UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Politécnica Superior de Gandia

Diseño e implementación de una arquitectura de software para una aplicación de análisis del mercado eléctrico

Trabajo Fin de Grado

Grado en Tecnologías Interactivas

AUTOR/A: Losa García, Alejandro

Tutor/a: Palanca Cámara, Javier

CURSO ACADÉMICO: 2022/2023

Resumen

El siguiente trabajo describe el proceso seguido para obtener una aplicación de análisis y trading del mercado eléctrico ibérico, desde la concepción de la idea y el análisis del modelo negocio, pasando por el diseño de la interfaz y la experiencia del usuario, hasta el diseño de la arquitectura y su consiguiente implementación y despliegue.

El eje central del trabajo está compuesto por las decisiones de arquitectura seguidas, los patrones de diseño utilizados, los distintos microservicios que conforman el proyecto y sus implementaciones siguiendo las prácticas modernas de programación.

Palabras clave

Arquitectura de software, Diseño dirigido por dominio (DDD), IA, Mercado eléctrico, Código limpio.

Abstract

The following work describes the process followed to obtain an analysis and trading application for the Iberian electric market, from the conception of the idea and analysis of the business model, through the design of the interface and user experience, to the design of the architecture and its subsequent implementation and deployment.

The central axis of the work is composed of the architecture decisions followed, the design patterns used, the different microservices that make up the project, and their implementations following modern programming practices.

Keywords

Software architecture, Domain Driven Design (DDD), AI, Electricity market, Clean code.

Índice

Figuras y tablas

1. Introducción

1.1 Contexto

Durante los últimos 10 años, el mundo ha vivido el mayor salto tecnológico de la historia. Debido a la rapidez con la que las nuevas tecnologías han irrumpido en nuestras vidas, muchas empresas y sectores no han podido adaptarse a los cambios que se han producido, perdiendo así una gran oportunidad de mejorar sus modelos de negocio, reducir costes, o simplemente, ofrecer un mejor servicio a sus clientes. Uno de estos sectores afectados por la falta de innovación es el sector energético (en parte). Si bien, la inversión en nuevos modelos de generación de energía verde han aumentado de forma exponencial, hay otras secciones de la cadena productiva que no han sido atendidas por el mercado. Una de estas secciones y motivo de este trabajo es la comercialización de dicha energía.

Actualmente, el precio de la electricidad en Europa (y en España) está determinado a partir de la oferta y la demanda. Esto ha dado lugar a una cadena productiva segmentada en la que se comercia con la electricidad desde su producción hasta su distribución final. El mercado se divide en los siguientes segmentos: producción, transporte, distribución y comercialización. Cada paso de un segmento a otro supone una transacción en el mercado, y por ende, una negociación en la que el vendedor y el comprador intentan cerrar un precio que sea beneficioso para ambos. En este contexto, surgen empresas de trading, que se encargan de comprar y vender electricidad estimando su precio a futuro con el fin de ejecutar sus operaciones en el momento idóneo para maximizar sus beneficios.

El siguiente trabajo describe el proceso seguido para desarrollar una aplicación web mediante la cual, empresas del sector energético puedan analizar, estimar y ejecutar operaciones de trading en el mercado eléctrico ibérico de una forma rápida y eficiente. Para lograr este objetivo, se ha implementado una arquitectura de software robusta y escalable, que permite adaptar el proyecto a cualquier cambio, además se ha desarrollado una herramienta de inteligencia artificial para la predicción de precios y se ha diseñado una interfaz de usuario moderna e intuitiva.

El lector encontrará en este documento un conjunto de prácticas modernas utilizadas comúnmente en grandes empresas del sector del software, que exprimen al máximo los conocimientos adquiridos durante el grado.

1.2 Motivación

Durante el transcurso de mi vida académica, y sobre todo profesional, he podido observar e incluso beneficiarme de la capacidad de resolución de problemas que aporta la ingeniería de software. Hasta un simple script de una línea, ejecutado en la situación adecuada, puede reducir los costes de un proceso de manera astronómica.

Por otra parte, el sector energético es especialmente transversal, ya que todas las personas, independientemente de su poder adquisitivo, edad o creencia religiosa, requieren de electricidad para poder vivir de forma digna. Por este motivo, y al ver una necesidad no

cubierta de forma eficiente por el mercado, me decidí a desarrollar este proyecto. Incluso, con vistas a escalarlo a un modelo de negocio real.

Esto último es muy importante, pues ha afectado de forma muy intensa a las decisiones de diseño y de arquitectura que se han tomado durante el desarrollo.

1.3 Objetivos

Este proyecto tiene como objetivo diseñar e implementar una arquitectura de software escalable que permita a agentes del mercado eléctrico obtener información del propio mercado y elementos adyacentes, así como poder estimar el precio a futuro la energía. Para ello, se han definido los siguientes hitos:

- Diseñar una interfaz de usuario moderna e intuitiva e implementarla con tecnologías web.

- Diseñar una arquitectura de software escalable que permita hacer crecer el proyecto sin la necesidad de rehacer código y teniendo en cuenta patrones de diseño y de arquitectura que aumenten la cohesión y reduzcan el acoplamiento.

- Implementar un frontend web escalable acorde a la arquitectura general del proyecto.

- Implementar un backend que nutra al frontend y sea resiliente a fallos, además de mantener concordancia con la arquitectura de la aplicación.

- Implementar un modelo de predicción de precios de la electricidad para la siguiente sesión horaria.

1.4 Requisitos

La aplicación debe cumplir una serie de requisitos que certifiquen que los usuarios pueden analizar el mercado eléctrico de forma correcta. Para ello, la aplicación debe contar con las siguientes funcionalidades:

- **Inicio de sesión:** Los usuarios deben poder iniciar sesión para poder acceder a la aplicación.

- **Dashboard:** La aplicación debe tener un dashboard principal donde se muestra información relevante como el precio actual del mercado diario e intradiario, así como las últimas alertas y movimientos.

- **Página de datos:** Los usuarios deben poder acceder a los distintos datos que afectan al mercado, como precios, oferta y demanda, distribución de la generación…

- **Pronósticos:** Se debe mostrar en una tabla las predicciones de precio a futuro.

- **Alertas:** Se debe notificar al usuario cuando un precio suba o baje de un límite.

- **Configuración:** El usuario debe poder controlar distintas configuraciones de su cuenta.

1.5 Relación con los ODS

Este proyecto tiene como foco obtener una mayor eficiencia de los operadores del mercado energético español y europeo, de forma que afecta directamente a los ODS 8 y 9, trabajo decente y crecimiento económico, e industria, innovación e infraestructuras respectivamente.

Además, al permitir que cualquier persona pueda tomar acción en el análisis e incluso la adquisición de energía, esto repercute en una mayor competitividad de del mercado; y dado que las energías renovables ofrecen un mayor margen de beneficios que los métodos de generación con emisiones, esto puede repercutir en menor medida en que las empresas generadoras inviertan más capital en fuentes de generación renovables para adaptarse a la mayor demanda del mercado y por tanto maximizar sus ingresos. Por tanto, puede favorecer también los ODS 7 y 13.

Se ha adjuntado una tabla con todos los ODS en el anexo 1, al final del documento.

1.6 Estructura del documento

El documento se ha dividido en los siguientes capítulos:

- **Capítulo 1. Introducción:** Es este primer capítulo, se define el contexto y los motivos por los que se ha decidido abordar el proyecto, así como los objetivos y la relación del trabajo con los ODS .

- **Capítulo 2. Metodología:** Esta sección está destinada a exponer el flujo de trabajo que se ha definido, así como las distintas etapas del proyecto y el enfoque seguido.

- **Capítulo 3. Estado del arte:** El tercer capítulo contiene el análisis de la situación técnica que atraviesa el sector, así como las tecnologías disponibles y las tendencias más utilizadas en la actualidad.

- **Capítulo 4. Desarrollo del proyecto:** Este apartado contiene el grueso del contenido. Aquí se explican las soluciones que se han implementado, el diseño de la interfaz, y el desarrollo del frontend, el backend y la IA.

- **Capítulo 5. Validación de los resultados:** En el quinto capítulo se comentan los resultados obtenidos después de todo el proceso de desarrollo.

- **Capítulo 6. Conclusiones:** En este apartado se presenta un análisis final, relacionando todo el proceso seguido y los resultados obtenidos con los objetivos propuestos inicialmente.

2. Metodología

Cualquier proyecto de desarrollo de software requiere de una metodología que organice el flujo de trabajo y permita a los desarrolladores, product owners, managers y stakeholders conocer el estado del proyecto. En este caso, se ha elegido Kanban para organizar el flujo de trabajo.

Flujo de trabajo con Kanban
Actualmente, es común implementar metodologías ágiles en los equipos ya debido a las necesidades del mercado de la ingeniería de software. Hay múltiples metodologías ágiles que permiten desarrollar proyectos con una entrega de valor continua, una mayor robustez frente a errores y una mayor calidad del software.

En este caso, solo hay una persona implicada y no se requiere de validación por parte del cliente, por lo que las bondades de Scrum no son tan aprovechables como en otros entornos, sin embargo, Kanban se ajusta muy bien a las necesidades del proyecto, ya que facilita la organización durante el desarrollo y permite conocer en todo momento qué partes del proyecto están pendientes, en desarrollo o bloqueadas.

Flujo de desarrollo con pull requests
Trabajar con Kanban implica organizar el trabajo en tareas, por tanto, esto repercute en la forma en la que los desarrolladores administran su flujo de desarrollo. En este caso, se ha asignado un identificador a cada tarea, por ejemplo DEV-1.

Todos los repositorios están divididos utilizando git flow, es decir, con una rama master que contiene el código que está listo para producción, una rama develop con el código en pre-producción y varias ramas “feature”. Cada una de estas ramas “feature” corresponde con una tarea del tablón Kanban, y por tanto, se le asigna el mismo identificador que la tarea. Para la tarea DEV-1, habría una rama feature/DEV-1 que contiene el desarrollo asociado a esa tarea.

También se ha seguido la metodología FDD (Feature Driven Development), con la que se intenta conseguir que las nuevas funcionalidades entren poco a poco y de forma continua en vez de en grandes cambios, de forma que se pueda ir testeando cada pequeño cambio y minimizar así los posibles errores. Para llevar a cabo esta metodología, es posible que una tarea tenga que ser desarrollada en varios Pull Requests, de forma que requerirían distintas ramas. En este caso, cada rama se nombraría con el identificador de la tarea y el número del pull request (features/DEV-1-1, features/DEV-1-2…).

3. Estado del arte

A continuación, se comentarán los puntos claves del negocio de la compra-venta de electricidad (a partir de los cuales se modela el dominio), así como la situación actual, tanto del sector del software profesional, como de las aplicaciones de compra venta de electricidad. En este proyecto, se ha hecho especial hincapié en el diseño de la arquitectura y en el uso de buenas prácticas, por lo que también se expondrán algunas técnicas muy comunes hoy en día.

3.1 Trading del mercado eléctrico

Para poder desarrollar este proyecto, es necesario comprender cómo funciona el mercado eléctrico y quienes participan en él.

Un mercado es el lugar donde se realizan intercambios de bienes o servicios. En el mercado eléctrico, el bien ofertado es la electricidad (en forma de kWh o múltiplos), que es intercambiado por dinero, como en la mayoría de mercados modernos. Los distintos agentes que intervienen en este mercado son las empresas productoras, las empresas distribuidoras, los comercializadores y los clientes finales.

Si bien es un sector muy amplio, lo más importante a saber es que en cada intercambio entre los agentes de la cadena productiva, el precio de la energía es determinado a partir de la oferta y la demanda de la misma, por tanto, la parte ofertante buscará vender al máximo precio posible, y la parte demandante al mínimo. Con lo cual, la lógica de negocio tiene como objetivo maximizar el rendimiento para ambas partes así como proporcionar los datos necesarios para que los agentes puedan tomar decisiones basadas en datos.

3.2 Software en el sector energético

En el siguiente apartado se comentarán distintas empresas y productos relacionados con el sector energético que afectan de forma directa a este proyecto.

3.2.1 Red Eléctrica Española

Red Eléctrica es una empresa española encargada de gestionar la red de transporte de energía eléctrica en España y de garantizar el suministro de energía eléctrica en todo el territorio. Es responsable de planificar y operar el sistema eléctrico español, asegurando la continuidad y calidad del suministro eléctrico en todo el país. Además, también se encarga de gestionar el acceso a la red eléctrica y de coordinar los intercambios internacionales de energía eléctrica con otros países.

Además, proporciona una API REST con distintos datos como el balance energético, la demanda, los intercambios de energía, el transporte o la generación eléctrica.

3.2.2 OMIE

OMIE (Organización de Mercado Ibérico de Energía) es una organización que gestiona los mercados eléctricos de España y Portugal. Se encarga de coordinar la operación y gestión del sistema eléctrico en la Península Ibérica y facilitar la compraventa de electricidad en los

mercados mayoristas de energía. Además, también se encarga de supervisar y controlar la calidad del suministro eléctrico en la región.

También proporciona de forma abierta los distintos datos del mercado eléctrico mediante ficheros csv, para que cualquier empresa o proyecto pueda acceder a ellos.

3.2.3 Gnarum

Gnarum es una empresa que desarrolla aplicaciones para el sector energético, entre las que destacan dos: Janus, una aplicación de análisis de datos del mercado eléctrico, y Trade, una plataforma de trading tanto en el mercado eléctrico ibérico, como en el italiano. Aunque estas aplicaciones cuentan con una pobre y anticuada experiencia de usuario y es difícilmente accesible.

Además de ofrecer una peor experiencia, la empresa ofrece los servicios de forma separada, por lo tanto, los usuarios que necesiten analizar y hacer operaciones, deben ir alternando entre las dos aplicaciones. En cambio, la solución aportada por este proyecto ya incluye ambas funcionalidades dentro de la misma aplicación.

Otra mejora respecto a Gnarum es la implementación de una IA de predicción de precios, ya que si bien esta empresa proporciona los datos, es el usuario el que tiene que hacer un análisis de los mismos.

3.2.4 Hojas de cálculo

Muchas empresas utilizan distintos programas como Excel para crear modelos matemáticos a partir de los cuales estiman el precio objetivo de la electricidad para varios intervalos de tiempo.

Esta solución permite a los agentes saber como se está calculando la exactamente la predicción, pero supone un esfuerzo mayor, al tener que programar el modelo, así como proporcionarle los datos cada vez que se quiere hacer un cálculo, haciendo que el trabajador pierda mucho tiempo en tareas que realmente no aportan un valor al negocio.

3.2.5 Isotrol

Isotrol es otra empresa que desarrolla aplicaciones para el trading de electricidad. Al igual que Gnarum, cuentan con un programa anticuado, aunque en este caso, está enfocado en las operaciones dentro del mercado europeo.

3.3 Diseño guiado por dominio

El diseño guiado por dominio, comúnmente conocido como DDD (Domain Driven Design), es un paradigma del mundo de la ingeniería de software, en el cual el desarrollo se centra en el dominio, que es la representación del modelo de negocio dentro de la aplicación.

DDD no sigue una implementación concreta, sino que proporciona una serie de reglas y conceptos que permiten construir software de calidad que se amolde correctamente a las necesidades del negocio y facilita el mantenimiento y expansión de los proyectos.

Es importante destacar que, como el DDD engloba tanto el mundo del software como el mundo de negocio, para que su implementación en un proyecto sea fructífera, hay que hacer partícipes tanto a los desarrolladores de software, como a los product managers y project managers.

Dentro del DDD distinguimos dos tipos de perspectivas, la estratégica, cuyo objetivo es ayudar a los desarrolladores con el modelado del dominio, y la táctica, que marca una serie de patrones necesarios para implementar las decisiones estratégicas.

A continuación se comentarán los conceptos más importantes utilizados en DDD.

3.3.1 Contexto acotado

Los contextos acotados, más conocidos como bounded contexts, son agrupaciones lógicas que delimitan un problema de negocio y su solución, es decir, delimitan los dominios. Por ejemplo, en una aplicación para un ecommerce podríamos encontrar varios contextos acotados como: la tienda online, la gestión de productos y stock o la gestión de pedidos.

Cada uno de estos contextos contiene una sección distinta del negocio de la aplicación. Así, la lógica de negocio, las validaciones, las responsabilidades y los problemas a resolver de cada sección permanecen limitados y encapsulados, lo que ayuda a mantener el proyecto ordenado y bien organizado.

En el caso de la aplicación para el ecommerce, los beneficios de dividir el negocio en varios bounded context se pueden apreciar en el siguiente ejemplo: Tanto en el contexto de la tienda online, como en el de gestión de productos encontramos el concepto de usuario, sin embargo, el significado del mismo no es igual para ambos marcos.

Para la tienda online, un usuario es una persona que navega por la web y que puede guardar productos en el carrito, hacer compras o marcar un producto como favorito; por otro lado, para la sección de gestión de productos, un usuario puede ser tan solo una persona que tenga los permisos necesarios para modificar el registro del stock o visualizar qué productos han sido los más vendidos. Si la aplicación no estuviese dividida en distintos contextos, esto se traduciría en una clase User, que contendría toda la lógica necesaria para abarcar todos los usos de un usuario, independientemente de si está en la tienda online o en el dashboard de stock. De esta forma, nos encontramos con dos clases User, en módulos distintos y con significados distintos, por lo que un usuario de gestión de stocks ya no tiene por qué tener definidos atributos que en su caso de uso siempre iban a estar vacíos.

3.3.2 Lenguaje oblicuo

El lenguaje oblicuo es un concepto que se utiliza para definir un uso del lenguaje común entre los miembros de un equipo de desarrollo y los propietarios del dominio (product owners, project managers, stakeholders, etc). El lenguaje oblicuo es importante, porque sin él, las distintas partes implicadas no podrían comunicarse con eficiencia.

Cada contexto acotado cuenta con su propio lenguaje oblicuo, de forma que en el ejemplo anterior, un usuario no significa lo mismo en la tienda online que en la gestión de stocks. Puede parecer un aspecto trivial, ya que no está relacionado con la programación en sí, pero

un lenguaje oblicuo bien escogido, consensuado y actualizado es clave para el buen mantenimiento y evolución de un proyecto de software.

3.3.3 Entidad

Una entidad es un objeto de dominio que representa parte del negocio y que tiene una identidad y propiedades únicas, cuyo estado puede ir cambiando en el tiempo. Las entidades permiten modelar conceptos del negocio que van cambiando con el tiempo.

En el contexto de tienda online del ejemplo anterior, la clase User sería una entidad, porque representa a un usuario real dentro de la aplicación y tiene una identidad única, ya que cada usuario tiene un identificador individual que lo distingue de cualquier otro usuario. Otro ejemplo de entidad en el mismo contexto podría ser una clase *Product*, que contiene los atributos necesarios para modelar un producto que es vendido en la tienda, y que además, cada *Product* es único y distinguible de los demás.

3.3.4 Value object

Los value objects, abreviados VOs, son similares a las entidades, ya que ambos conceptos contienen atributos que sirven para modelar las distintas partes del negocio, sin embargo, al contrario que las entidades, los value objects no contienen una identidad en sí, de forma que dos objetos con las mismas propiedades no podrían ser distinguidos entre sí.

Un ejemplo de value object en el módulo de tienda online podría ser el color o la talla, que estarían modelados como una clase Color y una clase Size. Si tenemos dos clases (o enums, que quizás sería más apropiado para este caso) Color que representan el color azul, no tenemos forma de distinguirlas, ya que no tienen una identidad propia.

Entidades y value objects interaccionan entre sí constantemente, como se puede apreciar en el ejemplo anterior, ya que una clase Product (entidad) puede contener un atributo Size y un atributo Color (VOs).

3.3.5 Agregado

Un agregado es un conjunto de objetos de dominio que funcionan como una única unidad transaccional. El objetivo de los agregados es mantener la consistencia entre todos los objetos de dominios que están relacionados cuando se producen cambios, de manera que se consigue mantener la integridad del dominio.

Cada agregado tiene un objeto de dominio que contiene a todos los demás, conocido como raíz del agregado. Este objeto es el encargado de realizar las validaciones necesarias para mantener la consistencia e integridad del dominio.

Siguiendo con el ejemplo de la tienda online, un agregado podría ser un objeto Order (pedido). Este objeto contiene a su vez un atributo User, un atributo Products, un atributo Address, etc. Sería la clase Order la que realizaría las validaciones necesarias, como verificar que la lista de productos no está vacía o que la dirección está completa, antes de almacenar la información del pedido en la base de datos o enviarla por correo.

3.3.6 Evento de dominio

Los eventos de dominio son representaciones de sucesos que ocurren dentro de nuestro negocio. Son utilizados para comunicar cambios de estado dentro del dominio. Estos eventos deben ser escuchados por todas las partes que necesiten actuar ante un cambio en el estado de una determinada sección del dominio. Además, sirven para comunicar información entre distintos contextos e incluso distintas aplicaciones sin necesidad de que se produzca acoplamiento.

Cuando un usuario finaliza una compra en nuestra tienda online y se almacena la información del agregado Order en la base de datos, se emite un evento de dominio *OrderPlaced* (desde el contexto de tienda online), que es escuchado por el contexto de gestión de stock para actualizar los inventarios con los productos que se han pedido.

3.3.7 Gran bola de barro

Gran bola de barro, o "big ball of mud" por su término en inglés, es un concepto usado para definir un código que está fuertemente acoplado, desordenado y que ha crecido sin un diseño coherente. Es un problema ya que ralentiza el desarrollo de nuevo código e impide a los nuevos programadores poder entender de forma correcta el funcionamiento del mismo.

3.3.8 Capa de anticorrupción

Refactorizar una gran bola de barro es un proceso muy complicado y costoso, y empezar desarrollos nuevos extendiendo ese código implica aumentar el problema y hace que el proyecto sea cada vez más difícil de mantener. Por eso, los nuevos desarrollos se modelan con un dominio independiente separado del código de la gran bola de barro y se comunican con este mediante mapeos fuera del dominio. Esta división lógica que se ha creado y que divide el nuevo código del código "legacy" se conoce como "anti-corruption layer" o capa de anticorrupción.

3.3.9 Shared kernel

En ocasiones, dos o más contextos acotados comparten parte del dominio, pues una sección del negocio puede ser común para ambos. Para no duplicar este código en ambos contextos, se utiliza el *shared kernel*. El *shared kernel* contiene aquellos objetos de dominio que son transversales y que se utilizan por múltiples contextos.

Por ejemplo, el identificador de un producto, modelado en una clase ProductId, es utilizado tanto en el contexto de la tienda online como en el de gestión de stocks, por tanto, se guarda en un módulo *SharedKernel*, separado de ambos contextos.

Como se ha mencionado anteriormente, el DDD no define la implementación, por eso, se apoya en distintas 'arquitecturas limpias' como la arquitectura hexagonal.

3.4 Arquitectura hexagonal

En 2012, Robert C. Martin acuñó el concepto de "clean architecture" para definir aquellas arquitecturas de software que, siguiendo una serie de principios (SOLID) permiten construir software robusto, entendible y fácilmente escalable y extensible.

Aunque Robert C. Martin le dio el nombre, ya se conocían arquitecturas que cumplían con estos parámetros. Concretamente, en 2005, Alistair Cockburn creó el concepto de 'arquitectura de puertos y adaptadores', también conocida como arquitectura hexagonal o arquitectura de cebolla.

En esta arquitectura, los puertos son las interfaces mediante las cuales la aplicación se comunica con el exterior, y los adaptadores, son las implementaciones de dichas interfaces. Por ejemplo, en una API que guarda datos en una base de datos, un puerto sería la interfaz que definiría las acciones a realizar por la base de datos, y un adaptador sería la implementación de dicha interfaz para un tipo de base de datos concreta, como Oracle o MongoDB. De esta forma, la lógica de negocio no interactúa con las implementaciones específicas de terceros que nada tienen que ver con el negocio en sí.

Esta arquitectura divide las aplicaciones en capas para evitar el acoplamiento entre distintas clases cuyos propósitos estructurales son distintos. Comúnmente, se emplean 3 capas:

- **Dominio**, que es la capa más interna, se encarga de representar el negocio y contiene la lógica de la aplicación.

- **Aplicación**, es la capa intermedia, contiene los casos de uso y funciona como orquestador entre el dominio y la infraestructura.

- **Infraestructura**, es la capa más externa, que contiene los adaptadores e interactúa con los elementos externos a la aplicación.

La clave de esta división está en la imposibilidad de las capas internas de comunicarse con las capas externas, de esta forma, la capa de aplicación solo conoce a la capa de dominio, y la capa de dominio solo se conoce a sí misma, lo que permite que la lógica de negocio no se expuesta a factores externos.

Esta división casa muy bien con el DDD debido al aislamiento del dominio del resto de capas, que permite que pueda ser modelado sin ninguna interferencia.

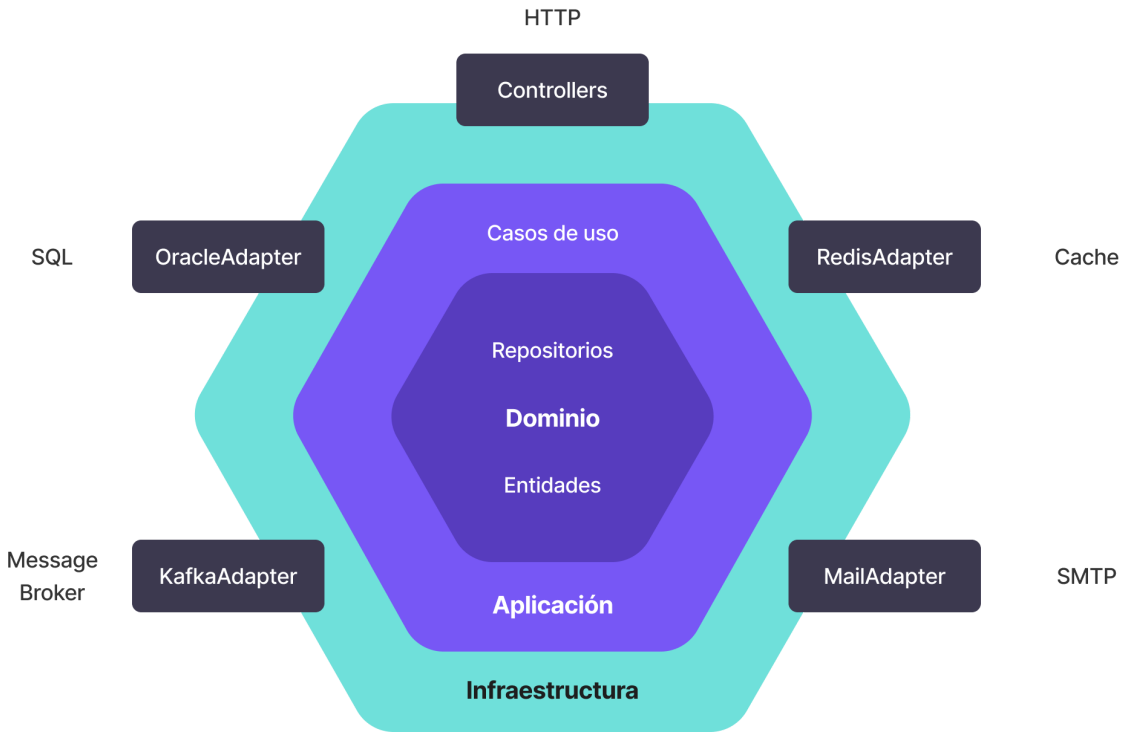


Figura 1. Arquitectura hexagonal

3.5 CQRS y Event Sourcing

CQRS (Command Query Responsibility Segregation) es un patrón de arquitectura que trata de separar la escritura de una base de datos, de la lectura de la misma. Esto permite asignar recursos de forma mucho más específica a los procesos necesarios para nuestra aplicación. Por ejemplo, en la API de una red social de publicación de posts, se leen muchas más publicaciones de las que se escriben, por eso, si ambos procesos están separados, se puede dedicar más recursos a la máquina encargada de la lectura que a la encargada de la escritura.

El Event Sourcing es un patrón de arquitectura que se basa en la captura de los cambios que se producen en la aplicación y en la emisión de eventos de dichos cambios. La utilidad de estos dos patrones aumenta exponencialmente cuando se implementan juntos, ya que nos permiten, entre otras cosas, activar procesos de lectura tras un proceso de escritura mediante la emisión de un evento sin que ambos servicios se acoplen de forma directa.

Por ejemplo, en un portal inmobiliario con una sección de últimos anuncios, una agencia publica un nuevo anuncio, que se guarda en la base de datos mediante un comando de escritura. Posteriormente, se genera un evento, que es escuchado por otro servicio que ejecuta una petición de lectura y actualiza la vista. En este caso, ambos servicios, el de publicación y el de representación de anuncios, estaban separados, pero pudieron comunicarse indirectamente mediante eventos sin estar acoplados.

También es muy común encontrar estos patrones en aplicaciones que provienen de una arquitectura de monolito y actualmente se han migrado a DDD. Rehacer el monolito siguiendo buenas prácticas es muy complejo y costoso, por eso, se aísla la 'gran bola de

lodo' emitiendo eventos que son escuchados por otros servicios. De esta forma, el monolito queda encapsulado y los nuevos servicios se pueden comunicar con él sin la necesidad de acoplarse.

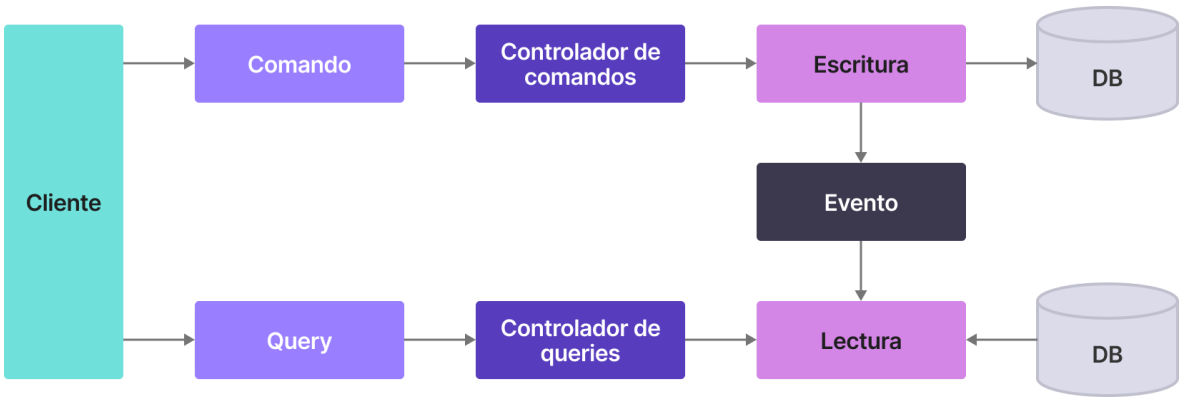


Figura 2. CQRS

3.6 Arquitectura de microservicios

Es un enfoque mediante el cuál, la arquitectura de una aplicación se divide en pequeños módulos independientes que realizan una función única y que pueden ser desplegados de forma individual e independiente. Esto facilita el crecimiento de una aplicación, aunque también tiene un punto negativo, y es que añade complejidad a la arquitectura.

3.7 Desarrollo de aplicaciones de escritorio

Actualmente, hay dos principales enfoques, el desarrollo nativo, y las PWA. El desarrollo nativo era el más utilizado hace unos años. Su principal ventaja es el alto rendimiento, pues las aplicaciones corren directamente sobre el sistema operativo en cuestión, aunque tienen limitaciones en cuanto a personalización de la UI, además de requerir duplicidad, pues hay que desarrollar una solución específica para cada entorno. Las soluciones más comunes para el desarrollo nativo son Qt y .NET para Windows y Swift para MacOS

Por otro lado, las PWA (Progressive Web Application) son aplicaciones desarrolladas con tecnologías web que corren sobre un navegador que se ejecuta como una app de escritorio. Aunque son menos eficientes, ya que no corren sobre el kernel del sistema operativo si no sobre un motor de JS, permiten que solo se tenga que desarrollar una solución. El framework más popular para el desarrollo de aplicaciones web de escritorio es Electron.

3.8 React

Para este proyecto se ha utilizado React, que es la librería más común en el desarrollo de frontend. Es muy común complementarlo con otras librerías de gestión de estado como Redux, aunque en este proyecto se ha implementado una solución propia basada en el patrón de diseño flux, que se expondrá más adelante.

4. Desarrollo del proyecto

Para llevar a cabo el proyecto, se ha dividido el desarrollo en varias secciones que se comentarán durante los siguientes apartados.

4.1 Interfaz de usuario

Uno de los factores diferenciales del proyecto respecto a otras aplicaciones de la competencia es la interfaz gráfica y la experiencia de usuario. Por eso, se ha prestado especial atención al proceso de diseño.

La interfaz de usuario debe ser moderna y fácil de usar. Todas las páginas de la aplicación deben seguir el mismo estilo, los mismos colores y la misma fuente. La fuente utilizada es Inter, que dispone de una licencia de uso libre. A continuación se mostrarán los colores empleados:

Color	Código	Uso
	#FFFFFF	Fondo
	#F8F8FC	Detalles del fondo
	#111315	Texto
	#2A2B2D	Menús
	#24C28B	Detalles positivos
	#DC5356	Detalles negativos
	#4950F0	Primario

Figura 3. Colores de la interfaz

4.1.1 Estructura

Esta aplicación está pensada para usarla en un entorno de trabajo, por eso se ha intentado que todas las funcionalidades sean accesibles empleando el menor número de interacciones posibles para que el usuario no pierda tiempo navegando entre menús.

Por eso, se ha decidido seguir una estructura de dashboard, con un menú lateral que contenga todas las funcionalidades principales. El contenido se muestra en una sección principal, que varía dependiendo de la página.

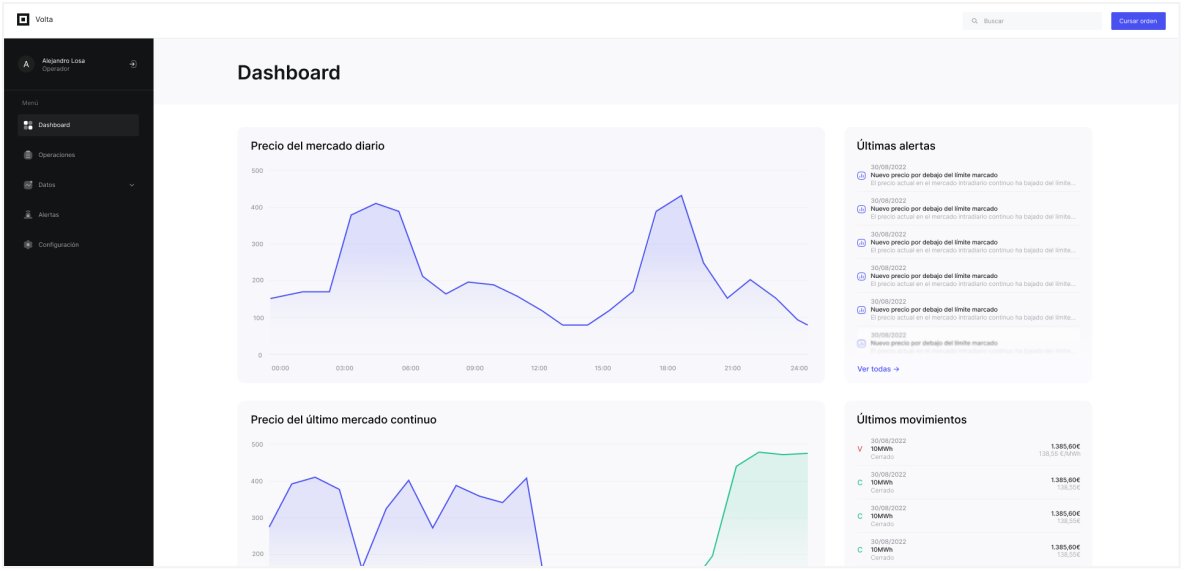


Figura 4. Estructura de la aplicación

4.1.2 Gráficas

En este proyecto se necesita mostrar una gran cantidad de datos, por eso, se ha diseñado una serie de gráficas que permiten al usuario visualizar dicha información con solo un vistazo.

- **Gráficas lineales**, utilizadas para representar series temporales.
- **Gráficas de barras**, empleadas para comparar varias características de un conjunto de datos.
- **Gráficas circulares**, utilizadas para representar datos binarios.
- **Gráficas compuestas**, que fusionan distintos tipos en una única representación.

Figura 5. Gráfica lineal

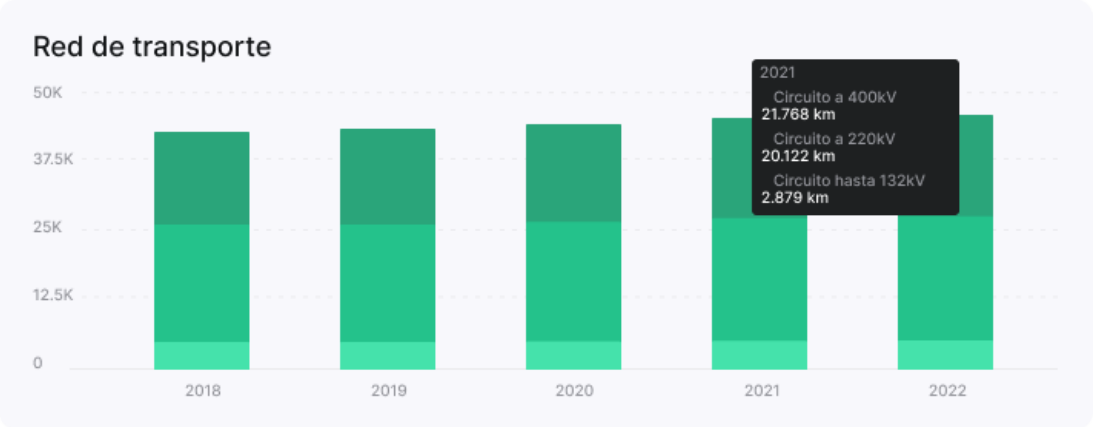


Figura 6. Gráfica de barras

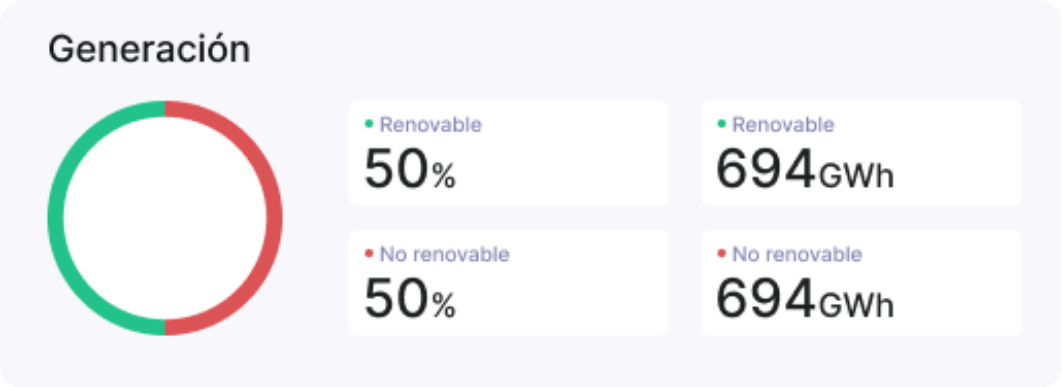


Figura 7. Gráfica circular con etiquetas

4.1.3 Etiquetas

Debido a la gran cantidad de datos de distinta índole que hay que representar (valores de energía, precios, textos, potencia, etc) surgió la necesidad de un componente que pudiese representar cualquier dato y que pudiese coexistir junto con otros componentes en cualquier posición. Teniendo esto en mente se ha diseñado la etiqueta.

Figura 8. Etiquetas

4.1.4 Tablas

Algunos usuarios tienen la necesidad de visualizar los valores de un conjunto de datos directamente en vez de fijarse en una gráfica, que es más visual pero menos precisa. Para satisfacer a estos se ha diseñado una tabla.

	H01	H02	H03	H04	H05	H06	H07	H08	H09	H10	H11	H12	H13	H14	H15
Demanda	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Eólica	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Fotovoltaica	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Termosolar	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Nuclear	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Hidráulica UGH	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Hidráulica no UGH	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6
Turbinación bombeo	161,5	148,4	137,2	130,5	123,2	127,1	135,5	142,5	143,6	118,6	81,4	42,3	25,1	15,4	8,6

Figura 9. Tabla

4.1.5 Listados

Se han creado varios tipos dependiendo de la cantidad de datos a mostrar y de su funcionalidad:

- **Listados simples:** Para listar elementos con poca información.
- **Listados complejos:** Para listar elementos con más información que los anteriores.
- **Listados interactivos:** Cada elemento contiene un botón que abre una modal configurable.

Figura 10. Listado simple

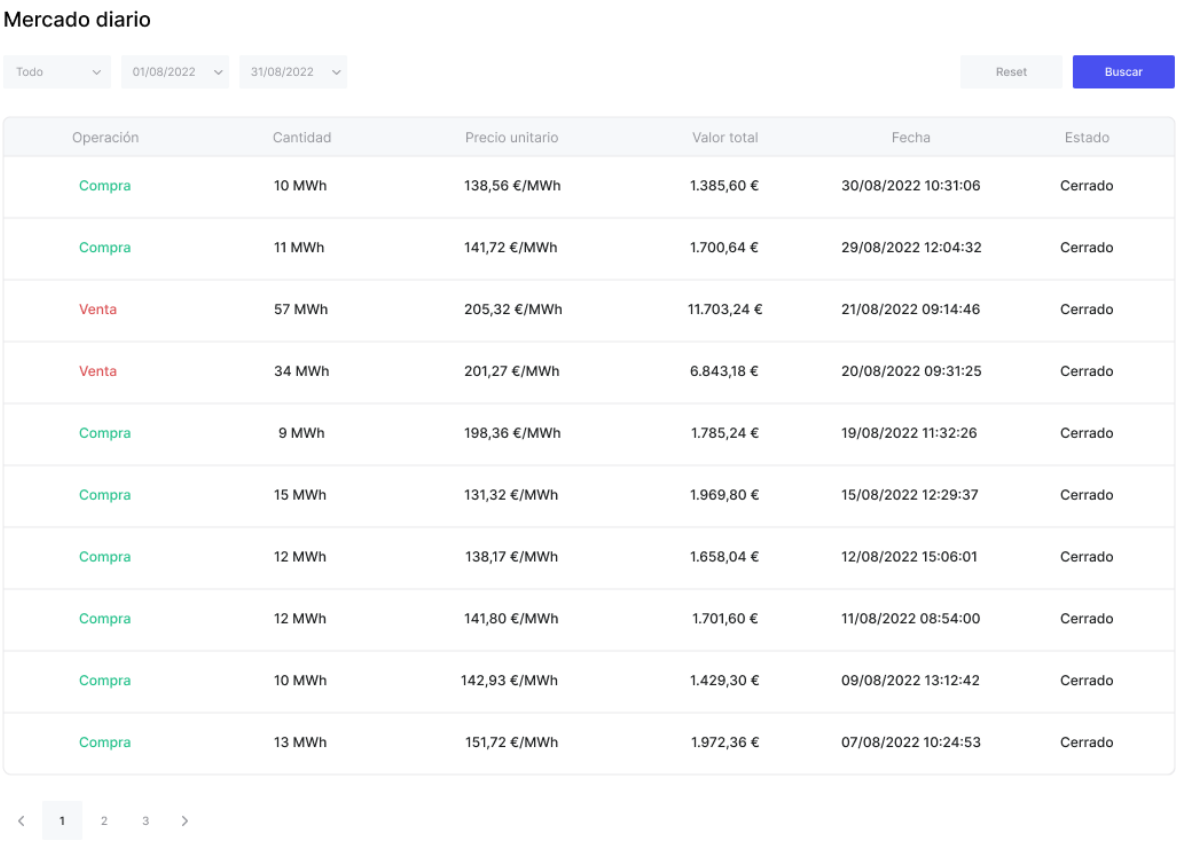


Figura 11. Listado complejo

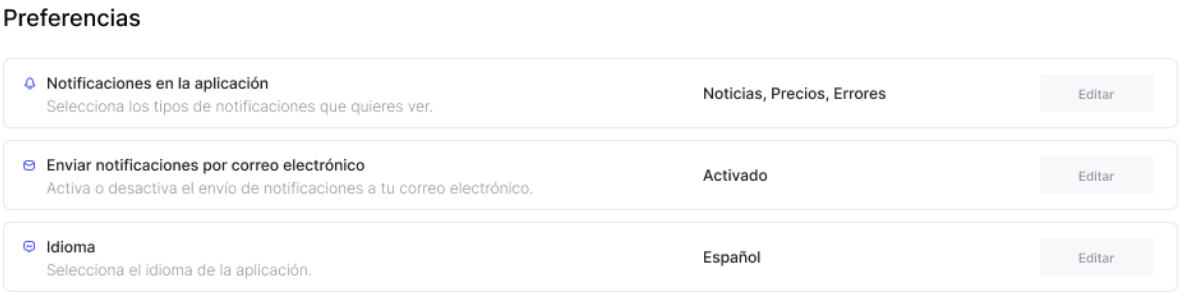


Figura 12. Listado interactivo

En definitiva, se ha planteado un diseño de interfaz limpia y moderna y con perspectiva en el usuario final. Se ha seguido un conjunto de colores definidos previamente que concuerdan con la identidad visual de la marca y se han creado distintos componentes que pueden ser modularizados y combinados entre sí.

4.2 Frontend

Para que los usuarios puedan interactuar con la aplicación, se necesita un cliente que implemente la interfaz previamente diseñada.

En este caso, se ha decidido desarrollar una aplicación web utilizando TypeScript, un lenguaje de programación que añade tipado estático y otras funcionalidades a JavaScript y

que es transpilado a este último antes de ser ejecutado; así como React, una librería que permite aislar el código de la vista en distintos componentes reutilizables y mejorar el rendimiento gracias a una simulación del DOM que solo actualiza el segmento de HTML necesario cuando se produce un cambio en el mismo.

Además de poder ejecutarlo en una web, la aplicación debe poder descargarse e instalarse en cualquier ordenador, por eso se ha hecho uso de Electron, un framework de JS que empaqueta el código ya desarrollado en una aplicación de escritorio (ejecuta un Chromium sobre el que corre el código de JavaScript) sin necesidad de realizar ningún cambio.

En las siguientes secciones se comentará como se ha implementado el frontend.

4.2.1 Arquitectura

La arquitectura más común y simple en el desarrollo de una aplicación web es el MVC (modelo-vista-controlador), correspondiendo el front con la vista. Si bien este patrón permite desarrollar aplicaciones pequeñas de una forma rápida, conlleva una serie de problemas para un proyecto con un tamaño mediano o con proyección a crecer.

En una aplicación con MVC, el modelo está acoplado a la vista, lo que rompe con el Principio de Inversión de Dependencia, ya que cualquier cambio en la vista implica un cambio directo en la lógica de negocio (modelo) y repercute en la mantenibilidad y facilidad de extensión del código.

En este proyecto, y con el objetivo de eliminar cualquier acoplamiento o dependencia entre distintos artefactos, se ha planteado una arquitectura de microservicios en la que cada parte del sistema está dividida en un servicio independiente, que puede tener a su vez una arquitectura interna determinada.

Como se puede deducir, el frontend se corresponde con un microservicio al que da forma una arquitectura hexagonal con una pequeña variación en su implementación. Como se ha mencionado en apartados anteriores, véase la sección 3.2, una arquitectura hexagonal está dividida en tres capas, a saber: dominio, aplicación e infraestructura. Es en esta última, la infraestructura, donde se debería encontrar el código respectivo a la interfaz gráfica, sin embargo, esto ha sido levemente modificado, separando la vista en una capa diferente con el fin de que resulte más intuitivo a cualquier programador que no esté habituado a esta arquitectura. Además, la capa de vista está dividida a su vez en UI (interfaz) y datos (Store).

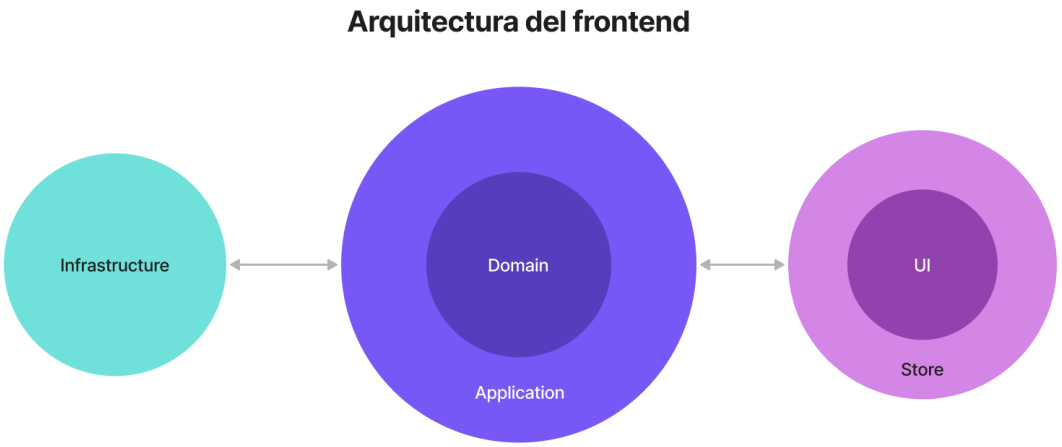


Figura 13. Arquitectura del frontend

La capa de dominio es la responsable de gestionar las validaciones y la lógica. El dominio está envuelto por la capa de aplicación, que sirve de orquestador entre este y la infraestructura/vista, y contiene los casos de uso, es decir, las acciones que pueden ser realizadas. A su vez, la capa de infraestructura se encarga de enviar y recibir peticiones del exterior, así como contener adaptadores para elementos externos a nuestro negocio. Finalmente, la capa de vista contiene todo lo necesario para representar los datos que se manejan en una interfaz gráfica. Para ello dispone de una capa intermedia llamada Store, que se encarga de mantener el estado del sistema y gestionar el flujo de datos que llega a la vista; y una capa UI, que es donde se encuentra React y HTML.

4.2.2 Patrón flux

El control de estado es una parte esencial en cualquier aplicación a la hora de manejar información, por lo que debe ser resiliente a inconsistencias, es decir, la representación que el usuario ve, debe ser fiel a los datos reales del sistema.

Flux es un patrón de arquitectura enfocado en gestionar el flujo de datos de una aplicación de manera unidireccional, lo que permite eliminar problemas de inconsistencia entre el estado de la aplicación y la representación. Además, reduce la complejidad debido a que todos los datos siguen el mismo flujo unidireccional.

El patrón flux está compuesto por los siguientes elementos:

- **Vista:** Contiene el componente de React (o cualquier otra librería) y el HTML necesario para representar información. Emite acciones cada vez que quiere transmitir algún tipo de dato y se actualiza a partir de una store.

- **Acción:** Es un evento que contiene información. Las acciones pueden ser emitidas desde la vista o desde cualquier componente que necesite comunicarse con la vista o influir en el estado de la aplicación.

- **Dispatcher:** Recibe las acciones emitidas y las redistribuye hacia las stores adecuadas.

- **Store:** Almacena y gestiona el estado de la aplicación. Los componentes store reciben acciones del dispatcher y las procesan para generar un nuevo estado mediante una función reductora. El estado es inmutable, por lo que no se puede actualizar, si no que se genera uno nuevo a partir del anterior estado y la acción entrante. Por cada cambio de estado, actualiza las vistas asociadas a dicha store.

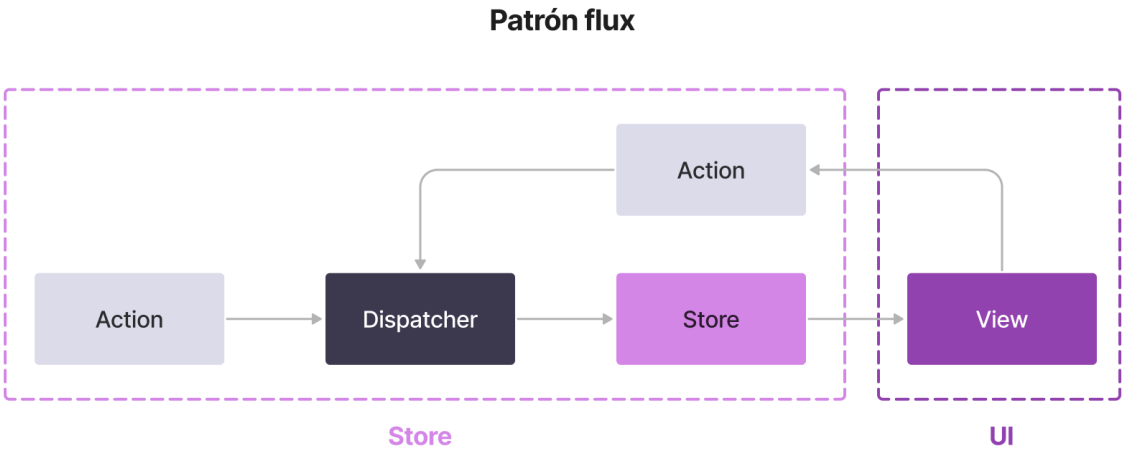


Figura 14. Patrón flux

4.2.2.1 Implementación

Todo lo comentado hasta ahora respecto al patrón flux se mantenía dentro del contexto teórico, pero para poder utilizar este recurso en un proyecto, es necesario implementar este patrón en la aplicación. Hay múltiples librerías open source que permiten utilizar este concepto, entre las que destacan Redux y Flux (librería de Meta).

Sin embargo, para este proyecto se ha decidido hacer una implementación propia con el objetivo de tener un mayor control y poder obtener una solución personalizada. Si bien Redux y Flux (Meta) permiten el uso de TypeScript, estas librerías no están pensadas para ello, y obligan al programador a trabajar con objetos de JS, por tanto, debido a la naturaleza de este proyecto, una implementación nativa en TypeScript se amolda mejor a los requerimientos del mismo.

Aunque la implementación de esta librería no es motivo de este trabajo, en el anexo 2 se puede encontrar un enlace a la solución desarrollada (que es gratuita y está abierta al uso de cualquier desarrollador).

4.2.2.2 Integración en código

Como se ha comentado en el apartado anterior, se ha optado por implementar una librería de flux desde cero utilizando TypeScript. En esta sección se comentará cómo se ha integrado en el código del frontend toda la teoría vista previamente.

Partimos de un componente de React que contiene el HTML necesario para representar una página. Este componente recibe el nombre de “contenedor” y representa una única página de la aplicación. Este contenedor puede tener a su vez otros componentes de React que son los encargados de representar partes específicas de la interfaz.

Cada contenedor debe estar asociado con una implementación de un objeto Store, que es escuchado por un listener que actualiza el contenedor cada vez que el estado almacenado en el store cambia. Como hemos visto en la figura 14, las vistas (contenedores) solo pueden ser actualizadas a través de un store. A continuación, se comentará cómo funciona un store.

Un store está formado por tres elementos: el estado actual del sistema, una lista de acciones y un conjunto de funciones reductoras. El estado almacena la información actual de la aplicación, la lista de acciones contiene los identificadores de las acciones que resultan relevantes porque afectan de una determinada forma al estado que dicho store está almacenando, y finalmente, cada acción del store está asociada a una función reductora que genera un estado a partir de la acción entrante y del estado actual del sistema.

Diagrama de un store

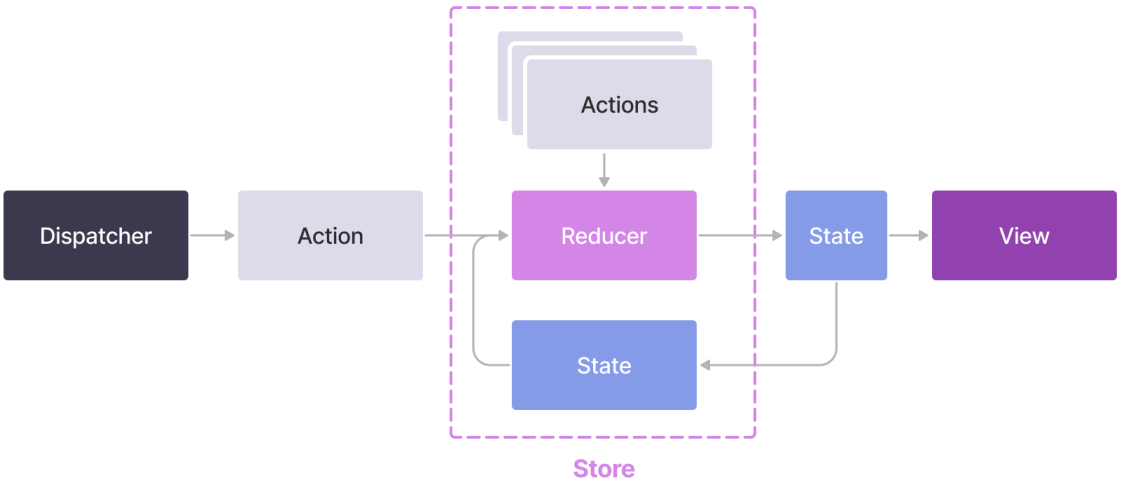


Figura 15. Diagrama de un store

Para que las acciones lleguen a una determinada instancia de un store, se necesita un dispatcher. Puede haber múltiples dispatchers distintos (uno por cada store normalmente), pero solo puede haber una única instancia de cada dispatcher, es decir, sigue un patrón Singleton. Un dispatcher se encarga de recibir un determinado tipo de acciones y enviarlas a los stores asociados. Además, puede contener una lista de funciones que se ejecutan únicamente cuando se recibe una acción o cuando se ha completado otra función anterior.

Estas funciones que están pendientes de ejecutar hasta que se recibe una determinada acción reciben el nombre de *thunk* y se definen en la creación del dispatcher. Son las responsables de, por ejemplo, ejecutar un servicio de aplicación que obtenga los datos del backend cuando llega una acción de carga de página.

Por último se encuentran las acciones, que son objetos simples que modelan un evento y que tan solo contienen un identificador y un payload, que normalmente corresponde con un objeto que modela el estado de la aplicación, ya que este payload será procesado por un *reducer* en un determinado objeto store.

En la siguiente imagen se muestra un diagrama con el funcionamiento completo del frontend con el patrón flux ya integrado.

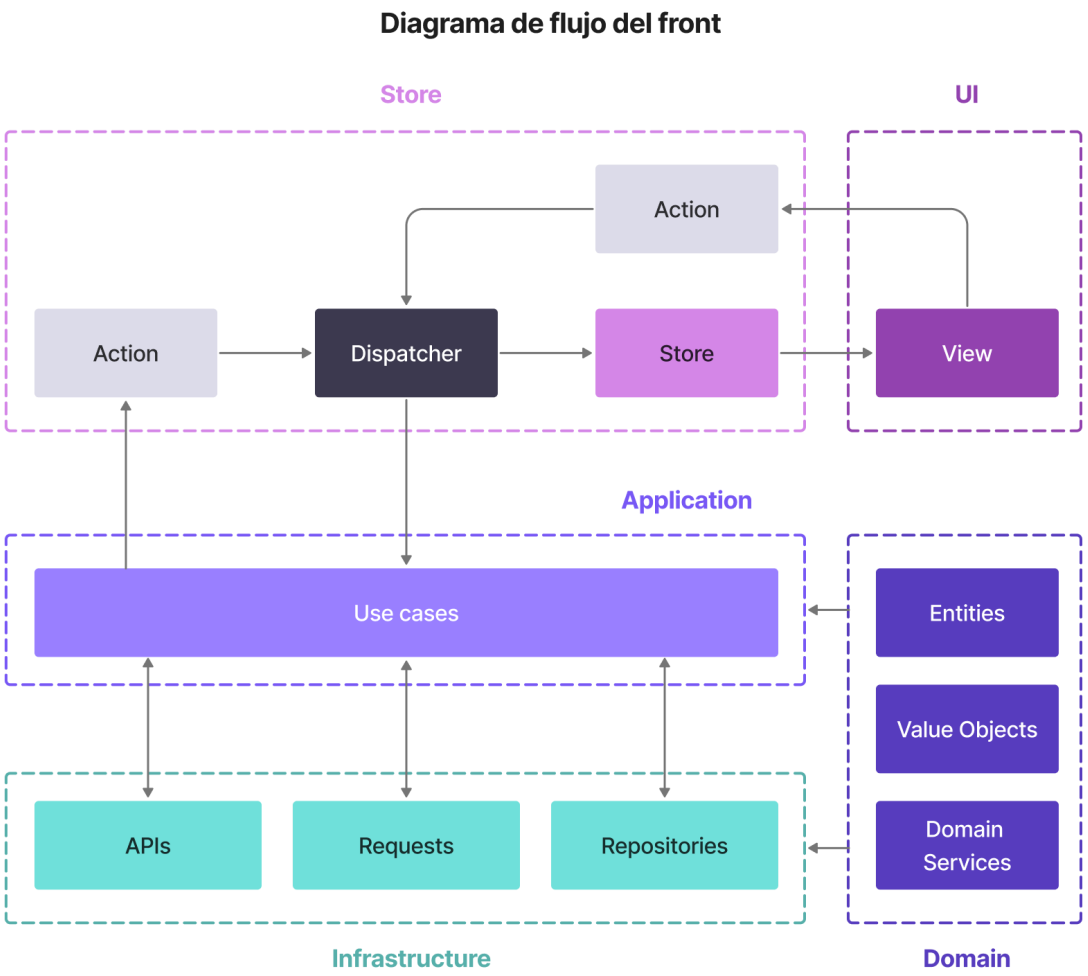


Figura 16. Diagrama de flujo del front

4.2.3 Paquetería

Como se ha comentado anteriormente, el frontend sigue una arquitectura hexagonal, pero, esto puede parecer un poco abstracto por lo que a continuación se mostrará la paquetería resultante a la hora de implementar la arquitectura.

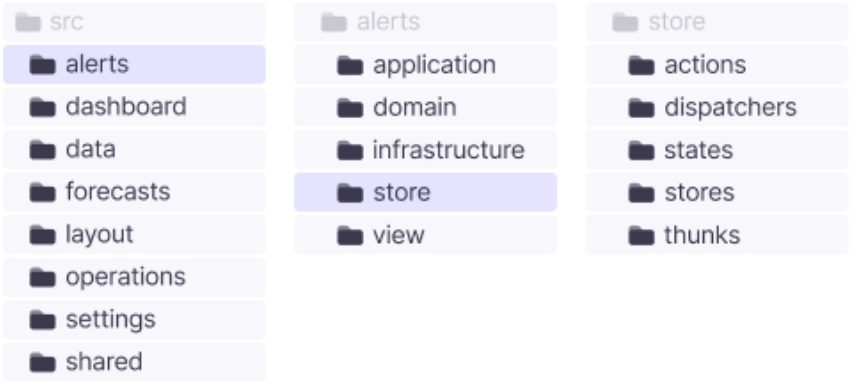


Figura 17. Paquetería

Como se puede ver en la anterior imagen, se ha agrupado cada página o sección de la web en un módulo (alerts, dashboard, data…). Además, hay dos casos especiales: layout y

shared. Estos dos módulos no corresponden con ninguna sección de la web, si no que contienen distintos submódulos que conforman un contexto propio.

Cada módulo genérico contiene una carpeta para cada capa de la arquitectura planteada. Esto se comentará con más detalle en una sección posterior. Por otro lado, los módulos de *shared* y *layout* contienen varios submódulos cuyas funcionalidades son utilizadas en los módulos generales. La carpeta *layout* contiene varios hexágonos —artefactos con capas de dominio, aplicación e infraestructura— relacionados con distintos elementos que conforman la estructura general de la aplicación como el menú lateral, el header o el sistema de grid (se comentarán más adelante). A su vez, la carpeta *shared* contiene varios elementos genéricos utilizados en todo el proyecto, como los componentes del listado, las gráficas, las tablas, etc.

Si bien las carpetas de *application*, *domain* e *infrastructure* pueden tener un contenido variable dependiendo del módulo —manteniendo siempre el sentido de la arquitectura—, tanto la carpeta *store* como la carpeta *view* siguen la misma estructura para todos los módulos. En la anterior imagen se puede apreciar cómo está compuesta la estructura de la carpeta *store*. Por otro lado, la carpeta *view* contiene, a su vez, otras carpetas como *components*, *containers*, *resources* y *styles* que se encargan de almacenar los distintos componentes de React utilizados en el módulo, los contenedores utilizados como raíz (ver sección 4.2.2.2), los recursos como imágenes o vectores, y los estilos de css respectivamente.

4.2.4 Módulos genéricos

Se ha creado un módulo específico para cada sección de la aplicación. De esta forma, cada uno de estos módulos puede contener un dominio específico que se ajuste a las necesidades de negocio del mismo. A continuación se comentará la implementación del módulo de Dashboard para exponer cómo es el desarrollo de uno de estos módulos, pero podría ser cualquier otro ya que todos siguen la misma estructura y principios.

Primeramente se comentará la capa de vista. En la Figura 19 podemos ver la estructura y el contenido que se muestra en la página de dashboard. Como ya se ha explicado, el menú lateral, el header superior y el banner del título son comunes a todas las secciones, por eso, forman parte del módulo de layout. Esto hace que el módulo de dashboard solo tenga que encargarse de representar varios componentes que contienen información básica para los clientes como un par de gráficas de precios del mercado diario e intradiario y una lista con las últimas operaciones y notificaciones. El contenido a mostrar forma parte de la lógica de negocio, por lo que no se define propiamente en el frontend sino que estos datos son consultados a un servicio en backend (se comentará más adelante).

Para representar la página de dashboard (o cualquier otra página), se necesita un “container” o contenedor. Este elemento es la base de toda la vista, pues es el que se encarga de coordinar todos los componentes de React y gestionar las distintas stores que afectan a la página. En el caso del módulo Dashboard, el contenedor tiene un atributo de clase: *DashboardStore*. El objeto *DashboardStore* implementa una *Store* y es el encargado de gestionar el estado, además tiene un listener asociado de forma que si este resulta

modificado y el estado no es el inicial se emite una acción de carga de página, que en este caso concreto tiene como identificador: *DASHBOARD_REQUEST_LOAD*.

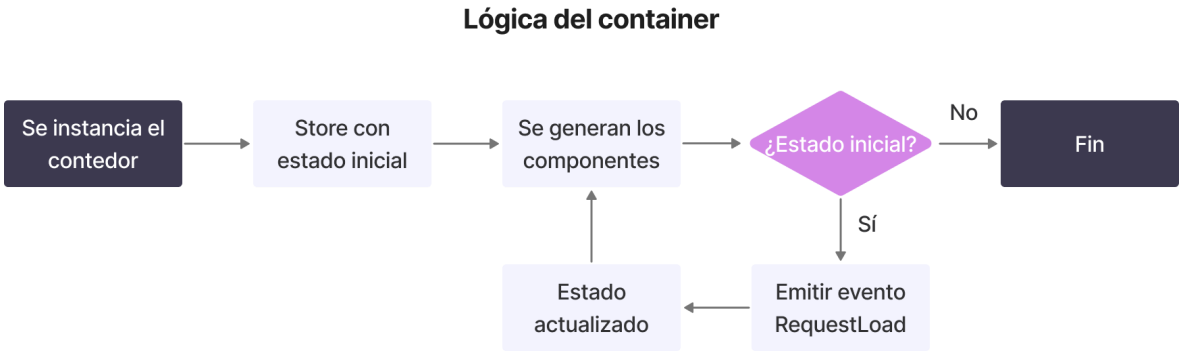


Figura 18. Lógica de un contenedor

Este contenedor tiene, a su vez, un método que determina qué tipo de componentes tiene que renderizar en función del estado. Por ejemplo, si el estado actual contiene un objeto *Chart* (dominio) entonces el container se encargará de instanciar un componente *GeneralChart* (vista, componente de React) y se le pasará el objeto del estado que contiene la información para dicha gráfica. Lo mismo para con las listas que hemos visto anteriormente.

Este punto es la clave de toda esta arquitectura, ya que nos permite delegar la lógica de que datos se quieren mostrar, que es parte del negocio, a un servicio del backend específico, y así, siguiendo el Principio de Responsabilidad Única, el frontend tan solo tiene que encargarse de representar dicha información. Si en un futuro se quisiera añadir un nuevo elemento al dashboard, el programador que tuviese que añadir esa feature tendría que hacerlo en el backend, esto permite que la aplicación del frontend —o aplicaciones, en caso de que hubiese web, app de escritorio y app móvil— no tenga que ser modificada (Principio Abierto/Cerrado). Cualquiera que haya sacado a producción una aplicación que tenga que ser descargada en el dispositivo del usuario puede ver los enormes beneficios que esto genera más allá de la organización del código.

Una vez se han comentado tanto el contenedor como los componentes que se encuentran dentro del mismo, es hora de ascender un nivel en la jerarquía de capas. Ya se ha visto cómo se utiliza un store dentro del contenedor, a continuación se desarrollará como se ha implementado ese store para el módulo Dashboard. Todo en la capa Store parte del estado, y para representar este estado se ha creado una clase *DashboardState*, que contiene un objeto de dominio, en este caso una instancia de *DashboardPage*, que es un agregado que contiene varios objetos de dominio que se comentarán posteriormente.

El objeto *DashboardState* es utilizado como atributo en la clase *DashboardStore* —que a su vez, como se ha comentado anteriormente, es atributo del contenedor—, y siempre debe tener un estado inicial por defecto, para que la store correspondiente pueda ser inicializada aún sin haber extraído los datos. Como se ha visto en la figura 16, los stores necesitan los identificadores de acciones así como la función *reducer* asociada a dicha acción. En este caso, el *DashboardStore* solo tiene un id de acción, que es *DASHBOARD_DISPLAY*, de forma

que cuando se emita esa acción, se genere un nuevo estado a partir del payload de la misma y del estado actual.

Otro elemento fundamental es el dispatcher. En este caso solo se ha utilizado uno, que como se puede intuir se ha llamado *DashboardDispatcher*. A este dispatcher se le ha registrado un thunk (véase la sección 4.2.2.2) llamado *RequestDashboardLoadThunk* que tiene como parámetro de entrada un objeto *Action* y no devuelve nada. En caso de que el identificador de esa acción sea *DASHBOARD_REQUEST_LOAD* —que es el que anteriormente se ha generado al instanciar por primera vez el contenedor— este thunk hace una llamada al caso de uso representado como una clase *LoadDashboardPage*.

Se está siguiendo una arquitectura hexagonal, y por tanto, el caso de uso se encuentra en la capa de aplicación. Este caso de uso se ha modelado a partir de una clase *LoadDashboardPage*. Esta clase contiene dos atributos: *DashboardClient* y *DashboardPageMapper*. La primera clase es una interfaz que se encuentra en dominio y define el contrato que tiene que seguir cualquier implementación que se quiera comunicar con el backend. La implementación concreta de dicha interfaz se encuentra en la capa de infraestructura, pues depende de elementos externos (en este caso, el backend, como se acaba de explicar). El mapper, en cambio, extrae la información necesaria del objeto *DashboardPage* y lo transforma en un *GeneralLayout* que posteriormente se enviará al módulo *Layout* para modificar el título de la aplicación (también se encuentra en infraestructura porque necesita conocer la existencia de otro hexágono). En definitiva, el caso de uso hace una llamada al backend a través del *DashboardClient* y a continuación emite dos acciones, una es *DASHBOARD_DISPLAY*, a la que le pasa un payload con el agregado proveniente del cliente, que será recibida por dispatcher y ejecutará el *thunk* que se había definido previamente en el *DashboardStore*; y la otra es *LAYOUT_DISPLAY*, cuyo payload es el objeto resultante del mapper previamente mencionado. Esta acción será escuchada en el store del módulo *Layout*.

La capa de infraestructura contiene la implementación del cliente del backend y el mapper utilizado para obtener el objeto necesario para enviarselo al layout. Ambos casos se han comentado en párrafo anterior.

Finalmente, la capa de dominio contiene todos los objetos necesarios para modelar el estado de la aplicación. Estos objetos son instanciados en el cliente del backend, después de mapear la respuesta del mismo. Posteriormente se agrupan todos en un agregado (*DashboardPage*) que se devuelve al caso de uso.

Este flujo, aunque ciertamente es complejo, resulta muy fácil de extender e implementar ya que distribuye los distintos artefactos entre capas bien definidas, y permite comunicar distintas partes de la aplicación sin necesidad de que estas se conozcan (elimina el acoplamiento directo). Si bien se ha expuesto el caso concreto del módulo *Dashboard*, el resto de módulos funcionan de la misma forma, por lo que no se hará mucho más hincapié en ello. En la siguiente imagen se puede ver un diagrama de flujo con el funcionamiento completo del módulo de dashboard.

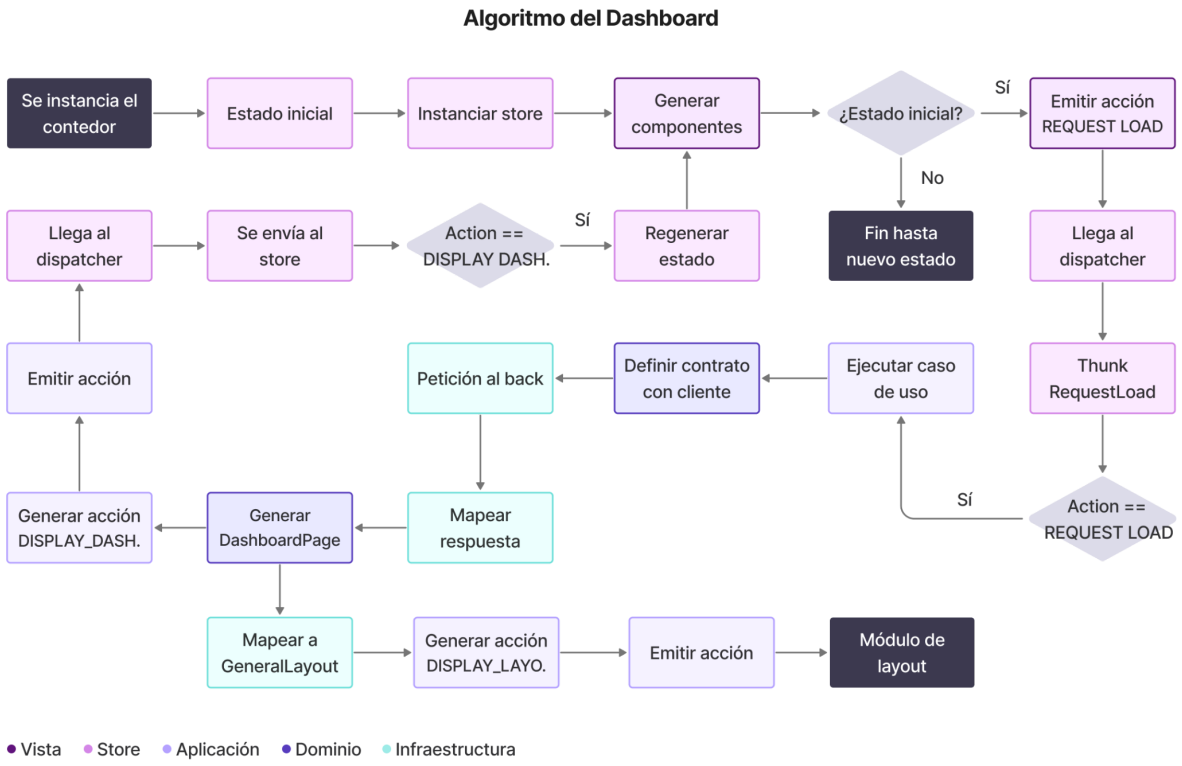


Figura 19. Algoritmo del módulo Dashboard

4.2.5 Layout

Los módulos genéricos que se han mostrado hasta ahora contienen un solo hexágono, sin embargo, el módulo *layout* contiene múltiples hexágonos que encapsulan distintas funcionalidades, aunque algunos de estos hexágonos no contienen todas las capas. A continuación se comentará cada uno de estos submódulos.

4.2.5.1 General Layout

Conforma la estructura general de la aplicación. Es utilizado como la raíz principal de la jerarquía en la que se encuentran el resto de contenedores y componentes. Es un módulo muy ligero que tan solo tiene tres capas: dominio, store y vista. El dominio está formado por dos clases, *GeneralLayout*, que se encarga de almacenar información como el título, el subtítulo, los filtros y las pestañas de la página actual; por otro lado, tenemos la clase *Session*, que se encarga de almacenar las últimas URLs en sesión.

4.2.5.2 Grid

Este submódulo tan solo tiene dos capas, dominio y vista, aunque es sumamente importante para la extensión del proyecto, ya que permite reutilizar el resto de componentes sin necesidad de tocar código a mano.

En la figura 4 se puede apreciar como el contenido de la aplicación se divide en varios recuadros independientes. Cada uno de estos recuadros es generado utilizando lógica de negocio en backend, por eso mismo no se puede fijar el tamaño que cada recuadro ocupa en frontend. Igualmente, el backend no es responsable de conocer qué reglas de diseño debe seguir la interfaz. Esto plantea un problema, ¿cómo permitir que cualquier

combinación de recuadros se represente de forma adecuada sin desarrollar una solución específica en frontend? El submódulo *Grid* proporciona la solución a este problema.

El agregado que modela el espacio donde se pueden colocar recuadros recibe el nombre de *Grid*. Esta clase, pese a ser un agregado, tan solo tiene un atributo, llamado *rows*, que es una lista de objetos de dominio *GridRow*. La clase *GridRow* representa, como su nombre indica, un fila dentro de la malla, por tanto, un *Grid* contiene una lista de filas. A su vez, un objeto *GridRow* contiene dos atributos, uno es un mapa que representa los elementos de la fila, cuya clave es el identificador del elemento y cuyo valor es el espacio que ocupa dicho elemento en la grid (sin tener en cuenta otros elementos); y el otro, es un objeto de dominio, *CapacityCalculator*, que se encarga de calcular si dado un nuevo elemento, la fila tiene el espacio necesario para poder admitir dicho elemento en ella. Esta lógica se ejecuta cada vez que se intenta añadir un nuevo elemento desde la grid, por tanto, si el resultado es negativo, el objeto *Grid* crea una nueva *GridRow* e inserta el elemento en ella. Cada vez que se añade un nuevo elemento, la *Grid* comprueba si hay una *GridRow* que soporte dicho componente, donde la lista de *GridRow* funciona como una cola FIFO, es decir, la primera fila con espacio es la primera que se llena. De esta forma, todos los componentes se recolocan dejando la menor cantidad de espacios vacíos posible.

4.2.5.3 Page Nav

Este hexágono es el encargado de representar la sección de navegación de cada página. Como se puede ver en la siguiente imagen, puede contener tres elementos principales: título, filtros y pestañas.

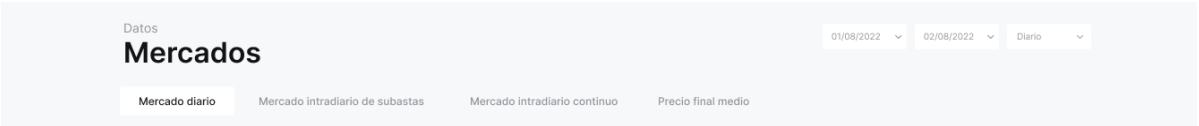


Figura 20. Page navigation

Lo más reseñable de este submódulo es que su estado, almacenado en la *PageNavStore*, es modificado en todas las páginas de la aplicación a través de una acción *DISPLAY_LAYOUT*. Es en casos como este donde se puede apreciar el potencial del patrón flux, ya que *PageNav* es agnóstico y desconocido por el resto de elementos que modifican su estado (al cambiar el título por ejemplo). Esto hace que la extensión del módulo sea independiente al resto de elementos y que se mantenga una alta cohesión entre componentes eliminando a su vez el acoplamiento entre los mismos.

4.2.6 Shared

El módulo no tiene mucho interés en sí mismo, pues es una agrupación de distintos elementos, en su mayoría componentes de React utilizados de forma diversa por los distintos módulos de la aplicación. El contenido de *Shared* no sigue una arquitectura hexagonal, lo cual es otra muestra de la flexibilidad de esta, ya que no es necesario que todos el código siga esta estructura.

4.3 Backend

Para que la aplicación que usa el usuario pueda ser funcional se necesita una aplicación en el servidor que gestione la lógica del negocio y provea de los datos necesarios a cada cliente. En esta sección se comentará la arquitectura del backend, así como cada uno de los microservicios que lo conforman.

4.3.1 Arquitectura

Como se ha comentado en el apartado 4.2.1, el backend está diseñado siguiendo una arquitectura de microservicios, donde cada servicio tiene una responsabilidad concreta y limitada, de esta forma, cada parte del negocio puede evolucionar de forma independiente y en relación a las necesidades de la misma. Al igual que en otras partes del proyecto, se ha utilizado una arquitectura hexagonal para implementar cada uno de los microservicios.

El backend se divide en cuatro microservicios principales: coreback, market data collector, market data reader y watchdog. Así como distintos artefactos de infraestructura como: instancias de una base de datos relacional (PostgreSQL), instancias de bases de datos clave-valor en memoria (Redis), brokers de mensajería (Kafka) y balanceadores de carga (Zookeeper).

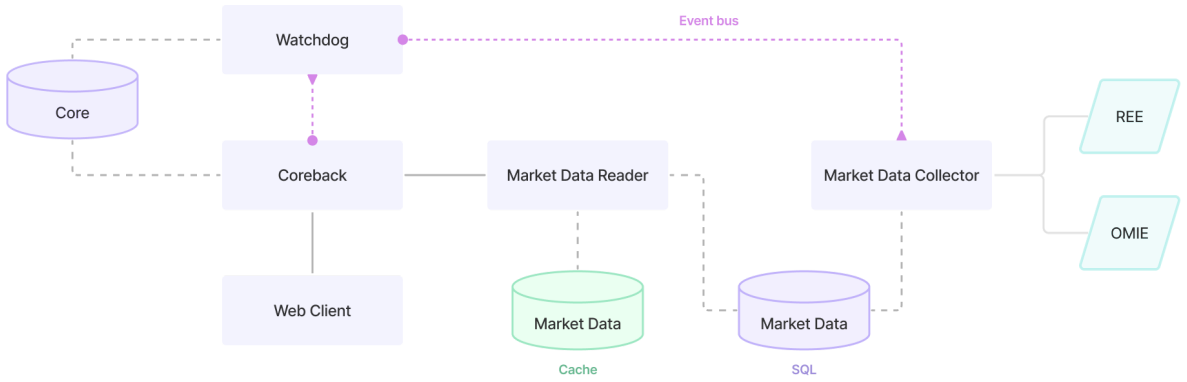


Figura 21. Arquitectura del backend

En esta figura, se puede observar cada uno de los microservicios, representados con un rectángulo gris, y las interacciones entre los mismos. Las líneas grises uniformes corresponden con peticiones directas mediante APIs, las líneas grises discontinuas representan peticiones a bases de datos, y las líneas rosas discontinuas representan comunicaciones mediante eventos. Los cilindros azules y verdes representan bases de datos SQL y cachés respectivamente, y finalmente, los trapecios celestes representan servicios externos.

4.3.2 Coreback

Coreback es el servicio principal del software y contiene la lógica de negocio de la aplicación. Es el encargado de recibir las peticiones del cliente, recuperar y procesar la información que se va a mostrar y mandar la respuesta al frontend. Este módulo está dividido en tres paquetes: *web*, *core* y *support*. A continuación, se comentará más a fondo el objetivo de cada una de estas secciones.

4.3.2.1 Web

Como su propio nombre indica, este paquete es el responsable de manejar todas las comunicaciones con la web, así como de gestionar la lógica de negocio inherente a la misma. Es aquí donde se encuentran los controladores que son llamados desde el frontend. Estos controladores encapsulan la información de las peticiones en DTOs (*Data Transfer Object*), y se los envían a los distintos casos de uso del negocio.

En función de si la parte de la web que se está atendiendo contiene mucha complejidad o no, se sigue una arquitectura hexagonal, o simplemente se construye un modelo de capas básico. Esto es así porque hay casos donde toda la lógica recae sobre un dominio que no guarda una relación directa con la web, de forma que, si construyeramos estos casos de uso siguiendo una arquitectura hexagonal, acabaríamos duplicando todo el dominio y mapeando la respuesta del hexágono que calcula realmente la lógica a un hexágono dentro de paquete *web* sin ninguna lógica adicional. Esto es lo que se conoce como modelo de dominio anémico, es decir, que se ha "modelado el mundo" en vez de los casos reales de la aplicación, por lo que el dominio carece de lógica.

El síntoma más claro de un dominio anémico es que este está compuesto por numerosas *Data class*. Una data class es un *code smell*, es decir, un síntoma que alerta sobre problemas en el código, que se da en situaciones en las que una clase no tiene lógica ninguna y tan solo se emplea para enviar datos de un sitio a otro (aunque esto no es algo malo per sé, cómo puede ser el caso de un DTO, sí que resulta perjudicial a la hora de modelar un dominio, ya que nos está indicando que contiene clases que no tienen ninguna lógica).

En este paquete, en definitiva, se encuentran los controladores para obtener los datos del mercado, el listado de alertas, modificar la configuración del usuario, cargar las distintas páginas de la web, etc. Posteriormente, la lógica necesaria para procesar esas peticiones se puede realizar dentro de *web*, para casos en los que la lógica está relacionada con la web, o se puede realizar en distintos módulos dentro del paquete *core*.

4.3.2.2 Core

Cuando una funcionalidad contiene una lógica de negocio que es transversal y que no está ligada exclusivamente a la web, se implementa dentro del paquete *core*. Un ejemplo de esto es el módulo *authentication*. La lógica relacionada con la autenticación del usuario debería estar en core, ya que si en un futuro se quiere expandir el proyecto y añadir, por ejemplo, una aplicación móvil, se puede reutilizar el módulo llamándolo desde un nuevo paquete *app*. Si esta lógica se situase en el paquete *web*, habría que o bien duplicar el paquete en *app*, o acoplar el controlador de *app* a *web*.

Este módulo está compuesto por distintos hexágonos como: alertas, autenticación o gestión de configuraciones. Cada hexágono es tratado como un servicio independiente del resto, y para que un módulo de la web u otro elemento acceda a cada uno de ellos, se expone un Facade (patrón Facade), que es una clase que mapea los datos entrantes en objetos del hexágono y devuelve el resultado del proceso.

4.3.2.3 Support

Este es el paquete menos relevante. Contiene distintos módulos pequeños que contienen funcionalidades cuyo uso se repite de forma continuada a lo largo de todo el código y que se deben centralizar para no tener que implementarlas de nuevo cada vez que necesiten ser usadas.

Aquí se encuentran módulos como *api*, que establece un contrato para todos los controladores que se comunican con el exterior, *i18n*, que contiene una serie de configuraciones y clases utilizadas para manejar las traducciones de texto, o *priceformatting* que contiene un par de clases utilizadas para formatear precios.

4.3.3 Market Data Collector

Debido a la gran cantidad de datos que se quieren mostrar en la aplicación, así como los requeridos para entrenar la IA, ha sido necesario diseñar un sistema de recolección automática y consulta de datos. En esta sección se comentará el funcionamiento del primero de los dos servicios de datos, encargado de recopilar de forma automática la información de distintos proveedores, procesarla y almacenarla en una base de datos.

4.3.3.1 Tipos de datos

El primer paso a la hora de desarrollar un servicio de recolección de datos, es precisamente, conocer qué datos se necesita obtener. Hay dos funcionalidades principales que se van a proporcionar a partir de esta información, la predicción de precios con IA por un lado, y la facilitación de datos al usuario por otro.

Para satisfacer con estas necesidades necesitaremos, principalmente, los siguientes datos:

Nombre	Tipo
Oferta de compra y venta	Mercado diario
Capacidad de importación y exportación	Mercado diario
Energía proporcionada por: carbón, nuclear, hidráulica, ciclo combinado, eólica, solar térmica, solar fotovoltáica y cogeneración + residuos	Mercado diario
Precio marginal, máximo, mínimo y medio en España y Portugal	Mercado diario
Oferta de compra y venta	Mercado intradiario
Capacidad de importación y exportación	Mercado intradiario
Contratación diaria: volumen y energía	Mercado intradiario
Precio marginal, máximo, mínimo y medio en España y Portugal	Mercado intradiario

Oferta de compra y venta	Mercado intradiario continuo
Precio marginal, máximo, mínimo y medio	Mercado intradiario continuo
Energía comerciada: España, Portugal y, MIBEL	Mercado intradiario continuo
Demanda final: Valor y variación del índice IRE General, industrial, servicios y otros.	Demanda
Demanda real, prevista y programada (total, peninsular, Canarias, Ceuta y Melilla)	Demanda
Estructura de la generación	Generación
Evolución (renovable y no renovable)	Generación
Emisiones	Generación
Disponibilidad de la red	Transporte
Pérdidas	Transporte
Distancia recorrida a: 400, 220 y 132 kV	Transporte
Balance internacional	Balance
PVPC vs Precio spot	PVPC vs Precio spot
Temperatura media, mínima y máxima	Clima
Velocidad y dirección del viento	Clima
Humedad y precipitaciones	Clima

Tabla 1. Tipos de datos.

4.3.3.2 Fuente del dato

Una vez se han definido los datos que se necesitan obtener, se debe encontrar el origen de la emisión de dicha información. En este caso, el mercado eléctrico está regulado por dos entidades, OMIE y REE. OMIE se encarga de regular el mercado diario e intradiario, donde se establece el precio de la electricidad, por tanto, es la fuente de todos aquellos datos que traten sobre precios y transacciones en el mercado. Por otro lado, REE (Red Eléctrica Española), gestiona la infraestructura mediante la cual circula la electricidad en España, por lo que puede proporcionar datos fiables sobre la generación eléctrica, los intercambios fronterizos, los balances, las fuentes de generación, etc.

Ambas organizaciones disponen de una API pública, lo que simplifica y abarata la obtención de datos. Para poder hacer peticiones y gestionar las respuestas de estas APIs, es necesario implementar un mapper que transforme el dominio de la aplicación según el contrato establecido por estas APIs. En este caso, REE sí que proporciona un contrato de tipo REST, sin embargo, para acceder a los datos de OMIE, hay que descargar y formatear

una serie de ficheros csv. En el siguiente apartado se comentará cómo se han implementado ambos casos.

4.3.3.3 Arquitectura del servicio

En esta sección, se presentará el diseño de la arquitectura elegida para implementar este servicio, que sólo cuenta con dos paquetes principales: *collect* y *ree*. El primer paquete contiene la funcionalidad principal del servicio, es decir, los casos de uso de captación de datos. Por otro lado, el paquete ree es un paquete complementario que contiene la implementación de un cliente para hacer peticiones a la API de Red Eléctrica Española.

El paquete collect tiene a su vez varios módulos para cada tipo de datos: *balance, demand, dailymarket*, etc. Cada submódulo contiene tres carpetas para el dominio, los casos de uso y la infraestructura.

4.3.3.4 Captación en REE

Como se ha comentado anteriormente, el paquete ree contiene la implementación del cliente que se usa para comunicarse con la API de REE. Se ha tomado la decisión de separarlo en un módulo distinto debido a dos puntos:

- Hay muchos casos de uso que necesitan comunicarse con REE.
- La lógica de negocio de un caso de uso no debe conocer el origen de los datos.

El primero punto se explica debido a que los casos de uso que implican obtener información de tipo balance, demanda, generación o transporte entre otros, necesitan acceder a la API de REE, ya que esta es la empresa que monitoriza las redes de transporte y distribución de la electricidad. Por otro lado, siguiendo la filosofía de DDD, la lógica de negocio de un caso de uso, como por ejemplo, obtener la evolución de la demanda en barras de central en España, no debe conocer al proveedor de los datos, ya que esta información no forma parte de la lógica de negocio en sí, sino que es un detalle de implementación que debe situarse fuera del dominio. Teniendo en cuenta ambas necesidades, se decidió separar el cliente en un hexágono propio, de forma que todas las comunicaciones están centralizadas en este módulo, se eliminan duplicidades (DRY) y a su vez, la lógica de negocio es agnóstica de la fuente de datos.

A continuación, se muestra un ejemplo del caso de uso de la obtención del balance eléctrico:

Dentro de la carpeta *collect*, que almacena todos los casos de uso del servicio, encontramos un módulo llamado *balance*, que a su vez, está dividido en *domain*, *application* e *infrastructure*. Como ya se ha explicado numerosas veces en este trabajo, en la carpeta *domain* se encuentra la lógica de negocio, en este caso, clases que modelan la información que se quiere obtener, además de contener las validaciones y los procesados que deben cumplir los datos antes de ser almacenados. Es aquí donde encontramos el puerto, *BalanceClient*, que define el contrato que debe seguir cualquier adaptador que lo implemente para proporcionar los datos obtenidos de un servicio externo.

A continuación, tenemos el paquete *application*, que contiene el caso de uso *ObtainAndStoreElectricBalance* cuya finalidad es obtener los datos y almacenarlos en una base de datos.

Para terminar con este hexágono, se encuentra la capa *infrastructure*, donde se encuentra el controlador que hace público el caso de uso con el exterior, el repositorio que accede a la base de datos para guardar los datos obtenidos, y finalmente, el adaptador del puerto *BalanceClient*, que en este caso se ha llamado *ReeBalanceClient*.

Es en *ReeBalanceClient* donde se puede apreciar las bondades de esta solución, ya que esta clase no conoce nada del contrato establecido con la API de REE, ni la url ni los datos que se le tienen que pasar en el cuerpo de la petición. Gracias a haber creado una capa de abstracción entre la API y los casos de uso (el módulo *ree*), este adaptador solo tiene que llamar a la fachada (patrón *Facade*) del módulo *ree* con las particularidades de cada caso de uso. En el anexo 18 se muestra un fragmento de código donde el adaptador del módulo *balance* hace una petición al *facade* del módulo *ree* para obtener los datos del balance desde la API de REE sin necesidad de conocer nada de la misma:

4.3.3.5 Captación en OMIE

En un principio, la idea de crear un cliente para obtener los datos de OMIE al igual que se ha hecho con REE puede resultar atractiva, pero en esta ocasión se ha optado por implementar directamente el cliente en la capa de infraestructura de cada caso de uso individual. La causa que motiva esta diferencia es la disparidad en la estructura tanto de la petición como de la respuesta que proporciona OMIE. Anteriormente se mencionó que OMIE no proporcionaba una API REST como sí lo hacía REE, sino que para acceder a los datos de esta organización, se debe descargar un fichero desde su web y formatear la respuesta a datos manipulables, esto hace que sea imposible de generalizar sin caer en la sobre-ingeniería.

Al igual que se ha hecho en el apartado anterior, a continuación se detalla el flujo que seguido para guardar datos de OMIE, en este caso, el precio en el mercado diario:

Este caso de uso corresponde con el módulo *dailymarket/energyprice*. Al igual que todos los demás módulos, contiene tres carpetas para las capas de *application*, *domain* e *infrastructure*. En la capa de aplicación hay dos casos de uso, *ObtainAndStoreDayPrices* y *ObtainAndStoreIntervalPrices*, que sirven para obtener y guardar los precios de un día (hay un precio por hora) y de un intervalo de fechas respectivamente.

El proceso de obtención de este dato es más lento y costoso que los anteriores ya que se tiene que descargar y procesar un archivo por cada día que se quiere consultar, por eso el dominio, además de modelar la información recogida, contiene la lógica necesaria para crear y procesar *bulk loads*. Por ejemplo, haciendo una petición al servicio para obtener los precios de un mes entero, la conexión quedaría abierta durante varios segundos o incluso minutos. Esto haría colapsar el servicio con unas pocas peticiones, además de que mantendría al cliente ocupado durante periodos largos de tiempo (si este no cierra la conexión). Para solucionar esto, cuando se realiza una petición de este tipo, se inicia un proceso asíncrono que encola los días solicitados y crea un carga masiva (*bulk load*). A

continuación, se envía la respuesta para que el cliente pueda cerrar la conexión, y es entonces cuando paralelamente, el servicio comienza a descargar y procesar cada uno de los días que han sido encolados con anterioridad mientras puede seguir atendiendo peticiones de otros clientes.

Cada vez que se procesa un día, se hace una llamada a un puerto (interfaz) de dominio, que establece el contrato para obtener precios a partir de un día. Esta interfaz es implementada por un adaptador en infraestructura, que es el que tiene la lógica para generar la url necesaria para descargar el fichero de OMIE y mapear la respuesta a un objeto de dominio. En el anexo 16 se muestra un fragmento del código del adaptador.

La clase *FileRetriever* se encuentra en una carpeta *shared* y su función es generalizar el proceso de hacer peticiones a urls y obtener los ficheros que se descarguen de esa url. Esto no es exclusivo de OMIE, sino que se puede utilizar con cualquier otra página web.

Continuando con la infraestructura, en este paquete también encontramos los controladores que reciben las peticiones para obtener los precios de un día o un intervalo de días, los DAOs (*Database Access Object*) para hacer consultas a la base de datos, el adaptador del cliente de *bulk loads* y el productor de eventos de Kafka utilizado para enviar eventos cada vez que se captura un precio, cuya finalidad se comentará más adelante.

4.3.4 Market Data Reader

Este servicio es el encargado de acceder a los datos almacenados en la base de datos por el Market Data Collector. Su función es proporcionar una API a los demás servicios, para que estos puedan consultar cualquier información con solo hacer una petición.

4.3.4.1 Arquitectura del servicio

Este servicio está dividido en dos módulos: *read* y *client*. El módulo *client* contiene la clase *Response* que se envía al exterior por los controladores del servicio así como los endpoints publicados. Por otro lado, el módulo *read* contiene toda la funcionalidad en sí del servicio, que le permite realizar búsquedas con distintos criterios y devolver los datos obtenidos al cliente. En la siguiente página se muestra el diagrama UML simplificado con el flujo principal del servicio. Hay muchas clases que no se han añadido para facilitar la comprensión del diagrama.

Este servicio debe ser capaz de leer una gran variedad de tipos de datos distintos, lo que resultó un desafío a la hora de diseñarlo. Por un lado, hay que proporcionar a los clientes un contrato que permita buscar cualquier tipo de dato con las peculiaridades de cada uno (cada tipo de dato tiene parámetros distintos) y por otro lado, hay que modelar el dominio impidiendo el acoplamiento a los datos para que la extensión del mismo sea sencilla y no se produzca una explosión de clases que haga imposible la escalabilidad del servicio.

Con estas dos necesidades claves se planteó la siguiente solución: El cliente puede realizar la búsqueda pasando en la petición una serie de parámetros como las fechas de inicio y fin, el intervalo de tiempo entre datos, el territorio o el tipo de dato. Este tipo de dato es una clase abstracta vacía que es extendida por n clases (una por cada tipo de dato). Cada una de estas clases contiene una anotación con su nombre en la petición (que será utilizado

para instanciar automáticamente la clase) y unos atributos que definen los campos que se quieren obtener de cada tipo de dato.
Posteriormente, la petición se mapea a un objeto de dominio *Criteria* que contiene una lista de *DataField*, que es otro objeto de dominio que contiene el tipo de dato que el cliente ha solicitado, que ha sido mapeado desde una clase hija de la clase abstracta previamente mencionada.

Luego, un servicio de dominio selecciona el repositorio adecuado para cada *DataField*, que lo mapea a una entidad para obtener los datos solicitados de la base de datos. Finalmente, los datos obtenidos se mapean a un objeto de respuesta para devolverlos al cliente.

En el anexo 6 se muestra un diagrama UML simplificado del servicio.

4.3.4.2 Implementación de la arquitectura

En el punto anterior se ha mostrado un diagrama UML con las clases más importantes que conforman el servicio y se ha explicado el funcionamiento del mismo. En este apartado se va a comentar detalladamente como se ha implementado cada uno de los pasos definidos con anterioridad.

El flujo comienza en el controlador, que mapea la información de la petición automáticamente a un DTO cuyo nombre es *Search*. En el anexo 15, se puede ver un fragmento de código con la URL de la petición y el cuerpo de la misma, así como el controlador que escucha dicha petición.

El objeto *Search* contiene la información de la petición como el territorio, las fechas o los tipos de datos solicitados. El mapeo de la petición al objeto *Search* se hace de forma automática gracias a las anotaciones *@JsonProperty* y *@JsonTypeName*.
Como se puede observar en el anexo 8, en la clase *Search* se ha definido que el atributo *types* que llega en el json de la petición se debe mapear a una lista de *Type*. Hay múltiples clases que extienden de *Type*, y cuyos atributos son los posibles parámetros de cada tipo de dato, que pueden ser solicitados o no en la petición. En el ejemplo, se le pasa un único tipo *dailyMarketPrice*, que como se puede ver en el código, tiene muchos posibles parámetros (precio marginal en España, precio máximo en España, etc) pero en la petición tan solo se han solicitado dos de ellos, *spainMarginalPrice* y *spainMaxPrice*. En este caso, ambos parámetros son booleanos, ya que pueden ser solicitados o no, pero hay otros casos como *negotiatedEnergy*, que en caso de ser solicitados, pueden ser devueltos con distintos valores (unidades de energía o porcentajes en este caso).

Una vez mapeada la petición, el controlador hace una llamada al caso de uso *ReadMarketData* con el DTO de la búsqueda. El caso de uso se encarga de transformar ese DTO (*Search*) a un objeto de dominio *Criteria*, y de pasárselo al servicio de dominio *MarketDataReader*, que tiene la lógica de obtención de datos. En el anexo 17 se puede observar el caso de uso.

A continuación, el servicio de dominio *MarketDataReader* extrae una lista de búsquedas por módulo (*ModuleSearch*) a partir del objeto de dominio de la búsqueda original (*Criteria*), y luego, recupera el repositorio concreto para cada módulo (los módulos corresponden con

los tipos de datos) y realiza la consulta a base de datos. Una vez se han obtenido los datos, se mapean a DTO salida y se le envían al cliente que realizó la petición.

4.3.5 Watchdog

Este pequeño servicio es el encargado de notificar cuando una alerta ha superado el límite definido por el usuario. Los usuarios de la aplicación pueden, por ejemplo, crear alertas que los notifiquen cuando el precio de la electricidad en el mercado diario supera un umbral.

4.3.5.1 Mensajería con Kafka

Para satisfacer esta necesidad se emplean colas de Kafka, que es un servicio de streaming de eventos, que permite a dos artefactos distintos comunicarse sin que estos se conozcan. Tiene cuatro conceptos principales:

- **Broker**: Es un servidor en el que se está ejecutando Kafka. Almacena el evento y permite que se pueda acceder a él. Varios brokers se pueden juntar en un cluster.
- **Evento**: Transporta información sobre un suceso en un determinado momento. Se envía a una dirección conocida como topic.
- **Producer**: Es el emisor. Crea eventos y los envía a un topic.
- **Consumer**: Es el receptor. Recibe el evento cuando es enviado a uno de los topics que están siendo escuchados.

4.3.5.2 Funcionamiento del servicio

En este servicio no encontramos controladores o fachadas, los puntos de entrada a nuestros casos de uso son *consumers* de Kafka. Estos consumidores se encuentran en la capa de infraestructura como es natural, y reciben la información de los eventos que llegan a los topics que tienen definidos.

Cuando un evento es emitido a uno de los topics que están siendo escuchados por un consumidor, este mapea la información contenida en el evento a un DTO de entrada —al igual que se hace en un controlador— y lo envía al caso de uso. Una vez en el caso de uso, se llama al dominio que contiene la lógica para cada funcionalidad.

En la sección 4.3.3.5 se comentó como una vez que el *Market Data Reader* obtenía el precio de la electricidad en el mercado diario desde OMIE, este persistía el dato en una base de datos y emitía un evento por Kafka. Cada vez que se captura un precio, este servicio emite un evento al topic *price.collected*, que a su vez, está siendo escuchado por la clase *PriceConsumer* en el servicio *watchdog*. Posteriormente, se obtiene el precio y la fecha del payload del evento y se construye un DTO que se envía al caso de uso *CheckAndNotifyExceededThresholds*. Una vez aquí, se recupera de la base de datos el precio en la sesión anterior para saber si la variación es positiva o negativa. En función del resultado de la operación anterior se obtiene la lista de alertas creadas por usuarios para precios que han sido sobrepasados superior o inferiormente, y finalmente, se crea un notificación sobre las alertas recuperadas que se persiste de nuevo en base de datos. Adicionalmente, se emite un nuevo evento al topic *alert.threshold-exceeded* con la información de la alerta que ha sido superada para que otro servicio externo pueda notificar al usuario en tiempo real.

Al centralizar la gestión de alertas en este servicio, se evita que el *MarketDataReader* deba absorber la lógica de negocio que se ha comentado en el párrafo anterior, y al usar Kafka, se consigue que ambos servicios permanezcan desacoplados casi por completo, ya que lo único que comparten es el contrato implícito establecido por el evento.

4.4 Predicción de precios con IA

Uno de los usos planteados para los datos que se han capturado previamente es el análisis de los mismos con el fin de obtener una predicción fiable en el tiempo. Dada la naturaleza volátil del precio de la electricidad en el mercado diario, se ha establecido como foco la predicción del valor en la siguiente hora.

4.4.1 Análisis del problema

Como se ha mencionado, el objetivo es obtener el precio de la electricidad en el mercado eléctrico diario en la siguiente hora, que está determinado en función de la oferta y de la demanda, por lo que se realizará en base a una serie de datos que influyen tanto en el consumo de electricidad (demanda) como en su generación (oferta).

Como queremos predecir en intervalos horarios, nuestros datos deben ajustarse a esa periodicidad. Por tanto, dado que tenemos múltiples entradas y queremos obtener un único valor a futuro en la salida, estaremos ante un problema que requerirá de un modelo multivariado unistep.

4.4.2 Limpieza de datos

A continuación, se introducirán los datasets empleados, así como el procesamiento de los mismos.

4.4.2.1 Datasets

La primera variable, y la más importante, es el precio de la electricidad, que es además la variable que queremos predecir. Incluso si solo se tuviese este dato, podría optarse por implementar un modelo ARIMA que estimase el precio a futuro.

El siguiente grupo de variables que se emplearán serán los distintos modos de generación (carbón, nuclear, ciclo combinado, eólica, solar fotovoltaica, solar térmica, hidroeléctrica…). Esto es relevante ya que a mayor porcentaje de generación renovable, menor es el precio de la electricidad, ya que estos métodos de generación tienen costes muy bajos, lo que permite a las empresas generadoras poner sus MWh prácticamente a coste cero, en cambio, las centrales de ciclo combinado y de carbón están sujetas al precio del gas y del carbón respectivamente, lo que implica que las empresas generadoras repercutan los costes en el precio de venta.

Como se puede suponer del punto anterior, la generación renovable depende del clima en un momento dado, por tanto, también se utilizará un dataset de datos climáticos en varios puntos de la península: Madrid, Barcelona, Bilbao, Valencia y Sevilla. Estas ciudades no solo están distribuidas por todo el territorio, si no que además, son los núcleos urbanos con mayor población del país, por lo que también podemos utilizar el dataset para representar

los hábitos de consumo de la población nacional, y por tanto, asociarlos con su repercusión en la demanda de electricidad.

Del mismo modo que los recursos naturales como el viento o la luz solar afectan a la generación renovable, el precio del gas natural y del carbón afectan a la producción no renovable. En este caso, se descartará el uso del precio del carbón, ya que la generación de electricidad por este medio es de menos del 3% del total. Por otro lado, no se encontró un dataset del precio del gas natural en España, pero ya que esta variable es muy homogénea en todo el mundo debido que a que es un mercado oligopolístico, se ha empleado el precio de los futuros de gas natural en Estados Unidos (podrían haberse utilizado los futuros del gas holandés, pero se disponen de menos datos históricos).

4.4.2.2 Preparación de los datos

Todos los datasets contienen datos que no son relevantes, ya sea porque contienen un gran número de datos vacíos o porque sus propios valores no son determinantes a la hora de predecir el precio de la electricidad. En el dataset de variables de generación se han eliminado algunas columnas como la generación hidroeólica (embalses con generadores hidroeléctricos que aprovechan la energía eólica para bombear el agua, solo hay una central en El Hierro), generación de autoproductores (individuos que generan su propia electricidad), generación por turbina de gas, por turbina de vapor o por motores diesel. Respecto al dataset de datos climáticos, se han eliminado algunas columnas no relevantes como la cantidad de nieve, el punto de rocío o la visibilidad.

Todos los datos deben estar medidos con la misma periodicidad, para lograr esto, se realiza un remuestreo horario para cada dataframe estableciendo un rellenado de datos con interpolación lineal. Por ejemplo, para el dataset del precio del gas natural, que está medido de día en día, se crean 23 entradas nuevas por cada día cuyos valores se definen utilizando la fórmula de interpolación lineal. Una vez los datos están en la misma escala temporal, se procede a separar el dataset de clima en 5 dataframes distintos (uno por cada ciudad), y finalmente, se fusionan junto con el resto en un único dataframe indexado por fecha.

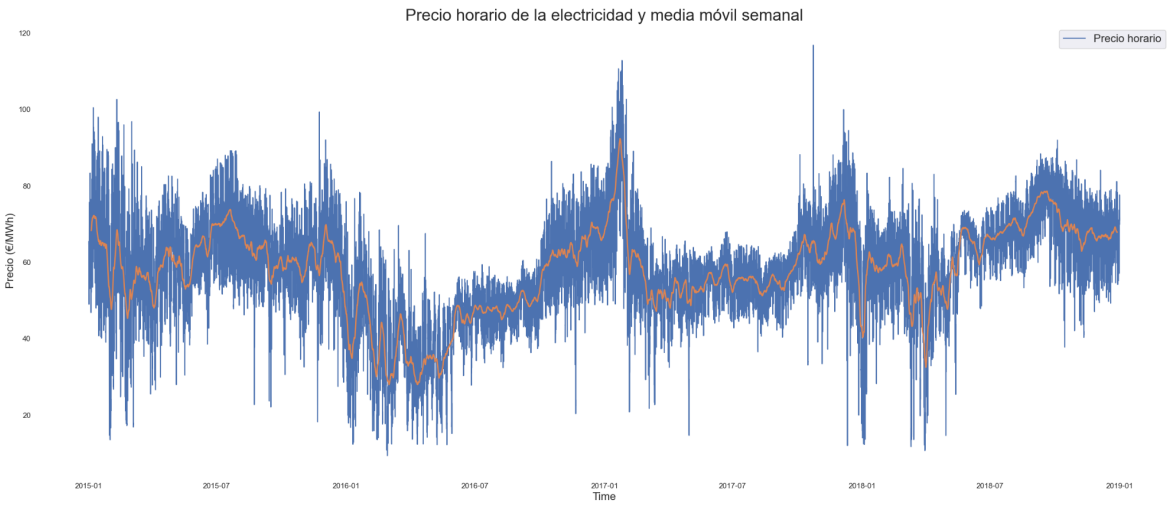


Figura 22. Precio horario de la electricidad y media móvil semanal

4.4.2.3 Prueba de estacionariedad

Para poder implementar un buen modelo que estime con fiabilidad, se necesita conocer si la serie temporal tiene estacionariedad o no. Para lograr esto, se ha sometido al dataframe a un test de Dickey-Fuller aumentado, que determina como de acoplada está la serie temporal a una tendencia a partir de dos hipótesis:

- **Hipótesis nula (H_0):** La serie temporal presenta una raíz unitaria, por lo que la serie no es estacionaria y puede incluir tendencias o patrones no constantes en el tiempo.

- **Hipótesis alternativa (H_1):** La serie temporal no contiene raíz unitaria, por lo que se puede afirmar que es estacionaria, o puede convertirse en estacionaria aplicando una diferenciación.

Parámetro	Resultado
Estadística ADF	-7.046
Valor P	0.0000
Retrasos utilizados	50
Valor crítico (1%)	-3.43
Valor crítico (5%)	-2.86
Valor crítico (10%)	-2.57

Tabla 2. Resultados del test de Dickey-Fuller aumentado.

El valor de la estadística ADF, -7.046, es menor al valor crítico del 1%, por tanto se puede descartar la hipótesis nula con una relevancia del 1%, con lo que se puede concluir que no hay raíz unitaria en la serie temporal y por tanto, o bien la serie ya es estacionaria o bien se puede conseguir la estacionariedad con una diferenciación de primer orden.

También se ha realizado un test de Kwiatkowski-Phillips-Schmidt-Shin (KPSS), para confirmar el resultado del test ADF.

4.4.2.4 Análisis de la correlación (ACF y PACF)

Analizar la autocorrelación permite conocer la relación directa que hay entre una observación en una unidad temporal determinada y sus valores pasados.

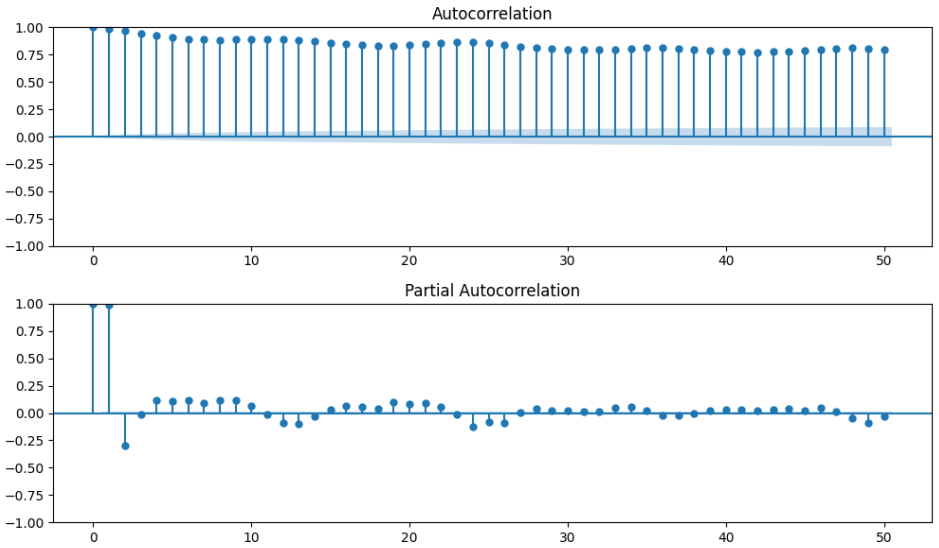


Figura 23. Autocorrelación

Como se puede ver en la gráfica de autocorrelación (ACF), a partir del valor 24 o 25 aproximadamente, la correlación empieza a bajar de 0.8, por lo que se utilizarán los últimos 25 valores de la serie temporal para desarrollar el modelo.

También se ha realizado un estudio de la correlación entre el precio de la electricidad y otras variables, siendo las más relevantes el precio spot (0.73) y el precio del gas natural (0.66).

4.4.3 Feature engineering

A continuación, se van a transformar los datos resultantes de los pasos previos para mejorar el rendimiento y la eficiencia del modelo. La primera “feature” que se va implementar será el día de la semana y el mes al que pertenece una hora concreta.

A partir de ese nuevo dato, se puede generar otra nueva feature que determine si una observación está en horario laboral o no (esto puede ser determinante ya que durante la jornada laboral se consume mucha más electricidad). Otra feature derivada de la primera es el horario de fin de semana, de forma que si una observación se encuentra en sábado o domingo se guarde como valor 1, y si es día laboral, se guarde como valor 2.

A partir del clima también se ha creado una nueva característica. En este caso, se ha asignado un peso a cada ciudad a partir del número de habitantes, ya que cuanta más población, mayor es el consumo.

4.4.4 Modelos

Una vez se tienen los datos procesados, se debe implementar un modelo que prediga el valor con la mayor exactitud posible sin caer en sobreentrenamiento. En este proyecto se han implementado varios modelos como ARIMA, SARIMA o modelos autorregresivos con bosques aleatorios, pero ninguno de ellos proporcionó un resultado óptimo, por lo que se terminó optando por una red LSTM.

El primer paso antes de entrenar el modelo es dividir el dataset en tres partes: entrenamiento (80%), validación (10%) y prueba (10%). A continuación, se normalizan los

datos. En este caso se ha optado por una escala de 0 a 1 y se ha utilizado la función MinMaxScaler de la librería scikit.

El siguiente paso es reducir en la medida de lo posible la dimensión del dataset sin perder información relevante. Con este fin se ha utilizado PCA (Principal Component Analysis). En el anexo 7 puede verse el resultado. Esto reduce la dimensionalidad transformando las variables originales en nuevos componentes que son combinaciones lineales de las mismas, ordenadas de mayor a menor variabilidad. También permite eliminar la correlación de las variables originales, gracias a la transformación mencionada anteriormente.

A continuación, se vuelve a dividir el dataset en batches teniendo en cuenta que se utilizarán los 24 valores previos para obtener el siguiente valor a futuro. Con estos valores definidos, se puede crear el modelo, para el que se ha utilizado la librería Keras:

```
model_lstm = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(100, input_shape, True),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(200, activation='relu'),
    tf.keras.layers.Dropout(0.1),
    tf.keras.layers.Dense(1)
])

model_checkpoint = tf.keras.callbacks.ModelCheckpoint('lstm', True)
optimizer = tf.keras.optimizers.Adam(lr=6e-3, True)

model_lstm.compile(loss=loss, optimizer=optimizer, metrics=metric)
history = model_lstm.fit(train,
                         epochs=50,
                         validation_data=validation,
                         ...)
```

Una vez entrenado y guardado el modelo, puede ser utilizado para predecir valores a futuro, aplicando el mismo procesamiento que se ha utilizado para el entrenamiento, y haciendo la transformada inversa de dicho procesamiento para el resultado. De esta forma, se puede calcular también la precisión del modelo. En este caso, se ha utilizado la raíz del error cuadrático medio para calcular la tasa de error, que ha arrojado un valor de 2.3 €/MWh, lo cuál es bastante aceptable dada la naturaleza volátil del precio de la electricidad.

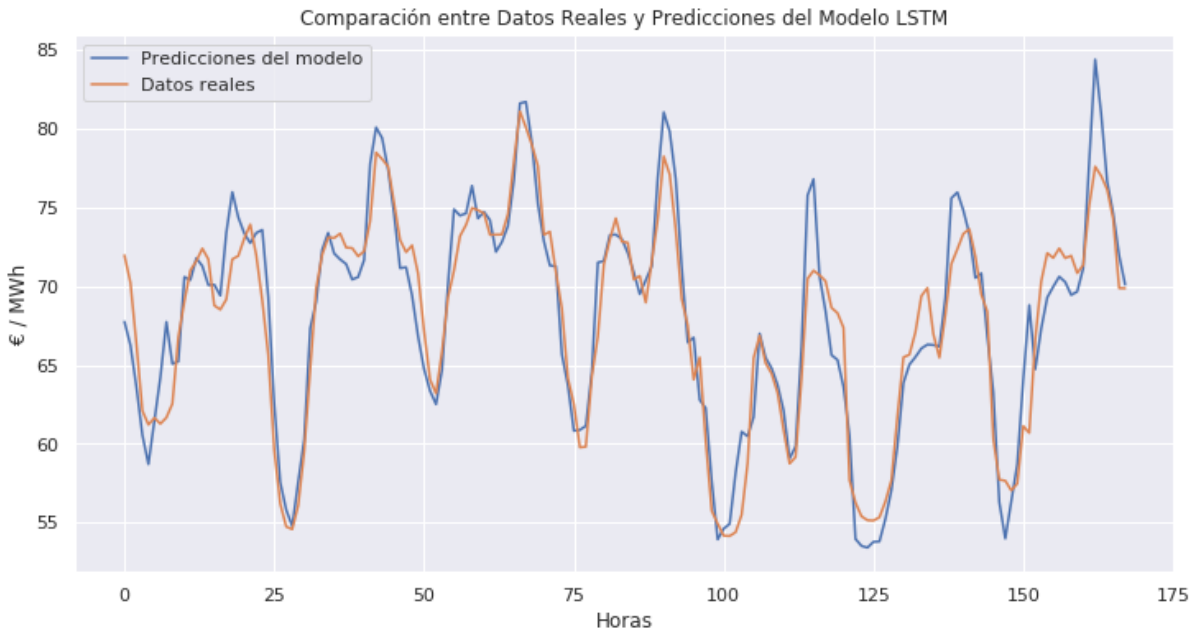


Figura 24. Comparación entre datos reales y predicciones del modelo LSTM

4.4.5 Servicio de predicción

Para poder consultar las predicciones en la web, es necesario hacer que el modelo ya guardado se ejecute y realice los pronósticos. El pronóstico se realiza sobre la hora siguiente a la actual, y es igual para todos los usuarios, por lo que exponer un servicio para que sea consultado por otros es contraproducente, ya que la respuesta será igual independientemente del cliente. Por tanto, la forma más eficiente de proporcionar los datos es a través de la persistencia del resultado, de forma que este solo sea calculado una única vez, y sean los clientes los que accedan a dicho recurso cuando sea necesario.

Esto se puede lograr programando un cron job, que sea el encargado de ejecutar el servicio de forma horaria. A su vez, el servicio consulta los datos correspondientes al intervalo horario que se quiere predecir, y posteriormente, guarda el resultado en la base de datos. De esta forma, es el coreback (o cualquier otro servicio que necesite la información) el responsable de leer la última entrada de la tabla correspondiente para obtener la predicción. Así se logra que cada servicio sea independiente y no esté acoplado, de forma que si uno de los servicios fallase, esto no repercutirá al normal funcionamiento del resto de servicios. Es importante tener en cuenta que esta solución puede causar otros problemas, ya que obliga tanto a los clientes a establecer un contrato con la base de datos, pero para este caso parece ser la solución más conveniente.

5. Validación de los resultados

En todo proyecto es importante validar que los requisitos previos se satisfacen correctamente. Hay varias formas de lograr este fin, pero en este proyecto se han utilizado principalmente tres: tests unitarios, API testing y pruebas manuales.

- **Tests unitarios:** Estos tests permiten construir software robusto y resiliente a los cambios, y además, permite al desarrollador asegurarse de que la funcionalidad implementada mantiene el correcto funcionamiento y cumple con el comportamiento deseado. En este proyecto se han utilizado numerosos test unitarios para asegurar que funcionalidades claves del negocio. En el anexo 3 se muestran algunos ejemplos.

- **API testing:** Si los tests unitarios comprueban que una pequeña funcionalidad se cumple, los test de API tienen el mismo objetivo para casos de uso completos, es decir, funcionan como tests de integración para funcionalidades completas de APIs. La mayor parte de servicios del proyecto utilizan como interfaz para los clientes una API REST, por lo que resulta muy sencillo comprobar si una funcionalidad cumple o no con los requerimientos mediante el uso de herramientas de testeo. En este caso se ha utilizado Postman, en el anexo 4 se han adjuntado distintos ejemplos.

- **Pruebas manuales:** Otra forma de comprobar que se cumplen los objetivos de una determinada funcionalidad es testear dicha funcionalidad de forma manual. Hay varios métodos para hacer esto, pero en este proyecto se han utilizado principalmente ejecuciones de servicios y debugging, que permite visualizar paso por paso el flujo que sigue la ejecución de un programa. En especial, se han utilizado los IDEs de JetBrains: IntelliJ IDEA, WebStorm, PyCharm y DataGrip, que proporcionan excelentes herramientas de debugging.

Además, siguiendo con la metodología Kanban, se han utilizado criterios de aceptación en aquellas tareas en las que se ha tenido que desarrollar alguna funcionalidad.

6. Conclusiones

En este apartado se realizará un análisis de los objetivos y de los requisitos del proyecto. Se comentará cómo se ha solucionado cada uno de los objetivos planteados. Además, se expondrán las soluciones implementadas para satisfacer los requisitos previos.

6.1 Análisis de los objetivos

El objetivo del proyecto, como se ha expuesto en la introducción, es diseñar e implementar una arquitectura de software para una aplicación de trading del mercado eléctrico. Este trabajo se ha llevado a cabo con éxito y se han completado los objetivos enumerados al inicio del documento gracias a los siguientes puntos:

- Se ha diseñado una interfaz gráfica siguiendo las últimas tendencias y que permite a los usuarios acceder a las funcionalidades del producto de forma intuitiva gracias a la disposición en forma de dashboard y a la uniformidad y homogeneidad de los distintos componentes que la conforman.

- Se ha diseñado una arquitectura de software escalable gracias a una correcta distribución en microservicios, que permiten una asignación de recursos específica y resiliencia a fallos. Además, se ha implementado el proyecto siguiendo la filosofía DDD y basándose en una arquitectura hexagonal, que permite escalar la lógica de negocio y aumentar su independencia eliminando su acoplamiento a factores externos.

- Se ha implementado un frontend siguiendo una arquitectura hexagonal y haciendo uso del patrón flux para aumentar la independencia de los distintos módulos y favorecer su crecimiento a futuro.

- Se ha implementado el backend a partir de distintos microservicios independientes específicos para cada parte del modelo de negocio. Cada microservicio se ha implementado siguiendo una arquitectura hexagonal.

- Se ha llevado a cabo un modelo LSTM que permite predecir el precio de la electricidad en el mercado diario en la próxima hora a partir de una serie de variables.

6.2 Análisis de los requisitos

En este punto se va ha exponer la solución aportada para cada uno de los requisitos que se han definido al inicio del proyecto. Todas las imágenes se adjuntarán en el anexo, al final del documento.

- **Inicio de sesión:** Se ha desarrollado una página de inicio de sesión, con dos inputs: email/teléfono y contraseña. Cuando el usuario rellena esta información, se envía una petición a un endpoint en el servicio *coreback*, que responde con la información del usuario en caso de validar la contraseña. En el anexo 9 puede verse el resultado.

- **Dashboard:** Cuando el usuario inicia sesión, llega a un dashboard que contiene información como el precio del mercado diario y mercado intradiario, los últimos movimientos realizados y las últimas alertas. Cuando se carga la página, se hace una petición a coreback, que realiza una petición al *market data reader* para obtener el precio de ambos mercados. Tanto las últimas alertas como los últimos movimientos los obtiene de la base de datos. Se ha adjuntado una imagen en el anexo 10.

- **Página de datos:** Se ha implementado una sección con varias páginas de datos divididas por temáticas: últimos datos, balance, demanda, generación, intercambios, transporte y mercados. El front realiza una petición al *coreback*, que es procesada para extraer las fechas y los distintos filtros aplicados por el usuario, posteriormente, se realiza una petición al *market data reader*. A continuación, se procesa la respuesta y se construye un objeto para que la aplicación muestre la información. En el anexo 11 se puede ver un ejemplo.

- **Pronósticos:** Se muestra una tabla con los precios del día actual y el precio pronosticado por el modelo LSTM. De nuevo, es el front el que envía una petición a *coreback*, que a su vez consulta al servicio de pronóstico. Se ha adjuntado un ejemplo en el anexo 12.

- **Alertas:** Cuando se realiza la carga de página, el *coreback* consulta la base de datos con las últimas alertas creadas por el *watchdog*. Posteriormente, se construye una lista en front con los resultados obtenidos. Ver en el anexo 13.

- **Configuración:** Para que el usuario pueda gestionar su configuración, se hace una petición al *coreback*, que obtiene y/o modifica dicha configuración y devuelve la información para que se pinte en front. En el anexo 14 se muestra una captura de pantalla.

Bibliografía

[1] **Amat, R., & Escobar, J.** (Abril de 2023). **Intermittent demand forecasting with skforecast.** *cienciadedatos.net*. URL: https://cienciadedatos.net/documentos/py48-intermittent-demand-forecasting.html

[2] **Auhl, M.** (6 de agosto de 2021). **What is an ARIMA Model?**. *Towards Data Science.* URL: https://towardsdatascience.com/what-is-an-arima-model-9e200f06f9eb

[3] **Avasthi, T.** (10 de noviembre de 2022). **Guide to Setting Up Apache Kafka Using Docker.** *Baeldung.* URL: https://www.baeldung.com/ops/kafka-docker-setup

[4] **Bellemare, A.** (2020). *Building Event-Driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media.

[5] **Brewer, C., Tidwell, J. & Valencia, A.** (2020). *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly Media

[6] **Cockburn, A.** (4 de enero de 2005). **Hexagonal architecture.** *Alistair Cockburn.* URL: https://alistair.cockburn.us/hexagonal-architecture/

[7] **Dehghani, Z.** (2022). *Data Mesh: Delivering Data-Driven Value at Scale.* O'Reilly Media.

[8] **Endesa.** (2023). *Todo lo que siempre quisiste saber sobre el mercado eléctrico.* URL: https://www.endesa.com/es/la-cara-e/sector-energetico/como-funciona-el-mercado-electrico-en-espana

[9] **Fernández, P.** (18 de diciembre de 2022). **Introducción al patrón flux.** *El rincón del front.* URL: https://elrincondelfront.substack.com/p/intruduccion-al-patron-flux

[10] **Ford, N., Richards, M., Sadalage, P. & Dehghani, Z.** (2021). *Software Architecture: The Hard Parts*. O'Reilly Media.

[11] **Fowler, M.** (12 de diciembre de 2005). **Event Sourcing.** *martinfowler.com*. URL: https://martinfowler.com/eaaDev/EventSourcing.html

[12] **Fowler, M.** (14 de julio de 2011). **CQRS.** *martinfowler.com*. URL: https://martinfowler.com/bliki/CQRS.html

[13] **Fowler, M.** (2018). *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.

[14] **Khononov, V.** (2021). *Learning Domain-Driven Design*. O'Reilly Media

[15] **Kleppmann, M.** (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media.

[16] **Kotlin.** (2023). *Get started with Kotlin*. URL: https://kotlinlang.org/docs/getting-started.html

[17] **Martin, R. C.** (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson.

[18] **Martin, R. C.** (2 de diciembre de 2009). **Getting a SOLID start.** *Clean Coder.* URL: https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

[19] **Martin, R. C.** (2012). **The clean architecture.** *The Clean Coder Blog.* URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[20] **Martins, J.** (10 de octubre de 2022). **¿Qué es la metodología Kanban y cómo funciona?** *Asana.* URL: https://asana.com/es/resources/what-is-kanban

[21] **Muñoz, A.** (20 de junio de 2021). **Así es el mercado intradiario y continuo de electricidad, el hermano pequeño del 'pool'.** *El periodico de la energía*. URL: https://elperiodicodelaenergia.com/asi-es-el-mercado-intradiario-y-continuo-de-electricidad-el-hermano-pequeno-del-pool/

[22] **Olah, C.** (27 de agosto de 2015). **Understanding LSTM Networks.** *Colah 's blog.* URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[23] **Peixeiro, M.** (2022). *Time Series Forecasting in Python.* Manning.

[24] **React.** (2023). *Getting Started.* URL: https://react.dev/learn

[25] **Roussis, D.** (25 de marzo de 2021). **Electricity price forecasting with DNNs (+ EDA).** *kaggle.com*. URL: https://www.kaggle.com/code/dimitriosroussis/electricity-price-forecasting-with-dnns-eda/notebook

[26] **Santra, R.** (8 de junio de 2023). **Introduction to SARIMA Model.** *medium.com.* URL: https://medium.com/@ritusantra/introduction-to-sarima-model-cbb885ceabe8

[27] **Shapira, G., Palino, T., Sivaram, R. & Petty K.** (2021). *Kafka: The Definitive Guide.* O'Reilly Media.

[28] **Stanke, B.** (Diciembre 2018). **Feature-Driven Development: The Pros, Cons, and How It Compares to Scrum.** *Bob Stanke.* URL: https://www.bobstanke.com/blog/feature-driven-development

[29] **Spring Boot.** (2023). *Spring Boot.* URL: https://spring.io/projects/spring-boot

[30] **Vernon, V.** (2013). *Implementing Domain-Driven Design.* Addison-Wesley Professional.

Anexos

Anexo 1. Objetivos de desarrollo sostenible

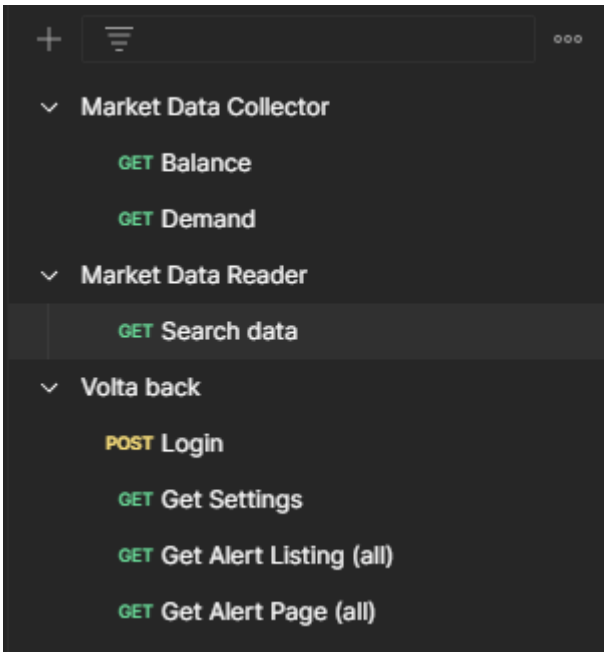
Objetivos de Desarrollo Sostenible	Alto	Medio	Bajo	No procede
1. Fin de la pobreza				x
2. Hambre cero				x
3. Salud y bienestar				x
4. Educación de calidad				x
5. Igualdad de género				x
6. Agua limpia y saneamiento				x
7. Energía asequible y no contaminante			x	
8. Trabajo decente y crecimiento económico	x			
9. Industria, innovación e infraestructuras	x			
10. Reducción de las desigualdades				x
11. Ciudades y comunidades sostenibles				x
12. Producción y consumo responsables				x
13. Acción por el clima			x	
14. Vida submarina				x
15. Vida de ecosistemas terrestres				x
16. Paz, justicia e instituciones sólidas				x
17. Alianzas para lograr objetivos				x

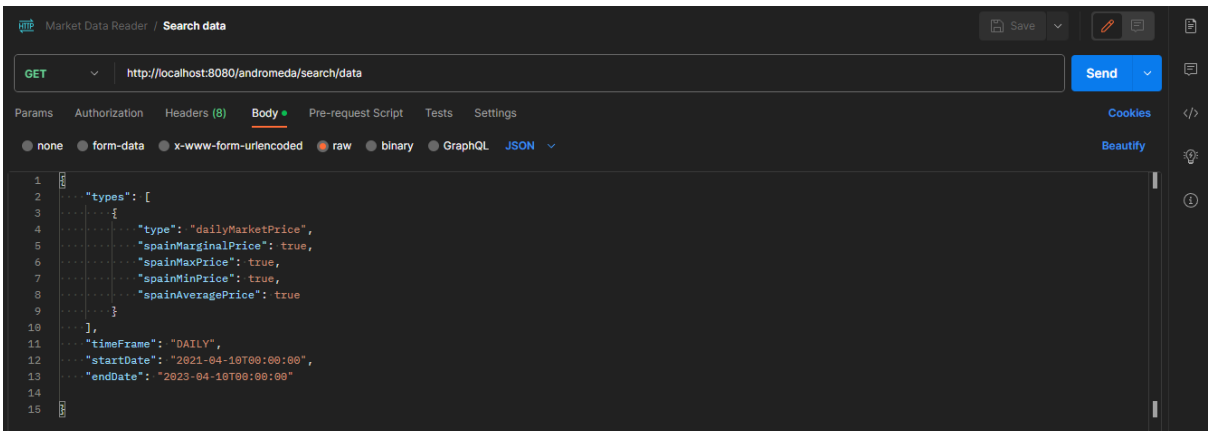
Anexo 2. Librería flux implementada
https://github.com/Kuasr/flux

Anexo 3. Tests unitarios

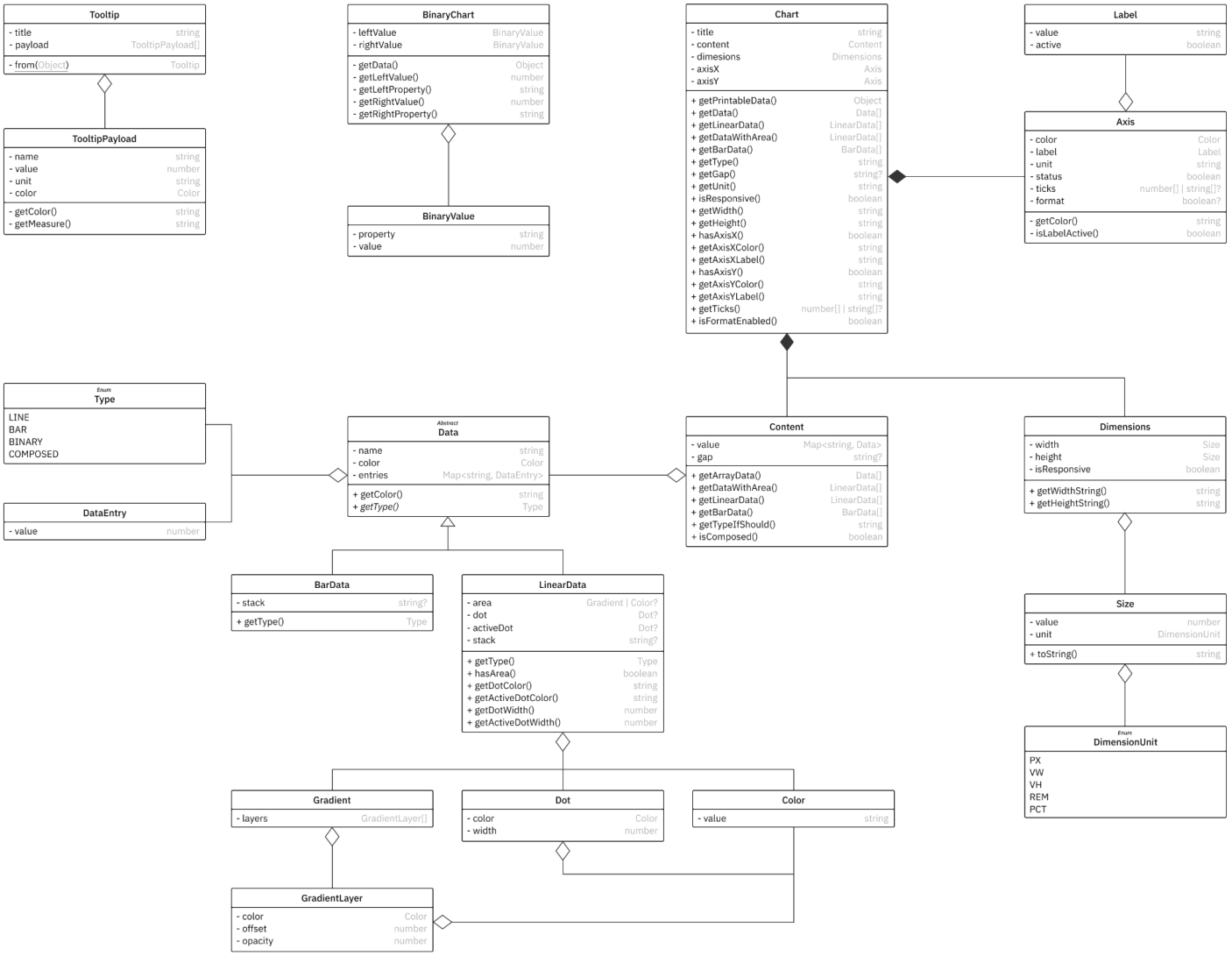
```
RUNS tests/Layout/GeneralLayout/store/states/GeneralLayoutState.test.ts
RUNS tests/Shared/ActionActuator/domain/ActuatorDirector.test.ts
PASS tests/Shared/ActionActuator/domain/ActuatorDirector.test.ts (11.488 s)
PASS tests/Layout/GeneralLayout/domain/GeneralLayout.test.ts (11.884 s)
PASS tests/Layout/TopHeader/store/states/TopHeaderState.test.ts (11.889 s)
PASS tests/Shared/ActionActuator/store/states/ActuatorState.test.ts (11.91 s)
PASS tests/Layout/GeneralLayout/domain/Session.test.ts (12.002 s)
PASS tests/Layout/GeneralLayout/store/states/GeneralLayoutState.test.ts (12.067 s)
PASS tests/Shared/ActionActuator/store/actions/ActionGenerator.test.ts (12.075 s)
PASS tests/Layout/GeneralLayout/store/states/GeneralLayoutStateGenerator.test.ts (12.082 s)
PASS tests/Layout/TopHeader/store/stores/TopHeaderStore.test.ts (12.089 s)
PASS tests/Layout/GeneralLayout/store/actions/ActionGenerator.test.ts (12.124 s)
PASS tests/Layout/GeneralLayout/store/stores/GeneralLayoutStore.test.ts (12.2 s)
PASS tests/Layout/TopHeader/view/components/TopHeader.test.tsx (12.222 s)
PASS tests/Shared/ActionActuator/view/ActionActuator.test.tsx (12.23 s)
PASS tests/Layout/GeneralLayout/view/container/ApplicationLayoutContainer.test.tsx (12.796 s)
```

Anexo 4. API testing con Postman

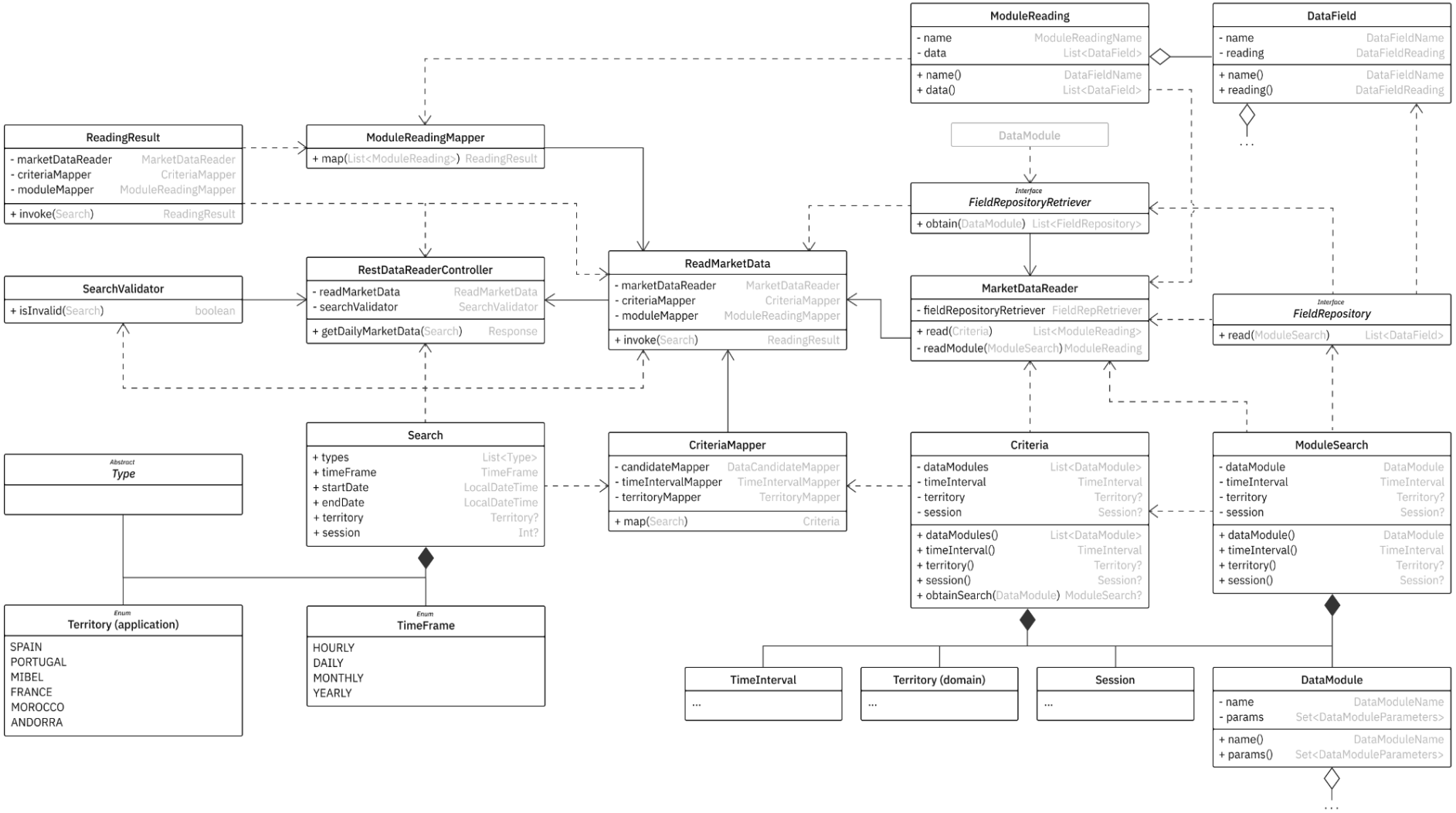




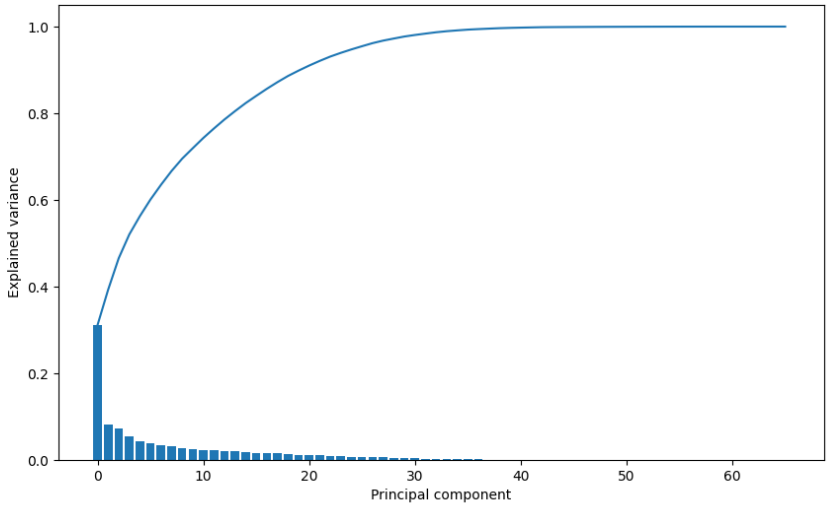
Anexo 5. UML de la funcionalidad de gráficas en el frontend



Anexo 6. Diagrama UML simplificado del servicio de lectura


ModuleReading
- name ModuleReadingName
- data List<DataField>
+ name() DataFieldName
+ data() List<DataField>
DataField
- name DataFieldName
- reading DataFieldReading
+ name() DataFieldName
+ reading() DataFieldReading
ReadingResult
- marketDataReader MarketDataReader
- criteriaMapper CriteriaMapper
- moduleMapper ModuleReadingMapper
+ invoke(Search) ReadingResult
ModuleReadingMapper
+ map(List<ModuleReading>) ReadingResult
DataModule
Interface
FieldRepositoryRetriever
+ obtain(DataModule) List<FieldRepository>
SearchValidator
+ isInvalid(Search) boolean
RestDataReaderController
- readMarketData ReadMarketData
- searchValidator SearchValidator
+ getDailyMarketData(Search) Response
ReadMarketData
- marketDataReader MarketDataReader
- criteriaMapper CriteriaMapper
- moduleMapper ModuleReadingMapper
+ invoke(Search) ReadingResult
MarketDataReader
- fieldRepositoryRetriever FieldRepRetriever
+ read(Criteria) List<ModuleReading>
- readModule(ModuleSearch) ModuleReading
Interface
FieldRepository
+ read(ModuleSearch) List<DataField>
Abstract
Type
Search
+ types List<Type>
+ timeFrame TimeFrame
+ startDate LocalDateTime
+ endDate LocalDateTime
+ territory Territory?
+ session Int?
CriteriaMapper
- candidateMapper DataCandidateMapper
- timeIntervalMapper TimeIntervalMapper
- territoryMapper TerritoryMapper
+ map(Search) Criteria
Criteria
- dataModules List<DataModule>
- timeInterval TimeInterval
- territory Territory?
- session Session?
+ dataModules() List<DataModule>
+ timeInterval() TimeInterval
+ territory() Territory?
+ session() Session?
+ obtainSearch(DataModule) ModuleSearch?
ModuleSearch
- dataModule DataModule
- timeInterval TimeInterval
- territory Territory?
- session Session?
+ dataModule() DataModule
+ timeInterval() TimeInterval
+ territory() Territory?
+ session() Session?
Enum
Territory (application)
SPAIN
PORTUGAL
MIBEL
FRANCE
MOROCCO
ANDORRA
Enum
TimeFrame
HOURLY
DAILY
MONTHLY
YEARLY
TimeInterval
...
Territory (domain)
...
Session
...
DataModule
- name DataModuleName
- params Set<DataModuleParameters>
+ name() DataModuleName
+ params() Set<DataModuleParameters>

Anexo 7. PCA



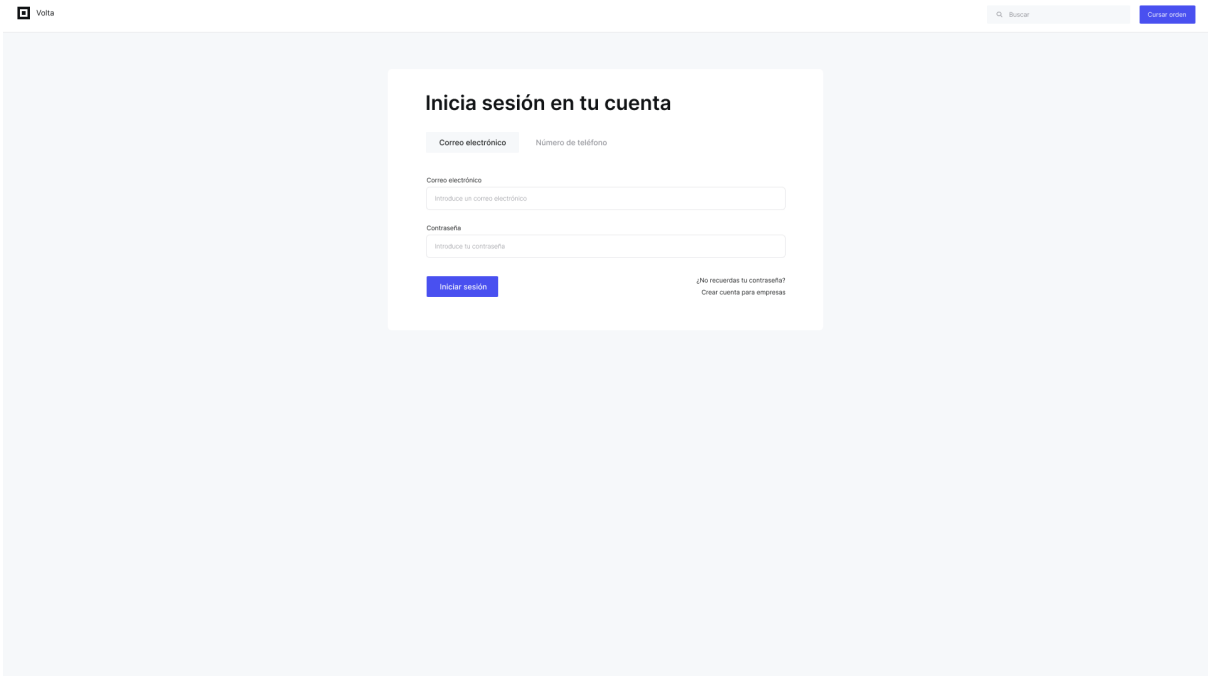
Anexo 8. Implementación de *Search*

```
class Search(
    @JsonProperty("types")      val types:      List<Type>,
    @JsonProperty("timeFrame")  val timeFrame:  TimeFrame,
    @JsonProperty("startDate")  val startDate:  LocalDateTime,
    @JsonProperty("endDate")    val endDate:    LocalDateTime,
    @JsonProperty("territory")  val territory:  Territory?,
    @JsonProperty("session")    val session:    Int?
)
```

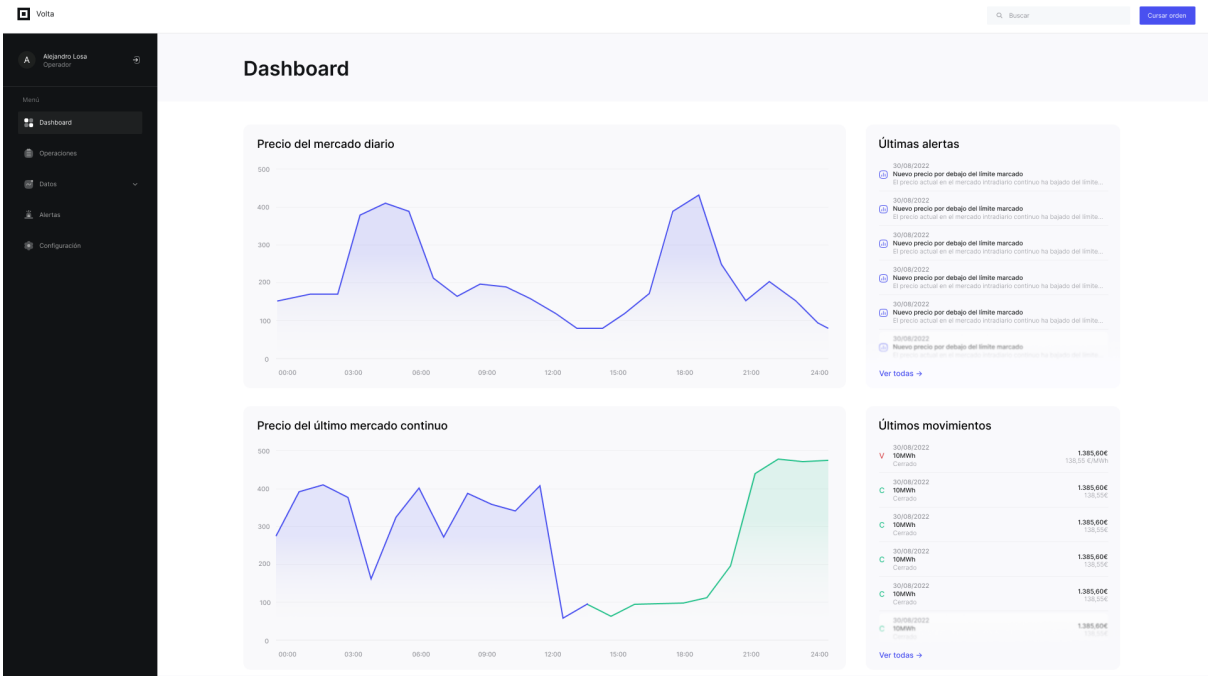
```
@JsonTypeInfo(property = "type")
@JsonSubTypes(
    JsonSubTypes.Type(DailyMarketPrice::class,   "dailyMarketPrice"),
    JsonSubTypes.Type(BusbarDemand::class,       "busbarDemand"),
    JsonSubTypes.Type(GenerationBalance::class, "generationBalance"),
    ... // Más de 50 tipos
)
abstract class Type
```

```
@JsonTypeName("dailyMarketPrice")
class DailyMarketPrice(
    val spainMarginalPrice:    Boolean?,
    val spainMaxPrice:         Boolean?,
    val spainMinPrice:         Boolean?,
    val portugalMarginalPrice: Boolean?,
    ...
    val negotiatedEnergy:      Energy?
): Type()
```

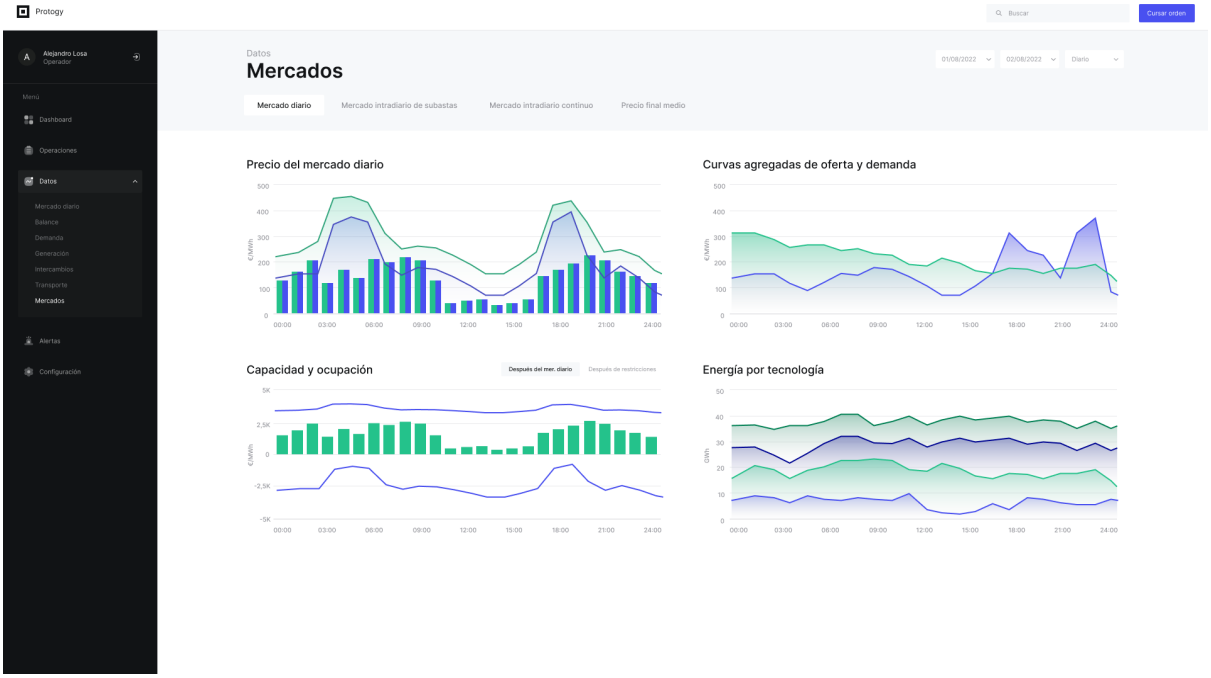
Anexo 9. Inicio de sesión



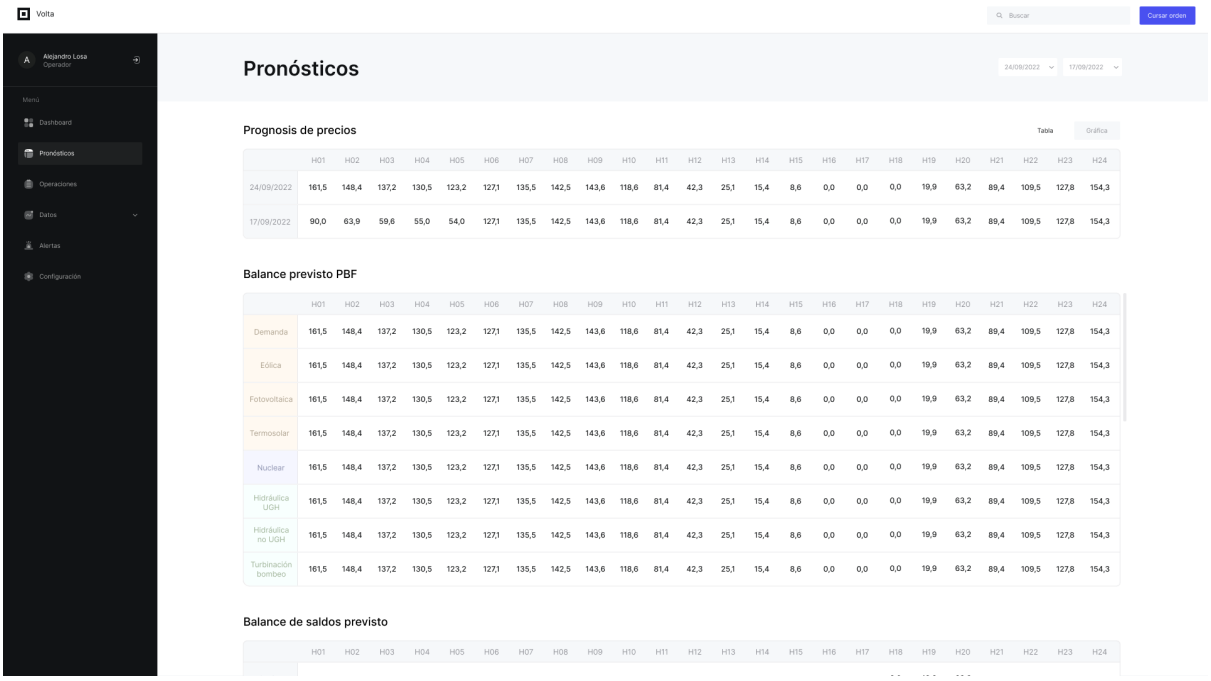
Anexo 10. Dashboard



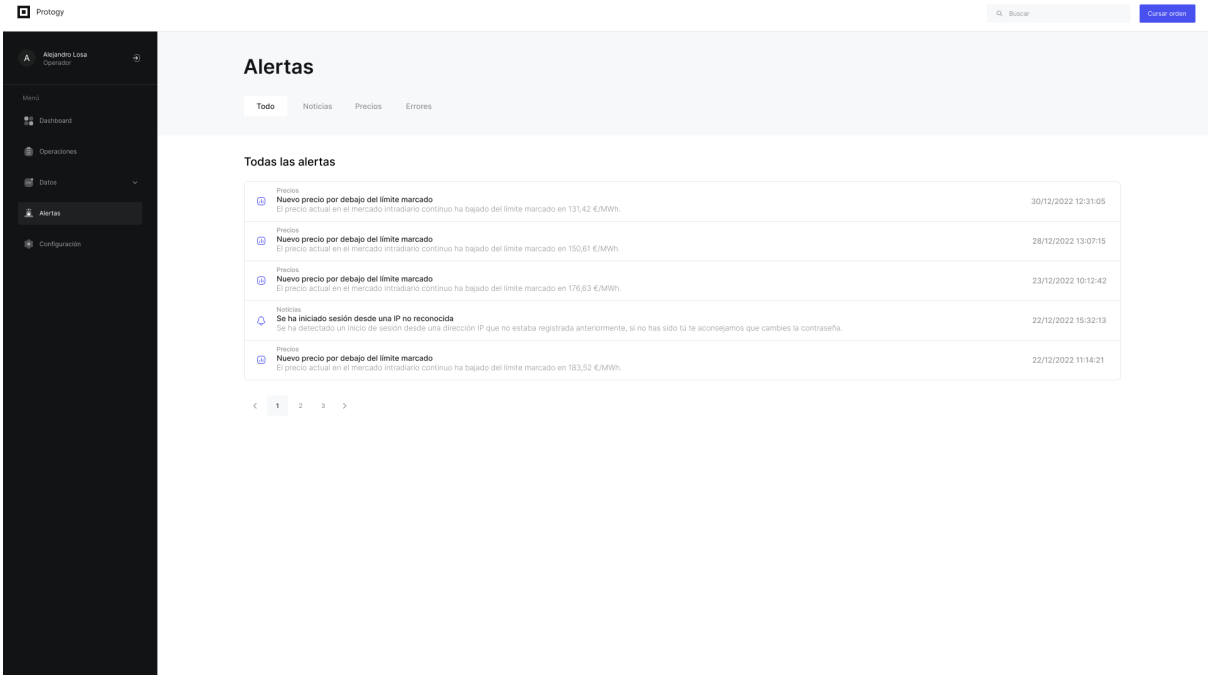
Anexo 11. Página de datos



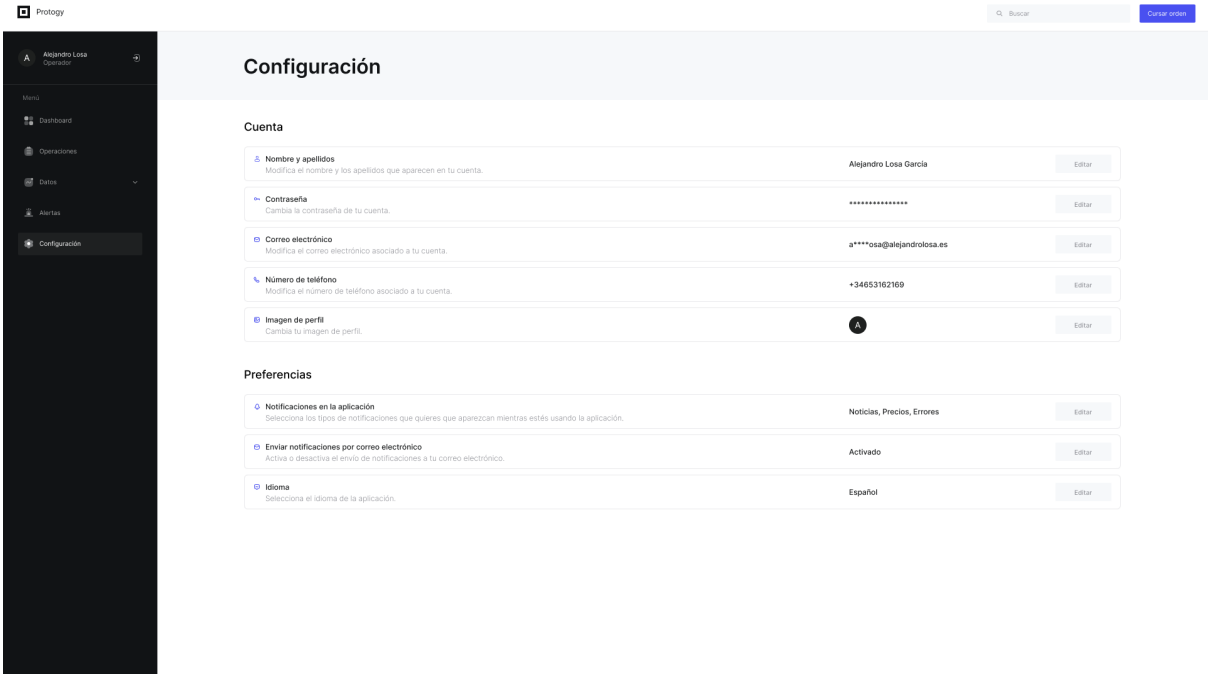
Anexo 12. Pronósticos



Anexo 13. Alertas



Anexo 14. Configuración



Anexo 15. Petición de búsqueda y controlador del market data reader

```
// URL de la petición
http://marketdatareader1.devel.kuasr.com/andromeda/search/data

// Cuerpo de la petición
{
    "types": [
        {
            "type": "dailyMarketPrice",
            "spainMarginalPrice": true,
            "spainMaxPrice": true,
        }
    ],
    "timeFrame": "DAILY",
    "startDate": "2021-06-01T00:00:00",
    "endDate": "2022-06-01T00:00:00"
}

// Controlador
class RestDataReaderController(private val readMarketData:ReadMarketData,
                               private val validator:SearchValidator) {
    @GetMapping(Routes.DATA)
    fun getData(@RequestBody search: Search): Response<ReadingResult> {
        if (validator.isInvalid(search))
            return Response.error(search);

        val readingResult = readMarketData(search);
        return Response.success(readingResult);
    }
}
```

Anexo 16. Adaptador de la API de OMIE

```
// collect/dailymarket/energyprice/infrastructure/OmiePriceCollector
class OmiePriceCollector (private val fileRetriever: FileRetriever,
                          private val dayMapper: DayMapper
): PriceCollector {
    ...
    override fun obtainDayPrices(day: Day): MibelDailyPrices? {
        val response = fileRetriever.makeApiCall(getOmieApiUrl(day));
        if (isEmptyResponse(response)) throw CollectorException(day);
        return map(response);
    }
    ...
}


// collect/shared/FileRetriever
class FileRetriever {

    fun makeApiCall(url: String): List<String> =
        getDataFromFile(url).split("\r\n");

    private fun getDataFromFile(url: String): String {
        val client = HttpClient.newBuilder().build();
```

```
        val request = HttpClient.newBuilder()
                                .uri(URI.create(url))
                                .build();

        return client.send(request, HttpResponse.bodyHandlers()).body();
    }
}
```

Anexo 17. Caso de uso para leer datos del Market Data Reader

```
class ReadMarketData(private val criteriaMapper:CriteriaMapper,
                     private val moduleReadingMapper:ModuleReadingMapper,
                     fieldRepositoryRetriever:FieldRepositoryRetriever) {

    private val dataReader: MarketDataReader =
        MarketDataReader(fieldRepositoryRetriever);

    operator fun invoke(@search: Search): ReadingResult {
        val criteria = criteriaMapper.map(search);
        val retrievedData = marketDataReader.read(search);

        return moduleReadingMapper.map(retrievedData);
    }
}
```

Anexo 18. Cliente de REE

```
class ReeBalanceClient(private val facade: Facade,
                       private val balanceMapper: BalanceMapper
): BalanceClient {
    ...
    override fun obtain(dateInterval: DateInterval,
                        location: Location?): List<BalanceData> {

        val data = facade.collectReeData(
            ReeCategory.BALANCE,
            ReeTopic.ELECTRIC_BALANCE
            dateInterval.startDate(),
            dateInterval.endDate(),
            mapPeriod(dateInterval.period()),
            Language.ENGLISH,
            ReeGeoScope.ELECTRIC_SYSTEM
            mapReeLocation(location)
        );

        if (data.error != null) throw ClientException(data.error);
        return data.sections.map(balanceMapper::invoke);
    }
    ...
}
```