Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN ENXEÑARÍA DO SOFTWARE

erlNote: Aplicación web de planificación personal

Estudante: Alberto Seoane Martínez
Dirección: Laura M. Castro Souto

A Coruña, setembro de 2019.

A A.B.F., *una fuente de inspiración inagotable, ¡sin duda!.*

Agradecimientos

A Laura, por su infinita paciencia, disponibilidad y crítica constructiva.

A todos los que tuvieron a bien poner a mi disposición tanto su software como conocimientos, de forma libre y desinteresada, para que yo pudiera usarlos en la construcción de este "puzzle".

A todos los creadores de contenido que hacen su obra disponible públicamente. En especial, a aquellos que utilizan licencias libres altamente permisivas. Siempre es un gusto trabajar con material sujeto a unas restricciones mínimas.

A Dennis Ritchie, por el lenguaje en el que más me he divertido escribiendo código.

En definitiva, a toda la buena gente con la paciencia suficiente para leer todo lo que sigue.

Resumen

El principal objetivo de este proyecto es estudiar la viabilidad del ecosistema de programación Elixir en el *backend*, así como una serie de nuevas tecnologías en el ámbito del desarrollo web, mediante la creación de una aplicación web/móvil, de página única, cuya temática gira en torno a la planificación personal, y que servirá como medio para la consecución del objetivo esencial.

Abstract

The main objective of this project is to study the feasibility of the Elixir programming ecosystem in the backend, as well as a series of new technologies in the field of web development. This is achieved by creating a single page web/mobile application, which theme revolves around personal planning, and that serves as a means to achieve the aforementioned essential objective.

Palabras clave:

- Aplicación de página única
- Bootstrap
- Ecto
- Elixir
- Erlang
- GraphQL
- Programación funcional
- React
- Redux
- Websocket

Keywords:

- Bootstrap
- Ecto
- Elixir
- Erlang
- Functional programming
- GraphQL
- React
- Redux
- Single Page Application
- Websocket

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Este documento es la memoria técnica correspondiente al trabajo de fin de grado titulado "**erlNote: Aplicación web de planificación personal**".

A lo largo de este documento, cuando se hable de una aplicación web, será para referirse a una aplicación web/móvil de página única (SPA). Con el desarrollo de una SPA se pretende conseguir una aplicación que se ejecute en el navegador del usuario mostrando una fluidez próxima a la que se obtendría en una aplicación de escritorio semejante.

En lo que resta de documento se explicarán los motivos para realizar este trabajo, el proceso llevado a cabo, al igual que las tecnologías empleadas. En el caso de las tecnologías, se tratarán de explicar los conceptos fundamentales, sin profundizar en aspectos demasiado técnicos, puesto que esto último es más propio de manuales de referencia, artículos técnicos, foros de dudas o similares.

1.1 Breve descripción

Este proyecto versa sobre el desarrollo de una herramienta de planificación personal, apoyándose, para ello, tanto en lenguajes de programación funcional como en protocolos/formatos estándar. El desarrollo se centrará, intensivamente, en el *backend*, dejando el del *frontend*, exclusivamente, a modo de demostración.

El concepto de *herramienta de planificación personal* puede resultar excesivamente genérico. Por ello, vamos a tratar de concretar. En este proyecto, cuando nos referimos a una *herramienta de planificación personal*, estamos haciendo alusión a una aplicación que nos permite tomar notas de texto, crear listas de tareas o realizar edición colaborativa de texto, con otros usuarios, mediante pizarras. Como rápidamente es posible intuir, se trata de un trabajo ambicioso, teniendo en cuenta el tiempo limitado y el casi nulo conocimiento del ecosistema a utilizar. Esto nos lleva a establecer prioridades y, en este caso, se ha preferido dar prioridad al desarrollo del servicio en el *backend*.

La herramienta, objeto de desarrollo, ha de poseer las siguientes características:

- *Multiusuario*: Cada usuario debe registrarse para tener acceso a los servicios disponibles. En algún caso, se puede permitir el acceso a servicios concretos, como invitado, sin necesidad de disponer de una cuenta. Opcionalmente, se deja en el aire el desarrollo de características extra, tales como un sistema de detección de bots en el acceso como invitado, por ejemplo.
- *Fácilmente extensible*: El sistema se aborda con un enfoque funcional, basado en módulos, para proveer microservicios. Por consiguiente, tanto la inserción/modificación de funcionalidad, como la reusabilidad de componentes, así como la integración de diferentes *frontends*, debería resultar de escasa complejidad.
- *Segura*: Las comunicaciones hacia/desde el sistema, por parte de los diferentes actores/componentes, han de realizarse por medio de canales y protocolos que permitan un nivel de seguridad aceptable. Es seguro que alcanzar un nivel de seguridad medio/alto queda fuera de los objetivos de este proyecto. En todo caso, la seguridad implementada debe servir como base para la transición, en un futuro, a un nivel superior.
- *Colaborativa/concurrente*: La herramienta permite la comunicación, en tiempo real blando, entre usuarios del sistema, habilitando la colaboración en la consecución de distintos objetivos.
- *Soporte de notificaciones* para usuarios registrados (opcional).
- *Soporte de alarmas/recordatorios* para usuarios registrados (opcional).

1.2 Objetivos

A continuación se pasan a enumerar y describir los objetivos concretos que se desean alcanzar en este proyecto.

- Desarrollo de una aplicación web/móvil, empleando un enfoque puramente funcional. En este caso, se optó por la aplicación descrita en la sección anterior, aunque la propia aplicación, y su temática asociada, no es el objetivo principal del proyecto, como quedó reflejado, en su momento, en el anteproyecto. El objetivo principal del proyecto es explorar las capacidades del ecosistema ofertado por Elixir, y su viabilidad, para el desarrollo de aplicaciones móviles/web.
- Favorecer la mantenibilidad y la calidad del código del proyecto. En todo momento, se tratará de escribir el código más sencillo y claro posible, facilitando su comprensión y posterior mantenimiento.

- Uso de lenguajes de programación extensibles por diseño. De forma que la aplicación del lenguaje a un dominio debe resultar natural y con un esfuerzo ínfimo.
- Construcción de un sistema escalable y tolerante a fallos, que toma como base la máquina virtual de Erlang (BEAM). Deseamos obtener un sistema que se mantenga estable, dentro de unos límites normales de operación, y con un tiempo de respuesta aceptable en las interacciones con el usuario.
- Gestión de la concurrencia poco compleja e intuitiva. Querencia a lenguajes que permitan explotar las capacidades hardware, disponibles actualmente, de forma sencilla.
- No uso de programación defensiva.
- Implementación del *backend* como un *pipeline* de funciones.
- Uso de GraphQL como alternativa a REST, para la comunicación entre *frontend* y *backend*.
- Uso de *WebSockets* para la comunicación, en tiempo real, entre usuarios de la aplicación.
- Integración de un sistema de gestión de base de datos SQL, de la forma más natural posible, teniendo en cuenta la tecnología empleada en el *backend*.
- Consecución de un sistema bien documentado a efectos de memoria, diseño, código, ….

Capítulo 2

Metodología

En este capítulo se aborda la metodología seguida durante el desarrollo del proyecto. Al tratarse de un proyecto con el equipo de desarrollo llevado a la mínima expresión, las principales metodologías, tanto clásicas como ágiles, no son de aplicación.

2.1 Filosofía aplicada durante el proyecto

Esta sección se podría adornar mucho, posiblemente escogiendo alguna metodología, describiendo los conceptos teóricos sobre los que se sustenta, realizando una aplicación ficticia al proyecto, creando diagramas Gantt que, seguramente, nunca fueron una realidad, calculando costes de proyecto sobre bases cuanto menos dudosas, etc. Bien, lejos de seguir esa senda, vamos a optar por el realismo más absoluto, alejándonos así de divagaciones e información ficticia. En este proyecto no se ha seguido metodología alguna, ni nada parecido que se pueda poner bajo el paraguas de ese término. Lo máximo a que se puede aspirar es a afirmar que se han seguido algunas prácticas que se corresponden con las incluidas en algunas metodologías ágiles, y punto.

La filosofía aplicada se asemeja a la predicada por *eXtreme Programming*: "*Hazlo lo mejor que puedas y, entonces, asume las consecuencias*". Para muchos "expertos" esto es escalofriante, para otros es nuestro día a día. Siempre es preferible la mediocridad a una larga espera. La mediocridad de hoy nos puede ayudar a comprender cómo hacerlo mejor mañana. Todo va mejor cuando se trata de ser positivo, actuando como si los recursos fueran a ser suficientes, aún en el supuesto de que no lo fueran. Divagar sobre las limitaciones nos aleja de nuestros objetivos. Un problema no debe verse como algo negativo, debe enfocarse como una oportunidad para aprender y mejorar.

Otra idea mantenida durante este trabajo es la de evitar el estancamiento mediante fallos. La idea consiste en que si uno se queda, en un momento dado, sin saber qué camino escoger, entonces lo que se debe hacer es fallar todo lo posible. Muchas veces, el fallo nos acerca a la

solución e incrementa nuestro entendimiento.

Dado que el tiempo para este proyecto es limitado e inflexible, lo único que podemos controlar es nuestro comportamiento. Desde luego, que lo que no podremos controlar de ninguna manera son las expectativas de los otros. Hacer cuadrar las expectativas generadas con la realidad no suele ser un procedimiento sencillo. Por ello, en este proyecto, sencillamente, se trató de hacer el mejor trabajo posible y comunicar la realidad obtenida de forma clara. En ningún caso se han tratado de esconder los errores, sino que el enfoque adoptado es justamente el contrario: hacerlos públicos. De cualquier manera, el sistema obtenido está muy lejos de ser perfecto, tampoco fue considerada la idea de esperar por la perfección, pero sí que conforma una buena base para un desarrollo futuro. Si alguien busca un punto de partida, casi siempre es más fácil empezar con algo ya hecho que partir de cero.

El proyecto se ha enfocado, desde su comienzo, con una mentalidad abierta. Desde este punto de vista, se han llevado a cabo debates sobre posibles soluciones a problemas que han ido surgiendo, empleando los canales que la comunidad Elixir [2, 3] tiene habilitados para tales fines. Conviene comentar que, dentro de dicha comunidad, se valora muy positivamente la disposición a colaborar y el grado de compromiso con la misma. Por tanto, como para el desarrollo del proyecto no se disponía de un equipo, se consideró que la opción más lógica era rodearse de gente experimentada en el lenguaje principal sobre el que se sustenta el trabajo, obteniendo de esta forma un *feedback* fiable en el que apoyarse. Cuando surgen problemas en el desarrollo, lo más probable es que alguien conozca alguna solución, y la comunidad Elixir lo que hace es facilitar que dicha solución llegue a la persona interesada. Por tanto, de este párrafo puede extraerse que en este proyecto se han tenido en cuenta valores comunes en varias metodologías ágiles como son: comunicación, *feedback* y respeto.

Existen otros valores, como la simplicidad, que ya van ligados al mundo Elixir. En este ecosistema, las soluciones siempre tienden a eliminar la complejidad no necesaria, en la medida de lo posible. Es frecuente, al plantear alguna duda a la comunidad, que la primera pregunta que se hace es: "¿Cuál es la cosa más sencilla que podría funcionar?"

Desde el inicio del proyecto se ha empleado un tablero Trello [4] 2.1. Dicho tablero permite organizar y priorizar las tareas del proyecto de forma fácil y flexible. En este caso, se ha utilizado un tablero con las opciones por defecto. En su configuración inicial, el tablero dispone de cuatro columnas: "lista de tareas", "en proceso", "hecho" y "descartado". En cada columna se pueden añadir tarjetas, que representan tareas o actividades. Según el estado en que se encuentra la tarea, la tarjeta se puede mover de una a otra columna, o modificar propiedades disponibles en la tarjeta, tales como fechas de inicio/fin, alarmas, etc. Mediante este sistema se consiguió planificar de forma flexible la implementación de funcionalidades, respondiendo de forma rápida ante el cambio en las necesidades.

La técnica seguida, para la implementación, consistió en añadir pequeñas funcionalidades

Figura 2.1: Tablero Trello

en intervalos de tiempo cortos. No queremos fases discretas con grandes entregables. Queremos incrementos pequeños y frecuentes. Los grandes cambios abortados, a menudo generan pérdidas de tiempo inasumibles.

Siempre se ha tratado de respetar el *principio de similaridad propia*, copiando estructuras de soluciones en diferentes contextos, por ejemplo, haciendo uso de patrones de arquitectura.

La evolución del proyecto se puede seguir, simplemente, echando un vistazo al log del sistema de control de versiones, puesto que, desde el inicio, está sometido al control de *git* en repositorios Github [5, 6, 7].

Inicialmente, se consideró escribir primero las pruebas, de forma que fallen, y luego codificar hasta que la prueba pase. Afortunadamente, esto no fue necesario. El código Elixir de una función no es, frecuentemente, muy extenso y, por otro lado, suele ser de una claridad excepcional. Por tanto, gracias al lenguaje elegido es bastante intuitivo saber si algo está haciendo

lo que realmente debe. Además, la información referente a errores que genera el lenguaje resulta, sencillamente, de un detalle envidiable.

2.2 Prácticas de metodologías ágiles adoptadas

Durante este proyecto se han tratado de seguir las siguientes prácticas de metodologías ágiles:

- Espacio de trabajo informativo: Que permita a alguien familiarizado con el proyecto hacerse una idea de su situación en un corto espacio de tiempo (*Story cards* (c.f. 2.2).
- Trabajo con energía: No dedicar más tiempo del que se pueda ser productivo. Es frecuente que el exceso de trabajo sea una forma de tratar de controlar una situación fuera de control.
- Historias (c.f. 2.2): Planificación empleando unidades de funcionalidades. Una vez escrita la historia, hay que tratar de estimar el esfuerzo necesario para implementarla. Las historias facilitan la estimación temprana. Las historias deben tener nombres cortos y descripciones breves.
- Ciclo semanal: Se planifica el trabajo de una semana. Cada semana se revisa el progreso realizado. Se escogen las historias para implementar y se dividen en tareas.
- Distensión: Incluir en la planificación tareas menores que pueden ser eliminadas.
- Código compartido.
- Base de código única: Pueden existir ramas temporales, pero tienen una vida muy corta.

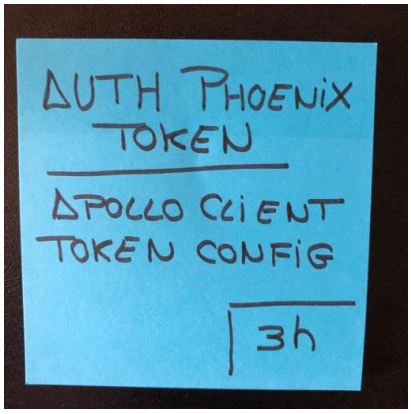

Figura 2.2: Historia

2.3 Coste del proyecto

2.3.1 Situación real

Esta sección se puede resumir en una frase: el coste del proyecto tiende a cero. Simplemente, si el salario del desarrollador se establece a 0€/seg, el único coste imputable sería el de suministros (hardware/electricidad/conexión/...). Teniendo en cuenta que en lo referente al hardware, lo único imputable es el coste de desgaste, el cual es despreciable, obtenemos que lo único relevante sería el coste del suministro eléctrico y el de conexión a internet. A los que hay que sumar los de un par de libros, no disponibles en la biblioteca, para documentación. Lo cual da una estimación del coste en una horquilla [200 - 300]€.

Debido al contexto del proyecto, no procede un análisis de riesgo económico, que se realizaría en un proyecto real para evitar el éxito sin valor.

Al no proceder el aspecto económico, no tiene sentido considerar el valor del dinero en el tiempo. Así que lo único, en lo referente a valor, tenido en cuenta, fue tratar de que las decisiones tomadas tengan un posible valor en el futuro.

En este contexto, podemos concluir que el trabajo tiene un elevado valor, en términos de satisfacción personal, pero un valor nulo en tanto a dinero se refiere.

2.3.2 Situación ficticia

A continuación se presenta una tabla en la que se desglosa, más o menos, el tiempo empleado en la realización de este proyecto. Más tarde, se realiza un cálculo, aproximado, del coste del trabajo en base a dos supuestos ficticios que, en ningún caso, deberían tomarse como base fiable en situaciones reales.

Tabla 2.1: Tareas del proyecto.

Tareas del Proyecto	
Tarea	Tiempo (horas)
Determinación de estado del arte	
Análisis de software viable (Angular, Ember, Material, etc.)	15
Total parcial	**15**
Adquisición de conocimientos necesarios para el desarrollo del *backend*	
Repaso del lenguaje Erlang	7
Elixir: Tutorial oficial	5
Elixir: Libro *Elixir in Action*	20
Elixir: Libro *Programming Elixir*	7

Continuación de Tareas del Proyecto 2.1	
Tarea	Tiempo (horas)
Persistencia Ecto: Documentación oficial	5
Persistencia Ecto: Libro *Programming Ecto*	10
Phoenix Framework: Guías oficiales	7
Phoenix Framework: *Plugs*	$\frac{1}{2}$
Phoenix Framework: Sistema Pub/Sub	$\frac{1}{2}$
Phoenix Framework: *Sockets*	1
Phoenix Framework: Libro *Programming Phoenix*	5
Phoenix Framework: *Phoenix Tokens*	$\frac{1}{2}$
GraphQL: Especificación oficial	2
GraphQL: Documentación oficial	3
GraphQL: Libro *Craft GraphQL APIs in Elixir with Absinthe*	21
Familiarización con iex	1
Estudio del sistema de Ports de Erlang	3
Familiarización con la herramienta Mix	1
Total parcial	**99 y $\frac{1}{2}$**
Determinación de la arquitectura del sistema	
Repaso patrones de arquitectura *software*	1
Arquitectura: Libro *Functional Web Development with Elixir, OTP and Phoenix*	3
Estudio arquitectura *backend*	1 y $\frac{1}{2}$
Estudio arquitectura *frontend*	2
Total parcial	**7 y $\frac{1}{2}$**
Debates, resolución de problemas e interacción con la comunidad	
Debates en Elixir Forum y Slack	120
Total parcial	**120**
Diseño del sistema	
Diseño *backend*	2
Diseño *frontend*	2
Creación modelo de datos	2
Total parcial	**6**
Adquisición de conocimientos necesarios para el desarrollo del *frontend*	
Estudio lenguaje JavaScript	20
Estudio CSS: Libro *CSS in Depth*	12

Continuación de Tareas del Proyecto 2.1	
Tarea	Tiempo (horas)
Redux: Tutorial	3
Redux: Documentación oficial	4
Bootstrap: Documentación oficial	8
React: Tutorial	1
React: Guías oficiales	2
React: Libro React Quickly	5
React: Libro React For Real	1
Total parcial	**56**
Configuración del sistema	
Phoenix Framework: Integración GraphQL (Absinthe)	1
Configuración entorno desarrollo/servidores *backend*	1 y $\frac{1}{2}$
Configuración entorno desarrollo/servidores *frontend*	$\frac{3}{4}$
Configuración política CORS backend	$\frac{3}{4}$
Configuración del cliente Apollo GraphQL para trabajar con Absinthe	3
Total parcial	**7**
Implementación del *backend*	
Implementación Ecto (Persistencia)	12
Implementación GraphQL/Absinthe (consultas, mutaciones, suscripciones)	19
Implementación autorización/autenticación *backend*	4
Implementación del patrón "me" GraphQL	3
Análisis del código fuente de suscripciones Absinthe	5
Total parcial	**43**
Pruebas del *backend*	
Pruebas unitarias *backend*	12
Prueba de la API con GraphiQL	8
Total parcial	**20**
Implementación del *frontend*	
Cliente Phoenix.js: Estudio código fuente/reconexión	5
Implementación *frontend*	43
Total parcial	**48**
Pruebas del *frontend*	
Pruebas manuales *frontend*	15
Total parcial	**15**
Control de versiones	

Continuación de Tareas del Proyecto 2.1	
Tarea	Tiempo (horas)
Familiarización con control de versiones Git	1
Estudio de versiones semánticas	$\frac{1}{4}$
Estudio de opciones de integración continua	1
Total parcial	**2 y $\frac{1}{4}$**
Tareas de documentación de código	
Documentación funciones *backend*	3
Colaboración en traducción del proyecto exDoc a español/gallego	30
Total parcial	**33**
Elaboración de la memoria del proyecto	
Memoria del proyecto	38
Total parcial	**38**
Total	**510 y $\frac{1}{4}$**
Fin de Tareas del Proyecto	

Coloquialmente, se denominan *"cárnicas"* [8, 9, 10, 11] a las empresas, en su mayor parte consultorías informáticas, que contratan trabajadores por salarios realmente bajos [12], con jornadas laborales ciertamente extensas y cuya plantilla acostumbraba a ostentar, en un elevado porcentaje, escasa o nula cualificación. Y decimos acostumbraba porque, en los últimos años, también muchos profesionales realmente cualificados se han visto obligados a aceptar estas, vamos a decir, no muy óptimas condiciones. Estamos hablando de técnicos bien formados, con salarios de 14000€ brutos anuales y dedicaciones semanales que, frecuentemente, superan las 50 horas (extraoficialmente). No es ningún secreto que estas empresas se ocupan/desarrollan proyectos para empresas públicas y grandes empresas. Desafortunadamente, aunque las "cárnicas" trabajen para grandes empresas, este último adjetivo está muy lejos de poder ser aplicado a las mismas. La mediocridad de su trabajo ha quedado patente al saltar a la luz pública escándalos como LeXNet [13, 14], RentaWeb [15], errores electorales [16, 17, 18, 19] o la escasa calidad de la web de Renfe [20, 21]. Todos estos casos no son otra cosa que la reafirmación de que la explotación de los empleados hasta límites insanos, las jornadas laborales maratonianas, los ambientes de trabajo tóxicos y estresantes, no parecen, vistos los resultados, el mejor camino para obtener un *software* de calidad.

Estas empresas son una verdadera lacra para los profesionales de la informática, y los que pasan por el aro, permitiéndoles subsistir cual parásitos, no son mucho mejores. Bien es cierto, que la anestesiada sociedad respecto a la movilización en contra del empleo precario crea un contexto que es el caldo de cultivo perfecto para este tipo de empresas. El *modus operandi* es muy fácil: la administración pública, en su afán de no adquirir una relación laboral

directa con un profesional cualificado, externaliza un servicio mediante una empresa "cárnica" interpuesta. Esta empresa interpuesta lo que hace a menudo, además de facturar a precios desorbitados, es delegar el trabajo en personas, muchas veces, infracualificadas. Por tanto, es habitual que estas empresas contribuyan al intrusismo laboral en el sector de las tecnologías de la información y, lo que es peor, este intrusismo, sumado a la pésima calidad del producto entregado, provoca la creación de una imagen realmente mala de los profesionales del sector a ojos de la sociedad. La mecánica del negocio es muy simple: cobrar mucho, pagar poco, horarios excesivos, trato personal lamentable y ambiente estresante.

Suele ser común que, al externalizar un servicio, no se contrate únicamente a una empresa interpuesta, sino a varias en cadena. De esta forma, se consigue oscurecer el rastro de la inversión realizada en el proyecto, quedando diluido en los eslabones de dicha cadena, al mismo tiempo que se dificulta la trazabilidad respecto a quién está realizando realmente el trabajo.

Observando la proliferación de las empresas "cárnicas", no deja de llamar la atención que el Estado gaste millones de euros, formando a gente muy cualificada, en centros públicos, para acabar derrochando otra magna cantidad de dinero, subcontratando empresas que realizan proyectos manifiestamente chapuceros a enorme coste, que además no muestran ni la más mínima sensibilidad hacia el bienestar del empleado. Como cualquiera puede deducir, vistos los productos entregados por este tipo de empresas, pongamos LeXNet, el código espagueti nunca estuvo tan bien pagado (esto no quiere decir que el que lo escribe esté bien pagado, más bien al revés). A modo de reflexión personal, simplemente afirmar que lo que estas empresas están perpetrando es una verdadera perversión de las metodologías aplicables a proyectos informáticos. Es verdaderamente vergonzoso que algunas incluso se jacten de emplear metodologías ágiles, cuando la filosofía real en la que encajan es, claramente, en la de la precariedad y la explotación. Lo único ágil que conocen es exprimir al empleado para que suba la velocidad y realizar la entrega en tiempo, aunque muchas veces no en forma.

Si asumimos un analista-programador, de una consultoría "cárnica", con una jornada de 40 horas semanales y un sueldo de 25000€ brutos anuales en 14 pagas, para nuestro proyecto de 510 horas y cuarto, nuestro trabajo estaría concluido, aproximadamente, en 13 semanas. El sueldo de nuestro desarrollador rondaría los 6800€ brutos y, teniendo en cuenta que estas empresas suelen aplicar sobre un 30% de beneficio, el trabajador nos saldría por unos 8840€. Si a esto le sumamos otros gastos relacionados con el proyecto, pongamos toma de requisitos, reuniones, ..., siendo generosos, por unos 18000€ tendríamos nuestro proyecto, con total seguridad en la fecha indicada. El principal problema sería la calidad del mismo, como quedó patente en los casos citados anteriormente. Probablemente, el sistema entregado funcione, probablemente cumpla casi todos, sino todos, los requisitos del cliente. Pero también, seguramente, el tratamiento de los datos no sea el deseable, las políticas de seguridad sean bastante

laxas, las pruebas mínimas y poco rigurosas, etc. Conviene recordar que, en este tipo de empresas, suele estar muy extendido el mantra *"Entregar y, más tarde, parchear"*. En conclusión, en la primera entrega es habitual recibir un "sistema hecho con chicles y alambres". Luego está, la duda ofende, el negocio de los parches; parches que, en su mayoría, podrían haberse evitado si se hicieran bien las cosas desde un principio.

Supongamos ahora que queremos contratar nuestro proyecto con una empresa éticamente intachable, que goza de una excelente reputación en el sector, al tiempo que cuida a sus empleados permitiéndoles conciliar su vida personal y familiar con horarios flexibles e, incluso, teletrabajo. Bien, en este caso, es impensable, como sucedía con el tipo de empresa anterior, que alguien haga de analista-programador sobre la marcha, los roles suelen estar claramente delimitados. Esta clase de empresas suele asignar un equipo de profesionales cualificados al proyecto, cada uno especializado en un área concreta (arquitectura, análisis, etc.). Pero para no crear un contraste demasiado radical, supondremos que se le asigna un analista y un programador. Estas personas disfrutan de una jornada laboral de 30 horas semanales, configurables a gusto, y un salario de 50000€ brutos anuales en 14 pagas. Luego, para nuestro proyecto de 510 horas y cuarto, nuestro trabajo estaría concluido, aproximadamente, en 17 semanas. Por tanto, nuestros desarrolladores costarían sobre 17723€/persona y, considerando un beneficio del 30%, saldría más o menos por 23040€/persona, y el equipo de desarrollo completo por 46080€. Al tratarse de una empresa de primer nivel, probablemente, los gastos que rodean el proyecto sean algo más elevados. Por lo que vamos a concluir que el precio final oscila en torno a los 60000€. El triple del precio obtenido en el caso anterior. Eso sí, seguramente obtendremos un producto mucho más cuidado, mejor documentado, y que genere muchos menos problemas/preocupaciones/sanciones, que en el primer supuesto.

Llegados a este punto, podemos afirmar que, a priori, todos los caminos son viables y la decisión de transitar por uno u otro depende, en gran medida, del riesgo que se esté dispuesto a asumir. Desde el punto de vista ético, el camino está claro. Desde el punto de vista económico ¿es, realmente, más barato un *software* sometido a continuos parches y que genere, probablemente, futuras sanciones a pagar?

Por último, hemos de comentar que no todo el software creado por empresas "cárnicas" es, ciertamente, mejorable. En estas empresas, por suerte para ellas, aún es posible encontrar, de vez en cuando, algunos magníficos profesionales. Pero, en su mayoría, se trata de gente que no está en esas empresas para hacer carrera, sino más bien de forma esporádica o para ganar experiencia. Hace unos años, a esta gente era frecuente contratarla directamente, puenteando así a la empresa "cárnica", de forma que se establecía un contrato directo con el cliente, permitiéndole cobrar un salario acorde a su trabajo. Desafortunadamente, estas empresas se dieron cuenta de la situación, introduciendo cláusulas en los contratos para evitar, precisamente, este tipo de situaciones.

Capítulo 3

Arquitectura software del sistema

En este capítulo presentamos y describimos la *arquitectura software* implementada en el proyecto. Esto es, las estructuras fundamentales del sistema software. Normalmente, las decisiones tomadas, cuando se define la arquitectura, son costosas de modificar una vez implementado el sistema.

3.1 Arquitectura software

La *arquitectura software* de un programa o sistema es la *estructura de estructuras* del mismo, incluyendo los *elementos software*, las *propiedades externamente visibles* de dichos elementos, y las *relaciones* entre ellos. La visión arquitectónica de un sistema se centra en el comportamiento e interacción de "cajas negras".

La *arquitectura software* se considera una fase previa al diseño, y puede verse como una manifestación de las primeras decisiones de diseño.

La arquitectura de un sistema determina ciertos requisitos, no funcionales, del mismo. Los *parámetros de calidad* (requisitos no funcionales) nos permitan evaluar la calidad de una forma objetiva. Esto nos lleva a concluir que: nuestras decisiones de arquitectura determinarán la calidad del sistema desarrollado, independientemente de la funcionalidad.

Los parámetros de calidad del sistema son, fundamentalmente, seis:

1. *Disponibilidad*: Resistencia a fallos (observables por el usuario). Se consigue con redundancia, que requiere de sincronización.

2. *Flexibilidad al cambio*: Medida del coste que un cambio tiene en el sistema.

3. *Rendimiento*: Respuesta a eventos temporales en un intervalo aceptable.

4. *Seguridad*: Capacidad del sistema de resistir el uso no autorizado, al mismo tiempo que sigue atendiendo peticiones legítimas.

5. *Facilidad de prueba*: Facilidad de conseguir que el *software* revele sus fallos, utilizando pruebas.

6. *Usabilidad*: Facilidad de usuarias o usuarios para realizar la tarea deseada, con el soporte que el sistema proporciona.

En nuestro caso, cabe comentar, que el sistema obtenido presenta:

- Rendimiento: El sistema obtenido presenta, a simple vista, un buen rendimiento, con una respuesta a eventos dentro de límites normales, sin apreciar retardo alguno en la interacción, tanto en el *frontend* como en el *backend*. Cabe resaltar que esto es una apreciación subjetiva a falta de pruebas que avalen dicha conclusión.

- Flexibilidad al cambio: El *backend* basado en Elixir [3] y GraphQL [22] B.7 presenta una extraordinaria flexibilidad al cambio. Introducir cambios en el *backend* no supone, en caso alguno, una complejidad excesiva, fuera de los límites que se consideran normales.

- Seguridad: Se ha obtenido un nivel de seguridad adecuado, teniendo en cuenta el tipo de sistema con el que estamos tratando. El uso de *tokens* de autenticación, junto con protocolos cifrados de comunicación, en producción, constituye una solución más que aceptable, en este aspecto.

- Facilidad de prueba: Aquí conviene diferenciar entre *backend* y *frontend*. El *backend* es extremadamente simple de probar, el cuidado detalle de fallos/errores, exhibido por Elixir, junto con el *framework* de pruebas ExUnit [23] B.12 (proporcionado por los creadores del lenguaje), hacen que revelar los fallos del *software* sea, prácticamente, un juego de niños. De todas formas, para la gente más exigente, aún es posible rozar la perfección mediante la adopción del *framework* CommonTest [24] de Erlang. Cabe comentar, que ExUnit B.12, a pesar de estar enfocado a pruebas de unidad, puede ser utilizado, adicionalmente, para pruebas de integración. En el caso del *frontend*, debemos lamentar que JavaScript no se acerca, ni de lejos, al nivel ofrecido por Elixir. Por falta de tiempo, las pruebas realizadas en el *frontend* se han limitado a pruebas manuales. Mediante el uso de la extensión Redux B.6.6, F para el navegador, las herramientas de desarrollo React y las propias herramientas de desarrollo ofrecidas por el navegador web, se ha conseguido probar el sistema con éxito de forma no muy compleja, pero más de la deseable. No obstante, la propia naturaleza de los *frontends*, hoy en día, donde mucha de la complejidad, antaño en el *backend*, ha sido trasladada al *frontend*, no hace esta parte de la arquitectura especialmente fácil de probar.

- Usabilidad: En lo referente a la usabilidad, el sistema desarrollado es, ciertamente, mejorable. El *frontend* presentado se considera una base sólida sobre la que evolucionar.

No obstante, está lejos de ser una versión destinada al usuario final. Si alguien desea hacerse una idea, el nivel se correspondería con una versión *alpha* que, en ningún caso, alcanzaría el nivel de *beta*. Más allá de la funcionalidad disponible, el usuario, casi al instante, echaría de menos un sistema bien documentado.

- La disponibilidad, en principio, no sería un problema, puesto que los sistemas basados en Erlang suelen gozar de una alta disponibilidad. Estos sistemas son frecuentemente distribuidos, con nodos replicados. Los procesos ligeros contribuyen a que el número de procesos, y por tanto la creación de los mismos, no sea un problema. Al mismo tiempo que el cambio de código en caliente ayuda, por ejemplo, a que el sistema permanezca disponible aún durante una actualización de código.

3.2 Arquitectura del sistema

Visto en su conjunto, el sistema se ajusta a una arquitectura cliente-servidor clásica. El servidor expone servicios, en este caso una API GraphQL B.7, que son consumidos por el cliente.

El sistema consta de dos partes completamente diferenciadas, un *frontend* (del lado del cliente) y un *backend* (del lado del servidor). *Frontend* y *backend* son desarrollos completamente independientes uno del otro, de hecho, ambos están alojados en sendos repositorios *git* en Github [6, 7].

La única restricción para una exitosa comunicación, entre cliente y servidor, no es otra que hablar y entender GraphQL, así como soportar protocolos estándar de comunicación cliente-servidor (que cualquier navegador web actual soporta sobradamente). El cliente debe generar documentos según el esquema GraphQL residente en el servidor, al mismo tiempo que entender las respuestas JSON, a las consultas enviadas, haciendo uso de dichos documentos. En desarrollo, se usarán protocolos de comunicación sin cifrar, mientras que en producción se cambiarán por sus homólogos cifrados.

3.2.1 Arquitectura del *backend*

El *backend* goza de una sencilla arquitectura (c.f. 3.1) basada en un simple servidor web. Dicho servidor constituye los cimientos en los que se apoyan los servicios suministrados por el *backend*. En este caso, los servicios de los que hablamos son una API GraphQL basada en HTTP [25] y la misma API basada en Websockets [26]. Finalmente, existe una capa de persistencia, basada en el clásico patrón Repositorio [27], para operar contra el sistema de gestión de bases de datos (SGBD).

La base arquitectónica y, por tanto, la piedra angular sobre la que se sustenta todo el *backend*, es la BEAM (máquina virtual de Erlang). Se trata del componente más crítico cuando

hablamos del servidor. Con esto, se está dando a entender, que la aplicación desarrollada no constituye un contraejemplo de uso de la BEAM (por ejemplo, contextos en los que se hace obligatorio el uso de código nativo).

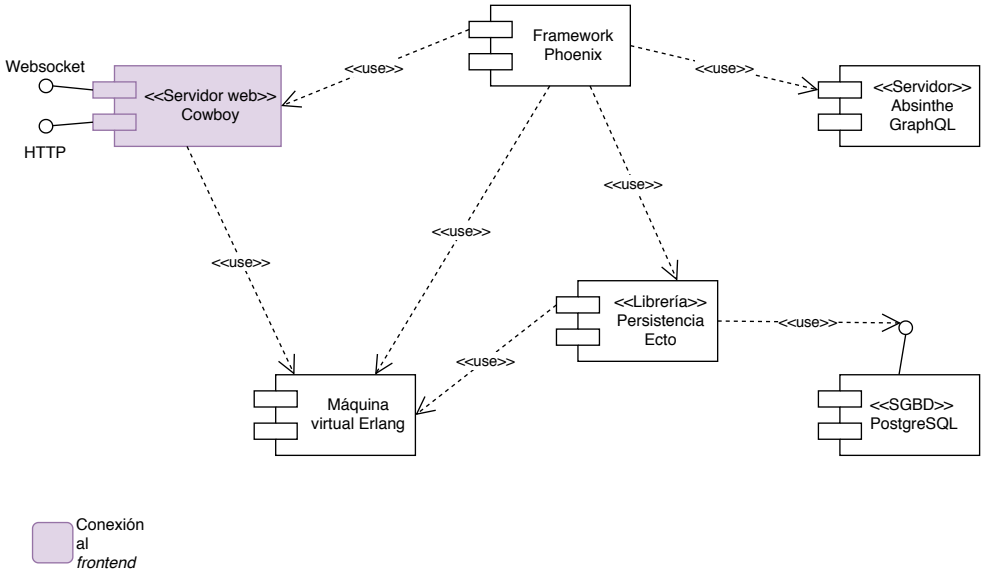

Figura 3.1: Arquitectura software del *backend*

3.2.2 Arquitectura del *frontend*

El *frontend* presenta una arquitectura (c.f. 3.2) algo más compleja de lo que se da en el caso del *backend*. El *frontend* consiste en una aplicación web de página única (SPA). Para conservar el estado local de la misma, se usa una implementación simplificada del patrón FLUX [28] B.6.5.

Con el fin de obtener una comunicación (c.f. 3.3) con el *backend* óptima y asíncrona, se emplea un cliente GraphQL genérico y un cliente *WebSocket* específico y hecho a medida, esto es, optimizado. El cliente GraphQL, a su vez, proporciona una caché que se ha de mantener actualizada. Es de vital importancia que no se rompa la fuente única de verdad impuesta por la versión empleada de FLUX, por lo que se debe tratar con sumo cuidado la interacción entre el cliente GraphQL y el almacenamiento de FLUX.

El *frontend* gira en torno a un motor de navegación capaz de entender estándares web, principalmente: HTML, ECMAScript y CSS. Para mantener cierta información puntual, se hace uso, en casos muy concretos, del almacenamiento local suministrado por el navegador. Esto complementa el almacenamiento obtenido al introducir el patrón FLUX, ofreciendo redundancia, en casos muy específicos, de datos críticos.

La vista de la interfaz gráfica de usuario (GUI) está soportada por una librería que nos per-

mite concebirla como un árbol jerárquico de componentes B.6.3. Esto contribuye a fomentar la reutilización de los distintos elementos/componentes de la GUI. Además se hará uso de un *framework* que permite a la GUI adaptar su aspecto en función del tipo de dispositivo en el que se visualiza.

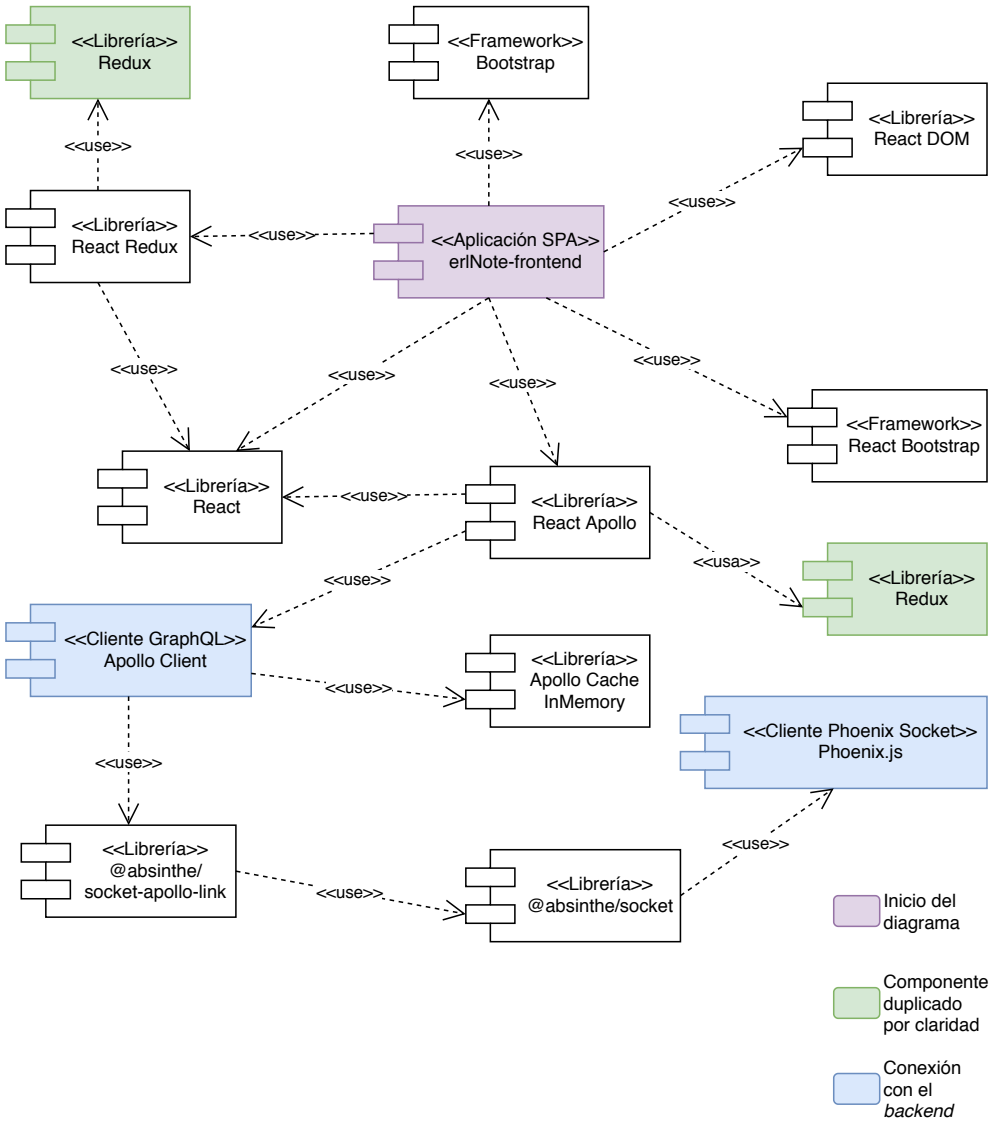

Figura 3.2: Arquitectura software del *frontend*

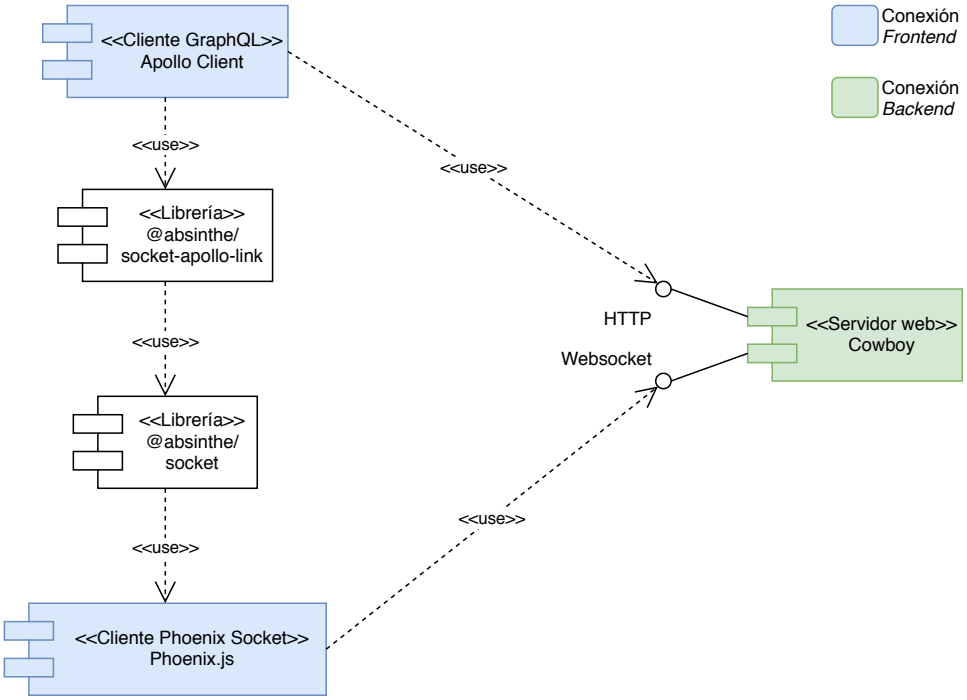

Figura 3.3: Arquitectura comunicación *frontend-backend*

Capítulo 4

Diseño e Implementación

En este capítulo de la memoria se describe el desarrollo del sistema llevado a cabo. Durante el mismo, se abordarán temas como el modelo de datos empleado, el diseño del sistema realizado o las decisiones de implementación adoptadas, así como las diferentes tecnologías empleadas.

4.1 Modelo de datos

En esta sección explicaremos el modelo de datos, empleado en el *backend* por el sistema de gestión de bases de datos (SGBD). Para ello haremos uso de diagramas entidad-relación clásicos, que utilizan una notación de máximos y mínimos. Como nota informativa, se aclara que: algunas entidades/relaciones, estrictamente hablando, no necesitarían un identificador numérico entero, no negativo, como clave principal. Dicho campo se incluye porque en la documentación de Ecto [29] B.10 se aconseja hacerlo así. Esto es, dejar las claves primarias que se generan por defecto, a no ser que haya un motivo, de mucho peso, para emplear otras diferentes. La toma de esta decisión puede resultar controvertida, pero debido a la falta de experiencia en el desarrollo con Ecto, se ha preferido seguir los consejos proporcionados por los desarrolladores de la librería.

El modelo de datos, aquí descrito, ha sido implementado haciendo uso de la librería de persistencia Ecto [29] B.10 de Elixir [3] B.9, exclusivamente. Aunque, como es lógico, se requiere un SGBD subyacente, en este caso PostgreSQL [30], no fue necesario ningún tipo de operación directa sobre dicho SGBD, más allá de lo que se refiere a la configuración de los usuarios y permisos pertinentes.

Como curiosidad debemos mencionar que, en un principio, la idea original fue utilizar una base de datos No-SQL basada en Erlang: CouchDB [31]. Esta base de datos documental se ajustaba, perfectamente, a las características del proyecto. No obstante, su uso requería del desarrollo, a mayores, de un adaptador específico para Ecto, el cual no existe, actualmente.

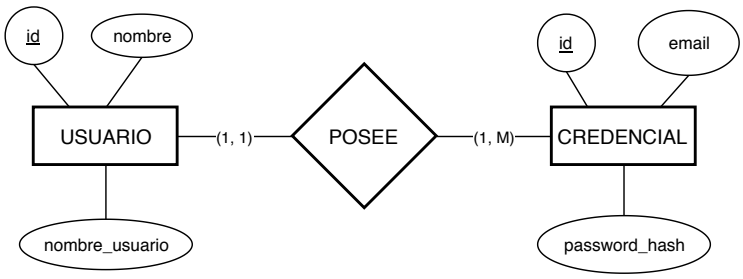

Figura 4.1: Diagrama entidad-relación para usuarios y credenciales

Esta opción, tras una intensiva evaluación, hubo de ser descartada debido a la mala relación complejidad vs. tiempo disponible.

4.1.1 Diagrama entidad-relación para usuarios y credenciales

Este diagrama E-R 4.1 describe la relación existente entre la entidad *usuario*, que representa a un usuario del sistema, y las credenciales que posee para identificarse en el mismo y ejercer sus derechos. Un usuario tendrá siempre un *identificador* numérico entero único, al igual que un *nombre de usuario* que también será único en el sistema. Como es frecuente, está permitido que múltiples usuarios tengan el mismo nombre, puesto que se trata de una situación típica y normal. Conviene destacar que no es lo mismo el *nombre de usuario* que el *nombre de un usuario*.

En este sistema, un usuario puede tener múltiples credenciales, pero una credencial solo puede pertenecer a un usuario. Actualmente, una *credencial de usuario* está compuesta por un correo electrónico, que es único en el sistema, y una contraseña asociada que se almacena cifrada. Estos dos últimos atributos son los que se utilizan para el inicio de sesión.

4.1.2 Diagrama entidad-relación para pizarras

El diagrama E-R 4.2 describe las entidades *usuario*, *pizarra* y las relaciones existentes entre ambas.

Como puntualización, hay que resaltar la diferencia entre una pizarra y una nota. Pizarras y notas pueden ser concebidas como un *buffer* compartido en el que los usuarios pueden escribir de forma concurrente. Pero, en este caso, las pizarras no poseen permisos de acceso. Esto es, cualquier colaborador invitado a contribuir en una pizarra, adquiere en ese mismo momento permiso de lectura y escritura en el *buffer* subyacente. En el caso de las notas, los permisos de lectura y escritura están diferenciados. Por lo tanto, un colaborador puede tener permiso de solo lectura sobre una nota o, en otro caso, permisos de lectura y escritura.

La entidad *pizarra* representa el concepto de una pizarra tradicional. Una pizarra tiene, siempre, un título (pueden existir varias pizarras con el mismo título). Cuando se crea una

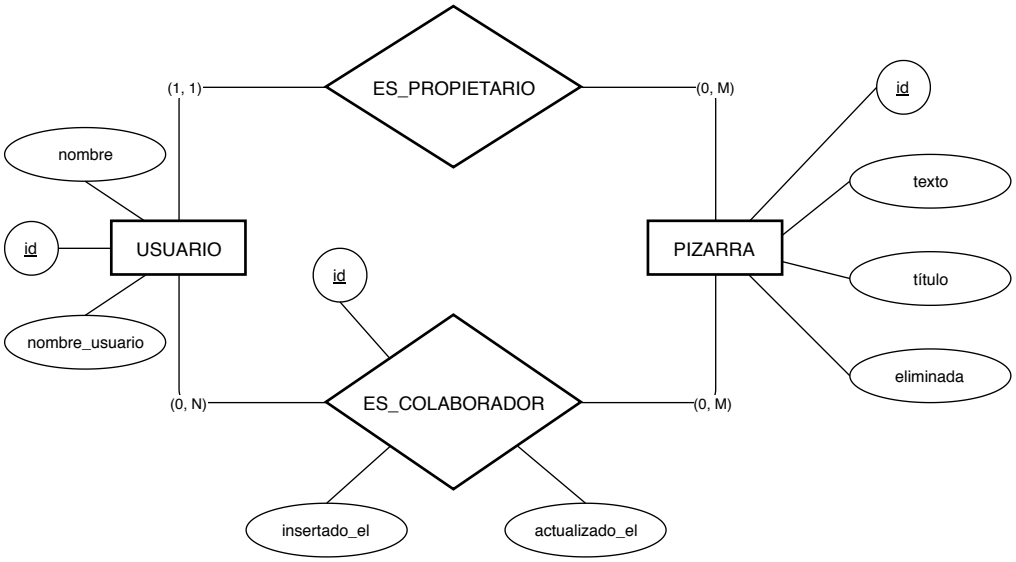

Figura 4.2: Diagrama entidad-relación para pizarras

pizarra, por comodidad, se le asigna un título compuesto, en parte, por un UUID. Toda pizarra, además del título, puede, opcionalmente, tener un *texto*. Las pizarras poseen un atributo *eliminada*, empleado para borrado lógico de la misma. Este campo se emplea para evitar el borrado físico de la pizarra cuando su creador la elimina y siguen existiendo colaboradores activos en la misma.

Como se muestra en el diagrama, un usuario puede tener un conjunto de cero o más pizarras del que es propietario. Análogamente, un usuario puede colaborar en cero o más pizarras de otros usuarios. La relación usuario-pizarra es una relación muchos a muchos, ya que un usuario puede colaborar en múltiples pizarras y una pizarra puede tener múltiples colaboradores. Eso sí, una pizarra siempre tendrá un único propietario, que además será el único que puede autorizar su eliminación definitiva. Aunque el borrado físico solo se llevará a cabo cuando no exista ningún usuario de la misma y el propietario haya manifestado su intención de eliminarla.

4.1.3 Diagrama entidad-relación para notas

El diagrama E-R para notas 4.3 describe las entidades *usuario*, *nota*, *etiqueta*, la relación entre usuarios y notas, y la relación entre notas y etiquetas.

Una nota tiene siempre un título y puede tener, opcionalmente, un cuerpo. Al igual que en el caso de las pizarras, visto anteriormente, la nota tiene un atributo *eliminada* para borrado lógico y su título también se genera, inicialmente, de forma automática, incluyendo un UUID en el mismo.

Una nota siempre tiene un propietario y puede tener cero o más colaboradores. Un usuario

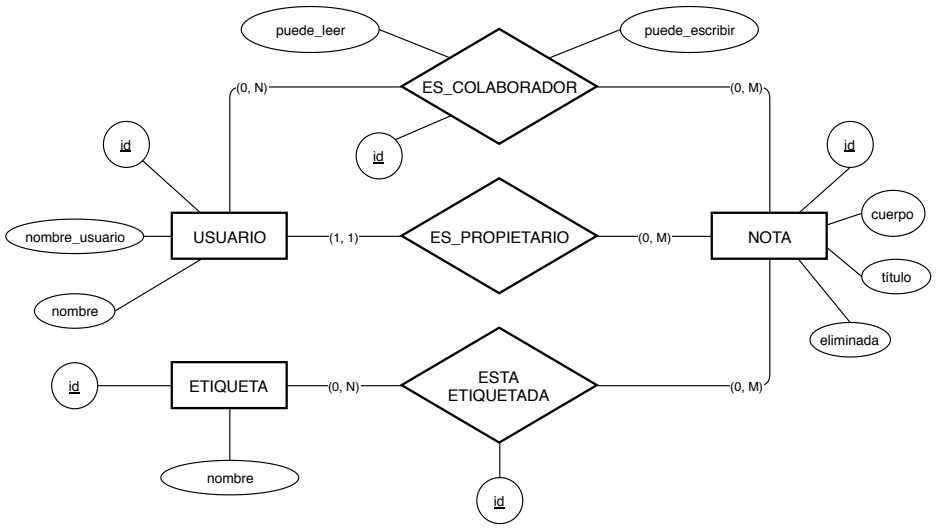

Figura 4.3: Diagrama entidad-relación para notas

que colabora en una nota puede tener permiso de solo lectura para la misma o bien permisos de lectura y escritura, como queda patente en la relación muchos a muchos *ES_COLABORADOR* que se muestra en el diagrama.

Las notas se pueden etiquetar o no. Esto es, una nota puede tener cero o más etiquetas y una etiqueta puede figurar en cero o más notas. En todo caso, cada etiqueta es única en el sistema y se emplea tanto para notas como para listas de tareas. Una etiqueta tiene un identificador entero único y, además, su nombre es único.

4.1.4 Diagrama entidad-relación para bloc de notas

En este diagrama 4.4 se muestran las relaciones de la entidad *bloc de notas* con las entidades *usuario*, *nota* y *etiqueta*.

La entidad *bloc de notas* tiene un atributo *id* que no es otra cosa que un número entero no negativo que permite identificarlo inequívocamente. Adicionalmente, un *bloc de notas* ha de tener, obligatoriamente, un nombre.

Como se puede apreciar en el diagrama, un usuario puede tener múltiples *blocs de notas*, pero un *bloc de notas* solo puede pertenecer a un *usuario*. Por otro lado, un *bloc de notas* puede tener asociadas cero o más notas y, a su vez, una determinada nota puede estar contenida, al mismo tiempo, en un único *bloc de notas*.

Al igual que pasa con entidades previamente descritas, un *bloc de notas* puede tener asociadas cero o más etiquetas, a la vez que una etiqueta puede describir a cero o más *blocs de notas*.

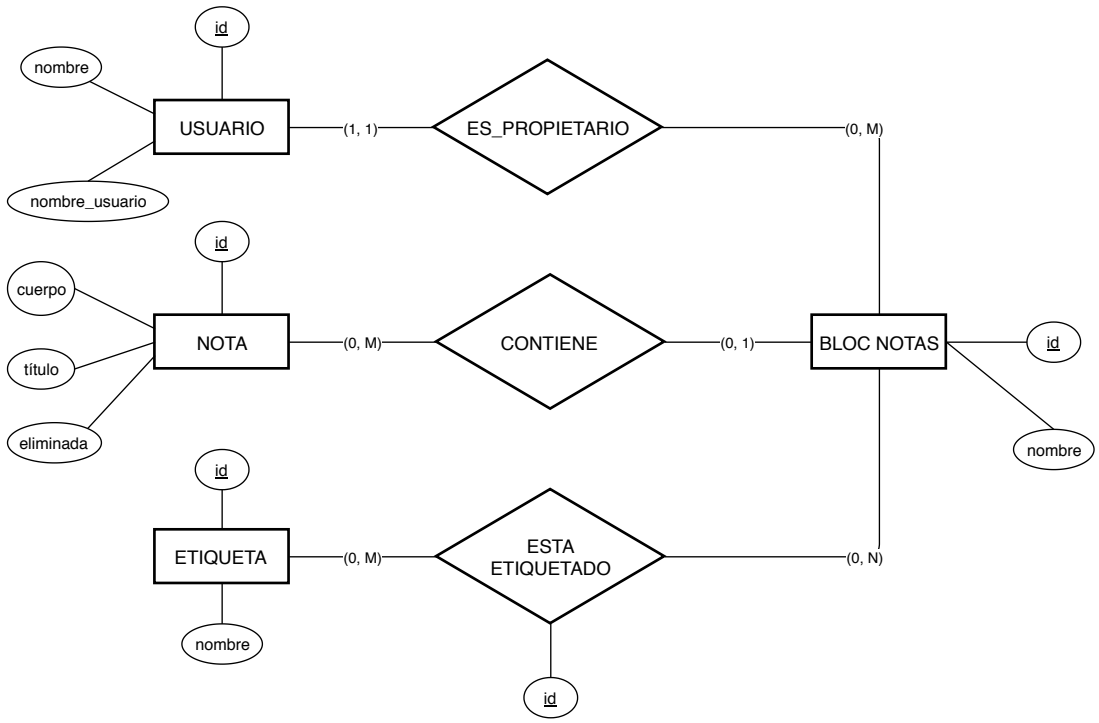

Figura 4.4: Diagrama entidad-relación para bloc de notas

4.1.5 Diagrama entidad-relación para lista de tareas

En este diagrama E-R 4.5 se describe la entidad *lista de tareas* o *TODO list* y las relaciones que presenta con otras entidades.

Una *lista de tareas* siempre tiene un propietario y, a su vez, un usuario puede tener cero o más *listas de tareas*. Adicionalmente, varios usuarios pueden colaborar en una misma *lista de tareas* y, para dichos usuarios, existen permisos independientes de lectura y escritura sobre la lista. Como es lógico, también está permitido que un mismo usuario colabore en múltiples *listas de tareas*, si lo desea.

Una *lista de tareas* tiene un identificador entero no negativo que sirve como clave primaria. Además, ha de tener, obligatoriamente, un título. Como parece razonable, cada *lista de tareas* no es más que un contenedor de *tareas* y, en este caso, dicho contenedor puede albergar cero o más tareas.

Una *tarea* tiene que estar, necesariamente, contenida en una *lista de tareas*. Esta entidad tiene que tener establecidos, obligatoriamente, los atributos *id*, *nombre*, *estado* y *prioridad*. El atributo *id* es autogenerado y sirve de clave primaria. El *estado* se establece a *iniciada*, por defecto. Y la prioridad, a no ser que se modifique explícitamente, es normal. Opcionalmente, una tarea puede tener *fechas de inicio/fin* y una *descripción*.

Por último, una *lista de tareas* puede tener múltiples *etiquetas* asociadas, que refuercen su

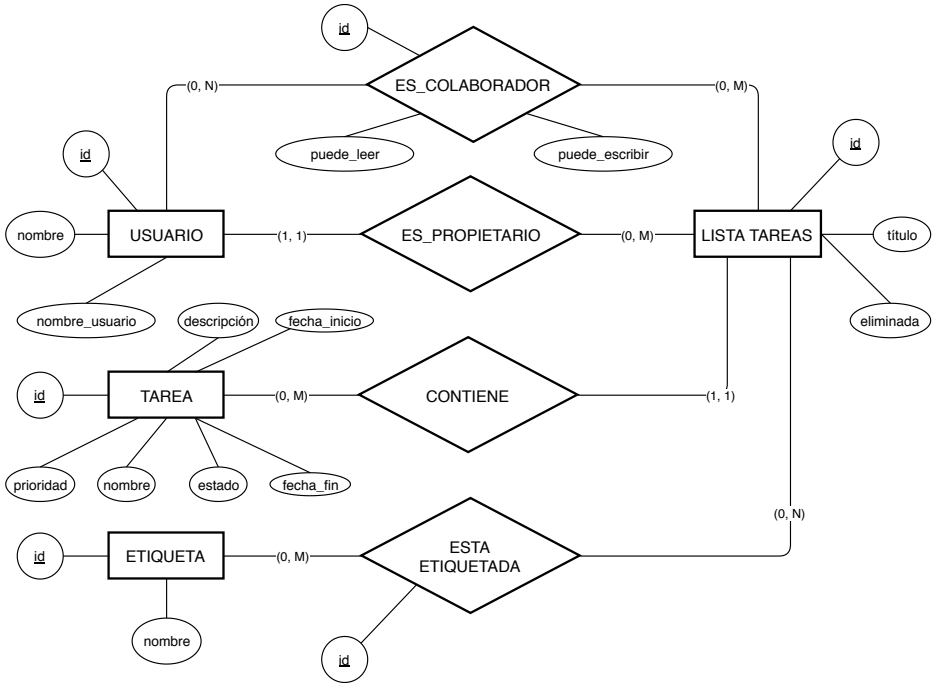

Figura 4.5: Diagrama entidad-relación para lista de tareas

semántica y faciliten la búsqueda e identificación de la misma.

4.2 Realización del sistema

En esta sección se describirá y explicará el diseño adoptado tanto en el lado cliente (*frontend*) como en el lado servidor (*backend*), así como las tecnologías escogidas para la implementación del mismo.

4.2.1 Diseño del *frontend*

GraphQL y su cliente

Como se ha mencionado, previamente, para la comunicación del *frontend* con el *backend* se emplea un cliente GraphQL. En este proyecto se ha optado por el cliente de código abierto Apollo. En el caso de Apollo [32], se ofrecen soluciones tanto para el cliente como para el servidor, pero en nuestro caso solo hacemos uso del cliente. La elección del mismo se debe, principalmente, a su simplicidad sobre su principal competidor, actualmente, Relay Modern [33]. Otros aspectos valorados, a la hora de la decisión, son su amplia documentación y ejemplos de uso, así como el soporte ofrecido por la comunidad de usuarios, aunque los ejemplos son ciertamente mejorables y la propia documentación presenta muchas lagunas que se deberían

pulir. La elección del cliente GraphQL se vio muy condicionada por las librerías Absinthe [34] (*The GraphQL Toolkit for Elixir*) disponibles para el *frontend*, que facilitan la configuración de la conexión al *backend*. Por tanto, hubo que buscar clientes compatibles con dichas librerías y, llegados a este punto, las únicas opciones serias y fiables parecen ser Apollo y Relay.

Una vez escogido Apollo como cliente GraphQL, es recomendable personalizarlo para lograr una mayor compatibilidad con la solución ejecutada en el *backend*. En el caso de la API HTTP, no hay nada que personalizar, de hecho, cualquier cliente HTTP estándar es una solución válida. Válido no significa recomendable, puesto que con el uso de un cliente HTTP "común", lo que estamos haciendo, en definitiva, es renunciar al potencial ofrecido por la caché suministrada por el cliente Apollo. No obstante, el problema crítico surge cuando se requiere el uso de la API basada en *Websockets* [26]. En este último caso, hemos de personalizar el cliente para que haga uso de los canales Phoenix (servidor) [35]. Para esto existe un cliente Javascript, denominado Phoenix.js [36], de los mismos creadores del *framework* Phoenix [37] B.11. El citado cliente está disponible desde cualquiera de los típicos gestores de paquetes, como pueden ser `npm` [38] o `yarn` [39]. Además de dicho cliente, se hace uso de los paquetes `@absinthe/socket` y `@absinthe/socket-apollo-link`. Ambos paquetes permiten gestionar la API GraphQL, basada en Websockets, de forma exitosa: `@absinthe/socket` nos dota con características muy interesantes, como es el caso de una API funcional inmutable, conexión perezosa o gestión de operaciones pendientes, si se produce una pérdida de conexión. Por tanto, el cliente Apollo, a bajo nivel, está delegando la conexión GraphQL, con el *backend*, al cliente Phoenix.js. Como detalle, conviene comentar que Phoenix.js no dispone de un método para reconectar al servidor. Después de analizar, detenidamente, el código fuente, se ha llegado a la conclusión de que la única forma existente, actualmente, para realizar una operación de reconexión es invocar, a muy bajo nivel, el método `close()`, sobre un objeto `conn` que representa la conexión. Cuando el cliente se da cuenta que la conexión se ha perdido, realiza intentos de reconexión empleando el *backoff* correspondiente. En lo referente a Phoenix.js, también es recomendable citar que si el servidor rechaza la conexión, los intentos de reconexión se realizan de forma indefinida. Aspecto que influye, en cierta forma, en la estrategia de autenticación/autorización adoptada en el *backend*.

En este proyecto, el cliente GraphQL se ha configurado para ejecutar las operaciones GraphQL B.7 de mutación y consulta empleando el protocolo HTTP, mientras que las operaciones GraphQL de suscripción se ejecutan mediante el protocolo WebSocket. Esta decisión ha sido adoptada para minimizar, en la medida de lo posible, el mantenimiento de estado en el servidor. Esto se debe a que para las operaciones HTTP no se mantiene el estado y, por tanto, minimizando el uso de *WebSocket*, logramos *mantener el estado* al mínimo. Se debe tener en cuenta que Apollo no crea la conexión *WebSocket* hasta que la utiliza por primera vez. Esto es, puede haber casos en los que el uso de la aplicación no requiera el uso de suscripciones, sino

tan solo de consultas y mutaciones, y, en estos casos, el servidor no mantendrá estado alguno referente al cliente. Con el fin de implementar esta partición, se ha hecho uso de la librería *@jumpn/utils-graphql*, la cual nos provee con la función `hasSubscription`, que combinada con la función `split` de Apollo, nos permite especificar el protocolo empleado para operaciones de suscripción y el protocolo utilizado para el resto de tipos de operaciones GraphQL, es decir, consultas y mutaciones. A modo informativo, hemos de decir que esta configuración puede visualizarse en el directorio `src/graphql-client` del repositorio del *frontend* [7].

El cliente Apollo emite preguntas y recibe respuestas de forma asíncrona. Es por ello, que sus creadores nos proporcionan una solución de caché. Existen dos usos, completamente diferentes, pero no mutuamente exclusivos, que se le pueden dar a la citada memoria. El primer uso, que es el que le damos en este proyecto, es para almacenamiento temporal de operaciones ejecutadas en remoto. Un segundo uso, más reciente y, actualmente, menos extendido, es el almacenamiento local del estado de la aplicación (*frontend*). En nuestro caso, para este último tema, hemos optado por una solución de uso común, como es Redux [40] B.6.6. Esta decisión nos permite cambiar el cliente GraphQL sin afectar de forma crítica a la arquitectura de la aplicación. Dicho esto, hemos de reconocer que el aspecto negativo es que se están introduciendo más dependencias en la aplicación. Hay que advertir que es imprescindible tener un especial cuidado con el funcionamiento de la caché, ya que un mínimo fallo en su uso puede conducir, directamente, a datos totalmente erróneos. La política de caché se puede gestionar haciendo uso del parámetro `fetchPolicy` del cliente. Por defecto, la política de caché está establecida a `cache-first`: esta política siempre intenta obtener los datos, para determinada consulta, primero desde la caché. Con esta política solo se recurrirá a la red en caso de que el resultado no esté cacheado. Evidentemente, esto minimiza el número de operaciones de red. Desafortunadamente, en nuestro caso ha sido imposible mantener dicha política, debido a un *bug* del cliente Apollo, referente a métodos que se deben ejecutar después de obtener el resultado y, sin embargo, no se ejecutan, generando comportamientos inesperados y complicados de depurar. Para corregir este mal funcionamiento del cliente, hemos optado por establecer la política de caché al valor `cache-and-network`: con esta política, si los datos para satisfacer la consulta se encuentran en la caché, se retornan de forma inmediata. No obstante, la consulta también será ejecutada en remoto, independientemente de si los datos están en la caché, con el fin de conservar la consistencia de los datos cacheados con el servidor. Por supuesto, esto último se logra a base de operaciones extra de red. No es la mejor solución posible, pero actualmente es la política que más se aproxima a la que debería de haber sido establecida por defecto, `cache-first`. Finalmente, existen otras tres políticas disponibles, pero que se adaptan peor a las necesidades de nuestra aplicación:

1. `network-only`: Nunca devuelve, inicialmente, los datos desde la caché. Siempre los solicita al servidor, a través de la red.

2. `cache-only`: Siempre retorna los datos desde la caché. Si no figuran en la misma, lanza un error.

3. `no-cache`: Evita el uso de la caché. Hace la solicitud al servidor y no escribe dato retornado alguno a la caché.

Para obtener los datos de las diferentes pizarras, notas y listas de tareas, que se muestran en los *dashboards* correspondientes, hubo que tomar una decisión ante dos posibles alternativas. Por un lado, cabía la opción de implementar las suscripciones GraphQL correspondientes, que nos notificaran de actualizaciones y creaciones. Por otro lado, el cliente GraphQL Apollo dispone de un mecanismo de *polling*, para las consultas. Esto es, ejecuta una consulta implementada cada cierto intervalo de tiempo, especificado por el implementador. En nuestro caso, hemos decidido emplear dicho *polling*, por resultar la opción más natural a la hora de mantener los *dashboards* actualizados.

El mecanismo de *polling* resulta de excelente utilidad, pero presenta un gran inconveniente: si la consulta renderiza directamente el resultado, se produce un parpadeo constante y molesto de la interfaz de usuario, al tiempo que se pierde el *offset* del *scroll* actual. Tras un exhaustivo estudio, empleando el método de prueba y error, al no encontrar información alguna al respecto, la solución hallada a ambos problemas (parpadeo y *scroll offset*) pasa por hacer circular los datos obtenidos por la consulta a través del *store* de Redux, previamente a su renderizado. En definitiva, estamos evitando que el componente Apollo renderice nada, es decir, lo estamos empleando únicamente como mecanismo para la obtención de datos. Vamos a tratar de explicar esto un poco más detalladamente: React-Apollo nos proporciona componentes React B.5 para ejecutar las operaciones GraphQL correspondientes. Esto es, el componente `<Query>` para consultas, el componente `<Mutation>` para mutaciones y el componente `<Subscription>` para suscripciones. Dichos componentes tienen dos partes diferenciadas. Una parte en la que se especifica lo que se debe renderizar mientras está cargando, si hay un error o si la operación es satisfactoria. Otra parte permite especificar *callbacks*, mediante *props*, que se ejecutarán cuando se complete la operación o haya un error. Por tanto, en el caso del *polling*, lo que nos interesa es la parte de las *callbacks*, pero no la del renderizado (será nuestro componente el que renderice, previa notificación de Redux). Para conseguir esto debemos hacer que la función de renderizado, del componente que implementa la operación GraphQL, retorne nulo.

Mecanismo de autenticación en el cliente

El *frontend*, durante el proceso de *login* satisfactorio de un usuario, recibe un *token* de autenticación (Phoenix Token). Por razones de eficiencia y rendimiento, dicho *token* se almacena en el almacenamiento local suministrado por el navegador web (`localStorage`), cuyas

funciones son siempre síncronas. Al mismo tiempo se conservará una copia del mismo en la fuente única de verdad en la que se mantiene el estado de la aplicación. El *token* es importante, para autenticación y autorización, y se mantiene en ambas ubicaciones por coherencia de operación. Para el navegador es más ventajoso albergarlo en el almacenamiento local (por ejemplo, si se usan varias pestañas), mientras que para los componentes React B.5, implementados por nosotros, tiene más sentido requerirlo desde la fuente única de verdad. El *token* recibido es completamente intranscendente para el cliente. Lo único que ha de hacer el cliente es recibirlo, almacenarlo y enviarlo en las peticiones HTTP (*stateless*), así como en el establecimiento de la conexión con el Websocket (*stateful*).

Este proceso se ilustra en el diagrama de secuencia 4.27, realizado para este fin.

Estado en el *frontend*

Para el mantenimiento del estado local del *frontend* se ha optado por la librería Redux [40] B.6.6. Esta solución no es más que una implementación simplificada del patrón FLUX B.6.5. A modo de anécdota, hemos de comentar que el cliente Apollo utiliza Redux internamente, por lo que es posible utilizar el mismo `store` del cliente Apollo, en lugar de crear otro independiente. Esta solución es justificada por muchos con la excusa de que menos código implica menos errores. Probablemente sea así, pero dicha solución no hace más que introducir una mayor rigidez en la aplicación referente a la tecnología. El uso de Redux nos hace considerar dos tipos de componentes en el *frontend*: componentes contenedor y componentes presentacionales. Los componentes contenedor son los que se ocupan de obtener los datos que, posteriormente, los componentes presentacionales mostrarán. Estrictamente hablando, en nuestra aplicación los componentes presentacionales no son "puramente" presentacionales, se trata de un híbrido entre componente contenedor y presentacional. Esto se debe a que, al hacer uso de la librería Apollo para React (React-Apollo [41]), los componentes que implementan las operaciones GraphQL, y en consecuencia obtienen datos, se ubican en el método `render` de los componentes presentacionales. En todo caso, el autor de este proyecto afirma que la división en componentes presentacionales y contenedor debe ser tomada de forma orientativa, en caso de resultar útil, pero en ningún caso como un mantra, ni siquiera muy en serio. El consejo del autor de este trabajo es: úsala si te resulta útil, en caso contrario recházala sin pensarlo demasiado, ya que en muchas ocasiones resulta demasiado estricta y forzada. Incluso grandes defensores de este enfoque, antaño, han terminado renegando del mismo. El propio Dan Abramov, voz ciertamente autorizada en este tema, actualizó su artículo [42], escribiendo cosas tales como:

> *"I wrote this article a long time ago and my views have since evolved. In particular, I don't suggest splitting your components like this anymore. If you find it natural in your codebase, this pattern can be handy. But I've seen it enforced without any*

> *necessity and with almost dogmatic fervor far too many times. The main reason I found it useful was because it let me separate complex stateful logic from other aspects of the component. Hooks let me do the same thing without an arbitrary division. This text is left intact for historical reasons but don't take it too seriously".*

Para que un componente React pueda usar Redux, para gestionar el estado de la aplicación, lo que se suele hacer es envolver dicho componente con otro componente que implementa las funciones `mapStateToProps` y `mapDispatchToProps`. La función `mapStateToProps` tiene como objetivo hacer los campos seleccionados, del estado de la aplicación, disponibles al componente envuelto, en sus parámetros de entrada (`props`. La función `mapDispatchToProps` hace disponibles *callbacks*, mediante `props`, al componente envuelto. Estas funciones, normalmente, disparan un creador de acción que se utiliza para solicitar cambios en el estado. Una vez creada la acción, la función *reducer*, teniendo en cuenta la propia acción y el estado actual, retornará el nuevo estado resultante de la ejecución de la acción. Durante este proyecto, se han utilizando, de forma frecuente, las herramientas de desarrollo para Redux [43] F, que permiten visualizar el estado del *store* inicialmente y después de cada acción. Estas herramientas son de gran ayuda para tareas de depuración y monitorización del estado.

A modo de reflexión personal, el autor de esta memoria cree que el desarrollo *frontend*, en la actualidad, tiende a abusar excesivamente de dependencias por comodidad. Después de leer una cantidad respetable de código de diversos desarrolladores en esta área, se ha llegado a la conclusión de que, muchas veces, es la inercia derivada del propio uso de gestores de paquetes la que lleva a introducir dependencias realmente innecesarias e injustificadas en la aplicación. Un caso muy fácil de observar es la inclusión en muchas aplicaciones de la librería Immutable.js [44], para gestionar el estado de Redux, cuando las versiones actuales de JavaScript suministran funciones igualmente simples con capacidades análogas, que permiten ahorrar en dependencias y simplificar el mantenimiento.

Es muy importante tener claro que, en todo momento, debe de existir una única fuente de verdad para la aplicación. Al disponer del *store* de Redux, por un lado, y de la caché del cliente GraphQL, por otro, alguien, por comodidad, podría caer en la tentación de tomar los datos de la fuente a la que más cómodamente pueda acceder en un determinado instante. ¡Camino equivocado!. Más pronto que tarde nos encontraremos con inconsistencias, a poco que la suerte cambie. Tampoco debemos olvidarnos del estado de los componentes React. Así, que en el peor de los casos tendríamos tres fuentes de verdad diferentes, situación que, mal gestionada, puede ser más peligrosa que pasear por la cima del Etna. El camino tomado en este proyecto es hacer circular los datos obtenidos por el cliente GraphQL (caché incluida, cuando proceda) hacia el *store* de Redux, siempre antes de hacer uso de los mismos. En lo que a los datos de salida (hacia el servidor) se refiere, circulan siempre hacia Redux y hacia el cliente GraphQL ...y, de ahí, al servidor. Finalmente, el soporte de estado proporcionado por React

para sus componentes, se usa para cosas muy menores, que no forman parte del estado de la aplicación, sino más bien describen una situación menor y puntual del componente, como puede ser el índice seleccionado de un *dropdown button*.

En definitiva, el procedimiento simplificado para modificar el estado del *frontend* es el siguiente:

- Se desean modificar los datos en el *store* de Redux.
- Se invoca un *creador de acción* que, como su propio nombre indica, es una función que creará una acción de determinado tipo y, opcionalmente, con el *payload* indicado. Dicha acción sirve para comunicar nuestra intención de modificar el estado.
- Se ejecuta el *reducer*. Esto es, una función que, tomando como entradas la acción y el estado actual, generará un nuevo estado. Conviene tener siempre presente que, en Redux, los objetos que representan el estado son inmutables. Por ello, el nuevo estado no es el estado actual modificado por la acción, es un nuevo objeto creado tomándolos como base.

La configuración de Redux, definición de acciones y creadores de las mismas, e implementación de funciones reductoras, están ubicados en el directorio `src/redux` del repositorio del *frontend* [7]. Este directorio contiene los subdirectorios `actions`, `constants`, `reducers` y `store`, donde figuran las partes correspondientes de Redux.

La interfaz gráfica de usuario

En lo referente a la interfaz de usuario, se ha optado por una solución que se sustenta sobre la librería React [45] B.5, escogida, fundamentalmente, por su enfoque claramente funcional, su excelente documentación y la poca pronunciada curva de aprendizaje que posee. Con esta decisión, queda claro que la interfaz de usuario estará construida empleando componentes reutilizables, los cuales serán implementados, dentro de módulos, como funciones o clases JavaScript, dependiendo de si carecen o no de estado. El resto de las tecnologías que la complementan son JavaScript [46] y CSS [47]. Finalmente, también se hace uso de la librería React-Bootstrap [48] B.4. Esta biblioteca se ocupa del diseño de la GUI para sitios web y aplicaciones. La elección de React-Bootstrap, y no de la biblioteca Bootstrap original, no es una decisión tomada al azar. Hay que tener en cuenta que la Bootstrap original emplea jQuery [49] y actúa directamente sobre el DOM [50]. Esto supone un gran problema cuando introducimos React en la ecuación. Hemos de ser conscientes que React utiliza un DOM Virtual [51], por lo que si tratamos de operar simultáneamente sobre el DOM real y el virtual, es muy probable que se generen graves problemas en el funcionamiento. Con esto no estamos afirmando que la citada solución sea incorrecta, pero sí es una solución complicada con una probabilidad muy

elevada de error. React-Bootstrap nos dota con los componentes de la librería original, pero completamente reescritos en forma de componentes React, al mismo tiempo que nos permite utilizar los estilos CSS originales. En este proyecto utilizamos la versión de React-Bootstrap para la versión vigente de Bootstrap, en este momento, que es la versión 4. A pesar de que la versión 4 de Bootstrap es estable, no sucede así con la homóloga de React-Bootstrap, que actualmente está en fase beta pública. Durante el desarrollo no ha surgido problema alguno con esta librería, no obstante, visto su estado, es evidente que existen *bugs* a resolver y que, en un momento dado, podrían causar algún comportamiento no deseado en la aplicación desarrollada.

Personalmente, preferimos referirnos a React [45] como una simple librería para *interfaces de usuario* en el *frontend*. Quizá la categoría de *framework* se le queda un poco grande, si uno se fija en las funcionalidades que proporciona. La idea en la que se basa esta librería es sumamente simple y, la misma, describe perfectamente el diseño de nuestro *frontend*: la interfaz de usuario no es más que una jerarquía de componentes, en la que los padres pasan información a sus hijos (mediante `props`) y los hijos pueden contactar con sus padres mediante eventos. Esta librería posee un lenguaje específico, pero prescindible, denominado JSX [52], que no es más que un guiño hacia HTML (escribir JavaScript como HTML). Se consideran dos clases de componentes: sin estado (implementados mediante funciones flecha de JavaScript) y con estado (implementados mediante clases JavaScript). La librería también suministra algunos trucos útiles de vez en cuando y en situaciones muy específicas, como es el caso de los contextos, que permiten compartir datos saltándose la jerarquía de componentes. Aunque, en el caso de este proyecto, este tipo de cosas han tratado de evitarse en la medida de lo posible.

En este trabajo se ha optado por emplear la herramienta oficial `create-react-app` [53] para la creación inicial del proyecto *frontend*. Esta herramienta no es más que un conjunto de librerías y *scripts* que permiten crear una aplicación, basada en React, de forma fácil y rápida, evitando así tediosos procesos de configuración e integración de dependencias. Al mismo tiempo, dicha herramienta suministra un servidor que posibilita la ejecución del *frontend* desde el instante inmediatamente posterior a su creación. Es muy importante resaltar que en el caso de ejecutar el *backend* en un servidor diferente del suministrado por `create-react-app`, surgen problemas asociados con la política de control de acceso CORS. Dichos problemas no fueron resueltos, de forma satisfactoria, siguiendo las instrucciones aportadas por los creadores de la herramienta, haciéndose necesaria la incorporación de una librería de gestión CORS [54, 55] en el *backend*.

Cuando hablamos de CORS, también conocido como Intercambio de Recursos de Origen Cruzado, nos estamos refiriendo a un mecanismo que permite que un agente de usuario obtenga permiso para acceder a recursos de un servidor en un origen distinto al que pertenece. Este mecanismo utiliza cabeceras HTTP específicas. Un agente crea una petición de origen

cruzado cuando solicita un recurso que tiene un origen diferente al de su propio origen. Los principales navegadores tienen restringidas las peticiones de origen cruzado iniciadas desde *scripts*, obviamente, por razones de seguridad, a excepción de que la respuesta del otro origen contenga las cabeceras CORS adecuadas. Es por esta última razón, por la que se hace uso en el *backend* de una librería CORS. En caso contrario, la comunicación entre cliente y servidor es inviable.

Una gran problemática encontrada, durante el desarrollo del *frontend*, fue el tratamiento del *offset* en las *textareas/textfields* que soportan edición compartida. Debemos tener en cuenta que cada vez que se actualiza el valor, en un campo o área de texto, el *offset* del mismo salta a la posición final. Esto debe ser considerado, tanto si la actualización procede de un usuario remoto o de un usuario local. Antes de actualizar cualquier campo o área de texto se debe guardar el *offset* actual del usuario y restablecerlo una vez actualizado el valor. No obstante, es importante tener presente que JavaScript es monohilo, guiado por eventos, y no dispone de semáforos o similares. En una primera aproximación, la actualización del *offset* fue implementada como parte de la operación GraphQL correspondiente. Concretamente, después de recibir los datos de la mutación (confirmación de la actualización) desde el servidor. Esto funcionaba bien en la mayoría de los casos, pero presentaba alteraciones insatisfactorias por manipulación inadecuada del *offset*. Particularmente, en el caso de pulsaciones de la tecla "borrar hacia atrás" continuadas, que generan una secuencia extremadamente rápida de eventos. En este caso, es posible que el manejador del evento sea invocado múltiples veces antes de que el resultado de la mutación correspondiente a la primera pulsación sea recibido. En este punto, el *offset* almacenado sería posterior, y no el correspondiente al resultado de la mutación. Como el valor del campo/área retornado por la mutación corresponde a un *offset* previo, el resultado es que hay caracteres que no se borran, es decir, el campo/área no es más que un contenedor de basura. Tras intensivas pruebas, se llegó a la conclusión de que dicho *offset* debe ser restablecido desde el manejador del evento asociado al cambio en el área o campo. Esto se consigue mediante el empleo de referencias `Refs` proporcionadas por React. Esto nos permite establecer el valor del campo y reestablecer el *offset*, accediendo directamente al nodo DOM correspondiente al área/campo, como paso previo a la operación GraphQL y posterior actualización del estado del *store* Redux. Con esto se consigue establecer el valor del campo/área de forma inmediatamente posterior a la ocurrencia del evento de teclado.

La aplicación *frontend* desarrollada es muy simple. Los diferentes elementos de la interfaz de usuario se disponen empleando CSS Flexbox [56], que no es más que un gestor de organización de elementos de la interfaz gráfica de usuario proporcionado por CSS [47] y que nos pone a disposición, para su uso, el *framework* Bootstrap [57] B.4. FlexBox es, excepcionalmente, simple de entender: los elementos de la interfaz de usuario se colocan siguiendo un flujo horizontal o vertical, punto final, ¡no hay más! …o seguimos el eje X o el eje Y. Adicio-

nalmente, Flexbox proporciona formas de especificar cómo se gestiona el espacio disponible lo que es, probablemente, su aspecto más complejo. En el caso de React-Bootstrap, el *framework* nos proporciona un componente `Container` que, conceptualmente, es una rejilla, con 12 columnas a lo ancho, y los diferentes componentes de la interfaz de usuario se disponen sobre dicha rejilla, empleando Flexbox. Bootstrap nos dota de formas de decirle a Flexbox: cuántas columnas de la rejilla debe ocupar un componente, de especificar si el componente debe colocarse al inicio, al final o en el centro (del eje que proceda), así como de establecer diversos valores para las propiedades de un componente (márgenes, relleno, etc). Finalmente, es importante comentar que Bootstrap considera 5 tamaños diferentes de dispositivo y hace posible especificar el número de columnas de la rejilla que debe ocupar un componente, en función de cada tamaño considerado. De esta forma, se facilita el soporte de la aplicación en la mayor parte de los dispositivos existentes en la actualidad.

Debido a la adopción de Bootstrap en este proyecto, la implementación de la interfaz de usuario ha consistido, fundamentalmente, en ubicar los contenedores pertinentes (puede haber contenedores embebidos en otros contenedores) y colocar los diversos componentes (botones, áreas de texto, etc.), proporcionados por Bootstrap, en dichos contenedores, siguiendo estrictamente las reglas de Flexbox.

Como es de esperar, Bootstrap nos proporciona componentes de construcción de la interfaz gráfica de usuario de uso común. Pero esto no implica que la construcción de la citada interfaz sea fácil. Es frecuente usar los componentes suministrados como base, pero personalizando los estilos CSS para adaptarlos a la aplicación en desarrollo y, esta labor, a menudo no resulta trivial y requiere de tiempo.

Finalmente, para conectar la interfaz de usuario a la lógica del *frontend*, lo que se hace es asociar manejadores de eventos a los componentes Bootstrap, por un lado. Por otro lado, también se asocian algunas propiedades de los componentes Bootstrap al estado de la aplicación, de forma que cuando se actualice el estado, el componente sea renderizado de nuevo, si procede. En este último caso, si asumimos el uso de Redux, para la gestión del estado, el procedimiento es el siguiente: cuando el estado cambia, React-Redux notifica a React del cambio. React es consciente de los componentes de la interfaz que hacen uso de la parte del estado que ha sido modificada, por lo que hace uso de su famoso algoritmo para determinar la parte de la interfaz gráfica de usuario que debe ser re-renderizada.

La aplicación dispone de un *dashboard* donde se muestran las notas, pizarras o listas de tareas del usuario con sesión iniciada, según proceda. Dependiendo del contenido del *dashboard*, una barra de menú tendrá controles para cambiar las entidades mostradas en el *dashboard* o crear una nueva. Esto es, crear una pizarra, una lista de tareas o una nota. Haciendo click en una entidad del *dashboard*, entraremos en el modo edición de dicha entidad, que nos permitirá editar sus propiedades (de forma individual o compartida), eliminar dicha entidad o añadir

usuarios colaboradores que contribuyan a la misma.

Cuando se crea una entidad (nota, pizarra o lista de tareas), no se accede (ni siquiera existe) a una pantalla para especificar los valores de las propiedades de la misma. Lo que se hace es crear la entidad con valores vacíos o por defecto (en los campos que proceda) y pasar inmediatamente al modo edición. Esta decisión nos permite tratar la creación, actualización y borrado de entidades en una sola pantalla de edición, lo que hace la aplicación extremadamente fácil de operar y simplifica bastante los flujos de trabajo.

Componentes del *frontend*

La aplicación se ha estructurado de un modo que resulta muy simple y fácil de entender. Básicamente, se han considerado dos tipos de componentes: contenedor y presentacionales no estrictos. Estos componentes se encuentran en los directorios `src/containers` y `src/-components` del repositorio del *frontend* [7], respectivamente.

En nuestra aplicación, los componentes contenedor, en el directorio `containers`, envuelven a sus homólogos presentacionales, en el directorio `components`. Concretamente, la función de estos contenedores es hacer disponibles el *router* y Redux a los componentes presentacionales. En el caso del cliente Apollo, en nuestro proyecto, es suficiente con importar los componentes correspondientes a las operaciones GraphQL desde la librería React-Apollo. Si se deseara, en algún momento, acceder al propio cliente, sí que sería conveniente habilitarlo desde un componente contenedor que envuelva al presentacional.

El renderizado de la aplicación comienza en el fichero `index.js`. En este fichero se crea la instancia del cliente GraphQL y el *store* de Redux que permanecerán asociados con nuestra aplicación. El componente raíz, con el cual comienza realmente la jerarquía de la aplicación, es `App`, ubicado en el fichero `App.js`. En este componente App es donde se cargan los componentes que implementan la cabecera y el pié de página, presentes en todo momento en la GUI, a la vez que se define el enrutamiento de la misma. A partir de aquí, dependiendo del enrutamiento definido y el estado actual, se cargarán los diversos componentes, según proceda. Por ejemplo, al iniciar la aplicación, como no hay sesión de usuario activa, se irá a la pantalla de inicio de sesión (componente `RunLogin`. Desde la pantalla de inicio de sesión, sin sesión iniciada, solo es posible ir a la pantalla de registro de usuarios (componente `SignUp`. Si se inicia sesión, se irá a la pantalla *home*, que tendrá una barra de menú principal (componente `MainBar`) y el *dashboard* de la entidad seleccionada, en cada momento, por el usuario (por ejemplo, componente `Boards` para pizarras). En definitiva, los componentes poseen nombres, que a juicio del autor, resultan sumamente descriptivos e intuitivos. Por otro lado, extraer la jerarquía de la aplicación, a partir del código fuente, resulta casi trivial.

Para concluir, hemos de decir que plasmar la estructura del *frontend* en un diagrama puede resultar algo denso. No obstante, tratándose de React, el disponer del código fuente hace que

la misma sea, relativamente, fácil y rápida de intuir.

4.2.2 Diseño e implementación del *backend*

El núcleo del *backend* es, sencillamente, un servidor web. En concreto, se trata del servidor web Cowboy [58]: un servidor HTTP pequeño, rápido y fiable para Erlang/OTP.

El servidor web Cowboy está escrito en Erlang y nos dota con una HTTP *fullstack*. Las características más destacables del mismo son su baja latencia y el poco consumo de memoria. Evidentemente, este servidor sería perfectamente viable reemplazarlo por cualquier otro, como sería el caso de Apache HTTP [59], pero echando un ojo al ecosistema por el que hemos optado, Cowboy parece una de las opciones más razonables, dada su elevada facilidad de integración en el mundo Erlang/Elixir.

El *software* Cowboy nos dota de los dos puntos de entrada a la API que nuestro servicio oferta. El primer punto de entrada es un servidor HTTP, que recibe las peticiones, basadas en ese protocolo estándar, desde el cliente. El segundo punto de entrada es el servicio WebSocket, que permite que el cliente establezca conexiones, basadas en el protocolo WebSocket estándar, con Cowboy. Conviene aclarar que HTTP es un protocolo sin estado, mientras que WebSocket es un protocolo con estado, por lo que el sistema que hemos diseñado es un sistema híbrido, en este aspecto. En todo caso, todo documento GraphQL recibido empleando tanto el protocolo HTTP, como por un canal WebSocket, será entregado al *Phoenix Framework* [37] B.11. Todas las operaciones GraphQL implementadas son susceptibles de ser ejecutadas indistintamente por HTTP o WebSocket. En una primera aproximación, se consideró la excepción de la mutación de *login*, que por motivos técnicos solo podría ser realizada empleando el protocolo HTTP: para conectar al socket es necesario un *token* de autenticación válido y, sin él, no es posible conectar al *socket* como un nuevo usuario, ya que sin dicho *token* el *socket* rechazará la conexión, por seguridad. En caso contrario, el sistema sería susceptible de asemejarse a un colador, en lo que a seguridad se refiere. Por tanto, en esta primera aproximación, la alternativa que se consideró más razonable fue la de forzar la obtención del *token* de autenticación mediante una mutación GraphQL de *login*, ejecutada bajo HTTP, como paso previo al acceso por WebSocket. Esta aproximación tiene un gran inconveniente: cuando el cliente ejecuta un *logout* y el *socket* del servidor se cierra, el cliente *WebSocket* Phoenix.js queda intentando reconectar al servidor indefinidamente. Este inconveniente no tiene fácil solución y el servidor queda sometido, indefinidamente, a un bombardeo de intentos de reconexión fallidos. Visto esto, después de una exhaustiva evaluación, se decidió proceder con una segunda aproximación. Esta vez, probamos a mantener la conexión *WebSocket* activa, cuando el usuario cierra la sesión, pero actualizando los datos de conexión de forma que se elimine el token de acceso (forzando una reconexión sin *token de acceso*). De esta forma, cuando un usuario ejecuta un *logout*, la conexión cliente-servidor permanecerá activa, pero solo se podrán realizar opera-

ciones que no requieren de autenticación, esto es, mutaciones de registro de usuarios y *login*. En principio, esta solución no compromete en modo alguno la seguridad, a la vez que nos permite salvar las limitaciones del cliente Phoenix.js en este aspecto.

Llegados a este punto, es obligatorio mencionar que: la única forma existente, actualmente, de actualizar los datos de conexión del canal *WebSocket*, es realizar un *hack*, a muy bajo nivel, en el cliente Phoenix.js, que "corta" la conexión con el servidor. Cuando Phoenix.js se da cuenta que la conexión se ha perdido, intentará reconectar automáticamente (con los nuevos datos). La parte más importante de todo esto es que se trata de una solución nada ortodoxa que, muy probablemente, resulte inestable.

Endpoint y Router

Cuando Cowboy recibe peticiones del cliente las encamina, directamente, al *Phoenix Framework* [37] B.11, que es el corazón del *backend*. En concreto, la función `connect` del módulo `ErlnoteWeb.UserSocket` es la que gestiona las conexiones del cliente GraphQL, basadas en *WebSocket*, al servidor. Una vez aceptada la conexión, los datos que se consideren importantes de la misma, se pueden almacenar en un `Map` (estructura de datos clave-valor) , denominada `Context`, que es accesible durante la vida de la conexión *WebSocket*. Como se puede imaginar, existe también una `Context` para las peticiones HTTP, con la diferencia, al ser *stateless*, que la vida de estos datos comienza cuando se recibe la petición y expira cuando se genera la respuesta. Esta estructura es de sumo valor, para almacenar datos tan importantes como puede ser el identificador del usuario con sesión activa. La cantidad de datos que se pueden almacenar en una estructura `Context` tiene como único límite la cantidad de memoria disponible. De todas formas, estas estructuras suelen albergar muy pocos datos pero muy valiosos.

Obviando el servidor web subyacente, la vida de nuestra aplicación, en el *backend*, comienza en el *endpoint* del *Phoenix Framework*. El *endpoint* constituye la frontera en la cual comienzan todas las solicitudes a nuestra aplicación web, al mismo tiempo que conforma la interfaz que nuestra aplicación suministra al servidor web subyacente. En *endpoint* es vital, puesto que forma parte de un árbol de supervisión, que hace nuestra aplicación especialmente buena en lo que a tolerancia a fallos se refiere.

Como hemos comentado previamente en esta memoria, el *Phoenix Framework* puede ser concebido como un *pipeline* de *Plugs*. Un *Plug* es, sencillamente, una función que produce y consume una estructura de datos denominada `Plug.Conn`. Dicha estructura representa el universo completo de una solicitud del cliente. Pues bien, el *endpoint* es, precisamente, el punto de inicio de dicho *pipeline* de *Plugs*. Esto nos lleva a concluir que, como nuestra aplicación es una cadena de *Plugs*, y un *Plug* no es otra cosa que una función, luego nuestra aplicación web es, ciertamente, una gran función, resultado de la composición de funciones más pequeñas, que se agrupan en módulos.

El módulo correspondiente al *endpoint* es, en nuestro caso, el módulo `ErlnoteWeb.Endpoint`. En la aplicación, este módulo invoca dos *Plugs* fundamentales, uno es `Corsica`, encargado de la gestión de la política de seguridad de recursos cruzados CORS, y el otro es `Router`, encargado del enrutamiento en el *backend*. En este módulo, también se especifica la URL del socket, el módulo que gestiona la conexión al mismo y el protocolo a emplear: *WebSocket* o *Long Poll*.

Las peticiones HTTP, que pasen el *endpoint*, irán a parar directamente al *router*, que no es más que otra sucesión de *Plugs* y está representado por el módulo `ErlnoteWeb.Router`. Como es obvio, las peticiones por *WebSocket* omitiran el *router*. Por ello, en el caso de las operaciones por *WebSocket*, la estructura `Context` será configurada en el módulo `ErlnoteWeb.UserSocket`, que recordamos, gestionaba dichas conexiones. Análogamente, en el caso de las peticiones HTTP, la estructura `Context` será establecida en `ErlnoteWeb.Router`.

Nuestro módulo `ErlnoteWeb.Router` es muy sencillo, ya que, al utilizar GraphQL, todas las peticiones contra la API han de ir dirigidas a una única URL. En nuestro caso: `http://localhost:4000/api` para HTTP y `ws://localhost:4000/socket` para *WebSocket*. En resumen, lo que hace nuestro *router* es aceptar las peticiones HTTP enviadas a la URL indicada anteriormente (contenido codificado en JSON), extraer la información contenida en el *token* de autenticación de la cabecera HTTP (el identificador del usuario activo), establecer el `Context` con dicha información y pasarle el documento GraphQL al *Plug* Absinthe, que implementa el servidor GraphQL. Al invocar el *Plug* Absinthe, en el *router*, se debe especificar el módulo que implementa el esquema GraphQL, en nuestro caso `ErlnoteWeb.Schema`.

Para configurar el `Context`, en nuestro *router* invocamos el *Plug* `ErlnoteWeb.Context`. Este *Plug* lo hemos implementado para que extraiga el *token* de autorización desde la cabecera `authorization` de la petición HTTP. Posteriormente, utiliza otro módulo desarrollado, `ErlnoteWeb.Authentication`, para verificar la validez del *token*. Finalmente, ubica la información contenida en el *token* dentro de la estructura `Context`, para que esté disponible para el servidor GraphQL Absinthe. Los *tokens* utilizados, en esta implementación, son proporcionados por el módulo `Phoenix.Token` y tienen una validez de 24 horas, período tras el cual el usuario debe volver a identificarse ante el sistema.

El módulo `Phoenix.Token` nos dota de funciones para generar y verificar *Bearer Tokens*, adecuados para autenticación. Estos *tokens* están firmados, para evitar que sean manipulados, pero no encriptados. Esto es, son adecuados para almacenar información de identificación, tal como IDs de usuario, pero no información confidencial.

Absinthe y el servidor GraphQL

Una vez que el *router* ha encaminado el documento GraphQL, con la solicitud del cliente, al `Plug Absinthe`, comienza un proceso, prácticamente, común a todo servidor GraphQL.

El servidor recurrirá al esquema GraphQL para comprobar que el documento, enviado por el cliente (mutación, consulta, subscripción), es un documento válido. En nuestra aplicación, el esquema GraphQL se encuentra soportado por el módulo `ErlnoteWeb.Schema`. Este módulo utiliza diversos tipos (de entrada, salida o entrada/salida) que hemos definido para implementar el esquema. Estos tipos son albergados por los módulos:

- `ErlnoteWeb.Schema.AccountsTypes`: Tipos usados para implementar las operaciones GraphQL relativas a cuentas de usuario.
- `ErlnoteWeb.Schema.BoardsTypes`: Tipos que se emplean en la implementación de operaciones GraphQL referentes a pizarras.
- `ErlnoteWeb.Schema.NotepadsTypes`: Tipos utilizados en la implementación de operaciones GraphQL sobre blocs de notas.
- `ErlnoteWeb.Schema.NotesTypes`: Tipos usados para implementar las operaciones GraphQL relativas a notas.
- `ErlnoteWeb.Schema.TagsTypes`: Tipos que se emplean en la implementación de operaciones GraphQL con etiquetas para notas, blocs de notas y listas de tareas.
- `ErlnoteWeb.Schema.TasklistsTypes`: Tipos utilizados en la implementación de operaciones GraphQL sobre listas de tareas.
- `ErlnoteWeb.Schema.TasksTypes`: Tipos usados para implementar las operaciones GraphQL relativas a tareas de una lista de tareas.

Además de en el interior de los módulos anteriores, es posible definir los tipos más generales dentro del propio módulo `ErlnoteWeb.Schema`, como así se ha hecho. El módulo del esquema GraphQL se divide en tres bloques muy importantes:

- `do query`: En este bloque se definen las diferentes consultas implementadas, empleando la macro `field`.
- `do mutation`: En este bloque se definen las diferentes mutaciones (creación, actualización o borrado) implementadas, empleando la macro `field`.
- `do subscription`: En este bloque se definen las diferentes suscripciones implementadas, empleando la macro `field`.

En el módulo `ErlnoteWeb.Schema` también es posible definir una función, por casos, denominada `middleware`. Esta función nos permite introducir diversos *middlewares* empleados durante las consultas, mutaciones o suscripciones. En nuestro caso concreto, empleamos esta función para proporcionar un *middleware* que formatea los errores de modo que sean más fácilmente comprensibles por el ser humano. El módulo que implementa esta función se denomina `ErlnoteWeb.Schema.Middleware.ChangesetErrors`.

Además de la función, por casos, `middleware`, existe una macro con el mismo nombre que también nos permite ejecutar *middlewares* donde proceda. En este proyecto, y más concretamente en el esquema GraphQL, se hace uso de dicha macro para ejecutar un *middleware* que comprueba si hay un usuario autenticado en el sistema y si se trata de un usuario válido. Este *middleware* está implementado en el módulo `ErlnoteWeb.Schema.Middleware.Authorize` y es ejecutado para toda operación no pública, esto es, que requiera tener sesión iniciada.

En el esquema se implementan las operaciones GraphQL que deseemos. Una operación GraphQL, en general, se define de la siguiente forma:

- La macro `field`, seguida por el nombre de la operación y el tipo retornado por la misma. Se puede retornar una lista especificando `list_of(tipo)`.
- Los argumentos de entrada. Cada argumento se define con la macro `arg`, seguida del nombre del argumento, el tipo del mismo y restricciones (Ej. non_null).
- Una función de resolución, especificada mediante la macro `resolve`. La función de resolución es la encargada de ejecutar, propiamente, la operación. Aquí es donde se implementa la lógica de la operación y se genera la respuesta.
- Los *middlewares* necesarios, se pueden introducir con la macro `middleware`, antes o después de la función de resolución, según proceda.

En este proyecto, las funciones de resolución se encuentran ubicadas en módulos independientes del esquema, y son invocadas desde este último. Los módulos *resolver* son los siguientes:

- `ErlnoteWeb.Resolvers.Accounts`: Funciones de resolución para operaciones GraphQL relativas a cuentas de usuario.
- `ErlnoteWeb.Resolvers.Boards`: Funciones de resolución para operaciones GraphQL relativas a pizarras.
- `ErlnoteWeb.Resolvers.Notepads`: Funciones de resolución para operaciones GraphQL relativas a blocs de notas.

- `ErlnoteWeb.Resolvers.Notes`: Funciones de resolución para operaciones GraphQL relativas a notas.
- `ErlnoteWeb.Resolvers.Tasklists`: Funciones de resolución para las operaciones GraphQL relativas a listas de tareas.
- `ErlnoteWeb.Resolvers.Tasks`: Funciones de resolución para operaciones GraphQL relativas a tareas de una lista de tareas.

La función de resolución más empleada es la de aridad 3, esto es `resolve/3`. El primer argumento es la entidad padre (por ejemplo, la función de resolución en el campo ID de un usuario, tendría como entidad padre al usuario), el segundo son los argumentos de entrada y el tercero una estructura de resolución, que alberga información sobre la resolución, tal como el contexto.

Una vez alcanzado el *resolver*, el resto de trabajo será delegado a los módulos que están en la frontera del modelo de negocio, denominados, en Phoenix, *Context modules*. Estos módulos nos permiten interaccionar con el modelo de negocio de nuestra aplicación sin necesidad de conocer nada relativo a su implementación.

Contextos de la aplicación

Cuando se usa el *framework* Phoenix, es recomendable seguir el patrón de contextos recomendado por sus creadores. Este patrón consiste en dividir el modelo de negocio en contextos o áreas funcionales.

Un contexto se ubica en un subdirectorio del directorio `lib/erlnote`, cuyo nombre se corresponde con el del contexto definido. En nuestro proyecto hemos definido cinco contextos:

1. `accounts`: Funcionalidades relacionadas con cuentas de usuario.
2. `boards`: Funcionalidades relacionadas con pizarras.
3. `notes`: Funcionalidades relacionadas con notas.
4. `tags`: Funcionalidades relacionadas con etiquetas.
5. `tasks`: Funcionalidades relacionadas con tareas.

El directorio de un contexto contiene un módulo, con el mismo nombre del directorio, denominado *módulo contexto* o *context module*. Este módulo tiene como objetivo servir de frontera entre la parte del modelo de negocio que representa el contexto y el exterior. Las funciones implementadas en este módulo invocarán, directamente, a las funciones ubicadas en los módulos de negocio. Al mismo tiempo, las funciones contenidas en los módulos contexto serán invocadas por los *resolvers* GraphQL, para satisfacer las operaciones solicitadas.

El directorio de un contexto no contiene solamente al módulo contexto, sino que, además, alberga a los módulos de negocio asociados a dicho contexto. En este proyecto, los directorios contexto contendrán el módulo contexto correspondiente y, concretamente, los módulos de negocio que se encargan de operar con los datos e interaccionar con el sistema de gestión de bases de datos. Estos últimos módulos son, fundamentalmente, los módulos que hacen uso de la librería de persistencia Ecto [29] B.10.

Migraciones y el modelo de datos

En este proyecto no hemos interaccionado directamente con el sistema de gestión de bases de datos, prácticamente, en ningún momento. Esto se debe a que la librería de persistencia oficial de Elixir, conocida como Ecto [29], nos permite manipular la base de datos sin conocer nada sobre ella. Lo único requerido es un usuario con los permisos suficientes.

Cuando creamos una aplicación con el *framework* Phoenix, el propio *framework* crea una base de datos, en el sistema de gestión, para la aplicación, junto con un repositorio que servirá para acceder a la misma. Este repositorio está representado por el módulo `lib/erlnote/-repo.ex`. En este módulo se puede observar, sin género de duda, el nombre del repositorio (`Erlnote.Repo`), el nombre de la aplicación asociada al repositorio (argumento `otp_app`), y el adaptador usado para la conexión al sistema de gestión de bases de datos, en nuestro caso `Ecto.Adapters.Postgres`, para acceder a PostgreSQL [30].

Para generar la estructura de la base de datos, Ecto emplea lo que se conoce como *migraciones*. Las migraciones son simples ficheros, ubicados en el directorio `priv/repo/migrations`, que contienen funciones. Dichas funciones son las que permiten crear tablas de la base de datos, eliminarlas, añadir/eliminar/modificar columnas de tablas, gestionar índices para acelerar el acceso a los datos y gestionar restricciones sobre los datos. El nombre del fichero que alberga la migración contiene una marca de tiempo y una descripción de lo que la migración hace. Cuando el usuario descarga una versión de la aplicación, del sistema de control de versiones, al inicializarla, se ejecutan las migraciones, para adecuar la estructura de la base de datos a los requerimientos actuales. Por razones de eficiencia, se mantiene un histórico de las migraciones ejecutadas, de ahí la marca de tiempo en el nombre de cada migración. Por tanto, en cada versión nueva instalada, solo se ejecutarán las migraciones que no han sido todavía llevadas a cabo, en lugar de ejecutar, innecesariamente, todo el conjunto de migraciones implementadas.

Una vez llegados a este punto, la base de datos estará preparada para atender a nuestra aplicación.

Esquemas Ecto

Un esquema Ecto [H] está vinculado, directamente, a una tabla de la base de datos del proyecto. La función del esquema es definir una estructura a la que leer los datos desde la base de datos, para poder operar con ellos usando el lenguaje Elixir.

Un esquema asocia un campo de la estructura a una columna de la tabla (de la base de datos) usando la macro `field`. El esquema también contiene campos, definidos con las macros `has_one`, `has_many`, `belongs_to` y `many_to_many`, que representan relaciones con otros esquemas. Esto no son sino las relaciones que emergen del modelo de datos subyacente. En conclusión, el esquema tiene campos de tipos "simples" y de tipo "otro esquema" (relaciones).

De todo esto se deduce que los datos se leen desde la base de datos y se colocan en una *struct* de Elixir, el esquema. También se concluye que: los esquemas se relacionan de la misma forma que sus tablas homólogas.

En este punto, es obligatorio comentar que los campos del esquema, que representan relaciones, no se cargan *nunca* automáticamente, al leer los datos. Ecto no implementa, en ninguna de sus formas, el concepto de *carga perezosa*. Esto se hace con la intención de evitar operaciones no deseadas, que se realizan en segundo plano, que las librerías que sí implementan dicho concepto ejecutan. Ecto trata de incrementar la eficiencia haciendo, exactamente, lo que el desarrollador le comunica, explícitamente, que debe realizar. La consecuencia de esto es que la librería es un poco menos cómoda de utilizar, pero lejos de suponer una complejidad insalvable.

En nuestro proyecto, los módulos que implementan los esquemas Ecto, residen en el directorio del contexto correspondiente. Por ejemplo, en el contexto `Boards` tenemos el esquema Ecto `Board`. Por tanto, el esquema reside en el directorio `boards`, en el módulo `board.ex`, y es accesible con la estructura `%Board{}`.

Changesets

Los *changesets* [H] son funciones, que normalmente residen en el mismo fichero que el esquema Ecto al que están asociadas, y que se ocupan de que solo lleguen los datos correctos a la base de datos.

Principalmente, un *changeset* se ocupa de la conversión entre tipos de datos, de las validaciones y de las restricciones.

La función `cast` realiza, casi siempre con éxito, la conversión de datos externos de un tipo al tipo del campo correspondiente del esquema.

Después de la conversión de tipos, se ejecutan las validaciones. Las validaciones son restricciones sobre los datos pero que no están asociadas a la base de datos. Por ejemplo, se puede validar que ciertos campos son obligatorios, de una longitud determinada o siguen un

formato deseado. También es posible definir validaciones personalizadas, adicionalmente a las funciones de validación proporcionadas por defecto.

Después de las validaciones se ejecutan las restricciones, más conocidas como *constraints*. Estas son restricciones asociadas a la base de datos. Por ejemplo, se puede requerir que el valor en una columna de una tabla sea único. Al igual que en las validaciones, se pueden definir restricciones personalizadas, como complemento a las suministradas por defecto (*unique*, *primary_key*, etc.).

Para concluir, hemos de comentar que: si una validación falla, las restricciones ya no se llegan a evaluar.

Supervisores

La aplicación desarrollada implementa la tolerancia a fallos mediante un árbol de supervisión. Este árbol de supervisión puede observarse en el fichero `lib/erlnote/application.ex`.

Si uno se analiza, detenidamente, este fichero, se puede percatar de la existencia de un supervisor general, para la aplicación. Adicionalmente, existen tres subárboles de supervisión para puntos estratégicos de la aplicación:

- Subárbol de supervisión para el repositorio que maneja la base de datos.
- Subárbol de supervisión para el *endpoint*, que constituye el punto de entrada a la aplicación.
- Subárbol de supervisión para las suscripciones Absinthe, que se corresponden con la parte asociada a *Websockets* y *Channels*.

Estos supervisores son los recomendados para esta aplicación y cubren, de forma notable, los posibles fallos de la misma.

La estrategia de supervisión seguida es *one_for_one*. Esto es, si un proceso hijo termina, solo ese proceso es reiniciado.

4.2.3 Diagramas de casos de uso del *backend*

En este apartado se incluyen los diagramas de casos de uso correspondientes con las funcionalidades desarrolladas para la parte servidora de la aplicación web (*backend*).

En este caso se han elaborado cinco diagramas diferentes, correspondiendo cada uno de ellos a las entidades con las que opera el sistema: cuentas de usuario 4.6, pizarras 4.7, listas de tareas 4.8, notas 4.9 y blocs de notas 4.10.

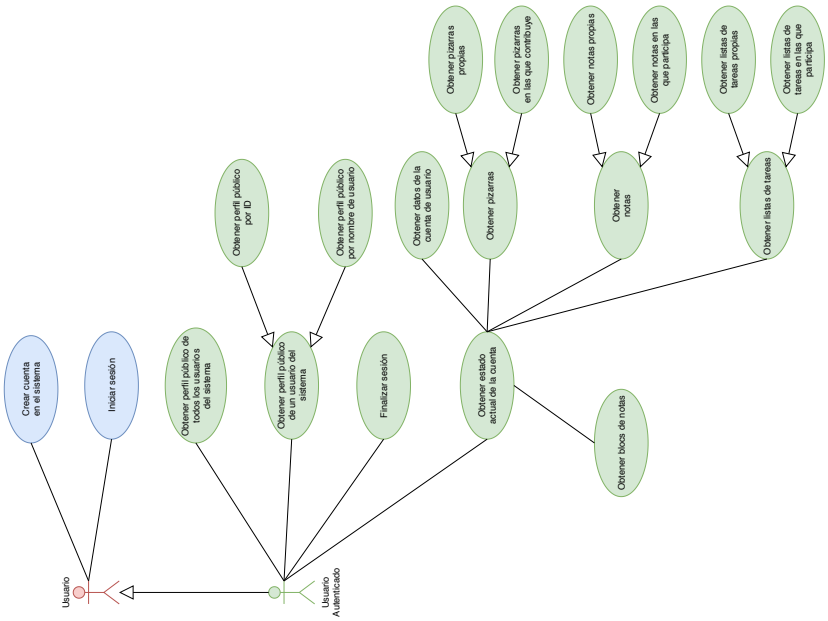

Figura 4.6: Backend: Casos de uso de cuentas de usuario

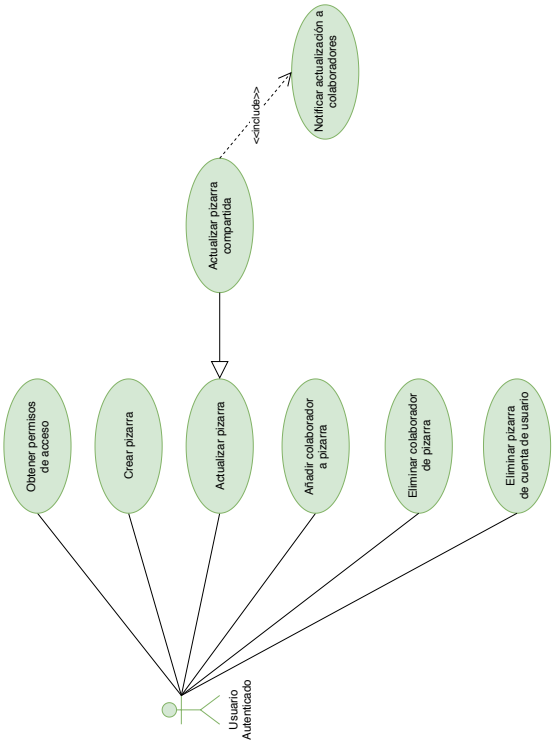

Figura 4.7: Backend: Casos de uso de pizarras

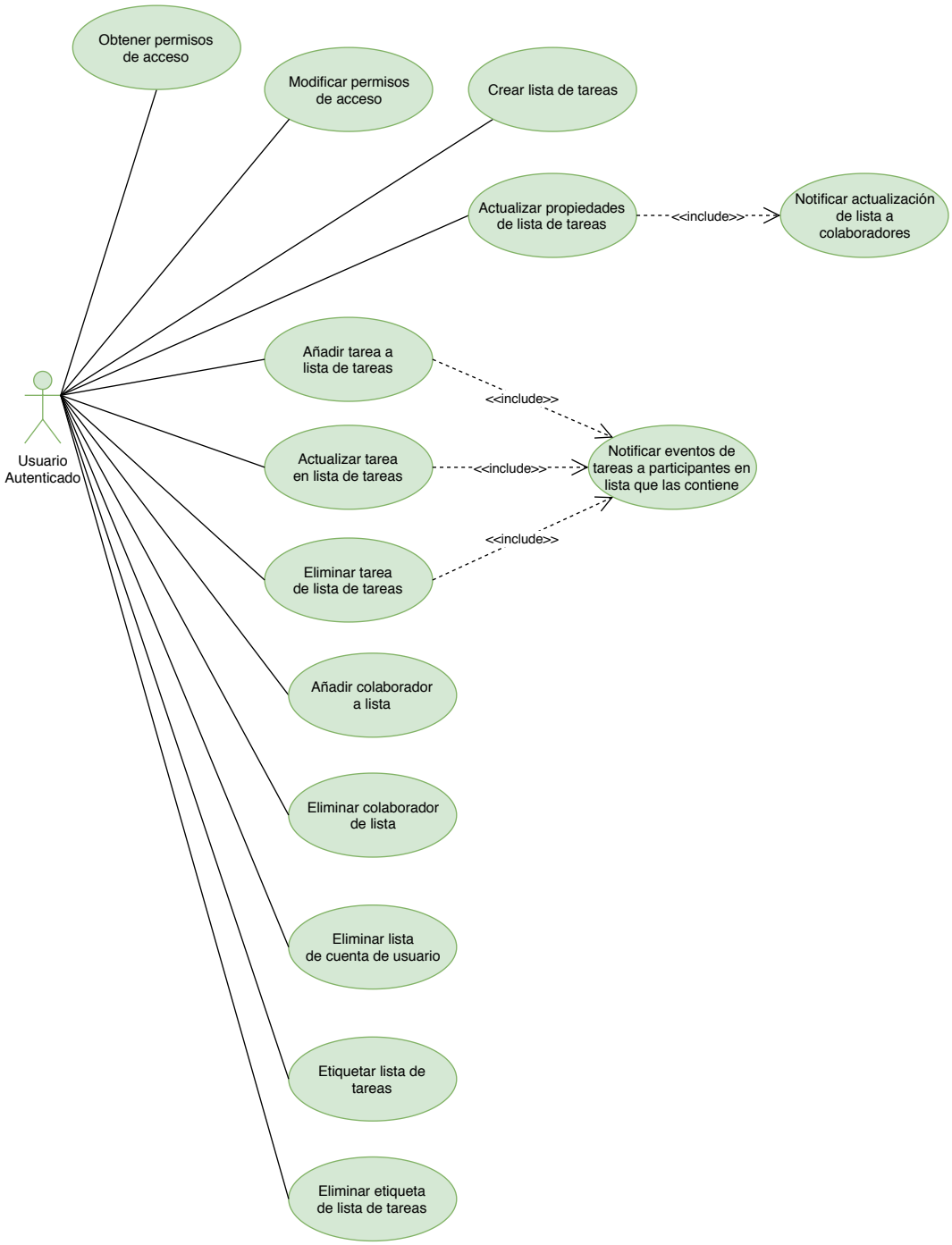

Figura 4.8: Backend: Casos de uso de listas de tareas

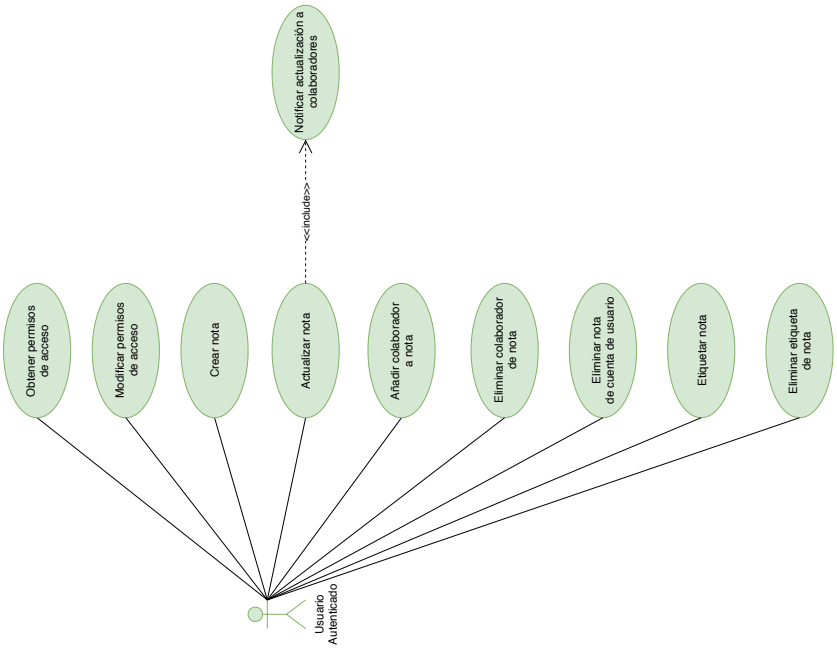

Figura 4.9: Backend: Casos de uso de notas

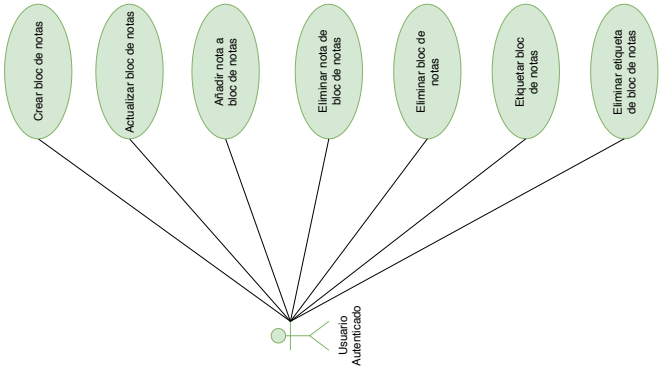

Figura 4.10: Backend: Casos de uso de bloc de notas

Figura 4.11: Frontend: Casos de uso del *frontend*

4.2.4 Diagramas de casos de uso del *frontend*

En este apartado se incluyen los diagramas de casos de uso [4.11] [4.12][4.13][4.14] correspondientes a las funcionalidades desarrolladas para la parte cliente de la aplicación web (*frontend*).

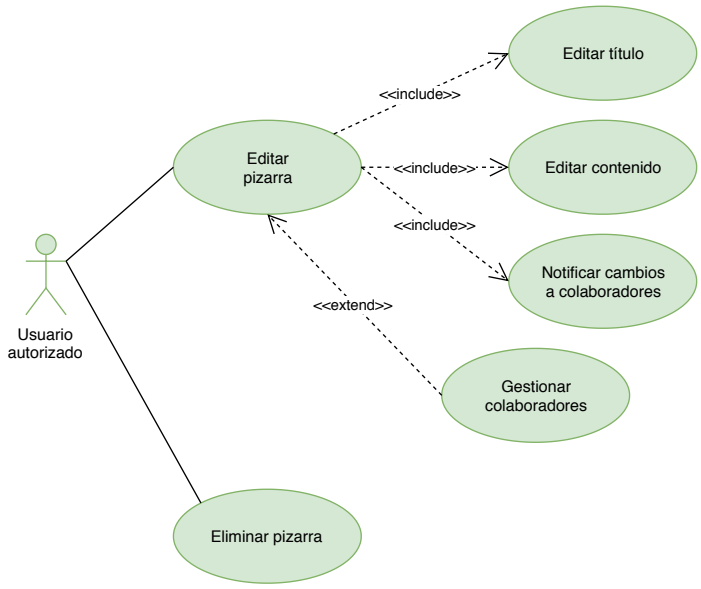

Figura 4.12: Frontend: Casos de uso del *frontend*

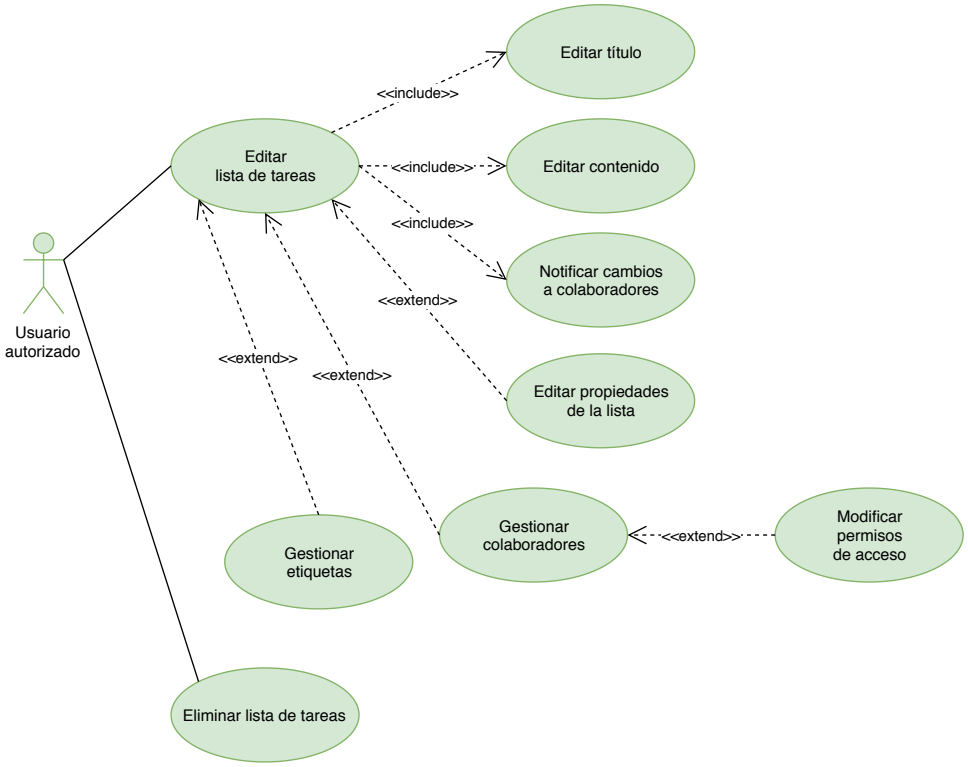

Figura 4.13: Frontend: Casos de uso del *frontend*

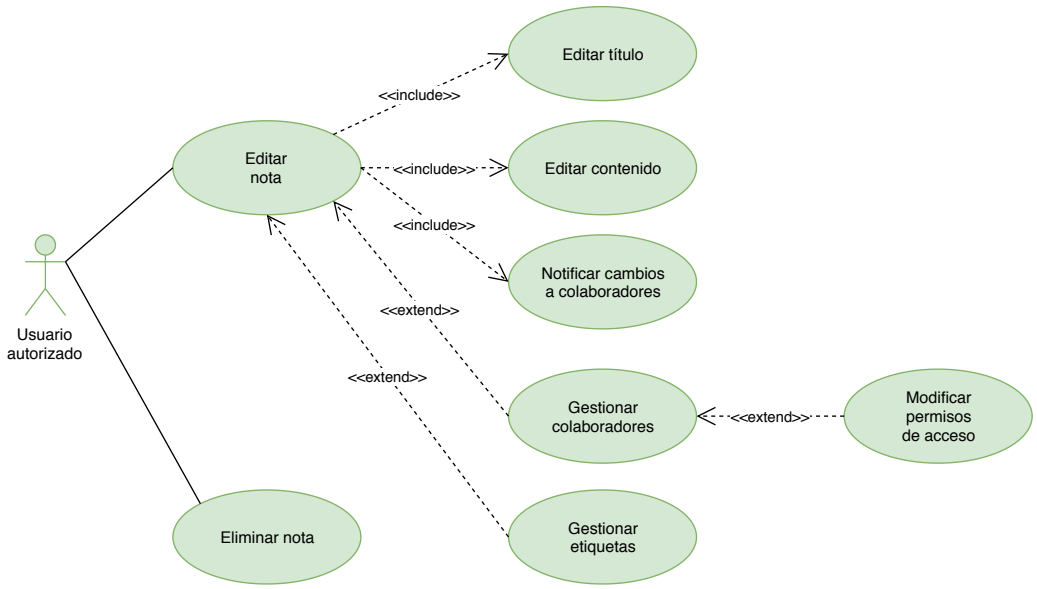

Figura 4.14: Frontend: Casos de uso del *frontend*

4.2.5 Diagramas de clases del backend

En esta sección se incluyen los diagramas de clases que muestran la estructura estática del sistema desarrollado: [4.16] [4.17] [4.15] [4.18] [4.19] [4.20] [4.21] [4.22] [4.23] [4.24] [4.25] [4.26].

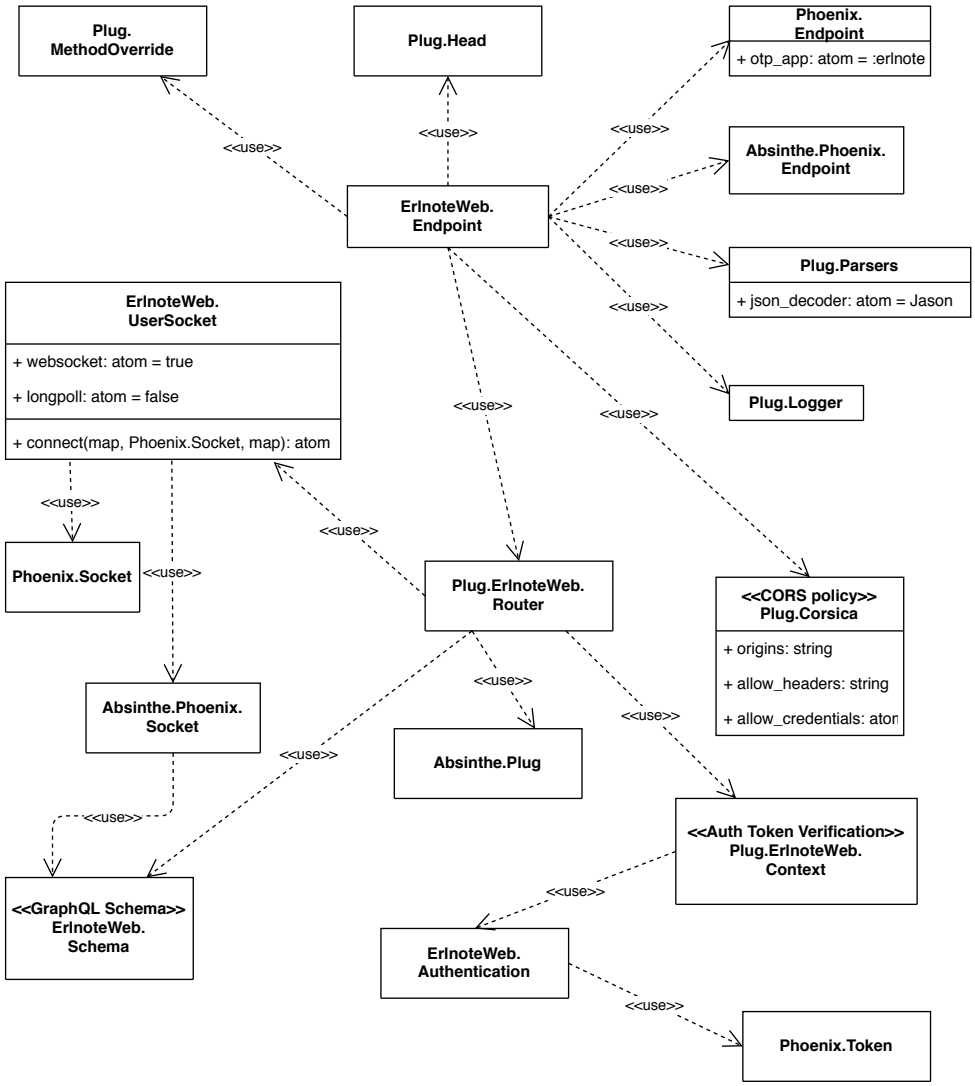

Figura 4.15: Diagrama de clases Backend Endpoint

4.2.6 Diagramas de secuencia

En este apartado se muestran diagramas de secuencia que tratan sobre: una operación de inicio de sesión 4.27, una operación GraphQL de mutación o consulta 4.28 y una operación GraphQL de suscripción 4.29.

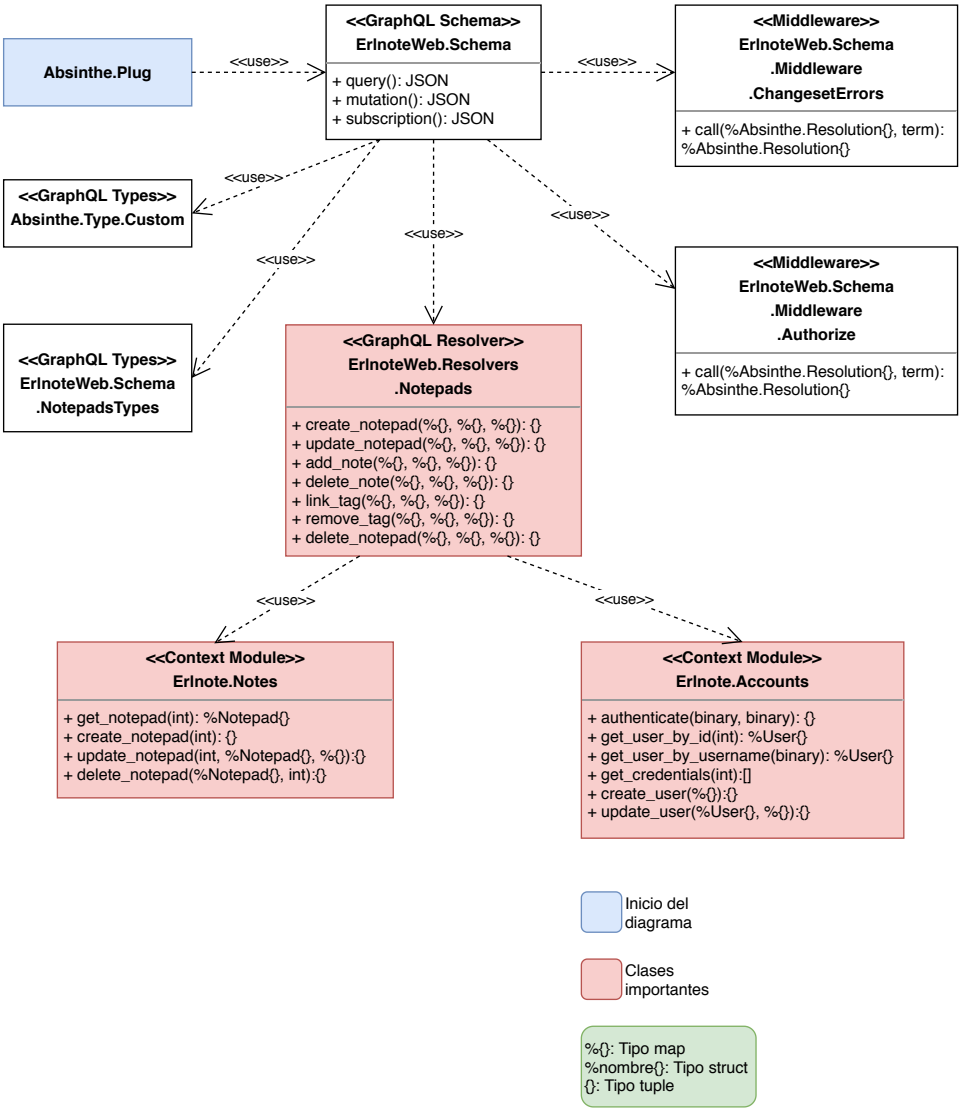

Figura 4.16: Diagrama de clases Backend: Bloc de notas

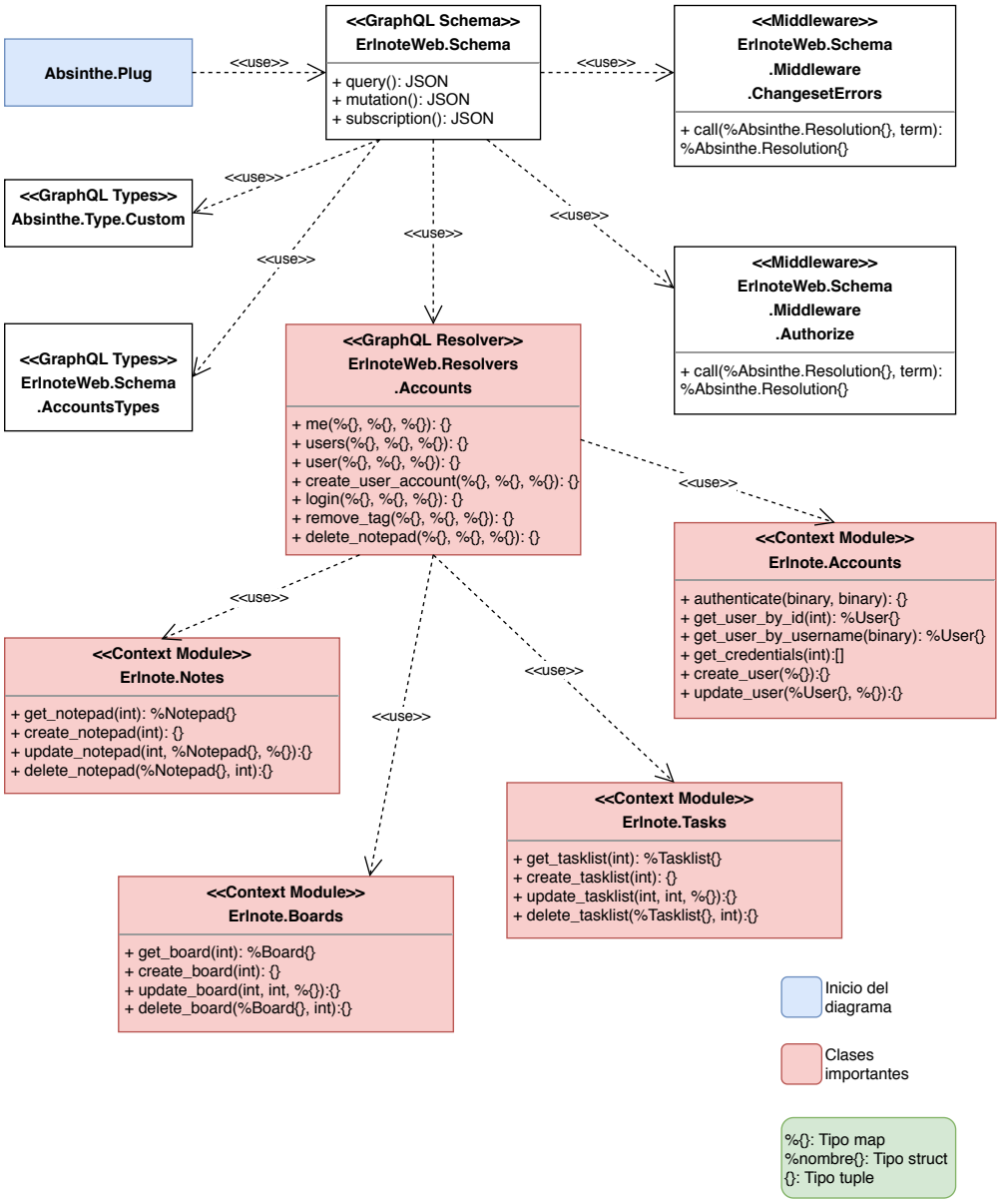

Figura 4.17: Diagrama de clases Backend: Cuentas de usuario

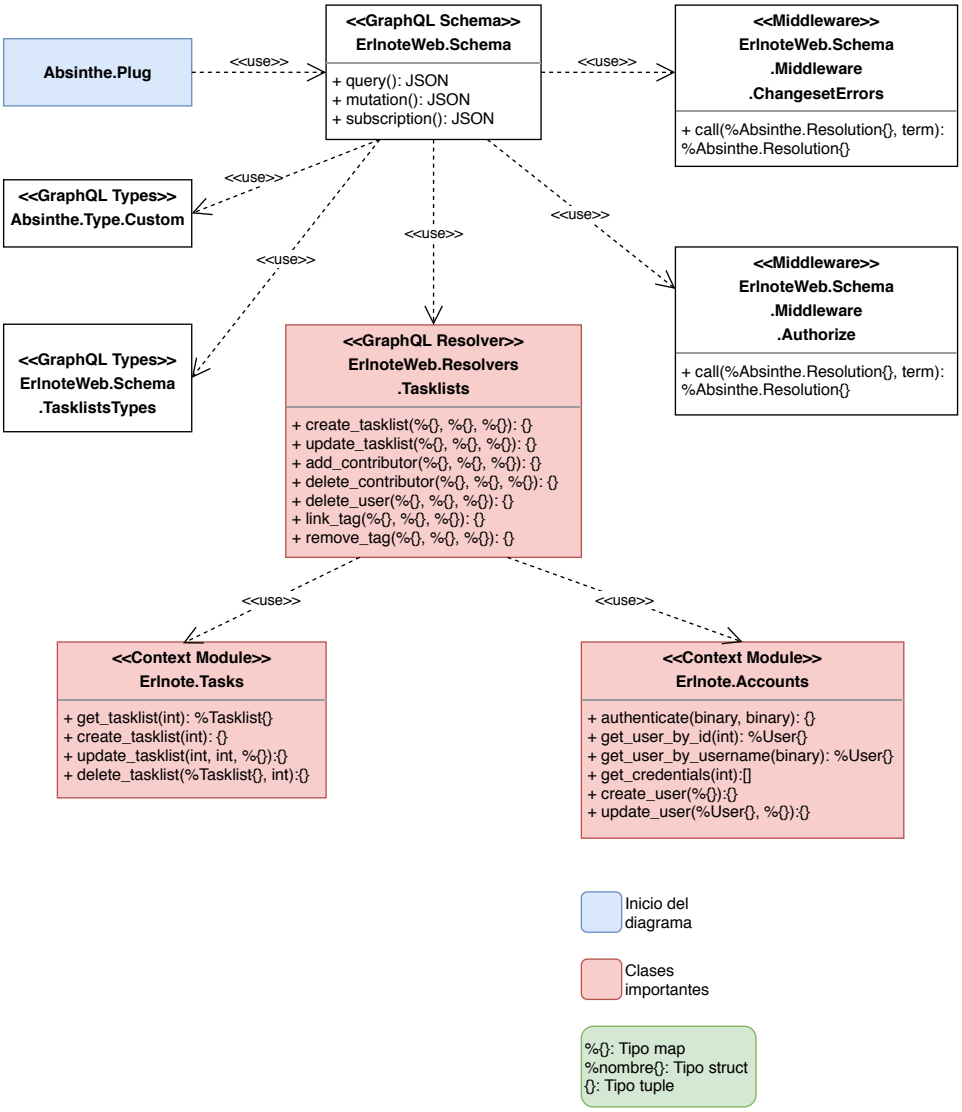

Figura 4.18: Diagrama de clases Backend: Lista de tareas

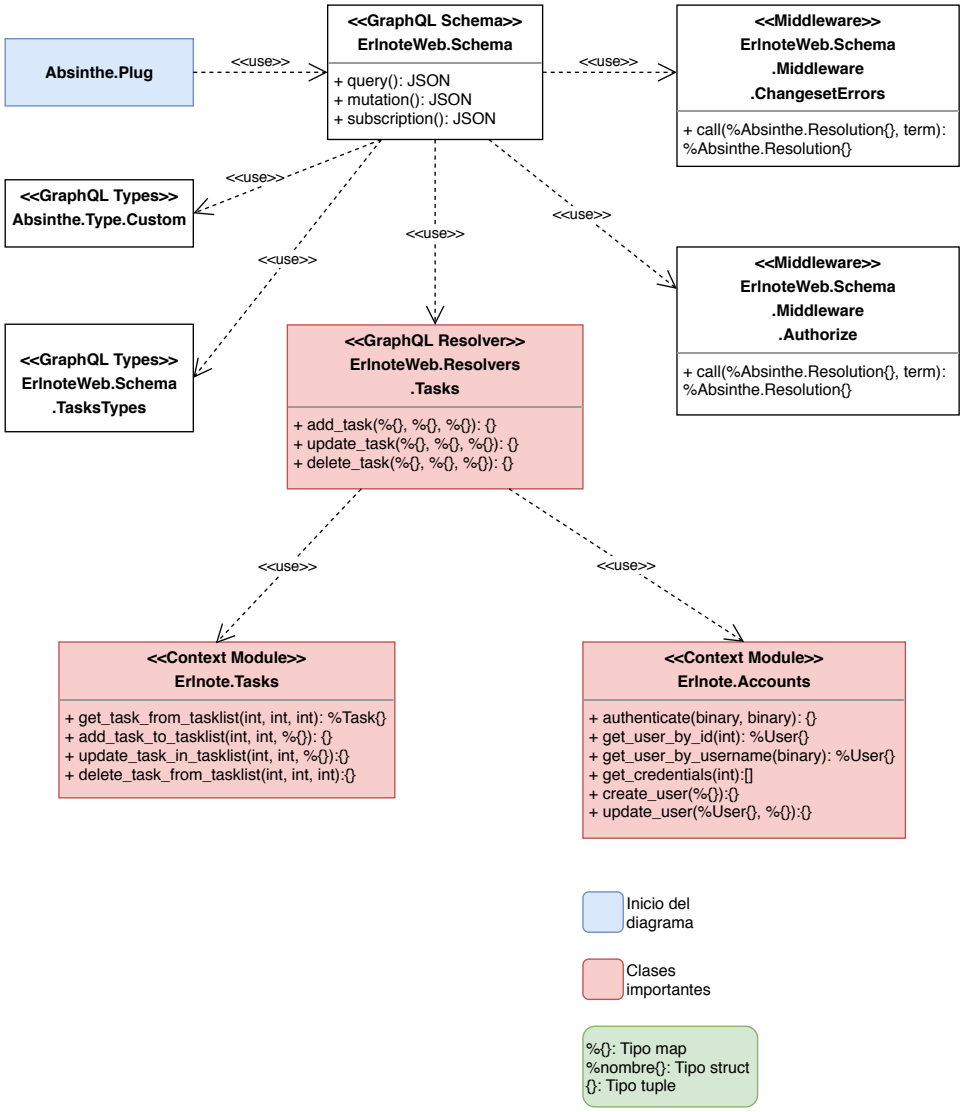

Figura 4.19: Diagrama de clases Backend: Tareas de lista de tareas

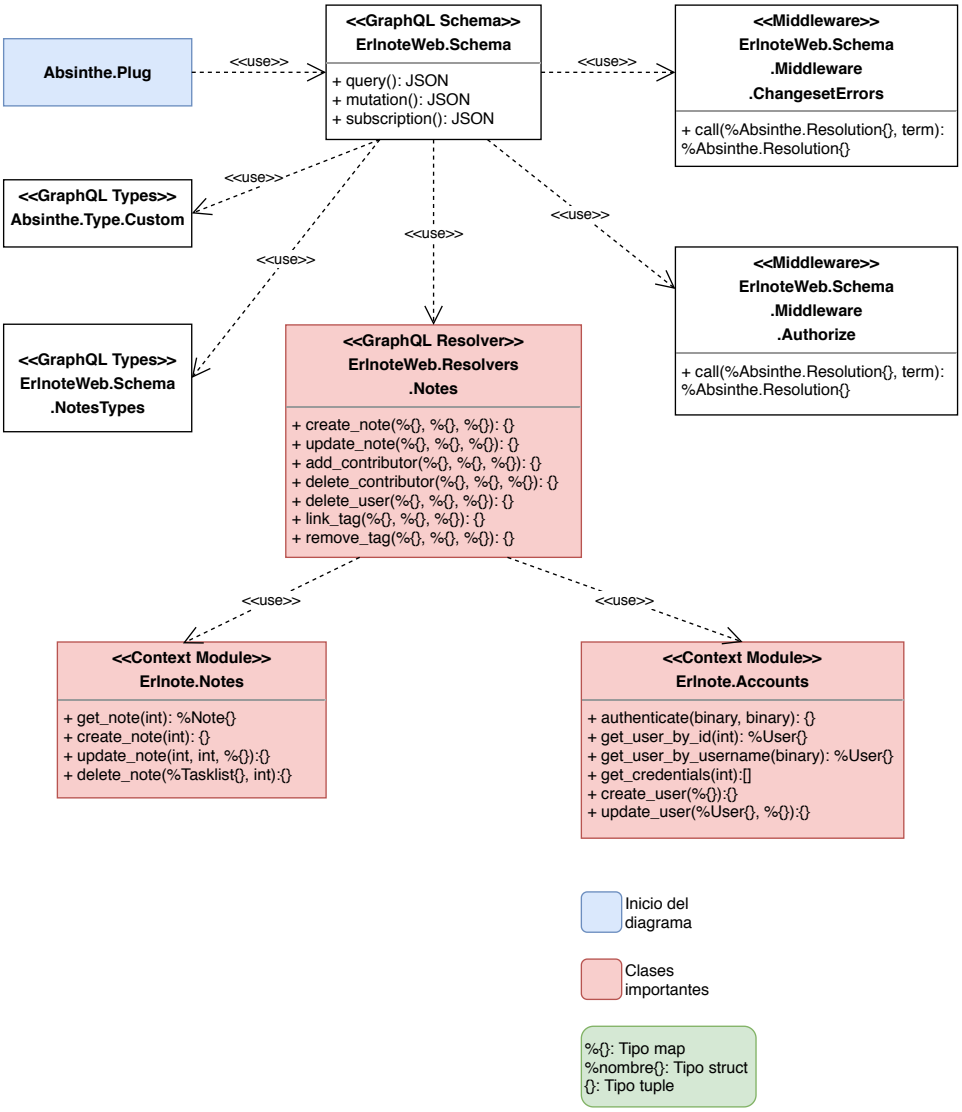

Figura 4.20: Diagrama de clases Backend: Notas

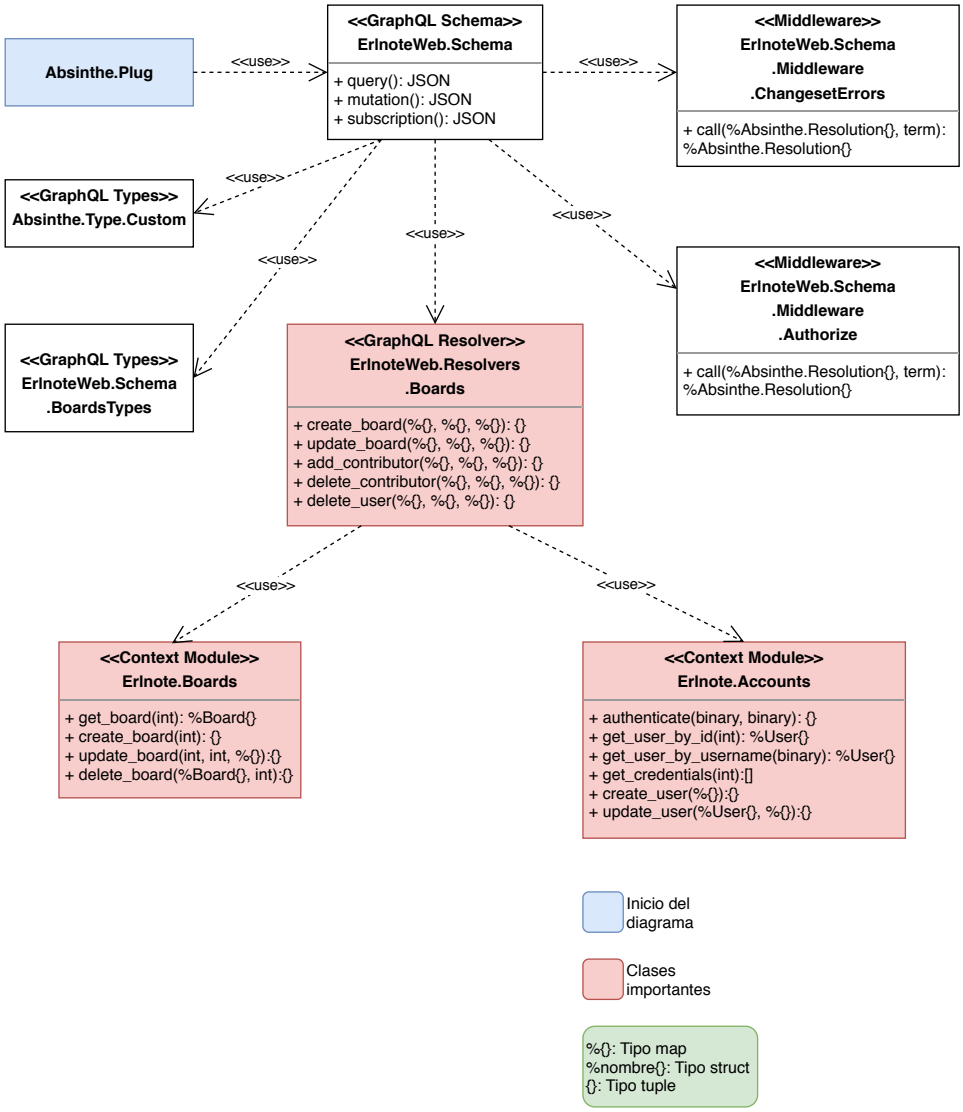

Figura 4.21: Diagrama de clases Backend: Pizarras

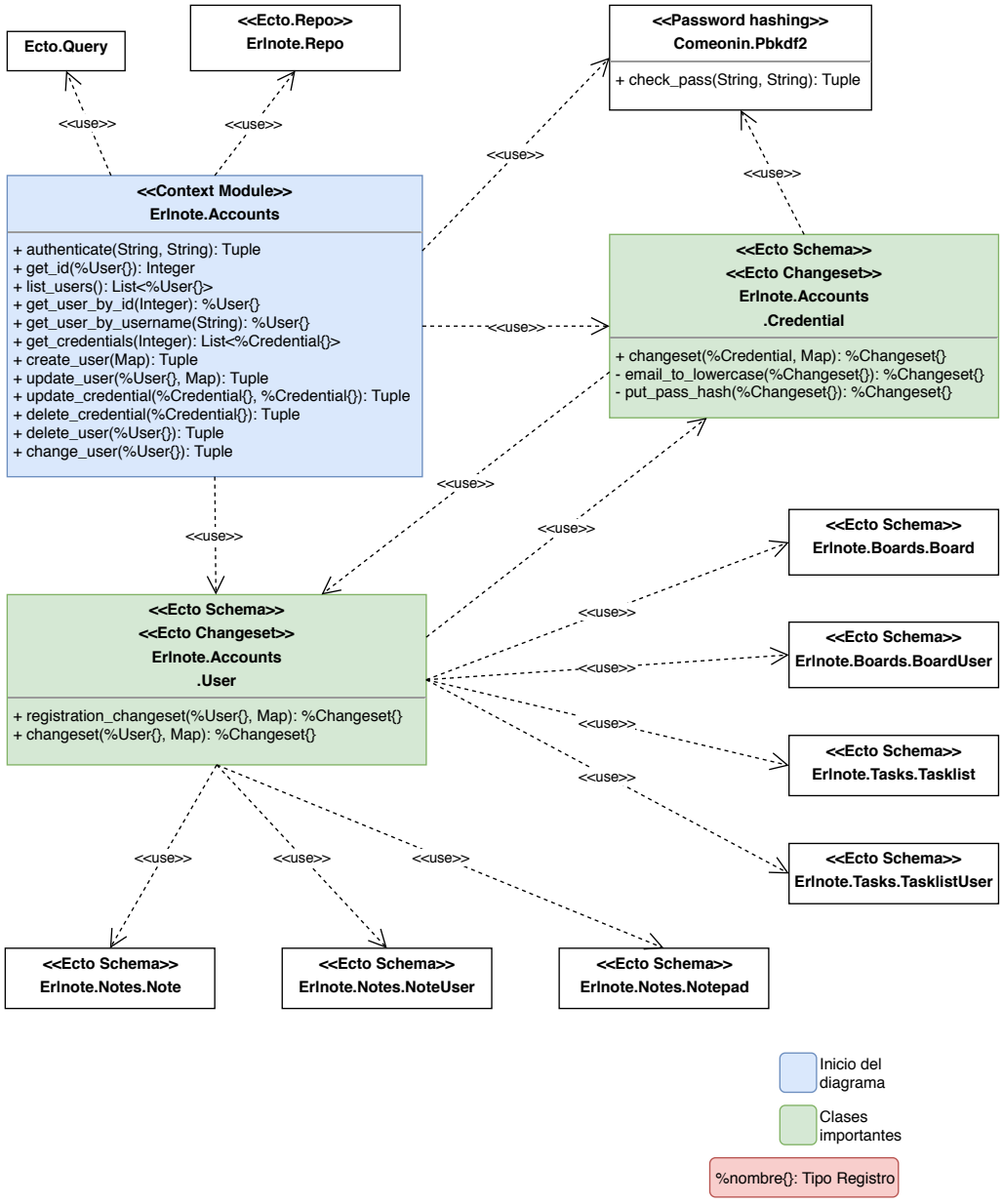

Figura 4.22: Diagrama de clases Backend: Contexto cuentas de usuario (negocio)

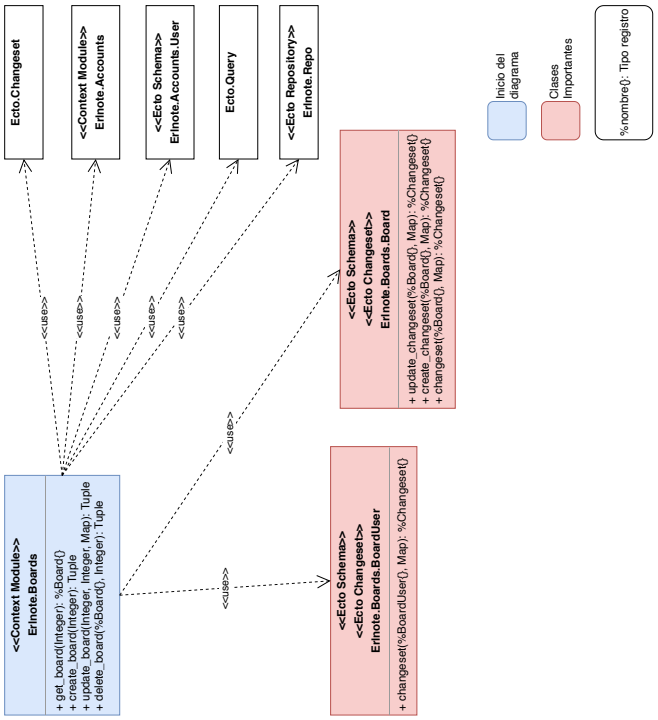

Figura 4.23: Diagrama de clases Backend: Contexto pizarras (negocio)

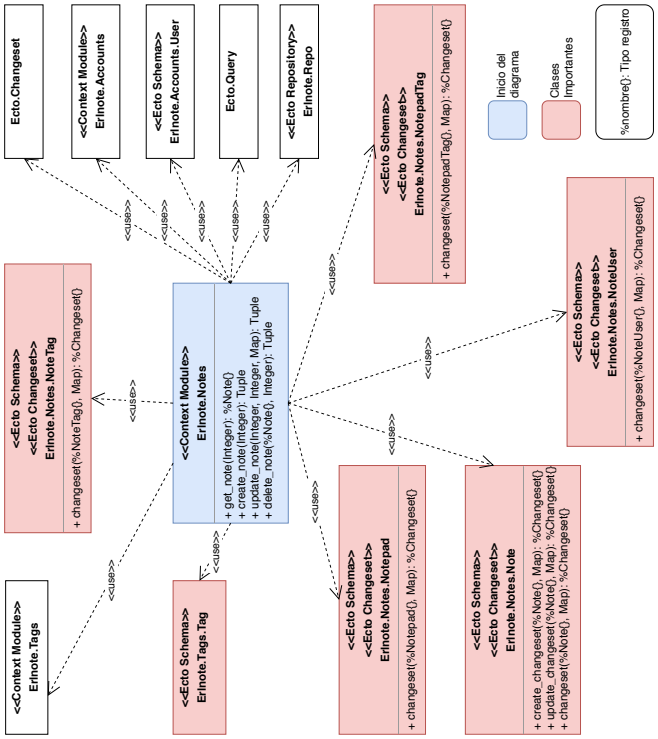

Figura 4.24: Diagrama de clases Backend: Contexto notas (negocio)

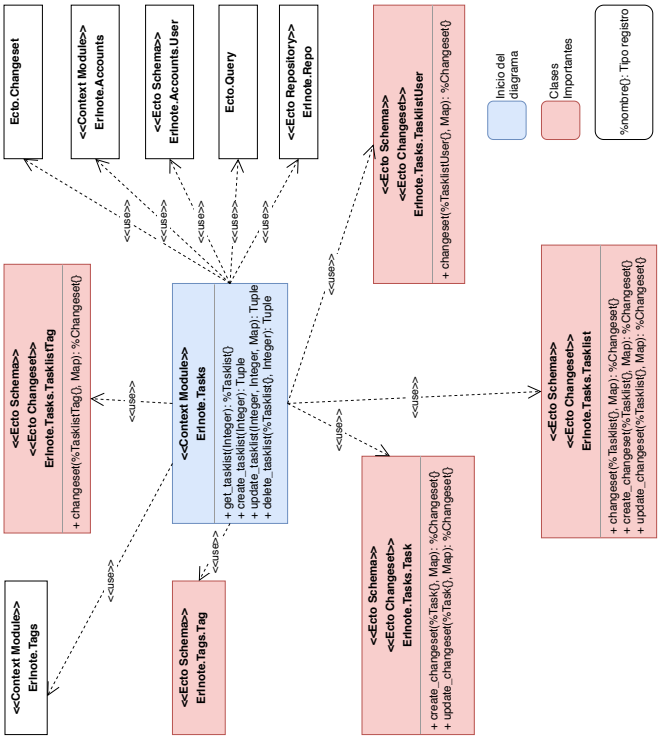

Figura 4.25: Diagrama de clases Backend: Contexto tareas (negocio)

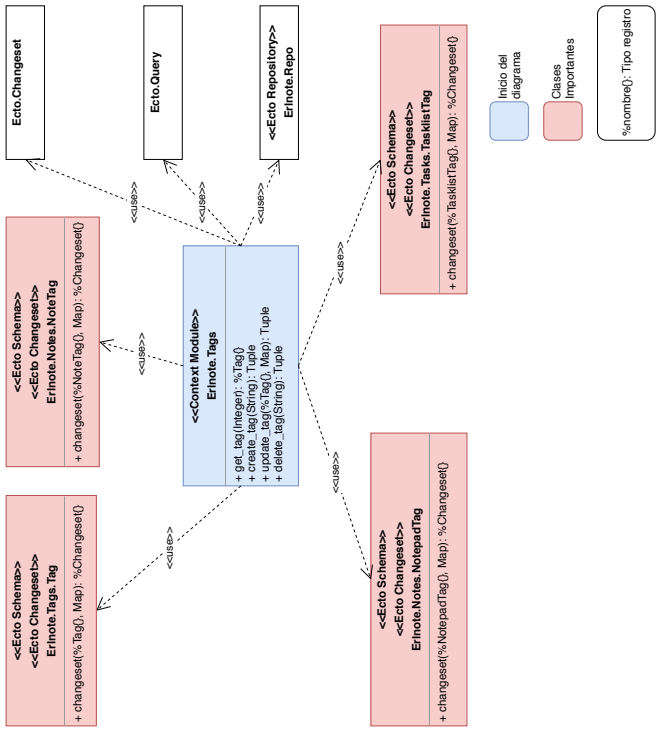

Figura 4.26: Diagrama de clases Backend: Contexto etiquetas (negocio)

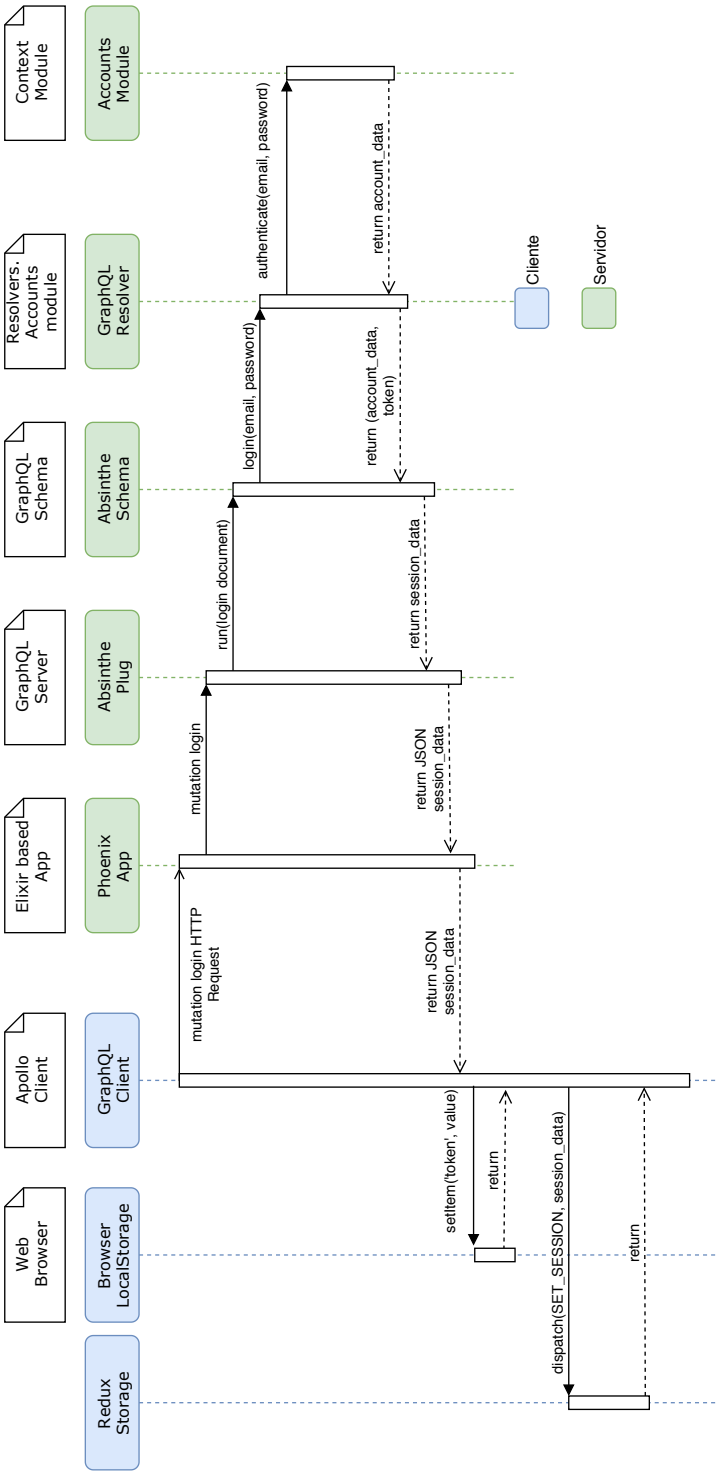

Figura 4.27: Secuencia de login

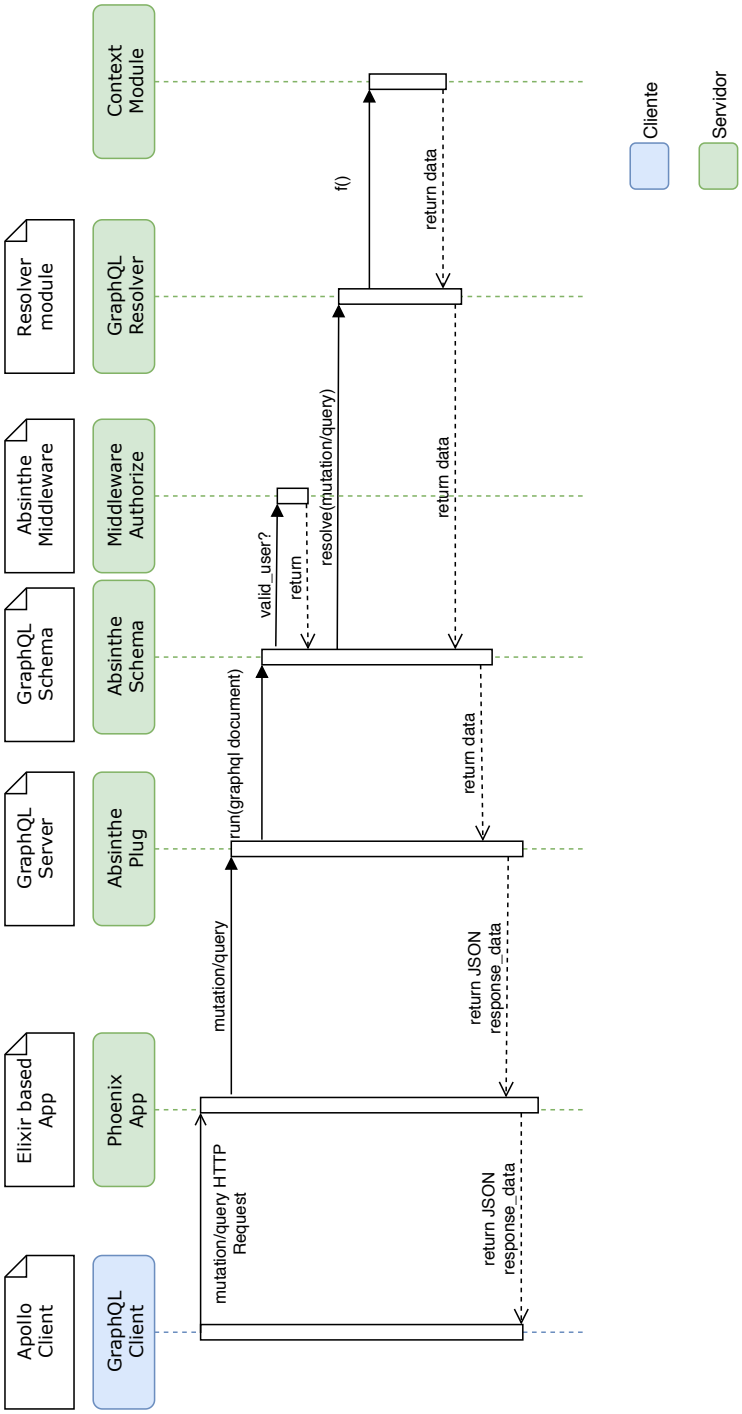

Figura 4.28: Secuencia en mutaciones/consultas GraphQL

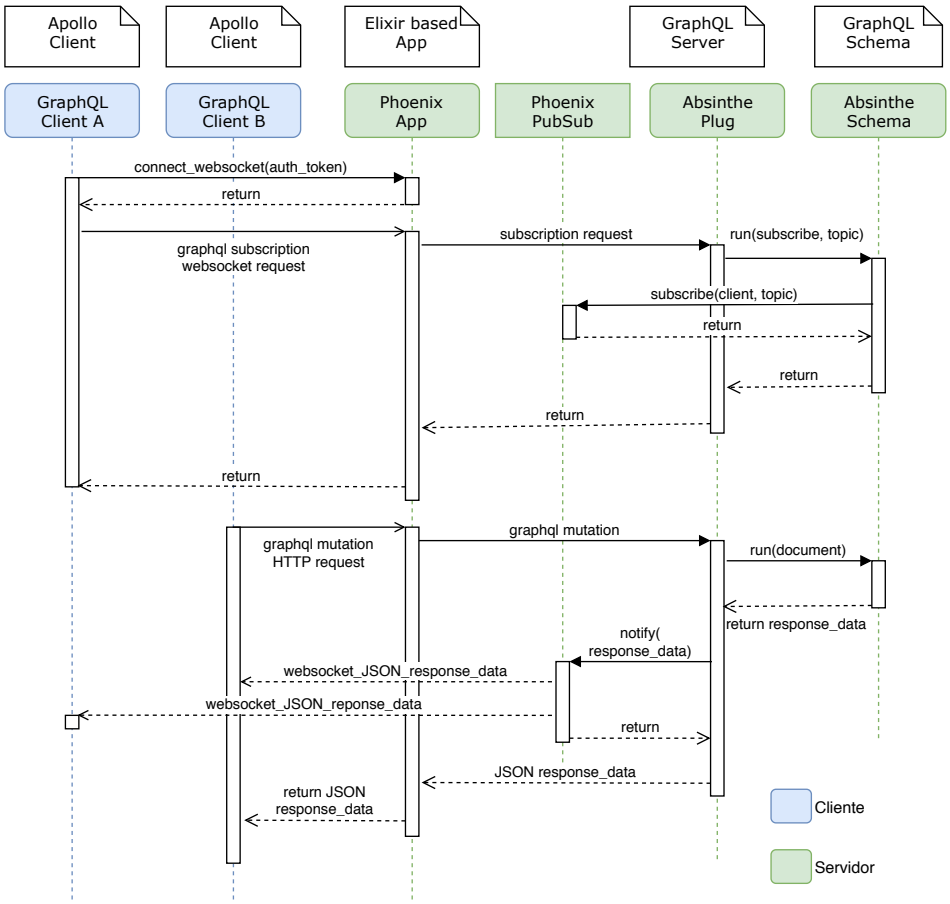

Figura 4.29: Secuencia en subscripciones GraphQL

Capítulo 5

Pruebas

En este capítulo de la memoria trataremos de describir, a grandes rasgos, las pruebas del sistema implementadas.

5.1 Inicialización de datos en el *backend*

Para realizar pruebas en el *backend*, o en el *frontend*, es necesario que el sistema disponga de un conjunto de datos de ejemplo. Una aplicación web, basada en el *framework* Phoenix, posee el fichero `priv/repo/seeds.exs`, en el cual se pueden incluir las instrucciones, consideradas oportunas, para poblar la base de datos. Este fichero es un *script* Elixir, que se suele invocar, de forma automática, al ejecutar `mix ecto.reset`, para reestablecer la base de datos.

A pesar de que los datos se pueden añadir, a la base de datos, directamente desde este fichero (usando las funciones del repositorio `insert!`, `delete!` o `update!`), es una buena práctica invocar, en su lugar, una función que realice este trabajo. En nuestro caso así lo hemos hecho, y el único contenido de este fichero es la invocación a la función `run` del módulo `Erlnote.Seeds`, que es la que realmente carga los datos de ejemplo.

5.2 Pruebas en el *backend*

Las pruebas del *backend* que se han implementado se encuadran, en su mayor parte, en la categoría de *pruebas unitarias*. Para la implementación de dichas pruebas se ha utilizado el *framework* suministrado por Elixir para dicha tarea, denominado ExUnit [23] B.12.

Al emplear ExUnit, obtenemos una herramienta para pruebas totalmente integrada en Elixir y en el Phoenix *framework*. Esto permite ejecutar las pruebas simplemente haciendo uso de la herramienta oficial de línea de comandos para gestión de proyectos de Elixir: `mix test`.

Cuando creamos nuestro proyecto Phoenix, se configura, en segundo plano, el entorno de

pruebas exUnit para el mismo. A partir de ahí, no es necesario hacer nada más que escribir los casos de prueba y los módulos *helper*, que dan soporte a las pruebas.

Las pruebas ExUnit se ubican en el directorio `test` de una aplicación Elixir. El *framework* de pruebas nos suministra tres módulos *helper*, que nos facilitan escribir las diferentes funciones de prueba que necesitemos. Estos módulos *helper* residen en el directorio `test/support`:

- Módulo *helper* `DataCase`: Este es el módulo que más utilizamos en este proyecto. Su función es facilitarnos el escribir pruebas que requieren el uso del *framework* de persistencia Ecto y, consecuentemente, suelen requerir interacción con un sistema de gestión de bases de datos. Este módulo crea un *sandbox* de la base de datos empleada, de forma que, en la ejecución de las pruebas implementadas, no se está operando contra la base de datos real, sino contra una base de datos ficticia. Esto nos permite realizar las pruebas, en un entorno análogo al real, pero sin miedo a dañar la base de datos real.
- Módulo *helper* `ConnCase`: Este módulo nos ayuda en la implementación de pruebas que requieren conexiones de red. Se usa frecuentemente en la realización de pruebas de integración, aunque no está restringido a ningún tipo de pruebas específico.
- Módulo *helper* `ChannelCase`: Este módulo nos asiste en la implementación de pruebas que hacen uso de los canales Phoenix para comunicación en tiempo real, haciendo uso del subsistema de publicación/suscripción.

El desarrollador puede escribir, obviamente, módulos *helper* personalizados. Dichos módulos también deben residir en el directorio `test/support`. A modo de prueba de concepto, hemos desarrollado un módulo *helper* personalizado, que está enfocado al desarrollo de pruebas de integración que hacen uso de suscripciones GraphQL. Este módulo se llama `SubscriptionCase` y su función es inicializar la base de datos con un conjunto de datos de prueba y establecer una conexión *WebSocket* con el servidor GraphQL Absinthe. Por tanto, las pruebas escritas, que hagan uso de dicho módulo, solo tendrían que limitarse a ejecutar las operaciones GraphQL empleando la conexión suministrada.

Los módulos *helper* son la piedra angular de ExUnit, puesto que todos ellos hacen uso de una *sandbox*, que permite al desarrollador despreocuparse por dañar el sistema real durante las pruebas. No obstante, ha de tenerse en cuenta que la escritura de módulos *helper* personalizados, suele requerir conocimientos muy avanzados del lenguaje Elixir, resultando casi imprescindible manejarse con soltura en el sistema de macros proporcionado por el lenguaje. Esto hace que la escritura de pruebas de un nivel aceptable requiera tiempo y experiencia, siendo una tarea poco recomendable para un *newbie*.

Las pruebas de una aplicación web Phoenix se alojan en dos directorios:

1. `test/nombre_del_proyecto`: Aquí deben colocarse los módulos con las pruebas del modelo de negocio y contextos (frontera del modelo de negocio).

2. `test/nombre_del_proyecto_web`: En este directorio residirán las pruebas asociadas a la parte web de la aplicación.

En nuestro caso, los módulos de pruebas unitarias ExUnit desarrollados, se ubican en el directorio `test/erlnote`, puesto que las pruebas escritas son para el modelo de negocio. Cada módulo consta de:

- Una función *fixture*: Esta función se encarga de preparar en entorno para la ejecución del test. Esto es, lleva el sistema al estado inicial de partida.
- Tests de prueba: Un test no es más que una función que se define empleando la macro `test`, seguida del nombre de la prueba. Para comprobar las condiciones bajo las que la prueba es o no satisfactoria se usa la macro `assert` o alguna de sus variantes.
- En algunos módulos se procede a agrupar varios tests en un bloque, lo cual se logra haciendo uso de la macro `describe`. Internamente, lo que se hace es utilizar el nombre del bloque `describe` como prefijo de los tests incluidos en dicho bloque.

En el caso específico de nuestro modelo de negocio, se han desarrollado 19 módulos (contextos incluidos). Para esos 19 módulos se han desarrollado 5 módulos de pruebas unitarias intensivas, para los 5 contextos existentes. Estos 5 módulos albergan 206 tests, para cada una de las funciones contenidas en los *módulos contexto*, en el momento de escribir esta memoria. Esto asegura, prácticamente, una total cobertura de pruebas unitarias del modelo de negocio. Se debe tener en cuenta que los 14 módulos restantes se corresponden con módulos Ecto asociados a *Schemas* y *Changesets*. En el caso de los *Schemas*, no proceden pruebas unitarias. En el caso de los *Changesets*, se trata de funciones de una lógica sumamente simple, que si bien se podrían probar, en realidad resultaría bastante excesivo y dichas pruebas tendrían sentido solo en caso de sistemas muy críticos.

A modo de ejemplo, se ha desarrollado lo que sería una prueba de integración con exUnit, usando el *modulo helper* para conexiones de red. Esta prueba se encuentra en el *path* `test/erlnote_web/schema/query/users_test.exs`. Dicha prueba está diseñada para ejecutarse de forma asíncrona. El test crea una conexión de red al *backend*. Posteriormente, realiza una consulta GraphQL, empleando el protocolo HTTP, contra la URL en la que se ubica la API. Finalmente, se comprueba que la respuesta HTTP tiene un código que no indica error y que el contenido de la misma se corresponde con lo preguntado. Este tipo de pruebas, utilizando exUnit, derivan, frecuentemente, en casos realmente largos y no triviales, que serían solo abordables con un equipo de desarrollo adecuado o con una gran cantidad de tiempo.

Como opinión personal, este autor cree que ExUnit suministra un nivel de prueba aceptable, pero todavía muy lejos del CommonTest [24] de Erlang, en lo referente a opciones y flexibilidad. Así que, en este caso, con tiempo suficiente, el enfoque se inclinaría más hacia

esta segunda opción, ya que además de ser mucho más completa, también es bastante más didáctica.

Una vez probado el modelo de negocio, resta probar la API GraphQL. Para esto hemos empleado la herramienta gráfica GraphiQL [60] E, integrada en Absinthe [34]. En este caso se trata de una herramienta gráfica que se ejecuta en el navegador. Dicha herramienta está habilitada en nuestro proyecto, desde el módulo `ErlnoteWeb.Router`, y es accesible en la url `http://localhost:4000/graphiql`. Esta herramienta tiene varias interfaces de usuario, pero en nuestro caso la que se ajusta a nuestras necesidades es la avanzada. La herramienta se divide en pestañas y, en cada pestaña, se puede crear una interacción con la API GraphQL. En las opciones de configuración se puede establecer la URL HTTP en la que se encuentra la API y la dirección/parámetros del socket empleado para suscripciones. Del mismo modo, es posible configurar un *token* de autenticación, que se incluirá en la cabecera HTTP con cada nueva operación. En el caso de *WebSocket*, el *token* de autenticación debe incluirse como un parámetro en la URI especificada para el *socket*. Además de todo esto, la herramienta posee dos áreas de texto independientes: un área para escribir la operación GraphQL que se desea ejecutar (así como establecer los valores de las variables que procedan) y otra área en la que se muestra la respuesta emitida por el servidor.

Con la herramienta GraphiQL se ha podido probar, exhaustivamente, el correcto funcionamiento de la API GraphQL, sin necesidad de disponer, en modo alguno, del *frontend*. Esto nos ha permitido el casi completo desarrollo del *backend*, dejando el *frontend* como fase final del desarrollo. Al fin y al cabo, lo que hace el *frontend* es, exactamente, lo mismo que se realiza desde GraphiQL, pero un poco más bonito visualmente hablando. Toda operación GraphQL implementada ha sido probada con GraphiQL, como paso previo a su uso en el *frontend*.

Si alguien se pregunta porqué no se han escrito pruebas unitarias relativas a la parte web de la aplicación, hemos de comentar que no resultan especialmente relevantes ni de interés. La parte que yace fuera del modelo de negocio consiste en, por un lado, el esquema GraphQL sobre el que no proceden pruebas unitarias y, por otro lado, las funciones *resolver* cuyo comportamiento consiste, fundamentalmente, en invocar funciones de los *módulos contexto*. Dichas pruebas sí tendrían relevancia, por ejemplo, en una arquitectura Modelo-Vista-Controlador.

En este proyecto no se han llevado a cabo pruebas no funcionales, como podrían ser pruebas de rendimiento, que serían necesarias para cualquier sistema enfocado a producción. Como es lógico, lo más deseable es que un sistema disponga de la mayor cantidad de pruebas posible. No obstante, debido a las limitaciones de tiempo y demás recursos, se hizo obligatorio recortar en algunas áreas. En este caso, se optó por recortar en pruebas antes que en funcionalidad, pero esto no es indicativo del nivel de importancia.

5.3 Pruebas en el *frontend*

En el caso del *frontend* no ha sido posible, por limitaciones temporales, escribir programáticamente las pruebas. Pero esto no significa que no se hayan llevado a cabo. Se han realizado pero de forma manual.

Puesto que, a todas luces, resultaba excesivo acometer el estudio de otro *framework* de pruebas, adicionalmente a los ya estudiados, se optó por enfocarse en un método un poco más rápido y menos complejo, que parece perfectamente viable teniendo en cuenta que no se está tratando con un sistema enfocado a producción. Como los cimientos del *frontend* son JavaScript, y este lenguaje se guía por eventos, se ha considerado el *frontend* como una máquina de estados. El procedimiento consiste en analizar un estado inicial, disparar el evento y comprobar que el estado final es el deseado. Este procedimiento es muy fácil de llevar a cabo gracias a las herramientas de desarrollo de React y Redux, que permiten visualizar tanto el estado de React como de Redux en cualquier instante. Como anécdota comentar que también existen herramientas de desarrollo que, en teoría, permiten visualizar el estado del cliente Apollo GraphQL. Pero, en realidad, estas herramientas fallan demasiado y no resultan muy útiles, puesto que en nuestro proyecto el estado de Apollo se refleja en el estado de Redux y los posibles errores se visualizan claramente en la consola del navegador.

En este trabajo, el *frontend* no resulta, ni de lejos, una parte crítica del sistema. Tal y como está implementado el *backend*, es muy improbable poder comprometerlo desde el *frontend* con entradas/acciones de usuario incorrectas, ya que debilidades tales como desbordamientos de *buffers* han sido contempladas. Por otro lado, el sistema trata con total normalidad entradas de usuario que hacen uso de caracteres Unicode y formatos requeridos, tales como direcciones de correo electrónico, son comprobadas en el *backend*, de forma que si el formato es incorrecto, los datos no llegarán a modificarse y el cliente recibirá el mensaje de error correspondiente. Lo que sí es mejorable es la información que recibe el usuario del *frontend* en tales casos, pero eso ya formaría parte de un sistema con un nivel de madurez mucho más alto, que difícilmente sería alcanzable en un trabajo de este tipo.

Capítulo 6

Conclusiones

En este último capítulo de la memoria, se presenta la situación final del trabajo, las lecciones aprendidas, y las posibles líneas futuras.

6.1 Situación final de la herramienta objeto de desarrollo

La herramienta objeto de desarrollo, en este caso una aplicación web/móvil de planificación personal, ha alcanzado los objetivos inicialmente fijados. Se ha de tener en cuenta que dicha herramienta nunca fue el objetivo principal del proyecto, sino una forma de darle visibilidad al trabajo realizado en el *backend*. Teniendo presente esta premisa, en ningún momento se pretendió desarrollar un *software* enfocado a producción y, de hecho, la herramienta carece de muchas funcionalidades que serían deseables en cualquier aplicación en producción que se encuadrara dentro de su categoría.

Claramente, se ha obtenido un sistema multiusuario y con una seguridad más que aceptable, considerando que no se trata de aplicaciones críticas tales como las bancarias, etc. El uso de protocolos seguros, en este caso HTTPS:// y WSS://, junto con *tokens* de autorización, dan al sistema soporte para una seguridad más que aceptable.

Por otro lado, también se ha obtenido un sistema fácilmente extensible. En el lado del servidor, el enfoque basado en módulos, exigido por Elixir, facilita mucho este requisito. Esto, junto con el uso de *context modules*, que separan los módulos de negocio del resto y los agrupan por áreas, hacen el *backend* extremadamente fácil de extender y comprender, sin mayor complejidad. Como añadido, el uso de GraphQL hace la ampliación de la API del servidor, prácticamente, trivial; puesto que extender el esquema GraphQL no resulta para nada complejo, una vez disponemos de una base, con un mínimo conocimiento del lenguaje. En el lado cliente, también disfrutamos de una gran facilidad para la extensión: la sustitución del cliente GraphQL por otro no debería de resultar un proceso dramático, reemplazar Redux por otra solución FLUX, tampoco sería difícil, y el enfoque de la interfaz de usuario, como un árbol je-

rárquico de componentes, hacen muy sencilla su modificación. En resumen, tenemos la base de un software muy flexible a la hora de adaptarse a cambios.

Gracias a la acertada arquitectura adoptada, haciendo *frontend* y *backend*, prácticamente, independientes, reemplazar un *frontend* por otro es extremadamente fácil y nada propenso a errores. Esto permitiría que los usuarios dispusieran de una GUI moderna, al tiempo que podemos conservar el *backend* inalterado por largos períodos de tiempo.

Por último, el software obtenido es colaborativo, ya que soporta múltiples usuarios colaborando, valga la redundancia, en un mismo recurso. Dicho esto, hemos de comentar que se ha llegado a la conclusión de que las operaciones GraphQL de suscripción son adecuadas para notificaciones, alarmas, etc. pero no así para edición de texto colaborativa. Esto viene dado porque existe cierta probabilidad de que se generen conflictos, entre clientes editando el mismo recurso de forma simultánea. Este problema resulta casi inevitable, a no ser que se acometa la implementación de una lógica extremadamente compleja de integrar, lo cual carece de sentido, actualmente, puesto que existen soluciones más simples susceptibles de una integración indudablemente más sencilla. Por ejemplo, usando tipos de datos replicados libres de conflictos, empleando el *backend* como *relay*. Otro comportamiento que se debe tener en cuenta, referente a las suscripciones, es aquel que consiste en que una suscripción envía la notificación del resultado tanto a los usuarios suscritos como al emisor del evento origen de la notificación. Esto hace que el usuario que ejecuta la mutación GraphQL, causante del disparo de la suscripción, reciba el resultado de la misma por duplicado. Esto tiene especial relevancia en la edición de texto compartido, puesto que si no se eliminan las respuestas duplicadas, se generan comportamientos no deseados complicados de depurar.

6.2 Objetivos alcanzados y pendientes

De los objetivos planteados al inicio del proyecto, se ha satisfecho, casi, la práctica totalidad.

Se ha desarrollado una aplicación web/móvil, que se puede utilizar en diferentes tipos de dispositivos, siguiendo un enfoque puramente funcional, tanto en el *frontend* como en el *backend*. Es comprensible que se puedan criticar las funcionalidades existentes en la aplicación, así como la ausencia de otras muchas, pero recordemos que el *frontend* se implementó solo como un modo de hacer emerger el trabajo realizado en el *backend*.

Como resultado final se ha obtenido un sistema extremadamente fácil de entender y mantener. Cualquier persona, con unos mínimos conocimientos técnicos, debería poder hacerse una idea, en un corto intervalo de tiempo, tanto del sistema en su conjunto como de las áreas específicas. El lenguaje Elixir permitió escribir un código fuente que goza de una alta claridad, así como funciones no excesivamente largas y muy sencillas. En el caso del cliente, el código

resulta algo más denso de leer y, es posible, que también de comprender, pero esto es debido a que HTML y JSX no son, precisamente, lenguajes que contribuyan mucho a estos objetivos.

El uso tanto de Elixir [3], como de Phoenix [37], nos ha permitido crear un *backend* especialmente bueno en lo que a tolerancia a fallos se refiere. Los árboles de supervisión empleados nos permiten disponer de un servidor muy robusto, el cual no ha presentado fallo alguno incluso durante el desarrollo. Los supervisores de persistencia, del servidor GraphQL y del propio Phoenix *Framework* constituyen los cimientos de una tolerancia a fallos realmente magnífica.

En lo referente a la tolerancia a fallos en el *frontend* no se ha podido evaluar, convenientemente, por falta de tiempo. De todas formas, es realista asumir que es muy inferior a la que presenta el *backend*.

El software obtenido demostró un buen tiempo de respuesta, en general. No se observan ralentizaciones destacables en ningún punto y la BEAM hace bien su trabajo, al mantener el sistema estable dentro de los límites que se esperan normales. No obstante, sería aconsejable ejecutar múltiples *benchmarks* para confirmar esto de un modo más formal.

El principal objetivo, que queda pendiente, es la obtención de un sistema bien documentado. La documentación escrita es poco más que a modo de ejemplo y anecdótica, pero es que documentar el software completo constituiría un proyecto en sí mismo. No obstante, en el afán de contribuir a la buena documentación de los sistemas software, durante este proyecto se ha colaborado, simultáneamente, en la localización del sistema de generación de documentación de Elixir (alias exDoc) para su traducción al gallego, al español, …. En general, para el soporte de múltiples *locales* [61, 62].

Otras tareas inalcanzables, por cuestión de tiempo nuevamente, son la optimización del *backend* y la implementación de todo tipo de pruebas que serían obligatorias en producción. Aunque hay que tener en cuenta que cualquiera de las dos tareas sería susceptible de un proyecto completo.

6.3 Trabajo futuro

- El tratamiento de caracteres con tilde, en las áreas de texto del *frontend*, es incorrecto por defecto. Esto se debe a que no se interpreta adecuadamente la pulsación de la tilde antes de la vocal correspondiente, dando lugar a "´a" en lugar de "á". Esto ha sido corregido mediante sustitución de la cadena correspondiente. No obstante, sería deseable buscar una librería más avanzada, una implementación de áreas de texto más potente o, en su defecto, desarrollar más el mapeo de caracteres.

- Existe un cuello de botella en el *backend* que merma, sensiblemente, el rendimiento de la aplicación. Esto se debería corregir mediante la introducción de una caché entre el *resolver* y Ecto, de forma que se implementara una especie de escritura retardada sobre

la base de datos. Permitiendo amplificar la potencia del *backend* como *relay*.

- Apollo 2.5 presenta un serio *bug*, de forma que no ejecuta el método `onCompleted` cuando se obtiene una consulta desde la caché y fetchPolicy es *cache-first*. Esto obliga a cambiar el valor de fetchPolicy a alguna política que contenga *network*. Esto es una solución temporal y se debería hacer un seguimiento de si futuras versiones corrigen el problema. Si la política correcta de caché funcionara correctamente, sería posible minimizar el número de operaciones de red. Aunque las realizadas con la política actual constituyen un número perfectamente asumible.

- Es necesario implementar pruebas "más formales" para el *frontend*. En este caso se han empleado los complementos de React y Redux, para navegadores web, junto con las herramientas para desarrolladores proporcionadas por el navegador web, para comprobar el correcto funcionamiento del *frontend* y los resultados de las pruebas manuales. Esto es posible porque el *frontend* no requiere de una lógica excesivamente compleja y no existen demasiadas restricciones sobre las entradas de usuario en la aplicación.

- Se requiere implementar las validaciones de los campos de los formularios empleados en el *frontend*. Esto resultó imposible de acometer por falta de tiempo.

- Es posible mejorar el código fuente del *frontend* de forma que quede más limpio y óptimo. Mucho código, en el cliente, fue escrito de forma rápida, debido a que el tiempo estimado para el proyecto se encontraba excedido muy sensiblemente.

- En este proyecto no se han desarrollado las funcionalidades, que serían deseables, relacionadas con la búsqueda textual. En principio, estas funcionalidades no son de especial complejidad a la hora de desarrollarlas. Como propuesta se han considerado dos alternativas: bien emplear las capacidades de búsqueda textual ofrecidas por PostgreSQL [63], bien escribir un módulo Elixir que suministre una API mínima [64] para hablar con un servidor de búsqueda textual tal como ElasticSearch [65].

6.4 Reflexiones técnicas

El framework Angular se presenta como una opción extremadamente potente y flexible para el desarrollo en el *frontend*. Sin embargo, en muchas ocasiones, puede no ser la mejor opción, teniendo en cuenta su tamaño y su pronunciada curva de aprendizaje. Su ventaja principal es que, prácticamente, todo lo necesario para el desarrollo está disponible dentro de la propia plataforma. Por ejemplo, se dispone de enrutador, librería *Material Design* [66] para la interfaz de usuario, ...No obstante, teniendo en cuenta el limitado tiempo disponible para el trabajo, hemos descartado su uso. En su lugar, hemos optado por una *stack* basada

en React, React-Router, React-Redux, React-Apollo y React-Bootstrap. Como ventaja hemos obtenido una curva de aprendizaje menos pronunciada y un entendimiento más rápido. Como desventaja hemos de comentar que estas librerías son "no oficiales" y, en consecuencia, más susceptibles de generar problemas de mantenimiento en un futuro.

GraphQL es una alternativa al omnipresente REST [67]. En concreto, la implementación basada en Elixir, Absinthe [34], se muestra muy completa y robusta. Absinthe es fácil de comprender y las APIs, basadas en el mismo, simples de implementar, una vez se comprenden los pocos conceptos básicos sobre los que se sustenta. Pero siempre hay que tener presente que, a priori, no es mejor ni peor que REST, simplemente es un enfoque diferente. Su idoneidad vendrá dada por las requisitos de la API a desarrollar y sus necesidades futuras. Quizá, el aspecto más negativo de Absinthe sea su documentación oficial. No es precisamente la librería mejor documentada del ecosistema Elixir, sobre todo en lo referente a suscripciones o al cliente Apollo. Bien es cierto que existe un libro específico [68] sobre la librería, pero flaquea, precisamente, en el punto comentado.

El proyecto se planteó, inicialmente, con la intención de utilizar el sistema de gestión de bases de datos No-SQL CouchDB [31]. La elección no fue realizada al azar, sino basándose en el lenguaje en que dicha base de datos está implementada, que no es otro que Erlang [69]. Sin embargo, su uso tuvo que ser descartado por un cúmulo de factores. No parecen existir librerías Erlang/Elixir actualizadas que faciliten la operación contra Couch. Las librerías encontradas son muy escasas y adolecen de un buen mantenimiento para las versiones actuales de Couch. No obstante, también se consideró la implementación de un módulo intermedio, escrito en Elixir, que permitiera operar contra CouchDB. Esta idea tuvo que ser descartada después de analizar la posible complejidad de la implementación en relación al tiempo disponible.

El lenguaje Elixir ha demostrado ser perfectamente viable para el *backend*, en el ámbito del desarrollo web. La mayoría de estos servicios no suelen ser CPU intensivos, por lo que la BEAM es una alternativa que se ajusta, perfectamente, a este tipo de aplicaciones. Cierto es, también, que Elixir presenta un ecosistema pensado, precisamente, para este tipo de desarrollos. Como punto negativo hemos de resaltar el escaso abanico de opciones disponibles entre las cuales elegir. Todo es perfectamente viable mientras uno se mantenga en las librerías "oficiales" suministradas por el lenguaje. En el momento que uno intenta buscar opciones alternativas, es difícil encontrar algo con una calidad suficiente, que muchas veces ni existe. Por ello, no es infrecuente el tener que recurrir a librerías Erlang. Esto nos lleva a concluir que es muy difícil concebir Elixir sin Erlang. Estos inconvenientes fueron previstos desde un principio, ya que se era plenamente consciente de que estamos tratando con un lenguaje de uso minoritario, aunque siempre en evolución.

Existen muy pocos *frameworks* para el desarrollo web basado en Elixir. De todas formas,

el *framework* Phoenix [37] es una buena elección, con características más que suficientes, en el lado del servidor, para cualquier aplicación web moderna. Su arquitectura basada en encadenamiento de *plugs* (funciones) lo hace fácil de comprender y de expandir.

La librería de persistencia Ecto [29], para tratar con el acceso a datos, ha demostrado una calidad muy elevada. Probablemente, de lo mejor, sino lo mejor, que se ha encontrado hasta la fecha. Una librería robusta, flexible, muy completa, bien documentada y con un enfoque realmente fácil de comprender. Escribir código Ecto resulta una tarea incluso divertida, pero esto ya es una apreciación personal. El libro oficial [70] en el que se explica el funcionamiento y capacidades de la librería posee una claridad en las explicaciones que merece la pena destacar. El gran problema que se observa en esta librería es que se centra, principalmente, en el sistema de gestión de bases de datos PostgreSQL [30]. Mientras usemos este sistema, no deberíamos tener grandes problemas. Una vez consideramos otras alternativas es cuando los problemas empiezan a aflorar: adaptadores poco cuidados, con mantenimiento deficiente o, incluso, inexistentes.

Una vez sumergidos en el ecosistema Elixir, hemos de concluir que: si se desea implementar una aplicación web, cuyo *backend* esté basado en Elixir, hoy por hoy, la única opción seria es emplear el *framework* Phoenix, la librería de persistencia Ecto y el sistema de gestión de bases de datos PostgreSQL. Salirse de ahí supondría un coste demasiado elevado, tanto en términos económicos como de complejidad, para proyectos no experimentales. Con estas opciones, prácticamente todas, sino todas, las aplicaciones web actuales parecen viables. Apartarse de las opciones por defecto, supondría un incremento notable del riesgo, pero esto, en principio, no tiene que ser sinónimo de baja calidad o garantía de fracaso, incluso puede ser parte del camino al éxito.

Con respecto al software empleado en el *frontend*: la librería React resultó ser una opción de calidad contrastada e interesante, sobre todo el enfoque funcional del que hace gala. No obstante, no estaría de más que integrase alguna opción de enrutamiento, recordemos que react-router es un proyecto independiente, a la par que alguna opción, tipo Redux, para la gestión avanzada del estado. Es una librería que se queda algo escasa en esos aspectos.

Lo más negativo encontrado en el *frontend* es, con absoluta seguridad, los clientes, tanto el Apollo GraphQL, como el Phoenix.js para Phoenix *sockets*. Apollo es un cliente muy completo, pero presenta *bugs* ciertamente llamativos y no menores. Como casi siempre, su documentación no es para recrearse. El cliente Phoenix.js tampoco es de lo mejor: presenta un conjunto escaso de operaciones, el código fuente no está suficientemente documentado y, de nuevo, la documentación no es de lo más detallada.

6.5 Reflexiones metodológicas

El desarrollo desde un enfoque funcional parece encajar, perfectamente, con el uso de metodologías ágiles.

En este proyecto se ha empleado algo de UML aplicado al backend. De todas formas, hay que resaltar que no se ha encontrado absolutamente ninguna literatura, asociada al BEAM, en UML. No es algo de uso frecuente entre desarrolladores Erlang/Elixir. En este tipo de contextos suele abundar más el "arte de pizarra".

Como opinión personal, creo que se debería incentivar la realización de trabajos de fin de grado por equipos de desarrollo. En un trabajo individual, se pierde mucha capacidad de aprendizaje respecto a lo que se podría obtener en un enfoque por equipos. En un enfoque individual, la sección de metodología es casi anecdótica. Las habilidades que se podrían adquirir con los sistemas de control de versiones, no se adquieren, puesto que las operaciones se reducen a poco más que una sucesión de *commits* y algún *merge*. Se pierde la posibilidad de obtener sistemas probados y documentados intensivamente. En conclusión, creo que sería conveniente darle un par de vueltas a la normativa de forma que se fomente dicha forma de trabajar.

6.6 Reflexiones personales

Una de las grandes debilidades del software, actualmente y desde sus inicios, es, en general, la documentación. ¡Esa gran olvidada!. Durante este proyecto nos hemos encontrado con muchas librerías, ciertamente muy usadas y potentes, pero que dificultan en exceso su adopción inicial, precisamente por la dejadez en la documentación. Ausencia de documentación, incompletitud de la misma, ejemplos irrelevantes, poco didácticos o excesivamente simples. Lo peor no es el estado actual, lo realmente preocupante es que la situación no tiene visos de mejorar. Basta con echar un vistazo a sitios como GitHub y buscar proyectos realmente bien documentados. En poco tiempo nos daremos cuenta que el porcentaje de los mismos es totalmente insignificante en relación al total. El lenguaje Elixir es un proyecto, especialmente, bien documentado. Desafortunadamente, muchas de las librerías más usadas no mantienen ese nivel de calidad en la documentación.

En lo que se refiere al desarrollo de *frontends* hay que reconocer la escasa experiencia que se posee. Pero aún así, hemos de señalar que, pese a las muchas facilidades existentes hoy en día, sigue pareciendo demasiado caótico. A poco que crezca una aplicación, el número de dependencias se dispara. Cierto es que los gestores de paquetes, como npm o yarn, alivian algo esta situación. Pero se debe tener en cuenta los inconvenientes que surgen de hacer una aplicación dependiente de un número tan elevado de librerías. Reemplazar una librería cuyo

mantenimiento cese puede ser una gran aventura, corriendo el riesgo de caer en una cascada considerable de modificaciones. Actualmente, solo parecen existir dos caminos: bien dependencias excesivas, bien plataformas descomunales como Angular (a mi parecer en muchos casos excesivo).

Como recomendación para futuros trabajos, no recomendaría optar por proyectos tan ambiciosos, inicialmente, como ha sido este. El tiempo dedicado ha excedido, de forma desorbitada, el asignado oficialmente para el proyecto. De forma personal, actualmente, optaría por proyectos de pequeño tamaño, si se van a basar en tecnologías minoritarias, o proyectos de tamaño medio, si se emplean tecnologías de uso frecuente. De esta forma, sería posible obtener un resultado convenientemente documentado, probado y que luzca visualmente. De otra forma, probablemente, lo que se obtendrá será una base, quizá susceptible de una mayor calificación, pero mucho más compleja de evaluar, puesto que se requiere profundizar más a nivel interno y no quedarse, simplemente, en la fachada visible. Evaluar aplicaciones realizadas en tecnologías minoritarias requiere, más allá de la lectura de una memoria, un estudio previo del contexto actual de dichas tecnologías, con el fin de establecer unos criterios de evaluación justos respecto a los empleados para tecnologías de amplio uso. Los valores de tiempo, trabajo y esfuerzo difieren, ampliamente, entre un caso y otro.

Cuando se desarrolla, empleando tecnologías minoritarias, es frecuente encontrarse con software con una documentación muy deficiente. Con algo de suerte, en ocasiones, recurrir al código fuente puede llevar a la solución, ya que algún comentario en el mismo, o su análisis, pueden resolver nuestras dudas.

6.7 Lecciones aprendidas

Cada lección aprendida es una reafirmación de la infinita ignorancia del ser humano.

- La calidad del trabajo de un desarrollador es directamente proporcional a la calidad del entorno que le rodea y de las interacciones que efectúa.
- El enfoque funcional es mucho más natural y menos aburrido que la orientación a objetos.
- Si algo no puede salir mal, saldrá mal.
- La BEAM no es adecuada para trabajos CPU-intensivos. Para tareas que requieren mucha CPU es mejor enfocarse en otras alternativas.
- En la comunidad Elixir, el respeto y la educación representan el 99% del camino que hay que recorrer hacia la obtención de una solución, como mínimo factible, a cualquier problema.

- En la comunidad Elixir, son habituales los debates sobre una posible solución a un problema. En dichos debates se suele compartir, comparar y modificar código, el cual pasa a considerarse público para beneficio de la comunidad. Por tanto, lo que en ciertos ámbitos se considera "copiar", en este contexto se considera hacer uso de un recurso, generado de forma colaborativa, para el beneficio de la comunidad.

- El ecosistema Elixir está en constante evolución, con desarrolladores altamente cualificados, que proporcionan una fuente de innovación permanente, a la vez que proveen soluciones realmente simples y eficientes. Sin ser una comunidad excesivamente grande, como puede ser el caso del mundo Python, sí que está cuando menos a su mismo nivel, en cuanto a calidad e innovación se refiere.

- Tanto el creador del lenguaje, como los autores de los libros que versan sobre el ecosistema, son personas tremendamente accesibles. Es habitual encontrarlos resolviendo dudas de cualquier nivel en los canales oficiales habilitados para ello. Por tanto, en el éxito de un lenguaje también influye el nivel de accesibilidad de los principales referentes asociados al mismo.

- Los procesos gen_server son uno de los tipos de procesos más baratos, a la par que más útiles, en la problemática diaria del desarrollador Elixir/Erlang.

- En Elixir es frecuente que surga la necesidad de implementar una función por casos. Conviene comentar que una función de 50 ó 60 casos no se considera, en modo alguno, desproporcionada, sino todo lo contrario.

- Todo lo que es factible en Erlang, es también factible en Elixir, y viceversa. Un aspecto que se echa de menos en Elixir es el soporte simple para la declaración de constantes globales, teniendo que recurrir a macros para habilitar el soporte.

- Es realmente penoso que existiendo lenguajes como Erlang o Elixir, en el entorno empresarial sigan, en gran parte, anclados en dinosaurios como Java y compañía. Lenguajes en los que no voy a entrar a juzgar si son mejores o peores, pero sí que generan más código y más difícil de leer, al resultar menos natural. Después de transitar muchos años por el mundo Java, debo reconocer que el salto a Elixir ha supuesto un magnífico cambio para bien, muchísimo menos código y muchísima más semántica.

...y este fue el *bit* que colmó el *buffer*. Muchas gracias y ¡hasta pronto!

Apéndices

Apéndice A

Versionado semántico

El versionado semántico, *semantic versioning o semver*, se refiere a una especificación formal cuyo objetivo es que los números de versión, de nuestro software, resulten de utilidad en el manejo de dependencias.

El versionado semántico, por un lado, trata de evitar especificaciones de dependencias excesivamente estrictas, de forma que no se caiga en el bloqueo de dependencias, cuando queramos publicar una nueva versión de nuestro software. Por otro lado, también trata de que no hagamos previsiones demasiado optimistas en lo referente a la compatibilidad de nuestro software con futuras versiones de sus dependencias. Esto último se conoce como promiscuidad de versiones.

El versionado semántico propone un conjunto de reglas y requisitos que nos dicen cómo asignar e incrementar los números de versión de nuestros paquetes.

Cuando se usa este tipo de versionado, lo más importante es que el software debe declarar una API pública clara y precisa. Una vez se dispone de tal API, los cambios a la misma se comunican por medio de incrementos en el número de versión.

El número de versión sigue el formato "X.Y.Z", donde X, Y y Z son números enteros no negativos, que se incrementan de uno en uno. Las letras representan lo siguiente:

- X: Representa la versión *major*. La versión con "major" igual a cero se corresponde con el desarrollo inicial y la API pública no puede considerarse estable. La versión 1.0.0 define la API pública. La versión "major" debe ser incrementada si se introduce cualquier cambio, en la API pública, no compatible con la versión anterior. Puede incluir cambios de los niveles "minor" y "patch", pero ambas deben ser reinicializadas a cero cuando se incrementa la versión "major".

- Y: Representa la versión *minor*. La versión "minor" debe ser incrementada si se introduce una nueva funcionalidad compatible con la versión anterior. También se debe incrementar si cualquier funcionalidad de la API es marcada como deprecada. Puede ser

incrementada si se añade funcionalidad o correcciones de importancia al código privado. Puede incluir cambios del nivel Z (*patch*). La versión Z (*patch*) debe ser reinicializada a cero cuando la versión Y (*minor*) es incrementada.

- Z: Representa la versión *patch*. Debe incrementarse cuando se introducen correcciones compatibles con la versión anterior.

Una vez que un paquete versionado es liberado *release*, los contenidos de esa versión no deben ser modificados. Cualquier modificación ha de ser liberada como una nueva versión.

El nombre de una versión *pre-release* se forma añadiendo al formato de una versión normal un guión al final, seguido por una serie de identificadores separados por puntos. Solo se admiten caracteres ASCII alfanuméricos y el guión. Estas versiones tienen una menor precedencia que la versión normal.

Los metadatos de la "build" se pueden representar añadiendo un signo "+" y una serie de identificadores separados por puntos después de la versión "patch" o "pre-release". Los metadatos se ignoran a la hora de calcular la precedencia de versiones.

Para calcular la precedencia de versiones se separan el "major", el "minor", el "patch" y el "pre-release". Los "major", "minor" y "patch" se comparan, siempre, numéricamente. La precedencia del "pre-release" se determina comparando cada identificador separado por puntos de la siguiente manera:

- Los identificadores que son números se comparan numéricamente.
- Los identificadores que contienen letras o guiones se comparan siguiendo el orden establecido por ASCII.
- Los identificadores numéricos siempre tienen una menor precedencia que los no numéricos.

Ejemplos de "pre-release":

- 2.0.0-alpha
- 3.0.0-alpha.1
- 4.0.0-0.3.7
- 5.0.0-x.7.z.92

Ejemplos de metadatos:

- 7.0.0-alpha+001
- 7.0.0+20130313144700

- 7.0.0-beta+exp.sha.5114f85

Ejemplo de precedencia: 8.0.0-alpha < 8.0.0-alpha.1 < 8.0.0-beta.2 < 8.0.0-beta.11 < 8.0.0-rc.1 < 8.0.0

Apéndice B

Estado del arte

Actualmente, existe un número muy elevado de tecnologías asociadas al *frontend*, en el desarrollo web. Debido a la rápida evolución en este campo, muchas de ellas pasan a considerarse *legacy*, o directamente se descartan, en un período relativamente breve de tiempo. A pesar de que, en este proyecto, el *backend* tiene asociada una mayor prioridad, exploraremos algunas de las tecnologías más relevantes, actualmente, en el desarrollo de *frontend*s.

Se debe tener en cuenta que esta sección no pretende ser, en ningún caso, una descripción técnica, excesivamente detallada, de las tecnologías que en ella se citan. Para estos menesteres ya existen los pertinentes manuales asociados, junto con la documentación/guías/tutoriales *online*. Lo que aquí se pretende es dar una descripción muy general o, cuando proceda, explicar los conceptos más relevantes asociados a dichas tecnologías. La terminología asociada figura en el glosario de este documento (apéndice I).

B.1 *Frontend* y *backend*

Hoy en día existen múltiples significados vinculados a los términos *frontend* y *backend*, dependiendo de la disciplina en la que miremos. Por tanto, es de vital importancia dejar claro el significado al que nos referimos cuando hacemos uso de estos vocablos, ya que muchas definiciones no tendrían sentido alguno si fueran asociadas a este proyecto.

Desde el punto de vista de la arquitectura del software, el *frontend* es la parte de la aplicación con la que interacciona, directamente, el usuario final. Por tanto, se podría concebir como la parte "visible" de la aplicación. Respectivamente, el *backend* es la parte asociada a la gestión de lógica de negocio y datos, con la que el usuario solo puede interaccionar de forma indirecta.

Un número significativo de aplicaciones web suelen encuadrarse dentro de la arquitectura cliente-servidor, como es el caso de este trabajo. En este contexto, el *frontend* suele estar asociado con la parte del cliente, mientras que el *backend* se ubica en la parte del servidor.

B.2 Tecnologías *frontend* alternativas

A continuación vamos a citar y describir, brevemente, algunas de las tecnologías más importantes, en la actualidad, en el campo de desarrollo de *frontends*.

B.2.1 Vue.js

Vue [71] es un *framework* JavaScript progresivo enfocado a la construcción de interfaces de usuario. Está diseñado para ser adoptado de forma incremental. Esto es, escala entre librería y *framework*. Su *core* está enfocado solamente a la capa de la vista. Este *framework* es extensible mediante *plugins*. Por ejemplo, el *plugin* Vuex implementa el patrón Flux B.5 para Vue, lo cual permite manejar el estado de la aplicación. La aplicación resultante es un árbol jerárquico de componentes. Vue utiliza un *DOM virtual*, proporciona componentes de vista reactivos, que admiten composición, y solo soporta *one-way data flow*. Admite, opcionalmente, la sintaxis *JSX* y el lenguaje *TypeScript*.

B.2.2 Ember.js

Ember [72] es un *framework* JavaScript, para el *frontend*, basado en el patrón de arquitectura software *MVVM* B.4 y pensado para sitios web que requieran interacciones con el usuario complejas.

B.2.3 Riot JS

En este caso, estamos hablando de una microbiblioteca para IU [73], similar a React, que promete una curva de aprendizaje mínima, a la vez que una sintaxis agradable.

B.2.4 Elm

Al hablar de Elm [74] nos estamos refiriendo a un lenguaje de programación funcional, que compila a JavaScript, que se puede considerar como una opción en el desarrollo de *frontend*s. Hace hincapié en la usabilidad, rendimiento y robustez. Entre sus principales características están el tipado estático, la inmutabilidad, la modularidad y la interoperabilidad con HTML, CSS y JavaScript. Este lenguaje promete la no existencia de errores en tiempo de ejecución.

B.2.5 Semantic-UI

Semantic-UI [75] es un *framework* de interfaz de usuario diseñado para *theming*. Posee un número respetable de elementos de interfaz de usuario y está diseñado para ser responsivo.

Puede integrarse con otros *frameworks*, como React, Angular, Meteor o Ember, para organizar la capa de interfaz de usuario. Emplea una disposición de componentes semejante a CSS Flexbox.

B.2.6 Foundation

Se trata de un *framework* [76] de interfaz de usuario adaptativo. Los *frontends* que lo usan trabajan bien tanto en dispositivos grandes como pequeños, puesto que usa CSS *media queries*. Contiene los patrones más comunes para maquetar un sitio adaptativo. El diseño de las páginas se ajusta dinámicamente según el dispositivo utilizado. Adopta un enfoque *mobile-first*, que consiste en diseñar y desarrollar primero para móviles y mejorar, posteriormente, para pantallas grandes.

B.2.7 Materialize CSS

Materialize [77] se puede describir como un *framework* CSS que sigue las líneas de diseño de *Material Design* de Google [66]. Contiene un buen número de interfaces, listas para utilizar, junto con código JavaScript asociado.

B.2.8 UIKit

UIkit [78]es un *framework* pensado para desarrollar interfaces web en el *frontend*. Se trata de un *framework* ligero y modular. Soporta una gran cantidad de navegadores, desde IE11 hasta la última versión de Google Chrome. Los estilos están escritos en LESS, un lenguaje dinámico de hojas de estilo. Puede ser usado, fácilmente, en conjunción con *frameworks* JavaScript como Vue.js.

B.2.9 Polymer

En el caso de Polymer [79], estamos hablando de una librería JavaScript, para la construcción de aplicaciones web, que usa *Web Components*. *Web Components* se puede definir como un conjunto de características que proporciona un modelo de componente estándar para la web y que permite la encapsulación e interoperabilidad de los elementos HTML individuales. Por tanto, hace posible crear y compartir elementos personalizados, que funcionan en cualquier sitio, y se integran bien con los diversos *frameworks* existentes. *Web Components* se sustenta en los siguientes estándares web:

- *Custom elements*: Permite diseñar y usar nuevos tipos de elementos DOM [80].
- *Shadow DOM*: Define cómo usar estilos encapsulados y marcado en *web components* [81].

- Especificación de módulos ECMAScript 6: Define la inclusión y reutilización de documentos JavaScript de forma eficiente, modular y basada en estándares [82].

- *HTML Template*: Define cómo declarar fragmentos de marcado, que no se usan durante la carga de una página, pero que pueden ser instanciados más tarde, en tiempo de ejecución [83].

B.2.10 Material-UI

Material-UI [84] es una librería que nos proporciona un conjunto de componentes React, que nos permiten construir una interfaz gráfica de usuario conforme a las especificaciones Material Design [66] de Google.

Esta librería fue una seria candidata a ser incluida en el proyecto. Inicialmente, en el análisis previo, se descartó por su mal soporte de las áreas de texto. En un análisis posterior, se volvió a descartar por su mala documentación: necesita ampliar sensiblemente el nivel de detalle e incluir mayor variedad de ejemplos didácticos y fáciles de entender. La calidad de la librería parece excepcional y, para un desarrollador experimentado, seguramente sea una buena opción. No obstante, es conveniente tener cuidado con este tipo de software cuya documentación flojea, ya que puede suponer riesgos importantes para el futuro, sobre todo a la hora de que otros puedan entender la implementación.

B.2.11 MDC React

MDC React [85] es la implementación oficial de los componentes Material Design [66] para React [45]. En su momento, se consideró como una de las posibles alternativas a escoger, para este proyecto. En un análisis más profundo, se detectó que su implementación, en ese momento, estaba sometida a cambios drásticos, lo que hizo su uso completamente inviable.

En la actualidad, la librería es un *wrapper* para *Material Components for the web* (MDC Web) [86].

B.3 Angular 2+

En esta sección describiremos la alternativa estudiada más en serio, junto con React, como posible tecnología candidata para el desarrollo del *frontend*. Esta opción, en concreto, fue finalmente rechazada; principalmente por su pronunciada curva de aprendizaje, comparada con la de la alternativa React analizada. El otro motivo de descarte es la querencia del autor por opciones que no hacen uso de *templates*. Esto no quita que Angular 2+ sea una de las plataformas más completas para el desarrollo de aplicaciones de página única (*SPAs*) en el *frontend*.

De hecho, si se quieren evitar problemas de dependencias y disfrutar de una documentación de altísima calidad, debería plantearse como una de las principales candidatas.

Angular es una plataforma de desarrollo que permite la construcción de aplicaciones web, tanto móviles como de escritorio. Se trata de un proyecto de código abierto, dirigido por Google.

B.3.1 ¿*Framework* o plataforma?

Actualmente, es fácil encontrar referencias a Angular bien como *framework*, bien como plataforma, bien como ambas cosas a la vez. Al utilizar esta última acepción, se suele considerar que se trata de una solución completa, para la creación de aplicaciones cliente, incluyendo todo lo necesario, tanto para el desarrollo como para el despliegue. Precisamente, esto es una de las ventajas de Angular sobre soluciones basadas en React, por ejemplo, donde estas últimas pueden depender de software de terceros, mantenimiento incluido, para proporcionar ciertas características, como pueden ser el enrutamiento o la inyección de dependencias.

B.3.2 Angular y AngularJS

En contra de lo que pueda parecer, Angular, poco o nada tiene que ver con su predecesor, AngularJS. Angular, inicialmente conocido como Angular 2, se trata de una reescritura completa de AngularJS. Desde la versión 2, la arquitectura permanece, prácticamente, inalterada.

B.3.3 Lenguajes de desarrollo

Para desarrollar una aplicación en Angular, el lenguaje recomendado a utilizar es TypeScript. De hecho, el propio Angular está escrito en dicho lenguaje. Sin embargo, existen otras alternativas en lo referente a la elección del lenguaje. Dos de las más extendidas son JavaScript y Dart.

B.3.4 Conceptos fundamentales

Sistema de módulos

Un módulo, en Angular, puede pensarse como un paquete que implementa cierta funcionalidad del dominio de negocio de la aplicación. Por tanto, es una práctica común que, una vez identificadas las distintas áreas funcionales de la aplicación, las mismas sean distribuidas en diferentes módulos. Desde otro punto de vista, un módulo puede verse como un contenedor que agrupa componentes, servicios, directivas y *pipes* relacionados de alguna manera (un *workflow*, una colección de utilidades, ...).

Un módulo Angular se denomina *NgModule*. Toda aplicación tiene, como mínimo, un módulo conocido como *root module*, utilizado en el inicio de la aplicación. El resto de módulos se catalogan como *feature modules*. Los *feature modules* dan lugar a una característica importante, proporcionada por Angular, denominada *carga perezosa*, y directamente asociada al patrón de diseño *Lazy loading*. Esto es, ciertos módulos solo se cargarán cuando se necesiten, lo que contribuye a aumentar la eficiencia y concentrarnos en una cantidad menor de código. La carga perezosa es opcional, existiendo, por tanto, su contrapartida, la *precarga*.

Existen cinco tipos de *feature modules*:

1. *Domain*: Son aquellos que proporcional una experiencia de usuario dedicada a un dominio particular de la aplicación. Por ejemplo, editar un cliente.

2. *Routed*: Son módulos cuyos componentes de nivel superior son el objetivo de las rutas de navegación del *router*. Todos los módulos objeto de carga perezosa son de este tipo.

3. *Routing*: Estos módulos posibilitan la configuración de enrutamiento para otro módulo.

4. *Service*: Suministran servicios.

5. *Widget*: Hacen componentes, directivas y *pipes* disponibles para módulos externos.

Un *NgModule* es una clase decorada con `@NgModule`. El decorador no es sino una función cuya entrada es un objeto con las propiedades que describen el módulo. Es decir, información sobre lo que hay declarado en el módulo, lo que importa/exporta, al igual que los servicios que suministra. Además de configurar un inyector de dependencias, cada módulo hace lo propio con el compilador.

Desde el punto de vista de Angular, las librerías son simples *NgModules*.

Directivas

Estrictamente hablando, una directiva no es más que una clase decorada con `@Directive`. Pero, quizá, lo verdaderamente interesante, es que permiten añadir comportamiento personalizado a los elementos en el DOM. Así, un uso frecuente, es usarlas para extender HTML.

En Angular, existen tres clases de directivas diferentes:

1. Componentes: Son directivas con una plantilla HTML. Un componente gestiona una región de HTML en forma de un elemento HTML nativo.

2. Estructurales: Modifican la estructura del DOM, añadiendo, manipulando o eliminando elementos del DOM. Cambian la estructura de la vista. Son las responsables de la disposición HTML. Son fáciles de reconocer, puesto que su nombre está precedido por un "*".

3. De atributo: Cambian la apariencia o comportamiento de un elemento, componente o de otra directiva. Se usan como atributos de elementos.

Plantillas

Una plantilla es una forma de HTML que informa a Angular sobre cómo renderizar un componente. Una plantilla define la vista de un componente.

Es común que las vistas se organicen de forma jerárquica. La plantilla asociada con un componente define la *host view*. El componente puede definir una jerarquía de vistas, que contenga vistas embebidas, alojadas por otros componentes.

Componentes

En Angular, un componente se puede pensar de diferentes formas. Es posible concebirlo como una directiva con una plantilla o como una clase, decorada con `@Component` y que contiene la lógica de aplicación, que junto a la plantilla asociada describen una vista (UI).

Llegados a este punto, conviene aclarar que un componente siempre tiene una vista asociada; una directiva es posible que tenga una vista asociada, pero no necesariamente.

Cuando hablamos de componentes, es útil recurrir a los patrones de arquitectura MVC (*Model-View-Controller*) o MVVM (*Model-View-ViewModel*). La clase que conforma el componente hace el rol de *controller/viewmodel*, mientras que la plantilla asociada representa la vista.

Un componente tiene un ciclo de vida. Angular suministra funciones asociadas con las distintas fases del ciclo, permitiendo así configurar el comportamiento del componente en cada fase de interés.

Servicios

Un servicio suministra datos o lógica que no se asocia con ninguna vista, pero que queremos compartir. Un servicio es una clase decorada con `@Injectable`.

Un componente puede utilizar un servicio, al igual que un servicio puede hacer uso de otro servicio. Los servicios se inyectan siguiendo el patrón de diseño "Inyección de dependencias", que en Angular está soportado por el *DI Framework*. Se denominan dependencias a los servicios u objetos que una clase necesita. Dicha clase, en lugar de crear ella misma tales dependencias, lo que hace es preguntar por ellas al inyector. Con esto se consigue aumentar la eficiencia, flexibilidad, robustez y facilidad de prueba de las aplicaciones, al tiempo que se simplifica su mantenimiento.

Es crítico comprender que existe una jerarquía de inyectores. Primeramente, hay un inyector, a nivel de plataforma, compartido por todas las aplicaciones en ejecución. También

existe un inyector a nivel de aplicación y más abajo, dentro de cada módulo, inyectores a nivel de directiva. Comprender esto es fundamental para determinar el ámbito y el tiempo de vida del servicio.

Pipes

Una *pipe* es una clase, decorada con `@Pipe` que define una función que transforma valores de entrada en valores de salida que se mostrarán en una vista.

Angular Elements

Se denominan *Angular Elements* a componentes Angular que están empaquetados como *custom elements*. Esto último es una característica del estándar *Web Components* que permite definir nuevos elementos HTML de forma *framework-agnostic*. Un *custom element* extiende HTML de forma que permite definir una etiqueta cuyo contenido es creado y controlado por código JavaScript. Existe un mapeo de una clase JavaScript instanciable a una etiqueta HTML. El estándar en cuestión está soportado por los principales navegadores del mercado.

Actualmente, se está trabajando para que los *custom elements* puedan ser usados por aplicaciones web construidas con otros *frameworks* distintos de Angular. Para lograr esto, una versión mínima y autocontenida de Angular debe ser inyectada como un servicio.

Service workers

Un *service worker* es un *script* que ejecuta en el navegador y gestiona el *caching* para una aplicación.

A la hora de conseguir hacer una aplicación web progresiva (PWA), es común hacer uso de *service workers*.

El funcionamiento de los *service workers* es igual que el de un *proxy* de red. La idea central es que interceptan todas las solicitudes HTTP, hechas por la aplicación, y deciden cómo responderlas. El *caching* es completamente programable.

Un *service worker* puede cargar una aplicación eliminando por completo o reduciendo la necesidad de acceder a la red. Esto probablemente suponga una mejora en la experiencia de usuario, en especial cuando tratamos con redes poco fiables o lentas.

Angular implementa *services workers* desde la versión 5. El servidor que trae Angular por defecto, `ng serve`, no soporta *service workers*, aunque es fácil utilizar otros como `http-server`, disponible bajo el gestor de paquetes `npm`.

B.3.5 Arquitectura de Angular

La arquitectura software actual de Angular se resume en la figura B.1. En la figura se aprecia cómo un componente, junto con una plantilla asociada, definen una vista.

El decorador, que figura en la clase componente, añade los metadatos referentes al mismo, incluido un puntero a la plantilla (*template*) asociada.

En Angular, el *binding markup* conecta los datos de la aplicación y el DOM. En esta plataforma existen dos tipos de *data binding*:

- *Event binding*: Permite a la aplicación responder a las entradas del usuario actualizando los datos de la aplicación.
- *Property binding*: Permite interpolar valores, calculados a partir de los datos de la aplicación, en el HTML.

Las directivas proporcionan "lógica de programa" (Ej. *ngFor, *ngIf, etc.).

Las directivas y el *binding markup*, en la plantilla asociada al componente, modifican las vistas en función de la lógica y los datos del programa.

Finalmente, en la figura se observa como el inyector de dependencias suministra los servicios requeridos por el componente.

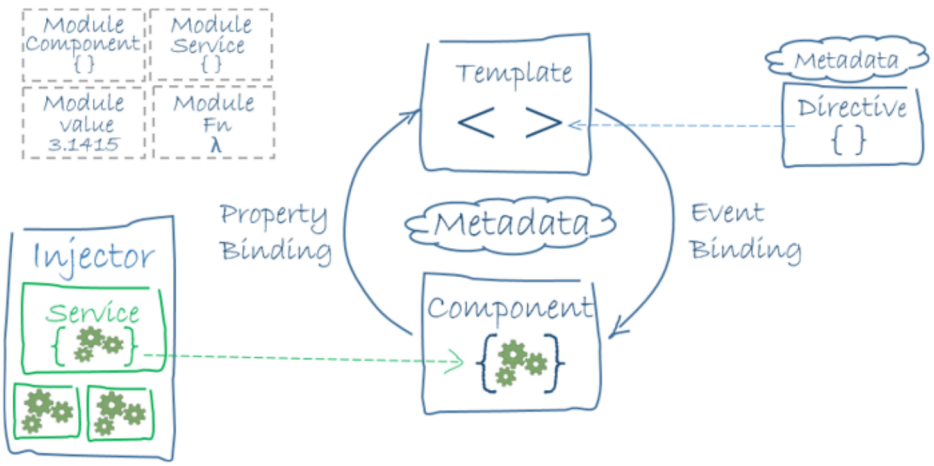

Figura B.1: Arquitectura de Angular [1]

B.3.6 Angular CLI

Angular CLI es la herramienta oficial de interfaz de línea de comandos de Angular. Permite crear, inicializar, desarrollar y mantener aplicaciones Angular, al igual que generar diversas

plantillas que sirven como punto de partida para la implementación.

B.3.7 Pruebas en Angular

Existen multitud de *frameworks* de pruebas para Angular. Como es imposible abarcarlos todos, nos centraremos en los dos, quizá, más conocidos: Jasmine y Protractor.

Jasmine es un *Behavior Driven Development Framework* que nos permite seguir un proceso de desarrollo guiado por comportamiento. Este *framework* posibilita escribir pruebas de unidad, en lenguaje natural, que describan lo que el software debería hacer. El uso del lenguaje natural facilita la comprensión de las pruebas con tan solo leerlas, al mismo tiempo que las habilita como documentación.

Jasmine se suele utilizar con un *test runner*, basado en línea de comandos, llamado `Karma`. Esta herramienta prueba código JavaScript, aunque en el caso de Angular las pruebas se suelen escribir en TypeScript. Si se usa *Angular CLI* para gestionar el proyecto, el uso de Jasmine/Karma es muy sencillo, ya que todo está prácticamente configurado. En otro caso, la configuración puede resultar algo más laboriosa, ya que se debe proceder de forma manual.

Cuando hablamos de *Behavior Driven Development* (BDD) debemos tener claros los conceptos a utilizar. Un caso de prueba se llama *spec*. Una *suite* se define como la combinación de una o más *specs*. La *suite* se define con la función *describe()*. Cada *spec* se programa como una función *it()*, que define el comportamiento esperado y cómo realizar la prueba. Una *assertion* es una forma de preguntar si cierta expresión, en una prueba, es cierta o falsa.

Angular contiene una librería de pruebas que incluye algunos *wrappers* o envoltorios para Jasmine.

Por otro lado, las pruebas E2E (*End-to-End*) se realizan con Protractor. En este tipo de pruebas se pretende probar el flujo de trabajo simulando la interacción de la aplicación con el usuario.

La librería Protractor simula las acciones del usuario, sin necesidad de interacción humana. Protractor está basada en *Selenium WebDriver*, en concreto utiliza *WebDriverJS*, que permite manejar el navegador mediante *scripts*. En esta librería se incluye una API, específica para Angular, que permite localizar los distintos elementos de la interfaz de usuario. Si el proyecto ha sido creado con *Angular CLI*, Protractor estará ya disponible sin mayor esfuerzo, de idéntica forma a lo que sucedía con Jasmine.

B.3.8 Material Design For Angular

Material Design For Angular es una librería de componentes de interfaz de usuario, que siguen la especificación de Material Design [66], contruidos con y para Angular, por el equipo de Angular. Estos componentes prometen un código de calidad, soportan accesibilidad e

internacionalización, fueron sometidos a pruebas de unidad y de integración, y están optimizados.

B.4 Bootstrap

Bootstrap [57, 87] es un *framework* CSS para desarrollo de *frontend*s, exclusivamente. Contiene elementos para la interfaz de usuario basados en HTML y CSS, tipografías, ..., así como extensiones JavaScript. Las versiones más recientes utilizan Sass para las hojas de estilo. Permite realizar proyectos con un enfoque *mobile-first* y responsivos.

B.5 React

React [45, 88, 89, 90] es una librería que facilita el desarrollo de *interfaces gráficas de usuario* (GUIs). Cabe destacar que se trata de una pequeña librería, y puesto que carece de las herramientas que serían de esperar en lo que clásicamente se entiende por *framework*, en ningún caso puede considerarse una solución completa para el desarrollo *frontend*. Caso aparte sería si hablásemos de *React stack*.

React nació con el objetivo de facilitar el manejo de interfaces de usuario complejas, en *frontends*, y uno de los principales problemas de los que se ocupa es de cómo gestionar los cambios en las vistas en respuesta a cambios en los datos. La librería posee un estilo declarativo, mediante el cual los desarrolladores escriben *qué hacer* y nunca *cómo hacerlo paso a paso* (estilo imperativo). La principal baza para elegir un estilo declarativo es que, para un humano, el código así escrito es más fácil de leer y comprender.

React es una librería muy rápida porque abraza el hecho de que generar elementos en memoria es muy rápido. La mayor pérdida de velocidad se produciría al renderizar en el DOM. Afortunadamente, encontraron la solución en un buen algoritmo.

La arquitectura de React es una arquitectura basada en componentes [91], que fomenta un bajo acoplamiento y la reutilización de código.

Esta librería diferencia entre elementos y componentes:

- Elementos React B.1: Son los bloques, esenciales, de construcción de una aplicación React. No se debe confundir, en ningún momento, los conceptos de elemento y componente. Un elemento describe lo que se ve en la pantalla, es inmutable y no es habitual que sea usado directamente. Lo más normal es que los elementos sean retornados desde los componentes.

- Componentes React B.2: Son pequeños trozos reutilizables de código, que retornan un elemento React que será renderizado en la página.

```
const element = <h1>Segmentation fault (coredumped)</h1>;
```

Listing B.1: Elemento React

```
function ParseError(props) {
  return <h1>Parse error before {props.error}</h1>;
}
```

Listing B.2: Componente React

React considera las interfaces de usuario como funciones. Existen dos tipos de componentes en React: con estado y sin estado. Los componentes sin estado se escriben como simples funciones JavaScript, que retornan un elemento React. Los componentes con estado se escriben como clases JavaScript. De todas formas, no debemos olvidarnos que, hasta ECMAScript 6, no existían clases, y estas eran implementadas como funciones. Esto es, las clases en JavaScript pueden verse casi como azúcar sintáctico o un guiño a la orientación a objetos.

Esta librería se aleja de la clásica separación de datos, estilo (CSS), estructura (HTML) e interacciones dinámicas (JavaScript). En su lugar, adopta un enfoque basado en la construcción de componentes de interfaz de usuario, reusables y escritos en JavaScript. Como es obvio, cada componente ha de estar ligado a una funcionalidad. La idea central es construir una GUI mediante la composición de componentes (bloques de funcionalidad autocontenidos). Este enfoque permite crear GUIs que son fáciles de mantener y de extender. Actualmente, no tiene mucho sentido separar HTML de JavaScript, puesto que están íntimamente relacionados, sobre todo en el caso de aplicaciones de página única.

Un componente React es el caso más claro de *one-way data flow*. El componente siempre recibe unos determinados datos de entrada y retorna lo que le diga su método `render()` que hay que mostrar. Para las mismas entradas, debería retornar las mismas salidas. Los datos de entrada se introducen en el componente vía *props* y, precisamente por medio de estas *props*, pueden ser accedidos desde la función `render()`. Lo que debería quedar claro es que lo que React denomina *props* no es otra cosa que las entradas arbitrarias que recibe un componente. Las *props* son datos, de solo lectura, pasados desde un componente padre a un componente hijo. Cabe destacar que, al afirmar que son de solo lectura, estamos indicando que no deben ser modificadas en modo alguno. Las propiedades susceptibles de modificación deberían estar ubicadas en el estado del componente (*this.state*), gestionado por él mismo. Finalmente, comentar que *props.children* alberga todo el contenido situado entre la etiqueta de apertura y cierre de un componente.

Una interfaz de usuario React es un árbol jerárquico de componentes. Los padres pueden comunicar su estado a los hijos haciendo uso de las *props*, mencionadas anteriormente. Pero los hijos también pueden comunicarse con los padres por medio de *callbacks*. Un padre puede

pasar a un hijo una función mediante una *prop*. Cuando el hijo invoque dicha función, el padre recibirá el valor de retorno.

Lo normal en una aplicación React es que los datos se pasen de padre a hijo, como acabamos de describir, a través de *props*. No obstante, existe una forma de saltarse esta limitación, denominada *Context*. *Context* nos permite que los datos circulen a través del árbol de componentes sin pasar como *props* por cada nivel. *Context* solo es adecuado para datos que se consideran globales al árbol de componentes, pongamos, por ejemplo, el tema de la interfaz de usuario.

Además de los componentes "estándar", existen otro tipo de componentes, denominados *componentes de orden superior* o *HOC*. Un componente "estándar" recibe unas determinadas *props* y retorna interfaz de usuario. Un *HOC* es una función que recibe un componente y devuelve otro componente. Por tanto, los componentes de orden superior son un patrón, que surge de la naturaleza composicional de React, y que permiten reutilizar lógica.

Todo componente React tiene un ciclo de vida. La librería nos proporciona un conjunto de métodos que podemos reescribir para establecer lo que pasa en cada fase del ciclo. Esto es, ¿qué pasa cuando el componente se carga? ¿qué se hace cuando se descarga?, etc.

Cuando usamos React, también es frecuente recurrir a la sintaxis JSX. Aunque no es para nada obligatorio. Esto no es otra cosa que escribir JavaScript como HTML.

Una de las características que más resaltan de React es su uso de un *DOM virtual*. La librería emplea este DOM virtual para encontrar las diferencias entre lo que se muestra, actualmente, en el navegador y la nueva vista. Estas diferencias se denominan *delta* y el proceso se llama bien *DOM diffing*, bien *Reconciliación entre el estado y la vista* (c.f. B.2) [92][93]. Y ... ¿qué ventaja tiene esto? Pues, fundamentalmente, que el desarrollador únicamente tiene que preocuparse de actualizar el estado de sus componentes. Luego, React ya se ocupará de actualizar la vista de la forma más conveniente. Cabe destacar que, como buen algoritmo de *diff* que implementa, la librería solo actualizará las partes de la UI que procedan.

Otra característica destacable es la abstracción, que hace la librería, del DOM, proporcionando propiedades y eventos sintéticos. Esto es, no se trabaja con eventos nativos. En su lugar, se utilizan eventos sintéticos que envuelven a los eventos nativos. La ventaja de esta solución es que obtenemos el mismo comportamiento independientemente del navegador en el que se ejecute la aplicación.

Finalmente, comentaremos que en React todo componente admite el atributo especial *Ref*. Este atributo permite tener acceso directo al elemento del DOM o a la instancia del componente, según proceda.

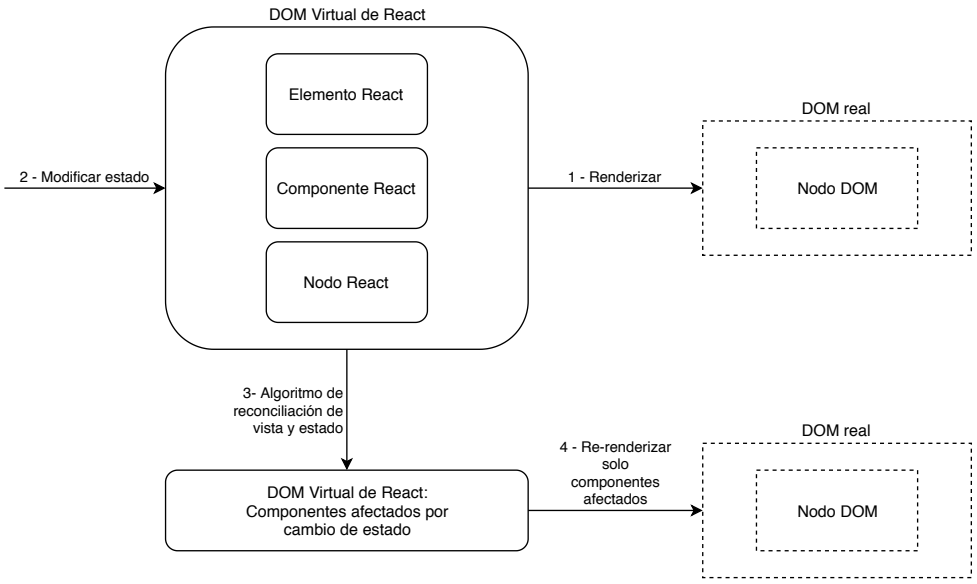

Figura B.2: Funcionamiento del DOM virtual de React

B.6 Arquitectura de datos en el Frontend

B.6.1 El patrón Modelo-Vista-Controlador

El patrón Modelo-Vista-Controlador [94], desde ahora MVC, es un patrón de arquitectura del software empleado, intensivamente, en el desarrollo de interfaces gráficas de usuario.

MVC separa las representaciones internas de información de la forma en que esta última es presentada al usuario o suministrada por el mismo.

Componentes del patrón Modelo-Vista-Controlador

El patrón MVC (c.f. B.3), como su propio nombre indica, divide la aplicación (o *frontend*) en tres partes interconectadas: modelo, vista y controlador.

El *modelo* es la representación de la información con la que el sistema opera. Este componente es el responsable de gestionar los datos de la aplicación. En su operativa, envía al componente *vista* la información que se le solicita en cada momento. También entra dentro de sus funciones atender las solicitudes de obtención/manipulación de información recibidas desde el componente *controlador*.

El *controlador* es un componente que acepta entradas y las transforma en comandos para el modelo o la vista. En cierta forma, realiza una labor de intermediación entre la vista y el modelo. Este componente responde a eventos, como pueden ser el caso de acciones del usuario, realizando solicitudes de operaciones sobre la información al modelo o enviando comandos a su vista asociada.

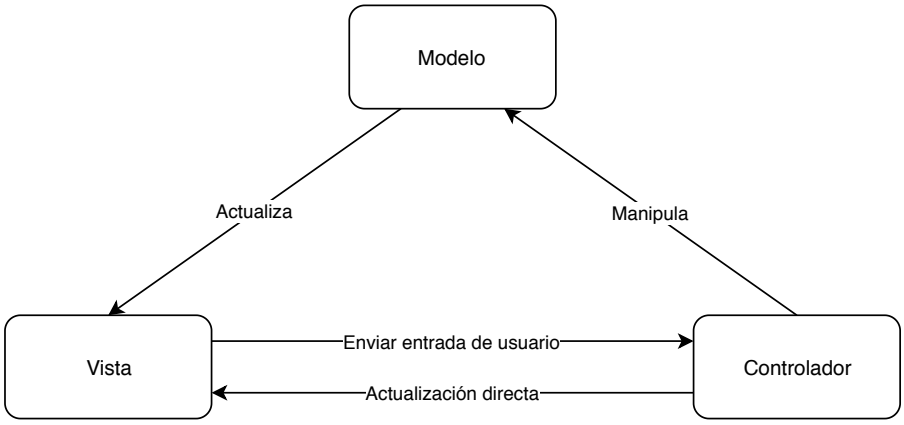

Figura B.3: Patrón de arquitectura Modelo-Vista-Controlador

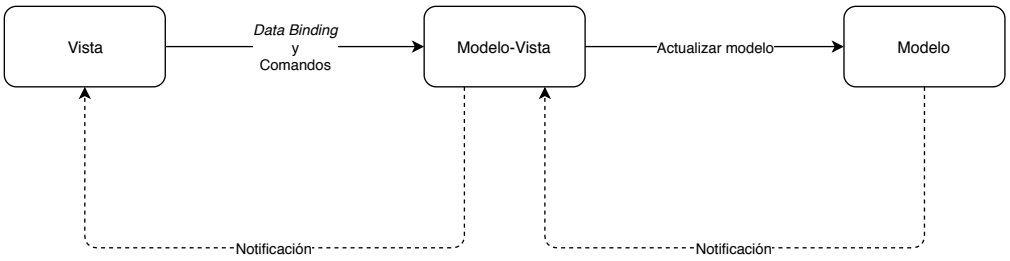

Figura B.4: Patrón Modelo-Vista-ModeloVista

El componente *vista* se encarga de presentar la información obtenida desde el componente *modelo* en un formato específico.

Ventajas del patrón Modelo-Vista-Controlador

El patrón MVC, al dividir la aplicación en componentes, está favoreciendo la reutilización del código, al mismo tiempo que permite trabajar, de forma simultánea, en el desarrollo de cada componente. La separación propuesta por el patrón, como es evidente, favorece el bajo acoplamiento.

B.6.2 El patrón Modelo-Vista-ModeloVista

Este patrón (c.f. B.4) es una evolución del patrón MVC clásico. Su objetivo principal es separar la lógica de negocio y la lógica de presentación de la interfaz de usuario. Este patrón es empleado, actualmente, de forma mucho más frecuente que el MVC, en lo que se refiere a arquitectura de *frontends*.

Tiene tres componentes diferenciados:

- Modelo: Se refiere a lógica de negocio o a la capa de acceso a datos.

- Vista: Muestra una representación del modelo y recibe la interacción del usuario con la vista y redirige dicha interacción para que sea manejada por el componente Modelo-Vista.
- Modelo-Vista: Maneja la lógica de presentación.

El componente Vista conoce al componente Modelo-Vista y, este último, conoce al componente Modelo. El Modelo no es consciente de que existe el Modelo-Vista y el Modelo-Vista tampoco es consciente de la Vista. Por tanto, el Modelo-Vista actúa de intermediario entre la Vista y el Modelo.

B.6.3 Arquitectura basada en componentes

Una de las arquitecturas usadas habitualmente, hoy en día, por las diversas plataformas/*frameworks* de desarrollo para el *frontend*, es la denominada "arquitectura basada en componentes". Este tipo de arquitectura se enfoca en descomponer el sistema en componentes funcionales o lógicos, que exponen interfaces de comunicación bien definidas. Es el caso, por ejemplo, de Angular 2+, que presenta una arquitectura basada en componentes los cuales, a su vez, tienen una arquitectura similar a MVC (c.f. B.3).

Dicho esto, no es demasiado interesante hablar de la arquitectura de la plataforma usada. Suele ser bastante más importante enfocarse en la propia arquitectura elegida para el *frontend*, como puede ser una arquitectura tipo MVVM o Flux. Dicho de otro modo, en este trabajo se considera más prioritario describir la arquitectura de un *frontend* que utiliza, por ejemplo, React, que describir la propia arquitectura de React.

B.6.4 Data binding en interfaces de usuario

En el mundo de las interfaces de usuario, el término *data binding* se refiere a la forma de asociar los campos de la interfaz de usuario a un modelo de datos. Actualmente, existen dos tipos principales de *data binding*: "One way data flow" y "Two way data binding".

One way data flow

El concepto de flujo de datos unidireccional (*one-way data flow*) abraza la idea de que los datos, en el *frontend*, solo pueden fluir en una única dirección. De esta forma, la vista nunca actualiza el modelo. Lo único que puede hacer la vista es disparar una acción, que representa la intención de actualizar el *store*, que es el lugar donde se alojan los datos de la aplicación. Pero, en última instancia, es el *store* quien decide cómo gestionar la acción.

Como se puede observar, este concepto permite llevar a cabo razonamientos, sobre el flujo de datos, de una manera muy sencilla e intuitiva. Echando un ojo a la vista, se puede

determinar, fácilmente, el conjunto de acciones que pueden ser disparadas. Analizando cómo se gestiona cada acción, en el *store*, podemos completar el razonamiento sobre el flujo de datos.

Two way data binding

Two way data binding se basa en la idea de que cualquier cambio en un campo de la interfaz de usuario actualiza el modelo, y viceversa. Este concepto parece, a primera vista, muy atractivo. Desafortunadamente, a medida que la aplicación crece, nos puede llevar a múltiples efectos laterales no deseados y una gran dificultad de depuración: un modelo podría actualizar una vista, que a su vez podría actualizar otro modelo, que a su vez podría actualizar otro modelo, y así entrar en una cascada de actualizaciones muy difícil de seguir.

Otro inconveniente que presenta este concepto es la excesiva dependencia que se crea entre el modelo y la vista, cuando se aplica.

B.6.5 El patrón Flux

Flux es un patrón de arquitectura, creado por Facebook, para el manejo del flujo de datos en el lado cliente (*frontend*) de las aplicaciones web. El eje en torno al cual gira este patrón y, al mismo tiempo, el concepto más importante, es el *flujo de datos unidireccional* (*one-way data flow*), descrito anteriormente. Este patrón fue desarrollado para reemplazar el clásico MVC en favor del citado flujo de datos unidireccional.

Partes clave de Flux

Flux tiene cuatro sencillas partes que conforman la clave de su funcionamiento. Estas partes son:

- Acciones: Las acciones definen la API interna de la aplicación. Son el modo que tiene Flux de capturar las posibles formas en las que se puede interactuar con la aplicación. Una acción expresa un deseo de interacción, con la aplicación, por parte de quien la dispara. Una acción no es más que un simple objeto con un campo "tipo" y otro para datos *payload*. Las acciones deben tener semántica y ser descriptivas, obviando detalles de implementación.

- *Stores*: Los *stores* alojan los datos de una aplicación y, dicha aplicación puede tener múltiples *stores*. Un *store* debe registrarse con el *dispatcher* para poder comenzar a recibir acciones. Es muy importante comentar que los datos, en un *store*, solo pueden modificarse en respuesta a una acción. No existen métodos *set* públicos en un *store*, solo métodos *get*. Un *store* decide a qué acciones desea responder. Cada vez que un *store* cambia, recordemos que como respuesta a una acción, debe emitir un evento de cambio.

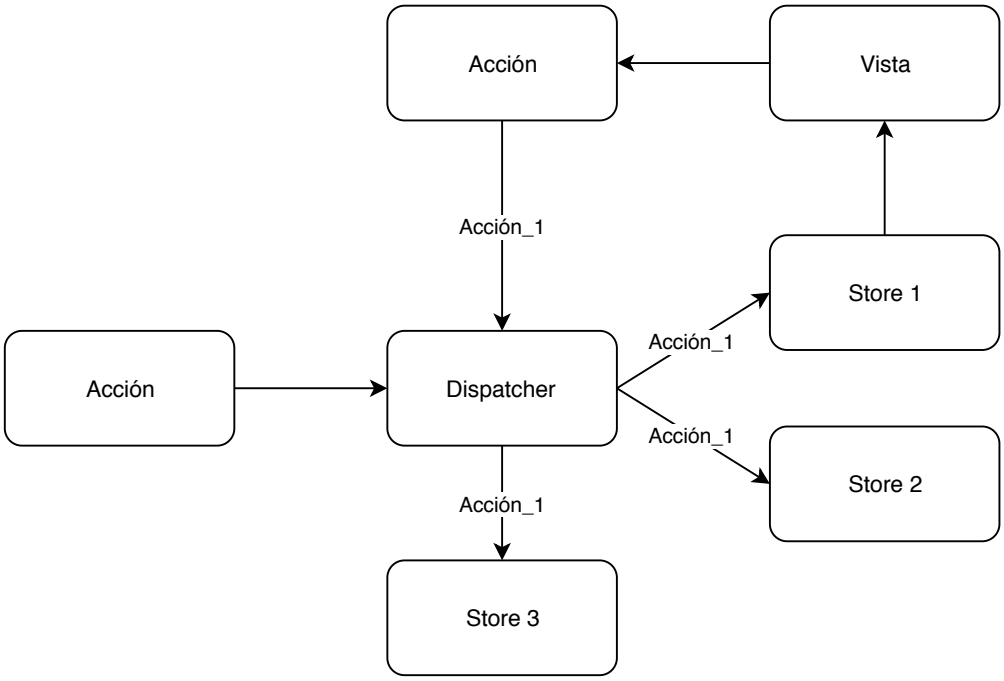

Figura B.5: Flujo de datos unidireccional típico en arquitectura Flux

- *Dispatcher*: El *dispatcher* es un componente que recibe acciones y las dispensa a los *stores* que tiene registrados. Debe haber un único *dispatcher* (*singleton*) en cada aplicación. Toda acción dispensada será recibida por todos los *stores*. Son estos últimos los encargados de decidir qué acciones atienden.

- Vistas: Los datos de los *stores* se muestran en vistas. Cuando una vista usa los datos de un *store*, debe suscribirse a los eventos de cambio que emite dicho *store*. Así, cuando los datos del *store* cambian, como consecuencia de una acción, la vista puede actualizarse en consecuencia. Las acciones suelen emitirse desde las vistas, cuando el usuario interactúa con la interfaz de usuario, pero esto no es, ni mucho menos, obligatorio.

Flujo de datos típico de Flux

A continuación se describen los pasos de un flujo de datos típico en la arquitectura Flux (c.f. B.5):

- La vista envía acciones al *dispatcher*.

- El *dispatcher* envía las acciones a todos los *stores* que tiene registrados.

- Los *stores* que deciden atender las acciones, envían datos a las vistas.

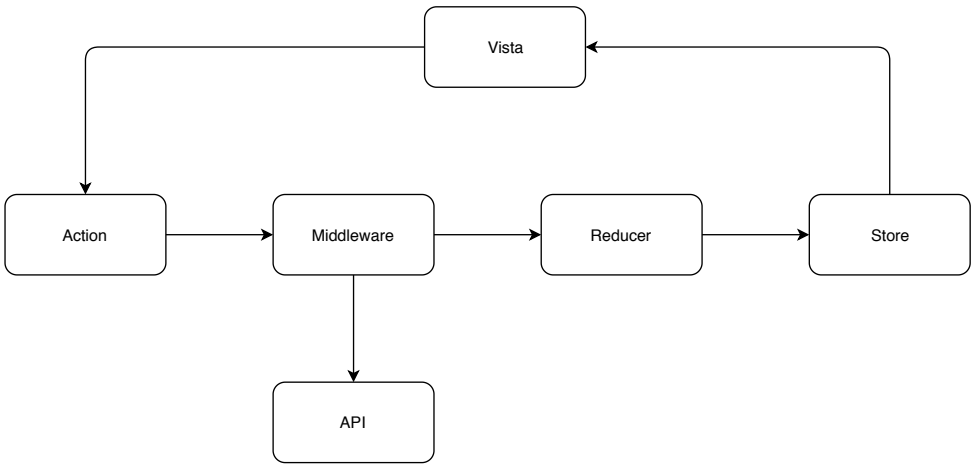

Figura B.6: Flujo de datos en una única dirección de Redux

B.6.6 Redux

Redux (c.f. B.6) es una librería que está basada en el patrón de arquitectura Flux, aunque no es, exactamente, Flux. Se trata de una librería extremadamente simple, ya que en su versión mínima solo tiene 99 líneas de código.

Al igual que Flux, Redux tiene acciones y *store*. Pero simplifica aún más la arquitectura Flux, eliminando el *dispatcher* e imponiendo ciertas restricciones en forma de tres principios:

- Existe una fuente única de verdad: Esto es, el estado completo de la aplicación se almacena dentro de un único *store*, en forma de árbol de objetos.
- El estado es de solo lectura: La única forma de alterar el estado es emitir una acción, que no es otra cosa que un objeto que describe lo que ha ocurrido.
- Los cambios en el estado se hacen empleando *pure functions*.

El tener una fuente única de verdad permite que el estado pueda ser serializado desde el servidor, en el cliente, sin ningún esfuerzo. Esto permite crear, fácilmente, aplicaciones universales. También es cierto que, al tener un único árbol de estado, se facilita la depuración de la aplicación, al mismo tiempo que hace más fácil salvar el estado.

Al no existir *dispatcher*, tampoco requiere el registro de los *stores* en el mismo.

Las acciones, igual que en Flux, expresan la intención de cambiar el estado. Cabe destacar que todos los cambios están centralizados y se ejecutan en orden estricto uno a uno, por lo que no caben *race conditions*. Al igual que en Flux, una acción es un objeto con un campo tipo y otro para datos (*payload*).

Los cambios en el *store* se hacen empleando *pure functions*, o lo que es lo mismo, funciones que retornan, siempre, el mismo resultado para idénticas entradas. Estas funciones, en Redux,

se denominan *reducers*. Un *reducer* no es más que una función (*pure*), que toma como argumentos de entrada el estado actual y la acción, y devuelve el estado siguiente. Aquí conviene comentar que el estado siguiente no es una mutación del estado anterior, ya que el estado es inmutable. El estado siguiente es un nuevo objeto.

Un *reducer*, como función que es, puede dividirse en *reducers* más pequeños, que gestionarán partes específicas del árbol de estado. Posteriormente, es posible combinar los *reducers* pequeños en uno más grande.

La inmutabilidad requerida por Redux facilita, enormemente, la implementación de técnicas de detección de cambios. Por ejemplo, es extremadamente fácil comparar el estado actual con el estado siguiente y, cuando proceda, analizar las diferencias. Evidentemente, esto también facilita las técnicas de depuración y hace más seguro el manejo de datos.

B.6.7 Alternativas a Redux

Existen bastantes implementaciones del patrón de arquitectura Flux. Algunas de las más conocidas son:

- MobX[95]: Librería para la gestión del estado que usa *observables* para responder a cambios en el estado.
- Reflux[96]: Librería Flux muy simple.
- Fluxible[97]: Un *framework* creado por Yahoo para aplicaciones Flux isomórficas (que pueden ser renderizadas en múltiples plataformas).

B.7 GraphQL

Hoy en día, construir una API puede convertirse, rápidamente, en una tarea altamente compleja. Si una API posee un amplio abanico de aplicaciones que la utilizan, podemos encontrarnos, más temprano que tarde, con un gran conjunto de necesidades requeridas pero divergentes entre sí. Dar soporte a diferentes tipos de clientes requiere flexibilizar los puntos de entrada a servicios y procesos (*endpoints*), y aquí es donde GraphQL puede ayudarnos. Hoy en día, los usuarios tienden a usar los diferentes servicios desde múltiples dispositivos (su móvil, su PC, etc). Por ello, cuando se diseña una API es conveniente que sea suficientemente flexible en lo referente a la variedad de dispositivos soportados. Debemos pensar que una API puede ser usada en parte por una aplicación, completamente por otra, etc. Por ello, considerar la capacidad de flexibilización futura de una API, a la hora de elegir la tecnología con la que implementarla, será casi siempre una buena decisión. Aunque se debe tener en cuenta que hay ciertas APIs que pueden mantenerse inmutables en el tiempo o su tamaño es

muy pequeño, por lo que tener en cuenta el grado de flexibilización soportado es un aspecto del todo irrelevante. No obstante, esto último no suele ser un caso muy predominante.

GraphQL [22] es un lenguaje de consulta y manipulación de datos, de código abierto, fuertemente tipado, jerárquico y enfocado a la construcción de APIs. Las solicitudes, escritas en lenguaje GraphQL, emitidas por clientes y con destino un servicio GraphQL, se denominan documentos. Al mismo tiempo, también provee un servicio para satisfacer las consultas con los datos disponibles. Quizá la ventaja más destacable de GraphQL es que permite a los usuarios describir sus necesidades de datos empleando un lenguaje de consulta declarativo, en cierta forma similar al archiconocido SQL, pero más sencillo.

En sus inicios, GraphQL fue desarrollado internamente por Facebook. Posteriormente, en el año 2015, fue liberado públicamente. Unos años más tarde, concretamente el 7 de Noviembre de 2018, el proyecto pasó de Facebook a la fundación GraphQL, bajo el paraguas de la fundación Linux.

GraphQL permite la creación de APIs flexibles, potentes, fáciles de usar y de mantener. Evidentemente, no se trata de una solución universal para todo tipo de APIs. Se debe tener en cuenta que todas las facilidades que proporciona se obtienen a costa de incrementar la complejidad de la solución. Por tanto, para APIs relativamente simples, no es una opción que suela compensar. Otro aspecto a considerar detenidamente son las cachés de las que suelen presumir las diferentes implementaciones. Es imprescindible, especialmente en proyectos del ámbito empresarial, analizar el nivel de efectividad de dichas cachés, pudiendo de esta forma evaluar las ventajas de GraphQL sobre otras arquitecturas de servicios web.

GraphQL proporciona:

- Un sistema de tipos.
- Un lenguaje de consulta.
- Semánticas de ejecución.
- Validación estática.
- Introspección de tipos, o lo que es lo mismo, capacidad para examinar el tipo de un objeto en tiempo de ejecución.

El lenguaje de consulta declarativo, que GraphQL proporciona, permite a los usuarios de la API describir, de forma sencilla, los datos que necesitan.

A poco que se busque información referente a GraphQL, es frecuente encontrar artículos en los que se presenta al mismo como el sucesor de REST. Esto no parece otra cosa que un craso error. Ambas arquitecturas de servicio presentan enfoques totalmente divergentes, en cierto modo hasta incomparables. Personalmente, no creo, en general, que una sea mejor que

la otra. Simplemente, dependiendo del tipo de servicio y el contexto, una puede ajustarse mejor que la otra a las necesidades demandadas.

B.7.1 Esquema GraphQL

El esquema GraphQL se aloja, habitualmente, en el servidor. Dicho esquema constituye la fuente canónica de información a la hora de atender cualquier solicitud del cliente. Lo que es más importante, el esquema define el modelo de dominio de la API GraphQL y determina la forma en la que los datos son recuperados.

En el esquema GraphQL se mantiene un grafo de tipos. Pasar de la teoría a la práctica es tan sencillo como implementar el código, por ejemplo Erlang, que se corresponde con dicho grafo. El grafo de tipos contiene nodos y aristas. Cada nodo representa un tipo. Conviene destacar que existe un tipo raíz (`QueryRootType` para consultas), que representa el "inicio" del grafo. Las aristas del grafo representan campos; en el caso de una consulta, representan los campos del documento de consulta emitido por el cliente (ver figura B.7). En conclusión, al construir un grafo de este estilo, lo que estamos creando es un esquema en el que se pueden observar las entidades existentes y, lo que es más importante, las relaciones entre las mismas.

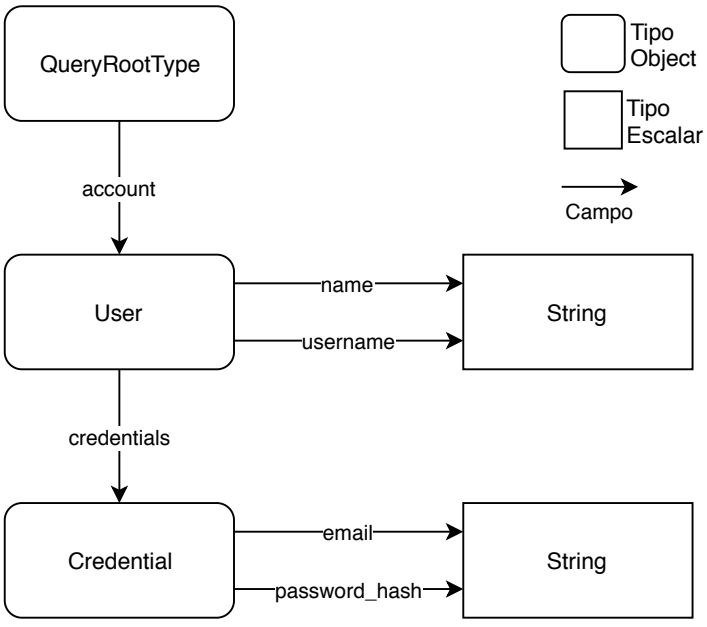

Figura B.7: Ejemplo de grafo en el esquema GraphQL

Cuando el servidor recibe un documento GraphQL, emitido por un cliente de la API, simplemente observando el esquema, puede determinar si se trata de un documento válido que contiene solicitudes de consulta/modificación de datos que puede atender.

Desde el punto de vista del grafo, si un tipo tiene campos (aristas), dicho tipo representa

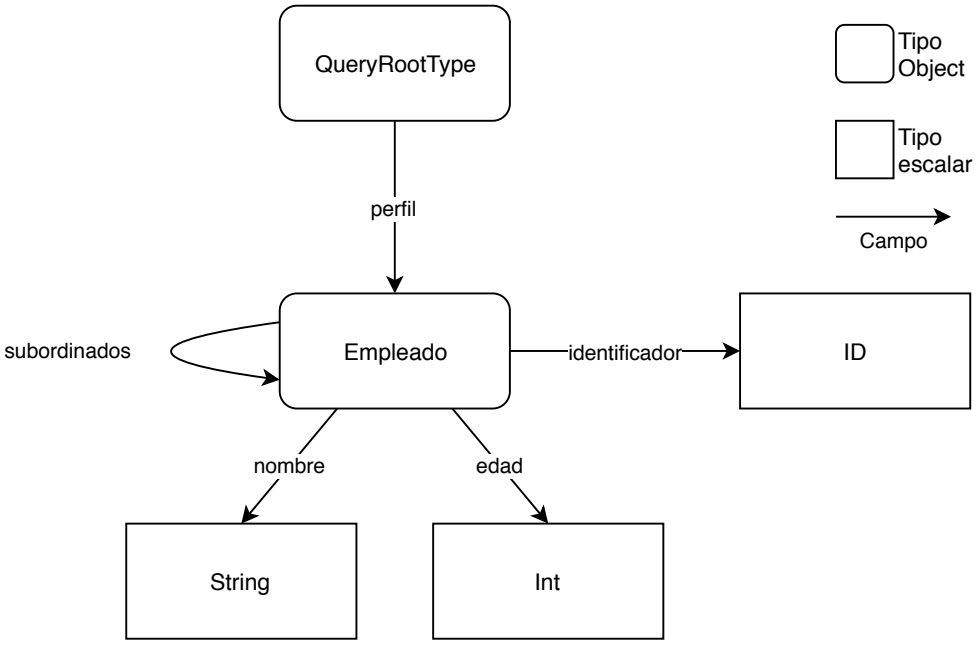

Figura B.8: Ejemplo de grafo recursivo en el esquema GraphQL

un valor complejo y se denomina tipo objeto. Un tipo objeto puede tener campos que apuntan al mismo tipo, esto es, aristas con origen y destino en el mismo nodo. Esto permite modelar relaciones recursivas (ver figura B.8).

Los tipos que representan valores simples se conocen como tipos escalares. Como es de esperar, los tipos escalares no tienen campos.

Del contenido de esta sección podemos extraer dos conclusiones. Primero, GraphQL, mediante el requisito de requerir un esquema, está forzando al desarrollador de la API a construir una representación de sus datos. Segundo, el esquema impone un formato restringido para las entradas que se pueden recibir desde el cliente.

B.7.2 Funcionamiento de GraphQL

Como acabamos de ver, el esquema GraphQL dota al servidor, poseedor de la API, con un conjunto de reglas y tipos.

El cliente de la API, en un momento dado, emitirá un documento en lenguaje GraphQL, con el fin de satisfacer sus necesidades de datos. Llegados a este punto, es recomendable señalar que, absolutamente, todas las operaciones GraphQL se realizan contra una única URL. Dicha URL siempre es la misma para toda operación contra la API. La petición se realiza empleando el protocolo HTTP, concretamente solicitudes GET o POST. Cabe destacar que el formato del documento GraphQL emitido por el cliente determina el formato de la respuesta emitida por el servidor.

Cuando el servidor recibe el documento se remite al esquema, antes de ejecutarlo. En este instante lo que hace el servidor es ver si el documento contiene los campos adecuados, relacionados de la forma correcta y con el tipo debido. Si esto es así, la operación se ejecutará.

Una vez ejecutada la operación GraphQL, solicitada por el cliente, se genera una respuesta que es enviada de vuelta al cliente, tal y como representa la figura B.9. Dicha respuesta, es frecuente que sea generada en formato JSON, aunque esto no es, ni mucho menos, obligatorio.

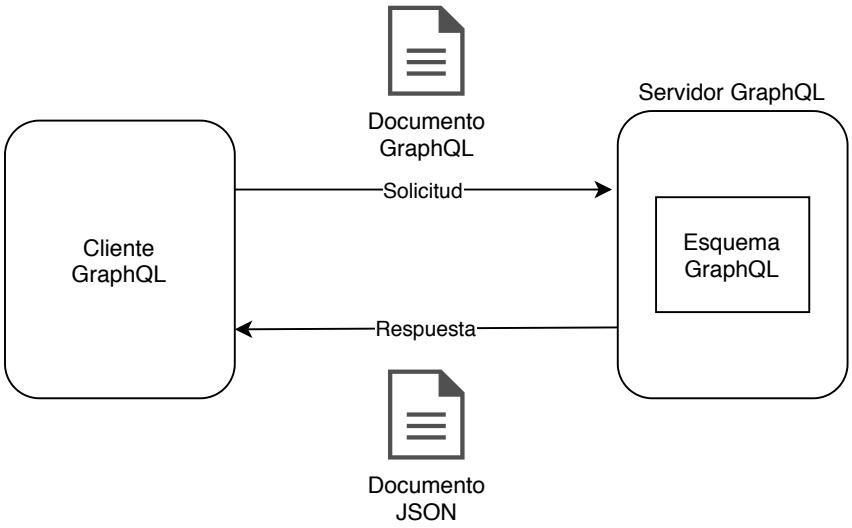

Figura B.9: Operación de GraphQL

B.7.3 Tipos soportados en GraphQL

Los tipos son la unidad fundamental en torno a la cual gira el esquema GraphQL.

Los tipos `Scalar` y `Enum` son los dos tipos más sencillos existentes en el lenguaje. El tipo `Scalar` representa un valor primitivo (`String`, `Int`, etc.). Por su parte, `Enum` especifica el espacio de valores posibles.

Los tipos `Object` definen un conjunto de campos (*Fields*), en el que cada campo es de otro tipo definido en el sistema. De esta forma se definen jerarquías.

El tipo `Input Object` permite definir exactamente los datos que el esquema espera recibir. Es un tipo diferente de `Object` y origen de errores comunes.

Un tipo `Interface` define una lista de campos. Los tipos `Object` que implementen dicha interfaz deben implementar, de forma obligatoria, los campos definidos en la misma.

Un tipo `Union` define una lista de posibles tipos. La diferencia con `Interface` la marca el término “posibles”.

Hasta aquí hemos visto tipos más o menos comunes en multitud de lenguajes. Los tipos vistos hasta ahora se conocen como tipos con nombre. No obstante, GraphQL nos provee con

otra clase de tipos, denominados tipos envoltorio (*wrapping types*):

- Tipo `List`: Este tipo envuelve otro tipo y permite indicar que un campo representa una lista de otros tipos.
- Tipo `Non-Null`: Al igual que el anterior, este tipo envuelve otro tipo, denotando que el valor resultante nunca será *null*.

En GraphQL, los tipos se suelen clasificar también en tipos de entrada (*input types*) y tipos de salida (*output types*). Esta clasificación no es un concepto menor en el universo GraphQL, puesto que se trata de un concepto básico de funcionamiento. Los tipos de entrada sirven para describir los valores aceptados como entrada para argumentos y variables. Los tipos de salida sirven para describir los valores aceptados como salida para campos. Conviene destacar que ambos conjuntos de tipos no son disjuntos. Dentro del conjunto de todos los tipos existentes en GraphQL, existen tipos solo de entrada (como `Input Object`), existen tipos solo de salida (como `Interface`, `Union` y `Object`) y, finalmente, existen tipos de entrada y salida (`Scalar` y `Enum`). El caso de los tipos envoltorio es un poco especial, puesto que depende de cómo el tipo envuelto pueda ser usado.

A modo de esquema mental, es útil tener una visión simplificada del sistema de tipos. Así una respuesta GraphQL puede imaginarse como un árbol jerárquico en el que los elementos intermedios son tipos `Object` y las hojas las conforman tipos `Scalar` o `Enum`. Pero no deja de ser una visión excesivamente simplificada.

B.7.4 Operaciones soportadas por GraphQL

Un documento GraphQL puede contener tres tipos diferentes de operaciones: consultas, mutaciones y suscripciones. Es además posible que contenga fragmentos. Un fragmento no es más que una unidad de composición que permite la reutilización de consultas.

Los tres tipos de operaciones permitidos se definen como sigue:

- Consulta (*query*): es una operación de obtención de datos (fetch) de solo lectura.
- Mutación (*mutation*): esta operación consta de una modificación de datos (escritura) seguida de una obtención de datos (fetch).
- Suscripción (*subscription*): se trata de una solicitud, a largo plazo, que obtiene datos en respuesta a eventos en el origen.

El esquema GraphQL define un tipo de operación raíz para cada operación, valga la redundancia, que soporta. El tipo raíz determina el lugar, en el sistema de tipos, en el que comienzan las operaciones.

El tipo de operación raíz para una consulta (`RootQueryType`) es obligatorio y debe ser un tipo `Object`.

El tipo de operación raíz para una mutación (`RootMutationType`) es opcional, dependiendo de si el sistema soporta o no mutaciones. En caso afirmativo debe ser un tipo `Object`.

El tipo de operación raíz para una suscripción (`RootSubscriptionType`) es opcional, dependiendo de si el sistema soporta o no suscripciones. En caso de que así sea, obligatoriamente ha de ser un tipo `Object`.

Las operaciones en GraphQL pueden tener un nombre o ser anónimas, pueden tener variables de entrada o no, es posible que tengan un alias y, concretamente, los tipos `Object` pueden tener argumentos. Además, el lenguaje permite descomponer operaciones en fragmentos que, posteriormente, se pueden combinar de diferentes formas, dando lugar a distintas operaciones y fomentando así la reutilización de código.

Las consultas se asocian con "*READ*". Las mutaciones suelen estar ligadas a "*CREATE/UPDATE/DELETE*". La diferencia entre *query* y *mutation* es, en muchos casos, mayormente conceptual.

La parte realmente obligatoria de GraphQL son las consultas. Mutaciones y suscripciones son opcionales. Las suscripciones se suelen emplear para notificar el resultado de una mutación ejecutada por un cliente. El aspecto más interesante de las suscripciones es que proporcionan soporte para notificaciones en tiempo real blando. Por tanto, cuando un cliente altera ciertos datos mediante una mutación, todos los clientes suscritos a ese tema (*topic*) serán notificados del resultado, prácticamente, de forma inmediata.

Las suscripciones funcionan por temas. Esto es, un cliente ha de suscribirse al tema en el que está interesado. Una vez solicitada la suscripción, GraphQL notificará al cliente si su suscripción fue exitosa. En caso afirmativo, a partir de ahí, comenzará a recibir notificaciones referentes a la temática a la que se ha suscrito. Cuando cese el interés del cliente en un determinado tema, ha de enviar un requerimiento al servidor para darse de baja en dicha temática.

Como es previsible, las suscripciones son interesantes pero requieren de la disponibilidad de ciertos recursos en el servidor. Por un lado, se suelen implementar utilizando *WebSockets* para las notificaciones a suscritos. Por tanto, se necesita un servidor web con soporte para el protocolo *WebSocket*. Por otro lado, también se requiere un sistema *software Pub/Sub* (Publicación/Suscripción), por encima del servidor web. La adopción de suscripciones GraphQL, en el servidor, no es barata. Estamos incrementando el software requerido, lo cual, como es evidente aumenta la complejidad. También se requiere una mayor capacidad de carga del sistema y concurrencia. Finalmente, estamos pasando de una *API stateless*, basada en HTTP, a una API *stateless*(HTTP)/*stateful*(Websockets), lo cual complica aún más su gestión, por poner un ejemplo, se requiere una gestión diferente de los *tokens* de autorización en el caso de *WebSockets* o de una simple petición HTTP.

B.7.5 Clientes GraphQL

Actualmente, son dos los clientes GraphQL de uso mayoritario: Apollo Client [32] y Relay Modern [33]. Ambos clientes cumplen, sobradamente, la función para la que fueron creados. Esto es, recuperación de datos y acceso modular a los mismos. Los dos implementan cachés con las que amortiguan las diferentes operaciones.

Básicamente, optar por uno u otro es más una cuestión de gustos que técnica. Como opinión totalmente personal, Relay parece sensiblemente más denso que Apollo a nivel documentación, y su curva de aprendizaje un poco más dificultosa. Apollo, por su parte, tiene una documentación que no es mala, pero sí ciertamente mejorable. También presenta algunos *bugs* bastante molestos a la par que preocupantes, (ciertos métodos que no se invocan cuando deberían, etc.) sobre todo a la hora de considerar su uso en sistemas en producción. Cierto es que también existen ”parches” para salir del paso.

En los últimos tiempos, está tomando fuerza la tendencia de utilizar el cliente GraphQL, empleado en el *frontend*, para mantener el estado local del mismo. Siempre es de agradecer la variedad de alternativas ante un problema. No obstante, se debe plantear que adoptando esta propuesta es posible que se esté creando una excesiva dependencia, en el *frontend*, de los citados clientes. Aún entendiendo el afán de las empresas en crear soluciones *software* lo más completas e integradas posibles, creemos firmemente que los desarrolladores deberían tomarse algún tiempo para meditar el grado de dependencia de cierto *software* que desearían que presentasen sus desarrollos. Hacer el estado local del *frontend* dependiente de un cliente GraphQL específico es muy posible que, a la larga, haga más dificultoso el mantenimiento del sistema o su reemplazo por otra solución. Como es obvio, abrazar esta solución implica adoptar la posibilidad de utilizar el lenguaje GraphQL para consultar el estado local de forma exactamente igual a como se escriben las consultas remotas. Lo dicho, no es una decisión trivial y deberían ser evaluadas las consecuencias de adopción de la misma a largo plazo.

B.8 Tecnologías *backend* alternativas

En esta sección daremos un par de pinceladas sobre tecnologías alternativas, que podrían haber sido empleadas en el desarrollo del *backend* de este proyecto. Debido al gran número de opciones existentes, esta sección no deja de ser, ciertamente, anecdótica.

B.8.1 API del backend

En este proyecto hemos implementado el servicio de planificación personal empleando GraphQL. En todo caso, hubiera sido perfectamente viable haber empleado una arquitectura REST (*Representational State Transfer*) para dicho servicio.

En realidad, estaríamos pasando de un lenguaje declarativo a un enfoque de solicitud de recursos basado en URLs. Lo que en GraphQL son mutaciones, en REST serían operaciones de creación, actualización y borrado. Y lo que en GraphQL son consultas, en REST serían lecturas.

Las tecnologías subyacentes serían prácticamente las mismas: HTTP y JSON. No obstante, hay que recordar que GraphQL emplea Websockets para suscripciones, por tanto, habría que buscar una solución adicional para suministrar suscripciones, ya que REST es una arquitectura sin estado.

A modo de idea, personalmente, para reemplazar las suscripciones GraphQL, emplearía alguna librería CRDT (*Conflict-free replicated data type*) [98]. Un CRDT es una estructura de datos que puede ser replicada a través de múltiples computadoras en una red. Las réplicas pueden ser actualizadas independientemente y de forma concurrente (sin coordinación entre ellas). Además, siempre es posible solucionar inconsistencias, matemáticamente, que puedan surgir. Una librería candidata sería, por ejemplo, y-js [99].

B.8.2 Sistema de gestión de bases de datos

Prácticamente, cualquier sistema de gestión de bases de datos, disponible hoy en día, es viable para este proyecto. El único requisito, más o menos deseable, es que disponga de una API, REST o similar, con la que se pueda interactuar. Simplemente, comentar que si se trata de un SGBD del que ya exista un adaptador para Elixir, siempre es más rápido y, posiblemente, más fiable. En caso contrario, hay que escribir dicha librería, tarea que es bastante compleja si lo que se busca es una solución óptima.

B.8.3 Lenguaje de programación para el backend

Actualmente, recomendar un lenguaje alternativo para el *backend* supondría entrar en una guerra de detractores y fans de la gran variedad de los mismos que existe hoy en día. A modo de elección personal, si no se hubiera elegido Elixir, muy posiblemente la opción hubiera sido Rust [100], lenguaje del que se habla extraordinariamente bien en los círculos de Elixir.

B.9 El lenguaje Elixir

Elixir [2, 3, 101, 102] es un lenguaje funcional, concurrente y de propósito general, que se ejecuta en la máquina virtual de Erlang [69, 103], más conocida como la BEAM. Como es de esperar, todo lo que se puede hacer en Erlang, se puede hacer en Elixir, y viceversa.

El entorno de ejecución de Elixir es una instancia de la BEAM. Cuando se inicia el sistema, Erlang toma el control. Se crea un proceso del sistema operativo, para la instancia de la BEAM, y todo ejecuta dentro de ese proceso. La mejor forma de encontrar el citado proceso es buscar,

```
$ iex -pa /path/a/mi/codigo -pa path/a/otro/codigo
```

Listing B.3: Intérprete interactivo de Elixir

por ejemplo, mediante el comando `top` o similar, un proceso con el nombre *beam*. Es importante tener en cuenta que, una vez que el sistema está iniciado, la máquina virtual de Erlang hace un seguimiento de todos los módulos que se encuentran cargados en memoria. Cuando se invoca una función de un módulo, lo primero que hace la BEAM es comprobar si el módulo ya está cargado. Si es así, ejecuta el código correspondiente a la función. En caso contrario, busca el fichero compilado del módulo (cada módulo compilado reside en un fichero con la extensión `.beam`) y lo carga para, posteriormente, ejecutar el código de la función invocada. Si se definen múltiples módulos dentro de un mismo fichero fuente, el compilador generará un fichero `.beam` para cada módulo. El intérprete interactivo de Elixir puede iniciarse fácilmente invocando `iex` B.3.

Un fichero fuente puede definir uno o varios módulos, en su interior, pero también puede contener algún código fuera de esos módulos. Supongamos que dicho fichero se llama *mis_modulos.ex*. Cuando nosotros ejecutamos "`elixir --no-halt mis_modulos.ex`", se inicia una instancia de la BEAM. Posteriormente, el fichero es compilado en memoria y los módulos resultantes se cargan en la máquina virtual. El código que reside fuera de los módulos es interpretado y no se genera ningún fichero `.beam` en el disco. Una vez hecho todo esto, la instancia de la BEAM debería detenerse, pero sigue viva porque hemos especificado la opción `--no-halt`.

Actualmente, Elixir sigue un esquema de versionado semántico [104][105], con solo una "major release" [Anexo A] liberada hasta la fecha. En resumen, la versión actual de su API pública es, totalmente, retrocompatible con la primera versión liberada de la API.

Algunos de los aspectos más destacables del lenguaje son los siguientes:

- Las estructuras de datos, en Elixir, son inmutables. Esto permite un código más claro. Una función puede devolver una versión modificada de su entrada, que reside en otra posición de memoria. La versión modificada compartirá tanta memoria como sea posible con la original.

- Es un lenguaje con tipado dinámico. El tipo de una variable lo determina el valor que posee.

- Es un lenguaje que hace un fuerte uso de concordancia de patrones (*pattern matching*) y expresiones regulares. Si tenemos algo tal como `baraja = {:parse, :error, :before, :la, :sota, :de, :bastos}`, el operador `=` se llama *match*. El lado izquierdo del operador se llama *pattern* y en el lado derecho tenemos una expresión que se evalúa

a un término (*term*). En Elixir, se emplea *term* como una abreviatura con el significado de "cualquier tipo". En tiempo de ejecución, se hace concordar el lado izquierdo con lo que hay en el lado derecho. En el ejemplo, hacemos concordar la variable `baraja` con el término `{:parse, :error, :before, :la, :sota, :de, :bastos}`. Como una variable siempre concuerda con el término del lado derecho, `baraja` quedará ligada a dicho término. No debe confundirse la concordancia de patrones con la asignación, puesto que se trata de un concepto más complejo.

- La programación defensiva brilla por su ausencia. El *try/catch* no está muy bien visto en la comunidad de usuarios. La tolerancia a fallos se suele dejar en manos de procesos supervisores que conforman árboles de supervisión y definen estrategias de recuperación a seguir.
- Admite intercambio de código en caliente. Esto es, no es necesario detener la aplicación para aplicar una actualización de código.
- Se pueden utilizar librerías de Erlang. Una gran ventaja, ya que pone a disposición del desarrollador el inmenso OTP (*middleware*, herramientas y librerías escritas para Erlang).
- Manejo fácil de la concurrencia B.10. Crear procesos es muy barato y sincronizarlos, mediante paso de mensajes, prácticamente trivial.
- Elixir, al utilizar la máquina virtual de Erlang, se muestra como una alternativa perfectamente viable para el desarrollo de sistemas distribuidos y, al igual que Erlang, escala muy bien. Los sistemas Elixir, de forma análoga a lo que es común en sus homólogos Erlang, suelen tener una alta disponibilidad.
- El lenguaje posee una sintaxis muy fácil e intuitiva (azúcar sintáctico). Dicha sintaxis permite no escribir excesivo código y, el que se escribe, es posible hacerlo de una forma muy clara.
- *Releases*: Un directorio autocontenido que alberga la aplicación, todas sus dependencias y la máquina virtual de Erlang. Una vez que está ensamblada, la *release* puede ser empaquetada y desplegada a cualquier objetivo, siempre que tenga la misma versión del sistema operativo.
- Mix: Una completísima herramienta para la gestión del proyecto, en línea de comandos. Genera plantillas/esqueletos, construye el proyecto, genera la documentación, gestiona dependencias, ejecuta las pruebas, etc. La capacidad de *scripting*, admitida por Mix, es uno de las grandes fortalezas de Elixir. Esta herramienta se suele emplear para el flujo de trabajo: desarrollo $\Rightarrow$ compilación $\Rightarrow$ pruebas.

- Soporta el gestor de paquetes Hex [106]. Un clásico para ecosistemas Erlang.
- Umbrella: Permite dividir un proyecto más o menos grande en múltiples aplicaciones. Así se tiende a simplificar una lógica compleja en partes más simples.
- El lenguaje posee un potente sistema de macros, que favorecen la extensibilidad del mismo, a la vez que permiten el desarrollo de lenguajes específicos del dominio con suma facilidad. Una macro no deja de ser código Elixir que recibe una representación del código de entrada Elixir, parseado, y retorna una versión alternativa de dicho código. Las macros permiten realizar transformaciones del código en tiempo de compilación y permiten modificar la semántica del código de entrada a la macro.
- *Protocols*: El polimorfismo no es otra cosa que la decisión, en tiempo de ejecución, referente a qué código ejecutar en función de la naturaleza de los datos de entrada. En Elixir, una de las maneras de obtener el polimorfismo es haciendo uso de *protocols*. Un *protocol* es un módulo que declara ciertas funciones sin implementarlas. Posteriormente, se debe realizar una implementación del *protocol* para distintos tipos de datos.
- *ets*: Elixir implementa la memoria compartida entre procesos en forma de tablas *ets*.
- Todo proceso en Elixir tiene un diccionario de proceso, clave-valor, en el que se pueden almacenar valores que el proceso podrá acceder en cualquier momento. No obstante, su uso genera mucha controversia, en la comunidad Elixir, puesto que se argumenta que emplear dicho diccionario hace el código más difícil de seguir y, por tanto, de comprender.

Elixir es un lenguaje funcional y la mayor parte de la programación, bajo el mismo, consiste en escribir módulos de funciones. Las funciones son ciudadanos de primera clase. Si se desea seguir cualquier otro tipo de paradigma, entonces no es la opción más aconsejable.

No hay tipo booleano, ni nulos. En su lugar se utilizan los átomos (constantes cuyo valor es su nombre) true, false y, por ejemplo, nil.

No hay tipo "*cadena de caracteres*". En su lugar se utiliza el tipo `binary`, que representa secuencias de *bytes* con codificación Unicode UTF-8.

El tipo *tupla* permite agrupar un pequeño número de campos de tamaño fijo. Los elementos se almacenan de forma contigüa en memoria. Ej. {:ok, "ok_msg"}.

El tipo *lista* se usa para colecciones de tamaño variable. Ej. [1 | [2, 3]].

El tipo `Map` es una estructura de datos clave-valor. Ej. %{:a => 1, 2 => :b}.

A modo de aclaración, conviene destacar que un proceso en Elixir no tiene, absolutamente, nada que ver con un proceso/hilo del sistema operativo. En Elixir, un proceso es una primitiva básica de concurrencia, y no es para nada raro ver millones de estos procesos ejecutando en

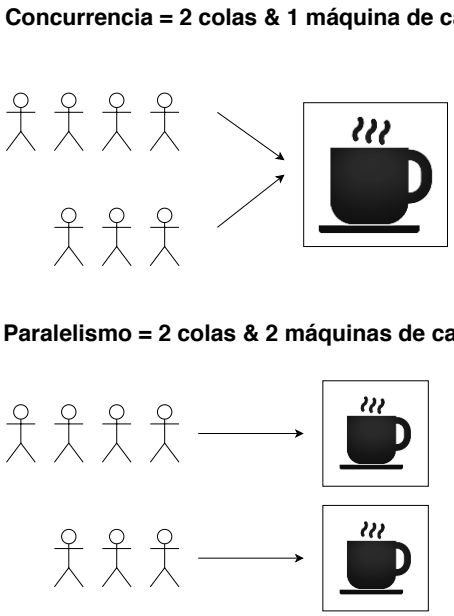

Figura B.10: Concurrencia vs. Paralelismo

la BEAM. La BEAM dispone de sus propios planificadores para distribuir los procesos sobre los núcleos de la CPU.

Erlang y Elixir no son, precisamente, lenguajes rápidos. Por tanto, si el desarrollo requiere mucha carga de CPU, definitivamente BEAM no es el camino a seguir. Otros lenguajes como C o C++, que compilan a código nativo, son mucho más rápidos, con diferencia. Por tanto, es difícil encontrar librerías pensadas para trabajos CPU intensivos, del estilo de Apache Lucene [107], escritas para Elixir. Con esto no queremos decir que está todo perdido. Existen soluciones como delegar dichas tareas en un servidor externo y escribir un módulo de comunicación para Elixir o, incluso, usar el sistema de *ports* que Elixir suministra. El módulo *Port* [108], de Elixir, nos dota de funciones para iniciar procesos del sistema operativo externos a la máquina virtual de Erlang, al mismo tiempo que nos permite comunicarnos con dicho procesos mediante paso de mensajes. Este mecanismo nos provee de una forma muy fácil de interaccionar, por ejemplo, con programas C o comandos del sistema operativo. Por supuesto, es recomendable tener unos conocimientos mínimos sobre sistemas de ficheros y, más concretamente, sobre descriptores.

Por otro lado, la velocidad no lo es todo. Por ello, Erlang/Elixir se centran más en mantener el rendimiento estabilizado, dentro de unos límites, que en situarse en el *top ten* de los lenguajes más veloces.

En lo referente al tiempo de respuesta de los sistemas basados en Elixir, suele ser bueno, en condiciones de carga normales; a no ser que el código escrito sea manifiestamente ineficiente.

El ecosistema Elixir es pequeño, aunque con la ayuda de Erlang su tamaño aumenta sensi-

blemente. No obstante, si lo que se desea, a la hora de realizar un proyecto, es disponer de un gran número de alternativas entre las cuales elegir, es más razonable enfocarse en lenguajes tales como JavaScript.

B.10 Persistencia en Elixir

Ecto [29, 70] es, probablemente, el *framework* para persistencia en Elixir más conocido, maduro y estable. El proyecto tiene entre sus colaboradores a varios miembros del equipo de Elixir, por lo que el soporte ofrecido es de lo mejor que se puede encontrar para este lenguaje.

Cuando enfocamos el desarrollo del *backend* hacia el ecosistema Elixir, es frecuente hacer uso del *framework* Phoenix [37], con el objetivo de mantener cierta "armonía tecnológica''. Basta con echar un vistazo a Phoenix para darse cuenta de que incluye a Ecto como la librería de bases de datos por defecto. En todo caso, Ecto puede incluirse en cualquier proyecto Elixir, independientemente del uso del *framework* Phoenix.

Para aquellos que hayan trabajado con *frameworks* parecidos, pongamos LINQ en .NET o Active Record en Rails, Ecto les puede resultar familiar. En cualquier caso, Ecto tiene su propia personalidad, como puede ser su sintaxis, de factura propia, para consultas, relaciones o migraciones. Es más, posee su propio lenguaje específico del dominio (DSL), construido haciendo uso de las macros de Elixir.

B.10.1 Características principales de Ecto

Ecto se emplea, mayoritariamente, con bases de datos relacionales. Existen adaptadores oficiales tanto para PostgreSQL como para MySQL. Además, hay disponibles desarrollos para otras bases de datos como MongoDB, Mnesia o SQLite. Sin embargo, no todas las bases de datos más importantes están soportadas, como puede ser el caso de CouchDB, para la que no existe un adaptador relativamente estable y maduro.

Es un *framework* muy flexible. De hecho, hay funciones cuyo uso no requiere siquiera de una base de datos. Esto puede resultar algo extraño, al principio, pero es habitual usar Ecto con otras fuentes de datos diferentes a sistemas de gestión de bases de datos. Otro aspecto a destacar es que hace posible la construcción de consultas complejas mediante la composición de otras más simples. Por supuesto, en caso de ser necesario, admite SQL puro.

Sin duda, la característica que más resalta en este *framework* es que requiere ser tremendamente explícito. Al contrario que otros *frameworks*, no hace absolutamente nada que no se le ordene de forma explícita. La "magia'' que llevan a cabo otros *frameworks* puede resultar muy cómoda y eficaz, al principio. Pero cuando la cantidad de datos crece, dicha "magia'' es fácil que se traduzca en una pérdida de rendimiento y la trazabilidad de lo que se está haciendo se difumine. Esta es la principal razón por la que Ecto exige este elevado nivel de explicitud.

Visto todo esto, como cabe esperar, Ecto no soporta carga perezosa sino precarga. La carga perezosa exigiría no ser explícito.

Se trata de un *framework* totalmente modular, aunque el único módulo que, mayoritariamente, resulta de uso obligatorio, es el módulo `Repo`, el cual se puede considerar, sin duda alguna, el núcleo de Ecto. Todas las operaciones contra la base de datos han de pasar, obligatoriamente, por el repositorio. El resto de módulos son prescindibles y se suministran con el objetivo de facilitar el uso de la librería y el manejo de datos.

La herramienta de gestión de proyectos de Elixir, Mix, posee diversas funciones generadoras para crear esqueletos que facilitan el desarrollo con Ecto. Esto es un claro ejemplo del nivel de integración que presenta esta librería en el ecosistema Elixir.

B.10.2 El patrón repositorio

La arquitectura de Ecto gira en torno al patrón de arquitectura repositorio. En aras de una mayor claridad, se va a omitir una explicación formal de este patrón, que poco aportaría más que complejidad en el razonamiento. Por tanto, aún a riesgo de no satisfacer las mentes más ávidas de conocimiento, vayamos por la senda de lo simple: existe una clase o módulo a través del cual pasa toda comunicación con la base de datos subyacente. Dicho módulo es el repositorio. Esto es, si se desea hacer una consulta, se envía al repositorio; si se desea actualizar un dato, se envía al repositorio, etc. Posteriormente, es el repositorio el que se encarga de comunicar con la base de datos, para realizar la operación, y, más tarde, obtener la respuesta para retornar a la aplicación.

Cabe destacar que el único punto de contacto, entre la aplicación y la base de datos, es el repositorio.

B.10.3 Migraciones

El concepto de migración no es nuevo. De hecho, existe también en *frameworks* tan veteranos como Rails para Ruby y Django para Python.

Una migración es, simplemente, una función. Ecto usa las migraciones para crear y modificar las tablas de la base de datos. La *migración* es, literalmente, un conjunto de comandos Elixir que contiene las instrucciones para hacer los cambios en la estructura de la base de datos. Lo normal, en Ecto, no es implementar todo el modelo de bases de datos de golpe. La idea es: se desea añadir soporte para cuentas de usuario al sistema en desarrollo, luego tenemos que crear una tabla “usuarios” y una tabla “credenciales”, entonces escribiremos una migración que crea esas tablas (con las columnas correspondientes). La migración se ejecuta siempre como una transacción. Si más tarde nos damos cuenta de que no hace falta la tabla “credenciales”, pues crearemos otra migración que crea, a su vez, una columna en la tabla “usuarios”, que copie en dicha columna los datos de la tabla “credenciales” y que elimine la

tabla "credenciales". Y así se va creando y modificando la estructura de la base de datos poco a poco, y según se vaya necesitando.

Cada migración reside en un fichero con una marca de tiempo. Cuando se crea una migración, se crea también ese fichero y Ecto lo añade a la cola de migraciones. Luego haremos un push y subiremos todo al repositorio git, por ejemplo, Github [5]. Cuando otro desarrollador se baje nuestro proyecto del repositorio git en Github, para configurarlo tendrá que ejecutar las migraciones `mix ecto.migrate`, y la estructura de su base de datos quedará exactamente igual que la nuestra. La fecha de las migraciones es necesaria porque si un desarrollador ya se ha descargado una versión anterior de nuestra aplicación, entonces ya habrá ejecutado algunas migraciones, por tanto, no necesita ejecutar toda la cola de migraciones sino sólo las que se han subido nuevas.

Estas serían las migraciones para un proyecto:

```
dark-mac:migrations fun2src\$ pwd
/Volumes/ExtHDD/fun2src/Proyectos/code/priv/repo/migrations
dark-mac:migrations fun2src\$ ls
20190308131742\_add\_users\_table.exs
20190308132308\_add\_credentials\_table.exs
```

Y este el contenido del primer fichero, por ejemplo:

```
defmodule MusicDB.Repo.Migrations.AddUsersTable do
  use Ecto.Migration

  def change do
    create table(:users) do
      add :name, :string, null: false
      add :email, :string, null: true

      timestamps null: true
    end

    create index(:users, :name)
  end
end
```

B.10.4 Esquemas

Para tratar de explicar el concepto de "esquema" en Ecto, vamos a introducir primero las definiciones de *struct* y `Map`. Un `Map`, en Elixir, es una estructura de datos clave-valor. En Elixir, una *struct* es una extensión, construida en la cima de un `Map`, que nos provee de comprobaciones en tiempo de compilación y valores por defecto. Una *struct* puede pensarse como un `Map` con un número fijo de campos y, en concreto, uno llamado `__struct__` que

contiene el nombre de la *struct*. Debe quedar claro que una *struct* no deja de ser un `Map` y, en consecuencia, todas las operaciones sobre *maps* son aplicables a *structs*.

Un esquema, en Ecto, permite mapear una fuente de datos a una *struct* de Elixir. Lo más frecuente es que dicho mapeo sea de una tabla de la base de datos a una *struct* de Elixir. Esto permite la circulación de datos entre el código Elixir y la base de datos.

La creación de esquemas se realiza por medio del DSL (*Domain Specific Language*) proporcionado por Ecto. Este lenguaje permite, como es de esperar, usar asociaciones para conectar estructuras relacionadas.

Un átomo, en Elixir, es una constante cuyo valor es su propio nombre. En otros lenguajes se suelen denominar *símbolos*. La conversión entre los tipos del esquema y los tipos de la tabla de la base de datos, normalmente, se realiza de forma automática. Cabe destacar que Ecto tiene definidos sus propios tipos, cada uno de los cuales se corresponde con un tipo del lenguaje Elixir. En este contexto, el equipo de Ecto recomienda que los mapas usen cadenas de caracteres como claves y no átomos, ya que cuando se recuperan de la base de datos siempre se retornan como cadenas. No obstante, es posible la creación de tipos personalizados en caso de que los predefinidos no satisfagan las expectativas.

B.10.5 *Changesets*

Las validaciones y las *constraints* son comprobaciones que generan errores en caso de que algo vaya mal. Casi en su totalidad, las validaciones pueden ser ejecutadas sin necesidad de interaccionar con la base de datos, mientras que las *constraints* confían en la base de datos. Por tanto, las validaciones se comprueban antes que las *constraints* y, si una validación falla, las constraints no llegan ni a comprobarse.

Un *changeset* encapsula el proceso de recibir los datos externos, realizar el *casting* de tipos requerido y filtrado de valores que no son de interés, realizar las validaciones pertinentes y comprobar las *constraints* definidas, antes de llevar a cabo la escritura a la base de datos. En conclusión, este mecanismo asegura que se seleccionan los datos correctos, del tipo adecuado, que cumplen ciertas propiedades y, solo cuando todo lo anterior es cierto, se modifica la base de datos.

B.10.6 Módulos de Ecto

En esta sección vamos a describir los principales módulos de Ecto:

- `Ecto.Repo`: Este módulo define un repositorio. El repositorio se mapea a un almacenamiento de datos subyacente, empleando un adaptador y las credenciales adecuadas. De este adaptador depende la capacidad de operar contra un determinado sistema de gestión de bases de datos. El adaptador utilizado, con mayor frecuencia, suele ser el

de PostgreSQL. Un repositorio puede verse como un *wrapper* alrededor de un almacenamiento de datos. Empleando el repositorio se pueden crear, actualizar, eliminar y consultar los datos.

- `Ecto.Schema`: Este módulo se emplea en la definición de esquemas que permiten mapear cualquier fuente de datos a una `struct` de Elixir.
- `Ecto.Changeset`: Los *changesets* se utilizan para filtrar parámetros externos y hacer conversión de tipos. También conforman un mecanismo para hacer un seguimiento de los cambios y validar los mismos, antes de que sean aplicados a los datos.
- `Ecto.Query`: Permite escribir consultas, con sintaxis Elixir, para recuperar información de un repositorio. Las consultas admiten composición, para formar otras más complejas.
- `Ecto.Multi`: Este módulo se emplea para empaquetar operaciones que deberían ser ejecutadas en una única transacción.

B.11 El framework Phoenix

Actualmente, el *framework Phoenix* [37, 109] es la opción más seria en lo que se refiere a desarrollo *backend* de aplicaciones web basadas en Elixir.

B.11.1 Arquitectura de Phoenix

Rigurosamente hablando, Phoenix se define, arquitectónicamente, como un *framework* MVC (Modelo-Vista-Controlador). Podríamos ocupar páginas y páginas hablando de esta arquitectura, pero no lo vamos a hacer por tres buenas razones: existe una amplia literatura al respecto, no vamos a utilizar el *framework* con este enfoque, ergo podría resultar cansino y hasta aburrido.

Cuando pensamos en el *backend* como en un servidor basado en una API GraphQL, tanto la vista, como el controlador, como las *templates*, dejan de tener sentido y resulta más recomendable tratar de entender el *backend* desde el foco proporcionado por el esquema GraphQL.

Personalmente, como autor de este trabajo, y aún a riesgo de que muchos puristas discrepen, prefiero concebir Phoenix como una arquitectura de *pipeline* de funciones.

En Phoenix, existe una estructura de datos, denominada `Plug.Conn`, que representa el universo completo para una solicitud dada (protocolo, parámetros, etc.). Un *Plug* es una función que consume un `Plug.Conn`, realiza alguna tarea y retorna otro `Plug.Conn` modificado. Una solicitud entra en el sistema por un *endpoint*, se le pasa al *router* y, en el caso de nuestro sistema, este último la encamina hacia un *pipeline* de Plugs, que irán generando estructuras `Plug.Conn` modificadas, siendo la última la respuesta a retornar al cliente.

En resumen, el *backend* de nuestra aplicación web es, por definición, un *pipeline* de funciones.

B.11.2 Contextos

En Phoenix, se denomina *Contexto* a una capa que envuelve la lógica de negocio, permitiendo abstraerse de los detalles del *backend*. Los contextos son módulos que agrupan funcionalidad relacionada y la exponen. Por tanto, proporcionan una API.

El uso de contextos permite reducir la complejidad y ocultar los detalles de implementación. En el mejor de los casos, el sistema se divide en partes independientes, más fáciles de comprender y de mantener, cada una accesible por medio de una API pública.

B.11.3 *Channels*

Para problemas altamente interactivos, Phoenix proporciona *Channels*. Esto es, cada cliente se conecta, directamente, con un proceso (*channel*) en el servidor.

Un *channel* se puede pensar como una conversación, en la que el mismo proceso envía/recibe mensajes y conserva el estado. Una conversación siempre es sobre un tema (*topic*). Cada conversación de un usuario, sobre un tema, tiene su propio proceso independiente. Esto garantiza que, en caso de fallo, no afecta al resto de suscriptores. Además, esto escala bien, porque Elixir gestiona, perfectamente, millones de procesos simultáneos. Lo único por lo que hay que preocuparse es por: abrir/cerrar conexiones y enviar/recibir mensajes. Recordemos que en la BEAM, los procesos son primitivas de concurrencia y no tienen nada que ver con los procesos del sistema operativo.

En concreto, los *channels* se emplean para implementar las operaciones de suscripción de GraphQL, que permiten la implementación de notificaciones en tiempo real blando. Por sí solos, los *channels* no son suficientes, para dicha implementación, y deben hacer uso del sistema PubSub (publicación/suscripción) del *Phoenix framework*.

B.12 Pruebas en Elixir

El ecosistema Elixir dispone de su propia librería para realizar pruebas unitarias: ExUnit [23]. Se trata de una librería muy simple y, fundamentalmente, enfocada solo a dicho tipo de pruebas. Si se desean realizar pruebas de integración, lo normal es recurrir a la librería Common Test [24] de Erlang.

Las pruebas de un proyecto generado con la herramienta *Mix* de Elixir, suelen residir en el directorio *test*, en ficheros que concuerdan con el patrón `*_test.exs` y pueden ejecutarse con `mix test`.

En Elixir, cuando se documenta una función de un módulo, es posible escribir, en dicha documentación, ejemplos de cómo sería su invocación en el intérprete y de los resultados esperados. ExUnit, permite la prueba de dichos ejemplos mediante `doctest`.

ExUnit implementa las pruebas como *scripts* y permite la ejecución de múltiples módulos de pruebas en paralelo. Eso sí, los *tests* dentro del mismo módulo son siempre síncronos. Cada *test* ejecuta en un proceso separado evitando, de esta forma, la compartición de datos por medio del diccionario de proceso.

ExUnit permite agrupar múltiples *tests* utilizando bloques `describe`. La librería proporciona las siguientes *callbacks*:

- `setup_all`: Define una *callback* que será ejecutada antes de cualquier *test* en un caso de prueba.
- `setup`: Define una *callback* que será ejecutada antes de cada *test* en un caso de prueba.
- `on_exit`: Define una *callback* que será ejecutada una vez que el *test* finalice.

Las pruebas se basan en diferentes macros. Las más empleadas son:

- `assert`: Se usa para probar que una expresión es verdadera. En caso contrario, se lanza un error y la prueba falla.
- `refute`: Se emplea para probar que una expresión es falsa. En caso contrario, se lanza un error y la prueba falla.
- `assert_raise`: Se utiliza para verificar que se ha lanzado un determinado error.
- `assert_receive`: Se usa para probar los mensajes que se envían los distintos procesos.

La librería Common Test de Erlang es, infinitamente, más potente que ExUnit. Permite, por ejemplo, un control muy personalizado sobre el orden y paralelización de los *tests* dentro de grupos, permite pruebas distribuidas, genera informes hiperdetallados, permite definir dependencias entre *tests*, facilita trabajar de forma simple con muchos protocolos de red, etc. Se trata de una librería de lo más completo que se puede encontrar hoy en día, pero su buen manejo requiere de mucho tiempo y dedicación. De lo contrario, su empleo resultar peor que su omisión.

Si lo que se desea son pruebas basadas en propiedades (*property-based testing*, y no simples aserciones, existe la librería StreamData [110].

Apéndice C

Software y técnicas utilizadas

En este capítulo se describen, brevemente, tanto el software como algunas de las técnicas empleadas durante el desarrollo del proyecto.

C.1 Importación de ficheros cabecera de Erlang

Como es ampliamente conocido, Elixir permite el uso de funciones definidas en módulos Erlang directamente. Desafortunadamente, la importación directa de ficheros cabecera de Erlang (*.hrl) no está soportada. Por suerte existen un par de soluciones que, aunque no son directas, son fáciles de comprender e implementar.

La primera solución consiste en crear un módulo Erlang que importe los contenidos del fichero cabecera, pongamos valores de constantes, y exporte dichos contenidos como valores de retorno de funciones. Posteriormente, es suficiente con emplear las funciones del módulo, desde Elixir, para obtener los contenidos de los archivos ".hrl".

Una segunda solución se sustenta en el uso del módulo "epp_dodger" de Erlang. Este módulo permite parsear el código fuente de Erlang sin expandir las directivas del procesador. En concreto, la función de interés es parse_file/1, que nos permite procesar el fichero cabecera objetivo.

```
1 with {:ok, forms} <- :epp_dodger.parse_file(file) do
2   for {:tree, :attribute, _, {_, {:atom, _, :define}, [{_, _,
      name}, {_, _, value}]}} <- forms,
3     do: {name, value}
4 end
```

Listing C.1: Importación de constantes desde fichero .hrl

```elixir
def load_list_of_constants(path) do
  path
  |> File.read!()
  |> Enum.filter(&String.starts_with(&1, "-define("))
  |> Enum.map(&parse_and_extract_values/1)
  |> Map.new()
end
```

Listing C.2: Importación de constantes desde fichero .hrl con Elixir

C.2 Guardado automático de sesiones iex/erl

Durante el trabajo con Erlang/Elixir, es común que surja la necesidad de grabar una sesión interactiva en un fichero de texto plano. Para grabar tanto las entradas como las salidas de dichas sesiones, lo más fácil es recurrir al comando BSD "script".

```
script [-adkpqr] [-F pipe] [-t time] [file [command ...]]
```

Listing C.3: Comando script de BSD

El citado comando grabará en un fichero toda la sesión de terminal, hasta que le indiquemos que finalice. El principal inconveniente es que si el terminal no está correctamente configurado, el comando introduce códigos de color, en el fichero de salida, que pueden confundirse fácilmente con basura. Por ello, existen múltiples soluciones. Las más fáciles consisten en redireccionar el fichero generado a herramientas que sean capaces de entender dichos códigos de color, como es el caso de cat o "less -r". Otra solución consiste en configurar el terminal con un tema monocromo. Finalmente, también es posible indicarle a iex que opere sin colores, ejecutando IEx.configure(colors: [enabled: false]) dentro de la sesión o poniendo dicha línea en el fichero de configuración ".iex".

C.3 React-Apollo

En lo referente al cliente GraphQL, empleado en el *frontend*, es muy importante indicar que se trata de la versión 2.5. Actualmente, acaba de publicarse la versión 2.6 que, muy posiblemente, solucione alguno de los *bugs* comentados en esta memoria. Con esta versión, el enfoque de la librería se ha actualizado para adoptar las últimas novedades de React y, como consecuencia, la documentación ha variado de forma bastante perceptible respecto a la de la versión anterior. En concreto, Apollo ha evolucionado para adaptarse a una nueva característica de React denominada *Hooks*. Los *Hooks*, básicamente, permiten emplear estados sin utilizar clases. Como consecuencia, se obtiene un enfoque funcional mucho más marcado que el existente hasta ahora. De todas formas, debe tenerse en cuenta que no hay planes de eliminar las clases de React, y los *Hooks* son completamente opcionales.

C.4 Erlang, Elixir y Phoenix

Durante el desarrollo de este proyecto se han empleado las últimas liberaciones tanto del lenguaje Erlang como de Elixir. En concreto, el *software* funciona adecuadamente con la liberación 22 de Erlang y la 1.9.1 de Elixir.

La versión del *framework* Phoenix empleada, en todo momento, se corresponde con la 1.4. La más reciente hasta la fecha, aunque dentro de esta, existen parches menores, los cuales no han sido aplicados en su totalidad.

C.5 Editores de programación

Para el desarrollo de este proyecto, prácticamente cualquier editor de programación actual sería viable. Recomendar uno u otro no llevaría a otro lado que a entrar en una *editor war*, análoga a la clásica GNOME vs. KDE, puesto que el editor no deja de ser una decisión muy personal de cada desarrollador. Por ello, solo vamos a comentar que en este proyecto se ha utilizado Visual Studio Code [111] en MacOS y Emacs [112] bajo Linux. La decisión no tiene más base que la de la comodidad en los mapeos de teclado bajo los dos sistemas operativos. En lo referente a las extensiones disponibles para edición Elixir, simplemente recomendar ElixirLS [113], aunque personalmente recomiendo monitorizar el consumo de CPU de la misma.

C.6 Monitorización del sistema Erlang

Para hacer un seguimiento del estado del sistema Erlang, en un momento dado, se ha empleado la herramienta *observer*. Dicho software nos permite visualizar información general del sistema, árboles de supervisión, información de procesos, etc., empleando una interfaz gráfica. Como Elixir emplea la BEAM, la herramienta es igualmente válida para sistemas Elixir.

C.7 Navegadores web y extensiones

Para el desarrollo del *frontend* se ha empleado la versión estable en cada momento de dicho navegador. La última versión probada fue la 76. Para este navegador se han instalado y configurado las herramientas de desarrollo Redux y React. También se ha hecho un uso intensivo de las propias herramientas para desarrollo web que vienen integradas en el navegador por defecto.

Por otro lado, también se ha empleado el navegador Safari Developer Preview para Mac OS Mojave 10.14.6, con el fin de comprobar que el *frontend* tiene una interfaz de usuario coherente, independientemente del navegador.

Apéndice D

Interfaz gráfica de usuario de la aplicación

Este capítulo contiene algunas capturas de pantalla de la interfaz gráfica de usuario de la aplicación desarrollada.

- En la figura D.1 se puede visualizar la pantalla de *login* para los usuarios del sistema.
- En la figura D.2 puede verse el aspecto de la GUI ante un intento de *login* fallido.
- En la figura D.3 se observa la pantalla de registro de usuarios en la aplicación.
- En la figura D.4 se aprecia el *dashboard* para las listas de tareas.
- En la figura D.5 se ve el *widget*, de navegación entre entidades, desplegado.
- En la figura D.6 se puede observar el aspecto de una lista de tareas para un usuario con permiso de solo lectura.
- La figura D.7 se corresponde con la pantalla de edición/creación de listas de tareas de la aplicación.
- En la figura D.8 se muestra la forma de añadir/eliminar colaboradores a una lista de tareas.
- En la figura D.9 se expone la manera de añadir/eliminar etiquetas a una lista de tareas.
- En la figura D.10 se puede ver cómo borrar una lista de tareas de la cuenta de un usuario.
- En la figura D.11 se ve el *dashboard* para las pizarras.
- En la figura D.12 se muestra la pantalla de edición, individual o compartida, de una pizarra.

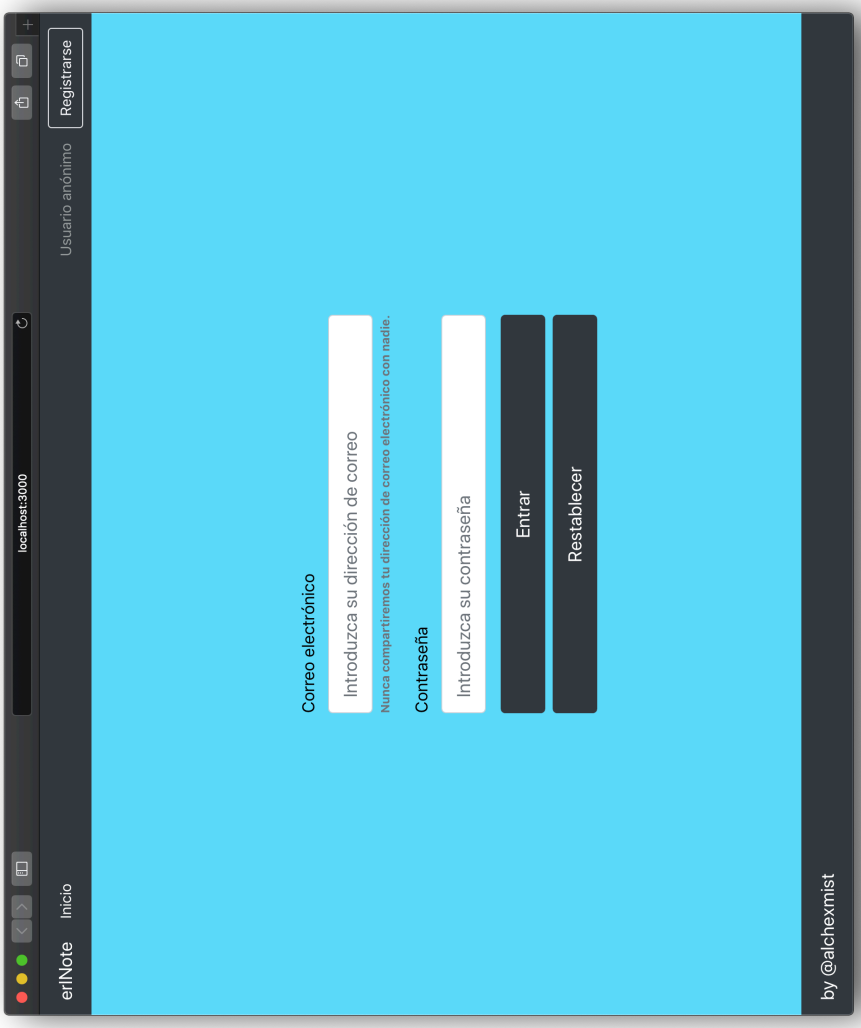

Figura D.1: Pantalla de acceso a la aplicación

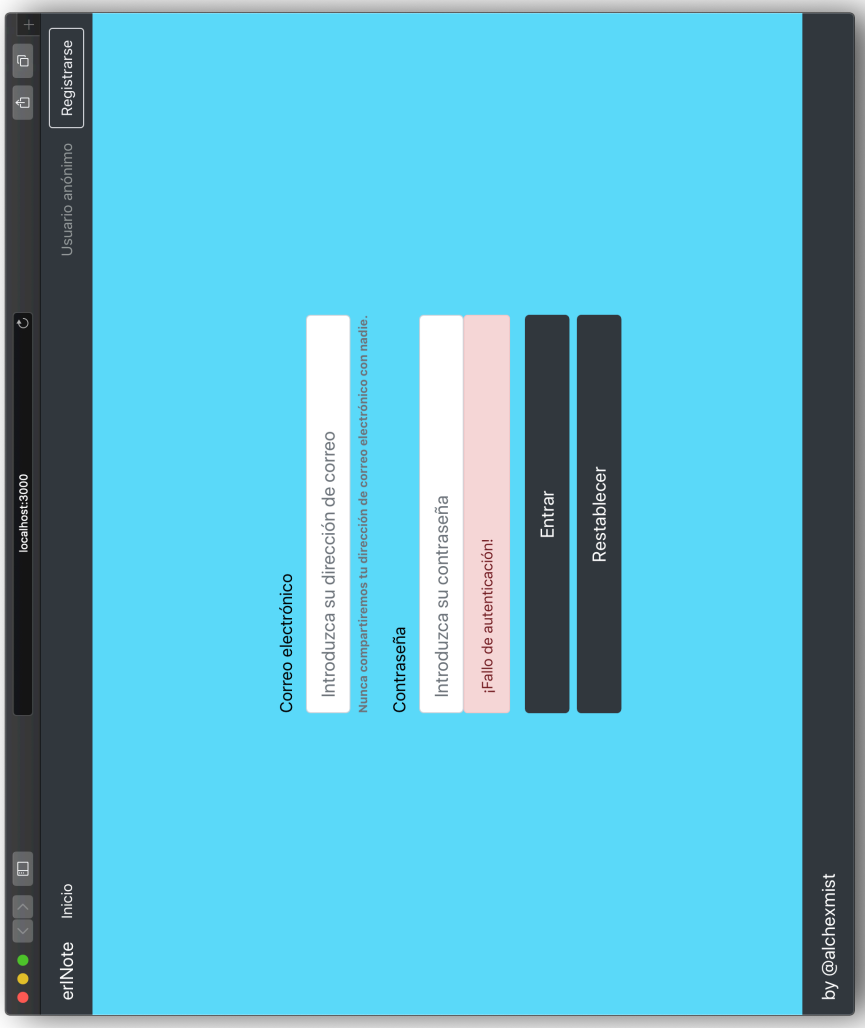

Figura D.2: Fallo de autenticación en el sistema

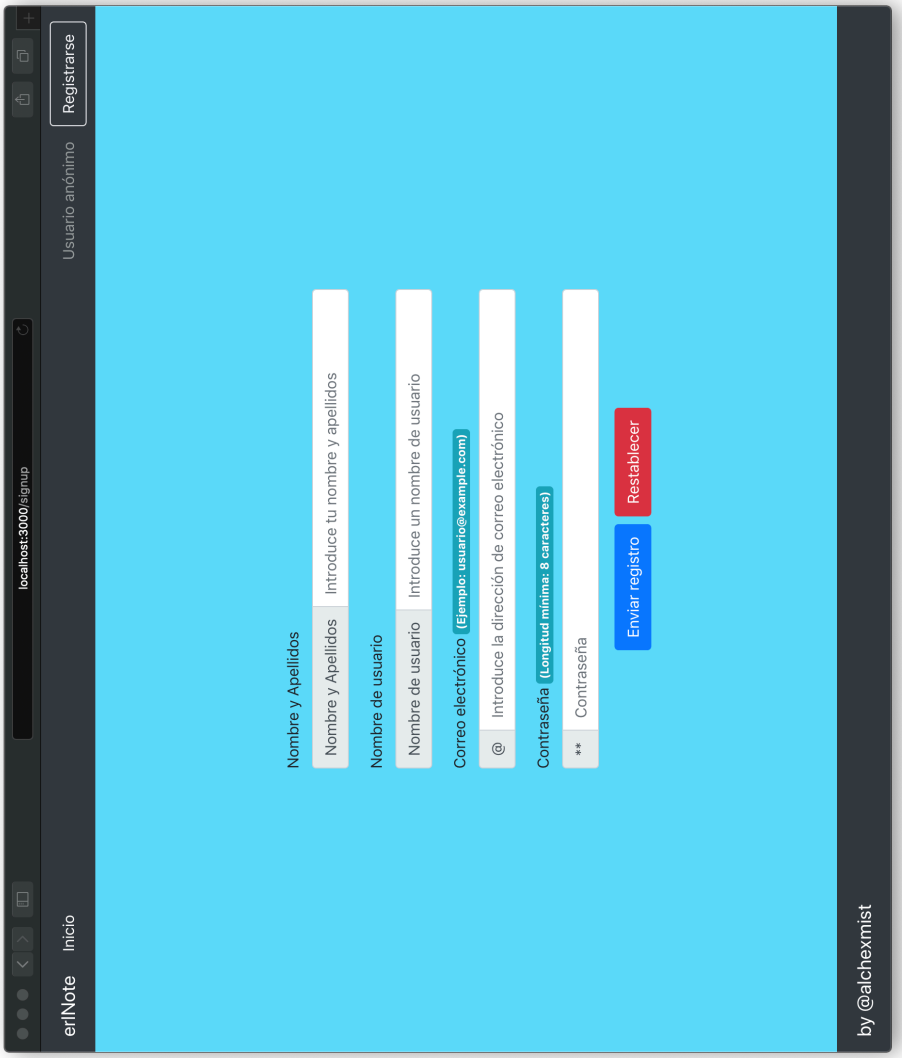

Figura D.3: Registro de usuarios en el sistema

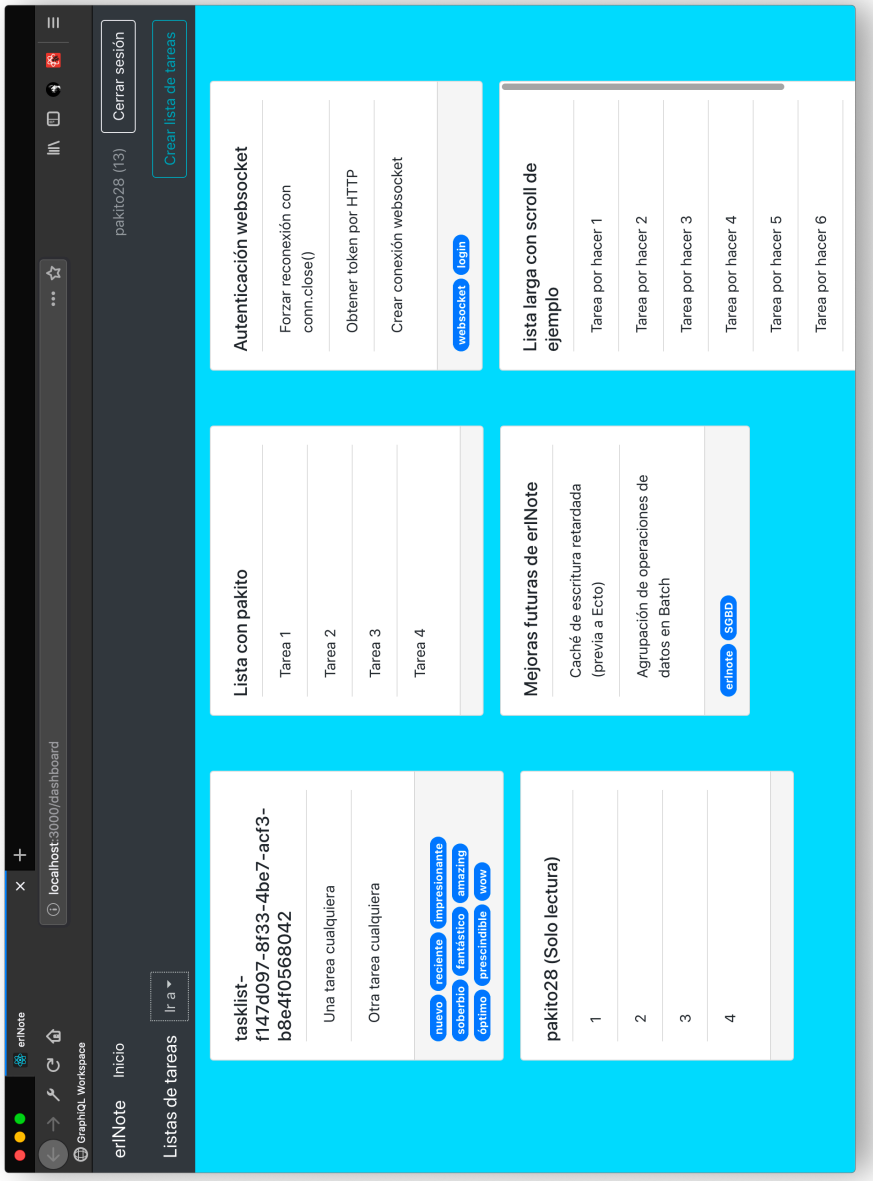

Figura D.4: *Dashboard* para listas de tareas

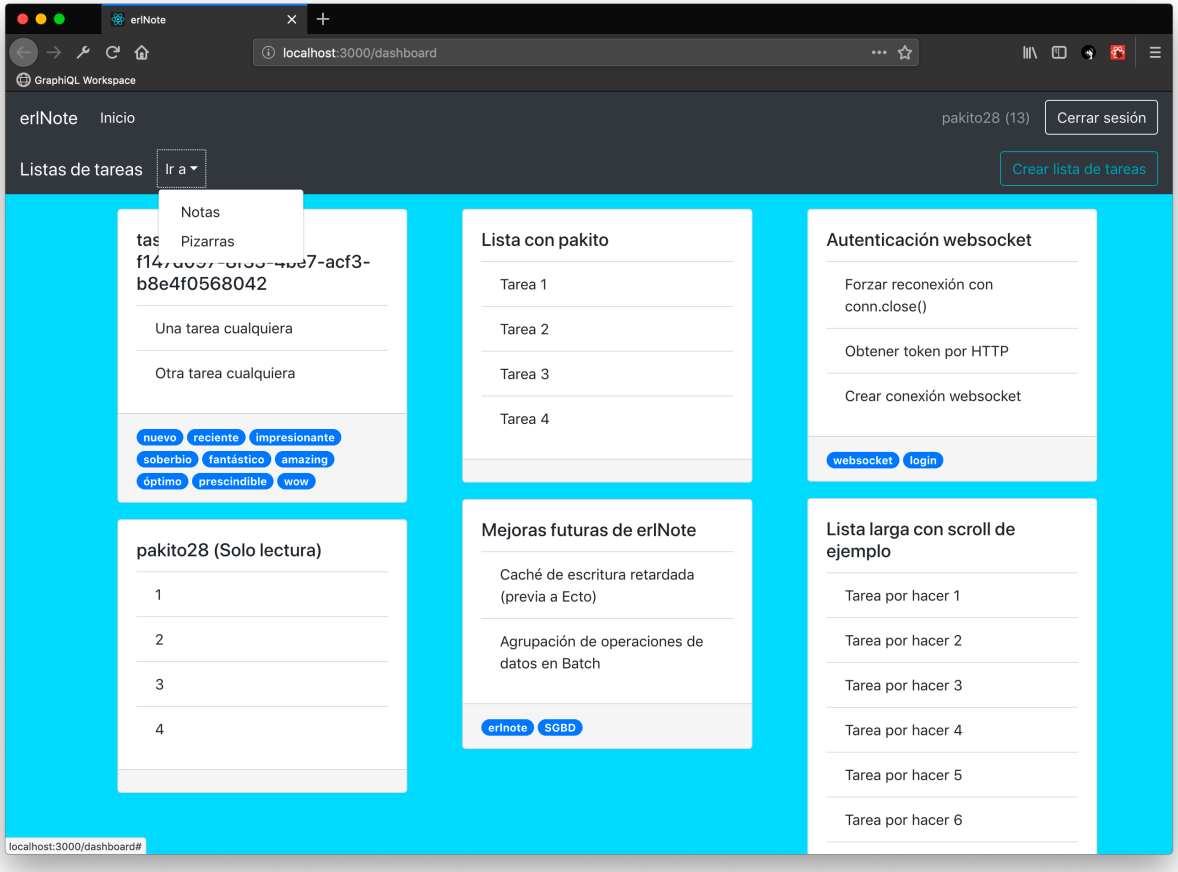

Figura D.5: *Widget* de navegación entre entidades

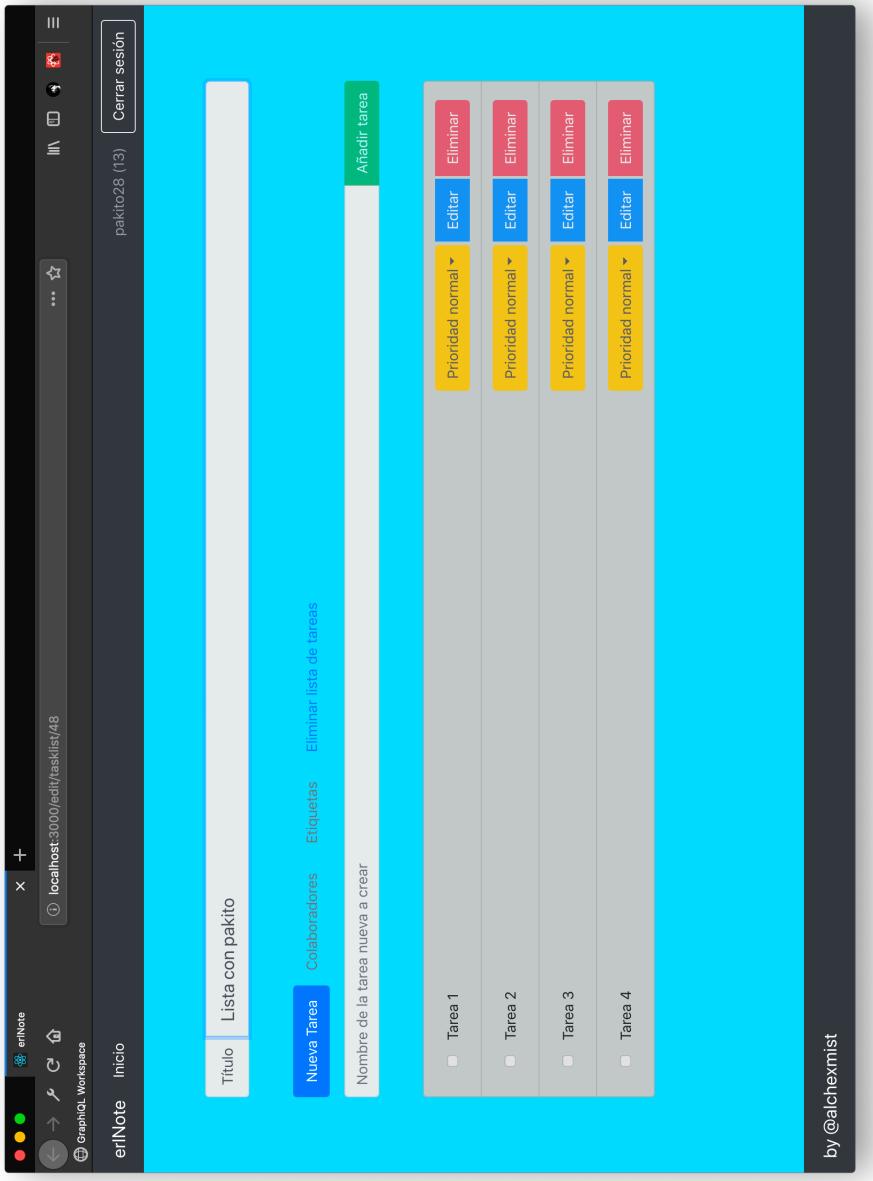

Figura D.6: Lista de tareas de solo lectura

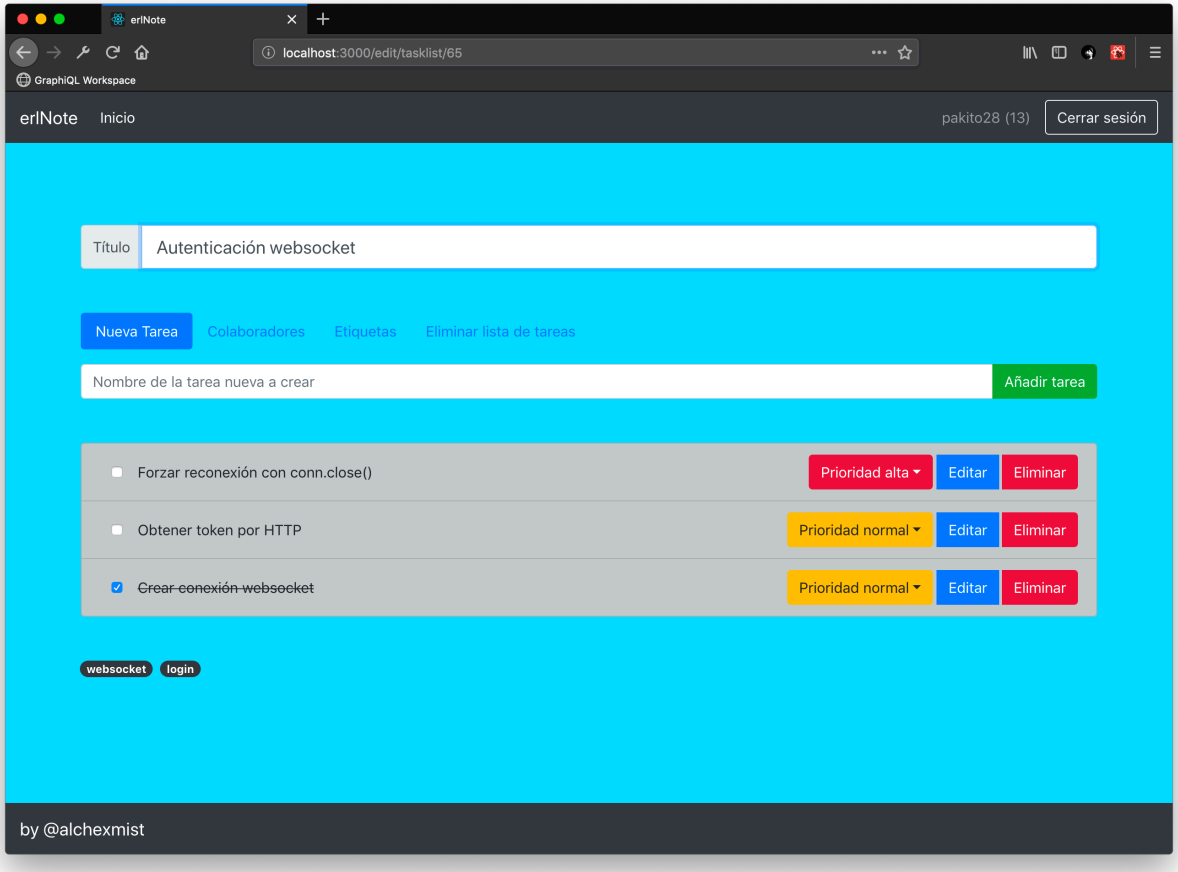

Figura D.7: Creación/edición de listas de tareas

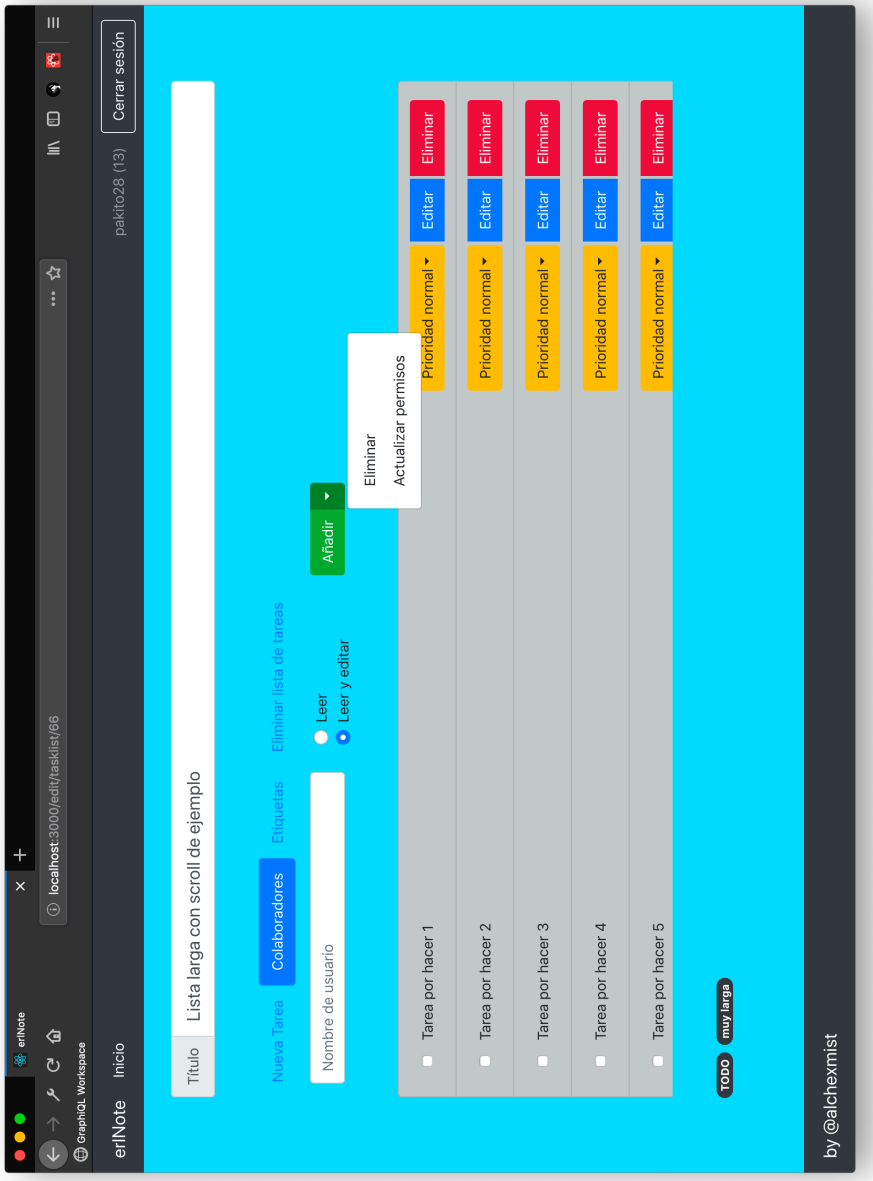

Figura D.8: Adición/eliminación de colaboradores

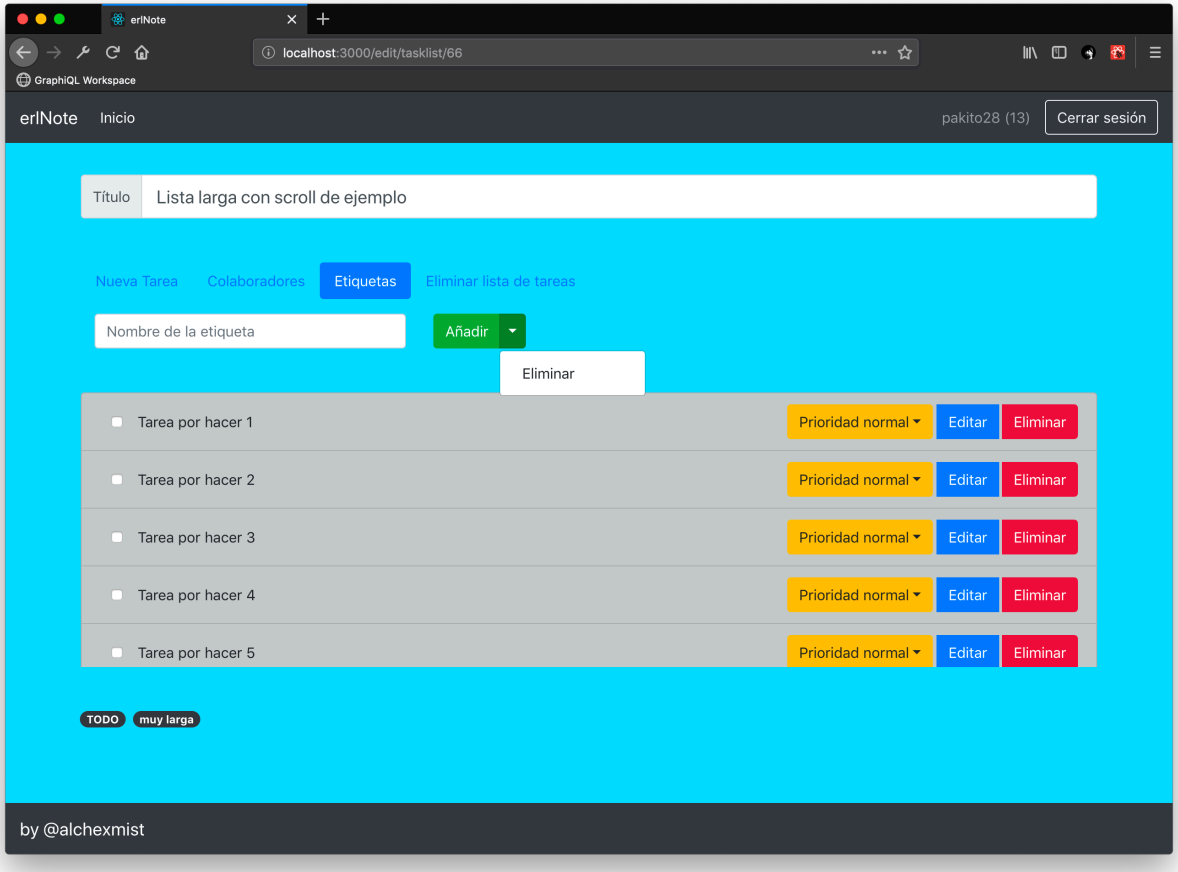

Figura D.9: Adición/eliminación de etiquetas

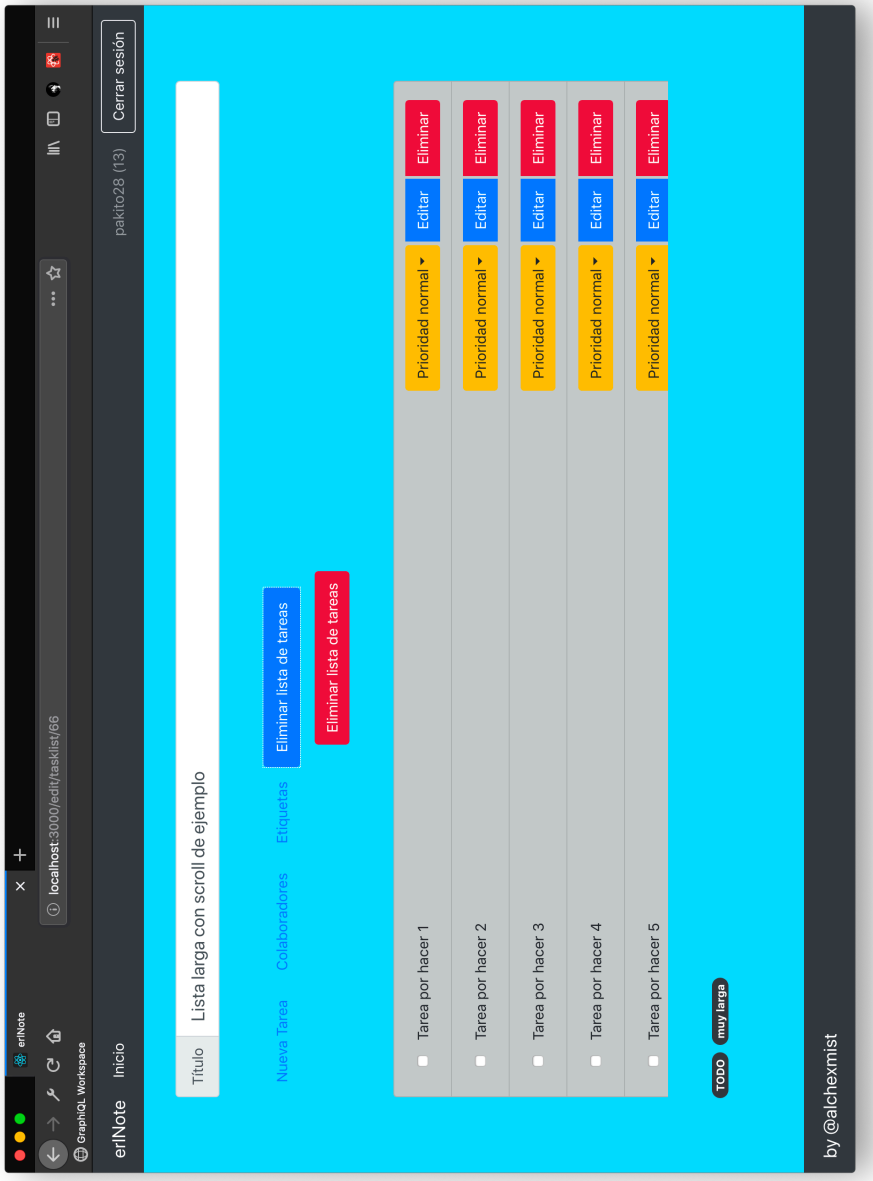

Figura D.10: Borrado de una lista de tareas

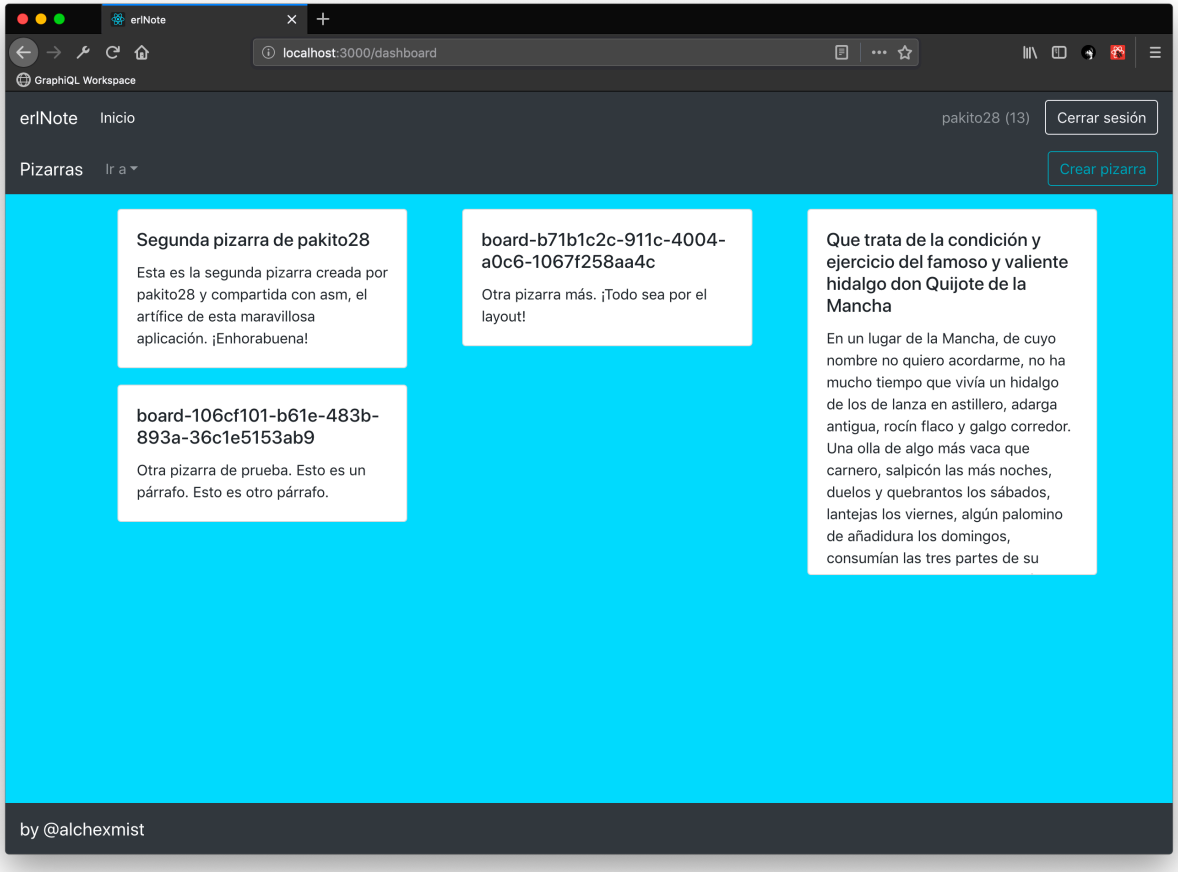

Figura D.11: *Dashboard* de pizarras

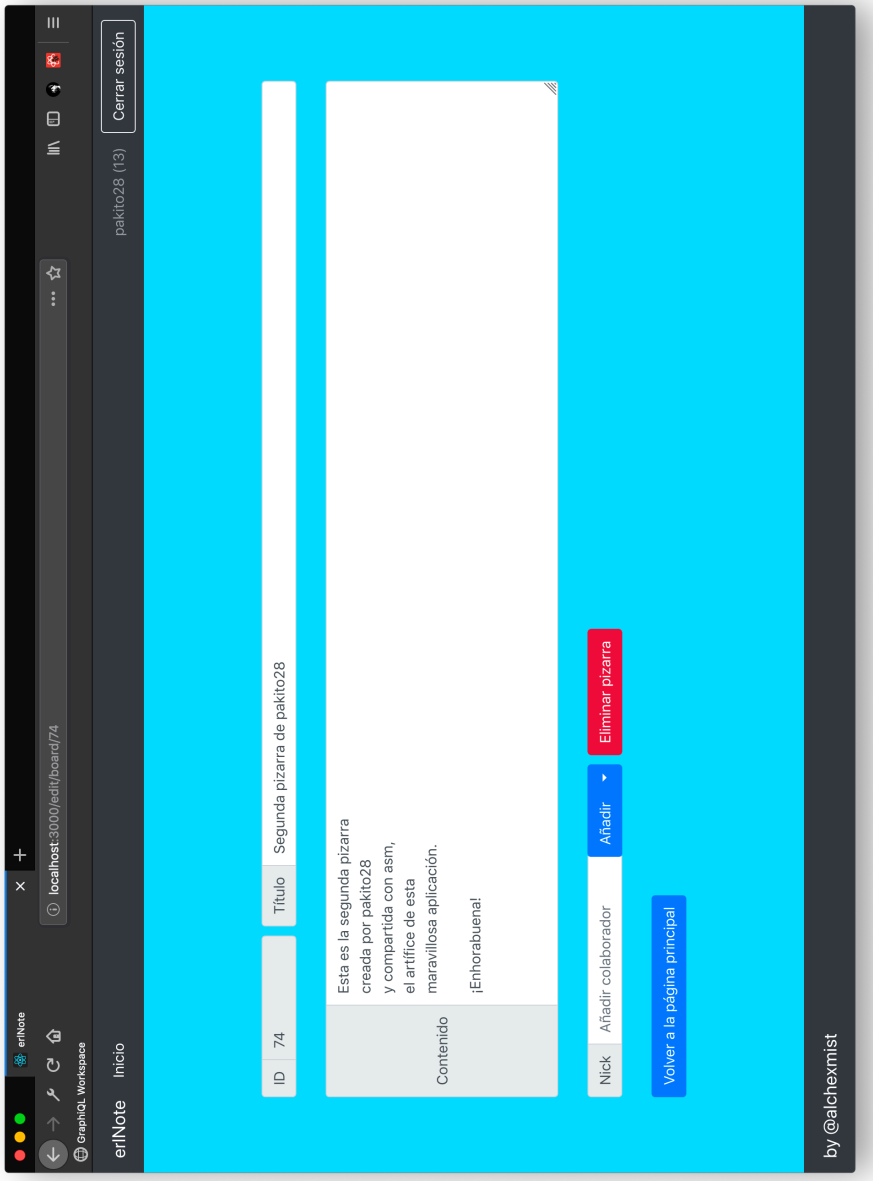

Figura D.12: Creación/edición de pizarras

Apéndice E

GraphiQL

Este capítulo contiene algunas capturas de pantalla de la interfaz gráfica de usuario de GraphiQL.

La figura E.1 muestra una consulta ejecutada en GraphiQL, de una usuario ya registrado en el sistema.

La figura E.2 ilustra la mutación de *login*, para un usuario anónimo, y el uso de la sección *Query variables*.

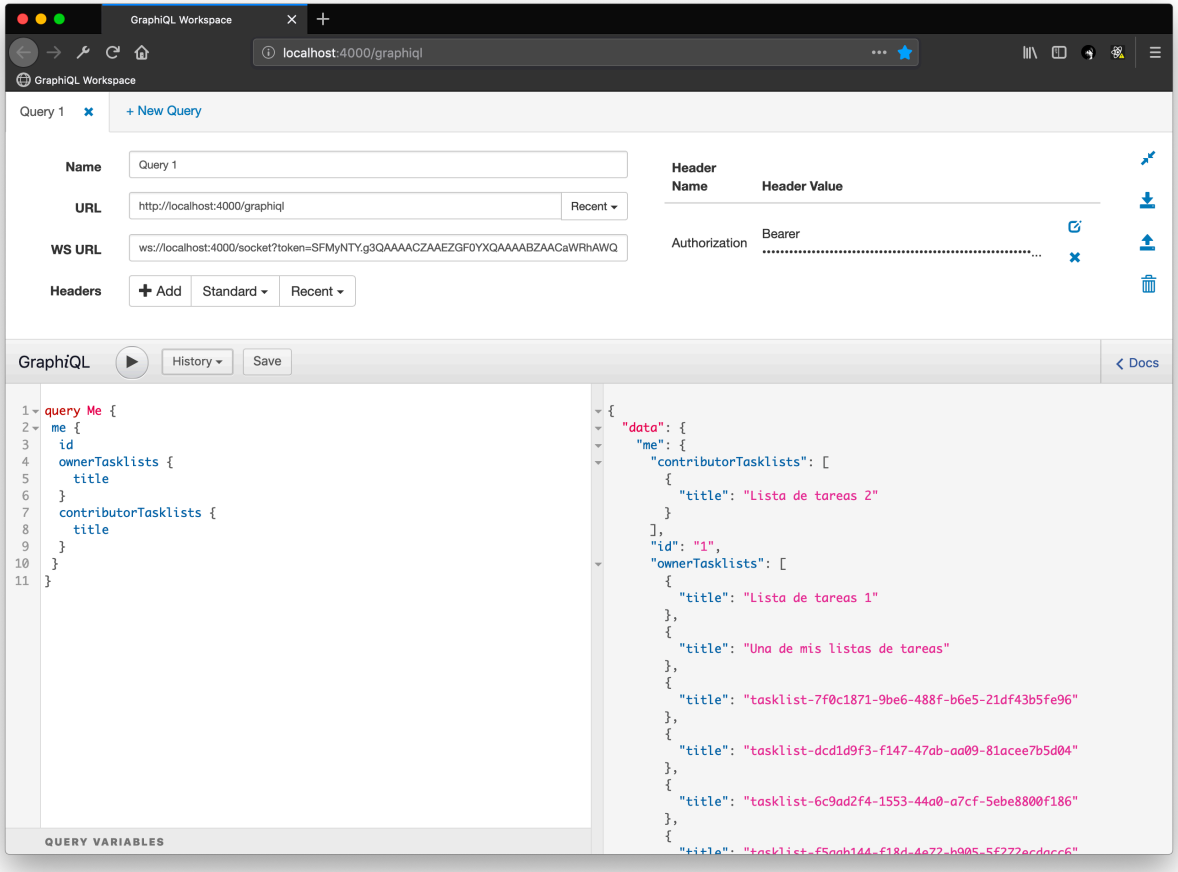

Figura E.1: Consulta ejecutada en GraphiQL

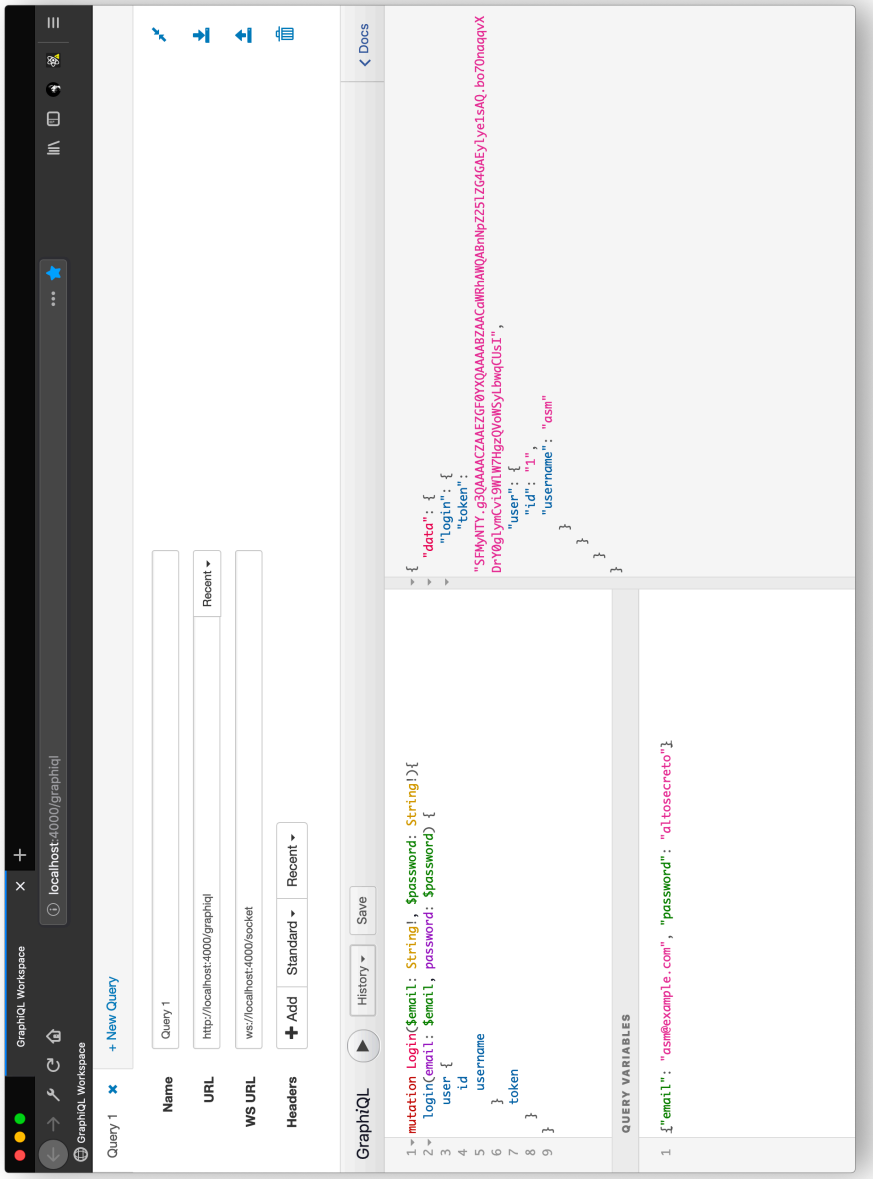

Figura E.2: Mutación ejecutada en GraphiQL

Apéndice F

Redux Developer Tools

Este capítulo contiene una captura de pantalla F.1 de las herramientas para desarrolladores Redux, en ejecución. En ella se puede apreciar el estado de la aplicación y las acciones ejecutadas, hasta un instante determinado.

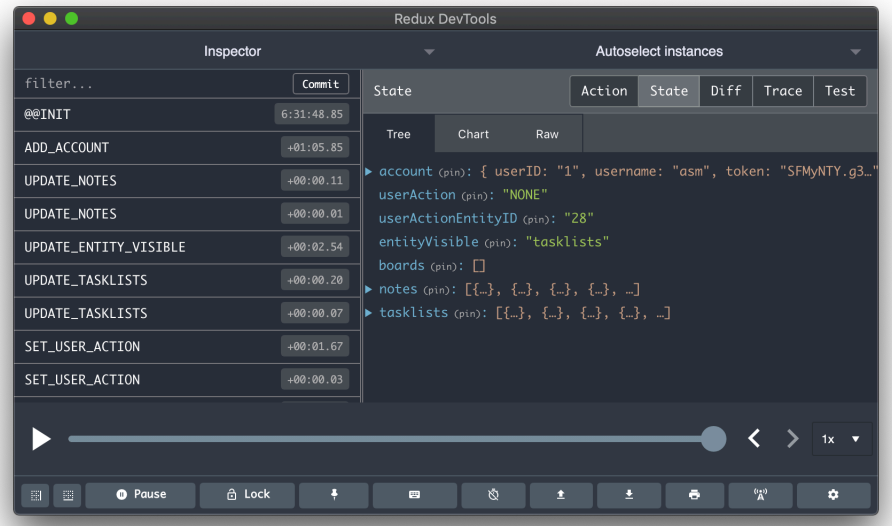

Figura F.1: *Redux Developer Tools*

Apéndice G

Ejemplos de operaciones GraphQL

Este capítulo contiene algunos ejemplos de las operaciones GraphQL implementadas. Se suministran algunos ejemplos de mutaciones, consultas y suscripciones. En el caso de las suscripciones, no existe respuesta alguna.

```
mutation CreateUserAccount($accountData: UserAccountInput!) {
    userAccount: createUserAccount(input: $accountData) {
      id
      name
      username
      credentials {
        email
        password_hash
      }
    }
  }
  # QUERY VARIABLES
  {
    "accountData": {
      "username": "whitehat",
      "name": "White Hat",
      "credentials": [
        {
          "password": "12345678910",
          "email": "whitehat@example.com"
        }
      ]
    }
  }
```

```
mutation CreateBoard {
  board: createBoard {
    id
    title
    text
  }
}
# RESPONSE
{
  "data": {
    "board": {
      "id": "20",
      "text": null,
      "title": "board-89ee2081-c177-43e3-83bd-7f478e12fef9"
    }
  }
}
```

```
mutation {
  login(email: "asm@example.com", password: "altosecreto") {
    user {
      id
      username
    }
    token
  }
}
# RESPONSE
{
  "data": {
    "login": {
      "token":
    "SFMyNTY.g3QAAAACZAAEZGF0YXQAAAABZAACaWRhAWQABnNpZ...",
      "user": {
        "id": "1",
        "username": "asm"
      }
    }
  }
}
```

```
mutation UpdateBoard($boardData: UpdateBoardInput!) {
  board: updateBoard(input: $boardData) {
    id
    text
    title
  }
}
# QUERY VARIABLES
{
  "boardData": {
    "id": "40",
    "text": "Hola que tal!",
    "title": "Tablero actualizado"
}
# RESPONSE
{
  "data": {
    "board": {
      "id": "40",
      "text": "Hola que tal!",
      "title": "Tablero actualizado"
    }
  }
}
```

```
  subscription TasklistUpdated($tasklistId: ID!) {
    tasklistUpdated(tasklistId: $tasklistId) {
      id
      title
      tasks {
        id
        name
        description
        state
        priority
        startDatetime
        endDatetime
      }
      tags {
        id
        name
      }
      updatedBy
    }
  }
```

```
query GetAccessInfo($entityId: ID!) {
    getAccessInfo(entityId: $entityId, entityType: TASKLIST) {
      ... on TasklistAccessInfo {
        ownerId
        userId
        canRead
        canWrite
        tasklistId
      }
    }
  }

#RESPONSE
{
  "data": {
    "getAccessInfo": {
      "__typename": "TasklistAccessInfo",
      "canRead":true,
      "canWrite":true,
      "ownerId":"2",
      "tasklistId":"2",
      "userId":"1"
    }
  }
}
```

Apéndice H

Schema y *Changeset* Ecto

Este capítulo contiene un ejemplo de código en el que se muestra la definición tanto de un esquema Ecto, como de un *changeset*.

```
defmodule Erlnote.Accounts.User do
  use Ecto.Schema
  import Ecto.Changeset

  alias Erlnote.Accounts.Credential
  alias Erlnote.Boards.{Board, BoardUser}
  alias Erlnote.Notes.{Notepad, Note, NoteUser}
  alias Erlnote.Tasks.{Tasklist, TasklistUser}

  # If your :join_through is a schema, your join table may be structured as
  # any other table in your codebase, including timestamps. You may define
  # a table with primary keys.

  schema "users" do
    field :name, :string
    field :username, :string
    has_many :credentials, Credential, on_replace: :delete
    # Los hijos los añadimos con build_assoc.
    has_many :owner_boards, Board, foreign_key: :owner, on_replace: :delete
    has_many :notepads, Notepad, on_replace: :delete
    has_many :notes, Note, on_replace: :delete
    has_many :owner_tasklists, Tasklist, on_replace: :delete
    many_to_many :boards, Board, join_through: BoardUser
    many_to_many :collaborator_notes, Note, join_through: NoteUser
    many_to_many :tasklists, Tasklist, join_through: TasklistUser

```

```
    timestamps(type: :utc_datetime)
  end

  @doc false
  def registration_changeset(user, params) do
    user
    |> changeset(params)
    |> cast_assoc(:credentials, with: &Credential.changeset/2,
    required: true)
  end

  @doc false
  def changeset(user, attrs) do
    user
    |> cast(attrs, [:name, :username])
    |> validate_required([:name, :username])
    |> validate_length(:name, min: 1, max: 255)
    |> validate_length(:username, min: 1, max: 50)
    |> unique_constraint(:username)
  end
end
```

Apéndice I

Licencia *software* utilizada

A continuación se muestra la licencia *software* bajo la cual ha sido liberado el software desarrollado en este proyecto.

En el caso de este trabajo se ha optado por una licencia libre altamente permisiva, en concreto la licencia Apache 2.

En lo referente al *software* de terceros empleado en este proyecto, se han elegido siempre *frameworks*, librerías y lenguajes liberados, todos ellos, bajo licencias de software libre, más o menos permisivas. Nótese la diferencia entre, por ejemplo, la GPL v3 y la MIT.

```
1
2                                 Apache License
3                           Version 2.0, January 2004
4                        http://www.apache.org/licenses/
5
6   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION
7
8   1. Definitions.
9
10     "License" shall mean the terms and conditions for use,
    reproduction,
11     and distribution as defined by Sections 1 through 9 of this
    document.
12
13     "Licensor" shall mean the copyright owner or entity
    authorized by
14     the copyright owner that is granting the License.
15
16     "Legal Entity" shall mean the union of the acting entity and
    all
17     other entities that control, are controlled by, or are under
    common
18     control with that entity. For the purposes of this definition,
```

```
19        "control" means (i) the power, direct or indirect, to cause
      the
20        direction or management of such entity, whether by contract or
21        otherwise, or (ii) ownership of fifty percent (50%) or more
      of the
22        outstanding shares, or (iii) beneficial ownership of such
      entity.
23
24        "You" (or "Your") shall mean an individual or Legal Entity
25        exercising permissions granted by this License.
26
27        "Source" form shall mean the preferred form for making
      modifications,
28        including but not limited to software source code,
      documentation
29        source, and configuration files.
30
31        "Object" form shall mean any form resulting from mechanical
32        transformation or translation of a Source form, including but
33        not limited to compiled object code, generated documentation,
34        and conversions to other media types.
35
36        "Work" shall mean the work of authorship, whether in Source or
37        Object form, made available under the License, as indicated
      by a
38        copyright notice that is included in or attached to the work
39        (an example is provided in the Appendix below).
40
41        "Derivative Works" shall mean any work, whether in Source or
      Object
42        form, that is based on (or derived from) the Work and for
      which the
43        editorial revisions, annotations, elaborations, or other
      modifications
44        represent, as a whole, an original work of authorship. For
      the purposes
45        of this License, Derivative Works shall not include works
      that remain
46        separable from, or merely link (or bind by name) to the
      interfaces of,
47        the Work and Derivative Works thereof.
48
49        "Contribution" shall mean any work of authorship, including
50        the original version of the Work and any modifications or
      additions
51        to that Work or Derivative Works thereof, that is
```

```
    intentionally
52    submitted to Licensor for inclusion in the Work by the
    copyright owner
53    or by an individual or Legal Entity authorized to submit on
    behalf of
54    the copyright owner. For the purposes of this definition,
    "submitted"
55    means any form of electronic, verbal, or written
    communication sent
56    to the Licensor or its representatives, including but not
    limited to
57    communication on electronic mailing lists, source code
    control systems,
58    and issue tracking systems that are managed by, or on behalf
    of, the
59    Licensor for the purpose of discussing and improving the
    Work, but
60    excluding communication that is conspicuously marked or
    otherwise
61    designated in writing by the copyright owner as "Not a
    Contribution."
62
63    "Contributor" shall mean Licensor and any individual or Legal
    Entity
64    on behalf of whom a Contribution has been received by
    Licensor and
65    subsequently incorporated within the Work.
66
67  2. Grant of Copyright License. Subject to the terms and
    conditions of
68    this License, each Contributor hereby grants to You a
    perpetual,
69    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
70    copyright license to reproduce, prepare Derivative Works of,
71    publicly display, publicly perform, sublicense, and
    distribute the
72    Work and such Derivative Works in Source or Object form.
73
74  3. Grant of Patent License. Subject to the terms and conditions
    of
75    this License, each Contributor hereby grants to You a
    perpetual,
76    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
77    (except as stated in this section) patent license to make,
    have made,
78    use, offer to sell, sell, import, and otherwise transfer the
```

```
Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least
```

```
   one
112        of the following places: within a NOTICE text file
   distributed
113        as part of the Derivative Works; within the Source form or
114        documentation, if provided along with the Derivative
   Works; or,
115        within a display generated by the Derivative Works, if and
116        wherever such third-party notices normally appear. The
   contents
117        of the NOTICE file are for informational purposes only and
118        do not modify the License. You may add Your own
   attribution
119        notices within Derivative Works that You distribute,
   alongside
120        or as an addendum to the NOTICE text from the Work,
   provided
121        that such additional attribution notices cannot be
   construed
122        as modifying the License.
123
124      You may add Your own copyright statement to Your
   modifications and
125      may provide additional or different license terms and
   conditions
126      for use, reproduction, or distribution of Your modifications,
   or
127      for any such Derivative Works as a whole, provided Your use,
128      reproduction, and distribution of the Work otherwise complies
   with
129      the conditions stated in this License.
130
131   5. Submission of Contributions. Unless You explicitly state
   otherwise,
132      any Contribution intentionally submitted for inclusion in the
   Work
133      by You to the Licensor shall be under the terms and
   conditions of
134      this License, without any additional terms or conditions.
135      Notwithstanding the above, nothing herein shall supersede or
   modify
136      the terms of any separate license agreement you may have
   executed
137      with Licensor regarding such Contributions.
138
139   6. Trademarks. This License does not grant permission to use the
   trade
```

```
   names, trademarks, service marks, or product names of the
Licensor,
   except as required for reasonable and customary use in
describing the
   origin of the Work and reproducing the content of the NOTICE
file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express
or
   implied, including, without limitation, any warranties or
conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for
determining the
   appropriateness of using or redistributing the Work and
assume any
   risks associated with Your exercise of permissions under this
License.

8. Limitation of Liability. In no event and under no legal
theory,
   whether in tort (including negligence), contract, or
otherwise,
   unless required by applicable law (such as deliberate and
grossly
   negligent acts) or agreed to in writing, shall any
Contributor be
   liable to You for damages, including any direct, indirect,
special,
   incidental, or consequential damages of any character arising
as a
   result of this License or out of the use or inability to use
the
   Work (including but not limited to damages for loss of
goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While
redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty,
```

```
    indemnity,
169   or other liability obligations and/or rights consistent with
    this
170   License. However, in accepting such obligations, You may act
    only
171   on Your own behalf and on Your sole responsibility, not on
    behalf
172   of any other Contributor, and only if You agree to indemnify,
173   defend, and hold each Contributor harmless for any liability
174   incurred by, or claims asserted against, such Contributor by
    reason
175   of your accepting any such warranty or additional liability.
176
177 END OF TERMS AND CONDITIONS
178
179 APPENDIX: How to apply the Apache License to your work.
180
181   To apply the Apache License to your work, attach the following
182   boilerplate notice, with the fields enclosed by brackets "[]"
183   replaced with your own identifying information. (Don't include
184   the brackets!)  The text should be enclosed in the appropriate
185   comment syntax for the file format. We also recommend that a
186   file or class name and description of purpose be included on
    the
187   same "printed page" as the copyright notice for easier
188   identification within third-party archives.
189
190 Copyright [yyyy] [name of copyright owner]
191
192 Licensed under the Apache License, Version 2.0 (the "License");
193 you may not use this file except in compliance with the License.
194 You may obtain a copy of the License at
195
196     http://www.apache.org/licenses/LICENSE-2.0
197
198 Unless required by applicable law or agreed to in writing,
     software
199 distributed under the License is distributed on an "AS IS" BASIS,
200 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
     implied.
201 See the License for the specific language governing permissions
     and
202 limitations under the License.
```

Lista de acrónimos

API *Application Programming Interface.*

BEAM *Bogdan's Erlang Abstract Machine.*

CSS *Cascading Style Sheets.*

DOM *Document Object Model.*

ERLANG/OTP *Erlang Open Telecom Platform.*

GUI *Graphical User Interface.*

HTTP *Hypertext Transfer Protocol.*

JSON *JavaScript Object Notation.*

JSX *JavaScript XML.*

LESS *Leaner Style Sheets.*

MVVM *Modelo-Vista-ModeloVista.*

Sass *Syntactically Awesome Stylesheets.*

SGBD *Sistema de gestión de bases de datos.*

SPA *Single Page Application.*

UUID *Universally Unique Identifier.*

Glosario

Aplicación web progresiva Aplicación web que carga como una página web normal pero ofrece funcionalidades típicas de aplicaciones nativas (trabajo *offline*, notificaciones *push*, ...). Las aplicaciones web progresivas se caracterizan por tener una alta fiabilidad, responder rápidamente a las interacciones con el usuario y dar la sensación de ser una aplicación propia del dispositivo.

Backend En un modelo cliente-servidor, el servidor es, normalmente, considerado el *backend*. En Ingeniería del Software, el *backend* se corresponde con la capa de acceso a datos.

BEAM Máquina virtual Erlang, parte del sistema en tiempo de ejecución, que ejecuta bytecode compilado.

Bytecode Código independiente de la máquina que generan compiladores de determinados lenguajes (Java, Erlang,...) y que es ejecutado por el intérprete correspondiente.

CORS El Intercambio de Recursos de Origen Cruzado o *Cross-origin Resource Sharing* es un mecanismo que emplea cabeceras HTTP adicionales para permitir acceder a recursos restringidos en otro dominio fuera del dominio desde el cual se sirvió el primer recurso.

CSS Flexbox Modelo unidimensional (que maneja una sola fila o columna de forma simultánea) de *layout* o disposición de contenido. Este módulo nos dota de un método para distribuir el espacio entre los ítems de una interfaz de usuario y gestionar las alineaciones.

Data binding Forma de asociar los campos de la interfaz de usuario a un modelo de datos.

Diseño web adaptativo Se denomina diseño web adaptable, también conocido como adaptativo o responsivo, a una filosofía de diseño y desarrollo que tiene como objetivo adaptar la apariencia de las páginas web en función del dispositivo, y sus características, que

se utilice para cargarlas. El diseño web adaptativo se hizo posible gracias a la creación de las *media queries*.

Endpoint Un extremo de un canal de comunicación. Un punto de entrada a un servicio o proceso.

Frontend En un modelo cliente-servidor, el cliente es, normalmente, considerado el *frontend*. En Ingeniería del Software, el *frontend* se corresponde con la capa de presentación.

Less Lenguaje dinámico de hojas de estilos.

Media query Módulo CSS3 (hojas de estilo en cascada) que permite adaptar la representación del contenido a las características del dispositivo.

One way binding Ver *One way data flow*.

One way data flow Un patrón de datos unidireccional surge se da cuando no existen referencias mutables entre partes con distinta funcionalidad. Por ejemplo, si tenemos un modelo y una vista, los datos solo podrán fluir desde el modelo a la vista, y nunca al revés. Dicho de otra manera, cambios en el modelo causarán cambios en la vista, pero la vista nunca puede modificar el modelo directamente. En un flujo de datos bidireccional, cambios en el modelo causarán cambios en la vista, y viceversa.

Pipeline Conjunto de elementos, de procesamiento de datos, conectados en serie, donde la salida de un elemento se corresponde con la entrada del siguiente.

Pure function Función cuyo valor de retorno siempre es el mismo para los mismos argumentos. Además su evaluación no tiene efectos laterales, es decir, no modifica variables estáticas, globales, argumentos pasados por referencia o flujos de entrada/salida.

REST Arquitectura software usada para crear servicios web.

Race condition Es el comportamiento de un sistema en el que dicho comportamiento depende de la secuencia o temporización de procesos o hilos para operar correctamente.

Theming Aplicación de temas a la interfaz de usuario.

Typescript Lenguaje de programación desarrollado por Microsoft. Es un superconjunto de JavaScript que añade tipado estático.

Websocket Protocolo de comunicaciones, que nos proporciona canales full-duplex sobre una conexión TCP.

Bibliografía

[1] (2019) Angular architecture. [Online]. Available: https://angular.io/guide/architecture

[2] (2019) Elixir forum. [En línea]. Disponible en: https://elixirforum.com/

[3] (2019) Elixir website. [En línea]. Disponible en: https://elixir-lang.org/

[4] (2019) Trello. [En línea]. Disponible en: https://trello.com/

[5] (2019) Github. [En línea]. Disponible en: https://github.com/

[6] (2019) erlnote repository. [En línea]. Disponible en: https://github.com/alchexmist/erlnote

[7] (2019) erlnote frontend repo. [En línea]. Disponible en: https://github.com/alchexmist/erlnote-frontend

[8] GenBeta, "El pernicioso círculo vicioso de las cárnicas. los que pagan, los clientes," 2013. [En línea]. Disponible en: https://www.genbeta.com/desarrollo/el-pernicioso-circulo-vicioso-de-las-carnicas-los-que-pagan-los-clientes

[9] "Consultoras tecnológicas cárnicas y cómo salir de la carrera de la rata," 2018. [En línea]. Disponible en: https://elfuturodelosdatos.com/consultoras-tecnologicas-carnicas-la-carrera-de-la-rata/

[10] El País, "Precarios de cuello blanco," 2018. [En línea]. Disponible en: https://elpais.com/economia/2018/06/28/actualidad/1530184992_366865.html

[11] El Diario, "La precariedad y los sueldos bajos irrumpen en el sector de las tic," 2016. [En línea]. Disponible en: https://www.eldiario.es/catalunya/trabajo/precariedad-sueldos-irrumpen-empresas-TIC_0_525298242.html

[12] Voz Populi, "Despidos en indra," 2016. [En línea]. Disponible en: "http://vozpopuli.com/economia-y-finanzas/65091-indra-ofrece-trabajo-a-100-informaticos-junior-mientras-anuncia-1-850-despidos"

[13] El Confidencial, "Engaños y precariedad: así es trabajar en las cárnicas del software en españa," 2017. [En línea]. Disponible en: https://www.elconfidencial.com/tecnologia/2017-08-05/consultoria-tecnologica-lexnet-carnicas-alten-sermicro-iecisa-digitex_1425318/

[14] ——, "Este software es una ruina: todo lo que tienes que saber sobre el desastre lexnet," 2017. [En línea]. Disponible en: https://www.elconfidencial.com/tecnologia/2017-08-03/desastre-lexnet-justicia-ciberseguridad-orfilia-rafael-catala_1424504/

[15] 20 Minutos, "Renta web vuelve a fallar," 2016. [En línea]. Disponible en: https://www.20minutos.es/noticia/2716190/0/cruce-borradores/agencia-tributaria/vista-previa-error-renta-web/

[16] El País, "Interior admite errores en los resultados electorales provisionales que ofrece su web," 2019. [En línea]. Disponible en: https://elpais.com/politica/2019/05/29/actualidad/1559083636_290025.html

[17] El Confidencial, "Múltiples fallos de la empresa del recuento del 26-m disparan las alarmas en interior," 2019. [En línea]. Disponible en: https://www.elconfidencial.com/elecciones-municipales-y-autonomicas/2019-05-28/interior-recuento-elecciones-fallos-scytl_2040906/

[18] El Independiente, "Fallos en el volcado de datos ponen en duda los resultados de las municipales en multitud de ayuntamientos," 2019. [En línea]. Disponible en: https://www.elindependiente.com/politica/2019/05/28/fallo-masivo-web-pone-duda-resultados-26m/

[19] La Voz de Galicia, "Falla el sistema informático de recuento de las elecciones municipales y europeas," 2019. [En línea]. Disponible en: https://www.lavozdegalicia.es/noticia/elecciones/2019/05/11/falla-sistema-informatico-recuento-elecciones-municipales-europeas/00031557572677563419149.htm

[20] Xataka, "Por qué 700000 euros es un presupuesto ridículo para arreglar la web de renfe," 2019. [En línea]. Disponible en: https://www.xataka.com/empresas-y-economia/que-700-000-euros-presupuesto-ridiculo-para-arreglar-web-renfe

[21] El Diario, "Renfe.com: una web saturada y rodeada de consultores," 2013. [En línea]. Disponible en: https://www.eldiario.es/sociedad/Renfecom-polemica-saturada-rodeada-consultores_0_102890083.html

[22] (2019) Graphql: A query language for your api. [En línea]. Disponible en: https://graphql.org

[23] (2019) Exunit. [En línea]. Disponible en: https://hexdocs.pm/ex_unit/ExUnit.html

[24] (2019) Common test. [En línea]. Disponible en: http://erlang.org/doc/apps/common_test/users_guide.html

[25] (2019) Http protocol. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/HTTP

[26] (2011) Websocket protocol. [En línea]. Disponible en: https://tools.ietf.org/html/rfc6455

[27] (2019) Repository pattern. [En línea]. Disponible en: https://martinfowler.com/eaaCatalog/repository.html

[28] (2019) Flux pattern. [En línea]. Disponible en: https://facebook.github.io/flux/

[29] (2019) Phoenix website. [En línea]. Disponible en: https://hexdocs.pm/ecto/Ecto.html

[30] (2019) Postgresql. [En línea]. Disponible en: https://www.postgresql.org/

[31] (2019) Couchdb. [En línea]. Disponible en: http://couchdb.apache.org/

[32] (2019) Apollo graphql. [En línea]. Disponible en: https://www.apollographql.com

[33] (2019) Relay. [En línea]. Disponible en: https://relay.dev

[34] (2019) Absinthe graphql toolkit. [En línea]. Disponible en: https://absinthe-graphql.org

[35] (2019) Phoenix channels. [En línea]. Disponible en: https://hexdocs.pm/phoenix/channels.html

[36] (2019) Phoenix.js documentation. [En línea]. Disponible en: https://hexdocs.pm/phoenix/js/index.html#socket

[37] (2019) Phoenix framework website. [En línea]. Disponible en: https://phoenixframework.org/

[38] (2019) Npm package manager. [En línea]. Disponible en: https://www.npmjs.com/

[39] (2019) Yarn package manager. [En línea]. Disponible en: https://yarnpkg.com

[40] (2019) Redux. [En línea]. Disponible en: https://redux.js.org

[41] (2019) React apollo. [En línea]. Disponible en: https://github.com/apollographql/react-apollo

[42] (2019) Presentational and container components. [En línea]. Disponible en: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0

[43] (2019) Redux developer tools. [En línea]. Disponible en: https://github.com/zalmoxisus/redux-devtools-extension

[44] (2019) Immutable.js library. [En línea]. Disponible en: https://immutable-js.github.io/immutable-js/

[45] (2019) React website. [En línea]. Disponible en: https://reactjs.org/

[46] (2019) Ecmascript. [En línea]. Disponible en: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[47] (2019) Cascading style sheets. [En línea]. Disponible en: https://www.w3.org/Style/CSS/

[48] (2019) React bootstrap. [En línea]. Disponible en: https://react-bootstrap.github.io/

[49] (2019) jquery. [En línea]. Disponible en: https://jquery.com

[50] (2005) Introducción al dom. [En línea]. Disponible en: https://www.w3.org/2005/03/DOM3Core-es/introduccion.html

[51] (2019) Dom virtual en react. [En línea]. Disponible en: https://es.reactjs.org/docs/faq-internals.html

[52] (2019) Presentando jsx. [En línea]. Disponible en: https://es.reactjs.org/docs/introducing-jsx.html

[53] (2019) create-react-app dev tool. [En línea]. Disponible en: https://create-react-app.dev

[54] (2019) Cors for developers. [En línea]. Disponible en: https://w3c.github.io/webappsec-cors-for-developers/

[55] (2019) Cors mdn. [En línea]. Disponible en: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[56] (2019) A complete guide to flexbox. [En línea]. Disponible en: https://css-tricks.com/snippets/css/a-guide-to-flexbox/

[57] (2019) Bootstrap website. [En línea]. Disponible en: https://getbootstrap.com/

[58] (2019) Cowboy web server. [En línea]. Disponible en: https://github.com/ninenines/cowboy

[59] (2019) Apache web server. [En línea]. Disponible en: https://httpd.apache.org/

[60] (2019) Absinthe graphiql tool. [En línea]. Disponible en: https://hexdocs.pm/absinthe_plug/Absinthe.Plug.GraphiQL.html

[61] (2019) exdoc issue. [En línea]. Disponible en: https://github.com/elixir-lang/ex_doc/issues/1027

[62] (2019) exdoc pull request. [En línea]. Disponible en: https://github.com/elixir-lang/ex_doc/pull/1030

[63] (2019) Postgresql full text search. [En línea]. Disponible en: https://www.postgresql.org/docs/11/textsearch.html

[64] (2019) Elixir: a minimal api to talk to elastic. [En línea]. Disponible en: https://gist.github.com/rodrigues/b12bc68d7b9c8b558a931d28c19d3b92

[65] (2019) Elasticsearch. [En línea]. Disponible en: https://www.elastic.co/es/products/elasticsearch

[66] (2019) Material design website. [En línea]. Disponible en: https://material.io/

[67] (2019) Rest. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Representational_state_transfer

[68] B. Williams and B. Wilson, *Craft GraphQL APIs in Elixir with Absinthe*, 1st ed. The Pragmatic Programmers, 2018.

[69] (2019) Erlang website. [En línea]. Disponible en: https://www.erlang.org/

[70] D. Wilson and E. Meadows-Jönsson, *Programming Ecto*, 1st ed. The Pragmatic Programmers, 2019.

[71] (2019) Vue.js website. [En línea]. Disponible en: https://vuejs.org/

[72] (2019) Ember.js website. [En línea]. Disponible en: https://www.emberjs.com/

[73] (2019) Riot.js website. [En línea]. Disponible en: https://riot.js.org/

[74] (2019) Elm website. [En línea]. Disponible en: https://elm-lang.org/

[75] (2019) Semantic ui website. [En línea]. Disponible en: https://semantic-ui.com/

[76] (2019) Foundation framework website. [En línea]. Disponible en: https://foundation.zurb.com/

[77] (2019) Materialize website. [En línea]. Disponible en: https://materializecss.com/

[78] (2019) Uikit website. [En línea]. Disponible en: https://getuikit.com/

[79] (2019) Polymer project website. [En línea]. Disponible en: https://www.polymer-project.org/

[80] (2019) Custom elements specification. [En línea]. Disponible en: https://w3c.github.io/webcomponents/spec/custom/

[81] (2019) Shadow dom specification. [En línea]. Disponible en: https://w3c.github.io/webcomponents/spec/shadow/

[82] (2019) Es 6 module specification. [En línea]. Disponible en: https://html.spec.whatwg.org/multipage/webappapis.html#integration-with-the-javascript-module-system

[83] (2019) Html template specification. [En línea]. Disponible en: https://html.spec.whatwg.org/multipage/scripting.html#the-template-element/

[84] (2019) Material-ui website. [En línea]. Disponible en: https://material-ui.com

[85] (2019) Material components for react website. [En línea]. Disponible en: https://github.com/material-components/material-components-web-react

[86] (2019) Material components for the web. [En línea]. Disponible en: https://github.com/material-components/material-components-web

[87] (2019) Bootstrap wikipedia. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework)

[88] A. Mardan, *React Quickly*, 1st ed. Manning Publications Co., 2017.

[89] A. Banks and E. Porcello, *Learning React: Functional Web Development with React and Redux*, 1st ed. O'Reilly Media, Inc., 2017.

[90] L. Fischer, *React for Real: Front-end code, Untangled*, 1st ed. The Pragmatic Programmers, LLC, 2017.

[91] (2019) Component-based software engineering. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Component-based_software_engineering

[92] (2019) Reconciliation. [En línea]. Disponible en: https://reactjs.org/docs/reconciliation.html

[93] (2013) React's diff algorithm. [En línea]. Disponible en: https://calendar.perfplanet.com/2013/diff/

[94] (2019) Model-view-controller wikipedia. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Model–view–controller

[95] (2019) Mobx website. [En línea]. Disponible en: https://mobx.js.org/

[96] (2019) Refluxjs website. [En línea]. Disponible en: https://github.com/reflux/refluxjs

[97] (2019) Fluxible website. [En línea]. Disponible en: https://fluxible.io/

[98] (2019) Conflict-free replicated data type. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

[99] (2019) y-js. [En línea]. Disponible en: https://github.com/y-js/yjs

[100] (2019) Rust programming language. [En línea]. Disponible en: https://www.rust-lang.org/

[101] S. Jurić, *Elixir in Action*, 2nd ed. Manning Publications Co., 2019.

[102] D. Thomas, *Programming Elixir* $\geq$ *1.6*, 1st ed. The Pragmatic Programmers, 2018.

[103] J. Armstrong, *Programming Erlang*, 2nd ed. The Pragmatic Programmers, 2014.

[104] (2019) Semantic versioning. [En línea]. Disponible en: https://semver.org/

[105] (2019) Software versioning. [En línea]. Disponible en: https://en.wikipedia.org/wiki/Software_versioning

[106] (2019) Hex package manager. [En línea]. Disponible en: https://hex.pm/

[107] (2019) Apache lucene website. [En línea]. Disponible en: https://lucene.apache.org/

[108] (2019) Elixir port module website. [En línea]. Disponible en: https://hexdocs.pm/elixir/Port.html

[109] C. McCord, B. Tate, and J. Valim, *Programming Phoenix* $\geq$ *1.4*, 1st ed. The Pragmatic Programmers, 2019.

[110] (2019) Streamdata. [En línea]. Disponible en: https://github.com/whatyouhide/stream_data

[111] (2019) Visual studio code. [En línea]. Disponible en: https://code.visualstudio.com/

[112] (2019) Gnu emacs. [En línea]. Disponible en: https://www.gnu.org/software/emacs/

[113] (2019) elixir-ls. [En línea]. Disponible en: https://github.com/JakeBecker/elixir-ls