КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп’ютерних наук та кібернетики

Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп’ютерні науки

на тему:

ГЕНЕРАЦІЯ WEB API БІБЛІОТЕК

Виконав студент 4 курсу

Олексій ТИЧКОВСЬКИЙ __________

(підпис)

Науковий керівник:

асистент, кандидат інженерних наук

Олексій ФЕДОРУС __________

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент ______________
(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри математичної інформатики
«____» _________________
2023 р., протокол №_____
Завідувач кафедри ______________
(підпис)
ТЕРЕЩЕНКО Василь

Київ – 2023

РЕФЕРАТ

Обсяг роботи: 49 сторінок, 28 ілюстрацій, 13 джерел посилань.

Об'єктом даної роботи є розробка механізму генерації Web API бібліотеки для існуючого додатку у сфері Fin-Tech. Основною метою проекту є дослідження наявних інструментів для генерації Web API бібліотек та створення програмного забезпечення для покращення функціональності вже існуючого додатку.

Для досягнення поставленої мети, в процесі розробки використовуються такі методи: аналіз існуючих підходів до генерації Web API бібліотек, розробка та тестування програмного забезпечення. Для зручності розробки та програмування використовується операційна система Windows 10 та середовище програмування PyCharm.

Одним з основних результатів даної роботи є створення зручного інструменту, який автоматизує сценарії використання додатку та полегшує аналіз отриманих результатів. Розроблений програмний продукт вже успішно використовується на реальних проектах, що свідчить про його ефективність та потенціал для впровадження в інші області.

У підсумку, розробка генератора Web API бібліотеки для існуючого додатку в сфері Fin-Tech є важливим кроком у поліпшенні функціональності та розширенні можливостей цього додатку. Результати роботи виявилися успішними, а отриманий програмний продукт вже знайшов своє застосування на практиці.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА ВИЗНАЧЕНЬ **4**

ВСТУП **5**

РОЗДІЛ 1. ОПИС ОСНОВНОГО ДОДАТКУ **7**

1.1. Загальний огляд 7

1.2. Архітектура додатка 9

1.3. Дані для генерації бібліотеки 10

РОЗДІЛ 2. ОПИС АРХІТЕКТУРИ БІБЛІОТЕКИ **12**

2.1. Серіалізатор 12

2.2. Десеріалізатор 15

2.3. Об'єкт контекст 16

2.4. Обгортка методів 18

2.5. Авторизація та сертифікати 19

2.6. Web Api Storage 23

РОЗДІЛ 3. РОЗРОБКА ГЕНЕРАТОРА КЛАСІВ **25**

3.1. Парсинг даних для генерації. 25

3.2. Генерування Python класів. 27

3.3. Генерування init файлів. 31

3.4. Написання генератора документації для згенерованих класів 32

3.5. Написання упакування згенерованих класів в бібліотеку. 34

РОЗДІЛ 4. ПОВНА АВТОМАТИЗАЦІЯ ЗБІРКИ **36**

4.1. Автоматичне вирішення проблем циклічного імпорту. 36

4.2. Автоматичний збір пакетів у коректну бібліотеку. 38

РОЗДІЛ 5. ПРИКЛАДИ ВИКОРИСТАННЯ **41**

5.1. Збереження нових об'єктів до бази данних. 41

5.2. Виклик методів для існуючих об'єктів. 43

5.3. Імпорт результатів та їх аналіз в Python. 44

ВИСНОВКИ **47**

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ **48**

ПЕРЕЛІК СКОРОЧЕНЬ ТА ВИЗНАЧЕНЬ

API - Application Programming Interface

JSON - JavaScript Object Notation

XML - eXtensible Markup Language

MTLS - Mutual Transport Layer Security

ADFS - Active Directory Federation Services

MSAL - Microsoft Authentication Library

ВСТУП

У сучасному інтегрованому цифровому середовищі, веб-інтерфейси API (інтерфейси прикладного програмування) здійснюють критичну роль у плавному обміні даними між різноманітними програмними системами. Враховуючи очікуваний попит на ефективні та надійні API, розробка бібліотек веб-аплікацій стає пріоритетною. Проблема є актуальною і описується, наприклад, в такій науковій статті [1], а в статті [2] описується аналіз генерація коду для аналізу API.

У контексті розробки веб-аплікацій, виникає виклик забезпечити ефективну та зручну взаємодію з API. Існує широкий спектр інструментів та бібліотек, призначених для полегшення створення та використання API.

Swagger[3] - це один з цих інструментів, що дозволяє автоматично генерувати документацію для веб-інтерфейсу API на основі анотацій у вихідному коді. Swagger пропонує зручні можливості для створення запитів до API, перевірки параметрів, документації ресурсів та їх параметрів. Проте, його ефективність може знижуватися у складних сценаріях, наприклад, при необхідності отримання інформації про всіх наслідників класу, який є параметром запиту, та рекурсивної обробки цієї інформації.

Також існують інші інструменти, які полегшують розробку та використання веб-інтерфейсів API. Наприклад, Postman[4] – це відомий інструмент для тестування та документації API, що дозволяє виконувати запити, тестувати їх, зберігати історію запитів та генерувати документацію на основі виконаних запитів.

Insomnia[5] відзначається як потужний інструмент для взаємодії з API, який надає можливості тестування, автоматичного документування та генерації коду для різних мов програмування.

Незважаючи на ці інструменти, важливо підкреслити, що вони не завжди можуть забезпечити повну автоматизацію всіх аспектів роботи з API. Ситуації, коли вимагається більш гнучка система для автоматизації взаємодії з API та документування його можливостей, є не рідкістю. В цій дипломній роботі представлено розробку механізму генерації бібліотеки для взаємодії з вже існуючим додатком, що надає можливість ефективної автоматизації сценаріїв взаємодії з додатком, аналізу результатів за допомогою Python бібліотек та безпосереднього взаємодію з базою даних.

РОЗДІЛ 1. ОПИС ОСНОВНОГО ДОДАТКУ

1.1. Загальний огляд

Програмний комплекс, що розглядається, репрезентує собою інтегровану обчислювальну платформу для комплексного аналізу та управління фінансовими ризиками. Ця платформа надає своїм користувачам розширений набір можливостей для створення, конфігурації та аналізу фінансових сутностей. Вона надає засоби для використання вже наявних фінансових структур, а також для створення нових, що дозволяє відповідати на різноманітні вимоги фінансового аналізу, включаючи гнучкість, персоналізацію та глибоку кастомізацію.

Цей додаток розроблено з урахуванням концепції середовищ (environments), які відображають різні фінансові сценарії та умови. Для кожного такого середовища передбачено зберігання відповідних даних у базі даних MongoDB, що гарантує надійне та оптимальне зберігання великих обсягів інформації.

Дані в системі класифікуються як загальноприйняті (common used) та специфічні дані, що стосуються конкретних дат використання. Завдяки структурі бази даних, для кожної фінансової сутності формується відповідний набір даних (dataset), що містить відомості, прив'язані до певної дати. Ця організація даних дозволяє користувачам з легкістю використовувати актуальні дані, а також створювати власні набори даних.

Можливості створення даних реалізовані через графічний інтерфейс користувача (GUI) та спеціалізований інструмент "Snapshot manager". Даний менеджер надає можливість формувати знімки даних на певну дату, при цьому

всі дані, що відповідають цьому знімку, будуть збережені в відповідному датасеті.

Результати обчислень представлені у відповідних viewer-ах що сприяє їх наочному відображенню та аналізу. У цілому, ця платформа є міцним інструментом для ефективного управління фінансовими ризиками та аналізу даних, комбінуючи в собі зручний інтерфейс користувача, гнучкість конфігурації та розширені можливості обробки фінансової інформації.

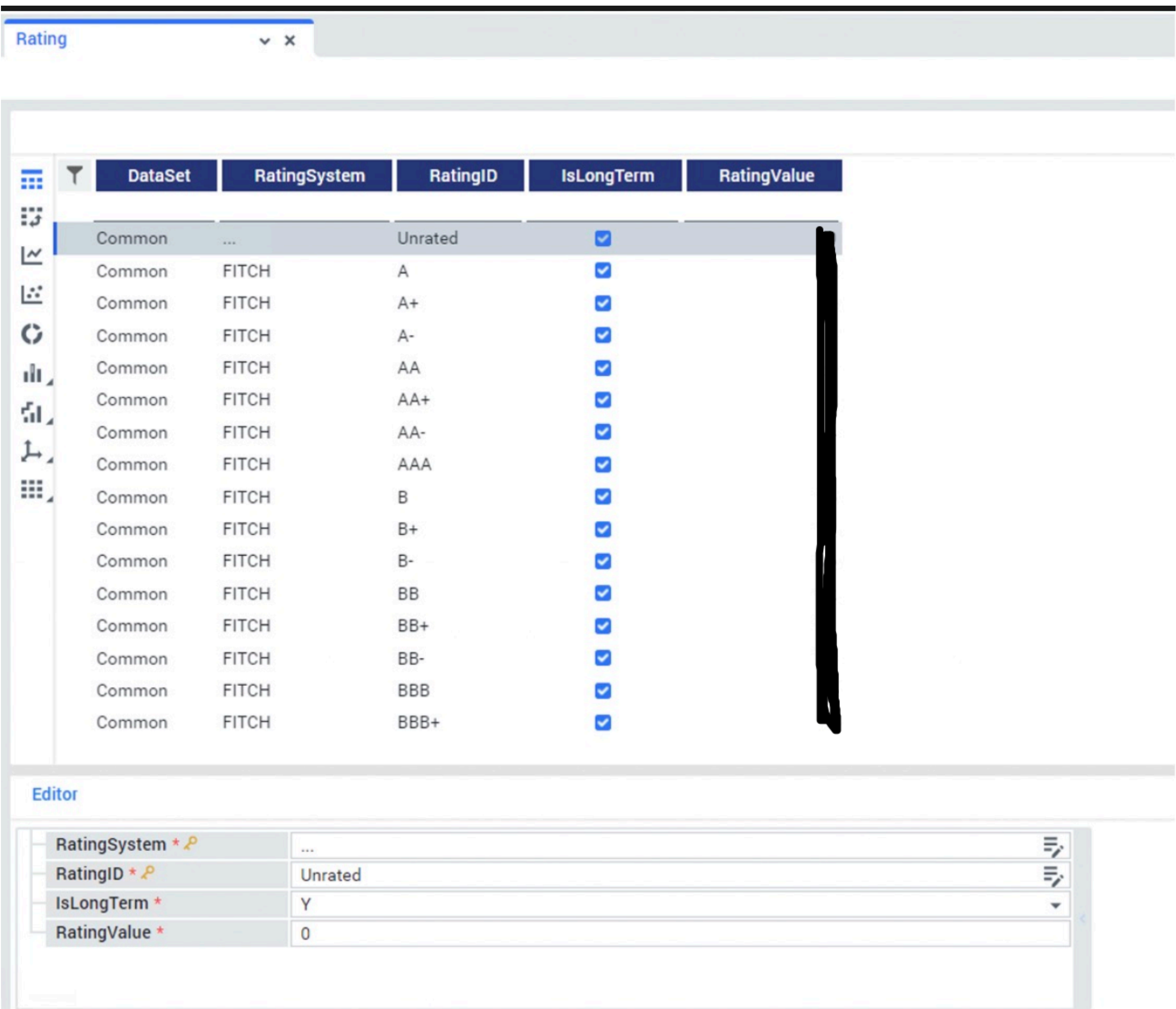

Рисунок 1.1 - Приклад сутностей Rating на графічному інтерфейсі

1.2. Архітектура додатка

Архітектура додатку використовує пакетну структуру, де кожен пакет представляє собою сукупність класів, що відповідають певній функціональній області. Наприклад, пакет "finance" містить класи, реалізовані для взаємодії з стандартними фінансовими інструментами.

З метою оптимізації процесу розширення та модифікації, всі класи унаслідуються від базового класу "Data". Це забезпечує уніфікацію базових функцій та спільних атрибутів для всіх класів у системі.

Для підвищення гнучкості роботи з даними, деяка інформація генерується автоматично, а структура і властивості кожного класу описуються відповідним XML-файлом. Ці декларації використовуються для генерації веб-інтерфейсу, що відображає дані класу та надає користувачам можливість взаємодії з ними.

Така архітектура забезпечує модульність, можливість розширення та гнучкість роботи з даними. Кожен пакет має чітко визначену область відповідальності, а наслідування від базового класу спрощує процес розробки і підтримки. XML-файли виконують роль документації для класів і слугують джерелом інформації для створення інтерфейсу користувача.

Додаток також надає універсальний механізм виклику методів будь-якого об'єкта, достатньо передати ключ об'єкта, назву методу і необхідні параметри. Це надає гнучкість при взаємодії з об'єктами, дозволяючи динамічно викликати методи без потреби в написанні спеціалізованого коду.

Альтернативно, додаток може використовувати окремі декларації для кожного методу, що включає вказівку на кінцеву точку для запиту і структуру

запиту. Такий підхід дозволяє ясно визначити доступні методи, необхідні параметри і очікувані результати.

Обидва підходи, як універсальний, так і на основі декларацій, надають гнучкість і різні можливості для взаємодії з об'єктами і виклику їх методів, відповідаючи різним потребам і особливостям додатку.

1.3. Дані для генерації бібліотеки

Архітектура додатку визначає використання XML-декларацій для детального опису структури класів і їх властивостей. Ці декларації включають інформацію про поля класу, такі як назва, опис, тип та модуль, в якому знаходиться тип поля, і вказують, чи є це поле ключовим або необов'язковим.

Ієрархію успадкування класів також вказано в деклараціях, що дозволяє розрізнити спадковість класу від інших класів. Це фактично означає, що кожен клас успадковує властивості та методи від свого батьківського класу, спрощуючи процес розробки та управління функціональністю.

Методи класу та їх параметри також відображаються в деклараціях. Для кожного методу визначаються параметри, включаючи їх типи та обов’язковість.

За допомогою такої архітектури, XML-декларації стають головним джерелом інформації для генерації даних. Ці декларації надають детальний і структурований опис класів та їх властивостей, що забезпечує єдність і точність даних. Використання XML-декларацій у поєднанні з архітектурою додатка сприяє ефективному управлінню даними та полегшує розробку інтерфейсу користувача.

Рефлексія[10] в програмуванні пропонує обширні можливості для автоматичного збору інформації про класи та їх структуру. Це дозволяє отримувати доступ до полів, методів, конструкторів, анотацій і навіть узагальнених типів.

Застосування рефлексії для генерації даних є особливо корисним, коли потрібно автоматично створювати об'єкти зі складною структурою або генерувати дані для серіалізації. За допомогою рефлексії можна переглянути всі поля класу, отримати їх назви, типи та інші важливі відомості. Наприклад, використовуючи рефлексію, можна автоматично генерувати значення для цих полів на основі типу поля, стандартних значень або власних правил.

Також, рефлексія дає можливість доступу до методів класу, що відкриває широкі можливості для автоматичного виклику методів з заданими параметрами. Це особливо корисно, коли є багато класів зі схожою логікою, але з різними параметрами.

Крім того, рефлексія дозволяє отримувати інформацію про успадкування класів, що дає змогу автоматично переглянути всю ієрархію успадкування та отримувати всі класи, які успадковані від певного батьківського класу.

Використання рефлексії для генерації даних дозволяє автоматизувати процес створення об'єктів та заповнення їх даними. Це зменшує кількість повторюваного коду і спрощує розробку, особливо в ситуаціях, коли структура даних змінюється часто або коли є велика кількість класів з багатьма полями.

РОЗДІЛ 2. ОПИС АРХІТЕКТУРИ БІБЛІОТЕКИ

2.1. Серіалізатор

Серіалізатор у Python, що призначений для створення коректних запитів до API головного додатку, є критично важливим компонентом бібліотеки. Враховуючи, що формат відповідей головного додатку може бути специфічним, серіалізатору необхідно адаптуватися для обробки особливих класів та серіалізації полів із стилю Python до стилю camelCase.

Ключовим аспектом цього серіалізатора є можливість налаштованої обробки специфічних класів. Оскільки головний додаток може використовувати свої класи із особливою логікою, серіалізатор повинен бути модифікований, щоб урахувати ці класи та правильно серіалізувати їх у формат JSON.

Крім того, серіалізатор повинен також обробляти конвертацію полів із стилю Python до стилю camelCase. Враховуючи, що в C# назви полів часто записуються в стилі camelCase, серіалізатор повинен гарантувати перетворення назв полів для відповідності цьому стилю, щоб забезпечити сумісність з головним додатком.

Іншою важливою характеристикою є перевірка повноти заповнення всіх полів, які не марковані як необов’язкові. Це забезпечує послідовність даних і допомагає уникнути проблем при обробці запитів до API. Серіалізатор повинен перевіряти, чи всі обов'язкові поля містять правильну інформацію перед відправкою запиту.

Крім того, серіалізатор виконує перевірку типів полів, тому що Python є динамічно типізованою мовою, яка не гарантує статичну типізацію.

Серіалізатор повинен перевіряти, чи всі поля мають правильні типи даних, щоб уникнути можливих помилок при взаємодії з головним додатком.

Основною метою серіалізатора є формування правильних запитів до API головного додатку, з урахуванням специфічних вимог і забезпечення правильності та послідовності даних.

На додаток до вищезазначеної інформації, роль серіалізатора в Python може бути важливою для візуалізації даних в середовищі Python. За допомогою серіалізатора, ми можемо перетворити об'єкти в формат JSON, а потім створити з даних JSON DataFrame - потужну структуру даних для зручного відображення та аналізу в Python. Таким чином, серіалізатор відіграє важливу роль у перетворенні та обробці даних, що дозволяє гнучко використовувати дані у широкому спектрі аналітичних та візуалізаційних задач.

Перетворення об'єктів в формат JSON дозволяє нам легко передавати та обмінюватися даними між різними системами або мовами програмування. JSON є популярним форматом обміну даними, оскільки його можна легко зрозуміти та обробити.

Далі, за допомогою серіалізатора, ми можемо перетворити об'єкти в JSON-рядки, зберігаючи структуру та типи даних. Після цього, з використанням відповідних бібліотек, ми можемо зручно перетворити JSON-дані в DataFrame. DataFrame - це потужна структура даних в Python, яка надає розширені можливості для маніпулювання та аналізу даних.

DataFrame забезпечує зручний спосіб відображення даних у форматі, зрозумілому для користувача. Він дозволяє виконувати різноманітні операції над даними, такі як фільтрація, сортування, групування та агрегація. Крім того, DataFrame може бути використаний для створення графіків, візуалізації та детального аналізу даних в середовищі Python.

Таким чином, серіалізатор відіграє важливу роль у візуалізації та аналізі даних в середовищі Python, дозволяючи зручно перетворювати об'єкти в формати, які найбільш підходять для відображення та обробки даних. Це робить процес роботи з даними більш ефективним та зручним для розробників.

Щодо реалізації серіалізатора то в цілому все досить стандартно, з самого класу отримаємо данні про кожне з полів і далі якщо це поле також згенерований клас то робимо аналогічне рекурсивно інакше ж в залежності від типу, назвемо його примітивним буде обчислено значення. З кастомних моментів в даному випадку буде типу Variant, який в собі містить тип і значення, також формат дати та дати + часу. Також для стиснення розміру використовується вказування типу лише за умови що це наслідник типу, а інакше не вказується відповідно. І відповідно до зарезервованності деяких слів (наприклад in, type, при генерації до назви цих полів додано _ і враховано це при серіалізації.

Загалом, процес серіалізації полягає у зборі даних про кожне поле з класу та його значення. Якщо поле є об'єктом власного класу, то проводиться рекурсивний виклик серіалізатора для цього об'єкту. У випадку примітивного типу поля, значення обчислюється відповідно до типу поля.

Одним зі специфічних аспектів реалізації є підтримка типу "Variant", який містить інформацію про тип та значення. Цей тип дозволяє гнучку обробку поліморфних даних, коли тип значення може змінюватись. Крім того, серіалізатор повинен правильно обробляти типи дат, такі як дата та дата + час для забезпечення коректної серіалізації цих типів.

Для економії розміру серіалізованих даних використовується вказівка типу лише у випадку, коли поле є нащадком відповідного типу. У протилежному випадку, тип не вказується для поля. Це дозволяє скоротити обсяг запиту та оптимізувати процес серіалізації.

Крім того, враховується зарезервованість деяких слів, наприклад "in" або "type". При генерації полів з такими назвами, до них додається підкреслювання (_) для уникнення конфліктів та забезпечення коректності серіалізації.

Такі підходи дозволяють розширити можливості серіалізатора та забезпечити його адаптивність до різних типів даних та специфічних вимог. Вони сприяють ефективному та надійному обміну даними між різними системами та забезпечують коректність та консистентність серіалізованих даних.

Також іноді данні можуть бути некоректними, але все таки є потреба їх оброблювати, для цього було додано спеціальний флаг, який би не перевіряв що всі обов’язкові поля заповнені.
Для зручності реалізації створено клас TypeDecl, який і зберігатиме всю інформацію про кожен з класів і буде далі використовуватися для десерелізації і генерації даних.

2.2. Десеріалізатор

Десеріалізатор у бібліотеці відіграє таку саму важливу роль, як і серіалізатор, відповідаючи за коректне розбирання та перетворення вхідних даних з формату JSON у внутрішні об'єкти або структури даних в середовищі Python. Додатково до цього, він виконує перевірку на заповненість всіх обов'язкових полів та перевірку типів полів.

Важливим механізмом, який використовує десеріалізатор, є рефлексія. Вона дозволяє динамічно отримувати інформацію про типи об'єктів, що спрощує процес знаходження потрібного класу для перетворення даних. При отриманні вхідних даних, десеріалізатор обробляє всі імпортовані типи, які

унаслідуються від Data, і за допомогою рефлексії він знаходить потрібний клас для десеріалізації.

Така система дозволяє автоматично переводити поля з camelCase-стилю у стиль Python (snake_case), що відповідає особливостям Python-екосистеми.

Загальна мета десеріалізатора полягає в забезпеченні правильної обробки та конвертації вхідних даних, щоб забезпечити їх використання в основному додатку. Десеріалізатор має застосовуватися у випадках, коли отримані дані потрібно розібрати та використати у внутрішніх процесах програми.

У своїй роботі десеріалізатор доповнює функціонал серіалізатора, забезпечуючи зворотний процес перетворення даних з формату JSON у внутрішні об'єкти Python. Це стає основою для правильної та надійної обробки даних у взаємодії з API основного додатку, забезпечуючи консистентність та коректність даних при їх розборі.

2.3. Об’єкт контекст

Важливим елементом для написання скриптів є контекст, оскільки користувачеві важливо забезпечити надсилання запиту на правильний URL, до правильного датасету та у відповідному середовищі (environment). Контекст включатиме в себе інформацію про згадані параметри і буде слугувати місцем збереження цієї інформації. Крім того, в ньому будуть розташовані методи для генерації самостійних запитів.

Основною задачею цих методів буде формування коректних запитів згідно з архітектурою системи. У даному випадку, це означатиме правильне встановлення ключа об'єкта, вказівку методу та його параметрів. Ці методи

будуть відповідати за створення правильно структурованих запитів, що дозволить забезпечити правильну взаємодію з API основного додатку.

В контексті можуть бути реалізовані різноманітні методи для генерації запитів, залежно від потреб користувача. Наприклад, можуть бути методи для створення GET- або POST-запитів з відповідними параметрами. Контекст буде відповідати за створення правильно сформованих запитів на основі наданої інформації та забезпечення їх відправки до API. Приклад формування запиту зображено на рисунку 2.1.

```
json = {
    "Action": {
        "Type": {
            "Module": {"ModuleName": _get_package_from_module(handler.__module__)},
            "Name": caller_name,
        },
        "Handler": to_pascal_case(handler_name),
        'Arguments': _process_method_arguments(handler, arguments),
    }
}
```

Рисунок 2.1 - Формування запиту для API

Крім формування запитів, контекст може також включати різноманітні методи для обробки отриманих відповідей. Наприклад, це можуть бути методи для розбору та аналізу отриманих даних, перетворення їх у зручний для подальшого використання формат або виконання додаткових операцій з отриманими результатами.

В цілому, контекст грає важливу роль у забезпеченні правильної взаємодії з API основного додатку. Він допомагає зберегти та керувати інформацією про параметри, необхідні для формування та виконання запитів.

Також він спрощує процес формування коректних запитів за допомогою наданих методів, що забезпечує зручність та ефективність роботи з API.

Далі ще вказати додатково параметри, вказані вище і надіслати запит до коректної кінцевої точки. Для цього просто була використана бібліотека requests[11].

2.4. Обгортка методів

Для полегшення генерації класів з методами можна використати декоратори в Python. Це зручний підхід, який дозволяє додавати функціональність до існуючих методів, не змінюючи їх вихідний код.

Декоратор можна написати для кожного методу, який відповідає за відправку коректного запиту з використанням методу та параметрів. При використанні декоратора, він буде застосовуватися до відповідних методів автоматично, надаючи їм необхідну функціональність для взаємодії з API.

Python надає підтримку декораторів функцій, що є потужним інструментом для розширення функціональності методів. За допомогою декораторів можна додати додаткову логіку до методів, таку як формування коректних запитів, встановлення необхідних параметрів, обробка відповідей та багато іншого.

Написання декоратора для методу дозволить створити шаблон, який можна застосовувати до багатьох методів, що відповідають за взаємодію з API. Це спростить процес генерації класів з методами і забезпечить консистентність та правильність формування запитів.

З використанням декораторів, код може мати більшу модульність та повторне використання. Можливо розробити декоратор, який приймає на вхід необхідну інформацію про метод та параметри запиту, і відповідно

налаштовує цей метод для взаємодії з API. Такий підхід спрощує розширення функціональності та забезпечує більш гнучку архітектуру програми.

Використання декораторів функцій в Python дозволить створити ефективний та зручний механізм для генерації класів з методами, що забезпечуватимуть коректну взаємодію з API основного додатку.

Відповідна обгортка зображена на рисунку 2.2.

```python
def cl_handler():
    def wrap(method: Callable):
        @wraps(method)
        def run_client(*args, **kwargs):
            self_param, params = _get_parameters(method, args, kwargs)

            client = params['context'].client
            params.pop('context')

            if self_param is not None:
                return client._run_handler_internal(method, params, self_param)

            return client._run_action_internal(method, params)

        wrapped_method = run_client

        wrapped_method._cl_handler = True
        return wrapped_method
    return wrap
```

Рисунок 2.2 - Обгортка методів

2.5. Авторизація та сертифікати

На певній стадії розвитку проєкта, його почали розгортати в хмарне середовище, і, звичайно, з'явилася необхідність використання сертифікату для перевірки HTTPS-запитів. Окрім цього, також знадобилася можливість надавати MTLS-ключ та сертифікат для авторизації.

Для забезпечення зручності та гнучкості використання, ці можливості були реалізовані за допомогою підходу впровадження залежностей (dependency injection). Такий підхід дозволяє впроваджувати різні реалізації сертифікатів та ключів безпосередньо в запити, а також забезпечує можливість авторизації.

Щодо сертифікатів та ключів, які не змінюються протягом тривалого часу, їх достатньо передати у запит. Це забезпечує зручну та безпечну автентифікацію при взаємодії з сервером.

Однак, у випадку авторизації ситуація трохи відрізняється. Клієнт використовував ADFS тому було реалізовано два варіанти отримання токену авторизації.

Перший варіант - менш безпечний, використовувався бібліотекою MSAL[7] та явним введенням логіна та пароля. Цей спосіб використовувався для повної автоматизації та забезпечення функціональності клієнта.

Другий варіант передбачав написання сервісу, наприклад за допомогою фреймворку Flask[14] через який клієнт мав спочатку авторизуватися. Сервіс отримував auth token, а потім мав можливість обмінювати його на refresh token, який необхідний для подальших запитів.

Таким чином сервіс має 3 кінцеві точки, auth, callback та token.

Реалізація кінцевої точки auth зображена на рисунку 2.3, ця точка перенаправляє користувача на сам сервіс вказувавши відповідний айді додатка та адресу перенаправлення після логування.

```
@app.route('/auth')
def auth():
    auth_url = f"{ADFS_AUTHORIZE_URL}?response_type=code&client_id={CLIENT_ID}&redirect_uri={REDIRECT_URI}"
    return redirect(auth_url)
```

Рисунок 2.3 - Кінцева точка auth.

Відповідно реалізація кінцевої точки callback, на яку перенаправить auth зображена на рисунку 2.4.

```
@app.route('/callback')
def callback():
    global auth_code
    auth_code = request.args.get('code')
    return 'Successfully received auth code. You can now use /token endpoint to get the token.'
```

Рисунок 2.4 - Кінцева точка callback.

І остання кінцева точка token, для обміну токена авторизації на поновлювальний токен зображена на рисунку 2.5

```python
@app.route('/token')
def token():
    if not auth_code:
        return 'No authorization code received', 401

    try:
        data = {
            'client_id': CLIENT_ID,
            'code': auth_code,
            'redirect_uri': REDIRECT_URI,
            'grant_type': 'authorization_code'
        }
        headers = {'Content-Type': 'application/x-www-form-urlencoded'}
        response = requests.post(ADFS_TOKEN_URL, data, headers=headers)
        response.raise_for_status()
        auth_token = response.json()['access_token']
        print(f"Obtained access token: {auth_token}")
        return {'token': auth_token}
    except requests.exceptions.HTTPError as error:
        print(f"Error during the token exchange: {error}")
        return "Error during the token exchange", 500
```

Рисунок 2.5 - Кінцева точка callback

Перший спосіб, незважаючи на його недоліки, був необхідним для повної автоматизації та задоволення певного функціоналу клієнта. Завдяки такому підходу, було досягнуто зручності та гнучкості при авторизації та взаємодії з сервером.

Так як це було зроблено за допомогою впровадження залежностей і середа запуску може було різною, як локально так і в клауді то логічним кроком є написання декількох конфігураторів залежностей і їх використання на початку скриптів.

Написання декількох конфігураторів залежностей дає можливість налаштовувати залежності для різних середовищ і режимів роботи.

Наприклад, можна мати окремий конфігуратор для локальної розробки та інший для хмарного середовища. Конфігуратори залежностей містять інформацію про те, які компоненти бібліотеки потрібно використовувати та як їх налаштувати для відповідного середовища.

Використання конфігураторів залежностей спрощує налаштування та управління залежностями у різних середовищах, дозволяючи швидко переключатися між різними конфігураціями. Це підвищує гнучкість та переносимість бібліотеки, спрощує її розгортання та використання в різних умовах.

2.6. Web Api Storage

Початково виникла необхідність взаємодії з базою даних напряму, і було прийнято рішення використовувати прямі запити до бази даних. Однак, такий підхід виявився незручним для клієнтів через складнощі в налаштуванні конфігурації для з'єднання з базою даних. Щоб полегшити використання та зробити інтерфейс більш зрозумілим, був реалізований сервіс для виконання запитів до бази даних, а також відповідний клас для його використання в Python.

У реалізації цього сервісу було визначено стандартні методи, які дозволяють виконувати операції збереження та завантаження даних. Зокрема, методи save_one (зберегти один запис), save_many (зберегти кілька записів), load_one (завантажити один запис), load_all (завантажити всі записи) та load (завантажити записи за певним критерієм). Додатково, був реалізований метод для завантаження записів з бази даних відповідно до заданого фільтру.

Для зручності формування запитів та створення складних умов для пошуку даних був впроваджений query_builder. Цей клас дозволяє будувати

складні запити з використанням різних умов, таких як рівність, нерівність, більше, менше тощо. Він допомагає уникнути потреби у ручному формуванні складних запитів та спрощує процес створення запитів до бази даних.

РОЗДІЛ 3. РОЗРОБКА ГЕНЕРАТОРА КЛАСІВ

3.1. Парсинг даних для генерації.

Спочатку необхідно проаналізувати декларацію для кожного класу, що включає парсинг XML-структури. Початково, XML-файл зчитується та конвертується в словникову структуру для подальшої обробки. Потім ця інформація десеріалізується у власний тип, який ми назвемо TypeDecl, за допомогою попередньо написаного десеріалізатора. У типі TypeDecl міститься різноманітна інформація, така як модуль, назва, коментар, успадкування, декларації методів (з назвами параметрів, типами повернутих значень і т. д.), а також інші необхідні атрибути.

Даний процес аналізу та десеріалізації декларації дозволяє отримати структуровану інформацію про кожен клас, яка буде використовуватися для подальшого генерування коду і роботи з цими класами. Ця інформація дозволяє отримати доступ до детального опису кожного класу, включаючи його залежності, коментарі та структуру методів.

Такий підхід забезпечує можливість динамічного аналізу та маніпулювання декларацією класів на основі зчитаних XML-даних. Це дозволяє програмістам зручно оперувати структурою класів та виконувати автоматичне генерування коду на основі цієї інформації.

Декларації типу Type0, для використання його в подальшому в Type1 зображено на рисунку 3.1, а приклад декларації Type1, з прикладом поля та метода зображено на рисунку 3.2. Відповідно приклад зчитаної декларації Type1 зображено на рисунку 3.3.

```
<Decl>
  <Module>
    <ModuleName>Module1</ModuleName>
  </Module>
  <Name>Type0</Name>
  <Label>Type 0</Label>
  <Comment>Type 0.</Comment>
  <Declare>
  </Declare>
</Decl>
```

Рисунок 3.1 - Декларація типу Type0.

```
<Decl>
<Module>
    <ModuleName>Module1</ModuleName>
  </Module>
  <Name>Type1</Name>
  <Label>Type 1</Label>
  <Comment>Type 1.</Comment>
  <Inherit>
    <Module>
      <ModuleName>Module0</ModuleName>
    </Module>
    <Name>TypeBase</Name>
  </Inherit>
  <Elements>
      <Value>
        <Type>Double</Type>
      </Value>
      <Name>Field1</Name>
      <Comment>Field1</Comment>
    </Elements>
  <Keys>Field1</Keys>
  <Declare>
    <Handlers>
      <Name>DoSomething</Name>
      <Label> Do Something </Label>
      <Type>Job</Type>
      <Return>
        <Data>
          <Module>
            <ModuleName>Module2</ModuleName>
          </Module>
          <Name>ResultOfModule2</Name>
        </Data>
      </Return>
    </Handlers>
   </Declare>
</Decl>
```

Рисунок 3.2 - Декларація типу Type0.

```
'Module' = {OrderedDict: 1} OrderedDict([('ModuleName', 'Module1')])
    'ModuleName' = {str} 'Module1'
  Value of a primitive type
'Name' = {str} 'Type1'
'Label' = {str} 'Type 1'
'Comment' = {str} 'Type 1.'
'Inherit' = {OrderedDict: 2} OrderedDict([('Module', OrderedDict([('ModuleName', 'Module0')])), ('Name', 'TypeBase')])
  'Module' = {OrderedDict: 1} OrderedDict([('ModuleName', 'Module0')])
    'ModuleName' = {str} 'Module0'
    __len__ = {int} 1
    Protected Attributes
  'Name' = {str} 'TypeBase'
  __len__ = {int} 2
  Protected Attributes
'Elements' = {list: 1} [OrderedDict([('Value', OrderedDict([('Type', 'Double')])), ('Name', 'Field1'), ('Comment', 'Field1')])]
  0 = {OrderedDict: 3} OrderedDict([('Value', OrderedDict([('Type', 'Double')])), ('Name', 'Field1'), ('Comment', 'Field1')])
    'Value' = {OrderedDict: 1} OrderedDict([('Type', 'Double')])
    'Name' = {str} 'Field1'
    'Comment' = {str} 'Field1'
    __len__ = {int} 3
    Protected Attributes
  __len__ = {int} 1
  Protected Attributes
'Keys' = {list: 1} ['Field1']
  0 = {str} 'Field1'
  __len__ = {int} 1
  Protected Attributes
'Declare' = {OrderedDict: 1} OrderedDict([('Handlers', [OrderedDict([('Name', 'DoSomething'), ('Label', 'Do Something'), ('Type', 'Job'), ('Return', OrderedDict([('Data', OrderedI
  'Handlers' = {list: 1} [OrderedDict([('Name', 'DoSomething'), ('Label', 'Do Something'), ('Type', 'Job'), ('Return', OrderedDict([('Data', OrderedDict([('Module', OrderedDict([(
    0 = {OrderedDict: 4} OrderedDict([('Name', 'DoSomething'), ('Label', 'Do Something'), ('Type', 'Job'), ('Return', OrderedDict([('Data', OrderedDict([('Module', OrderedDic
      'Name' = {str} 'DoSomething'
      'Label' = {str} 'Do Something'
      'Type' = {str} 'Job'
      'Return' = {OrderedDict: 1} OrderedDict([('Data', OrderedDict([('Module', OrderedDict([('ModuleName', 'Module2')])), ('Name', 'ResultOfModule2')]))])
      __len__ = {int} 4
```

Рисунок 3.3 - Приклад зчитаної декларації.

Для зручності для кожного для кожного з цих полів був написаний свій клас і словник десерелізувався попередньо написаним десереалізатором до більш зручного формату.

3.2. Генерування Python класів.

За допомогою десеріалізованих даних, що були отримані на попередньому етапі, можна згенерувати класи. Для цього необхідно скласти опис класу, включаючи його батьківський клас, методи та необхідні

залежності. Інформація про місцезнаходження параметрів дозволяє правильно виконати залежності без конфліктів у просторі імен.

Для зручності створення класів використовується бібліотека attr[7], яка надає зручний спосіб задавати структуру класу. Кожен метод класу слід огорнути декоратором `@cl_handler`, який був попередньо описаний, щоб створити логіка генерування запитів створилася автоматично.

Під час генерації класів також слід враховувати наявні коментарі, які були включені в декларацію. Вони допомагають краще розуміти код та зберігають цінну інформацію про функціональні особливості класу.

Окрім того, для кожного згенерованого класу необхідно створити відповідний ключ. Для цього достатньо створити клас, що містить лише ключові поля, і додати до назви класу суфікс "Key".

Крім цього потрібно згенерувати для кожного класу Query, щоб була змогу зручно задавати запити за кожним з типів, тут все простіше, для кожного з примітивних типів описується PrimitiveTypeQuery, а всі інші генерують поля базуючись на інших Quary типах.

Так для автогенерації класових методів слід додати ще такі відповідні декоратори як

```
'@attr.s(slots=True, auto_attribs=True)'
```

Також за необхідності зберегти додаткову інформацію (таку як чи є поле необов’язковим, або ж чи є ключовим, чи можливо задля визначення початкового ім’я, бо просто to_snake_case згенерував зарезервоване слово, можна це зробити в metadata)

```
f'default=None, kw_only=True, metadata={{{", ".join(query_element_metadata)}}}'
```

Для кожного поля значення за замовчуванням виставляється None здаля того, щоб автоматично згенерувався конструктор для кожного поля. Також

так як ключ клас все одно генерується логічним кроком є наслідувати клас від ключа і не переписувати функціонал.

Враховуючи всі ці фактори, якщо ми початково задали декларації двох класів, то на виході отримаємо згенеровані файли, рисунок 3.4.

Name	Type	Size
__init__.py	PY File	1 KB
type0.py	PY File	1 KB
type0_key.py	PY File	1 KB
type1.py	PY File	2 KB
type1_key.py	PY File	1 KB

Рисунок 3.4 - Згенеровані файли

Приклад структури звичайного типу без полів та методів з відповідним query типом зображений на рисунку 3.5.

```
from module1.type0_key import Type0Key

@cl_label('Type 0')
@attr.s(slots=True, auto_attribs=True)
class Type0(Type0Key):
    """Type 0."""

    pass


@attr.s(slots=True, auto_attribs=True)
class Type0Query(ClQuery):

    pass
```

Рисунок 3.5 - Згенерований файл type0.py

Відповідний ключ, згенерований до нього зображений на рисунку 3.6:

```
@attr.s(slots=True, auto_attribs=True)
class Type0Key(ClKey):
    """Type 0."""

    pass


@cl_data_class
@attr.s(slots=True, auto_attribs=True)
class Type0KeyQuery(ClQuery):

    pass
```

Рисунок 3.6 - Згенерований файл type0_key.py

Приклад типу у якого є ключове поле і є метод та відповідний query зображено на рисунку 3.7.

```
from cl.module0.type_base import TypeBase, TypeBaseQuery
from cl.module0.type1_key import Type1Key


@cl_label('Type 1')
@attr.s(slots=True, auto_attribs=True)
class Type1(TypeBase, Type1Key):
    """Type 1."""


    @cl_handler
    def do_something(self, context: ClContext) -> Type1Key:
        pass


@attr.s(slots=True, auto_attribs=True)
class Type1Query(TypeBaseQuery):

    field1: Optional[ClDoubleCondition] = attr.ib(default=None, kw_only=True)
    """Field1"""
```

Рисунок 3.7 - Згенерований файл type1.py

Відповідний ключ до нього зображений на рисунку 3.8.

```
@attr.s(slots=True, auto_attribs=True)
class Type1Key(TypeBaseKey):
    """Type 1."""

    field1: float = attr.ib(default=None, kw_only=True)
    """Field1"""


@attr.s(slots=True, auto_attribs=True)
class Type1KeyQuery(TypeBaseQuery):

    field1: Optional[ClDoubleCondition] = attr.ib(default=None, kw_only=True)
    """Field1"""
```

Рисунок 3.8 - Згенерований файл type1_key.py

3.3. Генерування init файлів.

Пошук оптимального рішення для ефективної та коректної десеріалізації виявився непростим завданням, що вимагало ретельного розуміння технічних вимог та системи. Особливу складність представляло забезпечення належного імпорту всіх необхідних файлів, що були включені в запит, оскільки це вимагало точного відтворення відповідних класів.

Початковий підхід до вирішення цієї проблеми передбачав імпорт усіх класів в самому початку. Проте цей варіант, хоча і простий, був неприйнятно довгим, займаючи декілька секунд лише на завантаження, що було неприпустимим у масштабах реального часу обробки.

У зв'язку з цим, було вирішено впровадити динамічне завантаження класів, використовуючи механізм лінивої ініціалізації, що передбачав імпорт

класів лише тоді, коли вони були дійсно необхідні. Для досягнення цього було створено спеціальний словник в кожному файлі ініціалізації, який зберігав важливу метаінформацію про кожен клас та модуль, в якому він знаходиться. Цей словник потім був інтегрований до файлу, відповідального за розпізнавання типів, спрощуючи та оптимізуючи процес десеріалізації.

Результатом цього удосконалення стало значне поліпшення процесу передопрацювання: час завантаження зменшився з 10 секунд до всього 20 мілісекунд. Тепер процес десеріалізації відбувається швидко та ефективно, гарантуючи точність та повноту завантажених класів, що, у свою чергу, покращує загальну продуктивність та надійність системи. Відповідний приклад словника зображено на рисунку 3.9.

```
 9  import_dict = {
10      'Type0': 'module1.type0',
11      'Type0Query': 'module1.type0',
12      'Type0Key': 'module1.type0_key',
13      'Type0KeyQuery': 'module1.type0_key',
14      'Type1': 'module1.type1',
15      'Type1Query': 'module1.type1',
16      'Type1Key': 'module1.type1_key',
17      'Type1KeyQuery': 'module1.type1_key',
18  }
19
20  ClassInfo.register_package('module1', import_dict)
21
```

Рисунок 3.9 - Згенерований словник для заданих файлів

3.4. Написання генератора документації для згенерованих класів

Забезпечення користувачам бібліотеки зручного доступу до документації є вельми важливим аспектом. Документація дозволяє користувачам швидко ознайомитися з класами, їх полями, методами та необхідними параметрами.

Для автоматичної генерації документації було використано просте рішення - бібліотеку під назвою pdoc[6].

Однією з переваг використання бібліотеки pdoc є його максимальна простота використання. Вона надає можливість автоматично генерувати документацію на основі декларацій класів. Це дозволяє користувачам швидко отримати повну інформацію про класи, що полегшує їх розуміння та використання.

Проте, варто зазначити, що бібліотека pdoc має певні обмеження. Одне з них полягає у відсутності можливості пошуку за класами у документації. Це означає, що користувачам доведеться самостійно знаходити потрібні класи в документації. Хоча це може становити певну не зручність, загальна простота використання бібліотеки pdoc забезпечує ефективну генерацію документації.

Отже, використання бібліотеки pdoc дозволяє забезпечити користувачам бібліотеки зручну документацію, яка надає повну інформацію про класи, поля та методи. Незважаючи на обмеження, пов'язані з відсутністю можливості пошуку за класами, простота використання цієї бібліотеки робить її ефективним інструментом для автоматичної генерації документації.

Приклад згенерованої документації для класу Rate Report зображено на рисунку 3.10.

Classes

```
class ClRateReport (*, context: ClContext = None, report_id: str = None,
                    report_type: Optional[ClReportTypeKey] = None,
                    report_sub_type: Optional[ClReportSubTypeKey] = None,
                    model: ClModelKey = None, rate: ClRateKey = None,
                    settings: Optional[List[ClCalculationSettingsKey]] = None,
                    observation_schedules: Optional[List[ClDatesScheduleKey]] = None,
                    quantiles: Optional[List[float]] = None)
```

Rate calculation report.

Method generated by attrs for class ClRateReport.

▸ EXPAND SOURCE CODE

Рисунок 3.10 - Згенерована документація для типу Rate Report

3.5. Написання упакування згенерованих класів в бібліотеку.

В Python існують зручні методи для упакування класів у вигляді бібліотеки. Один із таких методів - використання пакету (wheel) бібліотеки setuptools[7]. Для створення пакету необхідно визначити файл setup.py, в якому задати потрібні налаштування та інформацію про бібліотеку, таку як назва, версія, опис, автор і т.д.

Файл setup.py є конфігураційним файлом, що дозволяє встановити параметри для упаковки бібліотеки. У ньому можна вказати, які модулі та пакети потрібно включити до пакету, залежності, необхідні для правильної роботи бібліотеки, а також інші деталі, пов'язані з налаштуванням пакету.

Одним із ключових елементів setup.py є функція setup(), в якій передаються необхідні параметри. Наприклад, назва, версія та опис бібліотеки можуть бути визначені за допомогою аргументів name, version і description. Крім того, можна вказати автора, ліцензію, URL-адресу репозиторію, які файли включати в пакет за допомогою регулярного виразу, а які ні, це важливо в разі якщо в бібліотеці також лежать тести, що користувачу очевидно непотрібні, а полегшити бібліотеку зайвим не буде та інші важливі деталі.

Після визначення необхідних налаштувань у файлі setup.py, можна використовувати команду python setup.py bdist_wheel для створення пакету у форматі wheel. Цей пакет можна легко встановити за допомогою інструментів управління пакетами, таких як pip.

Загалом, використання методу упакування бібліотеки за допомогою wheel у Python дозволяє створити компактний та легко встановлюваний пакет, який містить класи та модулі. Визначення необхідних налаштувань у файлі setup.py дозволяє зручно задати інформацію про бібліотеку та її залежності.

Відповідний приклад конфігурації зображений на рисунку 3.11.

```python
from setuptools import setup, find_packages

setup(
    name='My Generated Lib',
    version='0.1',
    url='https://github.com/username/mylibrary',
    author='Author Name',
    author_email='author@gmail.com',
    description='Description of my package',
    packages=find_packages(),
    install_requires=['numpy', 'matplotlib'],  # List of dependencies
    classifiers=[
        'Programming Language :: Python :: 3',
        'Programming Language :: Python :: 3.6',
        'Programming Language :: Python :: 3.7',
        'Programming Language :: Python :: 3.8',
    ],
)
```

Рисунок 3.11 - Приклад setup.py.

РОЗДІЛ 4. ПОВНА АВТОМАТИЗАЦІЯ ЗБІРКИ

4.1. Автоматичне вирішення проблем циклічного імпорту.

Під час написання коду на Python, особливо з використанням анотацій, можуть виникати проблеми з циклічним імпортом. Це виникає, коли два або більше модулів взаємно посилаються один на одного при імпорті, що призводить до зациклення і несправностей у роботі програми.

Одним з найпростіших рішень для уникнення циклічного імпорту без необхідності змінювати архітектуру програми є використання анотацій замість прямого імпорту класів. Замість того, щоб імпортувати клас для анотації, просто використовуйте назву класу у лапках. При цьому, імпорт буде здійснюватися лише у спеціальному блоку TYPE_CHECKING.

Наприклад, якщо є модуль A і модуль B, які взаємно посилаються один на одного, ви можете використовувати анотації замість прямого імпорту класів. У модулі A, замість імпорту класу з модуля B, просто можна використовувати назву класу у лапках, як зображено на рисунку 4.1.

```
class A:
    b_instance: 'B'  # Замість: from moduleB import B

class B:
    a_instance: A  # Замість: from moduleA import A
```

Рисунок 4.1 - Усунення циклічних залежностей

Таким чином, імпорт класу B з модуля A не відбувається напряму. Але, для забезпечення коректності типів при виконанні майбутніх перевірок типів, додайте імпорт лише для блоку TYPE_CHECKING, рисунок 4.2.

```
if TYPE_CHECKING:
    from moduleB import B
```

Рисунок 4.2 - TYPE_CHECKING

Це дозволяє забезпечити коректну роботу анотацій та перевірок типів, але уникнути циклічного імпорту.

Використання анотацій замість прямого імпорту класів у лапках та обмеження імпорту до блоку TYPE_CHECKING є простим і ефективним рішенням для уникнення проблем з циклічним імпортом.

Хоча такі випадки на практиці відбуваються не часто (приблизно 5 випадків на 10000 файлів), але все ж вони можуть виникати і потребують розв'язання. Одним з простих та ефективних рішень є написання скрипту, який здійснює автоматичну заміну імпорту класів на анотації у відповідному файлі згенерованої бібліотеки.

Цей скрипт може виявити випадки циклічного імпорту та автоматично замінити імпорт на анотації у відповідному файлі. Наприклад, він може використовувати пошук та заміну тексту для заміни рядка імпорту класу на анотацію у вигляді назви класу у лапках.

Це рішення забезпечує зручність та легкість у використанні, оскільки автоматично заміняє прямий імпорт на анотацію, що дозволяє уникнути циклічного імпорту. Крім того, такий скрипт може бути легко виконаним для згенерованої бібліотеки, що робить його зручним і масштабованим у випадках,

коли потрібно вирішити проблему циклічного імпорту великої кількості файлів.

Отже, розробка скрипту для заміни імпорту на анотації є зручним і ефективним рішенням для уникнення проблем з циклічним імпортом у Python. Воно дозволяє легко та автоматично здійснювати заміну імпорту класів на анотації, що сприяє безперебійній роботі програми та полегшує розробку складних проектів.

4.2. Автоматичний збір пакетів у коректну бібліотеку.

У процесі розробки бібліотеки, яка складається з кількох пакетів та має залежності між ними, включаючи відносне розташування, виникає необхідність правильного керування цими залежностями. Ручне розташування та керування цими залежностями може бути часо- та ресурсозатратним завданням, особливо в разі збільшення розміру проекту та кількості пакетів.

Для забезпечення правильного розташування та керування залежностями у бібліотеці був розроблений окремий скрипт. Цей скрипт дозволяє автоматично визначити та налаштувати розташування пакетів залежностей, зокрема з урахуванням їх відносного розташування. Він дозволяє уникнути проблем, пов'язаних з неправильним імпортом між пакетами та забезпечити належну роботу бібліотеки.

Цей скрипт полегшує процес розробки та управління залежностями шляхом автоматичного розташування пакетів, забезпечуючи правильну структуру та функціонування бібліотеки. Він дозволяє ефективно керувати залежностями, зокрема при додаванні нових пакетів або внесенні змін до вже існуючих.

Цей скрипт є важливим компонентом в розробці бібліотеки, оскільки забезпечує правильне розташування та керування залежностями. Він сприяє

зручності та ефективності у розробці та підтримці бібліотеки, допомагаючи зосередитись на основному функціоналі та завданнях розробки, не витрачаючи час на ручне розташування та керування залежностями.

4.3. Отримання робочої бібліотеку.

У підсумку, для досягнення повної автоматизації процесу розробки бібліотеки необхідно написати скрипт, який викликатиме генерацію для кожного з відповідних пакетів. Цей скрипт буде вирішувати такі завдання, як виявлення та виправлення циклічних залежностей, налаштування серіалізаторів, десеріалізаторів та інших необхідних об'єктів, а також генерацію колеса та документації.

Цей скрипт є ключовим елементом в процесі автоматизації розробки бібліотеки, оскільки він забезпечує автоматичне виконання рутинних завдань, які раніше вимагали ручного втручання. Він дозволяє зосередитись на розробці основного функціоналу бібліотеки, прискорює процес відладки та тестування, а також забезпечує зручну генерацію колеса та документації для швидкої поставки бібліотеки.

Після реалізації усіх попередніх кроків, наступним етапом буде налаштування відповідного сценарію на TeamCity[12] або іншому збиральному сервері. Це дозволить автоматично виконувати процес збирання та генерації бібліотеки з оновленнями за визначеним графіком або після певних подій.

Важливим аспектом буде написання тестів для стандартних сценаріїв використання, наприклад, за допомогою бібліотеки pytest[13]. Ці тести дозволять перевірити правильність роботи бібліотеки та її здатність задовольняти очікувані результати в різних сценаріях використання.

Один раз налаштувавши процес на сервері, при змінах у деклараціях або коді бібліотеки все буде автоматично перегенеровуватись, а потім можна буде просто завантажити готову актуальну версію бібліотеки з артефактів відповідного сценарію. Це спростить процес розгортання та забезпечить наявність оновленої та працездатної версії бібліотеки для використання у різних проектах та середовищах.

Таким чином, реалізація відповідного сценарію на сервері та написання тестів для стандартних сценаріїв використання допоможуть автоматизувати процес генерації, оновлення та розгортання бібліотеки, забезпечуючи швидкий доступ до актуальної версії для розробки та використання.

РОЗДІЛ 5. ПРИКЛАДИ ВИКОРИСТАННЯ

5.1. Збереження нових об’єктів до бази данних.

Як приклад застосування можна взяти створення нового об’єкта, наприклад RateReport і відповідно до нього створення нового FxRate що базується вже на існуючих кривих. Відповідний Python код зображений на рисунку 5.1.

```
def main():
    # create storage for new objects
    storage = create_web_storage(SNAPSHOT_DATA_SET)
    # create fx rate
    fx_rate = ClFxRate()
    fx_rate.rate_id = "EURUSD Rate"
    fx_rate.base_curve = ClCurveKey(curve_id="EUR ████████████████")
    fx_rate.quote_curve = ClCurveKey(curve_id="USD ████████████████")
    storage.save_one(fx_rate)

    # create report
    fx_rate_report = ClRateReport()
    fx_rate_report.report_id = "EURUSD Rate report"
    fx_rate_report.rate = fx_rate.to_key()
    fx_rate_report.model = ClModelKey(model_id="EUR_USD_████████")

    # save report
    storage.save_one(fx_rate_report)
```

Рисунок 5.1 - Python код створення та збереження нових об’єктів.

В консолі при запуску з’явиться відповідне логування, що зображене на рисунках 5.2 та 5.3.

```
_save:
Request body:
{
    "Data": [
        {
            "@Type": ClFxRateData",
            "RateID": "EURUSD Rate",
            "BaseCurve": {
                "CurveID": "EUR
            },
            "QuoteCurve": {
                "CurveID": "USD
            }
        }
    ]
}
```


Рисунок 5.2 - Логи зберігання Fx Rate


```
_save:
Request body:
{
    "Data": [
        {
            "@Type": "ClRateReportData",
            "ReportID": "EURUSD Rate report",
            "Model": {
                "ModelID": "EUR_USD_FX_MODEL"
            },
            "Rate": {
                "RateID": "EURUSD Rate"
            }
        }
    ]
}
```


Рисунок 5.3 - Логи зберігання Rate Report

І аналогічно новий репорт можна спостерігати на графічному інтерфейсі, рисунок 5.4.

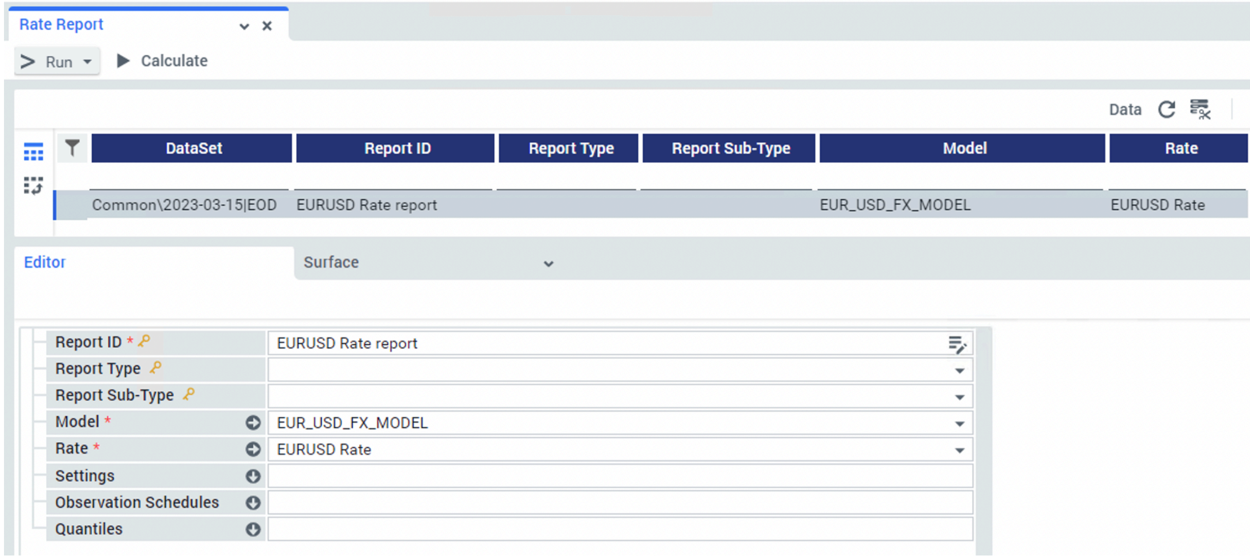

Рисунок 5.4 - Збережений об’єкт на графічному інтерфейсі.

5.2. Виклик методів для існуючих об’єктів.

Для створеного в попередньому випадку звіту, можна одразу викликати метод calculate для підрахунку репорта, для цього необхідно створити відповідний контекст і викликати метод. Python код відповідного сценарію зображений на рисунку 5.5, а відповідне логування на рисунку 5.6.

```
ctx = create_context(SNAPSHOT_DATA_SET)
# calculate report
fx_rate_report.calculate(ctx)
```

Рисунок 5.5 - Виклик методу у існуючого об’єкта.

```
Request body:
{
    "Method": "Calculate",
    "Key": {
        "@Type": "█████ClRateReportKey",
        "ReportID": "EURUSD Rate report"
    },
    "Arguments": {}
}
```

Рисунок 5.6 - Python логи реквесту виклику методу.

5.3. Імпорт результатів та їх аналіз в Python.

Базуючись на попередніх скриптах, після підрахунку репорта тепер є можливість експорту його результатів за допомогою viewer-а так наприклад можна скористатися viewer-ом surface-ів, фактично це шляхи що були згенеровані під час самих обчислень, для цього також треба передати відповідний контекст, після цього результати можна записати в dataframe, в даному випадку повертається кастомний контейнер, тож для нього треба написати додатково кастомний конвертер і після цього в пітоні можна візуалізувати наприклад 5 квантилів результатів і звичайно можна зберегти графік в файл, а числові результати в csv. Python код відповідного сценарію зображено на рисунку 5.7.

```python
# export results (surface)
data = fx_rate_report.view_surface(ctx)

# convert matrix to dataframe
dataframe = cl_matrix_to_dataframe(data)

# create quantiles
quantiles = dataframe.quantile([0.05, 0.25, 0.5, 0.75, 0.95], axis=1)
quantiles = quantiles.transpose()

# save quantiles
dataframe.to_csv(fx_rate_report.report_id + ".csv")
quantiles.to_csv(fx_rate_report.report_id + "_quantiles.csv")

# draw quantiles
quantiles.plot(figsize=(20, 10))
plt.title("EURUSD Quantiles")
plt.savefig(fx_rate_report.report_id + "_quantiles.png")
plt.show()
```

Рисунок 5.7 - Python код, імпорт результатів.

*З причин безпеки даних, за допомогою наступного скрипта шляхи були змінені на рандомізовані, скрипт рандомізаціх зображений на рисунку 5.8.

```python
for _ in range(num_paths):
    price_path = [initial_price]
    current_price = initial_price

    for _ in range(num_days):
        # Генеруємо випадкову зміну ціни в діапазоні -5% до +5%
        price_change = random.uniform(-0.05, 0.05)

        # Рахуємо нову ціну, враховуючи зміни
        new_price = current_price + (current_price * price_change)

        price_path.append(new_price)
        current_price = new_price

    price_paths.append(price_path)
```

Рисунок 5.8 - Python код, скрипт рандомізації результатів.

Саме ж графічне відображення результатів зображено на рисунку 5.9.

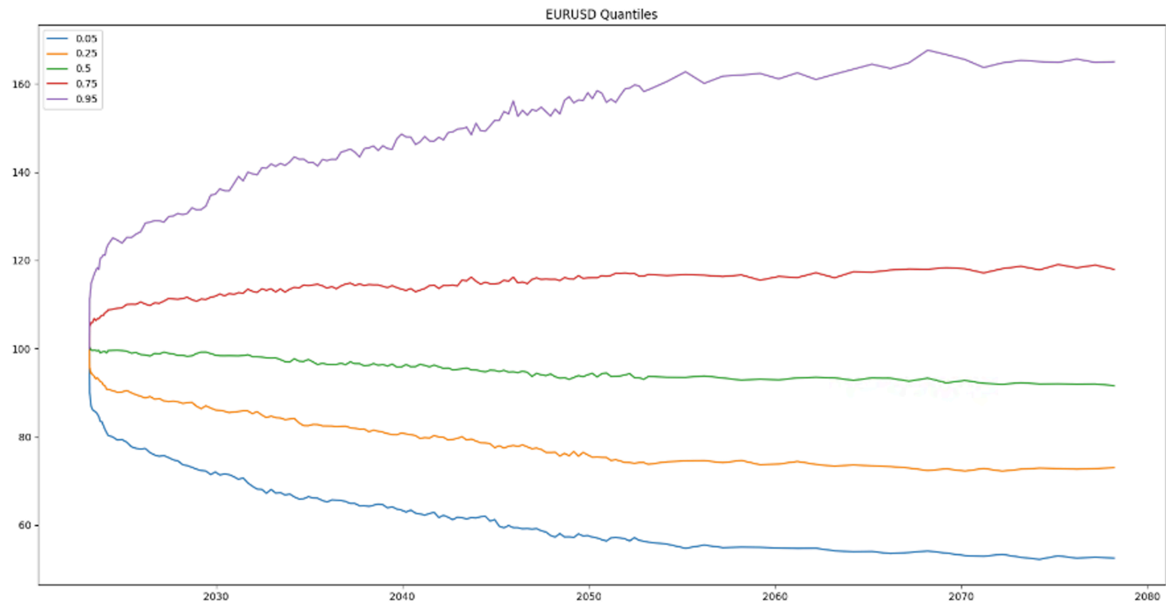

Рисунок 5.9 - Графічне відображення результатів.

ВИСНОВКИ

У дипломній роботі було розроблено Python Web API бібліотеку для взаємодії з додатком, яка має численні переваги порівняно з існуючими рішеннями, такими як Swagger, Postman та Insomnia. Основні переваги цієї бібліотеки полягають у зручності використання для складних запитів, де існує багато можливих варіантів параметрів.

У відміну від інших рішень, бібліотека надає можливість визначити тип запиту та отримати підказки щодо заповнення полів для цього типу. Це значно полегшує роботу зі складними запитами та дозволяє швидко і точно виконувати необхідні дії.

Крім того, бібліотека також забезпечує зручність написання скриптів для наповнення бази даних. Це особливо корисно, коли потрібно швидко та ефективно заповнити базу даних великим обсягом даних. Наша бібліотека дозволяє легко і зрозуміло написати необхідні скрипти для цих цілей.

Крім того, за допомогою цієї бібліотеки можна створювати тести продуктивності з бажаними інтервалами. Це дозволяє перевіряти швидкодію додатку та ідентифікувати можливі проблеми з продуктивністю.

У підсумку, розроблена Python Web API бібліотека виявляється вельми корисною та зручною у використанні. Вона надає практичні переваги в роботі зі складними запитами, наповненням бази даних та тестуванні продуктивності, спрощуючи та прискорюючи роботу розробників.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Automatic multi language program library generation for rest apis // Steiner T. /Diss. Master's thesis, University of Karlsruhe/cole Nationale Suprieure d'Informatique et de Mathmatiques Appliques de Grenoble. – 2007.
2. Specification and automated analysis of inter-parameter dependencies in web APIs // Martin-Lopez A. /IEEE Transactions on Services Computing. – 2021.
3. Документація та ресурси з інструменту Swagger [Веб-ресурс] – режим доступу: вільний. - swagger.io
4. Документація та ресурси з інструменту Postman [Веб-ресурс] – режим доступу: вільний. - postman.com
5. Документація та ресурси з інструменту Insomnia [Веб-ресурс] – режим доступу: вільний. - insomnia.rest
6. Документація та ресурси з бібліотеки pdoc [Веб-ресурс] – режим доступу: вільний. - pdoc3.readthedocs.io
7. Документація та ресурси з бібліотеки setuptools [Веб-ресурс] – режим доступу: вільний. - setuptools.readthedocs.io
8. Документація та ресурси з бібліотеки attr [Веб-ресурс] – режим доступу: вільний. - www.attrs.org
9. Документація та ресурси з бібліотеки msal [Веб-ресурс] – режим доступу: вільний. - github.com/AzureAD/microsoft-authentication-library-for-python
10. Рейфлексія в Python [Веб-ресурс] – режим доступу: вільний. - https://www.geeksforgeeks.org/reflection-in-python/
11. Офіційна документація бібліотеки Requests [Веб-ресурс] – режим доступу: вільний. - docs.python-requests.org
12. TeamCity документація. [Веб-ресурс] – режим доступу: вільний. - https://www.jetbrains.com/teamcity/documentation/

13. Документація фреймворку pytest. [Веб-ресурс] – режим доступу: вільний. - https://docs.pytest.org/
14. Документація фреймворку Flask. [Веб-ресурс] – режим доступу: вільний. - flask.palletsprojects.com