



# descuentor

Memoria del Proyecto Final del Ciclo Superior de  
Desarrollo de Aplicaciones Multiplataforma

Miguel Alfayate Gallego

# Tabla de contenido

Estudio del problema y análisis del sistema .....	3
Introducción .....	3
Finalidad.....	3
Requisitos .....	4
Servicios a los usuarios. ....	6
Recursos .....	8
Recursos Hardware .....	8
Recursos Software .....	8
Planificación temporal.....	10
Desarrollo y pruebas.....	12
Arquitectura.....	12
Implementación .....	25
Testing .....	38
Conclusiones finales.....	40
Grado de cumplimiento de requisitos.....	40
Mejoras de la aplicación.....	41
Nuevas funcionalidades.....	42
Guías.....	43
Guía de uso.....	43
Guía de instalación .....	47
Referencias bibliográficas .....	49

# Estudio del problema y análisis del sistema

## Introducción

En la actualidad, el comercio electrónico ha experimentado un crecimiento exponencial, transformando la manera en que los consumidores interactúan con las tiendas y realizan sus compras. Sin embargo, la dispersión de productos en diferentes plataformas y la volatilidad de los precios pueden dificultar a los usuarios la toma de decisiones óptimas de compra. En este contexto, nace Descuentor, una aplicación diseñada para mejorar y simplificar la experiencia de compra online de los usuarios.

Descuentor se presenta como una solución integral que permite a los usuarios realizar un seguimiento efectivo de los precios de sus productos favoritos a través de diferentes tiendas online. La aplicación se estructura en cuatro componentes principales:

1. Una extensión de Chrome que actúa como punto de entrada.
2. Una aplicación web que funciona como centro de control.
3. Un sistema automatizado de notificaciones por email.
4. Un servicio de web scraping que monitoriza periódicamente los precios de los productos almacenados.

La arquitectura del sistema se ha diseñado siguiendo principios SOLID y patrones de diseño modernos, lo que facilita su mantenimiento y permite la incorporación de nuevas funcionalidades de manera ágil y estructurada.

## Finalidad

La motivación principal detrás de Descuentor surge de la necesidad de simplificar y optimizar la experiencia de compra online. El proyecto nace con la visión de crear una solución centralizada que permita a los usuarios:

1. **Centralización de la información:** Ofrecer un punto único donde los usuarios puedan almacenar y gestionar sus productos favoritos de diferentes tiendas online homologadas, eliminando la necesidad de mantener múltiples pestañas abiertas o recordar diferentes URLs.
2. **Monitorización automática:** Implementar un sistema que realice un seguimiento continuo de los precios, liberando a los usuarios de la tarea repetitiva de comprobar manualmente los precios de sus productos de interés.
3. **Notificaciones proactivas:** Mantener a los usuarios informados sobre las variaciones de precios en sus productos seleccionados, permitiéndoles aprovechar las mejores ofertas en el momento oportuno.
4. **Experiencia de usuario fluida:** Facilitar la integración con el flujo natural de navegación del usuario mediante una extensión de Chrome, permitiendo añadir productos a la lista de seguimiento con un simple clic mientras navegan por sus tiendas favoritas.

5. **Toma de decisiones informada:** Proporcionar a los usuarios información histórica sobre los precios de los productos, permitiéndoles tomar decisiones de compra más informadas basadas en la evolución de los precios.

La finalidad de la aplicación también incluye implementar una serie de criterios técnicos y de desarrollo, como la utilización de una arquitectura robusta y escalable que permita la incorporación futura de nuevas funcionalidades y la integración con más tiendas online.

Descuentor representa un paso significativo hacia la optimización del comercio electrónico, proporcionando una herramienta que simplifica el seguimiento de precios y permite a los usuarios navegar eficientemente en el complejo panorama de las compras online, permitiéndoles ahorrar tiempo y dinero.

## Requisitos

### *Funcionales*

#### **Gestión de Usuarios**

El sistema debe proporcionar una gestión completa de usuarios que permita el registro mediante correo electrónico y contraseña, asegurando la unicidad de las cuentas creadas. Los usuarios registrados deben tener la capacidad de modificar sus datos personales y gestionar sus preferencias en cualquier momento. Es fundamental implementar un sistema robusto de recuperación de contraseñas que garantice la seguridad de las cuentas.

#### **Sistema de Autenticación**

La autenticación se implementará mediante un sistema basado en tokens que garantice la seguridad de las sesiones de usuario tanto en la interfaz web como en la extensión de Chrome. Los tokens de acceso tendrán un tiempo de expiración definido para mantener la seguridad, complementándose con un sistema de refresh tokens que permita mantener la sesión activa de manera segura. Los usuarios deben tener la capacidad de cerrar sesión en todos sus dispositivos cuando lo consideren necesario, proporcionando control total sobre sus sesiones activas.

#### **Gestión de Productos**

Los usuarios deben poder gestionar su lista de productos de interés de manera intuitiva. El sistema permitirá añadir productos desde las tiendas online homologadas mediante la extensión de Chrome, proporcionando una experiencia fluida de captura de datos. Una aplicación web permitirá mostrar el historial completo de precios de cada producto, permitiendo a los usuarios tomar decisiones informadas. Los usuarios tendrán la capacidad de eliminar o editar productos de su lista de seguimiento, manteniendo su espacio organizado y personalizado.

#### **Sistema de Notificaciones**

El componente de notificaciones debe mantener a los usuarios informados sobre las variaciones de precios relevantes en sus productos seguidos. El sistema enviará notificaciones por correo electrónico cuando se detecten descuentos significativos, permitiendo a los usuarios configurar sus propios umbrales de descuento para recibir alertas personalizadas. Las notificaciones incluirán información detallada sobre el producto y el descuento detectado, agrupándose de manera inteligente para evitar la saturación de la bandeja de entrada del usuario.

#### **Sistema de Scraping**

El sistema de scraping constituye el núcleo de la funcionalidad de seguimiento de precios. Debe ser capaz de extraer información actualizada de los productos de manera automática y periódica desde las tiendas online homologadas. El scraper debe implementar mecanismos robustos para manejar diferentes estructuras de páginas web y ser resistente a cambios en el diseño de las tiendas. Es esencial que el sistema maneje correctamente situaciones como productos agotados, páginas de error y variaciones en el formato de los precios. El scraping debe realizarse de manera respetuosa con los recursos de las tiendas online, implementando delays apropiados entre consultas y respetando las políticas de robots.txt. Además, debe mantener un registro detallado de las actualizaciones de precios para cada producto, permitiendo la construcción de un historial preciso de variaciones de precio.

## **Interfaz API REST**

La API REST, desarrollada con ASP.NET, sirve como columna vertebral de la aplicación, coordinando la comunicación entre todos los componentes. Este servicio gestiona la autenticación y autorización de usuarios, procesa las solicitudes provenientes tanto de la extensión como de la interfaz web, y coordina las operaciones de scraping y notificación. La API garantiza la consistencia de los datos y proporciona una capa de abstracción que facilita la escalabilidad del sistema.

## *No Funcionales*

### **Rendimiento**

El rendimiento del sistema se ha establecido como una prioridad crítica, estableciendo objetivos específicos para cada componente. La API debe mantener tiempos de respuesta inferiores a dos segundos para garantizar una experiencia fluida, mientras que el sistema de scraping debe completar actualizaciones diarias de precios de manera eficiente. La extensión de Chrome debe procesar las solicitudes de manera casi instantánea, con un tiempo máximo de respuesta de un segundo, para mantener una experiencia de usuario sin interrupciones. La infraestructura está diseñada para manejar una base de usuarios concurrentes significativa, con optimizaciones específicas en la base de datos para gestionar eficientemente las consultas frecuentes de precios.

### **Seguridad**

La seguridad se aborda de manera integral en toda la aplicación. El sistema implementa un robusto esquema de autenticación y autorización, utilizando algoritmos de hash seguros para el almacenamiento de contraseñas y tokens de acceso para la gestión de sesiones. La comunicación entre componentes se realiza de manera segura, con tokens transmitidos exclusivamente a través de headers de autorización. Se implementan medidas de protección contra ataques comunes como CSRF, asegurando la integridad de las operaciones del usuario.

### **Usabilidad**

En términos de usabilidad, la aplicación se ha diseñado priorizando la experiencia del usuario. La interfaz web utiliza un diseño responsive que se adapta a diferentes dispositivos y tamaños de pantalla, mientras que la extensión de Chrome se integra de manera natural con el flujo de navegación del usuario. El sistema proporciona retroalimentación clara e inmediata sobre todas las acciones realizadas, utilizando mensajes intuitivos y orientados al usuario. La arquitectura de navegación se ha optimizado para minimizar el número de interacciones necesarias para acceder a cualquier funcionalidad.

### **Escalabilidad**

La escalabilidad del sistema se ha considerado desde las etapas iniciales del diseño. La arquitectura permite el escalado horizontal de los servicios críticos, incluyendo el sistema de scraping que puede distribuirse entre múltiples instancias para manejar cargas crecientes. La base de datos está diseñada para soportar un crecimiento sostenido del volumen de datos sin degradación del rendimiento. La estructura modular del sistema facilita la incorporación de nuevas tiendas online y funcionalidades sin necesidad de modificar el código existente, así como la futura compatibilidad a diferentes navegadores web.

## **Mantenibilidad**

La mantenibilidad del código se garantiza mediante la adherencia estricta a los principios SOLID y la implementación de patrones de diseño establecidos. El sistema mantiene una cobertura de pruebas exhaustiva que facilita la detección temprana de problemas y asegura la calidad del código. La documentación sigue estándares establecidos y se mantiene actualizada, mientras que la arquitectura desacoplada permite realizar modificaciones y mejoras con un impacto mínimo en los componentes existentes.

## **Disponibilidad**

La disponibilidad del sistema se gestiona mediante una infraestructura robusta que garantiza un tiempo de actividad superior al 99.9%. Las actualizaciones y mantenimientos se realizan de manera transparente para los usuarios, sin interrupciones en el servicio. El sistema está diseñado para manejar de manera elegante las posibles interrupciones en servicios externos, implementando estrategias de fallback y recuperación. La monitorización continua del estado del sistema permite la detección y resolución proactiva de problemas potenciales.

## **Servicios a los usuarios.**

Descuentor ofrece una suite integrada de servicios que trabajan en conjunto para proporcionar una experiencia completa de seguimiento de precios. Cada servicio ha sido diseñado específicamente para cubrir una necesidad concreta del usuario.

La **interfaz web**, desarrollada con Blazor WebAssembly Standalone, actúa como el centro de control principal para los usuarios. A través de este portal, los usuarios pueden acceder a una vista completa y organizada de todos sus productos monitorizados. La interfaz presenta los datos históricos de precios mediante gráficas interactivas implementadas con el paquete ApexCharts de .NET, que transforma las listas de datos en visualizaciones dinámicas que facilitan la identificación de tendencias y oportunidades de compra. Los usuarios pueden personalizar sus preferencias de notificación y gestionar su lista de productos, eliminando aquellos que ya no desean seguir.

La **extensión de Chrome** representa el punto de entrada más directo para los usuarios. Esta herramienta se integra de manera natural en el flujo de navegación, detectando automáticamente cuando el usuario está visitando una tienda compatible. Cuando esto ocurre, la extensión permite añadir productos a la lista de seguimiento mediante un simple clic, extrayendo automáticamente la información relevante como el título, precio actual y URL del producto. La extensión proporciona feedback inmediato sobre el estado de cada operación, asegurando una experiencia fluida y sin fricciones.

El **sistema de scraping** funciona como el motor de monitorización automática de la aplicación. Este componente utiliza Playwright para simular la navegación real y extraer los precios actualizados de los productos guardados. El sistema implementa estrategias sofisticadas anti-bloqueo para garantizar la continuidad del servicio y gestiona eficientemente las consultas

mediante procesamiento paralelo. Toda la información recopilada se almacena de manera estructurada en la base de datos, manteniendo un historial completo de las variaciones de precios.

El **sistema de notificaciones** actúa como un asistente personal que mantiene a los usuarios informados sobre las oportunidades de compra. Este servicio está estructurado en dos componentes principales: un servicio de recopilación que identifica y analiza los productos que requieren notificación, y un servicio de envío de correos electrónico que utiliza el servidor SMTP de Mailtrap en modo sandbox. El sistema analiza las variaciones de precios detectadas justo después del proceso de scraping, calcula los porcentajes de descuento y genera correos electrónicos personalizados cuando se identifican ofertas relevantes. Los correos incluyen enlaces directos a los productos con descuento y la información se agrupa de manera inteligente para evitar el envío excesivo de notificaciones.

La interacción entre estos servicios sigue un flujo natural y automatizado. Cuando un usuario descubre un producto de interés mientras navega, la extensión de Chrome facilita su adición al sistema. La API procesa esta solicitud y almacena la información del producto. A partir de ese momento, el sistema de scraping comienza a monitorizar periódicamente el precio. Cuando se detecta un descuento significativo, el sistema de notificaciones alerta al usuario, quien puede entonces consultar los detalles completos a través de la interfaz web.

El verdadero valor añadido de esta integración radica en la automatización completa del proceso de seguimiento de precios. Los usuarios solo necesitan identificar los productos que les interesan, y el sistema se encarga automáticamente del resto. Las notificaciones se envían de manera oportuna y relevante, mientras que la interfaz web proporciona una visión completa y detallada del seguimiento. La extensión simplifica significativamente el proceso de añadir nuevos productos, eliminando la fricción típicamente asociada con este tipo de tareas.

# Recursos

## Recursos Hardware

Para el desarrollo de la aplicación se ha utilizado un MacBook Pro de 15 pulgadas (2018), equipado con un procesador Intel Core i7 de 6 núcleos a 2,2 GHz y 16GB de RAM. Esta configuración ha proporcionado la potencia de procesamiento necesaria para ejecutar simultáneamente los entornos de desarrollo, contenedores Docker y realizar las tareas de depuración de manera fluida.

La infraestructura de servidor se sustenta en una máquina con procesador Xeon de 10 núcleos y 16GB de RAM, que ejecuta Proxmox como hipervisor para la gestión de máquinas virtuales. Esta configuración proporciona un entorno robusto y flexible para alojar la base de datos y los servicios necesarios para el desarrollo y pruebas de la aplicación.

## Recursos Software

### *Entornos de Desarrollo*

El desarrollo se ha realizado principalmente utilizando dos IDEs complementarios:

- **JetBrains Rider** (última versión): Un potente IDE específicamente diseñado para el desarrollo en .NET, que ha sido fundamental para el desarrollo del backend y la aplicación web Blazor. Su integración con .NET y sus capacidades avanzadas de depuración han facilitado significativamente el desarrollo.
- **Visual Studio Code**: Utilizado principalmente para el desarrollo de la extensión de Chrome, proporcionando un entorno ligero pero potente para el desarrollo web.

Ambos entornos de desarrollo se han complementado con **GitHub Copilot**, una herramienta de IA que ha asistido en la resolución de problemas y la escritura de código.

### *Control de Versiones*

El proyecto utiliza **GitHub** como sistema de control de versiones, sirviendo no solo como repositorio y backup del código, sino también como plataforma para mostrar el trabajo realizado. El repositorio se mantiene público lo que permite que sirva como portfolio para potenciales empleadores.

### *Base de Datos y Contenedores*

- **PostgreSQL** (última versión): Seleccionado como sistema de gestión de base de datos por su robustez, capacidad de manejo de datos relacionales y excelente rendimiento en aplicaciones web.
- **Docker** (versión 26.1.4): Utilizado en dos contextos diferentes:
  - Durante el desarrollo, para proporcionar un entorno consistente y aislado para la base de datos PostgreSQL.
  - Como plataforma de despliegue, facilitando la distribución y ejecución de la aplicación en diferentes entornos.



- **Docker Compose:** Empleado para la orquestación de contenedores, simplificando la gestión de los diferentes servicios de la aplicación.

### *Frameworks y Paquetes*

El proyecto se sustenta en varios paquetes y frameworks clave de .NET:

- **Entity Framework Core:** Como ORM (Object-Relational Mapping) para la gestión de la base de datos.
- **Blazored.LocalStorage:** Para el manejo del almacenamiento local en el navegador.
- **ApexCharts:** Utilizado para la visualización de datos y gráficos.
- **MediatR:** Implementación del patrón mediator para la comunicación entre componentes. En la sección de "Arquitectura" se explicará en qué consiste este patrón de arquitectura.
- **Identity:** Para la gestión de la autenticación de usuarios

### *Frontend Blazor*

Utilizamos para el Frontend **Blazor WebAssembly** en su variante Standalone. Gracias a ella, la aplicación se ejecuta completamente en el navegador del cliente, sin necesidad de un servidor web ASP.NET Core para el renderizado.

WebAssembly (abreviado como WASM) es un formato de código binario de bajo nivel diseñado para ser ejecutado en navegadores web. Actúa como una especie de "máquina virtual" que permite ejecutar código compilado en el navegador a velocidades cercanas al código nativo. Esto significa que toda la aplicación .NET, incluyendo el runtime de .NET, se descarga y ejecuta directamente en el navegador del usuario.

### *Servicios Externos*

**Mailtrap:** Utilizado como sandbox SMTP para el desarrollo y prueba del sistema de notificaciones por email, permitiendo verificar el funcionamiento del sistema de correos sin enviar mensajes reales a los usuarios durante la fase de desarrollo.

### *Navegador*

El desarrollo y pruebas de la extensión se han realizado exclusivamente en **Google Chrome** (última versión), asegurando la compatibilidad con este navegador como objetivo principal del desarrollo.

### *Pruebas API*

Para el desarrollo de la API he utilizado **Insomnia** que es una herramienta parecida a Postman que simplifica la interacción con servicios web y API para probarlos y depurarlos.

### *Gestor de bases de datos*

Para gestionar y administrar la base de datos PostgreSQL se ha utilizado **Navicat**, una herramienta visual multiplataforma que proporciona una interfaz gráfica para la administración, desarrollo y monitorización de bases de datos.

## Planificación temporal

El desarrollo de Descuentor se llevó a cabo durante un periodo de 7 semanas, estructurado en fases bien diferenciadas que permitieron una evolución controlada del proyecto.

### *Fase de Investigación y Diseño (1 semana - Diciembre 2024)*

Durante esta fase inicial se establecieron las bases del proyecto:

- Análisis de soluciones existentes en el mercado
- Definición de la arquitectura del sistema
- Diseño de la base de datos
- Selección de tecnologías y frameworks

### *Fase de Desarrollo Core (3 semanas - Diciembre 2024)*

Esta fase constituyó el grueso del desarrollo, centrándose en:

- Implementación de la base de datos y sistema de usuarios
- Desarrollo de la API REST
- Implementación del servicio de scraping
- Sistema de autenticación y autorización

### *Fase de Desarrollo Frontend (2 semanas – Diciembre-Enero 2025)*

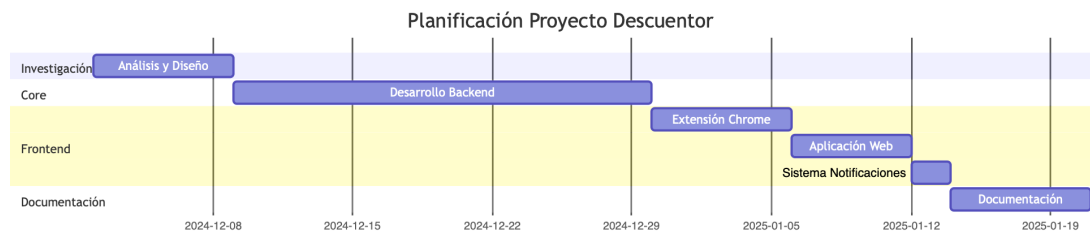
Se desarrollaron los componentes de interfaz de usuario:

- Extensión de Chrome para captura de productos (1 semana)
- Aplicación web Blazor para gestión de productos (6 días)
- Sistema de notificaciones por email (2 días)

### *Fase de Documentación (1 semana - Enero 2025)*

Fase final dedicada a:

- Documentación técnica de la API
- Creación de guías de usuario
- Preparación de la memoria del proyecto
- Organización del repositorio y documentación del código



*Figura 1. Gráfico Gantt de la planificación temporal del proyecto*

Esta estructura temporal permitió un desarrollo iterativo y controlado del proyecto, con hitos claros que facilitaron el seguimiento del progreso y la detección temprana de posibles desviaciones del plan original.

# Desarrollo y pruebas

## Arquitectura

### *Visión General*

#### Principios SOLID

En el desarrollo de Descuentor, se han aplicado los principios SOLID como base para crear una arquitectura robusta y mantenible. Estos principios, introducidos por Robert C. Martin, constituyen un conjunto de directrices que promueven el desarrollo de software de calidad.

- El **principio de Responsabilidad Única** (Single Responsibility Principle) establece que cada clase debe tener una única razón para cambiar. En Descuentor, este principio se refleja claramente en servicios como ScrapingService y NotificacionDescuentosService, donde cada uno maneja una funcionalidad específica sin mezclar responsabilidades.
- El **principio de Abierto/Cerrado** (Open/Closed Principle) sugiere que las entidades de software deben estar abiertas para su extensión pero cerradas para su modificación. La implementación de TiendaOnlineFactory demuestra este principio, permitiendo añadir nuevas tiendas sin modificar el código existente, aunque se podría mejorar extrayendo los selectores CSS a un archivo de configuración externo.
- La **Sustitución de Liskov** (Liskov Substitution Principle) establece que los objetos de una clase derivada deben poder sustituir a los objetos de la clase base sin alterar el comportamiento del programa.
- El **principio de Segregación de Interfaces** (Interface Segregation Principle) sugiere que los clientes no deberían depender de interfaces que no utilizan. En el proyecto, las interfaces como IEnvioEmailService y ITiendaOnlineFactory están bien definidas y cohesivas.
- Finalmente, el **principio de Inversión de Dependencias** (Dependency Inversion Principle) establece que los módulos de alto nivel no deberían depender de módulos de bajo nivel, sino que ambos deberían depender de abstracciones. Este principio se implementa efectivamente a través del sistema de inyección de dependencias en toda la aplicación, particularmente visible en los controladores y servicios.

La aplicación de estos principios ha resultado en un código más mantenible y flexible, aunque hay áreas donde la adherencia podría mejorarse, como en la gestión de configuraciones hardcodedas en algunas clases y en la granularidad de ciertas interfaces. Estas mejoras están identificadas como parte del trabajo futuro en el desarrollo de la aplicación.

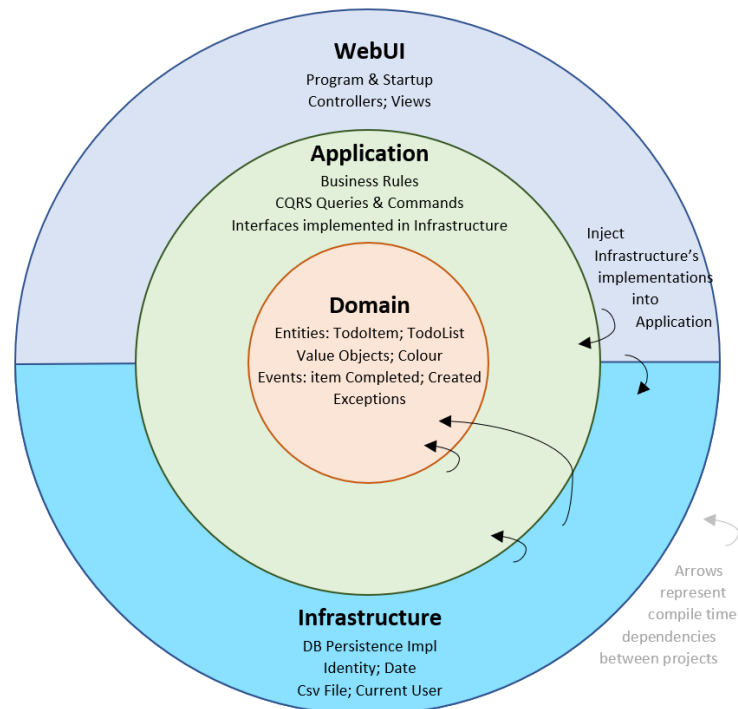


Figura 2. Diagrama de la arquitectura Clean. [Enlace](#)

Clean Architecture, propuesta por Robert C. Martin, es un enfoque de diseño de software que busca la separación de preocupaciones mediante la organización del código en capas concéntricas. En el núcleo se encuentran las entidades y reglas de negocio, mientras que los detalles de implementación y frameworks se mantienen en las capas exteriores. Este diseño garantiza que los cambios en aspectos externos, como la interfaz de usuario o la base de datos, no afecten a la lógica central del negocio.

En Descuentor, la implementación de Clean Architecture se materializa en cuatro capas principales:

- La **capa de Dominio** constituye el núcleo de la aplicación, conteniendo las entidades como *Producto*, *HistorialPrecio* y *TiendaOnline*, así como las interfaces que definen los contratos fundamentales del sistema. Las interfaces en esta capa son cruciales porque permiten la independencia de la lógica de negocio respecto a los detalles de implementación. Esta capa no tiene dependencias externas, manteniendo la pureza del dominio del negocio.
- La **capa de Aplicación** orquesta el flujo de datos y la lógica de negocio, implementando los casos de uso específicos de la aplicación. En Descuentor, esta capa contiene los servicios que coordinan las operaciones de scraping, notificaciones y gestión de productos, utilizando el patrón Mediator para gestionar la comunicación entre componentes. Los DTOs (Data Transfer Objects) en esta capa facilitan la transferencia de datos entre las diferentes partes del sistema.

- La **capa de Infraestructura** maneja los detalles técnicos y la implementación de las interfaces definidas en el dominio. Aquí se encuentran las implementaciones concretas de servicios como *ScrapingService* y *NotificacionDescuentosService*, así como la configuración de Entity Framework para la persistencia de datos. Esta capa es la única que conoce los detalles específicos de tecnologías y frameworks externos.
- Finalmente, la **capa de API** actúa como la interfaz de presentación, exponiendo los endpoints necesarios para interactuar con la aplicación. Los controladores en esta capa son simples y delgados, delegando toda la lógica de negocio a la capa de Aplicación a través de comandos y consultas mediados por MediatR.

Esta arquitectura ha proporcionado beneficios tangibles en el desarrollo de Descuentor. La separación clara de responsabilidades ha facilitado la implementación de pruebas unitarias, especialmente en las capas de Dominio y Aplicación. La independencia entre capas ha permitido realizar cambios significativos en la infraestructura, como la implementación del sistema de scraping, sin afectar a la lógica de negocio central. Además, la estructura organizada facilita la incorporación de nuevas funcionalidades y la identificación y resolución de problemas.

### Mediator

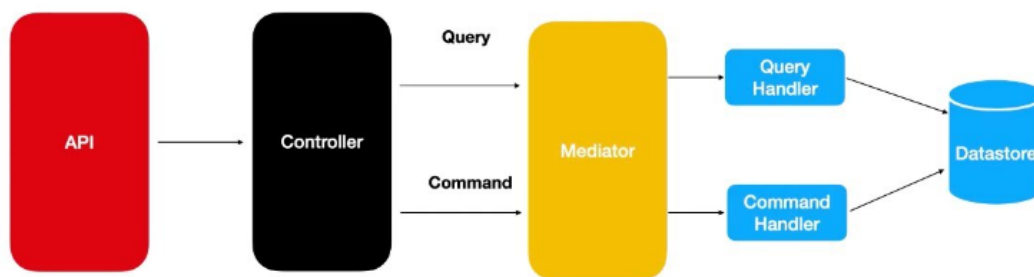


Figura 3. Diagrama de flujo del patrón Mediator. [Enlace](#)

El patrón Mediator es un patrón de diseño de comportamiento que promueve el bajo acoplamiento entre componentes de un sistema, introduciendo un objeto mediador que encapsula la comunicación entre diferentes objetos.

La implementación del patrón Mediator en Descuentor se materializa a través de un sistema de comandos (commands) y consultas (queries). Cada operación significativa en la aplicación se representa como un comando o una consulta, que se envía a través del mediador. Por ejemplo, cuando un usuario desea crear un nuevo producto, el controlador de la API no interactúa directamente con los servicios de la aplicación, sino que crea y envía un *CrearProductoCommand* al mediador, que se encarga de dirigirlo al manejador apropiado.

Los manejadores de comandos y consultas en Descuentor están diseñados para ser simples y enfocados, siguiendo el principio de responsabilidad única. Cada manejador implementa la interfaz *IRequestHandler<TRequest, TResponse>* de MediatR y contiene la lógica necesaria para procesar un tipo específico de comando o consulta. Esta estructura ha permitido mantener una separación clara de responsabilidades y ha facilitado la implementación de nuevas funcionalidades sin modificar el código existente.

El uso del patrón Mediador ha proporcionado beneficios significativos en la arquitectura de Descuentor. Ha simplificado la comunicación entre componentes, eliminando dependencias directas entre ellos y facilitará la implementación de funcionalidades transversales como logging y validación. Además, ha mejorado la testabilidad del código, ya que cada manejador puede probarse de forma aislada.

La flexibilidad proporcionada por este patrón ha sido especialmente valiosa en el manejo de operaciones asíncronas, como el scraping de precios y el envío de notificaciones. El mediador gestiona eficientemente estas operaciones, permitiendo que la aplicación mantenga su capacidad de respuesta incluso durante tareas que requieren un procesamiento significativo.

## CQRS

El patrón CQRS (Command Query Responsibility Segregation) es un principio arquitectónico que separa las operaciones de lectura (Queries) de las operaciones de escritura (Commands) en una aplicación. Esta separación permite optimizar cada tipo de operación de manera independiente, reconociendo que las necesidades y características de las lecturas y escrituras son fundamentalmente diferentes en la mayoría de las aplicaciones.

En Descuentor, CQRS se implementa en conjunto con el patrón Mediador a través de MediatR, creando una clara separación entre los comandos que modifican el estado del sistema y las consultas que recuperan información. La estructura del proyecto refleja esta separación en la capa de Aplicación, donde las operaciones se organizan en carpetas Commands y Queries, cada una con sus propios modelos y manejadores específicos:

- Los **Commands** en Descuentor representan operaciones que modifican el estado del sistema, como *CrearProductoCommand* para añadir nuevos productos o *ActualizarPrecioCommand* para actualizar los precios tras el scraping. Estos comandos son clases inmutables que contienen todos los datos necesarios para realizar la operación, y sus manejadores (handlers) correspondientes implementan la lógica de negocio necesaria para ejecutar los cambios en el sistema.
- Por otro lado, las **Queries** se utilizan para recuperar datos sin modificar el estado del sistema. Por ejemplo, *ObtenerProductosPaginadosQuery* permite recuperar productos con paginación, mientras que *ObtenerHistorialPreciosQuery* obtiene el histórico de precios de un producto específico. Los manejadores de consultas están optimizados para operaciones de lectura, utilizando proyecciones y consultas específicas que mejoran el rendimiento.

## Factory

El patrón Factory es un patrón de diseño creacional que proporciona una interfaz para crear objetos en una superclase, permitiendo a las subclases alterar el tipo de objetos que se crean. Este patrón es particularmente útil cuando se necesita crear objetos sin exponer la lógica de creación directamente al cliente.

En Descuentor, el patrón Factory se implementa únicamente en la clase *TiendaOnlineFactory*, que se encarga de proporcionar los selectores CSS necesarios para extraer información de diferentes tiendas online. Esta implementación centraliza la lógica de identificación y configuración de los selectores específicos de cada tienda, permitiendo que el resto del sistema trabaje con una interfaz unificada sin preocuparse por los detalles específicos de cada tienda.

La configuración podría moverse a un archivo de configuración externo o una base de datos en futuras iteraciones, permitiendo actualizaciones sin necesidad de modificar el código.

### *Repositorio*

El patrón Repositorio actúa como una capa de abstracción entre la lógica de negocio y la capa de persistencia de datos, encapsulando la lógica necesaria para acceder a las fuentes de datos (en este caso a la base de datos PostgreSQL). Este patrón permite trabajar con objetos de la capa de dominio de manera consistente, sin exponer los detalles de la base de datos subyacente, proporcionando una visión más orientada a objetos de la capa de persistencia.

En Descuentor, el patrón Repositorio se implementa mediante interfaces definidas en la capa de Dominio y sus implementaciones concretas en la capa de Infraestructura. Por ejemplo, la interfaz *IUsuarioRepository* define los contratos para las operaciones relacionadas con los usuarios, mientras que *UsuarioRepository* proporciona la implementación concreta utilizando Entity Framework Core.

```
public interface IUsuarioRepository
{
    Task<Usuario?> ObtenerUsuarioPorId(int id);
    Task<List<Usuario>> ObtenerUsuarios();
    Task<Usuario> CrearUsuario(Usuario usuario);
    Task ActualizarUsuario(Usuario usuario);
    Task EliminarUsuario(int id);
}

public class UsuarioRepository : IUsuarioRepository
{
    private readonly ApplicationDbContext _context;

    public UsuarioRepository(ApplicationDbContext context)
    {
        _context = context;
    }

    public async Task<Usuario?> ObtenerUsuarioPorId(int id)
    {
        return await _context.Usuarios
            .Include(u => u.Productos)
            .FirstOrDefaultAsync(u => u.Id == id);
    }
    // Implementación de otros métodos...
}
```

La implementación del patrón Repositorio en Descuentor ha proporcionado beneficios significativos en términos de mantenibilidad y flexibilidad. Los repositorios encapsulan la lógica de consulta compleja, como las inclusiones de entidades relacionadas y las transformaciones de datos, manteniendo los controladores y servicios de aplicación limpios y enfocados en la lógica de negocio. Además, facilita la implementación de pruebas unitarias al permitir la sustitución de repositorios reales por implementaciones simuladas.

Un aspecto particular de la implementación en Descuentor es la integración con Entity Framework Core. Los repositorios aprovechan las características del ORM como el seguimiento de cambios y las consultas LINQ, mientras mantienen estos detalles técnicos aislados del resto de la aplicación.



Esta encapsulación ha sido especialmente útil en operaciones complejas como el seguimiento de precios, donde se requieren consultas elaboradas y actualizaciones en lote.

Es importante mencionar que la implementación actual podría mejorarse. Por ejemplo, se podría implementar un Unit of Work para gestionar transacciones y asegurar la consistencia de los datos en operaciones que afectan a múltiples repositorios. También se podría considerar la implementación de un repositorio genérico para operaciones comunes, reduciendo la duplicación de código entre diferentes repositorios. Estas mejoras están contempladas en el trabajo futuro de la aplicación.

### *Inyección de Dependencias*

La Inyección de Dependencias (DI) es un patrón de diseño que implementa el principio de Inversión de Dependencias (el último principio SOLID). En lugar de que las clases creen o busquen sus dependencias, estas son "inyectadas" desde el exterior. En Descuentor, este patrón se implementa utilizando el contenedor de DI nativo de ASP.NET Core.

La principal ventaja de este enfoque es que permite un bajo acoplamiento entre componentes, ya que las clases no necesitan conocer los detalles de implementación de sus dependencias. Esto no solo hace que el código sea más mantenible y testeable, sino que también facilita el cambio de implementaciones en tiempo de ejecución. Por ejemplo, durante el desarrollo podemos usar una implementación simulada de un servicio de email, mientras que en producción usamos el servicio real, todo sin modificar el código que utiliza dicho servicio.

La aplicación utiliza este patrón extensivamente en todas sus capas, registrando las interfaces y sus implementaciones en el contenedor de servicios durante el inicio de la aplicación. Por ejemplo, interfaces como *IScrapingService* o *ITiendaOnlineFactory* se registran con sus implementaciones concretas, permitiendo que los controladores y servicios trabajen con abstracciones en lugar de implementaciones específicas. Esto facilita el testing, mejora la modularidad y permite cambiar implementaciones sin modificar el código que las utiliza.

### *Modelos y Diagramas*

A continuación vamos a exponer los diversos diagramas específicos de mi aplicación. Estos diagramas ayudan a visualizar la estructura, las relaciones y el flujo de datos dentro del sistema.

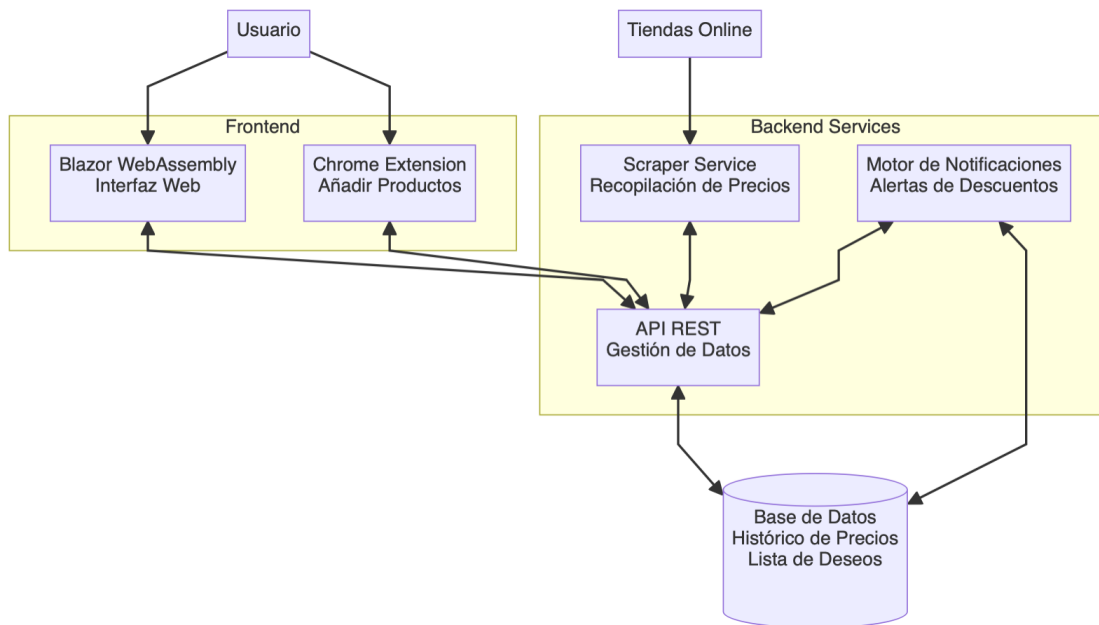


Figura 4. Diagrama de la arquitectura del sistema

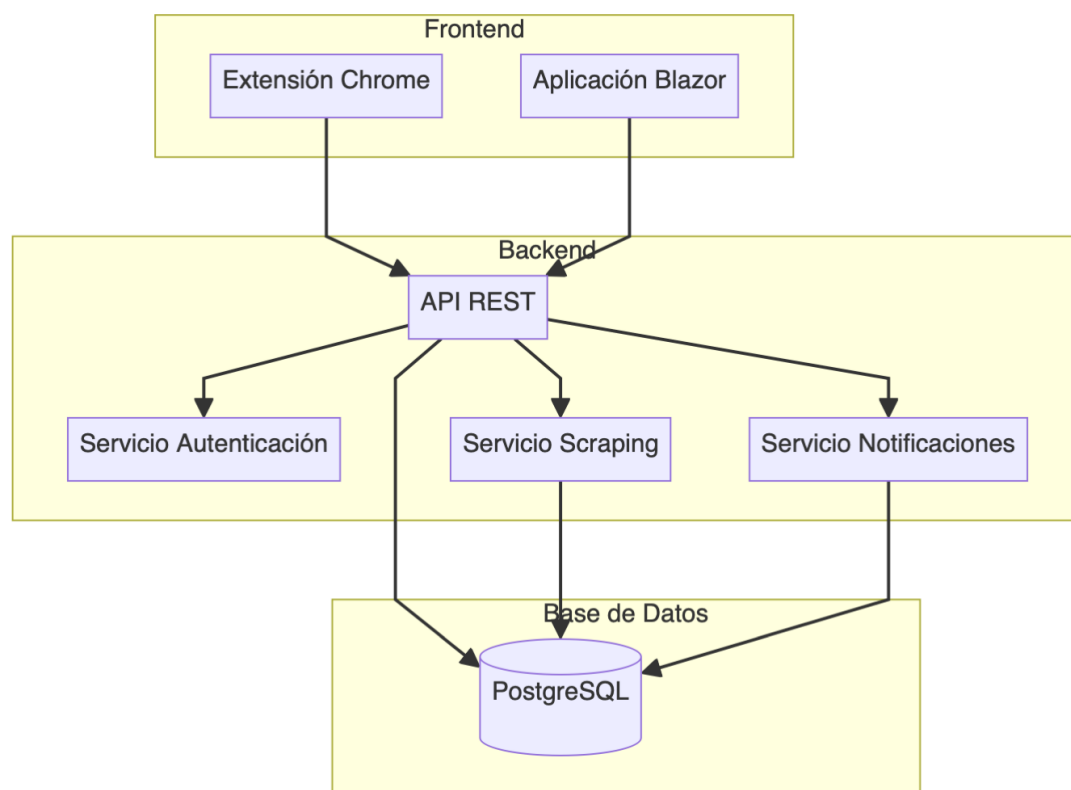


Figura 5. Diagrama de Componentes

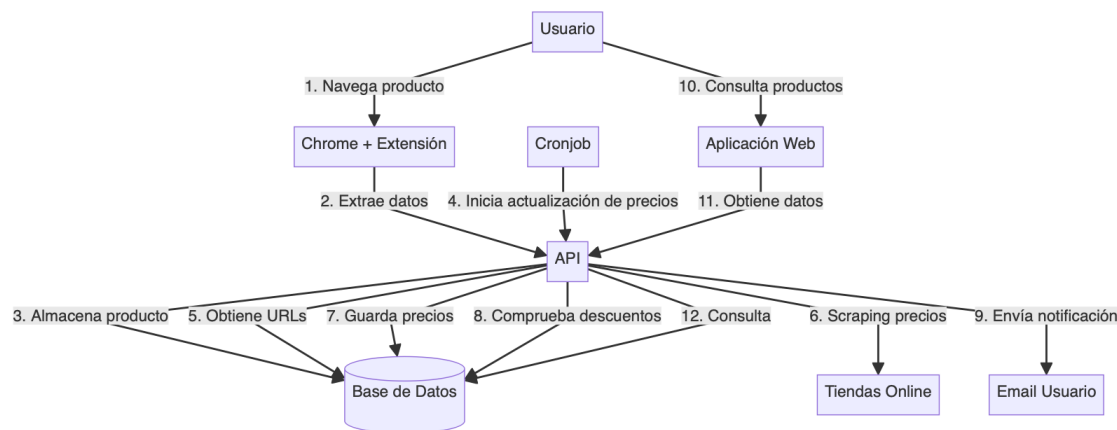


Figura 6. Diagrama de Flujo de Datos

## Base de Datos

### Entity Framework Core

Entity Framework Core (EF Core) es un Object-Relational Mapper (ORM) moderno desarrollado por Microsoft que permite a los desarrolladores trabajar con bases de datos utilizando objetos .NET. En Descuentor, se ha implementado utilizando el enfoque Code First, donde el modelo de datos se define primero en código C# y luego EF Core se encarga de crear y mantener la base de datos PostgreSQL con la que está vinculada.

El enfoque Code First en Descuentor comienza con la definición de las entidades del dominio como clases C#. Estas clases representan las tablas de la base de datos y sus relaciones. Por ejemplo, la entidad *Producto* incluye propiedades como *Nombre*, *Url* y *PrecioInicial*, así como navegación a otras entidades relacionadas como *HistorialPrecios*. Las relaciones entre entidades se establecen mediante propiedades de navegación, permitiendo a EF Core crear las claves foráneas y restricciones necesarias en la base de datos.

```

public class Producto
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Url { get; set; }
    public decimal PrecioActual { get; set; }
    public decimal PrecioInicial { get; set; }
    public DateTime FechaCreacion { get; set; }
    public int UsuarioId { get; set; }
    public Usuario Usuario { get; set; }
    public List<HistorialPrecio> HistorialPrecios { get; set; }
}

```

La configuración de las entidades se realiza en el *ApplicationDbContext*, donde se definen las relaciones, índices y otras restricciones utilizando el API Fluente de EF Core. Esta configuración permite un control preciso sobre cómo se mapean las entidades a las tablas de la base de datos, incluyendo la definición de claves primarias, relaciones uno a muchos y muchos a muchos, y la configuración de propiedades requeridas.

```

public class ApplicationDbContext : IdentityDbContext<UsuarioAplicacion, IdentityRole<int>, int>
{
    public DbSet<Producto> Productos { get; set; }
}

```

```

public DbSet<HistorialPrecio> HistorialPrecios { get; set; }

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    base.OnModelCreating(modelBuilder);

    modelBuilder.Entity<Producto>()
        .HasOne(p => p.Usuario)
        .WithMany(u => u.Productos)
        .HasForeignKey(p => p.UsuarioId);

    modelBuilder.Entity<HistorialPrecio>()
        .HasOne(h => h.Producto)
        .WithMany(p => p.HistorialPrecios)
        .HasForeignKey(h => h.ProductoId);
}
}

```

El uso de EF Core con el enfoque Code First ha proporcionado varios beneficios en el desarrollo de Descuentor, incluyendo un control preciso sobre el esquema de la base de datos, la capacidad de versionar los cambios en el esquema junto con el código fuente, y una forma orientada a objetos de trabajar con los datos de la aplicación. Además, facilita la realización de pruebas unitarias al permitir el uso de bases de datos en memoria para pruebas.

## Identity

ASP.NET Core Identity es un sistema de membresía que proporciona funcionalidades de autenticación y autorización para aplicaciones web. En Descuentor, Identity se integra con Entity Framework Core para gestionar usuarios, roles y claims, extendiendo sus capacidades predeterminadas para adaptarse a las necesidades específicas de la aplicación.

La implementación comienza con la clase `UsuarioAplicacion`, que hereda de `IdentityUser<int>` para personalizar el tipo de dato de la clave primaria y añadir propiedades adicionales necesarias para la aplicación:

```

public class UsuarioAplicacion : IdentityUser<int>
{
    public string? Nombre { get; set; }
    public string? Apellidos { get; set; }
    public DateTime FechaCreacion { get; set; }
    public virtual ICollection<Producto> Productos { get; set; }
}

```

El `ApplicationDbContext` hereda de `IdentityDbContext` para incorporar las tablas necesarias para Identity, mientras mantiene la capacidad de configurar entidades personalizadas:

```

public class ApplicationDbContext : IdentityDbContext<UsuarioAplicacion, IdentityRole<int>, int>
{
    // Resto del código
}

```

Identity crea automáticamente las tablas necesarias para la gestión de usuarios (`AspNetUsers`), roles (`AspNetRoles`), claims (`AspNetUserClaims`, `AspNetRoleClaims`) y otras tablas relacionadas. La integración con Entity Framework Core permite una gestión seamless de las relaciones entre las entidades de Identity y las entidades personalizadas de la aplicación, como la relación entre usuarios y productos.

Esta implementación se complementa con un sistema de roles predefinidos que se establecen durante la inicialización de la base de datos. El sistema incluye dos roles principales: "admin", que tiene acceso a funcionalidades especiales como la actualización masiva de precios y la gestión del sistema, y "usuario", que representa a los usuarios estándar de la aplicación con acceso a las funcionalidades básicas de seguimiento de productos. Estos roles se utilizan en conjunto con los atributos de autorización en los controladores para garantizar que cada endpoint solo sea accesible por usuarios con los permisos adecuados.

### Estructura y relaciones

La siguiente imagen representa el esquema completo de la base de datos de Descuentor, donde se puede apreciar claramente la diferenciación entre dos grupos de tablas. En rojo se muestran las tablas core de la aplicación, que implementan la lógica de negocio principal relacionada con el seguimiento de productos y precios. En azul se visualizan las tablas generadas automáticamente por ASP.NET Core Identity, que gestionan la autenticación y autorización del sistema.



## Migraciones

Las Migraciones en Entity Framework Core son un sistema de control de versiones para el esquema de la base de datos, permitiendo evolucionar el modelo de datos de manera controlada y manteniendo un historial de cambios.

En Descuentor, el proceso de migraciones se gestiona mediante comandos de la CLI de dotnet. Cuando se realizan cambios en las entidades o en la configuración del modelo, se genera una nueva migración usando el comando `dotnet ef migrations add NombreMigracion`. Este comando crea una clase que contiene dos métodos principales: `Up()` para aplicar los cambios y `Down()` para revertirlos.

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Productos",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                Nombre = table.Column<string>(nullable: false),
                Url = table.Column<string>(nullable: false),
                // Otras columnas...
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(name: "Productos");
    }
}
```

Para aplicar las migraciones a la base de datos se utiliza el comando `dotnet ef database update`, que ejecuta las migraciones pendientes en orden cronológico. Este sistema permite mantener sincronizado el esquema de la base de datos con el modelo de datos de la aplicación, facilitando el desarrollo colaborativo y el control de versiones del esquema.

## API Endpoints

A continuación voy a listar brevemente los endpoint que implementa el backend de la aplicación. Estos se dividen en los propios de la aplicación Descuentor, es decir, los necesarios para el ejercicio de las funciones core de la aplicación, y en los que utiliza Identity para autenticar a los usuarios.

### Endpoints de Descuentor

#### Controlador ActualizarPrecios

1. `POST /api/actualizarprecios`
  - No recibe ningún dato
  - Ejecuta los cuatro procesos necesarios para la actualización de precios. Extraer de la base de datos las URLs de los productos, lanzar el scraper para todos esos

productos y recuperar los precios actualizados, añadir los precios a la tabla HistorialPrecios y, por último, lanzar el servicio de notificación a los usuarios.

- Este endpoint solo pueden acceder a él los usuarios con rol "Admin". Por lo que el Bearer token tiene que ser el propio de estos usuarios.

## **Controlador Productos**

### **1. *POST /api/productos***

- Recibe los datos de un producto para darlo de alta.
- Este endpoint es utilizado por la extensión de Chrome.
- Es necesario que la llamada al endpoint vaya con el Bearer token propio de los usuarios con rol "usuario".

### **2. *GET /api/productos***

- Recibe datos de paginación y ordenamiento por campo en la query de la URL.
- Devuelve la lista de productos paginada y ordenada según los criterios planteados por un usuario específico.
- Es necesario estar autenticado con token de rol "usuario" para llamar a este endpoint.

### **3. *GET /api/productos/{id}***

- No recibe datos pero devuelve los datos de productos para un usuario con el historial de precios de los últimos 50 registros de esta tabla.
- Es necesario registrarse con Bearer token de "usuario"

### **4. *PUT /api/productos/{id}***

- Recibe los datos de los campos que se van a actualizar del producto.
- Es necesario el Bearer token de "usuario".

### **5. *DELETE /api/productos/{id}***

- No recibe ningún dato. Simplemente elimina el producto cuyo id aparece en la URL.
- Es necesario el Bearer token de "usuario".

## **Controlador Usuarios**

### **1. *GET /api/usuarios***

- No recibe ningún dato. Simplemente devuelve los datos de un usuario.
- Este endpoint es utilizado por la aplicación web para cargar los datos en el formulario de edición.

### **2. *POST /api/usuarios***



- Este endpoint sobreescribe, en cierto sentido, el endpoint que implementa el API de Identity */identity/register*.
- Recibe los datos de email y contraseña para dar de alta un usuario. La diferencia con el endpoint propio de Identity es que también le añade el rol de "usuario".

### 3. *PUT /api/usuarios*

- Recibe los datos para modificar datos de un usuario.
- Es necesario que la llamada al Endpoint vaya con Bearer token propio de los usuarios con rol "usuario".

## Endpoints de Identity

De los endpoints que implementa la plataforma Identity solo vamos a utilizar los siguientes:

### 1. *POST /identity/login*

- Para la autenticación de usuarios
- Recibe credenciales y devuelve token de acceso

### 2. *POST /identity/refresh*

- Renovación del token de acceso usando refresh token

### 3. *GET /identity/manage/info*

- Para recibir el email del usuario registrado.
- Este endpoint es utilizado por la aplicación web para comprobar que el usuario está logueado y qué usuario es.

## Implementación

A continuación, se detalla la implementación de cada capa de la arquitectura Clean, comenzando por la capa de Dominio que constituye el núcleo de la aplicación, seguida por la capa de Aplicación que orquesta la lógica de negocio, la capa de Infraestructura que proporciona implementaciones concretas de las interfaces definidas, y finalmente las capas de presentación que incluyen la API REST, la aplicación Blazor WebAssembly y la extensión de Chrome.

### *Capa de Dominio*

La capa de dominio es el núcleo de la aplicación en la arquitectura Clean. Contiene las entidades y las interfaces que definen la lógica de negocio y las reglas del dominio. Esta capa está definida dentro de la solución "Descuentor" en el proyecto "Descuentor.Dominio". A continuación, se detallan las entidades y las interfaces que forman parte de esta capa.

### Entidades

Las entidades son objetos del dominio que contienen datos y lógica de negocio. A continuación, se describen las entidades presentes en la carpeta *Entidades*:

#### 1. Configuración:

- Representa una configuración general del sistema. Estas opciones de configuración se usarían para proporcionar al usuario personalización sobre todo de la aplicación web (opciones de idioma, región horaria, ...).
- Propiedades:
  - *Id*: Identificador de la configuración.
  - *Clave*: Clave de la configuración.
  - *Valor*: Valor de la configuración.
- Esta tabla por falta de tiempo al final no se ha llegado a utilizar.

#### 1. **UsuarioConfiguracion:**

- Representa una configuración específica de un usuario.
- Propiedades:
  - *Id*: Identificador de la configuración.
  - *UsuarioId*: Identificador del usuario al que pertenece la configuración.
  - *Clave*: Clave de la configuración.
  - *Valor*: Valor de la configuración.

#### 2. **HistorialPrecio:**

- Representa el historial de precios de un producto.
- Propiedades:
  - *Id*: Identificador del historial.
  - *ProductoId*: Identificador del producto.
  - *Fecha*: Fecha del registro del precio.
  - *Precio*: Precio del producto en la fecha registrada.

#### 3. **Producto:**

- Representa un producto en el sistema.
- Propiedades (entre otras):
  - *Id*: Identificador del producto.
  - *Nombre*: Nombre del producto.
  - *Descripcion*: Descripción del producto.
  - *PrecioInicial*: Precio inicial cuando se capturó el producto.

#### 4. **TiendaOnline:**

- Representa una tienda en línea.
- Propiedades:
  - *Id*: Identificador de la tienda.
  - *Nombre*: Nombre de la tienda.
  - *Url*: URL de la tienda.

### Interfaces en la Capa de Dominio

Las interfaces en la capa de dominio definen los contratos para los servicios y repositorios. Estas interfaces son implementadas en otras capas (como la capa de infraestructura) para proporcionar la funcionalidad necesaria.

Estas son algunas interfaces presentes en la carpeta homónima:

1. IScrapingService
2. ITiendaOnlineFactory
3. IUserario
4. IUserarioRepository
5. IHistorialPrecioRepository
6. IHistorialPrecioRepository
7. IProductoRepository
8. ICurrentUser
9. IEnvioEmailService

En la arquitectura Clean, la capa de dominio es también independiente de las demás capas (como la capa de infraestructura, la capa de aplicación y la capa de presentación). Por lo tanto, el resto de capas dependen directa o indirectamente de ella ya que en ella están representadas todas las entidades del modelo de datos.

### Capa de Aplicación

Es una de las capas centrales y se encarga de orquestar la lógica de negocio y las operaciones de la aplicación. Esta capa está definida dentro de la solución "Descuentor" en el proyecto "Descuentor.Aplicacion".

La capa de Aplicación se estructura principalmente alrededor de la carpeta "Funcionalidades", que implementa los patrones Mediator y CQRS. Esta carpeta contiene los manejadores (handlers) que procesan las solicitudes, organizados en subcarpetas que separan claramente las operaciones de lectura (queries) de las de escritura (commands). Esta separación permite una mejor organización del código y facilita el mantenimiento de la aplicación.

## Dependencia de la Capa de Aplicación

La capa de Aplicación depende únicamente de la capa de Dominio, utilizando las interfaces y entidades definidas en esta. Esta dependencia unidireccional es un principio fundamental de Clean Architecture que permite mantener la lógica de negocio aislada de los detalles de implementación.

### Ejemplo de Funcionalidad: *ObtenerUsuarioQueryHandler*

A continuación, se presenta un ejemplo de cómo se implementa una funcionalidad en la capa de Aplicación utilizando el patrón CQRS y MediatR:

```
public class ObtenerUsuarioQueryHandler : IRequestHandler<ObtenerUsuarioQuery, IActionResult>
{
    private readonly IUsuarioRepository _usuarioRepository;
    private readonly ICurrentUser _currentUser;

    public ObtenerUsuarioQueryHandler(IUsuarioRepository usuarioRepository, ICurrentUser currentUser)
    {
        _usuarioRepository = usuarioRepository;
        _currentUser = currentUser;
    }

    public async Task<IActionResult> Handle(ObtenerUsuarioQuery request, CancellationToken cancellationToken)
    {
        var usuarioAplicacionId = !string.IsNullOrEmpty(_currentUser.Id) ? int.Parse(
            _currentUser.Id!) : 1;
        var resultado = await _usuarioRepository.ObtenerUsuarioPorId(usuarioAplicacionId);

        return new OkObjectResult(resultado);
    }
}
```

En este ejemplo, *ObtenerUsuarioQueryHandler* maneja la consulta *ObtenerUsuarioQuery* para obtener los datos de un usuario. Utiliza el repositorio de usuarios (*IUsuarioRepository*) y la información del usuario actual (*ICurrentUser*) para realizar la operación. El resultado se devuelve como un *IActionResult*, que es una respuesta estándar en aplicaciones ASP.NET Core y que hace de wrapper del resultado, que en este caso son los datos de un usuario.

## Capa de Infraestructura

Es responsable de proporcionar implementaciones concretas para las interfaces definidas en la capa de Dominio y de Aplicación. Esta capa está definida dentro de la solución "Descuentor" en el proyecto "Descuentor.Infraestructura".

En el proyecto *Descuentor*, esta capa incluye varios servicios y repositorios que interactúan con la base de datos, servicios externos y otros componentes del sistema. A continuación, se detalla la estructura y funcionalidad de los componentes principales de la capa de Infraestructura, así como los patrones de diseño utilizados y su relación con las capas de Aplicación y Dominio.

## Servicios Implementados

### QueryableExtensions

Este servicio proporciona una extensión para paginar consultas de tipo *IQueryable<T>*. La paginación es una técnica común para manejar grandes conjuntos de datos dividiéndolos en páginas más pequeñas.

```
public static class QueryableExtensions
{
    public static IQueryable<T> Paginar<T>(this IQueryable<T> queryable, int cantidadRegistros, int pagina)
    {
        return queryable
            .Skip((pagina - 1) * cantidadRegistros)
            .Take(cantidadRegistros);
    }
}
```

### ScrapingService

El *ScrapingService* es responsable de obtener precios de productos desde diferentes tiendas en línea. Utiliza la biblioteca *Playwright* para automatizar la navegación y extracción de datos de las páginas web.

Utiliza un enfoque de procesamiento paralelo para extraer precios de productos desde varias URLs. La función *ObtenerPrecios* emplea *Parallel.ForEachAsync* para realizar solicitudes concurrentes a las URLs de los productos, lo que permite manejar múltiples tareas de scraping simultáneamente y mejorar la eficiencia.

Dentro de cada solicitud concurrente se llama a la función *ObtenerPrecioProducto* que navega a la URL del producto, espera a que el elemento del precio esté presente en la página y luego extrae el texto del elemento. El selector del precio se obtiene mediante la interfaz *ITiendaOnlineFactory*, que proporciona el selector CSS adecuado basado en la URL de la tienda.

El manejo de excepciones se realiza principalmente en dos lugares:

1. **Navegación a la URL:** Si ocurre un error al navegar a la URL, se captura la excepción y se imprime un mensaje de error en la consola.
2. **Instalación de Playwright:** Si la instalación de Playwright falla, se lanza una excepción con el código de salida.

Estas serían las optimizaciones realizadas:

1. **Procesamiento Paralelo:** El uso de *Parallel.ForEachAsync* con un grado máximo de paralelismo de 10 permite realizar múltiples operaciones de scraping en paralelo, reduciendo significativamente el tiempo total de ejecución.
2. **Uso de *ConcurrentDictionary*:** Para manejar los resultados de manera segura en un entorno concurrente, se utiliza *ConcurrentDictionary*, lo que evita problemas de concurrencia.

3. **Navegador en Modo Headless:** El navegador se lanza en modo headless, lo que reduce el consumo de recursos y mejora la velocidad de las operaciones de scraping.

```
public class ScrapingService : IScrapingService
{
    private readonly ITiendaOnlineFactory _tiendaOnlineFactory;

    public ScrapingService(ITiendaOnlineFactory tiendaOnlineFactory)
    {
        _tiendaOnlineFactory = tiendaOnlineFactory;
    }

    public async Task<Dictionary<int, decimal>> ObtenerPrecios(Dictionary<int, string> productosUrl)
    {
        var result = new ConcurrentDictionary<int, decimal>();

        await Parallel.ForEachAsync(
            productosUrl,
            new ParallelOptions { MaxDegreeOfParallelism = 10 },
            async (kvp, token) =>
            {
                var price = await ObtenerPrecioProducto(kvp.Value);
                result.TryAdd(kvp.Key, price);
            });

        return new Dictionary<int, decimal>(result);
    }

    private async Task<decimal> ObtenerPrecioProducto(string url)
    {
        using var playwright = await Playwright.CreateAsync();

        var exitCode = Microsoft.Playwright.Program.Main(new[] { "install" });
        if (exitCode != 0)
        {
            throw new Exception($"Playwright exited with code {exitCode}");
        }

        await using var browser = await playwright.Chromium.LaunchAsync(new BrowserTypeLaunchOptions
        {
            Headless = true
        });

        var page = await browser.NewPageAsync();

        try
        {
            await page.GotoAsync(url);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error al navegar a la URL {url}: {ex.Message}");
        }

        var selector = _tiendaOnlineFactory.ObtenerSelectorPrecio(url);
```

```

    var priceElement = await page.QuerySelectorAsync(selector);

    decimal priceValue = 0;
    if (priceElement != null)
    {
        string price = await priceElement.InnerTextAsync();
        price = price.Replace("€", "").Trim();
        priceValue = decimal.Parse(price);
    }

    await browser.CloseAsync();

    return priceValue;
}
}

```

### TiendaOnlineFactory

El *TiendaOnlineFactory* es un servicio que proporciona los selectores CSS necesarios para extraer los precios de diferentes tiendas en línea. Utiliza un diccionario para mapear las URLs de las tiendas a sus respectivos selectores.

```

public class TiendaOnlineFactory : ITiendaOnlineFactory
{
    private readonly Dictionary<string, string> _selectors = new Dictionary<string, string>
    {
        { "amazon.es", "#priceblock_ourprice, .a-price .a-offscreen" },
        { "es.wallapop.com", "span[class*='ItemDetailPrice']" }
    };

    public string ObtenerSelectorPrecio(string urlTienda)
    {
        foreach (var entry in _selectors)
        {
            if (urlTienda.Contains(entry.Key))
            {
                return entry.Value;
            }
        }

        return ".default-selector";
    }
}

```

### NotificacionDescuentosService.cs

Es el servicio que se encarga de notificar a los usuarios sobre descuentos en productos a través de correo electrónico.

Utiliza el método *NotificarDescuentoEmailAsync* que recibe como parámetros el Id del usuario y la lista de sus productos con descuento. Realiza los siguientes pasos:

#### 1. Obtención del Usuario:

- Busca el usuario por su *Id* y obtiene su correo electrónico.

## 2. Construcción del Mensaje:

- Crea un mensaje HTML que incluye una tabla con los productos y sus descuentos.
- Utiliza `string.Join` y `LINQ` para generar las filas de la tabla con los datos de los productos.

## 3. Envío del Correo:

- Utiliza `IEnvioEmailService` para enviar el correo electrónico al usuario con el asunto y el mensaje construidos.

```
public class NotificacionDescuentosService : INotificacionDescuentosService
{
    // Inyección de dependencias

    public async Task<bool> NotificarDescuentoEmailAsync(int usuarioId, List<Producto
> productos)
    {
        var usuario = _userManager.Users.FirstOrDefault(u => u.Id == usuarioId);
        var email = usuario!.Email;

        var mensaje = $"
<html>
<head>
<meta charset='UTF-8'>
<style>
    body {{ font-family: Arial, sans-serif; }}
    .producto {{ margin-bottom: 20px; }}
    .producto h2 {{ color: #333; }}
    .producto p {{ color: #666; }}
    table, th, td {{ border: 1px solid black; border-collapse: collapse; }}
    th, td {{ padding: 8px; text-align: left; }}
</style>
</head>
<body>
    <h1>Hola {(string.IsNullOrEmpty(usuario.Nombre) ? usuario.Email : usuario.Nombre)}
},</h1>
    <p>Te notificamos que los siguientes productos tienen un descuento:</p>
    <table>
    <thead>
        <tr>
            <th>Producto</th>
            <th>Precio Inicial</th>
            <th>Precio Actual</th>
            <th>% Descuento</th>
        </tr>
    </thead>
    <tbody>
        {string.Join("", productos.Select(p => $"
        <tr class='producto'>
            <td><a href='{p.Url}'>{p.Nombre}</a></td>
            <td>{p.PrecioInicial:C}</td>
            <td>{p.HistorialPrecios!.OrderByDescending(hp => hp.FechaConsulta).First(
).Precio:C}</td>
            <td>{((p.PrecioInicial - p.HistorialPrecios!.OrderByDescending(hp => hp.F
echaConsulta).First().Precio) / p.PrecioInicial * 100):F2}%</td>

```



```

        </tr>"))}
    </tbody>
</table>
    <p>¡Aprovecha estas ofertas!</p>
</body>
</html>";

    var subject = "Descuentor - Descuentos en tus productos favoritos";

    await _envioEmailService.EnviaEmailAsync(email!, subject, mensaje);

    return true;
}
}

```

EnvioEmailService.cs

Es un servicio para enviar correos electrónicos utilizando el protocolo SMTP sirviéndose del servicio de Sandbox de la aplicación Mailtrap.

El método *EnviarEmailAsync*, que recibe del servicio *NotificacionDescuentosService* el cuerpo del email, la dirección del remitente y el concepto del email, envía un correo electrónico utilizando las credenciales y el servidor SMTP de Mailtrap.

### 1. Credenciales y Configuración del Cliente SMTP:

- Se configuran las credenciales (*username* y *pwd*) y se crea una instancia de *SmtplibClient* con el servidor SMTP de Mailtrap y el puerto 2525.
- Se habilita SSL para asegurar la conexión.

### 2. Creación del Mensaje de Correo:

- Se crea una instancia de *MailMessage* con el remitente, destinatario, asunto y cuerpo del mensaje.
- Se establece *IsBodyHtml* en *true* para indicar que el cuerpo del mensaje está en formato HTML.

### 3. Envío del Correo:

- Se utiliza el método *SendMailAsync* del *SmtplibClient* para enviar el mensaje de correo de forma asíncrona.

```

public class EnvioEmailService : IEnvioEmailService
{
    public Task EnviarEmailAsync(string email, string subject, string htmlMessage)
    {
        var username = "040ec699b7d160";
        var pwd = "dc13edff67f09f";

        var client = new SmtplibClient("sandbox.smtp.mailtrap.io", 2525)
        {
            Credentials = new NetworkCredential(username, pwd),
            EnableSsl = true
        };
    }
}

```

```

    var mailMessage = new MailMessage(from: "descuentor@example.com",
        to: email,
        subject: subject,
        body: htmlMessage
    );
    mailMessage.IsBodyHtml = true;

    return client.SendMailAsync(mailMessage);
}
}

```

## Dependencia con las Capas de Aplicación y Dominio

La capa de Infraestructura depende tanto de la capa de Dominio como de la capa de Aplicación, implementando las interfaces definidas en ambas capas. Por ejemplo, implementa interfaces del dominio como *IScrapingService* y *ITiendaOnlineFactory*, proporcionando la lógica concreta para el scraping de precios y la gestión de selectores de las tiendas online. Esta dependencia permite que las capas superiores trabajen con abstracciones mientras la capa de Infraestructura proporciona las implementaciones específicas, manteniendo así una arquitectura limpia y desacoplada.

## Capa de API

El proyecto *Descuentor.API* de la solución *Descuentor* representa la capa de presentación de una arquitectura Clean. Esta capa es responsable de manejar las solicitudes HTTP, procesarlas y devolver las respuestas adecuadas.

Contiene el archivo *Program.cs*, que es el punto de entrada de la aplicación. Este archivo configura y ejecuta la aplicación, y pone a disposición de los clientes los endpoints definidos en los controladores. Contiene los siguientes elementos:

1. **Configuración del Host:** Se configura el host de la aplicación, incluyendo la configuración de servicios y middlewares necesarios.
2. **Configuración de Servicios:** Se registran los servicios necesarios en el contenedor de dependencias, como *IMediator*, *ICurrentUser*, y otros servicios de la aplicación.
3. **Configuración de Middlewares:** Se configuran los middlewares que manejarán las solicitudes HTTP, como la autenticación, autorización, y el enrutamiento.
4. **Ejecución de la Aplicación:** Se construye y se ejecuta la aplicación, poniendo a disposición de los clientes los endpoints definidos en los controladores.

Los controladores son la característica principal de esta capa. Cuando se ejecuta este proyecto automáticamente se pone a disposición de los usuarios los endpoints que permitirán a los distintos componentes interactuar con los servicios y funcionalidades utilizados en el resto de capas del modelo Clean. Tenemos los siguientes:

- *ProductosController*: Este controlador maneja las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para los productos. Utiliza el patrón de diseño Mediator para enviar comandos y consultas a través de *IMediator*.

- *UsuariosController*: Este controlador maneja las operaciones relacionadas con los usuarios, como obtener, crear y actualizar usuarios. También utiliza el patrón Mediator para la comunicación entre la capa de presentación y la capa de aplicación.
- *ActualizarPreciosController*: Este controlador maneja la operación de actualización de los precios y la notificación de los descuentos al cliente por email.

### *Descuentor.Shared*

El proyecto *Descuentor.Shared* es un proyecto de clases que contiene únicamente los DTOs derivados de las entidades de la capa de Dominio. Este proyecto se ha añadido a la solución para facilitar la comunicación entre el backend (API) y el frontend (WebAssembly), permitiendo que ambos componentes compartan estos modelos de datos de manera consistente. Al centralizar estos DTOs en un proyecto independiente, se evita la duplicación de código y se asegura que tanto la API como la aplicación Blazor trabajen con las mismas estructuras de datos, manteniendo la integridad y coherencia en la transferencia de información entre las distintas partes del sistema.

### *Blazor WebApp*

El proyecto *Descuentor.WebAssembly* es el frontend de gestión de productos de la aplicación Descuentor. Está estructurado de la forma habitual de una aplicación Blazor WebAssembly. Incluye los siguientes archivos y carpetas:

- *Program.cs*: Configuración y arranque de la aplicación.
- *App.razor*: Configuración del enrutamiento de la aplicación.
- *Servicios*: Carpeta que contiene los servicios de gestión de estado *TokenService* y *AuthService*.
- *Layout*: Carpeta que contiene componentes de diseño como *MainLayout.razor* y *AuthLayout.razor*.
- *Pages*: Son componentes Razor que representan diferentes vistas o páginas de la aplicación.

### Gestión del estado

La gestión del estado se realiza principalmente a través de los servicios *TokenService* y *AuthService*.

La primera es una clase de servicio que gestiona los tokens de autenticación utilizando la librería *Blazored.LocalStorage* que, a su vez, comunica con el almacenamiento local de Chrome, y la segunda utiliza *TokenService* para realizar las operaciones de registro, logueo, refresco de token y deslogueo llamando a los endpoints específicos de la API.

### Páginas de Blazor

Cada Page está asociada a una ruta URL mediante la directiva *@page*. Las páginas permiten definir la interfaz de usuario y la lógica de la aplicación en un solo archivo.

Aquí tienes una breve explicación de algunas páginas en el proyecto *Descuentor.WebAssembly*:

- *Login.razor*: Proporciona un formulario para que los usuarios inicien sesión en la aplicación. Incluye campos para el email y la contraseña, y maneja la lógica de autenticación a través del servicio de autenticación.
- *Register.razor*: Proporciona un formulario para registrar un nuevo usuario. Incluye campos para el email, contraseña y confirmación de contraseña, y maneja la lógica de registro a través del servicio de autenticación.
- *Home.razor*: Página principal de la aplicación que muestra la lista de los productos capturados por el cliente. Actúa como la página de inicio y puede incluir enlaces a otras partes de la aplicación.
- *ProductoVer.razor*: Muestra los detalles de un producto, incluyendo su historial de precios, tienda, precio inicial y descripción. También proporciona enlaces para ver el producto en la web y un botón para volver a la página anterior.
- *ProductoEditar.razor*: Permite a los usuarios editar la información de un producto existente. Incluye campos para el nombre del producto, precio, descripción, etc., y maneja la lógica para actualizar el producto en el servidor.
- *UsuarioEditar.razor*: Permite a los usuarios editar su información personal, como nombre, apellidos y número de teléfono. También incluye la lógica para manejar la edición de usuario y la navegación.

Los archivos Razor (.razor) son componentes de Blazor que combinan C# y HTML para definir la interfaz de usuario y la lógica de la aplicación. Estos archivos permiten crear componentes reutilizables y páginas dinámicas en aplicaciones Blazor. Los archivos Razor utilizan la sintaxis de Razor para incrustar código C# dentro de HTML, lo que facilita la creación de aplicaciones web interactivas y modernas.

### Chrome Extension

La extensión se organiza en una serie de archivos clave que trabajan en conjunto para proporcionar la funcionalidad completa. En el corazón de la extensión encontramos el *manifest.json*, que actúa como punto de entrada y configuración principal, definiendo los permisos necesarios, los recursos accesibles y la configuración básica de la extensión. La interfaz de usuario se construye principalmente a través de *popup.html* y su correspondiente *popup.js*, que manejan la interacción principal con el usuario.

La autenticación se gestiona a través de *Login.html* y *Login.js*, que trabajan en conjunto con *AuthService.js* para proporcionar una capa robusta de seguridad. Para la configuración específica de las diferentes tiendas online soportadas, se utiliza *storeConfigs.js*, que centraliza toda la lógica de extracción de datos.

### Funcionalidades Implementadas

El sistema de autenticación implementado en *AuthService.js* proporciona una gestión completa de Bearer tokens, incluyendo su almacenamiento seguro en *chrome.storage* y un sistema de refresh automático que mantiene la sesión del usuario activa. La seguridad se refuerza mediante la validación constante del estado de autenticación y la gestión apropiada de los tokens de acceso y refresco.

Para la captura de productos, se ha implementado un sistema modular y extensible que permite añadir fácilmente soporte para nuevas tiendas online. Cada tienda tiene su propia configuración en *storeConfigs.js*, que define los selectores necesarios para extraer información como el nombre del producto, precio, descripción, imágenes y otros datos relevantes. Esta arquitectura modular facilita enormemente la expansión del sistema para soportar nuevas tiendas.

### Interacción con la API

La comunicación con el backend se realiza a través de endpoints bien definidos que manejan la autenticación (*/identity/login* y */identity/refresh*) y el almacenamiento de productos (*/api/productos*). Todas las peticiones se realizan con los headers de autorización apropiados, y se ha implementado un sistema robusto de manejo de errores que incluye la renovación automática de tokens expirados.

La seguridad en las comunicaciones se mantiene mediante Bearer tokens, y se ha implementado un sistema de refresh automático que se ejecuta cada 5 minutos para mantener la sesión activa. Además, se realiza una validación constante del estado de autenticación antes de realizar cualquier operación que requiera autorización.

### Consideraciones Específicas de Chrome

La extensión hace un uso extensivo de las APIs de Chrome, aprovechando *chrome.scripting.executeScript()* para la extracción de datos de las páginas web, *chrome.storage.local* para el almacenamiento persistente de datos de autenticación, y *chrome.tabs.query()* para interactuar con las pestañas del navegador.

Los permisos se han configurado de manera minimalista, solicitando solo los necesarios: *activeTab* para acceder a la pestaña actual, *scripting* para la inyección de código, y *storage* para el almacenamiento local. Esta aproximación mejora la seguridad y la confianza del usuario en la extensión.

### Arquitectura de la extensión

La arquitectura se ha diseñado pensando en la modularidad y la extensibilidad, con una clara separación de responsabilidades entre los diferentes componentes. La configuración de las tiendas se mantiene en un archivo separado, facilitando la adición de soporte para nuevas tiendas sin necesidad de modificar el código core de la extensión.

### Manejo de errores

El manejo de errores es robusto y proporciona feedback claro al usuario en cada operación, mientras que la interfaz de usuario se mantiene simple y efectiva, con estados claros que reflejan el progreso de las operaciones y posibles errores.

Esta implementación representa una solución robusta y segura para la captura de productos en diferentes tiendas online, con un fuerte énfasis en la seguridad, la experiencia del usuario y la facilidad de mantenimiento y extensión.

## Testing

### Estrategia

La estrategia de testing en Descuentor se beneficia directamente de la arquitectura Clean implementada y del uso extensivo de interfaces e inyección de dependencias. Este diseño permite sustituir fácilmente las implementaciones reales por mocks durante las pruebas, facilitando el aislamiento de los componentes a testear.

El proyecto *Descuentor.Tests* dentro de la solución se estructura en dos grandes áreas: tests unitarios y tests de integración. Los tests unitarios se centran en probar componentes individuales de la aplicación, como los servicios de scraping o la lógica de negocio, mientras que los tests de integración verifican la correcta interacción entre diferentes partes del sistema, como la comunicación entre la API y la base de datos.

Para los tests se utiliza xUnit como framework principal, junto con Moq para la creación de mocks de las interfaces. Esta combinación permite simular el comportamiento de dependencias externas y verificar el correcto funcionamiento de la lógica de negocio de manera aislada.

### Unit Tests

Los tests unitarios se centran en probar la lógica de negocio de manera aislada, utilizando mocks para simular las dependencias. Un ejemplo significativo es el testing del servicio de scraping, donde se mockea *ITiendaOnlineFactory* para devolver selectores predefinidos sin necesidad de acceder a sitios web reales:

```
[Fact]
public async Task ObtenerPrecio_DebeRetornarPrecioCorrecto()
{
    // Arrange
    var mockFactory = new Mock<ITiendaOnlineFactory>();
    mockFactory.Setup(f => f.ObtenerSelectorPrecio(It.IsAny<string>()))
        .Returns(".precio");
    var scrapingService = new ScrapingService(mockFactory.Object);

    // Act
    var resultado = await scrapingService.ObtenerPrecios(
        new Dictionary<int, string> { { 1, "https://ejemplo.com" } });

    // Assert
    Assert.True(resultado.ContainsKey(1));
}
```

Otros casos de prueba importantes incluyen la validación del servicio de notificaciones, donde se verifica que los correos se envían solo cuando hay descuentos significativos, y el testing de los handlers de MediatR, asegurando que las operaciones CRUD se ejecutan correctamente.

En el caso de mi proyecto solo he implementado el test unitario para el *ScrapingService*. Como forma de mostrar cómo se haría para el resto de servicios.

### Integration Tests

Los tests de integración se centran en verificar la correcta interacción entre los diferentes componentes del sistema. Para estos tests se utiliza una base de datos PostgreSQL en memoria y

un `TestServer` que simula el entorno de la API, permitiendo probar el sistema end-to-end sin dependencias externas.

Los escenarios principales de prueba incluyen el flujo completo de seguimiento de productos, desde su creación a través de la API hasta la actualización de precios y el envío de notificaciones. Se prueban específicamente las integraciones entre el servicio de scraping y la base de datos, así como la correcta propagación de eventos desde la detección de un cambio de precio hasta la generación de notificaciones.

La configuración del entorno de pruebas se realiza mediante un *TestStartup* personalizado que configura los servicios necesarios, incluyendo una base de datos limpia para cada ejecución de test. Se utiliza el patrón `WebApplicationFactory` de ASP.NET Core para simular el entorno de la aplicación:

```
public class IntegrationTestWebFactory : WebApplicationFactory<Program>
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // Configuración de la base de datos en memoria
            // Registro de servicios mock necesarios
        });
    }
}
```

Los resultados de estas pruebas son fundamentales para garantizar que los diferentes componentes de la aplicación funcionan correctamente en conjunto, validando aspectos como la persistencia de datos, la gestión de transacciones y el correcto manejo de los eventos del sistema.

Por falta de tiempo estas pruebas no se han implementado en el software a día de hoy, pero tenemos una base teórica sólida sobre la comenzar a trabajar para hacer las pruebas necesarias que ofrezcan la mayor cobertura ante los distintos casos de uso de la aplicación.

# Conclusiones finales

## Grado de cumplimiento de requisitos

El desarrollo de Descuentor ha alcanzado un nivel satisfactorio en la implementación de sus funcionalidades core, cumpliendo con la mayoría de los requisitos fundamentales establecidos inicialmente. La aplicación permite efectivamente realizar el seguimiento de precios de productos, notificar cambios significativos y gestionar una lista personalizada de productos a través de una interfaz web y una extensión de Chrome.

Sin embargo, es importante reconocer que algunos requisitos especificados inicialmente no han sido completamente implementados, principalmente debido a limitaciones temporales en el desarrollo del proyecto. A continuación, se detallan los aspectos pendientes de implementación:

### Seguridad y Autenticación

- El sistema de recuperación de contraseñas, aunque planificado en los requisitos iniciales, no ha sido implementado.
- La gestión de tokens de autenticación actualmente utiliza localStorage tanto en la extensión de Chrome como en la aplicación web, lo que representa una vulnerabilidad de seguridad que debe ser abordada en futuras actualizaciones.
- No se han implementado todas las medidas de protección contra ataques comunes como CSRF.

### Personalización y Experiencia de Usuario

A pesar de que en la capa de Dominio se definieron las entidades *Configuracion* y *UsuarioConfiguracion* para gestionar las preferencias de usuario, estas no han llegado a utilizarse en la implementación actual, quedando pendiente para futuras iteraciones del proyecto. Podemos incluir aquí la capacidad de configurar umbrales personalizados para las notificaciones de descuentos.

### Sistema de Scraping

Si bien el sistema básico de scraping funciona correctamente, no se han implementado todas las medidas de robustez planificadas:

- Manejo de casos especiales como productos agotados o páginas de error
- Implementación de delays entre consultas
- Respeto a las políticas de robots.txt de las tiendas online

### Calidad y Testing

No se ha implementado la cobertura de pruebas exhaustiva inicialmente planificada, lo que limita la capacidad de detectar problemas de manera temprana.

### Infraestructura y Rendimiento

- Los requisitos de rendimiento se cumplen actualmente de manera pasiva debido a la baja carga de usuarios, pero no se han implementado optimizaciones específicas.



- Los aspectos relacionados con la disponibilidad y monitorización del sistema no han sido implementados al no estar la aplicación en un entorno de producción.

A pesar de estas limitaciones, es importante destacar que la aplicación cumple con su objetivo principal de proporcionar un sistema funcional de seguimiento de precios, estableciendo una base sólida sobre la cual se pueden implementar las funcionalidades pendientes en futuras iteraciones del desarrollo.

## **Mejoras de la aplicación**

Además de los requisitos no implementados mencionados en la sección anterior, durante el desarrollo del proyecto se identificaron diversas áreas de mejora que serían fundamentales para el lanzamiento de la aplicación en un entorno de producción.

En el ámbito de la arquitectura y el diseño del sistema, se ha identificado la necesidad de reducir el código hardcodeado, especialmente en lo relacionado con los selectores de las tiendas online. Una mejora significativa sería migrar esta información a la base de datos, permitiendo una gestión más flexible y centralizada. En esta misma línea, sería conveniente optimizar el almacenamiento de URLs de productos, guardando únicamente los componentes necesarios para formar la URL completa, mejorando así la eficiencia en el almacenamiento de datos.

Respecto a la seguridad y autenticación, se propone migrar del sistema actual de tokens de Identity Core a JWT (JSON Web Tokens), que ofrecen mayor flexibilidad y capacidad para transportar información codificada. La implementación actual de autenticación en el frontend también requiere una revisión, reemplazando la lógica personalizada por los métodos de autenticación nativos de ASP.NET. Adicionalmente, se deberían implementar las características de seguridad propias del paquete Identity, como la confirmación de email, y establecer un sistema robusto de gestión de secrets para servicios externos.

La experiencia de usuario presenta múltiples oportunidades de mejora. Se propone un rediseño completo de las interfaces siguiendo criterios específicos de UX, implementando un sistema de filtrado en la aplicación web que permita agrupar productos por tiendas, y añadiendo funcionalidades de configuración de usuario como franjas horarias, preferencias de moneda, temas visuales (claro/oscuro), selección de idioma y umbrales personalizados para notificaciones. La implementación de un sistema multi-idioma y un tutorial interactivo mejoraría significativamente la accesibilidad de la aplicación.

En cuanto a la infraestructura y servicios externos, se recomienda migrar a un servidor de correo profesional como Mailchimp o Sendgrid, utilizando un dominio propio (descuentor.com). El sistema de scraping requiere mejoras significativas, incluyendo la gestión de artículos agotados, la implementación de un sistema de rotación de IPs para evitar bloqueos por parte de las tiendas online, y un mecanismo para gestionar cambios en los selectores CSS de las tiendas.

Por último, para garantizar la calidad y el mantenimiento del servicio, es crucial implementar un sistema de logging exhaustivo que funcione en todos los niveles de la aplicación, permitiendo la detección y resolución temprana de problemas. Esto, junto con las mejoras en el rendimiento del scraping y la gestión de productos no disponibles, aseguraría un servicio más robusto y fiable.

Estas mejoras, identificadas durante el desarrollo del proyecto, serían fundamentales para transformar Descuentor de un prototipo funcional a un producto comercial completo y competitivo.

## Nuevas funcionalidades

Más allá de las mejoras identificadas en la sección anterior, durante el desarrollo del proyecto surgieron ideas para nuevas funcionalidades que podrían enriquecer significativamente la propuesta de valor de Descuentor. Estas funcionalidades no solo mejorarían la experiencia del usuario sino que también abrirían nuevas oportunidades de monetización y expansión del servicio.

La implementación de un sistema de afiliación representa una oportunidad clave para la monetización del servicio. Mediante acuerdos con las tiendas online, se podrían generar ingresos a través de enlaces de afiliados cuando los usuarios realicen compras a través de la plataforma. Esta funcionalidad podría complementarse con un sistema de cupones de descuento, permitiendo a los usuarios acceder a ofertas exclusivas y mejorando así el valor percibido del servicio.

La integración con los sistemas de promociones de las tiendas homologadas constituiría otra mejora significativa. El sistema podría monitorizar activamente las campañas de descuentos de las tiendas y notificar a los usuarios sobre promociones relevantes para los productos en su lista de seguimiento. Esta funcionalidad permitiría a los usuarios aprovechar descuentos temporales y ofertas especiales que podrían pasar desapercibidas.

Para mejorar la gestión y supervisión del sistema, se propone desarrollar un panel de administración completo. Este entorno permitiría a los administradores visualizar estadísticas detalladas de uso, gestionar el alta de nuevas tiendas online, administrar usuarios y monitorizar logs del sistema. También podría incluir herramientas para analizar patrones de comportamiento de usuarios y tendencias de precios, facilitando la toma de decisiones estratégicas.

Adicionalmente, se podría implementar un sistema de recomendaciones basado en el historial de productos seguidos por los usuarios, sugiriendo productos similares o complementarios. La inclusión de funcionalidades sociales, como la posibilidad de compartir listas de productos o crear grupos de seguimiento colaborativo, fomentaría la interacción entre usuarios y potenciaría el crecimiento orgánico de la plataforma.

Por último, se podría implementar un sistema de alertas predictivas, utilizando algoritmos de machine learning para anticipar posibles bajadas de precios basándose en patrones históricos y tendencias estacionales. Esto permitiría a los usuarios tomar decisiones más informadas sobre el momento óptimo para realizar sus compras.

Estas nuevas funcionalidades transformarían Descuentor en una plataforma más completa y versátil, mejorando significativamente su propuesta de valor tanto para usuarios como para potenciales socios comerciales.


# Guías

## Guía de uso

Esta guía describe el funcionamiento básico de los dos componentes principales de Descuentor: la extensión de Chrome y la aplicación web.

### *Extensión de Chrome*

La extensión de Chrome permite añadir productos a tu lista de seguimiento mientras navegas por las tiendas online compatibles. Para comenzar, deberás iniciar sesión en la extensión utilizando tus credenciales de Descuentor.

La imagen muestra un formulario de inicio de sesión dentro de un popup. Hay dos campos de entrada: uno etiquetado 'Email:' y otro etiquetado 'Contraseña:'. Debajo de los campos hay un botón verde con el texto 'Iniciar Sesión'. Una etiqueta de ayuda 'Completa este campo' apunta al botón. En la parte inferior, hay un enlace que dice '¿No tienes una cuenta? Regístrate aquí.'

*Figura 8. Captura del popup de la extensión mostrando la pantalla de login*

Una vez autenticado, cuando visites un producto en una tienda compatible, el icono de la extensión se activará. Al hacer clic en él, verás la información del producto que se va a guardar.



*Figura 9. Captura del popup de la extensión mostrando los datos de un producto detectado*

Con un simple clic en el botón "Añadir", el producto se agregará a tu lista de seguimiento y comenzará a monitorizarse automáticamente.

### *Aplicación Web*

La aplicación web funciona como el centro de control donde puedes gestionar todos tus productos guardados. La primera vez que accedas, necesitarás crear una cuenta.

The image shows a registration form for 'descuentor'. At the top, the brand name 'descuentor' is displayed in red. Below it, the title 'Nuevo usuario' is centered. The form contains three input fields: 'Email:', 'Password:', and 'Confirma contraseña:'. Each field is represented by a white rectangular box with a thin border. At the bottom of the form is a blue button with the text 'Registrar' in white.

*Figura 10. Captura de la pantalla de registro mostrando el formulario de creación de cuenta*

Una vez creada la cuenta, podrás acceder mediante la pantalla de login.





















The image shows a login form for 'descuentor'. At the top, the brand name 'descuentor' is displayed in red. Below it, the title 'Login' is centered. The form contains two input fields: 'Email:' and 'Password:'. Each field is represented by a white rectangular box with a thin border. At the bottom of the form is a blue button with the text 'Login' in white. Below the button is a link that reads '¿No tienes una cuenta? Regístrate aquí'.

*Figura 11. Captura de la pantalla de login mostrando el formulario de acceso*

Tras iniciar sesión, accederás a la página principal que muestra tu lista de productos en seguimiento. La interfaz es intuitiva y minimalista, diseñada para proporcionar toda la información necesaria de manera clara y accesible. Los productos se presentan en una lista paginada, permitiéndote navegar fácilmente por tu colección de productos en seguimiento. También se permite ordenar los campos de la tabla por orden pulsando el encabezado de la misma. Al final de cada registro de la tabla se ubican unos botones dónde se podrá visualizar, editar y eliminar los productos.

## Lista de deseos

Puedes navegar por los productos, verlos, modificarlos o eliminarlos.

Foto	Nombre ↑	Tienda ↕	Precio Inicial ↕	Precio Actual ↕	% de variación ↕	Acciones
	A ADDTOP Cargador Solar 25000mAh Power Bank Portátil con 3 Ports 3A Output Batería Externa Impermeable con 4 Paneles Solar para Smartphones Tabletas	Amazon	32 €	29,99 €	-6,3%	  
	Auriculares Inalámbricos Bluetooth, Auriculares Bluetooth 5.3 con 4 HD Mic, 56H Reproducción HiFi Estéreo Auriculares Inalambricos, 13mm Controlador Dinámico, IP7 Impermeable con Pantalla LED, Negro	Amazon	18 €	23,99 €	33,3%	  
	Auriculares Inalámbricos Deportivos, Nueva Auriculares Bluetooth 5.4 con HD Mic, 75H Cascos Inalambricos Bluetooth HiFi Estéreo, Cancelacion Ruido ENC Auriculares Running, IP7 Impermeable/Pantalla LED	Amazon	16 €	35,99 €	124,9%	  
	Batería Externa Carga Rapida 10000mAh, PD 22.5W Power Bank con Pantalla LED, Salida 12V, Mini Batería Externa Movil con USB c, Adecuado para etc.	Amazon	14,72 €	14,72 €	0%	  
	CREATIVE Barra de Sonido Stage SE Debajo del Monitor con Audio Digital USB y Bluetooth 5.3, diálogo Claro y Envolvente por Sound Blaster	Amazon	48 €	55,99 €	16,6%	  

Anterior
1
2
Siguiente

Figura 12. Captura de la página principal de la aplicación web mostrando la lista de productos

La pantalla de visualización del producto ofrece los datos del producto y una gráfica interactiva de la evolución del precio. Se puede ampliar partes de la tabla con la opción de lupa y moverte en el eje temporal. Incluso se podrán descargar los datos en formatos SVG, PNG o CSV.

### Auriculares Inalámbricos Deportivos, Nueva Auriculares Bluetooth 5.4 con HD Mic, 75H Cascos Inalambricos Bluetooth HiFi Estéreo, Cancelacion Ruido ENC Auriculares Running, IP7 Impermeable/Pantalla LED

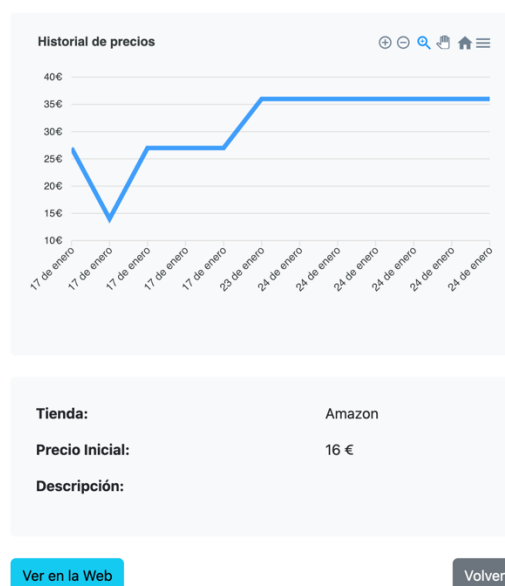


Figura 13. Captura del detalle de un producto mostrando la gráfica de evolución de precios

Para cada producto, podrás editar su información básica a través de un formulario dedicado.

### Editar producto

Nombre

Auriculares Inalámbricos Deportivos, Nueva Auriculares Bluetooth 5.4 con HD Mic, 7!

Descripción

URL Imagen

https://m.media-amazon.com/images/I/71iaWpqrPgL.\_AC\_SX679\_.jpg

URL

https://www.amazon.es/Auriculares-Inal%C3%A1mbricos-Inalambricos-Cancelacion-

Precio Inicial

16

Guardar Cancelar

*Figura 14. Captura de la pantalla de edición de producto mostrando el formulario con los campos editables*

La aplicación también te permite gestionar tu perfil de usuario, donde puedes actualizar tu información personal.

### Editar Usuario

Nombre:

Usuario1

Apellidos:

Gómez

Número de Teléfono:

8432323

Guardar Volver

*Figura 15. Captura de la pantalla de edición de usuario mostrando el formulario de datos personales*

## Guía de instalación

### *Instalación de la Extensión de Chrome*

La extensión de Descuentor actualmente no está disponible en la Chrome Web Store. Para su publicación, será necesario:

1. Crear una cuenta de desarrollador en la Chrome Web Store
2. Empaquetar la extensión en un archivo .zip
3. Completar la información requerida por Google, incluyendo descripción, capturas de pantalla y políticas de privacidad
4. Pagar la cuota única de desarrollador
5. Esperar la revisión y aprobación por parte de Google

Durante el desarrollo, la extensión se ha utilizado en modo desarrollador, que permite cargar extensiones no empaquetadas directamente desde el código fuente:

1. Abrir Chrome y navegar a `chrome://extensions`
2. Activar el "Modo desarrollador" en la esquina superior derecha
3. Hacer clic en "Cargar descomprimida" y seleccionar la carpeta que contiene los archivos fuente de la extensión

### *Instalación del Sistema Backend*

El sistema backend se despliega utilizando Docker Compose para orquestar varios contenedores. La instalación se puede realizar en cualquier máquina Linux que tenga Docker y docker-compose instalado, siguiendo estos comandos:

```
# Clonar el repositorio
git clone https://github.com/miguelalfayate/descuentor.git

# Navegar a la carpeta del proyecto
cd descuentor/src/descuentor

# Construir y ejecutar los contenedores
docker-compose up --build

# Para detener y eliminar los contenedores (opcional)
docker-compose down -v
```

El sistema se compone de cuatro contenedores principales:

#### API REST (Descuentor.API)

Contenedor que representa la API REST desarrollada en ASP.NET Core.

#### Aplicación Web (Descuentor.WebAssembly)

Al ser una aplicación Blazor WebAssembly Standalone, requiere un proceso de dos pasos:

- Primero, realizar el build de la aplicación
- Segundo, servir los archivos estáticos resultantes utilizando NGINX

### Base de Datos

Se utiliza un contenedor PostgreSQL para la persistencia de datos.

### Servicio de Actualización

Se implementa un contenedor ligero basado en Alpine Linux que ejecuta un cronjob diario para actualizar los precios. Este contenedor:

- Ejecuta un script que primero obtiene el token de autenticación usando las credenciales del usuario Admin
- Utiliza el token obtenido para hacer la llamada al endpoint de actualización de precios

Los secretos y configuraciones sensibles se manejan a través de un archivo .env que debe crearse antes del despliegue. El sistema estará disponible en:

- API: <http://localhost:8080>
- Aplicación Web: <http://localhost:80>
- Base de datos: localhost:5432 (acceso interno)



## Referencias bibliográficas

- *Documentación oficial ASP.NET Core*
- *Página oficial del experto en C# Netmentor, con tutoriales y documentación*
- *Webscraping en C#*
- *Implementación de Identity*
- *Patrón repositorio*
- *Mediatr en .Net*
- *Cómo desplegar aplicaciones Blazor WebAssembly en Docker*
- *Ejemplos de Blazor WebAssembly en Docker*
- *Principios SOLID en C#*
- *Tutorial sobre principios SOLID*
- *Más sobre principios SOLID*
- *Patrón Factory*
- *Tutoriales sobre testeo de aplicaciones ASP.NET*
- *Curso Udemy ASP.NET*
- *Curso Udemy de Entity Framework*
- *Tutorial para desarrollar Chrome Extensions*
- *Documentación del paquete ApexCharts para Blazor*