# 1 ВСТУП

На сьогоднішній день різні види цифрової валюти(далі **крипто валюти**) набирають стрімку популярність. Вони забезпечують прозору альтернативу фіатним грошам, а також дозволяють з легкістю проводити операції переводу, конвертації та зберігання валюти. В основі будь-якої валюти лежить так званий блокчейн, який відповідає за зберігання даних про транзакції користувачів. Стрімкий ріст популярності привів до того, що нові крипто валюти з’являються та зникають щодня. Також крипто валюти можна обмінювати на декількох торгових майданчиках, тому існує гостра потреба в інформаційному засобі для спостерігання за курсом різних валют на різних майданчиках.

Метою курсової роботи є розробка власного програмного продукту для спостерігання змін курсів крипто валют на різних торгових майданчиках. На початку курсової роботи було досліджено декілька існуючих рішень для спостерігання курсу крипто валют. В результаті дослідження були виявлені наступні риси гарного програмного продукту: простий і зрозумілий інтерфейс, крос-платформеність та відсутність потреби у встановлені додаткового програмного забезпечення на пристрій кінцевого користувача.

Загалом, розгляд доступного програмного забезпечення для спостеріганням за курсом крипто валют виявив головний недолік існуючих систем: всі вони прив’язані до певного торгівельного майданчику, тому основною метою цієї курсової роботи є створення сервісу для агрегації даних про курси крипто валют з існуючих торгівельних майданчиків. І сповіщення кінцевого користувача.

При створенні програми були використані основні принципи ООП: поліморфізм, інкапсуляція та наслідування – для полегшення процесу проектування програми та створення її функціоналу.

# 2 ПОСТАНОВКА ЗАДАЧІ

Метою цієї курсової роботи є створення програмного рішення для моніторингу курсу крипто валют на різних торгівельних майданчиках. Користувач повинен мати можливість отримувати оновлення в реальному часі. Рішення має бути легко масштабованим. До того ж рішення має бути розрахованим на розробку та роботи з відсутністю початкового бюджету.

Рішення має надавати запроваджувати наступний функціонал для користувача:

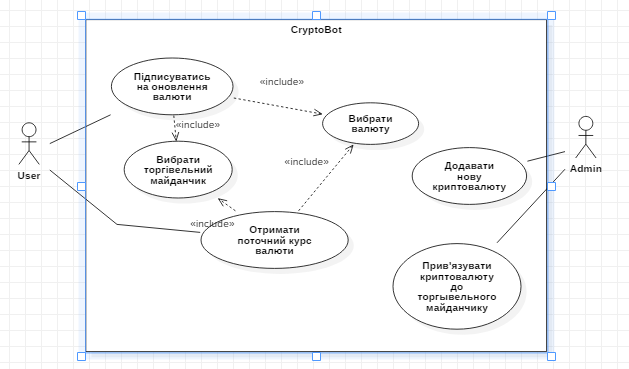
* Отримання поточного курсу крипто валюти без прив’язки до певного торгівельного майданчику;
* Отримання поточного курсу крипто валюти для певного майданчику;
* Підписуватись на оновлення курсу певної крипто валюти з прив’язкою до певного торгівельного майданчика;
* Отримувати оновлення курсу крипто валют згідно з підписками у реальному часі;

Також рішення має надавати можливість для адміністратора:

* Додавати нові крипто валюти до списку;
* Прив’язувати існуючі крипто валюти до торгівельних майданчиків;

Адміністратор авторизується за допомогою токену в хедері запиту, та виконує операції за допомогою REST API, без графічного інтерфейсу. Для детальнішого опису взаємодії користувачів з програмним продуктом було побудовано use-case діаграму. Use-case діаграма (також відома яка діаграма прецедентів) – представлення проектованої системи у вигляді скінченої системи акторів, що взаємодіють із системою за допомогою варіантів використання. Use-case діаграму для проектованої системи наведено на наступній сторінці (див. рис. 2.1)

Рис. 2.1 – Діаграма прецедентів



# 3 ОГЛЯД ІСНУЮЧИХ РІХЕНЬ ТА ОПИС МОДЕЛІ

2.1 Огляд та аналіз існуючих рішень

В ході аналізу існуючих рішень, мною були виявлені умовно безкоштовні аналоги реалізації функціоналу програмного рішення в мережі Інтернет. Нажаль, ці рішення, хоч і мали багато функціональних можливостей, були зосередженні на одному або декількох торгівельних майданчиках. Основною метою таких рішень є збільшення доступу користувача до особистого кабінету таких майданчиків. Оскільки метою цього програмного рішення є агрегація існуючих торгівельних майданчиків, було прийнято рішення розробляти цей продукт згідно з ліцензією GNU General Public License, зробивши вихідний код відкритим. Для взаємодії з іншими розробниками буде використовуватись ресурс [Github.com](http://github.com/).

Провівши аналіз існуючих рішень, з оглядом на важливість миттєвого сповіщення користувачів та дуже лімітований(або зовсім відсутній) бюджет додатку, було обрано найбільш підходящий – бот у системі миттєвого обміну повідомленнями. Такий підхід дає велику кількість переваг:

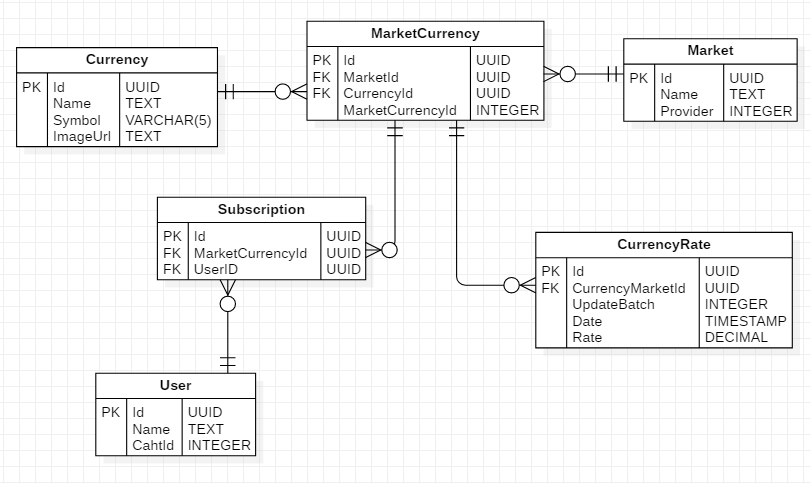
* зручність інтерфейсу забезпечується програмою для обміну повідомлень;
* гнучкість у системних вимогах;
* відсутність необхідності у встановленні додаткового програмного забезпечення на пристрій користувача;
* простота у користуванні;
* відсутність додаткових витрат на розробку або використання існуючої системи миттєвих сповіщень;
* Можливість створення ботів для різних систем обміну повідомленнями, не змінюючи серверну частину;

В якості кінцевої платформи для миттєвого обміну повідомленнями було орано відносно нову – Telegram. Такий вибір обумовлено зручністю створення та адміністрування боту у цій системі та великою кількістю програмних open-source бібліотек для створення ботів для цільової платформи(.Net core).

2.2 Аналіз предметної області

Для аналізу предметної області скористаємося Entity-Relation моделлю. Entity-Relation модель – модель даних, що дозволяє описати концептуальні схеми предметної області, такі як моделі та зв’язки між ними для відображення в базі них. Ця модель відображається за допомогою простої і наочної Entity-Relation діаграми. Побудова цієї моделі дозволяє наочно зобразити предметну область, що полегшує подальше проектування. На рисунку 2.1 зображена Entity-Relation модель для предметної області курсової роботи.

Рис. 2.1 – Entity-Relation модель



В предметній області існують такі сутності:

* **User** – представлення користувача у системі;
* **Currency –** представлення крипто валюти у системі;
* **Market –** представлення торгівельного майданчику у системі;
* **CurrencyRate –** оновлення курсу певної крипто валюти на певному торгівельному майданчику**;**
* **CurrencyMarket –** розв’язочна сутність для відношення багато до багатьох між **Currency** та **Market**. Це важлива сутність, оскільки вона також вказує на id валюти на зовнішньому ресурсі(торгівельному майданчику);
* **Subscription -** розв’язочна сутність для відношення багато до багатьох між **User** та **CurrencyMarket**. Це важлива сутність, оскільки вона є важливою частиною бізнес логіки додатку(наприклад, не можна да рази підписатись на певну валюту на певному торгівельному майданчику).

Детальніше розглянемо атрибути кожної сутності:

Таблиця 2.1 – Список атрибутів сутності **User**.

|  |  |  |
| --- | --- | --- |
| *Назва атрибуту* | *Тип атрибуту* | *Опис* |
| Id | UUID | Унікальний ідентифікатор |
| Name | TEXT | Ім’я користувача |
| ChatId | Integer | Ідентифікатор чату у Telegram |

Таблиця 2.2 – Список атрибутів сутності **Currency**.

|  |  |  |
| --- | --- | --- |
| *Назва атрибуту* | *Тип атрибуту* | *Опис* |
| Id | UUID | Унікальний ідентифікатор |
| Name | TEXT | Назва крипто валюти |
| Symbol | VARCHAR(5) | Коротка назва крипто валюти |
| ImageUrl | TEXT | Посилання на логотип валюти |

Таблиця 2.3 – Список атрибутів сутності **Market**.

|  |  |  |
| --- | --- | --- |
| *Назва атрибуту* | *Тип атрибуту* | *Опис* |
| Id | UUID | Унікальний ідентифікатор |
| Name | TEXT | Назва торгівельного майданчику |
| Provider | INTEGER | За допомогою цього поля визначається, який провайдер для агрегації буде використано |

Таблиця 2.4 – Список атрибутів сутності **CurrencyMarket**.

|  |  |  |
| --- | --- | --- |
| *Назва атрибуту* | *Тип атрибуту* | *Опис* |
| Id | UUID | Унікальний ідентифікатор |
| CurrencyId | UUID | Ідентифікатор валюти |
| MarketId | UUID | Ідентифікатор ринку |
| MarketCurrencyId | INTEGER | Ідентифікатор відповідної валюти у зовнішньому представленні торгівельному майданчику |

Таблиця 2.5 – Список атрибутів сутності **CurrencyRate**.

|  |  |  |
| --- | --- | --- |
| *Назва атрибуту* | *Тип атрибуту* | *Опис* |
| Id | UUID | Унікальний ідентифікатор |
| CurrencyMarketId | UUID | Ідентифікатор валюти |
| UpdateBatch | INTEGER | Номер агрегації валюти |
| Date | TIMESTAMP | Час проведення агрегації |
| Rate | DECIMAL | Курс валюти на час агрегації в USD |

Таблиця 2.6 – Список атрибутів сутності **Market**.

|  |  |  |
| --- | --- | --- |
| *Назва атрибуту* | *Тип атрибуту* | *Опис* |
| Id | UUID | Унікальний ідентифікатор |
| UserId | UUID | Ідентифікатор користувача |
| MarketCurrencyId | UUID | Ідентифікатор валюти на торгівельному майданчику (**CurrencyMarke**t) |

Тепер розглянемо відношення, які існують в предметній області:

* Відношення *багато до багатьох* між **Currency** та **Market** через розв’язочну сутність **CurrencyMarket**.Це відношення означає, що одна та сама валюта може бути на багатьох торгівельних майданчиках, та на одному торгівельному майданчику може бути представлено багато різних валют.
* Відношення *багато до багатьох* між **User** та **CurrencyMarket** через розв’язочну сутність **Subscription**.Це відношення означає, що один користувач може бути підписаний на декілька різних валют на декількох торгівельних майданчиках, та за оновленням певної валюти на певному майданчику може слідкувати декілька користувачів.
* Відношення *один до багатьох* між **CurrencyMarket** та **CurrencyRate.** Це відношення означає, що курс певної валюти на певному торгівельному майданчику буде оновлюватись багато разів(при кожному агрегувані).

Підводячи підсумок аналізу предметної області, можна сказати, що побудована модель повністю задовольняє вимогам, поставленим до програмного продукту. Також ця модель є простою для розуміння і модифікації, і може бути представлена за допомогою засобів ORM Entity Framework Core. Але, при розширені можливостей додатку варто врахувати наступні нюанси:

По-перше, при додаванні нових кінцевих платформ(наприклад, Facebook), варто створити нову сутність **UserProviderInfo**, яка буде відповідати за представлення користувача на різних кінцевих платформах. Це слід зробити для того, щоб надати можливість користувача прив’язати до одного облікового запису декілька засобів миттєвого обміну повідомлення. Також слід додати зв’язок один до багатьох між **User** та **UserInfoProvider**.

По-друге, не всі торгівельні майданчики будуть використовувати послідовності цілих чисел для унікального ідентифікатору валюти. Можливим рішенням є використання типу RAW або VARCHAR для атрибуту **MarketCurrencyId** в сутності **CurrencyMarket.**

# 4 ДЕТАЛЬНИЙ ОПИС АРХІТЕКТУРИ

4.1 Вимоги до архітектури системи

В умовах розробки open-source продукту до архітектуру проекти з’являються наступні вимоги:

* Простота та прозорість архітектури;
* Простота та повнота інтерфейсів програмних компонентів;
* Простота модифікації існуючого коду;
* Легкість розгортання проекту на локальній машині;
* Наявність гарної документації, або слідування принципу само-документованого коду;

Також важливим був вибір парадигми програмування. На сьогоднішній день домінуючими є **функціональна** та **об’єктно-орієнтована** парадигми. Через специфіку обраних технологій та більшу популярність, було застосовано об’єктно-орієнтовану парадигму.

4.2 Архітектура програмного продукту

Оскільки програмний продукт повинен мати змогу працювати з декількома кінцевими платформами( наприклад, бот у Telegram, чат-бот у Facebook або Twitter),то було прийняте рішення розділити бота та АПІ для агрегації та роботи з курсом крито валют. Також на етапі планування, була прийнята ідею використовувати Service Bus для оркестрування та синхронізації сервісів, але в процесі розробки цю ідею було відкинуто через високу складність реалізації цієї частини. Замість неї було використано простий task scheduler. Яка вже було сказано, система складається з двох компонентів: **бекенду боту** та **бекенду API**. Бекенд боту та API не мають стану та є REST-ful сервісами. Для зберігання даних використовується реляційна база даних.

4.3 Огляд використаних технологій

В якості цільової платформи було обрано екосистему .Net Core за зручність, наявність великої кількості бібліотек, фреймворків та крос-платформеність. В якості кінцевої платформи було обрано сервіс для миттєвого обміну повідомленнями – Telegram. Для збереження даних було обрано об’єетно-реляційну систему управління базами даних PostgreSQL. Для написання API було використано ASP .NET Core 2.1. Для роботи з БД було використано Entity Framework Core 6. Для написання боту було використано сторонній NuGet-пакет Telegram.Bot , а для task scheduler-a Quartz. Розглянемо використані технології більш детально.

4.3.1 .Net Core

**.NET Core -** це безкоштовний крос-платформний фреймворк з керованим кодом підтримуваний на Windows, Linux і Mac OSX. На відміну від .NET Framework вихідний код .NET Core є повністю відкритим. Він містить CoreCLR - повністю крос-платформну реалізацію CLR, віртуальну машину, яка керує виконанням програм в .NET середовищі. CoreCLR поставляється з оптимізованим «just-in-time» компілятором RyuJIT. .NET Core також включає в себе CoreFX, яка представляє собою часткове відгалуження FCL (стандартна бібліотека класів .NET фреймворку). Реалізації усіх класів також відкриті.

У той час як .NET Core розділяє підмножину API .NET Framework, він містить також власний API, який не є частиною .NET Framework. Крім того .NET Core містить CoreRT, оптимізований під інтеграцію в АОТ(компіляція перед виконанням) бінарні файли. Варіант бібліотеки .NET Core використовується для UWP (універсальна платформа Windows). UWP платформа, створена Microsoft і вперше представлена в Windows 10. Метою даної платформи є допомога у створенні універсальних додатків Windows, що запускаються як на Windows 10, так і на Windows 10 Mobile без зміни в коді. Інтерфейс командного рядка .NET Core пропонує точку входу для операційних систем і надає послуги для розробників, такі як компіляція і пакети управління.

.NET Core підтримує чотири крос-платформних сценарії: ASP.NET Core веб-аплікації, консольні додатки, бібліотеки і UWP (універсальна платформа Windows) додатки. Він не реалізує Windows Forms або WPF, які створюють стандартний графічний інтерфейс для настільних ПК на Windows. .NET Core також модульна, а це означає, що замість збірок, розробники працюють з пакетами NuGet. На відміну від .NET Framework, який обслуговується за допомогою служби Windows Update, .NET Core залежить від його менеджера пакетів при отриманні оновлень.

4.3.2 ASP .NET core

**ASP.NET Core -** вільне та відкрите програмне забезпечення каркасу веб застосунків, з продуктивністю вищою ніж у ASP.NET, розроблене корпорацією Microsoft. Це модульна структура, яка працює як на повній платформі .NET Framework, так і на платформі .NET Core.

Фреймворк являє собою повний перепис, який об'єднує раніше окремі ASP.NET MVC та ASP.NET Web API у єдину програмувальну модель.

Не зважаючи на те, що це є новим фреймворком, побудованим на новому веб-стеку, ASP.NET Core має високу ступінь сумісності концепцій з ASP.NET MVC, який об'єднує функціональність MVC, Web API та Web Pages. В попередніх версіях платформи дані технології реалізовані окремо і тому містять багато дубльованої функціональності. Тепер це об’єднано в одну програмну модель ASP.NET Core MVC. Веб-форми повністю вийшли в минуле. Програми ASP.NET Core підтримують програмні версії, в якій різні програми, що працюють на одному комп'ютері, можуть орієнтуватися на різні версії ASP.NET Core. Це не можливо з попередніми версіями ASP.NET Core.

4.3.3 PostgreSQL

**PostgreSQL -** об'єктно-реляційна система керування базами даних(СКБД). Є альтернативою як комерційним СКБД (Oracle Database, Microsoft SQL Server, IBM DB2 та інші), так і СКБД з відкритим кодом (MySQL, Firebird, SQLite).

Порівняно з іншими проектами з відкритим кодом, такими як Apache, FreeBSD або MySQL, PostgreSQL не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення.

Сервер PostgreSQL написаний на мові C. Зазвичай розповсюджується у вигляді набору текстових файлів із сирцевим кодом. Для інсталяції необхідно відкомпілювати файли на своєму комп'ютері і скопіювати в деякий каталог. Весь процес детально описаний в документації.

Головними перевагами PostgreSQL є:

* **Функції**

Функції дозволяють виконувати деякий код безпосередньо сервером бази даних. Ці функції можуть бути написані на SQL, який має деякі примітивні програмні оператори, такі як галуження та цикли. Але гнучкішою буде функція написана на одній із мов програмування, з якими PostgreSQL може працювати. Функції можуть виконуватись із привілеями користувача, який її викликав, або із привілеями користувача, який її написав;

* **Індекси**

У PostgreSQL є підтримка індексів наступних типів: B-дерево, хеш, R-дерево, GiST, GIN. При необхідності можна створити нові типи індексів;

* **Багатоверсійність (MVCC)**

PostgreSQL підтримує одночасну модифікацію БД декількома користувачами за допомогою механізму Multiversion Concurrency Control (MVCC). Завдяки цьому виконуються вимоги ACID, і практично відпадає потреба в блокуванні зчитування;

* **Типи даних**

PostgreSQL підтримує великий набір вбудованих типів даних. Крім того, користувач може самостійно створювати нові необхідні йому типи та програмувати для них механізми індексування за допомогою GiST;

* **Об'єкти користувача**

PostgreSQL може бути розширено користувачем для власних потреб практично в будь-якому аспекті;

* **Успадкування**

Таблиці можуть успадковувати характеристики та набори полів від інших таблиць (батьківських). При цьому дані, які додаються до породженої таблиці, автоматично будуть брати участь (якщо це не вказано окремо) в запитах до батьківської таблиці. Цей функціонал в поточний час не є повністю завершеним. Однак він достатній для практичного використання;

* **Тригери**

Тригери визначаються як функції, що ініціюються DML-операціями. Наприклад, операція INSERT може запускати тригер, що перевіряє доданий запис на відповідність певним умовам. Тригери можна писати різними мовами програмування. Вони пов'язані з визначеною таблицею. Множинні тригери виконуються в алфавітному порядку;

4.3.4 Entity Framework Core

Entity Framework (EF) Core - це легка, розширювана та крос-платформенна версія популярної технології доступу до даних Entity Framework.

EF Core може слугувати об'єктно-реляційним маппером (O / RM), що дозволяє розробникам .NET працювати з базою даних, що використовує об'єкти .NET, і виключає необхідність більшості коду доступу до даних, який вони, як правило, повинні писати. EF Core підтримує багато СУБД.

4.3.5 Telegram bot API

ЕTelegram bot API дозволяє підключати ботів до системи миттєвого обміну повідомленнями. Telegram Bots - це спеціальні облікові записи, для яких не потрібно встановлювати додатковий номер телефону. Ці облікові записи слугують інтерфейсом для коду, що працює десь на зовнішньому сервері.

Для цього не потрібно нічого знати про те, як працює протокол шифрування MTProto – сервер-посередник буде обробляти всі шифрування та спілкування з API Telegram. Зовнішній сервер зв'язуєтеся з цим сервером за допомогою простого HTTPS-інтерфейсу, який пропонує спрощену версію API Telegram.

4.3.6 REST

**REST –**  підхід до архітектури мережевих протоколів, які забезпечують доступ до інформаційних ресурсів. В основі REST закладено принципи функціонування Всесвітньої павутини і, зокрема, можливості HTTP. REST було розроблено паралельно з HTTP 1.1 базуючись на попередньому протоколі HTTP 1.0.

Дані повинні передаватися у вигляді невеликої кількості стандартних форматів (наприклад, HTML, XML, JSON). Будь-який REST протокол (HTTP в тому числі) повинен підтримувати кешування, не повинен залежати від мережевого прошарку, не повинен зберігати інформації про стан між парами «запит-відповідь». Стверджується, що такий підхід забезпечує масштабовність системи і дозволяє їй еволюціонувати з новими вимогами.

4.4 Огляд структури бота

При попередньому проектуванні, планувалась наявність додаткового компоненту системи: Service Bus-у. Цей компонент відповідав за орекестрацію та синхронизацію роботи бота та API. Але при більш детальному вивчені предметної області через складність реалізації двоетапних транзакцій було вирішено відкласти цей компонент на наступну ітерацію розробки, і замість нього використати простіший у реалізації Task Scheduler.

Бот є прошарком між API та користувачем. Він перетворює команди користувача на запити до API, обробляє відповідь та подає у зручному для користувача представлені. Також бот віддає команду на агрегацію зовнішніх ресурсів та розсилає оновлення курсу валют.

Для більш глибшого розуміння структури надалі буде розглянута структура найважливіших неймспейсів та інтерфесів боту.

4.4.1 Неймспейс Bot.Routers

Цей неймспейс є центральним у боті. У ньому знаходиться все необхідне для маршрутизації команд, отримання вхідних даних та виконання дій в залежності від маршруту. Головними елементами у цьому неймспейсі є:

* **Інтерфейс IRouter**

Містить наступні методи:

1. IRouter Bind(string route, Func<ParameterBag,IReply> action, string name = null) – метод для прив’язки делегату до маршруту. Повертає екземпляр IRouter для подальшо чейнінгу.
2. IReply Dispatch(string route) – виконує команду і повертає результат, придатний для відправки користувачу
3. Route GetRouteByName(string name) – повертає абстракцію маршруту згідно з прив’язаним ім’ям.

Цей інтерфейс є кінцевою абстракцію роутера;

* **Інтерфейс IRouterExpressionParser**

Абстракція, яка відповідає за перетворення маршруту до кінцевого регулярного виразу. Містить лише метод string ParseExpression(string command), який на вхід приймає строкове представлення маршруту, а повертає регулярний вираз.

* **Клас Route** – представляє абстракцію над аршрутом. Містить наступні властивості:

1. public string Name { get; set; } – назва маршруту;
2. public string RawRoute { get; set; } – необроблене представлення маршруту;
3. public string CompiledRoute { get; set; } – маршрут, перетворений на регулярний вираз;
4. public Func<ParameterBag, IReply> Handler { get; set; } – коллбек, який буде викликаний при обробці маршруту;

Також цей класс містить допоміжні функції для встановлення одного або деклькох значень маршруту;

* **Клас ParameterBag** – представляє собою абстракцію, яка зберігає параметри команди;

Також у цьому неймспейсі присутні реалізації відповідних інтерфейсів.

* + 1. Неймспейс Bot.Exceptions

У цьому неймспейсі знаходяться всі виключення, що можуть відбутись в ході виконання програми. Перелік классів:

* **DomainException** – базове виключення, яке представляэ помилку предметної області;
* **RouteException** – виключення, яке представляє те, що маршрут не було знайдено в маршрутизаторі;
* **RouteParamsException** – виключення, яке виникає при спробі використати невірний формат для строки маршруту;
  + 1. Неймспейс Bot.Entities

Представляє з собою необхідні сутності предметної області. Містить наступні класи:

* **Currencies** – представлення криптовалюти;
  + 1. Неймспейс Bot.APIs

Представляє собою абстракції над бекендом API. Містить в собі інтерфейс **I**API. Цей інтерфейс пропонує наступні методи:

* Task<SubscriptionDTO> AddSubscription(int userId, int currencyId, int marketId) – підписатись на оновлення певною валюти;
* Task<CurrencyRateDTO> GetCurrencyRate(int currencyId, int? marketId) – отримати поточний курс валюти;
* Task<AvailableCurrenciesDTO> GetAvailableCurrencies() – отримати список доступних криптовалют(для підписки або отримання поточного курсу);

Також цей неймспейс містить вкладений неймспейс з об’єктами, які представляють відповіді з серверу – Bot.APIs.DTOs. Оскільки, вони принципово не важливі для розуміння архітектури їх перелік і деталі реалізації будуть пропущені.

4.5.5 Неймспейс Bot.Services

Цей неймспейс слугує для групування колбеків для маршрутів по класам. Перелік класів не наводиться, оскільки вони не важливі для розуміння архітектури.

* 1. Огляд структури API

Оскільки API побудовано за допомогою фреймворку ASP .NET Core, який реалізує архітектурний паттерн MVC, у архітектурі API немає особливостей, на які варто було б звернути увагу. Єдиним неймспейсом, який заслуговує уваги є BotAPI.Aggregators. Цей клас відповідає за агрегацію зовнішніх API.

//TO DO: перелік інтерфейсів