Criptoanálisis y automatización de transacciones.

Nityananda Lorenzo Hernández,
I.E.S San Vicente, San Vicente del Raspeig.

Índice de contenidos

1 Introducción

Actualmente la idea de comercio mediante monedas completamente virtuales está en alza, el intercambio de este tipo de divisas se está convirtiendo en un mercado en auge que cada vez llama la atención de más gente. Se trata de un mercado volátil, fuertemente digitalizado, en el que existe la posibilidad de obtener beneficios aprovechando los márgenes y variaciones del propio poder adquisitivo de las diferentes divisas que emergen día tras día.

Para el proyecto haré uso principalmente del entorno .NET. El lenguaje de programación principal será C#, pero también contendrá algo de HTML y CSS, ya que la parte frontal la crearé empleando .NET Blazor. En cuanto a temas de persistencia, haré uso de una base de datos en Postgres, cuyo cometido principal será almacenar los resultados obtenidos a través de diferentes configuraciones de la aplicación.

En este caso, la aplicación generará un histórico de valores de transacción para poder usar como base frente a una posible ampliación futura de las funcionalidades del servicio, el cual no deja de tener valor en sí mismo. Este servicio se hará cargo de, en base a ciertos valores almacenados en la base de datos, realizar consultas sobre el estado actual de las varias criptomonedas que están siendo trabajadas y gestionar su almacenamiento con Postgres.

La aplicación será capaz de consultar cualquier endpoint que devuelva una respuesta en formato JSON, para ello se emplearán herramientas en la lógica de la aplicación como expresiones de JSONpath, genéricos y reflections.

En resumen, el objetivo principal de este proyecto es crear un servicio que permita consultar el estado actual de cualquier moneda virtual sin importar la estructura del api que ofrezca dicho dato, almacenando los resultados para disponer de un histórico de valores controlado, manteniendo una estructura normalizada y reutilizable cara a posibles expansiones futuras.

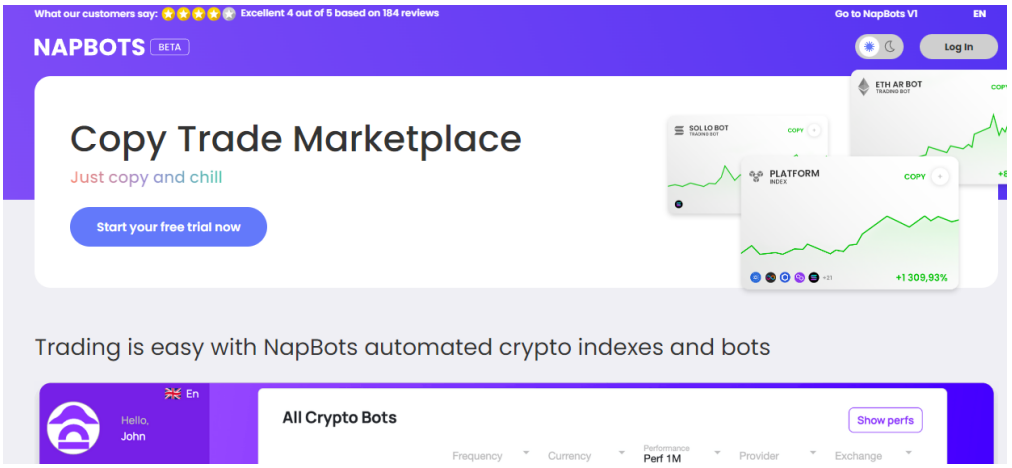
2 Antecedentes

Tras realizar un estudio de mercado he detectado que existe una cantidad considerable de servicios que ofrecen aplicaciones similares, pero siempre están sujetas a los listados de mercados con los que ellos trabajan. Por tanto, considero que tener la opción de obtener datos de cualquier mercado y ser capaz de presentarla a clientes potenciales puede ser una gran innovación.

2.1 Aplicaciones de consumo de datos

A continuación se incluyen múltiples opciones existentes en el mercado actual que consumen datos similares a los que he decidido capturar con este proyecto.

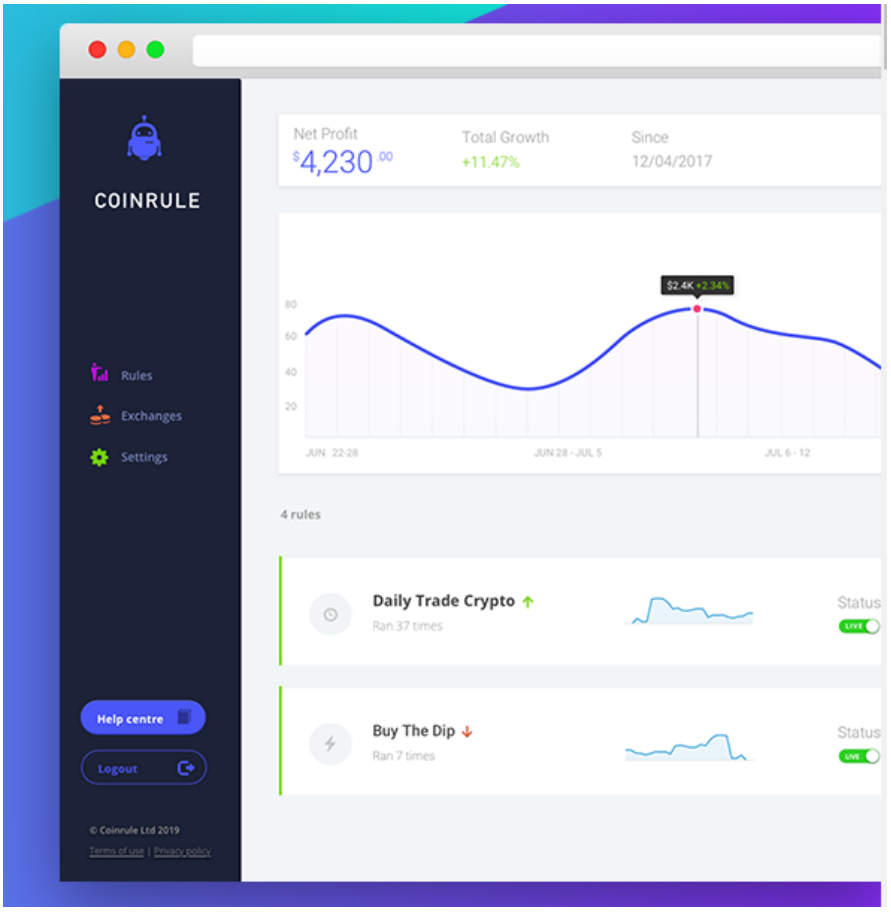
- **NapBots:** Ofrece estadísticas y algoritmos de búsqueda preconfigurados, adjunto a un coste de 83 € / mes en caso de querer hacer uso de todo lo que ofrece.



- **CryptoHopper:** Ofrece estadísticas, algoritmos junto a múltiples intercambios y monedas, a cambio de alrededor de 99$ / mes.

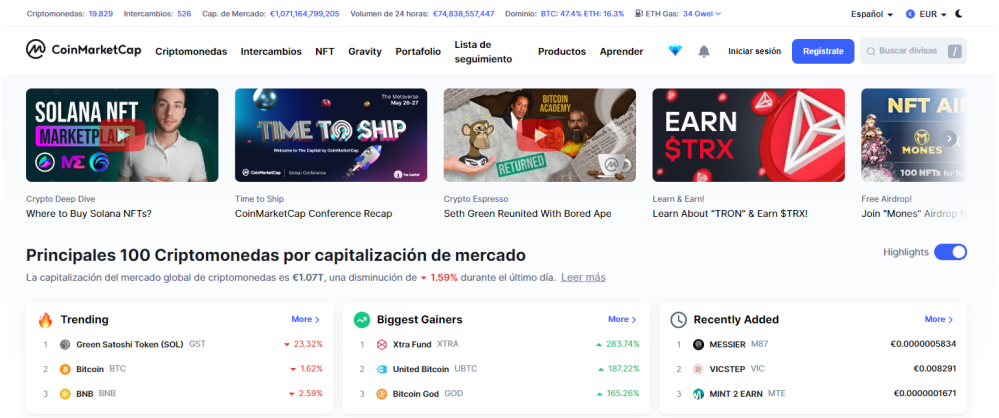
- **Coinrule:** Tal y como los antecedentes, ofrece estadísticas y múltiples opciones y servicios, con un

coste adjunto de 449 $ / mes.



2.2 Aplicaciones de consulta de datos

- **CoinMarketCap:** Es una aplicación masiva que ofrece un servicio muy similar al propuesto para mi proyecto, con la diferencia de que a la hora de necesitar algo concreto, o un nuevo intercambio menos conocido, dependes de que decidan incluirlo en sus listados.



En general, todo este tipo de empresas aplican su propia lógica de obtención de datos, mientras que en mi solución sería posible consultar estos mismos datos obtenidos desde múltiples fuentes de forma completamente normalizada y fácil de implementar, con la opción de ampliar a nuevos mercados y fuentes de datos de forma sencilla y transparente para la persona que administre el servicio.

3 Análisis

La aplicación deberá permitir a los usuarios introducir información relevante sobre cualquier intercambio de criptomoneda para que comience a realizar un seguimiento de las estadísticas del mismo. Por este motivo, será necesario un algoritmo de normalización de datos que permita extraer la información relevante a partir de cualquier estructura en formato JSON, sin importar la profundidad a la que se encuentren los datos, o los nombres de los atributos que los contengan. Este método deberá estar optimizado para que los tiempos de consulta y almacenamiento de los datos sean lo más bajos que permitan las tecnologías empleadas en el proyecto.

Por otro lado, para facilitar la forma en la que la aplicación será configurada, la propia configuración de la aplicación será almacenada en tablas relacionales dentro de la base de datos, cuya gestión será expuesta mediante servicios web desde el servidor que aloje este proceso paralelizable. La aplicación permitirá:

- Añadir o eliminar nuevas fuentes de datos de mercado con los valores más recientes.
- Añadir o eliminar los datos que se desean recopilar, en este caso parejas de monedas.
- Consultar los datos almacenados.

4 Diseño

Las principales funciones de la aplicación serán las siguientes:

- Un microservicio de información con:
    - Un punto de acceso público para realizar la configuración de consulta.
    - Un punto de acceso público para activar o desactivar las diferentes monedas intercambiadas.
    - Un sistema interno de normalización de datos que ajustará la respuesta de cualquier api a las necesidades de la base de datos de la aplicación.
    - Un trabajo de fondo que realizará peticiones a todas las apis conocidas de forma periódica y almacenará los resultados.
- Una pantalla de configuración externa al servicio principal que permita:
    - Incluir nuevas fuentes de datos
    - Eliminar fuentes de datos obsoletas
    - Especificar los datos consultados

El objetivo de esta arquitectura de aplicación es tener una torre centralizada con el proceso más pesado de obtención de datos de apis, mientras que el resto de procesos puedan distribuirse y replicarse con facilidad sin tener la necesidad de ser ejecutados en el mismo hardware.

Como metodología de desarrollo he decidido aplicar una metodología incremental, ampliando las funcionalidades de la aplicación tras cada iteración. A continuación indico un esquema contemplando los varios ciclos de desarrollo, las herramientas que he aplicado, y los requisitos que he necesitado cumplir para lograr los objetivos de mi aplicación:

- Estudio y análisis de múltiples fuentes de datos posibles para cumplir los objetivos del proyecto
    - Aprendizaje sobre cómo funcionan ciertos aspectos del comercio de criptomonedas
        - Estudio para descartar o aceptar posibles objetivos extra del proyecto
- Diseño de la base de datos
    - Análisis de los requisitos relacionales que mejor se adaptan a los requisitos de la aplicación
        - Búsqueda de maneras de optimizar el impacto de los requisitos sobre la fluidez del proceso de obtención de datos
        - Búsqueda del mejor diseño para conocer qué datos son consultados por qué usuario y ser capaz de monetizar la aplicación en un futuro
    - Creación de las tablas y relaciones
- Creación de endpoints básicos no relacionales
    - Aprendizaje sobre los requisitos mínimos de ASP .NET Core
        - Aprendizaje sobre el uso de dependencias inyectables con Asp .Net Core
            - Detección de mis propias clases con los repositorios y servicios que se encargan de controlar la persistencia de los datos
            - Inyección de Logger en mis repositorios, controladores y servicios, dejando la estructura preparada para guardar un registro de lo que sucede
            - Uso de Swagger durante el desarrollo para realizar las consultas sobre la Api
    - Aprendizaje de Entity Framework Core, herramienta usada para el control de la persistencia
        - Investigación sobre las formas de hacer funcionar EF con PostgreSQL
            - Empleada librería NPGSQL
        - Acceso a la capa de persistencia diseñado con múltiples capas de abstracción entre Repositorio y Controlador
            - Creado servicio intermediario con todos los repositorios necesarios para la persistencia para lograr que los requisitos del controlador sean transparentes para la aplicación
    - Inyección del servicio de persistencia de datos en los controladores
- Inclusión de las tablas relacionales
    - Estudio más profundo de Entity Framework Core
    - Aprendizaje sobre los requisitos mínimos y mejores prácticas para lograr un código más mantenible con EF Core
        - Aplicada configuración a nivel de contexto en lugar de nivel de entidad
- Creación de endpoints relacionales

    - Resolución de problemas del estado interno de las múltiples entidades gestionadas por Entity Framework
- Creación del proceso de obtención y normalización de datos
    - Estudio de varios acercamientos para resolver la complejidad de obtener datos a cualquier profundidad en un fichero JSON
        - Descartada la opción de trabajar de forma recursiva debido a problemas de rendimiento
    - Encontrada la existencia de expresiones tipo JSONPath avaladas por Oracle
    - Implementado método de normalización de datos a bajo nivel combinando genéricos, expresiones de JSONPath, reflections y atributos propios
        - Creado atributo de filtrado para asignar una propiedad por cuyo valor se filtran los datos durante el proceso
    - Detectado problema de control de memoria, el uso de memoria excede unos valores razonables debido al fuerte uso de EF Core por el proceso
        - Estudiada la forma de controlar el uso de memoria de la aplicación
            - Implementado método disposing a varios niveles de la aplicación para vaciar por completo el contexto de EF Core y controlar así un uso excesivo de memoria
    - Detectado problema de rendimiento, el proceso de filtrado es bastante pesado pero paralelizable
        - Aplicada paralelización del proceso en todos los bucles de filtrado internos donde ha sido posible para que se complete en el menor tiempo posible
- Inclusión del método de obtención de datos como procedimiento programado para repetirse cada cierto tiempo
    - Estudio de cómo crear un BackgroundService en Asp .Net Core
    - Implementación del servicio
        - Configuración para su inyección al inicio de la aplicación
- Creación de una interfaz básica para el proyecto con Blazor
    - Configuración de CORS para lanzar un servicio web local que apunte hacia mi Api
- Creación de una pantalla de configuración para incluir o eliminar nuevas fuentes de datos
    - Importación de librería BlazorStrap para agilizar el diseño de la interfaz
        - Estudio básico para el uso de las herramientas ofrecidas
- Creación de una pantalla de consulta de datos para mostrar una gráfica con algunos resultados obtenidos por el proceso de obtención y normalización
    - Importación de Ploty.Blazor para la creación de las gráficas

4.1 Base de datos

La base de datos es relacional, y emplea PostgreSQL. A continuación adjunto el esquema entidad relación y la estructura de las diferentes tablas que lo forman.

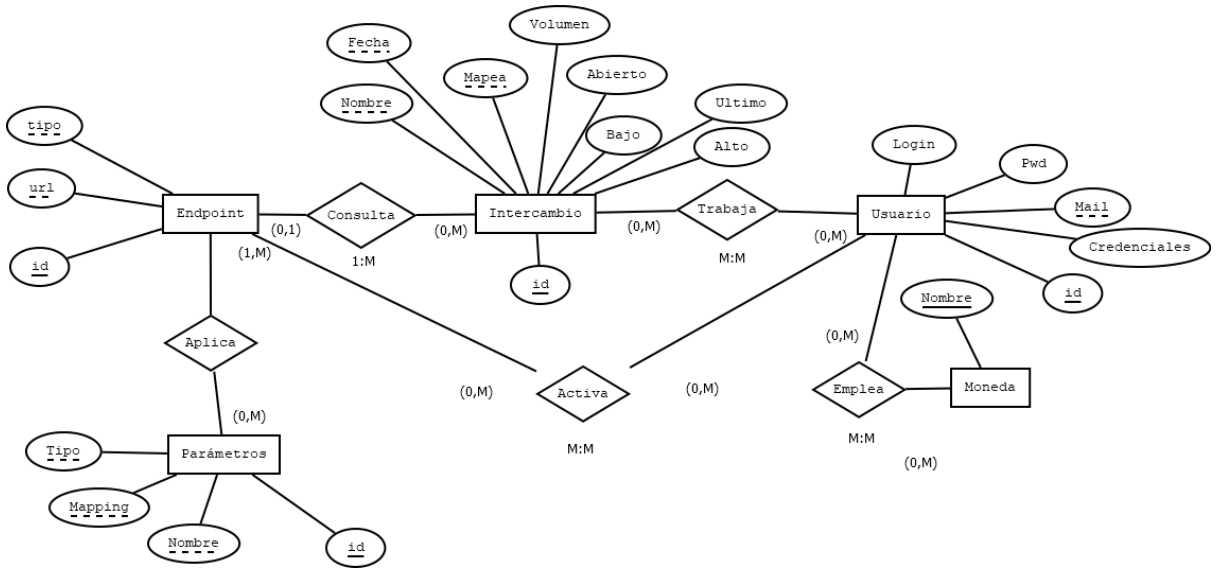


Fig. 1. Esquema Entidad-Relación.

Listado de tablas:

Id	Integer, Primary Key, Auto Increment	Clave primaria
Url	Character varying, Unique Key	Enlace
Tipo	Character varying	Tipo de enlace

Tab. 1. Tabla Endpoint

Id	Integer, Primary Key, Auto Increment	Clave primaria
Nombre	Character varying	Nombre de la moneda
Fecha	Timestamp with timezone	Fecha de la consulta
Intercambiado	Character varying	Nombre de la otra moneda
Volumen	Numeric	Volumen de intercambio
Abierto	Numeric	Precio al abrir el intercambio
Alto	Numeric	Precio más alto del intercambio
Bajo	Numeric	Precio más bajo del intercambio
Reciente	Numeric	Precio más reciente intercambiado
Cod_edp	Integer, Foreign Key(Endpoint)	Id del endpoint del que provienen los datos

Tab. 2. Tabla Intercambio

Id	Integer, Primary Key, Auto Increment	Clave primaria
Valor	Character varying	Nombre del parámetro en el JSON
Mapping_modelo	Character varying	Nombre del parámetro en el Modelo interno de la aplicación
Tipo	Character varying	Tipo de parámetro

Tab. 3. Tabla Parámetros

Id	Integer, Primary Key, Auto Increment	Clave primaria
Login	Character varying	Login del usuario
Pwd	Character varying	Clave del usuario
Mail	Character varying	Email del usuario

Tab. 4. Tabla Usuario

Nombre	Character varying, Primary Key	Nombre de la moneda o pareja de intercambio

Tab. 5. Tabla Moneda

Cod_usr	Integer, Foreign Key(Usuario)	Usuario relacionado con el intercambio
Cod_itc	Integer, Foreign Key(Intercambio)	Intercambio relacionado con el usuario

Tab. 6. Tabla relacional Usuario Intercambio

Cod_edp	Integer, Foreign Key(Endpoint)	Endpoint relacionado con el parámetro
Cod_prm	Integer, Foreign Key(Parámetros)	Parámetro relacionado con el endpoint

Tab. 7. Tabla relacional Endpoint Parámetros

Cod_edp	Integer, Foreign Key(Endpoint)	Endpoint relacionado con el usuario
Cod_usr	Integer, Foreign Key(Usuario)	Usuario relacionado con el endpoint

Tab. 8. Tabla relacional Endpoint Usuario

Cod_mnd	Integer, Foreign Key(Moneda)	Moneda relacionada con el usuario
Cod_usr	Integer, Foreign Key(Usuario)	Usuario relacionado con la moneda

Tab. 9. Tabla relacional Usuario Moneda

4.2 Servidor

Está basado en Asp .Net Core, se encarga de gestionar la configuración y ejecución del proceso de obtención de datos.

Listado de endpoints:

Descripción	Listado de endpoints con sus parámetros
Ruta	/api/ApiConfig/Endpoints
Método	GET
Parámetros de salida	Endpoints: IActionResult<IEnumerable<EndpointDto>>

Tab. 10. Consulta de Endpoints

Descripción	Listado de endpoints con sus parámetros en base a la id de usuario
Ruta	/api/ApiConfig/EndpointsPorUsuario/{userId}
Método	GET
Parámetros de entrada	userId: Integer
Parámetros de salida	Endpoints: IActionResult<IEnumerable<EndpointDto>>

Tab. 11. Consulta de Endpoints por usuario

Descripción	Listado de parámetros con los endpoints que los aplican
Ruta	/api/ApiConfig/Parametros
Método	GET
Parámetros de salida	Parámetros: IActionResult<List<ParametroDto>>

Tab. 12. Consulta de Parámetros

Descripción	Listado de todos los intercambios almacenados con el endpoint del que han sido consultados
Ruta	/api/ApiConfig/Intercambios
Método	GET
Parámetros de salida	IActionResult<List<IntercambioDto>>

Tab. 13. Consulta de Intercambios

Descripción	Listado de N endpoints, con filtrado por endpoint opcional, ordenados por fecha (descendiente)
Ruta	/api/ApiConfig/Intercambios/{limit}
Método	POST
Parámetros de entrada	Url: String, Limit: Integer
Parámetros de salida	IActionResult<List<IntercambioDto>>

Tab. 14 Consulta de Intercambios con límite y filtrado por url

Descripción	Listado de N endpoints filtrados por pareja, con filtrado por endpoint opcional, ordenados por fecha (descendiente)
Ruta	/api/ApiConfig/Intercambios/{limit}/{par}
Método	POST
Parámetros de entrada	Url: String, Par: String, Limit: Integer
Parámetros de salida	IActionResult<List<IntercambioDto>>

Tab. 15 Consulta de Intercambios con límite y filtrado por url y pareja

Descripción	Listado de usuarios
Ruta	/api/ApiConfig/Usuarios
Método	GET
Parámetros de salida	IActionResult<List<UsuarioDto>>

Tab. 16 Consulta de Usuarios

Descripción	Listado de monedas
Ruta	/api/ApiConfig/Monedas
Método	GET
Parámetros de salida	IActionResult<List<MonedaDto>>

Tab. 17 Consulta de Monedas

Descripción	Inserción de un nuevo endpoint
Ruta	/api/ApiConfig/NuevoEndpoint
Método	POST
Parámetros de entrada	Endpoint: EndpointCreateDto
Parámetros de salida	IActionResult<EndpointDto>

Tab. 18 Inserción de nuevo Endpoint

Descripción	Inserción de un nuevo endpoint directamente vinculado a un usuario
Ruta	/api/ApiConfig/NuevoEndpoint/{userId}
Método	POST
Parámetros de entrada	Endpoint: EndpointCreateDto, UserId: Integer
Parámetros de salida	IActionResult<EndpointDto>

Tab. 19 Inserción de nuevo Endpoint vinculado a un usuario

Descripción	Modificación de un endpoint ya existente
Ruta	/api/ApiConfig/ActualizarEndpoint/{endpointId}
Método	PUT
Parámetros de entrada	Endpoint: EndpointCreateDto, EndpointId: Integer
Parámetros de salida	IActionResult<EndpointDto>

Tab. 20 Modificación de un endpoint ya existente

Descripción	Inserción de un parámetro, existente o no, en un endpoint ya existente
Ruta	/api/ApiConfig/IncluirParametroEnEndpoint/{endpointId}
Método	PUT
Parámetros de entrada	Parametro: ParametroDto, EndpointId: Integer
Parámetros de salida	IActionResult<EndpointDto>

Tab. 21 Modificación de un endpoint ya existente y sus parámetros asociados

Descripción	Inserción de una nueva Moneda
Ruta	/api/ApiConfig/NuevaMoneda
Método	POST
Parámetros de entrada	Moneda: MonedaDto
Parámetros de salida	IActionResult<MonedaDto>

Tab. 22 Inserción de una nueva Moneda

Descripción	Eliminar una moneda
Ruta	/api/ApiConfig/EliminarMoneda
Método	DELETE
Parámetros de entrada	Nombre: String
Parámetros de salida	IActionResult<MonedaDto>

Tab. 23 Eliminación de una moneda por su nombre

Descripción	Eliminar un endpoint
Ruta	/api/ApiConfig/EliminarEndpointPorUrl
Método	DELETE
Parámetros de entrada	Url: String
Parámetros de salida	IActionResult<EndpointDto>

Tab. 24 Eliminación de un Endpoint por su Url

4.3 Proceso de consulta de datos

Es el núcleo de la aplicación, se encarga de recorrer un JSON string y extraer los datos de interés en un formato normalizado y persistible con Entity Framework Core. Está diseñado como BackGroundService para repetirse siempre que la aplicación esté en marcha.

Este proceso se encarga de, en base a la configuración almacenada en la base de datos, consultar un listado de endpoints, aplicando sus parámetros relacionados, filtrando por las parejas de monedas conocidas. El proceso está diseñado para aceptar cualquier clase que incorpore el atributo personalizado ideado para indicar la propiedad principal por la cual se debe aplicar el filtrado del JSON.

Haciendo uso de genéricos y reflections, se almacenan los valores de las propiedades relevantes en su sitio correspondiente, realizando un intento de conversión al tipo de datos de la propiedad. En caso de fallar la conversión directa a un tipo primitivo por reflections, se aplica un segundo intento de conversión, también haciendo uso de reflections, aplicando la información de tipo extraída de la propiedad al método de deserialización genérico de NewtonSoft.

Es un proceso más pesado a más grande el JSON string que almacena los datos, por tanto ha sido paralelizado en todos los puntos que así lo permiten para minimizar los tiempos de obtención de resultados.

4.4 Interfaz de usuario

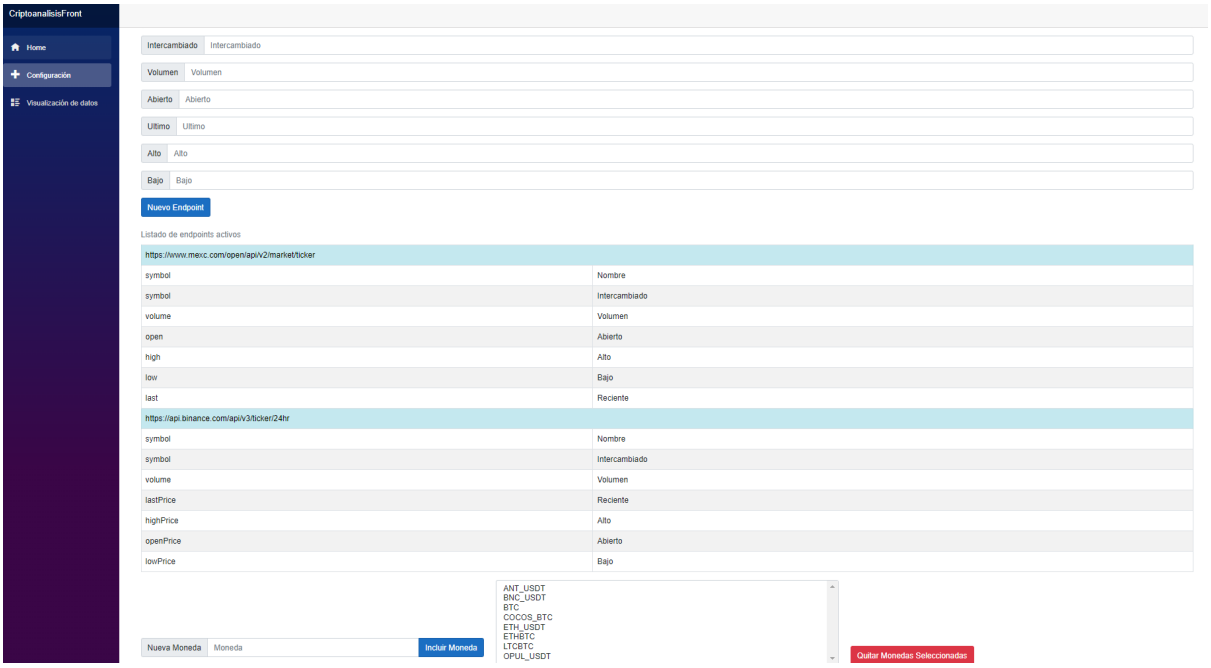
Cumple las veces de panel de control de la aplicación. Se ofrece una pantalla funcional, creada con Blazor y C#, para que el administrador de la aplicación sea capaz de incluir nuevas fuentes de datos de forma simple y directa.

También se incluye una estadística a modo de demostración en la cual se pueden observar algunos de los datos que van siendo almacenados por la aplicación. Esta estadística, creada con la librería de Ploty para Blazor, permite consultar cualquier intercambio almacenado, y muestra por pantalla el desglose de los valores más recientes para el par de monedas seleccionado y su evolución en el tiempo.

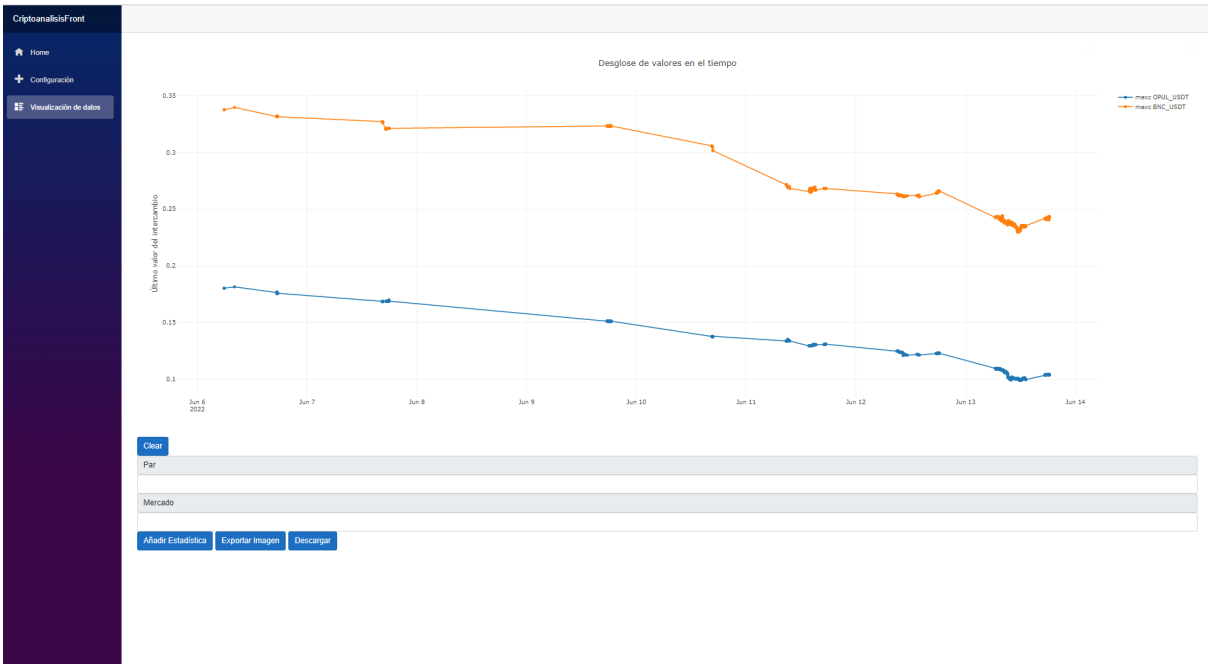
5 Resultados

Aunque se han cumplido todas las funcionalidades definidas por la parte de diseño, y el desarrollo del proyecto ha cumplido la función de aprendizaje en profundidad del funcionamiento de Entity Framework Core, los objetivos cumplidos son una buena base de la cual se puede partir hacia múltiples objetivos.

5.1 Pantallas de la aplicación



Img. 1 Pantalla de configuración



Img. 2 Pantalla de estadísticas

6 Conclusiones

La parte que más tiempo ha llevado del proyecto ha sido la de investigación. De las cien a ciento veinte horas que ha llevado completar el proyecto, más de la mitad se han invertido en encontrar fuentes de información válidas para poder cumplir los objetivos.

El proceso de normalización de datos ha sido la parte más compleja de implementar, partiendo de un acercamiento recursivo al análisis de un JSON string, descartado por varios problemas de rendimiento, ha acabado convirtiéndose en un método que emplea múltiples herramientas con cierto nivel de abstracción. Ha servido para aprender cómo realizar llamadas genéricas a partir del propio tipo que recibe el método, y para profundizar bastante en el manejo de objetos de C# a un nivel más bajo de lo que se ha estudiado durante el curso.

6.1 Problemas encontrados

6.1.1 Mal planteamiento inicial

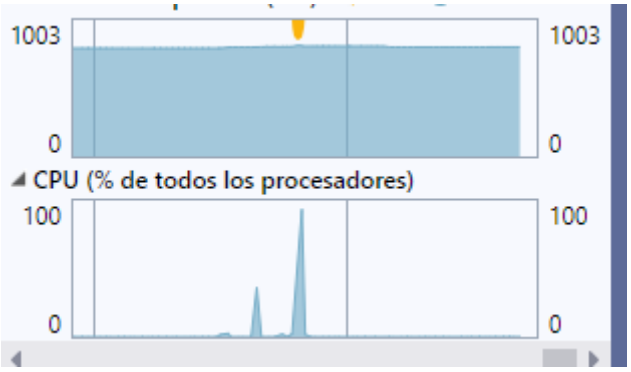
Mantenibilidad del código, uso de genéricos y abstracciones con EntityFramework. Durante el primer acercamiento a la implementación del proyecto, decidí tratar de crear un controlador abstracto, que hiciera uso de un repositorio también abstracto, los cuales aplicaran una serie de métodos estándar a cualquier crud.

Al ser la primera vez que trabajo con Entity Framework, en este punto del desarrollo desconocía por completo los requisitos del contexto de datos que emplea esta herramienta, y debido a un mal planteamiento desde el principio fue necesario descartar todo el código que llevaba hasta el momento en el que comencé a trabajar las relaciones de la base de datos.

Esto me forzó a rehacer por completo la implementación de controladores, repositorios e interfaces, pero me permitió iniciar un planteamiento diferente, el cual acabó en lo que presento en el proyecto. He tratado de desvincular al máximo posible las varias capas de la aplicación, creando varias capas de abstracción intermedias desde el controlador hasta el repositorio que implementa el contexto de persistencia, tratando de asegurar el menor impacto posible en el caso de necesitar realizar cambios en un futuro.

6.1.2 Problemas de rendimiento

El objetivo de este proyecto es tener un método fluido, lo más rápido posible, capaz de obtener y persistir datos con la menor latencia y espera posible. Por ello, el método de obtención y persistencia de datos fue rehecho unas cuantas veces hasta llegar a dar con la forma más paralelizable posible, con el menor número de llamadas a servidores externos posible.



Img. 3 Uso de CPU por un i7 7700k (4 núcleos / 8 hilos)

Otro de los problemas encontrados fue el uso de memoria. Tras varias pruebas iniciales detecté que el consumo de memoria excedía los parámetros normales esperables por una aplicación de estas características. Tras investigar esto más a fondo, me di cuenta de que era necesario implementar IDisposable por el servicio que inyecto en el proceso, y emplearlo para llamar al método Dispose de los repositorios, que a su vez hacen lo mismo con el DBContext de Entity Framework. Sin este paso, el uso de memoria de la aplicación estaba totalmente descontrolado.

6.2 Trabajos futuros

Como mejoras para la aplicación, el proceso actual sería fácilmente monetizable a partir de varios acercamientos.

En primer lugar, sería posible vender directamente el histórico de datos en base al número de peticiones y variedad de intercambios. También existen otras posibilidades, como el hecho de disponer de estos datos permitiría ciertas automatizaciones, consumiendo los datos para realizar compra-venta de criptomonedas.

7 Bibliografía

Para el desarrollo del proyecto se han empleado los siguientes recursos

- Asp .Net Core
  - https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-6.0
- Entity Framework
  - https://docs.microsoft.com/en-us/ef/core/
- Newtonsoft.Json
  - https://www.newtonsoft.com/json
- NPGSQL
  - https://www.npgsql.org/
- BlazorStrap
  - https://blazorstrap.io/
- Ploty.Blazor
  - https://github.com/LayTec-AG/Plotly.Blazor