Facultad de Ciencias

LA/BORATORY: DESARROLLO DE UNA PLATAFORMA PARA LA CREACIÓN Y EJECUCIÓN DE TESTS A/B

(LA/Boratory: Development of a platform for the creation and execution of A/B tests)

Trabajo de Fin de Grado
para acceder al título de

GRADO EN INGENIERÍA INFORMÁTICA

Autora: Marta Obregón Ruiz

Director: Pablo Sánchez Barreiro

Co-Director: Guillermo Ménguez Álvarez

Junio - 2022

Índice de Contenido

Índice de Figuras

Índice de Tablas

1. Introducción, Motivación y Objetivos

1.1 Introducción

Este Trabajo de Fin de Grado se ha desarrollado en colaboración con el programa *Observatorio Tecnológico* de la empresa *HP SCDS* (*HP Solutions Creation & Development Services*), una empresa sita en León que forma parte del grupo empresarial *HP*. *HP*, anteriormente conocida como *Hewlett-Packard*, es una multinacional que se centra en la fabricación y venta de computadores personales, periféricos e impresoras, siendo líder en este último apartado a nivel mundial. Dentro de *HP*, la empresa *HP SCDS* se especializa en el desarrollo de firmware y software para impresoras 2D y 3D. *HP SCDS* posee un programa llamado *Observatorio HP* cuyo objetivo es acercar a los alumnos al mundo empresarial mediante su participación en diversos proyectos ofertados a través de este programa. Algunos de estos proyectos son de interés para la propia empresa, que espera poder continuarlos y explotarlos, pero también se ofertan proyectos que puedan ser beneficiosos para la sociedad o les permitan explorar nuevas áreas. Dentro de este programa colaboran diferentes universidades, como la de Universidad de Santiago de Compostela, la Universidad de Castilla-La Mancha o la Universidad de Cantabria, entre otras.

Dentro de dicho programa, se ofertaba el proyecto que constituye este Trabajo Fin de Grado. El proyecto se denominaba *LA/Boratory*, y su objetivo era permitir la gestión de experimentos basados en pruebas A/B. Las pruebas A/B[1] son experimentos cuya misión principal es comparar dos o más versiones de un producto o aplicación para comprobar cuál de ellas es más adecuada con respecto a un objetivo concreto. Estas pruebas son utilizadas frecuentemente en el ámbito de la analítica web y el marketing digital.

El funcionamiento de un test A/B es el siguiente:

1) En primer lugar, se selecciona una intervención, es decir, una modificación, o conjunto de modificaciones, que se desea realizar a un producto con un objetivo concreto. Algunos ejemplos de intervención podrían ser bajar el precio de un producto para ver si así aumentan sus ventas, o cambiar la colocación de un enlace para ver en qué sitio de la interfaz se consigue que se seleccione más.
2) A partir de esto se definen las distintas versiones de la aplicación en las que se pondrán a prueba las diferentes opciones que forman parte de la intervención. En adelante, nos referiremos a cada opción dentro de una intervención como *variable de un experimento A/B,* o simplemente *variable*. Por ejemplo, en el caso de una tienda online, una de las variables podría mostrar un precio igual al 100% del valor original, otra mostrar un precio igual al 90% del valor original y una última mostrar un precio igual al 70% del valor original. En este caso el objetivo sería evaluar cuál de las tres versiones produce más beneficios. A la hora definir el experimento, se asigna a cada variable el porcentaje de usuarios de la aplicación que se desea la que vean o utilicen. Por ejemplo, para realizar el experimento de manera más o menos segura, se podría optar por que un 75% de los usuarios sigan viendo el precio del producto al 100% de su valor original, un 15% por ciento reciban el 10% de descuento y un 10% el 30% de descuento.
3) El siguiente paso es ejecutar el experimento. Para poder ejecutarlo, la interfaz del producto debe saber qué versión del experimento debe mostrar a cada usuario. En este punto, es importante que el usuario siempre vea la misma versión del producto. Es decir, si un usuario entra en la aplicación y ve que el producto cuesta 29.95€, las siguientes

veces que acceda a dicho producto deberá visualizar el mismo precio. Para conseguir esto, la interfaz de la aplicación, antes de mostrar la información al usuario, tendrá que preguntar a algún tipo de servicio qué versión del experimento le corresponde. En función de la versión que le corresponda a un determinado usuario, la interfaz deberá realizar las modificaciones que sean necesarias.

4) Finalmente, se procede a evaluar los resultados, que en el ejemplo anterior, serían los beneficios obtenidos con cada precio del producto. En función de las conclusiones que se obtengan se decidirá si asignar la variable que ha resultado más eficiente a todos los usuarios o, si no se han obtenido unos resultados concluyentes, modificar el experimento existente. Una práctica habitual dentro del mundo de los experimentos A/B es modificar los porcentajes asignados a las variables del experimento para, progresivamente, ir convergiendo hacia una determinada opción. En el caso de nuestro ejemplo, y en función de sus resultados, en una segunda iteración del experimento podría optarse por subir el porcentaje de usuarios que reciben un 10% de descuento al 50%, bajar los que ven el precio original al 45% y reducir los que reciben el descuento del 30% al 5%.

Las *variables* o versiones del experimento definen cómo va a ser la interfaz del producto. Como mínimo, existen dos variables: la de *control*, que muestra la interfaz original, y una variable *extra* que difiere ligeramente de la original en algún aspecto. Dicho esto, se pueden tener tantas variables como sea conveniente.

1.2 Objetivos

El objetivo principal del proyecto es, como se ha mencionado anteriormente, crear una aplicación para crear y gestionar pruebas A/B. La idea general del proyecto se ilustra en la Figura 1.

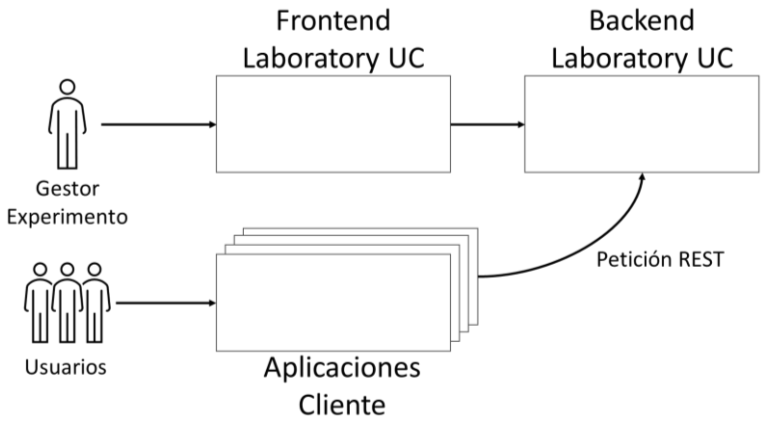

Figura 1. Esquema del Funcionamiento de LA/Boratory

En primer lugar, la persona u organización que quiera crear un experimento para uno de sus productos deberá acceder a nuestra aplicación, *LA/Boratory,* a través de su interfaz web y configurar un nuevo experimento. A continuación, deberá modificar la interfaz del producto al que desea aplicar el experimento para que, cada vez que un usuario acceda a ella, se realice una petición a un servicio web ofrecido por *LA/Boratory* que devuelva la variable del experimento asociada a ese usuario. En caso de que el usuario no tenga asignada ninguna variable aún, el servicio deberá asignarle una, cuidando de que las asignaciones se distribuyan aleatoriamente conforme a los porcentajes asignados a cada variable. En función del transcurso del

experimento, sus creadores podrán acceder a la aplicación *LA/Boratory* para modificar los porcentajes asignados a cada variable, o para seleccionar una de estas variables como la definitiva.

Además, los creadores de experimentos deberían poder definir y gestionar lo que se conoce como *overridden users.* Un *overridden user* es un usuario para el cual la asignación de un experimento no se realiza de manera aleatoria, sino que la establece explícitamente el gestor de un experimento. Estas asignaciones no se pueden modificar aunque se modifiquen los porcentajes asignados a cada variable. El objetivo de este tipo de asignaciones es tener usuarios bien identificados, normalmente ficticios, para los que sabemos con total seguridad qué versión de un experimento les corresponde. Esto se utiliza fundamentalmente para la realización de pruebas. Por ejemplo, si queremos verificar cómo se verá la interfaz con la opción C de un determinado experimento, accederemos a la aplicación con el *overridden user* asociado a dicha opción C y podremos ver esa versión concreta de la interfaz. Sin estos *overridden users,* para ver cómo queda una interfaz con una determinada opción, tendríamos que ir entrando con diferentes usuarios hasta que el servicio le asignase a uno de ellos la opción C.

Todas las funcionalidades descritas anteriormente se recogen de manera más sistemática en el modelo de objetivos del sistema de la Figura 2.

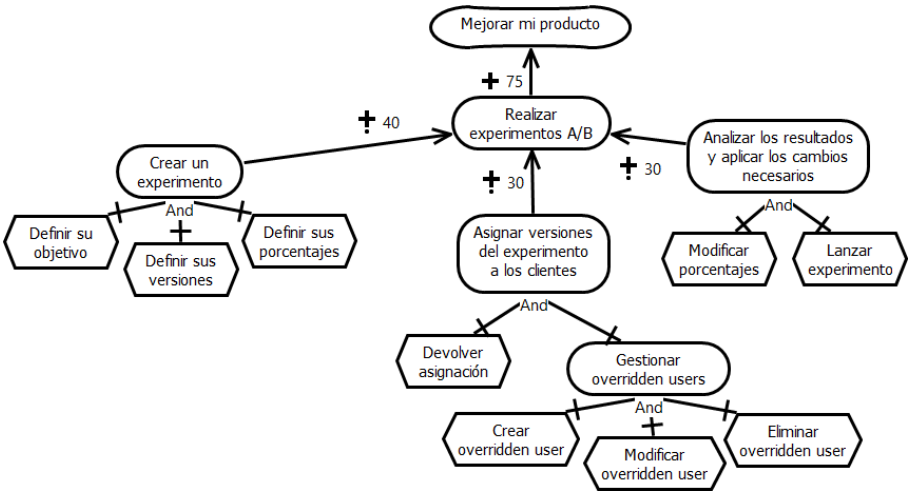

Figura 2. Modelo de Objetivos

1.3 Alcance del Trabajo Fin de Grado

A continuación se especifica, fase por fase, la contribución del proyecto a cada parte del ciclo de vida software.

1.3.1 Ingeniería de Requisitos

En este caso el proceso de obtención de los requisitos se realizó de dos maneras distintas. Por un lado, los directores de *HP* actuaron a modo de *Product Owners* y especificaron qué funcionalidades querían que tuviera el servicio. Por otro lado yo también les comenté otras posibles funcionalidades que podrían ser de utilidad para el usuario. Esta tarea se realizó al principio del proyecto, sin embargo, hubo requisitos que se fueron refinando y añadiendo durante el mismo. Esto se debió a que, con el transcurso del tiempo y la implementación de código, fueron surgiendo nuevas funcionalidades que resultaron de interés y se fueron incorporando al proyecto mediante el uso de una metodología de desarrollo ágil.

1.3.2 Arquitectura

Todas las decisiones referentes a la arquitectura fueron tomadas por mí. Dentro de ellas se incluye el diseño en 3 capas de la aplicación, el modelo de dominio, la interfaz REST junto con las representaciones de los recursos, y la selección de los patrones de diseño utilizados para facilitar la implementación de dichas capas.

1.3.3 Implementación

La parte de la implementación también ha sido realizada al completo por mí. Por un lado está la parte del servidor y por otro dos clientes que hacen uso de la API.

La implementación del servidor está organizada en controladores que se comunican con el repositorio a través de una capa de servicio. También se han añadido clases dedicadas a alimentar la base de datos y a realizar los cálculos necesarios para cumplir la lógica de negocio de la aplicación de manera precisa.

Los clientes se organizan en componentes que se comunican con el servidor a través de una capa de servicio. También se han definido las interfaces correspondientes a los tipos de datos de la aplicación, y se han utilizado diferentes APIs como, por ejemplo, *React Router*, para facilitar la navegación entre las distintas páginas de la interfaz, y *React Hook Form*, para facilitar la implementación de los formularios.

Aun así, cabe aclarar que la implementación del cliente que hace de demo de la página web tiene una implementación más sencilla y menos modularizada, porque, como se trataba de un ejemplo cuya funcionalidad no era crucial para el cumplimiento de los objetivos del proyecto, se decidió hacer más hincapié en el diseño del cliente dedicado a la gestión de experimento.

Toda la implementación se ha realizado manualmente a excepción del mapeo objeto-relacional que se ha generado automáticamente mediante anotaciones JPA[2].

1.3.4 Pruebas

En cuanto a pruebas unitarias, se ha realizado 1 prueba unitaria para el *back-end*, ya que las clases de dominio contenían una lógica muy sencilla y se consideró que no merecía la pena realizar pruebas exhaustivas sobre ello.

Respecto a las pruebas de integración, se realizaron 70 pruebas que comprueban tanto el código de respuesta como el contenido del cuerpo de los mensajes HTTP resultantes de hacer distintas peticiones al servicio.

Las pruebas de interfaz se realizaron para la parte del *front-end*, ya que es donde se implementó la UI. En total se realizaron 14 pruebas sobre 8 de los componentes, las cuales comprueban que se renderizan correctamente los elementos de cada componente y que el funcionamiento al utilizar los botones y cuadros de texto, entre otros, es el esperado. Cabe destacar que los componentes han ido modificándose y mejorando a lo largo del proyecto, por lo que algunos de los *test* se realizaron para versiones anteriores a las de los componentes finales.

Durante el desarrollo del proyecto y en las reuniones que fueron teniendo lugar, se realizaron pruebas de aceptación por parte de los tutores de *HP*, que comprobaban la funcionalidad del servicio siempre que se implementaba algo nuevo.

Finalmente, cabe destacar que se realizó una prueba para comprobar el funcionamiento del algoritmo de asignación de variables y constatar su fiabilidad. De esta manera se aseguró que el reparto de las asignaciones era correcto y se respetaban los porcentajes especificados.

1.3.5 Despliegue

En este caso no se realizó ningún despliegue del servicio, ya que, por las políticas del programa *Observatorio HP*, los productos realizados no se despliegan, únicamente quedan almacenados en el repositorio para su posterior uso por la empresa.

1.4 Metodología de Desarrollo

La metodología de desarrollo que se siguió para desarrollar el proyecto fue *Scrum* con una serie de modificaciones. Se podría decir que los tutores de *HP* ejercían el rol de *Product Owners*, ya que guiaban el rumbo del proyecto en cuanto a las funcionalidades que debía de tener. Lógicamente, mi rol era el de *Scrum Team* ya que era la encargada de desarrollar el proyecto.

A continuación, se describe secuencialmente cómo era el desarrollo de cada *sprint*, los cuales tenían dos semanas de duración. De esta manera, se puede observar con claridad qué diferencias guardaba respecto a la metodología *Scrum* original.

Primero tenía lugar una reunión al inicio de cada *sprint* que servía a modo de *Product Review* y *Sprint Planning Meeting.* En ella se revisaba el trabajo realizado durante el *sprint* anterior y luego se seleccionaban las historias de usuario y tareas que se iban a llevar a cabo en el siguiente ciclo. Para especificar las historias de usuario no se definían de manera detallada sus criterios de confirmación, ya que, normalmente, estas reuniones eran bastante extensas, y desde HP tampoco disponían de mucho tiempo porque tenían otras reuniones de trabajo programadas. Por otro lado, en la planificación del *sprint*, las historias de usuario se elegían en base a criterios subjetivos, ya que estas no tenían puntos de esfuerzo. Esto se debe a que para el desarrollo del proyecto se utilizaron nuevas tecnologías en las que no se tenía experiencia, por lo que no se podía estimar cuánto se iba a tardar en realizar cada una de ellas. Además, la carga de trabajo asignada también variaba en función de los exámenes y entregas que estuvieran previstos para esas semanas. Estas circunstancias hacían que la selección de las historias de usuario se realizara mejor de esta manera. Por otro lado, para evitar tener más reuniones, no se realizaron *Product Backlog Refinement* formalmente sino que la actualización del *backlog* se llevaba a cabo durante las reuniones quincenales cuando era necesario. Debido a que el *Scrum Team* estaba formado por una sola persona, no hubo necesidad de realizar *Daily Scrum Meetings*. Tampoco había *Sprint Retrospective* sino que, de manera informal, iba aprendiendo de los errores cometidos en los *sprints* para intentar ser más eficiente, y las prácticas que me servían de utilidad las mantenía. Por ejemplo, al comienzo del proyecto empecé realizando el modelo de dominio, detallando el diseño antes de la implementación. Como planificar la implementación me sirvió de ayuda, continué haciendo esquemas y pseudo-códigos en papel durante el resto del proyecto antes de escribir el código.

Dicho esto, las reuniones tenían lugar, al menos, dos veces al mes. Aun así, hubo que realizar alguna modificación en el calendario ya que en algunos casos coincidía que el día que debíamos reunirnos era festivo o alguno de los asistentes no podía acudir. También debíamos tener en cuenta que trabajábamos desde comunidades autónomas diferentes, por lo que no todos los días de vacaciones coincidían los mismos para todos.

El proyecto comenzó de manera oficial el 17 de enero de 2022 y finalizó el 2 de junio de 2022. En este tiempo tuvieron lugar 11 *sprints* y se desarrollaron 15 historias de usuario. La Tabla 1 muestra los datos de dos ciclos. La tabla completa se adjunta en el apéndice A a modo de ejemplo.

Sp	Fechas	Funcionalidades implementadas
9	12/05/2022 16/05/2022	Filtrar la lista de experimentos por nombre
		Crear *override*
		Modificar experimento
10	16/05/2022 02/06/2022	Acceder a una página web de ejemplo
		Eliminar *overrides* de un experimento
		Ver lista con los *overrides* de un experimento
		Modificar asignaciones de un experimento

Tabla 1. Extracto de la Información de los Sprints

Para realizar el seguimiento del proyecto se utilizó la plataforma *GitLab*. Para las tareas y las historias de usuario se crearon *issues*, que son unidades de trabajo que se utilizan para realizar una mejora en un sistema. Esta puede ser el arreglo de un fallo, una característica solicitada o incluso una tarea. Los *issues* iban etiquetados para determinar su tipo y su prioridad. Las etiquetas eran las siguientes:

- *Must*: funcionalidad imprescindible que se debía implementar.
- *Nice to have*: funcionalidad que podría resultar de interés pero su implementación no era crucial.
- *Task*: tarea.
- *User Story*: historia de usuario.
- *Improvement*: mejora a funcionalidades que ya habían sido implementadas.
- *Dropped*: funcionalidad que finalmente no se implementó.

La Figura 3 muestra un ejemplo de *issue*:

Figura 3. Ejemplo de Issue

1.5 Herramientas de Desarrollo

Las herramientas de desarrollo que se utilizaron para la implementación de la aplicación fueron las siguientes:

- Se eligió *Spring*[3] para el *back-end* por familiaridad con la herramienta, y por ser actualmente una de las de mayor penetración industrial dentro del lenguaje *Java*.
- Se escogió *React*[4] para el *front-end* porque es una de las más utilizadas hoy en día en el mundo laboral.
- Como base de datos se utilizó *H2*. Se trata de una base de datos en memoria que se eligió porque cumplía con las necesidades del servicio y resultaba útil para el proceso de desarrollo, ya que como se ha mencionado anteriormente, el proyecto no estaba pensado para ser desplegado. La ventaja principal de este tipo de bases de datos es que las velocidades de acceso son significativamente más altas, además de que en la fase de desarrollo la persistencia de los datos no era un factor de prioridad. De todas maneras,

en el caso de que se llegara a desplegar, se podría optar por la utilización de otra base de datos como por ejemplo *MySQL*.

Para la implementación de las pruebas se utilizaron las siguientes herramientas:

- *JUnit*[5] para las pruebas unitarias, ya que es la herramienta con la que tenía mayor familiaridad.
- *Postman*[6] para las pruebas de integración de la API, ya que es una conocida plataforma que facilita el diseño, construcción y fase de pruebas de APIs, y me fue recomendada por mis tutores.
- *React Testing Library* para los *tests* del *front-end*, porque permite probar los componentes de *React* con facilidad y además, debido a su popularidad, hay mucha información disponible sobre ella en la web.

En cuanto a la gestión de la configuración se crearon dos ramas: una principal llamada *main* y otra de desarrollo llamada *develop*. La idea inicial era crear una rama por cada funcionalidad e ir fusionándolas con la rama *develop* cuando estuvieran acabadas. Sin embargo, parte de la implementación de funcionalidades como *Crear Experimento*, sufrieron bastantes cambios y estuvieron en continua mejora, por lo que finalmente decidí no hacerlo así para evitar tener varias ramas vivas que tuviese que estar fusionando constantemente entre sí. Además, como solo estaba trabajando yo en el proyecto, había una menor necesidad de gestión de versiones concurrentes del proyecto. Por último, destacar que en cada *commit* detallaba cuáles eran los cambios realizados respecto a la versión anterior identificando si se trataba del *back-end* o del *front-end* cuando era necesario.

Respecto al repositorio, este está formado por dos carpetas principales: *backend* y *frontend*. Dentro de la primera hay una carpeta llamada *docs*, en la que se encuentra el documento que describe la API del servicio, y una carpeta llamada *laboratory-uc* donde se encuentra el proyecto *Spring Boot* con la implementación del servicio. La segunda carpeta principal se divide en otras dos subcarpetas, *client* y *laboratory-uc*. La primera de ellas contiene el proyecto *React* con la página web de ejemplo, y la segunda contiene la implementación de la interfaz de gestión de los experimentos.

No tuve la oportunidad de realizar integración continua ya que en *GitLab* la integración continua es de pago y *HP* no la tenía habilitada. Sin embargo, me habría gustado poder utilizarla porque resulta muy útil para la automatización de pruebas.

Finalmente, respecto a la gestión de la calidad, mi código era revisado por los tutores de *HP*. En estas inspecciones de código me informaban sobre cosas que podía mejorar o aspectos que estaban mal implementados, sobre todo en *React*, ya que era la primera vez que trabajaba con esa tecnología. Un caso concreto fue el del formulario para la creación de los experimentos, más específicamente la parte de validación. Inicialmente lo había realizado utilizando las propiedades *error* y *helperText* de los componentes de *MaterialUI*[7], pero como había que hacer muchas comprobaciones y había campos que dependían de otros, esto acabó siendo demasiado engorroso. Al revisar mi código, me advirtieron del problema y me dijeron que podía implementarlo de una forma más sencilla utilizando la API *React Hook Form*. Al usar esta API conseguí que la implementación fuera más sencilla y además me facilitó el resto de tareas en las que se necesitaba un formulario, como por ejemplo, en la modificación de los experimentos.

2. Ingeniería de Requisitos

Este capítulo únicamente recoge los requisitos funcionales y no funcionales del proyecto, ya que, como las funcionalidades a implementar ya estaban más o menos definidas desde el principio del proyecto, no hubo un proceso formal de captura de requisitos.

2.1 Requisitos Funcionales

En total hay 15 requisitos funcionales que se documentaron como historias de usuario. A continuación, la Tabla 2 muestra dos ejemplos de estas historias de usuario. El resto se pueden encontrar en el apéndice B. Tal como se comentó en el apartado anterior, no se definieron de manera formal criterios de confirmación para cada historia de usuario porque no disponíamos de tanto tiempo de reunión con los tutores de *HP*.

ID	Historia de Usuario
RF-07	Lanzar un experimento
Yo, como usuario, quiero lanzar un experimento de manera que, una vez finalizado el proceso de evaluación de los resultados, pueda asignar la variable que ha resultado más eficiente a todos los usuarios de un experimento respetando los *overridden users*. Tener esta funcionalidad me resultaría útil para poder aplicar los resultados de los experimentos de manera casi inmediata.	
RF-08	Filtrar la lista de experimentos por nombre
Yo, como usuario, quiero filtrar la lista de experimentos por su nombre de manera que pueda localizar los experimentos más rápido, independientemente del tamaño de la lista. Consecuentemente, conseguiría ahorrar tiempo.	

Tabla 2. Ejemplos de Historias de Usuario

2.2 Requisitos No Funcionales

Para la definición de los requisitos no funcionales se realizó un análisis de la ISO 25010, la cual cataloga y categoriza los requisitos no funcionales que suelen aparecer de manera recurrente en la mayoría de los sistemas software.

Dentro de estos requisitos no funcionales, nuestro sistema no tenía que presentar ninguna característica especial para la mayoría de ellos, más allá de lo esperable a cualquier sistema software. Por tanto, no se consideró necesario definir requisitos no funcionales concretos para estas categorías.

Por ejemplo, respecto a la categoría de rendimiento, se entiende que el sistema debe tener un consumo de memoria adecuado, pero no existe realmente ningún límite concreto para este consumo de memoria. Por tanto, el único requisito no funcional que podría establecerse dentro de esta categoría es que la aplicación debe tener un consumo de memoria razonable, pero esto es una obviedad que se aplica a todo sistema software y que, por tanto, no merece la pena especificar de manera explícita como requisito no funcional.

No obstante, dentro de la ISO 25010, existían dos categorías que sí eran merecedoras de un estudio más detallado dentro de nuestra aplicación. La primera de ellas era la de corrección funcional, dentro del apartado más genérico de adecuación funcional. La corrección funcional mide el grado de precisión de los resultados de una aplicación. En nuestro caso concreto, nuestro sistema debe asignar versiones de un experimento a los usuarios conforme a unos porcentajes deseados. No obstante, si estas asignaciones se realizan con verdadera

aleatoriedad, la distribución real de las versiones de un experimento deberá acercarse a los porcentajes especificados, pero diferirá levemente de dichos porcentajes. Por tanto, para que nuestra aplicación proporcione una adecuada corrección funcional, la diferencia entre el porcentaje de asignación especificado para un experimento y el porcentaje de versiones realmente asignadas deberá mantenerse dentro de unos márgenes de error adecuados, lo que constituiría el primer requisito funcional para nuestra aplicación.

En segundo lugar, por petición de los directores de *HP*, la interfaz de usuario del sistema debía ser capaz de detectar la mayor cantidad de errores posibles en los datos de entrada y alertar adecuadamente al usuario de ello, informándole además de cómo solucionar los problemas detectados. En particular, se puso especial énfasis en verificar antes de realizar cualquier petición al servidor que la suma de los porcentajes asignados a cada versión de un experimento fuese 100. Esto constituiría nuestro segundo requisito no funcional.

3. Arquitectura y Diseño

3.1 Diseño Arquitectónico

La Figura 4 muestra un esquema del diseño arquitectónico de la aplicación. Como se ha explicado anteriormente, la aplicación está organizada en 3 capas. Los creadores de experimentos utilizan una arquitectura en tres capas clásica, con presentación, negocio y persistencia; mientras que las aplicaciones que necesitan saber qué variable le corresponde a un usuario utilizan simplemente un servicio web con capa de negocio y persistencia.

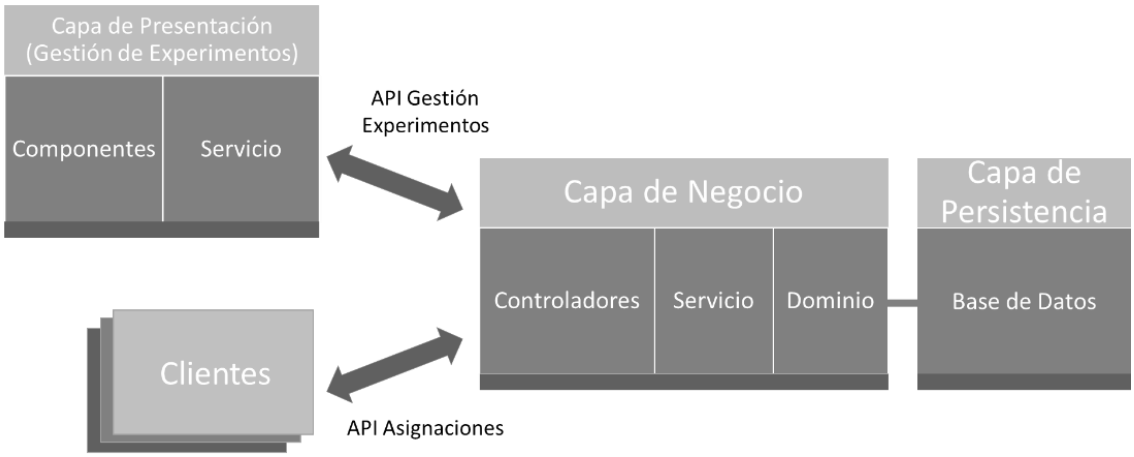

Figura 4. Esquema de la Arquitectura

Dentro de la arquitectura, se distinguen dos orígenes distintos para las acciones que tendrá que ejecutar el sistema: los creadores o gestores de los experimentos y las aplicaciones clientes. En el primero de ellos, el gestor de los experimentos accedería al *front-end* o interfaz gráfica para la definición y gestión de experimentos con el objetivo de, por ejemplo, crear uno nuevo. Esta interfaz gráfica, tras validar que los datos son sintácticamente correctos, es decir, el peso de cada variable del experimento es un número entre cero y 100 y la suma de todos ellos vale 100, entre otras comprobaciones, haría una petición AJAX a la API REST de la aplicación. Esta API REST comprobaría de nuevo los datos del experimento y, en el caso de que estos cumplieran con la lógica de negocio, los enviaría a la capa de servicio a través del método correspondiente, en este caso, el de creación de experimentos. Desde ahí, se llamaría al repositorio para que almacenara el nuevo experimento en la base de datos. Si el proceso de validación de datos desde la API REST se realiza con éxito, se devuelve un código 201 a la capa de presentación, sino, se envía un código de error que identifique el problema, de manera que la interfaz sepa por qué ha fallado el proceso.

En segundo lugar están las aplicaciones clientes, cuyo funcionamiento es similar al anterior. En este caso, se hace una llamada AJAX desde una interfaz desconocida a la API REST con el objetivo de conocer la asignación correspondiente a un usuario para un experimento concreto. El resto del proceso es el mismo que el descrito anteriormente, con la única diferencia de que, en vez de crear un experimento, las funciones estarán destinadas a obtener una asignación.

Por brevedad, la estructura del *back-end* y del *front-end* del proyecto se encuentra en el apéndice C.

3.2 Modelo de Dominio

La Figura 5 muestra una imagen con el modelo de dominio del sistema.

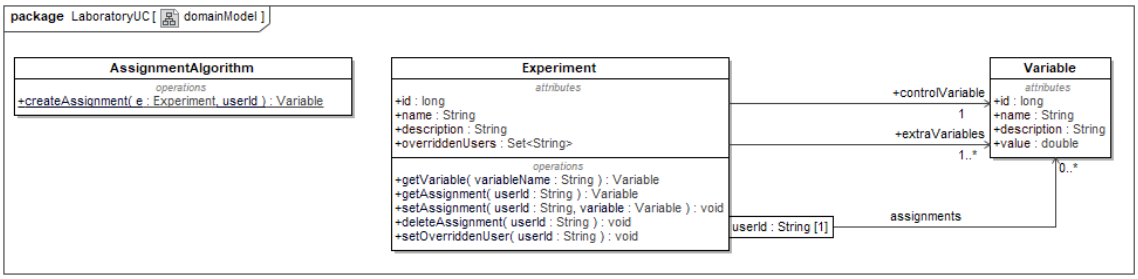

Figura 5. Modelo de Dominio

Podemos observar que se trata de un modelo sencillo en el que existen solo dos clases en torno a las cuales se organiza toda la lógica de negocio. La primera de ellas, *Experiment*, tiene un *id* único que es asignado por el servidor en el proceso de creación, un nombre que debe ser único para cada experimento y una descripción que define en qué consiste la prueba. Como se explicó anteriormente, cada experimento tiene una variable de control (*controlVariable*) y una o más variables extra (*extraVariables*).

En base a esta clase, se desarrolla el término *assignment*, que hace referencia a las asignaciones de versiones del experimento a usuarios concretos. Una asignación está formada por el identificador del usuario y la variable que tiene designada, y determinará cuál de las variantes visualizará el usuario al acceder a la página. Para definirlas se utiliza un mapa cuya clave son los *id* únicos de los usuarios, de manera que, a cada *id* le corresponde una variable. Finalmente, para representar a los usuarios cuya asignación se ha realizado manualmente, es decir, los *overridden users*, se utiliza una lista con sus identificadores.

- La operación *getVariable* permite obtener la variable del experimento a partir de su nombre.
- La operación *getAssignment* permite obtener la asignación de un usuario para un experimento concreto a partir de su identificador. Si el usuario no tuviese ninguna versión del experimento asignada aún, se le asignará una utilizando el servicio *AssignmentAlgorithm*.
- La operación *setAssignment* permite crear una asignación indicando el *id* del usuario y la variable a la que va a ser asignado. Este método se utiliza para crear todo tipo de asignaciones, tanto las manuales como las realizadas con la ayuda del algoritmo.
- La operación *deleteAssignment* permite eliminar una asignación a partir del identificador del usuario.
- El método *setOverriddenUser* permite añadir un usuario a la lista de usuarios con asignación manual, de manera que se puedan distinguir fácilmente del resto de usuarios cuya asignación ha sido realizada automáticamente.

La clase *Variable* representa a cada una de las variables o variantes del experimento. Cada una de ellas tiene un *id* que es asignado por el servidor, un nombre que la identifica unívocamente dentro del experimento, una descripción, y el porcentaje de usuarios a los que debe ser asignada. Por último cabe añadir que, por petición de los *Product Owners*, el nombre de la variable de control debe ser siempre “C”.

Por último, la clase *AssignmentAlgorithm* representa el servicio que se utiliza para obtener la variable que se debe asignar a un usuario cuando se trata de asignaciones aleatorias.

3.3 Diseño de API REST

La API está formada por 5 recursos, los cuales forman la base para la gestión de los experimentos y su información. La Tabla 3 detalla estos recursos y cómo pueden ser manipulados.

Recurso	URI	Métodos	Respuestas HTTP
API Gestión Experimentos			
Experiments	/experiments	GET	200
Experiment	URIBase/experiments	POST	201, 400, 409
	/experiments/{expName}	GET	200, 404
		PUT	200, 400, 404
		DELETE	200, 404
Variables	/experiments/{expName}/variables	GET	200, 404
		PUT	200, 404
Assignments	/experiments/{expName}/assignments	GET	200, 404
Assignment	/experiments/{expName}/assignments/{userId}	PUT	200, 400, 404
		DELETE	200, 404
API Asignaciones			
Assignment	/experiments/{expName}/assignments/{userId}	GET	200, 404

Tabla 3. Diseño API REST

El recurso *Experiments* representa la lista con todos los experimentos del servicio. En este caso únicamente dispone del método GET, y la respuesta siempre va a ser 200 (OK) ya que devolverá una lista, en el caso de que no exista ningún experimento, esta estará vacía. Por otro lado, el recurso *Experiment* representa un experimento concreto, y sobre él pueden realizarse operaciones de creación (POST – porque el *id* es asignado por el servidor), lectura (GET), modificación (PUT) y borrado (DELETE). Al crear el experimento, como los nombres tienen que ser únicos, se devolverá un código 409 (CONFLICT) si se intenta crear un experimento con un nombre que ya existe en la base de datos.

Respecto a las variables, se podrán obtener cuáles son las variables que pertenecen a un experimento concreto (GET). También se podrá modificar su porcentaje una vez creado el experimento (PUT). La Figura 6 muestra los parámetros que se deben enviar en esta última petición. Como se puede observar, se envía una lista con todas las variables del experimento, cada una de ellas identificada por su nombre. En el campo *expectedValue* se envía el nuevo valor que se desea asignar a cada variable. En esta operación en concreto, el valor del campo *realValue* no es relevante porque es recalculado automáticamente por el servicio después de hacer las reasignaciones.

El servicio es el que se encarga de detectar cuál es la variable *donante* y cuál es la *receptora*. Siguiendo las directrices establecidas por los directores del proyecto, únicamente se podrán reasignar porcentajes de una variable *donante* a otra *receptora*, enviando desde la variable donante un porcentaje, menor o igual a su valor, a la variable receptora. De esta forma se asegura que la suma de los porcentajes asignados a las variables sea 100.

```
[
  {
    "name": "C",
    "realValue": 0,
    "expectedValue": 40
  },
  {
    "name": "A2",
    "realValue": 100,
    "expectedValue": 60
  }
]
```

Figura 6. Parámetros Método PUT Variables

El recurso *Assignments* funciona de manera similar al de *Experiments*, con la única diferencia de que si no existe el experimento se devolverá un código 404 (NOT FOUND). Finalmente, el recurso *Assignment*, que representa una asignación para un usuario en un experimento concreto, dispone de operaciones de lectura (GET), creación (PUT) y eliminación (DELETE), estando estas dos últimas destinadas a las asignaciones manuales, es decir, para los *overridden users*.

Cabe destacar que algunas operaciones pueden devolver el error 400 (BAD REQUEST). Esto sucederá si el cliente intenta crear o modificar un experimento con unos datos que son incorrectos, es decir, que no cumplen con la lógica del sistema. En todos los casos, el código de respuesta 404 (NOT FOUND) se enviará cuando alguno de los elementos de la URI no exista.

Para completar la especificación de la API, las Figuras 7 y 8 muestran un modelo de respuesta para las operaciones GET sobre los recursos *Experiment* y *Assignment*. Tal como se puede observar, el servicio devuelve los datos en formato JSON, que es la práctica habitual hoy en día.

```
{
 "name": "PricingExp",
 "description": "This experiment displays different prices in a online shop depending
on the user who is buying the product. The aim of this experiment is to analyze at
what price do people buy the product more often.",
 "controlVariable": {
  "id": 14,
  "name": "C",
  "description":"The control variable should display the original price of the product.",
  "value": 35
 },
 "extraVariables": [
  {
   "id": 15,
   "name": "X",
   "description":"This variable should display 50% of the original price of the product.",
   "value": 10
  },
  {
   "id": 16,
   "name": "Y",
   "description":"This variable should display 125% of the original price of the product.",
   "value": 25
  },
  {
   "id": 17,
   "name": "Z",
   "description": "This variable should display 75% of the original price of the product.",
   "value": 30
  }
 ]
}
```

Figura 7. Representación de Experiment

```
{
  "variable": "C",
  "userId": "pepe",
  "overriden": false
}
```

Figura 8. Representación de Assignment

3.4 Diseño de las bases de datos.

En este proyecto, la base de datos se ha generado automáticamente a partir del modelo de dominio usando anotaciones JPA. La Figura 9 muestra, a modo de ejemplo, la clase *Experiment* anotada utilizando JPA.

```
1  @Entity
2  public class Experiment {
3        // Atributos
4        @Id
5        @GeneratedValue
6        private long id;
7        @Column(unique=true)
8        private String name;
9        private String description;
10       @OneToOne(cascade=CascadeType.ALL)
11       @JoinColumn(name="control_variable_fk")
12       private Variable controlVariable;
13       @OneToMany(cascade=CascadeType.ALL)
14       @JoinColumn(name="experiment_fk")
15       private List<Variable> extraVariables;
16       @ManyToMany(cascade = CascadeType.ALL)
17       private Map<String, Variable> assignments; // clave-valor
18       @ElementCollection
19       private Set<String> overriddenUsers;
20
21       // Constructores
22       // (...)
23
24       // Getters y setters
25       // (...)
26 }
```

Figura 9. Extracto de la clase Experiment

Conforme al estándar JPA, las entidades van etiquetadas con la anotación *@Entity* (línea 1). Al anotarse como *Entity* se creará una tabla en la base de datos para guardar las instancias de esta clase. En cuanto a los atributos, cada uno de ellos va etiquetado de manera distinta en función de sus características. A continuación se describe cada uno de ellos:

- El atributo *id* lleva la etiqueta *@Id* (línea 4), que especifica que el atributo funciona como clave primaria del objeto. Además, la etiqueta *@GeneratedValue* (línea 5) indica que esta clave es autogenerada.
- El atributo *name* lleva la anotación *@Column* (línea 7) con el valor *unique* a *true*, indicando que el valor de esta columna en la base de datos debe ser único, es decir, no puede haber nombres de experimentos repetidos. Este campo se podría haber utilizado como clave primaria pero finalmente se optó por usar un entero autoincrementado por petición de los *Product Owner*.
- El atributo *description* no lleva ninguna anotación porque es un atributo simple que se me mapea directamente a una columna de una tabla de la base de datos.
- El atributo *controlVariable* lleva la anotación *@OneToOne* (línea 10), que indica que esa relación entre la clase *Experiment* y la clase *Variable* es uno a uno. Es decir, todo

experimento tiene una única variable de control, y a cada variable de control le corresponde un único experimento. Además, las operaciones con la base de datos se propagan en cascada. Por ejemplo, si se inserta un nuevo experimento en la tabla de experimentos, también se insertará su variable de control en la tabla para las variables. Estas relaciones uno a uno se implementan a nivel de base de datos asignando una clave foránea a una de las dos tablas. En este caso, la anotación *@JoinColumn* (línea 11) especifica que la clave foránea estará en la tabla de los experimentos y su nombre será *control_variable_fk*.

- El atributo *extraVariables* tiene las mismas características que *controlVariable*, a excepción de que el tipo de relación es *OneToMany*, ya que un experimento tendrá una o más variables extra, y cada variable extra pertenecerá a un único experimento. En este caso, la clave foránea estará en la tabla *Variable*.
- El atributo *assignments* es el mapa que contiene las asignaciones del experimento, y se mapea utilizando la etiqueta *@ManyToMany* (línea 16).
- El atributo *overriddenUsers*, al tratarse de un conjunto cuyo tipo base es un tipo primitivo simple, *String* en este caso, se mapea a través de la etiqueta *@ElementCollection* (línea 18).

3.5 Diseño de las interfaces de usuario

Todas las decisiones sobre la interfaz y el modo de navegación del servicio se han tomado intentando favorecer la sencillez de uso del sistema. Además, se ha tenido muy presente el hecho de proteger al usuario de posibles errores que pudieran provocar un mal funcionamiento del sistema. Por ejemplo, como era de esperar, la interfaz de usuario evita que se asignen porcentajes a las variables cuya suma total no sea 100, informando adecuadamente al usuario de este problema cuando sucede. Para ello se ha optado por el uso de cuadros de diálogo, alertas y mensajes de ayuda, cuyos objetivos se detallan más adelante. A continuación se especifican las interfaces y la navegación de la capa de presentación del sistema.

Las Figuras 10 y 11 muestran un *mockup* para la interfaz de creación de experimentos. De modo general, la interfaz se organiza de la siguiente manera: a la izquierda se sitúa un menú fijo desde el cual se puede acceder a la lista de experimentos y al formulario de creación de un experimento.

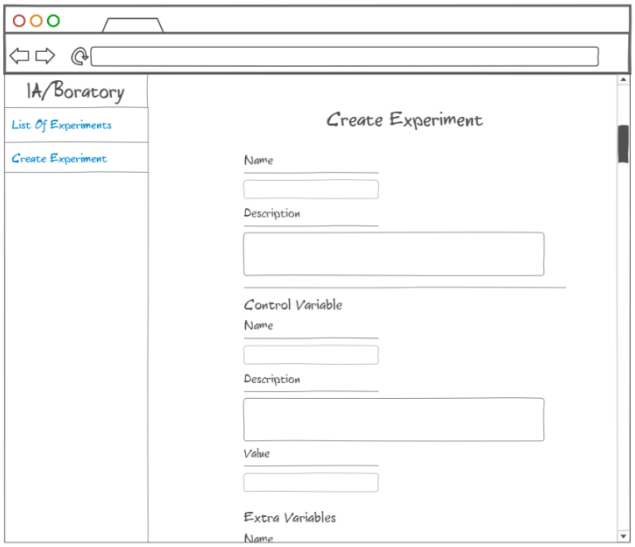

Figura 10. Interfaz Crear Experimento (parte 1)

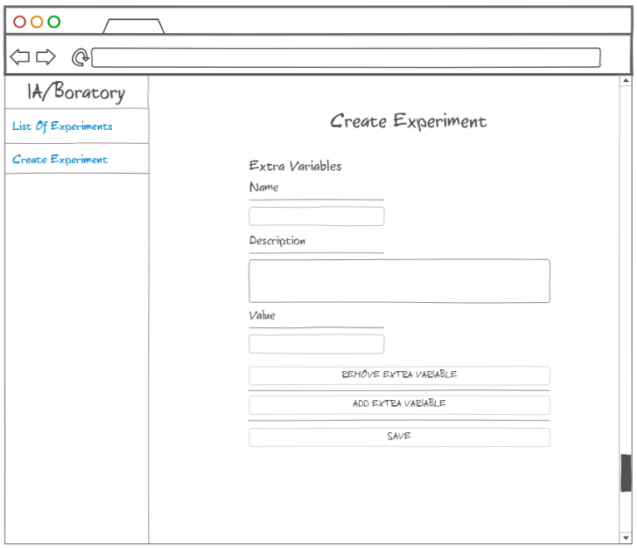

Figura 11. Interfaz Crear Experimento (parte 2)

Como elementos a destacar, observamos un botón para añadir nuevas variables extra y otro para eliminarlas. Además, en relación con la protección de errores mencionada anteriormente, siempre que se intente guardar un experimento cuyos datos no sean correctos, se mostrará un texto de ayuda informando al usuario de cómo solucionar el problema. Cuando se pulsa en el botón *SAVE*, si los datos del experimento son correctos, se guardará y se mostrará la pantalla de la lista de experimentos.

Por el contrario, si desde la interfaz inicial se pulsa el botón *List Of Experiments* del menú, se mostrará la lista con todos los experimentos directamente, tal y como muestra la Figura 12.

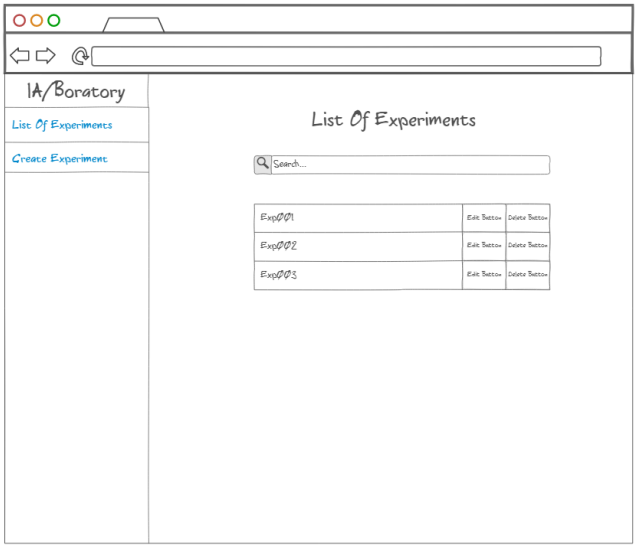

Figura 12. Interfaz Lista de Experimentos

Como se puede observar, se muestra cada experimento junto con un botón para el borrado y otro para la edición del mismo. Si se desea eliminar un experimento, se mostrará un cuadro de diálogo para que el usuario pueda confirmar la operación. Además, el buscador de la parte superior permite encontrar experimentos a partir de su nombre. La búsqueda se realiza dinámicamente según el usuario va escribiendo, por lo que no hace falta ningún botón para iniciarla.

A partir de esta última interfaz se puede acceder a la información de cada experimento pulsando en su nombre. Por brevedad, las interfaces para visualizar y editar un experimento concreto se juntan en el apéndice D.

Desde la interfaz de edición del experimento se puede acceder a la modificación de las asignaciones manuales, es decir, las pertenecientes a los *overridden user*s, y a la configuración de las variables. La interfaz para la modificación de las asignaciones manuales es muy sencilla y, por motivos de brevedad, se omite en este apartado, encontrándose disponible en el apéndice D.

La Figura 13 muestra la interfaz a través de la cual se configuran las variantes, permitiendo transferir porcentajes entre las distintas variables del experimento, la cual no es del todo trivial. Como se puede ver en la imagen, cada variable tiene dos porcentajes, el primero de ellos, *Expected Value*, es el valor que el usuario introdujo a la hora de crear o modificar el experimento, es decir, es el porcentaje de asignaciones que se desea que tenga esa variable. El segundo porcentaje, *Real Value*, representa el porcentaje real de asignaciones que el experimento tiene asociadas a esa variable. Lo normal es que estos valores difieran entre sí porque, como el algoritmo de asignación trabaja utilizando números aleatorios, es imposible que realice las asignaciones ajustándose al porcentaje exacto. En cuanto al proceso de transferencia de los porcentajes, el usuario debe escribir el porcentaje de asignaciones respecto al total que desea enviar desde la variable actual hacia la variable seleccionada en el desplegable. Después de pulsar el botón de enviar, el servicio se encargará automáticamente de cambiar los porcentajes y redistribuir las asignaciones.

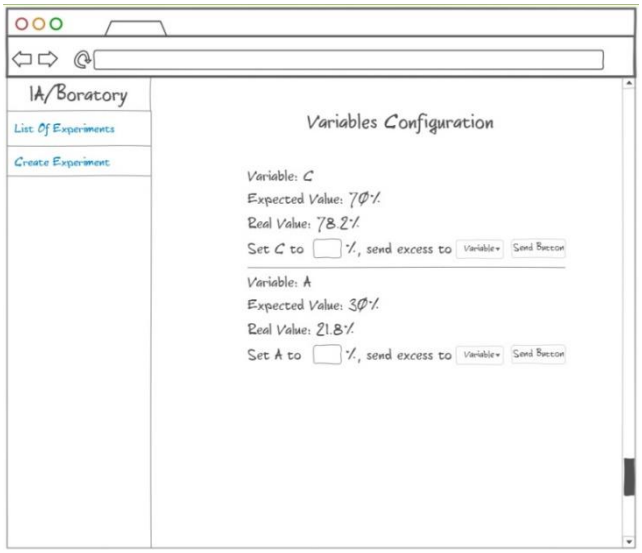

Figura 13. Interfaz Configuración de Variables

4. Implementación

En líneas generales, se ha intentado modularizar el código lo máximo posible para favorecer la legibilidad del mismo, y facilitar su depuración. Respecto al uso de patrones de diseño, cabe destacar la utilización del patrón *Data Transfer Object*, utilizado para la comunicación de datos entre el cliente y el servidor. Este permitió crear cuantas vistas fueron necesarias para enviar estructuras de datos independientes del modelo de dominio, de manera que el proceso resultó más eficiente.

A continuación se detalla la implementación de los distintos componentes de la aplicación.

4.1 Interfaz Gráfica

Esta sección proporciona algunos detalles acerca de cómo se ha llevado a cabo la implementación de la interfaz gráfica o *front-end* de la aplicación. Esta interfaz gráfica se ha implementado siguiendo el modelo *Single Page Application* (*SPA*), utilizando para ello el concepto de *router*.

Para poder explicar qué es un *router* en *React*, debemos conocer primero el concepto de *routing*. El *routing* sirve para navegar dinámicamente entre distintos componentes de un proyecto en función de su URL. De esta manera, cada componente tendrá asociado una URL única. Por un lado, podemos escribir manualmente la URL a la que queremos acceder, y por otro, la misma aplicación podrá modificar la URL por nosotros en función de las acciones que realicemos. Por ejemplo, si estamos visualizando la lista de experimentos, la ruta a la que estaremos accediendo será */list-of-experiments*. Sin embargo, si pulsamos en el primer experimento de la lista, por ejemplo, *Exp001*, la ruta pasará a ser */Exp001*.

En *React* la librería *React Router* nos proporciona los mecanismos necesarios para poder aplicar *routing* sobre nuestro proyecto. La principal ventaja de utilizar esta herramienta es la posibilidad de definir qué se tiene que renderizar en cada página de la aplicación. Además, la gestión de los cambios en la URL en función de las acciones que realice el usuario es realizada automáticamente por la librería.

A continuación, la Figura 14 muestra un fragmento del archivo *index*, en el cual se definen las distintas rutas de la aplicación.

```
root.render(
 <React.Fragment>
  <CssBaseline />
   <BrowserRouter>
    <Routes>
     <Route path="/" element={<App/>}>
      <Route path="list-of-experiments" element={<ListOfExperimentsComponent/>} />
      <Route path="create-experiment" element={<CreateExperimentComponent/>} />
      <Route path="/modify-experiment/:expId" element={<ModifyExperimentComponent/>} />
      <Route path="/modify-experiment/overriden-users/:expId" element={<OverridenComp/>} />
      <Route path="/modify-experiment/variables-configuration/:expId" element={<VComp/>} />
      <Route path=":expId" element={<ViewExperimentComponent/>} />
     </Route>
    </Routes>
   </BrowserRouter>
 </React.Fragment>
);
```

Figura 14. Extracto de index

Estas rutas se declaran dentro del componente *BrowserRouter* (línea 4). Existe una ruta raíz (línea 6) a partir de la cual se forman el resto de URLs, concatenando el *path* de la ruta raíz con el del resto de rutas. Para cada ruta se establece su *path* y el componente al que está asociada. Por ejemplo, el formulario de creación de un experimento (línea 8) se accede mediante la ruta */create-experiment* y está asociado al componente *CreateExperimentComponent*.

Como se ha mencionado anteriormente, la ruta raíz está asociada al componente *App*, por lo que este será el esqueleto principal de la interfaz de la aplicación. Tal y como se puede ver en la Figura 15, la disposición utilizada se trata de una cuadrícula con dos columnas, en la parte de la izquierda se sitúa un menú fijo (línea 5) mientras que la parte de la derecha es dinámica. Dependiendo de la ruta, de entre las definidas en la Figura 14, a la que se acceda, se mostrará un componente u otro. Esto último se declara mediante el componente *Outlet* (línea 8).

```
1 export default function App() {
2  return (
3   <Grid container spacing={0}>
4      <Grid item xs={2}>
5        <MenuComponent />
6      </Grid>
7     <Grid item xs>
8       <Outlet />
9     </Grid>
10   </Grid>
11  );
12 }
```

Figura 15. Componente App

A modo de ejemplo, en la Figura 16 se muestra un extracto del componente dedicado al formulario de creación de los experimentos, suprimiéndose algunos componentes estéticos para favorecer la legibilidad del código. Para su creación se utilizó la API *React Hook Form*, que cuenta con distintos métodos y componentes que facilitan el proceso.

```
1 export default function CreateExperimentComponent() {
2  let navigate = useNavigate();
3  const methods = useForm<IFormInputs>{ resolver: yupResolver(formSchema) });
4  const [saveResp, setSaveResponse] = useState<number>(0);
5  const onSubmit: SubmitHandler<IFormInputs> = (data) =>ExperimentService.saveExperiment(
6                               { data: data, onSaveStatusChanged: handleSaveStatusChange });
7
8  function handleSaveStatusChange(data: number) {
9   setSaveResponse(data);
10   if (data == 201)
11    navigate("/list-of-experiments");
12 }
13
14 return (
15  <Container maxWidth="sm">
16    <Typography variant="h4" component="h1" align='center'> Create Experiment </Typography>
17    <FormProvider {...methods}>
18     <form onSubmit={methods.handleSubmit(onSubmit)}>
19      <ExperimentComponent />
20      <ControlVariableComponent />
21      <Typography variant="h6" component="h1" align='left'> Extra Variables </Typography>
22      <ExtraVariableComponent />
23       <Button type="submit" variant="contained" color="secondary"> SAVE </Button>
24     </form>
25    </FormProvider>
26    <CreateAlertComponent status={saveResp}clicked={saveResp != 0 && saveResp != 201}
27                                              onStatusChange={handleSaveStatusChange} />
28  </Container >
30 );}
```

Figura 16. Extracto de CreateExperimentComponent

A continuación se explican sus aspectos más destacables:

- Se utiliza la función *useNavigate* (línea 2) de la API *React Router* para poder acceder a la URL del navegador y obtener parámetros de ella o modificarla.
- Se utilizan los métodos de *React Hook Form* para definir el esquema de validación a utilizar mediante *useForm* (línea 3).
- Se asigna la acción a realizar cuando el usuario pulse en el botón de enviar formulario a través de la constante *onSubmit* (línea 5).
- Se crea una función llamada *handleSaveStatusChange* (línea 8) para gestionar el código de respuesta después de enviar el formulario, si recibe un código 201 se navega a la página con la lista de todos los experimentos.

En la parte del *return* (línea 14) se organizan todos los componentes que se van a mostrar. Se optó por crear componentes pequeños y formar a partir de ellos la interfaz completa. De esta manera, el código es más legible, no se trabaja con documentos con muchas líneas de código, y los componentes se pueden reutilizar.

El formulario en cuestión está envuelto en la etiqueta *FormProvider* (línea 17), que sirve para asignar el esquema de validación al formulario. Después tenemos el componente *form* (línea 18), que se utiliza para asociar la función *onSubmit* al formulario. Dentro de esta destacan los componentes *ExperimentComponent* (línea 19), que se encarga de recoger la información principal del experimento (nombre y descripción); *ControlVariableComponent* (línea 20), que recoge la información de la variable de control (descripción y valor); y *ExtraVariableComponent* (línea 22), que recoge la información de las variables extra (nombre, descripción y valor) y gestiona su eliminación y creación. Finalmente se encuentra el botón de envío (línea 23). El componente *CreateAlertComponent* (línea 26) no se muestra inicialmente, solo es visible si al guardar el experimento recibe un código de respuesta distinto a 201 (CREATED).

Una vez completado el formulario correctamente, se pulsaría al botón de guardado. Al pulsar a este botón se hace una llamada a la función *saveExperiment* de la clase *ExperimentService* (líneas 5 y 6).

Como se ha mencionado anteriormente, para comprobar la validez de los datos introducidos por el usuario en el formulario, se utiliza un esquema de validación (línea 3). Para realizar este tipo de comprobaciones, la API *React Hook Form* última utiliza *yup*, un constructor de esquemas que permite la validación y el análisis sintáctico de los campos del formulario. Esta herramienta cuenta con métodos para comprobar características como la longitud mínima de un *input* o si un campo es requerido o no, entre otras.

A modo de ejemplo, la Figura 17 muestra el esquema utilizado para la validación de las variables del formulario de creación de experimentos. Podemos ver que se comprueba, entre otras cosas, que la descripción de las variables es un *string* de un máximo de 255 caracteres (línea 13). Sin embargo, la lógica de negocio solicitaba realizar otras comprobaciones más complejas y específicas: comprobar que la suma de los porcentajes era 100, y asegurarse de que los nombres de todas las variables eran únicos. Para solucionar esto, se añadieron funciones adicionales a las establecidas por defecto, *AllNamesAreUnique* (línea 4) y *AllValuesSum100* (línea 16).

```
export const variablesSchema: SchemaOf<IVariable> = object({
 name: string().max(20, "This field must be at most 20 characters")
  .required("This field is required")
  .test('AllNamesAreUnique', 'The names of the variables must be unique',
   function (value: any): boolean {
    const schema: any = this;
    const namesSet: Set<String> = new Set();
    schema.from[1].value.extraVariables.map(function (obj: IVariable) {
     namesSet.add(obj.name);
    });
   return namesSet.size === schema.from[1].value.extraVariables.length && !namesSet.has("C");
 }),
 description: string().max(255, "This field must be at most 255 characters"),
 value: number().min(0, "This field must be greater than or equal to 0").max(100, "This
   field must be less than or equal to 100").required("This field is required")
  .test('AllValuesSum100', 'The sum of the percentages must be 100',
   function (value: any): boolean {
    const schema: any = this;
    const result = Number(schema.from[1].value.controlVariable.value) +
     schema.from[1].value.extraVariables.reduce((total: number, current: any) =>
     total + Number(current.value), 0);
    return result === 100;
   })
});
```

Figura 17. Esquema de Validación de las Variables

La Figura 18 muestra la función *saveExperiment* como ejemplo de las llamadas AJAX que se realizan desde el *front-end* para comunicarse con el *back-end*. Esta función recibe como propiedades los datos del nuevo experimento y una referencia al método del componente al que debe llamar una vez reciba la respuesta del servidor. Primero se formatea el experimento, *data*, en JSON para que el servidor lo entienda cuando lo reciba (línea 3). Después se especifica la configuración del mensaje (línea 5): método, cabeceras y cuerpo. Finalmente, se envía la petición (línea 16) y se llama al método que se recibió como argumento de entrada junto con el estado del mensaje de respuesta (línea 17).

```
function saveExperiment(props: SaveExperimentServiceProps) {
  let { data, onSaveStatusChanged } = props;
  data = formatExperiment(data);

  const requestOptions = {
       method: 'POST',
       headers: { 'Content-Type': 'application/json' },
       body: JSON.stringify({
           name: data.name,
           description: data.description,
           controlVariable: data.controlVariable,
           extraVariables: data.extraVariables,
       })
  };

  fetch(urlBase, requestOptions).then((response) => {
     onSaveStatusChanged(response.status);
  })
}
```

Figura 18. Método saveExperiment de ExperimentService

4.2.Implementación del service

El *backend* cuenta con tres clases de servicio, una para los experimentos, otra para las variables y otra para las asignaciones. A modo de ejemplo, la Figura 19 muestra la cabecera de la clase *AssignmentService*. Esta clase utiliza el repositorio y el servicio de los experimentos, los cuales se inyectan automáticamente como dependencias utilizando la anotación *@Autowired* (líneas 4

y 7). Para definir que se trata de una clase de servicio, se ha anotado con la etiqueta *@Service* (línea 1).

```
1 @Service
2 public class AssignmentService {
3
4       @Autowired
5       private ExperimentService expSvc;
6
7       @Autowired
8       private ExperimentsSpringRepository experimentRepository;
9
10      (...)
11 }
```

Figura 19. Cabecera de AssignmentService

Esta clase contiene el método *getAssignment*, ilustrado en la Figura 20, que se encarga de devolver una asignación para un experimento y usuario concretos. Para ello, en primer lugar, trata de recuperar la asignación que corresponde al usuario en cuestión de la base de datos (línea 2), utilizando para ello el repositorio de experimentos. Si ya existe una asignación para este usuario se recupera el resto de información necesaria para construir el DTO que se devolverá al final del método (línea 12). Si no existe, se obtiene una nueva asignación para este usuario mediante el algoritmo de generación de asignaciones aleatorias, y se almacena en la base de datos (línea 9). Finalmente, se devuelve el DTO de la asignación indicando si se trata de un *overridden user* o no (línea 16).

```
1  public AssignmentDTO getAssignment(Experiment e, String userId) {
2   String variableId = experimentRepository.findAssignmentByUser(e.getId(), userId);
3   String variableName = null;
4
5   // Si no existe le creo
6   if (variableId == null) {
7     Variable v = AssignmentAlgorithm.createAssignment(e, userId);
8     e.setAssignment(userId, v);
9     expSvc.saveExperiment(e);
10    variableName = v.getName();
11  } else {
12    variableName = experimentRepository.findVariableNameById(Long.parseLong(variableId));
13  }
14
15  // Creo el DTO con la informacion que quiero devolver
16  return new AssignmentDTO(variableName, userId, isOverriden(e, userId));
17 }
```

Figura 20. Método getAssignment de AssignmentService

4.4 Implementación de un controlador

Dentro de la capa de negocio, existen tres controladores que se encargan de implementar la API detallada en el apartado 3.3. Su función es definir el comportamiento que debe tener el servicio ante las distintas llamadas HTTP que reciba por parte de los clientes. El controlador *ExperimentController* se encarga de implementar las operaciones de los recursos *Experiment* y *Experiments*, *VariablesController*, se dedica al recurso *Variables*, y *AssignmentController*, se encarga de las operaciones de los recursos *Assignment* y *Assignments*.

A continuación se procederá a explicar, a modo de ejemplo, la implementación de una de estas clases, en concreto la que se dedica a manejar los recursos *Experiments* y *Experiment*.

Empezando con la cabecera de la clase, la cual se muestra en la Figura 21, observamos que esta se identifica como controlador mediante la anotación *@RestController* (línea 1), mientras que la etiqueta *@ResquestMapping* (línea 2) especifica el prefijo común a todas las URLs que procesa este controlador. En este caso, este controlador procesa todas las peticiones que se hagan sobre las URL que comiencen con */experiments*. Además, se comunica con el servicio *ExperimentService* el cual se inyecta a través de la anotación *@AutoWired* (línea 4).

```
1  @RestController
2  @RequestMapping("/experiments")
3  public class ExperimentController {
4       @Autowired
5       private ExperimentService expSvc;
6
7       @GetMapping
8       @ResponseStatus(code = HttpStatus.OK)
9       public List<ExperimentDTO> getExperiments() throws SQLException {
10              List<ExperimentDTO> experiments = new ArrayList<ExperimentDTO>();
11              for (Experiment e : expSvc.getExperiments()) {
12                      experiments.add(new ExperimentDTO(e.getName(),
13                      e.getDescription(), e.getControlVariable(), e.getExtraVariables()));
14              }
15              return experiments;
16      }
17      ...
18 }
```

Figura 21. Extracto de ExperimentController

A continuación, cada método público del controlador se asocia con una URL y un verbo HTTP determinado, de manera que, cuando se recibe una llamada HTTP sobre esa URL y con ese verbo, se ejecute el cuerpo del método. Cada verbo HTTP tiene su propia anotación. Por ejemplo, para peticiones GET se usa la anotación *@GetMapping*, para peticiones PUT se usa la anotación *@PutMapping*, y así sucesivamente. Además, estas anotaciones pueden tener distintos parámetros especificados, siendo el más utilizado el del *path* al que se asocia el método. Este *path* se concatenaría con el definido en la clase del controlador. En caso de no especificarse ninguno, se entiende que la llamada se hace directamente sobre el *path* de la clase.

A modo de ejemplo, la Figura 21 incluye también el método controlador para la obtención de la lista de experimentos. Este método devuelve una respuesta HTTP (*HTTP Response*) conteniendo una lista con los DTO (*Data Transfer Object*) de los experimentos. Los DTO se utilizan para poder devolver proyecciones de un modelo orientado a objetos que contengan sólo la información necesaria para un determinado propósito y que, además, sean serializables, es decir, que no contenga ciclos. En nuestro caso, cuando se devuelve un experimento se elimina de este su mapa de asignaciones, ya que en la práctica puede llegar a haber miles de usuarios participando en cada experimento. Dado que el mapa de asignaciones no se utiliza para nada de la interfaz de gestión de los experimentos devolver esta información a la interfaz gráfica es añadir peso a cada respuesta HTTP innecesariamente. Además, como en este caso siempre se devuelve un código de respuesta 200, se declara la etiqueta *@ResponseStatus* (línea 8) especificándolo.

El cuerpo del método, tal como se puede ver, funcionaría de la siguiente forma. En primer lugar, se recupera la lista de experimentos invocando a la clase de servicio (línea 11). A continuación, cada experimento devuelto se transforma en su correspondiente DTO y se añade a una lista (líneas 12 y 13), la cual se devuelve al final del método (línea 15). Finalmente, sería el propio framework *Spring* el que se encargaría de crear una respuesta HTTP adecuada y de serializar la lista con los DTO de los experimentos en formato JSON.

Por otro lado el método *postExperiment*, ilustrado en la Figura 22, se encarga de crear los experimentos. Para favorecer la legibilidad del código, parte de las comprobaciones se han suprimido de la figura.

```
1  @PostMapping(consumes = "application/json", produces = "application/json")
2  public ResponseEntity<String> postExperiment(@RequestBody Experiment newExperiment) {
3    String json = null;
4    // 409 - Conflict (resource already exists)
5    if (expSvc.getExperiment(newExperiment.getName()) != null) {
6      return new ResponseEntity<>("An experiment with that name already exists.",
7                        HttpStatus.CONFLICT);
8    }
9    // 400 - Bad Request (malformed syntax)
10    if (newExperiment.getName() == null) {
11      return new ResponseEntity<>("The name of the experiment must be specified.",
12                                                              HttpStatus.BAD_REQUEST);
13   }
14   if (newExperiment.getName().matches(".*\\s.*")) {
15     return new ResponseEntity<>("The names of the experiment can't have blank spaces.",
16                                                              HttpStatus.BAD_REQUEST);
17   }
     (...)
56   // 201 Created
57   expSvc.saveExperiment(newExperiment);
58   ObjectWriter ow = new ObjectMapper().writer().withDefaultPrettyPrinter();
59   try {
60     json = ow.writeValueAsString(newExperiment);
61   } catch (JsonProcessingException e) {
62     throw new RuntimeException(e);
63   }
64   URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
65     .buildAndExpand(newExperiment.getId()).toUri();
66   return ResponseEntity.created(location).body(json);
67 }
```

Figura 22. Implementación del Método postExperiment

En la anotación del método se especifica qué tipo de documento recibe y retorna, en este caso se trata del formato JSON (línea 1). Primero se comprueba que el nombre del nuevo experimento no ha sido asignado aun (línea 5). En caso de que estuviera repetido se devolvería un código 409, ya que por lógica de negocio los nombres de los experimentos deben ser únicos.

A continuación se comprueba el formato del experimento enviado en el cuerpo del mensaje, asegurándose de que se cumple con la lógica de negocio para así evitar que se almacenen experimentos cuyos datos son incorrectos. En total se realizan las siguientes comprobaciones:

- El experimento tiene un nombre asignado y este no contiene espacios en blanco.
- El experimento tiene una descripción asignada.
- El experimento tiene el nombre de las variables asignado.
- El nombre de las variables es único para ese experimento y no contiene espacios en blanco.
- Los valores de las variables son mayores que 0 y menores que 100.
- La suma de los porcentajes del experimento es 100.

Finalmente, si el experimento está correctamente formulado se crea y se devuelve en el cuerpo de la respuesta junto con la cabecera *location* indicando el URI del nuevo recurso (línea 66).

Por brevedad, el resto de métodos del controlador *ExperimentController* se incluyen en el apéndice E.

4.5 Implementación de repository

Los repositorios son las clases que sirven como punto de acceso para la comunicación con la base de datos. Los repositorios se han implementado utilizando las facilidades que proporciona *Spring* para ello. Por ejemplo, simplemente heredando de la clase *JPARepository*, tal como se muestra en la Figura 23, tenemos automáticamente disponibles las operaciones CRUD básicas para la gestión de los experimentos. Además de estos métodos que se proporcionan por defecto, se añadieron 5 más mediante el uso de *custom queries*. Las *custom queries* permiten crear métodos que devuelven como resultado uno o más objetos resultantes de una consulta expresada en el lenguaje *JPQL* (*Java Persistence Query Language*). JPQL es muy similar a SQL, pero se aplica a estructuras orientadas a objetos en lugar de a modelos relacionales. En nuestro caso, se han definido las siguientes *custom queries*:

- El método *findByName* (línea 4) devuelve un experimento a partir de su nombre.
- El método *findAssignmentByUser* (línea 8) devuelve la asignación de un usuario para un experimento concreto.
- El método *findVariableNameById* (línea 11) devuelve el nombre de una variable en función de su identificador.
- El método *getNumberOfAssignments* tiene dos variantes, dependiendo de los argumentos de entrada.
    - Si recibe el identificador de un experimento, devuelve el número de asignaciones establecidas para el experimento (línea 15).
    - Si recibe el identificador de un experimento junto con el identificador de una de sus variables, devuelve el número de asignaciones para el experimento establecidas con esa variable (línea 19).

```
1 public interface ExperimentsSpringRepository extends JpaRepository<Experiment,Long> {
2
3       @Query(value = "SELECT * FROM experiment WHERE name = ?1", nativeQuery = true)
4       Experiment findByName(String name);
5
6       @Query(value = "SELECT assignments_id FROM experiment_assignments WHERE
7                       experiment_id = ?1 AND assignments_key = ?2", nativeQuery = true)
8       String findAssignmentByUser(long experimentId, String user);
9
10      @Query(value = "SELECT name FROM variable WHERE id = ?1", nativeQuery = true)
11      String findVariableNameById(long variableId);
12
13      @Query(value = "SELECT COUNT(*) FROM experiment_assignments WHERE
14                                              experiment_id = ?1", nativeQuery = true)
15      String getNumberOfAssignments(long experimentId);
16
17      @Query(value = "SELECT COUNT(*) FROM experiment_assignments WHERE experiment_id = ?1
18                                          AND assignments_id = ?2", nativeQuery = true)
19      String getNumberOfAssignments(long experimentId, long variableId);
20 }
```

Figura 23. Interfaz ExperimentsSpringRepository

4.4 Lógica de Negocio

En este apartado se muestra el algoritmo desarrollado para la creación de asignaciones como aspecto destacable de la implementación de la lógica de negocio.

4.4.1 Algoritmo de Asignaciones

Como se ha explicado en capítulos anteriores, existen dos formas de realizar asignaciones: se pueden hacer manualmente (*overridden users*), o se puede obtener una asignación de manera automática. Normalmente, se utilizará la segunda opción para crear las asignaciones, por lo que el servidor recibirá un identificador de usuario y el nombre de un experimento y, a partir de esos datos, tendrá que devolver una asignación. Aun así, su obtención no es trivial, ya que se tienen que respetar los porcentajes establecidos para cada variable. Siguiendo estas restricciones, se ha desarrollado un algoritmo cuyo pseudo-código se especifica a continuación:

1. Se genera una lista con todas las variables del experimento.
2. Se ordena la lista de mayor a menor en función de su valor (porcentaje) asignado.
3. Se genera un número aleatorio del 0 al 99.
4. Se crea una variable numérica auxiliar cuyo valor inicial es 0.
5. Se recorre la lista de variables y para cada una de ellas se realiza lo siguiente:
    a. Se comprueba si el número aleatorio es menor o igual que la suma del valor de la variable más el de la variable auxiliar.
        i. Si es menor o igual, se guarda esa variable para asignársela al usuario y se interrumpe la iteración.
        ii. Sino se actualiza la variable auxiliar sumándola el valor de la variable y se continua recorriendo la lista.
6. Después de recorrer la lista, si no se ha obtenido ninguna variable, se almacena la última variable de la lista.
7. Finalmente se devuelve la variable obtenida.

De esta manera se consigue que las asignaciones respeten, con un margen, los porcentajes establecidos por el usuario. Hay que tener en cuenta que los porcentajes real y teórico serán similares pero nunca iguales por los siguientes motivos: las asignaciones se realizan de manera aleatoria, y estas, al representar a personas, son indivisibles.

Por último, cabe mencionar que el generador de números aleatorios es compartido por todos los experimentos, lo que no resulta un problema, ya que estos generadores se comprueban primero con una batería de *tests* estadísticos para ver cómo funcionan en realidad. Dichos *tests* intentan comprobar que, si extraes una subsecuencia de números cualquiera, esta también tiene propiedades aleatorias. En concreto, la propiedad asociada a esta característica se conoce como *escalabilidad*. Por tanto, podemos asumir que el método *nextInt* de la clase Java *RandomGenerator* ha pasado por estas pruebas y, por tanto, cumple con esta propiedad.

5. Pruebas

Este capítulo describe las diferentes pruebas realizadas durante el desarrollo de la aplicación.

5.1 Pruebas Unitarias

Tal como se ha comentado anteriormente, se realizó una única prueba unitaria en el *back-end* sobre el método *getAssignment* de la clase *Experiment*. No se realizaron más pruebas unitarias porque la lógica de las clases de dominio era muy sencilla, y porque se consideró que la parte del servicio junto con los controladores se comprobaría mejor mediante pruebas de integración. El código de esta prueba se muestra en la Figura 24. Esta prueba se realizó con la ayuda de la librería *JUnit*.

```
1  @Before
2  public void setUp() throws Exception {
3       // Initialize the user
4       userId = "Paco";
5       // Initialize the experiment
6       Variable controlVariable = new Variable("C", 100);
7       List<Variable> extraVariables = new ArrayList<Variable>();
8       extraVariables.add(new Variable("A", 0));
9       sut = new Experiment("Exp", "This is an experiment", controlVariable, extraVariables);
10 }
11
12 @Test
13 public void testGetAssignmentUserNotExists() {
14      // User does not exist
15      Variable assignedVariable1 = sut.getAssignment(userId);
16      assertEquals(assignedVariable1, controlVariable);
17 }
18
19 @Test
20 public void testGetAssignmentUserExists() {
21      // User already exists
22      Variable assignedVariable2 = sut.getAssignment(userId);
23      assertEquals(assignedVariable2, controlVariable);
24 }
```

Figura 24. Extracto de ExperimentTest

Primero el método *setUp* (línea 2) se encarga de inicializar las variables que se van a utilizar en la prueba. La etiqueta *@Before* (línea 1) indica que ese método se tiene que ejecutar antes de los tests. Después se comprueban las dos posibles situaciones que se pueden dar al invocar ese método: que el usuario no tenga ninguna asignación para ese experimento, por lo que se le creará una, o que el usuario ya la tenga, por lo que se le devolverá la asignación existente.

El método *testGetAssignmentUserNotExists* (línea 13) se encarga de comprobar que la asignación se crea correctamente para un usuario que no existe. Sin embargo, las asignaciones se realizan aleatoriamente, por lo que era necesario saber de antemano qué variable se iba a asignar para poder realizar el *assertEquals* (línea 16). Para solucionar este problema, se otorgó un 100% de valor a la variable de control (línea 6), de manera que siempre se asigna esa variable.

Por último, el método *testGetAssignmentUserExists* (línea 20), se encarga de comprobar que la asignación de un usuario existente se devuelve correctamente.

5.2 Pruebas de Integración

En total se realizaron 70 pruebas de integración sobre la API diseñada. El objetivo de estas pruebas era asegurar el buen funcionamiento del servicio de manera que no se realizaran

operaciones que incumplieran la lógica de negocio. Para desarrollar estas pruebas se utilizó la herramienta *Postman*, en cuyos *tests* se comprobaba el código de respuesta, el cuerpo y la cabecera (cuando era necesario) de cada respuesta HTTP. La Figura 25 muestra, a modo de ejemplo, dos tests desarrollados para la operación POST del recurso *Experiment*. En este caso se comprobaba el caso de éxito, es decir, una petición en la que el experimento no tuviera un nombre repetido y que sus datos fueran correctos. El primer *test* (línea 1) comprueba que el código de respuesta es correcto y que se ha devuelto la cabecera *location* con la URI correspondiente. En el segundo *test* (línea 7) se comprueban los campos del cuerpo de la respuesta para ver que el experimento se ha creado correctamente.

```
1  pm.test("Successful POST request", function () {
2   pm.expect(pm.response.code).to.be.oneOf([201]);
3   pm.expect(pm.response.headers.get("location")).to.be
4                                                 .eql("http://localhost:8080/experiments/18");
5  });
6
7  pm.test("Experiment values are OK", () => {
8    const responseJson = pm.response.json();
9    pm.expect(responseJson.name).to.eql("Exp");
10   pm.expect(responseJson.description).to.eql("This is an experiment");
11   pm.expect(responseJson.controlVariable.name).to.eql("C");
12   pm.expect(responseJson.controlVariable.description).to.eql("Control description");
13   pm.expect(responseJson.controlVariable.value).to.eql(90);
14   pm.expect(responseJson.extraVariables[0].name).to.eql("A");
15   pm.expect(responseJson.extraVariables[0].description).to.eql("Extra value description");
16   pm.expect(responseJson.extraVariables[0].value).to.eql(6);
17   pm.expect(responseJson.extraVariables[1].name).to.eql("B");
18   pm.expect(responseJson.extraVariables[1].description).to.eql("Extra value description");
19   pm.expect(responseJson.extraVariables[1].value).to.eql(4);
20 });
```

Figura 25. Test POST Experiment (caso éxito)

5.3 Pruebas del Frontend

En este apartado se detallan las pruebas unitarias realizadas a nivel de interfaz en *React*. Tal y como se mencionó en el apartado 1.3.4, se realizaron 14 pruebas sobre 8 de los componentes.

- BasicExperimentComponent
- ControlVariableComponent
- CreateExperimentComponent
- ExperimentComponent
- ExtraVariableComponent
- MenuComponent
- ViewExperimentComponent
- ViewVariableComponent

A continuación, a modo de ejemplo, se detallan los *tests* del componente *CreateExperimentComponent*, el cual se encarga de mostrar el formulario para la creación de un experimento. La Figura 26 muestra el código del mismo.

Este *test* contiene dos pruebas, en la primera de ellas se comprueba que se renderizan todos los cuadros de texto del formulario junto con el botón de guardado. Después, otra prueba se encarga de introducir una serie de datos en el formulario y darle al botón de guardado. A continuación se detalla el funcionamiento de las pruebas,

```
1  describe("CreateExperimentComponent", () => {
2
3    const exp: Experiment = {
4        name: 'ExpTest',
5        description: 'This is a description',
6        controlVariable: {name: 'C', description: '', value: 90},
7        extraVariables: [{name: 'A', description: '', value: 10}]
8    }
9
10   beforeEach(() => {
11      const mockAddListener = jest.spyOn(ExperimentService, 'saveExperiment');
12      mockAddListener.mockImplementation((config) => { config.onSaveStatusChanged(200); });
13
14        render(
15            <MemoryRouter initialEntries={["/create-experiment"]}>
16                <CreateExperimentComponent />
17            </MemoryRouter>);
18    });
19
20    it('initial render', () => {
21        const nameTextField = screen.getByTestId("name-input");
          (...)
29        const submitButton = screen.getByRole('button', { name: /SAVE/i });
30
31        expect(nameTextField).toBeInTheDocument();
          (...)
38        expect(submitButton).toBeInTheDocument();
39    });
40
41    it('successful submit', async () => {
42        const nameTextField = screen.getByTestId("name-input") as HTMLInputElement;
          (...)
49        const submitButton = screen.getByRole('button', { name: /SAVE/i });
50
51        fireEvent.change(nameTextField, { target: { value: exp.name } });
          (...)
58        expect(nameTextField).toHaveValue(exp.name);
59        (...)
60        userEvent.click(submitButton);
61    });
62 })
```

Figura 26. Extracto de CreateExperimentComponentTest

En primer lugar, se inicializa un experimento ficticio (línea 3), cuyos datos serán introducidos más tarde en el formulario. Después, mediante la función *beforeEach* (línea 10), se crea un *mock* para el método *saveExperiment* (línea 11), y se define su implementación (línea 12). Esto se debe a que el botón de guardado llama a la función *saveExperiment* de la clase de servicio *ExperimentService*, por lo que hay que mockerla. Como estamos trabajando con el caso de éxito en el que todos los datos son correctos, se establece que el método devuelva un código 201. Además, se define que antes de cada test se renderice el componente que estamos testeando, *CreateExperimentComponent* (línea 16). Como estamos utilizando un *router*, para poder renderizar el formulario será necesario englobar este último en un componente *MemoryRouter* (línea 15), y definir en él la propiedad *initialEntries* con el *path* correspondiente a ese componente. De esta manera, el router sabrá sobre qué *path* iniciar la prueba.

En el primer *test* (línea 20) se comprueba que todos los cuadros de texto y botones del formulario se renderizan correctamente. Para una mayor legibilidad del código se han omitido la mayor parte de los elementos del formulario. Se han dejado *nameTextField*, cuadro de texto para el nombre del experimento, y *submitButton*, botón de guardado, a modo de ejemplo. Primero se cogen los cuadros de texto (línea 21) y el botón (línea 29) y a continuación se comprueba si se han renderizado correctamente (líneas 31 y 38).

En el segundo *test* (línea 41) se comprueba que se puede guardar con éxito un experimento cuyos datos cumplen con la lógica de negocio de la aplicación. Primero se identifican los cuadros de texto (línea 42), esta vez como *HTMLInputElement* para poder escribir sobre ellos, y el botón (línea 49). Luego, se escribe en cada cuadro de texto los datos del experimento, *exp*, que habíamos definido previamente (línea 51). Finalmente se pulsa en el botón de guardado.

5.4 Pruebas de End-to-End

No se realizaron pruebas End-to-End porque, desde *React*, únicamente recomiendan el desarrollo de este tipo de pruebas para procesos críticos de larga duración, como por ejemplo pagos o registros. Además, como era la primera vez que trabajaba con este paradigma de programación, se suprimieron este tipo de pruebas para facilitar el desarrollo. Las pruebas de integración del *back-end* y las de interfaz del *front-end* se complementan junto con las de aceptación permitiendo obviar las End-to-End. Aun así, si hubiese dispuesto de más tiempo para aprender las herramientas necesarias para realizar este tipo de pruebas, como por ejemplo el framework *Cypress*, sí que me habría gustado realizarlas.

5.5 Pruebas de No Funcionales

Uno de los requisitos funcionales mencionados en el apartado 2.2 era el de corrección funcional. Como demostración del cumplimiento de este requisito, se han realizado una docena de ejecuciones del algoritmo de asignaciones con el objetivo de demostrar que su comportamiento es el adecuado, es decir, las asignaciones se realizan de manera precisa siguiendo los porcentajes establecidos por el usuario. En cada ejecución se realizan 10000 asignaciones para un experimento ficticio de 5 variables (C 55%, A1 25%, A2 10%, A3 8% y A4 2%). La Tabla 4 se muestra el número de asignaciones obtenidas en cada ejecución junto con el valor teórico y la Figura 27 muestra una gráfica con la distribución de estas asignaciones.

		Ejecuciones											
Variable	Valor Teórico	1	2	3	4	5	6	7	8	9	10	11	12
C	5500	5534	5502	5584	5485	5445	5410	5471	5507	5462	5498	5433	5498
A1	2500	2501	2518	2455	2547	2495	2553	2480	2550	2572	2461	2551	2575
A2	1000	950	971	968	1000	1055	1054	1028	983	950	1058	1046	931
A3	800	816	794	816	787	797	779	813	783	815	778	777	809
A4	200	199	215	177	181	208	204	208	177	201	205	193	187

Tabla 4. Número de Asignaciones de la Prueba

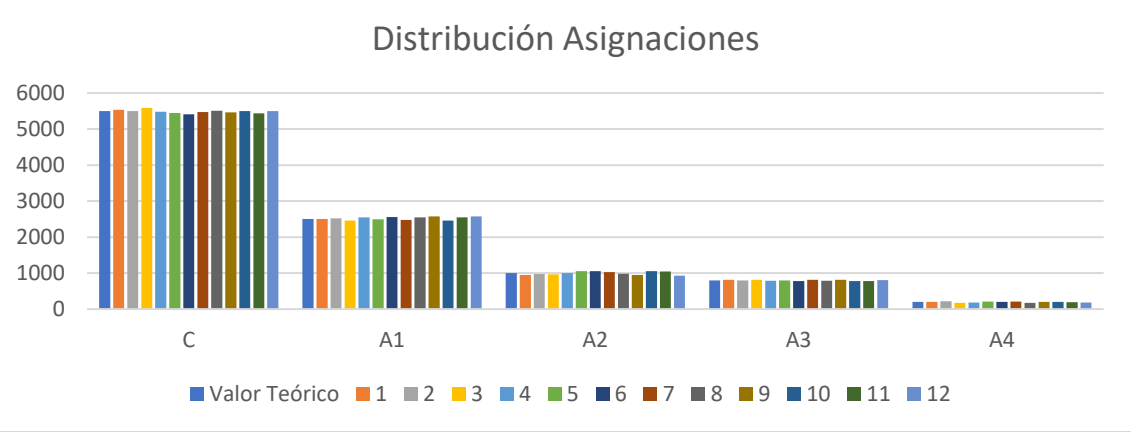

Figura 27. Gráfica de la Distribución de las Asignaciones

Como se puede observar, todos los valores son cercanos al valor teórico. Si calculamos el error medio a partir de estos datos, observamos que este tiene un valor de 28,1666667. Este valor se consideró suficiente por parte de los directores de HP para aprobar la validez del algoritmo.

5.6 Pruebas de Aceptación

Las pruebas de aceptación se realizaron teniendo a los tutores del proyecto como clientes. En todas las reuniones, a excepción de las primeras en donde mis tareas estaban orientadas al aprendizaje de las herramientas de desarrollo, se reservaba un tiempo al principio de la reunión para que les enseñara el trabajo realizado y su funcionamiento. Gracias a ellas se pudieron corregir errores del servicio y se adaptaron las funcionalidades de la aplicación y su interfaz a las necesidades de los clientes.

Durante la demostración de estas pruebas, mi tutor de HP iba tomando notas acerca de los avances que iba realizando y de los aspectos que tenía que mejorar. La Figura 28 muestra un ejemplo de las notas tomadas en una de las reuniones.

19/04/2022

- Ha hecho un cliente de prueba
- El algoritmo de asignación está implementado
    - Su persistencia también
- Usa hook form + yup para formularios y validaciones
    - Faltan validaciones multicampo
- Para el siguiente, poder balancear asignaciones

Figura 28. Extracto de las Notas del Tutor

5.7 Cliente de Prueba

A parte de todas las pruebas mencionadas anteriormente, también se realizó un pequeño cliente de prueba para poder hacer demostraciones visuales de la aplicación. Esta página simulaba una tienda online de productos informáticos y electrónicos, *My Store*, en la cual se mostraba uno de los productos que tenían a la venta junto con su precio y descripción, tal y como se puede ver en la Figura 29.

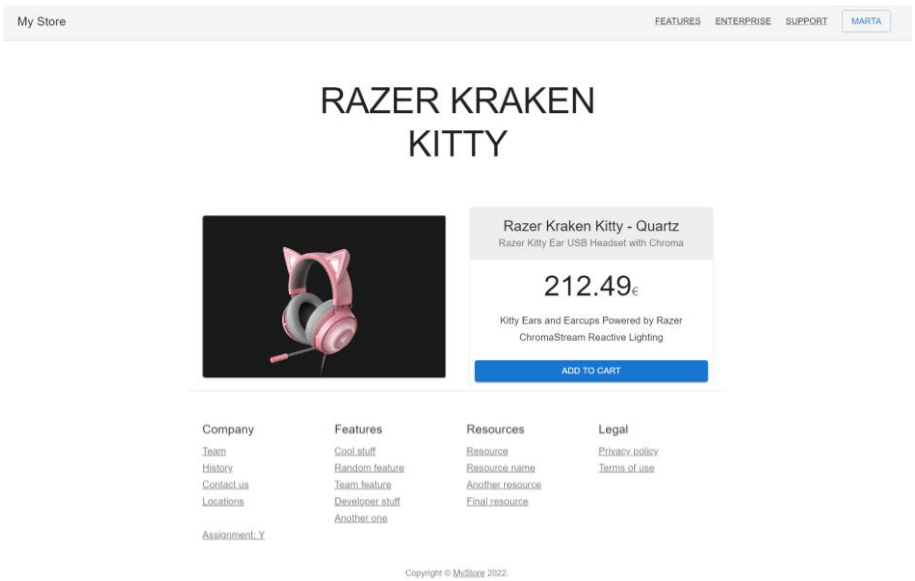

Figura 29. Interfaz Cliente de Prueba

Además, para facilitar la demostración del uso del servicio, y al tratarse de un cliente de prueba, en la parte inferior de la página se muestra la asignación que ha recibido el cliente, pero cabe destacar que esto no sería así si se tratase de una página normal.

Este cliente utiliza un experimento de prueba llamado *PricingExp*, en el cual se muestran diferentes precios en función del usuario que acceda a la página. Su objetivo es analizar a qué precio compran más personas el producto. Este experimento tiene 4 versiones.

- Versión C: el precio es un 100% de precio original (169.99€).
- Versión X: el precio es un 50% de precio original (85€).
- Versión Y: el precio es un 125% de precio original (212.49€).
- Versión Z: el precio es un 75% de precio original (127.49€).

De esta manera, dependiendo de qué versión se le asigne al usuario visualizará un precio u otro. En la Figura 29 podemos observar que al usuario con id “Marta” se le ha asignado la variable Y.

6. Conclusiones y Trabajos Futuros

6.1 Conclusiones

Una vez concluida la descripción completa del proyecto, se puede valorar si este ha cumplido con todos los objetivos propuestos. Principalmente, se pretendía que esta plataforma fuera capaz de crear y gestionar experimentos basados en pruebas A/B, lo que también incluye el manejo de las asignaciones y su correcta redistribución ante cambios en las variables. A partir de esto y una vez comprobado el funcionamiento del servicio, se puede concluir que el proyecto cumple con las expectativas iniciales de manera adecuada. Además, aparte de satisfacer sus necesidades básicas, también se han proporcionado funcionalidades extra como la capacidad de conocer la asignación para un usuario y experimento concretos, o la posibilidad de acceder a una página web a modo de demo que permite a los desarrolladores que quieran hacer uso del servicio ver su funcionalidad aplicada a un ejemplo concreto.

Durante el desarrollo de este proyecto he podido aplicar muchos de los conocimientos aprendidos a lo largo de la carrera y también he aprendido nuevas tecnologías que son frecuentemente utilizadas en la actualidad y que me servirán de ayuda en mi futuro laboral. Por último, considero que he aprendido a ser más autodidacta, a saber comprender y aplicar distintos paradigmas de programación, y a entender documentaciones de APIs y herramientas. Aun así, agradezco que, si había algún concepto que no entendía, mis tutores estaban siempre dispuestos a ayudarme y enseñarme, por lo que tampoco estuve sola durante el desarrollo.

6.2 Trabajos Futuros

Tal y como se ha explicado anteriormente, los propósitos principales del proyecto se han cumplido con éxito. Sin embargo, también se pensaron una serie de funcionalidades extra, no eran esenciales pero que podrían servir de utilidad, que finalmente no dio tiempo a implementar. Por motivos de calendario no fue posible añadirlas al proyecto, pero de cara al futuro su implementación sería de gran utilidad. Estas se ven reflejadas en las siguientes historias de usuario:

- Registrarse en la aplicación.
- Iniciar sesión en la aplicación.

Estas dos funcionales, junto con la implementación de los procesos de autorización necesarios, permitirían que cada usuario pudiera visualizar y manejar su propia lista de experimentos, ya que por el momento, el sistema no hace ninguna distinción, sino que muestra a todos los usuarios los mismos datos.

Siguiendo esta línea, también se podrían gestionar grupos de usuarios, como por ejemplo administrador o gestor, que compartieran una serie de permisos. De esta manera, se podría asignar a cada usuario de la plataforma un grupo o rol para que interactuara con la aplicación de una manera u otra en función de este. Estos grupos se podrían organizar entorno a jerarquías en las que el administrador estaría en la punta de la pirámide, pudiendo realizar cualquier cambio sobre la aplicación, y los usuarios no registrados estarían en la base, no pudiendo realizar más acciones que iniciar sesión en la plataforma.

A partir de estas nuevas funciones, sería conveniente utilizar un servidor de base de datos para poder garantizar su persistencia. Una vez realizado el cambio, el último paso sería desplegar el

servicio en *cloud*, de manera que el *back-end* se pudiera ejecutar de manera concurrente evitando así fallos o estados inconsistentes en la base de datos.

7. Referencias

[1] Dan Siroker. *A/B Testing: The Most Powerful Way to Turn Clicks Into Customers* Wiley, 2020.

[2] Cătălin Tudose. *Java Persistence with Spring Data and Hibernate* Manning, 2022.

[3] Craig Walls. *Spring in Action* Manning, 2022.

[4] Carlos Santana Roldán. *React 17 Design Patterns and Best Practices* Packt Publishing, 2021.

[5] Cătălin Tudose. *JUnit in Action* Manning, 2021.

[6] Dave Westerveld. *API Testing and Development with Postman* Packt Publishing, 2021.

[7] Adam Boduch. *React Material-UI Cookbook* Packt Publishing, 2019.

8. Anexo

8.1 Apéndice A

Sp	Fechas	Funcionalidades implementadas
1	17/01/2022 31/01/2022	Completar el Backlog con nuevas funcionalidades
		Iniciar el aprendizaje de las tecnologías de desarrollo elegidas
2	31/01/2022 07/02/2022	Aprender sobre el diseño de microservicios
3	07/02/2020 21/02/2022	Conectar el formulario del front-end al back-end
		Crear un esqueleto para el back-end
		Crear el controlador para los experimentos
		Crear un esqueleto para el front-end
		Almacenar los experimentos en la base de datos
4	21/02/2022 07/03/2022	Mejorar la validación del formulario para la creación de los experimentos
		Hacer pruebas sobre los componentes de React
		Dividir los componentes de React en otros más pequeños
		Mejorar la interfaz del formulario para la creación de los experimentos
		Crear experimento
5	07/03/2022 21/03/2022	Ver todos los experimentos
		Borrar experimento
6	21/03/2022 04/04/2022	Ver experimento
7	04/04/2022 18/04/2022	Acceder a un cliente de ejemplo
		Solicitar asignación para un experimento
8	18/04/2022 02/05/2022	Aplicar React Hook Form al formulario de crear experimento
		Lanzar experimento
9	02/05/2022 16/05/2022	Filtrar la lista de experimentos por nombre
		Crear *override*
		Modificar experimento
10	16/05/2022 02/06/2022	Acceder a una página web de ejemplo
		Eliminar *overrides* de un experimento
		Ver lista con los *overrides* de un experimento
		Modificar asignaciones de un experimento

Tabla 5. Información de los Sprints

8.2 Apéndice B

ID	Historia de Usuario
RF-01	Aprender sobre el diseño de microservicios
	Como desarrollador, quiero aprender más sobre el diseño de microservicios, de manera que pueda aplicar esos conocimientos al desarrollo e implementación del proyecto.
RF-02	Crear experimento
	Como usuario, quiero crear un experimento y establecer sus parámetros adecuadamente, de manera que pueda comparar la eficiencia entre la interfaz actual y las interfaces modificadas, con el objetivo de mejorar su rendimiento.

RF-03	Ver experimento
	Como usuario, quiero ver los detalles de un experimento concreto, de manera que pueda comprobar su información para simplemente revisarla, o para tomar alguna decisión basada en ella, como por ejemplo, modificar los valores de algún parámetro.
RF-04	Ver todos los experimentos
	Como usuario, quiero poder ver todos los experimentos que he creado previamente, de manera que pueda revisarlos de manera fácil y rápida.
RF-05	Modificar un experimento
	Como usuario, quiero modificar los parámetros de un experimento concreto, de manera que pueda adaptarlo a las necesidades de mi plataforma.
RF-06	Borrar experimento
	Como usuario, quiero eliminar un experimento concreto, de manera que si un experimento deja de ser de utilidad, bien porque ya no es de ayuda o bien porque ha finalizado, pueda deshacerme de él fácilmente.
RF-07	Lanzar experimento
	Yo, como usuario, quiero lanzar un experimento de manera que, una vez finalizado el proceso de evaluación de los resultados, pueda asignar la variable que ha resultado más eficiente a todos los usuarios de un experimento respetando los *overridden users*. Tener esta funcionalidad me resultaría útil para poder aplicar los resultados de los experimentos de manera casi inmediata.
RF-08	Filtrar la lista de experimentos por nombre
	Yo, como usuario, quiero filtrar la lista de experimentos por su nombre de manera que pueda localizar los experimentos más rápido, independientemente del tamaño de la lista. Consecuentemente, conseguiría ahorrar tiempo.
RF-09	Solicitar asignación para un experimento
	Como desarrollador externo, quiero solicitar una asignación para un experimento, de manera que para un usuario con un identificador concreto, el servidor me devuelva una asignación.
RF-10	Crear *override*
	Como usuario, quiero crear un *override* para un usuario con un identificador concreto, de manera que para un experimento X y un id único, pueda establecer una asignación manualmente. Así podré probar la interacción de mi aplicación con el servicio.
RF-11	Ver lista con los overrides de un experimento
	Como usuario, quiero ver la lista con los overrides de un experimento, de manera que pueda conocer el estado de la lista y saber qué variable tienen asignada estos usuarios.
RF-12	Eliminar *overrides* de un experimento
	Como usuario, quiero eliminar los overrides de un experimento, de manera que cuando estos ya no me sean de utilidad pueda poder asignar a esos usuarios una variable de manera automática.
RF-13	Acceder a un cliente de ejemplo
	Como desarrollador externo, quiero tener un cliente de ejemplo en el cual pueda obtener la variable que tiene asignada un usuario a partir de su identificador y el experimento al que está asociado. Esto me será de utilidad a la hora de hacer pruebas y además de poder tener conocimiento de las asignaciones realizadas a través del algoritmo.
RF-14	Modificar asignaciones de un experimento
	Como usuario, quiero cambiar las asignaciones de un experimento de manera que pueda redistribuir los porcentajes entre las variables. Esta operación se podrá hacer modificando el porcentaje de una variable y enviando la parte restante a otra. Se podrá realizar siempre y cuando se reparta un porcentaje menor o igual del que disponga la variable.

RF-15	Acceder a una página web de ejemplo
	Como desarrollador externo, quiero tener una página web a modo de demo para mi plataforma, de manera que pueda visualizar la aplicación de un experimento a un caso real. Así tendré una idea más clara del funcionamiento del servicio y podré hacerme una idea de lo que visualizará cada usuario.

Tabla 6. Historias de Usuario

8.3 Apéndice C

A continuación se detalla la estructura tanto del *front-end* como del *back-end* de la aplicación.

8.3.1 Front-end

La estructura utilizada para implementar el *front-end*, tal y como se muestra en la Figura 30, es la siguiente:

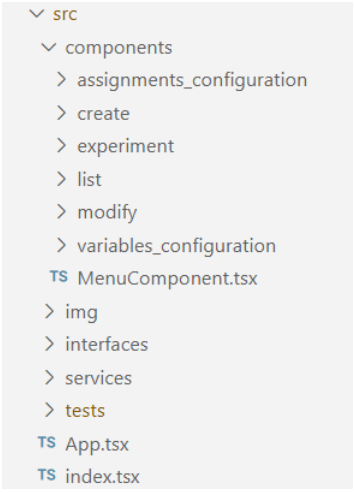

Figura 30. Estructura de clases del front-end

- La carpeta *components* contiene una subcarpeta por cada bloque de la aplicación.
  - En la carpeta *assignments_configuration* se encuentran todos los componentes que constituyen la vista de la configuración de las asignaciones: el número de *overridden assignments* total y el listado con todos ellos (incluyendo la funcionalidad de borrado).
  - En la carpeta *create* se encuentra el formulario para la creación de un experimento.
  - La carpeta *experiment* contiene los componentes necesarios para mostrar los detalles de un experimento y poder lanzarlo.
  - En la carpeta *list* se encuentran los componentes que muestran la lista de todos los experimentos, incluyendo su filtrado, eliminación y modificación.
  - La carpeta *modify* contiene el formulario para modificar un experimento.
  - Finalmente, en la carpeta *variables_configuration* se almacenan los componentes que permiten la reasignación de los porcentajes de las variables.
- La carpeta *img* contiene las imágenes que se utilizan en la interfaz.
- En la carpeta *interfaces* se encuentran las interfaces de todos los tipos de datos que se utilizan en el *front-end*.

- En la carpeta *services* se encuentran las clases que conforman la capa de servicio que se encarga de realizar las llamadas REST y comunicar los datos entre el servidor y la capa de presentación.
- La carpeta *tests* contiene las pruebas realizadas sobre los componentes.

Por último, los archivos *MenuComponent*, *App* e *index* contienen, respectivamente, la lógica del menú de la interfaz, el *layout* que permite que se visualice el menú a la izquierda de la pantalla y los distintos componentes a la derecha, y la lógica del *React Router* que permite definir las distintas rutas de la aplicación.

8.3.1 Back-end

La estructura utilizada para implementar el *back-end*, tal y como se muestra en la Figura 31, es la siguiente:

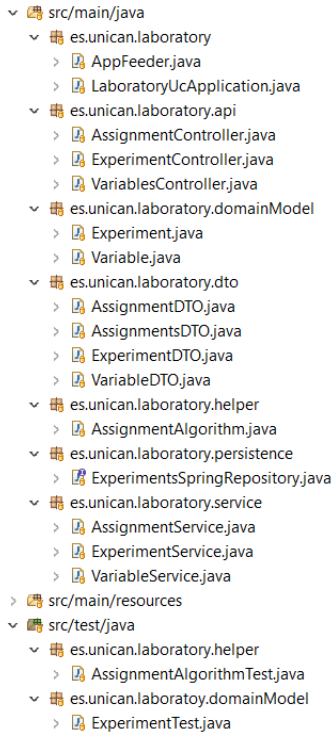

Figura 31. Estructura de clases del back-end

- El paquete *es.unican.laboratory* contiene el *launcher* del servidor y una clase que se encarga de alimentar la base de datos en el arranque de la aplicación. Esto se debe a que, como se trata de una base de datos en memoria, la información se borra cada vez que se cierra el servidor.
- El paquete *es.unican.laboratory.api* contiene las clases con todos los controladores del servicio.
- El paquete *es.unican.laboratory.domainModel* contiene las clases del modelo de dominio del servicio.
- El paquete *es.unican.laboratory.dto* recoge todos los DTO necesarios para comunicar información con el cliente del servicio. De esta manera, se evita el problema de *overfetching*, no llevando al *front-end* datos innecesarios que podrían ralentizarlo.
- El paquete *es.unican.laboratory.helper* contiene el algoritmo que se utiliza para la asignación de variables a los usuarios.

- El paquete *es.unican.laboratory.persistence* almacena la interfaz JPA que automatiza la comunicación con el repositorio de datos.
- El paquete *es.unican.laboratory.service* contiene las clases que conforman la capa intermedia entre los controladores y el repositorio.

Finalmente, la carpeta de *test* está formada por:

- El paquete *es.unican.laboratory.helper* que contiene un pseudo-*test* que se encarga de comprobar la fiabilidad y el funcionamiento del algoritmo de distribución de asignaciones.
- El paquete *es.unican.laboratory.domainModel* que contiene un *test* sobre una pequeña parte de la funcionalidad de la clase *Experiment*, ya que la mayor parte de la lógica de las clases de dominio se basa en *getters* y *setters* simples.

8.4 Apéndice D

A partir de la interfaz de la Figura 12, la cual muestra la lista de experimentos, si se pulsa el nombre de un experimento concreto de la lista, se mostraría la interfaz de la Figura 32. Para el experimento seleccionado se muestra su información principal: nombre, descripción y variables. Además, para cada variable se muestra su nombre, descripción y valor, junto con un botón que permite asignar dicha variable a todos los participantes del experimento, a excepción de los *overridden users*. El objetivo de este botón es, una vez haya finalizado un experimento, poder asignar una misma variable a todos los usuarios de manera rápida.

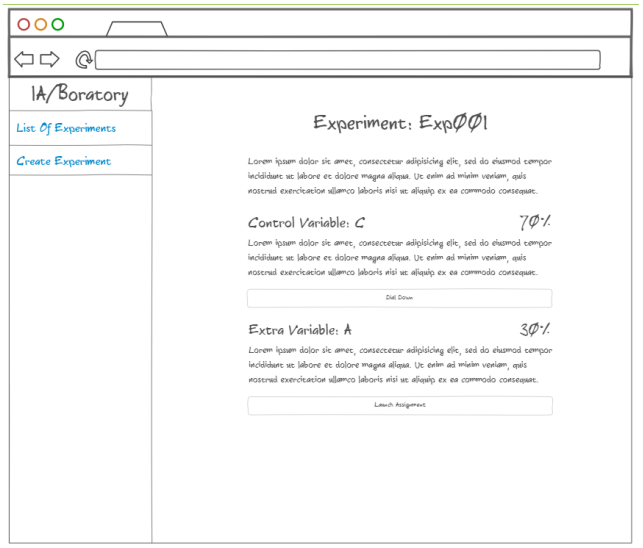

Figura 32. Interfaz Vista Experimento

Por otro lado, si se pulsa en el botón de modificar experimento ilustrado en la Figura 12, se mostraría la interfaz de las Figuras 33 y 34. Este formulario funciona de manera similar al de creación, a excepción de que en este no se pueden modificar ni el nombre del experimento ni el valor de las variables. Los cambios en el valor de las variables se realizan pulsando el botón Variables Configuration, mientras que la gestión de los *overridden users* del experimento se realiza pulsando el botón *Assignments Configuration*.

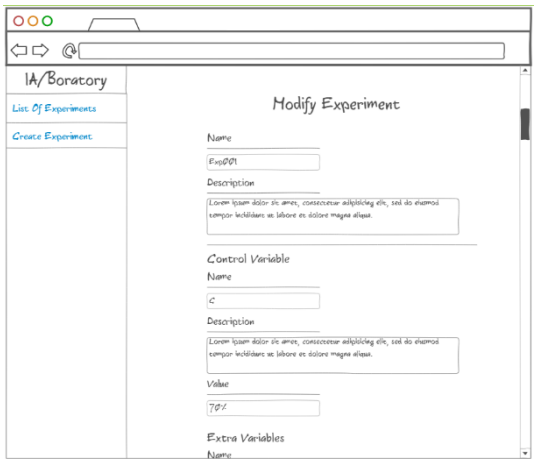

Figura 33. Interfaz Modificar Experimento (parte 1)

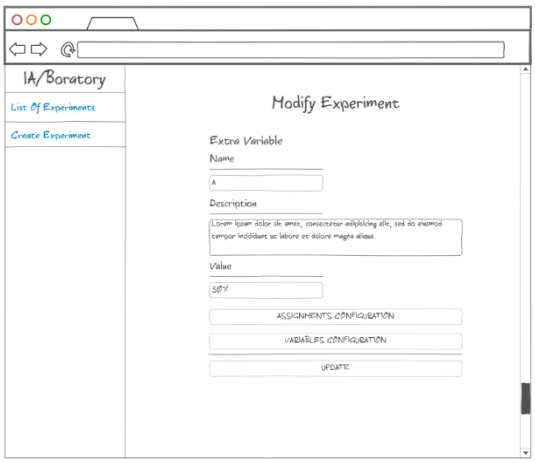

Figura 34. Interfaz Modificar Experimento (parte 2)

Finalmente, si se selecciona la opción de *Assignments Configuration*, se mostrará la interfaz de la Figura 35. Esta nos permite crear o eliminar *overridden users*. Para la creación de este tipo de asignaciones únicamente es necesario seleccionar en el desplegable qué variable queremos asignar y escribir el *id* del usuario para el cual queremos crear la asignación. Para la eliminación se utiliza el botón de borrado de la tabla donde se muestran todas las asignaciones manuales.

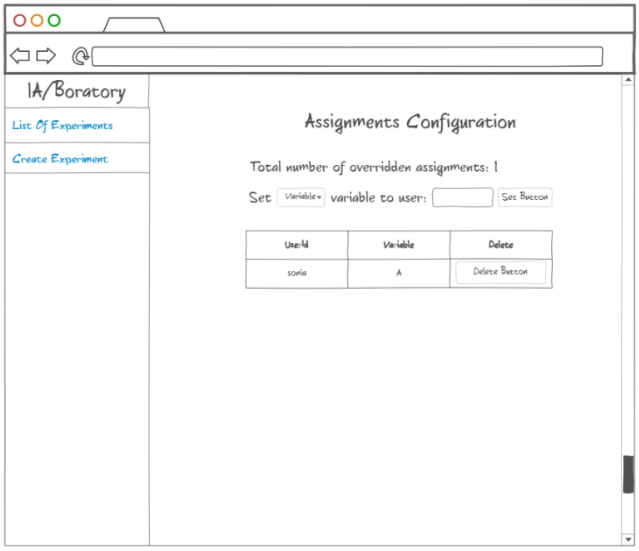

Figura 35. Interfaz Configuración de Asignaciones

8.5 Apéndice E

Para complementar el ejemplo anterior, la Figura 36 muestra la implementación del método controlador para recuperar un experimento concreto. Para poder recuperar un experimento concreto, de acuerdo con la especificación de nuestra API Rest, habría que invocar la URL */experiments/{experimentName}* con el verbo GET. Por tanto, en este caso especificamos como parámetro de la notación *@GetMapping* que hay que añadir el sufijo parametrizado *{experimentName}* a la URL base del controlador. Este fragmento parametrizado de la URL se puede recuperar en la cabecera del método como un parámetro de entrada mediante la anotación *@PathVariable*. En este caso, primero el método recupera el experimento a través del correspondiente servicio. A continuación, comprueba si el experimento existe. Si no existe, devuelve un código 404 especificando el error. Si existe, se devuelve un DTO del experimento solicitado en formato JSON.

```
@GetMapping("/{experimentName}")
public ResponseEntity<String> getExperiment(@PathVariable String experimentName) throws
SQLException {
	Experiment e = null;
	String json = null;
	e = expSvc.getExperiment(experimentName);

	if (e == null) {
		return new ResponseEntity<>("The experiment doesn't exist.",
				HttpStatus.NOT_FOUND);
	}

	ObjectWriter ow =
			new ObjectMapper().writer().withDefaultPrettyPrinter();
		try {
			ExperimentDTO result = new ExperimentDTO(e.getName(),
				e.getDescription(), e.getControlVariable(),
				e.getExtraVariables());
			json = ow.writeValueAsString(result);
		} catch (JsonProcessingException ej) {
			throw new RuntimeException(ej);
		}

		return new ResponseEntity<>(json, HttpStatus.OK);
	}

}
```

Figura 36. Implementación del método getExperiment

El método *putExperiment*, Figura 37, funciona de manera similar al método de creación. En concreto, se diferencia en dos aspectos: en este caso se añade a la anotación del método el fragmento del *path* correspondiente, y después de realizar las comprobaciones se almacena el experimento modificado en la base de datos en vez de crearse. Además, en el método de modificación no se devuelve la cabecera *location*. Para favorecer la legibilidad del código, la parte de las comprobaciones se ha suprimido de la figura.

```
@PutMapping(path="/{experimentName}", consumes = "application/json", produces =
"application/json")
public ResponseEntity<String> putExperiment(@PathVariable String experimentName,
			@RequestBody Experiment modifiedExperiment) throws SQLException {
	Experiment e = null;
	String json = null;
	e = expSvc.getExperiment(experimentName);

	if (e == null) {
		return new ResponseEntity<>("The experiment doesn't exist.",
								HttpStatus.NOT_FOUND);
	}

	// 400 - Bad Request (malformed syntax)
	// (...)

	// 200 Updated
	e = expSvc.modifyExperiment(e, modifiedExperiment);

	ObjectWriter ow = new ObjectMapper().writer().withDefaultPrettyPrinter();
	try {
		json = ow.writeValueAsString(new ExperimentDTO(e.getName(),
			e.getDescription(), e.getControlVariable(), e.getExtraVariables()));
	} catch (JsonProcessingException e1) {
		throw new RuntimeException(e1);
	}

	return new ResponseEntity<>(json, HttpStatus.OK);
}
```

Figura 37. Implementación del método putExperiment

Finalmente tenemos el método para el borrado *deleteExperiment*, ilustrado en la Figura 38. En este caso, si el experimento existe, se elimina y se devuelve en el cuerpo del mensaje de respuesta.

```
@DeleteMapping("/{experimentName}")
public ResponseEntity<String> deleteExperiment(@PathVariable String experimentName) throws
SQLException {
        String json = null;
        Experiment e = null;
        e = expSvc.getExperiment(experimentName);

        if (e == null) {
                return new ResponseEntity<>("The experiment doesn't exist.",
                                                                HttpStatus.NOT_FOUND);
        }

        ObjectWriter ow = new ObjectMapper().writer().withDefaultPrettyPrinter();
        try {
                ExperimentDTO result = new ExperimentDTO(e.getName(), e.getDescription(),
                                        e.getControlVariable(), e.getExtraVariables());
                json = ow.writeValueAsString(result);
        } catch (JsonProcessingException ej) {
                throw new RuntimeException(ej);
        }
        expSvc.deleteExperiment(e);
        return new ResponseEntity<>(json, HttpStatus.OK);
}
```

Figura 38. Implementación del método deleteExperiment