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

UPMoodle

Índice

Introducción 4

[R] Descripción del problema 4

[R] Importancia del problema 5

[R] Objetivos perseguidos 7

[R] Metodología para la resolución 9

[R] Resultados obtenidos 10

[R] Estructura del presente documento 11

Contexto 12

[R] Introducción técnica 12

[R] El servidor, aspectos generales 13

[R] Pila tecnica 13
[R] Arquitectura 16
[R] Puesta en producción: despliegues 19

[C] El cliente (web), aspectos generales 21

[R] Pila tecnica 21
[R] Arquitectura 25
[R] Despliegues 27

[R] Análisis de la metodología utilizada 28

Resultados 29

[V] Aspéctos lógicos del proyecto 30

[V] Jerarquía de usuarios 30

[C] API 31

[C] Lado servidor 44

[C] Capas de la arquitectura 44
[C] Elementos auxiliares a la arquitectura. Servicio. 44
[V] Pruebas 47

[C] Lado cliente 48

[C] Módulos 48
[V] Pruebas (ausencia de ellas) 51

Discusión de los resultados 52

[V] Interpretación y justificación de los resultados obtenidos. 52

[V] Indicación de hasta qué punto se ha resuelto el problema original con cierto

tipo de métodos. 52
[V] Indicación de los métodos que no han sido efectivos a la hora de resolver el problema. 52
[V] Indicación de si los resultados están de acuerdo/desacuerdo con lo publicado anteriormente. 52
Conclusiones 53
[V] Resumen general de las conclusiones que se han obtenido del trabajo realizado, justificándolas. 53
[V] Implicaciones futuras de los resultados obtenidos. Trabajos futuros. 53
Bibliografía 54

Introducción

[R] 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.

[R] 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.

[R] 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 minimalistas 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 Djando.
- 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.

[R] 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, impartida en este mismo grado.

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

[R] 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.

[R] 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

[R] 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.

[R] El servidor, aspectos generales

[R] 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".

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.

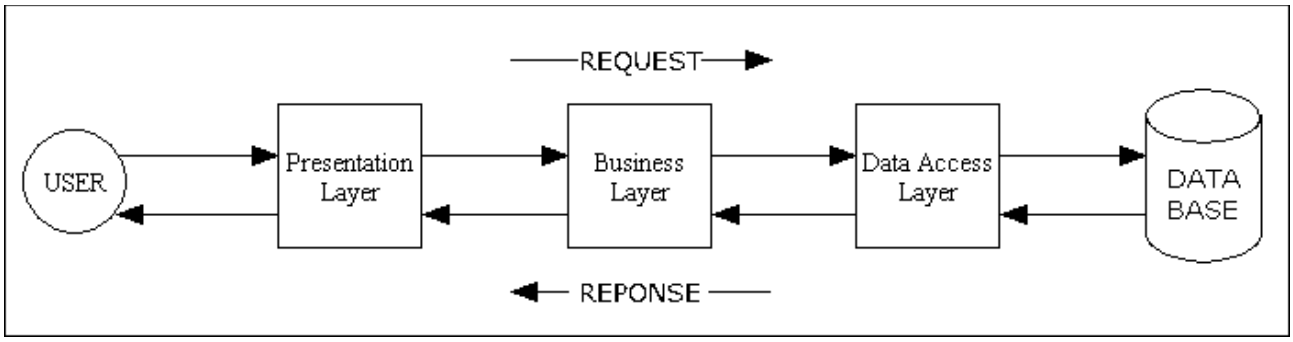
Gestores de tareas

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.

[R] 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).

[R] 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 ejecucción 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.

[R] El cliente (web), aspectos generales

[R] 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.

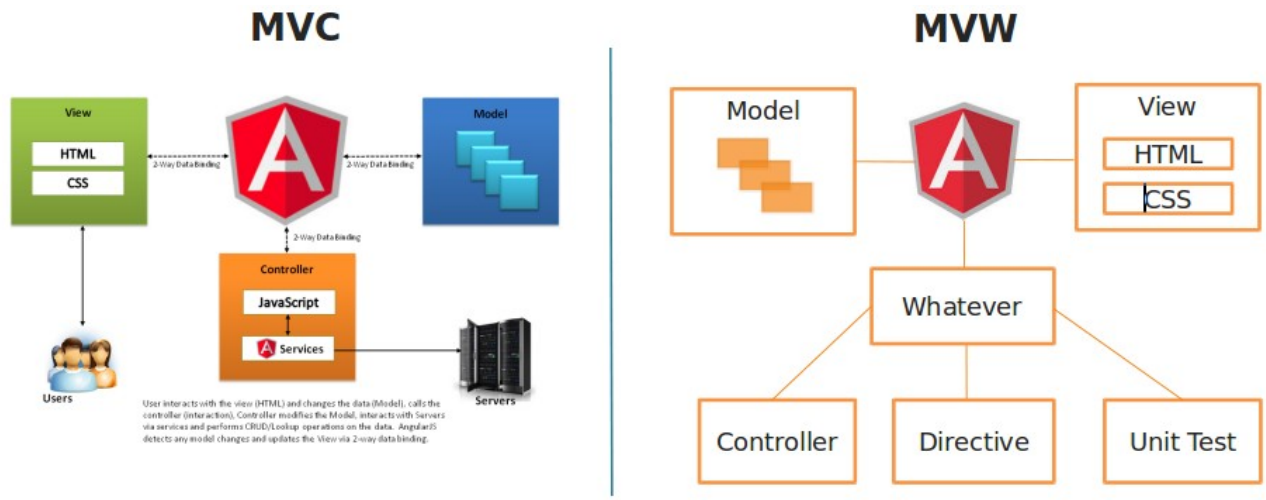
Inicialmente podríamos observar la necesidad de automatizar el proceso de compilación de los ficheros Sass como tarea mas primaria a solventar mediante automatización. Sin embargo, según crece la estructura del proyecto en su apartado Angular, se observa la necesidad de incluir tareas más complejas.

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.

[R] 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.

[R] 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.

[R] 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

Introducción

Aquí meter como va a analizarse el trabajo obtenido. Visión de vistas para hacer el puente back-front.

[V] Aspéctos lógicos del proyecto

[V] Jerarquía de usuarios

[C] API

Puente entre el cliente y el servidor

Login (/login)

Es la vista que permite ingresar nuestros credenciales de acceso a la plataforma.

front-end (loginController.js)

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).

API ([POST] login/)

Se envían dentro del form-data los datos de correo y contraseña de usuario.
De forma adicional a estos datos, se envían (y posteriormente se comprueban en el servidor) la cookie de sesión (cruasanPlancha).
El funcionamiento de las cookies de sessión ya ha sido explicado con anterioridad y como excepción (junto al logout) es el único caso en que dicha cookie es manipulada de forma activa por nuestro código.

Si el proceso es correcto, la API devolverá un código 200, en caso contrario un 400 (bad request), siguiéndose así los códigos dispuestos por el estándar para estos casos.

back-end

La llamada login afectará al modelo User en los casos en que se produzca un ingreso de credenciales correcto. Se modificarán los campos de sessionToken y lastTimeActive.
De forma necesaria, al igual que en todos los casos, el modelo utilizará códigos para devolver mensajes (bien sean de éxito o de error), viéndose en este caso una dependencia sobre los modelos Messages y ErrorMessages.

Tests.

En nuestro servidor, la clase LoginTestCase (con sus 9 métodos) está destinada a testear nuestra llamada login/ y comprobar que en el servidor no existen recovecos que permitan un acceso fraudulento a nuestro sistema.

SignUp (signup/)

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

front-end (signUpController.js)

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).

API ([POST] signUp/)

De forma análoga a la llamada de logueo, en nuestra llamada enviamos dentro del form-data los datos de usuario (correo, nombre, contraseña).

Si los datos son correctos y el regristro de ha completado correctamente en el servidor, la llamada devolverá un código 200, en caso contrario, un código 400 con un mensaje informando del problema.
Siendo una excepción dentro del sistema, esta vista no tiene explícitamente desacoplados los resultados de la llamada API pues en el caso de que el alta sido satisfactorio, la vista cambiará para mostrar un mensaje detallado que informe al usuario de los pasos a seguir.

En este caso, podría haber sido interesante haber introducido un pequeño hack en la respuesta de la API. En lugar de responder con un código 200 se podría haber devuelto un código 301 y en el campo location haber indicado una ruta (por ejemplo, la de confirmación de correo). Sin embargo, como ya se ha comentado anteriormente, con el fin de mantener los sistemas front y back desacoplados entre sí, no se asume la existencia de ningun estado o vista en el front-end y no se opta por dicha opción, aunque hubiese sido interesante de explorar desde el punto de vista técnico.

back-end

La llamada signup afectará al modelo User, creando un nuevo objeto (una nueva entrada SQL en la base de datos).
Igualmente, afectará al módulo email, en tanto que esta llamada necesita enviar un correo de confirmación a la cuenta del usuario.

Test

La clase D_SignUpTestCase es la encargada de comprobar la robusted del módulo creado, verificando que no existen grietas que permitan un registro de usuario fraudulento en nuestro sistema, verificando entre otros, que únicamente pueden realizarse registros a través de cuentas institucionales de la univiersidad deseada.

ConfirmEmail (confirmEmail/{token})

Es la vista que nos pemite confirmar la cuenta de correo de un usuario y por tanto, dentro de nuestros criterios de seguridad, al usuario registrado.

front-end (confirmEmailController.js)

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.

API ([GET] confirmEmail/)

Como se comentaba anteriormente, en el front-end se recibe una URL que contiene la clave que permite activar la cuenta del nuevo usuario.
Sin embargo, cuando se realiza la petición al servidor, dicha clave es enviada a través del form-data de la llamada POST, se hace de este modo para respetar al máximo las convenciones APIs, pues se entiende que en tanto se va a modificar información de la base de datos al realizar esta llamada (si el proceso es correcto), la cabecera utilizada en la llamada HTTP ha de ser POST y no cualquier otro.

back-end

En el lado de nuestro servidor se verá afectado el modelo User pues el campo booleano que indica si el correo ha sido confirmado pasará (en caso de un proceso correcto) de falso a cierto. (False, True respectivamente).
Más allá de estos cambios no existen grandes elementos a destacar del proceso, salvo quizás que a diferencia de muchas otras funciones que tramitan peticiones, las cuales comprueban la validez de las mismas a través de check_signed_in_request, en nuestro caso utilizamos sólo check_request_method, pues no es necesario comprobar si el usuario está logueado, en tanto que es imposible que ello suceda.

Otro detalle importante, no podemos asumir que la cookie de sesión será la misma en el momento del alta y en el momento de confirmación, si dicha suposición fuese cierta esta llamada se haría sin argumentos, pero en tanto que el usuario puede realizar el registro desde un terminal (un ordenador) y posteriormente confirmar desde otro (por ejemplo, su teléfono), no podemos dar por válido la solución sin parámetros.

RecoverPassword (recoverPassword/)

Es la vista que permite que un usuario establezca una nueva contraseña de acceso en caso de pérdida u olvido de la original.

front-end (recoverPassrwordController.js)

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.

API ([POST] recoverPassword/)

En dicha llamada únicamente es necesario enviar el correo deseado dentro de los campos del form-data.

En caso de que el proceso haya sido satisfactorio en el servidor, se recibe un código 200 y un mensaje informativo. En caso contrario, un códgo 400 y el correspondiente mensaje que explique el motivo del error.

back-end

De forma similar a lo que ocurre en el caso de la confirmación del correo, en el lado del servidor, esta llamada afecta al modelo User, modificando su contraseña en los casos correctos.
También existe una dependencia con el módulo de correo, pues de igual forma a como sucede en la confirmación de correo, necesitamos enviar un correo al usuario para proveerle una nueva contraseña.
En este caso, en el servidor se habrá de comprobar que se cumplen una serie de condiciones para poder cambiar la contraseña del usario. Esto significa que hemos de comprobar si el usuario tiene una cuenta activada y confirmada y que efectivamente, no se encuentra baneado dentro del sistema.
Igualmente, existe una dependencia con los modelos de Message & ErrorMessage para permitir una correcta información de resultado del proceso al usuario.

Tests

La clase H_RecoverPasswordTestCase comprueba a través de sus 5 métodos definidos que el proceso de recuperación de contraseña es adecuado a los requisitos definidos. Esto significa que únicamente los correos válidos y pertenecientes a usuarios activos en la plataforma pueden soliticitar la ejecución de dicho servicio.

Logout (sin vista propia)

Este método está carente de vista propia aunque es necesario descubrir la existencia de esta funcionalidad para comprender ciertos aspectos relativos al funcionamiento de las sesiones en nuestro sistema.

front-end (directives/navBar.js)

La funcionalidad que permite des-loguearse del sistema y borrar el navegador y el servidor los datos relativos a la sesión de usuario se encuentra ubicada dentro de la directiva del navbar del front-end. Esto sucede porque únicamente desde la barra superior de navegación se accede a esta funcionalidad.

De una forma similar a como sucede con funcionalidades como recuperar la contraseña o similares, nuestro "controlador" funciona a modo de pegamento entre la llamada de la factoría API y el HTML de la directiva que permite acceder al proceso de des-logueo.

API ([POST] logout/)

Esta llamada carece de argumentos dentro del form-data (conjunto de valores que van incluidos en nuestra petición HTTP).

Esto sucede porque el servidor únicamente necesita saber un dato que va implícito en la llamada: el valor de la cookie de sesión que permite al usuario logueado funcionar dentro del sistema.

Si la llamada fue correcta, el servidor responde con un código 200 y en caso contrario con un 400, en ambos casos, sin que se indiquen valores de mensaje en ninguno de los casos.

back-end

A nivel de servidor el proceso de deslogueo es relativamente sencillo. Si los datos de la petición son correctos (existe una sesión asociada a la petición, etc), la cookie de sesión será suprimida del registro de la BBDD y de las cabeceras de comunicación HTTP de nuestro servidor con el front-end (y viceversa).
El hecho de eliminar la sesión del registro del servidor se intuye con facilidad, aunque no así el motivo de suprimirlo de las cabeceras HTTP de cara a informar al front-end.
El motivo de ello es que de esta forma, si en lo sucesivo algun otro usuario desea loguearse en el sistema, recuperar la contraseña o cualquier otra acción, una nueva cookie de sesión será definida y enviada a través de las cabeceras HTTP, reiniciándose por completo el proceso. De cualquier otra forma, el proceso de des-logueado dejaría trazas en los sistemas que en ciertas ocasiones podrían suponer conflictos y comportamientos no deseados.

Tests

En el caso del proceso de deslogueo (clase test_1_logout_basic) únicamente se comprueba que el proceso es adecuado en su versión mas estándar, debido a las limitaciones y complejidad del sistema para testear escenarios (de integración) más complejos que buscasen casos límites.

Subjects (subjects/)

Es la vista que nos permite navegar a través de los distintos niveles de organización de una universidad hasta llegar a sus asignaturas.

front-end (subjectsController.js)

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.

API ([GET] subjectsTree/)

Como se ha explicado anteriormente, esta llamada devuelve los distintos niveles registrados en el sistema de forma anidada.
La llamada carece de argumentos de formulario. Si el proceso fue exitoso, devolverá un código 200 junto a la información deseada, en caso contrario, un código 400.

back-end

En nuestro sistema, la información de los distintos niveles de educación se almacenan de forma plana y no explícitamente relacional (salvo por el campo parent, que indica el nivel superior).
Por tanto, la concepción de una relación anidada es una construcción propia que simplifica la forma de relacionar la información entre sí y que intenta acercarse a la organización jerarquizada de una universidad real (en tanto que en el fondo estamos modelando en cierta forma la realidad).

El modelo Level es el que se ve “afectado” por esta llamada, que en realidad al ser de tipo GET no modifica en forma alguna la información ya presente en el sistema.

Test

//TODO.

Asignatura (Subject/{id})

Es la vista concreta de una asignatura, en ella se muestran los ficheros pertenecientes a la misma.
De forma extendida, se muestran a través de ventanas modales los diálogos que controlan la subida de nuevos ficheros a la plataforma, el visionado y edición de los mismos.

front-end (subjectController.js)

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.

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.

En el segundo controlador auxiliar (ModalEditFileInfo), tenemos la ventana modal que 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.

API ([GET] files/subject/{id}, [POST] file/f/, [GET][POST][DELETE] file , [GET] file/f/{id})

Veamos cada caso por separado.

[GET] files/subject/{id}. Obtiene el listado de ficheros pertenecientes a la asignatura indicada en la variable id.

[POST] file/. Es la llamada encargada de enviar nuevos ficheros al servidor. Junto al binario, ha de enviarse en el formulario la información relativa a la asignatura a la que pertenece el fichero, el autor del mismo, el nombre del fichero, la descripción de este y el tipo de fichero que es (apunte, examen, etc).
Esta llamada es una de las pocas que utiliza una cabecera form-data pura en lugar de la x-www-form-urlencoded utilizada en el resto de casos.

[GET][POST][DELETE] file/{id}. Son el conjunto de llamadas encargadas de gestionar un

fichero concreto. Siguiendo las reglas de nomenclatura definidas anteriormente, la cabecera GET devuelve la información del fichero en cuestión, la llamada POST modificará dicha información (si los valores y campos son correctos) y DELETE suprimirá del sistema el fichero en cuestión.

[GET] file/f/{id}. Es la llamada que devuelve el fichero en sí. La información y el fichero se encuentran en llamadas separadas ante la complejidad que reportaba mantener toda la información junta y la ineficiencia que supone enviar ficheros de tamaño indeterminado junto al resto de información asociada (de poco peso).

back-end.

A pesar de elevado número de llamadas API asociadas a esta vista, en nuestro back-end, el número de modelos afectados es relativamente reducido.

El modelo más afectado será File, el cual se encarga de gestionar toda la información relativa a los ficheros, tanto su información como los binarios. Es posteriormente a través del enrutador y los serializadores donde se crea la diferenciación entre el binario y su meta-información.
Además del modelo File, también existe uso y dependencia del modelo User (para comprobar los niveles de autorización a la hora de realizar ciertas acciones tales como [POST] file/{id}).

Test

//TODO.

EditSubjects (editSubjects/)

Es la vista que nos permite modificar la relación entre el usuario logueado y las asignaturas que sigue.

front-end (editSubjectsController.js)

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/.

API ([POST] user/subjects/)

Siguiendo los principios de composición de nuestra API, la llamada que modifica la información de las asignaturas seguidas por un usuario se define como una llamada “compuesta” dentro de las llamadas de usuario.
Como se explicaba en apartados anteriores, esto no obecede a unos motivos estructurales obligatorios, es decir, de cara a la API no es necesario definir la llamada de forma compuesta/anidada para que funcione el sistema, únicamente se hace por cuestiones conceptuales y de legibilidad.

En el lado técnico, tenemos que en el form-data de nuestra llamada enviamos en forma de lista las ids de todas las asignatuas a las que el usuario se asocia.
Si el proceso fue correcto, se devuelve un código 200 y en caso contrario un 400, enviándose en ambos casos sendos mensajes informativos que habrán de ser manejados en el front-end.

back-end.

En el lado del servidor, la llamada que modifica la lista de asignaturas adscritas afecta únicamente al modelo User.
A nivel ténico únicamente hay que transforma la lista en formato plano enviada por la API a un array de Python que contenga las ids en formato numérico para así poder crear la relación de usuario-asignaturas dentro del métido update_subjects del modelo User.

Tests

En la clase K_editSubjects comprobamos la resistencia del código elaborado a la hora de gestionar casos básicos (respuesta exitosa esperada), así como casos adversos donde no se transmiten ids en absoluto o los ids aportados no corresponden a asignaturas y por tanto no pueden ser registrados en el sistema.

Profile (profile/)

Es la vista que muestra (y edita) la información del usuario logueado en el sistema.

front-end (profileController.js)

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.

API ([GET][POST] user/, [POST] user/profile)

Veamos las llamadas por separado.

[GET][POST] user/. Estas dos llamadas son las que proporcionan la información del usuario y la que permite la modificación de los datos. En el caso de la llamada GET no existen valores o argumentos de llamada, pues como ya se ha explicado anteriormente, la única variable necesaria en este caso se transmite de forma implícita en la llamada a través de la cookie de sessión.
En el caso POST, se transmiten la totalidad de los campos, independientemente de si han sido modificados o no.

[POST] user/profile. Es una llamada similar a la vista anteriormente en [POST] file/ con una diferencia, no se transmiten valores adicionales en el formulario enviado.
Al igual que en la otra llamada análoga mencionada, en esta llamada el fichero es enviado al fichero a través de una llamada con un form-data puro.

back-end.

En el lado del servidor encontramos una profunda y esperada dependencia con el modelo User.
En el lado del manejo de un fichero cuando desea actualizarse la imagen de perfil, el proceso en sí es relativamente sencillo en tanto que Django empaqueta en el objeto request la información que deseamos de una forma sencilla. Es por esto que es sencillo acceder al fichero enviado a través de la API y asignarlo a nuestro usuario.

Test

En la clase I_userTestCase con sus 13 métodos de prueba encontramos el banco de pruebas que testean la robusted del diseño planteado y la imposibilidad de modificar los datos de usuario de forma equívoca o fraudilenta.

Notes (notes/)

Es la vista que gestiona de forma general todas las cuestiones relativas a las notas (noticias, eventos) registrados en el sistema.

front-end (notesController.js)

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).

Luego, entre los controladores auxilares encontramos primeramente el de 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.
El primero de ellos sería el controlador que gestiona la creación de noticias, realizando la labor de enlace con la llamada API debida ([POST] note/).
En segundo lugar tendríamos el controlador que gestiona la visualización de una nota/noticia así como su edición.
En el momento actual no es necesario recurrir a la llamda específica [GET] note/{id} para obtener toda la información de la misma, pues esta ya viene dada por la llamada [GET] note/level/{id}.
Sin embargo, si que realiza uso de la llamada [POST] note/{id} en tanto que cuando en la vista se activa el modo edición y si se disponen de los permisos adecuados, el usuario logueado en el sistema podrá modificar los datos relativos a la nota seleccionada.

API ([GET] note/level/{id}, [POST] note/, [POST][DELETE] note/{id})

Veamos las llamadas por partes.

[GET] note/level/{id}. De forma análoga a otras llamadas de similar sintaxis, carece de mayores argumentos de llamada y retornará un código 200 con las notas del nivel deseado si la llamada fue correcta o un 400 y un mensaje de error si la llamada no era adecuada.

[POST] note/. Es la llamada que permite dar de alta en el servidor nuevas noticias. De forma similar a la llamada [POST] note/{id} y como ocurría con el caso de la modificación de datos de usuario, se envian todos los campos pertenecientes a la nota, hayan sido estos modificados o no.

[POST][DELETE] note/{id}. La primera cabecera envía los datos de la nota al servidor, hayan cambiado estos o no y el servidor se encarga de reflejar los datos actualizados en el sistema si la llamada es realizada correctamente. En el caso de la cabecera DELETE, de forma similar a como sucede en el caso GET (no recogido en esta sección) no se requieren de argumentos de petición adicionales pues por ejemplo, un dato que proporciona el nivel

de autorización para realizar la petición es enviado de forma implícita con la llamada HTTP. Nos referimos, al igual que en ocasiones anteriores, a la cookie de sesión.

back-end

A pesar de las múltiples llamadas analizadas en el apartado anterior, el rango de afección de las mismas al sistema es limitado. De forma similar a como ocurre en el caso de los ficheros, existe una fuerte dependencia con el modelo Note y una relación de uso y acceso más debil con otros modelos como Messages, ErrorMessages, User (para comprobar permisos de acceso), etc.

Test

En la clase J_noteTestCase y sus 18 métodos de test de integración comprobamos la robusted del módulo de gestión de noticias de la universidad y de como no existen fallas de seguridad o acceso a los datos que puedan derivar en comportamientos erráticos del sistema.

Calendar
// TODO:

[C] Lado servidor

En el presente proyecto existen una serie de funcionalidades principales que pueden ser fácilmente identificadas como las distintas vistas del proyecto web.
Sin embargo, de forma transversal a dichas vistas y/o funcionalidades principales, encontramos una serie de módulos que bien por interés técnico o por cuestiones de arquitectura, es necesario detallar de forma separada.

[C] Capas de la arquitectura

[C] Elementos auxiliares a la arquitectura. Servicio.

Cookies

Nuestro servidor necesita de forma constante autenticar al usuario que intenta acceder a la información o incluso modificar la misma.
Para no tener que estar enviando constantemente los credenciales de acceso (usuario y contraseña) junto al resto de datos de las peticiones, utilizamos cookies de sessión para la autenticación.

El funcionamiento de las cookies de sessión como medio de autenticación es proporcionado por Django aunque en este caso, para obtener una mayor flexibilidad en el manejo de las sesiones, se ha optado por una implementación propia.

El funcionamiento es muy sencillo. Antes de que el usuario se loguee en el sistema, nuestra cookie se establece en el navegador y en el momento del logueo es enviada al servidor junto a los credenciales de acceso.
Si el logueo es satisfactorio, en nuestro servidor dicha cookie se asocia a nuestro usuario como la cookie que le identifica.
En sucesivas llamadas al servidor y de una forma automatizada (dado que es un uso estándar de cookies) la cookie de sessión que nos identifica es enviada dentro de los campos cabecera de nuestra llamada HTTP. Posteriormente en el servidor, dentro de las funciones concretas de cada caso, se comprueba la identidad del solicitante de la información a través de la cookie y se permite el acceso o no en función del resultado de la evaluación.

Email.

Algunas de las funcionalidades principales anteriormente mencionadas, tales como el registro o la recuperación de contraseñas requieren del uso de comunicación mediante correo.
Para llevar a cabo dichas comunicaciones es necesario hacer uso de una librería de python capaz de realizar envíos de correos electrónicos.

Serializadores/Deserializar.

Como ya se ha explicado en el apartado dedicado a la pila tecnológica, en nuestro proyecto, como parte de una serie de componentes que se encargan de proveer un camino entre los datos SQL de nuestra base de datos y las cadenas de texto JSON emitidas por la API, existe un módulo de vital importancia conocida como (des)serizalizadores. Estos módulos, que por comportamiento pueden considerarse uno solo, son los encargados de proveer una interfaz de interpretación y traducción entre un objeto diccionario JSON de python 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 DjandoRest provee opciones de deserialización (unserialization), por cuestiones de composición, complejidad de algunos casos 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 DjangoRest, 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.

Modelos.
Como se apuntó brevemente en el apartado dedicado al stack tecnológico, buena parte del manejo de datos en nuestro proyecto recae en la librería models.Model proporcionada por Django. models como referencia al paquete y Model como referencia de la clase.
Esta clase nos provee una serie de características, la más reseñable de todas ellas, la posibilidad de que nuestros objetos tengan persistencia en discos a través de una base de datos SQL. Es decir, la mentada clase Model no es sino la interfaz pública que Django provee de su ORM, siendo los métodos de dicha clase los que gestionar el necesario mapeo de los objetos.
Tal y como se indica en la documentación ofrecida por Djando, la clase Model ofrece una serie de métodos (los cuales pueden ser reescritos para ofrecer características extras a las originales) para la comprobación de valores, el guardado de datos en disco, actualización, etc.

**Messages & ErrorMessages. **

En el presente proyecto la información transmitida al usuario es muy importante. Ante ciertos eventos, la información que transmitidos en forma de errores o mensajes positivos recaen como responsabilidad única del servidor. Esto significa que es el servidor quien

posee y maneja en solitario todos los errores y mensajes que puedan necesitarse a lo largo del uso de la aplicación. El motivo de este diseño responde a la voluntad de mantener una integridad y control sobre la información mostrada lo más amplio posible. En los casos en que la información se encuentra desacoplada y distribuida entre los dos extremos del sistema, su mantenimiento y actualización puede resultar tedioso sino caótico.

En ciertos casos, dependiendo del nivel de seguridad y en función de la información expuesta a través de los mensajes y errores, es posible e incluso aconsejable el uso de una llamada API que devuelva todos los mensajes que pueda producir el back-end, de esta forma, pueden retornarse códigos de mensaje/error en el lugar de mensajes en sí. El objetivo final de este diseño no responde a una necesidad de ahorro de datos (que a nivel teórico sí podría producirse), si no que podría utilizarse para proveer al front-end de un listado de errores y mensajes que pudiera utilizar a discrección, incluso en casos en lo que no existiese comunicación con el servidor. Ejemplo, los mensajes de control de formato de un formulario. De esta forma los mensajes de error ya están disponibles para el cliente web, pero la integridad del sistema permanece.

==PUENTE==
API (entendido como el puente entre el back-end y el front-end).
Para la construcción de esta API se ha utilizado como referencia princial (con ciertas excepciones) una guía escrita por un antiguo alumnos de la escuela: Anthanh Pham.
En dicha guía se describen los aspectos más importantes a la hora de diseñar una API Rest. Entre dichos conceptos priman algunos sobre el resto a la hora de enfocarlo a nuestro caso particular. En concreto podría destacarse el esfuerzo por construir una API mínima, altamente legible, con una alta relación con las vistas y modelos e intentando reutilizar conceptos que varían en el tipo de llamada HTTP utilizado (GET, POST, etc).

Una de las pocas violaciones de la guía sería el consejo de no habilitar y utilizar llamadas tipo DELETE. Dicho consejo fue depreciado en pro de una mayor legilibilidad. Sin embargo, en caso de ser necesario en un futuro donde el proyecto fuese a ser usado en producción, la solución pasaría por añadir un flag/campo a las llamadas post equivalentes (todas lo tienen) e incluir un &method=DELETE.
Sin embargo, como se mencionaba anteriormente se ha preferido dotar de legibilibilidad a la API en tanto que se considera que el presente TFG es una vía de investigación y exploración que en términos generales, para adquirir el rango de código apto para despliegue en producción, necesita ligeros cambios.

Como se ha incidido en varias ocasiones en el tema de la legilibidad de la API, observemos un pequeño ejemplo.
En el caso de las llamadas de noticias (note/), tenemos los siguientes casos:
[POST] note/. Dar de alta una nueva nota.
[GET]|[POST]|[DELETE] note/{id}. Obtiene, modifica o elimina una nota en función de su identificación única.
[GET] note/level/{id}. Obtiene las notas de un nivel concreto (carrera, curso, asignatura).

Con estas 5 llamadas se gestionan la totalidad de los aspectos relacionados con las noticias de nuestro sistema. Las llamadas son sencillas de comprender y con una extensión adecuada.
Retomando la cuestión anterior de la legibilidad versus una API rígida en cuestiones de cabeceras, tendríamos que en el caso de [POST]|[DELETE] note/{id} quedaría fusionado en una sola llamada y que habría que recurrir a su estructura interna para comprender su funcionamiento.

Otro elemento que se ha intentado respetar aunque apenas si tiene peso en el proyecto: la composición. En el caso de que por ejemplo de la foto de perfil de nuestros usuarios tiene una llamada separada del [POST] user/, esta es, [POST] user/profile. Este sería un caso típico de composición/anidación de llamadas que en el capítulo correspondiente será analizado en detalle.

En cualquier caso, en las secciones posteriores (vistas) se tratarán a fondo cada una de estas llamadas, aunque es necesario explicar que la guía de estilo seguida en la cuasitotalidad de los casos es la explicada anteriormente.

[V] Pruebas

[C] Lado cliente

En el apartado front-end de nuestro proyecto encontramos una serie de módulos, que apoyándose módulos pre-existentes en los frameworks utilizados, son dotados de ciertas funcionalidades específicas.

[C] Módulos

App
Es el modulo principal del proyecto en su apartado front-end.
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.

API
Para ofrecer el paquete de llamadas API que permiten la comunicación con el servidor tenemos el módulo API. Dicho módulo es una implementación de una factoría angular.
Como ya se ha descrito anteriormente, el presente 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.
Nuestra factoría API simplemente sirve de enlace entre el puente API real y nuestro front.
Es por tanto que no es en este punto donde decidimos la arquitectura de la misma, ni los métodos expuestos, simplemente seguimos e implementamos lo que la API como tal expone declara públicamente que ofrece.

User
De igual forma que sucede con el módulo API, el caso del módulo User es una implementación de la inferfaz factoría que provee Angular.
En este caso se trata más del modelo de datos del usuario logueado que de las funciones que expone.
Es decir, si bien la factoría usuario posee una serie de funciones que permiten la manipulación de sus datos, lo reseñable de dicho módulo es la presencia de una variable tipo .model que posee los datos del usuario logueado, los cuales son retornados a través de una llamada API específica.
Lo interesante y potente de esta factoría es el hecho de que al ser un Singleton, los datos

del usuario se encuentran permanentemente actualizados de forma y manera que se garantiza la integridad de los datos a lo largo del front-end.

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.

NavBar

En el presente proyecto encontramos que en el apartado front-end 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 front-end 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 necesario 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.

Upload
https://github.com/danialfarid/ng-file-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
https://github.com/chieffancypants/angular-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.

Bootstrap
https://angular-ui.github.io/bootstrap/
Bootstrap es un framework web que incluye tanto aspectos visuales (css) como elementos dinámicos (javascript). Inicialmente es utilizado a lo largo del proyecto como esqueleto para hacer crecer las vistas html de una forma rápida y estable, sin embargo, según ha avanzado el desarrollo del front-end se ha visto en la necesidad de acudir a explotar su carácter más dinámico y utilizar las librerías js utilizadas (incluídas en la carpeta de librería de terceros).
Un motivo de esta pivotación sobre su uso puede encontrarse por ejemplo en el caso de las ventanas modales. Las ventanas modales son complejas de implementar si se desea obtener una UX pulida, sin embargo, acudiendo a la librería angular-bootstrap (su nombre lo dice todo) encontramos una implementación completa y cuidada de este patrón visual.
Como se explicaba anteriormente, bootstrap en su apartado más visual proporciona una serie de reglas y clases css muy sencillas pero que permiten construir vistas que se adaptan a las circunstancias de renderizado (adpatative o responsive).

[V] Pruebas (ausencia de ellas)

Discusión de los resultados

[V] Interpretación y justificación de los resultados obtenidos.

[V] Indicación de hasta qué punto se ha resuelto el problema original con cierto tipo de métodos.

[V] Indicación de los métodos que no han sido efectivos a la hora de resolver el problema.

[V] Indicación de si los resultados están de acuerdo/desacuerdo con lo publicado anteriormente.

Conclusiones

[V] Resumen general de las conclusiones que se han obtenido del trabajo realizado, justificándolas.

[V] Implicaciones futuras de los resultados obtenidos. Trabajos futuros.

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/