UNIVERSIDAD MIGUEL HERNANDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

"DISEÑO E IMPLEMENTACIÓN DE UNA PÁGINA WEB DESTINADA A CUESTIONARIOS DE TIPO TEST"

TRABAJO FIN DE GRADO

Enero - 2023

AUTOR: Francisco García López

DIRECTOR/ES: Pedro Pablo Garrido Abenza

Agradecimientos

Quisiera agradecer a Pedro Pablo Garrido Abenza, es decir, mi director del Trabajo Final de Grado por la paciencia durante el periodo de realización.

Every accomplishment starts with the decision to try.

John F. Kennedy

Índice general

Índice de figuras

Capítulo 1

Introducción

En este capítulo se pretende describir el objetivo de la aplicación a realizar, una descripción de las tecnologías empleadas y la estructura general del Trabajo Fin de Grado (TFG).

1.1. Objetivos

El objetivo general de este trabajo es la creación de una web para la realización de cuestionarios de tipo test donde se puedan ingresar datos de distintas temáticas. Estas temáticas estarán organizadas según una tecnología en concreto, o, una temática. Dentro de las temáticas tendremos cuestionarios tipo test, los cuales tendrán una cantidad de preguntas que el usuario decidirá. Todos estos datos estarán almacenados en una base de datos relacional.

Como objetivos específicos,

1. Se implementará una base de datos, en nuestro caso gestionada mediante PostgreSQL y se desarrollará una API que gestionará las consultas a la base de datos y, devolverá un código HTTP que se visualizará en el navegador de Internet del usuario.

2. También, se realizará la parte de interacción con el usuario, que estará comprendida por dos webs completamente diferentes.

La primera será una aplicación de administrador, donde se podrá hacer una consulta de los distintos temas, los test y las preguntas y respuestas, pudiendo editar, crear o borrar cada una de estas a decisión del administrador. La otra web es la parte de usuario, el cuál no será administrador, en la que el usuario podrá ver las temáticas y podrá realizar cualquier test contestando a cada una de las preguntas. Cuando el test se haya completado, el usuario podrá ver su puntuación. Toda la parte de web estará realizada en el *framework* de programación Vue que funciona mediante JavaScript.

3. Todas las aplicaciones desarrolladas se incluirán en un contenedor Docker, que nos permitirá virtualizarlas y así probarlas en cualquier ordenador siguiendo el anexo B.

4. Todo el código estará almacenado en GitHub y el desarrollo seguirá una metodología de GitHub denominada Gitflow, la cual permitirá tener un histórico de los cambios realizados y distintas ramas para trabajar.

5. Finalmente, se realizará un proceso de validación de la web para cerciorarse de que el sistema funciona como cabe esperar de cara a usabilidad. La validación se llevará a cabo mediante los denominados tests end-to-end con *nightwatch*, *framework* que se incluirá en Vue. Estos tests se ejecutarán en GitHub cada vez que se hagan *commits*, es decir, cuando el código sea modificado y se suba a esta plataforma.

1.2. Decisiones de software

En este apartado se justifica la elección de las tecnologías utilizadas para la consecución de los objetivos propuestos.

Lo primero será decir que HTTP será una tecnología obligatoria, ya que es, en esencia, como funciona Internet, es decir, una web devolverá una conexión HTTP y dependiendo de su código de respuesta se podrá acceder o no a esta página web. Los códigos de respuesta HTTP se verán más en profundidad en la sección 2.2.6 HTTP *status code*.

A continuación, deberemos seleccionar una base de datos que soporte el tipo de dato JSON, como podrían ser MySQL o PostgreSQL. En nuestro caso seleccionaremos la segunda opción ya que este esquema de base de datos implementa de forma nativa un tipo de dato denominado JSONB. La razón por la cual utilizaremos JSON en una base de datos principalmente será por eficiencia, es decir, podremos añadir más datos sin necesidad de crear tablas secundarias con relaciones complejas, ventaja que reducirá considerablemente el espacio de almacenamiento utilizado. También nos permitirá modificar datos y estructuras sin necesidad de modificar la base de datos. Otro aspecto a tener en cuenta sobre los JSON es que este formato nos permitirá la integración de la aplicación con otras aplicaciones o servicios. También utilizaremos GitHub para tener un histórico del trabajo realizado.

Finalmente, utilizaremos Python para la creación de la API y un *framework* de JavaScript para la parte de web. Python se ha seleccionado debido a que es un lenguaje muy usado hoy en día, tanto academicamente como en el mundo laboral, siendo un lenguaje de fácil implementación y con una rápida curva de aprendizaje.

1.2.1. Selección de PostgreSQL frente a MySQL

Como se ha dicho en la sección anterior, PostgreSQL se ha seleccionado por el manejo de datos tales como JSON. A continuación, se dan algunos puntos más que nos ha llevado a seleccionar dicho gestor de base de datos.

1. **Estándares de SQL:** PostgreSQL es considerado como uno de los sistemas de gestión de bases de datos más conformes con los estándares SQL y ofrece un mayor soporte para características avanzadas como transacciones complejas, subconsultas y vistas.

2. **Escalabilidad:** PostgreSQL es escalable y maneja bien grandes conjuntos de datos y altas cargas de trabajo.

1.2.2. Selección de Flask frente a Django

Al principio se propuso realizar el trabajo con Django, *Framework* bastante popular de Python, pero se descartó por los siguientes motivos.

1. **Tamaño y complejidad:** Flask es un *framework* web minimalista y ligero, mientras que Django es un *framework* web completo con una gran cantidad de características y herramientas. Por tanto, Flask es una herramienta más simple y fácil de aprender, pero que permite realizar nuestros objetivos.

2. **Flexibilidad:** Flask te permite tener control total sobre cómo organizar y estructurar tu aplicación, mientras que Django tiene una estructura predefinida para organizar tu aplicación.

3. **Escalabilidad:** Django tiene una gran cantidad de características y herramientas integradas para escalar aplicaciones de gran envergadura. Flask, como *framework* más ligero, es más adecuado para aplicaciones más pequeñas o proyectos de prueba.

1.2.3. Entorno de desarrollo

También comentar que la aplicación será desarrollada mediante Visual Studio Code [1], plataforma que permite implementar el subsistema de Linux para Windows [10], es decir, utilizaremos una consola de Linux para el desarrollo de todas las aplicaciones.

1.3. Estructura de la memoria

Para finalizar la introducción, la memoria se estructura del siguiente modo. En la siguiente sección 2 se presentan las distintas tecnologías escogidas para la realización del trabajo, así como los procedimientos empleados. En la sección 3 de describirá la base de datos y las aplicaciones desarrolladas, es decir, se desarrollarán los resultados obtenidos del trabajo. Por último, en la sección 4 se finaliza con las conclusiones e indicaciones a trabajos futuros.

Capítulo 2

Material y métodos

En este capítulo se pretende realizar una investigación de las tecnologías empleadas y se proporciona una breve explicación de los conceptos que se van a utilizar.

2.1. Web Services

Los servicios web, también conocidos como *web services*, permiten que las diferentes computadoras de una red se comuniquen entre sí. La comunicación más común es la que se da entre un cliente y un servidor, donde el cliente hace una solicitud y el servidor responde con la información solicitada. Brevemente, en la figura 2.1 se presenta la comunicación del sistema que se pretende desarrollar, siendo ésta de subida y bajada a la hora de utilizar GitHub y, de actualización cuando estamos hablando de contenedores Docker, es decir, la izquierda del diagrama.

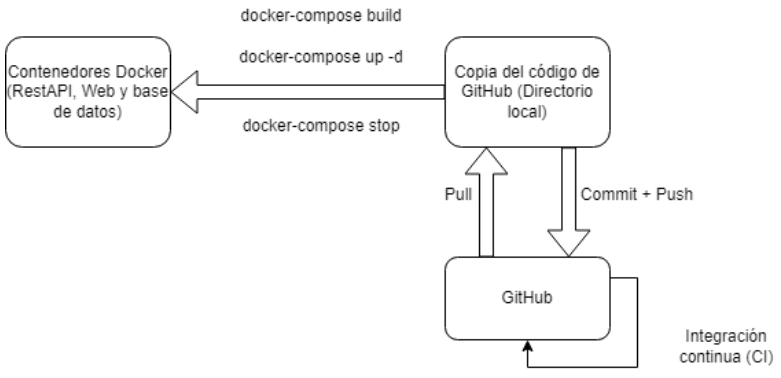

Figura 2.1: Diagrama Docker-GitHub

2.2. RESTAPI

El protocolo REST se refiere a la forma en la que un cliente se comunica con un servidor a través de la web utilizando el protocolo HTTP. Este tipo de comunicación es sencilla de implementar, por lo que se ha vuelto muy popular y se conoce como API REST. Al utilizar este protocolo se suele emplear el formato de datos JSON, y esto es lo que se emplea para seleccionar la base de datos.

En cuanto a la tecnología de REST, también podemos hablar de las características que posee, las cuales son las siguientes.

1. **Cliente-Servidor:** Característica principal de los REST que los define y arquitectura de la cual se deben basar.

2. **Sin estado:** los servicios REST pueden escalar a gran escala con la finalidad de atender a un gran número de clientes a la vez, lo que implica que se creen balanceos de carga para minimizar el tiempo de respuesta con servidores intermedios. Al utilizar servidores intermedios estos servicios deberán enviar toda la información dentro del paquete HTTP para generar una respuesta valida (HTTP response).

3. **Métodos HTTP:** Estos métodos serán *GET*, *POST*, *PUT* y *DELETE* los cuales son las consultas básicas HTTP y serán utilizadas por la API para modificar la base de datos.

4. **Respuesta en formato conocido:** El servicio REST debe mostrar al cliente toda la información en un formato conocido. El formato más habitual es el formato JSON para ver los datos.

En la figura 2.2 observamos un diagrama que muestra la interacción entre el cliente y el servidor, siendo el cliente nuestras aplicaciones de Vue y, el servidor, la parte de la RESTAPI y de la base de datos.

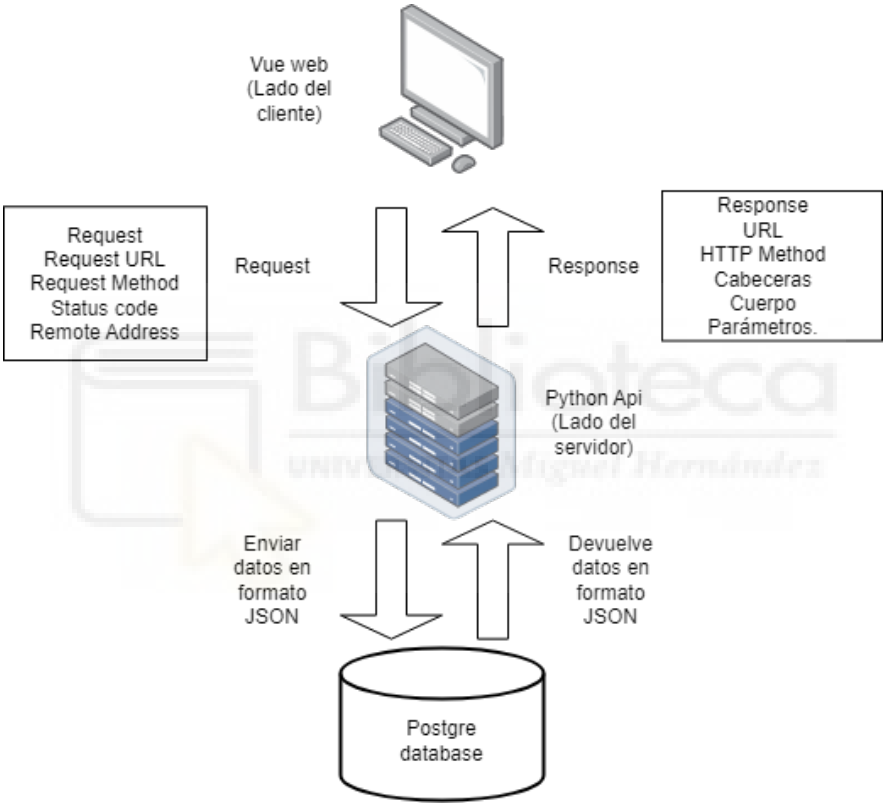

Figura 2.2: Request y Response cliente-servidor.

2.2.1. Elementos de la API

La API se compone de una serie de *scripts* escritos en Python, agrupados en 3 categorías:

```
api
├── controllers
│   ├── paths_tests_questions.py
│   ├── paths_tests.py
│   ├── paths.py
│   ├── questions.py
│   ├── resources.py
│   ├── tests.py
│   ├── topics.py
│   ├── user_login.py
│   ├── user_paths.py
│   ├── user_questions.py
│   ├── user_resources.py
│   ├── user_tests_questions.py
│   ├── user_tests.py
│   └── users.py
├── lib
│   ├── db.py
│   └── response_parser.py
├── routes
│   ├── paths_tests_questions_routes.py
│   ├── paths_tests_routes.py
│   ├── paths_routes.py
│   ├── questions_routes.py
│   ├── resources_routes.py
│   ├── tests_routes.py
│   ├── topics_routes.py
│   ├── user_login_routes.py
│   ├── user_paths_routes.py
│   ├── user_questions_routespy
│   ├── user_resources_routes.py
│   ├── user_tests_questions_routes.py
│   ├── user_tests_routes.py
│   └── users_routes.py
└── app.py
```

La primera de todas serán los controladores (*controllers*), ficheros que contendrán todas las sentencias SQL necesarias para que la web funcione correctamente. En segundo lugar, el directorio *lib* estará compuesto por dos ficheros *response_parser.py* que será el que devuelva un código HTTP una vez realizada la consulta SQL y, *db.py* que nos permitirá hacer sentencias SQL sin necesidad de escribir toda la sentencia mismamente, por ejemplo, en el caso de usuarios, cuando se pretende borrar uno se deberá borrar también todo su contenido como lo son paths, tests y questions que haya completado. Finalmente, los ficheros de las rutas *routes*, es decir, los que contendrán los endpoints, llamarán a los que crean las sentencias y, éstos, a su vez, estarán siendo incluidos en la aplicación principal *app.py*

En esta API, la estructura de una URL se crea a partir de URL's usadas para acceder a los *endpoints*, es decir, una URL específica que se utiliza para interactuar con un servicio o recurso específico de la API. Normalmente siguen un patrón que incluye

la URL principal, la dirección a la cual se envían las peticiones y, si es necesario, parámetros secundarios usados para especificar la solicitud. Un ejemplo de URL sería la siguiente: http://localhost:3000/users

En este caso, la URL principal sería el *localhost*, es decir, nuestro propio ordenador, con el puerto 3000 y la ruta a la cual enviaremos las peticiones será '/users'. En cuanto a los parámetros secundarios podríamos añadir un 'ok' en el caso de que la petición se haya realizado con éxito, pero en nuestro caso, devolveremos un código HTTP y un mensaje, que contendrá o no los datos dependiendo de si el usuario está registrado o no lo está.

2.2.2. Parámetros de consulta

Los parámetros de consulta son el cuerpo de la consulta y contienen la información para realizar la consulta satisfactoriamente. Estos parámetros son usados en consultas a bases de datos como POST o PUT, que necesitan de una información para poder utilizarlas, estando estas en formato JSON. Un ejemplo de esto es el siguiente:

Como vemos, los parámetros incluyen el nombre y apellidos el usuario, su apodo o *nickname*, el idioma preferido, su dirección de correo, y por último, su contraseña, que se guarda encriptada mediante la librería de JavaScript bcryptjs.

```
{
  "name": string,
  "surname": string,
  "nickname": string,
  "lang": string,
  "email": string,
  "password": string,
}
```

Bcryptjs se basa en el algoritmo de encriptación bcrypt, algoritmo de un solo sentido, es decir, que no se puede revertir para recuperar la información original. Su función principal es generar un *hash* de una contraseña, es decir, una cadena de caracteres que se utiliza para verificar la autenticidad de una contraseña sin tener que almacenar la contraseña en claro. Este proceso se realiza mediante un valor denominado salt, que es un valor aleatorio agregado a una contraseña antes de realizar el hash.

Su propósito es fortalecer la seguridad de la contraseña al hacer que dos contraseñas diferentes tengan valores hash únicos, incluso si son iguales.

Para verificar una contraseña, se utiliza el mismo salt para generar un hash de la contraseña proporcionada y se compara con el hash almacenado. Si los hashes coinciden, se considera que la contraseña es válida.

2.2.3. Cabeceras

Las cabeceras de las API son información extra que será visible tras cada llamada a ésta y llevan la siguiente información: [2]

- Requests: Petición del cliente a un servidor para obtener una información específica
- Responses: Información que el servidor envía de vuelta al cliente después de recibir una solicitud.
- Authorization headers: son un tipo de encabezado HTTP que se utiliza para proporcionar información de autenticación al servidor.

En este trabajo se utilizará una cabecera extra de autenticación, utilizando para ello los JWT o JSON Web Token. Este *token* nos permitirá almacenar información del usuario, tal como su identificador, el idioma que habla o, lo más importante, si se trata de un administrador o no. En el caso de que un usuario escriba credenciales no registradas en la base de datos, no se podrá acceder a la información. En la parte de web se hablará más a cerca de esto mismo, pero por ahora, deberemos saber que un JSON Web Token, sirve, principalmente, para la autenticación del usuario.

2.2.4. Seguridad JWT

Como se ha comentado en el punto anterior, JWT significa JSON Web Token y nos servirá para identificar a usuarios autenticados en nuestra aplicación. En nuestra API utilizaremos la librería Flask JWT extended para este propósito. Esto permitirá compartir información con la aplicación de cliente, estando ésta encriptada, y, la información que sea necesaria solo se devolverá en el caso de que el usuario este registrado en la web, es decir, éste, en resumen, es una cadena de caracteres que contiene información para ser verificada por el servidor. [12]

2.2.5. Métodos HTTP

Los métodos de una API son, principalmente, las 4 sentencias de SQL y se decidirán para cada una de las rutas, es decir, si se pretende obtener información, será un *GET*, si se pretende insertar información, un *POST*, para actualizar usaremos un *PUT* y, finalmente, si se desea borrar algún dato utilizaremos *DELETE*. Un ejemplo de lo comentado se muestra a continuación.

```
1 @app.route('/users', methods = ['POST'])
2 @app.route('/users', methods = ['GET'])
3 @app.route('/users/<int:user_id>', methods = ['GET'])
4 @app.route('/users/<int:user_id>', methods = ['PUT'])
5 @app.route('/users/<int:user_id>', methods = ['DELETE'])
```

En cuanto a `user_id`, se refiere a un identificador de usuario, que permitirá enfocarnos en uno solo y no en una lista de usuarios, como se hace en los dos primeros.

2.2.6. HTTP Status code

La API de consulta a la base de datos a través de la RESTAPI deberá devolver un código de estado, o *status code*, por cada consulta realizada, que identifica si la operación se ha realizado con éxito o, si, por el contrario, ha habido algún problema. A continuación, realizaremos una enumeración de los códigos más comunes, seleccionando los que nos interesen para nuestra API, sabiendo que del 100 al 199 son respuestas informativas, del 200 al 299 son respuestas satisfactorias y que los errores son entre 400 y 599. Los códigos que se generan en las consultas mediante la RESTAPI son, en su mayoría, informativos [11]. Los que nos interesan conocer a nosotros, por tanto, serán el código de respuesta 200, el 201, el 404 y el 500, explicados en la siguiente tabla.

CÓDIGO	DESCRIPCIÓN
200	La solicitud ha tenido éxito
201	La solicitud ha tenido éxito y se ha creado un nuevo campo en la base de datos.
404	El contenido solicitado no se encuentra disponible en la base de datos o no existe
500	El servidor ha encontrado una situación que no sabe cómo manejar (Internal server error)

2.2.7. SQLAlchemy

SQLAlchemy es una librería de Python que nos permitirá realizar las consultas SQL y conectarnos a una base de datos de la forma más sencilla posible. Esta librería puede utilizarse para abstraerse y evitar detalles de bajo nivel y consultas SQL, pero éste no es nuestro objetivo. Nuestro objetivo, por tanto, es utilizar sentencias SQL para las consultas a la base de datos. A la hora de realizar la configuración de la API para que ésta se conecte a la base de datos podemos mirar la documentación relacionada, pero, lo principal será crear un fichero con las siguientes líneas.

```
1 class TestingConfig():
2     TESTING = True
3     DEBUG = True
4     ENV = 'environment mode'
5     JWT_SECRET_KEY = "5iNySaf97&NW"
6     SQLALCHEMY_TRACK_MODIFICATIONS = False
7     SQLALCHEMY_DATABASE_URI = "postgresql://postgres:
          changeme@host.docker.internal:5432/tfg-db"
```

Este código define una clase llamada `TestingConfig` en Python. La clase contiene varios atributos que se utilizan para configurar el comportamiento de la aplicación en un entorno de pruebas.
Una explicación de lo que hace cada uno de los parámetros, sería la siguiente:

1. El atributo `TESTING` está establecido en *True*, línea 2 del código anterior, lo que habilita el modo de pruebas para la aplicación.

2. El atributo `DEBUG` está establecido en *True*, línea 3, lo que habilita el modo de depuración para la aplicación.

3. El atributo `ENV` está establecido en *environment mode*, línea 4, lo que establece la variable de entorno para la aplicación.

4. El atributo `JWT_SECRET_KEY` está establecido en '5iNySaf97&NW', línea 5, lo que se utiliza para encriptar y desencriptar los tokens JSON Web.

5. El atributo `SQLALCHEMY_TRACK_MODIFICATIONS`, línea 6, está establecido en *False*, lo que deshabilita la característica de SQLAlchemy que realiza un seguimiento de las modificaciones a los objetos y emite señales.

6. El atributo `SQLALCHEMY_DATABASE_URI`, línea 7, está establecido en "postgresql:/postgres:changeme@host.docker.internal:5432/tfg-db", lo que establece la cadena de conexión para la base de datos. La cadena de conexión utiliza el controlador PostgreSQL, y se conecta a una base de datos llamada "tfg-db", en host.docker.internal (dirección IP utilizada por Docker para acceder al host)

en el puerto 5432, con el nombre de usuario 'postgres', siendo la contraseña 'changeme'.

Además del código del fichero anterior, se requiere el siguiente, que define una función llamada `load_config()` en Python. La función toma un parámetro opcional llamado `mode`, el cual por defecto obtiene el valor del entorno establecido en la variable `'MODE'`. La función importa la clase correspondiente a cada modo de configuración. Si el modo es *'production'*, importa la clase `"ProductionConfig"`, si el modo es *'testing'* importa la clase `"TestingConfig"`, si el modo es *'development'* importa la clase `"DevelopmentConfig"`, y si no se establece un modo o es diferente a los mencionados anteriormente, importa la clase `'Config'`. La función se encarga de cargar las configuraciones correspondientes en función del modo establecido.

```
def load_config(mode=os.environ.get('MODE')):
    """Load config."""
    if mode == 'production':
        from .production import ProductionConfig
        return ProductionConfig
    elif mode == 'testing':
        from .testing import TestingConfig
        return TestingConfig
    elif mode == 'development':
        from .development import DevelopmentConfig
        return DevelopmentConfig
    else:
        from .default import Config
        return Config
```

2.3. Docker

Docker es una aplicación que permite crear, desplegar y ejecutar contenedores de manera sencilla, lo que facilita la tarea de desarrolladores y administradores de sistemas al permitirles trabajar con aplicaciones en diferentes entornos de manera estandarizada. La ventaja principal de Docker es que permite a los desarrolladores trabajar con una configuración de entorno similar a la que se encuentra en producción, lo que ayuda a reducir problemas de compatibilidad y facilitar la implementación de aplicaciones.

2.3.1. Dockerización de aplicaciones

Continuando con Docker, la *dockerización* de una aplicación se refiere al proceso de convertir una aplicación en un contenedor Docker. Esto implica crear una imagen Docker que encapsula todo lo necesario para ejecutar la aplicación, incluyendo el código fuente, las dependencias y las configuraciones necesarias. Una vez creada la imagen, ésta se puede utilizar para ejecutar una instancia de la aplicación en cualquier sistema que tenga Docker instalado, independientemente del sistema operativo subyacente.

La *dockerización* de una aplicación tiene varias ventajas, tales como:

1. Facilitar la distribución y el despliegue de la aplicación, ya que solo se requiere tener Docker instalado en el sistema para ejecutarla.

2. Aislar la aplicación de su entorno de ejecución, lo que reduce la posibilidad de conflictos de dependencias y configuraciones.

3. Facilitar la creación de entornos de desarrollo y pruebas, ya que se pueden crear contenedores con configuraciones específicas para cada entorno.

4. Facilitar la escalabilidad de la aplicación, ya que se pueden crear varias instancias de la aplicación en contenedores diferentes y escalarlas de manera independiente.

En resumen, la *dockerización* de una aplicación es un proceso que permite encapsular una aplicación en un contenedor Docker, para facilitar su distribución, despliegue y escalabilidad en diferentes entornos.

Capítulo 3

Resultados

En este capítulo se describe la base de datos y sus tablas, como también las aplicaciones web desarrolladas de administrador y usuario. También se describe cómo funciona la integración continua en la plataforma de GitHub.

3.1. Base de datos

A lo largo de este punto, hablaremos de la base de datos. La estructura y las distintas tablas se muestran en la figura 3.1. En el anexo B podremos seguir los pasos para descargar y configurar todo para, si se desea, probar las aplicaciones desarrolladas a lo largo del trabajo.

3.1.1. Estructura de la base de datos

Para la creación de la base de datos necesitaremos crear un fichero denominado docker-compose.yml, el cual está situado en el directorio general del proyecto.

Este fichero permitirá la creación mediante Docker, de un entorno virtual que nos permitirá conectarnos mediante una dirección local a la base de datos, en el proyecto, se utilizará para el efecto el puerto 5432, siendo éste el WKP de TCP para conectar con el servicio de PostgreSQL. Para poder visualizar la base de datos utilizaremos una aplicación que nos permita establecer una conexión segura con este puerto.

El fichero de Docker también iniciará los procesos correspondientes a la API y a la web, puertos 3000 y 3001 respectivamente.

A continuación, hablaremos de los conceptos que necesitaremos conocer para la futura creación de la base de datos que utilizaremos.

Una base de datos como la que vamos a utilizar está compuesta por tablas, las cuales almacenan los distintos datos que en un futuro crearemos. Estos datos pueden ser de distinto tipo, como caracteres, enteros, flotantes o fechas, por ejemplo.

Estarán relacionados entre ellos mediante claves primarias y foráneas para poder ejecutar las distintas operaciones que se necesitarán al realizar nuestra futura API.

Una clave principal o primaria viene a ser el dato de nuestra tabla de distintos elementos que identifica a qué dato nos estamos refiriendo. Este elemento será de utilidad en un futuro para poder utilizar correctamente nuestra API. Por otro lado, está la clave foránea, que es un elemento de una tabla que identifica a la clave principal de la tabla anterior, donde tenemos la clave principal. En cuanto a la relación que puede haber entre estas dos claves son las siguientes, pudiendo ser utilizadas cada una de ellas dependiendo de nuestras necesidades:

1. **Relación 1 a 1:** Esta relación ocurre cuando un elemento de una tabla está inmediatamente relacionado con un solo elemento de otra de las tablas. Un ejemplo de este tipo de relación puede ser la de una persona con su DNI, ya que no puede haber más de una persona con el mismo DNI.

2. **Relación uno a muchos o muchos a uno:** Este tipo de relación ocurre cuando un elemento de una de las tablas de la base de datos está relacionado con uno o más de un elemento de otra tabla, pero no al contrario. Un ejemplo de esto podría ser la relación de libros que ha escrito un escritor, es decir, un escritor tiene una o varias novelas a la venta, pero esas novelas solo podrán ser de ese escritor y no de otro.

3. **Relación muchos a muchos:** En este caso cada elemento de una tabla se encuentra relacionado con varios elementos de otra tabla y viceversa. Este puede ser el caso de la relación existente entre títulos de películas y nombres de actores.

La figura 3.1 muestra la estructura de la base de datos, las tablas y sus relaciones.

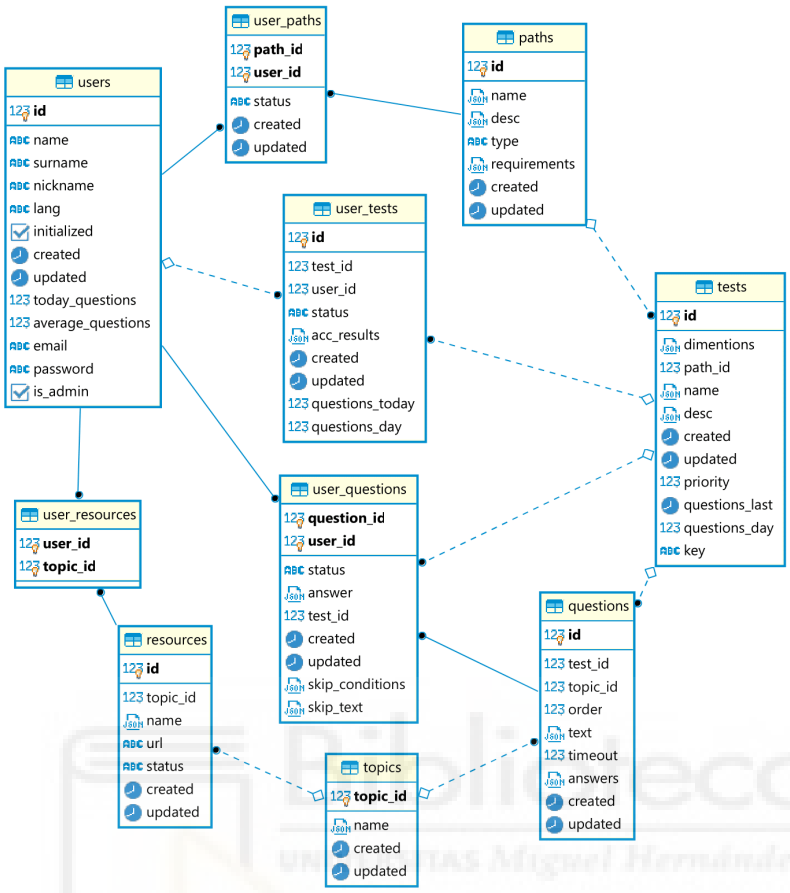

Figura 3.1: Diagrama de la base de datos.

A lo largo del capítulo, veremos el interfaz gráfico y funcionalidad de la web por la parte de administrador, esto implica los siguientes aspectos.

3.1.2. Tablas de la base de datos

Las tablas de la Figura 3.1 son las tablas que se utilizarán en nuestra base de datos. A continuación, se listarán cada una de ellas, y, posteriormente, se describirán cada una de las filas que intervienen.

CÓDIGO	DESCRIPCIÓN
users	Tabla con los datos del usuario
paths	Tabla que contendrá los datos de las temáticas
tests	Tabla donde estarán los datos de los tests
questions	Tabla que contendrá las preguntas referidas a los distintos tests
user_questions	Tabla que contendrá el estado de las preguntas, cada una de sus respuestas y el identificador del test al que hace referencia
user_tests	Tabla intermedia que contendrá los tests iniciados por un usuario y la puntuación en formato JSON del test realizado
user_paths	Tabla intermedia que contendrá las temáticas relacionadas con los usuarios.

users

La tabla de usuarios está compuesta por 6 datos del tipo VARCHAR, es decir, cadenas de caracteres que recogen los datos de un usuario como el nombre, los apellidos, un apodo, el idioma, el email y la contraseña de acceso a la web. También hay un dato booleano que permite identificar si un usuario es administrador o no. Este dato es *is_admin*, que por defecto estará inicializado a false. En la tabla también existen los datos Created, dato que tomará fecha y hora de creación de un usuario y Updated, dato que se modificará con cada actualización de los datos del usuario.

paths

En esta tabla tenemos el nombre de la temática, una pequeña descripción. Estos dos datos serán JSON, pudiendo así añadir varios idiomas a ellos. Created y Updated juegan la misma función que en la tabla de usuarios, pero definido en los datos que corresponden a *paths*

user_paths

Esta tabla relacionará a los usuarios con las temáticas mediante sus identificadores primarios, es decir, el path_id y el user_id

tests

En cuanto a la tabla de tests, si la observamos veremos que está compuesta por el identificador de la temática, el nombre del test y una breve descripción, estos últimos en formato JSON.

user_tests

Esta tabla relacionará a los usuarios con los tests que han hecho de un test mediante *user_id* y *test_id*. El dato *status* será completed una vez que el test estéx finalizado. También tenemos el dato *acc_results*, que guardará el número total de preguntas correctas frente a preguntas realizadas *(*'right'y 'total') del dato answer de la tabla de user_questions. Esto se realizará mediante una consulta SQL que realiza un reduce a datos estáticos. La siguiente sentencia SQL es la que se utiliza para realizar el reduce.

```
sql_statement = """
            UPDATE user_tests ut
            SET acc_results = (
            SELECT JSON_BUILD_OBJECT(
                'right', SUM(CAST(uq.answer->>'right' AS integer)),
                'total', SUM(CAST(uq.answer->>'total' AS integer))
            )
            FROM user_questions uq
            WHERE uq.test_id = ut.test_id AND uq.user_id = ut.
               user_id
            ), status = 'complete'
            WHERE ut.user_id = {0} AND ut.test_id = {1};
            """.format(user_id, test_id)
```

questions

También tenemos que describir la tabla de questions está formada por el identificador del test, el orden de aparición de las preguntas, la pregunta en sí, que se refiere al campo text y las respuestas, siendo este el campo de answers.

user_questions

La tabla user_questions, servirá para guradar las respuestas de un usuario una a una, es decir, cada pregunta que realice quedará almacenada en el dato *answer* y, posteriormente, cuando se finalicen las preguntas, éstas se utilizarán para calcular la puntuación total.

Finalmente, existe alguna tabla adicional, como la de topics, la de resources y la de users_resources, tablas que al comienzo del trabajo se crearon con la idea de incluir recursos que ayuden a los usuarios a tener información sobre cada una de las temáticas. Estas, como podrá observarse no han sido utilizadas en el trabajo final, pero se mantienen para la ampliación del proyecto, tal cual se comenta en las conclusiones.

3.2. Conceptos generales de la web

En esta sección veremos componentes globales de la web, así como la vista de acceso *login*, la cual es general para ambos usuarios. Los componentes podrían ser usados tanto en la aplicación de usuario como también en la aplicación de administrador, pero para hacer una distinción, se han seleccionado los que serán para administrador y los que no. En las secciones 3.3.1, se comentarán cuáles de estos componentes se han usado en la aplicación de administrador. Por otro lado, en la sección 3.4.1 comentaremos los componentes, que, por el contrario, se usarán en la aplicación de usuario. También vamos a hablar de otros elementos comúnes como.

1. Estructura de la web
2. La memoria local o *local store* utilizada para los token.
3. Estilos frontend
4. Rutas tanto de usuario como de administrador
5. Tests automáticos de *nightwatch*

3.2.1. Estructura

A lo que se refiere a la estructura de la aplicación, debería quedar parecida a la que se muestra en la figura 3.2. Un breve resumen de cada directorio podría ser el siguiente:

1. **logs:** Contendrá la salida de la consola cuando se ejecuten los *scripts* de linux.
2. **nightwatch:** Directorio que se crea al instalar el paquete de *nightwatch*.
3. **public:** Directorio que contendrá el fichero index.html general a todas las vistas.
4. **Scripts:** Directorio que contendrá 2 *scripts* para poder ejecutar todos los tests.

5. **Components:** Componentes que podrán ser utilizados en cualquier vista invocándolos. *cardView* será exclusivo para la vista de usuario en vez de la vista de administrador.

6. **Views:** Ficheros para la vista de usuario y de administrador.

7. **Store:** Directorio que contendrá los datos que se guardarán en el *local store* del navegador. También permitirá controlar el acceso a usuarios no registrados en la base de datos a las distintas vistas redirigiendo a la página de autenticación si es que no lo están.

8. **Tests:** Directorio que contendrá los tests end-to-end para la comprobación de la web, siendo éstos automáticos.

9. **Exportación a Moodle**

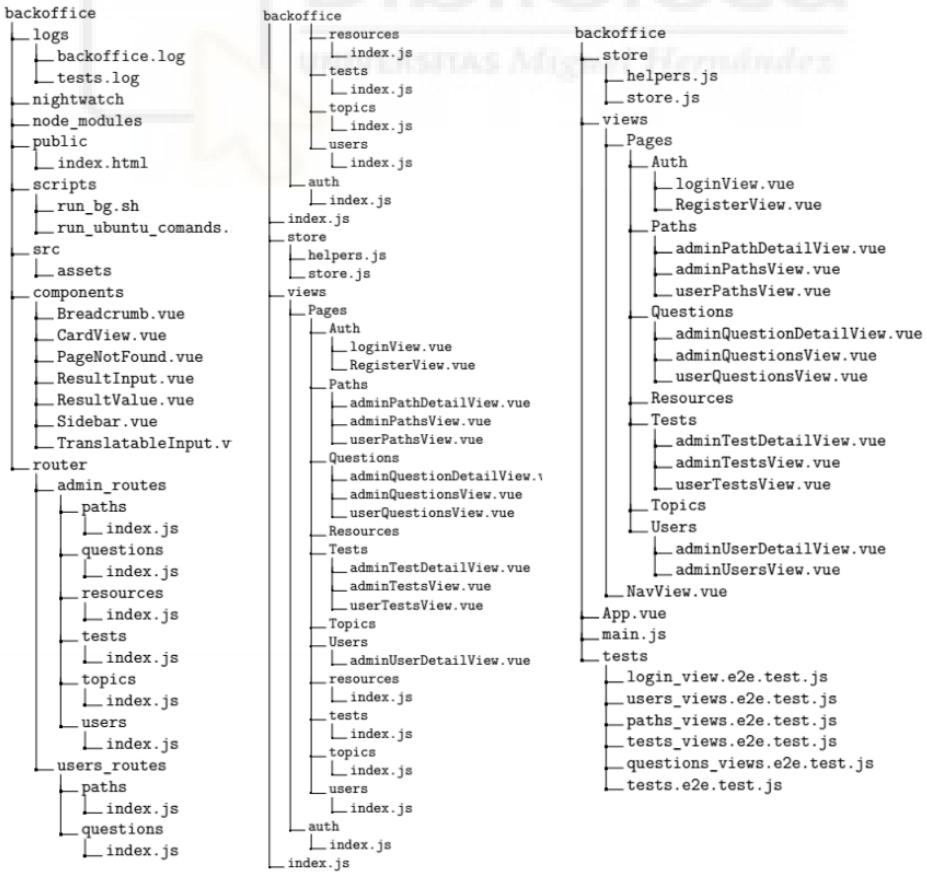

Figura 3.2: Estructura de la web.

Por último, comentar que los componentes podrán ser compartidos en ambas aplicaciones, pero que estarán pensados principalmente para una de las dos.

3.2.2. Local store

Si tenemos que hablar de la aplicación web, debemos también hablar del almacenamiento de datos en la memoria local, o *local store*. Según [3], el *local store* es una forma de almacenamiento web proporcionada por el navegador, pudiendo almacenar esta información sin fecha de expiración.

Para este proceso, se han desarrollado dos ficheros denominados helpers.js y store.js, los cuales se describen a continuación.

store.js

La primera parte de este fichero store.js se encarga de obtener un estado básico que define un usuario y lo inicializa a `null`.

```
1 import Vuex from 'vuex';
2 import axios from 'axios';
3 const getDefaultState = () => {
4     return {
5       user: null,
6     };
7 };
```

Posteriormente, se definen 2 métodos que permiten modificar el estado del usuario. El primero, denominado `SET_USER_DATA`, permite almacenar la información que le llegue desde el Backend en el *local store* de la web y permite hacer peticiones axios solo si el token JWT está presente.

El segundo método, denominado `CLEAR_USER_DATA`, permite borrar esta información del *local store* cuando el usuario se desconecte del sistema al presionar el botón de *logout*.

```
mutations: {
        SET_USER_DATA (state, userData) {
        state.user = userData
        localStorage.setItem('user', JSON.stringify(userData))
        axios.defaults.headers.common['Authorization'] = `Bearer ${
           userData.AccessToken}`
    },
    CLEAR_USER_DATA(){
        localStorage.removeItem('user')
        location.reload()
        }
}
```

Ahora mostraremos las funciones que se llamarán a la hora de conectarse *(login)* o de desconectarse *(logout)*.

```
actions: {
        async login ({ commit }, credentials) {
            const { data } = await axios.post('http://localhost
               :3000/login', credentials);
            commit('SET_USER_DATA', data);
        },
        logout ({ commit }) {
            commit('CLEAR_USER_DATA');
        },
        setUser ({ commit }, user) {
            commit('SET_USER_DATA', user);
        },

    },
    getters:{
        loggedIn(state){
            return !!state.user
        }
```

```
18        }
```

helpers.js

Para finalizar esta sección, hablaremos del otro fichero, helpers.js, que permitirá a los componentes acceder al estado del usuario y, de esta manera, podremos saber si éste está registrado o no, ya que anteriormente deberemos haber almacenado un JWT en la memoria local del navegador. En las herramientas de desarrollo, o presionando F12 en Chrome, podemos ver en la pestaña de Application las Figuras 3.3 y 3.4

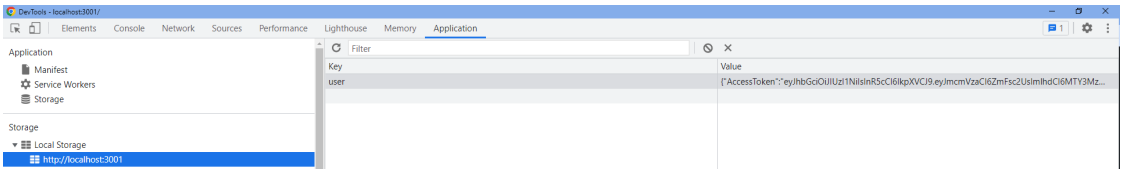

Figura 3.3: Local Storage in Chrome

Figura 3.4: userData in the local Storage

3.2.3. Vue Router

Continuando con el sistema de ficheros, ahora vamos a hablar del sistema de rutas. Este estará compuesto por distintos ficheros vue que, cada uno de ellos, contendrá en su interior una lista de rutas y, posteriormente, en un documento se incluirán por concatenación todas las rutas del trabajo. En cuanto a la estructura de rutas, si se representa de forma lógica, ésta debería ser con forma de árbol. En cuanto a la aplicación que se refiere al administrador, las rutas quedarían del siguiente modo.

```
/admin/users
/admin/paths
    /admin/paths/path_id
        /admin/paths/path_id/tests
            /admin/paths/path_id/tests/test_id
                /admin/paths/path_id/tests/test_id/questions
                    /admin/paths/path_id/tests/test_id/questions/question_id
```

Por otro lado, en la aplicación que se refiere al usuario no administrador, las rutas quedarían del siguiente modo.

```
/users/user_id/paths
└─ /users/user_id/paths/path_id/tests
   └─ /users/user_id/paths/path_id/tests/test_id
```

En los ficheros de *router*, para que la autenticación sea válida, también deberemos implementar algo de código, así, en caso de no ser un usuario válido, éste sea redirigido a la página de *login*.

```
{
        path: '/admin/paths/:path_id',
        name: 'adminPathDetailView',
        component: adminPathDetailView,
        meta: {
                requiresAuth: true,
        }
}
router.beforeEach((to, from, next)=>{
  const loggedIn = localStorage.getItem('user')
if(to.matched.some(record => record.meta.requiresAuth)&&!loggedIn)
   {
    next('/login')
  }else{
    next()
  }
})
```

3.2.4. Nightwatch tests y automatización

Considerando lo dicho al principio del capítulo, *nightwatch* es un *framework* que nos permitirá realizar tests automáticos end-to-end a nuestra aplicación de administrador.

Para esta tarea deberemos instalar *nightwatch*, posteriormente crearemos la carpeta de tests, que será la que deberemos ejecutar. Dentro de ésta, estarán contenidos los tests a las distintas páginas, los cuales se ejecutarán de forma ordenada y uno detrás del otro.

Estos test, por tanto, servirán para detectar errores en la interfaz de usuario, comprobando así el correcto funcionamiento de la aplicación. También comentar que *nightwatch* utilizará chromium en el caso de Google Chrome para ejecutar estos tests y, para poder decidir en qué componentes se deben ejecutar los tests, deberemos incluir selectores *CSS*, elementos que identifican una clase o identificador *HTML*.

Un ejemplo de test sería el que se puede ver a continuación, donde indica cuál es el test que se va a realizar y una descripción. Este es el ejemplo que presenta la página oficial de *nightwatch*. Su función es abrir un navegador, ir a la página de Escosia, en este navegador, realizar una búsqueda de *nightwatch* y finalizar la búsqueda. Una vez finalizada, el test terminará y, podremos observar que se ha completado con éxito.

```
describe('Ecosia', function() {
  // test() and specify() is also available
  it('demo test', function(browser) {
    browser.url('https://www.ecosia.org/')
      .setValue('input[type=search]', 'nightwatch')
      .click('button[type=submit]')
      .assert.containsText('.mainline-results', 'Nightwatch.js')
      .end();
  });
});
```

3.2.5. Estilos Frontend

Para nuestra aplicación, todos los estilos de la web estarán compuestos por *bootstrap*, *framework* que permitirá añadir estilos a nuestro *frontend* de forma relativamente sencilla, quitando la necesidad de utilizar *CSS* en nuestro código.

3.2.6. Login

Cuando entremos en la web, veremos una pantalla de *login*, figura 3.5, que redirigirá al usuario administrador a las vistas en las que podrá agregar, modificar o eliminar datos y, al no administrador solo se le permitirá ver las preguntas y responderlas de una en una.

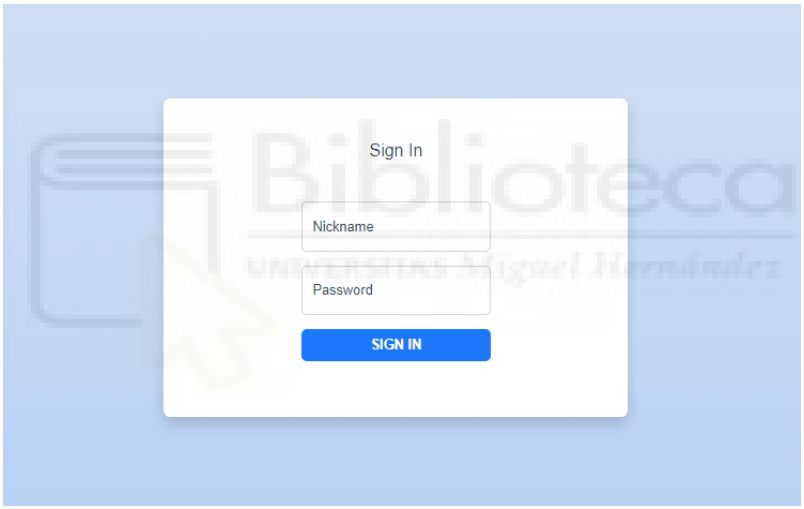

Figura 3.5: Vista de login.

3.3. Aplicación de administrador

A continuación, se describe la aplicación de administrador, que será la que permita a un usuario que es administrador controlar los datos que están en la base de datos. Esta aplicación se comunicará con la API mediante *axios*, es decir, biblioteca de JavaScript que se utiliza para realizar peticiones HTTP.

3.3.1. Componentes del administrador

Los componentes principales para esta primera web estarán compuestos por migas de pan o *breadcrumb* para la navegación, una barra lateral o *sidebar* y ResultInput, el cual invocará al ResultValue para funcionar correctamente, que a su vez invocarán, si es necesario, al componente TranslatableInput. Todos estos componentes se detallan a continuación.

BreadCrumb

Un Breadcrumb, en español, migas de pan, es, en esencia, un elemento gráfico que permite a un usuario saber en todo momento dónde se encuentra, pudiendo, si se desea, navegar a páginas de nivel superior. [9]

El funcionamiento de la siguiente función es la siguiente: Primero crearemos una ruta por defecto con el texto de "Home". Posteriormente, almacenamos en un array la ruta en la cual nos encontramos separado por barras laterales. A continuación, guardamos el primer elemento del array en una variable denominada reversePath e iteramos sobre los elementos restantes. Finalmente, lo añadimos con el método *push* al array de *crumbs* y lo devolvemos para usarlo en las distintas vistas de usuario.

```
1 getCrumbs() {
2         const crumbs = [{text: 'Home', link: '/admin/'}];
3         const pathSplit = this.$route.path.split('/');
4         const reversePath = [pathSplit.shift()];
5         for(var i=0; i< pathSplit.length; i++){
6                 const text = this.capitalizeString(pathSplit[i]);
```

```
7                   reversePath.push(pathSplit[i]);
8                   const link = reversePath.join('/');
9                   crumbs.push({text, link});
10          }
11          return crumbs;
12  }
```

SideBar

El componente del *sidebar* es muy parecido al anterior. Este componente permite navegar entre vistas, pero, a diferencia del anterior, solo permitirá navegar entre las 2 vistas principales, que son las de usuarios y las de temáticas. Una vez dentro, éstas estarán compuestas por una vista CRUD que permitirá al administrador Crear, Modificar, Editar y Leer datos de estas tablas en la base de datos, así como navegar a vistas relacionadas como los tests y las preguntas de cada uno de los tests.

Como conclusión, podemos decir que el Sidebar será una vista compuesta únicamente por *HTML*.

Componentes de input

Siguiendo con los distintos componentes, pasaremos a comentar, primero, el componente relacionado con la traducción de texto a otros idiomas, TranslatableInput. Este componente permitirá añadir o eliminar idiomas para los enunciados de los tests o las preguntas, pudiendo añadir idiomas incluidos en un array específico para esto. En el caso de que no queden idiomas por añadir, el botón relacionado con este componente se ocultará y no se podrá añadir ninguno más.

Después de éste, pasaremos a hablar del componente ResultValue, componente que permitirá crear la estructura de las respuestas, puesto que ésta es un array de elementos JSON para así poder incluir de forma sencilla las distintas respuestas para cada una de las preguntas. El fragmento siguiente es un ejemplo del formato que deben tener las respuestas.

```
[
    {
        "text": {
            "es": "por el contexto."
        },
        "result": {
            "right": 0,
            "total": 1
        }
    },
    {
        "text": {
            "es": "En ese caso se invocara al constructor por
               defecto."
        },
        "result": {
            "right": 1,
            "total": 1
        }
    },
    {
        "text": {
            "es": "No hay forma, se invocara al siguiente
               constructor escrito en el codigo."
        },
        "result": {
            "right": 0,
            "total": 1
        }
    }
]
```

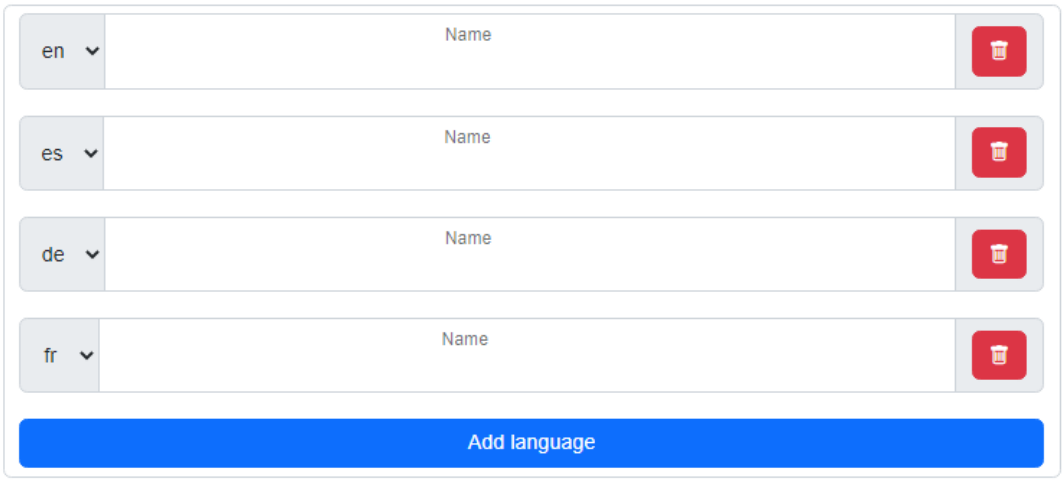

Figura 3.6: TranslatableInput para varios idiomas.

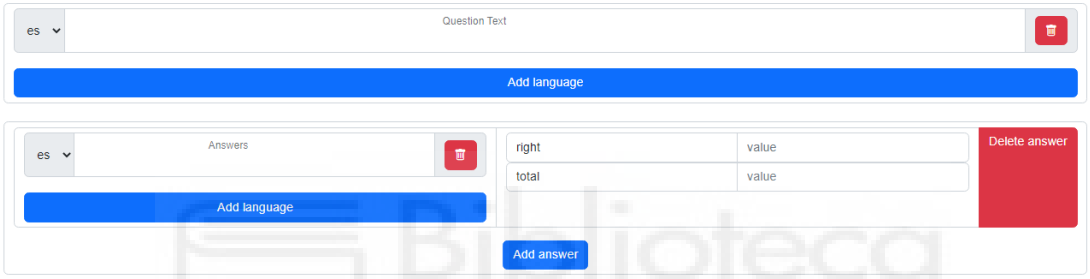

Figura 3.7: ResultInput que incluye un TranslatableInput y un ResultValue.

Para mostrar los distintos inputs, tenemos el ResultInput, que es el componente intermedio entre los dos anteriores, con el cual podemos invocar al primero y al segundo y así mostrar las distintas preguntas con sus respuestas y los idiomas incluidos en la base de datos en una sola vista. Estos componentes tendrán el aspecto que se muestra en las figuras 3.6 y 3.7.

En cuanto a la visualización de los idiomas, utilizaremos una propiedad llamada *computed*, la cual nos permite calcular de manera dinámica un valor basado en otras propiedades del componente, es decir, que no se vuelven a calcular cuando algo cambia en el componente.

3.3.2. Views

En Vue, las vistas son componentes que representan la interfaz de usuario de una aplicación. Éstas son las responsables de mostrar los datos al usuario y de recibir entradas del usuario.

En cuanto a las vistas de administrador, podemos diferenciar 2 bien definidas, la primera es la definida para ver una lista de datos en una tabla, como todos los usuarios o *paths*, éstas, en la estructura de ficheros se denominarán por admin(nombre)View.vue.

Por otro lado, tenemos las definidas como admin(nombre)DetailView.vue, que corresponden a vistas que permite ver un elemento en particular, es decir, se seleccionara el id del elemento que corresponda y éste se podrá editar, o, también, si se desea, podremos crear uno nuevo. Para estas vistas se utilizará axios, cliente HTTP para consultas a base de datos cuando sea necesario, es decir, a la hora de crear, ver, editar o eliminar datos.

En la sección 3.2.2, hemos hablado acerca de la seguridad en el *local store*. En las diferentes vistas en las cuales quisiéramos que el usuario esté registrado y se autentique, deberemos añadir el siguiente código. Este controlará lo que se desea, es decir, que un usuario no autenticado pueda acceder fácilmente a la información.

```
<script>
import { authComputed } from "@/store/helpers.js";
export default {
        computed: {
                ...authComputed,
        },
};
</script>;
```

3.3.3. Vistas de la aplicación de administrador

Una vez identificados, podremos ver los datos de los usuarios (figura 3.8) pudiendo editarlos, eliminar un usuario y todos sus datos y también añadir usuarios no administradores. Es decir, el usuario que se añada no será administrador. Los datos para el usuario administrador creado con el propósito de probar la página será *admin*, tanto en el usuario como en la contraseña.

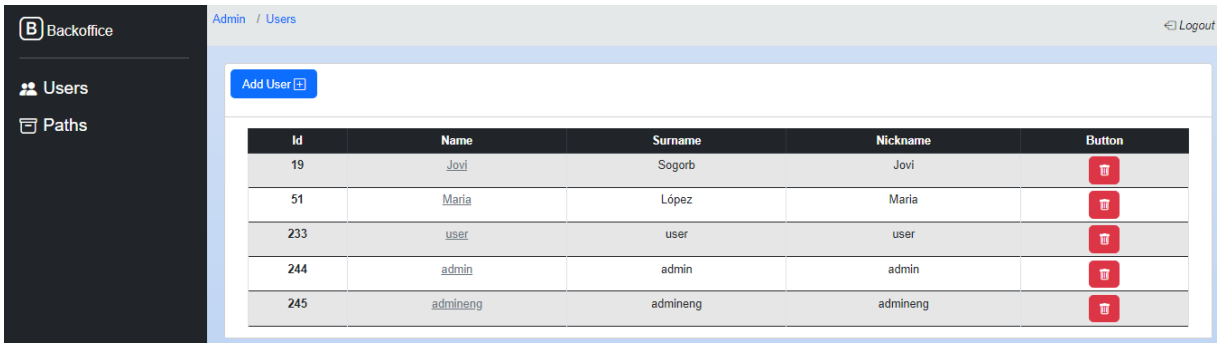

Figura 3.8: Vista de usuarios (adminUserView)

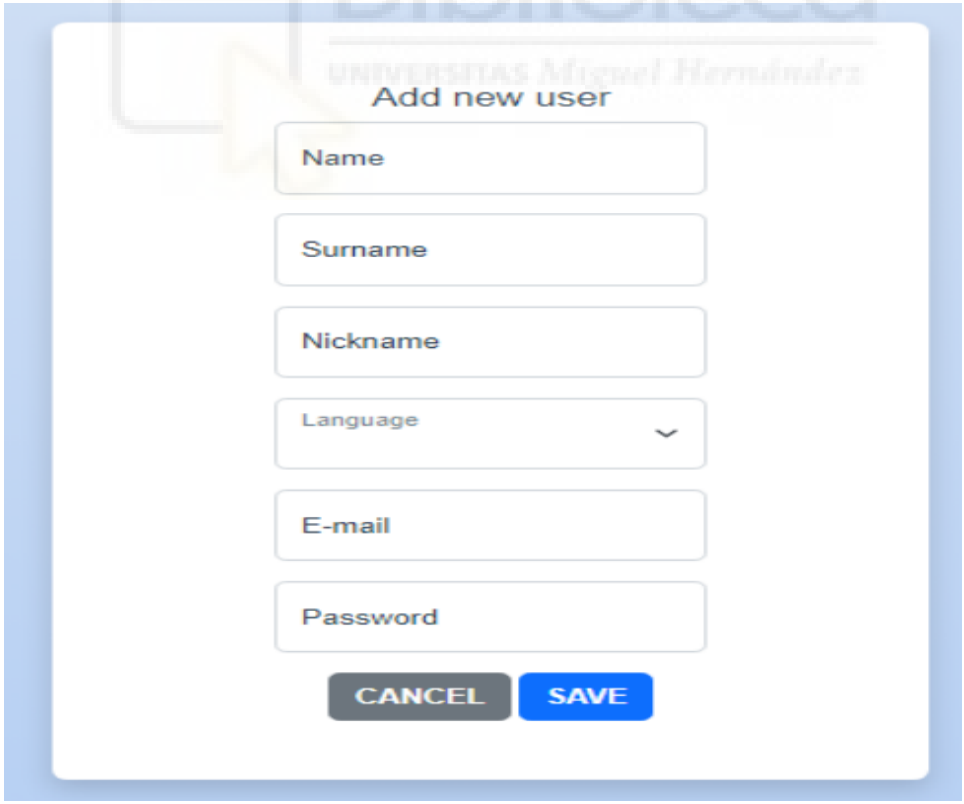

Figura 3.9: Vista de edición de usuarios (adminUserDetailView)

Para poder editar los datos de los usuarios, o, en general, cualquier dato que se presente en esta aplicación, deberemos hacer click sobre el nombre. Este nos llevará a la vista de edición, figura 3.9. Para borrar un usuario se deberá hacer click sobre el icono de la papelera. Finalmente, para añadir un nuevo usuario a la base de datos, haremos click sobre el botón "Add user". Si decidimos editar o añadir un usuario nuevo la vista será distinta, pero mediante una sentencia condicional podremos ver una u otra. Es decir, en una se mostrará la vista sin dato y en la otra se mostrará con datos en los componentes. Para lograr eso, utilizaremos el método *mounted*. La forma de proceder en las distintas vistas será la misma. Es decir, primero iremos a la página que se desee utilizar y, posteriormente se podrá manipular la información como se ha explicada en el caso de los usuarios.

Si queremos añadir preguntas a los distintos tests, estarán disponibles las vistas de preguntas con el componente ResultInput, que se mostró anteriormente en la figura 3.7. Las vistas para agregar preguntas serán visibles para los usuarios administradores solamente. Un ejemplo de vista para las preguntas podría ser la figura 3.10.

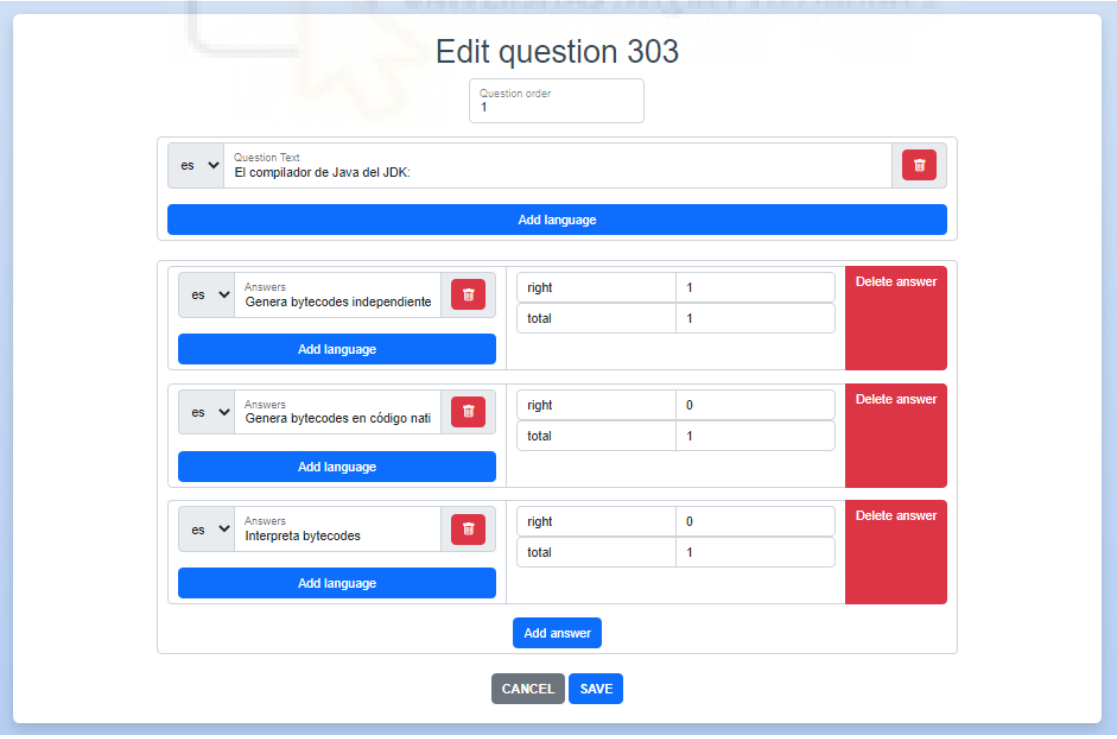

Figura 3.10: Vista de administración de preguntas (adminQuestionDetailView)

3.3.4. Exportar preguntas a Moodle

En esta sección se explica la funcionalidad de la aplicación para exportar preguntas de un test en formato XML para utilizar en Moodle. Esto permite utilizar la aplicación como editor de preguntas y poder crear cuestionarios importando el fichero generado con esta opción. Para facilitar este proceso, se incorporará un botón en la vista de preguntas relacionadas con un test que permitirá descargar un archivo llamado "test.xml", tal como se muestra en la figura 3.11.

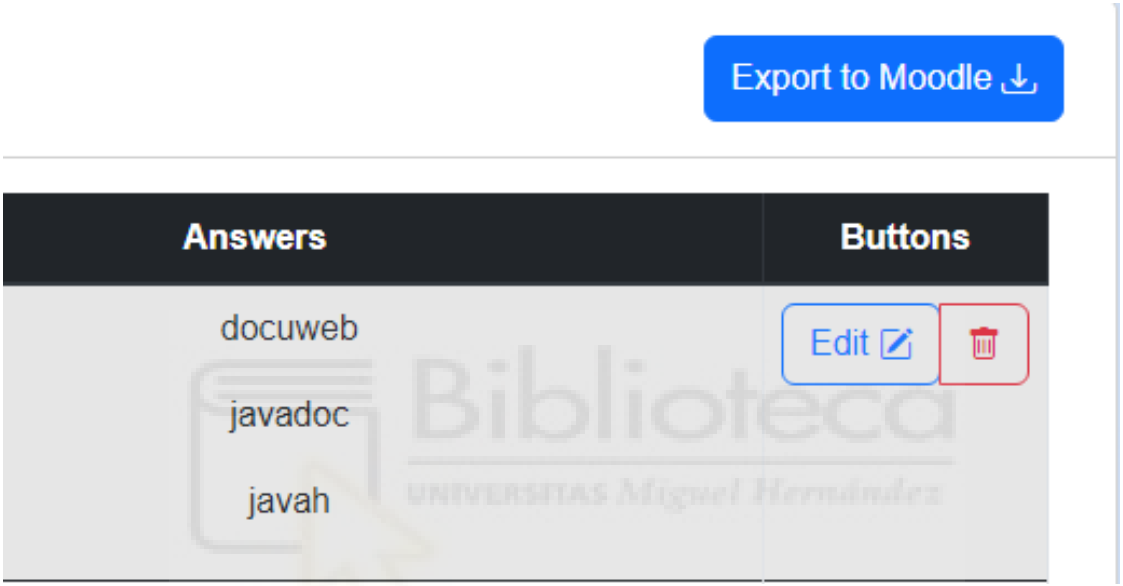

Figura 3.11: Botón para exportar preguntas a XML

Moodle permite la importación de ficheros en diversos formatos. El fichero generado utiliza el formato Moodle XML [13]. La Figura 3.12 muestra un fragmento de este fichero, donde se puede apreciar como está compuesto por 10 preguntas, y un ejemplo del formato de las preguntas.

Una vez descargado el archivo, podremos importar las preguntas a Moodle en la administración del curso. Para hacer esto, simplemente debemos importar el archivo y seleccionar las preguntas que deseamos utilizar en el cuestionario que crearemos más adelante. Para ello, según la figura 3.13, deberemos hacer clic en Importar en la sección "Banco de preguntas".

```
<?xml version="1.0" encoding="UTF-8"?>
<quiz>
  - <question type="multichoice">
      - <name>
            <text>Question 303</text>
        </name>
      - <questiontext format="html">
            <text>El compilador de Java del JDK:</text>
        </questiontext>
      - <answer format="html" fraction="100">
            <text>Genera bytecodes independientes de la plataforma.</text>
          - <feedback format="html">
                <text/>
            </feedback>
        </answer>
      - <answer format="html" fraction="0">
            <text>Genera bytecodes en código nativo según la plataforma de desarrollo.</text>
          - <feedback format="html">
                <text/>
            </feedback>
        </answer>
      - <answer format="html" fraction="0">
            <text>Interpreta bytecodes</text>
          - <feedback format="html">
                <text/>
            </feedback>
        </answer>
    </question>
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
  + <question type="multichoice">
</quiz>
```

Figura 3.12: Formato XML del fichero descargado

Figura 3.13: Importar fichero XML a Moodle

Cuando estemos dentro de la ventana 'Importar', tendremos que elegir el formato del archivo, que como hemos comentado es Moodle XML, y seleccionar el fichero test.xml, tal como se muestra en la figura 3.14.

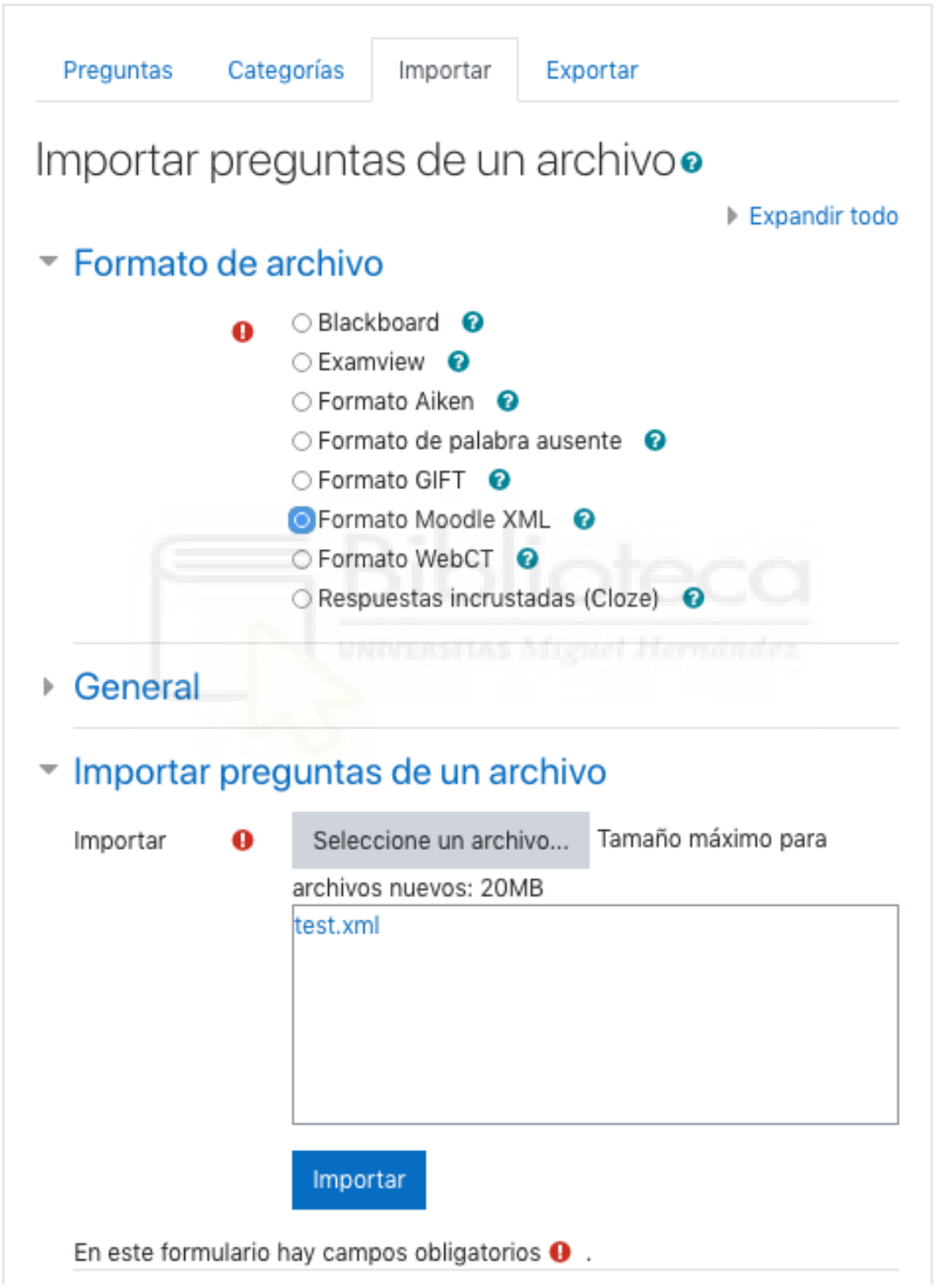

Figura 3.14: Importar fichero XML a Moodle

Para completar el proceso de importación, haremos clic en el botón 'importar' y se mostrará la Figura 3.15 y la Figura 3.16, lo que indicará que se han importado correctamente.

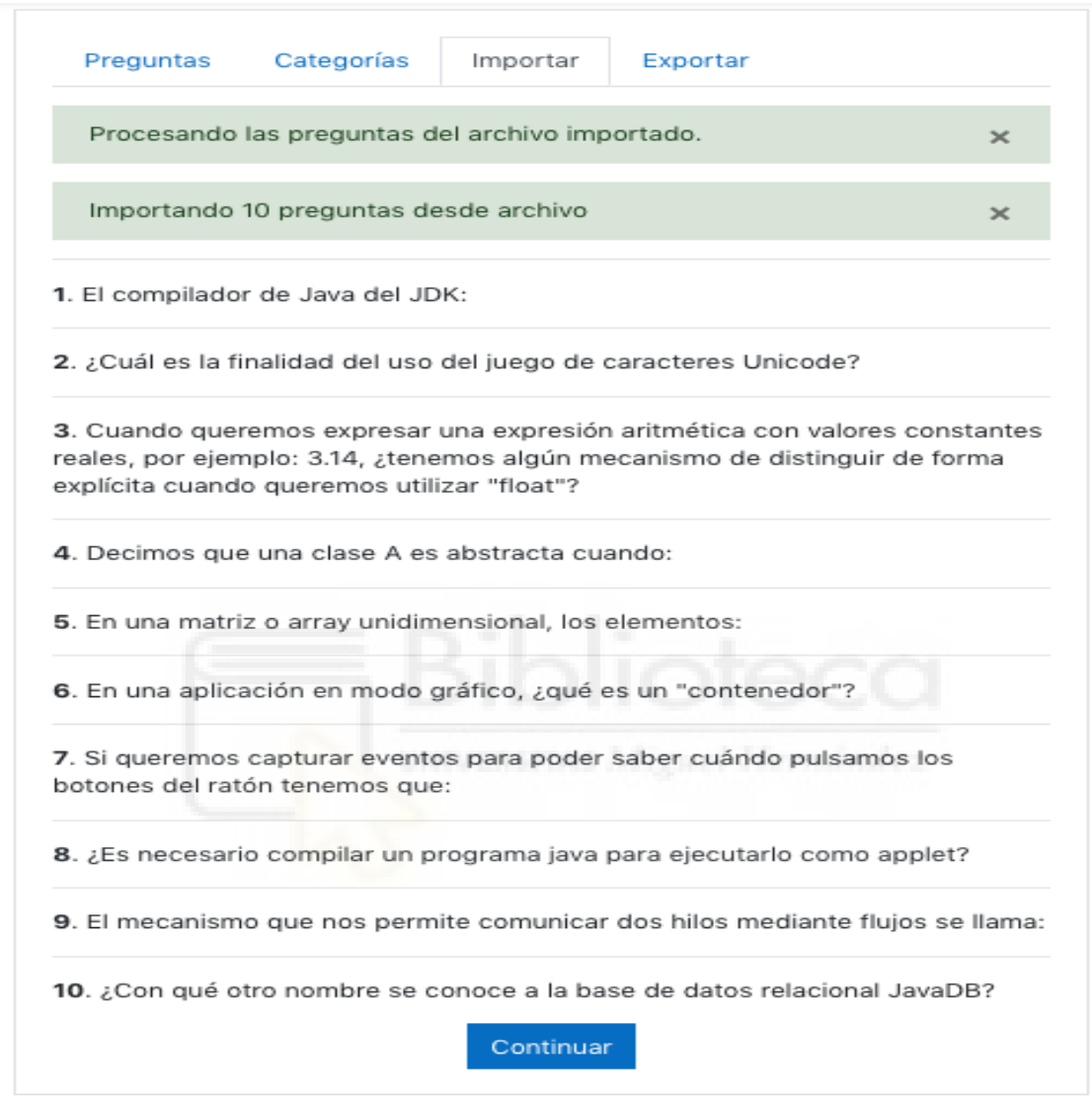

Figura 3.15: Importar fichero XML a Moodle

Figura 3.16: Banco de preguntas

En resumen, al incorporar un botón de descarga en la vista de preguntas relacionadas con un test, podremos exportar fácilmente estas preguntas en formato XML y utilizarlas en Moodle. Una vez importadas, podrán ser seleccionadas y utilizadas en un cuestionario de manera rápida y sencilla.

Creación del cuestionario

Una vez que hayamos importado las preguntas al banco de preguntas de Moodle, podremos crear una nueva actividad de tipo cuestionario. Para hacer esto, seguiremos los pasos indicados en las figuras 3.17 y 3.18.

Figura 3.17: Añadir cuestionario

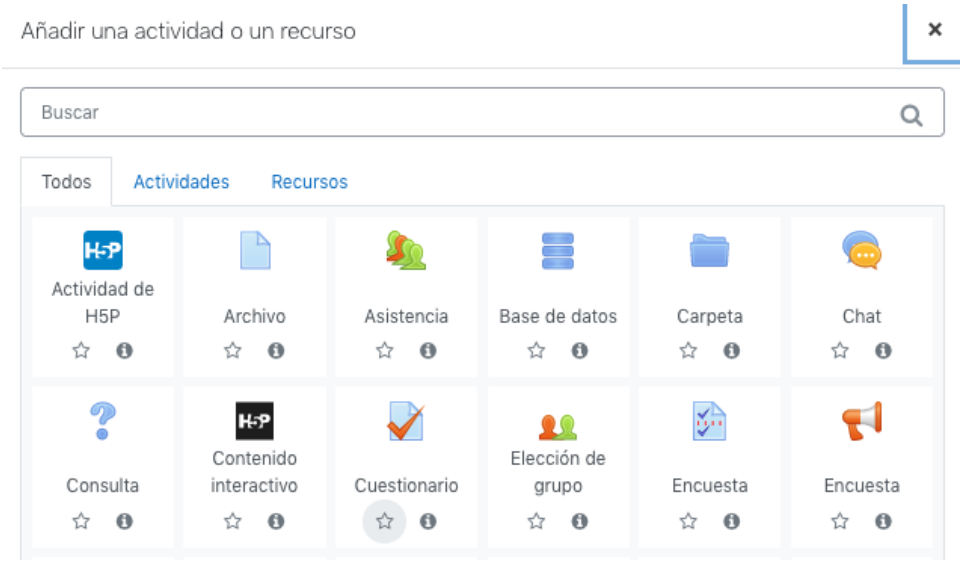

Figura 3.18: Añadir cuestionario (cont)

Luego, completaremos los campos requeridos por Moodle, como el nombre y la descripción del test. A continuación, podremos incluir las preguntas previamente importadas del banco de preguntas en nuestro test, tal como se muestra en la figura 3.19.

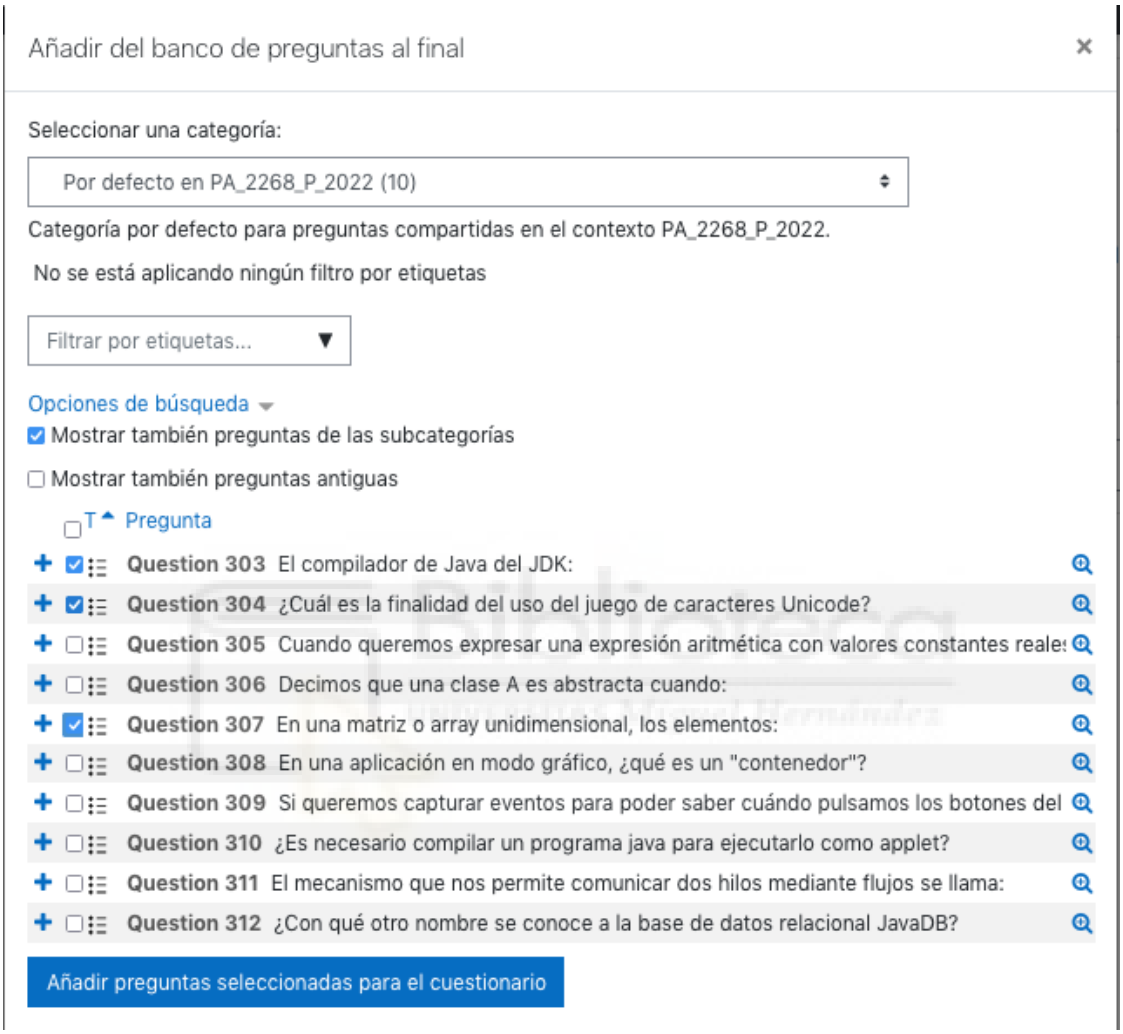

Figura 3.19: Añadir cuestionario (cont)

Una vez que hayamos configurado nuestro test y añadido las preguntas, podemos acceder a él y completar las preguntas para verificar que funciona correctamente. Es importante destacar que, en este ejemplo, se han seleccionado solo tres preguntas, tal como se muestra en la figura 3.20

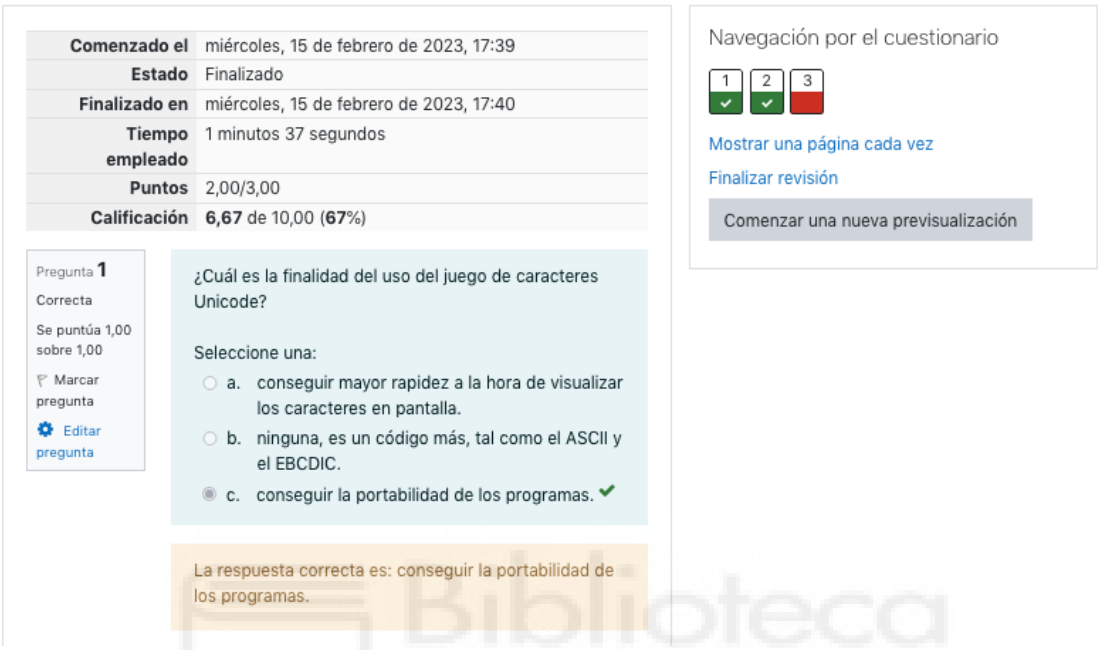

Figura 3.20: Respuestas de las preguntas

En conclusión, después de importar las preguntas al banco de preguntas, podemos crear un nuevo cuestionario en Moodle y añadir las preguntas deseadas para que los usuarios puedan completarlas. Siguiendo estos sencillos pasos, podremos crear fácilmente un cuestionario personalizado y ajustado a nuestras necesidades educativas.

3.4. Aplicación de usuario

En esta sección, vamos a ver los apartados que definen la aplicación de usuario, utilizada por los usuarios que no son administradores. Anteriormente mencionado, la web será la misma y se decidirá a qué páginas redirigir mediante un dato booleano que define quien es administrador y quien no lo es.

Por lo tanto, los conceptos de almacenamiento, de estilos y de rutas serán completamente los mismos, pero, por el contrario, los componentes a utilizar y las vistas, serán completamente distintas ya que el usuario que no es administrador no deberá tener acceso a la misma información que el que sí lo es.

Por ello, comentaremos las vistas de usuario, que serán 3, y también, los componentes que hemos utilizado a la hora de maquetarlas.

3.4.1. Componentes relacionados con el usuario

En esta parte del proyecto, tenemos tres componentes principales. Primero, la barra de navegación superior, la cual permitirá al usuario volver a la página de inicio definida para éste. Seguidamente, tendremos un *cardView*, que se mostrará en la vista de las temáticas y en la vista de los test. Finalmente, mostraremos una vista de preguntas, las cuales tendrán un orden predefinido por el usuario administrador, que mostrará una a una, siguiendo un orden ascendente, cuando el usuario elija una respuesta.

3.4.2. Vistas de usuario

Las 2 vistas principales de usuario serán la de temáticas y las de los tests relacionados con esas temáticas. Estas 2 vistas estarán formadas por un *cardView* que mostrará todos los elementos en la base de datos y, por un enlace que se actualizará dependiendo de en cuál de las vistas estemos.

Una vez claras estas vistas, pasaremos a la de las preguntas, que es una vista independiente que se actualiza cada vez que pulsamos una respuesta a una pregunta, pasando a la siguiente pregunta. Para este efecto, se han utilizado dos sentencias de bases de datos, una que obtiene todas las preguntas en un test por orden ascendente y, otra, que obtiene la última pregunta. Cuando el índice de la siguiente pregunta es mayor que el de la última, se enviará al usuario a la página de tests mostrando su puntuación total utilizando el método *reduce* para sumar todos los datos de ese test.

En la figura 3.21, podemos observar cómo se ve la vista de las temáticas para un usuario en concreto que no es administrador. Si hacemos *click* en el botón de *Tests*, pasaremos a ver los tests que corresponden a esa temática para un ese usuario.

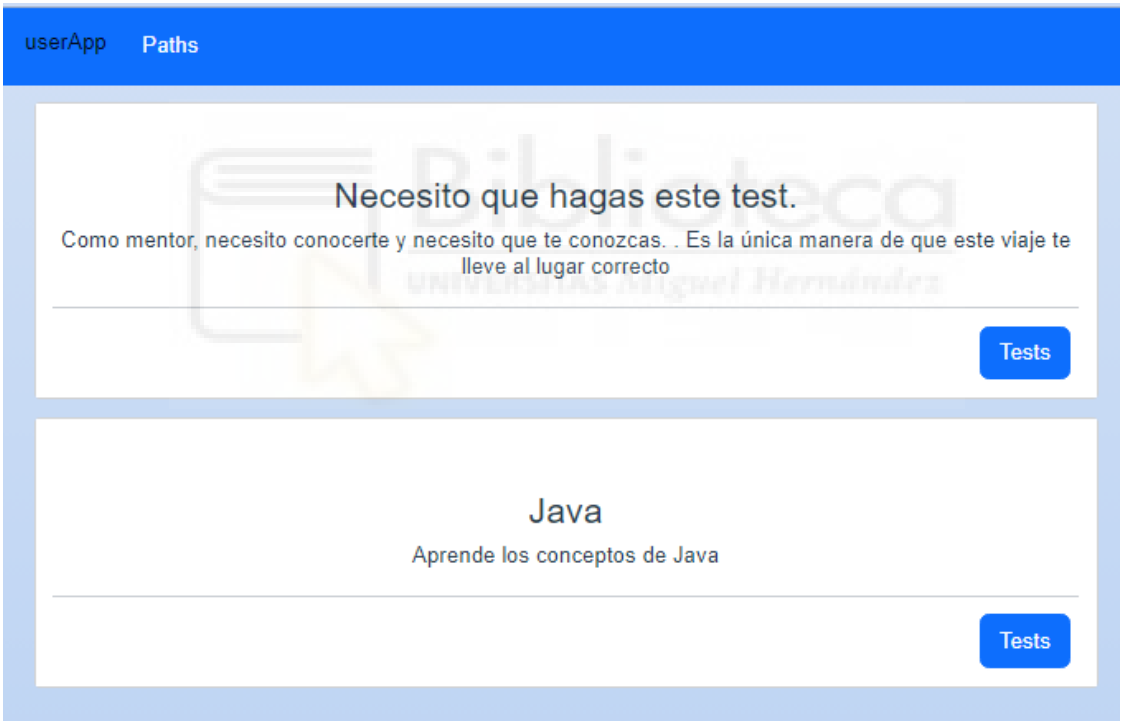

Figura 3.21: Vista de temáticas del usuario (userPathsView)

También se puede observar en las figuras 3.22 y 3.23 el resultado de que un usuario supere o no un test, respectivamente.

Figura 3.22: Test completado con éxito

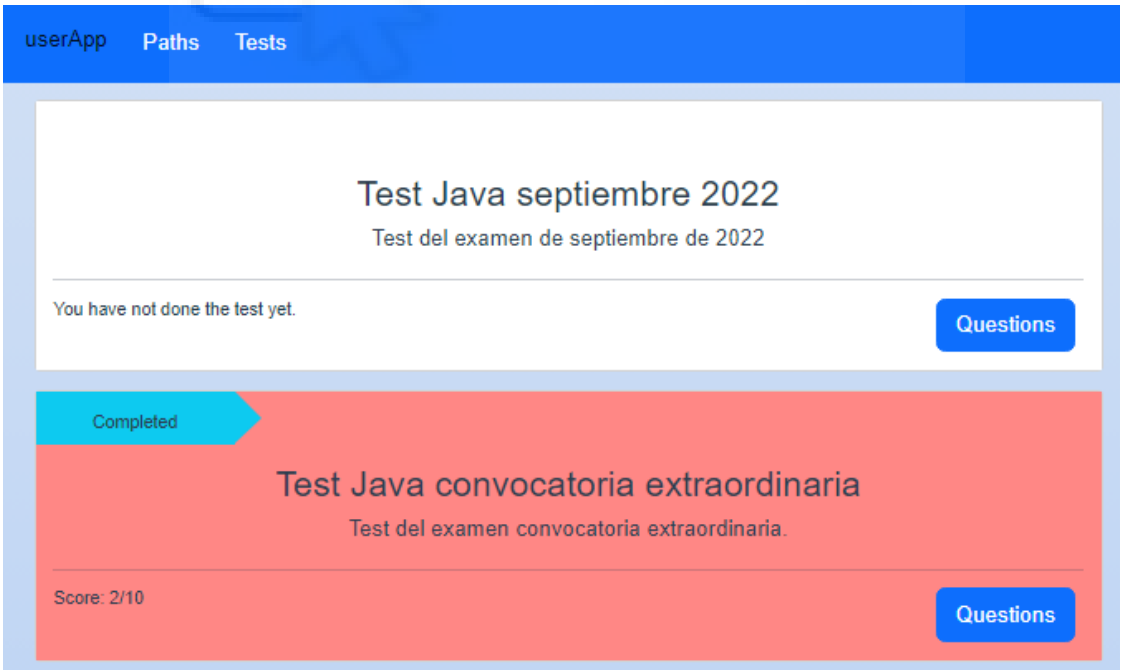

Figura 3.23: Test fallido

En la figura 3.24, podemos observar cómo se mostrarán las preguntas en cada uno de los tests, y, si el usuario hace click en una de las respuestas, ésta vista pasará a la siguiente pregunta.

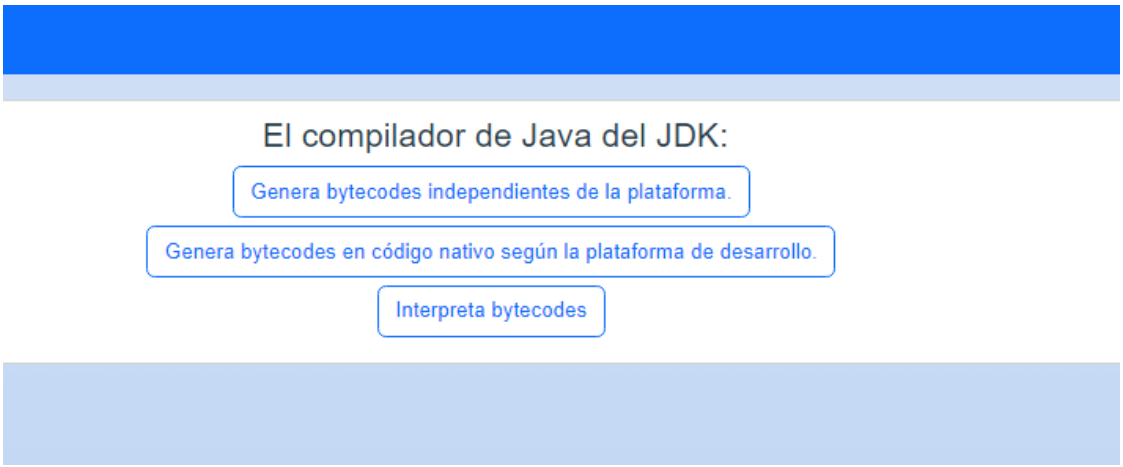

Figura 3.24: Vista de preguntas del usuario (userQuestionsView)

3.5. GitHub

Una vez comentadas las dos principales aplicaciones que se desarrollarán en cuanto a la parte de cliente se refiere, pasaremos a hablar de una herramienta muy utilizada en el mundo del desarrollo. Esta herramienta es una plataforma web para el control de versiones, la cual, utiliza Git, un sistema de control de versiones distribuido. Como está basada en el control de versiones, permite a los desarrolladores compartir los cambios en el código como así también almacenar los proyectos realizados. [7]

Entre sus técnicas, se encuentra la integración contínua, proceso automatizado que permite probar de forma continua los cambios en el código fuente. La idea principal de este proceso es solucionar problemas lo antes posible y así poder evitar la acumulación de errores.

Para llevar a cabo esto, se han desarrollado dos *shellscript* que permitirán, por un lado, instalar las dependencias necesarias para el correcto funcionamiento de los tests end-to-end, que se ejecutarán cuando un nuevo código sea subido al repositorio de GitHub.

3.5.1. Integración continua

En la sección anterior hemos visto que la integración continua nos permite detectar prematuramente errores en el código fuente. Para ver cómo funciona esto, deberemos crear un fichero con extensión yml.

3.5.2. Shellscript para la integración continua

El siguiente código se corresponde con los *scripts* que se ejecutarán cuando un *commit* se haga en GitHub y nos permitirá ver si la aplicación de administrador funciona como es debido.

El primero será el de instalar paquetes como *headless-chromium*, navegador que utilizará *nightwatch* para ejecutar los tests. También se instala la base de datos PostgreSQL, con la que se pretende restaurar la base de datos cada vez con la intención de hacer las pruebas y dejarla tal y como estaba. El siguiente script instala todo lo comentado.

```
1  #!/bin/bash
2  echo "Installing postgres on Ubuntu"
3  cd "../../api/backup/"
4  mkdir -p ./logs
5  echo "  - Installing postgres"    && sudo apt-get update &&
      sudo apt-get install -qq postgresql-client
      >> ./logs/postgres_logs.log
6  echo "  - Create database"        && psql postgresql://
      postgres:changeme@localhost:5432 -c "CREATE DATABASE \"tfg
      -db\"" >> ./logs/postgres_logs.log
7  echo "  - Loading data for tests"  &&  export PGPASSWORD=
      changeme && psql -q -h localhost -p 5432 -U postgres -d
      tfg-db < tfg-db.sql
8  psql postgresql://postgres:changeme@localhost:5432 \dt
9  echo "Installing headless chromium"
10 mkdir -p ./logs
11 echo "  - Installing dependencies..." &&  sudo apt-get
```

```
    install -y libappindicator1 fonts-liberation > ./logs/
    chromium_logs.log
12  echo "  - Downloading headless Chromium" && wget -q wget
    https://dl.google.com/linux/direct/google-chrome-
    stable_current_amd64.deb >> ./logs/chromium_logs.log
13  #echo "  - Installing dependencies..." &&  sudo apt-get
    install -f >> ./logs/chromium_logs.log
14  echo "  - Installing headless Chromium" && sudo dpkg -i
    google-chrome-stable_current_amd64.deb >> ./logs/
    chromium_logs.log
```

Por otro lado, en el siguiente *script* instalaremos todos los paquetes necesarios para el correcto funcionamieneto de la API y de las dos aplicaciones. Ejecutando los tests de la aplicación de administrador una vez las aplicaciones se hayan iniciado. Cuando los tests hayan finalizado, se detendrán la API y las aplicaciones mediante *kill*.

```
1   #!/bin/bash
2   echo "Starting API"
3   cd "../../api"
4   mkdir -p ./logs
5   echo "  - Enabling virtual environment" && virtualenv -p
    python3 venv && source "venv/bin/activate" > ./logs/api.
    log
6   echo "  - Installing dependencies" && pip install -r
    requirements.txt  >> ./logs/api.log
7   echo "  - Starting server" && python3 app.py >> ./logs/api.
    log &
8   echo "  - Server running"
9   sleep 10
10  echo ""
11  echo "Starting Backoffice"
12  cd "../backoffice"
13  mkdir -p ./logs
14  echo "  - Installing vue-cli-service" && npm install -g @vue/
```

```
    cli-service > ./logs/backoffice.log
15  echo "  - Installing dependencies" && npm install --legacy-
    peer-deps --loglevel=error >> ./logs/backoffice.log
16  echo "  - Starting service" && npm run serve >> ./logs/
    backoffice.log &
17  sleep 20
18  until $(curl --output /dev/null  --head --fail http://
    localhost:3001); do
19      printf '.'
20      sleep 5
21  done
22  echo ""
23  echo "Starting tests"
24  cd "../backoffice"
25  echo "  - Running tests" && npx nightwatch tests/tests.e2e.
    test.js --headless --reuse-browser
26  sleep 10
27  echo "Stopping services"
28  kill $(ps | grep npm | awk {'print$1'})
29  kill $(ps | grep node | awk {'print$1'})
30  echo "  - Backoffice stopped"
31  kill $(ps | grep python3 | awk {'print$1'})
32  echo "  - API stopped"
```

Para finalizar esta sección, veremos el fichero que ejecutara los dos *scripts* anteriores en un *workflow*, cada vez que se produzca un cambio.

```
1  name: backoffice-ci
2  run-name: ${{ github.actor }}
3  on: [push]
4  jobs:
5    build:
6      runs-on: ubuntu-latest
7      services:
```

```
8         postgres:
9           image: postgres
10          ports:
11            - 5432:5432
12          env:
13            POSTGRES_PASSWORD: changeme
14      steps:
15        - uses: actions/checkout@v3
16        - uses: actions/setup-node@v3
17        - run: cd ../quizer/backoffice/scripts && bash
             run_ubuntu_comands.sh
18        - run: cd ../quizer/backoffice/scripts && bash
             run_bg.sh
```


3.5.3. GitFlow

Por otro lado, GitFlow es una metodología de trabajo basado en la gestión de ramas para el control de versiones de un proyecto. En ésta metodología, se tiene una rama principal "master", la cual tiene código estable. Ésta rama es la que se utiliza para poner en marcha distintas versiones del software que se esté desarrollando.

Capítulo 4

Conclusiones

A la vista de los resultados presentados en el capítulo 3, se puede comprobar que se ha llevado a cabo lo propuesto. Se ha desarrollado una web para la realización de cuestionarios tipo test de forma online y se ha implementado una API para la comunicación con una base de datos implementada en PostgreSQL.

La web se ha llevado a realizado mediante el *framework* de programación Vue.js, que de forma local permitirá a un usuario que quiera acceder a éstas poder hacerlo a través del navegador con dirección local, activando previamente los contenedores Docker y siguiendo el anexo B. Esta web está compuesta por una parte, la cual corresponde a usuarios administradores, y, por otra, que corresponde a usuarios que no son administradores.

Otro objetivo que se ha logrado es el de crear contenedores Docker para la puesta en marcha en cualquier máquina de estas aplicaciones. Lo que permitirá a cualquier usuario que lo desee probarlas.

También podemos observar que una vez que las aplicaciones están en funcionamiento, se pueden crear, modificar, leer y borrar datos, utilidades que se perseguían. En cuanto, a las webs, para su acceso, como se ha nombrado a lo largo del proyecto, se deberá acceder con un usuario registrado, en el caso de la web para administrador, usaremos uno de los usuarios creados para esta tarea, es decir, *admin* o *admineng*, si se desea ver la página en inglés y, para acceder a la web de usuario, deberémos usar las credenciales *user* en ambos campos de la figura 3.5.

Por otro lado, podremos crear un usuario si se desea con nuestros datos para acceder a la web de usuario, pero previamente se ha creado uno con el cual nos permitirá acceder a esta web y probarla.

Finalmente, para el desarrollo se ha utilizado GitHub, que fue muy conveniente ya que proporciona un control de versiones con el cual podemos volver hacia atrás en nuestro código y ver estas versiones pasadas, teniendo esta herramienta la metodología de GitFlow, la cual se ha usado para el desarrollo de la aplicación.

4.1. Líneas de mejora

En cuanto a las líneas de mejoras abiertas, por un lado, tenemos las tablas de la base de datos que no se han usado. Estas servirán para introducir la teoría de la cual estarán compuestos los tests. La tabla de *topics* será la que utilizaremos para añadir el título a esta teoría. Seguidamente, la tabla de *resources* contendrá una URL que permitirá al usuario buscar esta información en Internet y, finalmente, la tabla *user_resources* será la que relacione estas dos tablas con el usuario.

Otra mejora sustancial del trabajo sería la documentación de la API mediante Postman o alguna aplicación similar, como lo es Swagger, herramienta utilizada para la documentación de API.

Para finalizar con las mejoras, también se puede actualizar la integración continua de GitHub de forma que se ejecute incluso cuando solo se han detectado cambios en el código, de forma que no permita subir a GitHub código incorrecto y sin comprobar previamente. Para llevar esto a cabo, habría que instalar y configurar herramientas que se utilicen para este propósito como pueden ser Jenkins, disponible en [8], o CircleCI, disponible en [4], entre otras.

Bibliografía

[1] Visual studio code, 2022. [Online]. Available: https://code.visualstudio.com/.

[2] Apipheny. Api headers: Definition, best practices, and examples, 2020. [Online]. Available: https://apipheny.io/api-headers/.

[3] D. Boateng. Javascript local storage explained, Jan 2022. [Online]. Available: https://dev.to/dboatengx/javascript-local-storage-explained-1di6.

[4] CircleCI. Circleci, 2022. [Online]. Available: https://circleci.com/.

[5] Docker. Docker, 2022. [Online]. Available: https://www.docker.com/.

[6] Docker. Docker on linux, 2022. [Online]. Available: https://docs.docker.com/desktop/install/linux-install/.

[7] GitHub. Github. [Online]. Available: https://github.com/FranGarciaLopez/quizer.

[8] Jenkins. Jenkins for ci, 2022. [Online]. Available: https://www.jenkins.io/.

[9] KINSTA. Breadcrumbs in web design: Examples and best practices. [Online]. Available: https://www.smashingmagazine.com/2009/03/breadcrumbs-in-web-design-examples-and-best-practices/.

[10] M. learn. Instalacion wsl 2 para windows. [Online]. Available: https://learn.microsoft.com/es-es/windows/wsl/install-manual.

[11] Mozilla Developer Network. Http status codes, 2019. [Online]. Avaliable: https://developer.mozilla.org/es/docs/Web/HTTP/Status.

[12] SuperTokens. What is jwt (json web token)?, 2022. [Online]. Available: https://supertokens.com/blog/what-is-jwt.

[13] F. M. XML. Formato moodle xml, 2022. [Online]. Available: https://docs.moodle.org/all/es/Formato_Moodle_XML.

Anexos

Anexo A

Acrónimos

Siglas	Significado
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
JWT	JSON Web Token
JSON	Java Object Notation
RESTAPI	Representational State Transfer Application Programming Interface
SO	Sistema Operativo
SQL	Structured Query Language
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
VSC	Visual Studio Code
WKP	Well-Known Port
WSL	Windows Subsystem for Linux

Anexo B

Manual de instalación

En este anexo veremos como instalar y poner en marcha todos los servicios necesarios para poder utilizar la web. Para ello necesitaremos instalar algunas herramientas, como lo es Docker, WSL y Visual Studio Code, así como realizar cierta configuración inicial. Cabe indicar que los pasos de instalación se han realizado en Windows y que estos podrían variar dependiendo del SO que se utilice.

B.1. Instalación de Docker

Lo primero que deberémos hacer es instalar Docker desde la página oficial [5]. Esta aplicación nos permitirá virtualizar los paquetes que necesitamos en unos contenedores que ejecutarán tanto la base de datos Postgre como la API y la web.

Si nos encontramos en Windows, a la hora de instalarlo nos dejará seleccionar mediante un *checkbox* si queremos utilizar WSL 2. Para éste proyecto se ha seleccionado dicha opción. Posteriormente, Docker nos solicitará que reiniciemos el sistema para la correcta instalación. Docker también se puede instalar en Linux desde [6]

B.2. Instalación de WSL2

Por otro lado, para instalar WSL2, máquina virtual que permite ejecutar una distribución de Linux en windows, deberémos ir a una consola de Powershell de windows, ejecutar el siguiente comando.

```
wls.exe --install -d Ubuntu-20.04
```

A continuación, deberemos seguir el tutorial proporcionado por Windows [10]. Este tutorial nos proporcionará la guia necesaria para configurar y utilizar correctamente el Windows Subsystem for Linux 2 (WSL 2).

B.3. Instalación de VSCode

Una vez instalado todo lo anterior, podrémos ejecutar una consola de Ubuntu en windows. Para la posterior visualización del código, deberemos instalar un editor de código como lo es Visual Studio Code desde [1], pudiendo usar cualquier otro con características similares como Atom, por ejemplo.

B.4. Configuración inicial

En la consola de WSL2 los comandos de git vendrán por defecto, por lo que no deberémos instalar ningún paquete adicional para su uso. Una vez tengamos la consola abierta y en el directorio en el que se desee crear una copia.

```
git clone https://github.com/FranGarciaLopez/quizer.git quizer
```

Para abrir el proyecto con la consola de Linux primero nos dirigiremos a la ruta en la cual se haya creado la copia del proyecto `cd quizer/` y, posteriormente haremos click donde se indica en la Figura B.1 y seleccionar la distribución deseada, en este proyecto la 20.04

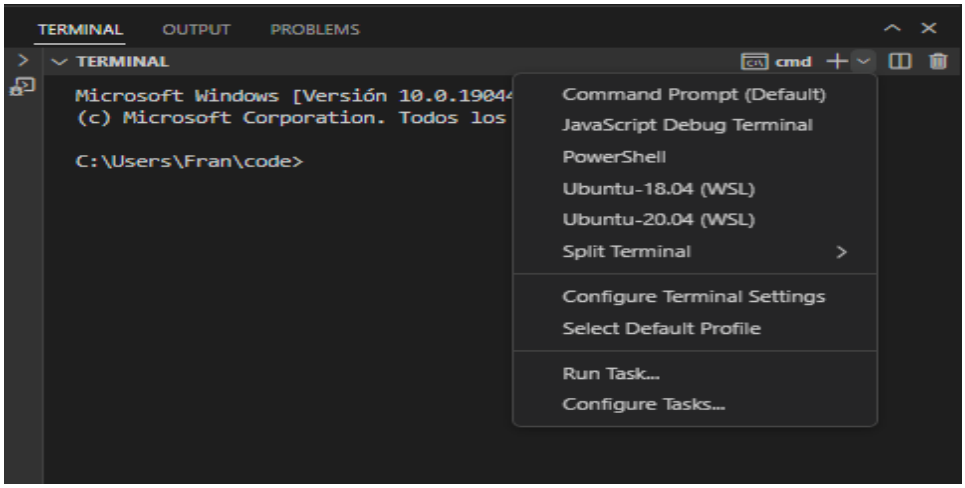

Figura B.1: WSL

Una vez abierta la consola pertinente, podemos observar que el directorio tiene un fichero que se denomina docker-compose.yml de forma global. Los contenedores Docker se crearán mediante el comando

```
docker-compose build
```

Y, posteriormente, para iniciar estos contenedores utilizaremos

```
docker-compose up
```

Estos comandos iniciarán tanto la base de datos como los dos servicios, la API y la web. A partir de este punto, podrémos probar la web, para ello, a continuación se indica como usarla.

B.5. Ejecutar script SQL en PostgreSQL

Cuando tengamos creados los contenedores de Docker y puestos en marcha, deberemos ejecutar el script SQL, el cual se encuentra en la raíz del proyecto. Para esto, en la consola de Linux que hemos habilitado anteriormente, primero crearemos la base de datos y, posteriormente, ejecutaremos el script con los siguientes comandos. Primero pasaremos a crear la base de datos en el contenedor docker iniciado anteriormente.

```
docker exec -i postgres_container psql
postgresql://postgres:changeme@localhost:5432
-c "CREATE DATABASE \"tfg-db\""
```

y, posteriormente insertaremos las tablas y los datos mediante.

```
docker exec -i postgres_container psql
-q -h localhost -p 5432 -U postgres -d tfg-db
< tfg-db.sql
```

B.6. Ejecutar tests end-to-end

A partir de este momento, podrémos usar los servicios creados como la API, la web o los tests end-to-end. Para ejecutarlos, irémos al directorio "backoffice", y usaremos el comando `npm test`. Este comando buscará en package.json la linea completa que debe ejecutar y podrémos ver como estos se pasan uno a uno.