Propuesta para la gestión integral de contenidos en espacios de e-educación

UPMoodle

Índice

Introducción

Descripción del problema

La gestión de aspectos relacionados con la educación en plataformas digitales afronta diversos problemas y retos a los cuales la UPM y nuestra escuela no es ajena.
En la actualidad, nuestra principal plataforma digital en la universidad, Moodle, presenta algunas carencias tanto a nivel técnico como a nivel resolutivo.

A nivel resolutivo encontramos tres problemas principales:

- No es una herramienta integral y parte de la información útil se encuentra diseminada en otros espacios.
- No da respuesta a las necesidades de acceso a material de estudio del cuerpo estudiantil. Siendo el principal motivo de este problema el que la información recogida en estos portales provee de una fuente centralizada, el profesorado.
- No existe coordinación entre los distintos niveles jerárquicos de la universidad cuando se habla de los contenidos digitales. Los distintos niveles y entidades pueden encontrarse bajo en mismo espacio de e-educación, pero la coordinación entre los mismos tiende a ser nula y ello contribuye a una información dispersa.

A nivel técnico se entienden diversos problemas siendo la visión general de todos ellos que el desarrollo ha evolucionado considerablemente desde que Moodle fue concebido.
El enfoque de (L)CMS que implementado en la plataforma actual ha sido desplazado por la madurez de los frameworks de desarrollo y sus comunidades libres.
Siendo así que en este aspecto el mayor cambio/atraso que se percipe reside en su arquitectura interna y quedando patente su obsolescencia cuando a resistencia a cambios se refiere.

El objetivo del presente PFG es proposición de una alternativa técnica que de resolución a los problemas detectados, tanto técnicos como resolutivo.
En otras palabras, se persigue la proyección y desarrollo de una plataforma de e-educación, basados en nuevas tecnologías y los enfoques que estas apoyan. Dicha plataforma ha de contener la información de forma integral y coodinada tanto entre las distintas entidades y niveles de la institución como su alumnado.

Importancia del problema

1. Apuntes

Los resúmenes de las clases magistrales, popularmente conocidos como apuntes presentan algunas carencias cuando se analiza su estado en las plataformas de e-ducación como Moodle.
Estos apuntos se presentan en algunos casos desfasado, tardíos, inexistentes, incorrectos o insuficientes.

Esta conclusión no es sin embargo en la mayoría de los casos culpa de los docentes, en realidad este PFG comprende que el mayor fracaso reside en un planteamiento que frena la existencia y actualización del material de estudio y trabajo.
El motivo principal para este freno reside en que de forma oficial y habitual, sólo los docentes pueden ser fuente de material a la plataforma

En la experiencia personal se han dado (no pocas) ocasiones en que los alumnos hemos sido capaces de organizar y elaborar material de trabajo sensiblemente superior al ofrecido por el docente.
Esto ha sucedido porque los alumnos han cooperado en la elaboración de los mismos.
No hay mayor premisa que esa (la colaboración y la cooperación) tras la idea de una plataforma de material de trabajo abierta a modificaciones por parte de cualquier usuario, independientemente de su rol dentro del sistema educativo.

El supuesto objetivo de nuestra educación superior es la reflexión, el aprendizaje bajo el fuerte convencimiento de que estudiamos materias que nos apasionan, que despiertan nuestro interés y curiosidad y que nos hacen evitar los procesos de memorización como elementos centrales de la superación de las asignaturas.
Siendo esta una de las premisas fundacionales de la educación superior, así ha de considerarse el desarrollo de la misma y por tanto, la colaboración como vía para unir mentes curiosas y ávidas de aprendizaje para el desarrollo del conocimiento. Ese ha de ser el modelo teórico y las plataformas de e-educación verdaderamente abiertas a la particiación y la colaboración el camino práctico.

2. Eventos

En el caso concreto de la UPM/ETSISI al igual que por lo comprobado en otras universidades y escuelas, las noticias de la escuela nunca se encuentran 2 veces en el mismo sitio o espacio web y eso choca terriblemente con una realidad cotidiana de nuestra vida en el 2016.

A día de hoy nuestros ojos se dirigen continuamente a los paneles de notificaciones de nuestros dispositivos móviles.
Es un concepto teórico sencillo. En un solo lugar encontramos todo (o casi todo) lo que necesitamos saber. La comunicación funciona de forma activa y no ya reactiva como en el pasado.

Frente a esta realidad tangible de nuestro día a día, la experiencia concreta de la universidad: webs de departamento, escuela, profesores, el correo, las plataformas de e-

educación, etc.

Este modelo de acceso a la información sencillamente es ineficiente y es obsoleto ya en su planteamiento por todas las fallas que presenta: dispersión, inconexión, desactualización, falta de acuse de recibo, entre otros.

Por tanto, el enfoque, el camino que guía la solución a implementar ha de ir orientada a acoplarse con los patrones de diseño y usabilidad actuales, es decir, enfocar virar el modelo de comunicación de reactivo a activo.
Si bien en el presente PFG no se desarrolla la solución completa, sí se desarrolla el aspecto más importante para poder alcanzar ese escenario que es unificación de los avisos y noticias que sean necesarios comunicar.

3. Horarios

De forma paralela y casi indistinguible de los eventos y noticias surge la problemática de los horarios.
Sumado a los problemas que hemos ya visto en el caso de las noticias, los horarios tienen una serie de condiciones añadidas.
Algunas de las diferencias más importantes sería en lo relativo a su carácter intrínsecamente temporal (dicho en sentido de tiempo, no te caducidad), la periocidad, el hecho de que los horarios tienen a colisionar unos con otros y a la necesidad de representar estos eventos de una forma más gráfica para poder presnetarlos de forma entendible.

Ante esta problemática, partiendo nuevamente de lo expuesto en el caso de los eventos y noticias se llega a la conclusión de que es necesario unificar los horarios, hacerlos fácilmente mantenibles y representarlos de una forma sencilla.

Hay que tener en mente que cuando se habla de unificar y representar todo aquello que tiene una relación temporal concreta, no se habla únicamente de los horarios de las asignaturas. Hay que entender que también hablamos de horarios de tutorías, charlas que se den en las escuelas, etc.

Objetivos perseguidos

El objetivo del presente proyecto ya ha sido esbozado ligeramente, aunque en la presente sección se realizará un perfil mas detallado del mismo.

Anteriormente ya hemos desgranado los problemas principales en lo relativo a los problemas que presentan las plataformas digitales, tanto si son analizadas a alto nivel y observamos sus características lógicas y técnicas o si analizando con un grano mas fino realizamos un análisis sobre sus elementos concretos.

El objetivo del presente PFG será por tanto crear un servidor y un cliente web que den solución a los fallos ya detectados y expresandos anteriormente.

Cada uno de los dos elementos técnicos principales, el servidor y el cliente web se habrán de enfocar en distintos sub-objetivos, habiendo presentar la suma de ambos la resolución de los objetivos globales perseguidos.

Siendo así, el cliente web tendrá como objetivos principales:

- Hacer incapié en el desarrollo basado en paquetes modulares y reutilizables. Esto se conseguirá mediante gestores de paquetes tales como NPM o Bower.

  Los detalles sobre esta cuestión se encontrarán más adelante.
- La simplicación, mejora de las experiencias de usabilidad de los usuarios en entornos web. Para ello se hará uso de componentes visuales mínimos pero que enriquezcan significativamente la capacidad de los usuarios para entender e interactuar con el sitio web.
- Demostrar la madurez de los frameworks web en la actualidad. En este caso se hará uso de Angular como framework de desarrollo principal y se analizará como la madurez de esta herramienta permite añadir funcionalidad de forma rápida y carente de apalancamiento en el código.

El servidor por otro lado, compartirá algunos objetivos con el cliente web y al mismo tiempo se diferenciará de este:

- Se persigue demostrar la madurez de los frameworks de desarrollo. Sin embargo en este caso se busca demostrar como en el caso de los servidores / backends, es posible delegar en menos herramientas y librerías para únicamente tener que dedicar esfuerzos al desarrollo de la lógica de negocio, quedando el resto de competencias delegadas al framework utilizando, en nuestro caso Django.

- Subsidiariamente al objetivo anterior, también se persigue demostrar la agilización de los procesos de desarrollo sin por ello renunciar a calidad, modularidad o alguno de los distintos irrenunciables princios actuales del desarrollo de software.
- El uso de una API y los principios REST como una buena práctica a la hora de permitir comunicar clientes y servidores.

Finalmente, la puesta en común de estos dos elementos puramente técnicos nos permitirá reflexionar sobre el producto final obtenido. Esto quiere decir que una vez unimos las dos partes técnicas de este proyecto podremos concluir que habremos alcanzado con satisfacción el cumplimiento de los objetivos lógicos o de negocio tales como:

- Permitir que todas las partes de la comunidad de una universidad se involucren en el mantenimiento y mejora de la información asociada a la vida universitaria.
- La centralización y unificación de los datos utilizados y necesarios para la comunidad.
- La simplificación y mejora en el acceso y modificación de la información generada o en uso. Implicando esto igualmente que los derechos de modificación responden a una jerarquía de responsabilidad y asignación de funciones.
- En el apartado técnico de la suma de las partes se habrá de conseguir un sistema confiable, respetuoso con las buenas prácticas del software y sencillo de mantener.

Metodología para la resolución

En la actualidad de los entornos de desarrollo de software las metodologías ágiles imperan en la mayoría en la mayoría de los casos.

Las denominadas ágiles han venido a desplazar a metodologías “pesadas” previamente existentes tales como el desarrollo por prototipo o en cascada.

Sin embargo, el resultado de la aplicación de las metodologías ágiles es desigual en la actualidad, el factor humano y las características del proyecto influyen notablemente en el éxito a la hora de su implementación.

Por todo lo anterior y atendiendo a razones de limitación de personal (una sola persona), tiempos y experiencia, a la hora de elegir una metodología de resolución del presente proyecto se ha optado por una forma mixta entre el mundo ágil y el pesado.

Dicha metodología/flujo de trabajo está ampliamente basado en un trabajo previo realizado durante la práctica principal de la asignatura de Calidad del Software [1], impartida en este mismo grado.

Un análisis más detallado puede ser encontrado en el apartado “Análisis de la metodología utilizada”.

Resultados obtenidos

El resultado final obtenido en este proyecto está en consonancia con los objetivos perseguidos.
Si bien esta misma cuestión será analizada de forma detallada en los útlimos capítulos de la memoria, a modo introductorio podemos resumir los resultados en dos niveles.

A nivel lógico. Los objetivos de negocio perseguidos desde el comienzo han sido resueltos satisfactoriamente. Aún siendo escuetos pero concluyentes: existe alternativa. Partiendo de un supuesto análisis correcto de la problemática, haciendo uso de pricipios actuales y modernos de la ingeniería del software es perfectamente posible desarrollar satisfactoriamente plataformas orientadas al educación digital que den respuestas a problemas y necesidades de la sociedad actual.

A nivel técnico. Los resultados obtenidos en este caso son igualmente satisfactorios, aunque sí es necesario reconocer que han sufrido un mayor refinamiento y corrección durante la fase de desarrollo.
Aunque en los últimos capítulos del proyecto se analizará en profundidad los casos concretos, sí podemos exponer una serie de elementos previos comunes.

- La capacidad para la detección precoz de los errores durante el desarrollo que tanto fomentan las tecnologías ágiles no impiden en ningún caso que las correcciones no se den, mas bien al contrario.
- El nivel de exigencia técnico ha ido creciendo con el tiempo. Por un lado por adquirir experiencia laboral al mismo tiempo que el proyecto era desarrollado. Por otro lado porque al analizar algunas cuestiones técnicas en detalle han surgido distintos enfoques erróneos. Principalmente en cuestiones relacionadas con la arquirectura del código.

Es por todo lo anterior que los resultados obtenidos durante este proyecto son satisfactorios.

Estructura del presente documento

Esta memoria ha sido estructurada basándose en modelos conocidos, estandarizados y probados en su efectividad.

En los primeros apartados de este documento hemos realizado una exposición de las cuestiones lógicas principales. Hemos expuesto y analizado la problemática observada y que se intenta resolver. Posteriormente se ha trazado una introducción a los elementos técnicos de más alto nivel que habŕan de hacer de soporte para la solución final.
Igualmente, antes de dar paso a cuestiones técnicas en detalle, seguimos ampliando cuestiones como la forma de organización durante el desarrollo.

En los próximos apartados hablaremos de aspectos técnicos y para desarrollar adecuadamente todos los elementos implicados, haremos análisis iterativos de una serie de elementos comunes (cliente, servidor, tecnologías, arquitectura, et).
Para ello partiremos de un análisis de alto nivel, con una fuerte relación con los elementos lógicos/de negocio del proyecto.
A medida que se realicen más iteraciones sobre los elementos de relato inmutable, los aspectos lógicos se irán difuminando para dar paso a concrecciones técnicas, hasta llegar a los niveles más bajos y definidos.

Por último pero no menos importante, llegarán las conclusiones a todo lo que habrá sido expuesto.
Una recapitulación extensa y crítica entre objetivos y resultados darán cierre al documento.

Contexto

Introducción técnica

Como ya ha sido explicado anteriormente, este proyecto está compuesto por dos entidades técnicas principales: el cliente (frontal o front-end) y el servidor (o back-end).

La responsabilidad de cada uno es clara:

- El cliente es responsible de las interacciones con el usuario y es aquí donde reside el apartado visual.
  El cometido del frontal es presentar al usuario la información transmitida por el servidor. Y, en caso de producirse, comunicar al servidor los cambios en los datos que los usuarios desean.
  Por tanto, nuestro cliente no poseerá lógica de negocio alguna, simplemente actuará como mero intérprete a través de una interfaz gráfica entre el usuario y el servidor.
- El servidor está encargado del almacenamiento de datos y de su procesamiento.
  Este mismo componente tendrá la responsabilidad de exponer su API (Rest) al frontal a modo de contrato sobre las capacidades y características que el servidor ofrece.

Por tanto, el coste de este proyecto a la hora de resolver los problemas de lógica y negocio observados durante el análisis y diseño, ha sido superior al que hubiese supuesto una solución monolítica.
Sin embargo, el coste no es exactamente doble, no podemos olvidar que la definición de una buena API Rest, en el fondo una buena interfaz de comunicación entre ambas partes, supone diversas ventajas.
Entre las más importantes es comprender que le acomplamiento entre partes se realiza sin fricción, pues ambas partes llegan al elemento común de “mutuo acuerdo”.
Por otro lado, una buena interfaz API sienta las bases una buena arquitectura.

Al inicio de la memoria, fue expuesto que para resolver técnicamente el proyecto se haría incapié en conceptos tales como software libre, comunidades, modularidad, escalabilidad y buenas prácticas.
La elección del conjunto de lenguajes, frameworks y herramientas no ha sido condicionante para estos principios, sino al contrario. Fueron estos mismos principios quienes señalaron a las soluciones de software escogidas como las mejores opciones en cada caso.

El servidor, aspectos generales

Pila técnica

Lenguaje

Python es el lenguaje principal de nuestro servidor.
Este lenguaje fue concebido hace más de 20 años, está fuertemente basado en principios de software libre y con una comunidad en internet muy activa y participativa.

Entre sus características técnicas destaca:

- Es un lenguaje interpretado.
- Multiparadigma. A distintos niveles permite una programación imperativa, funcional u orientada a objetos.
- Dinámico, tanto en la interpretación de variables como en la capacidad de modificación en tiempo de ejecución.
- Paso por referencia de objeto. Una definición que no encaja en ninguna de las dos formas clásicas de paso de argumentos (valos o referencia).
- Indentación obligatoria.

Aún con todas estas características podría discutirse si Python era la mejor opción disponible.
Es indiscutible que otra serie de lenguajes podrían haber sido una base ideal para nuestro servidor, sin embargo, la elección del lenguaje estaba condicionada al framework a utilizar y Django era un claro ganador sobre el resto cuando se observaban los objetivos.

Otros lenguajes

Como menciones adicionales a los lenguajes utilizandos en el servidor estaría el uso de Bash, de forma exclusiva en tareas de automatización de la plataforma.

Framework

Como ya se ha dicho brevemente, Django ha resultado en nuestra elección final.
Este framework ofrece una serie de herramientas que garantizan desarrollos rápidos, desacoplamiento entre las partes de la arquitectura que ofrece, mapeador de objetos-relacional.

Es conveniente en este punto recordar que uno de los objetivos perseguidos era

demostrar como los frameworks actuales mostraban una madurez en su desarrollo y capacidades tales como ofrecen modularidad, escalabilidad y una curva de aprendizaje corta a la hora de trabajar con ellos.
Es este mismo hecho el que convierte a Django en una, sino la mejor, opción posible.

Django ha sido acortado en sus capacidades de vistas, autenticación de usuarios y gestión de sesiones y extendido en su capadidad de exposición de API. Estos dos cambios, junto a otros de menor consideración han podido ser realizados sin excesiva complejidad.

Esta versatilidad y/o maleabilidad ha sido una característica que otros lenguajes y frameworks no podían igualar.
Por poner un ejemplo, al ser comparado con Spring (sobre Java), en cierta parte Java podría haber resultado tan bueno como Python a nivel de lenguaje para nuestros propósitos, sin embargo, Spring obligaba empaquetar manualmente muchos pequeños proyectos distintos, añadiendo complejidad no deseada para nuestros objetivos y donde finalmente no tendríamos garantizada la misma flexibilidad y capacidades a igual esfuerzo.

Otros frameworks y librerías

El fichero “requirements.txt” presenta un abultado listado de dependencias, siendo la inmensa mayoría dependecias de terceros. Sin embargo cabría destacar entre las pocas que son dependencias de primer orden el uso de “django-rest-framework” y “django-cors-hearders”.

- Django-rest-framework es una librería dependiente de Django que nos permite exponer de una forma sencilla nuestra API Rest y enlazarla con nuestro proyecto. Adiccionalmente incluye herramientas para la mejora en el mapeo de objetos.
- Django-cors-hearders por otro lado es una pequeña librería que nos permite mejorar las posibilidades de nuestro servidor, permitiendo hacer llamadas de tipo CORS.
  En cuestiones de comunicaciones HTTP, los servidores tienen una serie de reglas para permitir la comunicación y envío de datos con servidores más allá del propio dominio (Cross-origin resource sharing).

  De forma y manera que para poder permitir que nuestra API sea descubrible y usable, no ya por servicios de terceros, sino por nuestro propio front-end, es preciso que ciertas cabeceras sean modificadas para permitir la mentada comunicación "más allá de dominio".
- Postman. La excepción de la lista pues Postman no es una librería sino una herramienta, concretamente una aplicación para Google Chrome. Esta herramienta es usada con el fin de mantener una API documentada y poder realizar pruebas (e incluso tests) de forma dinámica.
  Una forma rápida, visual y amena de probar la API Rest escrita puede ser mediante la instalación de la app en un navegador Chrome y posteriormente importando el

fichero que representa la API [2].

Gestores de dependencias

Ya ha sido comentado que uno de los objetivos en este proyecto es el rápido desarrollo y la delegación en frameworks y librerías tantas tareas y funcionalidad como sea posible. Para alcanzar estos objetivos, es necesario utilizar herramientas que permitan simplificar y hacer accesible el trabajo de instalación y portación de librerías.

Estas herramientas son comúnmente denominadas “gestores de dependencias”, “repositorios de paquetes” o algún nombre equivalente.

En Python, Pip es la herramienta utilizada para dicho cometido. Pip nos pemitirá instalar con facilidad nuestras librerías y obtener una instantánea de las dependencias existentes en nuestro proyecto para permitir el despliegue del proyecto en cualquier máquina.

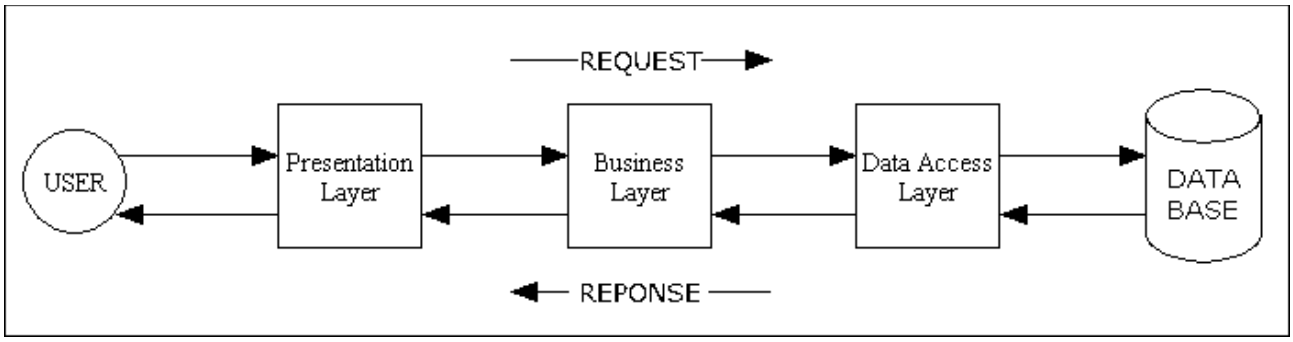
Gestores de tareas

En tanto hablemos de servidores y automatización/gestión de tareas, generalmente nos refereriremos a aquellos procesos que sirvan para realizar pruebas, despliegues o mantenimiento de forma automática o sistemática.

En este caso concreto no ha sido necesario realizar una integración de la plataforma con dichas herramientas, muy al contrario de lo que sucede en el caso del cliente web.

Un pequeño conjunto de scripts han sido desarrollados con el fin de tener bajo control las tareas más relativas al despliegue, aunque en uno de los siguientes apartados se desarrolla con más detalle lo relativo al despliegue.

Arquitectura



Una arquitectura de 3 capas

El dibujo anterior representa la arquitectura “clásica” que podríamos esperar ver en un servidor desarrollado desde 0, sin librerías adicionales y que además hubiese respetado unas normas y capas mínimas a la hora de construir dicho servidor.

En el caso de este proyecto, la ilustración es una buena aproximación a la arquitectura exacta que contiene el servidor, aunque algunas apreciaciones deben ser realizadas.

Capa 0. API

Entre el usuario y la capa de presentación encontramos una serie de elementos, que si bien carecen de lógica y por tanto, es discutible otorgarles la consideración de capa, es positivo resalzar su existencia.

Esta capa 0 es la que tiene la responsabilidad de capturar las llamadas realizadas a nuestra API y derivarlas al lugar adecuado de la capa de presentación. La resolución de esta capa en parte está sustentando en la librería django-rest-framework y en parte sobre desarrollo propio, aunque como se indica, el porcentaje de desarrollo propio es testimonial.

Capa 1. Capa de presentación.

Es el nombre clásico utilizado en esta capa, aunque quizás no es especialmente descriptiva en este proyecto.

El objetivo existencial de una capa de presentación es tratar con las peticiones de red que llegan a la máquina.
De hecho, cuando hablamos de una API Rest, estamos definiendo que esta capa tenga la responsabilidad de recibir peticiones HTTP y devolver igualmente llamadas HTTP. Esa es su responsabilidad y las clases y objetos pertenecientes a dicha capa no deben superar estos límites.

Al estar esta capa detrás de una capa previa, el nombre de presentación quizás no sea el más apropiado y resultaría más interesante hablar de capa de enrrutación. En el fondo, esa es el fin de esta capa, enrrutar llamadas.

Dicho más en detalle. La capa de enrrutación de este proyecto, recibe un objeto que representa la llamada HTTP, únicamente se analiza si el objeto/la llamada en sí se cumple con las restricciones marcadas y enrruta la llamada o no a la siguiente capa.
En caso positivo el objeto que representa la llamada HTTP no es transferido a la siguiente capa y únicamente serán los datos específicos asociados a la llamada los que serán pasados a la siguiente capa.

Igualmente, en caso negativo, en caso de que la petición recibida no tenga un formato adecuado, entra dentro de la responsabilidad de esta capa el devolver un error acorde. Aún es posible afinar más el grano en este sentido y exponer que las peticiones HTTP que no superan esta capa devolverán un código 400 (petición incorrecta) o 401 (no autenticado). Existirán muchos otros códigos y errores posibles, pero en esta capa únicamente estos dos deberían ser conocidos y utilizados.

Capa 2. Lógica de negocio.

Esta capa tiene la responsabilidad de tratar con los datos, procesarlos, reaccionar ante ellos y devolver una respuesta adecuada a los mismos.
Es la única de las 4 capas que veremos que sabe interpretar los datos como tal más allá de si existen (capas 1 y3), tienen el tipo adecuado (capa 3), etc.

En el caso del servidor desarrollado, la capa está compuesta por servicios, pues dicho nombre responde a una nomenclatura algo más actual sobre el propósito de existencia de los mismos. Esta capa ofrecerá un servicio para unos datos dados.

Es importante resaltar que esta capa es un híbrido entre los módulos ofrecidos por Django y desarrollo propio.
Esto sucede por un motivo principal, es Django en solitario el que compone la cuarta capa del servidor. Es el framework el encargado de gestionar la base de datos y la persistencia de los mismos.
Por tanto, si bien en esta capa existe muchísima lógica de negocio desarrollada, el uso de los objetos de tipo Model ofrecidos por Django, fuerzan a la simbiosis entre el código propio y el de framework.
Esta acoplación no es en absoluto una situación negativa, pues uno de los principios del framework es permitir la extensión y personalización de las capacidades ofrecidas.

Capa 3. Persistencia de datos.

Aunque brevemente, ya se ha comentado que Django gestiona esta capa en exclusiva y este es uno de las enormes ventajas del framework.

La escritura del código de comunicacción con BBDD es repetitivo, enlazante, vacío y plagado de los mismos aciertos y errores en su concepción e implementación (código boilerplatte).
Es por tanto, que delegar en una solución ya probada permite avanzar con tangible rapidez en el desarrollo de otros aspectos del proyecto.

Otra característica importante, es que es aquí donde reside la potencia del uso de un mapeador de objetos (ORM) que permite entregar objetos en memoria que serán traducidos a objetos de base de datos, la cual puede permancer completamente opaca en características y funcionamiento.

Con respecto al uso de mapeadores de objetos, por características complejas de las relaciones entre los modelos de objeto utilizados en este código, algunas clase adicionales han sido desarrolladas con el fin de mejorar el proceso de mapeo, independientemente de la dirección del mismo (serialización o deserialización).

Puesta en producción: despliegues

Cuando la fase de desarrollo está completada, es el momento de desplegar el código en la máquina que alojará y ejecutará la instancia del proyecto en cuestión.
Cuando un back-end es desarrollado, hay que distinguir entre el código propiamente y los datos de configuración que permiten ejecutarlo correctamente en una máquina.

El código y los datos de despliegue/configuración deben estar correctamente separados permitiendo así que un proyecto sea lo más agnóstico posible a los datos finales y concretos que alojará.

Esta mentada es muy diversa, pues varía entre claves de firma de una instancia, usuarios maestros que deben existir en cualquier sistema en ejecución hasta datos pertenecientes a la lógica que negocio que son necesarios para un correcto funcionamiento.

En el caso del servidor de este proyecto encontramos la necesidad de definir una serie de pasos, concretados en forma de pequeños "scripts" que nos permitan ejecutar una instancia completa.

Así, es necesario resaltar algunos de los pasos más importantes durante el despliegue de nuestro proyecto:

- Instalar las dependencias del proyecto. Como se ha comentado anteriormente, instalar los paquetes y librerías es un requisito indispensable cuando nuestro código posee dependencias con librerías y herramientas de terceros.
- Crear usuarios administradores para Django. Para hacer uso del panel de administración de Django, es necesario crear usuarios con permisos elevados, independientes del framework en sí y que puedan modificar en un momento determinado los datos de la base de datos del proyecto.
- Inicializar la base de datos de Django. En términos de framework esto se llama migración de modelos, pues el proceso sirve tanto para instancias nuevas como para actualizaciones, siendo obligatorio en casos de nuevas instancias.

  Este paso únicamente genera las entradas en la base de datos, aunque no proporciona datos.
- Provisionar con datos por defecto. Una vez nuestra base de datos existe y las tablas de los modelos están reflejados en la misma, algunos datos deberán ser añadidos sistemáticamente en las nuevas instancias con el fin de garantizar la existencia de unos mínimos de lógica de negocio en el sistema.
  En este aspecto es importante destacar que Django permite provisionar una instancia de diversas formas, pudiendo hacerse el proceso durante la migración de modelos o cargando ficheros de tipo JSON directamente en la base de datos (relacional).
  Es este proyecto se ha optado por la segunda opción por permitir alojar esta

información fuera del proyecto y por tratarse de representaciones planas de información, cuando por contra, en el caso de las migraciones, los datos son representados como objetos, haciendo mas complejo su modificación y manejo.

Con todos los elementos anteriormente enumerados, nuestros código puede ser desplegado y ejecutado en cualquier máquina UNIX que tenga Python 3.X instalado en ella y en la cual se tenga permiso de administración.

Como apunte final, el uso de Docker o Chef como solución en la que delegar los despliegues fue considerada, si bien, por el tamaño de del proyecto y por alejarse levemente de los objetivos del proyecto, su uso no fue finalmente concretado y se ha optado por desarrollar sencillos scripts en bash y python que lleven a cabo la tarea de provisionar una nueva instancia.

El cliente (web), aspectos generales

Pila técnica

Lenguaje

El lenguaje principal del cliente web es Javascript.

Javascript se caracteriza por ser interpretado y multiparadigma al igual que Python, si bien gana en velocidad y flexibilidad con respecto al lenguaje del servidor.
En construcción de sitios web es un referente absoluto y casi la única alternativa, esto se debe a que su enorme flexibilidad permite la construcción de framework extremadamente diferentes entre sí.

Otros lenguajes

Junto a Javascript (JS), el cliente web usa HTML, para la construcción/jerarquización de la estructura visual de la información representada y CSS para definir las propiedades visuales de cada uno de dichos componentes.

Framework

Angular es el framework principal, utilizado sobre javascript y desarrollado por Google. Como la mayoría de frameworks desarrollados en los últimos años, apuesta por los principios de single-page y por un patrón oficialmente definido MVC, aunque en términos relativos (y los propios desarrolladores del core así lo creen) puede considerarse un MVW (Model-View-Whatever). Esta definición nace de la abstracción, flexibilidad o mutabilidad que puede sufrir el concepto Controller según el punto de vista del programador y el modelo arquitectónico que utilice.

Igualmente, sería interesante mencionar que el concepto de vista no es inmutable en el desarrollo y que especialmente en entornos front-end se ve continuamente cuestionado. Si bien nadie discute la indivisible asociación de los términos vista & interfaz de usuario, existen distintas implementaciones o presentaciones finales del concepto vista. Es tangible que dependiendo del framework de javascript utilizado, la potencia y versatilidad del concepto vista varía y esto puede comprobarse objetivamente observando la discusión sobre patrones y en concreto patrón MVVM.

Como conclusión explicar que se elige Angular por su flexibilidad (entendido como reto para la construcción de un buen código sin que sea el framework quien fuerce a ello), por los retos a superar en sus puntos más débiles y por su comunidad y documentación disponible para consulta.

En el apartado dedicado a los módulos del proyecto (concepto anteriormente explicado en la introducción tecnica) se analizarán más en detalle aquellas librerías que se han incluído (o desarrollado) en la parte Angular del proyecto.

Otros frameworks

- SASS. Al igual que sucede con javascript, css hoy en día requiere de librerías y frameworks encima del lenguaje para alcanzar cierta potencia. En nuestro caso se apuesta por el uso de SASS como lenguaje de estilo.

  SASS (al igual que su principal rival: LESS) es un metalenguaje compilado, queriendo esto decir que el resultado de la compilación de su código da por resultado código CSS.

  CSS es un lenguage con una serie de limitaciones que en el mundo actual del desarrollo y maquetado web no ofrece todo lo necesario, sin embargo, la solución intermedia (y esperemos temporal) utilizada en ese aspecto es el uso extendido y masivo de metalenguajes que extiendan las capacidades y posibilidades de css.

  Todo esto se traduce en que los dos principales metalenguajes de CSS (SASS y Less) ofrecen características como la anidación, las variables, herencia, reutilización de código, etc, etc.

  Entre la posibilidad de utilizar Sass o Less se ha apostado por el primero. Es una opción más novedosa (más potente, un concepto más avanzado) pero con una comunidad y documentación igualmente asentada. Al ser la opción más novedosa incluye algunas funciones extremadamente interesantes que hacen que la balanza se decante positivamente sobre Sass (como por ejemplo su capacidad de reutilizar código). Uno de los últimos motivos de utilizar Sass es Compass. Compass es una herramienta de compilación de Sass pero que incluye funciones adicionales (como el uso de sprints por defecto) que hacen que la oferta de Compass haga de Sass una opcion mucho más interesante que su rival.

- Boostrap. Este framework tiene muchas vertientes. Inicialmente se hizo famoso por su conjunto de clases de CSS que permitían estructurar con seguridad y comodidad una web. Sin embargo, Bootstrap ha ido evolucionando con el tiempo y actualmente ofrece soluciones donde Javascript, CSS y HTML están involucradas. En cualquier caso, son las clases CSS de Boostrap el principal objetivo de su inclusión. Es preciso recordar que uno de los objetivos del proyecto es demostrar que la delegación en librerías de algunos aspectos del desarrollo aportan velocidad sin comprometer el resultado. Bootstrap cumple esta función a la perfección en el apartado CSS.

Gestores de dependencias

En la actualidad la mayoría de disciplinas o entornos poseen un grado de madurez tanto propio como de comunidad que permite que reusables conjuntos de código sean compartidos en forma de librerías o frameworks.

Si bien cada una de los entornos (web, móvil, servidor) tiene unas necesidades distintas, el principio que sustenta los gestores de dependencias de cada uno tienen bases comunes.

Dichos gestores de dependencias son herramientas extras que nos permiten acceder a aquellos paquetes (entendido como concepto abstracto de librería, framework o X) e intregrarlos en nuestros sistemas.

En muchas ocasiones, la filosofía detrás de los gestores de dependencias y comunidades de paquetes es el código libre. Entre sus muchos principios, en lo que compete a nuestro caso, una de las máximas de la teoría de estas comunidades es que muchos ojos observando y colaborando es extremadamente positivo.

En nuestro caso, para la gestión de dependencias de nuestro front-end utilizaremos Bower, NPM (node package manager) o gem (gemas de ruby) para la gestión de dichas librerías de terceros, sirviéndonos de las circunstancias para elegir en cada caso uno u otro, siendo el motivo principal de elección la disponibilidad y la capacidad de acceso a versiones específicas que garanticen la mayor compatibilidad con el resto de librerías.

Como excepcionalidad y violando brevemente la estructura de la memoria apuntar que un gestor de dependencias extremadamente famoso y muy utilizado en el desarrollo python es pip que en muchos aspectos recuerda a Bower.

Gestores de tareas

En la actualidad, nuestros proyectos web requieren de ciertas herramientas de gestión, control y despliegue adicionales que hasta hace unos pocos años no estaban maduras ni al alcance de cualquier desarrollo.

Para cumplir dichas necesidades los gestores de tareas, programas que simplifican los procesos de scripting asociados a las compilaciones y despliegues de código. En Android podríamos considerar Gradle o Ant y en términos de desarrollo web hablaríamos de Grunt o Gulp como los dos programas icono.

En nuestro caso, Grunt es la elección utilizada como gestor de tareas.

Como se explicaba anteriormente, Grunt nos ofrecerá incialmente introducir el concepto de automatización en el proceso de despliegue de nuestro proyecto. Sin embargo, este concepto se volverá más complejo según vaya escalando las necesidades y la complejidad del proyecto.

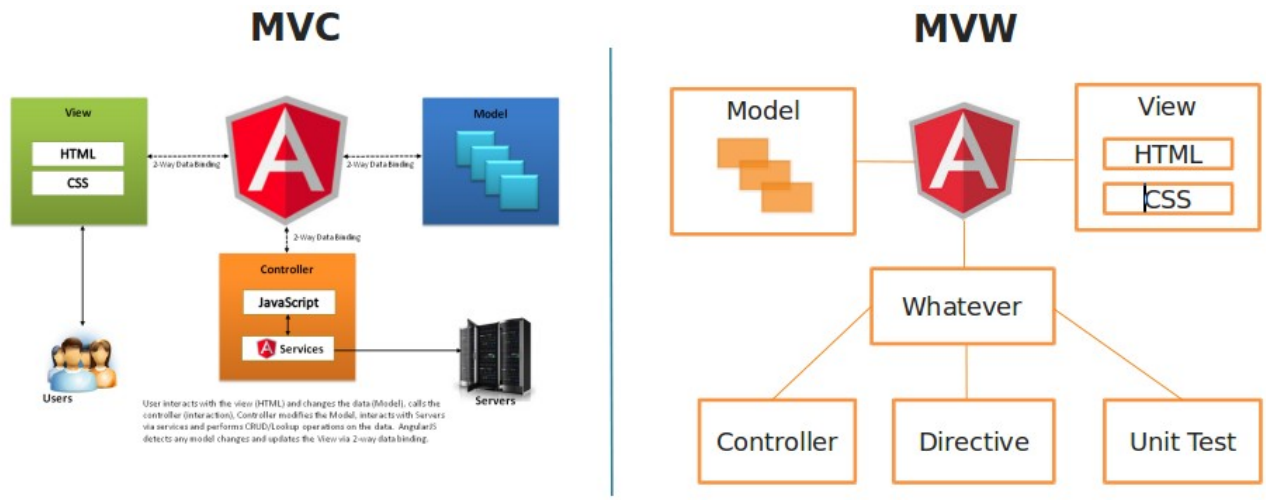
Inicialmente podríamos observar la necesidad de automatizar el proceso de compilación de los ficheros Sass como tarea mas primaria a solventar mediante automatización. Sin embargo, según crece la estructura del proyecto en su apartado Angular, se observa la necesidad de incluir tareas más complejas.

Entre las muchas opciones que ofrece grunt, las más comunes son la configuración para compilar y preparar nuestros ficheros scss. También puede ser utilizado para las tareas de ofuscación o concatenación de código. En el apartado predecesor dedicado a Angular ya se explicaba la estructura a nivel de ficheros seguida, pues bien, dicha estructura obedece a razones de legibilidad, estructura visual y también a razones de simplificación de las tareas de despliegue y es aquí donde Grunt ofrece todo su potencial.

Gracias a la estructura de carpetas utilizado, en Grunt únicamente hemos de indicarle que por cada carpeta de nuestra librería de Angular, devuelva un fichero minimizado (min.js) con una estructura concatenada y ofuscada. Así, en un solo fichero tenemos todos nuestros servicios, directivas, controladores y librerías de terceros integrados para funcionar.

De igual forma, en nuestro index.html que soporta toda la ingeniería single-page, tiene un número muy reducido de dependencias a incluir, facilitando enormemente las labores de desarrollo y mantenimiento.

Arquitectura



Interpretaciones de arquitecturas Angular (MVC vs MVW)

Una caracterísitca de Angular ya mencionada, es la creación o reinterpretación de los elementos jerárquicos que pueden utilizarse en un proyecto que use Angular. Es por ello que es necesario conocer ligeramente algunos de estos elementos:

- Directiva. Un elemento html que simboliza un modulo definido en el apartado js y que engloba un comportamiento completo asociado a una vista y un controlador principal.

- Servicios. Módulos de código reutilizable a lo largo de la aplicación pero que comparten memoria. En cierta manera son los singletons de angular y en múltiples ocasiones en la documentación oficial así como los propios desarrolladores del framework lo han defendido.

- Factorías. Las factorías y los servicios son muy similares en la mayoría de aspectos, habiendo diferencias significativas en la flexibilidad ofrecia (las factorías son algo más flexibles). En general puede entenderse que las factorías sirven para ofrecer comportamientos (funciones) y los servicios para ofrecer datos (modelos).

- Controladores. son una pieza clave de Angular y ofrecen comportamientos asociados a una vista. Si bien en términos generales el nombre de controlador es apropiado, debido a lo comentado anteriormente sobre MVW, a la modularidad pretendida en Angular y el diseño de módulos intercambiables, los controladores han de entenderse más bien como un pegamento, un nexo entre las vistas y los servicios y factorías.

Si bien la estructura a nivel de código no puede ser discutido (en tanto partimos de un framework definido por terceros), la forma en que el código sea organizado en las carpetas del proyecto sí que puede ser elegido a la hora de desarrollar.

Como se verá más adelante, cada uno de los conceptos de Angular señalados en el apartado anterior, es agrupado junto a sus semejantes en carpetas que los identifiquen. Es decir, los servicios se encuentran en la carpeta de servicios, las vistas en la carpeta de vistas, etc.

Esta organización responde a una voluntad de limpieza organizativa pero también buscando el objetivo se mejorar las tareas de compilación e inyección, cuestión que será visto con mayor detenimiento en el apartado dedicado a los gestores de tareas.

Despliegues

A diferencia del servidor, las características del cliente web hacen que el proceso de despliegue de una instancia sea autocontenido e indistinguible del proceso de ejecucción.

En el apartado dedicado al servidor, se vió como el proceso de despliegue del código en una máquina implicaba principalmente de la instalación de dependencias y provisión de datos en la base de datos.

Por el contrario, cuando hablamos del cliente, esto ya no se cumple en tanto que este es completamente agnóstico a los datos, pues su competencias es el enlace entre el apartado visual y los mismo, sin entrar a interpretarlos.
Es por esto, que los pasos para realizar un despliegue no se distinguen de los pasos para ejecutar una instancia. O dicho de otra forma, cuando se desea ejecutar una instancia el proceso es idempotente, es indiferente si la máquina está ejecutando el cliente web por primera vez o no, el resultado es el mismo.

Los pasos más importantes de este proceso de despliegue/arranque son los siguientes:

- Instalación y actualiación de dependencias.
- Empaquetar y minificar los ficheros JS necesarios.
- Compilar los SCSS en CSS
- Arrancar el servidor Node y exponer el cliente web en el puerto deseado.

Con estos sencillos pasos el cliente web se encontraría en ejecucción y listo para ser utilizado.
Como ya ha sido explicado anteriormente, todos estos pasos han sido llevado a cabo haciendo uso de Grunt.

Análisis de la metodología utilizada

En términos concretos, para abordar este proyecto se optó por realizar un análisis previo exhaustivo de todo el conjunto del problema, someterlo a diseños propositivos y rectificaciones previas. Este proceso fue llevado a cabo varias veces hasta tener la seguridad de tener una visión amplia asentada, realizable y en concordancia con el problema planteado.

Una vez concluída esta fase, con muchas similitudes a las que presentaría la fase de diseño o requisitaje de una metodología clásica o pesada, comenzaría la fase de desarrollo del proyecto. Esta segunda etapa es la que viene a tener más equivalencias con las ágiles. Teniendo en cuenta el tamaño del proyecto y la cantidad de personal involucrado en el mismo, las diferencias son notables y no se ha pretendido seguir un flujo de trabajo 100% Scrum o Kanban, por poner algunos ejemplos.

Sin embargo, en la etapa de desarrollo sí se ha hecho incapié en la idea de desarrollar pequeñas unidades completas, autocontenidas y testeables/testeadas de código en periodos cortos y cerrados de tiempo. O lo que en las ágiles se denominan “sprints”. Igualmente, la coordinación con los directores de proyecto ha sido fundamental en este aspecto, actuando ellos mismos en ciertos aspectos como propietarios (“owners”) del proyecto, en tanto que han sido ellos quienes al final del cada fase/sprint han comprobado los avances realizados sobre la fase anterior.

La elección de un enfoque ágil para la etapa de desarrollo ha permitido tener una comunicación fluida y continua. Esto ha desembocado en excelente retroalimentación en todo momento con los directores, de forma que las incorrecciones que han surgido del desarrollo han sido rápidamente detectadas y subsanadas.

Resultados

A continuación se exponen en detalle el resultado a nivél técnico, tanto en el lado del servidor como en el cliente web.
Este apartado es una continuación y profundización del las introducciones previas que se hicieron durante el capítulo “Contexto”.

De igual forma, en este capítulo se parte de un nivel medio de especificación para llegar a los niveles más bajos/detallados de análisis que serán expuestos en esta documentación y servirán a modo de punto y seguido para el último de los capítulos de la memoria, las conclusiones.

El servidor

Como ya se vió en el apartado anterior, las arquitecturas en los servidores siguen unos patrones muy definidos y comprobados.
A esto hay que sumar la guía de capas y componentes que Django marca a la hora de construir software haciendo uso de este framework.

Por ello y sumando el hecho de haber modificado algunas capas con respecto al diseño original de Django, en las 4 capas desarrolladas se ha seguido una guía de responsabilidades asignadas a cada una de estas capas. A saber:

- Capa 0 – API. En esta capa reside únicamente la responsabilidad de enlazar las llamadas HTTP entrantes con la siguiente capa.
- Capa 1 – Enrrutadores. Esta capa es responsable de tramitar peticiones y respuestas de red. No tiene mayor conocimiento que del servicio particular al que enrruta, los códigos de mensajes y la factoría de respuestas.
- Capa 2 – Servicios. Conocen los modelos, aunque no su funcionamiento interno explícito. En esta capa se lleva a cabo la gestión de la lógica de negocio propia y el manejo y mapeo de las excepciones.
- Capa 3 – Modelos. Esta capa, delegada mayoritariamente en el framework tiene por función validar los datos que se inyectan en los modelos, gestionar los accesos a base de datos y formatear los objetos en distintos tipos de formatos de salida.

Capa 0. API

A continuación se presentan y explican las rutas de la API expuesta por el servidor y su uso.

Esta información puede ser analizada estudiada de una forma más visual y amena haciendo uso de Postman e importando en el mismo el siguiente fichero.

Registro de usuario

```
POST /auth/signup/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{
    "email":"fool@eui.upm.es",
    "password":"password",
    "nick":"fooUser",
    "name":"foo foo"
}
```

Es la llamada que nos permitirá registrar a nuevos usuarios en la plataforma.

Se relaciona con:

- Front-end: app/source/controllers/signUpController.js
- Back-end: routers.auth.signup
  - Test: integration.auth.signup.SignUpTestCase

Autenticación de usuario

```
POST /auth/login/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{
    "email":"fool@eui.upm.es",
    "password":"password"
}
```

Esta ruta nos permitirá autenticar a un usuario en la plataforma y otorgarle distintos accesos en las llamadas sucesivas.

Se relaciona con:

- Front-end: app/source/controllers/loginController .js
- Back-end: routers.auth.login

- Test: integration.auth.login.LoginTestCase

Confirmación de correo

POST /auth/confirm_email/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{
"token":"token"
}

Es la llamada que nos permitirá registrar a nuevos usuarios en la plataforma.

Se relaciona con:

- Front-end: app/source/controllers/confirmEmailController.js
- Back-end: routers.auth.confirm_email
  - Test: integration.auth.confirmEmail.ConfirmEmailTestCase

Desautenticación de usuario

POST /auth/logout/ HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
Content-Type: application/json

Es la llamada que suprime los permisos de autenticación de un usuario.

Se relaciona con:

- Front-end: app/source/directives/navBar.js
- Back-end: routers.auth.logout
  - Test: integration.auth.logout.LogoutTestCase

Recuperación de contraseña

POST /auth/recover_password/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

```
{
   "email":"fool@eui.upm.es"
}
```

Es la llamada que permite a un usuario obtener una nueva contraseña.

Se relaciona con:

- Front-end: app/source/controllers/recoverPasswordController.js
- Back-end: routers.auth.recover_password
  - Test: integration.auth.recoverPassword.RecoverPasswordTestCase

Obtención de datos de usuario

```
GET /user/ HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Es la llamada que permite a un usuario autenticado obtener sus propios datos.

Se relaciona con:

- Front-end: app/source/factories/userFactory.js
- Back-end: routers.user.user_endpoint
  - Test: integration.user.user.UserTestCase

Modificación de datos de usuario

```
POST /user/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{
  "nick":"newNick"
}
```

Es la llamada que permite a un usuario autenticado modificar algunos de sus propios datos. Los campos modificables son el apodo, el nombre, la contraseña y el correo.

Se relaciona con:

- Front-end: app/source/controllers/profileController.js
- Back-end: routers.user.user_endpoint
  - Test: integration.user.user.UserTestCase

Eliminación de una cuenta de usuario

```
DELETE /user/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache
```

Es la llamada que permite a un usuario autenticado eliminar su propia cuenta.

Se relaciona con:

- Front-end: app/source/factories/userFactory.js
- Back-end: routers.user.user_endpoint
  - Test: integration.user.user.UserTestCase

Modificación de avatar

```
POST /user/avatar/ HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
Postman-Token: ef26221a-18f2-3094-f347-fa485510a5b6
Content-Type: multipart/form-data; boundary=----
WebKitFormBoundary7MA4YWxkTrZu0gW

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="avatar"; filename=""

Content-Type: image/png

----WebKitFormBoundary7MA4YWxkTrZu0gW
```

Esta llamada permite a un usuario autenticado modificar su foto de perfil (avatar).

Se relaciona con:

- Front-end: app/source/controllers/profileController.js
- Back-end: routers.user.user_avatar
  - Test: integration.user.user.UserTestCase

Modificación de las asignaturas adscritas

```
POST /user/subjects/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{

  "ids": [4,5,6]
```

}

Es la llamada que permite a un usuario autenticado modificar las asignaturas a las que está subscrito.

Se relaciona con:
- Front-end: app/source/controllers/editSubjectsController.js
- Back-end: routers.user.user_subjects
  - Test: integration.user.subject.EditSubjectTestCase

Obtención de datos de un usuario definido

```
GET /user/1/ HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Es la llamada que permite a un usuario autenticado obtener un conjunto de datos de otro usuario.

Se relaciona con:
- Front-end: app/source/factories/userFactory.js
- Back-end: routers.user.user_by_id
  - Test: integration.user.user.UserTestCase

Obtención de una noticia determinada

```
GET /note/2/ HTTP/1.1

Host: 127.0.0.1:8000

Cache-Control: no-cache
```

Esta llamada permite obtener una nota mediante su id.

Se relaciona con:
- Front-end: app/source/controllers/notesController.js
- Back-end: routers.note.note_by_id
  - Test: integration.note.note.NoteTestCase

Modificación de una noticia determinada

```
PUT /note/4/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
```

```
Cache-Control: no-cache

{
   "text": "New text"
}
```

Esta llamada permite modificar una nota ya existente mediante su id.

Se relaciona con:

- Front-end: app/source/controllers/notesController.js
- Back-end: routers.note.note_by_id
  - Test: integration.note.note.NoteTestCase

Eliminación de una noticia determinada

```
DELETE /note/4/ HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Esta llamada permite eliminar una nota ya existente mediante su id.

Se relaciona con:

- Front-end: app/source/controllers/notesController.js
- Back-end: routers.note.note_by_id
  - Test: integration.note.note.NoteTestCase

Publicación de una nueva noticia

```
POST /note/ HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{
   "text": "Body text",
   "topic": "New (bis)",
   "level_id": 3
}
```

Es esta llamada la que permite crear una nueva noticia.

Se relaciona con:

- Front-end: app/source/controllers/notesController.js

- Back-end: routers.note.note_endpoint
    - Test: integration.note.note.NoteTestCase

Obtener el árbol de niveles

```
GET /level/_tree HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Esta llamada devolverá el arbol de niveles de entidades/cuerpos estudiantiles de un nivel padre (universidad, escuela, curso, etc).

Se relaciona con:

- Front-end: app/source/subjectsController.js
- Back-end: routers.level.level_tree
    - Test: integration.level.level.LevelTestCase

Obtener todas las noticias de un nivel

```
GET /level/1/notes?recursive=true HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Esta llamada devuelve las noticias pertenecientes a un nivel organizativo. Si el parámetro de recursividad es verdadero, entonces devolverá también las noticias asociadas a niveles dependientes/inferiores.

Se relaciona con:

- Front-end: app/source/controllers/notesControllers.js
- Back-end: routers.level.level_notes_list
    - Test: integration.level.level.LevelNotesTestCase

Obtener un listado de ficheros de un nivel (de tipo asignatura)

```
GET /level/4/files HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Esta llamada devuelve la información de los ficheros asociados a una asignatura (nivel más inferior del árbol de niveles).

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.level.level_files_list
  - Test: integration.level.level.LevelFilesTestCase

Obtener los tipos de fichero existentes

```
GET /filetype/_all HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Con esta llamada obtendremos todos los tipos de ficheros existentes en nuestro sistema.

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.filetype_list
  - Test: integration.file.filetype.FileTypeTestCase

Obtener los metadatos de un fichero mediante su valor resumen (hash)

```
GET /file/ce6c281c447c5da1c85b97943cc3ab83ba409f636ba070a44c49555d4e410b8e
HTTP/1.1
Accept: application/json
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Con esta llamada el servidor devuelve los metadatos asociados a un fichero.

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.file_by_hash_endpoint
  - Test: integration.file.file.FileTestCase

Obtener el binario de un fichero mediante su hash

```
GET /file/ce6c281c447c5da1c85b97943cc3ab83ba409f636ba070a44c49555d4e410b8e
HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Esta llamada hará que el servidor devuelva el fichero binario asociado a nuestro objeto fichero.

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.file_by_hash_endpoint
  - Test: integration.file.file.FileTestCase

Obtener los metadatos de un fichero mediante su valor resumen (hash)

```
GET /file/ce6c281c447c5da1c85b97943cc3ab83ba409f636ba070a44c49555d4e410b8e
HTTP/1.1
Accept: application/json
Host: 127.0.0.1:8000
Cache-Control: no-cache
```

Con esta llamada el servidor devuelve los metadatos asociados a un fichero.

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.file_by_hash_endpoint
  - Test: integration.file.file.FileTestCase

Actualizar los metadatos de un fichero mediante su hash

```
PUT /file/ce6c281c447c5da1c85b97943cc3ab83ba409f636ba070a44c49555d4e410b8e
HTTP/1.1
Host: 127.0.0.1:8000
Content-Type: application/json
Cache-Control: no-cache

{
   "text": "New text"
}
```

Esta llama es la encargada de modificar los campos permitidos de los metadatos de un objeto fichero.

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.file_by_hash_endpoint
  - Test: integration.file.file.FileTestCase

Eliminar un fichero mediante su hash

```
DELETE /file/ce6c281c447c5da1c85b97943cc3ab83ba409f636ba070a44c49555d4e410b8e
```

HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache

Mediante esta llamada, un usuario con suficientes permisos, puede eliminar un archivo del sistema.

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.file_by_hash_endpoint
  - Test: integration.file.file.FileTestCase

Añadir un nuevo fichero

POST /file/ HTTP/1.1
Host: 127.0.0.1:8000
Cache-Control: no-cache

Content-Type: multipart/form-data; boundary=----
WebKitFormBoundary7MA4YWxkTrZu0gW

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="subject_id"

4

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="uploader_id"

1

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="name"

Filename

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="text"

Description

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="fileType_id"

1

----WebKitFormBoundary7MA4YWxkTrZu0gW

Content-Disposition: form-data; name="file"; filename=""

Content-Type:

----WebKitFormBoundary7MA4YWxkTrZu0gW

Mediante esta llamada, un usuario con suficientes permisos, puede añadir un nuevo archivo a un nivel definido (de tipo asignatura).

Se relaciona con:

- Front-end: app/source/controllers/subjectController.js
- Back-end: routers.file.file_add_endpoint
  - Test: integration.file.file.FileTestCase

Capa 1. Enrutación

La capa de enrutación, la cual se encuentra en el módulo rest.routers ya ha sido explicada en sus objetivos en el capítulo anterior, quedando aún por analizar el resultado de la capa construída.

En primer lugar podemos observar viendo únicamente la división de los ficheros que componen el módulo que se ha respetado los prefijos de ruta utilizados en la API (capa 0). Esto es importante en tanto que viene a reflejar una buena organización y división de las competencias de cada ruta.

Por otro lado se observa el acceso a un módulo adicional en la ruta rest.controllers llamada decorators.
En este módulo se encuentran una serie de funciones, donde si observamos aquellas que son públicas (carentes del prefijo _) se ve como los nombres coinciden con los de los decoradores utilizados en todas las funciones de los enrrutadores. Más concretamente hablamos de los decoradores @authenticated, @method y @methods.
Estos decoradores, llamados así por ser el patrón de diseño que representan, permiten realizar operaciones antes y después de llamar a una función, en este caso aquella que llama al decorador.

Estos decoradores permiten reutilizar funcionalidades necesarias en la mayoría de las funciones de la capa de enrrutación. Más en detalle:

- Es necesario comprobar que el método (o métodos) HTTP utilizado para pasar una llamada a capa de servicios es la esperada. Esta comprobación es competencia de esta capa y por tanto es aquí donde para cada uno de los casos se tiene que hacer la comprobación. Para ello haciendo uso de @method y @methods se comprueba que la llamada cumple las restricciones deseadas.
  En caso de no cumplirse, en todos los casos se devuelve un error indicando que no se ha construido correctamente la llamada HTTP.
  En caso de haberse realizado correctamente la llamada HTTP, se pasarán los datos del cuerpo de la llamada a la función invocadora.

- En una mayoría de casos, es necesario que la llamada HTTP realizada haya sido hecha por un usuario autenticado (logueado) en el sistema. Para ello se hace uso del decorador @authenticated.
  Esta función nos pemite saber de forma sencilla si un usuario se autenticó en el sistema previo a realizar una operación que requiere de ciertos permisos de acceso.
  En caso de no cumplirse esta restricción, se devolverá un error avisando de esta eventualidad.
  Cuando la llamada fue realizada correctamente, a la función invocadora se le pasará de forma específica la cookie (token) que verifica al usuario en el sistema.

Un detalle importante a explicar es la razón de extraer cierta información de las peticiones HTTP que llegan y pasarlas de forma separada a las funciones que utilizan los decoradores. Para poder construir una arquitectura bien cohesionada pero con bajo acoplamiento, es necesario que cada capa tenga conocimiento de un número limitado (y en la medida de lo posible único) de clases y/u objetos.

Por ello, a la capa de servicios (o a las clases de tipo servicio) no se les pasa los objetos que representando las llamadas HTTP, sino que estas clases solo adquieren acceso a un conjunto limitado de datos: el token de autenticación (cookie) y el cuerpo de la llamada.

Al hacer uso de los decoradores mencionados y realizando la extración de datos, se obtiene una capa de enrrutación con un código mínimo y limpio que respeta y se ciñe en el plano práctico a las responsabilidades que se le otorgan de forma teórica.

Capa 2. Servicios

El propósito de esta capa ya fué explicado en el capítulo anterior, siendo el objetivo de este apartado explicar los usos reales que se han conseguido en los distintos servicios desarrollados.

Al analizar el resultado de los servicios escritos, vemos que el acoplamiento predicho que vendría a existir entre nuestros servicios y los (vitaminados) modelo de Django se ha cumplido.

Cuando vemos los llamados modelos en Django, vemos que estos no responden a un concepto original de modelo. No solo definen los atributos de un objeto y exponen funciones que procesen/transforman dichos atributos, adicionalmente a su cometido original, también poseen la capacidad de comunicarse con la base de datos entre algunas otras funciones. Por ello, cuando hablemos de modelos de Django, normalmente se hará refiriéndose a ellos como modelos vitaminados o modelos con acceso a repositorio.

Este acoplamiento existente en sí mismo no es negativo, pues por muy al contrario que pueda parecer un anti-patrón de primeras, en realidad lo que ha venido a suceder es que se han conseguir extraer y des-acoplar competencias de los modelos de Django.
Hay que recordar que entre los objetivos fijados en este proyecto es la demostración de la madurez de los frameworks de desarrollo, significando esto que si es nuestro deseo, podemos flexibilizar el uso y re-estructurar las herramientras y objetos que provee el framework.

Por todo lo anterior, cuando se observa un servicio cualquiera, se observa como tiene una correspondencia exacta con un modelo de Django. P.ej: FileService (servicio) y File (modelo).

Existen de forma subsidiaria un segundo motivo que hacen necesaria la diferenciación entre los servicios y los modelos vitaminados de Django, es el uso de django-rest-framework y sus serializadores.
Esto sucede porque Django está pensado originalmente para servir vistas HTML, no objetos de texto plano (p.ej. JSON).
La capacidad de servir JSON a través de nuestra api lo proporciona django-rest-framework, pero a cambio tendremos que especificar la transformación de los objetos.
La transformación/serialización de los objetos es una competencia dada a los modelos (explicado en el siguiente capítulo).
Pero, una competencia que ha de pertenecer al servicio es discernir la validez de las modificaciones. Es decir, aquellas lógicas (de negocio) que han de conocerse por la manipulación de la información entrante, es responsabilidad del servicio y en tanto que estas lógicas son una parte importante dentro del flujo, se hace necesarioa la existencia esta capa.

Un tercer motivo que complementa al anterior que visibiliza y concluye una buena estructura elegida es el uso de mensajes embebidos en las respuestas. Estos mensajes, tanto de error como de aceptación son escogidos en case a los casos de error que los datos (no ya las peticiones, como se vio en la capa 1) contengan.

Es por todo lo anterior que la existencia de los servicios responde a una necesidad de

diseño y requisitos de proyecto. Por tanto, el mentado acoplamiento existente, es bajo, incluso menor del que pudiera inicialmente preveerse y productivo a la hora de construir el proyecto.

Los servicios desarrollados y a tener en cuenta son:

- AuthService. El servicio de autenticación y entrada al sistema. A través de él realizamos registros de usuarios, recuperación de contraseñas, autenticación de sesión, etc.
    - Relación: models.user.User, services.email.EmailService

- UserService. El servicio para el tratamiento de datos de los usuarios. Es aquí donde dichos datos pueden ser modificados, suprimidos o devueltos, tanto de forma aislada, como en grupo.
  No confundir con las competencias del módulo de autenticación. En este espacio sólo se trata con el paso y/o entrada de datos, no la veracidad y seguridad de los mismos (competencia directa de AuthService).
    - Relación: models.user.User, services.email.EmailService

- FileService. Servicio encargado de trabajar con los servicios, en este espacio añadiremos nuevos ficheros asociados a niveles autorizados, o bien será modificada sus metadatos o serán suprimidos del sistema.
    - Relación: models.user.User, models.level.Level, models.file.File,

- LevelService. En el sistema existen los llamados niveles, entidades que representan niveles organizativos en una organización educativa. P.ej: universidad, curso, clase, asignatura, etc.
  Es en este servicio concreto donde la información relativa a los niveles será expuesta o modificada.
  Como novedad con respecto a los otros servicios y modelos vistos hasta el momento, en este caso no se expone la capacidad de modificar los modelos existentes.
  El motivo de ello es que al ser datos sensibles y críticos del sistema, los cuales ademas no son modificados con frecuencia, su modificación deberá llevarse a cabo mediante el panel de administración externo, no a través de la API en sí.
    - Relación: model.level.Level

- NoteService. En este servicio se lleva a cabo el procesamiento de las noticias (o notas). En este espacio será posible llevar a cabo la modificación, creación, devolución o eliminación de las noticias asociadas al espacio educativo.
    - Relación: models.user.User, models.level.Level, models.note.NoteBoard,

- Rol. De forma similar lo que sucede con los niveles, en el sistema y de forma asociada a las entidades educativas, existen roles de las miembros de estos espacios. Estos roles vienen a ser una de las formas de permisos existentes en el sistema más básicas.
  Igual que en el caso de los niveles, al ser datos que apenas sufren modificaciones a lo largo del tiempo y siendo una cuestión crítica, ni en el servicio ni en la API es posible llevar a cabo modificaciones sobre dichos datos, únicamente su lectura.
    - Relación. models.rol.Rol

Un detalle que es necesario explicar son las relaciones entre servicios y modelos. El

ejemplo más claro de ello son las llamadas tipo “level/{id}/(notes|files)”. Estas llamadas pasan por el enrutador de nivel, pero acceden al servicio de notas o ficheros.
Esto sucede porque la teoría REST invita a la creación de rutas donde el primer nivel vaya en singular, seguido de identificador y el elemento de relación en plural.
Por ello, una buena construcción de las capas 0 y 1 obligan a hacer pasar la llamada por el enrrutador de nivel, capturar sus datos (los identificadores), pero, delegar en los servicios de nivel y ficheros el servicio final de datos.

Capa 3. Repositorio / Base de datos.

En esta capa se encuentran los modelos de Django, los cuales ya han sido explicados brevemente en capítulos anteriores, aunque es necesario hacer algunas puntualizaciones y análisis a la información ya conocida.

En Django, los modelos tienen un nombre que no responde a las competencias de un modelo clásico en teoría del software.
Estos modelos proporcionan acceso a objetos con lógica propia, siendo los más importantes de ellos los “managers”, pues estos son la pasarela de acceso a la base de datos o en otras palabras, los objeto repositorio.

Otro aspecto importante ya anotado anteriormente es que para la idea original de Django, el retorno de los modelos como objetos completos es un buen diseño, pues estos objetos permiten crear otros componentes como las vistas HTML con sencillez.
Sin embargo, esta particularidad no es útil en este proyecto, pues lo que se desea es obtener los objetos ya transformados en objetos más primitivos, es decir, en un formato diccionario (clave-valor) que sirva de antesala a la obtención de una cadena de texto que represente un objeto JSON, siendo este formato el que se desea que sea la salida y respuesta a las peticiones.

Por ello, los modelos desarrollados en este proyecto heredan todos ellos de una clase llamada BaseModel.
Esta clase expone una serie de métodos de clase que permiten acceder a la base de datos con rapidez y obtener los objetos deseados ya en el pre-formato JSON deseado.
Con esta estrategia, se evita dar a los servicios una competencia tal como el formateo de la información, pues esta competencia debe residir en esta capa.

Por otro lado, otro aspecto importante de esta capa son los validadores.
Ya ha sido analizado cómo cada capa tiene una responsabilidad de validación sobre los datos que entran en el sistema y el caso de la capa repositorio no es una excepción.
En este caso, la competencia de validación se basará en comprobar que los datos, a nivel de tipo de datos y forma es correcta.
Es decir, el tipo de primitiva que representa informaciones individuales así como su tamaño y composición interna.
Esto podría confundirse como una responsabilidad de la capa anterior, pero sería erróneo, pues en la capa de servicios se trata con la lógica de la existencia de conjuntos de valores, no su estado interno.

Un último elemento de importancia en esta capa son los mapeadores de objetos de Django.
Es importante saber que si bien las respuesta del servidor son objetos planos, informativamente autocontenidos, teóricamente no relacionales, esta situación no es

cierta a nivel de base de datos.
Lo que ocurre es que Django, una de las funcionalidades que ofrece cuando hacemos uso de los objetos de tipo repositorio, son los mapeadores de objetos.
En otras palabras, la información relacional de la base de datos (en este caso SQL) es enlazada, recogida, juntada en un solo espacio y devuelta de forma autocontenida.

Es por todo lo anterior que se entiende que la capa ha sido construida en base a unas competencias claras y correctas, que la extensión que ha sufrido una capa ya de por sí con múltiples responsabilidades está sobradamente justificado y ayuda a mantener una buena cohesión entre las capas.

Como cierre, recordar algunos de los modelos mas importantes en el sistema:

- File. Este modelo representa la meta información asociada a un fichero así como el mismo binario. En el modelo Django conviven la totalidad de esta información y en la capa servicios se hace el paso de una u otra información.
- Level. Con este modelo se representan las diferentes estructuras y partes que componen una institución educativa (escuelas, cursos, asignaturas, etc). El modelo contiene entre otros campos que determinan la posición dentro de una organización.
    - LevelType. Para permitir conocer el tipo de nivel en un momento determinado se utiliza este modelo adicional de forma que se consigue respetar la teoría de las formas normales en bases de datos relaciones.
- BaseMessage. Este modelo es especial debido a su carácter abstracto. Esta clase servirá de base a las clases hijas. El propósito general de esta clase es representar un mensaje de retorno de HTTP. Las particularidades de los mensajes de error y corrección impiden hacer uso de una única clase, sin embargo, con el fin de respetar las buenas prácticas de programación, esta clase se hace necesaria.
  Otra particularidad de esta clase, común a sus hijas, es el uso de la clase anidadad “Type”, la cual es de tipo enumerado. Con dicha clase se consigue acceder con rapidez a objetos concretos haciendo únicamente referencia a su nombre en clave.
    - OkMessage. Con este modelo se representan los mensajes devueltos en algunas llamadas cuando la petición fue satisfactoriamente resuelta.
    - ErrorMessage. Esta clase es el modelo utilizado cuando una petición de red no fue correcta en alguna de sus formas.
- NoteBoard. Este modelo representa noticias del sistema.
- User. Modelo utilizado para tratar a los usuarios del sistema.
  Como particularidad mencionar que Django proporciona un modelo de usuarios por defecto, pero que debido al acoplamiento que presentaba el modelo de

autenticación propuesto en este caso, se optó por escribir este modelo (y el módulo de autenticación) desde cero.

- Rol. Como parte de la determinación de los permisos de un usuario a la hora de realizar ciertas acciones es necesario determinar una jerarquía dentro de la organización. Con este módulo se consigue dicho propósito y se consiguen estructurar los diversos niveles, así como sus permisos, con sencillez.

Elementos transversales a las capas.

En el código servidor, algunos elementos existentes o bien no encajan de forma única en una sola capa o bien son utilizados de forma transversar a todas ellas.
Es por ello que estos módulos son analizados en el presente apartado de forma separada a la visión de capas.

Serializadores

Este conjunto de clases, son los encargados de proveer una interfaz de interpretación y traducción entre un objeto primitivo (antesala del formato JSON) y un objeto heredero de models.Model perteneciente a los módulos de mapeo de objetos de Django.

En términos más concretos, este módulo tiene la capacidad de generar un diccionario (pares de clave-valor) a partir de un objeto (incluso compuesto) de tipo Model.
Incluso, sin mayores problemas, puede realizarse el paso inverso.

En este punto es necesario explicar que si bien a través de django-rest-framework provee opciones de serialización/deserialización, por cuestiones de composición, complejidad de algunos casos, falta de cumplimiento de principios REST y rigidez general del sistema, se optó por la generación de funciones propias de serialización.

Aún más en detalle. En nuestro caso, por cuestiones de seguridad es necesario restringir según casos el acceso a la edición de algunos campos.

Es por ello que se opta por imitar el estilo del framework django-rest-framework, pero implementando unas serie de funcionaliades propias.

De esta forma, podemos elegir rápidamente qué campos pueden ser editados o qué campos pueden considerarse opcionales (evitando bajar las últimas capas para descubrir el error y así lanzar mensajes de error personalizado).

En cualquier caso, no se considera un ejemplo de sobrefactorización/sobreingeniería en tanto que está justificada la sobreescritura de funcionalidades en tanto las mismas por rigidez y falta de madurez no aportan la versatilidad necesaria para nuestra situación.

Excepciones ligadas a los errores.

Si se observa en el código la clase MessageBasedException[3] se encontrará una clase de heredera de la clase Exception.

El fin de esta clase es proveer una excepción que contenga información útil para ser devuelta como respuesta a una petición.
Esta situación no es posible y/o sencilla haciendo uso de otras excepciones del lenguaje o de Django, pues no son es natural a estas excepciones devolver mensajes asociados al sistema construido.

Por ello, esta clase resulta de gran utilizad a la hora de recoger las excepciones lanzadas y transformarlas en un objeto con información interpretable con rapidez por el sistema.

El funcionamiento de esta clase es sencillo, es posible lanzar una excepción con un mensaje de error asociado y de forma opcional, es posible incluir la excepción original que produjo el error y recorrer los mensajes de error almacenados en esta.

De esta forma, en el decorador @map_excepctions se garantiza la transformación de cualquier excepción en nuestra excepción deseada.
Y posteriormente, haciendo uso de @zero_exceptions, se garantiza que una excepción de tipo MessageBasedException producirá una respuesta de tipo JSON con los debidos mensjaes incluídos en la misma.

Todo este proceso permite reducir la captura de excepciones en lugares muy concretos del código, mejorando enormemente la legibilidad del mismo.

Factoría de respuestas HTTP

Si se observa la clase ResponseFactory [4], se encontrará una clase que implementa el patrón constructor (Builder) y que su cometido en la construcción de un objeto HttpResponse que contenga la necesaria información de una petición HTTP de respuesta.

Esta clase expone una serie de métodos que son necesarios conocer:

- Ok/Error. Estos dos métodos permiten definir si estamos ante una llamada que fué correcta o no.
  Ambas dos son muy similares, pues hacen uso de clases heredadas de BaseMessage para conocer el mensaje a pasar en la respuesta HTTP a devolver.
  Únicamente “error” es algo más compleja en tanto que añade la posible información asociada a las excepciones que pudiera existir.
- Body. Permite definir el cuerpo de la respuesta HTTP. Este cuerpo puede ser tanto complejo (un objeto heredero de los modelos de Django) o pre-formateado (preparado para ser convertido a JSON).
  Este método es tanto útil en llamadas GET (uso clásico) como en llamadas PUT.
- Identity. Es muy similar a “body”, con la particularidad de que en este caso solo se expone la id del objeto dado. Este método es muy útil a la hora de dar información de utilidad en llamadas de tipo POST o incluso PUT.
- Cookies. Permiten definir esta cabecera en la futura llamada HTTP a devolver.
  Considerando la versatilidad de llamadas que hacen uso de la factoría es necesario exponer este método con el fin de capacitar la definición de las cookies que se usarán. Esto es necesario, entre otros, en los casos de autenticación o salida del sistema.

Pruebas

Durante la construcción del servidor, de forma paralela, un conjunto de pruebas fueron desarrolladas con el fin de probar la validez y correción del código y funcionalidades escritas.

Estas pruebas encajan en el tipo de test de integración, es decir, pruebas que tratan al sistema en su totalidad como un solo elemento al que se le dan entradas y se analizan sus salidas.
La elección de la escritura de tests de integración en detrimento de los tests unitarios obedecen a varios elementos.

- El hecho de basar el código fuertemente en un framework hacen de los tests unitarios algo redundante, pues los frameworks ya se entienden probados contra su propio conjunto de test.
- El hecho indiscutible de una limitación de tiempo y recursos para llevar a cabo el desarrollo la codificación. En tanto que existe una limitación tangible de las capacidades de desarrollo, es necesario priorizar la correción de los casos de uso (que es el foco de los tests de integración) sobre las funcionalidades concretas de capa módulo del sistema.

En cualquier caso, el desarrollo de las pruebas unitarias se marca como una de las futuras amplicaciones que habría de tener el proyecto.

Test de integración

Para la escritura de las pruebas de integración se ha hecho uso de la clase TestCase ofrecida por Django.
Esta clase nos permite llevar a cabo los test de integración de forma aislada entre sí, permitiendo la división lógica de los mismos en clases e identificándose las pruebas en base al prefijo “test_” en nombre de los métodos.

Las pruebas serán independientes entre sí por defecto y permitiendo que las pruebas pertenecientes a la misma clase compartan una base de datos común, reiniciada entre prueba y prueba. Por tanto, el orden de la ejecución de los tests no afecta al resultado parcial o final del conjunto de pruebas.

En Django, las pruebas originalmente ofrecidas por el framework consiste en un único fichero llamado “tests.py”.
Esta organización de las pruebas no es en absoluto práctico y se ha optado por otorgar a cada una de las clases de tipo test, un fichero propio. Esta división mejora la compresión de las pruebas escritas, su organización e incluso su relación con el código del servidor en

sí.
Para poder llevar a cabo este cambio en necesario bordear los sistema de Django ligeramente.

Cuando Django intenta ejecutar pruebas asociadas a su framework, carga una carpeta/módulo indicado e intenta localizar aquellas clases y sus métodos que representan tests. En caso de encontrar tests, levanta una base de datos separada para las pruebas y ejecuta las pruebas.

La trampa reside en hacer uso de imports en el fichero __init__.py del módulo "tests.integration" para puentear el proceso y hacer que Django llegue a las clases directamente, pues en Python la carga de clases es reactiva a lo indicado en los __init__ de cada módulo y no es ni recursiva ni automática.

Una vez se ha creado esta estructura, es posible elegir la mejor organización en módulos/carpetas que se quiera, pues ya se tiene la garantía de que las pruebas serán ejecutadas.

De igual forma, con el fin de construir un sistema preparado para la integración continua y despliegue continuo, independientemente del sistema que lo analice, el fichero "run-tests-integration.sh" permite ejecutar las pruebas en un entorno shell.

Durante la escritura de las pruebas se fijó como obligatorio escribir al menos un caso de uso positivo y al menos uno negativo por cada mensaje de error que el sistema podría llegar a dar ante una petición mal construida.
Es por ello, que contando un número de rutas de API algo superior a la veintena, existen cerca de un centenar de pruebas de integración.

Utilidades creadas para las pruebas

Con el fin de hacer más sencilla la tarea de condificación de las pruebas y ante la evidencia del patrones comunes en todas las pruebas, algunas funciones y clases han sido escritas para hacer más rápido el proceso y seguir la filosofía KISS. Entre otros:

- Los clientes. Django ofrece un atributo "client" en su clase TestCase y que permite simular peticiones HTTP. Para simular correctamente que se está haciendo uso de las cabeceras json durante las llamadas, dicho atributo fue sobreescrito con la clase JSONClient con el fin de llevar a cabo de forma sistemática el uso de dichas cabeceras.
  Para los casos en que es necesario el envío de ficheros (cabeceras de tipo form-data), se hace uso del cliente por defecto de Django, el cual hará uso de estas misma cabecera.
- La base de datos de pruebas. Como se mencionó brevemente con anterioridad, Django crea una base de datos separada y temporal para la ejecucción de las

pruebas. Como dicha base de datos está vacía por defecto, es necesario dotarla de diversos datos al arrancar el sistema.
Existiendo varias estrategias para ello, para mantener la integridad y evitar la duplicidad de los datos, se ejecuta la misma llamada que durante la provisión. Lo que en Django se llama carga de “fixtures”.
Esta estrategia de carga de datos permite evitar duplicidades y garantizar que estamos haciendo pruebas con los datos núcleo reales de un entorno de producción.

- Asercción de resultados. En la inmensa mayoría de los casos, se desea comprobar que la respuesta devuelta por el servidor es la esperada.
  Para hacer esta comprobación siempre llevaremos a cabo una comprobación inicial del código de respuesta y de los datos de mensaje devueltos.
  Dado que estas comprobaciones son recurrentes, se han desarrollado una serie de funciones (assert_*_response).
  Estas funciones permiten hacer rápido y sencillo el proceso de testing gracias al uso de las clases de mensajes “OkMessage” y “ErrorMessage” así como sus enumerados que hacen aún más rápido e intuitivo el proceso.

El cliente web

Como ya fue visto en el capítulo anterior, las diferencias entre una arquitectura en un cliente web y un servidor son enormes y en muchos aspectos no son comparables, siendo uno de los principales motivos la existencia (o no) de una interfaz gráfica y la persistencia de datos en uno y otro caso.

En el caso del cliente web, la enorme versatilidad del framework utilizado en este caso, sumado al protagonismo del servidor (y sus funciones) en este proyecto hace que la proporción de código entre uno y otro sea de 3:1. Es decir, tres líneas en el servidor por cada en el cliente web.

Es por ello, que el capítulo que viene a continuación, si bien es de gran importancia, su alcance es algo menor en los objetivos marcados hacia el proyecto y así en tanto lo será su extensión en el análisis de los resultados.

Factorias

Como ya se explicó en el anterior capítulo, las factorías de Angular permiten enlazar un conjunto de funciones y datos en un mismo objeto y mantenerlo compartido a lo largo de la aplicación web.

A la hora de estructurar el cliente web, un componente que permite mantener su estado de forma compartida a lo largo de más componentes supone una gran utilidad a la hora de evitar un consumo de memoria o llamadas en red innecesarias, permitiéndose así una mayor eficiencia en el cliente.

Por otra parte, la sencillez de las factorías para ser construidas y utilizadas, así como su la sencillez de su resultado final permiten cumplir con los objetivos marcados en este proyecto a la hora de demostrar un desarrollo rápido, confiado en el framework y altamente legible.

Así mismo, es necesario analizar brevemente las factorías desarolladas y los motivos que hacen que los componentes desarrollados hayan sido construídos bajo este patrón y no otros:

- Api. Esta factoría permite la comunicación con el servidor. Este módulo es el último punto de comunicación dentro del cliente y es quien es encargará tanto de realizar las pertinentes comunicaciones con el debido servidor como de proveer una API pública dentro del cliente front-end que permita conocer las opciones de comunicación con el servidor.

  Dento de las mejoras necesarias observadas, sería hacer más rico en funcionalidades a esta factoría y permitir resolver parcialmente las promesas de las llamadas de red, así como dividir la factoría a lo largo de varios ficheros, agrupando por funcionalidades comunes para mejorar su legibildiad y compresión.

- SubjectsTree. El árbol de organizativo de nuestra entidad de estudio es un concepto muy reutilizado a lo largo de diversas vistas. Por ello es necesario poder disponer de un componente único que permita acceder a él de forma sencilla y con un uso ligero en memoria.
  Debido a esta necesidad, surge esta factoría, donde se permite recuperar tanto la jerarquía completa como parcial.
  Al hacerse un uso de promesas, el proceso se mantiene asíncrono y no bloqueante para el usuario, permitiendo obtener una buena experiencia de uso.

- User. La factoría que permite manejar la información del usuario actualmente

autenticado en el sistema.
Es gracias a esta sencilla factoría que es posible mantener los valores de un usuario compartido y actualizado en todo momento a lo largo de todo el sistema.

Directivas

Las directivas son consideradas la antesala de un patrón de diseño muy común en la actualidad como son los componentes web.
Al fin y al cabo las directivas de Angular permiten encapsular cierto código HTML a una serie de funciones en javascript y permitir enlazar estos componentes con sencillez directamente en nuestro código HTML y manteniendo estas parte autónomas al resto de los elementos de las vistas (y sus objetos Javascript asociados).

En el caso de el presente proyecto, el uso de las directivas se observó como una necesidad real cuando se hacía necesario mantener en múltiples vistas una serie de elementos recurrentes como han sido las barra lateral y superior. Más en detalle:

NavBar

En el presente proyecto encontramos que en el apartado web algunas vistas poseen una barra de navegación y otras no. En general puede dividirse que aquellas que pertenecen al ámbito de las vistas post-loguin poseen un navbar y el resto no.
Es por ello que es necesario que exista un módulo específico para la gestión de dicha barra de navegación.
Como es de suponer por lo exlicado anteriormente, el tipo de módulo de Angular elegido para desarrollar esta parte del proyecto es una directiva.

Hay que considerar que en Angular, por defecto, la anidación de vistas no está soportado. Es cierto que existen librerías de terceros que permiten esta funcionalidad explotando los límites del sistema, pero en tanto que las directivas suponen una excelente forma de cumplir nuestros objetivos y respetando la arquitectura original del framework, se opta porque esta sea nuestra elección.

Hay que destacar que el controlador de nuestra directiva será mínimo y solo provee una serie de funcionalidades mínimas de navegación, siendo una forma de uso de los controladores aún más minimalista que en el resto de vistas&controladores.

Sidebar

Al igual que sucede con el NavBar, en nuestro cliente web nos encontramos con que en la mayoría de casos (aunque no en todos ellos), es necesario disponer de un fragmento de código HTML con un comportamiento asociado. En este caso particular tratamos el concepto de sidebar y como en esta integramos información de acceso rápido para simplificar la navegación a lo largo de la web.

Igual que ocurría con el Navbar, habrá ocasiones en que sea necesaio disponer de una barra lateral y otras veces no, por lo que se opta por utilizar una directiva para la implementación.

En términos generales, ambos dos módulos son muy parecidos, pues en ambos dos casos se busca proveer una rápida navegación al usuario, siendo quizás la característica más diferenciadora que el sidebar depende en mayor medida de los datos y relaciones de usuario en tanto que este muestra información relacionada con el mismo de una forma mucho más completa y personalizada de lo que lo hace el navbar.
Además de que por su diseño vertical está pensado para ser extendido en funcionalidades dinámicas, una característica que el navbar tiene más limitada.

Controladores

Finalmente, entre los patrones de Angular utilizados en este proyecto se encuentran los controladores.

Como ya he mencionado anteriormente, los controladores hacen las veces de piedra angular entre las vistas los componentes reutilizables como las directivas Y las factorías.

Así se ve que existe un relación intrínseca entre todos controladores, las vistas y diversos niveles lógicos de negocio mencionados en el capítulo dedicado al servidor y sus diferentes divisiones.

Así, se ve de forma recurrente la relación entre una entidad lógica en el lado servidor, un grupo de llamadas en la factoría API y su uso en un conjunto definido y relacionado de controladores.
Este resultado en la construcción de los componentes, cuando se ve el efecto en los controladores permiten ver como se cumplen los objetivos a nivel arquitectura a la hora de una construcción limpia, haciendo uso de las relaciones y patrones necesarios en cada momento.

Finalmente, es necesario hacer un análisis de los distintos controladores construidos.

LoginController

Únicamente se encarga de realizar la llamada API “login” a través de la factoría “API”. Como se ha explicado anteriormente, en base a los principios de desacoplamiento de los componentes de interfaz, el resultado de la llamada se traslada al usuario a través de un snackbar.

En los casos en que se produce un ingreso de credenciales correcto, se redirige a la página principal de la plataforma (/dashboard).

SignUpController

El controlador encargado de gestionar las nuevas altas en el sistema. Únicamente es necesario introducir un correo (de la universidad), un nombre de usuario y una contraseña. Si los datos de registro son correctos la vista cambiará para mostrar al usuario los próximos pasos a seguir en el registro (activación de la cuenta mediante la confirmación del correo).

ConfirmEmailController.

De forma similar al controlador signUpController.js, nuestra vista únicamente realiza el enlace entre la iteración del usuario con el HTML y la llamada API que confirma correos en

el servidor (recoverPassword).

Como se ha explicado en secciones anteriores y como se ha visto en vistas anteriores, este diseño simple busca utilizar a los controladores como pegamento de unión entre el HTML y las factorías, manteniendo una gran simplicidad y legibilidad a nivel de código, obteniéndose una arquitectura modular y escalable.

Es una de la pocas vistas que requiere de una variable para el correcto funcionamiento de la misma. Dicha variable no forma parte de la URL, si no que se envía dentro de los campos de formulario que proporciona las llamdas de tipo POST.
A nivel interno es la cookie de sesión que estaba presente en el equipo del usuario registrado en el momento del alta.

RecoverPasswordController

Al igual que se ha visto en los casos de logueo, registro y similares, nuestro controlador únicamente tiene por código el enlace entre el HTML y la llamada API correspondiente.

En concreto, el controlador recibe un correo que es el enviado en la llamada API y espera una respuesta.

Manteniéndose el patrón de diseño, la comunicación de los resultado de las llamadas de red, está desacoplado y se informa al usuario mediante los ya conocidos snackBars.

SubjectsController

Esta vista se apoya principalmente en la lógica creada en el front-end partiendo de una única llamada a la API.

En concreto, el controlador pide vía subjectsTree una relación anidada de los distintos niveles de la universidad, partiendo de la universidad en el nivel más alto y llegando hasta las asignaturas de cada curso.

El controlador se encargara de que una vez obtenido correctamente esta información, la navergar entre las opciones, se muestre en la vista las opciones anidadas sucesivas.

SubjectController

A nivel de código javascript/Angular, el fichero de la vista contiene 3 controladores. El principal de la vista y dos correspondientes a las ventanas modales de visualización/edición y subida de ficheros.

En el controlador principal se ejecutan las funciones de obtención de la información de la asignatura en cuestión (a través de la factoría de asignaturas, subjectsTreeFactory).
Una vez se tiene la información de la asignatura, a continuación se obtiene mediante llamada a la API el listado de ficheros de la asignatura que estemos visualizando.

Este controlador tiene una serie de controladores subsidiarios asociados de forma interna:

- ModalNewFileInfo. En el caso de que se desee subir un fichero a la plataforma (arrastrando ficheros a la ventana), se activará el controlador auxiliar de subida de ficheros, el cual a nivel de interfaz, lanza la venta modal.
  Este controlador es el que se encarga de solicitar al usuario información relativa al fichero que está subiendo (tipo, nombre, etc) y realiza posteriormente el enlace a la API para la subida del fichero a través de la llamada uploadFile.

- ModalEditFileInfo. Este controlador permite visualizar la información de un fichero que ya está en nuestro sistema, editar su información (si tenemos los permisos adecuados) y acceder al fichero en sí.

  Este controlador accede a varias llamadas de la API, de forma directa accedemos a las llamadas DELETE y POST del fichero, las cuales ejecutan las llamadas de eliminación y modificación respectivamente. De forma indirecta accedemos posteriormente a la llamada [GET] file/f/{id}, la cual es la encargada de devolver el fichero en sí que estamos visualizando.

EditSubjectsController

En este controlador realizamos en enlace entre una lista en nuestro HTML y la llamada API que actualiza los valores para el usuario logueado en el sistema.

En el Html se representa el arbol de jerarquía de la universidad en cuestión, teniendo las asignaturas un checkbox asociado, donde se marca si el usuario está cursando la asignatura en cuestión.

Posteriormente, en nuestro controlador, procesaremos la lista de asignaturas que tienen su checkbox marcado y enviaremos dicha lista en forma de array dentro del form-data de la llamada [POST] editSubjects/.

ProfileController

En el controlador tenemos las funciones que realizan el enlace con la API para las llamadas que actualizan la información personal del usuario y su foto de perfil (en una llamada separada).

Fuera de estas dos responsabilidades, el controlador de esta vista no tiene mayores responsabilidades, existen un par de valores para controlar el modelo de datos y la activación del modo edición.

NotesController

En este fichero encontramos un controlador principal para la gestión principal de la vista y

3 controladores auxiliares que gestionan los casos de visualización, edición, alta y restricción de nivel.

En el caso del controlador general y al igual que se hace en el resto de vistas del proyecto, creamos el enlace entre la vista y la llamada API correspondiente (notesByLevelId).

Este controlador tiene una serie de controladores subsidiarios asociados de forma interna:

- ModalLevelsLists. Tiene por competencia el filtrado por nivel. Este controlador posee una ventana modal que muestra el nido de niveles de la universidad en cuestión y nos permite elegir un nivel a partir de el cual mostrar las noticias asociadas. Posteriormente, encontramos otros dos controladores relacionados directamente con el concepto noticia como tal.
- ModalNewNote. Gestiona la creación de noticias, realizando la labor de enlace con la llamada API debida.
- ModalEditNote. Gestiona la visualización de una nota/noticia así como su edición.

Otros componentes y librerías

Alrededor de los componentes creados partiendo de patrones ofrecidos por Angular, existen un conjunto de otros tantos componentes que no encajan en las divisiones anteriormente analizadas y expuestas.

Esto ocurre en su mayoría al estar hablando de librerías que han sido integradas en el sistemas a través del uso de gestores de dependencias, los cuales, por su y funcionalidad ya han sido explicados.

Sin embargo, independientemente de su integración automatizada y delegada en el sistema, sus funcionalidades y relaciones con los componentes creados no pueden ser ignorados.

App

Es el modulo principal del proyecto en el cliente web.
Este modulo sirve de punto de entrada principal al resto de módulos, así como realiza las tareas de gestión de la enrutación.

Es decir, en angular en la mayoría de casos, se define un módulo principal de tipo genérico en torno al cual se organizan el resto de módulos de propósito específico.

Este módulo genérico indica las dependencias/librerías globales de las cuales tendrá capacidad de uso el resto de módulos.

De igual forma, tal y como se mencionaba anteriormente, es en este módulo de propósito general donde se realizan las tareas de enrutamiento y control de rutas de la webapp.

Es decir, de acuerdo a las capacidades ofertadas por el framework, las rutas introducidas en nuestro navegador web pueden ser condicionadas a la redirección a una u otras vistas o el envío a una página general de eror.

Snacks

Al igual que en el apartado del servidor los mensajes cobraban una especial antención, de forma equivalente y paralela, los mensajes de sistema cobran importancia en el cliente web.

En términos concretos se ha optado por el uso de una librería que imprime snacks en pantalla, el cual ha sido modificado para que mediante el uso de códigos de color, se indique la severidad del mensaje.

Hay que explicar que los snacks son un concepto moldeado por Google con el motivo del

lanzamiento de su lenguaje visual bautizado como Google Material.

Este patrón visual es muy práctico a la hora de desacoplar los mensajes de error de la vista específica en que que produzcan. De esta forma se evita que tengan que realizarse bindings o asociaciones repetidamente a lo largo del código de las vistas para mostrar los distintos mensajes del servidor en nuestra pantalla.

Como curiosidad destacar que para simplificar y hacer más robusto el código de la librería, esta soporta que le sea enviado la totalidad de la respuesta del servidor (y posteriormente busca la clave pertinente en el mensaje), dando como se decía antes un mayor control sobre los posibles casos de error.

Upload

$upload es una directiva desarrollada por @danialfarid que controla todo el proceso de envío de ficheros desde el navegador hasta el servidor.

Esta librería es muy importante este proyecto pues hace sencillo el proceso de subida de ficheros al servidor, permitiendo que sea sencillo compartir y colaborar en los contenidos.

A nivel técnico de esta librería solo se aprovechan parte de sus funcionalidaes, esto se debe a que si bien el código de declaración del área Drag&Drop es muy útil, el resto de funcionalidades ya se encuentran definidas de forma propia, bien en el front-end o en el back-end.

Es decir, no es necesario que la directiva $upload maneje la petición POST que envía el fichero (y el formulario asociado) al servidor, nuestra factoría API ya se encarga de ello sin problemas.

La elección de seleccionar sólo una parte de las funcionalidades viene fomentado en gran parte porque la integración pura con python por parte de $upload no era la mejor solución. En el repositorio de la librería se indican varios ejemplos de código back-end para soportar la petición generada, sin embargo python no se encuentra entre uno de ellos. Esto supone una falla de documentación, sumado al hecho de que la propuesta de estructura HTTP generada por la librería choca parcialmente con el diseño ya implementado en nuestro servidor.

Son la suma de esta serie de circunstancias lo que invita a utilizar la librería únicamente en el estado inicial de captura de fichero por D&D, pero quedando el resto del procedimiento bajo implementación propia.

Loading bar

Esta librería nos permitirá mostrar progresos de carga en nuestro navegador.

En un esfuerzo por mantener desacoplados los conceptos visuales de alerta e información de los elementos que los originan (como se vio anteriomente en el apartado de las snacks, si un formulario falla, no es el formulario quien muestra el mensaje de error, sino nuestras

snacks), esta librería supone una continuación de dicha política.

La librería angular-loading-bar nos permitirá mostrar los progresos de una petición HTTP, de carga de un recurso, etc, etc, de una forma increíblemente sencilla, pues este módulo simplemente es indicado como una dependencia general de la aplicación y el resto es gestionado por la librería.

Como se comentaba anteriormente, el que la librería sea autónoma permite mantener desacoplados los conceptos visuales de los elementos precursores, simplificándose la interfaz, los casos de uso y mejorando notablemente la experiencia de usuario.

Pruebas

A diferencia del caso del servidor, el cliente web carece de un conjunto de pruebas que permitan verificarlo.

Esta ausencia ha sido fruto de una decisión tomada en base a varios elementos:

- En primer lugar y como ya ha sido mencionado en algún momento de esta memoria, haciendo una recapitulación del tamaño del proyecto, los objetivos marcados y la evidencia de una limitación de recursos personales y temporales hacen imposible abarcar la absoluta totalidad de los elementos que se hacen necesarios en un proyecto en un ambiente laboral.
  En tanto que se está tratando con un proyecto de carácter académico y atendiendo a las debidas restricciones, es necesario priorizar algunos elementos sobre otros, quedando las pruebas contra el código del cliente web en uno de los últimos puestos.
- El motivo de la baja prioridad de las pruebas del cliente web son dos
  - Los tests unitarios de esta parte habrían de basarse principalmente en aquellos componentes con relación con la comunicación en red y el comportamiento de los objetos.
    Estas pruebasn habrían requerido de una inversión de recursos superiores a los con los que se contaban.
    Igualmente, para unos tests verdaderamente útiles para comprobar la validez de lo construído se hacía necesario llevar a cabo una refactorización que permitiera hacer testeable la plataforma, sin que esto hubiese supuesto una necesaria mejora de la arquitectura final.
  - Los tests de integración visual requieren de un conjunto de herramientas y una definición de flujos muy elevada como para ser asumida en este proyecto.

Es por todo lo anterior, que si bien se entiende que las pruebas en este apartado hubiesen resultado útiles y sin que lo anterior signifique en entornos de producción fuesen obligatorias, el hecho final es que la realización de las pruebas para el cliente web no han sido realizadas.

Conclusiones

Autocrítica, mejoras futuras.

Algunos aspectos de este proyecto han podido quedar deslucidos por la falta de experiencia personal así como por la descompensación entre objetivos y recursos para llevarlos a cabo.

En este sentido, algunos aspectos técnicos han quedado con mejoras pendientes, entre otros cabría destacar:

- Refactorizacion de los usuarios para hacer uso de los modelos de usuario aportados por Django, así como su módulo de autenticación.
  Si bien se mantiene la defensa sobre la decisión y construcción realizada, en tanto se pierde la capacidad de acceder a las actualizaciones que sufre el framework, cabe buscar una integración de unos y otros modelos.
- Tests en el cliente web. Esto ya fue analizado debidamente, las pruebas de cliente son necesarias si se busca hacer crecer el proyecto y por tanto en futuros pasos del proyecto, deberían ser llevados a cabo.
- Un servidor más preparado para entornos de producción. Esto implica la integración de sistemas de logs y alertas sobre uso de la máquina, pero también integrar los despliegues y provisiones en herramientas especializadas a tal efecto.
- Pruebas de stress y rendimiento en el servidor. En tanto se hable de entornos de producción, este tipo de pruebas son críticas para la evolución de un proyecto.
- Actualización radical del cliente web. Para lo bueno y para lo malo, el mundo del front-end y sus herramientas evolucionan a una enorme rapidez. En el tiempo de desarrollo de este proyecto, algunas de las librerías utilizadas en el cliente web quedaron retrasadas en versión y funcionalidad.
  Cuando se busca la excelencia y un proyecto en producción, el código obsoleto no es una opción y por tanto habría de buscarse una actualización y puesta a punto en este aspecto.
  En el servidor no es necesaria esta actualización, pues se ha hecho uso de la última versión de Django y una versión elevada de Python.
- Panel de administracion. De cara a un uso en producción de este proyecto, se hace muy prioritario desarrollar un panel de administración que simplifique ciertos procesos a un número limitado de usuarios con privilegios elevados.

Conclusiones técnicas

Llegados a este punto es necesario recapitular sobre el resultado final de todo este proyecto y preguntarse si existe la debida correlación entre los objetivos declarados y perseguidos desde el principio con los resultados finalmente conseguidos.

Durante el capítulo dedicado al análisis del apartado técnico ya se ha venido realizando este mismo desarrollo, aunque con una perspectiva muy corta enfocada en aspectos muy específicos y técnicos.

Es por ello, que en este apartado ve vendrá a realizar un análisis y recapitulación desde una posición más abstracta y elevada.

Servidor.

El pilar principal en este proyecto por su tamaño e importancia.

Finalmente, el framework escogido (Django) resulto ser la elección adecuada. Ha permitido obtener un resultado de un gran nivel técnico en pocas líneas.
Ha demostrado una gran flexibilidad para la construcción de modelos complejos y una gran versatilidad a la hora de permitir construcciones fuera de sus planteamientos originales.

Esto no hubiese sido en absoluto posible sin Python y su dinamismo cuando se observan las partes desarrolladas fuera del planteamiento de Django.
Pero tampoco podría haberse llegado tan lejos con tan poco código de no ser por el cierre conceptual y funcional que ofrede Django-rest-framework.

Estos tres elementos sumados y alineados correctamente han permitido cumplir con satisfacción el objetivo de demostrar la madurez, rapidez y flexibilidad de los lenguajes y frameworks existentes hoy en día.

No hay que olvidar tampoco que entre los objetivos inicialmente perseguidos estaba la construcción de una API fundamentada en principios RESTs que permitiese desacoplar cliente y servidor manteniéndolos en comunicación a través de un contrato/interfaz sencilla y potente.
Este objetivo se ve claramente cumplido, los principios Rest han sido plasmados en la API construída, pero además, la correcta compresión de los mismos ayudó a conseguir que las arquitecturas fruto de la API estuviesen bien construídas y siguiesen unas buenas prácticas en términos de construcción de software.

Aunque no era un objetivo incialmente enunciado ni priorizado, la totalidad de la pila técnica escogida ha sido de gran utilidad a la hora de desarrollar los scripts y sistema de

despliegue y desarrollo.
Un aspecto que de haber sido descuidado podría haber sido un auténtico problema en términos reales, ha sido muy satisfactoriamente resuelto simplemente haciendo uno de Django y un conjunto reducido de scripts en Bash desarrollados para este proyecto.

Cliente web

En el cliente web se buscaban varios objetivos, algunos de ellos compartidos con el servidor en tanto se persigue la construcción de un software limpio, sencillo y apoyado en un framework en el que delegar los aspectos mas farragosos de la programación.

En este caso, Angular ha cumplido su función muy satisfactoriamente, demostrando que en Javascript se encuentra en un momento de madurez y mejora increíble y que es posible desarrollar arquitecturas y componentes bien cohesionados entre sí y respetuosos con las buenas prácticas de la programación.
En el cliente web se ha visto además del uso de gestores de tareas para hacer tanto del desarrollo como del entorno de producción algo ágil.

Igualmente, una gran parte de la sencillez a la par que potencia del cliente web construido se debe gracias al uso de gestores de dependencias que han permitido integrar con enorme flexibilidad nuevos componentes y funcionalidades ya existentes en el sistema.

Por otro lado no hay que olvidar que unido a lo anterior se encuentra la experiencia visual de los usuarios.
Este aspecto, exclusivo del cliente web era un de los aspectos más importantes del cliente web y nuevamente, la integración de librerías y herramientas como bootstrap, compass y componentes de Angular han permitido cubrir la necesidad y el objetivo muy meritoriamente y con una relación de coste en recursos muy asumible.

Conclusiones finales

Llegados a este punto cabe realizar una profunda reflexión sobre la suma de las dos partes técnicas principales, que representan juntas y cómo vienen (o no) a dar respuesta a los objetivos marcados en sus aspectos lógicos y de negocio.

Como ya ha sido explicado, ambas dos partes son dos piezas de software que han sido construídas de forma respetuosa con las buenas prácticas de programación, representando qué debe ser un buen software. O representando la totalidad de su proceso cómo debe lucir un proceso de ingeniería de software bien llevado desde el principio hasta el final.

No menos importante, todo el proceso ha sido transparente y colaborativo en su aspecto técnico. Se buscó desde el principio y se ha conseguido, el proyecto está apoyado en las comunidades de software libre y los proyectos que estas mantienen.
El código escrito, las arquitecturas desarrolladas, toda la suma de elementos tienen un fuerte componente de conocimiento colaborativo.
El valor devuelto a la comunidad en este aspecto no es muy grande en tanto que hasta este punto solo se ha hecho un desarrollo académico de un problema, sin embargo, en muchos aspectos, este proyecto tiene la capacidad de devolver parte de lo desarrollado a la comunidad. Bien por la experiencia obtenida en el uso de los framework en su visión oficial, como en los apartados en los que se ha trabajado fuera del camino marcado para conseguir nuevos enfoques y soluciones.

Si bien se defiende la corrección púramente técnica de lo obtenido, aún queda preguntarse si este proyecto da respuesta a una necesidad, al objetivo perseguido de mejora para las comunidades educativas.

En este aspecto cabe destacar que el proyecto vendría a identificarse con los conceptos de sencillez, confianza, modularidad, escalabilidad, etc.
Esto, en tanto además está arropado con una buena experiencia visual y de usuario, hace que el proyecto desarrollado sea un producto resolutivo para las necesarias mejoras que las plataformas de e-ducación necesitan en este momento.

Y este es sin duda el objetivo final, proporcionar una opción, una alternativa, una buena alternativa para ser utilizada por las instituciones educativas par acceder a mejor software y más capaz en diversos aspectos.
Software que vendrá a dar un mejor servicio a las personas pertenecientes al mundo universitario, independientemente del papel que vengan a jugar en el mismo.

Además, es necesario recordar que este proyecto buscaba que la plataforma construída

permitiese involucrar más (no sólo mejor) a los miembros de los estramentos educativos. Eso ha sido perseguido y buscado en todo momento, haciendo una plataforma abierta a todos, pero basada en principios de jerarquía que permitan un escalado en la calidad de la información y servicios ofrecidos.

Bibliografía

[1] Dev Non Dos – Manual de calidad

https://www.djangoproject.com/

https://github.com/ottoyiu/django-cors-headers

http://www.django-rest-framework.org/

http://manuel.kiessling.net/2014/06/09/creating-a-useful-angularjs-project-structure-and-toolchain/

https://github.com/danialfarid/ng-file-upload

https://github.com/chieffancypants/angular-loading-bar

[2] Enlace al fichero de Postman

[3] Enlace a la clase MessageBasedException

[4] Enlace a la case ResponseFactory