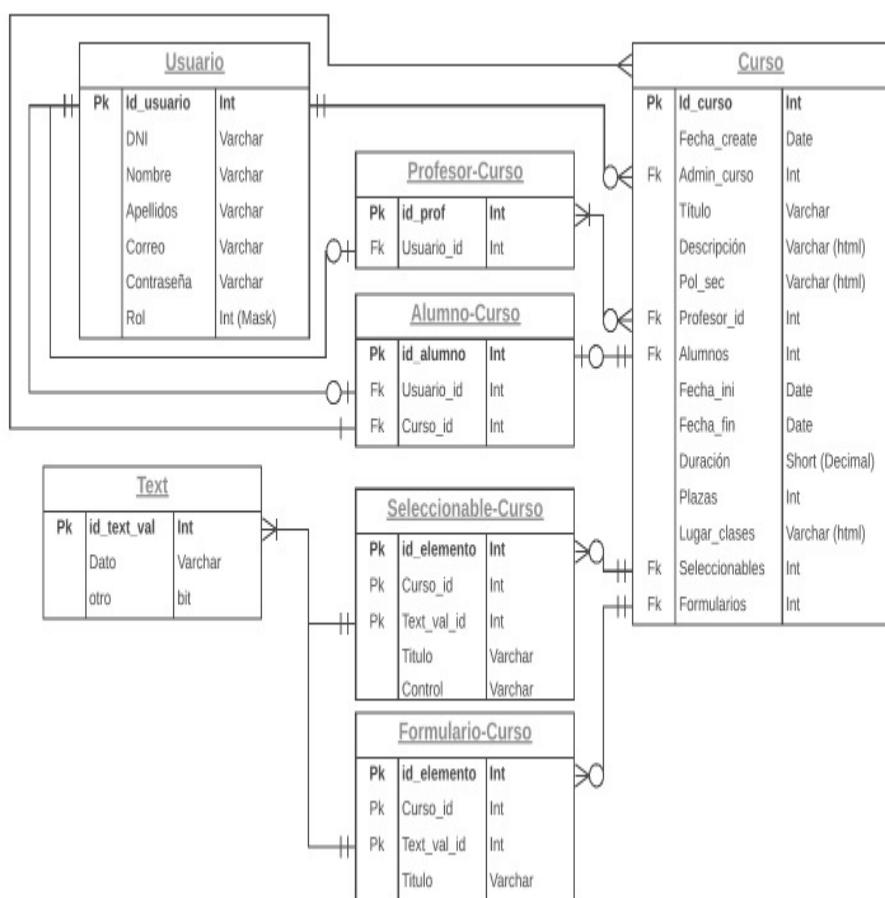


De esta forma, es la base de datos la que almacena todos los datos en crudo. Una de las principales ventajas de esta aproximación es que resulta muy cómoda a la hora de generar y visualizar el contenido. No obstante, esta no es la forma idónea a la hora de llevar a cabo la aplicación. Los inconvenientes son la falta de seguridad<sup>4</sup>, la imposibilidad de extraer datos de calidad de estos campos<sup>5</sup> y genera dependencia del servidor hacia el cliente<sup>6</sup>.

La otra estrategia que se podría seguir es una que separe la interfaz de los datos. Es decir, la base de datos solamente guarda los datos y es el cliente quien genera las partes de manera independiente. Para ello, la base de datos tiene que almacenar palabras clave, a modo de API, y es el cliente el encargado de implementar la interfaz con su correspondiente tecnología, ya sea con html, javaFX, etc.

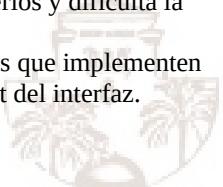
Esta perspectiva hace posible mejoras futuras y separa los datos de la interfaz de usuario, haciendo que el conjunto sea más claro y simple. Se considera así una mejor forma de proceder a la hora de alcanzar los requisitos exigidos. Además en un futuro se podrían llegar a separar por tipos de datos, haciendo posible comprobaciones y testeos de mejor calidad.

Así, el diseño final del modelo se queda de la siguiente manera (fig. 8).



**Figura 8: Diseño lógico final**

- 4 Se podría introducir código javascript que luego se podría ejecutar en el navegador de otro usuario.
- 5 Al encontrarse los datos mezclados con el código de la interfaz resulta laborioso extraerlos y dificulta la obtención de datos para la generación de estadísticas.
- 6 Al guardar el código de la interfaz, si se diera la ocasión de que hubieran varios clientes que implementen diferentes tecnologías, alguna de ellas podría no ser compatible con el html y javascript del interfaz.



## 2.4. Introducción a REST

REST es un patrón de arquitectura cliente-servidor ideado por Roy Fielding y explicado en su Tesis doctoral. Se utiliza para definir un patrón de implementación de sistemas en red y/o para definir sistemas distribuidos. Sus siglas corresponden con “Representational State Transfer” siendo uno de los patrones más utilizados actualmente.

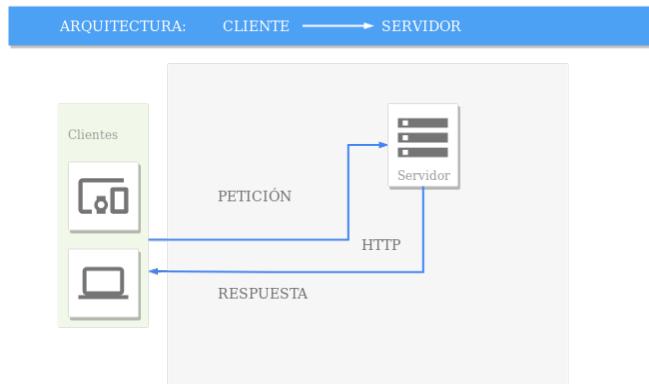


Figura 9: Estructura cliente-servidor

### 2.4.1. Acciones básicas de REST

REST se basa en el protocolo http para funcionar, pero no utiliza todos los métodos de este protocolo. Los métodos http más conocidos son:

Método http	Acción
GET	Ler o conseguir un recurso.
POST	Crea un recurso
PUT	Actualiza un recurso.
DELETE	Borrar un recurso.
PATCH	Actualiza sólo una parte de un recurso.
HEAD	Obtener las cabeceras de la petición.
OPTIONS	Especificar al servidor las opciones que se usarán.
...	...

REST se fundamenta principalmente en 4 acciones. Estas acciones forman el acrónimo conocido como CRUD (crear, leer, actualizar y borrar<sup>8</sup>).

Método http	Acción
GET	Ler un recurso
POST	Crear un recurso
PUT	Actualizar un recurso
DELETE	Borrar un recurso

<sup>7</sup> Hay más métodos http como TRACE o CONNECT, pero son importantes cuando implementamos un api basado en REST.

<sup>8</sup> Acrónimo inglés CRUD (create, read, update, delete).



La arquitectura REST está orientada a los recursos de base. Es decir, incentiva a los desarrolladores a usar los métodos http de manera consistente con la definición del protocolo. Cada recurso se identifica por una URL y cada uno de ellos debería soportar las peticiones principales de la anterior tabla. REST también permite tener diferentes representaciones del recurso, es decir el mismo recurso puede estar en texto, en xml, en json, etc., y el cliente puede realizar una petición de una en particular.

Los tipos de datos que se usan en REST son:

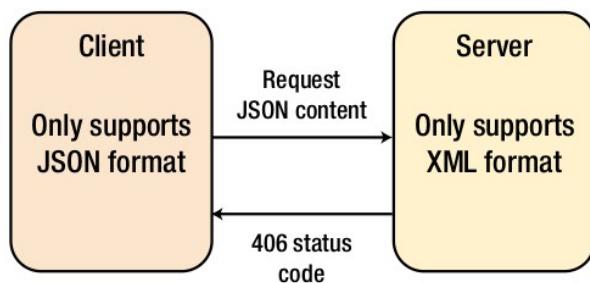
Tipo de elemento	Descripción
Recurso	Es el concepto REST.
Identificador de recurso	Es la ruta específica para alcanzar el recurso.
Metadatos del recurso	Información del api sobre el recurso.
Representación	Es el recurso en sí, el archivo en sí (una imagen, un vídeo, etc.,)
Metadatos de la representación	Datos sobre el tipo de archivo y cómo tratar con él.
Datos de control	Información que describe como optimizar el proceso de respuesta

Si combinamos las dos tablas anteriores obtenemos la especificación REST al completo.

Algunos ejemplos:

- Recursos:
  - <http://mirestapp/cursos> → con un get obtenemos todos los cursos
  - <http://mirestapp/cursos/3641> → con un get obtenemos el curso con id 3641
- Representaciones:
  - <Curso><id>3641</id><name>Mi curso</name></Curso>
  - {"Curso": {"id": "3641", "name": "Mi curso"}}

Http también soporta de forma nativa un mecanismo de control de tipo de datos, haciendo posible una comunicación entre el cliente y el servidor que lleva a cabo la gestión de los diferentes tipos de datos en la respuesta.



**Figura 10: Negociación de formatos (Pro Restfull APIs)**



## **2.5. Diseño del A.P.I.**

El diseño de un API tiene que resolver una gran cantidad de preguntas, pero lo primero que hay que determinar es el objetivo final. Para nosotros este será la implementación de un API sencilla de fácil uso y rápido desarrollo.

Una vez resuelto esto tendremos en cuenta el número de desarrolladores, el número de aplicaciones que interactuarán con la API, el número de usuarios que la utilizarán así como también la forma en que mejoraremos la API con el tiempo o como esta va a ayudar a la empresa a alcanzar sus objetivos. En nuestro caso los usuarios del API será solamente la aplicación en sí aunque se tendrá en cuenta posibles reutilizaciones o acceso a datos desde otras aplicaciones.

Para acometer el diseño de nuestra API se han seguido los siguientes pasos:

### **2.5.1. Análisis de Dominio o descripción de la API**

Aquí se analiza y propone la estructura organizativa del acceso a los recursos por parte de los clientes. Para realizar esto hay que tener muy presente que en una API-REST sólo podemos tener 3 tipos de recursos.

- Recurso independientes: Es un recurso que existe por sí mismo, es decir no necesita que exista otro recurso. Un ejemplo de esto es el recurso Usuario, etc.
- Recurso dependiente: En este caso no puede existir sin otro. Por ejemplo, la asociación que existe entre Profesor y Usuario. Un profesor no puede existir si no existe el recurso usuario antes.
- Recurso asociativo: Un recurso que existe de manera independiente pero que tiene algún tipo de relación, es decir puede ser conectado por una referencia. La relación entre un Curso y un Alumno no se puede determinar si no existen ambos. Por lo tanto Alumno es asociativo y dependiente de Curso.

**Recursos independientes:** Usuario, Curso

**Recursos dependientes:** Alumno, Profesor, Elemento\_curso

**Recursos asociativos:** Alumno-Curso, Profesor-Curso, Usuario-Curso.

Ahora se pasa a identificar las transiciones entre estados. Estas transiciones nos proporcionan un indicador de las necesidades que se tiene que soportar la API así como de los métodos http debería soportar la misma. Por ejemplo un alumno, que puede inscribirse a un curso, también se podría ver como un alumno que puede ser añadido a un curso.

### **EJEMPLO CON EL RECURSO CURSO**

Acción	Operación HTTP	Objeto de Dominio	Descripción
Crear	POST	cursos/	Crea un curso nuevo y lo añade a la base de datos.
Actualiza	PUT	cursos/{id_curso}	Actualiza los datos del curso.
Eliminar	DELETE	cursos/{id_curso}	Elimina los datos del curso con el identificador.



Leer	GET	cursos/{id_curso}	Obtiene los datos del curso con el identificador.
Leer	GET	cursos/	Obtiene los datos de todos los cursos disponibles en el sistema.

### **2.5.2. Análisis de los requisitos de la Arquitectura**

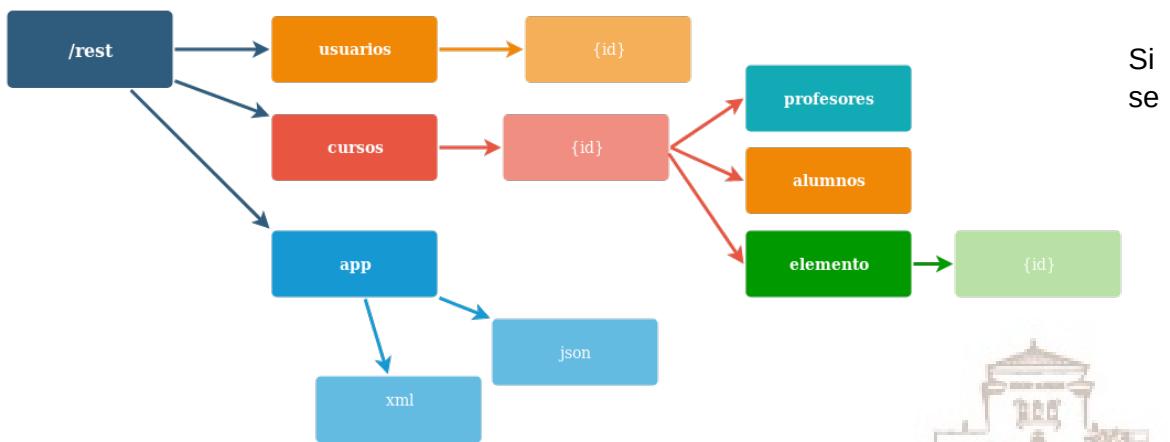
La arquitectura de la API debe cumplir una serie de requisitos marcados en las especificaciones. A continuación se enumeran los requisitos más importantes que debe cumplir para poder llevar a cabo la aplicación.

- La api debe proporcionar una forma de control de los protocolos (json, xml).
- La api debe permitir (leer, borrar, editar y crear) los cursos, usuarios y profesores de manera independiente.
- La api debe permitir leer los cursos, los usuarios y los profesores de manera conjunta (es decir debe devolver una lista de los mismos)
- La api debe permitir acceder a un recurso de acceso distinto y único para el usuario administrador de cursos.
- La api debe permitir el acceso al usuario administrador de la aplicación (el usuario root que añade y quita permisos)
- Debe implementar algún tipo de seguridad, Oauth
- La api se debe mantener sencilla y entendible
- Implementar los estados opcionales usando el símbolo “?”

### **2.5.3. Diseño básico de la Arquitectura**

En este apartado se especifican las bases de la arquitectura de manera que sea lo más gráfica y entendible posible.

Cabe destacar del diagrama de rutas, de la figura 11, que la parte representada como “app” se implementará usando la técnica negociación de formatos o contenido que se explicó en el apartado 2.4.1.



**Figura 11: Diagrama de rutas del API REST**



analiza el diseño de las rutas se puede apreciar que es bastante sencillo y que independiza los recursos unos de otros de manera que las llamadas que se realicen desde el cliente puedan ser independientes. También se intenta así que el API sea lo bastante flexible como para realizar añadidos con posteridad. Ya que se podrían incluir en un futuro, recursos como usuarios o administradores o incluso más protocolos.

No obstante hay que destacar que no todos los recursos tendrán asociadas todas las acciones del acrónimo CRUD. Así, por ejemplo el recurso “cursos” sólo podrá llevar asociado GET que devolverá una lista con todos los cursos y la API no responderá a ningún otro verbo como PUT, POST o DELETE.

También hay que decir que para utilizar estados opcionales y/o atributos se añadirán a la ruta puestos detrás de interrogante. Pero esto se explicará más en detalle en la parte del cliente.

Si se desea conocer la API en su totalidad y al detalle se puede mirar la documentación pública del API en este [link](#).

#### **2.5.4. Herramientas para el diseño y documentación del API**

Hay numerosas herramientas y plataformas web que nos proporcionan ayuda a la hora de diseñar y documentar APIs. Estas herramientas se basan en una serie de especificaciones o código que dictamina la estructura. Algunas de las herramientas más conocidas del mercado son:

*Swagger, Dapper Dox, ReDoc, RAML 2 HTML, Snowboard, Slate, Spectacle*

De entre las citadas, la más extendida es Swagger que utiliza la especificación [openAPI](#). Esta especificación, es de licencia libre y está implementada no sólo por swagger sino por muchas aplicaciones más, se puede implementar usando YAML o JSON para diseñar el API. Esto es una gran ventaja sobre todo si se conoce JSON o si se utiliza como uno de los formatos de intercambio y serialización de datos de la aplicación, para mantener una cierta regularidad y cohesión.

#### **2.5.5. Conceptos básicos de Open Api**

Open Api es muy sencillo y explícito. Está orientado al diseño y documentación de APIs REST y si se conoce la estructura de este tipo de servicios, se puede utilizar Open Api con facilidad. La estructura de esta especificación, se basa en meta-information del API, rutas con sus métodos y definiciones de objetos.

La meta información que se puede proporcionar suele ser, el autor, los datos de contacto, la versión, el host del API, etc.,

Las rutas y sus métodos, nos permiten especificar sus parámetros, las respuestas e incluso aportar ejemplos de objetos o documentos devueltos por el servidor.

Las definiciones nos permiten definir modelos y reutilizar objetos a lo largo del documento.

Combinando estos conocimientos y utilizando de plantilla el fichero de ejemplo que viene con la mayoría de herramientas podemos crear la documentación del API de la aplicación de manera muy sencilla.



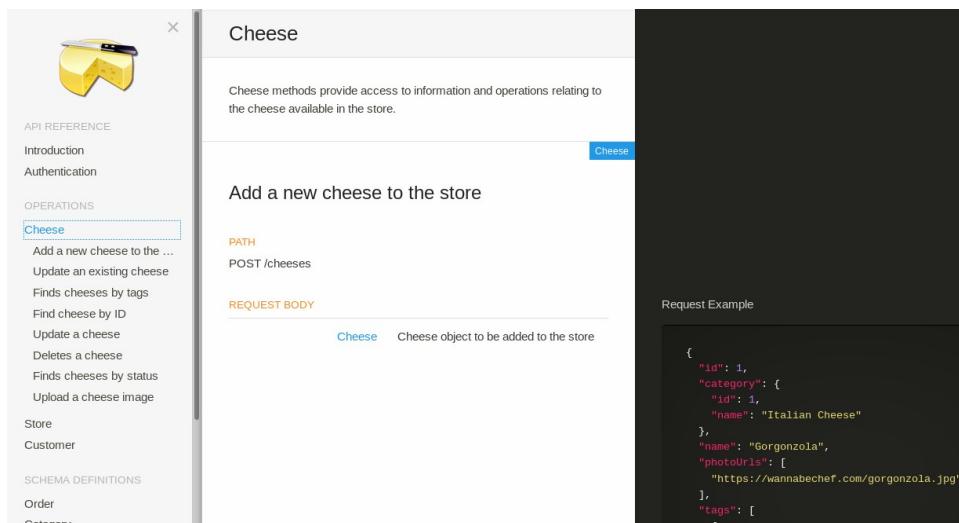
### **2.5.6. Publicación del API usando Spectacle**

Spectacle es una herramienta creada para usarse en la consola, que nos permite generar la documentación del API de una manera muy sencilla. Lo primero que se debe hacer es crear el fichero JSON e introducir el código de diseño de la API usando la especificación Open API<sup>9</sup>. Tras esto, instalamos spectacle-docs usando npm<sup>10</sup>, y luego se debe ejecutar el comando siguiente.

```
$ spectacle miAPI.json
```

*Código 1: Comando de ejecución de programa Spectacle*

Obtenemos así la documentación en formato HTML que se almacena en el directorio public creado por Spectacle en el directorio donde nos encontramos. Esta herramienta es muy útil y tiene bastantes más opciones, como la auto-generación de documentación tras guardar los cambios en el documento del api y otras. El funcionamiento básico, instalación y opciones se pueden consultar en su [web](#).



```
{  
  "id": 1,  
  "category": {  
    "id": 1,  
    "name": "Italian Cheese"  
  },  
  "name": "Gorgonzola",  
  "photourls": [  
    "https://wannabechef.com/gorgonzola.jpg"  
  ],  
  "tags": [  
    ...  
  ]  
}
```

*Figura 12: Ejemplo de documentación html del API con Spectacle*

### **2.5.7. Desarrollo del API en el proyecto**

Aunque en principio se decidió utilizar Spectacle para el diseño y creación de la documentación del API, por tener que desarrollar un API muy sencilla y poco extensa, se acabó usando Swagger<sup>11</sup> también por la facilidad que aporta el editor y la reutilización y conversión de código YAML a JSON. La primera versión del diseño del api se puede encontrar [aquí](#) junto con la [documentación generada](#) por el programa spectacle.

---

9 Spectacle usa la especificación Open Api v.2.0.

10 Ver apéndice Fundamentos de NPM para aprender a instalar un programa usando “npm”.

11 Para ver cómo se utiliza el editor de APIs Swagger ves al anexo E



## 2.6. Diseño y Estructura del Servidor

La parte del servidor debe ser visto en forma de pila de tareas como un servicio REST apoyado sobre un DAO que ejecuta acciones sobre la Base de Datos. Todo ello es gestionado por un Servidor Tomcat. El servidor Tomcat se encargará de gestionar la base de datos y hacer más eficiente el manejo de la misma por parte de la aplicación agrupando las conexiones a la B.B.D.D. en grupos. Esto hará que se reduzca enormemente los tiempos de respuesta de la base de datos.



**Figura 13: Esquema estructura interna del servidor**

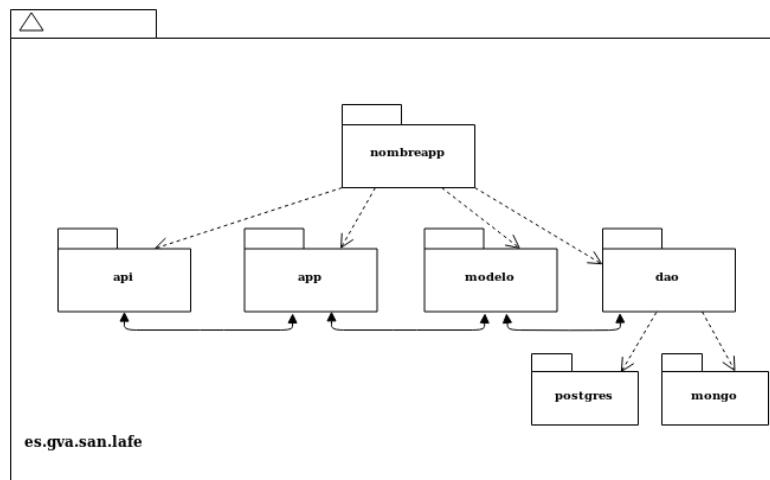
## 2.7. Estructura de paquetes y clases

Ya se ha hablado sobre las capas del servidor y cómo deben de estar separadas, ahora bien la pregunta que surge es, ¿Cómo implementamos este patrón?

La forma más sencilla es separar el código de nuestra aplicación en 4 paquetes diferenciados. Estos paquetes deben representar el API, la aplicación, el modelo y el DAO. El primero, se encarga de describir la API con las anotaciones del framework Jersey y realiza llamadas al paquete de aplicación. La segunda capa, que es la capa aplicación, utiliza los objetos DAO y el modelo para devolver al API el resultado final de la computación.

Finalmente la capa DAO es la que se encarga de manejar la Base de Datos y traducir los datos obtenidos de ella a un modelo entendible por la capa aplicación (modelo) basado en objetos.





**Figura 14: Diagrama de estructura de paquetes de la aplicación**

## 2.8. Análisis de las herramientas

Justo antes de comenzar a desarrollar el proyecto se realizó un estudio de las herramientas y de las posibilidades que hay en el mercado. Este estudio se plasmó en una presentación en diapositivas que analizaba los problemas y necesidades de la aplicación. Dicha presentación se puede encontrar en el siguiente [enlace](#).



## 3. DESARROLLO DEL PROYECTO

### 3.1. Implementación del modelo de la B.B.D.D.

Para la implementación del modelo de datos se siguieron los siguientes pasos comentados a continuación.

Se utilizó Squirrel SQL para modelar y probar la base de datos. Tras dar mucho la tabarra al tutor, y gracias a sus indicaciones, se fue clarificando la estructura del modelo y se pasó a implementar una base de datos en la que poder realizar pruebas.

Se crearon las tablas directamente con código SQL sobre el programa.

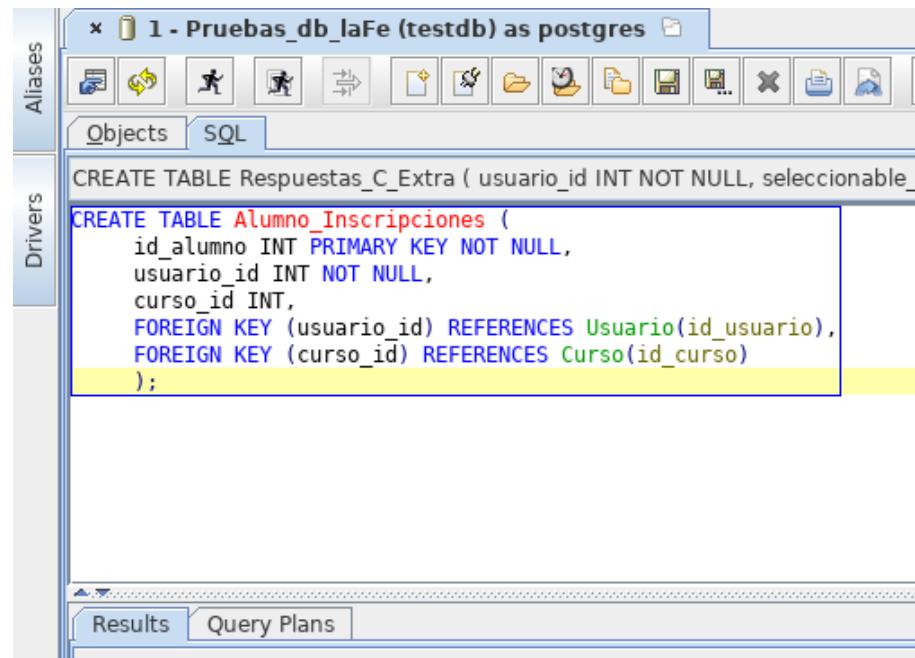


Figura 15: Creando tablas con Squirrel SQL

También se fue viendo qué cambios había que realizar para la inclusión de campos que se pudieran personalizar.

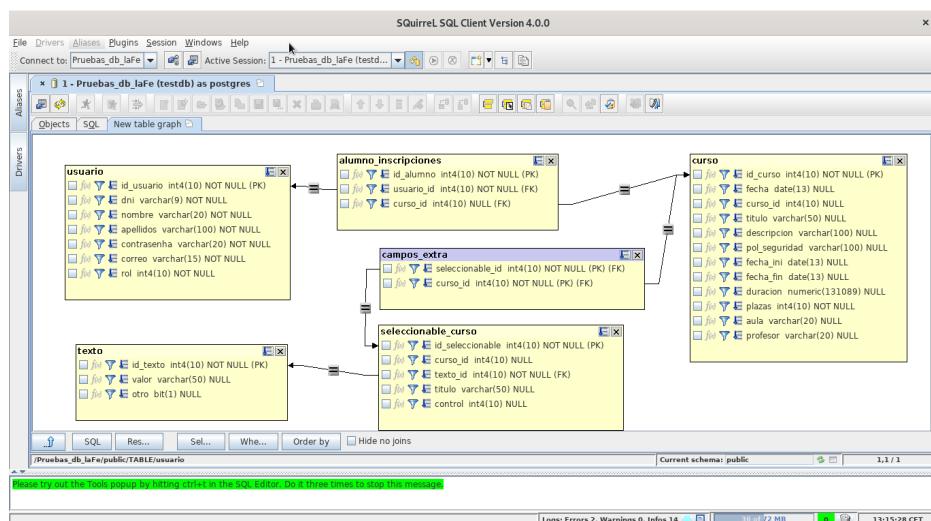


Figura 16: Revisando la estructura con Squirrel SQL graph plugin

Así, dadas las necesidades y los test realizados con SQL sobre el programa<sup>12</sup> se fue reduciendo, simplificando y clarificando la forma de la base de datos. Finalmente la estructura del modelo se recoge en la figura siguiente.

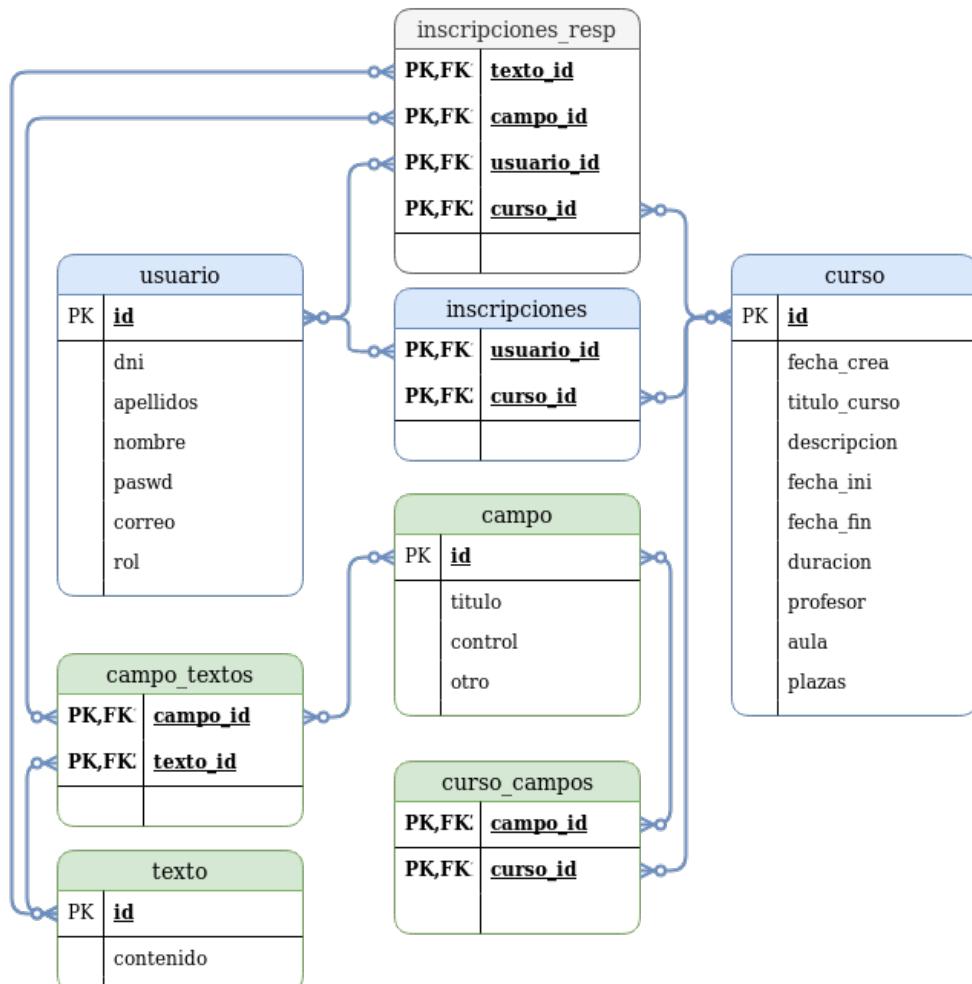


Figura 17: Diagrama de relación del modelo de datos

Las cajas de color azul son el modelo de datos básico, la funcionalidad básica del programa que se desarrollará en la primera versión de pruebas de la base de datos. Las cajetillas verdes representan la funcionalidad personalizada del editor de cursos. Y en la tercera versión, se implementarán las funcionalidades básicas para llevar a cabo el almacenamiento de los datos seleccionados por los usuarios sobre los campos extra.

El código de creación de la primera parte del modelo, se puede descargar desde [aquí](#).

Para comprobar la primera parte de la B.B.D.D. se insertaron 7 usuarios y 3 cursos. Luego se combinaron usuarios y cursos en la tabla también con el comando INSERT.

Posteriormente se borró un usuario y se comprobó que se borraban todas las filas que tenía ese usuario en la tabla inscripciones. Se realizó la misma comprobación para la tabla curso. Todo funcionó correctamente pues se insertó “ON DELETE CASCADE” a las claves foráneas.



```
INSERT INTO usuario (nombre) VALUES ('usu1');
INSERT INTO curso (titulo) VALUES ('cur1');
INSERT INTO curso (titulo) VALUES ('cur2');
INSERT INTO inscripciones VALUES ( 1, 2);
INSERT INTO inscripciones VALUES ( 1, 1);
DELETE FROM curso where id_curso=1;
SELECT * FROM inscripciones;
```

*Código 2: Ejemplo de comprobación de la base de datos*

Para comprobar que se pueden mostrar todos los cursos asociados a un usuario se utilizó el siguiente código.

```
SELECT usuario.nombre, curso.titulo FROM usuario
    INNER JOIN inscripciones ON usuario.id_usuario = inscripciones.usuario_id
        INNER JOIN curso ON curso.id_curso = inscripciones.curso_id
WHERE
    usuario.id_usuario=3;
```

*Código 3: Ejemplo de comprobación de la base de datos*

Para comprobar que se pueden mostrar todos los usuarios asociados a un curso se utilizó la siguiente sentencia.

```
SELECT usuario.nombre, curso.titulo FROM usuario
    INNER JOIN inscripciones ON usuario.id_usuario = inscripciones.usuario_id
        INNER JOIN curso ON curso.id_curso = inscripciones.curso_id
WHERE
    curso.id_curso=2;
```

*Código 4: Ejemplo de comprobación de la base de datos*

A continuación se creó la segunda parte, la marcada en color verde. Para ello se creó el siguiente [código](#) que incluye la creación de las tablas.

Luego se introdujeron datos en las tablas datos, para poder así, testear el añadido realizado a la base de datos.



```
INSERT INTO campo (titulo) VALUES ('categorias');
INSERT INTO campo (titulo) VALUES ('años de residencia');

INSERT INTO texto (valor) VALUES ('1');
INSERT INTO texto (valor) VALUES ('2');
INSERT INTO texto (valor) VALUES ('3');

INSERT INTO texto (valor) VALUES ('médico');
INSERT INTO texto (valor) VALUES ('enfermera');
INSERT INTO texto (valor) VALUES ('celador');
INSERT INTO texto (valor) VALUES ('auxiliar');

INSERT INTO campo_texto VALUES (1,4);
INSERT INTO campo_texto VALUES (1,5);
INSERT INTO campo_texto VALUES (1,6);
INSERT INTO campo_texto VALUES (1,7);
INSERT INTO campo_texto VALUES (2,1);
INSERT INTO campo_texto VALUES (2,2);
INSERT INTO campo_texto VALUES (2,3);

INSERT INTO campos_extra VALUES (1,2);
INSERT INTO campos_extra VALUES (2,3);
```

*Código 5: Introduciendo datos en la segunda estructura de la base de datos*

Tras esto, se realizaron dos consultas SQL para ver si se podían extraer los resultados que necesitamos. De esta forma tan básica, se comprueba que la estructura funciona correctamente.

```
-- select para ver la relación entre curso y campos extra
SELECT curso.titulo, campo.titulo FROM curso
    INNER JOIN campos_extra ON curso.id = campos_extra.curso_id
        INNER JOIN campo ON campos_extra.campo_id = campo.id

WHERE
curso.id_curso=6;

-- select para ver el título del curso, campos extra y los valores de cada campo extra
SELECT curso.titulo, campo.titulo, texto.valor FROM Curso
    INNER JOIN campos_extra ON curso.id = campos_extra.curso_id
        INNER JOIN campo ON campos_extra.campo_id = campo.id
            INNER JOIN campo_texto ON campo.id =
campo_texto.campo_id
                INNER JOIN texto ON campo_texto.texto_id =
texto.id

WHERE
curso.id_curso=5;
```

*Código 6: Código de test para curso-campo-varlo\_campo*



Los resultados fueron buenos y se consiguió lo que se pretendía, como se aprecia en la figura 18.

Results	Meta data	Info	Overview / Charts	Rotated table	Results as text
	curso			campo_extra	contenido
cur2				categorias	médico
cur2				categorias	enfermera
cur2				categorias	celador
cur2				categorias	auxiliar
cur2				años de residencia	1
cur2				años de residencia	2
cur2				años de residencia	3

**Figura 18: Resultados de búsqueda en SQuirreL SQL**

Antes de pasar a implementar la última parte del modelo, se realizó un último test de precaución. Se borró un curso que hiciera referencia a un campo extra que también lo tuviera otro curso. De esta manera se vería si al borrar el curso, se borraba el campo extra. Tras realizar esto se vio que no se producía este hecho. Esto ocurre gracias a tener una tabla intermedia con dos claves primarias.

```
DELETE FROM Curso WHERE id = 6;
```

### *Código 7: Borrado de dato de curso*

Results	Meta data	Info	Overview / Charts	Rotated table	Results as text
curso		campo_extra		contenido	
cur2		años de residencia		1	
cur2		años de residencia		2	
cur2		años de residencia		3	

**Figura 19:** Resultados tras la eliminación del curso

Finalmente, se implementa la última parte, que consta solamente de una tabla, la cual funciona de enlace entre los textos, los campos, los cursos y los usuarios. Así se puede establecer una relación entre los desplegables y la elección de cada uno de los usuarios.

La forma de realizar este añadido es similar a la anterior. Se crea la tabla como se puede apreciar en este [archivo](#). Y luego se introducen valores en la tabla.

```
INSERT INTO inscripciones_resp VALUES (3, 7, 1, 2);
INSERT INTO campos_extra VALUES (2,3);
```

*Código 8: Introduciendo datos en la segunda estructura de la base de datos*

Tras esto, se comprueba que se puedan extraer los datos con una búsqueda usando el lenguaje SQL.

```

SELECT curso.titulo, usuario.nombre, campo.titulo, texto.valor
FROM inscripciones_resp
    INNER JOIN curso ON inscripciones_resp.curso_id = curso.id
        INNER JOIN usuario ON inscripciones_resp.usuario_id = usuario.id
            INNER JOIN campo ON inscripciones_resp.campo_id = campo.id
                INNER JOIN texto ON inscripciones_resp.texto_id =
text.id
WHERE usuario.id_usuario = 5

```

Código 9: Código de test para curso-campo-valor\_campo

Una vez comprobado esto, se pensó bien todos los campos del modelo. Se renombraron tablas y se creó un archivo con una serie de sentencias SQL para crear una base de datos ya más cercana a la que se implementará finalmente.

Poco a poco, se va acotando el modelo que finalmente se usará de pruebas para la realización de la aplicación. Tras otro repaso de concepto, se cambiaron claves primarias introduciendo identificadores para esas tablas, se corrigieron errores en las convenciones de nombres, se añadieron restricciones “UNIQUE” en algunos campos y se cambió de tabla el atributo “otro”. También se introdujo la creación automática del campo fecha de la tabla inscripciones.

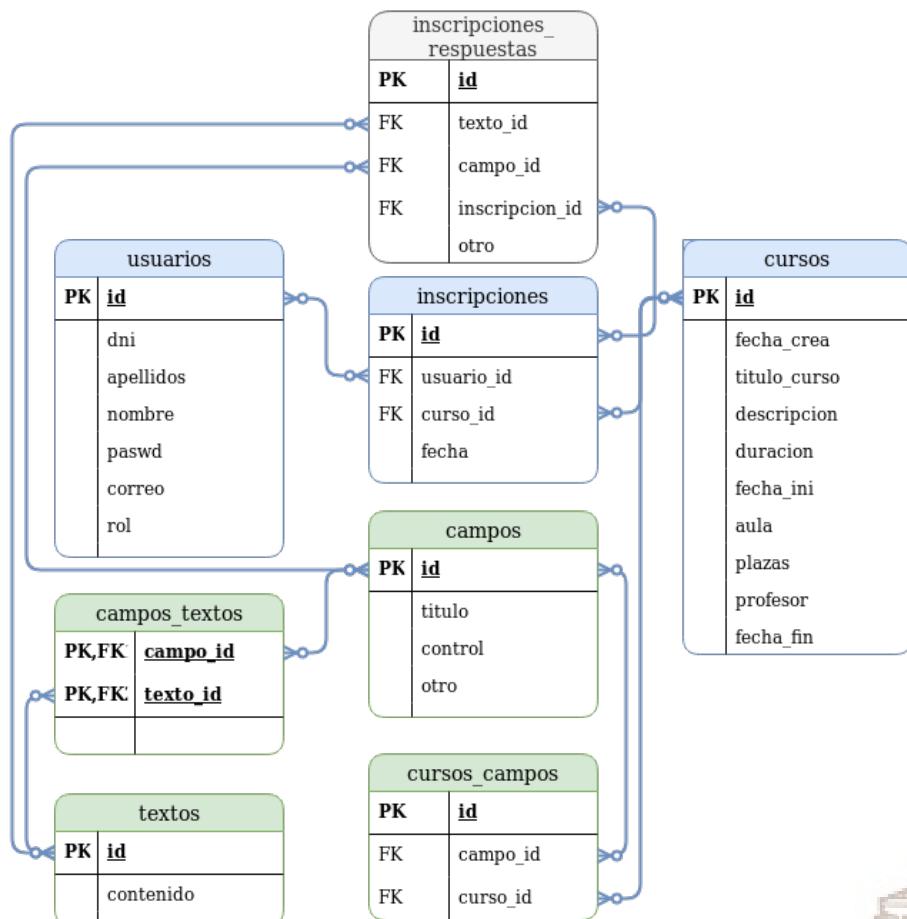


Figura 20: Mejora del modelo



Finalmente se añadió el campo activo a la tabla usuarios para evitar borrar usuarios de la base de datos. Todos los scripts generadores de esquemas, y datos que se implementaron en las primeras versiones del programa se pueden encontrar en el siguiente [repositorio](#)<sup>13</sup>.

### 3.1.1. Scripts de creación de la estructura de la base de datos

Posteriormente y tras comprobar el funcionamiento correcto de la base de datos, se pensó en la manera de crear un script de generación de base de datos usando el programa SQuirreL SQL. La manera de generarla es mediante el uso de los menús contextuales del programa. Se seleccionaron las tablas que se incluirán en el script, luego se pulsó con el botón derecho del ratón y en el menú desplegable sobre “Scripts → Create Table Script”.

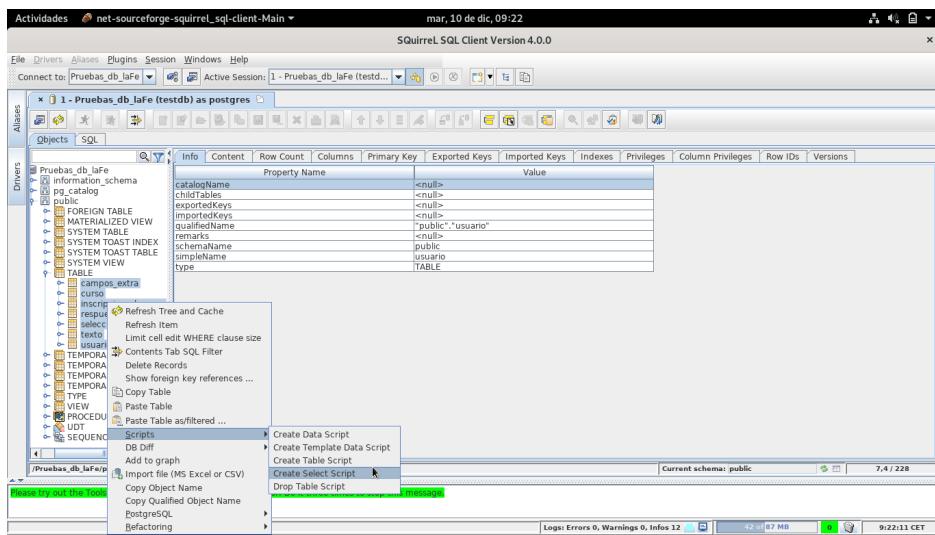


Figura 21: Creando Script de Tablas con SQuirreL SQL

Esta otra forma de generar un [script](#) para la base de datos, resulta no ser una forma tan explícita aunque funciona muy bien y se ahorra bastante tiempo una vez ya se tiene la estructura y sólo hay que cambiar algunos detalles, evitando realizar búsquedas y cambios en el código que nos podrían generar errores si no se analizan en profundidad.

Cabe destacar que el diseño de esta base de datos se realizó intentando que no hubiera redundancia en los datos. Es decir separando al máximo, las entidades de las acciones. No fue tarea fácil y se pudo llevar a cabo gracias a la ayuda y guía del tutor de las prácticas.

13 La relación de nombres de archivos y esquemas se muestra al final del documento.



### **3.2. Implementación de la arquitectura REST**

Hay muchas y diversas maneras de implementar una arquitectura REST. Como ya se ha comentado en este proyecto se utilizará el lenguaje java, un servidor Tomcat<sup>14</sup> y Jersey. A continuación se explican las bases, contenido y construcción de la misma.

Para la construcción se utilizó el IDE Eclipse, propuesto por el tutor de las prácticas.

#### **3.2.1. ¿Qué es JAX-RS?**

JAX-RS define un grupo de APIs de Java para el desarrollo de servicios Web construidos de acuerdo con la transferencia de estado representacional (REST) (JSR 370, 339, 311).

JAX-RS, define la forma correcta de empaquetar una aplicación Web en un archivo .war que será desplegado en un Servidor de Servlets de Java. Para ello se empaquetan las clases en el directorio WEB-INF/clases o en WEB-INF/lib y las librerías solamente en este segundo directorio.

Es bueno estudiar estas especificaciones para conocer lo que ocurre en una aplicación web REST a “bajo nivel” y cómo funciona internamente, aunque en este documento no se pretende profundizar tanto.

Algunos de los principios que sigue JAX-RS son:

- Debe asignarse un “id” a todo recurso.
- Los recursos deben enlazarse entre ellos.
- Debe usarse un set común de métodos.
- Se deben permitir múltiples representaciones.
- Se debe mantener la comunicación sin estado (stateless).

Las principales características de JAX-RS son:

- Una API de recursos basada en POJO<sup>15</sup>s.
- Ejecución de clases usando HTTP.
- Independencia de formato (content types).
- Independencia del contenedor (servidor-web).
- Fácil inclusión de Java EE.

La especificación JAX-RS 2.0 se mantiene consistente con la temática principal de Java EE 7 pero además integra también una serie de APIs muy esperadas. A estas APIs se les suele llamar la API Simplificada y se puede estructurar en temáticas como las siguientes.

**API de Cliente:** Permite llamadas HTTP de bajo nivel, compartir la API con el Servidor y permite la compatibilidad con las implementaciones de JAX-RS 1.0.

**Filtros e interceptores:** Integra el Loggin, la compresión y otras fórmulas que adjudican seguridad a la especificación.

---

<sup>14</sup> Tomcat Versión 9

<sup>15</sup> POJO es lo que se denomina una clase java básica, las de toda la vida.



**Cliente y Servidor asíncronos:** Permite ejecutar diferentes hilos de ejecución en el servidor, soporte para peticiones asíncronas y “Servlet3 async”.

**Mejora en la negociación de la conexión:** Permite determinar automáticamente la respuesta si la especificación de cliente está integrada también.

**Validación:** Permite la validación de datos a través de servicios, restricciones a través de anotaciones, validación de rutas a través de expresiones regulares, etc.,

**HATEOAS<sup>16</sup>:** Esta característica es muy importante en las arquitecturas REST. Nos permite proporcionar hyperlinks/URIs en la petición y la respuesta de los servicios web. Es similar a los enlaces en un formulario HTML.

### 3.2.2. ¿Qué es Jersey?

Jersey es la implementación de referencia de JAX-RS<sup>17</sup>. Sería como decir que JAX-RS es la API definida por Oracle para usar REST con Java y Jersey es el framework que nos facilita la tarea de implementarlo en nuestra aplicación.

Este framework se sustenta sobre varios pilares básicos como el uso de WADL pero sobre todo hace uso de las anotaciones Java para facilitar el uso del framework. Estas anotaciones se escriben con el símbolo “@” y son un tipo de metadato que proporcionamos al framework para ejecutar sus funcionalidades sobre las clases y métodos de nuestro código.

Para crear una aplicación usando Jersey se debe incluir la librería y todas sus dependencias. Se debe configurar el fichero web.xml que enlaza los paquetes del contenedor Jersey con nuestros paquetes de recursos. Finalmente, el programa o aplicación propia, que incluye anotaciones Java que se utilizan para integrar la aplicación con el framework. El uso de estas anotaciones nos puede servir también para sustituir el enlazado de paquetes que se realiza en el fichero web.xml.

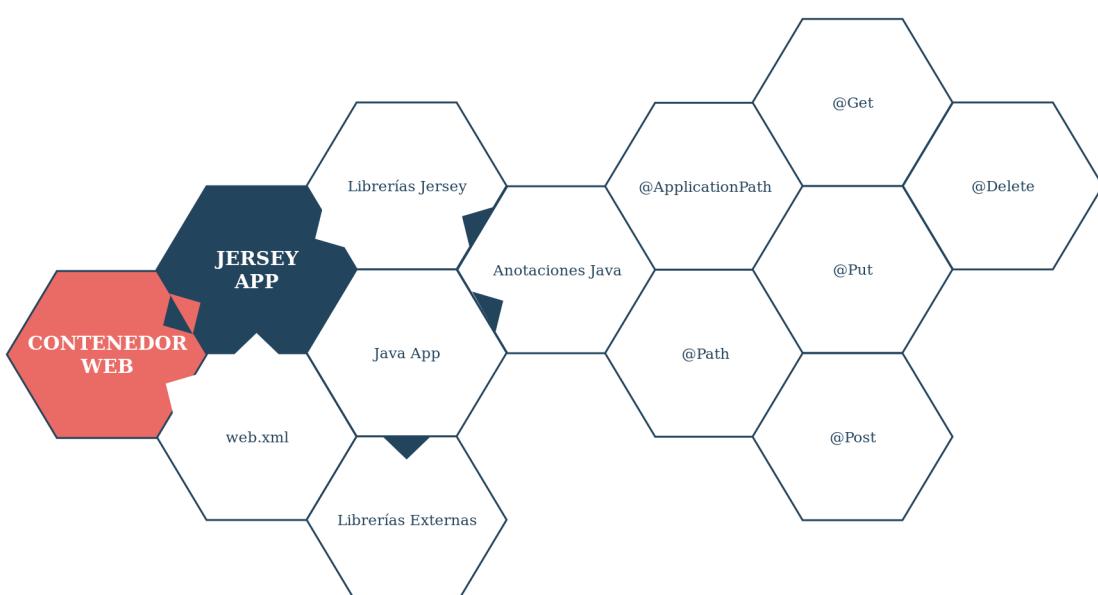


Figura 22: Estructura de aplicación web Jersey

---

16 Del inglés → Hypermedia as the Engine of the Application.

17 Se hizo un resumen de las principales anotaciones y metadatos usados con JAX-RS y Jersey y se colocó en el Apéndice F.



### **3.2.3. La representación de los datos cliente-servidor**

A la hora de devolver los datos de la aplicación servidor al cliente hay que serializar dichos datos, pues si se devolvieran los objetos Java del modelo, resultaría muy difícil y limitado trabajarlos con el cliente. Además, los clientes Web no son muy buenos trabajando con objetos Java. Así, para trabajar con clientes de tipo navegador web<sup>18</sup>, es muy usual utilizar XML o JSON. Particularmente, para la realización del presente proyecto se decidió implementar una serialización de objetos Java a JSON. Más adelante, si procediera se implementará también la serialización a XML.

Se escogió JSON por su facilidad a la hora de integrarlo con Angular, así como también por su enorme popularidad. Este tipo de formato, está sustituyendo poco a poco a XML en entornos web.

### **3.2.4. Librerías Java para serializar JSON**

Hay numerosas librerías que nos pueden ayudar a realizar esta acción. Las más populares son; Jackson, Google Gson, JsonMarshaller y JSON.simple.

Para la realización del proyecto se decidió utilizar la librería “Google Gson” por ser una librería muy sencilla de utilizar e implementar.

### **3.2.5. Google Gson**

Para instalar esta librería sólo hay que [descargar](#) e insertar el archivo “.jar”, en a la carpeta “WEB-INF” del proyecto web de eclipse. Para utilizarla hay que seguir las instrucciones de uso de la [guía de usuario](#).

También se probó con la librería Jackson, pero esta última necesita de más archivos para su instalación y es más tediosa de utilizar. La librería Gson, realiza la tarea de manera mucho más sencilla<sup>19</sup> y sólo se necesita crear un objeto para hacer la traducción a y de JSON. Así, nos permite tener un código más limpio, con un menor número de líneas.

---

<sup>18</sup> En el caso de esta aplicación, se utilizará un navegador que renderice una web estática creada con Angular.

<sup>19</sup> Jersey también implementa la serialización de manera automática, cosa que resulta aún más sencilla de implementar. No obstante se eligió una librería por proporcionar una mejor conversión de clases Java complejas.



### **3.3. Implementar herramientas en el servidor**

#### **3.3.1. El log de la aplicación**

Casi toda aplicación de cierta envergadura, incluye su propio “log” o API de trazas. Este tipo de técnicas, se usan con la finalidad de depuración de la aplicación, así como para registrar la interacción del usuario. Las aplicaciones que cumplen función de servidor suelen usar el “loggin” para registrar también errores y otros mensajes de utilidad para la aplicación cliente que se conecte a su servicio separado que está ocurriendo en el “back-end”.

#### **3.3.2. Para qué utilizar los logs**

Una de las preguntas que se suelen realizar al empezar a utilizar estas técnicas es, ¿por qué no utilizar simplemente “System.out.println()”?.

Principalmente porque se requiere de una mayor flexibilidad. Esta flexibilidad nos aporta seleccionar niveles de prioridad, mostrar mensajes de sólo ciertos módulos o clases, controlar el formato de los mensajes así como decidir el destino de los mismos.

Estas son sólo algunas de las razones por las que se desarrollaron los “loggers”.

#### **3.3.3. Tecnologías**

Para aplicar estas técnicas en aplicaciones Java, encontramos una serie de tecnologías que nos facilitan la tarea. Aunque hay multitud de loggers, las tecnologías más importantes actualmente son:

- `java.util.loggin`: Viene de serie con el lenguaje Java aunque no es muy utilizado.
- Log4J: El estándar de facto hasta hace unos pocos años.
- Logback: El sucesor de Log4J, creado por el mismo desarrollador y actualmente usado en multitud de proyectos.
- Log4J 2: Última versión de Log4J que aporta muchas más funcionalidades y arregla problemas que han ido apareciendo en Log4J y Logback. Para más información sobre las ventajas de este logger, se puede consultar la introducción del [manual de usuario](#) de Log4J 2 de apache.

A pesar de que simplemente podríamos elegir uno e instalarlo, sería bueno tener una cierta independencia de la tecnología a aplicar. Pues con el paso del tiempo, como se ha visto, se ha ido cambiando de Log4J a Logback y ahora finalmente a Log4J 2. Para salvar esta dificultad se crearon los “frameworks de loggin”. Este tipo de tecnologías, nos ayudan a crear logs y posteriormente, si se diera el caso de querer cambiar de una tecnología subyacente de loggin, a otra distinta, poder realizarlo sin grandes dificultades..

Hay dos frameworks que se utilizan mucho en la actualidad, tinylog<sup>20</sup> y SLF4J. Para el desarrollo del proyecto se gastará este último, combinado con Log4J 2, por lo que se tiene que aprender algo sobre cómo usar SLF4J en la aplicación así como también, aprender a configurar Log4J 2.

---

20 Un framework minimalista diseñado para ser fácil de usar.



### 3.4. ¿Qué es DAO?

DAO<sup>21</sup> es un objeto de acceso a datos, un componente que suministra una interfaz común y uno o más dispositivos de almacenamiento de datos. Es un patrón de diseño y es considerado una buena práctica. Los objetos de acceso a datos pueden usarse para aislar una aplicación de la tecnología de persistencia subyacente (la B.B.D.D.) y así lograr una independencia a la hora de alterar o cambiar por completo la capa inferior (persistencia) sin tener que alterar la aplicación en sí misma.

La principal desventaja que conlleva este patrón, es el aumento de trabajo en el desarrollo pues hay que desarrollar una capa intermedia entre la aplicación y la base de datos, pero se gana flexibilidad e independencia entre capas.

#### 3.4.1. ¿Cómo implementar un DAO?

Para implementar un DAO hay que crear una serie de clases e interfaces que hacen de puente entre la aplicación y la base de datos. Las interfaces nos ayudarán a la hora de crear una nueva implementación del DAO en un futuro. En el siguiente gráfico se expone la estructura básica de un objeto DAO.

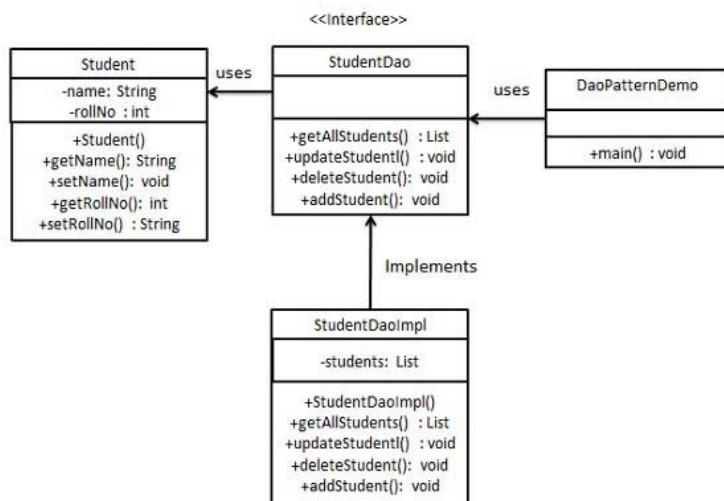


Figura 23: Esquema de DAO de una clase cualquiera ([Tutorialspoint](#))

En el desarrollo del proyecto, se utilizan pocos objetos DAO, por lo que los concentraremos en un administrador de DAOs que nos permita inicializar y ejecutarlos desde una única interfaz.

Posteriormente, si diera tiempo, se implementará también un DAO para otra base de datos como por ejemplo MongoDB.

21 En inglés, “Data Access Object”.



### 3.5. Implementación básica de Tomcat y servicio REST

Para implementar un servidor Apache Tomcat y desplegar una aplicación web en un entorno de producción usaremos el entorno de desarrollo Eclipse.

Crearemos un servidor y una aplicación web dinámica pulsando sobre nuevo proyecto, web dinámico del menú “Archivo”.

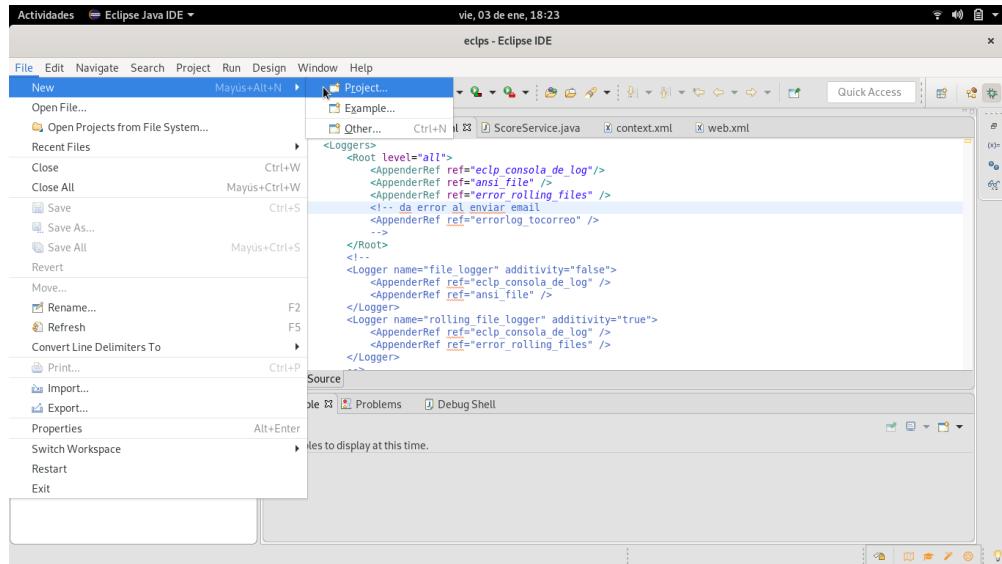


Figura 24: Creando proyecto eclipse

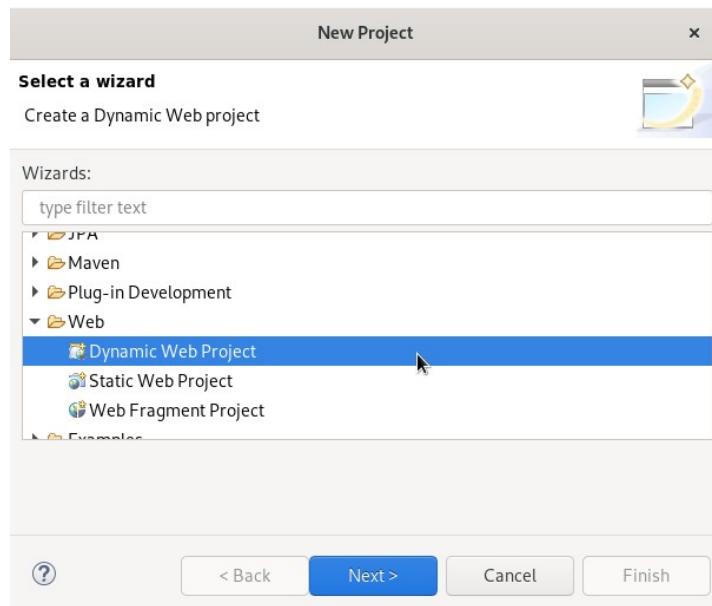
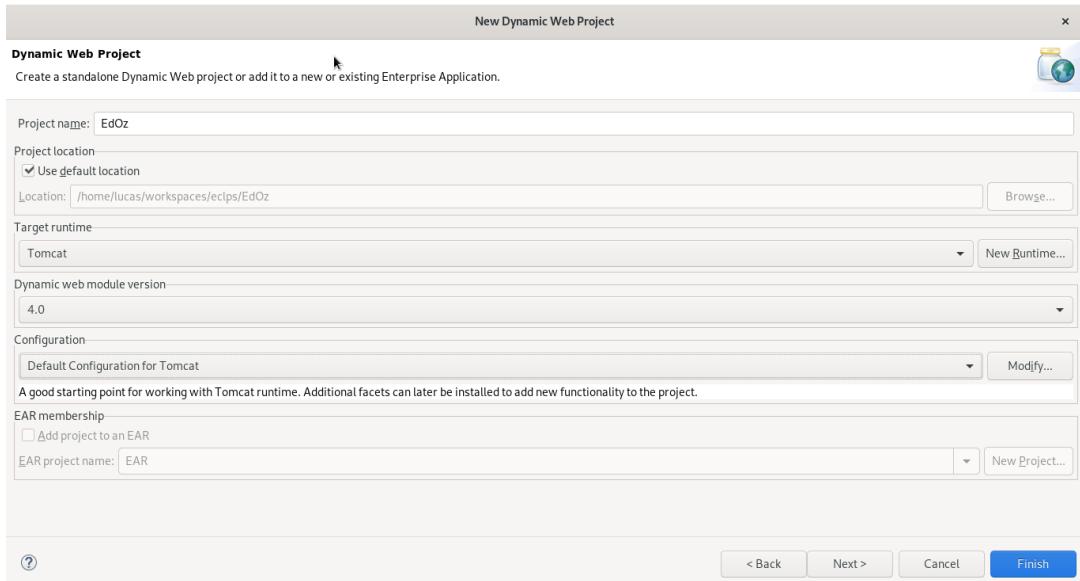


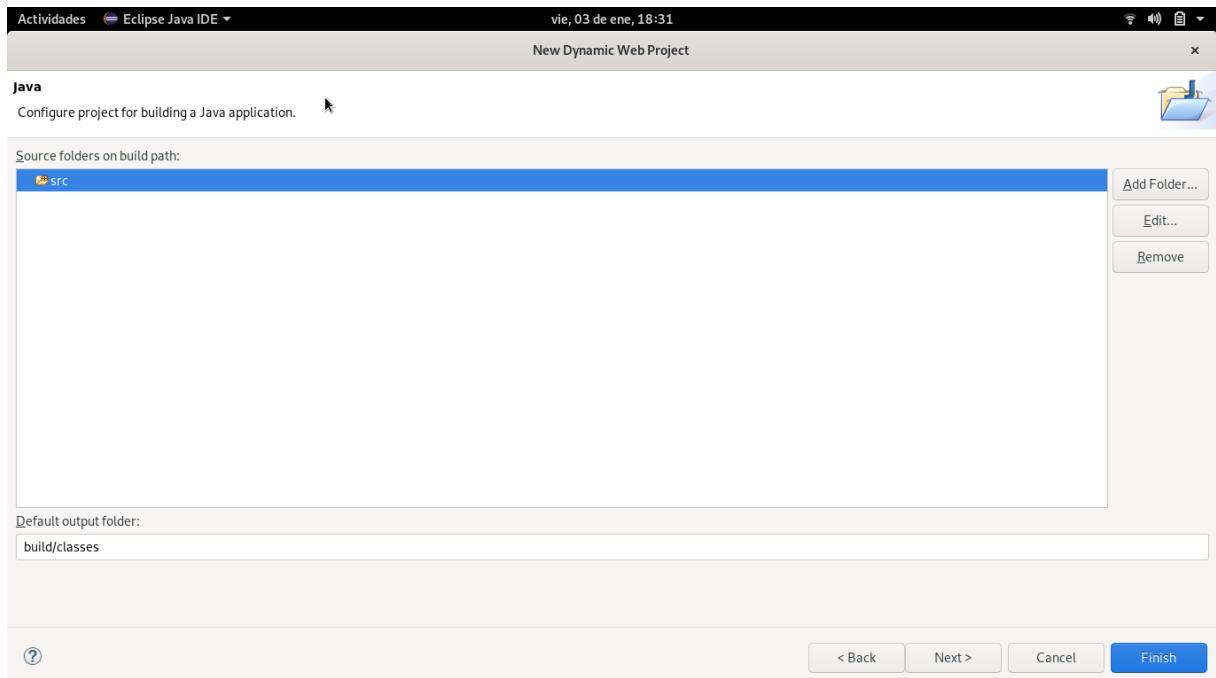
Figura 25: Seleccionando proyecto web dinámica.

Luego a través del generador de proyectos, añadimos los datos del proyecto, es importante generar el archivo web.xml para ahorrarnos la tarea de crearlo después.



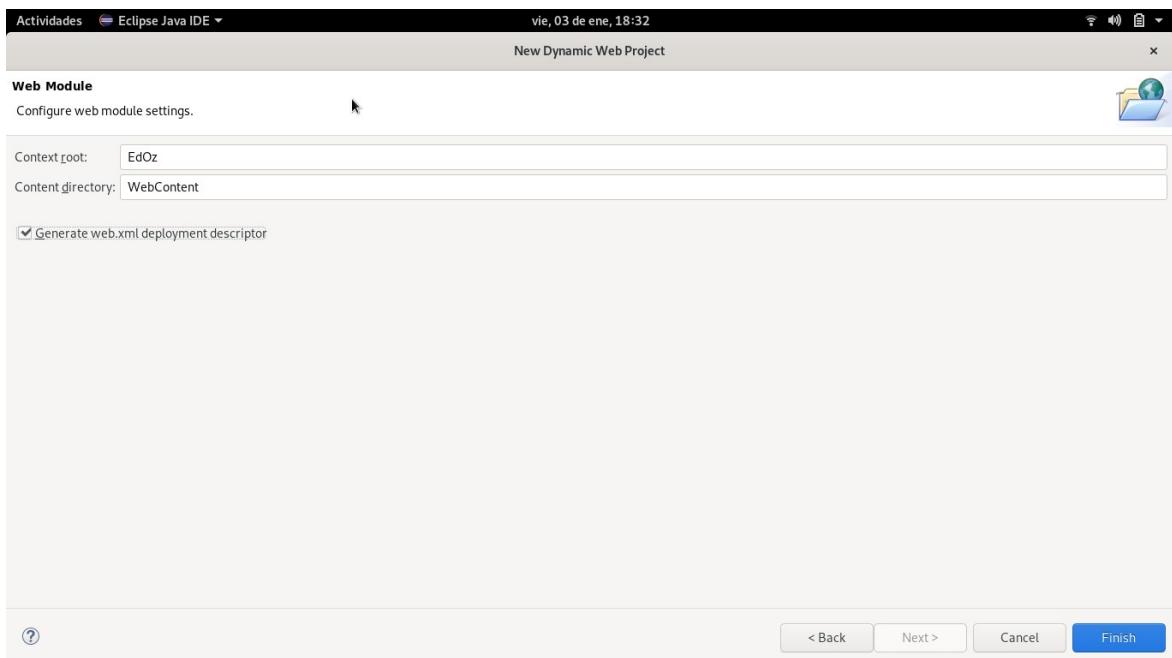


**Figura 26: Añadiendo datos al proyecto web**



**Figura 27: Añadiendo directorios al "Build Path"**



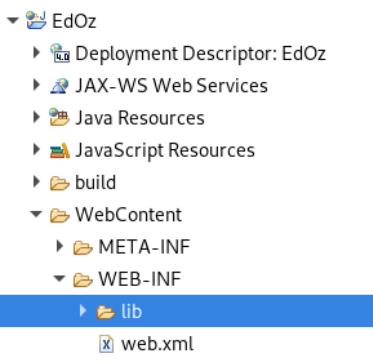


**Figura 28: Confirmando la creación del fichero web.xml**

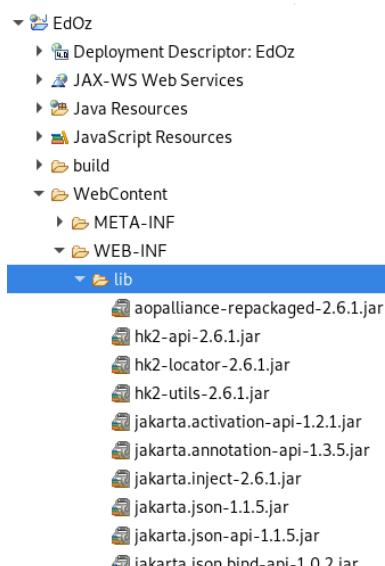
Finalmente, si se desea se puede utilizar la perspectiva Java EE o seguir con la que utilizamos normalmente.

### **3.5.1. Añadiendo las librerías del framework Jersey**

Para utilizar el framework Jersey es necesario [descargar](#) las librerías (archivos .jar) de la web. Luego, tras la creación del proyecto añadimos todas las librerías del framework Jersey en el directorio lib de “WEB-INF”.



**Figura 29: Añadiendo librerías Jersey**



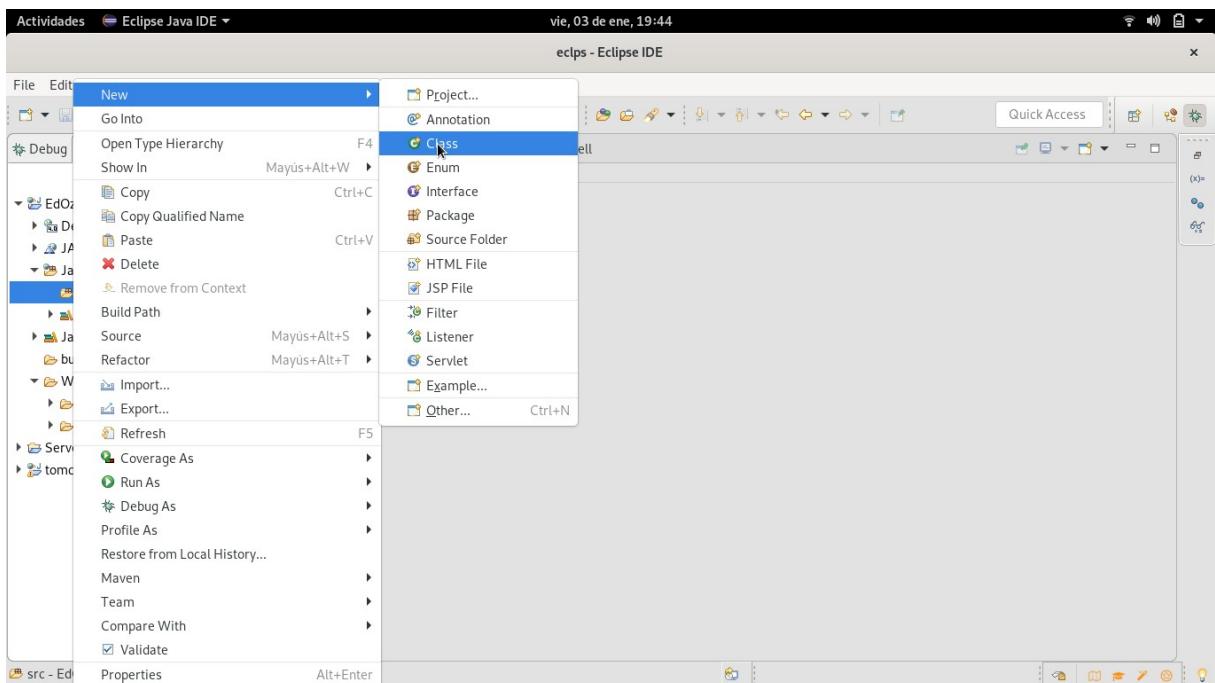
**Figura 30: Añadidos archivos Jersey**

Hay que cerciorarse de copiar todos los archivos que sean necesarios.

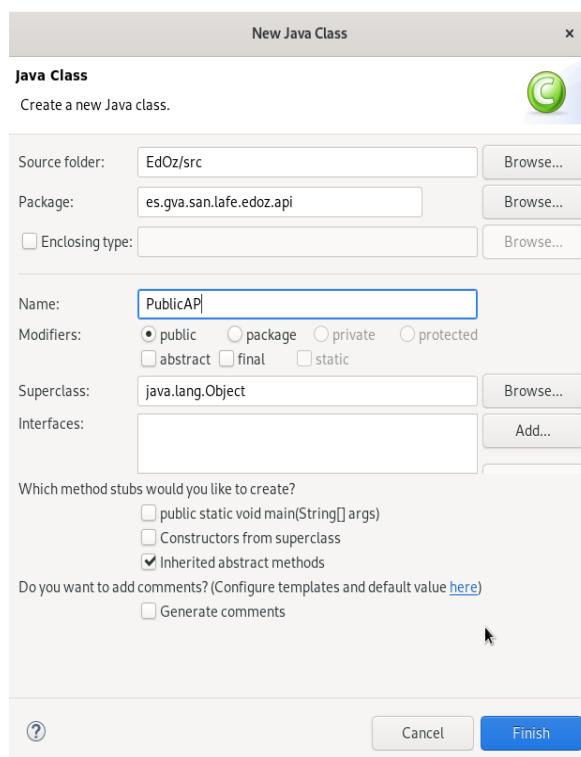


### 3.5.2. Código de implementación básica con Tomcat y Jersey

Ahora, una vez creada la aplicación y colocados todos los archivos vamos a crear las clases con las anotaciones básicas para la creación de la aplicación. En este ejemplo sólo se crea un recurso del API, un GET. El resto de anotaciones se implementan en la aplicación y se han explicado en el apartado Anotaciones de Java con Jersey. Se crea también una clase llamada API dentro del paquete “es.gva.san.lafe.edoz” e implementamos el código con las anotaciones JAXRS.



**Figura 31: Creación de clase para la API**



**Figura 32: Creando clase para API, añadiendo datos**

Se implementa así una única acción GET con una ruta y una negociación de contenido. Consiguiendo que se pueda entregar al cliente una respuesta en varios formatos distintos.

```
import javax.ws.rs.*;
import es.gva.san.lafe.edoz.app.*;

// API rest pública de la aplicación servidor

@Path("/")
public class PublicAPI {
    @Path("/test")
    @GET
    @Produces({"application/json"})
    public String getTest() {
        // conexión con la aplicación
        String pattrn = "{ \"woow\": \"%s\"}";
        return String.format(pattrn ,(new
ConnectionDB().test()));
    }
    @Path("/test")
    @GET
    @Produces({"text/plain"})
    public String getText() {
        // conexión con la aplicación
        return (new ConnectionDB.test());
    }
    @Path("/test")
    @GET
    @Produces({"application/xml"})
    public String getXML() {
        // conexión con la aplicación
        String pattrn = "<etiqueta>%s</etiqueta>";
        return String.format(pattrn ,(new ConnectionDB().test()));
    }
}
```

*Código 10: Código de implementación de Clase del API con negociación de contenido*

Finalmente tendremos que implementar un “hook” entre la aplicación y el contenedor de aplicaciones Tomcat. Este archivo le dice al framework Jersey dentro de qué paquetes tiene que mirar para encontrar las anotaciones JAX-RS.

```
import javax.ws.rs.ApplicationPath;
import org.glassfish.jersey.server.ResourceConfig;
// Esta clase sirve como enlace entre la aplicación rest y tomcat

@ApplicationPath("/rest")
public class Enlaceservidor extends ResourceConfig {
    public Enlaceservidor() {
        packages("es.gva.san.lafe.edoz.api");
    }
}
```

*Código 11: Código de enlace entre la aplicación y el servidor*



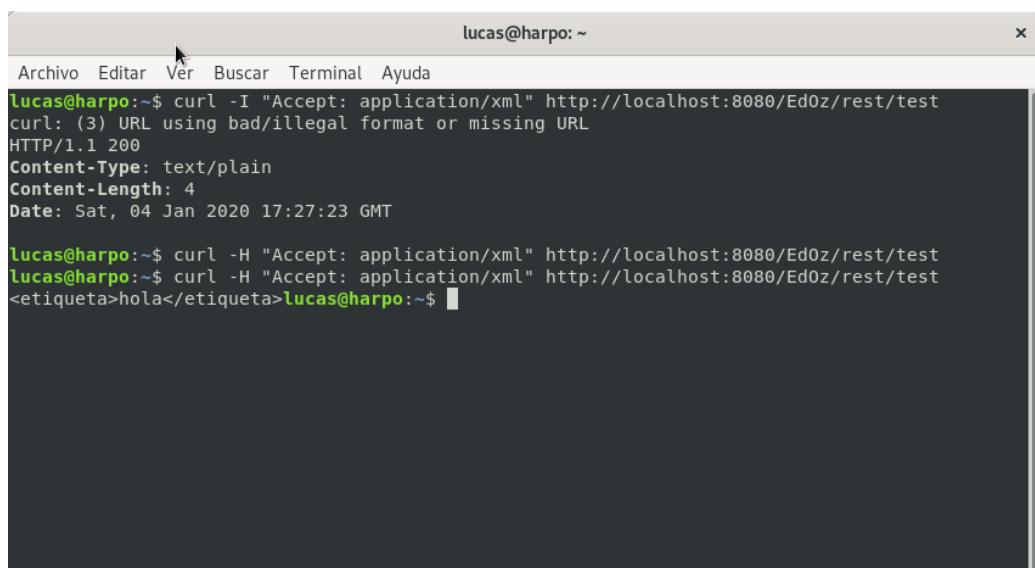
### **3.5.3. Comprobación básica del funcionamiento de la aplicación**

Una vez creado esto ya podemos comprobar con el comando “curl” que la negociación de contenido es la idónea. Arrancamos la aplicación usando el servidor Tomcat y en la terminal de linux se introduce el siguiente código.

```
$ curl -H "Accept: application/xml" http://localhost:8080/EdOz/rest/test
```

*Código 12: Test básico con el comando curl - get con cabecera*

También se pueden utilizar “Accept: applicatioon/xml” o “Accept: text/plain” para comprobar que se obtienen los resultados de formato correctos.

A screenshot of a terminal window titled "lucas@harpo: ~". The window contains two separate curl command executions. The first execution shows an error message about a bad URL format. The second execution shows the correct response, which includes XML data: '<etiqueta>holo</etiqueta>'. The terminal has a standard Linux-style interface with a menu bar at the top.

*Figura 33: Respuestas obtenidas con el comando "curl"*

En este caso el comando curl hace de cliente web al que se le sirven datos en xml. Bajo estos conceptos la aplicación crece exponencialmente en cantidad de servicios y funcionalidades y poco a poco se van construyendo el servidor y el cliente. Obteniendo resultados como los que se ven en el vídeo de demostración de la presentación o en las imágenes del anexo.

### **3.5.4. Alternativas de testeo de la aplicación**

Una alternativa muy efectiva para el testeo de la aplicación puede ser crear un cliente web en java utilizando una librería sencilla y asociar una batería de test con JUnit. Este tipo de testeos son habituales y mucho más precisos. No obstante, se debe emplear más tiempo para crear este tipo de tests y aunque son muy útiles este empleo extra de tiempo no es bien venido en programas sencillos que se quieran poner en producción en poco tiempo.



### 3.6. Login y Seguridad en la Aplicación

Una vez creado el servidor y el cliente, hay que proteger la aplicación. Para ello se pueden utilizar numerosas técnicas y metodologías.

#### 3.6.1. Login con el Directorio Activo

Como la aplicación se va a utilizar en la intranet del Hospital, se decidió utilizar el Directorio Activo como herramienta de login. Es decir, mediante el usuario y la contraseña de cada uno de los profesionales se valida la identidad del mismo y se obtienen sus datos. En función de esto, la aplicación muestra diferentes roles de usuario, con sus diferentes acciones a realizar.

Para realizar esto fue necesario aprender el lenguaje de consultas LDAP, y utilizar un recurso JNDI para la conexión y consulta del Directorio Activo. Para el entorno de producción se levantó un servidor Open LDAP y con el programa Apache Directory Studio se crearon los usuarios de prueba del mismo como se aprecia en la siguiente imagen.

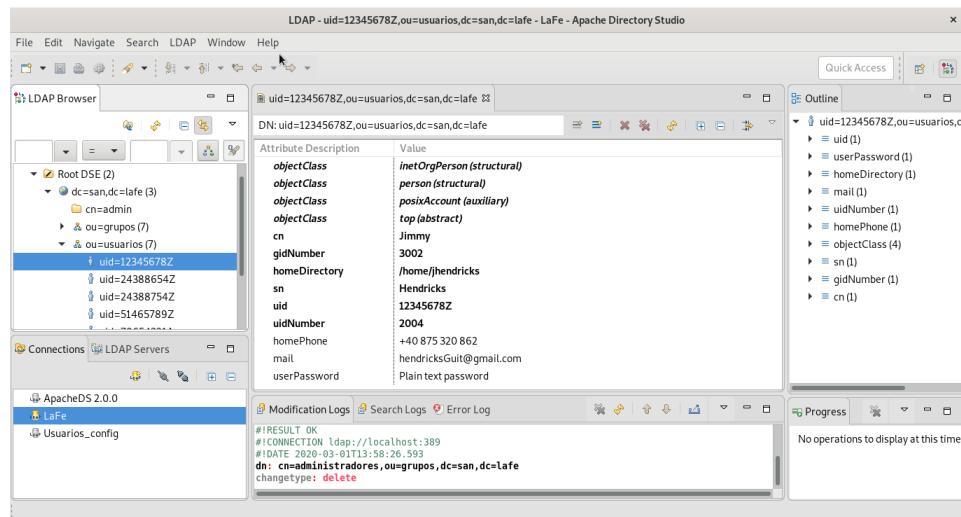


Figura 34: Usuarios en Open LDAP con Apache Directory Studio

Para el acceso al login, se creó una interfaz de acceso (Fig.35) que envía el usuario y la contraseña a la aplicación servidor. Esta, comprueba que el usuario y la contraseña existen y son coincidentes y entonces crea un JWT<sup>22</sup> que contiene los datos de identificación del usuario (Figs. 36,37). Devuelve el JWT firmado digitalmente al cliente que tendrá que incluirlo en las cabeceras de las peticiones HTTP que realice a recursos protegidos.

Cuando se envíe una petición a un recurso protegido del servidor, en caso de no tener los permisos, o en caso de que el JWT no esté validado<sup>23</sup> se devuelve una respuesta de no autorizado y no se realizan las acciones relacionadas a dicho recurso. El código de conexión con el servidor Open LDAP se puede ver en el anexo.

22 JSON Web Token

23 La comprobación de la firma falle.





**Figura 35: Interfaz de login**

### Administración de Cursos

ID	Título	Descripción	F.Crea..	F.Inic..	F.Fin	Profesor	Acciones

Toolbar: Inspector Consola Depurador Editor de estilos Rendimiento Memoria Red Almacenamiento Accesibilidad Adblock Plus

Filtros: Filtrar salida

Router Event: NavigationStart

```
> Object { dni: "514657892", paswd: "camarones_en_vinagre" }
> eyJraWQoIjJzZWNyZXQiJhbGciOiJSUzI1NiJ9
> .eyJpc3Mi0iJib3NwaXRhbCBVbml2ZXJzaXRhcmkgaSB0b2xpMoY25pYyAtIEZlIiwiYXVkiJoiQXBSawNHY2nDs24gZnJvbNrbmQgcXViIHNlIGN
vbmVjdGUgYSBhckGkIcIleHA1oJE100M2n10Mj0sImp0asIEInlcEVR3RzNqV1koVc2MjwzUVMwCilCjpxYQ010E100M2Njg4Mj0sImSiZlI6MTU4MjY200cwNhCwic3Vi1joiVGPrZWhgZGUGYXV0b3Jp
emFjacOzbibKzSB1c3VhcmIvIwzG5pIjoiNTE8NjU30DlaIiwiibmtynJ1Ijoi02FtYXLDs241LCJhcGvsbG1kb3M1o1JEZ5b5ySBlJc2xhIiwiawQ10jE21CJy2w1ojeIsImVtYwlsIjoiY2FtYXJvbkbNbWFpb
C5jb20ifQ.RurJ-F4x2ZvhMIAV-mwzoqffynnCJAx0150E8JaPrZMeFH1HfcaeCs71fcjMVhpFhKXlp34sJAzcDfiiigJnpWMR-EeVwfRTXARioVag7eqv6zRIsaWeLhZm4rqz1eAkK33sZta0R-
xXntbXKKIE4_YBiuv00fQ48I0a8zsWsz_6FXPH05Pe3km25BRc_aFY62LJY9EHTcfMXIoIdZ4MHRKEuJhs1VQGbzPduTpkv0joy9qlYgyst6a51iobUSdmkIvcn-
qUy0exU_ciDeGmD8rlw4EZ_5G_jc18J106tb_wa2S4f13A03eVE5ttCEsuLeUAOkpxN0Ld08Tw"
```

**Figura 36: Consola web de firefox con el JWT devuelto del servidor**

**Figura 37: Decodificación del JWT en la web JWT.io**



### **3.6.2. Asegurar nodos o “endpoints” en REST**

Otra acción interesante a desarrollar en un proyecto REST es la protección de los nodos o “endpoints” del servicio REST. Si no se aseguran estos nodos, cualquiera con un poco de paciencia podría alterar nuestra base de datos usando un programa como curl o httpget.

Existen varias formas de realizar acciones de asegurado de nodos en el servidor, cada una de ellas sirve para proteger ante ciertas amenazas.

La opción principal que se utilizó en la empresa fue la de utilizar un servidor proxy reverso a modo de intermediario. Este servidor comprueba si las peticiones al servidor (a nuestro Tomcat) vienen o no de dentro de nuestra red. Si esta condición se cumple y son internas, se permite el acceso, en caso contrario se bloquea.

No obstante, este proceso funciona exclusivamente con peticiones externas. Si hubiera un atacante infiltrado en nuestra propia red deberíamos utilizar otro tipo de técnicas. La más sencilla de implementar sería utilizar un filtro<sup>24</sup> que compruebe los permisos y credenciales del “JSON Web Token” y en función de los resultados de acceso o no a los recursos. Se suele también crear una anotación Java para aplicar ese filtro sobre los diferentes nodos. Así se podría realizar la misma comprobación sobre múltiples recursos y aplicar diversos filtros para asegurarlos como en el ejemplo número 13.

```
@Path("/{id : \d+)/borrar")
@DELETE
@Segura
@Produces({"text/plain"})
public Response deleteCurso(@Context HttpHeaders headers,@PathParam("id")
String
id) {
    SrvcCursos srvCurs = new SrvcCursos();
    return srvCurs.borraCurso(headers, id);
}

// Con la anotación sólo se borra el curso si el filtro lo permite
```

*Código 13: Código de uso del filtro sobre nodo REST*

### **3.6.3. Trampas, HoneyPots en el login y los formularios**

Otra acción muy común para las aplicaciones REST es utilizar honeypots en los formularios y realizar comprobaciones de los mismos en el servidor. Esta técnica se utiliza sobre todo para evitar el auto completado de formularios mediante bots o programas que saturan e incluso bloquean los servidores. Esta técnica es equivalente e incluso compatible con los llamados “Chapta” típico de las aplicaciones web.

Para implementar esta técnica, se crean una serie de campos vacíos ocultos al usuario y mediante los filtros que se vieron en el punto 3.6.2 se comprueba que los formularios trampa no han sido rellenados por el supuesto bot. De esta manera, se da y se quita el acceso a los recursos del servidor en función de los campos trampa.

---

24 Como en el ejemplo de código número 14



```
import java.io.IOException;
import javax.annotation.Priority;
import javax.ws.rs.*;
import es.gva.san.lafe.edoz.Segura; //importamos la anotación creada

// definimos la aplicación
@Segura
@Provider
@Priority(Priorities.AUTHENTICATION)
public class FiltroSeguridad implements ContainerRequestFilter {

    private static final String AUTHENTICATION_SCHEME = "Bearer";

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException{
        // Empieza el filtrado de autenticación
        // Conseguimos el objeto autorización de la cabecera http
        String authorizationHeader =
            requestContext.getHeaderString(HttpHeaders.AUTHORIZATION);
        // Validamos la cabecera
        if (!isTokenBasedAuthentication(authorizationHeader)) {
            abortWithUnauthorized(requestContext);
            return;
        }
        // Extraemos el token de autorización
        String token = authorizationHeader
            .substring(AUTHENTICATION_SCHEME.length()).trim();

        // Validamos el token
        try {
            validateToken(token);
        } catch (Exception e) {
            abortWithUnauthorized(requestContext);
        }
    }

    private void abortWithUnauthorized(ContainerRequestContext requestContext) {
        // Aborta y devuelve un código 401
        requestContext.abortWith(
            Response.status(Response.Status.UNAUTHORIZED)
                .header(HttpHeaders.WWW_AUTHENTICATE, AUTHENTICATION_SCHEME)
                .build());
    }

    .... // el resto de comprobaciones se realizan siguiendo el mismo esquema
}
```

*Código 14: Filtro de bloqueo de nodos REST*



### **3.7. Implementación del cliente Web**

Para la creación del cliente se utilizaron una serie de lenguajes y tecnologías que se describen y explican brevemente a continuación.

#### **3.7.1. Angular**

Es el “front-end” o interfaz de usuario. Es el framework que se utiliza para generar los objetos con los que el usuario interactua. Es un pequeño ecosistema en sí mismo y necesita de otras tecnologías para poder funcionar o explotarlo en su totalidad<sup>25</sup>.

#### **3.7.2. Angular CLI**

Es la interfaz de comandos del framework. No es necesario usarla para trabajar con el framework, pero sí nos ahorra mucho trabajo generando archivos, código y compilando o levantando servidores de pruebas.

#### **3.7.3. NodeJS**

Es el entorno sobre el que están construido Angular CLI y otras dependencias que podríamos usar para realizar el proyecto. Es como si fuera nuestra terminal bash de linux, el motor o ejecutor de código.

#### **3.7.4. NPM**

El Node Package Manager que nos servirá para gestionar esas dependencias. Sería similar a lo que es el comando apt, yum, rpm o pacman de las diversas distribuciones linux. Nos sirve para instalar o añadir funcionalidades, comandos o librerías a nuestro entorno de trabajo Node.

#### **3.7.5. TSC**

TSC es el compilador de TypeScript creado por Microsoft. Este “compilador” lo que realiza es convertir el lenguaje TypeScript a lenguajeJavaScript que es entendible por cualquier navegador o por el entorno de ejecución Node.

#### **3.7.6. WebPack**

Es un constructor automatizado. Es la herramienta que gasta el framework de Angular para construir nuestro proyecto. Utiliza todos nuestros scripts, los combina y los minifica<sup>26</sup>. Es también el responsable de recargar la aplicación cada vez que realizamos un cambio en nuestro código<sup>27</sup>.

#### **3.7.7. Editor de texto**

Se puede utilizar cualquier editor de texto para crear la interfaz de nuestra aplicación. No obstante se recomienda usar Visual Studio Code, particularmente el fork VSCodium de código corregido y que proporciona funcionalidades específicas para TypeScript de manera nativa. No obstante, hay numerosos [plugins](#) y accesorios que se pueden añadir a los editores más conocidos para lograr las mismas características de resaltado de sintaxis y auto completado que las logradas en VSCodium.

---

25 Para más datos sobre cómo desarrollar un proyecto con Angular se puede consultar el Anexo D.

26 La minificación en javascript se utiliza para optimizar recursos y se trata de eliminar los saltos de línea tabulaciones y demás signos introducidos en el código para llevarlo a la mínima expresión. El resultado es una única línea de código entendible por el computador y que se ejecuta más rápido. Es una optimización.

27 En inglés este término se conoce como Hot Module Replacement (HMR).



### **3.7.1. Ejemplo de creación de un cliente con Angular 2**

Para crear un cliente web usando Angular 2 o cualquiera de sus versiones posteriores, es necesario seguir una serie de pasos sencillos que describimos a continuación.

#### **3.7.1.1. Crear la aplicación**

Creamos la aplicación usando la consola. Para ello primero creamos una carpeta donde crearemos nuestros proyectos.

```
$ mkdir proyectos_ang
```

*Código 15: Crea un directorio nuevo*

Luego, tras responder a las preguntas sobre nuestro proyecto<sup>28</sup>, instalamos la aplicación de consola de Angular en caso de no tenerla instalada.

```
$ npm install -g angular-cli
```

*Código 16: Instala el módulo angular-cli de nodejs*

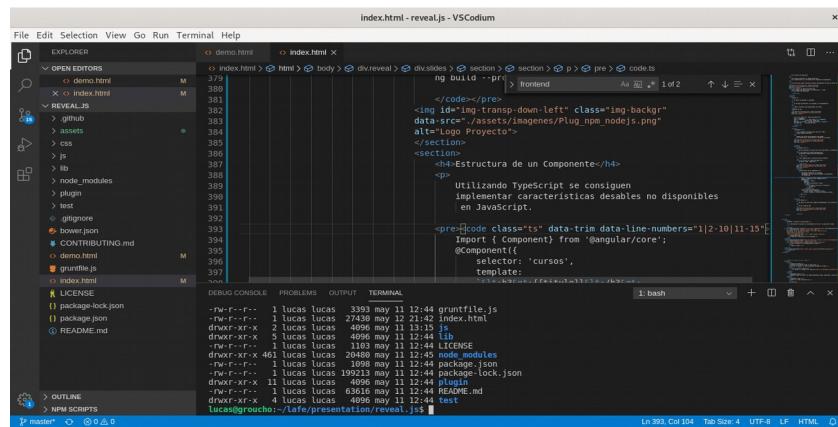
A continuación utilizamos el comando “ng”<sup>29</sup> para crear el proyecto, este proceso suele tardar, pues instala todos los paquetes necesarios para que nuestra aplicación funcione.

```
$ ng new miApp
```

*Código 17: Crea la estructura de un proyecto Angular*

#### **3.7.1.1. Desarrollar el código**

Ahora ya podemos abrir nuestro proyecto con el editor de texto VSCode. Al abrirlo observamos una serie de ficheros<sup>30</sup> que nos servirán para trabajar con las diferentes partes del framework.



*Figura 38: Ejemplo de VSCode funcionando*

28 Puedes consultar el funcionamiento de npm en el anexo B.

29 Más información sobre cómo se estructura y funciona Angular 2 en e anexo D.

30 Para ver la estructura de ficheros acudir al anexo D.iv.a.



Tras ello, en la terminal de VSCodium le pedimos a Angular-cli que nos cree una plantilla de un componente.

```
$ ng g component ./components/micomponente
```

*Código 18: Crea un componente en Angular 2*

Como vamos a conectarlo con la base de datos crearemos también un servicio.

```
$ ng g service ./services/miservicio
```

*Código 19: Crea un servicio en Angular 2*

Ahora dentro del servicio usaremos el módulo HttpClient y lo enlazaremos al servidor como se indica en el anexo D.iv.e. Este tipo de conexión que gestionamos como un servicio, es la encargada de traernos los datos al cliente (frontend) para poder colocarlos en la página web usando las diferentes herramientas del framework. Así, como se puede ver en el anexo dedicado a Angular, nos proporciona varias formas de formas de realizar esto. Para nuestro componente lo mejor sería utilizar un bucle for como se puede apreciar en el código de ejemplo número 80.

### **3.7.1.2. Testear la aplicación**

Básicamente para el testeo de este tipo de aplicaciones se usan dos tipos de pruebas y diferentes paquetes de nodeJs:

- Para las pruebas Unitarias<sup>31</sup>, se utiliza Karma y Jasmine. Síplemente hay que escribir el código<sup>32</sup> de testeo de los componentes en sus respectivos archivos y ejecutar el siguiente comando.

```
$ ng test
```

*Código 20: Ejecutando Tests Unitarios con Angular*

Karma es el motor, es el que ejecuta los test y nos los presenta en la consola y en el "Jasmine HTML Reporter" que se muestra en las siguientes imágenes .

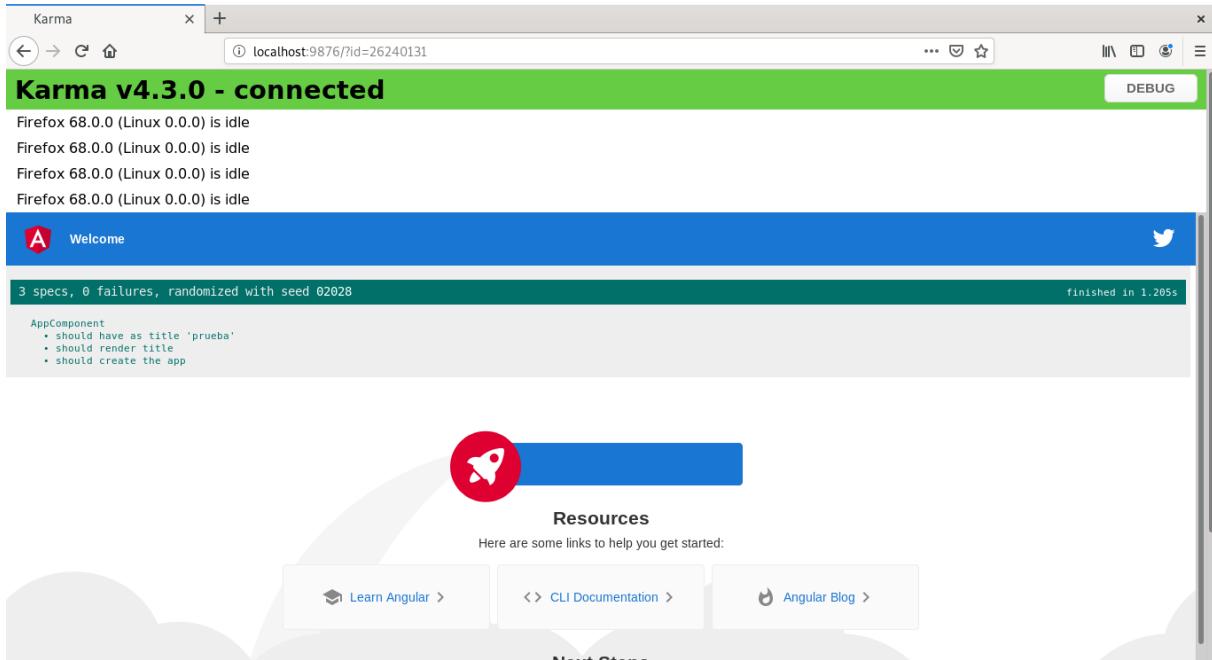
Para que karma funcione en otros navegadores hay que instalar el "launcher" mediante npm y añadirlo tanto a browsers como a plugins del archivo karma.config.js

---

<sup>31</sup> Conocidos como Unit Test, sirven para testear ciertas funciones, áreas o partes del código.

<sup>32</sup> La forma correcta de hacer esto usando Jasmine se puede ver en la documentación del [paquete](#) o en la documentación de [Angular](#).





**Figura 39: Karma ejecutando tests y mostrando resultados en el navegador firefox**

```
lucas@groucho:~/sandbox/prueba/prueba$ ng test
1% building 2/2 modules 0 active
17 05 2020 17:45:43.294:INFO [karma]: No captured browser, open http://localhost:9876/
17 05 2020 17:45:43.295:INFO [launcher]: Karma v4.3.0 server started at http://0.0.0.0:9876/
17 05 2020 17:45:43.295:INFO [launcher]: Launching browsers Firefox, FirefoxDeveloper, FirefoxAurora, FirefoxNightly with concurrency unlimited
17 05 2020 17:45:43.302:INFO [launcher]: Starting browser Firefox
17 05 2020 17:45:43.340:INFO [launcher]: Starting browser FirefoxDeveloper
17 05 2020 17:45:43.368:INFO [launcher]: Starting browser FirefoxAurora
17 05 2020 17:45:43.395:INFO [launcher]: Starting browser FirefoxNightly
17 05 2020 17:46:06.986:WARN [karma]: No captured browser, open http://localhost:9876/
17 05 2020 17:46:09.645:INFO [Firefox 68.0.0 (Linux 0.0.0)]: Connected on socket QadgNRJF-vddY9elAAAA with id 93001203
17 05 2020 17:46:09.863:INFO [Firefox 68.0.0 (Linux 0.0.0)]: Connected on socket Jgln087a-IHsstqAAAB with id 76335840
17 05 2020 17:46:09.946:INFO [Firefox 68.0.0 (Linux 0.0.0)]: Connected on socket wUqmMoOpj9Mt_pjWAAC with id 19899269
17 05 2020 17:46:10.103:INFO [Firefox 68.0.0 (Linux 0.0.0)]: Connected on socket SavNaXJN7TsVInzNAAAD with id 26240131
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.169 secs / 1.028 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.327 secs / 1.155 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.283 secs / 1.178 secs)
Firefox 68.0.0 (Linux 0.0.0) ERROR
  DisconnectedClient disconnected from CONNECTED state (transport close)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.169 secs / 1.028 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.327 secs / 1.155 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.283 secs / 1.178 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.206 secs / 0.991 secs)
Firefox 68.0.0 (Linux 0.0.0) ERROR
  DisconnectedClient disconnected from CONNECTED state (transport close)
17 05 2020 17:46:53.519:ERROR [launcher]: FirefoxDeveloper crashed.

17 05 2020 17:46:53.519:ERROR [launcher]: FirefoxDeveloper stdio:
17 05 2020 17:46:53.520:ERROR [launcher]: FirefoxDeveloper stderr:
17 05 2020 17:46:53.564:INFO [launcher]: Trying to start FirefoxDeveloper again (1/2).
17 05 2020 17:46:59.352:INFO [Firefox 68.0.0 (Linux 0.0.0)]: Connected on socket mVv4yD6kMawa_WjUAAAE with id 26240131
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.604 secs / 1.429 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.692 secs / 1.49 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.772 secs / 1.555 secs)
Firefox 68.0.0 (Linux 0.0.0): Executed 3 of 3 SUCCESS (1.282 secs / 0.934 secs)
TOTAL: 12 SUCCESS
TOTAL: 12 SUCCESS
^Clucas@groucho:~/sandbox/prueba/prueba$
```

**Figura 40: Karma mostrando el resultado de los tests en la consola**



- Para las pruebas de punto a punto<sup>33</sup>, se utiliza Protractor y Jasmine. Se debe importar algunas funcionalidades del módulo protractor y colocar los tests de Jasmine envolviendo estas funcionalidades. Estas pruebas, a diferencia de los test unitarios se colocan en el directorio e2e del proyecto Angular. Los resultados esta vez los obtenemos solamente por consola. Para ejecutar los test e2e hay que usar el siguiente comando.

```
$ ng e2e
```

*Código 21: Ejecutando Tests de navegación con Angular*

### **3.7.1.3. Construir la aplicación**

Finalmente cuando tengamos la aplicación desarrollada y lista para desplegar usaremos el comando siguiente.

```
$ ng build --prod --base-href ./
```

*Código 22: Construye la aplicación*

Este comando nos construye la aplicación en una carpeta llamada “dist” dentro de nuestro directorio. La opción “--prod” indica a angular que queremos que cree el programa usando las opciones de producción que le indicamos en el archivo “environment”<sup>34</sup>. La otra opción “--base-href ./” le dice al constructor de angular la ruta donde se situa la aplicación en el servidor, si esto no se configura bien al acceder a la aplicación en el servidor obtendremos diversos fallos en la descarga de los archivos necesarios para el funcionamiento de nuestra aplicación.

### **3.7.1.4. Desplegar la aplicación cliente**

Para desplegar la aplicación en el servidor Tomcat, es decir para que nuestros usuarios puedan acceder a la misma es necesario desplegarla en la ruta que le indicamos al constructor de angular-cli. Desplegar una aplicación Angular en Tomcat es tan sencillo como copiar los archivos de dentro de la carpeta “dist” y pegarlos en la ruta indicada de nuestra aplicación web hecha en Java.

Recordamos que el directorio raíz de nuestra aplicación está situado en la carpeta WebContent del proyecto web creado en eclipse.

Ahora, para acceder a la aplicación sólo habría que entrar en la ruta principal de nuestro Servlet de Java. Si arrancamos el servidor Tomcat de prueba y acudimos a la ruta “localhost:8080/MiAppDeTomcat/” obtendremos la aplicación cliente en el navegador.

---

<sup>33</sup> Conocidas como End to End Tests o e2e, ejecutan la aplicación en el navegador y simulan el comportamiento del usuario.

<sup>34</sup> Más información en el apéndice D.iv.a



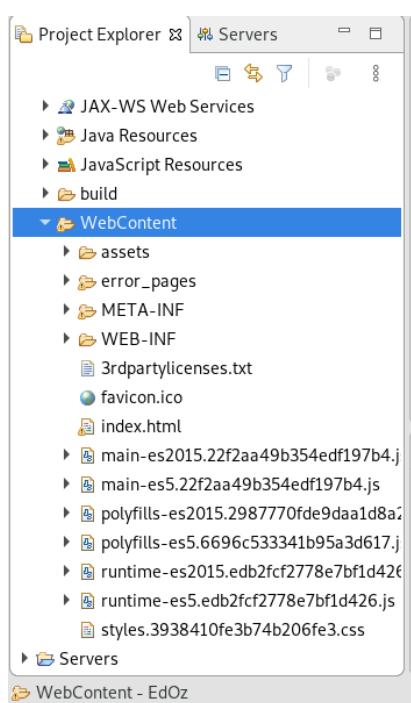


Figura 41: Aplicación cliente situada en la raíz del servlet

Para lograr que esta aplicación funcione con el módulo de rutas que proporciona Angular es necesario configurar el Servidor Tomcat y redireccionar todas llamadas al servidor (las que no sean puntos de acceso REST) hacia el archivo index.html del cliente Angular.

Esto se realiza a través del fichero web.xml que como ya se sabe sirve para indicar el punto de entrada al servidor (index.html) los recursos JNDI como la base de datos o el servidor mail, el accesos al servidor desde otros servidores remotos, los métodos permitidos, las cabeceras, las redirecciones o incluso las páginas de error que mostrará el servidor. Se puede ver un ejemplo de archivo de configuración en las páginas 102 y 103.

Es esta la forma más sencilla de desplegar la aplicación Angular en un servidor Tomcat aunque no la única, puesto que es en estas fases donde toma mayor relevancia la automatización de tareas vía npm, ant, gradle, grunt u otros programas. Incluso se puede hacer uso de eclipse que nos permite ejecutar acciones y comandos en la consola antes y después de construir la aplicación.

### 3.7.1.1. Construir el Servlet

Este es el último paso. Una vez ya se tiene la aplicación testeada y funcionando en producción, se debería construir un archivo "WAR" que aglutine tanto la aplicación cliente, como la aplicación servidor o servlet.

Para realizar esta tarea sólo hay que dirigirse al menú superior del editor Eclipse y en "File" pulsar sobre "Export". También obtendremos los mismos resultados si se hace click derecho sobre el proyecto que queremos convertir y pulsamos sobre "Export" en el menú contextual.

Luego, este war se coloca en el directorio de aplicaciones del servidor Tomcat de preproducción donde será testeado de nuevo y se comprobarán que todas las funcionalidades de la aplicación funcionan como es debido.



## 4. EVALUACIÓN Y CONCLUSIONES FINALES.

### 4.1. Evaluación

El proyecto se finalizó con éxito. Se cumplieron los objetivos no obstante cabe destacar las siguientes conclusiones.

#### 4.1.1. Análisis del Servidor

Se realizaron pocas pruebas<sup>35</sup> de carga de datos y rendimiento contra la aplicación Servidor, y aunque estas dieron bastantes buenos resultados, no obstante, se piensa que la elección de una base de datos relacional no fue la más indicada. Esto es debido a que con este tipo de base de datos hay que realizar diversas consultas en las tablas para ir construyendo los objetos del DAO. Si tenemos presente que las conexiones a la base de datos siempre son lo más costoso en recursos, se cree que se deberían realizar las mínimas consultas posibles.

Se piensa que dado que el programa tiene una estructura sencilla, un mejor acercamiento o mejora de esto sería la utilización de una base de datos noSQL como MongoDB o usando las llamadas “bases de datos orientadas a objetos” ayudarían y mejorarían el rendimiento de la aplicación. No obstante, este cambio se podría llevar a cabo en poco tiempo gracias a la aplicación del patrón de diseño DAO.

También se cree que se podrían utilizar gestores de dependencias como Maven o Graddle para la gestión de dependencias del proyecto, no sólo consiguiendo un ahorro de tiempo en la búsqueda e integración de las diferentes librerías que integra el proyecto sino también en la gestión de la versión y posibles actualizaciones de las librerías. Esto último es muy importante para mejorar la seguridad y mantenibilidad de cualquier proyecto.

#### 4.1.2. Análisis del Cliente

A la hora de analizar los datos del cliente, también se observa una buena respuesta en cuanto a tiempos y carga de datos. Se cree pues acertada la elección del framework que a pesar de resultar complicado en cuanto al aprendizaje, facilita enormemente el desarrollo y obliga a crear un código más manejable, entendible y con menos fallos que el tradicional JavaScript.

En cuanto al rendimiento, aunque actualmente se dice que es un framework bastante lento, esto va a cambiar en las siguientes versiones dándonos la posibilidad de actualizar la aplicación realizando cambios mínimos en el código.

#### 4.1.3. Posibles mejoras

Se pueden realizar infinidad de mejoras a la aplicación, desde intentar mejorar el rendimiento de la misma, refactorizar el código para que al recibir una petición el servidor ejecute un menor número de acciones, hasta añadir nuevos campos en la base de datos y crear nuevas funcionalidades. No obstante, siendo una aplicación relativamente sencilla, el margen de mejora en cuanto a nuevas funcionalidades básicas es ciertamente limitado y se cree que ya se ha alcanzado el estado básico de funcionalidades, es decir, se puede trabajar perfectamente usando sólo lo creado.

---

<sup>35</sup> Pruebas basadas en lanzar scripts con cientos de peticiones al servidor con el comando curl y pruebas unitarias usando JUnit y un cliente web básico en Java.



## **4.2. Conclusiones finales**

Aplicando lo aprendido en clase, en las prácticas y demás, se piensa que el resultado obtenido fue satisfactorio. Se consiguieron los objetivos, y se creó la aplicación que ahora mismo se encuentra en un entorno de Pre-Producción.

No obstante, cabe destacar la dificultad manifiesta de implantar un cierto grado de seguridad en este tipo de aplicaciones cliente-servidor. Es importante seguir las recomendaciones y los procedimientos de las diversas metodologías que se pueden encontrar, pues es muy sencillo, crear agujeros o fallos de seguridad que se podrían aprovechar para borrar, robar o manipular los datos de los usuarios.

Se cree que con la elaboración de este proyecto se aprendieron a usar numerosas tecnologías, pero sobre todo, se aprendió sobre arquitectura computacional. Se aprendió que es de enorme importancia diseñar y estructurar bien una aplicación a fin de poder ser creada, mantenida y mejorada con el tiempo.



¡FIN!



## BIBLIOGRAFÍA

ng-book The Complete Book on Angular 4. Murray Nate, Coury Felipe, Lerner Ari, and Taborda Carlos. 2017. San Francisco, California

Angular 2 From Theory To Practice. Hussain Asim. 2016. S.P. S.ISBN

Pro Angular, Second Edition. Freeman Adam. 2017. MILTON KEYNES MK6 3PA, United Kingdom. ISBN-13 (pbk): 978-1-4842-2306-2

Pro TypeScript: Application-Scale JavaScript Development. Fenton Steve. 2018. United Kingdom. ISBN-13 (pbk): 978-1-4842-3248-4

Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS. Sanjay Patni. Santa Clara, California, USA. 2017.ISBN-13 (pbk): 978-1-4842-2664-3.

Developing RESTful Web Services with Jersey 2.0. Sunil Gulabani. 2014. Birmingham B3 2PB, UK. ISBN 978-1-78328-829-8

The Complete log4j Manual. Ceki Gülcü. 2002. S.P. S.ISBN

Data Visualization with JavaScript. Stephen A. Thomas. 2015. USA. ISBN-10: 1-59327-605-2, ISBN-13: 978-1-59327-605-8

Apache Log4j 2 v. 2.13.0 User's Guide. Apache Software Foundation. 2019. <http://logging.apache.org/log4j/2.x/manual/index.html>

Documentación Oficial del Proyecto Apache Tomcat. S.F. The Apache Software Foundation. <https://tomcat.apache.org/tomcat-9.0-doc/>

Respuestas sobre Java, TypeScript, Angular 2 y Apache Tomcat. S.F. StackOverflow. <https://stackoverflow.com/questions>

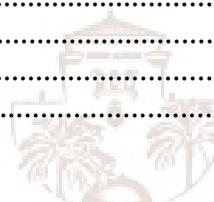
Documentación Oficial del Proyecto Eclipse. 2019. Eclipse Foundation. <https://help.eclipse.org/2019-12/index.jsp>

Documentación Oficial del Framework Angular. S.F. Google. <https://angular.io/docs/>

Páginas del Manual de Linux, vía online y vía terminal. S.F. Linux. <https://linux.die.net/man/>

## ÍNDICE DE FIGURAS

Diseño de interfaz de usuario para la edición de cursos.....	7
Diseño de interfaz de usuario básica de gestión de cursos.....	8
Diseño de interfaz de usuario para la inscripción al curso o jornada.....	9
Vista general estrategia a 4.....	10
Vista general estrategia a 2.....	10
Modelo de datos usando un acercamiento a 4.....	11
Posible aproximación con almacenamiento de html en crudo.....	11
Diseño lógico final.....	12
Estructura cliente-servidor.....	13
Negociación de formatos (Pro Restfull APIs).....	14
Diagrama de rutas del API REST.....	16
Ejemplo de documentación html del API con Spectacle.....	18
Esquema estructura interna del servidor.....	19
Diagrama de estructura de paquetes de la aplicación.....	20
Creando tablas con SquirreL SQL.....	21
Revisando la estructura con SquirreL SQL graph pluggin.....	21
Diagrama de relación del modelo de datos.....	22
Resultados de búsqueda en SQuirreL SQL.....	25
Resultados tras la eliminación del curso.....	25
Mejora del modelo.....	26
Creando Script de Tablas con SquirreL SQL.....	27
Estructura de aplicación web Jersey.....	29
Esquema de DAO de una clase cualquiera (TutorialsPoint).....	32
Creando proyecto eclipse.....	33
Seleccionando proyecto web dinámica.....	33
Añadiendo datos al proyecto web.....	34
Añadiendo directorios al "Build Path".....	34
Confirmando la creación del fichero web.xml.....	35
Añadiendo librerías Jersey.....	35
Añadidos archivos Jersey.....	35
Creación de clase para la API.....	36
Creando clase para API, añadiendo datos.....	36
Respuestas obtenidas con el comando "curl" .....	38
Usuarios en Open LDAP con Apache Directory Studio.....	39
Interfaz de login.....	40
Consola web de firefox con el JWT devuelto del servidor.....	40
Decodificación del JWT en la web JWT.io.....	40
Ejemplo de VSCode funcionando.....	44
Karma ejecutando tests y mostrando resultados en el navegador firefox.....	46
Karma mostrando el resultado de los tests en la consola.....	46
Aplicación cliente situada en la raíz del servlet.....	48
Ejecución de programa en NodeJS.....	55
Ejecución de programa con módulos.....	56
Inicializando un proyecto con npm.....	58
Árbol de componentes en Angular.....	70
Lista de directorios y ficheros creados con angular-cli.....	71
Lista de cursos usando ngFor.....	76
Renderizado de ngIf antes de pulsar el botón.....	77



Renderizado de ngIf después de pulsar el botón.....	77
Vista del entorno de trabajo del editor Swagger.....	80
Exportando documentación del API con el editor Swagger.....	81
Ejemplo de documentación generada con el editor Swagger.....	81
Ejecutando Swagger en modo local.....	82
Padre Palomino.....	89
Edna Carapápel.....	89
Modelo de datos usando un acercamiento sencillo.....	90
Possible acercamiento intermedio.....	90
Estructura rechazada por unir las inscripciones y los campos, creando posibles problemas.....	91
Abriendo la configuración del servidor tomcat.....	92
Configuración del servidor tomcat en eclipse.....	92
Incluir archivos .jar a nivel de aplicación.....	93
Despliegue de html en proyecto web tomcat-eclipse.....	93
Botones para uso del ngSwitch.....	94
ngSwitch pulsado el primer botón.....	94
ngSwitch pulsado el segundo botón.....	94
ngSwitch pulsado el último botón.....	95
Herramientas de desarrollo de firefox, reenviando petición.....	96
Cabeceras de envío de petición http.....	96
Edición de cabeceras y cuerpo de petición http con firefox.....	96



## A. ANEXO DE NODEJS

### i. NodeJS nociones básicas

NodeJS es un entorno de ejecución de JavaScript. Es decir, no es un servidor, es un entorno de ejecución que nos permite desarrollos tanto de back-end, front-end, aplicaciones nativas o IOT (Arduino, etc.,).

Para desarrollar en Node, se suele usar JavaScript aunque ahora también se puede utilizar un compilador de TypeScript que convierta nuestro código a JavaScript. NodeJS está basado en el motor V8 del navegador Chrome. También hay una versión que implementa el motor Chakra de Microsoft pero no se utilizará en este documento. Node está pensado para realizar operaciones de E/S<sup>36</sup> sin bloqueos y orientado a Eventos. Es decir, que se pueden realizar desarrollos asíncronos mediante estos eventos.

Es un entorno muy liviano y eficiente para la realización de servicios Web, para el desarrollo de Sockets o para programar placas Arduino.

### ii. Instalación de NodeJS en Unix

Hay varias formas de instalar NodeJS en linux, pero hay que cerciorarse de usar una versión moderna. Este entorno, ha evolucionado muy rápido, y las últimas versiones incorporan características y sintaxis que no se dan en las antiguas, además de mejorar el rendimiento del propio entorno y eliminar bugs.

La instalación de Node en Debian 10 se puede realizar mediante el instalador de paquetes apt.

```
$ sudo apt update
```

*Código 23: Reiniciando la cache del gestor de paquetes deb*

Primero reiniciamos la memoria del gestor de paquetes. Es decir actualizamos la cache del programa.

```
$ sudo apt install -y nodejs
```

*Código 24: Instalando Node con el comando apt*

Instalamos la última versión estable disponible desde los repositorios oficiales de debian.

La instalación de NodeJS en Windows es también muy sencilla, simplemente hay que descargar el instalador y ejecutarlo como administrador.



### **iii. Hola mundo en NodeJS**

Este entorno de ejecución se asemeja mucho al intérprete de Python. Es decir, abrimos una terminal ya sea de windows o de linux y escribimos el comando “node” para obtener un intérprete.

Este intérprete nos es útil para probar sentencias o algún tipo de código sencillo, pero programar así, resulta muy farragoso y poco práctico.

Al igual que python, Node proporciona otra forma de ejecutar código. Esta forma se basa en introducir la ruta del archivo como parámetro al comando node.

Como ejemplo, se crea un archivo llamado miarchivo en el directorio de trabajo y se introduce el siguiente código.

```
console.log("<!--PROGRAMA--");
console.log("-----");
console.log("Hola NodeJS");</pre>
```

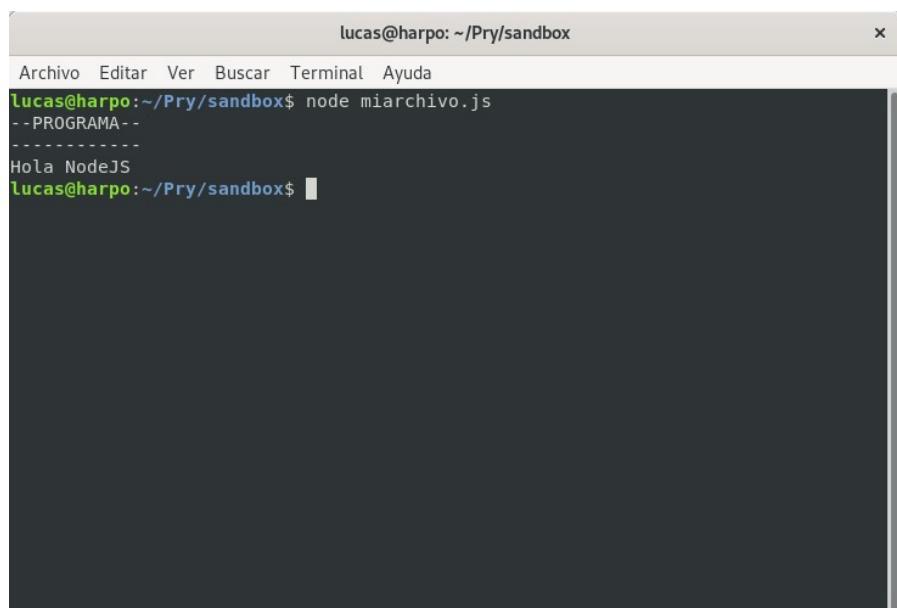
*Código 25: Hola mundo con JavaScript*

El código del cuadro 25 muestra el código JavaScript que se usa generalmente para imprimir texto por consola. Este código sirve para escribir en la consola del navegador, en el intérprete e nodeJS del que hablamos anteriormente o como programa secuenciado, ejecutando el siguiente comando en la terminal de UNIX.

```
$ node miarchivo.js
```

*Código 26: Ejecutando un programa con NodeJS*

El resultado de la ejecución del anterior comando es la siguiente.



The screenshot shows a terminal window titled "lucas@harpo: ~/Pry/sandbox". The window has a menu bar with "Archivo", "Editar", "Ver", "Buscar", "Terminal", and "Ayuda". The command entered is "node miarchivo.js". The output displayed is:

```
lucas@harpo:~/Pry/sandbox$ node miarchivo.js
--PROGRAMA--
-----
Hola NodeJS
lucas@harpo:~/Pry/sandbox$
```

*Figura 42: Ejecución de programa en NodeJS*



## iv. El API de NodeJS

El entorno de ejecución Node nos permite la realización de múltiples tareas, como son por ejemplo, el acceso a disco, creación de procesos hijos, grabación de archivos, lectura de bits de dispositivos, etc., Si se quiere tener una visión más detallada del mismo, cosa que se aleja del objetivo de este documento siempre se puede consultar la [API](#) de NodeJS.

## v. El concepto de módulos de NodeJS

NodeJs permite la división de programas en múltiples archivos y el uso de ese código en nuestros programas. Estos módulos o paquetes requieren de unas directivas especiales para sacar partido a esta funcionalidad.

Como ejemplo, se usarán dos archivos diferentes principal.js y modulo.js. En el archivo modulo.js incluimos el siguiente código.

```
var diLoQueQuiero= function(a){  
    return console.log("Lo que digo " + a);  
}  
  
module.exports.diLoQueQuiero = diLoQueQuiero;
```

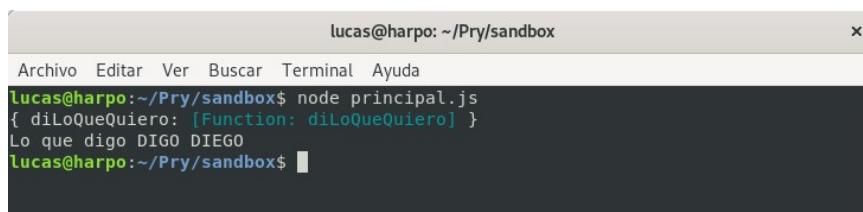
Código 27: Archivo modulo.js

Es necesario asignar al módulo el nombre de la variable que contiene la función. Así se puede invocar el módulo desde otro archivo y utilizar sus métodos. Para ello utilizamos las siguientes sentencias.

```
var imprime = require('./modulo.js');  
// para ver el objeto  
console.log(imprime);  
// para usar la función  
imprime.diLoQueQuiero("DIGO DIEGO");
```

Código 28: Archivo modulo.js

Así, finalmente se puede utilizar el comando “node principal.js” y obtendremos el siguiente resultado.



A screenshot of a terminal window titled "lucas@harpo: ~/Pry/sandbox". The window shows the command "node principal.js" being run. The output of the script is displayed, showing that the module "diLoQueQuiero" is an object with a function "diLoQueQuiero" assigned to it, and when called with the argument "DIGO DIEGO", it logs "Lo que digo DIGO DIEGO" to the console.

```
lucas@harpo:~/Pry/sandbox$ node principal.js  
{ diLoQueQuiero: [Function: diLoQueQuiero] }  
Lo que digo DIGO DIEGO  
lucas@harpo:~/Pry/sandbox$
```

Figura 43: Ejecución de programa con módulos

Observa que el módulo es un objeto JavaScript, porque está rodeado por los corchetes “{ }”. La función es asignada como una propiedad de dicho objeto.



Este es el concepto tan sencillo<sup>37</sup>, se puede volver algo complicado de manejar cuando se tienen múltiples dependencias, de múltiples archivos que a la vez tienen dependencias de otros módulos. Para solventar este problema se creó otra de las tecnologías que vamos a usar. NPM o “Node Package Manager”. Un gestor que nos ayuda en la gestión y manejo de dependencias.

---

<sup>37</sup> Aunque no se usarán, en la actualidad este concepto se ha diversificado, no sólo con ES6 (ECMASCRIPT 2015) y la inclusión de clases en el lenguaje, sino también con la directiva “imports” que añade otro tipo de funcionalidades y reusabilidad.



## B. ANEXO DE NPM

Node Package Manager o npm es el gestor de paquetes de node. Este administrador de paquetes no sólo nos permite gestionar las dependencias, sino que también nos provee de un repositorio con multitud de módulos que podemos utilizar con la misma facilidad con la que se instalan paquetes con “apt” en la distribución linux debian.

Este es uno de los programas más importantes que vamos a utilizar. Sirve para instalar Angular CLI así como los módulos de terceros que queramos añadir a Angular.

### i. Instalación de npm en UNIX

Al igual que node, se puede instalar npm de diversas manera. No obstante, la más sencilla es a través del gestor de paquetes “apt”.

```
$ sudo apt install -y npm
```

Código 29: Instalando NPM con el comando apt

### ii. Creación de un proyecto npm

NPM nos permite generar un proyecto, con todos sus características asociadas a un archivo llamado “package.json”. Para iniciar el proyecto tan sólo hay que introducir el comando siguiente.

```
$ npm init
```

Código 30: Iniciando un proyecto con npm

El comando “npm init” nos despliega un interfaz en el que iremos respondiendo preguntas importantes para el proyecto. Por ejemplo, el nombre del proyecto, la versión, la licencia, etc.,

```
Lucas@harpo:~/Pry/sandbox/npm-curs$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
See 'npm help json' for definitive documentation on these fields
and exactly what they do.

Press ^C at any time to quit.
package name: (npm-curs) mi-app
version: (1.0.0) 0.0.1
description: Es una aplicación iniciada con npm.
entry point: (index.js)
test command:
git repository:
keywords:
author: Lucas C.B.
license: (ISC) GPL
Sorry, license should be a valid SPDX license expression (without "LicenseRef"), "UNLICENSED", or "SEE LICENSE IN <filename>" and license is similar to the valid expression "GPL-3.0-or-later".
license: (ISC) GPL-3.0
About to write to /home/lucas/Pry/sandbox/npm-curs/package.json:

{
  "name": "mi-app",
  "version": "0.0.1",
  "description": "Es una aplicación iniciada con npm.",
  "main": "index.js",
  "scripts": {
    "test": "echo \\'Error: no test specified\\' && exit 1"
  },
  "author": "Lucas C.B.",
  "license": "GPL-3.0"
}

Is this OK? (yes)
```

Figura 44: Inicializando un proyecto con npm



Sobre este comando cabe aclarar unas cuestiones.

- Entry point → indica el archivo a ejecutar con node. Es el “main” de nuestra aplicación.
- Test command → si queremos especificar un grupo de comandos para lanzar test con npm.<sup>38</sup>
- git Repository → si deseamos indicar la url a un repositorio que albergue un proyecto controlado con git.

Es importante observar que npm sólo genera un archivo llamado “package.json”. Este archivo, proporciona “meta datos” no sólo a npm sino como veremos más a delante a otras muchas aplicaciones del entorno de node.

### **iii. Fundamentos de NPM**

NPM nos permite gestionar nuestros módulos. Para instalar o borrar módulos, se utilizan los parámetros típicos de estos programas.

```
$ npm install modulo3os
```

*Código 31: Instala el módulo modulo3os de nodejs*

```
$ npm uninstall modulo3os
```

*Código 32: Desinstala el módulo de nodejs modulo3os*

Usando estos dos comandos, se puede instalar o desinstalar un módulo del repositorio de npm. Ahora bien, estos módulos serán instalados y colocados en la carpeta “node\_modules” del proyecto en el que estemos trabajando<sup>39</sup>. Es decir, sólo serán accesibles desde la raíz del proyecto, no podremos utilizarlos en otro proyecto diferente.

No obstante el comando npm nos permite gestionar estas dependencias también de forma global añadiendo el parámetro “-g”. Y no sólo eso, también nos da la opción de separar los módulos o herramientas que usamos para el desarrollo de las dependencias del programa en sí. Así podemos incluir las herramientas que utilizaremos para el desarrollo en un directorio diferente que será excluido para la compilación, gestión de versiones o cualquier otra acción de construcción de código para producción.

```
$ npm install -g tsc
```

*Código 33: Instala el compilador de typescript de manera global*

---

38 Utilizando “npm run test”.

39 Donde estemos situados con la terminal (pwd)



```
$ npm install sass --save-dev
```

*Código 34: Instala el módulo sass como dependencia de desarrollo*

Npm también proporciona fórmulas para instalar versiones específicas de un módulo, o de un autor en particular, versiones anteriores o posteriores a una en particular o incluso permite instalar paquetes desde otros repositorios como github, gitlab, etc.,

```
$ npm install -g @autor2/modullos3os
```

*Código 35: Instala el módulo del autor especificado*

```
$ npm install modulo3os@1.5.0
```

*Código 36: Instala la versión 1.5.0 del paquete*

Para más información sobre la instalación de paquetes se puede consultar la página de la documentación referida a "[npm install](#)".

El resultado final de todos estos comandos no es más que un archivo package.json que incluye "meta datos" de los archivos situados en el directorio node\_modules.

En este documento se tratará el directorio node\_modules como si de una caja negra<sup>40</sup> se tratase, pues no es objetivo del mismo responder al funcionamiento interno del programa npm. En cambio si que se analizará un poco más en profundidad el fichero "package.json" que gestiona npm.

```
{
  "name": "mi-app",
  "version": "1.0.0",
  "description": "Es un proyecto típico.",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1",
  },
  "author": "Lucas C.B",
  "license": "GPL-3.0",
  "dependencies": {
    "modulo1": "~0.1.8",
    "modulo2": "^1.4.2"
  }
  "devDependencies": {
    "modulo3os": "3.4.0"
  }
}
```

*Código 37: Archivo package.json de ejemplo*

---

<sup>40</sup> Abstracción mediante la cual se separa la funcionalidad de la implementación.



Los módulos dentro de “dependencies” son módulos de los que depende la aplicación y los incluidos dentro de “devDependencies” los que se utilizan como herramientas para la creación de la aplicación.

#### **iv. Automatizando con NPM**

Otra característica importante del gestor npm es que nos permite ejecutar scripts a través de su interfaz. Esta funcionalidad es de suma utilidad cuando se crea un “workflow”<sup>41</sup> que separe acciones definidas. Por ejemplo, se pueden arrancar programadores de tareas<sup>42</sup> o grupos de comandos y herramientas diversas agrupados por diversos comandos.

Para programar esta funcionalidad sólo hay que añadir los comandos que se quieran ejecutar dentro del apartado el scripts del archivo “package.json” que genera npm.

```
{  
  "name": "prj-npm",  
  "version": "1.0.0",  
  "description": "Es un proyecto típico.",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\"$Error: no test specified\\\" && exit 1",  
    "surf": "grunt build; nodemon ./build/index.js;"  
  },  
  "author": "Lucas C.B",  
  "license": "GPL-3.0"  
}
```

*Código 38: Archivo package.json de prj-npm*

Y para disparar todos los comandos con npm se puede utilizar el siguiente comando.

```
$ npm run surf
```

*Código 39: Ejecuta el grupo de comandos asociados al script surf*

De esta manera se pueden automatizar procesos y ejecutarlos con sólo un comando.

La herramienta npm es muy completa, y posee muchas más funcionalidades de las especificadas en este documento, no sólo sirve para publicar módulos en el repositorios, instalar versiones específicas de programas, realizar búsquedas y demás. Es una herramienta muy potente y eficiente. Para más información se puede acudir a la web oficial para consultar la [documentación](#) o utiliza el comando “npm --help -l” en la consola.

#### **v. ¿Por qué usar NPM?**

Es la mejor manera de manejar las dependencias de un proyecto. Así, teniendo simplemente el fichero package.json de la aplicación se pueden instalar todas las dependencias con el comando “**npm install**”.

---

41 Flujo de trabajo.

42 Task Managers



## C. ANEXO DE TYPESCRIPT

### i. TypeScript y TSC

TypeScript es un superset de funcionalidades añadidas a JavaScript con los que se reescribió el framework de Angular. En este apartado se explican los fundamentos de este “lenguaje”, muy similar a JavaScript y que goza de muy buena reputación por añadir funcionalidades que aún no podemos encontrar hasta las últimas especificaciones de Ecmascript.

TypeScript y su transpilador<sup>43</sup> aportan las siguientes funcionalidades extra:

- Tipado estático → el compilador nos avisa de errores en conversión y utilización de tipos de datos. Esta característica hace que las aplicaciones sean más predecibles y fáciles de depurar.
- Características orientadas a objetos → podemos usar clases, interfaces, constructores, modificadores de acceso<sup>44</sup>, propiedades y otros.
- Al tener a nuestra disposición de un compilador, así como también una herramienta de autocompletado, se gana en seguridad y agilidad a la hora de crear nuestras aplicaciones.
- Todo código válido<sup>45</sup> en JavaScript es también válido en TypeScript.

### ii. Instalación del compilador

La instalación del compilador TSC es muy sencilla, sólo hay que utilizar el administrador de paquetes npm.

```
$ npm install -g typescript
```

*Código 40: Instalación de typescript*

### iii. Hola mundo con TypeScript

Tras la instalación podemos crear un archivo llamado “main.ts” e introducir algo de código javascript<sup>46</sup> en el mismo. Tras esto introducimos el siguiente comando en la terminal.

```
$ tsc main.ts && node main.js
```

*Código 41: Ejecución de archivo typescript*

Lo que acabamos de hacer es crear un archivo llamado “main.js” que ejecutamos con Node a partir del archivo “main.ts” que hemos compilado con tsc. Es decir, si todo sale como es debido el transpilador crea un archivo de igual nombre que el proporcionado como entrada y

43 En realidad TSC no es un compilador es un transpilador, pues convierte un código a JavaScript, no realiza una compilación a lenguaje máquina. No obstante en este documento se gastan indistintamente ambos términos. Transpilador ~ Compilador.

44 Modificadores como public o private tan típicos de otros lenguajes

45 De cualquier versión de ecmascript.

46 Recuerda que el compilador acepta código JavaScript.



tras acabar la conversión, es el programa NodeJS quien ejecuta ese nuevo archivo y nos muestra los resultados por consola.

Esto es sólo un ejemplo muy sencillo, de lo que en realidad hace la herramienta AngularCLI en segundo plano, pero esto ya lo veremos más adelante.

#### **iv. Variables y TypeScript**

En JavaScript hay dos formas de declarar una variable. La forma tradicional típica de Ecmascript 5<sup>47</sup> y la nueva forma incluida en Ecmascript 2015.

La primera, que todo el mundo conoce hace uso de la palabra reservada “var” y permite el acceso a nuestra variable desde varios ámbitos (scopes). Esto, puede crear problemas a la hora de escribir código, pues puede llevar a conflictos y errores extraños en la aplicación.

```
// forma clásica
Function cuenta(){
    for (var i = 0; i<5; i++){
        console.log(i);
    }
    console.log(i);
// el valor que se muestra en consola es 5
}
```

*Código 42: Formas de declarar un tipo en TypeScript*

La segunda forma, que usa la palabra reservada “let” sólo funciona dentro del ámbito donde se la declaró. Esto es una ventaja y aunque a veces nos obliga a hacer más trabajo, aporta la posibilidad de encontrar estos errores de forma sencilla en tiempo de compilación, es decir antes de desplegar la aplicación.

```
// forma clásica
Function cuenta(){
    for (let i = 0; i<5; i++){
        console.log(i);
    }
    console.log(i);
// el valor que se muestra en consola es 5
// pero tsc nos da un error de compilación
}
```

*Código 43: Formas de declarar un tipo en TypeScript*

Aunque TypeScript nos permite utilizar la segunda forma, y es la que se usará para el código, hay que conocer que el resultado final siempre será del tipo de la primera, pues TSC transpila a Ecmascript 5. No obstante, si que nos permite saber cuando se está realizando una mala práctica a la hora de declarar variables.

Otra funcionalidad importante a tener en cuenta con las variables, es la asignación de tipos. Es decir, el compilador nos va a producir errores en la conversión de tipos y también en la

---

<sup>47</sup> La versión de JavaScript soportada por la mayoría de navegadores.



asignación de los mismos. La forma correcta de asignar el tipo a una variable es la que se muestra en el siguiente cuadro.

```
// correcto obtendremos avisos
let a: number;
a = 5;
a = 'error no es un número';
// incorrecto
let b;
b = 3;
b = 'no hay error es un texto';
```

*Código 44: Formas de declarar un tipo en TypeScript*

Los tipos de datos que tenemos disponibles para usar en TypeScript se muestran en la siguiente tabla:

Tipo	Valores	Ejemplo
Boolean	True, false	let a: boolean;
Number	Números con o sin decimales, binario, hexadecimal, octal, ect.,	let a: number = 5; let bin: number = 0b1010; let oct: number = 0o744;
String	Cadena de caracteres, se puede utilizar los acentos graves para introducir variables.	let a: string; a = "Lucas C. B."; let b: string = `El estudiante se llama \${a}`;
Any	Cualquier tipo, no produce errores de compilación (var).	let a: any; a=2; a= "abc";
Array	Array de un tipo definido o any si es de diversos tipos.	let a: number[] = [1, 2, 4]; let b: any[] = [1,'dos',false];
Tuple	Dupla o tupla de valores.	let a: [string, number]; a = ["Página", 70]; a = ["hola", "cocacola"]; // da error console.log(a[0]);
Enum	Enumeración, como en otros lenguajes.	enum Color {Rojo, Verde, Azul}; let c: Color = Color.Azul; let nombreColor: string = Color[1];
Void	La ausencia de cualquier tipo. Se usa en returns y declaraciones de funciones.	Function warnUser(): void{ let nulo = null; }



Null / Undefined Cuando es nulo o está sin definir el tipo.

```
let u: undefined = undefined;
let n: null = null;
```

Object Cuando el tipo no es primitivo.

```
declare function crea(o: object | null):void;
crea({prop:0});
crea(null);
crea(false); // produce error
```

---

A veces a typescript le resulta difícil inferir el tipo y no nos proporciona sugerencias en el autocompletado. Para arreglar esto se pueden usar la assertividad de tipos<sup>48</sup>. Esta fórmula nos permite recalcarle al sistema de autocompletado de typescript que una variable es de un tipo en particular, como se aprecia en el siguiente cuadro.

```
// ejemplo de assertividad de tipos
let a;
a = "abc";
let acaba = a.endsWith("c");
// no proporciona sugerencias en cambio
// las dos fórmulas siguientes sí
let acabac = (<string>a).endsWith("c");
let acabab = (a as string).endsWith("b");
```

*Código 45: Assertividad de tipos TypeScript*

## v. Funciones flecha

Es una nueva forma de declarar funciones. Es el equivalente a las expresiones lambda de C# y otros lenguajes.

```
let mifunc = function (mens) {
    console.log(mens);
}
// es lo mismo que
let miflech = (mens) => console.log(mens);
```

*Código 46: Función flecha en TypeScript*

## vi. Interfaces de TypeScript

En TypeScript se pueden implementar el mismo concepto de interfaces que encontramos en Java y otros lenguajes. Dado que en JavaScript es muy normal pasar un objeto con múltiples propiedades, como parámetro a una función, la implementación de interfaces se hace tremadamente útil. Aporta reusabilidad y nos proporciona errores en tiempo de compilación si alguna de las propiedades del objeto enviado como argumento no es correcto.

---

48 En inglés type assertion.



```
let creaCoche = ( coche ) => {
    ...
}

creaCoche({
    marca: "Ford",
    puertas: 4
    // conductor: "Bill Murray"
}); // si añadimos conductor,
// no hay manera de encontrar el fallo
// --> en cambio hay dos formas de arreglarlo

// 1 - inline annotation con las característica del objeto

let creaCoche = ( coche: { marca:string, puertas: number}) =>
{
    ...
}
// ahora lo siguiente produce error
creaCoche({ conductor:"Bill", años:60});

// 2 - interfaz

interface Coche = { marca: string, puertas: number};
let creaCoche = (coche: Coche) =>{
    ...
}
creaCoche({conductor: "Bill"}); // da error
```

*Código 47: Interfaz en TypeScript*

## vii. Clases en TypeScript

En el anterior punto se explicó la forma de implementar interfaces en TypeScript. En cambio, para seguir las directivas de la programación orientada a objetos, necesitamos otros conceptos. Por un lado tenemos la unidad o esquema llamado clase. Este tipo de abstracción agrupa todos los elementos que se relacionan con un objeto en una misma estructura.

Al igual que en Java, C++, y otros lenguajes para la construcción de una clase se utiliza la palabra reservada Class.

```
class Coche{
    marca: string;
    puertas: number;
    motor: string;

    getMarca(){
        return this.marca;
    }
}
```

*Código 48: Ejemplo básico de clase en TypeScript*



También se incorporan otros conceptos como los constructores o la reserva de memoria mediante la palabra reservada “new”.

```
class Coche{
    marca: string;
    puertas: number;
    motor: string;

    constructor(marca:string, puertas:number,m?:string){
        this.marca = marca;
        this.puertas = puertas;
    }
    getMarca(){
        return this.marca;
    }
}
let coche = new Coche("Ford", 5);
Console.log(`Un coche marca ${coche.getMarca()}`);
```

*Código 49: Ejemplo básico de clase más completo en TypeScript*

Para obtener más funcionalidades con los constructores, se puede utilizar el signo de interrogación para declarar un parámetro opcional del constructor. Si se hace un parámetro opcional, todos los parámetros que se encuentren a la derecha deben también ser opcionales. En nuestro caso, si se añadieran parámetros al constructor, los que estén a la derecha de “m” deben también ser opcionales.

TypeScript también nos proporciona la característica típica de encapsulamiento que se observa en otros lenguajes de programación. Esta característica reduce las posibilidades de aparición de bugs y resultados inesperados en tiempo de ejecución.

Para manejar correctamente la encapsulación en TypeScript tenemos tres palabras reservadas que nos ayudarán a gestionarla en función de las necesidades.

- **public:** Por defecto, todos los miembros de una clase son públicos. Es decir, accesibles desde cualquier sitio del programa.
- **private:** El miembro/s de la clase que gisten esta palabra sólo pueden ser accedidos por otros miembros de la misma clase. Es decir sólo se puede usar en la clase. Con esta palabra podríamos aislar nuestros campos o propiedades de la clase, de accesos indeseados o imprevistos desde otra clase.
- **protected:**

TypeScript incorpora una funcionalidad para tener una sintaxis más sencilla a la hora de crear “setters y getters”. Usando las palabras reservadas “get” y “set” se consiguen los mismos resultados con una sintaxis más sencilla.



```
class Coche{  
    marca: string;  
    // ...  
  
    get marca(){  
        return this._marca;  
    }  
}  
// ahora se puede acceder así  
let coche = new Coche("Ford", 4);  
console.log(coche.marca);
```

*Código 50: Ejemplo de uso de palabra reservada get en TypeScript*

En este punto, se puede observar que TypeScript y JavaScript, poco a poco están realizando un viaje hacia Java. Cada vez proporciona más en características y funcionalidades del paradigma de Orientación a Objetos.

No obstante, hay que tener en cuenta que este tipo de “Setters y Getters” sólo son compilables a Ecmascript 5 y superior. Es decir, no serán utilizables por navegadores que no implementen ES5. Sin embargo, si queremos compilar este tipo de código hay que añadir parámetros adicionales a TSC.

```
$ tsc main.ts --target ES5 && node main.js
```

*Código 51: Ejecución de compilación tsc con “accessors”*

## viii. Modulos en TypeScript

En TypeScript podemos pensar en cada fichero como si de un módulo se tratara. El concepto es similar al que vimos en el apartado A.v de NodeJS. La diferencia radica en el uso de la palabra export al principio del fichero. No se usa exports ni require. Es un concepto un poco más refinado y similar a los imports de Java.

```
export class Coche{  
    marca: string;  
    puertas: number;  
    motor: string;  
  
    constructor(marca:string, puertas:number,m?:string){  
        this.marca = marca;  
        this.puertas = puertas;  
    }  
    getMarca(){  
        return this.marca;  
    }  
}
```

*Código 52: Ejemplo de módulo en TypeScript en fichero Coche.ts*



Se utiliza entre corchetes el tipo a importar que si son varios, se concatenan con coma. Lo que escribimos entre comillas, es la ruta relativa desde el archivo que importa al fichero que alberga la librería o módulo a importar. El nombre del fichero no lleva la terminación.

```
import { Coche } from './ruta/a/mi/modulo/Coche'; //Coche.ts No!
let coche = new Coche("Ford", 5);
Console.log(coche);
```

*Código 53: Importando un módulo al fichero main.ts en TypeScript*

Como ya veremos, en Angular no haremos referencia a los módulos usando una ruta relativa, sino que usaremos el nombre de la librería precedido por el símbolo arroba<sup>49</sup>.

En TypeScript se pueden exportar tipos, que pueden ser clases, funciones, objetos o simples variables y para usarlas hay que importarlas primero. Una vez se incluya la palabra import o export en la parte superior del archivo, ese se convierte en un módulo. Los módulos de TypeScript se diferencian de los de Angular en que estos últimos no organizan el código en diferentes ficheros, sino que son una forma de organización funcional para la aplicación.

---

<sup>49</sup> "@angular/core" que se encuentra en algún lugar dentro de la librería descargada en el directorio node\_modules que explicamos en npm.



## D. ANEXO DE ANGULAR 2

### i. Fundamentos de Angular

Angular es un framework de JavaScript que nos permite crear SPAs<sup>50</sup>. Este tipo de aplicaciones son aplicaciones web que cargan el interfaz completo en el navegador y que haciendo uso de JavaScript reaccionan a las interacciones del usuario. Las peticiones de datos que se realizan a la base de datos, se realizan en “background” o segundo plano. De esta manera, el usuario no se percata de estas peticiones y se consiguen aplicaciones de alto rendimiento para entornos web o incluso aplicaciones.

Pero, ¿por qué usar Angular y su ecosistema en vez de sólo JavaScript? Se usa Angular porque nos ayuda a estructurar nuestra aplicación, nos facilita mucho la reutilización de código y lo más importante, nos hace más fácil testearla.

### ii. Versiones del Framework

Es importante saber que hay dos versiones de Angular. Angular 1 o AngularJS que fue la primera versión del framework y Angular 2, 4, 6, 7 o 8 o simplemente Angular, que es el perfeccionamiento de la versión 2 del framework en el que se realizó una reestructuración y cambio de rumbo del framework. En este proyecto se habla de código para la versión 8 de Angular que se ejecuta de igual manera en cualquier versión anterior excepto la primera o AngularJS que es un framework totalmente diferente.

### iii. Arquitectura de Angular

El framework de Angular se estructura bajo dos conceptos básicos. El módulo y el componente. El componente o componentes, pues suelen haber muchos en una aplicación de Angular, es lo que sería una pieza del motor de un coche. Cada pieza, cumple una función y juntas forman un todo, el motor. Así, un tornillo cumple la función de sujetar la tapa del cárter al bloque motor, pero se puede utilizar también para cerrar una abrazadera de un tubo del turbo-compresor o para atornillar un faro al chasis del vehículo. Este es el punto fuerte del framework, la reusabilidad. Crear código reusable, agrupado como si de objetos o piezas de lego para construir algo más grande y complejo.

Toda aplicación de Angular tiene un componente raíz, del que cuelgan el resto de componentes. Estos, a su vez se pueden dividir en otros componentes que estén integrados dentro. Así se obtiene una estructura de árbol.

El otro concepto sobre el que se articula Angular, es el concepto de módulo. A diferencia de los módulos de NodeJS o los de TypeScript en angular se agrupan los componentes en módulos según su funcionalidad. Es decir, es una manera que tenemos de organizar la

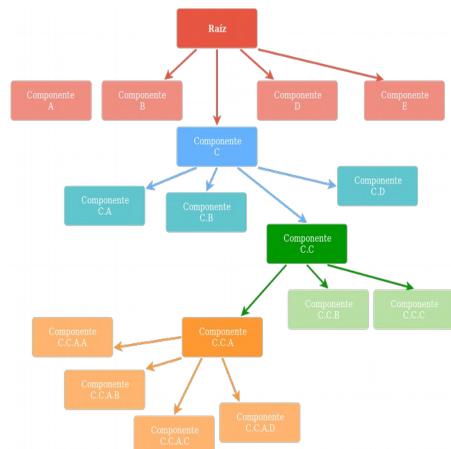
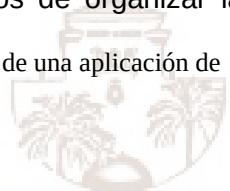


Figura 45: Árbol de componentes en Angular

50 En inglés Single Page Applications, aplicaciones web que simulan el comportamiento de una aplicación de escritorio.



aplicación, es similar a los paquetes de Java. A medida que nuestra aplicación crece en funcionalidades, podemos dividirla en módulos más manejables y fáciles de mantener.

Hay que destacar que toda aplicación de Angular tiene al menos un módulo (App-module) y un componente. Es decir, un módulo que integra un componente raíz.

#### **iv. Estructura de un proyecto Angular**

A la hora de crear una aplicación del framework de Google, se suele utilizar la herramienta “angular-cli” porque como se comprobará, se necesitan crear un montón de archivos e introducir bastantes líneas de código para hacer funcionar Angular. En cambio gracias a esta herramienta con un sólo comando podemos crear la estructura básica de una aplicación y tener listos todas las herramientas necesarias para empezar el proyecto, en apenas unos segundos.

```
$ ng new mi-angular-app
```

*Código 54: Creación de aplicación con angular-cli*

Se debe llamar a angular-cli usando el comando “ng” con el argumentos “new” que especifica que queremos que cree una aplicación nueva y como segundo argumento le decimos el nombre de la aplicación. Esto genera automáticamente una estructura con todos los archivos y componentes necesarios para implementar dicha web-app.

##### **a. Estructura de ficheros**

En cuanto acabe el proceso de la consola se observa que ahora hay una carpeta en el directorio base, con el nombre de la aplicación. Este nuevo contenedor de archivos se estructura de la siguiente manera.

```
4096 dic 25 08:55 .
4096 dic 25 08:54 ..
4096 dic 25 08:54 e2e
4096 dic 25 08:55 .git
28672 dic 25 08:55 node_modules
4096 dic 25 08:54 src
3633 dic 25 08:54 angular.json
429 dic 25 08:54 browserslist
246 dic 25 08:54 .editorconfig
631 dic 25 08:54 .gitignore
1023 dic 25 08:54 karma.conf.js
1297 dic 25 08:54 package.json
479715 dic 25 08:55 package-lock.json
1028 dic 25 08:54 README.md
270 dic 25 08:54 tsconfig.app.json
543 dic 25 08:54 tsconfig.json
270 dic 25 08:54 tsconfig.spec.json
1953 dic 25 08:54 tslint.json
```

*Figura 46: Lista de directorios y ficheros creados con angular-cli*

- e2e → directorio para introducir test “end to end”, que no son mas que test automatizados que simulan la interacción de usuarios reales.



- .git → este directorio que crea angular-cli de manera automática es donde se almacenan los datos del programa git que es el control de versiones que se suele utilizar en estos proyectos.
- node\_modules → ya se habló de este directorio con anterioridad. Aquí es donde están todas las librerías de terceros que ayudan a la creación del proyecto. Cuando despleguemos la aplicación, parte de estas librerías serán minificadas y colocadas en un “bundle” o paquete que se subirá al servidor.
- src → es donde se emplaza el código fuente de nuestra aplicación. Dentro de este directorio tenemos:
  - app → el directorio que almacena los módulos y componentes de angular. Al inicio hay un módulo y un componente:
    - app.**component**.css
    - app.**component**.html
    - app.**component**.spec
    - app.**component**.ts
    - app.**module**.ts
  - assets → donde se almacenan los activos estáticos de nuestra aplicación. Es decir, los archivos de imágenes, de iconos, de texto, etc.,
  - environments → aquí se encuentran las configuraciones para los diferentes entornos:
    - environment.prod.ts → para el entorno de producción
    - environment.ts → para el entorno de desarrollo
  - otros archivos que encontramos en el directorio src son:
    - favicon.ico → el ícono que se muestra en el navegador.
    - index.html → un simple fichero html que contiene la aplicación de Angular. En este fichero no hay referencias ni a scripts ni a estilos, sólo una etiqueta <app-root></app-root> que es donde se inserta la aplicación de manera dinámica.
    - main.ts → que es el punto inicial de la aplicación. Parecido al concepto de método Main de C, Java, etc., la función que se encarga de arrancar la aplicación es → bootstrapModule(AppModule);
    - polyfills.ts → es el archivo necesario para la integración de angular con los navegadores. Se podría decir que adapta el código generado por el framework para que sea utilizado en los navegadores actuales. Se pueden añadir más polyfills, pero ya se explicará más adelante.
    - styles.css → donde colocamos los css globales de la aplicación.
    - test.ts → para testear la aplicación.
  - .angular-cli.json → un archivo de configuración para angular-cli
  - .editorconfig → un archivo que especifica la configuración del editor de texto para que todos los desarrolladores que trabajen en el proyecto lo utilicen.



- .gitignore → especifica los archivos a ignorar por el sistema de control de versiones git.
- karma.conf.js → es un configurador del programa Karma que es un lanzador de tests<sup>51</sup>.
- package.json → ya se ha hablado de este archivo con anterioridad, es el encargado de almacenar los metadatos de las dependencias de la aplicación y del entorno de desarrollo. Este archivo se puede cambiar a voluntad, pero hay que saber lo que se hace para que todo funcione correctamente.
- protractor.conf.js → el archivo de configuración del programa protractor que es una herramienta para ejecutar test "end to end".
- tsconfig.json → configuración para el compilador de TypeScript. En función de este archivo el programa TSC transpilará o creará los archivos de una manera u otra.
- tslint.json → que configura la herramienta tslint que es un analizador de código TypeScript para mantenimiento, legibilidad y errores de funcionalidad.

Aunque se aprecian más archivos, estos son los fundamentales a la hora de trabajar con Angular. Poco a poco, si es necesario se introducirán más conceptos y aclaraciones sobre la estructura del proyecto.

### **b. Creación de componentes**

Para crear un componente es necesario completar tres pasos:

- Crear el componente y todos sus archivos asociados: El único archivo necesario es nombre.component.ts, pero pueden crearse también nombre.component.html, nombre.component.css y nombre.component.spec.ts.

Dentro del archivo nombre.component.ts se incluye un decorador que especifica las características que tiene dicho componente.

```
Import { Component } from '@angular/core';
// importamos las características de Component
// y sus decoradores

@Component({
    selector: 'cursos',
    template: '<h3>Hola Cursos</h3>'
})
export class CursosComponent{}
```

*Código 55: Ejemplo de cursos.component.ts*

El decorador @Component, si se adjunta a una clase, esta clase será convertida en componente. Ahora se analizan las partes que podemos definir dentro de este decorador.

- selector: referencia al elemento html mediante selectores css, de esta manera, el constructor de la aplicación, extiende el lenguaje de marcas html y nos permite estructurar la aplicación de manera muy sencilla.



- Ej - ‘cursos’ → en el html principal usaremos → <cursos>  
-‘.cursos’ → en el html principal usaremos → <div class="cursos">  
-‘#cursos’ → en el html principal usaremos → <div id="cursos">
- template: es la estructura html que forma el elemento en sí. Es decir, cada componente tiene su estructura interna y es aquí donde la definimos. También, si se prefiere, en lugar de añadir el código html directamente en el fichero “.ts”, se puede referenciar un archivo externo<sup>52</sup>. Esta es la forma en que se separa la lógica del “front-end” de la estructura visual.
  - styleUrls: Al igual que en la propiedad anterior, se definen los ficheros css que dan forma estética al componente. Es un array que especifica los archivos que contienen el estilo del componente, no es necesario incluir esta propiedad en el decorador.
  - Registrar el componente en un módulo: Se debe importar el archivo TypeScript del componente creado e incluirlo en el decorador de módulo de Angular.

```
Import { CursosComponent } from './cursos.component'
Import { BrowserModule } from '@angular/platform-browser';
Import { NgModule } from '@angular/core';

@NgModule({
  declarations:[
    AppComponent,
    CusosComponent,
  ],
  imports: [
    BrowserModule,
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule{}
```

*Código 56: Ejemplo de registro de componente en app.module.ts*

Así el constructor de Angular sabe que el componente “CursosComponent” forma parte del módulo AppModule.

- Añadir la etiqueta del componente en el HTML: Si va dentro de otro componente, se debe incluir la nueva etiqueta en el html de ese componente o en el de app.component.ts en el caso de sólo existir este fichero. De esta forma se anidan y estructuran los componentes del framework.

Bien, ya que se conoce la forma de crear un componente, da la sensación de que es una tarea tediosa y muy laboriosa. No importa si hay que crear uno, pero cuando la aplicación tiene cientos de componentes, nos encontramos con que se tiene que crear mucho código repetitivo. Para resolver este problema, los creadores de Angular integraron en la

---

52 Es por esto que hablamos que se pueden tener 1, 2 ,3 o 4 archivos que definen el componente. La convención para el título del fichero es → nombre.component.html



herramienta Angular-cli un comando para crear estos componentes y actualizar algunos de los archivos.

Así de manera sencilla, podemos crear plantillas de componente muy rápidamente.

```
$ ng generate component nombrecomponente
```

*Código 57: Creando componente con angular-cli*

Con el anterior comando, angular-cli se genera un directorio de nombre “nombrecomponente” y dentro de él crea los 4 archivos (html, css, ts y spec) del componente y actualiza el archivo app.module.ts registrando el componente. Si quisieramos que el componente se almacenara en un directorio llamado por ejemplo, miscomponentes podríamos utilizar el mismo comando con la ruta relativa del directorio. O también podríamos utilizar la forma abreviada<sup>53</sup> del comando.

```
$ ng g c ./miscomponentes/nombrecomponente
```

*Código 58: Creando componente con angular-cli abreviado*

### **c. Templates y renderizado dinámico**

Otra particularidad de Angular es la facilidad que nos otorga el framework para el renderizado dinámico de contenido. Para hacer uso de esta particularidad, debemos definir una variable dentro de la clase del componente. Luego, para indicar el lugar en el que queremos que se muestre este valor, se utilizan los dobles corchetes dentro del código o fichero html del componente. Angular cuando construya la aplicación, insertará el contenido en el DOM del navegador y si en algún momento del futuro, dichos datos cambiaran, el framework lo actualizará automáticamente. Todo esto, que se realiza en tiempo de ejecución, otorga un gran poder y facilidad de desarrollo al programador, es conocido como enlace de datos<sup>54</sup>.

```
Import { Component } from '@angular/core';
// importamos las características de Component
// y sus decoradores

@Component({
    selector: 'cursos',
    template: '<h3>Hola {{titulo}}</h3>'
})
export class CursosComponent{
    titulo: string = "cocacola";
}
```

*Código 59: Ejemplo de cursos.component.ts*

Pero esto no es sólo lo que podemos hacer. Dentro de los dobles corchetes, el enlace de datos, nos permite introducir el código JavaScript que se desee, es decir, se puede llamar a

53 Todos los comandos de Angular-Cli tienen una forma abreviada de escribirse

54 En inglés data binding



métodos, realizar operaciones matemáticas, concatenar caracteres, etc., Este tipo de sintaxis se suele llamar interpolación de cadenas<sup>55</sup>.

#### d. El DOM del navegador y las Directivas de Angular

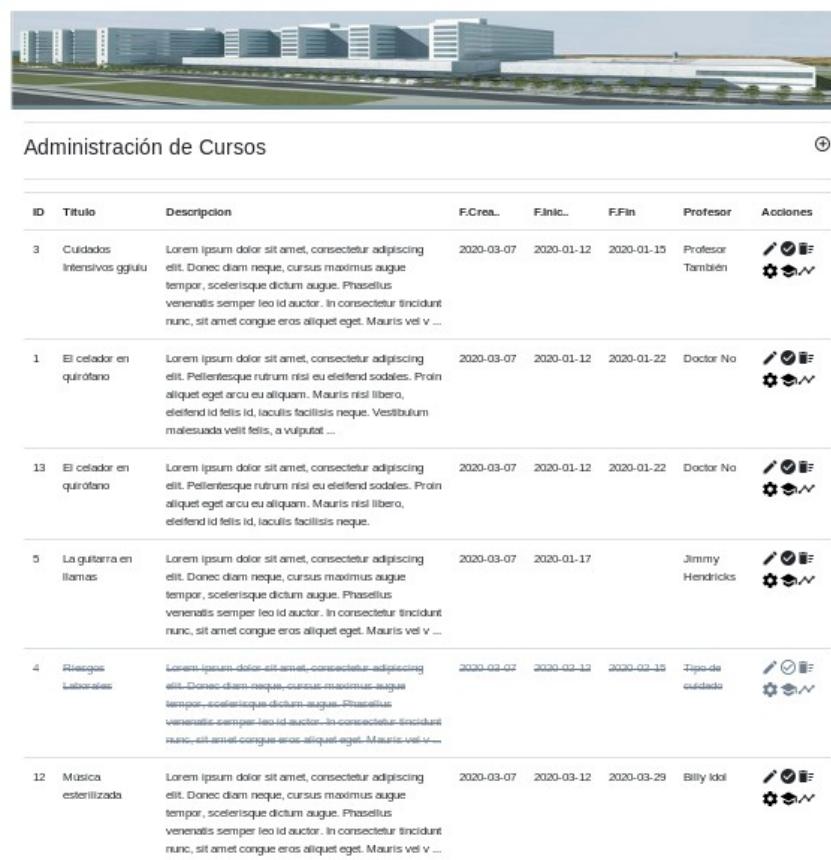
Además de lo anterior, el framework de Angular nos proporciona las “Directivas estructurales” como una forma muy potente de interactuar con el árbol DOM del navegador. Estas directivas, no son más que otra sintaxis específica que se utiliza sobre los “template” o plantillas html que nos permiten realizar tareas muy frecuentes. Las directivas estructurales, son muchas<sup>56</sup>, e incluso se pueden crear nuevas pero sobre todo sirven de gran ayuda a la hora de crear interfaces basadas en condiciones y bucles.

Las directivas siempre van precedidas de un asterisco, pues es la manera que tiene Angular de reconocerlas.

A continuación se explican las directivas más típicas de Angular:

- \*ngFor → Es un bucle. Se utiliza sobre todo para la repetición de elementos

En el ejemplo de código número 80 se utiliza la directiva ngFor para iterar sobre el array de cursos. Para mostrar cada uno de los elementos del array, Angular utiliza el enlace de datos y los dobles corchetes para mostrar el valor de la variable curso, creada por la directiva. Se utilizó este tipo de directiva para listar diferentes elementos en el cliente web.



ID	Título	Descripción	F.Crea..	F.Einic..	F.Fin	Profesor	Acciones
3	Cuidados Intensivos ggjuiu	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec diam neque, cursus maximus augue tempor, scelerisque dictum augue. Phasellus venenatis semper leo id auctor. In consectetur tincidunt nunc, sit amet congue eros aliquet eget. Mauris vel v ...	2020-03-07	2020-01-12	2020-01-15	Profesor También	
1	El celador en quirófano	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque nutrum nisi eu eleifend sodales. Proin aliquet eget arcu eu aliquam. Mauris nisi libero, eleifend id felis id, facilisis facilisis neque. Vestibulum malesuada velit felis, a vulputate ...	2020-03-07	2020-01-12	2020-01-22	Doctor No	
13	El celador en quirófano	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Pellentesque nutrum nisi eu eleifend sodales. Proin aliquet eget arcu eu aliquam. Mauris nisi libero, eleifend id felis id, facilisis facilisis neque.	2020-03-07	2020-01-12	2020-01-22	Doctor No	
5	La guitarra en llamas	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec diam neque, cursus maximus augue tempor, scelerisque dictum augue. Phasellus venenatis semper leo id auctor. In consectetur tincidunt nunc, sit amet congue eros aliquet eget. Mauris vel v ...	2020-03-07	2020-01-17		Jimmy Hendricks	
4	Riesgos Laborales	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec diam neque, cursus maximus augue tempor, scelerisque dictum augue. Phasellus venenatis semper leo id auctor. In consectetur tincidunt nunc, sit amet congue eros aliquet eget. Mauris vel v ...	2020-03-07	2020-03-12	2020-03-15	Tipo de actividad	
12	Música esterilizada	Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec diam neque, cursus maximus augue tempor, scelerisque dictum augue. Phasellus venenatis semper leo id auctor. In consectetur tincidunt nunc, sit amet congue eros aliquet eget. Mauris vel v ...	2020-03-07	2020-03-12	2020-03-29	Billy Idol	

Figura 47: Lista de cursos usando ngFor

55 En inglés string interpolation

56 Puedes consultar las básicas en la [documentación](#)



- **\*ngIf** → En el desarrollo web, hay veces que queremos mostrar o esconder una parte de la aplicación en función de una condición.

Hay que tener en cuenta que dentro del “\*ngIf=” puede haber cualquier tipo de expresión Javascript, incluso métodos del componente o de un servicio. La única condición para que esta directiva estructural funcione bien, es que dicha expresión o método devuelva un valor booleano.

Se utilizó esta directiva, como herramienta básica para mostrar y ocultar elementos, por ejemplo en la siguiente imagen.

Este es un supercampo  
Y tiene mogollón de botones, es requerido y tiene la opción otro  
 jfjasdf  lsdoa4  sdofosdf  oasdijfasdf  asdfasdf  oidjfasdfasdf

**Figura 48: Renderizado de ngIf antes de pulsar el botón**

Título del Campo  
Este es un supercampo  
Descripción o Subtítulo  
Y tiene mogollón de botones, es requerido y tiene la opción otro  
Un texto de apoyo para el usuario, no es obligatorio.

Tipo de Control

Botones de Respuesta Única

Etiqueta del botón de Respuesta Única  
Introduce el texto de la etiqueta  
\*Recuerde pulsar intro para añadir el texto del botón

Botones:

jfjasdf

lsdoa4

sdofosdf

**Figura 49: Renderizado de ngIf después de pulsar el botón**

- **\*ngSwitchCase** → Es un elemento que nos permite realizar lo mismo que “ngIf” pero no funciona sólo con valores “true” and “false”.

Esta directiva se usó sobre todo para desplegar diferentes menús en función de la selección. Por ejemplo con una lista desplegable de elementos. Como se puede apreciar en la serie de figuras que van de la imagen número a la imagen número .

#### **e. Módulo HttpClient**

El framework de Angular nos proporciona una serie de mecanismos para manejar las peticiones al servidor. Este concepto de cliente Http está fundamentado en los conceptos del conocido antiguamente como Ajax.

Se utiliza este módulo en prácticamente todas las interfaces de usuario desarrolladas para el proyecto, pues es el encargado de obtener el JSON con los datos del servidor.



A continuación se muestra un ejemplo de código implementado en la aplicación para utilizar la API “cursos” del servidor.

```
import { Injectable } from '@angular/core';
import { Curso } from '../models/curso';
import { HttpClient, HttpHeaders } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class CursoService {
  miJson: string;
  selectedCurso: Curso;
  cursos: Curso[];
  readonly URL_API = 'http://localhost:8080/EdOz/rest/cursos';
  curso: Curso;
  cursoId: number;

  // para la lectura sólo
  httpCab = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json')
  }
  // para el put y el post
  httpOptions: Object = {
    headers: new HttpHeaders({
      'Content-Type': 'application/json'}), responseType: 'text'
  }

  constructor(private http: HttpClient) {
    this.selectedCurso = new Curso();
  }

  getCursos(){
    return this.http.get(this.URL_API, this.httpCab);
  }
  putCurso(obj){
    this.miJson = JSON.stringify(obj);
    return this.http.put(this.URL_API + `/${obj.id}`,this.miJson,
      this.httpOptions);
  }
  postCurso(curso: Curso){
    return this.http.post(this.URL_API, curso, this.httpOptions);
  }
  desactivaCurso(id: string){
    return this.http.delete(this.URL_API + `/ ${id}` + "/desactivar",
      {responseType: 'text'});
  }
  activaCurso(id: string){
    return this.http.delete(this.URL_API + `/ ${id}` + "/activar",
      {responseType: 'text'});
  }
  borraCurso(id: string){
    return this.http.delete(this.URL_API + `/ ${id}` + "/borrar",
      {responseType: 'text'});
  }
}
```

*Código 60: Ejemplo de cursos.component.ts*



#### **f. Más sobre Angular**

El framework Angular tiene muchas más funcionalidades y características de las enseñadas aquí. Tiene la opción de crear servicios para compartir datos entre elementos. Tiene módulos que se encargan de la comunicación web<sup>57</sup>, tiene “Guards” y un “Router” para gestionar y filtrar las direcciones y mostrar los diversos componentes. En fin, Angular 8 es capaz de proporcionar todas las herramientas necesarias para crear el cliente web que nos hemos propuesto hacer.



## E. ANEXO DE SWAGGER

### i. Publicación del API usando Swagger y openAPI

A pesar de tener que implementar un API sencilla, también se puede usar un editor de APIs algo más complejo y completo como es Swagger.

Las principales razones para elegir el editor Swagger sobre otros es por ser el programa que creó la especificación [openAPI<sup>58</sup>](#). Otra buena razón para utilizar este sistema es la posibilidad de usar un editor específico para la misma. Este editor nos permite exportar la documentación del API a html con unos pocos clicks, así como resalta errores de sintaxis en el archivo YAML. Swagger sólo trabaja con YAML, aunque si puede exportar la API a JSON. De todas formas YAML, este es muy similar a JSON. Se podría decir que es un JSON optimizado para ser un poco más legible.

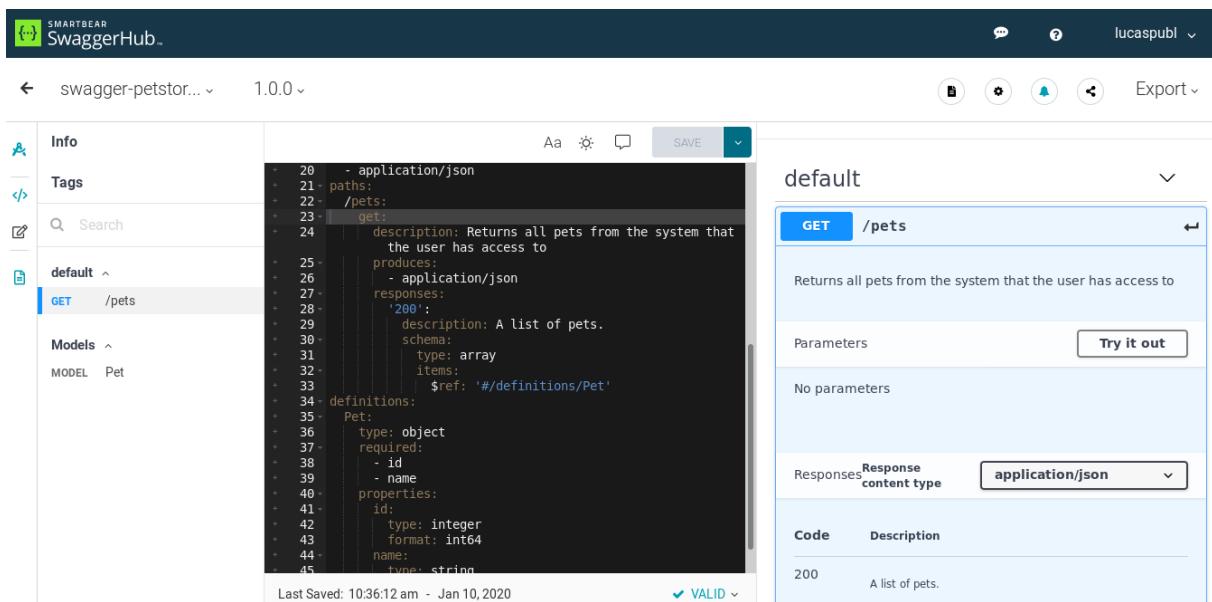


Figura 50: Vista del entorno de trabajo del editor Swagger

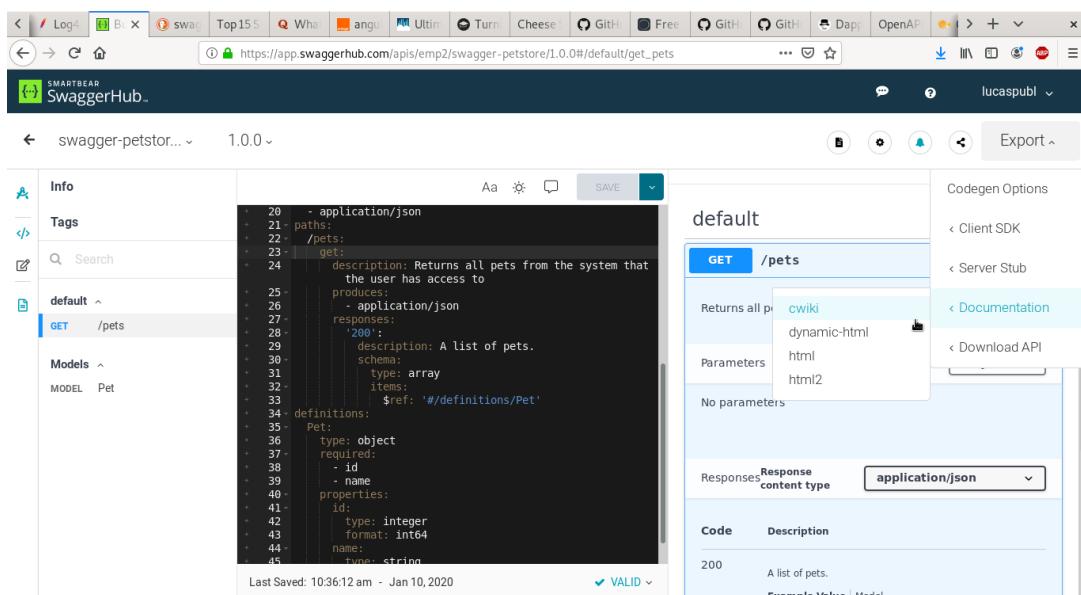
Para utilizar el editor no es necesario instalarlo, se puede usar de manera [online](#) y usa la especificación v2.0 ó v3.0 y para crear la documentación del API sólo hay que utilizar el menú de exportar API, como se muestra en la siguiente imagen.

La documentación generada en html tiene un formato distinto al de Spectacle, ofreciendo un html más básico y estático, pero con casi las mismas funcionalidades, pues lo que al final se pretende es documentar la API.

---

58 Anteriormente conocida como especificación Swagger.





**Figura 51: Exportando documentación del API con el editor Swagger**

**Figura 52: Ejemplo de documentación generada con el editor Swagger**

Para instalar el programa Swagger en modo local hay que introducir los siguientes comandos en la consola de linux.



```
$ git clone https://github.com/swagger-api/swagger-editor
```

Código 61: Instalando Swagger en el ordenador

Con esta acción, el programa git descarga el código fuente del programa. Tras lo cual, nos tenemos que situar en el directorio de nueva creación swagger-editor y ejecutar el comando “npm start”. Este otro comando descarga e instala las dependencias del programa, levanta un servidor en modo local y lanza el editor que se visualiza en el navegador en el puerto 3001.

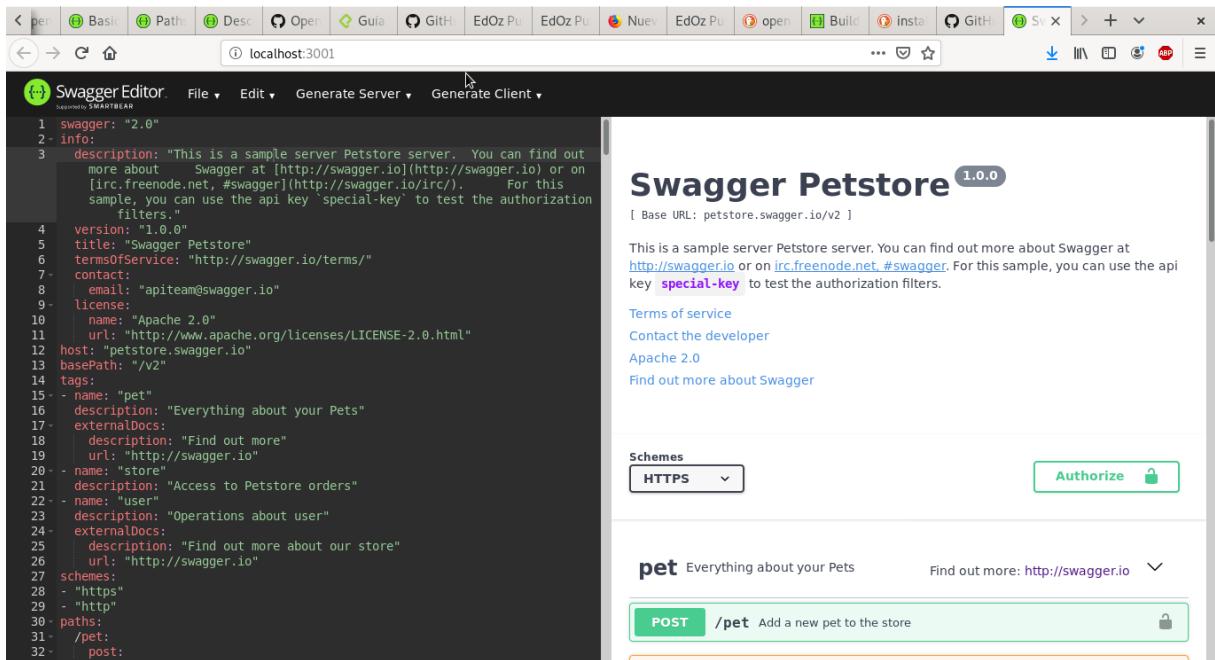


Figura 53: Ejecutando Swagger en modo local



## F. ANEXO DE JERSEY

### i. Anotaciones de Java con Jersey

Jersey implementa una serie de clases POJO que se pueden invocar a través de anotaciones Java.

Las anotaciones más importantes de la implementación Jersey son:

**@Path("url")** → Se usa para definir la ruta a la URL base o al recurso. La URL se configura mediante el nombre de la aplicación, el servlet y el patrón de URL establecido en el fichero web.xml.

Esta anotación se puede usar de forma anidada, es decir puede haber una en la clase y otra en un método/s que incorpore la clase. También soporta el control de rutas a través de expresiones regulares en la forma {"variable-name": "[regular-expression]"} como se puede ver en el ejemplo .

```
@Path("mi_primer_rest")
public class MiPrimerRest {
    @GET
    @Produces("text/plain")
    public String diAlgo() {
        return "Hola
caracola!";
    }
}
```

*Código 62: Ejemplo básico de anotación Path*

**@GET** → Indica que el método que lo lleva asociado, responde a una petición HTTP GET.

```
@GET
public String getUser() {
    System.out.println("Recibido GET");
    return "Hola con GET";
}
```

*Código 63: Ejemplo básico de anotación GET*

**@PUT** → Indica que el método que lo lleva asociado, responde a una petición HTTP PUT.

```
@PUT
public void updateUser(String usrdt) {
    System.out.println("Recibido PUT");
    System.out.println("Datos d usr: " + usrdt);
}
```

*Código 64: Ejemplo básico de anotación PUT*



**@POST** → Indica que el método que lo lleva asociado, responde a una petición HTTP POST.

```
@POST  
public void addUser(@FormParam("id") String id, @FormParam("name") String name){  
    System.out.println("Recibido POST");  
    System.out.println("Id: " + id);  
    System.out.println("Nombre: " + name);  
}
```

*Código 65: Ejemplo básico de anotación POST*

**@DELETE** → Indica que el método que lo lleva asociado, responde a una petición HTTP DELETE.

```
@DELETE  
@Path("{id}")  
public void delete(@PathParam("id")String id) {  
    System.out.println("DELETE: " + id);  
}
```

*Código 66: Ejemplo básico de anotación DELETE*

**@Produces(MediaType.TEXT\_PLAIN)** → Define que tipo MIME se devuelve con el método que lo lleva asociado. Varios ejemplos son (“text/plain”) (“application/json”) (“application/xml”). Si está colocado a nivel de clase, todos los métodos de esa clase pueden generar este tipo MIME por defecto. Si está a nivel de método, sobrescribe el tipo MIME por defecto que lleva asociado por la clase. Esta anotación también puede proporcionar múltiples tipos. Es importante usar esta anotación ya que si no se incluye, el servidor devolverá text/html como tipo MIME por defecto.

```
@Path("/rest")  
@Produces("text/plain")  
public class MiPrimerRest {  
    @GET  
    public String miMetodoA() {  
        ...  
    }  
    @GET  
    @Produces("text/html")  
    public String miMetodoB() {  
        ...  
    }  
}
```

*Código 67: Ejemplo básico de anotación Produces*

En el ejemplo, el primer método produce texto plano como respuesta y el segundo, devuelve HTML.



**@Consumes(type)** → Indica que tipo MIME consume el método o la clase. Sigue las mismas pautas que @Produces en cuestiones de “scope”<sup>59</sup> y valor por defecto, pero el error que devuelve es un 415 en lugar del 406 de la otra anotación. Al igual que @Produces, esta anotación también permite aceptar múltiples tipos MIME.

```
@Path("/rest")
@Consumes("multipart/related")
public class HelloWorldResource {
    @POST
    public String recibMultipart(MimeMultipart multipartData) {
        ...
    }
    @POST
    @Consumes("application/x-www-form-urlencoded")
    public String recibForm(FormURLEncodedProperties formData)
{
    ...
}
```

*Código 68: Ejemplo básico de anotación Consumes*

En el ejemplo anterior, se aceptan el tipo MIME Multipart y el tipo MIME de formulario por URL<sup>60</sup>.

Hay anotaciones que también se pueden colocar a nivel de parámetros como ya vimos en los ejemplos 65 y 66. Estas anotaciones son las siguientes:

**@PathParam** → Sirve para injectar los parámetros que envía el cliente. Estos parámetros que van en la URL se consiguen de la siguiente manera que podamos utilizarlos en nuestro código. Ejemplos de esto son @PathParam(value = "name") @PathParam("name"). Ver ejemplo de código 75.

**@QueryParam** → Sirve para injectar los parámetros colocados en la url, que envía el cliente que tienen el interrogante delante<sup>61</sup>.

```
@Path("/rest")
public class EjemploQueryParam {
    ...
    @GET
    @Path("/queryParam")
    public String MetodoConQuery(@QueryParam("name")String name) {
        System.out.println("Name: " + name);
        return name;
    }
    ...
}
```

*Código 69: Ejemplo básico de anotación QueryParam*

---

59 Si está escrito a nivel de clase o si lo está a nivel de método.

60 Estos son los datos que envía el típoco formulario html. Es lo que utilizaremos con Angular.

61 En inglés Query Params



En el ejemplo anterior el método tendrá en cuenta el parámetro que le pasa el cliente en el URI cuando accede al recurso de la siguiente manera; <http://localhost:8080/mi-app/rest/queryParam?name=Lucas>

**@FormParam** → Esta anotación se usa cuando el cuerpo (body) de la petición está codificado como application/x-www-form-urlencoded. De esta manera podemos injectar parámetros simples de manera individual.

```
@POST  
@Path("/addUser")  
public void addUser( @FormParam("name") String name, @FormParam("id") String id){  
    System.out.println("Add User:");  
    System.out.println("Id: " + id);  
    System.out.println("Name: " + name);  
}
```

*Código 70: Ejemplo básico de anotación FormParam*

**@HeaderParam** → Es una forma de mapear la cabecera HTTP (header) con nuestro código. Esta anotación puede resultar de utilidad a la hora de comprobar el tipo MIME que envía el cliente y así, realizar una respuesta acorde.

```
@GET  
@Path("/getUserAgent")  
public String getUserDevice(@HeaderParam("user-agent") String userAgent,  
                           @HeaderParam("Content-Type") MediaType contentType) {  
  
    return "User Agent: " + userAgent + ", Content-Type: " + contentType ;  
}
```

*Código 71: Ejemplo básico de anotación HeaderParam*

**@DefaultParam** → Sirve para asignar el valor por defecto de un parámetro, así se consigue generar parámetros opcionales. Este tipo de anotación funciona con @PathParam, @QueryParam, @MatrixParam, @FormParam, @HeaderParam y @CookieParam.

```
@GET  
@Path("/queryParam")  
public String getUser( @QueryParam("name")String name, @DefaultValue("15")  
                      @QueryParam("age") String age){  
    System.out.println("Nombre: " + name);  
    System.out.println("Edad: " + age);  
    return name;  
}
```

*Código 72: Ejemplo básico de anotación DefaultValue*



Hay más anotaciones como @CookieParam, @BeanParam, @MatrixParam que pueden ser de utilidad para usarse en los parámetros de los métodos. Estas anotaciones no se incluyen en este documento aunque tal vez se incluya algún ejemplo en el Anexo H.i, Ejemplos de código de anotaciones.

## **ii. Manejo de respuestas**

Aunque la aplicación del servidor tenga un log, hay veces que necesitamos responder a la aplicación cliente y decirle si ha ido todo correctamente así como también, cuál ha sido el error que se ha producido. Para ello, debemos implementar una parte de la especificación JAX-RS<sup>62</sup> que nos ayuda a manejar las respuestas del servidor de manera específica.

Lo más interesante de este tipo de acercamiento, es la posibilidad de crear respuestas estándar, típicas del servidor de una manera sencilla como se muestra en el siguiente cuadro.

```
@GET  
@Path("/usuarios")  
public Response getUsers(){  
    // Procesamos los datos  
    ...  
    // respondemos  
    return Response.status(Status.OK).entity(datos).build();  
}
```

*Código 73: Ejemplo básico de respuesta estándar*

Como se puede apreciar, se llama al método estático “Response” y se le asigna un estado y unos datos a devolver<sup>63</sup>. Este estado, se puede seleccionar de entre todas las respuestas típicas que se dan en los servidores, y se pueden consultar en la siguiente [documentación](#).

## **iii. Manejo de cabeceras**

Muchas veces en las aplicaciones REST se requiere tratar las cabeceras http de la petición. Esto es particularmente útil para la realización de la negociación de contenido así como para conocer el idioma u otras muchas acciones posibles.

```
@GET  
@Path("/primermediatype")  
public Response getUsers(@Context HttpHeaders cabeceras){  
  
    MediaType JsonMedia = new MediaType("application", "json");  
    // Extraemos y comparamos el Content-Type  
    String m = cabeceras.getAcceptableMediaTypes().get(0);  
  
    if ( m.isCompatible(JsonMedia) ){  
        // procesamos los datos json  
        ...  
        return Response.status(Status.OK).entity(stringDatosJSON).build();  
    }else{  
        return Response.status(Status.UNSUPPORTED_MEDIA_TYPE)  
            .entity(stringerror).build();  
    }  
}
```

*Código 74: Ejemplo básico de respuesta estándar*

---

62 Esa parte está incluida en el paquete **javax.ws.rs.core.Response**

63 Estos datos van en el cuerpo de la respuesta http



Para resolver estas situaciones el framework Jersey nos da la opción de usar la anotación “@Context” que nos devolverá la cabeceras HTTP. Así, se pueden obtener datos de las cabeceras y siguiendo la [documentación](#), se pueden realizar diferentes acciones en función de las mismas.



## G. ANEXO DE FIGURAS

### i. Imágenes varias



Figura 54: Padre Palomino



Figura 55: Edna Carapápel



## ii. Imágenes extra para explicaciones del modelo

Este anexo aglutina imágenes adicionales para la explicación de conceptos en el apartado de diseño del modelo.

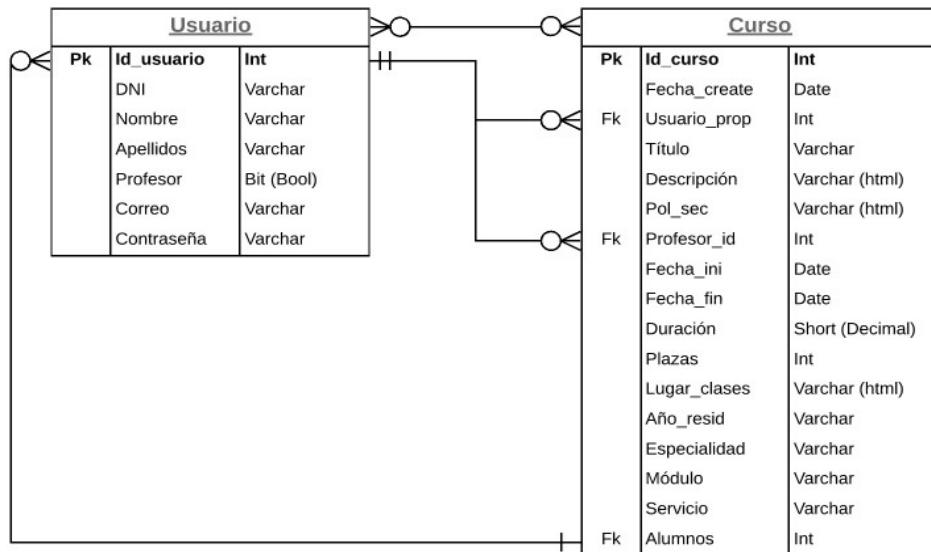


Figura 56: Modelo de datos usando un acercamiento sencillo

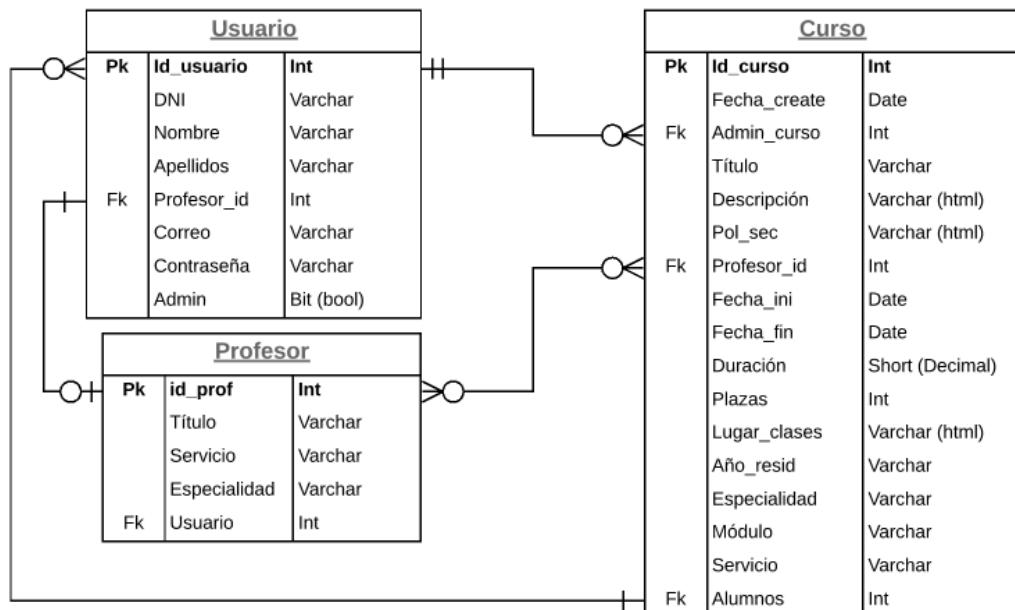
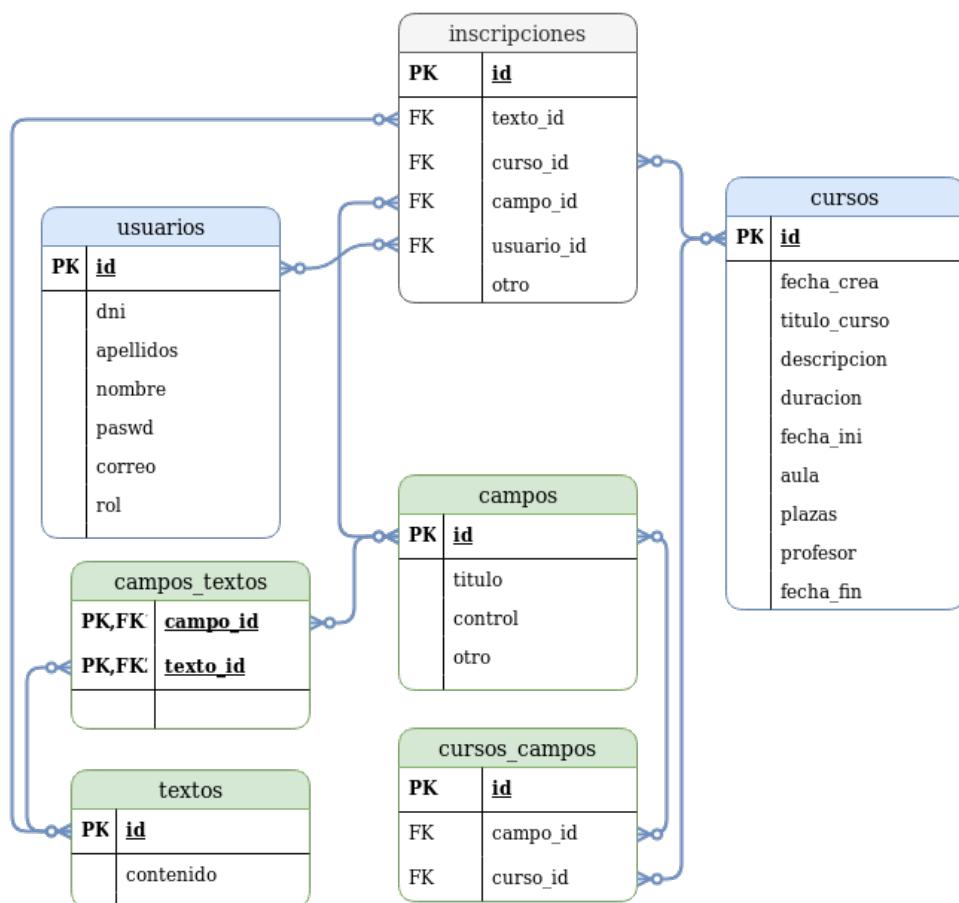


Figura 57: Posible acercamiento intermedio





**Figura 58: Estructura rechazada por unir las inscripciones y los campos, creando posibles problemas**



### iii. Imágenes extra del servidor tomcat y eclipse

A continuación se dejan unas imágenes que pueden ayudar en algunos de los apartados en que se explica cómo configurar el servidor tomcat desde eclipse.

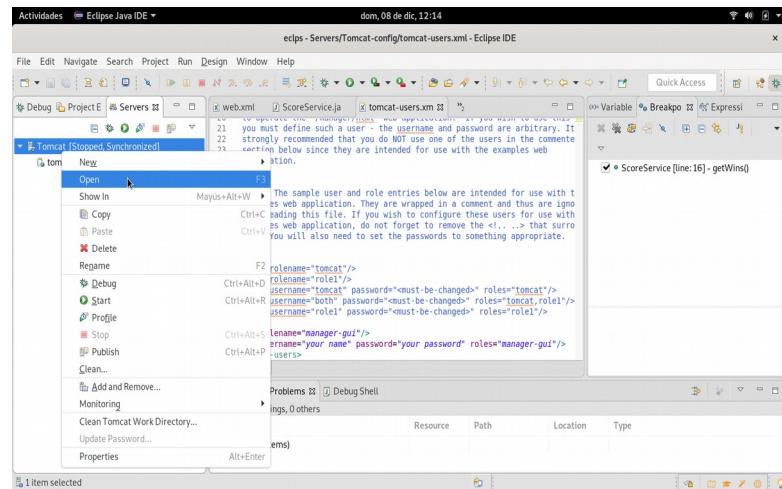


Figura 59: Abriendo la configuración del servidor tomcat

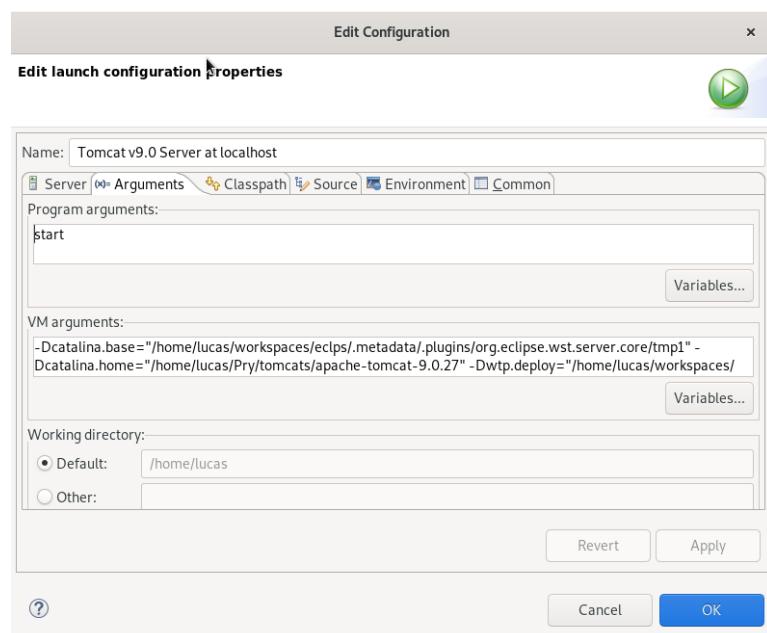
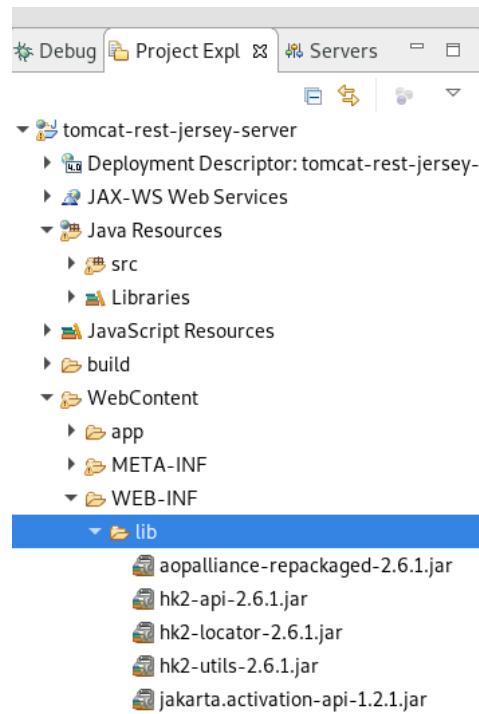
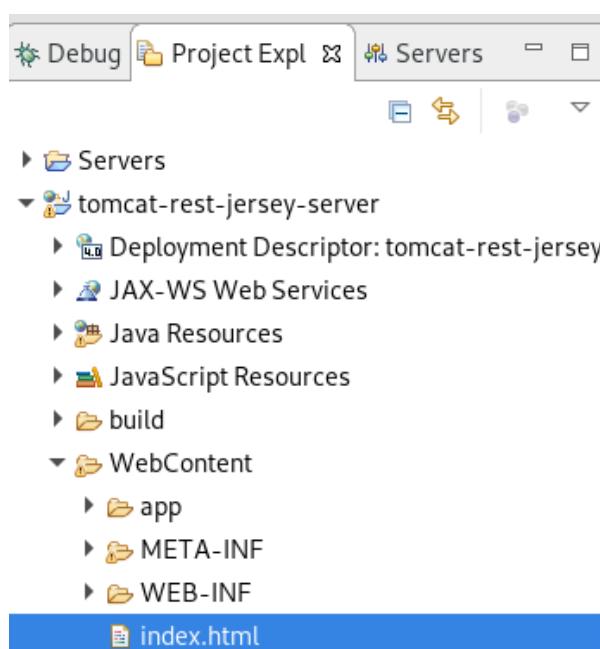


Figura 60: Configuración del servidor tomcat en eclipse





**Figura 61: Incluir archivos .jar a nivel de aplicación**



**Figura 62: Despliegue de html en proyecto web tomcat-eclipse**



## iv. Imágenes extra del cliente

Algunas imágenes de la interfaz de usuario.



**Figura 63: Botones para uso del ngSwitch**

### Crea Campos Personalizables

Título del Campo

Introduce el título

Título que aparecerá en grande, no es obligatorio.

Descripción o Subtítulo

Introduce la descripción

Un texto de apoyo para el usuario, no es obligatorio.

Tipo de Control

Caja de Texto

Respuesta Requerida

Etiqueta Superior

Introduce el texto de la etiqueta superior del texto

\*Este campo se tiene que llenar obligatoriamente

**Crear**

**Figura 64: ngSwitch pulsado el primer botón**

### Vincula, desvincula y borra Campos Personalizables

<b>Id</b>	<b>Título</b>	<b>Subtítulo</b>	<b>Tipo de Control</b>	<b>Acciones</b>
1	Categoría Profesional	Por favor introduzca a qué categoría profesional pertenece.	Caja de Texto	
2	Años de Residencia	Introduce el año de residencia en el que te encuentras actualmente	Lista de Selección	
3	Experiencia Profesional	Por favor introduce el número de años que llevas trabajados	Botones de Respuesta Múltiple	
4	Instrumento, sólo hay título		Lista de Selección	
5	Prueba de Campo oculto	Este control debe estar oculto en el curso 2 y mostrarse en el curso 3.	Caja de Texto	
7	Prueba Botones con campo Otro activado	Por favor, comprueba que funciona la respuesta otro	Botones de Respuesta Única	

**Figura 65: ngSwitch pulsado el segundo botón**





#### **Lista de Campos Personalizables del Curso**



##### **Prueba de Campo oculto**

Este control debe estar oculto en el curso 2 y mostrarse en el curso 3.

Médico

Introduce tu respuesta



dfjsdf  asfasf  hyfyfyfskdk asif asdf asifhuasf



##### **Este es un supercampo**

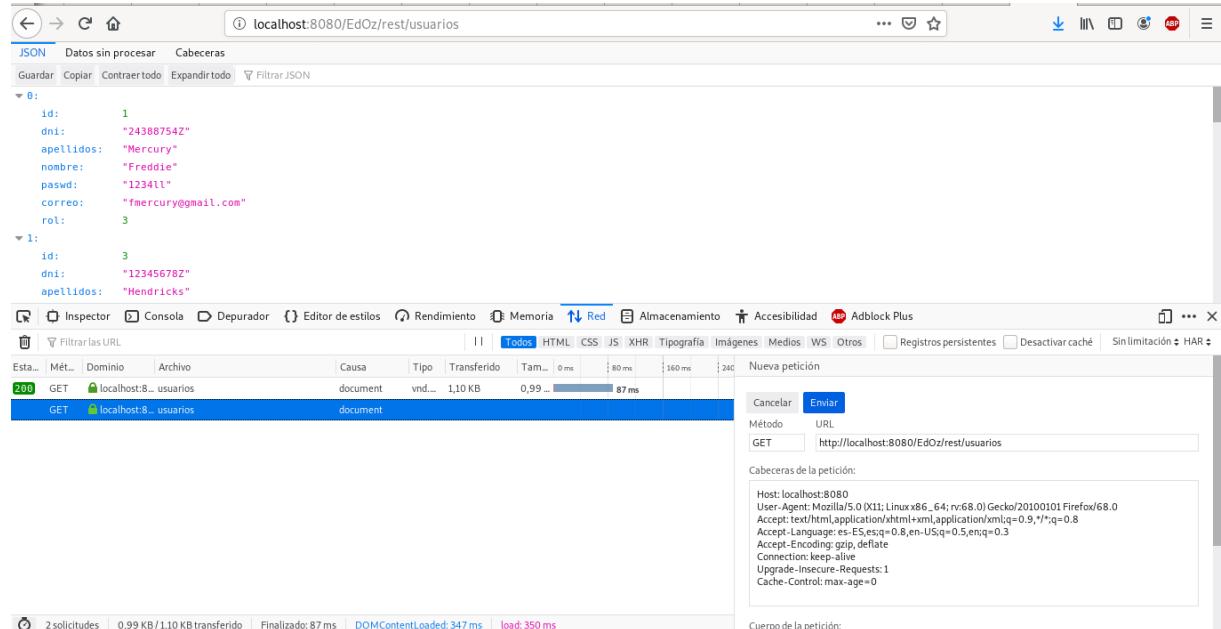
Y tiene mogollón de botones, es requerido y tiene la opción otro

**Figura 66: ngSwitch pulsado el último botón**

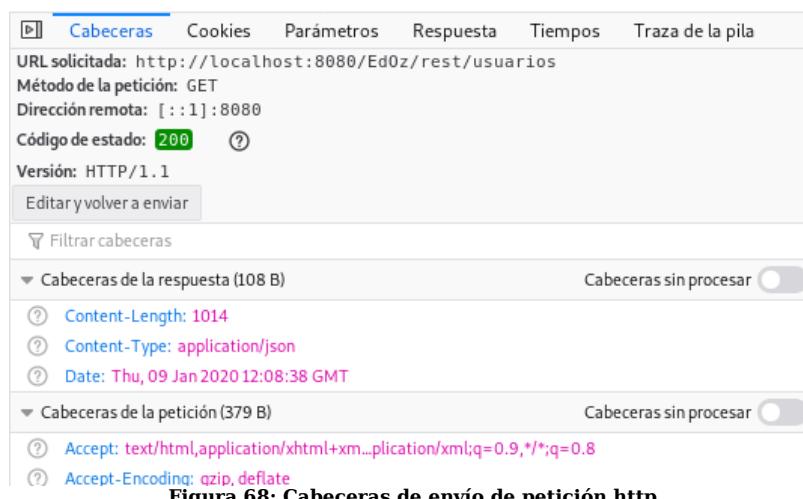


## v. Imágenes extra de envío de peticiones

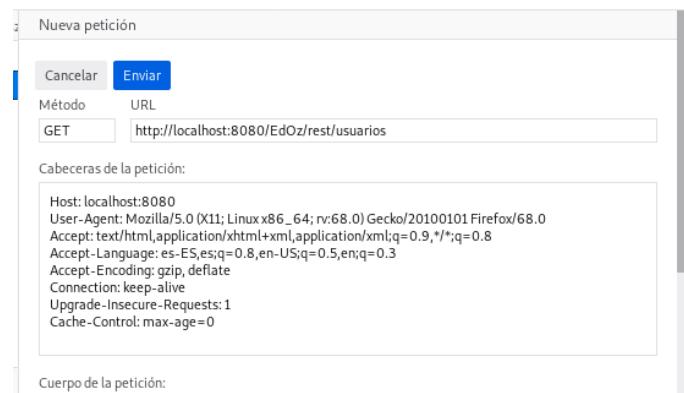
Aquí se incluyen imágenes extra de las herramientas de desarrollador del navegador mozilla firefox.



**Figura 67: Herramientas de desarrollo de firefox, reenviando petición**



**Figura 68: Cabeceras de envío de petición http**



**Figura 69: Edición de cabeceras y cuerpo de petición http con firefox**



## H. ANEXO DE EJEMPLOS DE CÓDIGO

### i. Ejemplos de código de anotaciones

Las anotaciones suelen ser muy sencillas, no obstante hay veces que se complican, por ellos e incorpora este anexo con ejemplos algo más complejos de cómo se realizan las acciones más típicas.

```
@Path("/rest/{name}")
public class MiPrimerRest {

    @GET
    @Produces("text/plain")
    public String diAlgo(@PathParam("name") String name){
        return "Hola, " + name;
    }
}
```

*Código 75: Ejemplo de Path a nivel de clase*

```
@Path("/rest")
public class MiPrimerRest {

    @GET
    @Produces("text/plain")
    @Path("{name}")
    public String diAlgo(@PathParam("name") String name){
        return "Hola, " + name;
    }
}
```

*Código 76: Ejemplo de Path a nivel de método*

Estos dos ejemplos realizan lo mismo, pero el segundo nos permite añadir más métodos a la ruta. Ambos códigos producirán la misma respuesta ante la llamada "<http://localhost:8080/mi-app/rest/Lucas>" → >> Hola Lucas.

```
@Path("/rest")
public class MiPrimerRest {
    @GET
    @Produces("text/plain")
    @Path("{name: ([a-zA-Z])*}")
    public String diAlgo(@PathParam("name") String name) {
        return "Hola, " + name;
    }
}
```

*Código 77: Ejemplo de anotación Path con expresiones regulares*

En este ejemplo el servicio rest sólo devuelve la respuesta correctamente si la petición sólo incluye letras. Es decir devolverá "status 404, not Found" si se envía lo siguiente "<http://localhost:8080/mi-app/rest/Lucas133>".



```
@POST  
public String addUser(MultivaluedMap<String, String> formData) {  
    System.out.println("Form Data: " + formData);  
    return "User added successfully.";  
}
```

*Código 78: Ejemplo de anotación Post*

```
@GET  
@Produces({"application/xml", "application/json"})  
public String miMetodoAB() {  
    ...  
}
```

*Código 79: Ejemplo de anotación Produces para múltiples tipos MIME*

```
Import { Component} from '@angular/core';  
// importamos las características de Component  
// y sus decoradores  
  
@Component({  
    selector: 'cursos',  
    template: `  
        <h3>{{titulo}}</h3>  
        <ul>  
            <li *ngFor="let curso of cursos">  
                {{ curso }}  
            </li>  
        </ul>  
`  
)  
export class CursosComponent{  
    titulo: string = "Lista de cursos";  
    cursos: string[] = ["curso1","curso2","curso3"]  
}
```

*Código 80: Ejemplo de uso de directiva ngFor*



```
Import { Component} from '@angular/core';

@Component({
    selector: 'cursos',
    template: `
        <h3>{{titulo}}</h3>
        <div *ngIf="bo0ol">
            El valor de bo0ol es true.
        </div>
    `
})
export class CursosComponent{
    titulo: string = "Ejemplo NgIf";
    bo0ol: boolean = false;
}
```

*Código 81: Ejemplo de uso de directiva ngIf*

```
Import { Component} from '@angular/core';

@Component({
    selector: 'cursos',
    template: `
        <h3>{{titulo}}</h3>
        <div *ngIf="bo0ol; else noTrue">
            El valor de bo0ol es TRUE.
        </div>
        <ng-template #noTrue>
            El valor de bo0ol es FALSE.
        </ng-template>
    `
})
export class CursosComponent{
    titulo: string = "Ejemplo NgIf";
    bo0ol: boolean = false;
}
```

*Código 82: Ejemplo de uso de directiva ngIf-else*



```
Import { Component } from '@angular/core';

@Component({
  selector: 'cursos',
  template: `
    <h3>{{titulo}}</h3>
    <ul>
      <button [class.active] (click)="cond = 'A'">A</button>
      <button [class.active] (click)="cond = 'B'">B</button>
      <button [class.active] (click)="cond = 'C'">C</button>
    </ul>
    <div [ngSwitch]="cond">
      <div *ngSwitchCase=" 'A' ">Dentro de A</div>
      <div *ngSwitchCase=" 'B' ">Dentro de C</div>
      <div *ngSwitchDefault>Otra cosa</div>
    </div>
  `
})
export class CursosComponent{
  titulo: string = "Ejemplo NgSwitchCase";
  cond: string = "A";
}
```

*Código 83: Ejemplo de uso de directiva ngSwitchCase*



## **ii. Ejemplos de configuración de recursos JNDI**

La mayor parte de la configuración que se realiza en el servidor Tomcat se realiza mediante XML. A continuación se colocan unos ejemplos de configuración a tener en cuenta.

```
<resource-ref>
    <description>Mail Server</description>
    <res-ref-name>mail/session</res-ref-name>
    <res-type>javax.mail.Session</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

*Código 84: Configuración MailSession en web.xml*

```
<Resource name="mail/session" auth="Container"
          type="javax.mail.Session"
          mail.smtp.host="smtp.gva.es"
          mail.smtp.port= "25"
          mail.transport.protocol="smtp"
          mail.smtp.auth="false"
        />
```

*Código 85: Configuración MailSession en context.xml*



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://
  xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-
  app_4_0.xsd" id="WebApp_ID" version="4.0">
  <display-name>EdOz</display-name>
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
  <resource-ref>
    <description>DB Connection</description>
    <res-ref-name>jdbc/postgres</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <resource-ref>
    <description>Mail Server</description>
    <res-ref-name>mail/session</res-ref-name>
    <res-type>javax.mail.Session</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
  <filter>
    <filter-name>CorsFilter</filter-name>
    <filter-class>org.apache.catalina.filters.CorsFilter</filter-class>
    <init-param>
      <param-name>cors.allowed.origins</param-name>
      <param-value>http://localhost:4200</param-value>
    </init-param>
    <init-param>
      <param-name>cors.allowed.methods</param-name>
      <param-value>GET,POST,HEAD,OPTIONS,PUT,DELETE</param-value>
    </init-param>
    <init-param>
      <param-name>cors.allowed.headers</param-name>
      <param-value>Content-Type,X-Requested-With,accept,Origin,Access-
Control-Request-Method,Access-Control-Request-Headers</param-value>
    </init-param>
    <init-param>
      <param-name>cors.exposed.headers</param-name>
      <param-value>Access-Control-Allow-Origin,Access-Control-Allow-
Credentials</param-value>
    </init-param>
    <init-param>
      <param-name>cors.support.credentials</param-name>
      <param-value>true</param-value>
    </init-param>
    <init-param>
      <param-name>cors.preflight.maxage</param-name>
      <param-value>10</param-value>
    </init-param>
  </filter>
```

Código 86: Ejemplo completo de archivo de configuración web.xml parte 1



```
<filter-mapping>
    <filter-name>CorsFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<servlet>
    <servlet-name>index</servlet-name>
    <jsp-file>/index.html</jsp-file>
</servlet>
<servlet-mapping>
    <servlet-name>index</servlet-name>
    <url-pattern>/cursos</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>index</servlet-name>
    <url-pattern>/admin</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>index</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>index</servlet-name>
    <url-pattern>/desinscribirse</url-pattern>
</servlet-mapping>
<error-page>
    <error-code>404</error-code>
    <location>/error_pages/notFound.html</location>
</error-page>
<error-page>
    <error-code>500</error-code>
    <location>/error_pages/error500.html</location>
</error-page>

<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error.jsp</location>
</error-page>

<env-entry>
    <env-entry-name>URL_DESINSCRIPCION</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>http://localhost:4200/desinscribirse</env-entry-value>
</env-entry>
</web-app>
```

*Código 87: Ejemplo completo de archivo de configuración web.xml parte 2*



### **iii. Configuración Log4J**

La mayoría de las configuraciones se explican en el apartado . No obstante aquí se incluye algún ejemplo interesante así como también configuraciones realizadas con otros lenguajes.

```
# nombre de la configuración
name = logConfig
# nombre de los appenders que creamos
appenders=xyz

# define el tipo de appender y su nombre
appender.xyz.type = Console
appender.xyz.name = myOutput

# define el formato del log
appender.xyz.layout.type = PatternLayout
appender.xyz.layout.pattern = %d [%p] [%c{1}:%L] - %m%n

rootLogger.level = debug
rootLogger.appenderRefs = abc
rootLogger.appenderRef.abc.ref = myOutput
```

*Código 88: Configuración Log4J con archivo log4j2.properties*



#### **iv. Códigos alternativos de Angular-cli**

Algunos comandos útiles o alternativas para la herramienta angular-cli.

```
$ ng g c nombrecomponente
```

*Código 89: Código alternativo para generar un componente con angular-cli*

