Integración de Qwizer con Moodle
Qwizer's integration with Moodle

Trabajo de Fin de Grado
Curso 2022–2023

Autor
Vicentiu Tiberius Roman
José Luis Bartha de las Peñas

Director
Manuel Montenegro Montes

Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

Integración de Qwizer con Moodle
Qwizer's integration with Moodle

Trabajo de Fin de Grado en Ingeniería Del Software

Autor
Vicentiu Tiberius Roman
José Luis Bartha de las Peñas

Director
Manuel Montenegro Montes

Convocatoria: *Junio 2023*

Grado en Ingeniería del Software
Facultad de Informática
Universidad Complutense de Madrid

29 de mayo de 2023

Agradecimientos

Primero de todo, queremos agradecer a nuestro tutor Manuel Montenegro Montes, por habernos ayudado y aconsejado a lo largo de todo el proyecto. Su enorme paciencia para resolver nuestras dudas y su flexibilidad para mantener nuestras reuniones a lo largo del curso, han permitido que podamos sacar este proyecto adelante.

Y en segundo lugar, personalmente, yo, José Luis Bartha, quiero a agradecer a mi compañero Tiberius por haberse involucrado tanto en este proyecto. Gracias a él se han cumplido muchos de los objetivos planteados para este Trabajo Fin de Grado e incluso algunos que no estaban ni planificados. Asimismo, he aprendido numerosas tecnologías que no pensaba que iba aplicar en este proyecto, que además reconoceré, que si no hubiera sido por él, no creo que las hubiera aprendido nunca y menos a utilizar. Definitivamente, sin él, no hubiéramos avanzado tanto como hemos hecho.

Y por último, nos gustaría agradecer a nuestros amigos y familiares, aquellas personas que han hecho que nuestro paso por la facultad sea lo más agradable posible y siempre tratan de apoyarnos a seguir en los peores momentos. Si ellos no hubiéramos sido capaces de llegar hasta aquí.

Vicentiu Tiberius Roman y José Luis Bartha de las Peñas

Resumen

Integración de Qwizer con Moodle

Qwizer es una aplicación web que permite la realización de cuestionarios dirigida sobre todo al ámbito académico. A diferencia de otras herramientas como los cuestionarios de Moodle, añade la característica de ser una aplicación web progresiva, es decir, permite al usuario utilizar la aplicación mientras que este no tenga una conexión a Internet activa. En este caso el usuario/alumno tendrá la capacidad para poder realizar los cuestionarios que tenga pendientes. Una vez que el alumno finalice un cuestionario se le generará un código QR que el profesor tendrá que escanear para obtener constancia de las respuestas enviadas por el estudiante. En el momento que el usuario recupere la conexión, las soluciones se mandarán al servidor y se registrará su intento.

Este proyecto es una ampliación de la aplicación Qwizer, en la que se han añadido nuevas funcionalidades como las siguientes:

- **Aleatorización de cuestionarios**: La aplicación contendrá la posibilidad de poder entregar a los alumnos cuestionarios aleatorizados, de modo que cada estudiante recibe una variante distinta de un cuestionario. En estos las preguntas elegidas para el cuestionario se aleatorizarán una vez el alumno se descargue el mismo.
- **Inclusión de enunciados en formato Markdown**: El lenguaje Markdown es un lenguaje de marcado que permite la inclusión de fórmulas y enunciados con distintos formatos de texto. Los cuestionarios y preguntas en Qwizer soportarán ahora este lenguaje.
- **Integración con Moodle**: Moodle es una de las herramientas más conocidas relacionadas con la gestión del aprendizaje. Una de las nuevas funcionalidades añadidas es la interoperabilidad entre Qwizer y la plataforma del Campus Virtual de la UCM para la gestión de calificaciones y cuestionarios.

Este proyecto no ha sido solo una ampliación, si no que también se ha centrado en una buena parte de estabilidad, refactorización y despliegue de la aplicación, haciendo que la misma sea más robusta, segura y amigable.

Enlace al proyecto en Github: https://github.com/Qwizer-UCM/Qwizer

Palabras clave

Qwizer, Campus Virtual, aplicación web progresiva, Aleatorización, Markdown, Docker, React, Django

Abstract

Qwizer's integration with Moodle

Qwizer is a web application that allows the completion of tests, aimed above all at the academic field. Unlike other tools such as Moodle quizzes, it adds the feature of being a progressive web application, which allows the user to use the application while they do not have an active Internet connection. In this case, the user/ student will have the possibility to complete the tests that are pending. Once the student completes a test, a QR code will be generated that the teacher will have to scan to obtain proof of the answers sent by the student. As soon as the user recovers the connection, the solutions will be sent to the server and their attempt will be recorded.

This project is an extension of the Qwizer application, in which new features have been added, such as the following:

- **Randomization of tests**: The application will contain the possibility of being able to give students randomized tests, so that each student receives a different variant of the test. In these, the questions chosen for the test will be randomized once the student downloads it.
- **Inclusion of Markdown and mathematical formulas**: The Markdown language is a markup language that allows the inclusion of formulas and statements with different text formats. Tests and questions in Qwizer will now support this language.
- **Integration with Moodle**: Moodle is one of the best-known tools related to learning management. One of the new features added is the interoperability between Qwizer and the UCM Virtual Campus platform for the management of qualifications and questionnaires.

This project has not only been an expansion, but has also focused in a good part of stability, refactor and deployment of the application, making it more secure and friendly to use for the users.

Link to the project on Github: https://github.com/Qwizer-UCM/Qwizer

Keywords

Qwizer, Virtual Campus, Progressive Web App, Randomization, Markdown, Docker, React, Django

Índice

Índice de figuras

Capítulo 1

Introducción

En esta memoria hablaremos del desarrollo de la ampliación y mejora de una aplicación progresiva para realizar cuestionarios en línea: Qwizer [4]. Empezaremos presentando que nos llevó a seguir este proyecto, qué objetivos hemos tenido a lo largo del proyecto y finalmente describiremos nuestro plan de trabajo a lo largo del curso académico.

1.1. Motivación del proyecto

El año pasado nos encontrábamos en la situación de elegir un trabajo que de verdad nos motivará a desarrollar nuestro último proyecto dentro de la facultad. Además, como estudiantes de la Facultad de Informática, queríamos un proyecto que de verdad impactase en la evolución de la misma y nos permitiera ayudar a mejorar sus servicios.

Manuel nos habló de una posible mejora y ampliación de una aplicación web progresiva basada en la realización de cuestionarios que permita realizarlos sin conexión a internet: Qwizer. Esta aplicación estaba dirigida a usuarios como nosotros, alumnos que a lo largo de nuestra trayectoria dentro de la facultad hemos tenido que realizar numerosos cuestionarios, sin añadir, que a mitad de nuestro paso, estalló una pandemia mundial debido al COVID-19 y este tipo de formato para la evaluación de nuestras asignaturas fue algo rutinario.

Inicialmente, la propuesta ya nos llamó la atención pero además identificamos una serie de causas que nos decantaron por progresar y continuar este proyecto. Entre ellas podemos nombrar las siguientes:

- **Nuevas tecnologías**: Esta aplicación utilizaba **tecnologías con las cuales no estábamos familiarizados** ninguno de los dos. Tanto React como Django eran dos tecnologías que desconocíamos pero sabíamos que iban a ser esenciales de dominar para nuestro desarrollo personal y profesional. Además, el hecho de ser una aplicación que funciona sin conexión a Internet, nos permite involucrarnos en el mundo de **las aplicaciones progresivas**, un mercado que hoy en día esta en auge por lo que es muy interesante desarrollar y descubrir las posibilidades de estas aplicaciones.
- **Uso práctico en un futuro**: Otro punto importante que nos hizo decidir-

nos por este proyecto, fue el **uso práctico** que le vimos a la aplicación. Bajo un buen desarrollo y el añadido de un mínimo de funcionalidades necesarias, veíamos que este proyecto podría llegar a ser útil. A lo largo de la carrera, hemos desarrollado numerosos proyectos con el único fin de ser calificados y focalizados a una simple entrega. Posteriormente, quedan olvidados o abandonados por falta de tiempo o motivación para seguirlos. En cambio, con este proyecto veíamos que si conseguíamos cumplir los objetivos podríamos hacer una **aplicación digna que podrían utilizar nuestros compañeros de la facultad** en un futuro.

- **Proyecto de innovación educativa de la UCM (proyectos INNOVA)**: Finalmente, además de los otros motivos, Manuel nos informó, como ya hemos comentado anteriormente, que este proyecto estaba destinado a seguir siendo desarrollado por algunos profesores de la facultad. De hecho, habían planteado un proyecto de innovación de la facultad, para que la aplicación fuera utilizada en los laboratorios de algunas asignaturas para calificar exámenes. Esto nos llamó bastante la atención y, ligado a lo que comentábamos al principio, veíamos que finalmente este proyecto **podría dejar huella en nuestra facultad**.

1.2. Objetivos

Una vez ya teníamos la propuesta del proyecto tuvimos que pensar en unos objetivos para desarrollar a lo largo de todo el curso académico. Esta tarea fue algo sencilla debido a que nuestros compañeros del año pasado y también nuestro tutor, nos ayudaron a identificar qué nuevas funcionalidades podrían añadirse a la aplicación Qwizer. Por lo que, finalmente, entre los consejos recibidos llegamos a plantear y establecer los siguientes primeros objetivos para el nuevo trabajo:

- **Aleatorización de cuestionarios**: Uno de los objetivos principales a futuro de nuestros compañeros del año pasado, fue el hecho de la **aleatorización dentro de los propios cuestionarios**. Ofrecer la capacidad de aleatorizar las preguntas de los cuestionarios es esencial en este tipo de aplicaciones para evitar copias entre los estudiantes. Por esta razón, llegamos a la conclusión que esta característica ofrecería a Qwizer un valor añadido.

  Para la implementación de esta faceta tuvimos que pensar detenidamente como aplicaríamos la aleatorización dentro de la aplicación, dado que dentro de un cuestionario se puede aplicar esto de numerosas maneras. Finalmente, identificamos tres objetivos principales:

  - Aleatorización a nivel de preguntas, es decir, las **preguntas de un cuestionario** deberían aparecer en un orden aleatorio para cada usuario que realice el mismo.
  - Aleatorización a nivel de opciones, es decir, las **opciones dentro de las preguntas** tipo *test* deberían mostrarse en un orden aleatorio para cada usuario.

    - Preguntas de selección aleatoria, es decir, dentro de los cuestionarios habrá una o varias preguntas que para cada usuario se elegirá en base a un **subconjunto de preguntas** del banco de preguntas.

- **Adaptación de aplicación a dispositivos móviles**: El diseño adaptable es un paso fundamental en las aplicaciones web. Dar a los usuarios que utilicen dispositivos móviles la posibilidad de ver nuestra aplicación de manera correcta es muy importante y más en el contexto de nuestra aplicación. Por tal motivo decidimos **intentar adaptar nuestra interfaz a los dispositivos móviles** y intentar que la **navegación por la aplicación** fuera lo más cómoda posible.

- **Inclusión de Markdown y fórmulas matemáticas**: Markdown es un tipo de lenguaje de marcado que permite **añadir a la aplicación texto enriquecido** permitiendo la inclusión de palabras en negrita, cursiva, imágenes, etc... Nuestro tutor identificó la **incorporación de Markdown** como uno de los objetivos principales del TFG y agregándole también la **posibilidad de redactar enunciados y soluciones con fórmulas matemáticas**, permitiendo la formulación de preguntas más complejas.

- **Integración con Moodle**: Finalmente, el último objetivo que planteamos fue la **integración con los formatos** de la plataforma **Moodle**. Moodle fue una de las herramientas que inspiró el desarrollo de Qwizer, de hecho, actualmente es la plataforma en la que se basa el **Campus Virtual** de la nuestra facultad. Dado que, a día de hoy, se sigue utilizando, la capacidad de integrar Qwizer con Moodle fue uno de nuestros objetivos más esperados. Para ello nos quisimos centrar en que se pudieran **importar preguntas del banco de preguntas de Moodle** y **permitir exportar las notas de los estudiantes que realizaron cuestionarios en Qwizer a Moodle**.

1.3. Plan de trabajo

A lo largo de este curso académico, hemos pasado por numerosas fases dentro del desarrollo de nuestro TFG. El hecho de ampliar un proyecto completo, implicaba etapas de estudio no solo de las nuevas tecnologías con las que no estábamos familiarizados, sino también grandes estructuras de códigos que hacen uso de las mismas. Debido a todo esto, necesitábamos un plan de trabajo que nos permitiera ir estableciendo pequeños objetivos para completar las tareas que teníamos planteadas.

Gracias a las reuniones que realizábamos con nuestro tutor Manuel Montenegro, las cuales se hacían cada dos semanas, nos organizábamos para ir cumpliendo una serie de tareas para cada reunión. Esta metodología nos ayudaba, al principio, a ir entendiendo poco a poco partes del código y empezar a cumplir las primeras metas, y posteriormente, en etapas más avanzadas del proyecto, a finalizar los objetivos principales del trabajo. Dicho esto, la distribución general de las fases de nuestro proyecto fueron las siguientes:

- **Refactorización, *testing*, actualización y despliegue de la aplicación** (*septiembre de 2022 - febrero de 2023*) Una de las etapas más largas de desarrollo. Al principio, nos dedicamos a entender gran parte del código y posteriormente refactorizar tanto el *frontend* como se comenta en el capítulo 3.5.1 como en el *backend* como se comenta en el capítulo 3.5.2. También actualizamos todas las librerías a la ultima versión intentando evitar conflictos por los cambios de versiones. Por último, preparamos mejor el entorno de desarrollo añadiendo `linters`, variables de entorno para los secretos, diseñando algunos *tests* para el *back-end*, usando un gestor de paquetes para este último y usando Docker para desplegar el proyecto de manera correcta.

- **Aleatorización de cuestionarios** (*marzo de 2023*): Durante esta etapa, se implementó la funcionalidad de aleatorización de cuestionarios. Este proceso involucró realizar varios cambios en el modelo relacional de la aplicación y en lógica de la aplicación como se detalla en profundidad en el capítulo 4, lo que requirió un tiempo significativo para actualizar el código correspondiente.

- **Markdown, preguntas aleatorias e integración con Moodle** (*abril de 2023*): Durante esta fase, se abordaron varios aspectos clave. La implementación del formato Markdown para las preguntas y la integración con Moodle fueron relativamente sencillas, a excepción de la gestión de las imágenes en el caso de Markdown. Sin embargo, la incorporación de preguntas aleatorias resultó ser un desafío considerable, ya que tuvo un impacto en gran parte del código del *back-end* de la aplicación. Estas incorporaciones están descritas en los capítulos 4, 5 y 6.

- **Comprobaciones finales y memoria** (*mayo de 2023*): Durante el último mes de desarrollo, nos enfocamos en realizar los ajustes finales en la interfaz de la aplicación, puliendo los detalles para lograr una experiencia de usuario mejorada. También dedicamos tiempo a redactar la memoria del Trabajo de Fin de Grado (TFG), documentando todo el proceso de desarrollo, las decisiones tomadas y los resultados obtenidos.

Capítulo 2

Selección de herramientas y tecnologías

En nuestro proyecto hemos hecho uso de numerosas tecnologías que nos han ayudado a la implementación final de nuestra aplicación. Dado que este proyecto es una continuación de uno anterior, hemos tenido que, no solo adecuarnos a muchas de las tecnologías que nuestros compañeros utilizaron el año pasado, sino que también hemos tenido que investigar numerosas tecnologías para implementación de nuestros nuevos objetivos. En esta sección vamos a explicar qué herramientas y tecnologías hemos utilizado finalmente para el desarrollo de Qwizer.

2.1. Lenguajes de programación, gestión de bases de datos y *frameworks* utilizados

Primero de todo, empezaremos nombrando que tipos de lenguajes de programación, gestores de bases de datos y marcos de trabajo han sido utilizados para el desarrollo del proyecto.

2.1.1. Python

Python es un lenguaje de programación interpretado y altamente conocido en términos de aplicaciones web y desarrollo de software. En nuestro caso es el principal lenguaje que incorpora el marco de trabajo que hemos utilizado en la parte de servidor de nuestra aplicación. Dado que fue una decisión de nuestros compañeros del año pasado, este año hemos decidido seguir con este mismo lenguaje para la implementación del servidor. Además, este año hemos hecho uso de entornos virtuales con la librería `Poetry`, permitiéndonos no tener que instalar todas las librerías que requiere el proyecto en nuestros equipos. Esta posibilidad nos ha ayudado a añadir una capa de aislamiento con nuestros sistemas y hacer más liviana su instalación.

2.1.2. Django

Django [3] es uno de los *frameworks*, o los también conocidos como marcos de trabajo, que más se utilizan en el desarrollo de las aplicaciones web. Su robusta y vasta capacidad de posibilidades para el control de la seguridad sobre las vulnerabi-

lidades más frecuentes en el entorno web, un panel de administración sencillo y bien integrado y un extenso ORM fueron las razones por las que nuestros compañeros del año pasado decidieron utilizar esta herramienta para el desarrollo de la aplicación Qwizer en la parte del servidor. Pese a estar más familiarizados con *Node.js* hemos decidido seguir con esta tecnología para aprovechar el código ya escrito.

2.1.3. JavaScript

JavaScript es un lenguaje de programación interpretado altamente conocido por su uso en el desarrollo web. Para nosotros, es el principal lenguaje de programación utilizado por el *framework* que utilizamos para el lado del cliente en la aplicación. Además, este lenguaje es esencial para una de las funcionalidades más atractivas de Qwizer, los *Service workers*. Estos permiten que la aplicación funcione aun estando el cliente sin conexión. Esta propiedad hace que las aplicaciones web sean conocidas como *aplicaciones web progresivas*. Agregado a lo anterior, dado que la aplicación se puede utilizar en el navegador, podemos hacer uso del almacenamiento interno del mismo gracias a este lenguaje. Con esto podemos hacer que los usuarios guarden datos en este pequeño almacenamiento, como son sus datos de sesión y algunos cuestionarios descargados que podrán realizar en caso de que la aplicación se quede sin conexión. También, gracias a su vasto número de librerías y de facilidades, Javascript ofrece una estructura sólida para realizar llamadas HTTP a nuestro servidor de manera sencilla.

2.1.4. React

React [13] es una de las bibliotecas más conocidas de Javascript para el desarrollo de aplicaciones web. Su arquitectura basada en el modelo *SPA* de aplicaciones web, consistentes en una sola página, fue una de las razones principales por la que nuestros compañeros del año pasado decidieron utilizar esta tecnología.

Una de las ventajas más importantes de React es su estructura por componentes. La programación por componentes se basa en la creación de pequeñas partes de la vista con incluso lógica propia. Esto permite que se pueda reutilizar y hacer mucho más entendible el código. La programación en React se basa en la creación de componentes estructurados mediante clases o mediante funciones. El año pasado, nuestros compañeros decidieron escribirlos por clases. Hoy en día las definiciones mediante clases, han quedado prácticamente obsoletas por lo que hemos decidido darle un enfoque funcional a estas. En este tipo de definiciones los componentes se encapsulan en funciones y en la función de retorno es donde se escribe el código JSX que más adelante se mostrará al usuario. El formato JSX, permite integrar código HTML dentro de JavaScript lo cual hace mas cómodo el desarrollo con React.

Anteriormente se utilizaba `create-react-app` para la compilación del código. Sin embargo, debido a los numerosos avisos que nos salían en la consola al ejecutar la aplicación y al lento mantenimiento que se le daba a la herramienta decidimos in-

vestigar y buscar alguna alternativa. Llegamos a una herramienta denominada `Vite` que, además de estar mejor mantenida, también ofrecía mejoras en el rendimiento a la hora de desarrollar ya que en vez de recompilar el proyecto entero cada vez que se realizaba algún cambio, solo se recompilaban los ficheros con cambios. Al probar la herramienta por primera vez nos sorprendimos por la rapidez con la que se ponía el proyecto en funcionamiento.

2.1.5. SQL

SQL es un lenguaje utilizado para el acceso y gestión de las bases de datos. En nuestro proyecto, gracias al ORM de Django, hemos podido evitar tener que utilizar este lenguaje para muchas de las consultas de la aplicación. Aun así, en algunos casos puntuales hemos necesitado de este lenguaje para hacer consultas muy especificas que el ORM Django no soportaba.

2.1.6. Service workers

Un *Service worker* es la tecnología que permite generar una aplicación web progresiva. Básicamente, consiste en un fichero JavaScript que se ejecuta en segundo plano, ofreciendo al cliente soluciones en caso de que se quede sin conexión a Internet o el servidor al que vaya a realizar las peticiones no esté disponible. En nuestra aplicación, utilizamos esta tecnología para guardar los recursos necesarios para que mientras el cliente se encuentre sin conexión, pueda disfrutar de una experiencia limitada de nuestros servicios. Otras de las tareas fundamentales que desempeña, es la de almacenar las peticiones que realice el cliente en el periodo que este sin conexión, y una vez que la recupere, volverá a mandar estas peticiones al servidor.

2.1.7. Bootstrap

Bootstrap [8] en una biblioteca CSS de JavaScript que hemos utilizado para el estilado general de toda la aplicación. Aparte que nuestros compañeros ya la utilizaron para este mismo motivo, nosotros hemos decidido mantenerla dado que hemos utilizado esta herramienta en numerosas asignaturas y estamos bastante familiarizados con ella. Además, nos permite generar una interfaz adaptable a las dimensiones de cualquier dispositivo y así cumplir uno de los objetivos principales. Cabe destacar también, que esta librería dispone de una documentación muy extensa e incluso con ejemplos explicativos.

2.2. Lenguajes de intercambio de datos y marcado

Como segundo apartado hablaremos de los lenguajes que hemos tenido que utilizar para intercambiar datos entre el servidor y el cliente de nuestra aplicación.

Aparte también introduciremos uno de los principales lenguajes de marcado del proyecto que ha permitido el desarrollo de uno de los objetivos de nuestro TFG.

2.2.1. JSON

JSON es un formato de fichero de texto ampliamente utilizado para el intercambio de datos dentro de las aplicaciones. Dadas su flexibilidad y vasto uso dentro de las implementaciones de las APIs y el desarrollo web hemos decidido utilizarlo para intercambiar información a través de nuestra API. Su sintaxis es ligera, lo que facilita los trabajos de decodificación y codificación de los datos utilizados. Como otra ventaja a añadir, la posibilidad de ser utilizado en cualquier lenguaje de programación nos ayuda en el tratamiento de datos entre el lado cliente y servidor de la aplicación.

2.2.2. YAML

YAML es un formato de serialización de datos, con similitudes a los formatos XML y JSON. El principal atractivo de este formato es su fácil lectura y legibilidad. Dado que esta aplicación va dirigida a usuarios que a lo mejor no son expertos en formatos algo más técnicos como pueden XML y JSON, nuestros compañeros del curso anterior decidieron elegir este formato para la subida de archivos referentes a la creación de cuestionarios y preguntas.

2.2.3. XML-Moodle

XML-Moodle [9] es un formato que utiliza la plataforma de Moodle para importar y exportar preguntas. En nuestro caso lo hemos utilizado para cumplir el objetivo de la importación de preguntas desde Moodle a Qwizer. El uso de este formato se detalla más en el Capítulo 6.

2.2.4. Markdown

Markdown es un lenguaje de marcado que permite la creación de texto especializado sin utilizar ningún formato especifico. Este lenguaje permite escribir fórmulas matemáticas, agregar imágenes o mejorar el estilo a textos con una sintaxis muy sencilla. Este lenguaje es esencial para uno de nuestros objetivos del TFG, por lo que la adición a nuestro proyecto era esencial. En capítulos posteriores explicaremos las principales razones de su uso pero se encargará de profesionalizar más nuestra aplicación.

2.3. Otras herramientas o tecnologías

En este último apartado añadiremos tecnologías o herramientas que no podemos agrupar y explicaremos en general de manera independiente el uso que tienen dentro de nuestra aplicación.

2.3.1. Git

Git es uno de los gestores más conocidos por la industria del desarrollo para el control de versiones. En estos proyectos en los que se tratan con una alta cantidad de archivos, llevar un control de todos ellos es algo complicado y de increíble importancia. Git nos ayuda y libera de este problema, con una lista de comandos muy sencilla y fácil de aprender. Además, con su posibilidad de restaurar cualquier versión anterior de los archivos nos permite trabajar tranquilos ante errores inesperados o no deseados.

2.3.2. Github

Github es un plataforma de alojamiento de proyectos que utilizan el control de versiones de Git. Dado que utilizamos Git como sistema de control de versiones, alojar nuestro proyecto en Github era sumamente sencillo y nos aportaba numerosos beneficios, como la mejor visualización del proyecto y la facilidad para seguir el trabajo de ambos.

2.3.3. Docker

Docker [17] es una herramienta de despliegue de software en sus llamados *contenedores virtuales*. Con la finalidad de desplegar nuestro proyecto, hemos utilizado Docker para preparar la aplicación para un entorno de producción y no solo de desarrollo.

Una parte importante de cualquier proyecto es el despliegue del mismo. Para facilitar el despliegue de Qwizer en cualquier tipo de maquina y sin necesidad de muchos conocimientos en las tecnologías usadas decidimos usar Docker para modularizar nuestra aplicación mediante contenedores. Empezamos por investigar como se desplegaban correctamente las aplicaciones con *React* y *Django*. Con la ayuda de `docker-compose` para ejecutar varios contenedores hemos hecho uso de las siguientes imágenes:

- PostgreSQL
- Node
- Python

- Nginx

La creación de los contenedores se aborda en mayor detalle en el Capítulo 7, donde se proporciona una explicación exhaustiva de este proceso.

2.3.4. PostgreSQL

PostgreSQL o Postgres, es un sistema gestor de base de datos relacional que soporta el almacenamiento de objetos. En nuestro caso, hemos decidido utilizarlo debido a que cumple de forma mucho más estricta con el estándar de SQL, además de que tiene licencia de código abierto.

2.3.5. LaTeX

LaTeX es una herramienta de composición de textos para documentos especializados. En nuestro caso hemos decidido utilizar LaTeX para redactar la memoria con el fin de mejorar su aspecto y facilitar el proceso de redacción, a diferencia de otras herramientas como Word o LibreOffice que son más complicadas de utilizar para el desarrollo de un documento más extenso como es nuestro caso.

2.3.6. Visual Studio Code

Visual Studio Code es un editor de código fuente ampliamente utilizado en el desarrollo software. Para nosotros ha sido el principal IDE donde hemos desarrollado la mayor parte del código. Dentro de nuestras principales razones para su uso están:

- **Legibilidad**. Es un entorno muy amigable donde su paleta de colores ayuda en hacer un código limpio.
- **Sencillez**. Su interfaz es sencilla y entendible, no tiene un exceso de elementos que oculten las funcionalidades principales del entorno.
- **Extensiones**. El amplio catalogo de extensiones que tiene facilita y ahorra tiempo en el desarrollo. Las extensiones que más hemos utilizados son las siguientes:
  - **Docker**: Dado nuestro uso de contenedores para el desarrollo, Visual Studio contiene una extensión propia de Docker que permite el manejo de los contenedores e imágenes.
  - **TODO Tree**: A lo largo del desarrollo hemos ido dejando pequeñas anotaciones con tareas pendientes por hacer para un futuro. Esta extensión permite remarcar esas anotaciones en cada archivo y ofrece una interfaz para saber en cada archivo dónde quedan pendientes cosas por hacer.

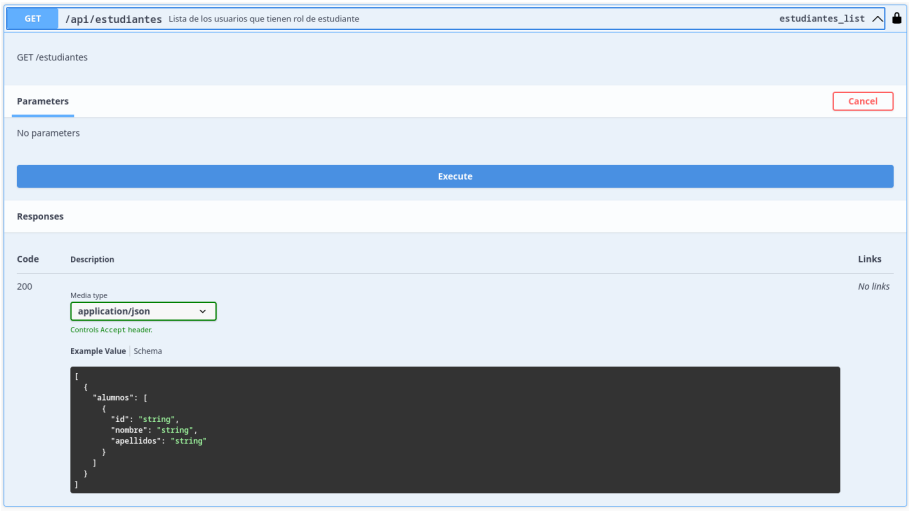

Figura 2.1: Ejemplo de uso de Swagger

- **Github**: Esta extensión permite realizar las operaciones básicas de Git directamente en el entorno de trabajo. Además permite ver versiones anteriores de cada archivo y ver las modificaciones realizadas por cada uno de nosotros.

2.3.7. Swagger

Swagger [16] es un conjunto de herramientas de código abierto diseñadas para documentar servicios web basados en arquitecturas *RESTful*. En nuestro proyecto, consideramos fundamental priorizar la documentación de la *API* con el fin de facilitar el desarrollo futuro de la aplicación. Para lograr esto, descubrimos Swagger, una herramienta que, siguiendo el estándar de *OpenAPI 3*, genera una interfaz web donde se documenta de manera detallada la *API*, además de brindar la posibilidad de realizar pruebas interactivas. En la Figura 2.1 se muestra un ejemplo de cómo se utiliza Swagger.

El enfoque de Swagger en la generación de documentación automatizada permite ahorrar tiempo y esfuerzo al proporcionar una representación clara y estructurada de los puntos finales de la *API*, los parámetros requeridos, los tipos de datos admitidos y las respuestas esperadas. Además, al ofrecer una interfaz interactiva, Swagger permite a los desarrolladores probar la *API* directamente desde la documentación, lo que facilita la depuración y el desarrollo ágil.

2.4. Tecnologías descartadas

2.4.1. JQuery

JQuery [7] es una biblioteca de JavaScript que permite una interacción más simplificada con los elementos HTML directamente desde un fichero JavaScript. El año pasado, nuestros compañeros la utilizaron para actualizar el contenido de algunos textos de las vistas, pero nosotros hemos eliminado todo comportamiento relacionado con esta tecnología y hemos llegado a descartarla.

2.4.2. Ionic

Ionic [11] es un *framework* de estilado que permite desarrollar interfaces de usuario para dispositivos móviles con tecnologías web. Consideramos utilizar esta tecnología para conseguir un diseño adaptable exclusivo para los dispositivos móviles, en vez de utilizar nuestra librería de estilado, *Bootstrap*. Sin embargo, por falta de tiempo decidimos abandonar esta idea y no utilizar esta opción.

Capítulo 3

Arquitectura global del sistema

En este capítulo explicaremos el núcleo y la organización principal de nuestra aplicación. Qwizer, como muchas aplicaciones web, está compuesto de un parte diseñada en el lado del servidor, también conocida como *back-end*, y además otra parte diseñada para el lado del cliente, comúnmente denominada *front-end*. Para el correcto funcionamiento de la aplicación es esencial una buena comunicación entre ambos lados. Aparte, nuestra aplicación se apoya en una *base de datos* que almacena la información necesaria para que la aplicación pueda funcionar.

3.1. Base de datos

Para gestionar la base de datos decidimos utilizar PostgreSQL. Este sistema gestor de bases de datos es uno de los más potentes, en temas de rendimiento, del mercado. Como explicamos en el capítulo anterior, este gestor contiene una licencia de código abierto por lo que de cara al futuro desarrollo de esta aplicación y su despliegue va a ser un factor esencial. Además, su analizador sintáctico es algo más precisa que los otros gestores de base de datos conocidos como pueden ser *MYSQL* o *MariaDB*, donde ligeros errores en las consultas o modificaciones del propio lenguaje SQL pueden funcionar sin seguir el verdadero estándar del lenguaje original SQL. Esta estrictez nos lleva a escribir un código más fiel al que hemos aprendido en estos años anteriores y a no depender en ningún aspecto del propio gestor de base de datos.

Dicho esto, como también explicamos anteriormente, Django ofrece un ORM muy extenso. Este mapeador de objetos nos presenta la posibilidad de la utilización de los objetos llamados *modelos*. Estos modelos permiten una sencilla implementación de los objetos a guardar en la base de datos, posibilitando modificar cualquier propiedad de los atributos a guardar, como puede ser la declaración de claves externas, o las conocidas como *foreign keys*, o los tipos de campos únicos, o *unique*.

En la figura 3.1, podemos observar la implementación de uno de los modelos de Django.

Esta transformación que pasa el modelo a objeto SQL es lo que se conoce en Django como una **migración**. Las migraciones son generadas por el propio Django una vez ejecutamos dentro de nuestro proyecto el comando de `python manage.py makemigrations`. Una vez ya tengamos todas las migraciones de nuestros mode-

los, podremos ejecutar el comando `python manage.py migrate` para ejecutar todo el código SQL generado por el comando anterior en el gestor de base de datos y así formar todas las tablas consecuentes.

El uso de esta característica de Django ayuda enormemente no solo en la creación de base de datos si no también en la propia modificación de la misma. En nuestro caso, dado nuestros objetivos como proyecto, hemos tenido que realizar incontables cambios en la base de datos, por lo que tener a nuestra disponibilidad esta herramienta ha sido de enorme ayuda.

3.1.1. Modelo entidad-relación

Como todo proyecto donde se involucre una base de datos, se debe planificar la estructura de la misma con un modelo entidad-relación.

Como explicamos anteriormente, dadas a las especificaciones del nuestro proyecto, la base de datos ha sufrido numerosos cambios a lo largo de todo el transcurso del trabajo, no solo con atributos o propiedades modificadas sino que también hemos añadido nuevas entidades y hemos reestructurado todo el modelo en general.

Este año recibimos un modelo entidad-relación como el que podemos ver en la figura 3.2. Este modelo contaba con numerosas entidades y relaciones que deberíamos entender con rapidez para empezar a trabajar en el proyecto.

Tras numerosas modificaciones y ajustes llegamos al modelo entidad-relación final que se presenta en la figura 3.3. En las siguientes secciones explicaremos más en detalle como fue el desarrollo de las principales modificaciones de la base de datos.

3.1.2. Modelo relacional

El modelo entidad-relación es una manera de explicar la base de datos a muy alto nivel, por lo que la creación de un modelo relacional es fundamental para el buen entendimiento de la estructura de la base de datos. Django tiene herramientas para generar de manera automática un modelo relacional a partir del código con todas las relaciones y atributos explicados con alto detalle, pero su poca legibilidad nos motivó a hacer un modelo relacional propio para, no solo entender mejor el

```
1 class Asignatura(models.Model):
2     objects = AsignaturaManager()
3     nombreAsignatura = models.CharField(blank=True,max_length=254,
4                         verbose_name="nombreAsignatura",
5                         unique=True)
6
```

Figura 3.1: Modelo de Django implementado

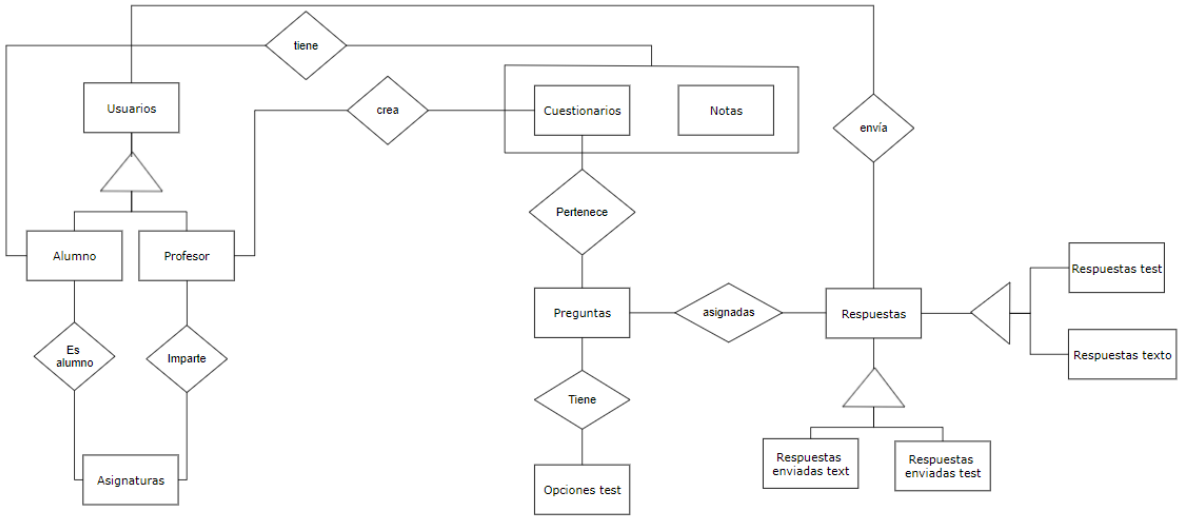

Figura 3.2: Modelo entidad-relación Qwizer del año pasado [4]

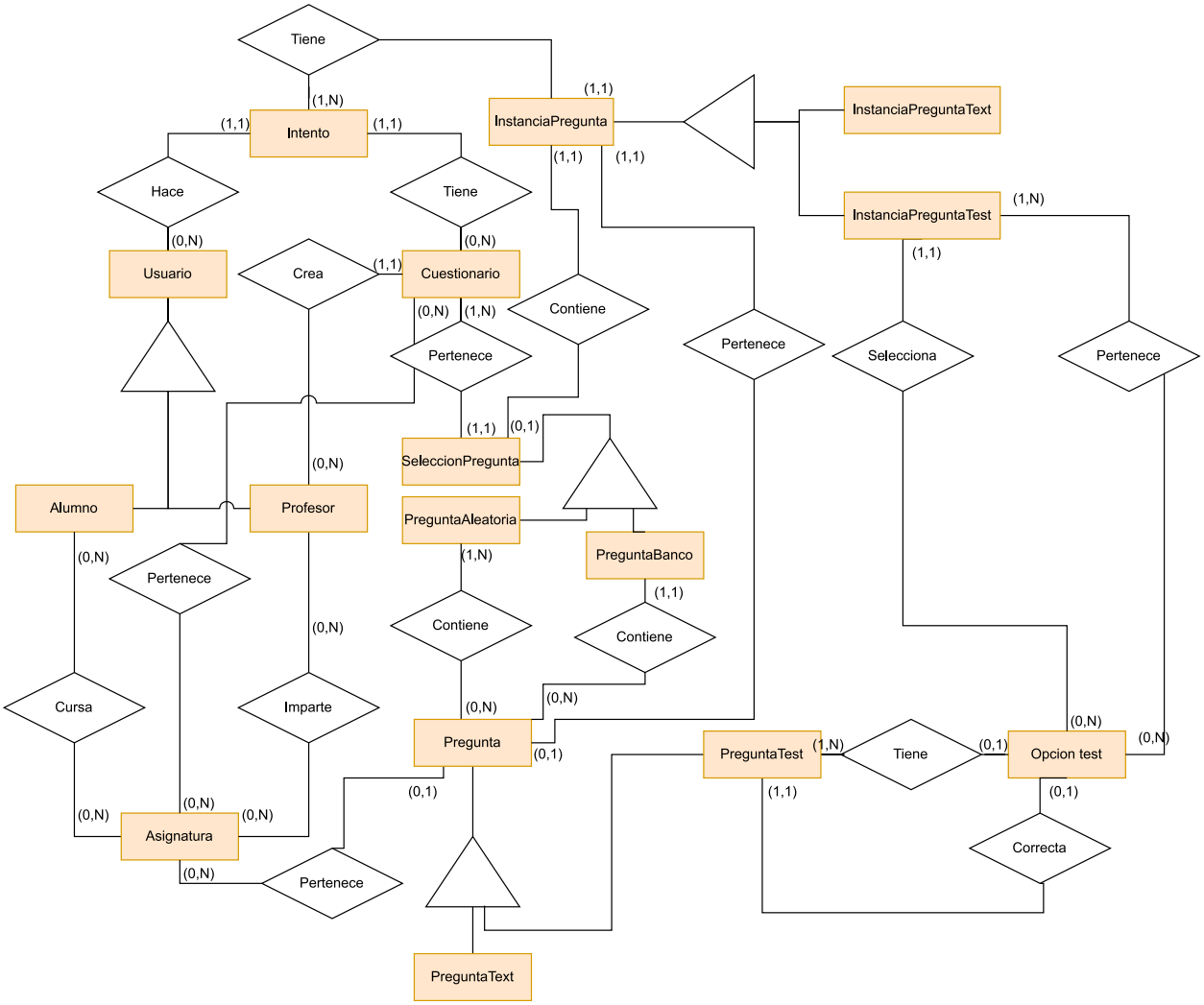

Figura 3.3: Modelo entidad-relación final

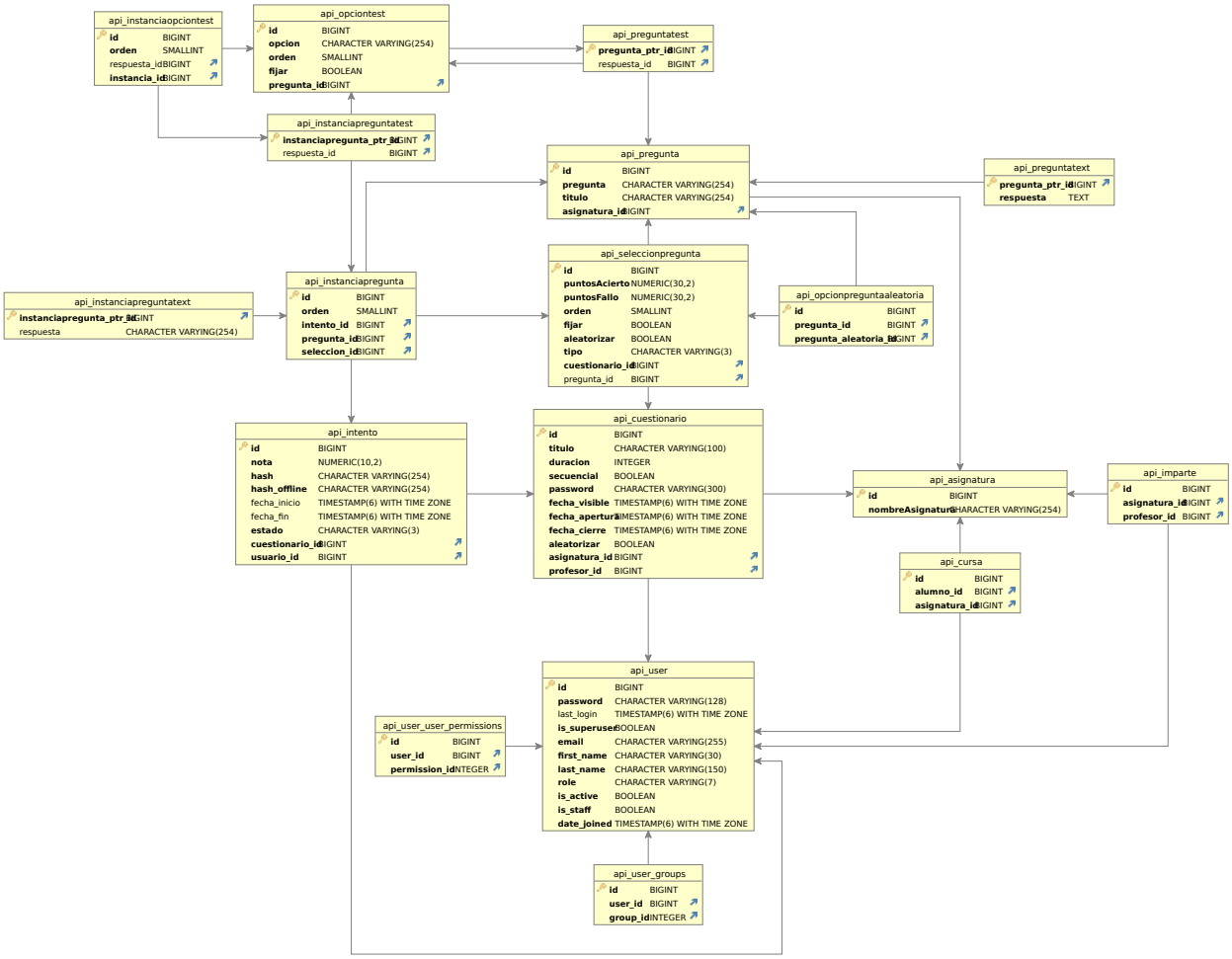

Figura 3.4: Modelo relacional

esquema de la base de datos que heredamos de nuestros compañeros, sino también para ayudar en un futuro a posibles compañeros que decidan proseguir con nuestro trabajo.

Como podemos ver en la figura 3.4, realizamos un modelo relacional en la aplicación con la herramienta *Draw.IO* que nos ayudó enormemente a lo largo de este proyecto.

3.2. API de comunicación entre *back-end* y *front-end*

Como hemos ido anticipando, la comunicación entre los dos lados de la aplicación es importante para el funcionamiento de la misma. Para ello se realizan las APIs, que permiten que el *back-end* y el *front-end* se comuniquen, intercambiando datos o ejecutando acciones especificas. Como proyecto continuación de otro anterior, heredamos una API ya realizada con anterioridad por nuestros compañeros, que hemos

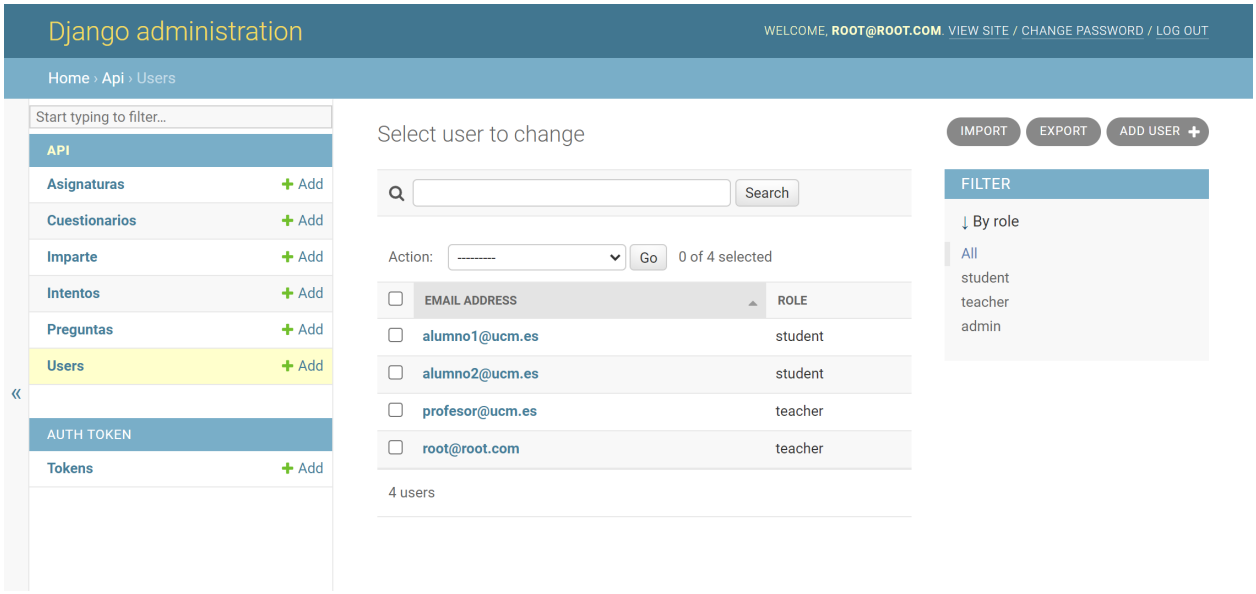

Figura 3.5: Vista del panel de administración de Django

modificado y adaptado mucho mejor al entorno de Django y React, nuestras dos tecnologías principales para cada lado respectivamente.

3.3. Aspectos de Django

En esta sección hablaremos sobre los aspectos más específicos del *framework* Django que hemos utilizado para el desarrollo *back-end* de nuestra aplicación.

3.3.1. Panel de administración

Una de las herramientas más potentes de Django es su terminal de administración. Esta herramienta ofrece la posibilidad a los usuarios *."administradores"* de realizar consultas e inserciones en la base de datos con mucha facilidad. De hecho, tal y como también hicieron nuestros compañeros del curso pasado, es posible delegar en esta herramienta de administración algunas de las funcionalidades que no se han implementado directamente en la aplicación web de Qwizer, tales como el registro de profesores o el registro de nuevas asignaturas. En la figura 3.5 podemos ver la vista de la herramienta.

Lo que sí hemos tratado de hacer durante este año es mejorar el uso del panel de administración, exprimiéndola lo máximo posible y ajustándola solo a las funcionalidades que necesitamos que tenga. Entre las mejoras realizadas podemos encontrar las siguientes:

- **Focalización en las opciones más relevantes**: Anteriormente, el panel de administración ofrecía el acceso a todas las tablas de la aplicación, donde

algunas de ellas no eran necesarias, ya que dentro de Qwizer estaban esas funcionalidades. Por lo que decidimos eliminar el acceso a las tablas innecesarias y dejar la interfaz mucho más limpia y escueta para los propios administradores.

- **Registro de usuarios**: Cuando se incluye la tabla de usuarios en el panel de administración, por defecto, se utiliza la entidad `Usuarios` de Django, la cual tiene numerosos campos que nosotros no tenemos en cuenta para el almacenamiento de los mismos. Esto hacía que la creación de usuarios fuera algo lenta e incluso pesada. Por ello decidimos eliminar los atributos innecesarios y así simplificar el proceso de registro.

- **Inserción en lote de usuarios**: Una nueva funcionalidad que nos pareció bastante útil fue el hecho de añadir la posibilidad de insertar usuarios en lote. Anteriormente, los usuarios se tenían que crear uno a uno. Por tanto permitimos que los *administradores* pudieran añadir varios usuarios de una sola vez. Para ello, en la tabla de usuarios hay una opción de importar usuarios donde podemos introducir un fichero CSV que contenga toda la información de los usuarios para registrarlos directamente. Un ejemplo de esto sería la figura 3.6.

email	firstname	lastname	password	role
alumno1@ucm.es	alumno	1	1	student
alumno2@ucm.es	alumno	2	2	student
profesor@ucm.es	profesor		3	teacher

Figura 3.6: Fichero CSV con usuarios

3.4. Librerías externas

En esta sección hablaremos sobre las librerías externas de Django y React que hemos utilizado en la aplicación.

3.4.1. Librerías para el *back-end*

- **djangorestframework [19]**: Permite la creación de una API REST de manera sencilla. El año pasado, entre otras cosas, nuestros compañeros la utilizaron por el sistema de *Tokens* de sesión que utilizaba esta librería, la cual simplificaba el complejo sistema de sesiones de Django y conseguía los objetivos relacionados con el apartado *offline* de la aplicación. Nosotros hemos decidido ir un poco más lejos y aprovechar más esta librería utilizando sus componentes básicos para la creación propia de la API. Dado que esto supone una mejora sustancial en la implementación, hablaremos más extensamente sobre el uso de esta librería en el apartado 3.5.2.

```
#qwizer_be/.env.sample
REACT_APP_API_URL=
REACT_HOST=
REACT_PORT=

#qwizer_fe/.env.sample
SECRET_KEY=
DEBUG=
ALLOWED_HOSTS=
DATABASE_ENGINE=
DATABASE_NAME=
DATABASE_USER=
DATABASE_PASSWORD=
DATABASE_HOST=
DATABASE_PORT=
CORS_ALLOW_ALL_ORIGINS=
CORS_ORIGIN_WHITELIST=
CSRF_TRUSTED_ORIGINS=

```

Figura 3.7: Fichero `env` de Django

- **django-environ [15]**: Permite la creación de un entorno de desarrollo en Django más amigable y seguro. Tanto en el despliegue como en el desarrollo del proyecto, Django maneja numerosas variables de configuración y hacen que sus ficheros sean difíciles de seguir. Además, el uso de esta biblioteca permite encapsular en un **solo** fichero `.env` todas las variables de configuración que se necesiten. Esto facilita el entendimiento general de la configuración del proyecto y ayuda a realizar modificaciones de manera más simple. En la figura 3.7 podemos ver un ejemplo de este fichero de configuración.

- **django-import-export [2]**: Permite importar y exportar objetos relacionados con las tablas de la base de datos. En la sección anterior hablamos del uso del panel de administración dentro de nuestro proyecto. Esta librería es la que nos ha permitido añadir usuarios en lote.

- **djoser [18]**: Ofrece una serie de *endpoints* relacionados con acciones básicas como son el registro, inicio y cierre de sesión. Esta librería se integra con el modelo de *Tokens* de la biblioteca **djangorestframework** para intentar mejorar el sistema de sesiones sobre todo con *frameworks* como React, que mantienen un modelo SPA que dificulta el mantenimiento de las propias sesiones.

- **drf-spectacular [6]**: Permite la documentación de la API en Django. Esta librería, que trabaja con el estándar de Open API 3, permite la integración con Swagger(ver subsección 2.3.7) nos ha ayudado a una generación de la documentación de la API muy detallada.

- **gunicorn [1]**: También conocido como Green Unicorn, es un servidor HTTP

compatible con Django que se utiliza para ejecutar aplicaciones web Django. Actúa como un servidor intermediario entre el servidor web y la aplicación Django, gestionando las solicitudes HTTP y enviando las respuestas correspondientes. Gunicorn implementa el estándar WSGI (Web Server Gateway Interface), que define una interfaz común entre los servidores web y las aplicaciones web en Python. En resumen, Gunicorn es una parte integral del entorno de Django, brindando un servidor HTTP robusto y eficiente para ejecutar aplicaciones web Django en producción.

3.4.2. Librerías para el *front-end*

- **axios [10]**: Permite tener un cliente HTTP para hacer llamadas a la API de manera más cómoda. En la sección de mejoras (ver sección 3.5) hablaremos más extensamente de cómo hemos añadido esta librería a nuestro proyecto, pero el año pasado, nuestros compañeros utilizaban la propia llamada `fetch` de JavaScript puro, donde tenían que construir ellos sus propias peticiones para llamar a la API. Con **axios** este proceso se simplifica con una sintaxis muy sencilla y permitiendo centralizar todo el manejo de llamadas a la API en un solo componente.

- **localforage [12]**: Permite el acceso al almacenamiento local y a la base de datos indexada en el navegador a través de un programa en JavaScript. En nuestra aplicación hacemos un gran uso del almacenamiento local para guardar cuestionarios, *tokens* e incluso información del usuario. Esta biblioteca nos permite acceder a este almacenamiento y también a la base de datos indexada que hay en los navegadores, la cual permite almacenar mucho más contenido que los escasos 5MB del almacenamiento local. Además, la librería ofrece una interfaz muy fácil de usar muy parecida al esquema de clave-valor que estamos acostumbrados a usar.

- **react-router-dom [14]**: Permite simular en React la función de servir distintas páginas al cliente desde diferentes rutas. Normalmente, en las páginas web que no utilizan React, cuando accedemos a un ruta especifica el servidor nos devuelve un fichero HTML único para esa ruta. Como React se basa en un modelo SPA no podemos permitirnos esta característica, por lo que tenemos que simularla de alguna manera. Esta librería nos permite devolver un componente de React por cada ruta instanciada en la aplicación manteniéndose siempre en una sola página.

- **hello-pangea/dnd [5]**: Permite un sistema de *drag and drop* en las listas de nuestra aplicación. Algunas veces, dentro de Qwizer necesitamos que nuestros usuarios coloquen una serie de preguntas en orden, por lo que con el fin de facilitarles esta tarea hemos decidido añadir esta librería, que aparte de hacer más estético este proceso, permite cambiar el orden entre los elementos dentro de una lista. En la figura 3.8 podemos ver un ejemplo de esta librería en uso.

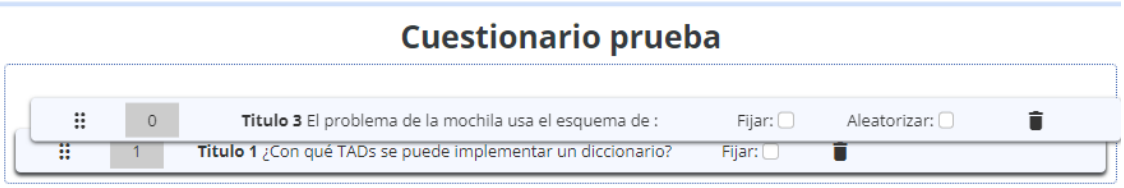

Figura 3.8: Ejemplo de librería hello-pangea/dnd

3.5. Mejoras realizadas sobre el trabajo anterior

En esta sección vamos a hablar de las mejoras que hemos realizado a lo largo del proyecto sobre los elementos que ya venían implementados en el trabajo del año pasado.

3.5.1. *Front-end*

3.5.1.1. Refactorización de clases a funciones

El código de un componente en React se puede estructurar de dos maneras: mediante clases o mediante funciones. Según los propios desarrolladores de React, el sistema mediante clases ha quedado totalmente obsoleto debido al cambio en el estándar de JavaScript y a problemas de rendimiento. Por esta razón, decidimos que debíamos cambiar todos los componentes de nuestros compañeros del año pasado a un enfoque funcional, es decir, componentes basados en funciones. Esto, además de mejorar el rendimiento de la aplicación, nos ayudaría a actualizar el código y programar en la manera más correcta de React.

A lo largo de la refactorización, también nos encontramos con numerosas variables declaradas con el tipo `var` de JavaScript que según el estándar se deberían de evitar por su comportamiento dentro del código, por lo que fuimos cambiando las variables a la declaraciones actuales de `let` y `const`. Otro dato a mencionar fue el hecho de cambiar las extensiones de todos los ficheros de los componentes de React a ficheros JSX. Anteriormente, todos los componentes estaban escritos en formato JSX pero los ficheros estaban declarados como ficheros de JavaScript.

Aparte de estos cambios, también decidimos ordenar y organizar el código para un mejor entendimiento a la hora de desarrollar, estructurando mejor sobre todo la carpeta `src` del proyecto, separando componentes, *hooks*, servicios, ficheros CSS e imágenes.

3.5.1.2. Actualización de librerías

Debido al cambio de estructura del código de React y el paso del tiempo, muchas de las librerías que se utilizaban el proyecto del año pasado quedaron algo desactua-

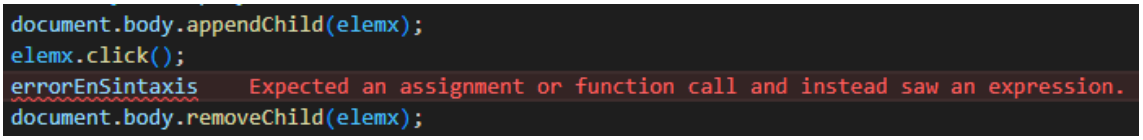

Figura 3.9: Ejemplo de *linter* de React

lizadas o obsoletas. Por ese motivo necesitábamos actualizar las librerías y eliminar las que no se utilizarán. Gracias a que cualquier proyecto de React trabaja sobre un entorno de Node.js, podemos actualizar todas las librerías que utilicemos de manera prácticamente instantánea ejecutando el comando de `npm update`, el cual actualizará todas las librerías que tengamos declaradas en el fichero `package.json`. En el caso que quisiéramos eliminar cualquier biblioteca, simplemente debemos ejecutar el comando `npm uninstall <nombre-del-paquete>`.

Aparte de estos cambios, queremos destacar el cambio de versión de la propia librería de React. La versión anteriormente utilizada por nuestros compañeros, React 17, tenía numerosos problemas de ejecución con la definición de componentes mediante funciones, por lo que tuvimos que realizar un cambio a la versión más actualizada, React 18, en la que cambiaba como se trataba el *renderizado* principal de la página.

3.5.1.3. Arreglo de errores y *linting*

Para el arreglo de errores generalizado durante la refactorización del código y nuestro futuro desarrollo en la aplicación, decidimos que lo mejor sería añadir *linters* a nuestro editor de texto, *Visual Studio Code*. Los *linters* son extensiones de *Visual Studio Code* que permiten mostrar errores en la sintaxis o dar consejos sobre cómo estructurar mejor el código bajo un estándar. En nuestro caso, añadimos *linters* tanto de React como de JavaScript para que nos ayudasen a encontrar errores más deprisa y evitar que cometiésemos errores innecesarios. En la figura 3.9 podemos ver una de las recomendaciones que nos hace el *linter* de React.

3.5.1.4. Configuración del entorno del desarrollo

Para facilitar el desarrollo a lo largo del proyecto pensamos que una buena configuración de nuestro entorno y editor de trabajo es esencial. Anteriormente, ya hemos hablado de la mejora con los *linters* para la identificación de errores y mejoras en nuestro código, pero también cabe destacar la configuración del entorno con nuestro nuevo compilador de React, *Vite*.

De la misma manera que en Django, React maneja una serie de configuraciones básicas para ejecutar su código. *Vite* permite, al igual que hacíamos con ***django-environ*** (ver sección 3.4.1), almacenar en un fichero tipo `.env` todas las variables de configuración necesarias, abstrayendo así todo en un simple fichero y mejorando la legibilidad del código en general.

```
export const Subjects = {
getAll: (_data = {}, config = {}) =>
    client.get('subject', config),
getTests: ({ idAsignatura }, config = {}) =>
    client.get(`subject/${idAsignatura}/cuestionarios`, config),
getQuestions: ({ idAsignatura }, config = {}) =>
    client.get(`subject/${idAsignatura}/preguntas`, config),
getFromStudentOrTeacher: (_data = {}, config = {}) =>
    client.get('subject/me', config),
enrollStudents: ({ alumnos, asignatura }, config = {}) =>
    client.post(`subject/${asignatura}/enroll`, { alumnos }, config),
deleteStudentsFromSubject: ({ alumnos, asignatura }, config = {}) =>
    client.post(`subject/${asignatura}/delete_enroll`, { alumnos }, config)
    ↪ ,
};

```

Figura 3.10: Fragmento del fichero de comunicación con la API

Además, *Visual Studio Code* permite configurar comandos personalizados para ejecutar dentro de nuestros proyectos. En nuestro caso hemos configurado una serie de comandos para arrancar la aplicación, arrancando tanto el *back-end* como el *front-end* de manera automática en sus respectivos puertos, iniciar *tests* de prueba y *depurar nuestra aplicación*. Todo estos comandos se encuentran especificados en el fichero *tasks.json*.

3.5.1.5. Mejoras generales

- **Abstracción de la API**: Anteriormente, las llamadas a la API estaban esparcidas por todos los componentes de la aplicación, donde por cada llamada se formaba la petición, repitiendo así muchísimo código y haciendo que el control y mantenimiento de todas las llamadas fuera complicado de gestionar. Por esta razón, primero, formamos con la nueva librería **axios** un método que creará la petición para evitar repetir lineas de códigos por cada llamada a la API y después, decidimos unificar en un fichero todas las llamadas para mejorar el control y la expansión de la API. En la figura 3.10 podemos ver como quedó parte del dicho fichero, donde se aprecian todas las llamadas relacionadas con las asignaturas.

- **Abstracción de rutas**: Anteriormente, también todas las rutas que hay en el lado del cliente se colocaban directamente en el código, y como buena práctica decidimos añadir un fichero de constantes que reuniera todas las rutas utilizadas y así hacer más legible el código.

- `React Router y ProtectedRoutes`: Como ya comentamos en la sección de librerías externas (ver sección 3.4.2), el uso de **react-router** ayudaba a man-

tener la arquitectura SPA de React aunque se cambiará de ruta. Esta librería ofrece un componente *Router* que engloba a todas las rutas de la API, donde a cada una se le asigna un componente especifico de React que se debe devolver cuando se acceda a esa ruta. El año pasado, nuestros compañeros no llegaron a aplicar esta librería correctamente, debido a que declararon un componente Router por cada ruta de la API y pensaban que tenían que estructurar dentro toda la página que querían devolver al usuario. Esto desembocó en un mal uso de esta librería y en un componente `App.js` demasiado extenso. Por lo que este año hemos conseguido implementarla correctamente, reduciendo enormemente el código del componente principal y hemos añadido que ciertas rutas sean exclusivas solo para ciertos tipos de usuario. Por ejemplo, si un estudiante tratara de acceder a la vista de crear cuestionarios introduciendo directamente la ruta en la barra de direcciones del navegador, no se le permitiría el acceso.

- `useFetch`, ***hook*** **para llamadas asíncronas**: Cuando el estado de un elemento de React depende de datos asíncronos se suele utilizar el *hook* `useEffect` de React para controlar cuándo debe de *renderizarse* otra vez el componente una vez actualizado dicho elemento. Como esto es algo recurrente dentro de nuestra aplicación, con el afán de reutilizar código decidimos crear un *hook* personalizado que tratara todas estas situaciones de igual manera.

- `useAuth`, ***hook*** **de autenticación**: Con el fin de centralizar toda la lógica relacionada con la autenticación de los usuarios, decidimos crear un *hook* personalizado que englobara todas las operaciones de autenticación y manejara la información del usuario que está *logueado* en ese momento.

3.5.2. *Back-end*

3.5.2.1. Refactorización de código

Lo primero que intentamos hacer al principio del proyecto, era comprender el código del proyecto tanto a nivel de *back-end* como *front-end*. A nivel de *back-end*, ya que nunca habíamos utilizado Django, entender todo el código al principio fue una tarea bastante dura. La organización del código era algo compleja y difícil de entender, con ficheros extensos y estructuras muy profundas. Por ello empezamos un proceso de refactorización de código en el que nuestro objetivo era fundamentalmente organizar el código de una manera que nosotros lo entendiéramos.

Primero de todo, empezamos organizando los nombres de las rutas de la API. Antes, se utilizaban nombres poco específicos que no dejaban claro qué hacía cada método. Después, empezamos a estructurar el código de una manera organizada, eliminando redundancias y haciendo más legible cada método. Además, todas las llamadas a la API estaban con métodos POST, cuando deberían ser GET y también los ajustamos.

Finalmente, pasamos a estudiar más en profundidad la librería de **djangorest-framework**. En esta librería explicaban el uso de algunas herramientas que nos

servirían a mejorar la estructura general de la API. Dentro de estos consejos se encontraban los conocidos como *viewsets*, que explicaremos en el apartado de mejoras generales (ver sección 3.5.2.5).

3.5.2.2. Actualización de librerías

De la misma manera que actualizamos las bibliotecas del *front-end* también nos enfocamos en poner al día todas las librerías utilizadas en el *back-end*. Para empezar, antes de actualizar todas las bibliotecas, decidimos implementar un entorno virtual de Python en nuestra aplicación. Los entornos virtuales permiten tener una instancia de Python especifica con una serie de dependencias instaladas asociadas al proyecto, sin obligar al cliente a tener que instalar todas estas en su propio equipo. Anteriormente, para poder trabajar sobre el proyecto teníamos que instalar todas las dependencias y librerías asociadas una a una en nuestros equipos, con el riesgo de tener conflictos con nuestras librerías propias instaladas y además haciendo este proceso tedioso y molesto para los desarrolladores. Con un entorno virtual evitamos esto y abstraemos así el entorno del proyecto de la configuración propia de nuestros equipos.

Para mantener el proyecto de manera más segura y estable, necesitábamos un gestor de dependencias para todas las librerías del proyecto y una herramienta que creará el entorno virtual. Por este motivo, al principio, utilizamos *pipenv*. *Pipenv* es un gestor de dependencias que trabaja con entornos virtuales. Permite crear entornos virtuales y a través de un fichero *Pipfile*, donde se guardan todas las dependencias del proyecto e instalar en el mismo entorno virtual todos las librerías especificadas. Durante parte del periodo de desarrollo, utilizamos este gestor hasta que más adelante decidimos pasarnos a otro gestor de dependencias llamado *Poetry*.

Poetry, del mismo modo que *pipenv*, es otro gestor de dependencias, pero en este caso permite almacenar todas las dependencias del proyecto con sus configuraciones en un solo fichero *.toml*. Según los estándares de Python más actuales, la gestión de un proyecto se debe mantener mediante este tipo de ficheros, por lo que para mantener nuestro proyecto lo más actualizado y correcto posible decidimos cambiar *pipenv* por *poetry*.

Por último, añadimos la actualización de Django a su versión más actualizada, Django 4.1. En el proyecto anterior, se utilizaba Django 3, por lo que temíamos que su actualización diera muchos problemas de compatibilidad. Aún así, no hubo muchas dificultades en este proceso.

3.5.2.3. Arreglo de errores y *linting*

Al igual que en el *front-end*, para el arreglo de errores decidimos utilizar los *linters* de Django y Python. Estos nos ayudarían a detectar errores básicos en la sintaxis de Python, como errores de espaciados, excepciones o fallos generales. Respecto a Django, también incluía consejos de cómo utilizar determinados elementos y dónde

```
# Shell Plus Django Imports
from django.core.cache import cache
from django.conf import settings
from django.contrib.auth import get_user_model
from django.db import transaction
from django.db.models import Avg, Case, Count, F, Max, Min, Prefetch, Q, Sum, When
from django.utils import timezone
from django.urls import reverse
from django.db.models import Exists, OuterRef, Subquery
# Shell Plus User Imports
from api.serializer import *
Python 3.11.2 (tags/v3.11.2:878ead1, Feb  7 2023, 16:38:35) [MSC v.1934 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 8.11.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Figura 3.11: Consola avanzada de Django

era mejor utilizarlos.

3.5.2.4. Configuración del entorno del desarrollo

Como ya explicamos en la sección de las librerías externas de *back-end* (ver sección 3.4.1), el uso de **django-environ** nos permitió almacenar todas las variables de configuración del proyecto en un solo fichero *.env* y poder utilizarlo en cualquier lado del proyecto facilitándonos enormemente el entendimiento de la configuración del proyecto al principio.

Además de esto, también configuramos una serie de extensiones de Django, con la librería **django-extensions**. La extensión que cabe destacar y más hemos utilizado ha sido la consola avanzada de Django. Esta consola permite ejecutar consultas en Python sobre cualquiera de los modelos de Django y ver qué resultados devuelven, permitiéndonos hacer pruebas y asegurarnos de nuestras decisiones. En la figura 3.11 podemos ver cómo hacemos uso de esta consola.

3.5.2.5. Mejoras generales

Dentro de las mejoras generales que hemos podido aplicar al *back-end* de la aplicación, está el uso de algunos elementos clave de la librería **djangorestframework** (ver sección 3.4.1).

- ***Serializadores***: Los *serializadores* son herramientas que ofrece la librería para convertir los objetos utilizados por Django, como pueden ser los *querysets* o los modelos, en tipos nativos de Python, como por ejemplo JSON o XML. Esto es clave para el tratamiento de datos en algunos elementos de esta librería y además nos permite devolver directamente información al *front-end* sin tener que transformar previamente los objetos obtenidos de la base de datos.

- ***Viewsets***: Generalmente, todas las rutas que se manejan en Django se deben colocar en el fichero de *views.py*, donde por cada ruta se escribe un método en el que se debe tratar la petición, aplicar cierta lógica de negocio y devolver una respuesta al *front-end*. Esta estructura, en proyectos tan amplios como este, conlleva a que el fichero *views.py* sea extremadamente largo. Para solucionar este problema hemos aplicado los *viewsets* de la librería **djangorestframework**.

  Los *viewsets* ofrecen una serie de rutas determinadas para un modelo de la base de datos especifico, ofreciendo rutas como por ejemplo, la creación de un objeto o el listado de todos los objetos de ese modelo. Además de las rutas que ofrecen, se pueden añadir más rutas con el decorador de `@action`, en el que escribimos un método como lo hacíamos anteriormente en el fichero *views.py*.

  Las ventajas de los *viewsets*, es que permiten desacoplar las rutas en varios ficheros y además ahorran código y trabajo con llamadas anteriormente configuradas.

- ***Managers*, cambio en el ORM**: Anteriormente, cuando queríamos recuperar información sobre cualquier modelo de nuestra aplicación, utilizábamos directamente el ORM de Django para filtrar y conseguir solamente la información que necesitábamos. Esto normalmente se define como una mala práctica dado que genera numerosa redundancia y empeora el mantenimiento del código. Por ello decidimos utilizar los *managers* de Django.

  Estas herramientas permiten generar una interfaz para las llamadas a la base de datos de los diferentes modelos de la aplicación. Debe haber un *manager* por cada modelo dentro del proyecto y en estos básicamente añadimos métodos que nos permitan recuperar cierta información de la base de datos. Un ejemplo de un *manager* es el que podemos ver en la figura 3.12.

- **Transacciones**: Conseguir un entorno transaccional en las operaciones de una aplicación software es algo de gran importancia para mantener la consistencia en los datos almacenados en la base de datos. Por eso nosotros, gracias a los mecanismos que ofrece Django para ello, hemos conseguido implementar transacciones en las llamadas a la API de la aplicación. Anteriormente, nuestros compañeros no utilizaron transacciones, provocando que cualquier fallo en cualquier operación resultase en el almacenado de datos erróneos en nuestra de base de datos. Por ende, hemos trabajado en implementar transacciones en las operaciones que podían desencadenar en el guardado de datos inconsistentes en nuestra aplicación. Para ello hemos hecho uso de la biblioteca `transaction` que ofrece Django, la cual nos permite marcar las operaciones que necesitan transacciones con el decorador `@transaction.atomic` y además ofrece un sistema de *savepoints* para volver a diferentes estados de la transacción en caso de errores.

```
class InstanciaPreguntaManager(models.Manager):
    def create_instancia(self, id_intento, id_pregunta, orden, id_seleccion
    ↪ , commit=False, **extra_fields):
        obj = self.model(intento_id=id_intento,
                         pregunta_id=id_pregunta,
                         orden=orden,
                         seleccion_id=id_seleccion,
                         **extra_fields)
        if commit:
            obj.save()
        return obj

    def get_by_intento_pregunta(self,
                                id_intento,
                                id_pregunta):
        return self.get_queryset()
                    .filter(
                            intento_id=id_intento,
                            pregunta_id=id_pregunta).first()
    def get_by_intento(self, id_intento):
        return self.get_queryset()
                    .filter(intento_id=id_intento)

```

Figura 3.12: Ejemplo de un *manager* de la aplicación

3.5.2.6. Documentación de la API

Un apartado fundamental en la creación de una API es su documentación. Con el objetivo de facilitar una documentación entendible para los lectores de nuestro trabajo y posibles futuros compañeros, utilizamos la librería **drf-spectacular**. Esta biblioteca permite redactar una documentación de API bajo el estándar de Open API 3, permitiéndonos añadir por cada llamada una descripción de la misma, los parámetros esperados y detalles sobre las posibles salidas o respuestas de cada petición. En la figura 3.13 podemos poder ver un pequeño ejemplo de cómo se implementa esta librería en el propio código.

Además, otra de las razones por la que utilizamos **drf-spectacular** es su buena integración con la herramienta *Swagger*. Esta herramienta, como ya comentamos en el capítulo de herramientas (ver capítulo 2), además de ofrecer una interfaz gráfica donde se pueden ver todas llamadas a la API, también permite realizar pruebas interactivas y ver cómo sería la comunicación entre *back-end* y *front-end*.

En el apéndice de la memoria adjuntaremos toda la documentación generada en el caso de que se desee leer (ver apéndice A).

3.5.2.7. Testing

Del mismo modo que la documentación es muy importante en el desarrollo software, también lo son las pruebas del mismo. Aunque no estaba dentro de nuestros objetivos iniciales, queríamos añadir algunos casos de prueba a la aplicación, sobre todo para probar que nuestra lógica de negocio era correcta y funcionaba como nosotros queríamos a lo largo de los diferentes *tests*. Asimismo, esto también ayudaría en un futuro a mantener mejor el código y también mejorar la compresión de la complicada lógica que a veces tienen nuestros métodos.

Con ello, lo primero que tuvimos que hacer es preparar nuestro entorno de desarrollo para las pruebas. *Visual Studio Code* lo pone fácil ofreciendo y identificando rápidamente las pruebas de nuestra aplicación una vez que lse introducen en el fichero *settings.json* del proyecto. En este fichero se identifican, aparte de otras configuraciones, dónde se encuentran los ficheros dedicados a las pruebas.

Ahora, para el desarrollo propio de las pruebas, utilizamos una clase de la librería de **djangorestframework.test**, `ApiTestCase`. Con esto podíamos declarar una clase que herede de `ApiTestCase` y dentro de esta nueva clase, tendríamos que declarar una función preparatoria llamada `setUp`, donde se inicialicen todas las variables que vayamos a utilizar en los *tests* y después ya escribiríamos todas las pruebas que quisiésemos separadas cada una en funciones de Python. El esquema general de los *tests* es el que se puede ver en la figura 3.14.

Finalmente, también añadimos la librería **coverage**, que nos permitía saber qué porcentaje del código se probaba una vez que se lanzaban los *tests* de nuestra aplicación, además de generar un fichero *coverage.xml* que contaba por cada fichero dentro de nuestro proyecto cuántas veces los *tests* pasaban por cada línea.

```
@extend_schema_view(
    list=extend_schema(
        summary="Lista de asignaturas",
        responses={
            200: OpenApiResponse(
                response={
                    "type": "object",
                    "properties": {"asignaturas": {"type": "array", "items
    ↪ ": {"type": "object", "properties": {"id": {"type": "string"}, "
    ↪ asignatura": {"type": "string"}}}}},
                },
            ),
        },
    ),
    cuestionarios=extend_schema(
        summary="Lista de cuestionarios de una asignatura",
        parameters=[
            OpenApiParameter(name="id", type=int, location=OpenApiParameter
    ↪ .PATH, description="Id de la asignatura"),
        ],
        responses={
            200: OpenApiResponse(
                response={
                    "type": "object",
                    "properties": {
                        "cuestionarios": {"type": "array", "items": {"type
    ↪ ": "object", "properties": {"id": {"type": "string"}, "titulo": {"
    ↪ type": "string"}}}},
                        "nombre": {"type": "string"},
                    },
                }
            )
        },
    ),

```

Figura 3.13: Ejemplo de implementación de documentación de API

```
1
2 class Question(APITestCase):
3     def setUp(self):
4         self.user = User.objects.create_user(email="test@root.com",
    ↪ first_name="root", last_name="root", password="root", role="teacher
    ↪ ")
5         self.client.force_authenticate(user=self.user)
6         self.asignatura = Asignatura.objects.create_asignaturas(
    ↪ nombreAsignatura="Test", commit=True)
7
8     def test_upload_question(self):
9         ...
10        ...
11        self.assertEqual(response_upload.status_code, status.HTTP_200_OK)
12        self.assertEqual(response.status_code, status.HTTP_200_OK)
13
```

Figura 3.14: Ejemplo de un fichero de pruebas

3.5.3. Nuevas funcionalidades extras

Además de todas las mejoras incluidas en el proyecto, también hemos añadido algunas funcionalidades nuevas que no estaban presentes en la aplicación del año anterior. Por eso, en esta sección vamos a hablar de ellas.

- **Listado de alumnos matriculados**: Anteriormente, en el momento en que un profesor quería matricular un alumno dentro de una asignatura, solo podía ver un listado de los alumnos que todavía **no** estaban matriculados. Ahora, en la vista donde antes aparecían los alumnos a matricularse en una asignatura, aparece un listado de todos los alumnos matriculados en la asignatura elegida.
- **Nuevo registro de usuarios en asignaturas**: Aunque antes sí se podía matricular alumnos en asignaturas desde la aplicación, este año nos hemos visto obligados a añadir una vista para esta funcionalidad, ya que la anterior vista que servía para matricular alumnos se ha sustituido con el listado total de matriculados en una asignatura.

  Para esta nueva vista hemos diseñado una ventana modal donde el registro se hará de igual manera que se hacía anteriormente: aparecerán en una tabla los alumnos sin matricular en la asignatura elegida y el profesor deberá añadir a los alumnos que él decida. En la figura 3.15, se muestra esta nueva ventana modal.
- **Baja de usuarios en asignaturas**: Además de poder ver los alumnos matriculados en una asignatura, ahora se podrá, una vez que se marque un alumno en el listado, dar de baja a uno o varios alumnos de la asignatura marcada.

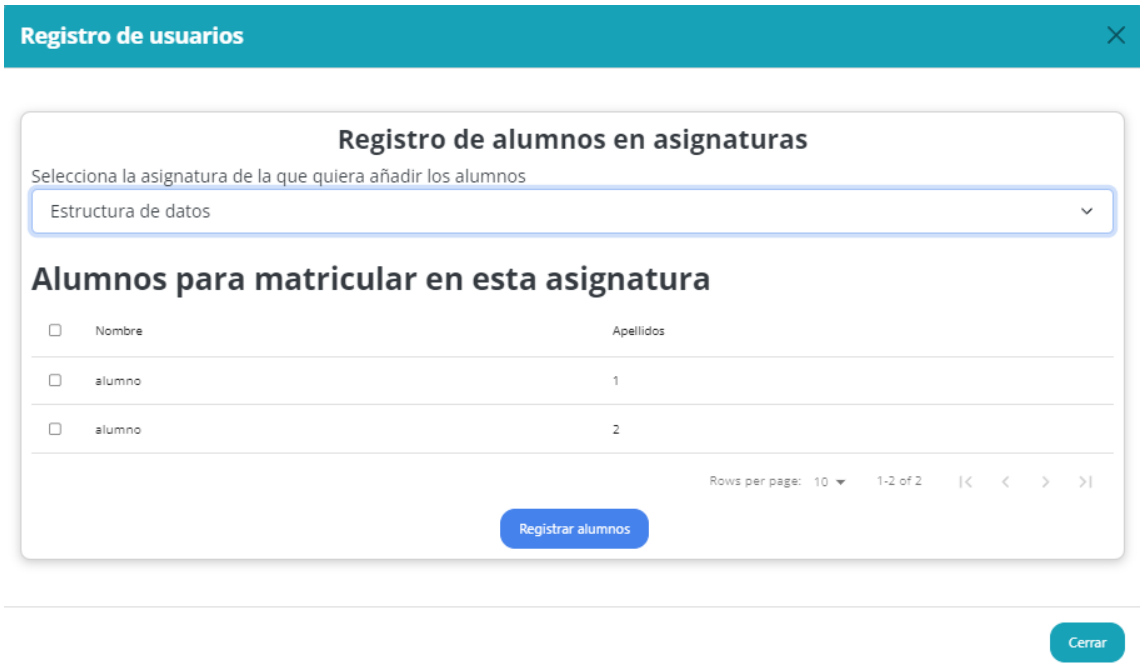

Figura 3.15: Nueva ventana modal para la matriculación de alumnos

Capítulo 4

Aleatorización de cuestionarios

En este capítulo vamos a explicar cómo a lo largo de este proyecto hemos implementado uno de los principales objetivos planteados para nuestro proyecto: la aleatorización de cuestionarios.

4.1. Objetivo principal / Introducción

La aleatorización de los cuestionarios es un objetivo muy amplio que abarca numerosos puntos a desarrollar. Principalmente, el objetivo inicial era ofrecer la posibilidad de la aleatorización de las preguntas de un cuestionario, es decir, en el caso que un profesor decidiese elaborar un cuestionario en el que sus preguntas se mostrarían de forma aleatoria, cada estudiante vería su cuestionario en el que las preguntas aparecen en un orden distinto al de los demás estudiantes. Poco a poco, esta idea se fue agrandando y fuimos extendiendo este objetivo a algo más complejo, en los que podemos identificar los siguientes objetivos:

- **Aleatorización de las opciones**: Qwizer cuenta con un banco de preguntas con diferentes tipos de pregunta. Dentro de los tipos que nos podemos encontrar, se encuentran las llamadas preguntas tipo *test*. En este tipo de preguntas se le ofrece al usuario un número de opciones especifico con diferentes respuestas ante una pregunta. Al tratar el tema de la aleatorización, decidimos ofrecer esta propiedad también a las opciones dentro de una pregunta de tipo *test*, por lo que el profesor tendría la opción de elegir si quiere que las opciones se barajen de manera aleatoria a los alumnos.
- **Fijación de preguntas y opciones**: La aleatorización de las preguntas o de las opciones de un cuestionario supone cambiar el orden relativo de las mismas. Sin embargo, a lo mejor los profesores en la creación de sus cuestionarios quieren mantener una pregunta o una opción en un lugar fijo. Por ejemplo, en las opciones de una pregunta tipo *test*, nos podemos encontrar la opción de `Ninguna de las anteriores`. En este caso, dado el significado de la misma opción, lo correcto sería colocarla en la última opción posible. Si el profesor aún así quiere que las opciones se dispongan de manera aleatoria, tendría que tener la posibilidad de mantener algunas opciones en una cierta posición fija.
- **Preguntas aleatorias**: Aparte de la aleatorización de las preguntas dentro de un cuestionario, el profesor también agradecería la posibilidad de, en un nú-

mero de pregunta especifico de su cuestionario, agregar una serie de preguntas a aparecer ante los usuarios. Por ejemplo, en un cuestionario un profesor decide que en la pregunta 3 no quiere que aparezca una pregunta especifica, sino que quiere que la aplicación elija entre un par de preguntas que él señale, es decir, siguiendo el ejemplo anterior y teniendo en nuestro banco de preguntas las preguntas 1,2,3 y 4, el profesor quiere que una determinada pregunta de su cuestionario, puedan aparecer la pregunta 1,2 o 3 del banco de preguntas. Esto es lo que nosotros hemos identificado como preguntas aleatorias.

4.2. Cambios en la base de datos

La incorporación de mecanismos de aleatorización en nuestros cuestionarios provocó grandes cambios en la estructura general de la base de datos, haciendo que la misma pasase por numerosas modificaciones para la buena adaptación a esta nueva característica. En esta sección iremos viendo la progresión de la base de datos a medida que íbamos cumpliendo los objetivos marcados.

4.2.1. Creación de entidad `Intento` e instancias

El añadido de la aleatorización de las preguntas a un cuestionario planteaba un grave problema ante nuestro modelo entidad-relación inicial. Anteriormente, las respuestas de los estudiantes a las preguntas de un cuestionario se almacenaban en una entidad llamada `Respuestas enviadas`. Cada vez que cualquier estudiante respondía una pregunta de un cuestionario, se guardaba en esta tabla la respuesta del mismo. Esta entidad no tenía ninguna relación con el cuestionario al que se contestaba y saber el orden en el que un cuestionario se había contestado y qué respuesta pertenecía a dicha pregunta era algo altamente complicado, por lo que por el afán de centralizar mejor la lógica propia de los cuestionarios y sus realizaciones, creamos la entidad `Intento`.

La entidad de intento almacenaría toda la información que se almacenaba antes entre varias entidades en una sola, es decir, se guardaría que **estudiante** respondió dicho intento sobre un **cuestionario** y qué **respuestas** había mandado ese estudiante ante las diferentes preguntas. Más adelante, decidiríamos que las respuestas de los estudiantes se podrían interpretar como instancias de preguntas que el usuario responde, por lo que con esto inició el concepto de las `Instancias` de preguntas.

Las instancias de preguntas almacenarían la **respuesta** de un usuario a una **pregunta** y el **orden** en la que apareció en un intento. Esto nos ayudaría enormemente después para el tratamiento de la aleatorización dentro de la aplicación, de cara, a recuperar el orden final de las preguntas de un cuestionario para un usuario.

Con los modelos de Django, creamos estas dos nuevas entidades de una manera muy sencilla y rápida. En el caso de las instancias tendríamos instancias para las preguntas tipo *test* y las tipo *text*. Por lo que el modelo final de las instancias

quedaría algo así:

```
class InstanciaPregunta(models.Model):
    objects = InstanciaPreguntaManager()
    intento = models.ForeignKey(Intento,
                                on_delete=models.CASCADE)
    seleccion = models.ForeignKey(SeleccionPregunta,
                                  on_delete=models.CASCADE)
    pregunta = models.ForeignKey(Pregunta,
                                on_delete=models.CASCADE)
    orden = models.PositiveSmallIntegerField()

    class Meta:
        ordering = ["pregunta"]

class InstanciaPreguntaTest(InstanciaPregunta):
    objects = InstanciaPreguntaTestManager()
    respuesta = models.ForeignKey(OpcionTest,
                                  on_delete=models.CASCADE
                                  ,null=True, blank=True)


class InstanciaPreguntaText(InstanciaPregunta):
    objects = InstanciaPreguntaTextManager()
    respuesta = models.CharField(null=True, blank=True,
                                  max_length=254,
                                  verbose_name="respuesta")
```

y el modelo de intento llegaría a ser algo así:

```
class Intento(models.Model):
    objects = IntentoManager()
    usuario = models.ForeignKey(User,
                                on_delete=models.CASCADE)
    cuestionario = models.ForeignKey(Cuestionario,
                                     on_delete=models.CASCADE)
    nota = models.DecimalField(default=0, max_digits=10,
                                decimal_places=2,
                                verbose_name="nota")
    hash = models.CharField(blank=True, max_length=254,
                                  verbose_name="hash")
    hash_offline = models.CharField(blank=True,max_length=254,
                                  verbose_name="hash_offline")
    fecha_inicio = models.DateTimeField(null=True,blank=False,
                                  verbose_name="fecha_inicio")
    fecha_fin = models.DateTimeField(null=True,blank=False,
                                  verbose_name="fecha_fin")

    class Estado(models.TextChoices):
        PENDIENTE = 'PEN', _('Pendiente')
        ENTREGADO = 'ENT', _('Entregado')
```

```
22
23      estado = models.CharField(
24          max_length=3,
25          choices=Estado.choices,
26          default=Estado.PENDIENTE,
27      )
```

Una vez tenemos un modelo entidad-relación más óptimo y preparado para esta nueva característica, ya podemos seguir con nuevas propiedades.

4.2.2. Aleatorización de preguntas

Inicialmente, nuestros compañeros del año pasado, mostraban las preguntas de un cuestionario de manera secuencial, es decir, según el orden en el que hubieran añadido el profesor las preguntas al cuestionario en sí. Para conseguir la aleatorización de las preguntas simplemente tuvimos que añadir un nuevo atributo en la entidad `Cuestionarios` que controlase si el profesor quiere que las preguntas aparezcan de manera aleatoria o no dentro de un cuestionario.

Como nos apoyamos en el ORM de Django, tuvimos que modificar el modelo de la clase `Cuestionario`, añadiéndole el atributo nuevo de aleatorización, quedando así un modelo como el siguiente:

```
1 class Cuestionario(models.Model):
2     objects = CuestionariosManager()
3     profesor = models.ForeignKey(User,
4                                  on_delete=models.CASCADE)
5     asignatura = models.ForeignKey("Asignatura",
6                                    on_delete=models.CASCADE)
7     ...
8     ...
9     aleatorizar = models.BooleanField(default=False)
```

Dentro de la aplicación, como podemos ver en la figura 4.1, hay un pequeño ejemplo de cómo un profesor podría modificar este atributo a su antojo.

En el caso en que el profesor decidiese añadir un cuestionario a través de un fichero YAML, simplemente tendría que añadir un pequeño campo de `aleatorizar` como se puede ver en la figura 4.2.

Cabe recordar que, gracias a nuestro modelo basado en **instancias** de las preguntas, podríamos guardar en qué posición apareció cada pregunta del cuestionario en cada intento, permitiéndonos con este sistema recuperar de manera sencilla el **orden** de todas las preguntas dentro de un intento.

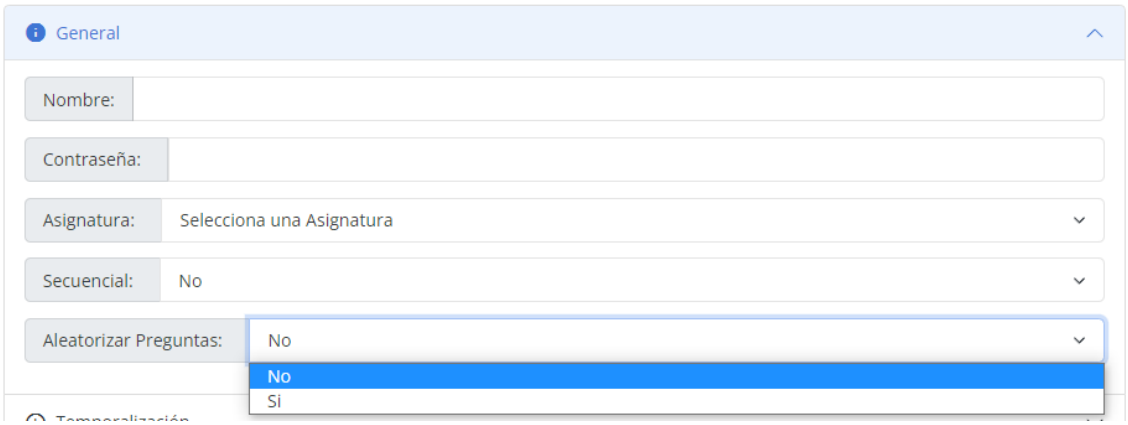

Figura 4.1: Modificación de aleatorización en creación de cuestionarios

```yaml
cuestionario:
  titulo: "Tema 8"
  password: "1234"
  asignatura: "Estructuras de datos"
  secuencial: 0 #0 es que no es secuencial, 1 es secuencial
  duracion: 10 #minutos que durara el test
  fecha_apertura: '21/02/20 11:00:00' #Formato esperado: yy:mm:dd hh:mm:ss
  fecha_cierre: '23/03/24 11:59:59' #Formato esperado: yy:mm:dd hh:mm:ss
  fecha_visible: '21/02/20 11:00:00' #Formato esperado: yy:mm:dd hh:mm:ss
  aleatorizar: True #Indica que se quiere aleatorizar el cuestionario

```

Figura 4.2: Aleatorización en cuestionarios vía YAML

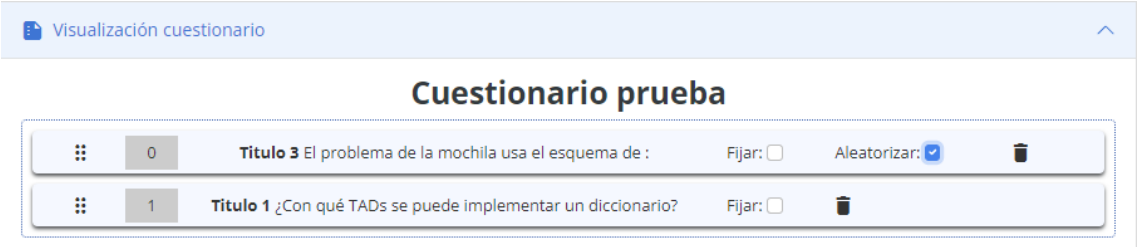

Figura 4.3: Elección de aleatorización de opciones

4.2.3. Aleatorización de opciones

La aleatorización se debe hacer también a nivel de opciones dentro de las preguntas tipo *test*. El profesor debería de poder elegir si las opciones de una pregunta *test* deben mostrarse de manera aleatoria dentro de un intento. Al igual que nos pasó con las preguntas dentro de los cuestionarios, nuestro modelo entidad-relación no estaba todavía lo suficientemente preparado como para almacenar si una pregunta dentro de un intento debería mostrar sus opciones de manera aleatoria y tampoco estaba estructurado para guardar el orden con el que se mostrarían las opciones a los estudiantes.

Para el primero de los problemas, añadimos un **nuevo atributo** en la clase que almacenaba las preguntas pertenecientes a un cuestionario, al igual que en la aleatorización de las preguntas, este nuevo atributo guardaría si la pregunta dentro del cuestionario debería mostrar sus opciones de manera aleatoria. A consecuencia de esta solución, el profesor podría marcar si las opciones de una pregunta tipo *test* se mostrarían de forma aleatoria. Como podemos ver en la figura 4.3 es muy fácil de controlar.

En el caso en que el profesor decida añadir un cuestionario a través de un fichero YAML, lo único que tendrá que hacer es añadir un nuevo atributo en las preguntas tipo *test*, como se puede ver en la figura 4.2, pero en vez de añadirlo a nivel de cuestionario, deberá añadirlo a nivel de pregunta.

Conforme a esto podríamos controlar perfectamente el primero de los problemas, pero para buscar una solución al segundo problema necesitamos pensar en cómo almacenar el distinto orden de las opciones en los diferentes intentos de los estudiantes. Para solucionar esto utilizamos de nuevo nuestra mentalidad de las **instancias** y con ello creamos así la clase de `InstanciaOpcionTest`. En este nuevo modelo de la base de datos se guardaría una referencia de la opción *test* que guardase esa nueva instancia y además el **orden** en el que apareció esta opción en el intento a la que esta asignada.

El nuevo modelo en Django está indicado en la Figura 4.4, donde se puede observar que guarda todos los datos anteriormente redactados:

```
    class InstanciaOpcionTest(models.Model):
        objects = InstanciaOpcionTestManager()
        instancia = models.ForeignKey(InstanciaPreguntaTest,
                                      on_delete=models.CASCADE)
        respuesta = models.ForeignKey(OpcionTest,
                                      on_delete=models.CASCADE,
                                      null=True, blank=True)
        orden = models.PositiveSmallIntegerField()

```

Figura 4.4: Modelo `InstanciaOpcionTest`

4.2.4. Fijación de preguntas y opciones

Como ya hablamos con anterioridad, la aleatorización presentaba el problema de preguntas o opciones de preguntas tipo *test* que el profesor quisiese que se quedarán fijadas en una posición concreta. Por ejemplo, en el caso de la opción de `Ninguna de las anteriores`, colocarla en una posición diferente a la última provocaría confusión en la propia pregunta. Por esta razón, para solucionar este pequeño contratiempo, decidimos almacenar un nuevo campo `fijar` en los modelos donde se guarda la información correspondiente a las preguntas pertenecientes a un cuestionario, `PerteneceCuestionario` y al modelo donde se guarda la información sobre las opciones de las preguntas de este tipo.

Gracias a esta implementación, una vez se recuperase una pregunta o una opción de un intento se podría comprobar si la misma debería ir fijada a la posición en la que se encuentra o no. En el apartado de la aplicación, en la vista de la creación de un cuestionario, para la fijación de las preguntas, el profesor simplemente, una vez elegido el orden en el que una pregunta debe aparecer, marcará si quiere que se fije a esa posición como podemos ver en la figura 4.3 al lado del botón de aleatorizar.

En el caso en que el profesor decida subir un cuestionario mediante un fichero YAML, una vez que ordene las preguntas en el orden canónico que quiere que aparezcan, podrá añadir un nuevo campo `fijar` como podemos ver en las preguntas de un cuestionario en la figura 4.5.

En relación a la fijación de las opciones de una pregunta, dado que nuestra aplicación solo permite la incorporación de preguntas vía YAML, el protocolo para ello es muy parecido a como se muestra en la figura anterior. Simplemente se debe colocar el campo `fijar` en la opción que se quiera mantener en una posición concreta.

4.3. Preguntas aleatorias

Anteriormente, introducimos el concepto de **preguntas aleatorias** dentro de nuestra aplicación Qwizer. Esta funcionalidad no estaba prevista dentro de nues-

```yaml
preguntas:
  - tipo: "test"
    pregunta: "El problema de la mochila usa el esquema de :"
    titulo: "Titulo 3"
    opciones:
      - op: "Divide y Venceras"
        fijar: False
      - op: "Vuelta atras"
        fijar: True

```

Figura 4.5: Fijación en cuestionarios vía YAML

tros objetivos iniciales, pero, como nos comentó nuestro tutor, esta característica es muy habitual en aplicaciones relacionados con cuestionarios, como es el ejemplo de nuestra mayor inspiración, Moodle. La posibilidad de crear una pregunta dentro de nuestros cuestionarios donde a cada usuario le apareciese una pregunta elegida de un conjunto de preguntas seleccionadas específicamente para que esa pregunta, era algo motivador y, desde nuestro punto de vista, algo complicado de realizar.

Desde un inicio, veíamos complicado la integración de esta funcionalidad dentro de nuestro sistema, dado a varios factores, entre ellos:

- **Cambios en la base de datos**: Desde un principio, nos dio la sensación de que este nuevo añadido en la aplicación podría provocar otro gran cambio en la base de datos. El hecho de tener que cambiar la estructura general después de haber ajustado todo a un modelo mucho más limpio y útil para nuestras funcionalidades principales, nos provocó, a primera vista, una sensación de rechazo.
- **Impacto en el código**: Pensando más en el código y su organización, el añadido de las preguntas aleatorias provocaría seguramente muchos cambios en la lógica principal de muchas de nuestras rutas en la API, provocándonos una sensación de miedo a generar errores en algunas partes concretas del código.

Aún así con estos factores en mente y gracias a la motivación de nuestro tutor, decidimos añadir esta funcionalidad para aumentar el valor de nuestra aplicación y hacerla mucho más competente en su ámbito.

4.3.1. Nuevo modelo, `SelecciónPregunta`

Primero de todo, la implementación de esta funcionalidad provocaba, como ya habíamos estimado, un gran cambio en la base de datos. La manera en la que se almacenaban las preguntas pertenecientes a un cuestionario solo permitía almacenar preguntas del propio banco de preguntas, por lo que añadir a un cuestionario

una pregunta que fuera de otro tipo, en este caso una pregunta aleatoria, era algo imposible por como estaba estructurado el modelo entidad-relación.

Para ello tuvimos que cambiar la mentalidad con la que almacenaríamos las preguntas pertenecientes a un cuestionario, llegando a la conclusión y a la ejecución de un nuevo modelo `SelecciónPregunta`. Este nuevo modelo cambiaría totalmente la perspectiva de la incorporación de preguntas a un cuestionario. Ahora, en vez de añadir directamente preguntas del banco se añadirían **selecciones de pregunta**. Estas **selecciones** pueden ser directamente **preguntas del banco de preguntas**, o bien **otro tipo de preguntas**, como pueden ser las preguntas aleatorias.

Este modelo nos permitiría añadir, en un futuro, otros tipos de preguntas que no tengan ninguna relación con el banco de preguntas o solo parcialmente, ampliando así la capacidad de la aplicación para escalar de manera mucho más sencilla. Como podemos apreciar en la figura 3.2 se puede ver cómo partíamos de un modelo más escueto y pasamos a un modelo más complejo con la incorporación del modelo `SeleccionPregunta` como podemos ver en la figura 3.3.

4.3.2. Uso dentro de la aplicación

Esta nueva funcionalidad está disponible para los cuestionarios que se suban vía YAML. En el momento que los profesores quieran añadir una pregunta aleatoria, tendrán que seguir los siguientes pasos para conseguirlo:

1. En el apartado de las preguntas del fichero YAML, añadir una pregunta nueva con el nuevo tipo de pregunta, `aleatoria`.
2. A continuación, añadir el campo de *preguntas_elegidas* después del campo `tipo` antes explicado. En este nuevo campo se introducirá el subconjunto de preguntas del banco que queremos que puedan aparecer. Para referenciar las preguntas del banco, recordar utilizar el campo `ref` y escribir el título de la pregunta a referenciar.
3. Al igual que las preguntas regulares del banco, debemos añadir una puntuación positiva y una puntuación negativa, con los campos *punt_positiva* y *punt_negativa* respectivamente.

La Figura 4.6. muestra un ejemplo de inclusión de una pregunta aleatoria.

Además, dentro de los ejemplos de cuestionarios ya creados, se encuentra uno con un *test* con una única pregunta aleatoria para que sirva de ayuda a los profesores que comiencen a utilizar la aplicación. Este cuestionario se llama *test2.yml*.

```yaml
preguntas:
  - tipo: "aleatoria"
    preguntas_elegidas:
      - ref: "Titulo temporal2"
      - ref: "Titulo temporal1"
    punt_positiva: 10.0    #puntos que suma
    punt_negativa: 0.0 #puntos que resta

```

Figura 4.6: Ejemplo de pregunta aleatoria en YAML

Capítulo 5

Introducción de Markdown y fórmulas matemáticas en enunciados

En este capítulo abordamos unos de los principales desafíos en este trabajo: la introducción de texto en formato Markdown, imágenes y fórmulas en los enunciados de las preguntas. Esta parte del trabajo conllevaba una fase previa de investigación para determinar las bibliotecas más adecuadas para la consecución de este objetivo.

5.1. Librerías utilizadas

Desde el inicio de nuestro proyecto, consideramos el uso de MathJax para implantar las formulas matemáticas en nuestra aplicación. Esta biblioteca nos pareció adecuada debido a que permite la visualización de fórmulas matemáticas en navegadores web mediante LaTeX, y además, nuestro tutor nos la recomendó. Sin embargo, nos encontramos con un obstáculo al tener que renderizar no solo fórmulas matemáticas, sino también Markdown. Esto implicó la necesidad de distinguir dentro del propio texto en Markdown qué partes correspondían a fórmulas matemáticas.

Después de investigar y probar diversas bibliotecas para nuestro proyecto, nos encontramos con tres que resultaron ser la solución: `react-markdown`, `remark-math` y `rehype-katex`.

- **react-markdown** es una biblioteca de React que nos permitió renderizar texto en formato Markdown en nuestra aplicación web. Con esta biblioteca, pudimos mostrar el contenido enriquecido con textos en negritas, cursivas, listas, enlaces, imágenes, entre otros elementos.

- **remark-math**, por su parte, es una biblioteca de procesamiento de Markdown que nos permitió renderizar fórmulas matemáticas dentro del texto escrito en formato Markdown.

- **rehype-katex** es una biblioteca que nos permitió procesar el HTML generado por `remark-math` y renderizar las fórmulas matemáticas de una manera más legible y estilizada. En otras palabras, `rehype-katex` mejoró la apariencia visual de las fórmulas matemáticas en nuestra aplicación web.

En resumen, `react-markdown` nos permitió mostrar texto en formato Markdown, `remark-math` nos permitió renderizar fórmulas matemáticas dentro del texto y `rehype-katex` mejoró la apariencia visual de las fórmulas matemáticas. Juntas, estas bibliotecas nos permitieron mostrar de manera clara y atractiva el contenido matemático y escrito en nuestra aplicación web.

5.2. Gestión de imágenes

A pesar del uso de las bibliotecas mencionadas en la sección anterior, todavía quedaba un pequeño detalle por implementar, y era el hecho de añadir imágenes dentro de los textos Markdown de la aplicación con el fin de mejorar los textos enriquecidos de la misma. A priori, nos pareció algo sencillo de implementar, pero al final, resultó en la parte más complicada de este objetivo.

5.2.1. Subida de imágenes

Primero de todo, empezamos con la implementación de la subida de imágenes en el *front-end* de la aplicación. Dado que el formato Markdown se aplica solo a los enunciados y/o a las opciones de las preguntas tipo *test*, nos centramos en la lógica de la subida de preguntas.

Para conseguir este objetivo, simplemente incluimos un nuevo selector de archivos en el que los profesores deberían añadir las imágenes que deseen que se almacenen con las preguntas.

Posteriormente, en el momento que el profesor pulse el botón de `Subir preguntas`, el contenido de ambos formularios pasarán a tratarse en el *back-end*.

En la figura 5.1 podemos ver el resultado de estos cambios en la vista principal de la subida de preguntas.

5.2.2. Tratamiento de imágenes en Django

Con las imágenes ya subidas al servidor, la información de las preguntas e imágenes llega al *back-end* de la aplicación donde esta información debe ser validada y procesada. Por consiguiente, teníamos que encontrar una manera en la que el código pudiese validar si las imágenes procedentes del *front-end* eran las mismas que había en los enunciados u opciones de las preguntas tipo *test* en el documento YAML.

Para conseguir este objetivo, diseñamos una expresión regular que a partir de un texto Markdown, filtrase su contenido y devolviese los nombres de las imágenes que se debían de guardar. Posteriormente, el código comprobaría si el profesor había subido las imágenes detectadas anteriormente. En caso negativo, se le devolvería un error al usuario especificándole las imágenes restantes por subir y en caso afirmativo, tanto las preguntas como las imágenes se almacenarían en nuestro servidor.

Con respecto al almacenamiento de imágenes, hemos utilizado el sistema de ficheros de nuestro servidor para guardar, en una carpeta llamada `media`, todas las imágenes provinientes de la subida de preguntas. Para el acceso al control del sistema de ficheros hemos hecho uso la clase de Django, `FileSystemStorage`. Esta clase, una vez configurada, ofrece una serie de métodos que nos han sido útiles para el almacenado de archivos. Entre ellos podemos encontrar:

- `save`: Permite almacenar un archivo dentro de nuestro sistema de ficheros, guardándolo preferiblemente con el nombre del fichero que se le pase por parámetro. En el caso que exista un archivo con el mismo nombre, el sistema de ficheros modificará el archivo para almacenarlo con un nombre único.

  En nuestro caso, esto nos puede pasar si varios profesores utilizan la misma imagen en sus enunciados, por lo que esta funcionalidad nos ayudaría a diferenciar qué imágenes pertenecen a cada uno. Por esta razón, algunas veces los profesores observarán que cuando colocan una imagen en sus enunciados, el nombre de la misma ha sido sustituido. Esto se debe a que se ha reemplazado por el nuevo nombre que el sistema de ficheros le dio al archivo.

- `path`: Permite recuperar la ruta de almacenamiento de un archivo dado un nombre. Gracias a esto podemos abrir y leer ficheros con la propia función de Python, `open` que devuelve los datos de un fichero a través de una ruta. Más adelante, esto nos ayudará a recuperar imágenes guardadas para después mostrarlas en el *front-end* de la aplicación.

- `delete`: Esta función posibilita eliminar un archivo del sistema de ficheros dado un nombre pasado por parámetro.

5.2.3. Recuperación de imágenes

Finalmente, nos falta abordar el tema de la recuperación de las imágenes dentro de la aplicación. En la sección anterior, ya hemos adelantado que con el método `path`, podemos encontrar las rutas de las imágenes dentro de nuestro sistema de ficheros y cómo con el método `open` podemos recuperar toda la información de la misma.

Como ya comentamos en la sección de mejoras (ver sección 3.5.2.5), tenemos que recordar que para devolver objetos al *front-end* utilizamos los *serializadores* que transforman los objetos de la base de datos a formatos sencillos como JSON para facilitar la comunicación entre ambos lados. Debido a esto, tuvimos que hacer un cambio en los *serializadores* de las preguntas y opciones tipo *test* para que pudieran devolver las imágenes correctamente.

Primero, identificamos todas las imágenes que había dentro de los textos con la misma expresión regular que utilizamos para la subida de preguntas y después, codificamos las imágenes en *base64* para poder recuperar las imágenes correctamente. Como podemos ver en la figura 5.2, este sería el nuevo código que aplicaríamos a nuestros serializadores.

Como podemos observar en esa misma figura, encontramos un pequeño inconveniente con las imágenes de tipo `svg` al pasar el contenido de la imagen a *base64*. El atributo `src` de la etiqueta `img` acepta los `MIME types` que estan recogidos en el registro de **IANA**. En nuestro caso necesitábamos `svg+xml` y por eso tuvimos que añadir esa parte a la extensión del fichero.

5.2.4. Consecuencias

Tras la implementación de imágenes en los cuestionarios, nos encontramos con un importante problema: el espacio ocupado por las imágenes de los cuestionarios descargados en *localStorage*.

Desde el inicio del proyecto, nos percatamos de que guardar los cuestionarios en *localStorage* no era una solución adecuada, debido a que los cuestionarios son objetos que se convierten en cadenas de texto para ser almacenados, y posteriormente se convierten de nuevo en objetos para su uso. Después de buscar alternativas, encontramos *IndexedDB*, una *API* de bajo nivel del lado del cliente que permite almacenar grandes cantidades de datos.

En un principio, descartamos esta opción porque nos centramos en otras funcionalidades, pero más adelante, cuando surgió el problema mencionado, retomamos esta opción y descubrimos una biblioteca llamada *localForage*[1] que simplificó mucho el código ya que funcionaba de manera similar a *localStorage*, pero con promesas.

Finalmente, adaptamos nuestro código para que funcionara con promesas y pudimos prescindir de la transformación del objeto a texto. Con esto, logramos implementar la funcionalidad requerida de manera eficiente y sin los problemas del espacio generados por las limitaciones del *localStorage*.

5.3. Resultados

Por último, queremos enseñar como se refleja todo lo explicado en este capítulo dentro de nuestra aplicación.

5.3.1. Subida de preguntas con Markdown

Empezaremos enseñando como, ahora, los profesores son capaces de incluir Markdown en sus enunciados e incluso en las opciones de las preguntas tipo *test*, sin olvidar que también pueden escribir fórmulas matemáticas (ver figura 5.3). Además cabe recalcar la posibilidad de incluir imágenes dentro de las preguntas, como podemos ver en el ejemplo de la figura 5.4.

[1] https://github.com/localForage/localForage

5.3.2. Visualización de preguntas con Markdown

Los profesores pueden publicar cuestionarios que contengan preguntas con texto Markdown, que en el momento que los estudiantes realicen podrán ver renderizadas correctamente. Las preguntas pueden contener texto Markdown con fórmulas matemáticas (Figura 5.5) e incluso imágenes (Figura 5.6). Asimismo, ya que hemos tratado también de hacer nuestra aplicación lo más adaptable a distintos dispositivos, las imágenes también se auto-escalan para que se puedan apreciar bien en un dispositivo móvil (ver figura 5.7).

Figura 5.1: Nueva vista para subir preguntas

```
def get_opcion(self,obj):
    fs = FileSystemStorage()
    opcion = obj.opcion
    imagenes_devolver = re.findall(
        r"!\[(.*?)\]\(([\w\/\-\:\._]+?)\)", obj.opcion)
    for img in imagenes_devolver:
        name = str(img[1])
        format = name.split(".")[1]
        if format == "svg":
            format +="+xml"
        path = fs.path(name)
        with open(path, "rb") as image_file:
            encoded_string = base64.b64encode(image_file.read())
                                        .decode('utf-8')
            imagen_base64 = '%s'
            % (format, encoded_string)
            opcion = opcion.replace(name, imagen_base64)
    return opcion

```

Figura 5.2: Cambios en *serializadores para recuperación de imágenes*

```
- tipo: "test"
    pregunta: "La formula de la energia es"
    titulo: "Titulo Markdown Test 2"
    opciones:
        - op: "$E = mc^2$"
          fijar: False
        - op: "$E = mc^3$"
          fijar: False
        - op: "$E = mc * 2$"
          fijar: False
        - op: "Ninguna de las anteriores"
          fijar: True
    op_correcta: 0 #id de opcion correcta

```

Figura 5.3: Pregunta con texto Markdown y fórmulas matemáticas

```
preguntas:
- tipo: "test"
    pregunta: "El simbolo del framework de React es:"
    titulo: "Titulo Markdown SVG"
    opciones:
        - op: "No se"
          fijar: False
          - op: "![Image](react.svg)"
          fijar: False
          - op: "![Image](html.svg)"
          fijar: False
          - op: "Ninguna de las anteriores"
          fijar: True
    op_correcta: 1 #id de opcion correcta


```

Figura 5.4: Pregunta con imágenes

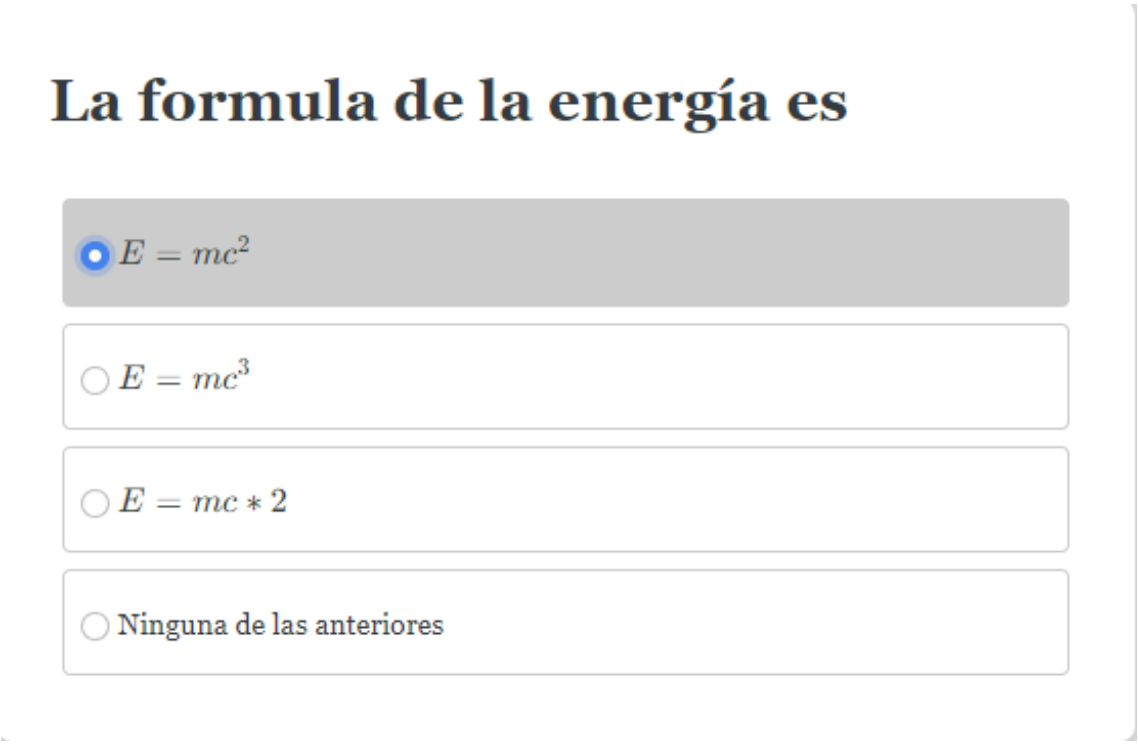

Figura 5.5: Visualización de preguntas con fórmulas

Figura 5.6: Visualización de pregunta con imágenes

Figura 5.7: Visualización de pregunta con imágenes en dispositivo móvil

Capítulo 6

Integración con Moodle

Uno de los principales objetivos del Trabajo de Fin de Grado era facilitar la migración de preguntas y calificaciones entre Moodle y Qwizer. Nos propusimos alcanzar los siguientes objetivos:

- **Importar preguntas desde Qwizer** generadas a partir de un banco de preguntas de Moodle. Esto permitiría aprovechar el contenido existente en Moodle y utilizarlo en Qwizer.
- **Exportar las calificaciones desde Qwizer** para que puedan ser importadas en Moodle. De esta manera, se podría mantener un seguimiento coherente de las calificaciones de los alumnos, utilizando Qwizer para evaluar su desempeño y luego reflejar esas notas en Moodle.

Estos objetivos estaban orientados a facilitar la interoperabilidad entre las plataformas y brindar a los profesores la capacidad de aprovechar el contenido y las calificaciones existentes en Moodle en el entorno de Qwizer, y viceversa.

6.1. Migración de las preguntas de Moodle

En esta sección, abordaremos el proceso de migración de preguntas desde Moodle a nuestro sistema. Para llevar a cabo esta migración, utilizamos el formato de Moodle XML, que nos brinda la capacidad de importar y exportar preguntas en Moodle.

6.1.1. Formato Moodle XML

El formato de Moodle XML[1] permite importar y exportar preguntas desde Moodle. En nuestro caso solo necesitábamos saber como estaban definidas las preguntas de respuesta corta (Figura 6.1) y las preguntas de opción múltiple (Figura 6.2), para insertarlas en nuestra base de datos posteriormente.

Teniendo conocimiento de la estructura del fichero, extrajimos selectivamente los datos necesarios para nuestra aplicación haciendo uso de la librería `ElementTree` como se detalla en la sección 6.1.2.

[1] https://docs.moodle.org/all/es/Formato_Moodle_XML

```
<question type="shortanswer">
    <answer fraction="100">
        <text>La respuesta correcta</text>
        <feedback><text>Correcto</text></feedback>
    </answer>
</question>

```

Figura 6.1: Pregunta corta

```
<question type="multichoice">
   <name>
      <text>Title</text>
   </name>
   <questiontext format="html">
       <text><![CDATA[<p>Choose one:</p>]]></text>
   </questiontext>
   <answer fraction="100" format="html">
      <text><![CDATA[<p>option</p>]]</text>
   </answer>
   <answer fraction="0" format="html">
      <text><![CDATA[<p>option</p>]]></text>
   </answer>
   <answer fraction="0" format="html">
      <text><![CDATA[<p>option</p>]]></text>
   </answer>
   <answer fraction="0" format="html">
      <text><![CDATA[<p>option</p>]]></text>
   </answer>
</question>

```

Figura 6.2: Pregunta de opción múltiple

6.1.2. ElementTree

Para procesar el archivo XML de Moodle, empleamos la biblioteca *ElementTree*. Esta herramienta nos brinda la capacidad de recorrer el XML de manera jerárquica, como si estuviéramos navegando por un árbol, y nos permite buscar elementos específicos utilizando expresiones *XPath*.

Gracias a estas funcionalidades, logramos realizar una manipulación eficiente y precisa de los datos que necesitábamos extraer del archivo de Moodle. Una vez que procesamos el archivo XML, creamos las preguntas en la base de datos, pero antes tuvimos que realizar otro tratamiento en el texto de las preguntas. Como se puede observar en la figura 6.2, las preguntas están en formato HTML, por lo que tuvimos que utilizar una expresión regular adicional para eliminar todas las etiquetas innecesarias.

Además, es importante mencionar que este proceso de manipulación de datos se lleva a cabo de manera similar al caso del YAML, con la diferencia de que en este caso los datos necesarios para las preguntas se extraen del archivo XML en lugar del archivo YAML.

6.2. Exportar notas

La exportación de calificaciones de los estudiantes involucra un proceso que consta de **varias etapas**. Para comprender mejor este proceso, es necesario establecer el contexto en el que se utiliza Moodle. **Moodle** ofrece la capacidad de **importar calificaciones en formato CSV** (valores separados por comas). Esto permite a los profesores transferir y registrar las calificaciones de los estudiantes de manera eficiente.

Para esto Moodle puede **generar una tabla que contiene un listado de estudiantes**, donde cada estudiante tiene su información personal como correo electrónico, DNI, entre otros datos. **Una de las columnas** de esta tabla corresponde a las calificaciones, y es en esta columna donde el profesor debe **completar las notas** correspondientes a cada estudiante.

Una vez que el profesor ha completado las calificaciones en la columna correspondiente de la tabla CSV, el siguiente paso es cargar el archivo CSV con las calificaciones en Moodle. Al hacer esto, Moodle incorpora la información de calificaciones proporcionada en el archivo CSV a su base de datos.

Para poder rellenar adecuadamente las notas de los estudiantes en nuestra aplicación de forma sencilla y automatizada se siguen los siguientes pasos:

- En Qwizer se carga el listado de estudiantes en formato CSV de Moodle, como se muestra en la Figura 6.3. Este archivo contiene la lista de estudiantes junto con la columna de calificaciones que se desea completar, como se ilustra en el ejemplo de la Figura 6.4.

Figura 6.3: Subida de la lista de estudiantes

- En Qwizer, se selecciona la columna correspondiente a las notas a rellenar y se indica la asignatura a la que pertenecen, como se muestra en la Figura 6.5. Esto permite a Qwizer identificar correctamente la columna de calificaciones en el archivo CSV, ya que puede haber varias.
- A continuación, se elige el cuestionario pertinente en Qwizer, que se utilizará para asignar las calificaciones a los estudiantes correspondientes, como se indica en la Figura 6.6. Cabe aclarar que para identificar a que alumno pertenece cada nota usamos la columna de *Dirección de correo*, que es la que se usará para identificar unívocamente a cada alumno en nuestra aplicación.
- Por último, se realiza la descarga del archivo final, que incluye la columna de calificaciones debidamente rellenada. Este archivo final puede ser cargado nuevamente en Moodle para que las calificaciones se incorporen en la base de datos de Moodle.

De esta manera, nuestra aplicación Qwizer facilita el proceso de importación de notas, permitiendo a los profesores completar las calificaciones en un listado de alumnos exportado desde Moodle para su posterior carga en la misma plataforma.

Nombre	Apellidos	DNI	Direccion de correo	Item de calificacion
root	root	1234567Q	root@root.com	
x	x	2222222Q	x@x.com	

Figura 6.4: Lista de estudiantes generada por Moodle

Figura 6.5: Selección de la columna y la asignatura

Figura 6.6: Selección del cuestionario

Capítulo 7

Creación de contenedores Docker

El despliegue de una aplicación es una etapa fundamental en cualquier proyecto de software, ya que permite poner en funcionamiento y hacer accesible la aplicación a los usuarios. Es durante esta fase donde se lleva a cabo la configuración de los recursos necesarios para ejecutar la aplicación de manera eficiente y segura.

Al revisar el proyecto del año pasado, notamos que la parte de despliegue no había sido abordada. Conscientes de su importancia, decidimos dedicar tiempo y esfuerzo este año para implementar adecuadamente esta etapa.

Optamos por utilizar Docker, una tecnología de contenedores, para facilitar el proceso de despliegue. Docker nos permite crear un entorno aislado y reproducible para nuestra aplicación, junto con todas las dependencias necesarias. Esto simplifica la configuración y asegura la consistencia del entorno de ejecución en diferentes plataformas.

7.1. Despliegue con Docker Compose

Hicimos uso de Docker Compose, una herramienta que nos permitió definir y gestionar múltiples contenedores de Docker como un conjunto de servicios interconectados. Con Docker Compose, pudimos configurar fácilmente todos los componentes necesarios para el despliegue de nuestra aplicación en un solo archivo YAML.

En este archivo de configuración, especificamos los servicios que componen nuestra aplicación, junto con sus dependencias y configuraciones correspondientes. Además, pudimos definir redes virtuales para facilitar la comunicación entre los diferentes servicios. El fichero de configuración esta detallado en la Figura 7.1.

Además de los servicios previamente mencionados, también incorporamos una base de datos *PostgreSQL* y una versión de desarrollo tanto para el *backend* como para el *frontend*. Si bien estos servicios no están diseñados para el despliegue en producción, resultaron útiles para el proceso de desarrollo, permitiéndonos trabajar de manera eficiente en la implementación de funcionalidades y en la detección de errores.

7.2. Servidor web con Nginx

Uno de los componentes clave en nuestro despliegue fue Nginx, un servidor web de alto rendimiento. Utilizamos Nginx como proxy inverso para enrutar las solicitudes entrantes a los servicios apropiados, como el *backend* de Django y el *frontend* de React. La configuración de este servidor está detallada en la Figura 7.2.

7.3. Seguridad con HTTPS

En nuestro despliegue, nos preocupamos por garantizar la seguridad de la comunicación entre los usuarios y nuestra aplicación. Por ello, implementamos HTTPS (Hypertext Transfer Protocol Secure) para cifrar la conexión y proteger la integridad de los datos transmitidos.

Para habilitar HTTPS, obtuvimos un certificado SSL (Secure Sockets Layer) auto firmado para realizar pruebas, que posteriormente se reemplazará con un certificado firmado por una autoridad de certificación. Configuramos Nginx para que utilizara este certificado y habilitara la comunicación segura a través del protocolo HTTPS.

```
version: '3.9'
services:
  nginx:
    image: nginx:latest
    restart: unless-stopped
    ports:
      - 80:80
      - 443:443
    depends_on:
      - front
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf
      - ./nginx/ssl/site.crt:/etc/ssl/certs/site.crt
      - ./nginx/ssl/site.key:/etc/ssl/private/site.key
      - back:/static
      - front:/dist

  back:
    build:
      context: ../
      dockerfile: ./.docker/back/Dockerfile
    command: sh -c "./start.sh"
    depends_on:
      db:
        condition: service_healthy
    environment:
      - DEBUG=False
      - DATABASE_HOST=db
    volumes:
      - back:/app/staticfiles

  front:
    build:
      context: ../
      dockerfile: ./.docker/front/Dockerfile
    environment:
      - REACT_APP_API_URL=/api
      - REACT_PORT=3000
      - REACT_HOST=0.0.0.0
    volumes:
      - front:/app/dist

volumes:
  pg_data:
  back:
  front:

```

Figura 7.1: Fichero de configuración docker-compose.yml

```
http{
  access_log /var/log/nginx/qwizer.access.log;
  error_log /var/log/nginx/qwizer.error.log;
  server_tokens off;

  server {
      include /etc/nginx/mime.types;
      server_name localhost;

      listen 80;
      listen 443 ssl http2;

      ssl_certificate /etc/ssl/certs/site.crt;
      ssl_certificate_key /etc/ssl/private/site.key;

      location / {
          root /dist/;
          try_files $uri /index.html;
      }

      location /static {
          root /;
          #autoindex on;
      }

      location /api {
          proxy_pass http://back:8000/api;
      }

      location /admin {
          proxy_pass http://back:8000/admin;
      }

      location /swagger {
          proxy_pass http://back:8000/swagger;
      }
  }
}

```

Figura 7.2: Configuración de Nginx

Capítulo 8

Conclusiones y trabajo futuro

En esta sección, expondremos nuestras conclusiones una vez que ha finalizado el desarrollo del proyecto. Abordaremos temas como pueden ser: el nivel de cumplimiento de los objetivos principales, las dificultades encontradas durante el desarrollo del proyecto y las posibles mejoras que podrían implementarse en el futuro.

8.1. Objetivos alcanzados

Durante el transcurso del proyecto, se lograron alcanzar todos los objetivos planteados inicialmente (sección 1.2):

- **Aleatorización de cuestionarios**: Se implementó con éxito la funcionalidad de aleatorización tanto a nivel de preguntas como de opciones dentro de los cuestionarios. Además, se amplió la variedad de preguntas con la adición de las preguntas de selección aleatoria (una pregunta al azar de una serie de preguntas elegidas). Ahora, los profesores pueden ofrecer cuestionarios únicos a sus estudiantes lo que ayuda a evitar la copia entre ellos.

- **Adaptación de aplicación a dispositivos móviles**: se consiguió ofrecer un diseño adaptable a nuestra aplicación, en la que se trabajó arduamente para poder adecuar su interfaz a todas las pantallas y mejorar la experiencia de usuario para que fuese cómoda e intuitiva.

- **Inclusión de Markdown y fórmulas matemáticas**: Se logró la implementación exitosa de Markdown, lo que permite a los usuarios enriquecer el contenido con formatos como negrita, cursiva, imágenes, entre otros. Además, se incorporó la capacidad de redactar enunciados y soluciones con fórmulas matemáticas, lo que amplía las posibilidades para la creación de preguntas más complejas y detalladas.

- **Integración con Moodle**: Se logró una integración exitosa de Qwizer con los formatos utilizados en la plataforma Moodle. Ahora los usuarios pueden importar preguntas del banco de preguntas de Moodle, lo que facilita la reutilización y la migración del contenido existente. Asimismo, se implementó la funcionalidad de exportar las notas de los cuestionarios realizados en Qwizer a Moodle, brindando una gestión más sencilla y centralizada de las calificaciones.

Por otra parte, conseguimos completar otros objetivos, los cuales fueron apareciendo a medida que el proyecto avanzaba. Entre ellos podemos encontrar los siguientes:

- **Cambios en el diseño principal**: Inicialmente, no teníamos pensado cambiar el diseño de las vistas de nuestros compañeros, pero, ligado al objetivo del diseño adaptable, nos vimos obligados a realizar algunos cambios sobre casi todas las vistas iniciales para mejorar la experiencia general de nuestros usuarios. Incluso llegamos a introducir nuevas funcionalidades de las que previamente Qwizer no disponía (sección 3.5.3).
- **Documentación de la API**: Como ya explicamos en la parte de mejoras (ver sección 3.5.2), una buena documentación de una API nos parece algo esencial ya que sin ella se dificulta enormemente entender cada llamada y el porqué de la misma. Al principio, nosotros nos encontramos en esta situación y por el bien de unos posibles futuros compañeros o simplemente alguien que quiera entender nuestro trabajo, hemos desarrollado una documentación extensa e interactiva sobre cada llamada de la API, facilitando su compresión a ojos de cualquiera.
- ***Testing***: Al igual que una buena documentación, cualquier desarrollo de software implica una etapa de *testeo*, donde se hagan numerosas pruebas que comprueben si el código desarrollado funciona como lo esperado. Por este motivo, desarrollamos un apartado de *testing* para la parte *back-end* de la aplicación, en la que se probaba la lógica de las llamadas a la API.
- **Despliegue de aplicación**: Finalmente, si queríamos probar nuestra aplicación en un entorno de producción teníamos que explorar la posibilidad de prepararla para su despliegue, por lo que configuramos y adaptamos nuestro entorno de desarrollo para desplegar la aplicación en cualquier momento. De hecho en el capítulo 7 hemos desarrollado más en detalle la solución a este objetivo.

Además también conseguimos completar parte de los objetivos que tenían nuestros compañeros planificados para trabajo a futuro. Entre ellos conseguimos implementar:

1. La realización de cuestionarios de manera secuencial, donde ahora los profesores pueden elegir si un cuestionario se debe hacer seguidamente, es decir, sin poder volver a la pregunta anterior una vez se pase a la siguiente.
2. Resolución de más de un cuestionario a la vez de manera *offline*. Anteriormente, solo se podía resolver un cuestionario si el estudiante perdía la conexión. En cambio, ahora gracias a la nueva gestión del almacenamiento de cuestionarios, se puede realizar más de un cuestionario de manera *offline*.
3. Mejora de interfaz de la creación de cuestionarios y, como ya hemos comentado en los objetivos principales, adaptación de interfaz a dispositivos móviles.

8.2. Dificultades encontradas

A lo largo del desarrollo del proyecto, se encontraron algunas dificultades que afectaron el proceso de implementación de los objetivos:

- **Código heredado**: Uno de los mayores desafíos fue lidiar con código heredado del proyecto anterior. El uso de código preexistente presentó dificultades para comprender su funcionamiento y adaptarlo a los nuevos requisitos, desembocando en numerosas semanas de refactorización y mejoras al código inicial. Por esta razón, el periodo de adaptación y familiarización con el proyecto se extendió más de lo normal, impidiendo cumplir algunos objetivos secundarios planteados.

- **Desconocimiento de las tecnologías**: Cabe destacar que ambos integrantes del equipo partíamos sin una base de las tecnologías utilizadas dentro del proyecto, por lo que la utilización de estas nuevas tecnologías y herramientas durante el desarrollo del proyecto implicó un período de aprendizaje y familiarización. Tanto Django como React eran *frameworks* que presentaban mecanismos complicados de entender y asimilar al principio. Por ello podemos agregar los principales problemas encontrados durante el desarrollo con ambos marcos de trabajo:

  - **Django**: Una de las librerías que se ha utilizado más a fondo este año es la llamada *djangorestframework*. El uso de la misma está explicado en la sección de 3.5. La ampliación en el uso de esta librería acarreó numerosos problemas dado que el uso de *serializadores*, *viewsets* y otros mecanismos de la misma fueron algo complicados de implementar en el código heredado recibido. Esto, sumado a la poca compresión que teníamos de Django en las primeras etapas de desarrollo nos provocaron grandes complicaciones al inicio del proyecto, que saldamos gracias a la documentación oficial tanto de Django como de la librería.

  - **React**: Aparte del desconocimiento general de React, el hecho de afrontar componentes con tanta lógica resultó algo abrumador inicialmente. Tras varias semanas de desarrollo, este sentimiento de presión se fue reduciendo e incluso acabó en numerosas mejoras en cada uno de los componentes.

- **Tiempo**: El factor temporal fue otro desafío significativo. Gran parte del desarrollo fue adecuar y mejorar el código anterior, y eso dio lugar a un menor tiempo para desarrollar las nuevas funcionalidades. Aunque se cumplieran los objetivos iniciales marcados para el trabajo, una etapa de comprobación y *testeo* en un entorno real hubiera ayudado al proyecto a llegar a un nivel de refinamiento más elevado. Todo esto sin contar la poca disponibilidad en algunos momentos del proyecto debido a la presión del curso y en momentos puntuales, de los exámenes.

8.3. Trabajo futuro

Si bien los objetivos planteados se han logrado con éxito, consideramos que el proyecto aún tiene margen de mejora. Para ello, proponemos las siguientes modificaciones:

- **Accesibilidad**: Aunque hemos adecuado la mayor parte de la aplicación para todos los dispositivos, creemos que todavía se podría ampliar la accesibilidad de nuestra aplicación permitiendo que todo tipo de usuarios, independientemente de sus capacidades físicas o cognitivas, pueda hacer uso de Qwizer. El uso de técnicas de accesibilidad como variación de colores, cambios en los temas, descripciones con textos de ayuda, uso de tipografías sencillas y botones y paneles grandes y fáciles de localizar ayudarían a ajustar Qwizer a un ámbito más inclusivo.
- **Testing en el *front-end***: Como ya explicamos anteriormente, uno de los objetivos secundarios cumplidos fue el hecho de documentar y testear el *back-end* de nuestra aplicación. Esto ha concluido en numerosos beneficios a lo largo del desarrollo de la aplicación. En cambio, uno de los trabajos que nos hubiera gustado realizar y por falta de tiempo no se ha podido completar es el hecho de realización de pruebas de la interfaz y de los componentes principales del *front-end* de nuestra aplicación. En numerosas ocasiones, hemos tenido que comprobar el correcto funcionamiento de la lógica de los componentes, en los que a veces provocaba llamadas a la API y consecuentemente, cambios en el estado de la base de datos. Este tedioso proceso nos hizo plantearnos realizar una serie de pruebas a las vistas más esenciales de la interfaz, permitiéndonos probar esta lógica y realizar las pruebas bajo una base de datos preparada para los *tests*. Además, este tipo de pruebas encamina el código por un camino más sostenible y estable, por los que futuros desarrollos se verían agradecidos por las mismas.
- **Interfaz propia para móviles**: A lo largo del desarrollo del proyecto, ha habido numerosas ocasiones en los que nos hemos replanteado hacer una interfaz propia para los dispositivos móviles, porque aunque hayamos dedicado tiempo a conseguir un diseño adaptable de nuestra aplicación, siempre quedan algunos componentes o algunos fragmentos de las vistas que quedan ligeramente fuera del estilo de una aplicación móvil. Para esto estudiamos *Ionic*, un *framework* centrado en el desarrollo de interfaces de usuarios en dispositivos móviles, el cual, dado que su sintaxis es muy parecida a nuestro principal *framework* de estilado, *Bootstrap*, nos permitiría haber diseñado una interfaz única y exclusiva para los dispositivos móviles. Al final, terminamos abandonando la idea debido a que era un objetivo muy complicado de completar en el poco tiempo que nos quedaba, pero en un futuro, esta idea podría darle a Qwizer un valor fundamental para su implementación dentro de las asignaturas de la facultad.
- **Más tipo de preguntas**: Qwizer es una aplicación basada en la realización de cuestionarios y uno de los apartados en los que todavía cojea es su pequeño

catalogo de preguntas. Aunque este año hemos dedicado parte del tiempo a la creación de un nuevo tipo de pregunta (las preguntas de selección aleatoria) pensamos que podrían añadirse muchos más tipos de preguntas, como por ejemplo, preguntas de relacionar, preguntas de rellenar huecos, etc... Además, gracias a la nueva estructura de la base de datos, añadir estos nuevos tipos de pregunta sería algo sencillo de realizar. Por tal motivo, pensamos que esta nueva incorporación de preguntas podría darle otra imagen a Qwizer y impulsarla a un entorno más llamativo.

- **Importar más tipos de preguntas**: En relación a la importación de preguntas, también tenemos numerosos frentes abiertos que podríamos abarcar, como puede ser el hecho de importar otro tipo de preguntas de Moodle, ya que ahora mismo, solo se pueden importar preguntas de respuesta corta y preguntas de multiopción. Al mismo tiempo, nos hemos replanteado la idea de importar preguntas con otros formatos. Actualmente, Qwizer solo permite importar preguntas con formato *XML-Moodle* centrando toda esta funcionalidad en un solo formato. Por lo que, en resumen, importar otros tipos de preguntas de Moodle y aumentar la cantidad de formatos para importar preguntas, nos parece un objetivo que se podría plantear para futuro.

- **Usar API de moodle. Mejorar la conectividad con Moodle**: Ligado a la exportación de las calificaciones de Qwizer a Moodle, creemos que este objetivo se puede mejorar haciendo más cómoda la interoperabilidad entre Qwizer y Moodle. Para ello, pensamos en que sería mejor utilizar la propia API de Moodle para ampliar las posibilidades de integración con nuestra aplicación.

Capítulo 9

Contribuciones personales

En este capítulo vamos a hablar de nuestras contribuciones por separado al proyecto. Ambos hemos trabajado de manera síncrona, tocando aspectos tanto del *front-end* como del *back-end*, por lo que en nuestras explicaciones algunas veces se combinarán tareas en las que ambos hemos trabajado. Aún así, cabe remarcar que ambos hemos trabajado juntos en los siguientes apartados:

- **Refactorización del código**
- **Actualización de librerías**
- **Nuevo diseño de la base de datos**: Los principales cambios en la estructura general de la base de datos fueron desarrollados entre los dos.
- **Corrección de errores**: Ambos hemos trabajado en corregir los errores que iban surgiendo tanto en el *front-end* como en el *back-end*.

Vicentiu Tiberius Roman

Durante el desarrollo del Trabajo de Fin de Grado he realizado las siguientes aportaciones:

- **Actualizaciones**: Me encargué de actualizar React, Django y todas sus dependencias a las últimas versiones estables. Además elimine múltiples dependencias innecesarias que no se usaban en el proyecto.
- **Preparación del entorno de desarrollo**: Dado que íbamos a estar numerosos meses desarrollando este proyecto, queríamos tener un entorno de desarrollo preparado y cómodo para trabajar. Por esta razón, preparé nuestro entorno y editor de trabajo añadiendo lo siguiente:
  - *Linters* para mejorar la calidad del código y mantener la consistencia del estilo en todo el proyecto.
  - *Scripts* para el iniciar, detener y depurar React y Django.
  - Extensiones y configuraciones para facilitar el desarrollo.
- **Gestión segura de secretos mediante archivos .env**: Identifiqué un importante problema en el proyecto relacionado con la exposición de contraseñas en el repositorio de GitHub. Para abordar esta preocupación, realicé la separación de todos los secretos en archivos individuales, los cuales se leen posteriormente para cargar su contenido. Además, proporcioné una plantilla de estos archivos para que cada desarrollador conozca las variables necesarias, pero sin la necesidad de subirlas al repositorio. Esto garantiza una gestión segura de los secretos en el proyecto además de permitir cambiar configuraciones del proyecto sin tocar el código y únicamente cambiando las variables de entorno. En la Figura 3.7 podemos ver un ejemplo de los ficheros de configuración que gestioné.
- **Docker**: Para facilitar el despliegue, utilicé Docker para el empaquetado y la distribución de las aplicaciones React y Django junto con el servidor web Nginx. Realice los ajustes necesarios para desplegar Django correctamente, ya que no estaba preparado para esa tarea y me encargué de añadir certificados SSL a la aplicación. Además de este proceso, también ayudé a José Luis a configurar todo para que pudiera utilizarlo correctamente, ya que él desarrolla en Windows y la configuración con Docker es diferente que en Linux.
- **Vite**: Al ver la gran cantidad de advertencias que salían al iniciar la aplicación de React, decidí probar Vite, una alternativa que resultó ser mucho más rápida y además facilitó el desarrollo al refrescar las páginas sin eliminar el estado anterior.
- **Refactorización de componentes**: Dentro de la refactorización de los componentes de React, además de traducir algunos componentes de clases a funciones yo me encargué de arreglar y reescribir la lógica de los componentes

eliminando numerosos errores en su lógica, adaptando las variables al estándar más actual de JavaScript, ajustando los componentes a la sintaxis más reciente de React y arreglando el manejo del estado de los componentes que se habían hecho visibles a causa del cambio de versión de React, al no seguir las recomendaciones sobre cómo tratar el estado en versiones anteriores.

- **Rutas en la aplicación**: Previamente, el manejo de las rutas era errático. No se utilizaba `React Router`, una de las soluciones más populares para integrar las rutas en React y se realizaba de forma manual. Esto implicaba lo siguiente:

  - No tener rutas reales ya que la *URL* de la barra de navegación no cambiaba en ningún momento.
  - El punto de entrada a la aplicación (`App.js`) era de una extensión inasumible al usar condiciones anidadas para saber en qué ruta se encuentra en cada momento.
  - Todo el estado de la aplicación estaba concentrado en `App.js`, es decir, los componentes no eran independientes y tenían su estado en sus antecesores.

  Todo esto implicó un gran trabajo para reescribir casi todos los componentes, separando su lógica y haciéndolos más legibles y escalables, además de implementar de cero la gestión de los permisos para cada ruta.

- **Abstracción de API**: Realicé una completa abstracción de la API en el *front-end* de la aplicación. En lugar de tener las llamadas dispersas por varios componentes, centralicé la comunicación entre el *back-end* y el *front-end* en un solo archivo. Para lograr esto, primero configuré nuestro cliente *axios* para incluir el token de usuario en cada llamada, en el archivo `client.js`. Luego, estructuré todas las llamadas a la API en el archivo `API.js`. Esta implementación simplificó enormemente el desarrollo futuro, ya que cualquier cambio en la API solo requería modificar un único archivo.

- **Creación de *hooks* personalizados**: He creado también los *hooks* personalizados que explicamos en la sección de 3.5.1. De esta manera evité la repetición del código y una mejor gestión de las llamadas a la API.

- **Cuestionarios**: Me encargué de corregir todos los errores relacionados con la realización y revisión de cuestionarios, además de implementar las siguientes mejoras:

  - Permitir la realización y descarga de múltiples cuestionarios de manera simultánea.
  - Agregar la posibilidad de eliminar cuestionarios descargados.
  - Paso a `IndexedDB` para evitar la transformación de los cuestionarios a cadenas de texto, además de evitar los problemas con el límite de almacenamiento del `localStorage`.

- **Interfaz de la aplicación**: A lo largo del proyecto, me he encargado de realizar múltiples cambios para adaptar la aplicación a dispositivos móviles, además de crear los modales para la gestión de los errores en toda la aplicación.
- **Markdown**: Con respecto al Markdown, yo me encargué de la investigación de las librerías que nos podían ayudar a la implementación de esta funcionalidad e hice un ejemplo de como se usarían y después fue José Luis el que se encargó de implementarlo dentro de la aplicación.
- **Gestión de dependencias en Django**: También me encargué de mejorar la gestión de dependencias en Django, ya que anteriormente la única manera de saber qué dependencias tenía el proyecto era ir añadiendo dependencias según salían errores hasta que no faltara ninguna. Por este motivo me decidí finalmente por usar **Poetry** lo que me permitió crear un único fichero `pyproject.toml` donde están definidas todas las dependencias del proyecto y sus configuraciones, permitiendo así instalar las mismas en un entorno aislado para evitar conflictos con otras dependencias del equipo.
- **Viewsets, Managers y Serializadores**: Al ir aprendiendo Django, me di cuenta de que no se estaban usando todas sus capacidades, así que decidí reescribir las vistas a viewsets y crear varios serializadores para la aplicación, además de crear managers para todos los modelos, permitiendo una refactorización más sencilla en el futuro.
- **Documentación de la API**: Una de las cosas de la que me encargué personalmente fue de cómo documentar la API. Desde el principio del proyecto, estuve investigando cómo podía documentar la aplicación para facilitarnos el entendimiento de la misma tanto a mí como a mi compañero. Para esta tarea, encontré una librería denominada `drf-spectacular`, que por detrás utilizaba Swagger para generar la documentación de la API. Me encargué de documentar llamada a llamada, explicando tanto las entradas, salidas y operaciones esperadas. Finalmente el resultado fue una documentación muy extensa y elaborada que cualquier desarrollador que prosiga con el proyecto puede disfrutar.
- ***Testing* de API**: En relación al *testing* de la API, asumí la responsabilidad de investigar las mejores prácticas para realizar pruebas en nuestra aplicación, añadiendo una opción para ver la cobertura del *testing*, para poder observar qué porcentaje del código se había testado.
- **Aleatorización, *back-end***: En el tema de la aleatorización, mi rol fue sobre todo la preparación del *back-end* para la aleatorización de cuestionarios y las opciones de las preguntas tipo *test*. Me centré en añadir a los modelos los atributos necesarios para la aleatorización, preparar la lógica para devolver y almacenar cuestionarios aleatorios y modificar la lógica de subida de cuestionarios para almacenar los nuevos datos de los cuestionarios.
- **Preguntas de selección aleatoria**: Respecto al nuevo tipo de preguntas, me encargué sobre todo de ayudar a José Luis a entender toda la nueva lógica que había realizado para la aleatorización de los cuestionarios y las opciones de las

preguntas tipo *test* que he explicado anteriormente. Una vez que terminó de implementarlas le ayudé con algunos errores que habían quedado pendientes y terminamos entre los dos esa funcionalidad.

- **Integración con Moodle**: Mientras mi compañero trabajaba en el nuevo tipo de preguntas, me dediqué a avanzar en nuestro siguiente objetivo, la integración con Moodle. Investigué sobre el formato *XML-Moodle* y sobre qué librerías nos podrían ayudar para implementar estas funcionalidades. Durante la investigación empecé a realizar una serie de pruebas con la librería *ElementTree* y conseguí el objetivo de importar preguntas de Moodle. Más adelante, tras una reunión con nuestro tutor, nos explicó más en detalle que debíamos permitir exportar calificaciones desde Qwizer a Moodle y decidí también implementarlo, ya que había estado investigando en ese tema.

José Luis Bartha de las Peñas

Durante el desarrollo del Trabajo de Fin de Grado he realizado las siguientes aportaciones:

- **Refactorización de componentes**: Como ya hemos comentado, el año pasado nuestros compañeros implementaron todos sus componentes de React mediante clases y este año por recomendación de los propios desarrolladores de React y por su versión más actual reescribimos todos los componentes a una implementación mediante funciones. En este caso, yo me encargué más de refactorizar de clases a funciones, donde entre ellos cambié:
  - `LoginComponent`
  - `TarjetaAsignatura`
  - `UploadQuestions`
  - `UploadFile`
  - `IndexContainer`
  - `QuestionNav`
  - `AvailableOffline`
  - `QrContainer`
  - `BancoPreguntas`
  - `CuestionarioPassword`
  - `CrearCuestionario`
  - `CuestionariosContainer`
  - `RegisterContainer`
  - `RevisionNotasContainer`
- **Diseño adaptable de la aplicación**: Conseguir un diseño adaptable fue una tarea que nos dividimos entre Tiberius y yo. En mi caso, me encargué más de modificar los componentes que no estuvieran adaptados a todos los dispositivos, utilizando siempre nuestro *framework* de estilado, Bootstrap. Me centré sobre todo en adaptar las siguientes secciones de la aplicación:
  - **Creación de cuestionarios**
  - **Subida de preguntas y cuestionarios**
  - **Banco de preguntas**
  - **Realización y revisión de cuestionarios**
  - **Componentes de representación asignatura y cuestionarios**: Adapté los componentes donde se ve la información de las asignaturas y cuestionarios.
- **Ajuste rutas de la API**: Una de las primeras cosas que quisimos cambiar nada más comenzar el proyecto fue la nomenclatura general de las llamadas. Normalmente, una buena práctica a la hora de crear las llamadas a la API, es tener una nomenclatura explicativa en cada ruta para que se entienda la funcionalidad de la misma. En cambio, el año pasado todas las rutas eran algo complicadas de entender. Por esta razón, me encargué de darle a cada ruta un nombre y una estructura que permitiese entenderla a cualquier persona.

- **Creación de nuevas funcionalidades y diseños**: A lo largo del proyecto hemos añadido nuevas funcionalidades como las que explicamos en la sección 3.5.3 e incluso hemos pensado en rediseñar algunas de las vistas ya existentes dentro de la aplicación, ya que eran algo toscas de utilizar o simplemente queríamos darle un toque personal y utilizar un poco de estilado propio. Por esto mismo, me encargué personalmente de implementar estas nuevas funcionalidades y además de rediseñar componentes enteros como por ejemplo, la creación y subida de cuestionarios.

- **Adaptación a nuevos modelos**: En la etapa de refactorización, decidimos cambiar el modelo entidad-relación heredado entre Tiberius y yo, produciéndose cambios en los nombres, relaciones y creación de nuevas entidades. Esto desencadenó que todos los modelos utilizados en el *back-end* de la aplicación quedarán desactualizados, derivando en una refactorización a gran escala de todos los modelos de la aplicación y sus respectivas referencias a lo largo del código. En este caso yo me encargué de gran parte de esta refactorización con ayuda de Tiberius también.

- ***Testing* de API**: Respecto al *testing* de la aplicación, tras la investigación y explicación de Tiberius, me dediqué a realizar las pruebas respectivas sobre las rutas que tenían relación con las preguntas y asignaturas. Para conseguir esto, generé las pruebas para todas las llamadas sobre estas entidades en los ficheros de `test-subject.py` y `test-question.py`.

- **Aleatorización, *front-end***: En relación con la aleatorización, mi papel fue el de implementar esta nueva funcionalidad en el *front-end* de la aplicación. Mi trabajo se dividió en las siguientes partes:

  - Incluir en la vista de creación de cuestionarios una manera para que los profesores pudieran elegir si las preguntas de sus cuestionarios y/o las opciones de una pregunta tipo *test* apareciesen en un orden aleatorio.
  - Incluir la posibilidad para fijar preguntas a una posición especifica de un cuestionario y fijar una opción a una posición especifica en la subida de una pregunta tipo *test*.
  - Ajustar los ficheros *YAML* de ejemplo para que todos utilicen la nueva sintaxis, incluyendo los atributos de aleatorización y fijación.

- **Preguntas de selección aleatoria**: Después de los cambios realizados en la estructura de la base de datos para las preguntas de selección aleatoria, decidí encargarme personalmente de implementarlo dentro de nuestra aplicación. Tras comprender cómo funcionaba bien toda la lógica de la aleatorización que había hecho Tiberius en el *back-end*, realicé primero la lógica para poder almacenar este nuevo tipo de pregunta en la base de datos, cambiando los modelos de Django y ajustando la lógica de creación de cuestionarios e intentos para que soportará esta nueva pregunta. Finalmente, comprobé la correcta creación de los cuestionarios con este nuevo tipo de pregunta creando un nuevo fichero de ejemplo *YAML*. El fichero que creé de prueba se llama *test2.yml* y se ha

mantenido en la carpeta de ejemplos para los usuarios que quieran probar la aplicación.

- **Markdown**: Con respecto al Markdown, yo me encargué de la implementación de esta nueva funcionalidad en la aplicación. Dado que ya almacenábamos en una cadena de caracteres los enunciados y las opciones de las preguntas tipo *test*, la inclusión de fórmulas matemáticas y estilo enriquecido en los textos fue algo trivial de hacer gracias a las bibliotecas que utilizamos. Sin embargo, la subida de imágenes en texto Markdown, fue lo más complicado de hacer, donde tuve que configurar la clase de Django, *FileSystemStorage*, que permitía gestionar el sistema de ficheros del proyecto, cambiar la lógica de la subida de preguntas para que soportará imágenes y además cambiar los *serializadores* de `Preguntas y OpcionesTest` para poder devolver las imágenes almacenadas correctamente en los enunciados y en las opciones de las preguntas. Además de esto también generé un fichero de ejemplo con preguntas con Markdown, tanto con fórmulas como con imágenes para los usuarios que quieran utilizar la aplicación. El fichero se encuentra en la carpeta de ejemplos con el nombre de *questions2.yml*.

Introduction

In this document we will talk about the development of the expansion and improvement of a progressive application to take online questionnaires: Qwizer. We will begin by presenting what led us to follow this project, what objectives we have had throughout the project and finally we will describe our working plan throughout the academic year.

Background and motivation

Last year we found ourselves in the situation of choosing a project that will really motivate us to develop our last project within the faculty. In addition, as students of the Computer Science Faculty, we wanted a project that would really impact its evolution and allow us to help improve its services.

Manuel told us about a possible improvement and extension of a progressive web application based on the completion of tests without an internet connection: Qwizer. This application was aimed at users like us, students who throughout our studies within the faculty have had to complete numerous questionnaires, without adding that halfway through our path into our degree, a global pandemic broke out due to COVID-19 and this type of format for the evaluation of our subjects was something routinary.

Initially, the proposal already caught our attention, but we also identified a series of causes that led us to progress and continue this project. Among them we can name the following:

- **New technologies**: This application **used technologies which neither of us were familiar with**. Both React and Django were two technologies that we were unaware of, but we knew that they would be essential for our personal and professional development. Besides that, the fact of being an application that works without an Internet connection, allows us to get involved in the world of **progressive applications**, a market that is glowing today, so it is very interesting to develop and discover the possibilities of these applications.
- **Practical use in the future**: Another important point that made us decide on this project was the **practical use** that we saw for the application. Under good development and the addition of a minimum of necessary functionalities, we saw that this project could become useful. Throughout our studies, we have worked in numerous projects with the sole purpose of being qualified

and focused on a simple work delivery. Subsequently, they are forgotten or abandoned due to lack of time or motivation to follow them. On the other hand, with this project we saw that if we were able to meet the objectives we could **make a decent application that our colleagues could use in the future**.

- **Educational innovation project of the UCM (INNOVA projects)**: Finally, in addition to the other reasons, Manuel informed us, as we have already mentioned, that this project was destined to continue being developed by some teachers of the faculty. In fact, they had proposed an innovation project for the faculty, so that the application could be used in the laboratories of some subjects to evaluate exams. This drew our attention and, bound to what we were telling at the beginning, we saw that this project could **finally leave its mark on our faculty**.

Goals

Once we already had the project proposal, we had to think about some objectives to develop throughout the entire academic year. This task was somewhat easy because our classmates from last year, and also our tutor, helped us identify what new features could be added to the Qwizer app. Therefore, finally, among the advice received, we came to propose and establish the following first objectives for the new work:

- **Randomization of questionnaires**: One of the main future objectives of our colleagues last year was the fact of **randomization within the questionnaires** themselves. Offering the ability to randomize quiz questions is essential in this type of application to avoid cheating among students. For this reason, we came to the conclusion that this feature would offer an added value to Qwizer.

  For the implementation of this new functionality we had to think carefully about how we would apply randomization within the application, since within a questionnaire this can be applied in numerous ways. Finally, we identified three main goals:

  - Randomization at the question level. **The questions in a quiz** should appear in a random order for each user who takes the quiz.
  - Randomization at the option level. **The options within the multiple choice questions** should be displayed in a random order for each user.
  - Random selection questions. Within the questionnaires there will be one or several questions that will be chosen for each user based on a subset of questions from the question bank.

- **Application adaptation to mobile devices**: Responsive design is a fundamental step in web applications. Giving users who use mobile devices the possibility to see our application correctly is very important and more so in the context of our application. For this reason we decided to try to **adapt our interface to mobile devices** and try to **make browsing through the application as comfortable as possible**.

- **Inclusion of Markdown and mathematical formulas**: Markdown is a type of markup language that allows **adding rich text to the application** allowing the inclusion of words in bold, italics, images, etc... Our tutor identified the **incorporation of Markdown** as one of the main objectives of this project and also adding the **possibility of writing statements and solutions with mathematical formulas**, allowing the formulation of more complex questions.

- **Integration with Moodle**: Finally, the last objective we set was the integration with the formats of the Moodle platform. **Moodle** was one of the tools that inspired the development of Qwizer, in fact, it is currently the platform on which the Virtual Campus of our faculty is based on. Since it is still in use to this day, the ability to integrate Qwizer with Moodle was one of our most anticipated goals. To do this, we wanted to focus on the fact that **questions could be imported from the Moodle** question bank and **allow exporting the marks of the students who took questionnaires in Qwizer to Moodle**.

Work plan

Throughout this academic year, we have gone through numerous phases in the development of our final degree project. The fact of extending a complete project implied stages of study not only of the new technologies with which we were not familiar, but also large code structures that make use of them. Due to all this, we needed a work plan that would allow us to establish small objectives to complete the tasks we had set.

Thanks to the meetings that we held with our tutor Manuel Montenegro, which were held every two weeks, we organized ourselves to fulfill a series of tasks for each meeting. This methodology helped us, at the beginning, to gradually understand parts of the code and begin to meet the first goals, and later, in more advanced stages of the project, to finalize the main objectives of the work. That being said, the general distribution of the phases of our project were the following:

- **Refactor, testing, updating and deployment of the application** *(September 2022 - February 2023)* One of the longest stages of development. At first, we dedicated ourselves to understanding a large part of the code and later refactor both the frontend, as discussed in chapter 3.5.1, and the backend as

discussed in chapter 3.5.2. We also updated all libraries to the latest version trying to avoid conflicts due to version control. Finally, we prepared the development environment by adding `linters`, environment variables for secrets, designing some tests for the back-end, using a package management tool and using Docker to deploy the project correctly.

- **Questionnaire Randomization** *(March 2023)*: During this stage, the questionnaire randomization functionality was implemented. This process involved making several changes to the application's relational model and application logic as detailed in depth in Chapter 4, which required significant time to update the corresponding code.
- **Markdown, Random Questions and Moodle Integration** *(April 2023)*: During this phase, several key aspects were addressed. The implementation of the Markdown format for questions and the integration with Moodle were relatively easy, with the exception of image management in the case of Markdown. However, incorporating random questions proved to be a significant challenge, as it impacted much of the application's back-end code. These incorporations are described in chapters 4, 5 y 6.
- **Final checks and memory** *(May 2023)*: During the last month of development, we focused on making final adjustements to the app's interface, polishing the details for an improved user experience. We also dedicate time to writing the Final Degree Project document, documenting the entire development process, the decisions made and the results obtained.

Conclusions and future work

In this section, we will present our conclusions once the development of the project has finished. We will address issues such as: the level of compliance with the main objectives, the difficulties encountered during the development of the project and the possible improvements that could be implemented in the future.

Achieved goals

During the course of the project, all the objectives initially set (in section 1.2) were achieved:

- **Questionnaire randomization**: The randomization functionality was successfully implemented both at the question and option level within the questionnaires. In addition, the variety of questions was expanded with the addition of random selection questions. Now, teachers can offer unique quizzes to their students which prevents them from cheating.
- **Adaptation of the application to mobile devices**: We managed to offer an adaptable design to our application, in which we worked hard to be able to adapt its interface to all screens and improve the user experience so that it was comfortable and intuitive.
- **Inclusion of Markdown and mathematical formulas**: The successful implementation of Markdown was achieved, which allows users to enrich the content with formats such as bold, italics, images, among others. In addition, the ability to write statements and solutions with mathematical formulas was added, which expands the possibilities for the creation of more complex and detailed questions.
- **Integration with Moodle**: A successful integration of Qwizer with the formats used in the Moodle platform was achieved. Now users can import questions from the Moodle question bank, making it easy to reuse and migrate existing content. Likewise, the functionality to export the notes of the questionnaires carried out in Qwizer to Moodle was implemented, providing a simpler and more centralized management of qualifications.

On the other hand, we managed to complete other objectives, which appeared as the project progressed. Among them we can find the following:

- **Core design changes**: Initially we did not plan to change the design of our fellow colleagues views, but tied to the goal of responsive design, we were forced to make some changes to almost all of the initial views to improve the overall experience of our users. We even introduce new features that Qwizer did not previously have (section 3.5.3).
- **API documentation**: As we already explained in the improvements part (see section 3.5.2), a good documentation of an API seems essential to us since without it it is extremely difficult to understand each call and the reason for it. At the beginning, we found ourselves in this situation and for the sake of possible future colleagues or just someone who wants to understand our work, we have developed an extensive and interactive documentation on each API call, making it easy for anyone to understand.
- **Testing**: Just like a good documentation, any software development involves a testing phase, where numerous tests are carried out to check if the developed code works as expected. For this reason, we developed a testing section for the back-end part of the application, in which the logic of the API calls was tested.
- **Application deployment**: Finally, if we wanted to test our application in a production environment, we had to explore the possibility of preparing it for deployment, so we configured and adapted our development environment to deploy the application at any time. In fact, in chapter 7 we have explained the solution to this objective in more detail.

In addition, we also managed to complete part of the objectives that our colleagues had planned for future work. Among them we managed to implement:

1. The completion of questionnaires sequentially, where teachers can now choose whether a questionnaire should be done immediately, that is, without being able to return to the previous question once they move on to the next one.
2. Resolution of more than one questionnaire at the same time offline. Previously, a quiz could only be solved if the student lost connection. Instead, now thanks to the new questionnaire storage management, more than one questionnaire can be taken offline.
3. Improvement of the interface for the creation of questionnaires and, as we have already commented in the main objectives, adaptation of the interface to mobile devices.

Difficulties encountered

Throughout the development of the project, some difficulties were encountered that affected the process of implementing the objectives:

- **Legacy code**: One of the biggest challenges was dealing with legacy code from the previous project. The use of pre-existing code presented difficulties to understand its operation and adapt it to the new requirements, leading to many weeks of refactoring and improvements to the initial code. For this reason, the period of adaptation and familiarization with the project lasted longer than normal, preventing some secondary objectives from being met.
- **Lack of knowledge**: It should be noted that both members of the team started out without a base of the technologies used within the project, so the use of these new technologies and tools during the development of the project implied a period of learning and familiarization. Both Django and React were frameworks that had complicated mechanisms to understand and assimilate at first. Therefore we can add the main problems encountered during development with both frameworks:
  - **Django**: One of the libraries that has been used most extensively this year is called *djangorestframework*. Its use is explained in section 3.5. The extension in the use of this library caused numerous problems since the use of *serializers*, *viewsets* and other mechanisms of the same were somewhat complicated to implement in the received inherited code. This, added to the little understanding that we had of Django in the early stages of development, caused us great complications at the beginning of the project, which we resolved thanks to the official documentation of both Django and the library.
  - **React**: Aside from the general unfamiliarity of React, dealing with components so logically heavy was initially overwhelming. After several weeks of development, this feeling of pressure lessened and even led to numerous improvements in each of the components.
- **Time**: The time factor was another significant challenge. Much of the development was to adapt and improve the previous code, and that resulted in less time to develop new features. Even if the initial objectives set for the work were met, a verification and testing stage in a real environment would have helped the project to reach a higher level of refinement. All this without regarding the little availability at some moments of the project due to the pressure of the course and at specific moments, of final exams.

Future work

Although the proposed objectives have been successfully achieved, we believe that the project still has room for improvement. To this end, we propose the following modifications:

- **Accessibility**: Although we have adapted most of the application for all devices, we believe that the accessibility of our application could still be extended

allowing all types of users, regardless of their physical or cognitive abilities, to use Qwizer. The use of accessibility techniques such as changing colors, changing themes, descriptions with help texts, use of simple typography, and large, easy-to-locate buttons and panels would help adjust Qwizer to a more inclusive environment.

- **Testing in the front-end**: As we explained before, one of the secondary objectives accomplished was the fact of documenting and testing the back end of our application. This has resulted in numerous benefits throughout the development of the application. On the other hand, one of the tasks that we would have liked to carry out and due to lack of time has not been able to complete is the fact of testing the interface and the main components of the front-end of our application. On numerous occasions, we have had to check the correct functioning of the component logic, which sometimes caused API calls and consequently, changes in the state of the database. This tedious process made us consider carrying out a series of tests on the most essential views of the interface, allowing us to test this logic and carry out the tests against a database prepared for the tests. In addition, this type of testing directs the code on a more sustainable and stable path, for which future developments would be grateful for them.

- **Own interface for mobiles**: Throughout the development of the project, there have been numerous occasions in which we have reconsidered making our own interface for mobile devices, because even if we have spent time getting an adaptable design of our application, there are always some components or some fragments of the views that are slightly out of the style of a mobile application. For this we study *Ionic*, a framework focused on the development of user interfaces on mobile devices, which, since its syntax is very similar to our main styling framework, *Bootstrap*, would allow us to have designed a unique and exclusive interface for mobile devices. In the end, we ended up giving up the idea because it was a very difficult goal to complete in the short time we had left, but in the future, this idea could give Qwizer a fundamental value for its implementation within the subjects of the faculty.

- **More types of questions**: Qwizer is an application based on completing questionnaires and one of the sections in which it still leaks is its small catalog of questions. Although this year we have spent some time creating a new question type (random selection questions) we thought that more question types could be added, such as matching questions, fill-in-the-blank questions, among others. Also, thanks to the new database structure, adding these new question types would be easy to do. For this reason, we think that this new addition of questions could give Qwizer another image and propel it into a more attractive environment.

- **Import more types of questions**: In relation to the import of questions, we also have numerous open fronts that we could cover, such as the fact of importing other types of questions from Moodle, since right now, only can be imported short answer and multi-choice questions. At the same time, we

have rethought the idea of importing questions with other formats. Currently, Qwizer only allows you to import questions with *XML-Moodle* format, concentrating all this functionality in a single format. Therefore, in summary, importing other types of questions from Moodle and increasing the number of formats to import questions, seems to us to be an objective that could be considered for the future.

- **Use moodle API. Improve connectivity with Moodle**: Linked to the export of Qwizer grades to Moodle, we believe that this objective can be improved by making interoperability between Qwizer and Moodle more comfortable. For this, we thought that it would be better to use Moodle's own API to expand the possibilities of integration with our application.

Bibliografía

[1] Benoit Chesneau. gunicorn. https://gunicorn.org/, 2023.

[2] Django Import Export. django-import-export. https://django-import-export.readthedocs.io/, 2023.

[3] Django Software Foundation. Django. https://docs.djangoproject.com/en/4.2/, 2023.

[4] El Fakhri Ouajih, Z. y Martínez Gamero, P. Aplicación web progresiva para la realización de cuestionarios, 2022. Trabajo de Fin de Grado en Ingeniería Informática, Facultad de Informática UCM, Departamento de Sistemas Informáticos y Computación, Curso 2021/2022.

[5] Hello Pangea. hello-pangea/dnd. https://github.com/hello-pangea/dnd, 2023.

[6] Jens Neuhalfen. drf-spectacular. https://drf-spectacular.readthedocs.io/, 2023.

[7] John Resig. jQuery. https://api.jquery.com/, 2023.

[8] Mark Otto and Jacob Thornton. Bootstrap. https://getbootstrap.com/docs/5.3/getting-started/introduction/, 2023.

[9] Martin Dougiamas. Moodle XML. https://docs.moodle.org/all/es/Formato_Moodle_XML, 2023.

[10] Matt Zabriskie. axios. https://github.com/axios/axios, 2023.

[11] Max Lynch, Ben Sperry, and Adam Bradley. Ionic. https://ionicframework.com/docs, 2023.

[12] Mozilla Contributors. localforage. https://localforage.github.io/localForage/, 2023.

[13] React Contributors. React. https://reactjs.org/, 2023.

[14] React Training. react-router-dom. https://reactrouter.com/web/guides/quick-start, 2023.

[15] Rolf Håvard Blindheim. django-environ. https://github.com/joke2k/django-environ, 2023.

[16] SmartBear Software. Swagger. https://swagger.io/, 2023.

[17] Solomon Hykes. Docker. https://docs.docker.com/, 2023.

[18] Sunscrapers. djoser. https://djoser.readthedocs.io/, 2023.

[19] Tom Christie. djangorestframework. https://www.django-rest-framework.org/, 2023.

Apéndice A

Documentación de la API

En este apéndice se encuentra la documentación de la API generada a partir del archivo JSON de Swagger y convertida en formato PDF. Sin embargo, es importante tener en cuenta que la interfaz de Swagger es considerablemente superior a este documento en PDF.

API Reference

Qwizer API

API Version: 1.0.0

This is Qwizer's official API documentation.

INDEX

Security and Authentication

SECURITY SCHEMES

KEY	TYPE	DESCRIPTION
tokenAuth	apiKey	Token-based authentication with required prefix "Token"

API

1. AUTH

Autenticación de usuarios

1.1 POST /api/auth/token/login

Use this endpoint to obtain user authentication token.

REQUEST

REQUEST BODY - application/json

```
{
  password  string
  email     string
}
```

FORM DATA PARAMETERS

NAME	TYPE	DESCRIPTION
password	string	
email	string	

FORM DATA PARAMETERS

NAME	TYPE	DESCRIPTION
password	string	
email	string	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  password  string
  email     string
}
```

1.2 POST /api/auth/token/logout

Use this endpoint to logout user (remove user authentication token).

REQUEST

No request parameters

RESPONSE

STATUS CODE - 200: No response body

1.3 GET /api/auth/user/me

REQUEST

No request parameters

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  first_name  string   max:30 chars
  last_name   string   max:150 chars
  role*       enum     ALLOWED:student, teacher, admin
                       * `student` - student
                       * `teacher` - teacher
                       * `admin` - admin
  id*         integer  READ-ONLY
  email*      string   READ-ONLY
}
```

2. ESTUDIANTES

Estudiantes

2.1 GET /api/estudiantes

Lista de los usuarios que tienen rol de estudiante
GET /estudiantes

REQUEST

No request parameters

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
[{
Array of object:
   alumnos* [{
   Array of object:
      id*         string
      nombre*     string
      apellidos*  string
   }]
}]
```

2.2 GET /api/estudiantes/{id}

Lista de los usuarios que tienen rol de estudiante de una asignatura
GET /estudiantes/{id_asignatura}

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
   alumnos* [{
   Array of object:
      id*         string
      nombre*     string
      apellidos*  string
   }]
}
```

2.3 GET /api/estudiantes/{id}/disponibles

Lista de los usuarios que tienen rol de estudiante que no estan cursando una asignatura

GET /estudiantes/{id_asignatura}/disponibles

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  alumnos* [{
    Array of object:
      id*         string
      nombre*     string
      apellidos*  string
  }]
}
```

3. QR

Qr

3.1 POST /api/qr

Insertar hash para un intento

REQUEST

REQUEST BODY - application/json

```
{
  idUsuario*       integer
  idCuestionario*  integer
  hash*            string
}
```

FORM DATA PARAMETERS

NAME	TYPE	DESCRIPTION
idUsuario	integer	
idCuestionario	integer	
hash	string	

FORM DATA PARAMETERS

NAME	TYPE	DESCRIPTION
idUsuario	integer	
idCuestionario	integer	
hash	string	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  inserted*  string
  message*   string
}
```

STATUS CODE - 400: Error: Bad Request

STATUS CODE - 403:

RESPONSE MODEL - application/json

```
{
  inserted*  string
  message*   string
}
```

3.2 GET /api/qr/{id_usuario}/{id_cuestionario}

Comprobación hash qr

GET /{idUsuario}/{idCuestionario}

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id_usuario	integer	
*id_cuestionario	integer	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  alumnos* [{
    Array of object:
      id*         string
      nombre*     string
      apellidos*  string
  }]
}
```

4. QUESTION

Preguntas

4.1 POST /api/question

Crear preguntas a partir de un fichero csv o xml

REQUEST

FORM DATA PARAMETERS

NAME	TYPE	DESCRIPTION
file	string(binary)	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  inserted string
  message  string
}
```

4.2 PUT /api/question/{id}

Actualizar una pregunta

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id de la pregunta

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  message string
}
```

4.3 DELETE /api/question/{id}

Borrar una pregunta

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id de la pregunta

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  message  string
}
```

4.4 POST /api/question/imagen

REQUEST

No request parameters

RESPONSE

STATUS CODE - 200: No response body

---

5. SUBJECT

Asignaturas

5.1 GET /api/subject

Lista de asignaturas

REQUEST

No request parameters

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
[{
Array of object:
  asignaturas [{
  Array of object:
    id          string
    asignatura  string
  }]
}]
```

5.2 GET /api/subject/{id}/cuestionarios

Lista de cuestionarios de una asignatura

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id de la asignatura

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  cuestionarios [{
  Array of object:
    id      string
    titulo  string
  }]
  nombre   string
}
```

5.3 POST /api/subject/{id}/delete_enroll

Desmatricular una lista de estudiantes

DELETE /asignatura/{pk}/enroll

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id de la asignatura

REQUEST BODY - application/json

```
{
  alumnos [{
  Array of object:
    id         integer
    nombre     string
    apellidos  string
  }]
}
```

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  borrados  integer
  errors    [undefined]
}
```

5.4 POST /api/subject/{id}/enroll

Matricular una lista de estudiantes

POST /asignatura/{pk}/enroll

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id de la asignatura

REQUEST BODY - application/json

```
{
  alumnos [{
  Array of object:
    id         integer
    nombre     string
    apellidos  string
  }]
}
```

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  insertados integer
  errors     [undefined]
}
```

5.5 GET /api/subject/{id}/preguntas

Lista de preguntas de una asignatura

Devuelve todas las preguntas de una asignatura para el banco de preguntas

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id de la asignatura

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  preguntas
    ONE OF
    OPTION 1{
      id          integer
      question    string
      title       string
      type        enum     ALLOWED:text
      correct_op  string
    }
    OPTION 2{
      id          integer
      question    string
      title       string
      type        enum     ALLOWED:test
      options [{
      Array of object:
        id  string
        op  integer
      }]
      correct_op  integer
    }
}
```

5.6 GET /api/subject/me

Estado de los cuestionarios de un usuario

REQUEST

No request parameters

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  asignaturas [{
    Array of object:
      id          integer
      nombre      string
      cuestionarios {
        nCuestionarios  integer
        nCorregidos     integer
        nPendientes     integer
      }
  }]
}
```

6. TEST

Cuestionarios

6.1 POST /api/test

Crear cuestionario

REQUEST

REQUEST BODY - application/json

```
{
  cuestionario {
    testName        string
    testPass        string
    testSubject     string
    secuencial      string
    testDuration    string
    fechaApertura   integer
    fechaCierre     integer
    fechaVisible    integer
    questionList
      ONE OF
      OPTION 1 {
        id              integer
        question        string
        title           string
        tipo            string
        correct_op      string
        punt_positiva   integer
        punt_negativa   integer
        fijar           boolean
        aleatorizar     boolean
      }
      OPTION 2 {
        id              integer
        question        string
        title           string
        tipo            string
        options [{
        Array of object:
          id  integer
          op  string
        }]
        correct_op      integer
        punt_positiva   integer
        punt_negativa   integer
        fijar           boolean
        aleatorizar     boolean
      }
    aleatorizar     string
  }
}
```

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  inserted  string
  message   string
}
```

6.2 GET /api/test/{id}

Descargar cuestionario

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	Id del cuestionario

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  id                        integer
  titulo                    string
  duracion                  integer
  secuencial                boolean
  password                  string
  fecha_visible             string
  fecha_apertura            string
  fecha_cierre              string
  aleatorizar               boolean
  profesor                  integer
  asignatura                integer
  iv                        string
  encrypted_message         string
  formatted_fecha_apertura  string
  formatted_fecha_cierre    string
}
```

6.3 POST /api/test/{id}/enviar

Responder a un cuestionario

POST /enviar -> POST /tests/1/enviar

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	A unique integer value identifying this cuestionario.

REQUEST BODY - application/json

```
{
  respuestas {
    id
      ONE OF
      OPTION 1{
        id     integer
        type   enum     ALLOWED:text
        answr  string
      }
      OPTION 2{
        id     integer
        type   enum     ALLOWED:test
        answr  integer
      }
  }
  hash        string
}
```

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  nota  number
}
```

6.4 GET /api/test/{id}/info

Información de un cuestionario para un alumno

POST /get-quiz-info -> GET /tests/{pk}/info

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	A unique integer value identifying this cuestionario.

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  duracion                 integer
  formatted_fecha_apertura string
  formatted_fecha_cierre   string
  formatted_fecha_visible  string
  fecha_apertura           string
  fecha_cierre             string
  fecha_visible            string
  corregido                integer
  nota                     number
}
```

6.5 GET /api/test/{id}/nota/{id_alumno}

Nota de un cuestionario de un estudiante

También se devuelven las respuestas del usuario

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	A unique integer value identifying this cuestionario.
*id_alumno	string PATTERN: ^\d+$	

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  titulo        string
  nota          number
  questions
    ONE OF
    OPTION 1{
      id          integer
      question    string
      type        enum     ALLOWED:text
      correct_op  string
      user_op     string
    }
    OPTION 2{
      id          integer
      question    string
      type        enum     ALLOWED:test
      correct_op  integer
      user_op     integer
    }
}
```

6.6 GET /api/test/{id}/notas

Lista de notas de todos los alumnos de un cuestionario

POST /get-quiz-grades -> GET /tests/{pk}/grades

REQUEST

PATH PARAMETERS

NAME	TYPE	DESCRIPTION
*id	integer	A unique integer value identifying this cuestionario.

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  notas {
    email {
      id        integer
      nombre    string
      apellidos string
      nota      number
    }
  }
}
```

6.7 POST /api/test/subir

Creación de un cuestionario a partir de un yaml

POST /upload -> POST /tests/upload

REQUEST

REQUEST BODY - application/json

```
{
  fichero_yaml string
}
```

RESPONSE

STATUS CODE - 200:

RESPONSE MODEL - application/json

```
{
  inserted string
  message  string
}
```
