TecnoCampus
Escola Superior Politècnica

Centre adscrit a la
upf. Universitat Pompeu Fabra Barcelona

Data Scraping y Explotación con Big Data

Trabajo de fin de grado

Doble Titulación en Grado en Informática de Gestión y Sistemas de Información y Grado en Diseño y Producción de Videojuegos

Aitor Garcia Diez

Tutor: Eugeni Fernández González

Universidad: TecnoCampus Mataró-Maresme

Dedicatoria

A mi hermana, para que tenga un buen trabajo al que referenciar en cinco años.

Agradecimientos

A mi familia, por siempre confiar en mí y apoyarme en todo.

A mis padres, por absolutamente todo.

Resumen

Este proyecto intenta tanto diseñar y aplicar una estructura centrada en el dominio con el mínimo acoplamiento posible como implementar un servicio de scraping en dicha estructura junto a una aplicación multiplataforma para hacer uso del servicio. Todo el proyecto está realizado para Casa Ametller.

Abstract

This project tries both to design and apply a domain-centric structure with the minimum possible coupling and to implement a scraping service in this structure together with a cross-platform application to make use of the service. The whole project is done for Casa Ametller.

Resum

Aquest projecte intenta tant dissenyar i aplicar una estructura centrada en el domini amb el mínim acoblament possible com implementar un servei de scraping en aquesta estructura al costat d'una aplicació multiplataforma per a fer ús del servei. Tot el projecte està realitzat per a Casa Ametller.

Tabla de contenidos

Índice de figuras

Glosario

TFG – Trabajo de fin de grado

MVC – Model View Controller / Modelo Vista Controlador

MVP – Minimum Viable Product

MA – Monolith Architecture

SOA – Service Oriented Architecture

DDD – Domain Driven Development

BI – Business Intelligence

SPA – Single Page Application

DSL – Domain Specific Language

DI – Dependency Injection

DTO – Data transfer object

DAO – Data Access object

1 Marco teórico y análisis de referentes

Los softwares a gran escala siempre han supuesto un problema a la hora de mantener, actualizar y escalar. Uno de los primeros libros de diseño de patrones es "*Design Patterns: Elements of Reusable Object-Oriented Software*" [1]. Aunque en este libro se definen muchos de los patrones más usados hoy en día, ya se usaban y eran muy conocidos algunos de los presentes, como el patrón de "*Model View Controller (MVC)*" que se usa para escribir frontends y aplicaciones con una capa de presentación[2].

1.1 Monolith Architecture

La Monolith Architecture, MA, es la forma tradicional de desarrollar software. El principio básico, y prácticamente el único, es que todo está en una aplicación.[3] Aunque al principio sea más fácil desarrollar la aplicación, tanto el escalado como el mantenimiento se dificultan contra más crece la complejidad y el tiempo.[4]

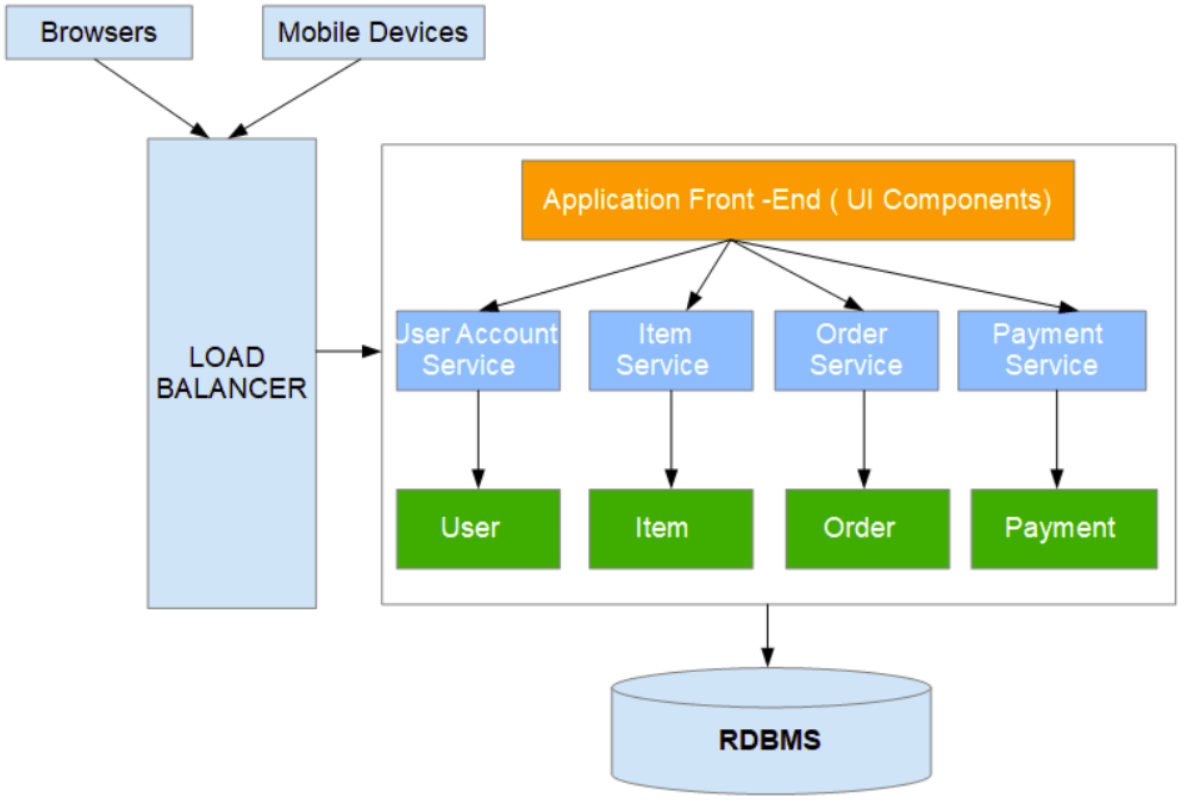


Fig. 1.1.1 Monolith Architecture. Fuente: oracleappshelp.com

1.2 Service-Oriented Computing

Uno de las primeras arquitecturas es "*Service-Oriented Computing (SOC) / Service-Oriented Architecture (SOA)*". Empieza como un paradigma emergente para negocios electrónicos y para computación distribuida que tiene como origen la programación orientada a objetos[5]. El concepto de SOA es que haya un servicio central que ofrezca todas las funcionalidades a otros componentes a través de mensajes. Para bajar el acoplamiento se desacoplan las interfaces de la implementación.

1.2.1 Beneficios

El primer beneficio de SOA es el escalado ya que se pueden crear nuevas instancias para hacer load balancing y dividir la carga de los sistemas.
El segundo es que es independiente de la plataforma ya que los componentes se comunican a través de mensajes, que podrían ser tcp, udp, etc.
El tercer gran beneficio es la independencia entre servicios, cuando se diseñan servicios siguiendo los principios de la arquitectura orientada a servicios suelen tener un bajo acoplamiento entre ellos.

1.2.2 Desventajas

Posiblemente debido a ser una de las primeras arquitecturas, SOA tiene unas desventajas muy claras. Que todos los servicios se comuniquen a través de mensajes suele significar que necesitan un gran ancho de banda. La mayor desventaja es sin duda su gran coste de implementación y ejecución. Hay un gran coste de recursos tecnológicos, monetarios y de desarrolladores.

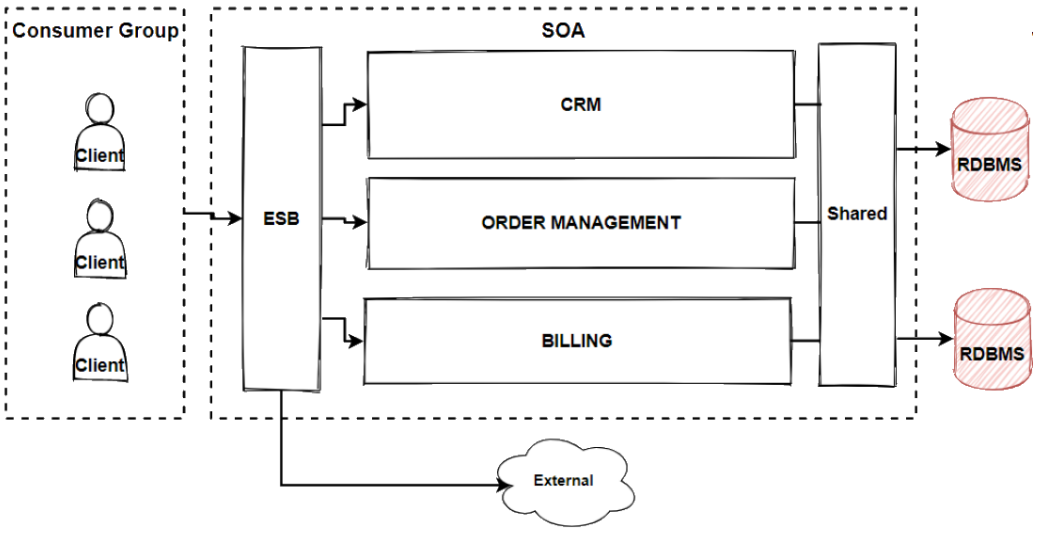


Fig. 1.2.1 Service Oriented Architecture. Fuente: medium.com

1.3 Arquitectura basada en microservicios

La arquitectura basa en microservicios es la evolución de las ideas de SOA. En este caso los microservicios están totalmente aislados de la solución principal y entre ellos. Uno de los puntos principales es que al ser un servicio tan básico se suelen usar contenedores ligeros como Docker o Podman que facilitan su deploy y escalado [6].

Según Rober "*Uncle Bob*" Cecil Martin, los microservicios no son una arquitectura sino un modo de hacer deploy. Si estás ligando tu arquitectura a que sea microservicios no estás implementando una arquitectura de software, sino simplemente una arquitectura de devops.

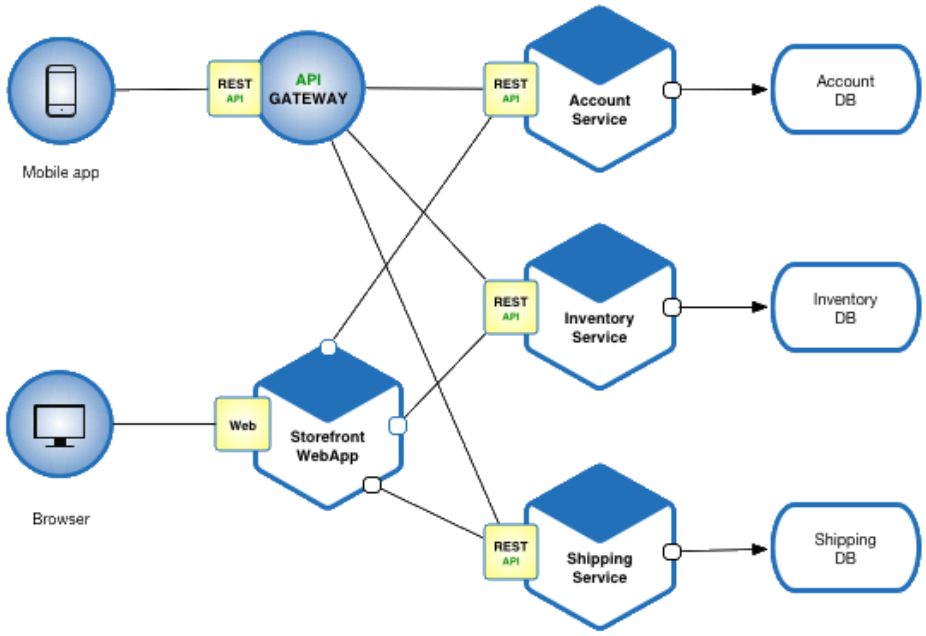


Fig. 1.3.1 Microservices Architecture. Fuente: microservices.io

1.4 Domain Driven Design (DDD)

DDD es una metodología que se centra en tener el negocio como el centro de todo[7]. Aunque muchas arquitecturas, metodologías y patrones se centran en cómo hacer un código mejor, más limpio o más rápido, DDD se centra en cómo tiene que funcionar el software y no en cómo ha de estar implementado por detrás[8]. Hay muchas arquitecturas basadas en DDD como la arquitectura Hexagonal, también conocida como arquitectura de Puertos y Adaptadores, la arquitectura llamada "*Data Context Integration*" (DCI) o la arquitectura "*Boundary Control Entity*" (BCE) que, aunque se autodefine como "*Use-Case Driven*" tiene muchas similitudes con DDD [9].

Lo que todas tienen en común es la independencia. Independencia de frameworks, independencia de la UI, independencia de la base de datos, independiente de cualquier agente externo a al negocio [9].

1.4.1 Clean Architecture

Uncle Bob hace un acercamiento a DDD que intenta integrar todo lo comentado en el punto anterior en una sola arquitectura y la denomina "*Clean Architecture*" [9], se puede observar en la Fig. 1.4.1.

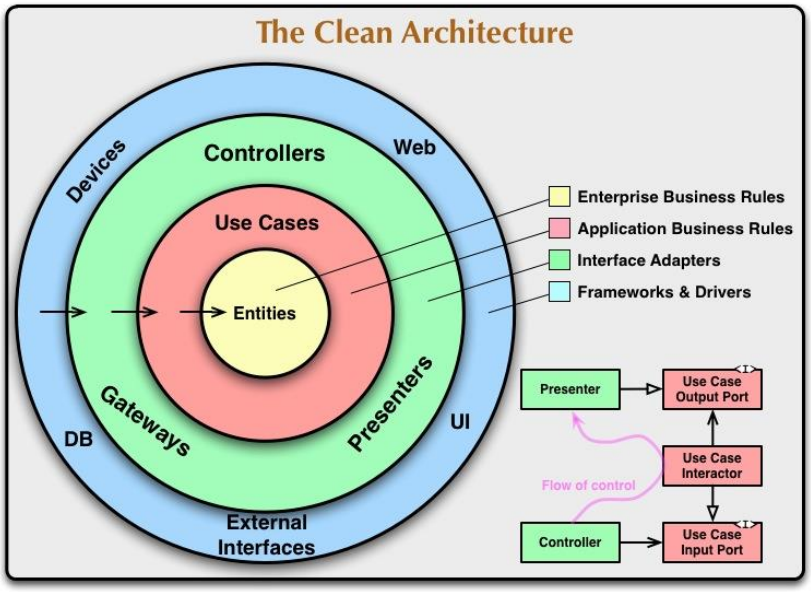


Fig. 1.4.1 Clean Architecture. Fuente: blog.cleancoder.com

1.4.1.1 Dependency Rule

La regla básica de la Clean Architecture es la regla de dependencia. Una capa solo puede depender de su misma capa o la inferior, nunca una superior y a poder ser no depender de una capa que esté dos niveles por debajo.

Siguiendo la Fig. 1.4.1, la capa de dominio o negocio "*Entities*" no puede depender de nada. La capa de aplicación "*Use Cases*" solo puede depender de la capa de dominio. En la capa de Adaptadores, "*Controllers*" podría depender de "*Gateways*" en la misma capa, de la capa de aplicación y, aunque se recomienda no hacerlo, podría depender de la capa de dominio.

1.4.1.2 Capa de dominio o negocio

En la capa de dominio van todo lo relacionado con el negocio:

- excepciones
- value-objects

- entidades
- reglas de negocio

Esta capa cambia únicamente cuando cambia el negocio y no puede tener dependencias a ninguna otra capa.

1.4.1.3 Capa de aplicación

Al contrario de lo que su nombre pueda indicar, en esta capa no se hace ninguna implementación, sino que se definen todas las interfaces necesarias para los casos de uso.

Clean Architecture se basa en casos de uso y es en esta capa donde se definen los casos de uso. Al depender solo de la capa de dominio, los casos de uso han de trabajar con las interfaces definidas en esta capa, lo que hace que no sea dependiente de ninguna implementación específica y asegura que el funcionamiento del negocio es el deseado.

La capa de aplicación no afecta a la capa de dominio; los casos de uso pueden afectar las interfaces de la capa de aplicación según su necesidad. La capa de aplicación tampoco es afectada por absolutamente ninguna externalidad fuera de las capas de aplicación y dominio, aunque si puede afectar a capas superiores cuando se hagan cambios en esta capa.

1.4.1.4 Capa de adaptadores

La capa de adaptadores es la capa donde se implementan las interfaces definidas en la capa de aplicación y donde se transforman los datos a lo más conveniente para los casos de uso y entidades. Normalmente se definen "*Data Transfer Objects*" o *DTOs* paralelos a las entidades con únicamente los datos necesarios para hacer las operaciones y sin ninguna regla de negocio.

Es en esta capa donde se implementa la persistencia. Es en esta, y exclusivamente en esta, capa donde se pueden especificar las implementaciones de base de datos. De ese modo se desacopla la lógica de negocio de las bases de datos ya que su uso no es importante para el negocio.

En esta capa es donde van las API. Es común que una API dependa de una capa de persistencia. Las dependencias a la misma capa están permitidas.

1.5 Soluciones comerciales

Actualmente ya hay soluciones comerciales de monitorización, recolección y explotación de datos.

1.5.1 Minderest

En Minderest se definen como “Expertos en monitorización de precios, promociones y stock”.

En Minderest tienen dos herramientas de extracción de datos.

1.5.1.1 Pricing Tool

Pricing Tool es una suite de herramientas en la nube para retailers y marcas. El SaaS de pricing permite a retailers reaccionar rápidamente a las estrategias de sus competidores, y a su vez ayudar a marcas anticipándose a las guerras de precio de su canal de distribución. [10].

Pricing Tool permite mantener un histórico de precios de la competencia, como se puede observar en Fig. 1.5.1, así como tener un sistema de “Dynamic Pricing” que pone los precios gracias a una Inteligencia Artificial y a reglas de negocio.

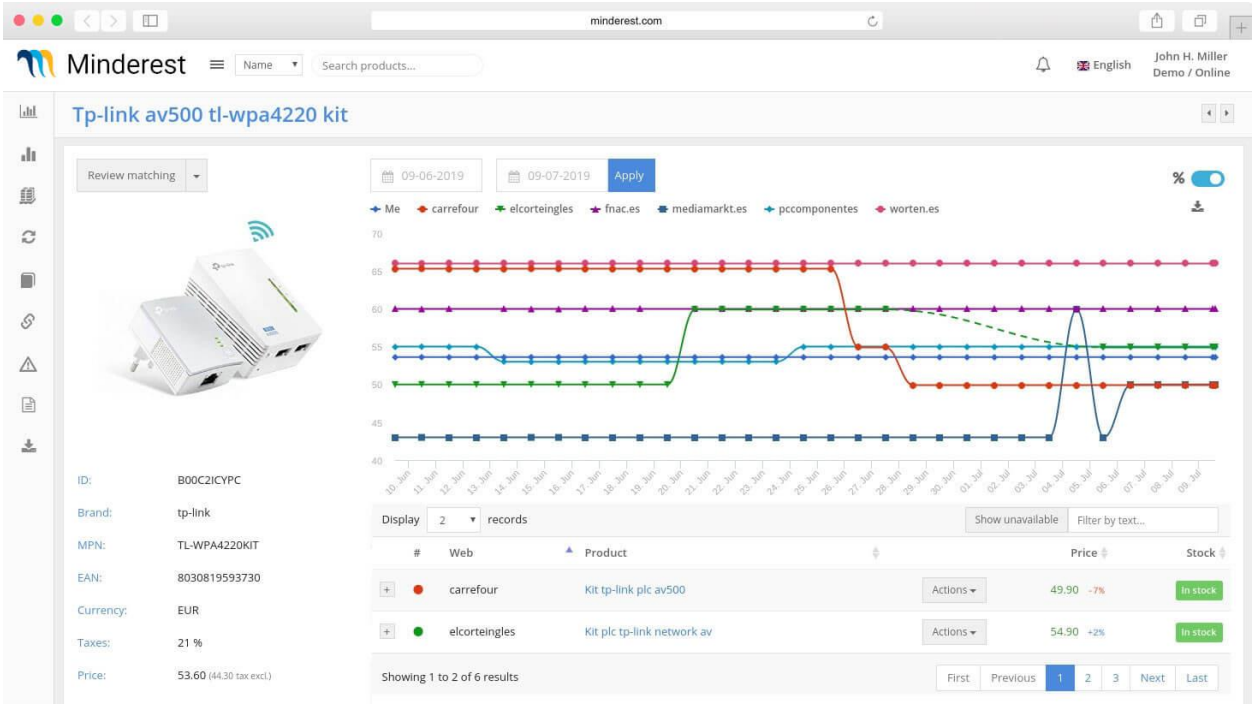


Fig. 1.5.1 Pricing Tool de Minderest. Fuente: minderest.com

1.5.1.2 Product Data API

Product Data API es una herramienta sencilla, solo se indican los portales ecommerce que se quieran monitorizar y Minderest se encarga del resto. Se paga un crédito por producto que se quiera recoger información cada vez, no se paga por el uso de la API [11].

Se pueden mezclar los datos con la herramienta de Business Intelligence, como se observa en la Fig. 1.5.2.

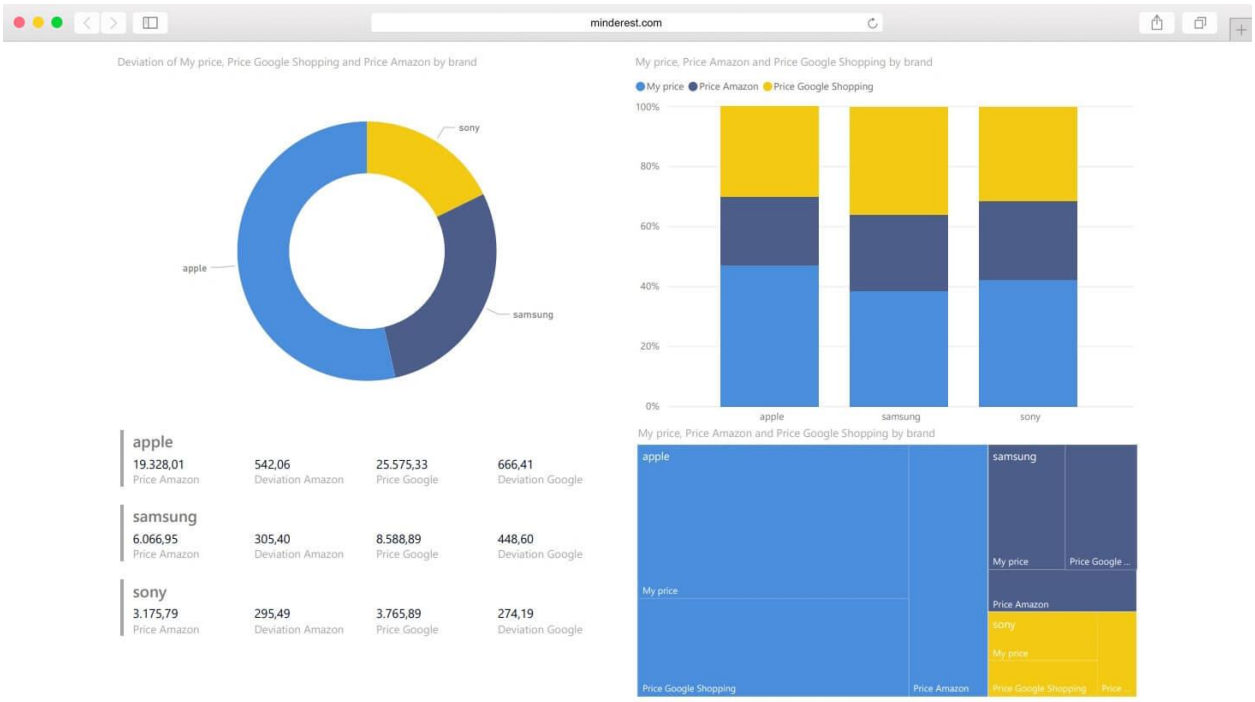


Fig. 1.5.2 Product Data API, Herramienta de BI. Fuente: minderest.com

1.5.2 FlipFlow

Flipflow, un partner tecnológico en monitorización de catálogo, precios, competidores, marketplaces y cualquier dato de productos en internet con sistemas de alertas de cambios para dar los datos más relevantes en todo momento[12]. FlipFlow tiene monitorización de Google Shopping, de marketplaces como Amazon.

También tiene una herramienta de BI, que se puede observar en la Fig. 1.5.3, donde explotar todos los datos recogidos. La aplicación tiene un set de herramientas completo para monitorizar, analizar y establecer alertas sobre precios, disponibilidad, competidores y contenidos. Ofrecen una opción para exportar los datos de su plataforma, ya sea en formato csv, a través de su API o incluso bajo demanda[13].

Fig. 1.5.3 Herramienta BI de FlipFlow. Fuente: flipflow.io

2 Objetivos y abastecimiento

Los objetivos de este proyecto vienen definidos por las necesidades del cliente interno ya que son el usuario final del producto.

Los objetivos son:

1. Recoger y almacenar datos internos de ventas
2. Recoger y almacenar datos externos mediante técnicas de scraping
3. Proporcionar una interfaz que permita hacer diferentes comparaciones mediante los datos almacenados.

3 Metodología

El trabajo de campo de este TFG se ha dividido en cinco grandes fases.

3.1 Fases

3.1.1 Diseño

La primera de diseño. Se han diseñado los patrones y arquitecturas del Backend, el diseño de la base de datos y del servicio de scraping. Se ha hecho un diseño con el mínimo acoplamiento para depender del mínimo número de tecnologías externas.

3.1.2 Almacenamiento de datos Internos

En la segunda fase se ha implementado el primer servicio de scraping, este servicio hace scraping a una página web.

3.1.3 Implementación del Backend

La implementación del Backend es una fase que no necesita estar acabada hasta el final. Solo se necesita implementar el método de insertar productos que es el que usa el servicio de scraping.

3.1.4 Almacenamiento de datos externos

La siguiente fase es implementar el servicio de scraping para los objetivos necesarios.

3.1.5 App

La última fase es hacer la App para los empleados internos de Ametller.

3.2 Agile

A pesar de que las fases se pueden hacer en waterfall siguiendo el orden en el que está escrito, se usa la metodología Agile que se implementa en la propia empresa. Esto permite que después de hacer el diseño global y el almacenamiento de datos interno, al menos una parte básica, se desarrollen las siguientes partes según sea necesario.

3.3 Gantt

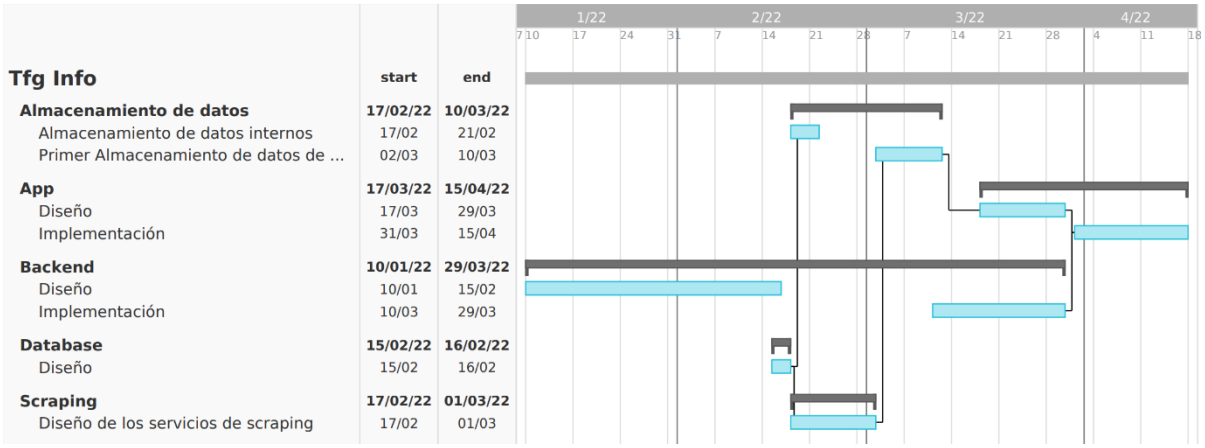


Fig. 3.3.1 Gantt del proyecto. Fuente: elaboración propia

Aunque la implementación básica se podría completar para mediados de abril, eso es teniendo en cuenta que se trabajan 4 horas al día durante cinco días a la semana. Son aproximadamente 332 horas de implementación básica. La mayor parte es el diseño ya que un buen diseño hace más fácil el cambio, escalado y mantenimiento.

4 Diseño

4.1 Requisitos

4.1.1 Tecnológicos

1. Utilizar Scala para el Backend.
2. Usar Python para el scraping.
3. Usar PostgreSQL como base de datos.
4. Usar BigQuery como Data warehouse.
5. Usar Angular para el Frontend.

4.1.2 Funcionales

1. Recoger y almacenar datos internos.
2. Recoger y almacenar datos externos.
3. Relacionar productos internos con externos.
4. Mantener un control de usuarios
5. Tener un frontend multiplataforma.
    a. Comparar históricos de precios de productos internos.
    b. Comparar históricos de precios de productos externos.
    c. Comparar históricos de precios de productos internos y externos.
    d. Comparar las ventas de productos interno con la diferencia de precio de la competencia.
    6. Relacionar productos internos con externos.
    a. Tener inicio de sesión

4.1.3 No funcionales

1. Diseñar una arquitectura escalable.
2. Tener el mínimo acoplamiento posible.
3. Automatizar la extracción de almacenamientos internos.

4.2 Arquitectura actual en Casa Ametller

Actualmente se usa una arquitectura basa en microservicios, que, aunque al principio estaba bien, actualmente se constan con más de cincuenta librerías internas y más de una docena de microservicios. Esto genera una problemática cada vez que hay que cambiar cosas internas del negocio o de alguna de

las librerías más básicas se tarden literalmente horas. Por este motivo se ha estudiado en profundidad arquitecturas que tengan el negocio como base y que sean lo menos dependientes posibles.

4.2.1 Elección de patrones y arquitectura

A pesar de haber dicho que en Ametller hay una arquitectura basa en microservicios, microservicios no es una arquitectura de código sino una forma de hacer deploy y comunicarse entre servicios. El único requisito necesario para tener microservicios es dividir el código en partes que hagan trabajos muy concretos.

Cuando se habla de arquitecturas para negocios hoy en día siempre aparece la misma palabra, DDD. Domain Driven Design, una arquitectura que centra todo en tener el negocio en el centro, aunque hay gente que la asocia con Data Driven Design, una arquitectura que se basa en que los datos son el centro de la solución y basan todo alrededor de las bases de datos, ya que estas suelen cambiar menos que el código, si es que cambian siquiera. Por toda la explicación de la arquitectura limpia y cómo funciona, que se puede encontrar en la sección Clean Architecture, se ha decidido utilizar Clean Architecture.

4.3 Diseño del producto.

4.3.1 Punto de partida

El primer paso fue visualizar el concepto del producto final y para esto se contó con la ayuda del jefe del departamento de IT, que se reunió con otras personas claves para acabar de definir el concepto general aplicado al negocio y a las necesidades de los clientes internos.

4.3.2 Diseño de las entidades de scraping necesarias

El 25 de febrero del 2022 se hizo la siguiente reunión con el jefe del equipo de integraciones para definir las entidades necesarias en base al producto final y al negocio. De esa reunión se pudo sacar que entidades son necesarias de almacenar en base de datos, en qué tipo de base de datos, y que datos son necesarios recolectar del scraping a ser posible.

La base de datos es PostgreSQL ya que toda la arquitectura de la empresa va alrededor de PostgreSQL. Se debe hacer un volcado cada cierto tiempo en BigQuery para no perder los datos y seguir manteniendo la performance.

4.3.3 Diseño del roadmap

El 4 de marzo se hizo una reunión en la cual se definió las tarjetas necesarias y los Story Points asignados a cada una de ellas para ir computándolas en cada Sprint.

4.3.4 Reunión con el cliente final interno

A mediados de mayo se hizo una reunión con el cliente final para enseñar el trabajo hecho y recibir feedback directo de las características implementadas y futuras, así como del diseño y de la experiencia de usuario.

4.4 Diagramas

Para ayudar a conceptualizar el proyecto, se han hecho diversos diagramas de diseño.

4.4.1 Diagrama de roles

El diagrama de roles se puede observar en la Fig. 4.4.1, que sirve para hacer una idea de los recursos necesarios tanto para desarrollar como mantener y usar toda la solución al completo.

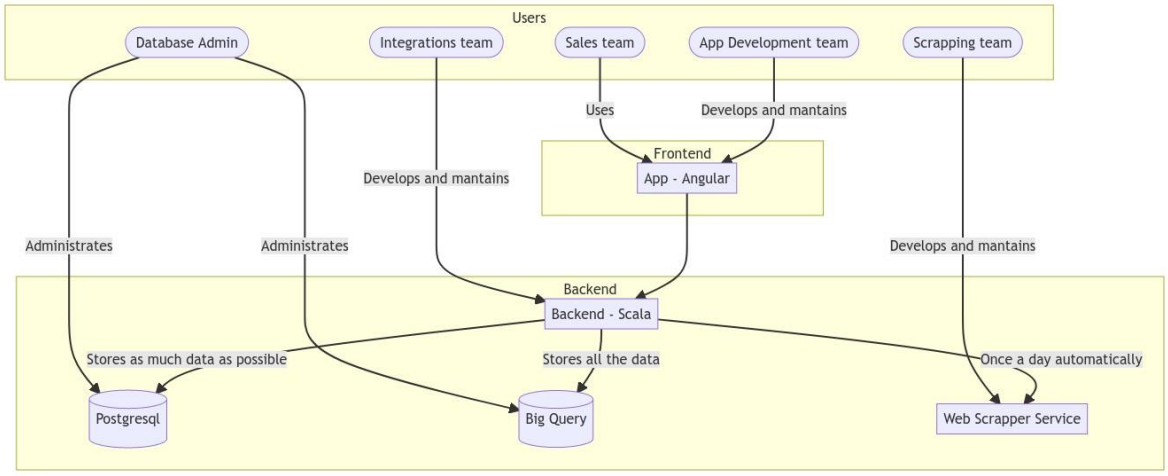


Fig. 4.4.1 Diagrama de roles. Fuente: Elaboración propia

4.4.2 Diagrama de tecnologías

Se puede observar en la Fig. 4.4.2 las tecnologías que se usan en este proyecto y como se relacionan entre ellas.

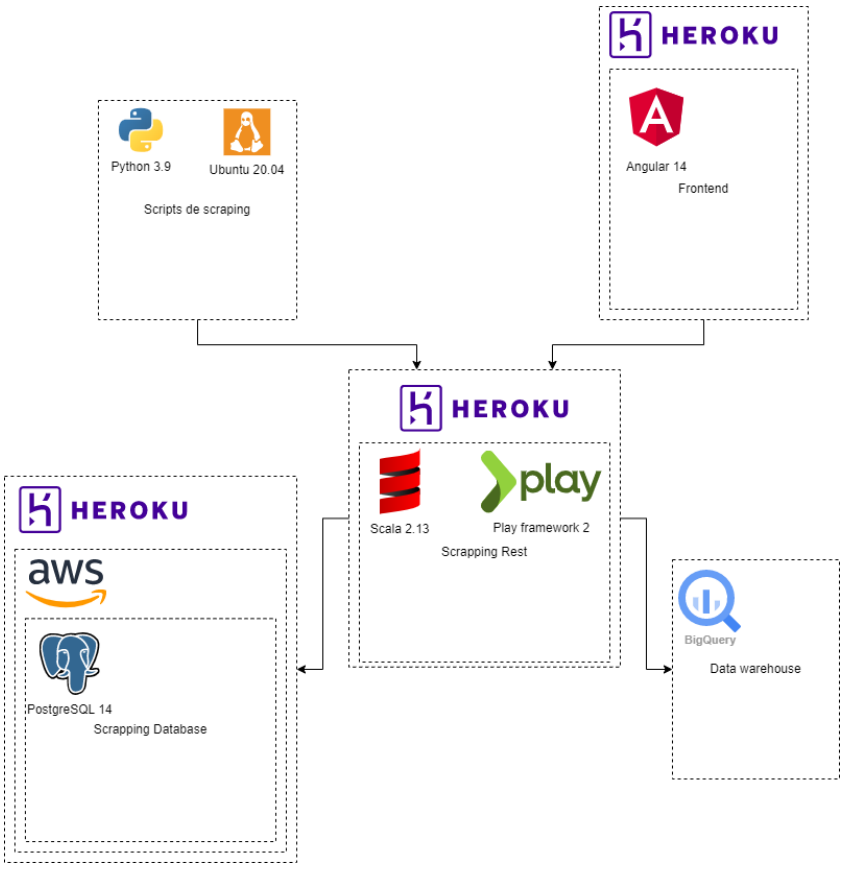


Fig. 4.4.2 Diagrama de tecnologías usadas. Fuente: Elaboración propia

4.4.3 Diagrama de la arquitectura

En la Fig. 4.4.3 se muestra un diagrama de la arquitectura, los elementos pintados de color verde claro son los que se han diseñado e implementado para este proyecto.

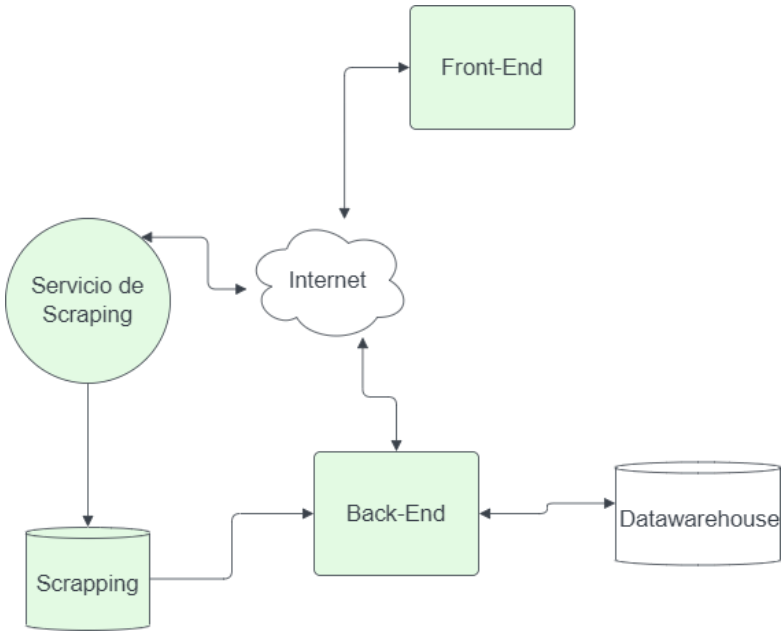


Fig. 4.4.3 Diagrama de la arquitectura. Fuente: Elaboración propia

4.4.4 Diagrama de los procesos que intervienen en el scraping

Para que sea más sencillo visualizar todo el proceso, se deja en la Fig. 4.4.4 un diagrama con el proceso de scraping.

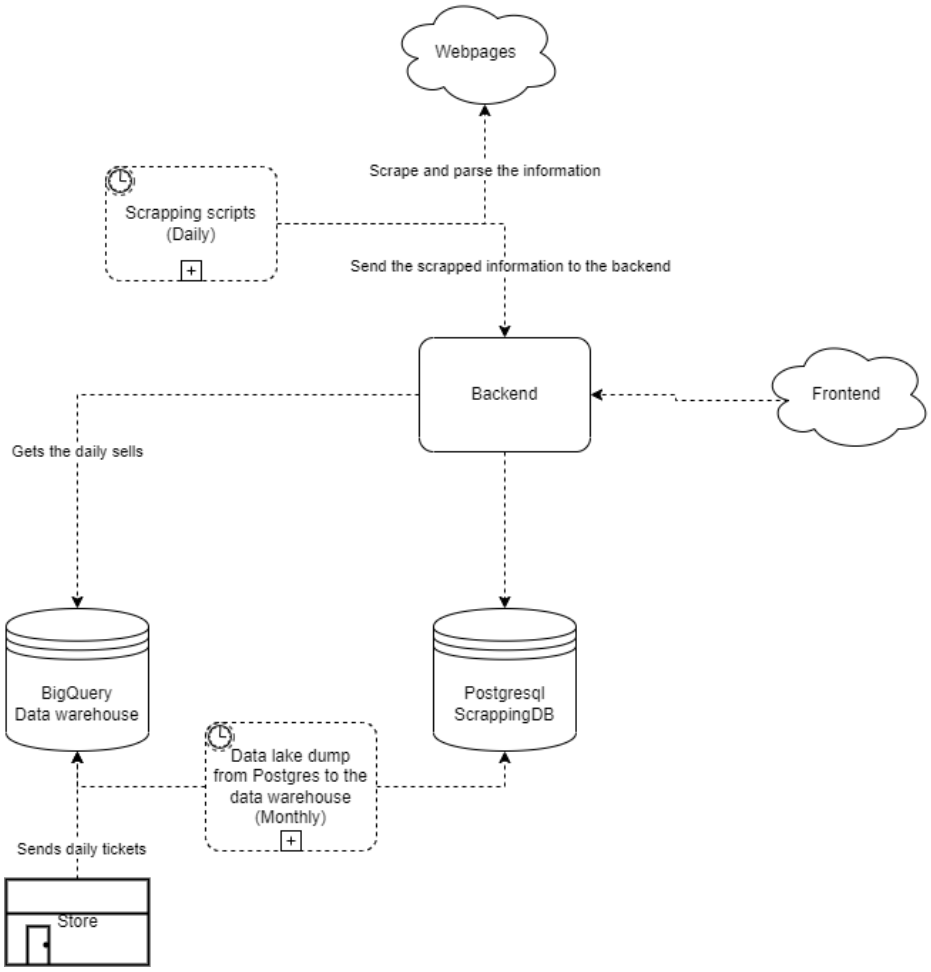


Fig. 4.4.4 Diagrama del proceso de scraping. Fuente: Elaboración propia

4.5 Diseño de la arquitectura del backend

4.5.1 Elección del lenguaje

La elección del lenguaje no es algo trivial ya que va a condicionar a que herramientas hay acceso.

En el caso de Ametller, se utiliza Scala, así que esta es una decisión la cual no se puede cambiar ya que viene dada por el negocio.

En caso de tener la oportunidad, se recomienda algún framework de PHP como Symfony o Laravel ya que PHP es un lenguaje que lleva establecido como uno de los lenguajes principales de backend y web desde hace más de 15 años [14] y hay centenares de miles de herramientas en repositorios como *packagist* que se acceden gracias a composer tanto en el caso de Laravel o de Symfony. Otra opción sería un entorno .Net como ASP.NET, de aquí en adelante ASP, que actualmente tiene un 7% de uso según W3 technologies [15]. ASP tiene la gran ventaja tener la API de .NET y sus magníficas

herramientas, en concreto Entity Framework, un mapeador de objetos a base de datos que utiliza LINQ para hacer todas las consultas en base de datos con los mismos métodos que se usan para las listas en memoria con el añadido de soportar múltiples bases de datos [16].

4.5.1.1 Mediator

Para conseguir un acoplamiento bajo se puede usar un patrón de tipo mediador que consiste en tener un objeto mediador que está acoplado a otros componentes. Cada componente puede enviarle un mensaje al mediador, sin saber el receptor o receptores final, y el mediador se encarga de hacer llegar el mensaje a quien lo necesite, si es que está presente.

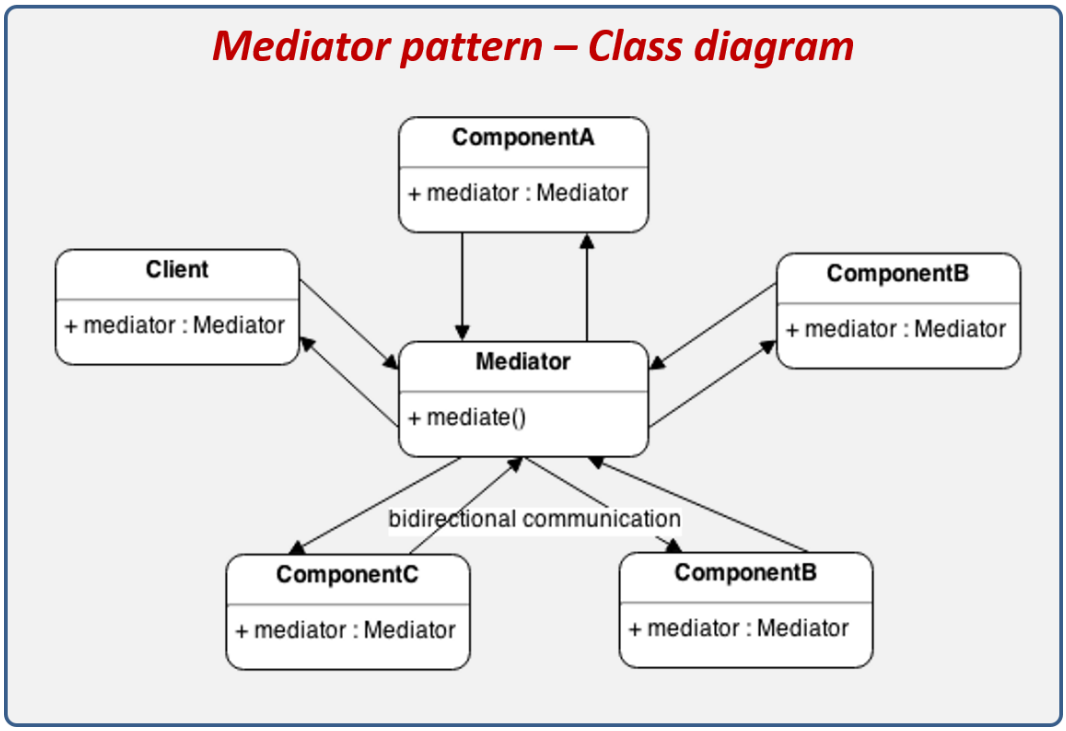


Fig. 4.5.1 Patrón Mediador fuente: reactiveprogramming.io

A pesar de que te hace acoplar a un patrón en específico para la mayor parte de la arquitectura, mejora en gran cantidad el acople ya que la aplicación en sí misma no necesita de conocer los componentes acoplados al mediador.

4.6 Generación de código automática

La arquitectura limpia es una arquitectura que necesita de una estructura de carpetas muy específica y que, por ende, se beneficia mucho de los generadores de código. A la hora de crear entidades es común que, simplemente en la capa de dominio y aplicación, se tenga que crear:

1. La entidad
2. El *DTO* de la entidad
3. Un repositorio base para esa entidad
4. Casos de uso para esa entidad.

Y en la mayoría de casos es una copia de las entidades base cambiando partes arbitrarias.

Si sumamos lo anterior a que la estructura que se ha de seguir es muy específica, se presentan las bases para que la generación de código brille en su máximo esplendor.

4.7 Backend

4.7.1 Capa de dominio

La capa de dominio es la capa más importante de toda la arquitectura, ya que es la que se encarga de tener todas las entidades relacionadas con el negocio, que no sean específicas de alguna capa, y sus reglas de negocio.

La mejor opción es escribir la capa de dominio en C++ con el mínimo de librerías posibles y desde ahí hacer wrappers a los idiomas necesarios. Todo esto con *SWIG* es sencillo de conseguir ya que permite exportar a una gran variedad de lenguajes. Al tener escrito el dominio en C++ y llevarlo con wrappers a distintos idiomas se hace más consistente el hecho de tener un solo dominio compartido con toda la arquitectura interna de la empresa.

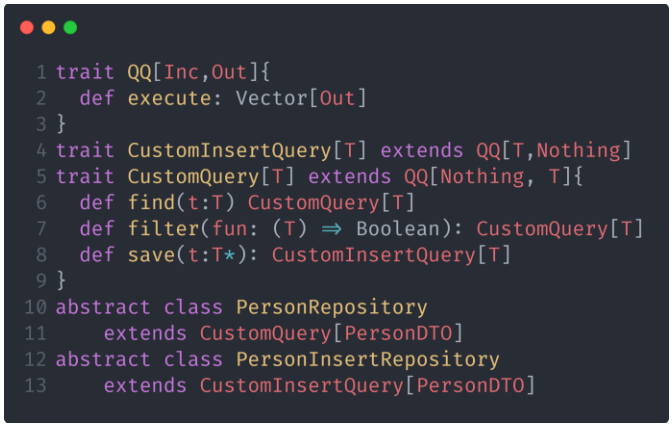
Debido a los requisitos de la solución, no se puede aplicar.

4.7.2 Capa de Aplicación

La capa de aplicación depende únicamente de la capa de dominio, hay solo un programa en la capa de aplicación, y es la capa de aplicación la única que puede depender de la capa de dominio.

En esta capa se definen todos los *DTOs* necesarios, así como interfaces que han de implementar capas superiores, como por ejemplo repositorios. Ya que los *DTOs* van a necesitar pasar los datos de cualquier sitio, existe la posibilidad de que se hayan de almacenar relacionalmente y necesiten de un identificador, por eso mismo se ha añadido en la entidad base una propiedad llamada *id* de tipo *Any* para que esté abierta a cualquier tipo de identificador exterior, incluso tenerlo nulo.

Para recoger datos de las capas de persistencia se han valorado y probado distintos métodos de repositorios, modos de hacer entidades y como hacer la comprobación de que son válidas. El mayor problema se tuvo a la hora de escoger como iban a ser los repositorios, ya que un acercamiento como el observable en la Fig. 4.7.1 es muy interesante, pero en la práctica es muy complejo de implementar y acaba suponiendo una deuda técnica, ya que es difícil de mantener, y por lo tanto es muy difícil de sustituir en capas superiores lo que puede generar un mayor acoplamiento que un acercamiento de tipo repositorio.



```
trait QQ[Inc,Out]{
  def execute: Vector[Out]
}
trait CustomInsertQuery[T] extends QQ[T,Nothing]
trait CustomQuery[T] extends QQ[Nothing, T]{
  def find(t:T) CustomQuery[T]
  def filter(fun: (T) ⇒ Boolean): CustomQuery[T]
  def save(t:T*): CustomInsertQuery[T]
}
abstract class PersonRepository
    extends CustomQuery[PersonDTO]
abstract class PersonInsertRepository
    extends CustomInsertQuery[PersonDTO]
```

Fig. 4.7.1 Repositorio tipo query. Fuente: Elaboración propia

4.7.3 Capa de persistencia

La capa de persistencia es la primera capa por encima de la capa de aplicación. Al estar por encima de la capa de aplicación no puede depender ni usar la capa de dominio, así que se han de usar los *DTOs* en todo momento.

En esta capa se implementan los repositorios que se definen en la capa de aplicación. Aunque no es necesario implementarlos todos, en este caso solo se van a implementar los repositorios de las entidades de scraping.

También se definen *DAOs*, *Data Access Objects*, que son un acercamiento lo más cercano posible a las entidades de bases de datos. Como se observa en la Fig. 4.7.2, en vez de tener una lista de autores en la clase *ScrappingBookDAO* se tiene una clase complementaria que relaciona un Id de libro con un Id de autor, del mismo modo no se tiene la entidad *EditorialDAO, sino que tiene únicamente el ID. Todas estas clases necesitan de un modo de conversión rápido para pasar de DTOs a DAOs, ya que los DAOs* son específicos de cada módulo e incluso dos módulos en el mismo nivel de capa, como podrían ser

una API REST y una capa de persistencia, no necesitan compartir los *DAOs*, de hecho, serían malas prácticas que lo hicieran.

```scala
case class ScrappingBookDAO(
    id: BookId,
    name: String,
    isbn: String,
    editorial: EditorialId,
    rating: Optional[Int]
)
case class EditorialDAO(
  id: EditorialId,
  name: String
)
case class AuthorBookDAO(
  bookId: BookId,
  authorId: AuthorId
)
```

Fig. 4.7.2 Ejemplo de DAOs con Libros. Fuente: Elaboración propia

4.8 Diseño de los servicios de scraping

4.8.1 Elección del lenguaje

Se ha elegido Python debido a su versatilidad. Para empezar, se puede correr Python en prácticamente cualquier entorno moderno y llamarlo desde prácticamente cualquier lenguaje, incluido Scala. Para continuar, tiene un gran repertorio de librerías muy útiles para scraping como *requests*, *BeautifulSoup4*, *lxml*, *Selenium* o incluso un framework completo de scraping como *Scrapy*. Por último, aunque, Python sea claramente más lento que otros lenguajes [17], aquí no es necesaria la performance ya que está pensado para ejecutarse una vez al día y con cada script en paralelo.

4.9 Diseño de la arquitectura del frontend

4.9.1 Elección del lenguaje

Del mismo modo que en el backend, en Casa Ametller se usa Angular para los frontend web, aunque a diferencia del backend, Angular si es una opción que se recomienda.

Aunque angular por sí solo no funciona como framework multiplataforma, en caso de necesitar si o si un framework multiplataforma se optaría por una de las tres siguientes opciones.

4.9.1.1 Ionic

Ionic es un framework multiplataforma de desarrollo para móviles[18]. Ionic tiene sus propios componentes para UI, muy similares a los de Material Design, que están diseñados para funcionar a la perfección tanto en IOS como en Android como en Web. Ionic deja acceder a componentes nativos con sus Native APIs[19] lo que facilita las cosas cuando se necesitan componentes físicos como la cámara o los sensores biométricos de autentificación. Actualmente se puede usar Ionic con Angular, React, Vue y JS puro.

Debido a toda la versatilidad, es la primera opción que se escogería.

4.9.1.2 MAUI

MAUI es el acercamiento de Microsoft con .Net a la multiplataforma, siendo una evolución directa de Xamarin.Forms pero que por debajo ha recibido una reestructuración interna para modernizarse y mejorar la compatibilidad[20]. MAUI trae un gran rendimiento ya que compila de forma nativa a todas las plataformas que promete, Windows, macOS, Android y iOS. Viene integrado y preparado para utilizar Fluent Design, la guía de estilos de Microsoft.

En caso de necesitar una aplicación que hiciera mejor uso de la CPU y de los componentes nativos, MAUI sería la elección, tiene un gran equilibrio entre rendimiento, versatilidad y comunidad.

4.9.1.3 Flutter

Flutter, un Framework open source de Google para Dart que compila de forma nativa en ARM, X86 y JavaScript[21]. Debido a que está siendo desarrollado por Google, todos los componentes de Flutter siguen las guías de diseño de Material Design, también de Google. Flutter permite un desarrollo rápido y flexible.

Debido a que es un framework relativamente nuevo, tuvo su lanzamiento inicial en 2017, Flutter suele hacer muchos cambios entre versiones. Si se le suma Dart, un lenguaje también creado por Google no muy conocido, que el rendimiento en escritorio está por debajo de lo que debería y que Google perdió credibilidad después de anunciar el fin del desarrollo de AngularJS de forma abrupta[22], sin duda es una elección que está por debajo de las anteriores. Es una buena elección si el target principal es solo Android y iOS.

4.9.2 Arquitectura de Angular.

Angular está pensado para usarse con una arquitectura MVC, model-view-controller, con componentes. Y está estructurado alrededor de esa arquitectura. A pesar de eso, no hay nada que

impida que se use Clean Architecture en Angular. Después de una reunión interna con el equipo de Casa Ametller se decidió no usar Clean Architecture ya que no se podía beneficiar tanto de los casos de uso y su abstracción como en Backend. Uno de los problemas fue el tener la capa de dominio en Scala y no en C++ como se comenta en el apartado 4.7.1.

4.10 Frontend

4.10.1 Diseño de la interfaz gráfica

Para el diseño de la interfaz gráfica se ha usado el software *Figma*. Se partió de una base que se necesitaba una lista de productos, una gráfica y un selector de fechas. Se puede ver el primer Mockup que se presentó a los clientes internos en la Fig. 4.10.1.

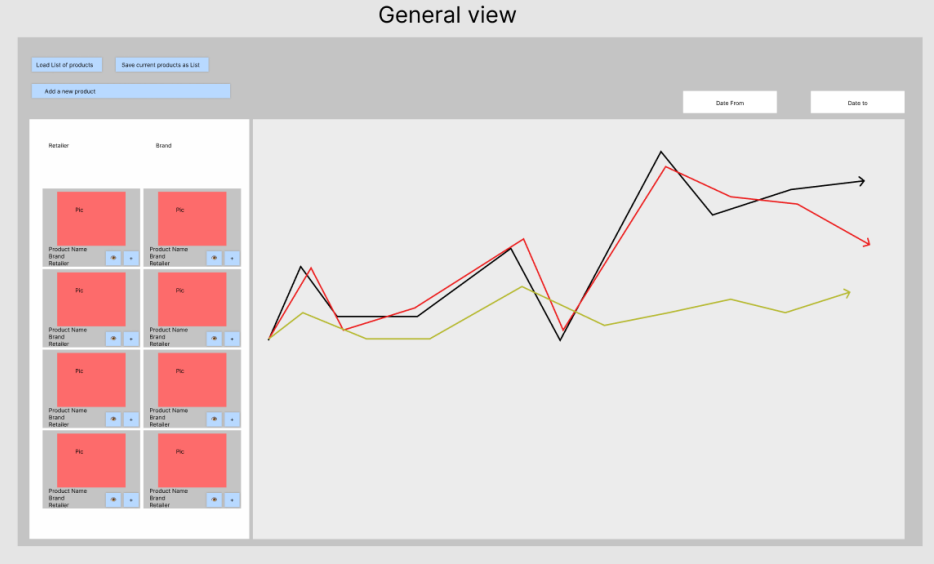


Fig. 4.10.1 Mockup de la primera versión. Fuente: Elaboración propia

Gracias al feedback recibido después de mostrar la primera versión, se decidió que la lista de productos iba a ser una tabla. En esa tabla cada fila será un producto de Ametller, y cada columna será el precio del día actual. Se puede ver en la Fig. 4.10.2.

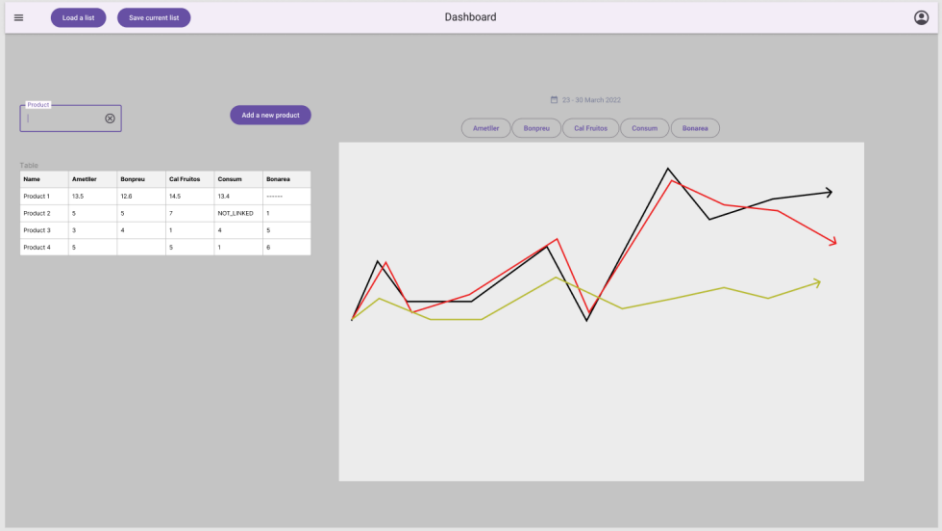


Fig. 4.10.2 Mockup de la segunda versión. Fuente: Elaboración propia

Después de más revisiones con el cliente interno, la última versión del diseño añade una columna extra por distribuidor, haciendo que haya dos columnas por distribuidor. La primera siendo el precio manual que han podido encontrar yendo a las tiendas físicas y la segunda siendo el precio de scraping. En el caso de Ametller la primera columna es el precio de coste y la segunda el precio de venta.

Se añaden un selector de fecha para la tabla, que decide sobre qué día se tienen que observar esos datos.

En vez de un gráfico se añaden tres gráficos. Si se pulsa a una fila, se carga el gráfico de comparación histórico, mientras que si se pulsa a la columna de algún distribuidor se carga un gráfico que compara la diferencia de precio de cada producto con el enlazado a ese distribuidor y las ventas de ese día. Este último gráfico ha de tener dos ejes Y distintos, uno para la diferencia de precio y otro para las ventas, y en el eje X ha de tener el código de producto. El último gráfico es una mezcla entre el primero y el segundo, ya que se mantienen los ejes Y del segundo, pero el eje X es un eje temporal, igual que el primero. Para activar el segundo o el tercero hay un *toggle-slider*.

5 Desarrollo

5.1 Backend

5.1.1 Falta de librerías

Una de los mayores problemas de escala, sino el mayor, es su falta de librerías y el problema de compatibilidades entre ellas. Scala es un lenguaje que *puede* correr en la JVM, cosa que hace por defecto, habilitando así el uso de prácticamente cualquier librería en Java. Siempre se intenta que las librerías sean específicas para Scala y no pasen por librerías Java ya que Scala tiene ciertos problemas al pasar de Java a Scala con los tipos genéricos lo que puede inducir a comportamientos no esperados [23]. Aún si el repertorio de paquetes en Scala fuera lo suficientemente grande, tiene un problema todavía más grande, que son las incompatibilidades entre librerías, binarios y versiones de Scala. Es común encontrarte con paquetes que son compatibles solo para Scala 2.12 e inferiores ya que de Scala 2.12 a 2.13 hubo un gran cambio de sintaxis para que la migración a Scala 3 fuera más sencilla [24] haciendo así que un gran número de librerías Open Source no se puedan actualizar a Scala 2.13 simplemente por dependencias con otras librerías.

5.1.1.1 Generación automática de código con Noot-scaffold

Como se menciona anteriormente, hacer generación automática de código facilita tanto la creación de código siguiendo la estructura y normas de la arquitectura como la velocidad a la que se trabaja ya que se elimina mucho trabajo repetitivo y tedioso. El problema es que la mayoría de generadores de código son para un lenguaje o frameworks específicos, y los que no lo son acaban dependiendo de algún lenguaje poco común o tienen algún fallo grande, como podría ser el generador de giter8 que depende de *sbt* y tiene problemas con algunos terminales.

Debido a no existir una solución que se ajuste a las necesidades de multiplataforma e independencia entre lenguajes y frameworks se ha creado una librería de scaffoling que está subida en forma de una herramienta de NuGet, así que para instalarla solo se necesita .net6 y el comando *dotnet tool install --global noot-scaffold*. El código es público en GitHub para que cualquier persona pueda hacer un fork y modificarlo a su gusto[25].

Al no estar pensada para un lenguaje o situación específica la librería funciona con todo tipo de estructura de carpetas y archivos.

Su funcionamiento es sencillo pero escalable, la librería escanea en el directorio actual en búsqueda de una carpeta que se llame "*.scaffold*". Al encontrar la carpeta busca todas las carpetas que haya dentro, cada carpeta encontrada será un scaffold de los posibles a escoger. En la raíz de cada scaffold ha de haber un archivo llamado *scaffold.properties* en donde irán las palabras claves que se sustituirán a posterior, así como una descripción opcional de que hace ese scaffold en concreto.

Fig. 5.1.1 Scaffold de ejemplo. Fuente: Elaboración propia

5.1.1.1.1 Ejecución

Teniendo la estructura de la Fig. 5.1.1 si se ejecuta el comando *nootscaffold* sin ningún argumento, como se observa en la Fig. 5.1.2, preguntará que scaffold se quiere seleccionar. En este caso no hace falta que se introduzca nada ya que entity, que es el ejemplo, es el primero. A continuación, se muestra la descripción y se pregunta una por una todas las claves que se encuentran el archivo *scaffold.properties* y en caso de introducir un texto, se sobrescribe esa variable. Al acabar con todas se muestra todas las carpetas y archivos que se crean. El resultado del scaffold se observan en la Fig. 5.1.3.

```
> nootscaffold.exe
No scaffold specified, please select one of the following:
[0] entity
[1] usecase

Scaffold selected [0]:

Description: this is an scaffold that generates a book with a configuration
entity [book]:
package [hello world]:hello world scaffold
configuration [release]:debug
Created file C:\Users\Aitor\dev\playground\scaffold\book.scala
created folder C:\Users\Aitor\dev\playground\scaffold\book
created folder C:\Users\Aitor\dev\playground\scaffold\book\configurations
Created file C:\Users\Aitor\dev\playground\scaffold\book\configurations\debug.sbt
```

Fig. 5.1.2 Output de ejemplo. Fuente: Elaboración propia.

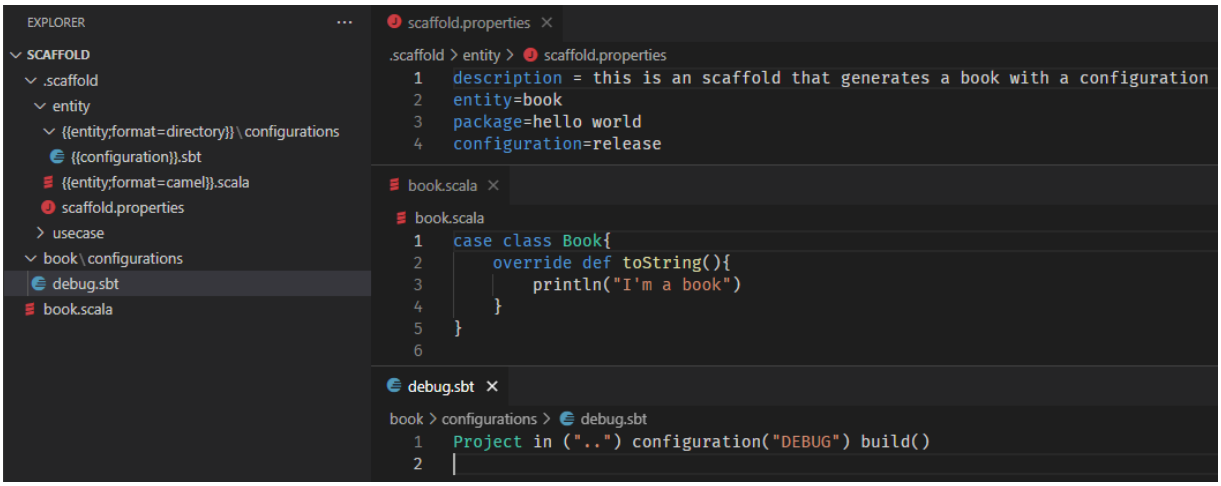


Fig. 5.1.3 Resultado de generar el scaffold entity. Fuente: Elaboración propia.

Es posible pasar como primer argumento el nombre del scaffold y en caso de existir selecciona directamente el scaffold, como se muestra en la Fig. 5.1.4. Para que sea más fácil generar muchos scaffolds en los que se repiten muchos parámetros también se puede pasar como argumento una palabra clave junto a lo que se quiere reemplazar, como en la Fig. 5.1.5.

```
> nootscaffold.exe entity

Description: this is an scaffold that generates a book with a configuration
entity [book]:
```

Fig. 5.1.4 Ejemplo de escoger un scaffold por argumento. Fuente: Elaboración propia.

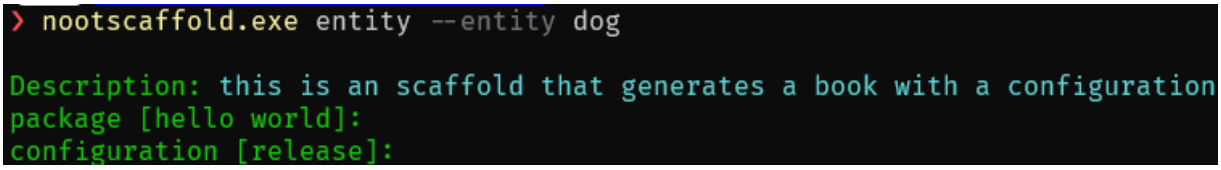


Fig. 5.1.5 Ejecución con un scaffold y una palabra clave por defecto. Fuente: Elaboración propia

En caso de tener archivos repetidos el generador los detecta y pregunta sobre si se quiere saltar, sobrescribir o cancelar la ejecución. En caso de cancelar la ejecución los archivos creados no se borran.

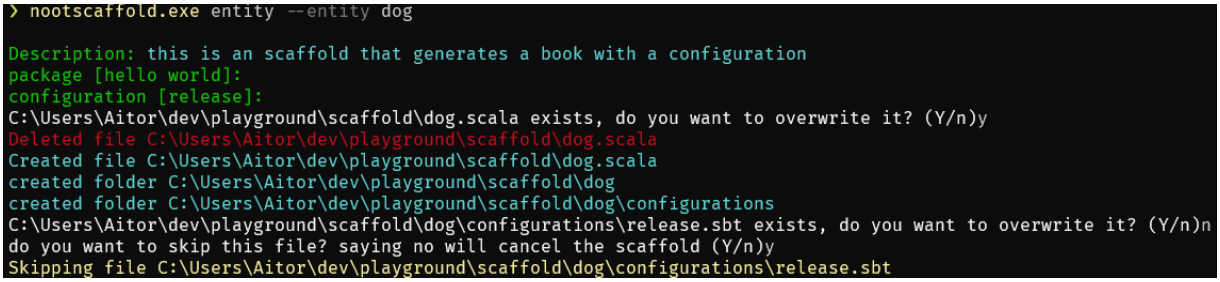


Fig. 5.1.6 Ejecución con archivos ya existentes. Fuente: Elaboración propia.

5.1.1.2 Implementación de mediator en Scala con Scala-mediator

Mientras que en .NET existe la librería MediatR, que está totalmente integrada con la inyección de dependencias, Dependency Injection en inglés y de aquí en adelante DI, de Microsoft, en Scala no existe ninguna librería que haga una implementación del patrón mediator, así que se tuvo que hacer la librería de 0. Una vez hecha, se publicó en GitHub[26]. Aunque tiene pocas líneas es un código muy compacto que usa funcionalidades de Scala avanzadas.

```scala
sealed abstract class Command[+RETURN_TYPE: TypeTag] {

}

abstract class MultiCommand extends Command[Unit] {

}
abstract class SingleCommand[RETURN_TYPE: TypeTag] extends Command[RETURN_TYPE] {

}

sealed abstract class CommandHandler[COMMAND <: Command[TARGET_TYPE] : TypeTag, TARGET_TYPE] {
  def handle(cmd: COMMAND): TARGET_TYPE

  def messageType: Type = typeOf[COMMAND]
}

abstract class SingleCommandHandler[COMMAND <: SingleCommand[TARGET_TYPE] : TypeTag, TARGET_TYPE] extends CommandHandler[COMMAND, TARGET_TYPE] {
  def handle(cmd: COMMAND): TARGET_TYPE

  override def messageType: universe.Type = typeOf[COMMAND]
}

abstract class MultiCommandHandler[COMMAND <: MultiCommand : TypeTag] extends CommandHandler[COMMAND, Unit] {
  def handle(cmd: COMMAND): Unit
}
```

Fig. 5.1.7 Ejemplo de la librería scala-mediator. Fuente: Elaboración propia

5.1.2 Capa de dominio

La estructura de las carpetas en la capa de dominio se puede estructurar de distintos modos, pero en este caso se ha optado por hacer una distinción entre el tipo de objecto en el primer nivel, entidades, value-objects, excepciones, etc, e ir especificando cada vez que se añade un nivel de carpeta según sea necesario. Como se puede observar en la Fig. 5.1.8 *Product* No está en ninguna subcarpeta dentro de entities ya que es una entidad que afecta a todo el negocio.

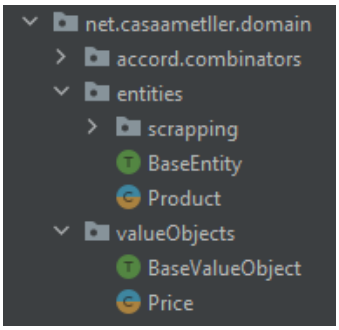


Fig. 5.1.8 Estructura de carpetas de la capa de dominio. Fuente: Elaboración propia

5.1.2.1 Dependencia con Accord

Accord es una librería de validación de datos para Scala que provee su propio DSL. Con Accord se pueden validar las reglas de negocio sobre las entidades de una forma más natural, como se puede observar en la Fig. 5.1.9. El tener un DSL tan natural ha sido el motivo principal de depender de esta librería para la capa de Dominio, aunque se pueda implementar sin necesidad de la librería, esto lo hace más legible para cualquier persona de negocio.

```scala
case class Person( firstName: String, lastName: String )
case class Classroom( teacher: Person, students: Seq[ Person ] )

implicit val personValidator = validator[ Person ] { p ⇒
  p.firstName is notEmpty
  p.lastName as "last name" is notEmpty
}

implicit val classValidator = validator[ Classroom ] { c ⇒
  c.teacher is valid
  c.students.each is valid
  c.students have size > 0
}
```

Fig. 5.1.9 Ejemplo de validación con Accord. Fuente: Elaboración propia

5.1.2.2 Testing

No se ha encontrado ninguna necesidad de escribir tests ya que para eso están los validadores de Accord, y conceptualmente no tiene sentido hacer tests para los validadores ya que los validadores son el test en sí mismo de cada entidad.

5.1.3 Capa de aplicación

En este caso se ha creado una *BaseEntityDTO, que se puede observar en la* Fig. 5.1.10, para que las todas las entidades extiendan de ahí.

```scala
abstract class BaseEntityDTO[T <: BaseEntity[T]] {
  val id: Any
  def toEntity: T
  def validate(): Result
}
```

Fig. 5.1.10 Entidad base. Fuente: Elaboración propia

En la Fig. 5.1.11 se puede observar un ejemplo de entidad que para el id utiliza un *AbstractId*, una interfaz pensada para que sea extendida en capas superiores.

```scala
case class ScrappingRetailerDTO(
    override val id: Option[AbstractId],
    val name: String
) extends BaseEntityDTO[ScrappingRetailer] {
  override def toEntity: ScrappingRetailer = this.toScrappingRetailer
  override def validate(): Result = accord.validate(this.toEntity)
}
```

Fig. 5.1.11 Entidad para un retailer. Fuente: Elaboración propia

```scala
trait AbstractId extends Any {
  def value: Any
  def ==(other: AbstractId): Boolean
  def !=(other: AbstractId): Boolean
  def >(other: AbstractId): Boolean
  def <(other: AbstractId): Boolean
  def >=(other: AbstractId): Boolean
  def <=(other: AbstractId): Boolean
}
```

Fig. 5.1.12 AbstractId. Fuente: Elaboración propia

Los repositorios tienen todos unos métodos base que vienen del *BaseRepository*, una interfaz de la que extenderán todos los repositorios de la capa de aplicación Se puede observar en la Fig. 5.1.13 y un ejemplo de implementación en la Fig. 5.1.14. En la Fig. 5.1.14 no se necesitan métodos extra así que se dejan sin implementar ni definir nuevos campos, ya que va a ser esta clase la que va a ser inyectada en el código y extendida por otras capas como se observa en la Fig. 5.1.18.

```scala
trait BaseRepository[T] {
  def find(t: T): Option[T]
  def save(t: T*): Unit
  def delete(t: T*): Unit
  def update(t: T*): Unit
  def saveOrUpdate(t: T*): Unit
}
```

Fig. 5.1.13 Interfaz base para los repositorios. Fuente: Elaboración propia

```
1 abstract class ScrappingProductRepository extends BaseRepository[ScrappingProductDTO]
```

Fig. 5.1.14 Interfaz específica para inyectar en un caso de uso.

La estructura de carpetas es similar al anterior, donde cada carpeta es un elemento compartido hasta que se llega a una unidad mínima donde está la entidad concreta. En cada carpeta ha de haber una y solo una entidad junto con su repositorio, y en este caso debido a limitaciones de Scala un object para hacer conversiones del DTO al objeto del dominio. Y en cada carpeta de entidad puede haber las carpetas de *commands*, *queries* y *exceptions* según se necesiten. En la carpeta commands van todos los casos de uso que necesiten de modificar datos. En la carpeta *queries* van todos los casos de uso relacionados con hacer búsquedas. Y en la carpeta *exceptions* van todas las excepciones directas de la entidad. En la Fig. 5.1.15 se enseña una parte de la estructura de carpetas de la capa de aplicación.

Es esta la capa que más se beneficia de usar scaffolds ya que a la hora de generar un caso de uso se sustituyen más de veinte campos predefinidos y a la hora de crear una entidad más cincuenta campos predefinidos. Aunque actualmente se han usado más los scaffolds de generar entidades que casos de uso, al acabar el proyecto habrá muchos más casos de uso que entidades.

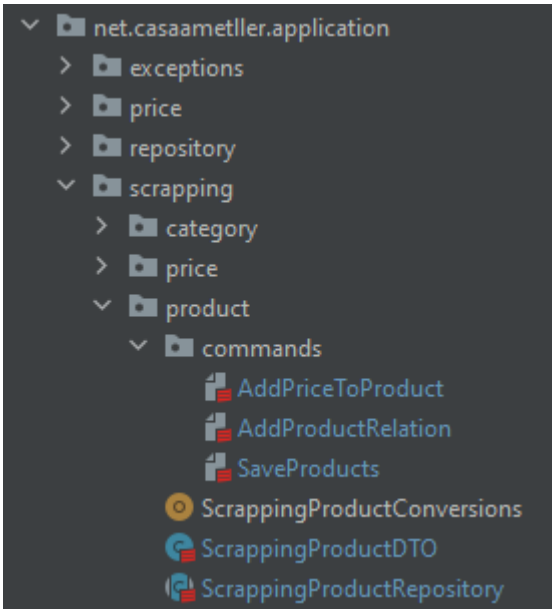


Fig. 5.1.15 Estructura de carpetas de la capa de aplicación

5.1.3.1 Dependencia con Scala-Mediator

En esta capa es donde empieza la dependencia con scala-mediator ya que cada caso de uso implementa un *SingleCommandHandler*.

```
1  sealed class SaveProducts(val products: ScrappingProductDTO*)
2      extends SingleCommand[Boolean]
3  sealed class GetByDateHigherHandler @Inject() (
4      val provider: Provider[IMediator],
5      val repository: ScrappingProductRepository
6  ) extends SingleCommandHandler[SaveProducts, Boolean] {
7    override def handle(command: SaveProducts): Boolean =
8      command
9        .products
10       .find(product ⇒
11         product.validate() match {
12           case Failure(_) ⇒ true
13           case _          ⇒ false
14         }
15       ) match {
16       case Some(product) ⇒
17         throw new IllegalArgumentException(
18           product.validate().toString
19         )
20       case None ⇒
21         repository
22           .save(command.products: _*)
23     }
24 }
```

Fig. 5.1.16. Caso de uso de guardar un producto. Fuente: Elaboración propia

5.1.3.2 Dependency Injection

En la línea 3 de la Fig. 5.1.16 se puede observar un *@Inject*, aunque parezca que se ha acoplado funcionalidad a la capa, es totalmente sustituible la DI. Sustituir la DI es un cambio que es totalmente externo de la capa de aplicación, así que se cumple que no esté acoplado.

5.1.3.3 Testing

En esta capa en concreto no hay tests ya que no hay implementaciones definidas además de las conversiones. Si algún caso de uso falla es porque no se ha definido bien en los requisitos, o bien porque los requisitos no se han seguido de una forma exacta, en ambos casos eso es algo que no se puede testear por código de forma específica. Se podrá testear la implementación de las interfaces definidas donde se implementen dichas interfaces, es decir, en capas superiores a las que no se tiene acceso desde la capa de aplicación.

5.1.4 Capa de persistencia

5.1.4.1 Quill

Para esta capa se ha decidido usar quill, una librería con un *QDSL*, *Quoted Domain Specific Language*. Quill permite hacer queries a una base de datos usando las mismas funciones de una lista, como se puede observar en la línea 12 de la Fig. 5.1.18 pero que compilan en tiempo de compilación para tener una query rápida en runtime. Si la tabla es diferente de la entidad, sobre todo por diferencias entre la nomenclatura, se puede crear un esquema personalizado como se muestra en la Fig. 5.1.17.

```
1 final val scrappingBookDAO = quote {
2     querySchema[scrappingBookDAO](
3       entity = "book",
4       _.id → "id",
5       _.name → "name",
6       _.isbn → "ISBN",
7       _.editorial → "editorial_id",
8       _.rating → "rating"
9     )
10  }
```

Fig. 5.1.17 Ejemplo de un esquema personalizado. Fuente: Elaboración propia

```
1 case class QuillScrappingProductRepository(database: PgContext) extends ScrappingProductRepository {
2   def delete(t: ScrappingProductDTO*): Unit = {
3     val dbQ = PublicSchema.scrappingProductDAO
4     val lis = t.map(prod ⇒
5       (prod, database.run(quote(findQuery(dbQ, prod).take(1))).headOption)
6     )
7     if (lis.exists(_._2.isEmpty)) {
8       throw new ObjectNotFoundException()
9     }
10    val l = lis.map(_._2.get.id)
11    val deleteQuery = quote {
12      dbQ.filter(p ⇒ liftQuery(l).contains(p.id)).delete
13    }
14    database.run(deleteQuery)
15  }
16 }
```

Fig. 5.1.18 Implementación de un repositorio para ScrappingProduct con Quill para PostgreSQL. Fuente: Elaboración propia

5.2 Servicios de scraping

5.2.1 Servicio base

Se han creado los archivos base necesarios, entre los que se incluyen los archivos de dominio. En caso de hacer el dominio en C++ hubiera sido tan sencillo como hacer un wrapper para Python y utilizar todo lo escrito anteriormente.

Se ha creado un archivo base del cual extenderán todos los servicios, se puede ver en la Fig. 5.2.1 .

```python
import abc
from base_files.product import Product
class ScrappingServiceBase(metaclass=abc.ABCMeta):
    retailer: str
    @abc.abstractmethod
    def run(self) -> list[Product]:
        pass
```

Fig. 5.2.1 Clase base para los servicios de scraping. Fuente: Elaboración propia

5.3 Frontend

5.3.1 Separación de componentes

Angular permite la creación de componentes, partes de código con HTML, CSS y código que son reutilizables y están asilados. Al estar aislados, no permiten interferencias externas, cosa que ayuda mucho con CSS.

A pesar de estar aislados, es posible tanto enviar objetos como recibir eventos. Esto se hace mediante los decoradores *@Input* y *@Output()*. En la Fig. 5.3.1 se puede ver un ejemplo de cómo se declaran las variables de input y output.

```
@Component({
  selector: 'app-name-selector',
  templateUrl: './name-selector.component.html',
  styleUrls: ['./name-selector.component.sass']
})
export class NameSelectorComponent implements OnInit {
  @Input() names: Array<string>;
  @Outut() onNameSelected: EventEmitter<string>;
}
```

Fig. 5.3.1 Ejemplo de componente. Fuente: Elaboración propia

Para utilizar esos componentes es tan sencillo como poner el selector en HTML. En caso de tener variables de *Input* se escribe el nombre entre corchetes, y en caso de tener variables de output se escriben entre paréntesis, como se observa en la Fig. 5.3.4. Es posible crear variables de Input que sean obligatorias con decoradores personalizados. También es posible tener una variable de input y output a la vez, aunque esto no se recomienda.

```
<app-name-selector
                  [names]="filteredNames"
                  (onNameSelected)="changeCurrentName($event)">
</app-name-selector>
```

Fig. 5.3.2 Ejemplo de componentes con variables de input y output. Fuente: Elaboración propia

Se han creado componentes para todas las cosas necesarias. El criterio que se ha seguido para determinar si algo es necesario o no de hacer componente ha sido que una parte de código se pueda aislar del resto y se gane un beneficio haciéndolo, aunque que ese beneficio sea simplemente un código más limpio o estructurado.

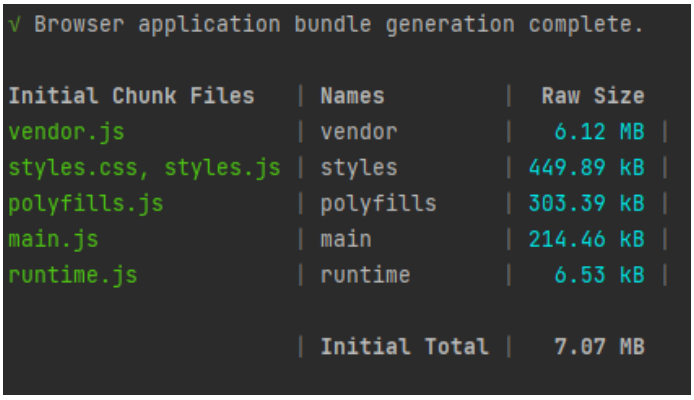
5.3.1.1 Lazy Loading

Angular por defecto carga todos los componentes de forma inmediata, eso significa que, aunque no usemos un componente de nuestra aplicación, este está cargado. Eso significa una mayor carga de memoria, así como un mayor tiempo de inicio. Para solventar ese problema, angular permite hacer *lazy loading*, es decir, que cargue un componente solo cuando esté en uso. Una de las partes más

importantes con *lazy loading* es separar correctamente los componentes y modulos. En caso de necesitar componentes compartidos entre módulos, estos se ponen en un módulo compartido, *shared module*. Del mismo modo que es importante saber separar los componentes y módulos, es importante saber que ha de ir en el módulo compartido, ya que se cargará en todos los módulos. Un mal uso del módulo *shared* puede conllevar a un rendimiento inferior al que se podría llegar a tener con todos los componentes en *eager loading*

En este caso, hay un módulo de autentificación llamado *auth* que se compone de los componentes de Login y de Logout, otro llamado *shared* que se encarga de los componentes compartidos y el último llamado *dashboard* que se encarga de todos los componentes y diálogos que pertenecen al *dashboard*.

Observando la diferencia del *Initial Total* entre la Fig. 5.3.3 y la Fig. 5.3.4 se puede ver que hay una diferencia total de 2,43 MB de carga inicial.

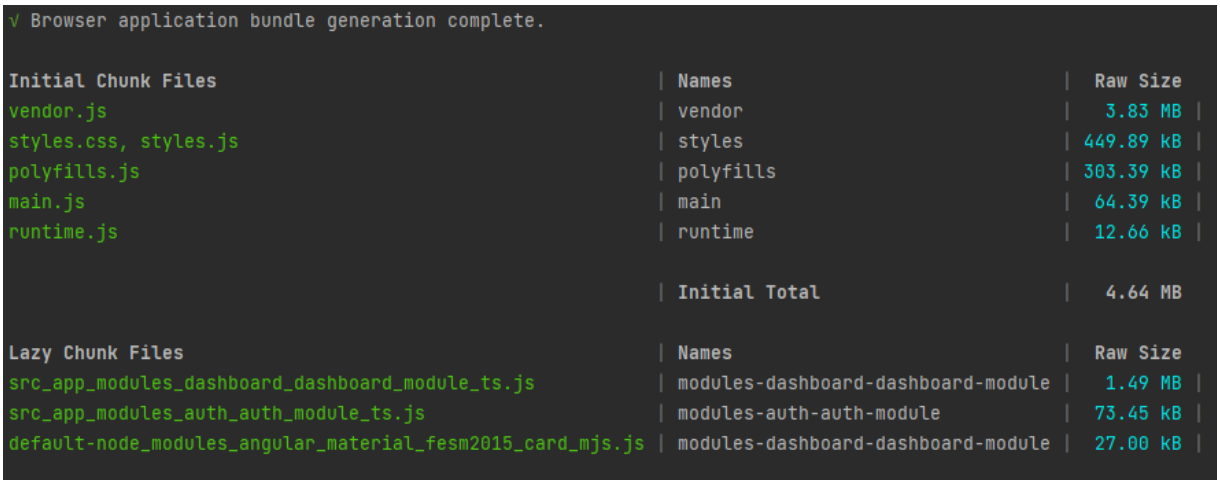


```
√ Browser application bundle generation complete.

Initial Chunk Files   | Names         |  Raw Size
vendor.js             | vendor        |   6.12 MB |
styles.css, styles.js | styles        | 449.89 kB |
polyfills.js          | polyfills     | 303.39 kB |
main.js               | main          | 214.46 kB |
runtime.js            | runtime       |   6.53 kB |

                      | Initial Total |   7.07 MB
```

Fig. 5.3.3 Bundle de la aplicación en modo desarrollo sin lazy loading. Fuente: Elaboración propia



```
√ Browser application bundle generation complete.

Initial Chunk Files                                | Names                                | Raw Size
vendor.js                                          | vendor                               |   3.83 MB |
styles.css, styles.js                              | styles                               | 449.89 kB |
polyfills.js                                       | polyfills                            | 303.39 kB |
main.js                                            | main                                 |  64.39 kB |
runtime.js                                         | runtime                              |  12.66 kB |

                                                   | Initial Total                        |   4.64 MB

Lazy Chunk Files                                   | Names                                | Raw Size
src_app_modules_dashboard_dashboard_module_ts.js   | modules-dashboard-dashboard-module   |   1.49 MB |
src_app_modules_auth_auth_module_ts.js             | modules-auth-auth-module             |  73.45 kB |
default-node_modules_angular_material_fesm2015_card_mjs.js | modules-dashboard-dashboard-module |  27.00 kB |
```

Fig. 5.3.4 Bundle de la aplicación en modo desarrollo con lazy loading. Fuente: Elaboración propia

5.3.2 Enrutamiento

Al estar pensado para páginas web, angular permite enrutamiento de forma sencilla. El enrutamiento de angular permite navegar entre componentes interpretando la URL del buscador. De ese modo, entre otros beneficios, se pueden hacer páginas de redirección y accesos directos.

Uno de los puntos positivos del enrutamiento, es que es lo que habilita que haya *lazy loading*. Se puede observar en la Fig. 5.3.5 la diferencia entre una ruta sin y con *lazy loading*. También permite redirecciones, incluida una redirección por *wildcard* que sirve para hacer una página personalizada de error HTTP 404.

```
const routes: Routes = [
  {
    path: 'eager/login',
    component: LoginComponent,
  },
  {
    path: 'lazy/login',
    loadChildren: () =>
      import('./modules/auth/auth.module').then((m) => m.AuthModule),
  },
];
```

Fig. 5.3.5 Enrutamiento sin y con lazy loading, respectivamente. Fuente: Elaboración propia

Una de las características del enrutamiento por angular es que permite hacer *custom matchers*, lo que habilita un enrutamiento más complejo, en caso de ser necesario. De forma predeterminada y sin usar custom matchers angular permite el uso de rutas parametrizadas, rutas que aceptan parámetros que luego son pasados en forma de variable al componente. Siguiendo la Fig. 5.3.6, si usaramos la ruta *book/Cristopher%20Paolini* angular pasaría *Cristopher Paolini* como variable *author* al componente. Del mismo modo que los parámetros por ruta son posibles, también se permiten los *query params*, aunque no se han usado en este desarrollo.

```
{ path: 'book/:author', component: BookAuthorComponent }
```

Fig. 5.3.6 Ejemplo de ruta parametrizada. Fuente: Elaboración propia

5.3.2.1 Seguridad

En cada objeto de ruta se puede definir la propiedad *canActivate* con una clase que implemente *CanActivate*. Si el método canActivate devuelve true entonces la ruta activa el componente o módulo, en caso contrario no lo activa.

```
1
2  const routes: Routes = [{
3      path: 'dashboard',
4      canActivate: [AuthGuard],
5      loadChildren: () => import('./modules/dashboard/dashboard.module')
6        .then(m => m.DashboardModule)
7    }];
8  export class AuthGuard implements CanActivate {
9    constructor(private authService: IAuthService,private router: Router) {}
10   canActivate(route: ActivatedRouteSnapshot,state:RouterStateSnapshot)
11   :Promise<boolean | UrlTree> {
12     return this.authService.isLogged().then(
13       value => {
14         if (value) return true;
15         this.router.navigate(['/login']);
16         return false;
17       },
18       _ => {
19         this.router.navigate(['/login']);
20         return false;
21       }
22     );
23   }
24 }
```

Fig. 5.3.7 Ejemplo de ruta con authGuard. Fuente: Elaboración propia

En el caso del proyecto, se comprueba si el usuario tiene la sesión iniciada, que básicamente comprueba que haya un apitoken y que sea válido, en caso de que no esté la sesión iniciada hace una redirección al login.

5.3.3 Dependency Injection

La inyección de dependencias en angular es muy sencilla de hacer.

1. Crea un componente que servirá como servicio con el decorador *@Injectable*
2. Declara el componente como un *Provider* en el módulo del componente que vaya a inyectarlo
3. Declara un parámetro del tipo del servicio en el constructor.

Al declarar un parámetro en el constructor, se crea un atributo en el componente que se autoasigna. Si se necesita que el atributo sea privado solo se tiene que declarar el parámetro del constructor como privado. Como se puede observar en la Fig. 5.3.8, la llamada al atributo es válido desde la primera línea del constructor.

```
@Injectable({
  providedIn: 'root',
})
export class AuthService {
  constructor() { }
  isLoggedIn(): boolean{
    retun true;
  }
}
export class LoginComponent {
  constructor(private authService: AuthService){
    console.log(this.authService.isLoggedIn()); //valid call ⇒ true
  }
}
```

Fig. 5.3.8 Ejemplo de servicio inyectado en un componente. Fuente: Elaboración propia

5.3.3.1 Mocking

Para facilitar el desarrollo, se han creado servicios de mocking. Lo primero que se ha hecho es crear una interfaz que es lo que los componentes van a implementar, después en la declaración de los servicios se hace uso de la función *isDevMode*, propia de angular, que devuelve false cuando está en producción para asignar una clase u otra a la interfaz. Se puede ver en la Fig. 5.3.9 como se definen los servicios y como se inyectarían en una clase.

```
const providers: [
    {
      provide: IProductsService,
      useClass: isDevMode() ? MockProductsService : ProductsService
    },
    {
      provide: IAuthService,
      useClass: isDevMode() ? MockAuthService : AuthService
    }
  ];
export class LoginComponent{
  constructor(private authService: IAuthService){}
}
```

Fig. 5.3.9 Ejemplo de inyección de dependencias por interfaces según el entorno. Fuente: Elaboración propia

5.3.4 Dashboard

5.3.4.1 Chart.JS

Para hacer las gráficas se ha usado una librería llamada Chart.JS. Chart.JS es una librería open source para javascript que utiliza los canvas nativos de HTML para hacer hasta ocho tipos de gráficas de forma nativa. Los tipos de gráficas que se pueden hacer son de líneas, de barras, de anillos, circulares, de

áreas polares, de radar y de dispersión. Gracias a plugins de la comunidad se añaden gráficas como diagramas de cajas, financieros, geográficos, matrices, diagramas de Venn y Euler, etc.

Chart.JS permite animar todos los cambios entre valores para tener una experiencia de usuario más rica, además lo hace por defecto. También tiene un buen rendimiento ya que gracias a la reducción de datos que tiene, se pueden representar gráficos con más de un millón de puntos. Chart.JS permite también representar distintos tipos de gráficos a la vez en un mismo gráfico, así como mezclar varios datasets al mismo tiempo. Los gráficos son totalmente configurables y se permite la inclusión de cualquier plugin, además se pueden sobrescribir todas las llamadas del ciclo de vida de los gráficos para poder personalizarlos todo lo necesario.

Además de todo esto, gracias a ng2-charts, hay una integración para angular directa, lo que facilita su inclusión.

Se ha creado un componente de gráficos llamado *ChartComponent* que dependiendo de los datos que se le envían, muestra una gráfica distinta, haciendo que toda la lógica sobre que gráfica mostrar esté en el propio componente.

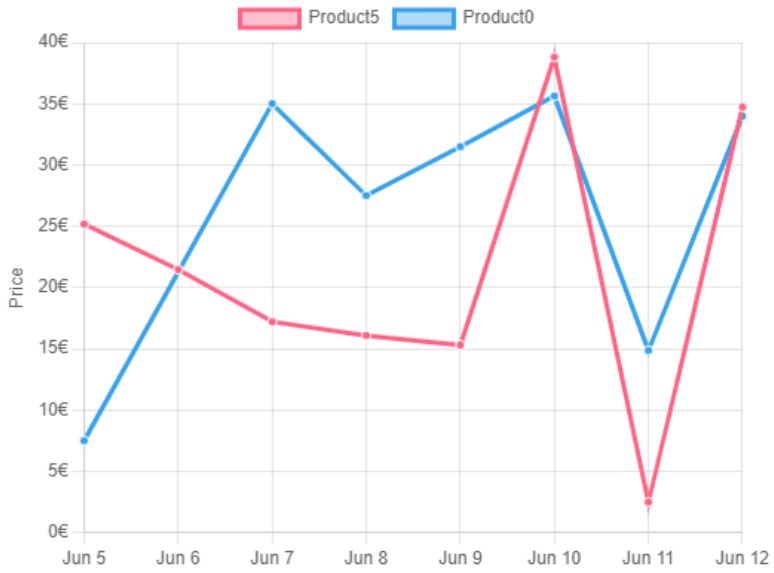


Figure 5.1 Ejemplo de gráfico de comparación de productos. Fuente: Elaboración propia

5.3.5 Testing

Angular está preparado tanto para hacer *unit testing* como *end to end* testing. Aun así, actualmente no se usa ningún tipo de testing.

5.4 Bases de datos

Para este proyecto se han usado dos bases de datos, una relacional PostgreSQL 14 y otra NoSQL

5.4.1 Diseño de la base de datos de scraping.

La base de datos se ha diseñado en conjunto al jefe del equipo de integraciones para que se acerque lo más al negocio posible. En este caso se ha diseñado con la herramienta *Moon Modeler* de Datensen, que además de disponer de una versión freeware todas sus licencias son de compra permanente, algo poco común últimamente.

Como se puede ver en la Fig. 5.4.1, toda la base de datos tiene el centro el producto. El producto consiste de un identificador auto incremental, el nombre del producto, la unidad de medida del producto, una llave foránea al id del retailer y otra al id de la marca del producto y dos campos opcionales que son el código del producto y el ean identificativo.

De todas las clases, la más importante es sin duda la de *product_price* ya que acabaran habiendo millones de filas en esa tabla ya que relaciona un producto con un precio de ese dia. Por ese motivo hay un índice parcial en la columna *date* descendiente. La tabla de *product_image* se compone de únicamente una llave compuesta ya que es posible que un producto tenga múltiples imágenes y que una imagen tenga múltiples productos.

Para el precio se ha creado un tipo compuesto *Price* que consiste de dos columnas, *value* de tipo double y *currency* de tipo text.

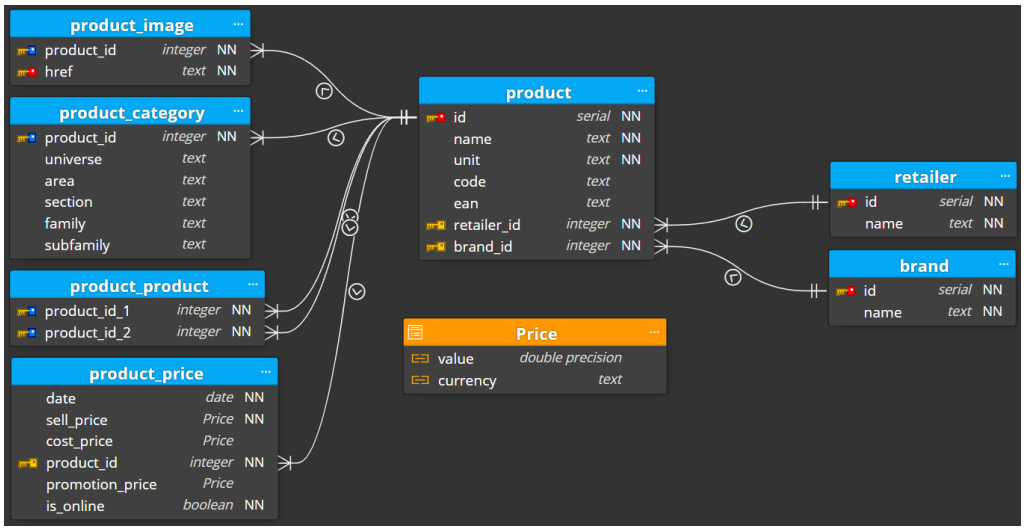


Fig. 5.4.1 diseño de la base de datos de scraping con Moon Modeler. Fuente: Elaboración propia.

6 Conclusiones

Poder hacer un proyecto entero desde el inicio con Clean Architecture ha sido una gran oportunidad para poder ver las flaquezas y los puntos fuertes de esta arquitectura. Uno de los objetivos de este proyecto es valorar si Clean Architecture es una buena arquitectura y vale la pena hacer el cambio en la empresa, y después de casi un año de trabajo puedo decir con confianza que vale mucho la pena. En mi opinión habría que aplicar también la recomendación de que el dominio esté hecho en c++ ya que hace que el dominio, el centro de la arquitectura, esté compartida entre todos los lenguajes que se vayan a usar.

Scala es un lenguaje con muchos problemas, el primero es que muchas de sus versiones son incompatibles entre sí, lo que hace que haya muchas librerías que no se puedan usar a partir de ciertas versiones. El segundo es que Scala no es multiparadigma, sino que es de paradigma funcional, y la mayoría de librerías utilizan Scala como si fuera multiparadigma u orientado a objetos, lo que hace que se pierda el punto principal de Scala, así que no tiene ningún beneficio usar Scala.

7 Futuras ampliaciones

Este proyecto nunca va a acabar de ampliarse, ya que está pensado para expandirse según las necesidades de la empresa. Las futuras ampliaciones que se han acordado con el cliente interno son:

1. Añadir más competidores a los servicios de scraping.
2. Añadir alertas dinámicas para los precios de los productos que los usuarios puedan personalizar en su totalidad.
3. Sistema de notificaciones.
4. Permisos según roles y roles para cada usuario.
5. Añadir más características y distintos tipos de dashboard.

Estas ampliaciones están sujetos a cambio según el cliente interno crea necesario.

8 Bibliografía

[1] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” p. 416, Nov. 1994.

[2] M. Fowler, D. Rice, M. Foemmel, R. Mee, and R. Stafford, “Patterns of Enterprise Application,” *Wesley, Addison*, p. 560, 2002.

[3] L. de Lauretis, “From monolithic architecture to microservices architecture,” *Proceedings - 2019 IEEE 30th International Symposium on Software Reliability Engineering Workshops, ISSREW 2019*, pp. 93–96, Oct. 2019, doi: 10.1109/ISSREW.2019.00050.

[4] R. Chen, S. Li, and Z. Li, “From Monolith to Microservices: A Dataflow-Driven Approach,” 2017, doi: 10.1109/APSEC.2017.53.

[5] N. Dragoni *et al.*, “Microservices: Yesterday, Today, and Tomorrow,” *Present and Ulterior Software Engineering*, pp. 195–216, Nov. 2017, doi: 10.1007/978-3-319-67425-4_12.

[6] T. Cerny, M. J. Donahoo, and M. Trnka, “Contextual Understanding of Microservice Architecture: Current and Future Directions,” 2017, doi: 10.1145/3129676.3129682.

[7] E. Evans, “Domain-Driven Design: Tackling Complexity in the Heart of Software”.

[8] Vaughn. Vernon, *Implementing domain-driven design*. Addision-Wesley, 2013.

[9] R. C. Martin, “Clean Architecture”.

[10] “Pricing Tool SaaS para ecommerce | Minderest.” https://www.minderest.com/es/pricing-tool-saas-ecommerce (accessed Jan. 22, 2022).

[11] “Product Data API para ecommerce | Minderest.” https://www.minderest.com/es/product-data-api-ecommerce (accessed Jan. 22, 2022).

[12] “Flipflow - Monitorización de precios, catálogos y stocks.” https://www.flipflow.io/ (accessed Feb. 05, 2022).

[13] “Business Intelligence - Flipflow - Monitorización de precios, catálogos y stocks.” https://www.flipflow.io/business-intelligence/ (accessed Feb. 05, 2022).

[14] "Most Popular Backend Frameworks – 2012/2021 - New Update - Statistics and Data." https://statisticsanddata.org/data/most-popular-backend-frameworks-2012-2021/ (accessed Apr. 17, 2022).

[15] "Usage Statistics and Market Share of Server-side Programming Languages for Websites, April 2022." https://w3techs.com/technologies/overview/programming_language (accessed Apr. 17, 2022).

[16] "Entity Framework documentation | Microsoft Docs." https://docs.microsoft.com/en-us/ef/ (accessed Apr. 17, 2022).

[17] M. Fourment and M. R. Gillings, "A comparison of common programming languages used in bioinformatics," *BMC Bioinformatics*, vol. 9, no. 1, pp. 1–9, Feb. 2008, doi: 10.1186/1471-2105-9-82/TABLES/1.

[18] "Cross-Platform Mobile App Development: Ionic Framework." https://ionicframework.com/ (accessed Jun. 08, 2022).

[19] "Ionic Native." https://ionicframework.com/native (accessed Jun. 08, 2022).

[20] "dotnet/maui: .NET MAUI is the .NET Multi-platform App UI, a framework for building native device applications spanning mobile, tablet, and desktop." https://github.com/dotnet/maui (accessed Jun. 08, 2022).

[21] "Flutter - Build apps for any screen." https://flutter.dev/ (accessed Jun. 08, 2022).

[22] "Google Graveyard - Killed by Google." https://killedbygoogle.com/ (accessed Jun. 08, 2022).

[23] "Type Erasure in Scala." https://squidarth.com/scala/types/2019/01/11/type-erasure-scala.html (accessed Apr. 17, 2022).

[24] "Source Level | Scala 3 Migration Guide | Scala Documentation." https://docs.scala-lang.org/scala3/guides/migration/compatibility-source.html (accessed Apr. 17, 2022).

[25] "MinZe25/noot-scaffold." https://github.com/MinZe25/noot-scaffold (accessed Jun. 15, 2022).

[26] "MinZe25/scala-mediator." https://github.com/MinZe25/scala-mediator (accessed Jun. 15, 2022).