UPMoodle
Una vuelta de tuerca a las plataformas de e-ducación.

Víctor Pérez Rey

Agradecimientos

Índice

Intro no técnica

El mundo actual ha cambiado irremediablemente con la irrupción de internet en nuestras vidas y la educación (en todos sus niveles) no es una excepción.
En la actualidad, si bien hay numerosas herramientas orientadas a integrar la educación universitaria en las posibilidades tecnológicas actuales, estas no logran alcanzar las expectativas y necesidades de profesores y alumnos.

Esta perspectiva, fruto de la propia experiencia personal en mis pasados años como estudiante son el punto de partida del presente proyecto.
Es una idea, una opinión sencilla: en las aulas docentes estamos perdiendo se está perdiendo una gran oportunidad al no proveer a estudiantes y profesores las herramientas digitales adecuadas y algunas de ellas son realmente sencillas de implementar.

Sin necesidad poner el foco en una materia o situación concreta, siempre hay a unas necesidades comunes hacia el cuerpo estudiantil que se dan de forma repetitiva.

- Se necesita material de trabajo. Enfóquese en corto y llámense apuntes, trabajos, bibliografía, ejemplos, exámenes. En cualquiera de los casos, dicho material de trabajo es una pieza vital para desarrollar tareas de estudio y trabajo que permitan transmitir conocimientos a los alumnos o a estos adquirirlos.

- Es necesario un canal comunicativo que permita transmitir noticias de distintas formas (general o focal). Es decir, necesitamos conocer las noticias de la institución. De las más generales a las más concretas. Las generales, las grandes noticias que sacuden la vida del campus (huelgas, cortes de luz, etc) hasta las focales, como puede ser la suspensión de una clase por una baja médica.
  En realidad no importa el mensaje o el público objetivo, lo esencial es que la necesidad existe y por tanto es visible la necesidad de satisfaccer la carencia con distintas herramientas.

- El cuerpo estudiantil necesitan organizar sus horarios. Hablamos tanto de sus tediosos horarios de clases que se habrán de repetir por todo un semestre, pero también de los eventos que habrán de ocurrir sólo una vez, o incluso con frecuencia irregular.
  Igual que sucede con las noticias, en el caso de los horarios/eventos, existen distintos tipos de situaciones, público objetivo, etc. Sin embargo, vuelve a suceder que la relación existencia-necesidad-herramientas está presente y que por tanto es necesaria la búsqueda de soluciones apropiadas.

Los arriba enunciados son 3 pilares sencillos, transversales a carreras y universidades.
Sin embargo, la motivación que lleva a la puesta en marcha de este TFG es el vergonzante y plausible hecho de que en nuestra escuela, así como en muchas otras universidades españolas, las herramientas aportadas por las escuelas a los estudiantes para satisfacer las necesidades anteriormente descritas no existen, no funcionan por estar mal planteadas o no cubren las necesidades de sus usuarios.

Análisis de la situación actual con respecto a los elementos a solventar.

1. Apuntes.

Desfasados, tardíos, inexistentes, incorrectos o insuficientes. Estos son los adjetivos que en muchos casos pueden atribuirse al material subido a las plataformas de e-educación universitaria.
No es en muchos casos simple culpa de los profesores, en muchas ocasiones el mayor motivo de fracaso en los planteamientos de estas plataformas reside en el hecho de que el profesor es el único que es capaz de aportar 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 colaboraban, en la elaboración de los mismos.
No hay mayor premisa que esa (la colaboración) tras la idea de una plataforma de material de trabajo abierta a modificaciones por parte de cualquier usuario, independientemente de su rol dentro del aula.
El 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.
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 de juntar 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 y eso choca terriblemente con una realidad cotidiana de nuestra vida en el 2015.
Hoy en día 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, fin.
Frente a esta realidad tangible de nuestro día a día, la experiencia concreta de la universidad: webs de departamento, de escuela, de profesores, el correo, las plataformas de e-educación, etc.
Simplemente no funciona. Por disperso, por inconexo, por desactualizado, por falta de acuse de recibo.
Por tanto, el enfoque, el camino que guía la solución a implementar ha de ir por tanto orientada a acoplarse con los patrones de diseño y usabilidad actuales, es decir, enfocar la solución a la existencia de paneles de notificaciones en los que se centralice la información.

3. Horarios

De forma paralela y casi indistinguible de los eventos surgen los horarios.
En el caso concreto de la UPM & la ETSISI, los horarios relativos a la actividad docente no se

encuentran centralizados.
Primeramente porque existen horarios de clase, de exámenes, de mantenimientos, de tutorías, etc.
Seguidamente porque estos vuelven a encontrarse dispersos a lo largo del moodle, webs de departamento, sitios personales, correos enviados a inicio de curso, etc.
Y esto en el caso de que la información realmente se encuentre disponible, lo cual en muchas ocasiones no se cumple.
Ante esta problemática, partiendo nuevamente de la teoría de notificaciones anteriormente escrita surge el diseño de aquello que pretende ser la solución.
Unificar en un solo espacio (al igual que con los eventos) todos aquello que tiene una relación temporal concreta. No hablamos únicamente de los horarios de las asignaturas, también de sus horarios de tutorías, charlas que se den en las escuelas y en definitiva cualquier cosa que como miembros de la universidad deseamos estar conocer con facilidad.

Conclusiones

Hay que ser conscientes del reto que se está poniendo encima de la mesa. En el mundo del desarrollo existe una cierta tendencia en bramar sobre la carencia que presentan algunos estándares y en el como hay demasiados que intentar dar una solución única. El resultado de estas discusiones en comunidad suelen dar por resultado que donde antes existían 13 estándares incompletos, ahora existan 14.
Es por tanto importante comprender que si se quiere señalar acusatoriamente a diversos niveles de la institución universitaria, la solución aportada debe verdaderamente solucionarlos con una calidad a la altura de la acusación, quedando este juicio de valor a discrección del jurado que habrá de evaluar el presente TFG.
Existirán más oportunidades en el futuro para presentar una solución que verdaderamente satisfaga a los miembros de la universidad, la presente solución aportada no es la última oportunidad para ello. Sin embargo, es mi objetivo prestar un alto valor en la solución aportada para que las nuevas promociones que ingresen en las aulas así como los profesores que habrán de continuar en ellas cuando nosotros ya no estemos, finalmente posean una herramienta a la altura de las necesidades educativas de estos tiempos digitales.

Intro técnica

El presente TFG es un proyecto de tipo full-stack, es decir, presenta elementos tanto de servidor (back-end) como de cliente final (front-end).
Partiendo de esta división principal, hay que introducir un tercer elemento, que si bien pertenece al back-end, en cierta forma puede ser entendido como el puente que une ambos mundos. Nos referimos a la API/Rest o en términos coloquiales, la API.

El API es nuestro contrato, el escaparate del back-end, el puente que permite la comunicación entre ambos entornos. Como se decía anteriormente, si bien la API es de facto un elemento del back-end, para comprender con profundidad cual es la arquitectura del proyecto es importante presentarlo como un elemento de unión entre entornos más que un módulo sobreexpuesto de uno de ellos.

Por tanto, partiendo de estos 3 elementos principales (de los que posteriormente se presentarán sus distintas capas y módulos) podemos empezar a definir las bases ténicas del proyecto.

En términos generales, en tanto que en secciones posteriores se hablará en profundidad de todos estos elementos, en cada entorno encontraremos 3 capas de concepto diferente: vistas/modelos, módulos y el stack técnico.

Vistas

Nuestras vistas o modelos van a ser uno de los principios fundamentales de esta memoria. Son el concepto final que permite aglutinar el resto de principios anteriormente descritos, en tanto es el concepto que se sitúa en lo alto de la lasaña de capas que componen el sistema.
El término vistas hace referencia en exclusiva al entorno front-end. Será cada una de las rutas principales de la web que posee el proyecto. Es por tanto que existirán vistas tales como asignatura, notas, calendario, registro, login, etc.
Cada una de estas vistas (atendiendo a distintas excepciones) tienen una correspondencia con un modelo del back-end.
Antes de continuar hay que definir brevemente el concepto modelo. Un modelo en nuestro back-end es una clase (dentro del paradigma OO) que representa un objeto de nuestro sistema.
Es por tanto, que en nuestras vistas se sirve/presentan datos que tienen una correspondencia con un modelo definido en el back-end, cada uno de estos modelos tiene unos atributos y unas propiedades (atendiendo a teoría basica de OO).
Como decimos, existe una correlación entre cada una de las vistas que habremos de tener en nuestro frontal web y los objetos que serán manipulados en el back-end. Y como se comentaba al inicio de la sección, es la api del sistema la que permite que se tienda un puente entre ambos mundos, pues son nuestros modelos del servidor los que son enviados al frontal para ser representados dentro de las vistas. De igual forma, en los casos en que la información pueda ser modificada, la información viajará de vuelta de las vistas al servidor para que se produzca las pertinentes actualizaciones.
Este es uno de los puntos principales de la teoría full-stack que se presenta en este proyecto. El cómo existe una fuerte relacíón entre ambos entornos, donde el concepto que sirve de referencia son las vistas y los modelos y donde el puente comunicativo es la apiRest expuesta.
Igualmente, es la sencillez y potencia de este principio el que permite la creación de un

código que respete los principos de arquitectura, limpieza, sencillez y claridad que se busca a la hora de programar.
Es el respeto a este punto de partida el que permite entender cómo puede conseguirse que el código permanezca limpio y organizado a la hora de la construcción del sistema. Es el comprender que no hay que dividir conceptos, ni adaptar los objetos utilizados en cada entorno para permitir la comunicación por la API lo que hace que el código y su estructuración permanezca clara y concisa a lo largo del tiempo y que su testeabilidad sea sencilla.

Más allá de aquí, lo que tendremos son elementos concretos que ya pertenecen a cada uno de los entornos. Principalemente, hablamos de la siguiente capa del sistema que serían los módulos y posteriorment el stack técnico que lo soporta todo.

Módulos

En ambos casos el nombre y el concepto es el mismo. Son los distintos componentes que a través de diversas reglas se relacionan para dar forma a la arquitectura general de cada entorno, que en muchos casos pueden comprenderse como funcionalidades adicionales o laterales al núcleo del sistema y que en otros casos habrán de entenderse como elementos abstractos del mismo.
En las secciones sucesivas, se definirán y explicarán con mayor profundidad cuales son estos módulos, su relacion con otros, su importancia y necesidad.
De forma extra, será en este punto donde se explique brevemente las característica de nuestra API y la teoría que la sustenta.

Stack técnico
A la hora de construir este proyecto ha sido necesario recurrir a soluciones técnicas ya existentes. Hablamos tanto de las bases técnicas más básicas como son los lenguajes utilizados (y el porqué de su uso) así como todo el ecosistema de frameworks, librerías que se utilizan sobre dichos lenguajes.
Igualmente, dentro del stack técnico se incluyen aquellas cuestiones más meta del proyecto. Los paradigmas de programación utilizados, los patrones de diseño implementados, etc, etc.
El stack técnico tiene una importancia muy elevada pues son los cimientos del resto del sistema. La elección de un stack u otro supone la elección del marco o el corsé que contendrá nuestros módulos y vistas y que la elección implicará grandes ventajas y sacrifios en el momento del desarrollo.

Stack técnico

Como se definía brevemente en la introducción técnica de la memoria, nuestro stack técnico es en pocas palabras la pila técnica del proyecto, es decir, lenguajes, herramientas, frameworks y librerías utilizadas a lo largo del proyecto del proyecto son múltiples y variadas.

###back-end

En nuestro back-end el lenguaje principal es Python. Este lenguaje se distingue por ser multiparadigma (se adapta tanto a una estructura tipo script como a OO), por su sencillez y verstailidad y por su potencia.

Por encima de Python se utiliza Django, un framework orientado a facilitar el desarrollo de proyectos web. Si bien puede utilizarse para proyectos full-stack (back-end + front-end), en nuestro caso particular solo se utiliza para la parte de servidor.
Este framework nos provee de múltiples y potentes herramientas. En nuestro caso, optamos por utilizar únicamente una pequeña parte de toda ellas, en concreto los módulos que refieren al ORM (mapeador de objectos relacionales), el enrutador de urls, etc.

Para facilitar la tarea de enrutación y des/serialización de objetos (de base de datos), por encima de Django se encuentra DjandoRestFramework. Este framework, dependiente de Django para su uso, proporciona una capa muy útil a la hora de transformar los datos de entrada de nuestras peticiones HTTP en objetos que nuestra base de datos pueda comprender (y viceversa).
El motivo principal de la elección de este diseño es que Django está pensado para ser el framework principal de todo el sistema, pues además del módulo de modelos también ofrece módulos para tratar vistas, plantillas, formularios. Esto significa que Djando puede encargarse en solitario de servir vistas donde el modleo de datos está integrado en la propia vista y no es necesario nada más para controlar los datos.
Sin embargo, atendiendo al diseño arquitectónico dispuesto, donde las vistas son delegadas al front-end y el canal de comunicación es nuestra ApiRest, es necesario utilizar DjangoRestFramework y como posteriormente se explica, mejorar algunas de las funcionalidades proporcionadas por dicho framework.

Persistencia de datos

En el presente proyecto la persistencia de los datos recae en una base de datos (motor sqlite) pero con una serie de particularidades. A nivel real la base de datos es sql, pero sin embargo la capa más alta de acceso a nivel de framework es una base de datos de tipo objeto. Esto significa que es el framework de django quien se encarga de realizar el mapeo orm para transformar las instancias de nuestros modelos (objetos) a datos sql con persistencia en disco.
El ORM proporcionado por Djando se podría resumir en que proporciona un objeto base Model de la librería models, todo objeto que en nuestro proyecto haga herencia de dicha clase será susceptible se tener capacidades de mapeo a datos SQL.
En el apartado dedicado a los des/serializadores se analiza el módulo encargado de convertir nuestras instancias heredada de Model a texto con formato JSON, preparado de esta forma para que nuestros clientes finales consuman los datos.
En términos generales puede establecerse el siguiente hilo conductor extremo a extremo

de un dato tipo.
Datos SQL <=Mapeador ORM=> Objeto de una clase heredera de models.Model <=Des/Serializador=> Diccionarios de Python (clave-valor) <=JSONRender=> Cadena de texto con formato JSON que representa pares de clave-valor.
Como se explicaba en la introducción nuestros modelos son una pieza fundamental del entorno back-end. Y es aquí, en el principio técnico que soporta la persistencia de datos donde se ve la relación última entre un objeto de tipo modelo y la pila técnica que lo sustenta (pasando por el elemento intermedio, el módulo, en nuestro casos el conjunto de clases Model que son proporcionadas por el framework Django).

Requests.

Request es una librería tipo navaja-suiza (swiss-knife) de Python. Es la librería tipo de PYthon para la comunicación HTTP.
Esta provee un acceso completo a la totalidad de los campos/cabeceras de una petición HTTP y de igual permite una completa personalización de las mismas.
De esta forma, nos es posible gestionar de forma propia las cookies/tokens de sesión, los tipos de petición HTTP que realizamos, el contenido de las respuesta, el formato de los datos que se envían en las respuestas y en definitiva, obtener un control total sobre la comunicación HTTP y llevarlo a nuestras necesidades.

De forma manual y complementaria a nuestras peticiones HTTP esambladas a través de librerías dependientes de HTTPResponse, tenemos que configurar algunos valores de cabecera de nuestras peticiones para permitir la comunicación entre servidores. Pero estas cuestiones son tratadas en el apartado dedicado a CORS.

HTTP 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".
Django proporciona una capa de distintos middelware (código embebido entre distintas capas y procesos del sistema) para permitir una comunicación cross-domain completa.
De esta forma, mediante la activación de una serie de middelwares ubicados en el settings.py de nuestro back-end, podremos definir las reglas que nuestra api expone en las cabeceras de las respuestas HTTP.
Por ejemplo, algunas de estas reglas nos permiten activar las peticiones de tipo DELETE, que por defecto no son accesibles fuera del dominio del back-end y que tan necesarias y que en el apartado dedicado a la teoría y práctica de nuestra ApiRest se encuentra explicado la importancia de un correcto y completo manejo de las capacidades del protocolo HTTP.

###front-end

Como todo proyecto web basado en páginas dinámicas, nuestro proyecto hace uso de HTML y Javascript. Pero más allá de estas dos tecnologías tan básicas y necesarias hoy en dia en cualquier proyecto web que se precie, por encima de ello tenemos implementado un amplio y rico ecosistemas de frameworks y librerías.

Angular
El primero de todos ellos sería Angular. Angular es un framework basado en javascript 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.

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).
Los 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.

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.

Gestores de tareas y despliegue
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.

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.

Modulos

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.

back-end.

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 sesión para la autenticación.

El funcionamiento de las cookies de sesión 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 sesión 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.

####front-end

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.

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).

Vistas

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:

Modelos

Bibliografía

Terminología.

Front-end. Cliente final. Conjunto de tecnologías que se encargan de las interacciones con los usuarios, fundamentalmente web.

back-end. Servidores. Con el conjunto de teconologías que residen en equipos remotos y que son accesibles a través de diversos protocolos comunicativos.

Full-stack. Proyectos que tienen implementación tanto en el lado front-end como en el lado back-end.